Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Курсова робота

освітній ступінь – бакалавр

на тему: «**Розробка back-end застосунку на NestJS**»

Виконав: студент 3-го року навчання,

Спеціальності

121 «Інженерія Програмного Забезпечення»

Студента Козира Владислава

Керівник Бабич Т.А.

магістр комп’ютерних наук, асистент

«7» червня 2022 р.

Київ – 2022

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Освітній ступінь бакалавр

Спеціальність 121 «Інженерія Програмного Забезпечення»

Освітня програма бакалавр

ЗАТВЕРДЖУЮ

Завідувач кафедри інформатики

Гороховський С. С.

“13” жовтня 2021 року

ЗАВДАННЯ
ДЛЯ КУРСОВОЇ РОБОТИ СТУДЕНТУ

Козир Владиславу

1. Тема роботи **«Розробка back-end застосунку на NestJS»**, керівник роботи Бабич Трохим Анатолійович, магістр комп’ютерних наук, асистент

2. Строк подання студентом роботи 7 червня 2022

3. План роботи

Анотація

Вступ

Розділ 1. Дослідження та аналіз предметної області

1.1 Аналіз архітектурних патернів проектування систем

1.1.1 Багатошарова архітектура

1.1.2 Клієнт-серверна архітектура
1.1.3 Модель-представлення-контролер
1.1.4 Архітектура керована подіями
1.2 Порівняння монолітної та мікросервісної архітектури
1.2.1 Виявлення сервісів (Service Discovery)
1.2.2 Автоматичний вимикач (Circuit Breaker)

Розділ 2. Проектування та розробка системи

2.1 Ознайомлення з фреймворком NestJS
2.2 Моделювання схеми системи
2.3 Архітектура застосунку
2.4 Керування ролями

Висновки

Список використаних джерел

Додатки

ГРАФІК ПІДГОТОВКИ КУРСОВОЇ РОБОТИ ДО ЗАХИСТУ

№ з/п	ПЕРЕЛІК РОБІТ	Термін виконання	Дата ознайомлення наукового керівника	Підпис наукового керівника	Примітки
1.	Вибір теми, затвердження її на засіданні кафедри та закріплення наукового керівника Узгодження календарного графіка підготовки кваліфікаційної роботи. Ознайомлення студента з критеріями оцінювання кваліфікаційної роботи (п. 8.5).	13 жовтня 2021			
2.	Вивчення джерел літератури, матеріалів архівів, періодичних видань, збір та узагальнення фактів, даних	13 жовтеня 2021 – 4 листопада 2021			
3.	Складання плану каліф. роботи та узгодження з науковим керівником	4 листопада 2021			
4.	Написання розділів роботи	4 листопада 2020 – 03 березня 2022			
5.	Проміжний контроль виконання роботи	14 лютого 2022			
6.	Написання кваліфікаційної роботи в цілому, ознайомлення з її першим варіантом наукового керівника	10 січня 2022 – 28 березня 2022			
	Розділ 1 (постановка проблеми, теоретичні основи, огляд літературних джерел)	26 січня 2022			
	Розділ 2 (аналітично-дослідницька частина)	03 березня 2022			
	Розділ 3 (проектно-рекомендаційна частина)	28 березня 2022			
7.	Повне завершення написання кваліфікаційної роботи, оформлення її згідно з вимогами й подання на відгук науковому керівнику	01 квітня 2022 – 6 червня 2022			
8.	Подання кваліфікаційної роботи для перевірки письмових робіт студентів НаУКМА на відповідність вимогам академічної доброчесності,	7 червня 2022			
9.	Публічний захист кваліфікаційної роботи перед екзаменаційною комісією	згідно з розкладом роботи ЕК			

Графік узгоджено 13 жовтня 2021 р.

