МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мережних технологій

Кваліфікаційна робота

освітній ступінь – магістр

на тему: **“Автоматизована генерація та налаштування мікросервісів для спрощення процесу розробки“**

Виконав: студент 2-го року навчання
спеціальності 121 Інженерія програмного забезпечення
Колінько Павло Володимирович
Керівник Волинець Є.А., ст. викладач, к.н.
Рецензент _____
Кваліфікаційна робота захищена
з оцінкою _____
Секретар ЕК _____
«____» _____20____ р.

Київ – 2024

Міністерство освіти і науки України
Національний університет «Києво-Могилянська академія»
Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Зав.кафедри інформатики,
к.ф-м.н., доц. Гороховський С.С
_____
(підпис)
„____”_____2023 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на кваліфікаційну роботу

студенту 2-го курсу, факультету інформатики
Колінько Павлу Володимировичу
Розробити Автоматизована генерація та налаштування мікросервісів для спрощення процесу розробки
Зміст ТЧ до кваліфікаційної роботи:

Зміст
Анотація
Вступ
1 Огляд мікросервісної та монолітної архітектур
2 Огляд scaffolding застосунків на базі платформи Node.js
3 Розробка програмного застосунку
4 Аналіз роботи програмного застосунку
Висновки
Список літератури
Додатки

Дата видачі „___” _____ 2023 р. Керівник _____
(підпис)
Завдання отримав_____
(підпис)

Графік підготовки кваліфікаційної роботи до захисту

Графік узгоджено «_____» _____ 2024 р.

№ з/п	Перелік робіт	Термін	Підпис	Дата	Примітка
1.	Отримання теми кваліфікаційної роботи	16.10.2023			
2.	Ознайомлення з існуючою інформацією за темою курсової роботи	23.10.2023			
3.	Створення плану роботи	12.11.2023			
4.	Робота з науковою літературою	13.11.2023			
5.	Аналіз предметної області дослідження, аналіз існуючих рішень та алгоритмів	02.01.2024			
6.	Аналіз математичних методів реалізації алгоритмів	02.02.2024			
7.	Підготовка даних до аналізу, використання обраних алгоритмів, оцінювання точності та ефективності	05.04.2024			
8.	Аналіз практичної частини, її корегування	14.04.2024			
9.	Початок написання текстової частини	21.04.2024			
10.	Подання проміжної версії текстової частини	11.05.2024			
11.	Завершення написання текстової частини роботи	22.06.2024			
12.	Створення презентації	23.06.2024			
13.	Захист кваліфікаційної роботи	10.06.2024			

Зміст

Анотація

Метою даної роботи є огляд існуючих рішень та розробка програмного застосунку для генерації мікросервісної архітектури засобами платформи Node.js.

У роботі розглянуто теоретичні відомості про мікросервісну архітектуру та проведено порівняльний аналіз із монолітною архітектурою. Практичною частиною роботи є дослідження scaffolding в програмуванні, тобто автоматизовану генерації коду та архітектури, проведено аналіз існуючих рішень на базі платформи Node.js. Кінцевим етапом практичної частини є розробка власного застосунка здатен генерувати мікросервісний проєкт.

Вступ

Розробка веб застосунків завжди починається з аналізу бізнес вимог та підбору архітектури для їх найоптимальнішого втілення. Інженери програмного забезпечення розробили безліч теоретичних і практичних підходів до вибору і впровадження архітектурних рішень, кожен з яких є доцільним у використанні для специфічних потреб бізнесу.

Одним з важливих питань, що постає на початку розробки бекенд застосунків, є вибір поміж монолітною та мікросервісною архітектурами, кожна з яких має свої переваги та недоліки. Проте, варто зазначити, що у кожної з вищезазначених архітектур є спільний етап конфігурації, що передбачає налаштування середовища, встановлення бібліотек, підключення бази даних та багато супутніх кроків. У випадку розробки мікросервісного застосунку, процес конфігурації повторюється велику кількість разів, адже кожен сервіс є окремим застосунком в ізольованому середовищі, що значно сповільнює процес імплементації.

Для вирішення проблеми повільної розробки та запобіганню помилок у момент конфігурації використовують scaffolding-утиліти. Scaffolding в перекладі з англійської мови означає «будівельні ліса» - це підхід в програмуванні, що передбачає генерацію коду, шаблону, архітектури проєкту. На базі платформи Node.js існують такі генеративні утиліти як Nest CLI, Yeoman, Prisma CLI, тощо. Однак, жодна з перерахованих бібліотек не вирішує проблеми автоматизованої генерації мікросервісів.

Актуальність теми полягає в двох аспектах – теоретичному та практичному. Теоретичний огляд дозволяє виділити вже існуючі новітні бібліотеки та засоби генерації коду на базі платформи Node.js, тоді як практичною частиною роботи є розробка застосунку, що здатен генерувати мікросервіси по заданому шаблону.

Розділ 1.

1.1 Монолітна архітектура

Перед початком огляду мікросервісної архітектури, необхідно розглянути її попередника, що до сих пір має широкий вжиток - монолітну архітектуру.

Монолітна архітектура - це класичний підхід в розробці програмного забезпечення, особливо для бекенд застосунків, який передбачає, що проєкт є цілісною та незалежною сутністю. Він збирає всі частини, необхідні для стабільної роботи продукту в рамках однієї кодової бази та єдиного середовища розгортання. Кожен компонент моноліта є пов'язаним та залежним із іншими внутрішніми компонентами, проте є абсолютно незалежним від зовнішніх [1]. Такі компоненти в бекенд застосунка, можна поділити на два рівня:

- рівень «бізнес логіки», загальний функціонал проєкту, як аутентифікація, агрегація, створення подій, тощо;
- рівень зберігання даних, зазвичай відповідає за збереження та обробку даних.

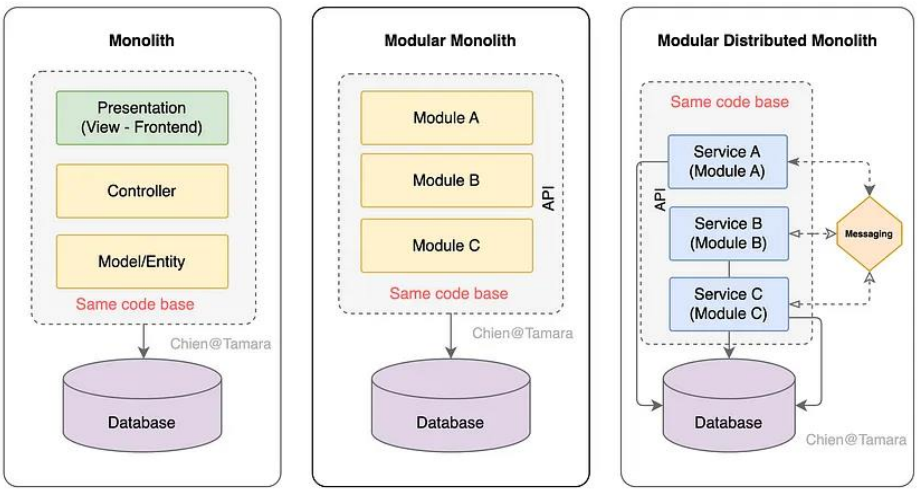


Рис. 1.1 Різновиди монолітної архітектури

Монолітна архітектура має наступні характеристики[1]:

- єдине середовище розгортання, тобто застосунок обов'язково є розгорнутий на одному сервері;
- центральний модуль контролює всі процеси додатку;
- тісна зв'язаність компонентів проєкта;

- спільне використання пам’яті;
- єдина технологія (фреймфорк, мова програмування, тощо) для всього проєкту.

Такий тип архітектури має ряд ключових переваг, наприклад, використання одного фреймворки чи мови програмування значно пришвидшує швидкість розробки, також такі додатки дуже легко деплоїти, тобто розгортати на сервері.

Проте, існує і ряд вагомих недоліків, що можуть викликати суперечки стосовно доцільності існування такого архітектурного підходу, одна з яких - поступове “забруднення” кодової бази із збільшенням проєкту. Розробка та впровадження нового функціоналу завжди збільшує кодову базу, що може призвести до нечитабельності та незрозумілості монолітного проєкту.

Програмні помилки також є критичним аспектом, що впливає на роботу застосунку, адже лише одна помилка призведе до відмови роботи всього застосунку. Така відома проблема пов’язана із спільним середовищем розгортання та тісною взаємодією компонентів продукту.

Монолітний підхід в розробці є широковживаним і в сучасних проєктах, що стартували нещодавно, проте, вибір на користь тієї чи іншої архітектури має бути виваженим, та базуватись на потребах конкретного продукту [1]. Наприклад, використання моноліту є доцільним для невеликих або середніх проєктів, для яких не передбачене збільшення навантаження та кількість необхідного функціоналу є сталою величиною.

1.2 Мікросервісна архітектура

Таке поняття, як мікросервісна архітектура, для скорочення використовують термін “мікросервіси” або “розподілені системи”, з’явився на початку 2010-х років. На противагу розглянутій у розділі 1.2 монолітній архітектурі, підхід до розробки в мікросервісному архітектурному стилі передбачає розподілену кодову базу, що зазвичай є розбитою на модулі, кожен з яких відповідає лише за певний функціонал [2].

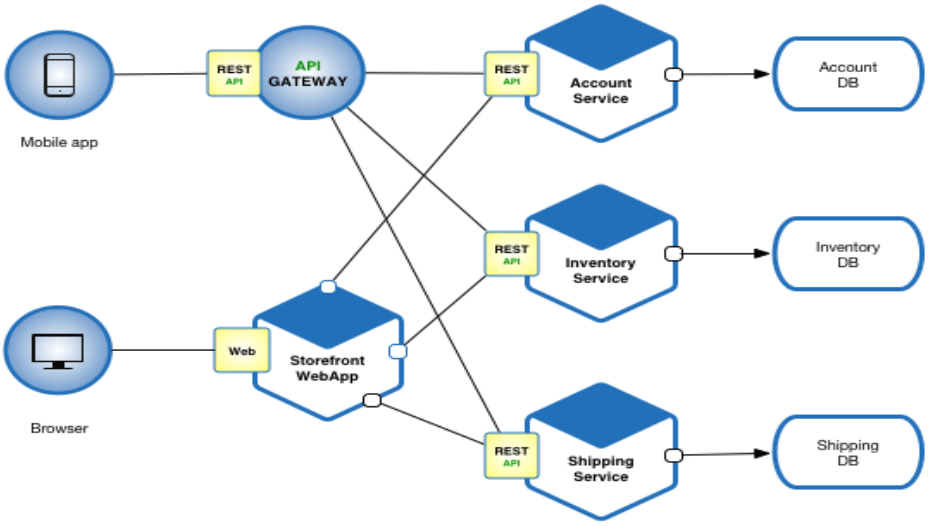


Рис 1.2 Архітектура мікросервісного проєкту

Найчастіше мікросервісну архітектуру закладають на етапі проєктування продукту, проте, існують випадки, коли є необхідним переписувати стару кодову базу, що була розроблена у вигляді моноліта. Тоді, можуть використовувати суміжній підхід, який описав Мартін Фаулер в своїй роботі MonolithFisrst про перехід від моноліту до розподіленої системи: *"Найпоширеніший підхід, коли система створюється у вигляді моноліту, після чого по краях від неї поступово починають відсікатися мікросервіси. При такому підході в серці мікросервісної архітектури залишається велике монолітне ядро, але більшість нових розробок припадає на мікросервіси, тоді як моноліт залишається у такому ж стані."* [3]

Найбільшою незалежною одиницею розподіленою системи є один мікросервіс що представлений у вигляді атомарного застосунку. Зазвичай кожен такий сервіс побудований довкола конкретної бізнес-логіки, наприклад, окремий сервіс для аутентифікації, листування, тощо. Кожен з них є розгорнутим в окремому середовищі, зазвичай має власну базу даних.

1.2.1 Порівняння мікросервісної та монолітної архітектур

Мікросервісний архітектурний підхід надає ряд переваг порівняно з монолітної архітектурою:

- незалежність в розробці та розгортанні;
- відмовостійкість;

- масштабованість;
- доступність.

Оскільки кожен мікросервіс є незалежним від інших мікросервісів в системі, вносити зміни в його кодову базу простіше, аніж моноліт. Таким чином, якщо необхідно ввести зміни лише, в сервіс, що відповідає за надсилання листів користувачам, немає жодної потреби вносити зміни в сервіс для менеджменту користувачів, що дозволяє розробникам працювати незалежно одне від одного [4]. Незалежність сервісів також дозволяє використовувати в рамках одного проєкту різні технології та підходи, наприклад, фреймворки або навіть різні мови програмування, що найкраще підійдуть для успішного виконання технічного завдання. В свою чергу, окреме середовище, де був розгорнутий сервіс, використання окремої бази даних для кожного проєкту дозволяють досягати вищого рівня відмовостійкості порівняно із монолітом. У разі виникнення критичної помилки в одному із сервісів, інші будуть доступним для користувачів [4].

Таким чином, два концептуально різні підходи є заміною одне одному, а інструментами, що мають бути використані за підходящих умов.

1.2.2 Структура проєкту зазснованого на мікросервісній архітектурі

Через особливості своєї структури, розподілені системи мають додатковий функціонал для правильної взаємодії між собою. Оскільки під час розробки мікросервісів легко втратити контроль над їх кількістю, Завершений мікросервісний додаток має наступну структуру:

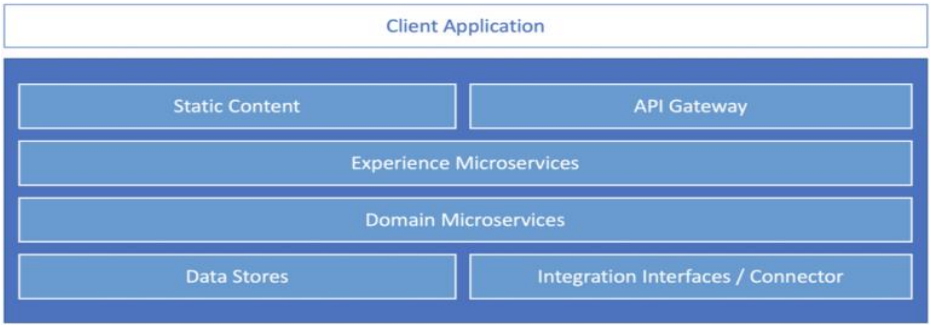


Рис. 1.3 Структура мікросервісного застосунку

Найвищий рівень завершеного доробку є клієнт. Зазвичай фронтенд застосунок що має графічний інтерфейс зрозумілий людині, з яким взаємодіє користувач. Комунікація між клієнтом та сервером в монолітній та мікросервісній архітектурах не відрізняється між собою, тобто запити надсилаються за допомогою API. Для взаємодії може бути використана Rest, GraphQL або gRPC архітектури [4]. Це такі патерни взаємодії клієнта та сервера що передбачають набір правил взаємодій. Проте, оскільки кожен мікросервіс є незалежним застосунком, комунікація клієнта мала б відбуватись з кожним з них окремо. Аби цього уникнути, використовують API Gateways утиліти. В рамках мікросервісної архітектури, вони є незалежними сервісами, що слугують єдиною точкою входу для комунікації, через яку проходять всі API запити, тоді як gateway розсилає їх на необхідні сервіси. Більш детально API Gateways будуть розглянуті у розділі 1.3.

Як зображено на рисунку *1.3*, наступним рівнем є функціональна бізнес логіка, що відповідає за обробку інформації, збір статистики, впровадженню відкладених задач, тощо. Він поділяється на два підтипи:

- experience;
- domain.

Experience рівень відповідає за бізнес логіку та взаємодію з даними, клієнтом та іншими навантаженими частинами застосунку, тоді як domain відповідає за вузькоспеціалізований функціонал, такий як, наприклад, шифрування даних [4].

Використання окремого середовища для сервісів передбачає необхідність комунікації між ними. Обмін даними та повідомленнями поміж сервісів відбувається за допомогою Event Brokers, Message Queues або інших патернів взаємодії сервісів. Такі додаткові утиліти дозволяють асинхронно передавати інформацію від сервісу до сервісу. Більш детально підходи до імплементації будуть розглянуті у розділі 1.4.

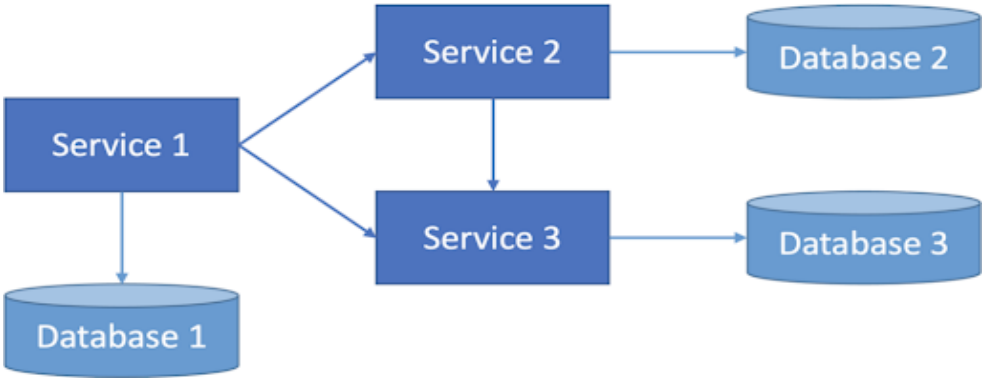


Рис.1.4 Приклад взаємодії мікросервісів

1.2.3 Патерни взаємодії мікросервісів із базою даних

Для читання, запису та агрегації інформації, аналогічно до монолітної архітектури, розподілені системи використовують бази даних. Архітектурний рівень, що відповідає за обробку інформації іноді називають “найнижчим”. Існують три основні принципи розгортання бази даних в рамках розподіленої системи [5] :

- Private-tables-per-service – кожен сервіс має приватні таблиці, що є доступними лише для нього;
- Schema-per-service – кожен сервіс має власну схему в базі даних, доступну лише для нього;
- Database-server-per-service – кожен сервіс має незалежну базу даних, доступну лише для нього.

Архітектурний підхід розділення сервісів, кожен з них має власне середовище та спосіб комунікацій із базою даних гарантує високий рівень відмовостійкості, тоді як відмова від використання зазначених підходів є поганою практикою.

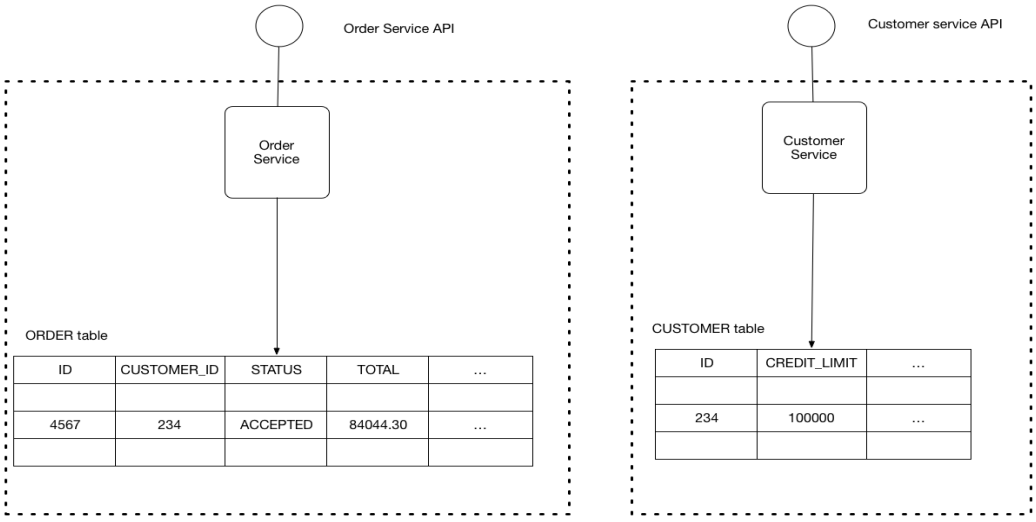


Рис. 1.5 Взаємодія сервісу із базою даних

З переліку представлених патернів взаємодії найбільших вживаним є Database-server-per-service патерн, адже він дозволяє досягнути найвищого рівня відмовостійкості та ізольованості окремого сервісу. Проте, нерідко виникає необхідність об’єднувати дані в рамках двох окремих мікросервісів або проводити операції над даними для двох баз, що знаходяться у середовищі різних сервісів [5]. Для вирішення такої задачі використовують різноманітні підходи, наприклад, таблиці та дані в рамках різних сервісів можуть дублюватись, що допомагає зберігати необхідні дані у разі відмови одного з сервісів.

1.2.4 Обробка типових помилок

Під час розробки мікросервісіного застосунку виникає ряд розповсюджених проблем, для більшості з яких існує розроблене рішення. Наприклад, головна проблема суміжна з основною перевагою - ізольованість середовища та обмін даними між ними. При обміні даних можуть виникнути мережеві або програмні помилки, що призведуть до відмови системи, неоднорідності даних, тощо.

Другою вагомою проблемою є збереження даних в однорідному стані поміж різних баз даних. Візьмемо за приклад ситуацію розробки банківської системи, в якій розроблено окремий сервіс, що відповідає за менеджмент користувачів, а другий відповідає за роботу з їх рахунками. Для переказу коштів поміж рахунками користувачів, валідним буде лише один з двох сценаріїв:

- гроші будуть списані з одного рахунку і переведені на інший;
- не відбудеться жодної зміни даних, і повернено помилку.

Такий підхід є необхідним для того, аби зберігати однорідність (конзистентність) даних. В рамках однієї бази даних при використанні монолітної архітектури, використовуються транзакції. Транзакції - це така агрегація декількох операцій над даними в базі, яка виконається лише за умови виконання всіх вкладених операцій послідовно, або не буде виконана взагалі. Для обробки такої ситуації в рамках розподіленої системи, використовують патерни обробки транзакцій в розподілених системах. Такі патерни обробки працюють зазвичай або “ланцюговою реакцією”, тобто передачею даних від сервісу до сервісу, або за допомогою так званих “оркестраторів”, окремих застосунків, що керують передачею даних поміж різних баз даних [6]. Важливо зазначити, що вразі неуспішності виконання стан буде повернений до попереднього. Існує чимала кількість патернів обробки розподілених транзакцій [6], основні з яких є:

- 2PC, двофазний коміт, який використовує ланцюговий підхід передачі даних поміж сервісів, як зазначено в назві, обробляється в два етапи, перший з яких є спробою, а другий затвердженням;
- 3PC, надбудова над 2PC, має аналогічну дію, проте додатково виконує фазу запиту готовності ресурсу, перед спробою запису даних;
- SAGA, поділяється на два основних типи, оркестрація та хореографія, використовує єдину точку входу, фактично окремий додаток, що керує транзакціями;
- Local Message Table, підхід, за якого, всі транзакцій обробляються в локальній таблиці оркестратора та потім розсилаються на необхідні сервіси.

Загалом, типові помилки мікросервісів можна поділити на два типи – мережеві (системні) та програмні. Говорячи про програмні помилки, способи запобігання їм є аналогічними до обробки в будь-якій іншій архітектурі, за допомогою передбачення граничних випадків дій користувача, використання try/catch блоків, тощо. Обробка мережевих та системних помилок є більш

фундаментальним підходом, адже окрім стандартних помилок коду передбачає обробку непередбачуваних «зовнішніх обставин» [6], таких як нестабільне мережеве підключення, неоднорідність даних в базі.

1.3 API Gateway

Як було зазначено у розділі 1.2.2, комунікація клієнта і сервера в рамках мікросервісної архітектури відбувається або з кожним сервісом окремо, або за допомогою шлюза, що також називаються API Gateway.

Шлюз - це єдина точка входу в додаток [7]. Фактично, це окремий застосунок, котрий розсилає запити на дочірні сервіси всього застосунку, та повертає відповідь клієнту. Принцип його роботи є дуже простим:

- отримання HTTP запит;
- валідація даних;
- перевірка аутентифікації/авторизації;
- надсилання запиту до відповідного бекенд ресурсу;
- надсилання відповіді назад до клієнта.

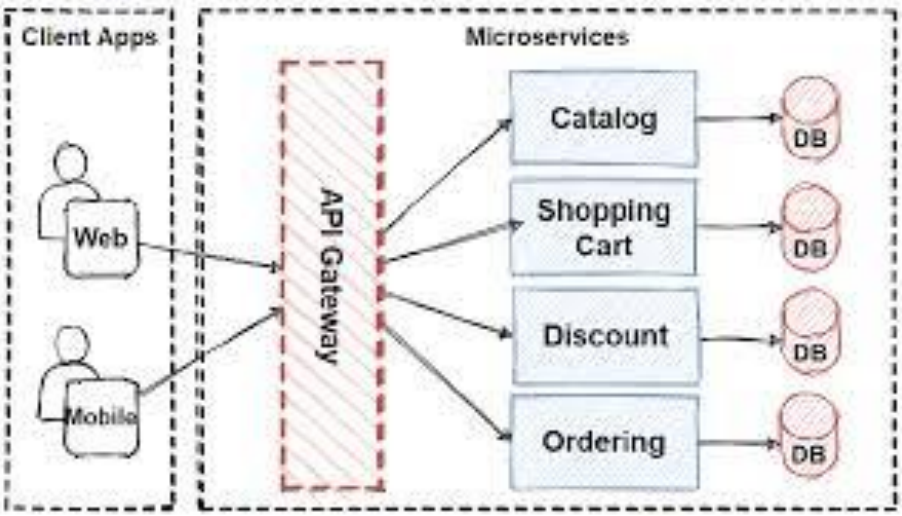


Рис 1.6 Використання API Gateway

Окрім розсилки, API Gateway також виконує додаткові функції обмеження кількість запитів, що надходять з однієї IP адреси за допомогою спеціальних утиліт, що називаються Rate Limiters. Також, він зачасту виконує роль

моніторингу, тобто записує джерело надходження запитів, дані про користувача та будь-яку необхідну інформацію.