Науковий керівник Бабич Трохим Анатолійович

Виконавець курсової роботи Козир Владислав

ЗМІСТ

Анотація 1

Вступ 2

Розділ 1. Дослідження та аналіз предметної області 3

1.1 Аналіз архітектурних патернів проектування систем 3

1.1.1 Багатошарова архітектура 3

1.1.3 Клієнт-серверна архітектура 7

1.1.4 Модель-представлення-контролер 8

1.1.5 Архітектура керована подіями 9

1.2 Порівняння монолітної та мікросервісної архітектури 9

1.3.1 Виявлення сервісів (Service Discovery) 14

1.3.2 Автоматичний вимикач (Circuit Breaker) 14

Висновки 15

Розділ 2. Проектування та розробка системи 16

2.1 Ознайомлення з фреймворком NestJS 16

2.2 Моделювання схеми системи 17

2.3 Архітектура застосунку 19

2.4 Керування ролями 22

Висновки 24

Список використаних джерел 25

Анотація

Дана робота присвячена дослідженню та аналізу переваг, недоліків та сфер використання архітектурних патернів проектування систем. Проведене якісне порівняння підходів до розробки веб-застосунків на базі монолітної та мікросервісної архітектур. Розглянуто NestJS, як уособлення системи за патерном проектування MVC на базі мікросервісної архітектури.

У резільтаті даної роботи розроблено back-end застосунок “Управління рестораном” з використанням мікросервісного підходу, архітектури MVC та вичерпних переваг реалізації фреймворку NestJS.

Ключові слова: MVC, N-tier, Pipe and Filter, Client Server, EDA, Monolith, Microservice, API Gateway, Service Discovery, Circuit Breaker, Controller, Pipes, IoC, Docker

Вступ

В реаліях сучасного життя, використання веб-застосунків спрощує нам життя. Сфери використання подібних застосунків дуже різноманітні – від медицини до різноманітних ігрових систем. Все частіше велика кількість фізичних бізнесів намагається переходити до інтернету, щоб збільшити свої можливості та маркетинговий впливу на сферу їх діяльності. Позитивне враження про функціональність та зручність використання веб-застосунку компанії безпосередньо впливає ринок загалом. Такі впровадження дозволяють сильно збільшити користувацьку базу та покращити досвід взаємодії з брендом компанії.

Розробка веб-сайтів тягне за собою багато архітектурних питань. У ході розробки можуть з’явитися декілька проблем пов’язаних з архітектурою, масштабуванням баз даних та сервісів. Як результат – загалом успіх сайту та бізнесу залежить від початкового вибору технологій та платформи. З часом системи стають все більш складними. Це спричиняє додаткові затрати по підтримці або ж масштабуванні подібних систем. У випадку, якщо на початку допустити мінімальну кількість помилок, то можна дуже сильно економити гроші бізнесу та збільшити життєздатність проєкту загалом.

Наразі якогось єдиного універсального рішення, щодо вибори певних технологій та платформи. Через це, для кожного вибору технологій потрібно підходити індивідуально.

Розділ 1. Дослідження та аналіз предметної області

1.1 Аналіз архітектурних патернів проектування систем

У розробці існують архітектурні шаблони ПЗ, що являють собою певний звіт “гарних практик” для вирішення розповсюджених проблем проектування. Такі патерни дозволяють виразити схеми систем, базуючись на сферах відповідальності та зв’язках. Розглянемо найрозповсюджені приклади таких патернів проектування [3]:

- Багатошарова архітектура (N-tier);
- Фільтрований потік (Pipe and filter);
- Клієнт-серверна архітектура (Client server);
- Модель-представлення-контролер (Model-View-Controller);
- Архітектура керована подіями (Event Driven Architecture);

Розглянемо найбільш популярний патерн проектування – N-tier.

1.1.1 Багатошарова архітектура

Цей архітектурний патерн проектування доволі популярний серед інженерів програмного забезпечення та розробників. Загалом не існує обмежень щодо кількості та типів цих шарів, але прийнято виділяти ці чотири шари [11]:

- Шар представлення (Presentation Layer);
- Шар бізнес логіки (Business Layer);
- Шар збереження даних (Persistence Layer);
- Шар бази даних (Database Layer);