API Gateways в класичному підході використовують або програмні, тобто ті, що написані власноруч, або ті, що надаються різноманітними хмарними сервісами, такий як AWS API Gateway. AWS, абревіатура, що розшифровується як Amazon Web Services, хмарна платформа, що надає велику кількість готових рішень, починаючи від оренди серверних потужностей і закінчуючи вищезазначеним шлюзом [7]. Такі готові програмні рішення в хмарних сервісах є більш надійними та стабільними рішеннями, адже використовують потужні сервери, мають широкий готовий функціонал та є інтегрованими в екосистему AWS, наприклад, використовує потужну систему моніторингу та розподілення навантаження.

Таким чином, підхід у використанні API Gateway є дуже потужним та стабільним рішенням, що значно спрощує процес розробки та інтеграцію API на стороні клієнта, дозволяє аналізувати дані та обмежує навантаження, відслідковуючі кількість запитів зі сторони клієнта.

1.4 Патерни взаємодії сервісів

Ізольованість та незалежність кожного мікросервісу від іншого передбачає можливість комунікації між ними задля обміну станом та даними. Типовим прикладом такої взаємодії є необхідність отримання спільної інформації про ролі користувача в системі. Для вирішення цієї проблеми існують декілька підходів до передачі повідомлень поміж сервісів.

1.4.1 Комунікація за допомогою HTTP протоколу

Одним з потенційних способів вирішення вищевказаної задачі є комунікація сервісів за допомогою HTTP/HTTPS протоколу, що може мати декілька реалізацій, в залежності від потреб бізнес-логіки. Для імплементації такого патерну передбачено розробку окремого клієнту в рамках бекенд застосунку, що надсилатиме запити на сервіси та пересилати відповідь від них у зворотному

напрямку. Це може бути реалізовано, у спрощений спосіб, у вигляді невеликої кодової бази для кожного сервісу окремо, так і його більш складною варіацією, розробкою окремого потужного застосунку-оркестратора, котрий буде виконувати аналогічну функцію. Переваги другого методу над першим є відсутність дублювання коду та контроль всього внутрішнього трафіку з однієї точки.

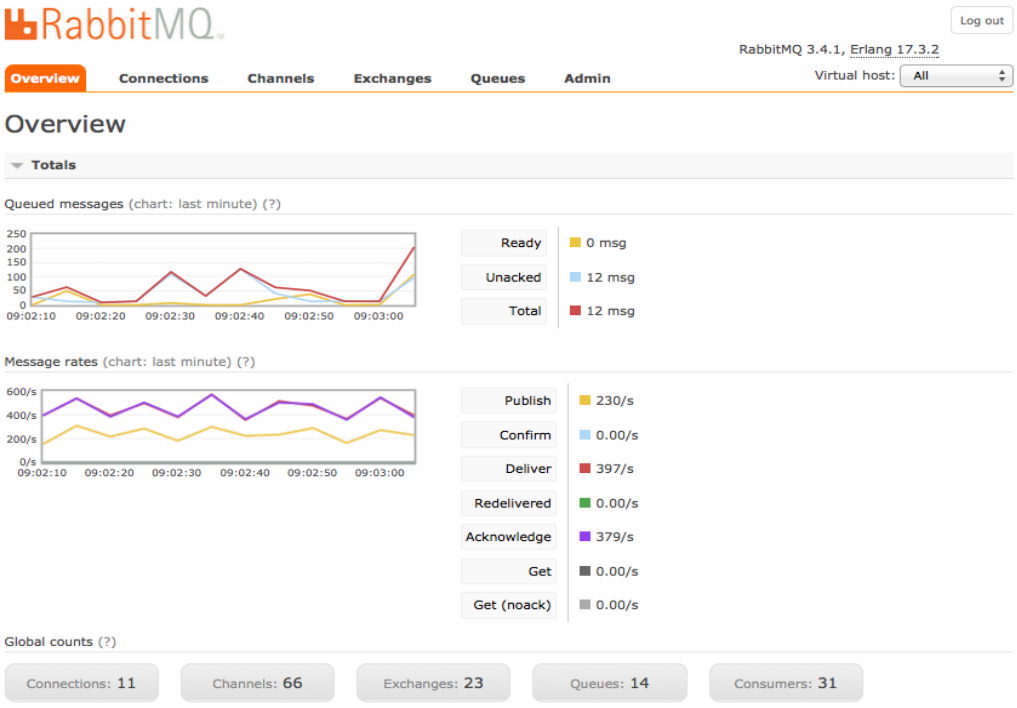
Такий підхід має ряд переваг та недоліків. Основна перевага полягає в швидкості розробки, адже такий підхід не потребує багато людських ресурсів. Проте, швидкість обміну даними кратно програє іншим таким методам як черги повідомлень та брокери подій.

1.4.2 Черги повідомлень

Черги повідомлень, такі як RabbitMQ, Apache Kafka - це зовнішні утиліти, що дозволяють асинхронно обмінюватись даними поміж різних компонентів застосунку безпосередньо. Окрім мікросервісної архітектури, можуть бути використані в рамках моноліту чи хмарної архітектури. Вони працюють по принципу обміну повідомлень від одного ресурсу, які називаються publisher або producer до другого, що називається consumer. Оскільки операція є асинхронною [8], тобто такою, що час на її виконання є невідомим, повідомлення публікуються в так звану чергу, де зберігаються по принципу “отримати і забув”. Цей принцип передбачає, що як тільки consumer маркує повідомлення як отримане від publisher, повідомлення буде одразу видалене з черги, та повернутись до нього буде неможливим.

Рис 1.7 Передача повідомлень від producer до consumer

Окрім можливості обміну повідомлень, всі вище перераховані утиліти надають можливість моніторингу стану та швидкості обробки повідомлень, менеджменту різних черг, тощо.



1.8 Графічний інтерфейс RabbitMQ

Черги повідомлень є потужними утилітами для обміну даними поміж сервісів [8], що значно переважають за ефективністю внутрішню комунікацію за допомогою HTTP протоколу.

1.4.3 Брокери подій

Брокери подій (англ. - Event Brokers) є схожим механізмом до черги повідомлень, тобто такі посередники, що обробляють дані в момент їх передачі від publisher до subscriber. Вони мають доступ до контексту та метаданих самого повідомлення та можуть робити з ними такий набір операцій як читання та запис. Проте, на відміну від черг повідомлень, брокери подіють реалізовані таким чином, що publisher та subscriber не обов’язково мають бути напряму пов’язані між собою, адже за керування “напрямком” інформації відповідає саме такий посередник [9], що надсилає дані від ресурсу-публікатора повідомлення до одного або декількох підписників.

1.9 Брокери подій

Event brokers завжди в тій чи іншій формі використовуються в Event-Driven архітектурах. Такий тип архітектури є загальною абстракцією, що описує кодову базу, як систему, що «спостерігає» за подіями, наприклад, отримання даних або помилки, закриття потоку передачі даних, тощо. Брокери подій бувають трьох типів - програмні, системні та хмарні, та мають реалізовувати наступний функціонал:

- потокову обробку даних;
- різні режими доставки інформації, такі як “рівно один раз”, “не більше одно разу” та “хоча б один раз”;
- логування всіх “подій” для аналізу та зберігання;
- захист на різних рівнях;
- розширюваність та масштабованість кодової бази брокера повідомлень;
- механізми запобігання помилок та висока доступність в одиницю часу.

Таким чином, Event Brokers є потужним інструментом доставки інформації поміж програмних компонентів або ресурсів, що використовують децентралізований підхід обробки подій за допомогою оркестатора. Найпопулярнішими брокерами подій є такі програмні продукти, як Microsoft Azure Event Grid, IBM MQ, Confluent Cloud, TIBCO Messaging, PubSub+ Platform [9].

1.5 Розгортання та контейнеризація застосунку

Кожен сервіс в рамках мікросервісної архітектури є незалежним додатком, тому він має бути розгорнутим в окремому середовищі. Для спрощення цього процесу, використовується так званий підхід контейнеризації, що передбачає

розгортання програмного застосунку в ізольованому віртуальному середовищі, умовному “контейнері”, що не взаємодіє із зовнішніми компонентами системи, проте має змогу звертатись до нього.

Для створення такого ізольованого середовища використовують Docker, продукт компанії Docker inc. Програмне забезпечення є простим в експлуатації, використовує набір CLI команд та Dockerfile, такий файл, що описує пошарову структуру майбутнього “контейнера” у вигляді простих команд. Окрім конфігураційного файлу, було створено додаткову утиліту для роботи з декількома контейнерами одночасно, що має назву docker-compose. Цей продукт був розроблений тим же постачальником програмних послуг використовує конфігураційний docker-compose.yaml файл та нескладні CLI для взаємодії. Таким чином, схематичний вигляд контейнеризованих мікросервісів виглядає наступним чином:

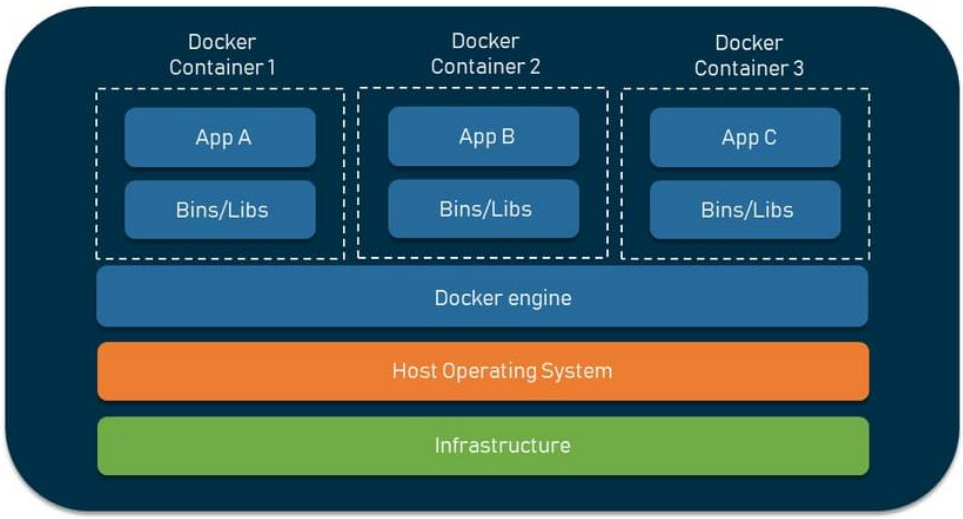


Рис. 1.10 Ізольовані Docker “контейнери”

Проте, з часом кількість сервісів зростає в арифметичній прогресії. Для керування великою кількістю таких мікросервісів використовують продукт корпорації Google, названий Kubernetes [10]. Це програмне забезпечення є оркестратором, тобто “центром керування” всіх контейнеризованих сервісів, що спрощує та пришвидшує процес інтеграції, розробки та паралельного розгортання декількох версій продукту.

Контейнеризація застосунку є великою перевагою для розгортання проєкту, наприклад, на хмарних серверах, адже дозволяє уникнути налаштування цих серверів під потреби розробленого додатку, а оркестрація контейнерів спрошує керування [10].

Розділ 2.

2.1 Scaffolding в програмуванні

У момент старту розробки будь-якого додатку виникає доволі вагома проблема, що займає чималу кількість часу - налаштування та загальна конфігурація проєкту. Налаштування кожного окремого сервіса в рамках одного бекенд застосунку зазвичай співпадають, тому така робота є одноманітною та займає чималу кількість часу. Якщо говорити про розробку мікросервісного застосунку, процес налаштування повторюється нескінченну кількість разів, адже, особливо в бекенд застосунках середнього та великого розміру, кількість сервісів зростатиме з плином часу та функціоналу проєкта [11]. Тому, аби заощадити час та мінімізувати кількість помилок на етапі налаштування, було розроблено певний набір утиліт, що виконують функцію конфігурації. Більша частина таких утиліт, особливо під час для платформи Node.js не є застосунками для генерації коду, а набором правил та рекомендації в текстовому вигляді.

В розробці програмного забезпечення існує такий термін як scaffolding (англ. - будівельні ліса). Відповідно до назви, цей підхід передбачає генерацію по якомусь загальному шаблону або “скелету”. Такий підхід необов’язково передбачає генерацію кінцевої версії продукту, що є готовим до використання, навпаки, частіше згенерована частина є лише основою, що спрощує та значно пришвидшує подальшу розробку застосунку.

Scaffolding можна розділити на дві основні категорії:

- Генерація коду;
- Генерація архітектури.

У першому випадку, утиліта допомагає генерувати шматки типового коду, такі як інтерфейси, функції getter та setter, тощо. Цей підхід підтримується деякими model-view-controller (MVC) фреймворками. MVC - це шаблонний патерн розробки програмного забезпечення, де буква M означає Model, тобто таку сутність що описує взаємодію з базою даних, V - це View, тобто шаблон відображення даних для клієнтської сторони додатка, C - Controller, тобто код,

що відповідає за взаємодію Model та View. Окрім нативних бібліотек фреймворків, процес генерації коду доставляють різноманітні бібліотеки, такі як Lombok написаний мовою програмування Java, що за допомогою дозволяє генерувати такі типові методи як getter,setter,toString,equals і тд.

Scaffolding для генерації коду, в свою чергу, також поділяється на два підкласи - design генерацію та runtime генерацію. Design генерація передбачає створення коду у момент збірки проєкта та дозволяє розробнику самостійно модифікувати створені файли. Runtime генерація навпаки, генерує код “на ходу”, тобто, безпосередньо під час написання коду. Runtime генерація більше підходить для великих за розміром проєктів, адже модифікація файлів вручну вже після design генерації може викликати конфлікти та призведе до повної недієздатності проєкту [11].

Типовий приклад design генерації коду - scaffolding у RoR (Ruby on Rails), шо працює за допомогою CLI. Ця абревіатура, що розшифровується як comand line interrface котрий передбачає набір консольних команд що виконують будь-який функціонал в рамках проєкту, зазвичай, різноманітні програмні скрипти. Scaffolding у RoR надає наступні переваги [11]:

- Пришвидшує розробку коду;
- Надає однорідність структури проєкта;
- Генерація використовує найкращі практики “з коробки”.

Типова команда для генерації виглядає наступним чином:

```
rails generate scaffold Post title:string body:text
```

Рис 2.1 Команда для генерації MVC засобами фреймфорку RoR

Ця команда згенерує Post модель за патерном MVC, а саме - модель під назвою Post для взаємодії з базою даних, контролер Post де будуть описані всі CRUD операції та view файли для кожної із згенерованих CRUD операції. CRUD - абревіатура, що розшифровується як створення (англ. - create), читання (англ. - read), оновлення (англ. - update), видалення (англ. - delete). І хоча перелічені файли мають мінімальний набір функціоналу, вони суттєво пришвидшать процес

розробки, адже всю рутинну роботу вже була виконана [11], тоді як розробнику лишається лише процес розширення коду.

```
class PostsController < ApplicationController
  before_action :set_post, only: %i[show edit update destroy]

  def index
    @posts = Post.all
  end

  def show
  end

  def new
    @post = Post.new
  end

  def edit
  end

  def create
    @post = Post.new(post_params)

    if @post.save
      redirect_to @post, notice: 'Post was successfully created.'
    else
      render :new
    end
  end

  def update
    if @post.update(post_params)
      redirect_to @post, notice: 'Post was successfully updated.'
    else
      render :edit
    end
  end
```

Рис.2.2 Згенерований контролер на RoR

Важливо зазначити, що scaffolding у веб застосунках поділяється на серверну на клієнтську частину. Більшість scaffold-based утиліт використовуються саме для кодогенерації на серверній частині, адже нерідко генерація коду передбачає взаємодіє з базою даних проєкта, наприклад, генерація інтерфейсів для кожної сутності, тоді як генерація на клієнтській частині застосунку зачасту використовує трансфер даних як основу для генерації коду, що є менш затребуваним через неможливість виділити достатній рівень абстракції під будь-який проєкт. Варто згадати такі приклади генерації коду на серверній частині, що взаємодіє із сутностями бази даних на платформі Node.js - Prisma ORM та TypeORM. Ці ORM автоматично генерують інтерфейси, тобто використовує runtime генерацію коду, відносно кожної сутності в базі даних. Такий підхід суттєво спрощує розробку та заощаджує час.

На противагу генерації коду, генерація проєкту є більш масштабною та складною процедурою, адже кожен продукт має свої індивідуальні

характеристики та особливості. Тому, доволі важко побудувати таку систему, що буде достатньо гнучкою та добре розширюваною водночас. Як приклади непоганої генерації проєктів, що суттєво спрощують процес розробки можна назвати вже вищезгаданий приклад генерації цілого сервіса засобами RoR або аналогічна генерація за допомогою CLI команд фреймворком Nest.js.

2.2 Огляд існуючих рішень на Node.js

Node.js - це платформа для виконання Javascript коду поміж браузерного середовища за допомогою рушія V8. Вона має як дуже обширний функціонал, так і велику кількість фреймфорків та бібліотек, має можливість взаємодіяти з різними операційними системами, такими як Windows, Linux, MacOS [12]. Ця платформа розширює звичайні можливості мови програмування Javascript, забезпечуючи взаємодію із пристроєм за допомогою вводу-виводу, дозволяє підключати бібліотеки, написані на інших мовах програмування, наприклад - crypto.js, бібліотека написана мовою розробки Python, використовується для шифрування.

Окрім функціоналу та бібліотек розробки, в Node.js входять утиліти для менеджменту, такі як npm - node package manager, та npx - node package executor. За допомогою першої утиліти відбувається встановлення та контроль версіонування зовнішніх бібліотек, а друга дозволяє виконувати пакетні скрипти. Такі інструменти роблять scaffolding на базі платформи Node.js зручними та простими у використанні [13].

В цьому розділі буде розглянуто приклад генерації коду та генерації проєкту, що були розроблені на базі платформи Node.js.

2.2.1 Nest CLI

Існує чимала кількість бібліотек та утиліт що здатні роботи кодову та проєктну генерацію. Наприклад, у попередньому розділі було розглянуто RoR scaffolding, що є потужним помічником під час розробки застосунку.

Nest.js - потужний фреймворк написаний на базі Node.js засобами мови typescript. Цей фреймворк покликаний розробляти з його допомогою ефективні та розширювані server-side застосунки на платформі Node.js. Він має повноцінну підтримку “з коробки” мову програмування Typescript, об’єктно-орієнтований підхід розробки, функціональне та реактивне програмування. Nest.js є надбудовою над іншою, менш потужною та менш оптимізованою бібліотекою Express, що, в свою, чергу, пропонує лише базовий набір функцій для роботи з бекенд частиною застосунка [14].

Окрім потужного набору інструментів, таких як готові модулі та класи (наприклад, для валідації, роботи з криптографією, тощо), розробники також впровадили scaffold утиліту для спрощення розробки та зниження порогу входу в нову технологію. Вона реалізована у вигляді CLI, що має набір команд, за допомогою яких можливо гнучко описати базові налаштування проєкту [14].

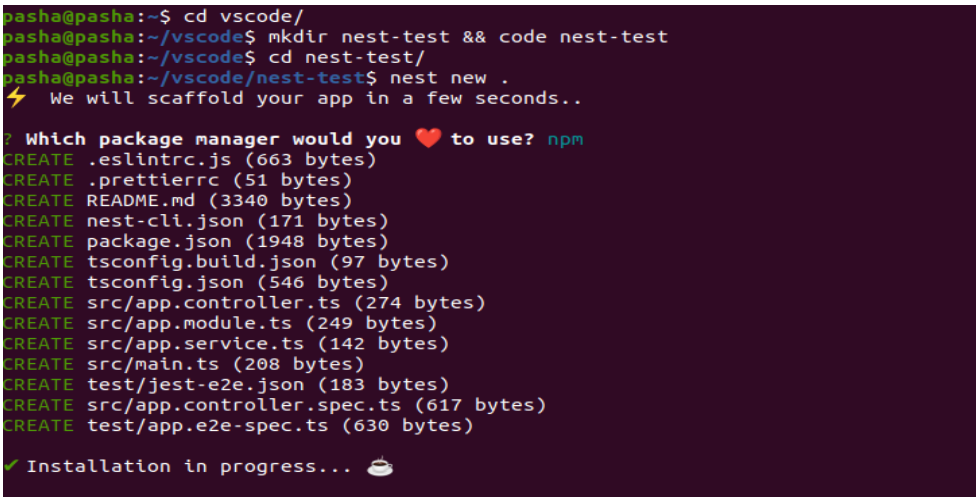


Рис. 2.3 Генерація нового проєкту засобами Nest CLI

За допомогою однієї консольної команди, розробник отримує згенерований застосунок з базовим функціоналом. Також, процес генерації передбачає налаштування Typescript, налаштування допоміжних утиліт, такий як prettier та eslint. Prettier - є стандартом розробки як серверних, так і клієнтських застосунків написаних за допомогою мов програмування Javascript або Typescript, він допомагає форматувати код відносно певного правила, щоб різні розробники мали однаковий “стиль” коду. Eslint є схожою на prettier утилітою, що, на відміну від prettier, перевіряє не стиль, а коректність написаного коду відносно заданого

розробником патерну, наприклад, перевіряє, чи була використана оголошена змінна.

Nest CLI автоматично встановлює всі необхідні залежності для роботи застосунку, такі як Express, Nest та інші. Останнім кроком, CLI ініціалізує порожній репозиторій та налаштує .gitignore файл.

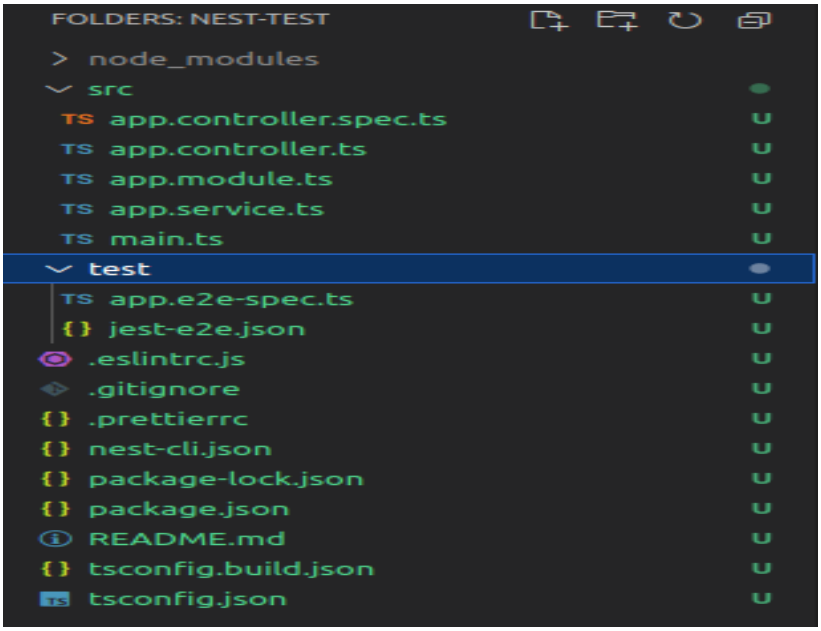


Рис 2.4 Структура згенерованого проєкту

На виході отримуємо представлену на малюнку *2.4* структуру проєкта. Проте, генерація базового налаштування це далеко не всі можливості scaffolding в Nest.js [13]. За допомогою bash аргументів, можна пропустити, або додати той чи інший крок на етапі генерації. Нижче представлений набір таких аргументів:

- --dry-run не вносити запропоновані зміни, а лише відзвітувати про них;
- --skip-git пропустили ініціалізацію порожнього гіт репозиторію;
- --skip-install попустити встановлення node_modules;
- --language обрати мову програмування Typscript або Javascript;
- --collection побудувати проєкт за готовою схемою;
- --strict налаштувати тайпскрипт із strict-параметрами.