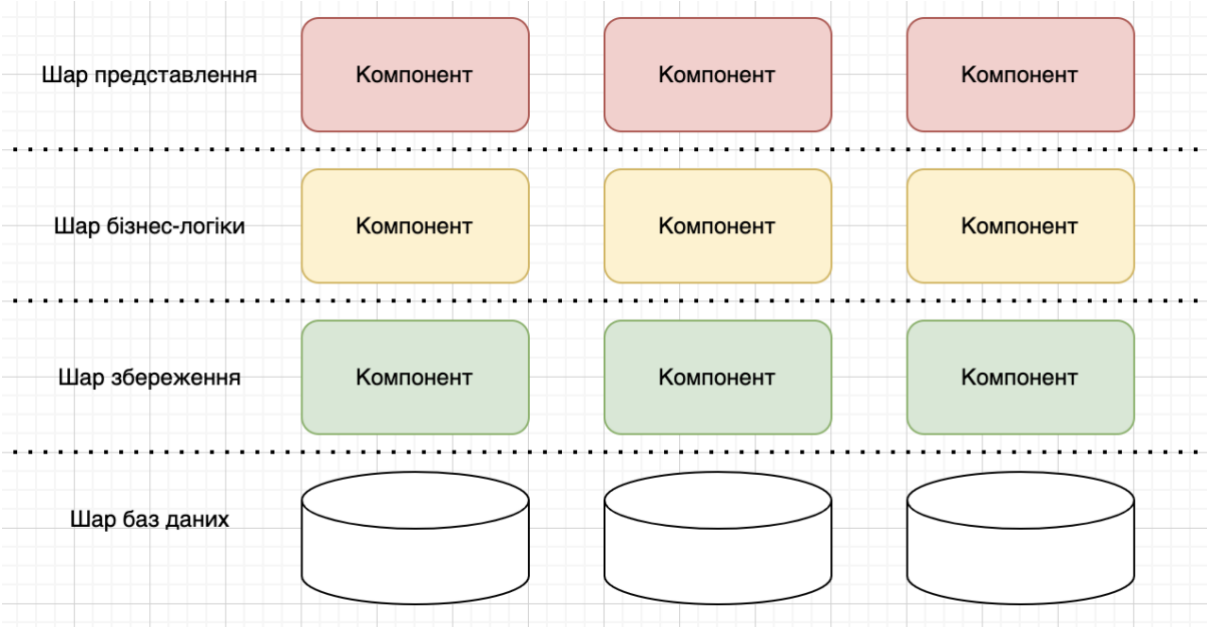

Рис 1.1 Схема багаторівневої архітектури

Зазвичай у великих, складних схемах потрібно мати можливість доволі гнучко забезпечувати паралельну розробку [3]. В реаліях подібних проектів, чим більший проєкт, тим більше буде коштувати розробка далі. Тому дуже важливо мати можливість підтримувати окремі частини системи. Для подібних ситуацій доволі гарно підходить даний архітектурний патерн, бо дає змогу зекономити гроші компанії під час розробки та підтримки.

Основне завдання даного підходу розділити ПЗ на модулі так, щоб була вони були мінімально зв’язані залежностями та забезпечували легке редагування та повторного використання.

Для використання такого підходу потрібно все ПЗ розділити на сутності – шари, а кожен з них потрібно розділяти на окремі модулі. Всі шари застосунку мають мати публічні протоколи для спілкування між собою, скриваючи реалізацію функціоналу. Також усі зв’язки між шарами мають бути одно направленими, тобто немає бути циклів в графі

залежностей між шарами. Дану архітектурний підхід можна інтерпретувати декількома способами:

- Кожен з шарів відповідає за певну роль. Для прикладу, шар представлення відповідає суто за обробку користувацького інтерфейсу;
- Групувати модулі по схожим признакам компонентів, а не доменної області;
- Позначати кожен з рівнів відкритим або закритим. Кожен з запитів має оброблятися кожним з шарів та проходити далі чи ні;