Ці параметри є доволі узагальненими та не надають змоги обробляти специфічні випадки, проте дозволяють налаштувати початкову конфігурацію

бекенд застосунка. Окрім генерації “основи” проєкта, Nest CLI також надає можливість генерації коду за допомогою так званих схем або schematic colletion.

```
$ nest generate <schematic> <name> [options]
$ nest g <schematic> <name> [options]
```

Рис 2.5 Кодогенерації засобами Nest CLI

Нижче наведено перелік всіх доступних схематиків [13]:

- app новий застосунок в стилі monorepo;
- library нова бібліотека, або MVC модуль проєкта;
- class новий клас;
- controller новий порожній Nest.js котролер;
- decorator новий порожній декоратор, тобто така функція, що надсилаю метадані функції, котру огортає;
- filter новий порожній фільтр, тобто елемент валідації даних з контексту Nest.js;
- gateway нова єдина точка входу;
- guard новий порожній декоратор, що проводить первинну валідацію даних, зазвичай використовується в якості аутентифікації для різних ендпоінтів;
- interface новий інтерфейс;
- interceptor нова interceptor функція, що має доступ до контекста Nest.js додатку;
- middleware новий пустий middleware, тобто така функція, що має доступ до об’єктів Request/Response;
- module новий Nest.js модуль;
- pipe новий Nest.js pipe, функція для обробки потоку даних;
- provider новий Nest.js provider, Injectable клас;
- resolver новий Nest.js resolver, наприклад, для роботу з Graphql архітектурою;
- resource новий Nest.js CRUD модуль.

Згенеровані за допомогою CLI бібліотеки (nest g library <name>) будуть збережені в конфігураційному файлі nest-cli.json, що дозволить розробникам мати спільну структуру проєкту та заново генерувати необхідні модулі однією командою. Саме цей підхід взятий за основу архітектури для розробки власного проєкту scaffolding мікросервісів.

Nest.js scaffolding утиліта Nest CLI є стандартом використання фреймворку для спрощення розробки. Проте вона може бути використана лише для одного MVC фреймворку, що добре працюватиме в рамках поставленої задачі розробки монолітного бекенд додатку, тоді як для задачі генерації проєкту на базі на іншого фреймворку [14], генерація застосунку для взаємодії з базою даних ця утиліта не підходить.

2.2.2 Yeoman

Yeoman - це scaffolding застосунок, який так само використовує CLI для взаємодії з користувачем, та в основі генерації коду має так звані генератори (англ. - generators). Yeoman генератор - це застосунок, що розроблений на базі npm пакету yemoan-generator, котрий здатен за допомогою директив цієї бібліотеки генерувати проєкт за заданим шаблоном. Можливість створювати нові генератори розробниками поза штатом основної команди продукту Yeoman дозволяє розробника з усього світу розширювати базової функціонал бібліотеки для власних потреб [17].

Кожен такий генератор, розроблений як CLI застосунок використовує не команди безпосередньо, як у випадку із Nest CLI, а так звані «опитування». Користувачу надається можливість обрати один із запропонованих варіантів і на основі обраного буде згенеровано ту чи іншу кінцеву версію застосунка.

Розробка нового модуля для генерації коду є доволі простою. Для цього, необхідно встановити вищевказаний пакет yeoman-generator за допомогою менеджера пакетів npm [17]. Ініціалізація проєкту відбувається наступним чином:

```
var prompts = [{
 type: 'prompt',
 name: 'appName',
 message: 'Could you tell me the name of your new
project?',
}];

this.prompt(prompts, function (answers) {
 this.appName = answers.appName;

 done();
}.bind(this));
```

Рис 2.6 Базовий код Yeoman-генератора

Після етапу створення питань та варіантів відповідей, необхідно створити директиву. Директива - це окремий скрипт, який у випадку yeoman-generator буде відповідати кодогенерації на кожен з обраних варіантів [17]. Нижче неведено приклад директиви, що генерує нову папку та копіює конфігураційні файли:

```
app: function () {
 this.mkdir('app');
 this.mkdir('app/templates');

 this.copy('_package.json', 'package.json');
 this.copy('_bower.json', 'bower.json');
}
```

Рис 2.7 Yeoman-директива

Засобами бібліотеки yeoman можливо робити як проєкту генерацію, так і генерацію коду.

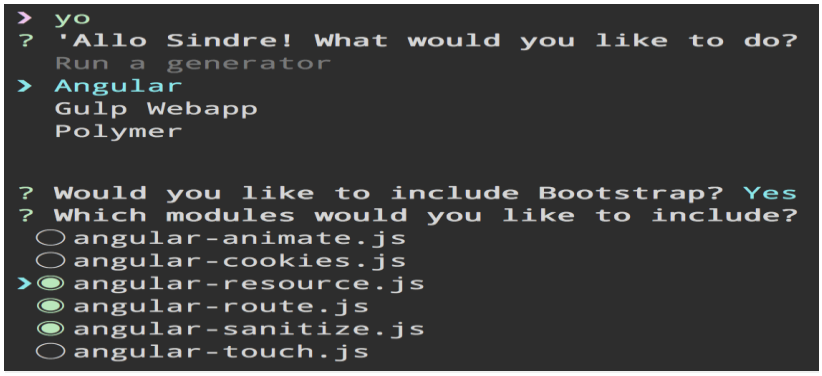


Рис. 2.8 Приклад генерацію коду для фреймвороку Angular за допомогою Yeoman

Бібліотека розроблена засобами мови Javascript на базі Node.js, та генератори можуть бути так само написані лише з використанням цієї мови програмування. Yeoman також може генерувати код на інших мовах програмування, і загалом будь-що, що може бути написане за допомогою Javascript та її ресурсів.

Yeoman є застосунком-обгорткою, що, фактично, уніфікує використання генераторів, тобто створює єдину точку входу для будь-яких потреб в генерації архітектури та модулів проєкту. Проте, навіть попри широкий вибір готових yeoman-generators, не завжди можна знайти такі, що задовільнять конкретну специфіку задачі [17].

2.3 Постановка задачі

Аналіз існуючих рішень надає широку картину переваг та недоліків генерації коду на платформі Node.js. Такі застосунки як Nest CLI вирішують лише одну конкретну проблему, таку як генерацію архітектури нового проєкту або його атомарного модуля в рамках фреймворку Nest.js. Попри те, що готові модулі цього фреймворку підтримують розробку мікросервісів, наприклад, створення message queueing транспортного протоколу за допомогою різних брокерів подій, або ж підтримку обробки розподілених транзакцій, як наприклад SAGA оркестрація, всі ці налаштування наразі є недоступними в рамках Nest CLI генерації.

На противагу генерації в рамках окремого фреймворку, така утиліта як Yeoman має набагато більш широкий спектр використання, до прикладу, генерація фронтед та бекенд застосунків на різних мовах програмування. Проте, наразі не було створено такого yeoman-генератора, що вирішував би задачу генерацію мікросервісної архітектури та окремих сервісів.

Цілком можливо розробити необхідний генератор засобами утиліти Yeoman, що вирішував би проблему генерації мікросервісів, проте такий підхід має декілька недоліків. По-перше, необхідно повністю підв'язуватись під можливості Yeoman, тобто у разі припинення підтримки коду, генератор буде так само важко підтримувати. По-друге, можливості SDK є обмеженими і нерідко застарілими, тоді як Node.js має ширший та новіший функціонал.

Таким чином, розробка застосунку для генерації мікросервісів, що є глобально незалежним від сторонніх бібліотек, є більш оптимальним і стабільним рішення, адже у цьому випадку, єдина технологія, що є необхідною для «життя» проєкту - це платформа Node.js. Ця платформа з часом все більше і більше набирає популярності, що означає її постійну підтри з боку розробників. Тому, кінцевим варіантом застосунку для генерації мікросервісів, стало об'єднання кращих практик з кожної дослідженої утиліти - простота у використання за допомогою CLI команд, та узагальнений широкий підхід, заснований на філософії yeoman-generators.

2.4 Архітектура програмного застосунку

Перед початком розробки власного scaffolding застосунка для генерації мікросервісів, необхідно визначитись з архітектурою. Головна ідея полягає в тому, аби створити максимально зрозумілий користувацький інтерфейс за допомогою CLI, в якому не буде потреби довго розбиратись. Тому з метою уникнення необхідності проходити "опитування" як у випадку роботи застосунку Yeoman, було вирішено описувати всі необхідні конфігурації в одному джерелі - конфігураційному файлі. Опис конфігурацій у єдиному файлі запозичена з підходу Iac (англ. - Infrastructure as a Code). Iac (англ. – Infrastructure

as a Code) [18], або інфраструктура як код, це підхід для опису та керування інфраструктурою хмарних сервісів за допомогою конфігураційних файлів. Взявши за приклад ситуацію, коли Devops інженеру треба налаштувати певну кількість серверів для розгортання майбутнього додатку, проблеми навряд виникнуть, якщо таких серверів лише декілька. Проте, у випадку їх кратного чисельного збільшення до декілька десятків, налаштування займатиме велику часу, а ризик виникнення помилки під час такого налаштування зросте в рази. Для вирішення такої проблеми, був розроблений застосунок terraform, що за допомогою консольних командами і конфігураційного файлу розгортає будь-яку кількість серверів.

Говорячи про мікросервісну архітектуру, маємо дуже схожу проблему - велика кількість окремих застосунків із власною екосистемою, кожен з яких необхідно налаштовувати незалежно один від одного. На етапі проєктування, передбачалось, що архітектура матиме наступний вигляд:

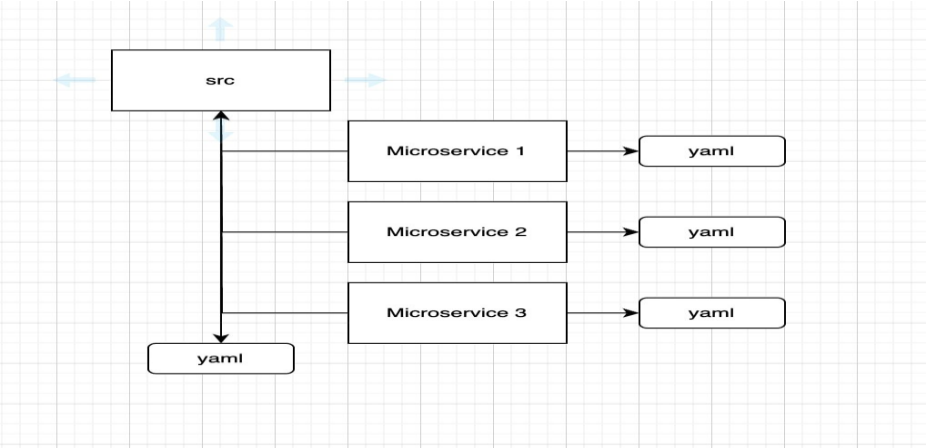


Рис. 2.9 Первісна архітектура застосунку

На етапі підготовки до генерації застосунку, користувач мав би створити кореневу папку проєкту, головний конфігураційний файл для налаштування, та необхідну кількість мікросервісів із власним конфігураційний файлом кожного сервісу безпосередньо, та на етапі генерації отримати наступний результат:

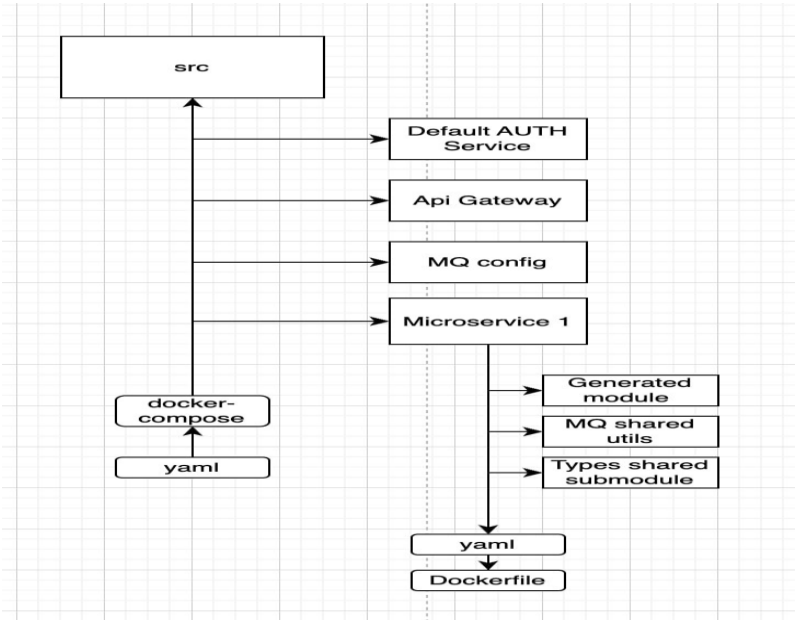


Рис.2.10 Архітектура згенерованого проєкту

Як показано на рисунку *2.10*, в кореневій папці згенеровано загальний docker-compose.yaml файл, для того, аби була можливість запустити всі сервіси однією командою, згенеровано gateway, конфігурації брокерів повідомлень, та безпосередньо мікросервіс, що відповідатиме обраній специфікації. Кожен з цих параметрів можливо обирати під потреби конкретного проєкту та сервісу, що дозволяє використовувати scaffold застосунок достатньо гнучко.

Проте, під час роботи над застосунком, багато первісних рішень було замінено на більш оптимальні. Наприклад, після дослідження вище описаного Iac підходу, велику кількість менших конфігураційних файлів було прийнято рішення об’єднати в єдиний файл, що відповідатиме за налаштування проєкту та кожного сервіса. Таким чином, весь додаток можна буде контролювати з “одного джерела” і користувачу не доведеться робити зайву роботу, таку як створення різних папок та файлів

Друга важлива зміна, що була впроваджена – це зміна формату конфігураційного файлу. На рисунку *2.10* зазначено, що планувалось використати .yaml розширення. Yaml - це абревіатура, що англійською розшифровується як “YAML Ain't Markup Language”, і дослівно перекладається як “YAML - не є мовою розмітки. Такий формат як .yaml або його скорочена

версія .yml - це формат серіалізації даних, який, хоч семантично і схожий на мови розмітки, не є такою. Його головне покликання це бути максимально зрозумілим людині, універсальним та мати схожі структури даних до більшості мов програмування, такі як масиви, об’єкти, тощо. Проте, в даному випадку більш підходящим варіантом є формат файлу .json. Абревіатура розшифровується як Javascript object notation, текстовий формат обміну даними, в основі якого є саме мова програмування Javascript.

І хоча .yaml є більш універсальне рішенням, адже не є прив’язаним до жодної мови програмування, .json є більш підходящим, адже він є ідеально сумісним з мовою програмування Javascript та не має жодної необхідності встановлювати якісь додаткові парсери формату, як би це було у випадку з першим вибором.

З урахуванням прийнятих рішень, користувачу необхідно створити лише один файл з описом конфігурації проєкта та необхідних мікросервісів. Була обрана робоча назва розробленого застосунку “microgen”.

Головний конфігураційний файл обов’язково матиме назву *microgen.json*. Як можна побачити, його структура є простою та зрозумілою, вона поділяється на два основних блоки - загальні конфігурації проєкта, тобто такі налаштування, що є спільними, та масив, що містить повторювану за структурою конфігурацію кожного мікросервісу.

Перший блок загальних конфігурації:

- `name` назва проєкту, не впливає на подальшу генерацію;
- `externalModules` зовнішні мікросервіси, які потенційно можуть підходити під теперішній проєкт, наприклад, сервіс для аутентифікації, бібліотека утиліт, тощо. Будуть скопійовані в кореневу папку проєкта;
- `gateway` приймає значення так/ні, додає до мікросервісного застосунку API gateway;
- `eventBroker` генерує директорію з налаштуванням для брокера подій або черги повідомлень, можливо обрати поміж двох варіантів (RabbitMQ, Kaffka).

Рис. 2.11 Приклад microgen.json файла

Другий блок конфігурацій мікросервіса:

- name назва мікросервіса та його директорії відповідно;
- framework фреймворк, який буде в основі сервіса, можливо обрати поміж двох варіантів (Express, Nest.js);
- language мова програмування, можливо обрати поміж двох варіантів (Javascript, Typescript);
- deps основні npm пакети, що будуть встановлені під час генерації;
- devDeps допоміжні npm пакети, що будуть встановлені під час генерації;

- dockerFile приймає значення так/ні, генерує Dockerfile для мікросервіса;
- database генерація та підключення бази даних, що буде розгорнута як docker image;
- driver конкретна субд, можливо обрати поміж двох варіантів (MySQL, PostgreSQL);
- orm встановлення та генерація для orm, можливо обрати поміж двох варіантів (Prisma, Typeorm).

Для старту генерації необхідно запустити команду *npm run generate - - - path=/path-to-microgen.json*. Консольна змінна -path вказує на абсолютний шлях до кореневого конфігураційного файлу.

2.5 Опис використаних технологій

Як мову програмування, було обрано саме Typescript, а не Javascript, адже перша пропонує більший функціонал для роботи, хоч і є, фактично, лише надбудовою над другою. Javascript, в свою чергу, кросплатформена мова програмування, котра, як прийнято вважати, обробляється в одному потоці за допомогою моделі обробки асинхронних операції, так званого циклу подій (англ. - event loop). Асинхронні операції, це такі операції, час на виконання яких неможливо точно визначити (на відміну від синхронних), тому необхідно “зачекати” до того моменту, як така операція поверне відповідь. До етапу відповіді, асинхронний код повертає так званий Promise замість конкретних даних. Promise, в перекладі з англійської означає “обіцянка”, є гарантією того, що рано чи пізно будуть отримані або дані, або помилка.

Розробка будь-якого програмного застосунку за допомогою Typescript має свій ряд переваг та недолік. З недоліків варто зазначити необхідність більшої витрати часу на розробку безпосередньо. Проте, для того застосунку як microgen, критично важливим є правильний, односторонній потік даних та “виразний” опис кожної окремої сутності проєкту. Такий вибір мови програмування

дозволяє підтримувати існуючий код, розширювати існуючий функціонал, мінімізує кількість помилок пов’язаних з неоднорідністю кодової бази.

Посилаючись на розділ 2.4, у якому була зазначена команда для старту генерації, необхідно звернути увагу на консольні аргументи, що передаються під час ініціалізації, а саме змінну *path*. Оскільки скрипт має бути універсальним, і повинен дозволяти генерацію для будь-якої файлової системи, без прив’язки до конкретної директорії, необхідно надати змогу користувачу динамічно зазначати абсолютне розташування microge.json файлу. І для того, щоб отримати такий шлях до файлу, необхідно передати консольні аргументи і розпарсити їх. Для обробки такого випадку, було прийнято рішення відмовитись від сторонніх бібліотек, та запровадити власне рішення, по причині того, що всі існуючі варіанти не працюють належним чином. Розробка такого функціоналу передбачає необхідність виділення двох типів даних, що надходять із консольних аргументів - змінні, тобто такі одиниці даних, що містять довільне строкове значення, та флаги, що можуть приймати значення так або ні.

Імплементація методу має наступний вигляд:

```
export const parseArgs = <T>(args: string[]): ParsedArgs<T> => {
 const slicedArgs: string[] = args.slice(1);
 const parsedArgs: ParsedArgs<T> = {
  flags: [],
  variables: {} as T,
 };
 for (const arg of slicedArgs) {
  if (arg.startsWith(«—«)) {
   const rawArg: string = arg.replace(«—«, «»);
   parsedArgs.flags.push(rawArg);
  } else if (arg.startsWith(«-«)) {
   const [key, value]: string[] = arg.replace(«-«, «»).split(«=»);
   parsedArgs.variables = {
    ...parsedArgs.variables,
```

```
      [key]: value,
    };
  }
}
return parsedArgs;
};
};
```

Рис 2.12 Функціонал для читання консольних аргументів

Для генерації файлів, копіювання, зчитування, встановлення залежностей, та інших операції, необхідно було обрати модулі їх виконання. В платформі Node.js існують дві вбудовані бібліотеки які добре підійдуть для виконання поставленої задачі - fs та node:child_process.

Назва бібліотеки fs є абревіатурою, що розшифровується як file system(англ. - файлова система). Вона має обширний перелік операції, зокрема створення нового файлу, створення нової директорії, запис та читання з файлу, додавання тексту до файлу, перевірка на існування ресурсу, та багато інших функцій. Також, окрім класичної версії бібліотеки, існує її модифікована версію, так звані fs:promises, що дозволяє обробляти асинхронні операції за допомогою синтаксису async/await, котрий з’явився у специфікації ES6 мови програмування Javascript, та є одним з методів роботи з Promise. Рішення обрати саме модифікований варіант бібліотеку гарантує конзистентність даних, тобто запобігає ситуації початку виконання наступного кроку асинхронної операції до завершення виконання попередньої.

Для виконання такого функціоналу, як обробка консольних команд було обрано програмний модуль платформи Node.js node:child_process дозволяє створювати синхронні та асинхронні процеси, що будуть оброблятись окремо від основного потоку. Ця бібліотека має велику кількість функціоналу, головним з яких є три функції. Функція *spawn* створює абсолютно новий дочірній процес, в якому буде оброблятись зазначена консольна команда. API цієї функції дозволяє передавати необхідні аргументи до команди динамічно [19]. Функція *fork* є дуже схожою на функцію *spawn*, проте вона створює не унікальний дочірній процес, а

копіює вже існуючий. Такий підхід є корисним у випадку необхідності клонування контексту виконання процесу. В свою чергу, функція *exec* використовується для виконання будь-якої консольної команди без створення ресурсоємного потоку. Для поставленої задачі, а саме виконання консольних команд, така функція підходить найкраще.

Для логування даних була обрана зовнішня бібліотекою Winston, що імплементує головні методи виводу даних в консоль, відповідно до патерну Logger, такі як *log, info, warn* та *error*. Також, вона записує повну інформацію про помилку в окремому файлі, що стає в нагоді під час розробки скрипта. Логування, в рамках цього проєкту, є необхідних, аби користувач міг повністю відслідковувати інформацію про процес виконання скрипта та виправити потенційні помилки.

Таким чином, обрані технології повноцінно покривають потреби для розробки scaffold застосунку. Важливо зазначити, що зазначений перелік обраних технологій дозволяє базуватись виключно на популярних бібліотеках, що є частиною стабільної платформи Node.js і робить додаток незалежних від зовнішніх постачальників програмного забезпечення.

2.6 Модулі програмної реалізації застосунку

У цьому розділі описано, які рішення були впроваджені під час розробки програмного забезпечення та аргументовано причину їх використання. Організація роботи такого застосунку заснована по принципу переходу від найменшої задачі, до найбільшої. Такий підхід є необхідним, адже кожний наступний крок в генерації є фактично надбудовою над попереднім. Для спрощення розуміння та читабельності коду, сам процес генерації було розбито на атомарні модулі, кожен з яких відповідає виключно за конкретну одиницю функціоналу.

2.6.1 Валідація конфігураційного файлу

Перший етап передбачає парсинг та валідацію головного конфігураційного файлу *microgen.json*. Читання файлу відбувається засобами бібліотеки fs:promises та записом даних в RAM. Оскільки формат файлу є .json, не має необхідності в додаткових кроках, адже такий формат добре підходить для обробки мовою Javascript.

Валідація файлу впроваджується окремою функцією, та є простою перевіркою, на відповідність даних заданому шаблону. Таким чином, відбувається перевірка, чи заповнені всі обов'язкові поля в загальній конфігурації проєкту або окремого мікрсервісу, чи існує імплементація обраного варіанту. У випадку невідповідності даних такому шаблону, користувачу буде виведено повідомлення про помилку, в якій буде описано, що необхідно змінити для продовження роботи.

2.6.2 Збереження загальної конфігурації проєкту

Оскільки конфігурація генеруємого проєкту є спільної і повторюваною, прийнято рішення зберігати такі дані в спільній для всього коду структурі даних, котра була названа __PROJECT_METADATA__.

```
export const __PROJECT_METADATA__: ProjectMetadata = {
  mainFileName: "microgen.json",
  projectPath: "",
  projectFolderName: "",
  microservices: [],
 };
```

Рис.2.13 Порожній об'єкт __PROJECT_METADATA__

На етапі парсингу файла, конфігураційні дані записуються в такий об'єкт, що є необхідним для отримання інформації про налаштування в будь-якій точці кодової бази.

```
__PROJECT_METADATA__.gateway = parsedMainConfigFile.gateway;
```

```
__PROJECT_METADATA__.externalModules =
parsedMainConfigFile.externalModules;
__PROJECT_METADATA__.eventBroker = parsedMainConfigFile.eventBroker;

const microservicesConfigs: MicroserviceConfig[] =
  parsedMainConfigFile.microservices;

for (const config of microservicesConfigs) {
  const deepPath = join(path, config.name);

  __PROJECT_METADATA__.microservices.push({
    ...config,
    language:
      config.language ??
      (config.framework === Frameworks.NEST && Languages.DEFAULT),
    exists: fs.existsSync(`${deepPath}/package.json`),
    folderName: config.name,
    absolutePath: deepPath,
  });
```

Рис. 2.14 Запис конфігурації

Таким чином, ми можемо використати дані про налаштування, наприклад, окремого мікросервісу будь-де в проєкті і гарантовано отримаємо одні і ті самі дані, без необхідності читати microgen.json заново, що суттєво заощаджує ресурси.

2.6.3 Генерація коду засобами Node.js

Першочергово, важливо розглянути процес генерації коду засобами Node.js безпосередньо. Генерація коду відбувається в рамках атомарного модуля, що представлена як окрема функції генерації. Для виконання консольних команд розроблено клас ChildProcessBuilder, що є імплементацію патерну Builder [19]. Це такий патерн, що за породжує шаблон об’єкту, або, як у випадку генерації

коду, шаблон консольної команди, що потім буде виконана за допомогою метода exec.

```
export class ChildProcessBuilder {
  private query: string = "";
  private readonly executionHandler = null;
  constructor(customHandler?: any) {
    this.executionHandler = customHandler;
  }

  append(command: string): typeof this {
    if (!command) {
      return this;
    }
    if (!this.query) {
      this.query = command;
    } else {
      this.query = `${this.query} && ${command}`;
    }
    return this;
  }

  exec(): void {
    logger.info(`Executing query: ${this.query}`);
    nodeExec(this.query, (error, stdout) => {
      stdout && logger.info(stdout);
      error && logger.error(error);
    });
  }

  async execAsync(): Promise<any> {
    return util.promisify(require("node:child_process").exec)(this.query);
  }
}
```

Рис. 2.15 Імплементація класу ChildProcessBuilder

За допомогою методу *append* клас об'єднує консольні команди в одну та зберігає їх у вигляді строки. Цей клас має на меті виконання ланцюга команд послідовно, оскільки процес створений методом exec є ізольованим, і такий підхід - єдина можливість виконати набір команд послідовно. Наприклад, для створення директорії та ініціалізації нового Node.js проєкту команди мають бути виконані в рамках одного процесу, для доступу до контексту виконання попередньої команди.

Наступна проблема, яка виникла під час розробки застосунку, є асинхронність виконання команд. Вищевказана команда exec працює таким чином, що час на виконання команди є невизначеним, проте виклик цього самого методу виконується синхронно. Тому, на етапі генерації виникала ситуація, що директорія ще не встигла створитись, проте виконання команди по ініціалізації нового Node.js проєкту вже стартувало, що, очевидно, призводило до конфлікту та помилок. Для вирішення такої проблеми, були використані різні підходи перший з яких був невдалим. Він полягав в тому, щоб викликати кожну наступну

функцію генерації відкладено, тобто пізніше на декілька секунд за попередню за допомогою нативного методу setTimeout. Цей вбудований метод дозволяє викликати синхронну функцію не одразу, а через зазначений час. Проте, це рішення не є стабільним, оскільки немає гарантії, що вказаний час відкладеного виконання є достеменно правильним, а справжній час роботи консольної команди може займати різну кількість часу. Проте, було впроваджено кращий варіант обробки такої ситуації, за допомогою пакету utils.promisify. Цей вбудований пакет працює схоже до fs:promises і дозволяє обробляти проміси за допомогою async/await синтаксису, що дозволяє очікувати нового завершення виконання консольної команди. Нижче наведено приклад створення команд для кодогенерації за допомогою ChildProcessBuilder:

```
export const initializeNodeProjects = (): Promise<any>[] => {
  const promises = __PROJECT_METADATA__.microservices
    .filter(({ framework }) => framework !== Frameworks.NEST)
    .map(({ absolutePath }) => {
      return new ChildProcessBuilder()
        .append(osExecutableCommands.changeDirectory(absolutePath))
        .append(npmExecutableCommands.npmInit())
        .execAsync();
    });

  logger.info("Initializing Node projects");

  return promises;
};
```

Рис.2.16 Генерація директорій та ініціалізація порожнього Node.js проєкту

Оскільки розробка розподіленої системи передбачає створення більше ніж одного сервісу, було прийнято рішення всі проміси, що відповідають за виконання команд повертати із зазначеної функції та обробляти нативним методом Promise.all. Такий стандартний підхід обробки ланцюга промісів існує для того, щоб обробити таку послідовність як один проміс, іншими слова зачекати, поки кожен з них виконається.

Список консольних команд, було описано у вигляді пари ключ-значення, де ключ – назва команди, а значення функція, яка повертає текстову строку із

сигнатурою команди. Вони розділені на два типи - npm команди та команди os. Npm список команд відповідає за весь менеджмент, що пов’язаний із Node.js та npm, тоді як OS список відповідає за команди операційної системи, такі як створення директорії, копіювання файлів, тощо. Такий підхід дозволяє вибудувати гнучку систему розширення, і у разі необхідності додавання нових директив генерації, розширити вже існуючий функціонал шляхом додавання нової пари ключ-значення.

```
export const npmExecutableCommands: NpmCommands = {
  npmNoop: () => "",
  npmInit: () => "npm init -y",
  npmInstallDeps: (deps: string[]) =>
    deps.length ? `npm i ${deps.join(" ")}` : "",
  npmInstallDevDeps: (deps: string[]) =>
    deps.length ? `npm i --save-dev ${deps.join(" ")}` : "",
  npmInitExpress: () => "npm i --reset-cache express cors",
  npmInitNest: () => `npm i -g @nestjs/cli && nest new . -p npm `,
  npmInitTypeScript: () => "npm i --save-dev @types/node typescript",
  npmInstallExpressTypes: () => "npm i --save-dev @types/express",
  npmInstallPgDriver: () => "npm i pg",
  npmInstallPgTypes: () => "npm i --save-dev @types/pg",
  npmInstallMySqlDriver: () => "npm i mysql",
  npmInstallMySqlTypes: () => "npm i --save-dev @types/mysql",
  npmInstallPrisma: () => "npm i @prisma/client",
  npmInstallTypeorm: () => "npm i typeorm",
  npmInitPrisma: (database: string) =>
    `npx prisma init --datasource-provider ${database}`,
  npmInstallRabbitmq: () => `npm i amqplib`,
  npmInstallRabbitmqTypes: () => `npm i --save-dev @types/amqplib`,
  npmInstallKafka: () => `npm i kafkajs`,
  npmInstallKafkaTypes: () => `npm i --save-dev @types/kafkajs`,
};
```

```
const ifExists = (
  path: string,
  executableCommand: string,
  typeFlag: ObjectTypes
) => {
  return `if [ ${typeFlag} ${path} ]; then
      echo "${path} exists."
    else
      ${executableCommand}
    fi`;
};

export const osExecutableCommands: OsCommands = {
  changeDirectory: (path: string) => `cd ${path}`,
  copyFile: (toCopy: string, toInsert: string, fileName: string) =>
    ifExists(
      join(toInsert, fileName),
      `cp -i -u ${toCopy} ${toInsert}`,
      ObjectTypes.FILE
    ),
  copyDirectory: (toCopy: string, toInsert: string) =>
    `cp -r -u ${toCopy} ${toInsert}`,
  renameFile: () => ``,
  initGit: () => `git init`,
  createDirectory: (dir: string) => `mkdir -p ${dir}`,
  rmRf: (dir: string) => `rm -rf ${dir}`,
  osNoop: () => "",
};
```

Команди, що використовуються для генерації коду

Кожна з цих команд описана у вигляді функції, а не у вигляді строки, тому що деякі командам для роботи необхідно динамічно приймати аргументи. Завдяки такому підходу та можливостям мови програмування Typescript, це дозволяє уникнути помилки, у випадку якщо у функцію буде передано аргумент, що не буде використаним. Таким чином, можливо співставити варіанту з конфігураційного файлу відповідну директиву, наприклад, ініціалізацію фреймворку, або мови програмування.

```
const frameworkConfigs: Record<string, CallableFunction> = {
  [Frameworks.EXPRESS]: npmExecutableCommands.npmInitExpress,
  [Frameworks.NEST]: npmExecutableCommands.npmInitNest,
  [Languages.TS]: npmExecutableCommands.npmInitTypeScript,
  [Languages.JS]: npmExecutableCommands.npmNoop,
  [Languages.DEFAULT]: npmExecutableCommands.npmNoop,
};      Pasha, last month • Added validation and nest generatio
```

Рис. 2.17 Приклад використання словника npm команд для ініціалізації фреймворка

Головна проблема, з якою довелось стикнутись під час розробки вищевказаної логіки полягає у події, коли файл вже був створений або скопійований, але етап генерації передбачаю аналогічну повторну дію. Наприклад, під час генерації нового сервісу, файли були створені лише частково, з помилкою, тому користувач запускає скрипт другий раз, аби перенести та створити всі ті файли, що залишились. Для цього було розроблена функція-обгортка ifExists. По суті, вона є звичайною частиною bash команди, що перевіряє, чи вже існує такий файл або директорія, і не створює, якщо він вже існує. Такий підхід було обрано на противагу видаленню та створенню нового файлу для запобігання втручання скрипту в ті зміни, що потенційно могла внести розробника, надавши їм вищого пріоритету за генеративні зміни.

Для роботи сервісу, окрім загальних конфігурацій, необхідні ще конкретні файли з кодом, які будуть «скелетом» майбутнього мікросервісу. Для цього було прийнято рішення власноруч розробити такий набір коду, названий ресурсами, або англійською assets, що буде автоматично скопійований в майбутній проєкт користувача. Такі ресурси містять як різноманітні конфігураційні файли, такі як .gitignore, tsconfig.json та інші, так і шматки готового коду, такі як налаштування бази даних, MVC файлова структура, налаштування брокера повідомлень. Повний перелік доступних конфігурацій описані у розділі 2.3.

З причини того, що проєкт написаний на мові програмування Typescript, який в свою чергу транспілюється, тобто перекладається в Javascript, всі файли, що мали розширення не .js або .ts не потрапляли в кінцевий Javascipt bundle, тобто побудований проєкт, готовий до використання. Такий підхід є

стандартною поведінкою tsc білдера, за допомогою якого Typescript перетворюється в Javascript, тому необхідно було розробити підхід, аби всі ресурси проєкту зберігались в кінцевому продукті. Існує декілька варіантів, як наприклад використання додаткових білдерів, таких як webpack, що дозволяють зберігати файли, що не містять Javascript або Typescript коду. Окрім білдерів, можливо впровадити генерацію таких файлів на етапі білда microgen застосунку динамічно, наприклад, за допомогою unix команди cat “text” >> echo “something.extc”. Проте це рішення є громіздким та не універсальних, адже залежить від операційної системи, де буде використаний скріпт. Тому, було прийнято рішення на етапі білда проєкту, копіювати всю директорію із Typescript директорії в аналогічну директорію Javascript білда.

Вибудувана архітектура проєкту дає чималу кількість переваг, таких як стабільність та безвідмовність роботи, універсальність, а головне - розширюваність, адже у розроблену кодову базу дуже легко додати нові фреймворки та конфігурації.

Розділ 3

3.1 Аналіз розробленого програмного забезпечення

В результаті розробки програмного забезпечення було досягнено основної мети по створенню стабільного застосунку, що є мінімально залежним від сторонніх бібліотек та має простий у використанні інтерфейс.

Єдина “точка входу”, що представлена конфігураційним файлом, дозволяє не тільки полегшити процес розробки на початкових етапах, а і легко керувати розширенням продукту, що використовує утиліту microgen. Наприклад, першочергові бізнес умови проєкту вимагали створення лише двох сервісів, під назвою user та bookings. Тоді, конфігураційний файл та результат роботи застосунку матимуть наступний вигляд:

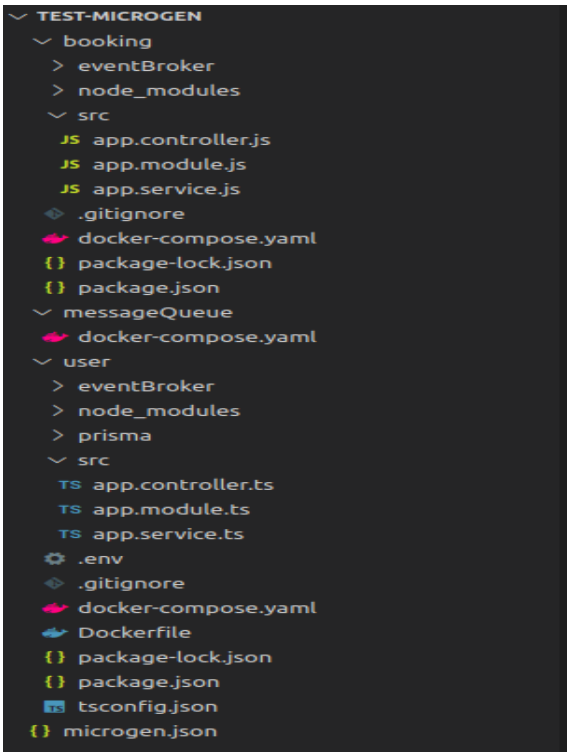


Рис. 3.1 та 3.2 Приклад конфігураційного файла та результат роботи microgen

Окрім мови програмування, в конфігураціях сервісів був обраний фреймворк Express, необхідні node_modules пакети, що треба встановити на етапі генерації, бази даних postgresql та mysql із додатковими ORM prisma та typeorm

відповідно. Для мікросервісу під назвою user також був обраний параметр dockerFile що відповідає за створення контейнера для згенерованого сервісу. Як зазначено на рисунку *3.1*, бажані сервіси були згенеровані з дотриманням конфігураційних вимог.

Окрім основних описаних конфігурацій, в сервіси було також додано сервіс для взаємодії із чергою повідомлень Apache kafka, зазначеного в загальній конфігурації проєкту.

У випадку якщо бізнес вимоги до проєкту будуть розширені, та з’явиться необхідність створення нового мікросервісу, наприклад, засобами фреймворку Nest.js, додавши в конфігураційний microgen.json файл опис такого сервісу та повторно використавши CLI команду старту генерації, сервіс буде додано, а всі інші, що вже були згенеровані, не матимуть жодних змін, адже всі сервіси, що містять package.json файл будуть проігноровані для повторної генерації. Такий підхід є необхідним для запобігання програмного втручання в зміни внесені розробниками. Проте, будуть проігноровані лише декілька ключових етапів, таких як ініціалізація фреймворку чи мови програмування, тоді як зміни конфігурації вже існуючих сервісів, наприклад, додавши нові node_modules пакети або необхідність створити Dockerfile, такі зміни будуть додані. Ці зміни не є «конфліктними» лише у випадку додавання раніше не існуючих файлів, якщо файл вже існує в сервісів, зміни також будуть проігноровані застосунком microgen. Тоді, результат роботи матиме наступний вигляд:

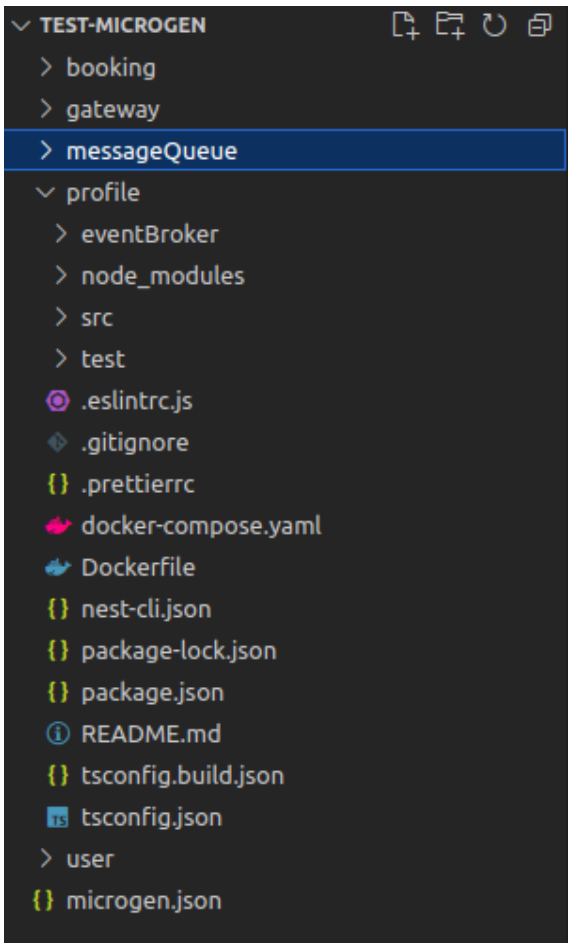


Рис. 3.3 та 3.4 Розширений конфігураційний файл та результат повторної генерації коду

Такий підхід дозволяє дуже гнучко контролювати продукт, розширення кодової бази, внесення змін у вже існуючі сервіси, встановлення залежностей лише за допомогою зміни конфігураційного файлу, тощо.

Логування даних є також стійкою перевагою розробленої утиліти. Під час виконання процесу генерації коду, кожна дія логується та виводиться в терміналі. Окрім повідомлень, доданих вручну, тобто таких повідомлень, що описують теперішню дію, реалізований вивід системних логів сторонніх бібліотек, що дозволяє отримувати всю актуальну інформацію про етапи генерації .

3.2 Порівняння з існуючими рішеннями - переваги і недоліки

Посилаючись на розділ 2.2 варто зазначити, що повноцінних аналогів утиліт, що виконують функцію генерації мікросервісної архітектури засобами Node.js не існує. Найближчі відповідники, описані у розділах 2.2.1 та 2.2.2 є Nest CLI та Yeoman. Проте, перший з вищезазначених ресурсів здатен виконувати задачу кодогенерації лише для монолітної архітектури в рамках окремого фреймворку Node.js, тоді як другий наразі просто не має необхідного генератора для виконання такої функції. Такий генератор може бути розроблений та мати відповідний функціонал для кодогенерації мікросервісів, проте дане рішення не є оптимальним. Бібліотека Yeoman є лише загальною “обгорткою”, що дозволяє виконувати генератор, не надаючи жодного готового функціоналу необхідного для виконання умов поставленої задачі. Тому, прив’язка до сторонньої бібліотеки є надлишковим рішенням, що не надасть фактичних переваг, проте спричинить ситуацію так званого vendor-lock, тобто прив’язки до постачальника послуг.

Розроблений додаток microgen має ряд вагомих переваг, таких як незалежність від бібліотек, що є нестабільними у підтримці, але і використовує вже розроблені рішення, такі як Nest CLI або Prisma CLI. Такий підхід дозволяє позбутись зайвої розробки додаткового коду та ресурсів застосунку, згенерувавши скелет мікросервісу, що використовує фреймворк Nest.js або ж ініціалізацію конфігураційного Prisma файлу за допомогою однієї команди без необхідності власноруч прописувати код. Гнучка архітектура застосунку microgen дозволяє швидко додавати нові ресурси та розширювати функціонал.

Проте розроблений додаток також має набір недоліків. Наприклад, наразі функціонал додатку є обмеженим певним набором технологій, тобто він здатен працювати лише з двома мовами програмування, двома фреймворками - Express та Nest.

Ще один з вагомих недоліків є відсутність функціоналу збереження версіонування (попереднього стану) проєкту. Підхід Iac та утиліта terraform, з

яких була запозичена ідея єдиного конфігураційного файлу як основи генерації проєкту, мають функціонал, що дозволяє порівнювати попередню версію застосунку із теперішньою конфігурацією на наступному етапі генерації. Наприклад, такий функціонал є корисним для видалення конфігурацій, таких як зміна бази даних, відмова від використання docker-compose.yaml, тощо. Проте, цей функціонал не є імплементованим, через обмеження пов’язані із «ручними» змінами, що можуть вносити розробники у вже згенерований мікрсервіс. Наприклад, у разі необхідності зміни мови програмування такі впровадження зачеплять вже існуючий функціонал у разі використання суміжних мов програмування, таких як Javascript та Typescript. А за необхідності повної міграції на іншу мову програмування потрібно створювати транспілятори, тобто перекладачі мов програмування.

Підсумовуючі, варто зазначити, що розроблена утиліта не має чітких відповідниках в рамках поставленої задачі. Вона спрощує та пришвидшує розробку в мікросервісів на базі платформи Node.js, що можуть бути згенеровані на базі найпопулярніших технологій цієї платформи. Проте, можливості застосунку обмежуються використанням невеликого переліку найпопулярніших технологій та відсутністю версіонування.

Висновки

У даній роботі було проведено огляд переваг та недоліків мікросервісної архітектури, розглянуто бібліотеки для scaffolding у Node.js, розроблено та описано застосунок для автоматизованої генерації кодової бази для мікросервісної архітектури.

Мету роботи було досягнуто, отриманий застосунок для кодогенерації виконує поставлену задачу – на основі простого CLI та єдиного конфігураційного файлу утиліта здатна генерувати кодову базу проєкту та контролювати додавання нових сервісів та конфігурацій до вже існуючої кодової бази. Програма microgen значно пришвидшити процес розробки та налаштування на початковій стадії розробки проєкта, є простою у використанні, стабільною та масштабованою. Проте, розроблений застосунок має пункти, які необхідно покращити, зокрема, розширення можливості вибору конфігурації - різних бібліотек, баз даних, технологій та мов програмування. Наразі, імплементація застосунку передбачає використання лише найбільш популярних бібліотек в рамках платформи Node.js.

Отримані результати роботи є корисними для генерації кодової бази на початкових етапах життя продукту. Такий застосунок може бути корисним тим інженерам, що обрали розробку бекенд частини додатку в рамках мікросервісної архітектури, та бажають автоматизувати процес налаштування задля економії часу та запобігання виникненню механічних помилок.

Список використаних джерел

1. Hoang C. Monolith Architecture. *Medium*.
URL: https://tech.tamara.co/monolith-architecture-5f00270f384e
2. Nagpal A. Monolithic vs Microservices Architecture: Advantages, Disadvantages, And Differences. *Medium*.
URL: https://medium.com/@jasminepuno/monolithic-vs-microservices-architecture-advantages-disadvantages-and-differences-2bee6d1da8ca#:~:text=Monolithic%20architecture%20is%20a%20conventional,closely%20connected%20and%20centralized%20system.
3. Fowler M. Monolith First. *martinfowler.com*.
URL: https://martinfowler.com/bliki/MonolithFirst.html.
4. Global Logic. Мікросервісна архітектура для початківців. Частина I. *GlobalLogic Ukraine*.
URL: https://www.globallogic.com/ua/insights/blogs/microservices-architecture-for-beginners-part-one/
5. Microservices.io. Microservices Pattern: Pattern: Database per service. *microservices.io*. URL: https://microservices.io/patterns/data/database-per-service.html.
6. dtm. The seven most classic solutions for distributed transaction management. *Medium*. URL: https://medium.com/@dongfuye/the-seven-most-classic-solutions-for-distributed-transaction-management-3f915f331e15.
7. Milanović D. M. What is API Gateway?. *Medium*.
URL: https://medium.com/@techworldwithmilan/what-is-api-gateway-91387e19dbd9.
8. IBM. What Is a Message Queue? | IBM. *IBM in Deutschland, Österreich und der Schweiz*. URL: https://www.ibm.com/topics/message-queues.
9. Solace. What is an Event Broker? | Solace. *Solace*.
URL: https://solace.com/what-is-an-event-broker/.

10. Prokofiev I. Kubernetes у мікросервісному підході. Як ми налаштували розгортання динамічних оточень для оптимізації розроблення. *dou.ua*. URL: https://dou.ua/forums/topic/45211/.

11. Altcademy Team. What is Scaffolding in Ruby on Rails?. *Altcademy Blog*. URL: https://www.altcademy.com/blog/what-is-scaffolding-in-ruby-on-rails/#:~:text=Scaffolding%20in%20Ruby%20on%20Rails%20is%20a%20helpful%20feature%20that,practices,%20and%20speeds%20up%20development (date of access: 20.05.2024).

12. Asian Digital Hub. What is Node.js and How it Work?. *Medium*. URL: https://medium.com/@asiandigitalhub/what-is-node-js-and-how-it-work-490f5ecba665.

13. Біденко Дмитро. Що таке npm і навіщо він потрібен. *robot_dreams - онлайн-курси для фахівців у сфері big data, machine learning, data science | Робот Дрімс*. URL: https://robotdreams.cc/uk/blog/271-chto-takoe-npm-i-zachem-on-nuzhen.

14. NestJS. Documentation | NestJS - A progressive Node.js framework. *Documentation | NestJS - A progressive Node.js framework*. URL: https://docs.nestjs.com/.

15. Naik A. How to scaffold ExpressJS server and test it. *Medium*. URL: https://medium.com/craft-academy/how-to-scaffold-expressjs-server-and-test-it-d2a2ab1d30e0.

16. GarryPas. Scaffolding a backend CRUD app using NestJS and Postgres. *Medium*. URL: https://medium.com/@garry.passarella/scaffolding-a-backend-crud-app-using-nestjs-and-postgres-6da6a97c5853.

17. Yeoman. Getting started with Yeoman | Yeoman. *The web's scaffolding tool for modern webapps | Yeoman*. URL: https://yeoman.io/learning/.

18. Microsoft. What is infrastructure as code (IaC)? - Azure DevOps. *Microsoft Learn: Build skills that open doors in your career*. URL: https://learn.microsoft.com/en-us/devops/deliver/what-is-infrastructure-as-code.

19. NodeJS. Child process | Node.js v22.2.0 Documentation. *Node.js – Run JavaScript Everywhere*. URL: https://nodejs.org/api/child_process.html.

20. Halliwell E. Using RabbitMQ and Kafka to send messages between applications. *Medium*. URL: https://medium.com/@edhalliwell/using-rabbitmq-and-kafka-to-send-messages-between-applications-84c962420149.

21. Tarek C. How to integrate kafka with nodejs ?. *DEV Community*. URL: https://dev.to/chafroudtarek/how-to-integrate-kafka-with-nodejs--4bil.

22. Vider O. Message Broker vs. Event Broker: When to Use Each One of Them. *Medium*. URL: https://medium.com/riskified-technology/message-broker-vs-event-broker-when-to-use-each-one-of-them-15597320a8ba.