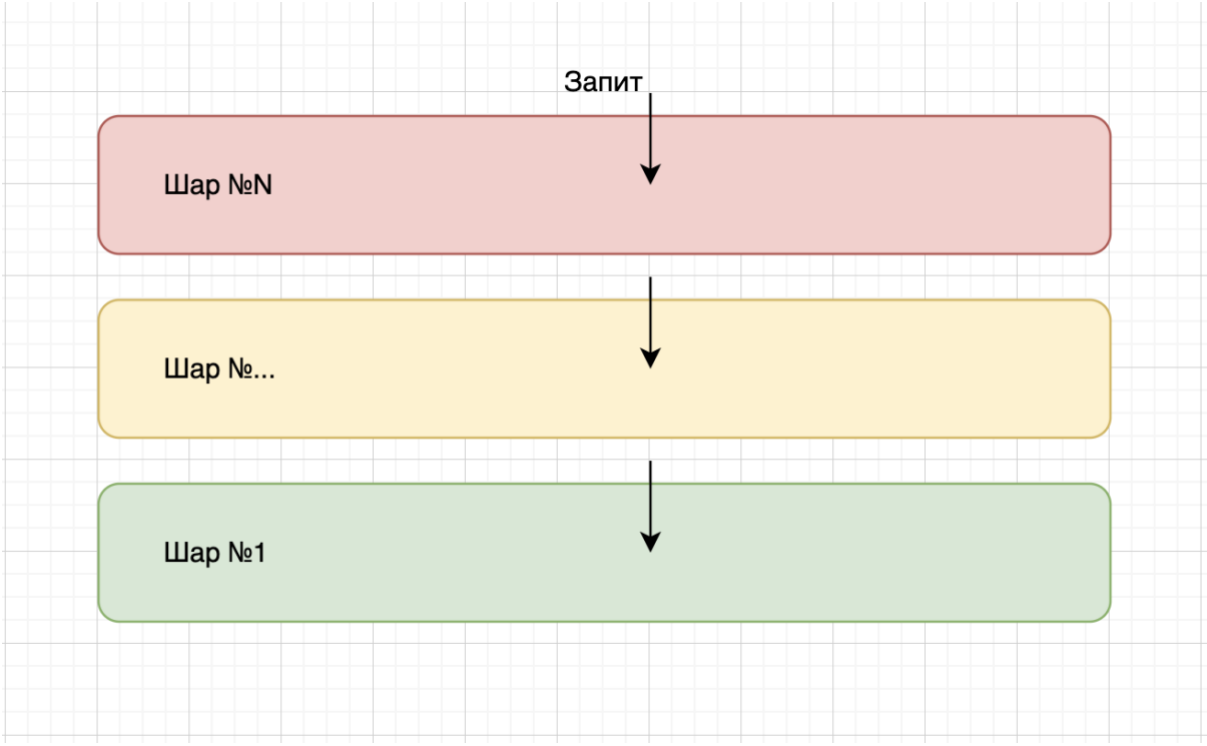

Рис 1.2 Відкриті та закриті шари

Серед недоліків можна виділити performance показники при проходженні запиту через всі шари нашого застосунку.

1.1.2 Фільтрований потік

Бувають задачі коли нам потрібно до певного набору даних застосовувати фільтрацію. Наприклад, стрічка Facebook фільтрується в залежності від ваших інтересів та релевантної інформації. В таких ситуаціях було б зручно виокремити фільтри як окремі компоненти, щоб мати можливість використовувати повторно.

При проектуванні подібних систем потрібно розділяти на слабо пов’язані компоненти, які мають змогу викликати певний інший компонент-фільтр. Через таку гнучку взаємодію, ми маємо можливість запускати виконання подібних компонентів паралельно.

Подібні канали з фільтрами є одно направленими – приймаєш дані та викликаєш наступну сутність з набором оброблених параметрів. В подібних підходах виокремлюють чотири видів фільтрів:

- Джерело (source – початок каналу з фільтрами);
- Трансформатор (map – обробляє дані);
- Фільтр (reduce – обробляє критерії фільтрів);
- Кінець (sink – поглинач каналу з фільтрами);

Рис 1.3 Схема фільтрованого потоку

Подібна архітектура гарно себе показує в застосунках, що займається односторонньою обробкою даних.

1.1.3 Клієнт-серверна архітектура

Існують системи де потрібно забезпечувати доступ до різноманітних розподілених клієнтів. Наприклад, той же самий Facebook – потрібно розробляти веб-клієнт та мобільні клієнти, котрі написані нативно для кожної з платформ: Android та iOS [4].

Така архітектура дозволяє не дублювати спільну бізнес логіку, а досить легко модифікувати спільну кодову базу. Також такий підхід дозволяє покращити підходи до масштабування, оскільки вся бізнес-логіка обробляється більш-менш централізовано [4]. Ця властивість дозволяє обробляти доволі затратні запити не на слабких мобільних клієнтах, а на більш потужних віддалених машинах.

Схема роботи подібної архітектури доволі розповсюджена. В нас є умовний клієнт, що відправляє певний запит, та сервер, що обробляє його.

До недоліків подібних систем можна віднести слабку продуктивність через занадто велику завантаженість серверу. Потрібно крок за кроком вносити зміни спочатку на сервері, а потім на клієнтах. Також, у випадку мобільного клієнту потрібно зберігати обернену сумісність до протоколу спілкування між клієнтом та сервером.

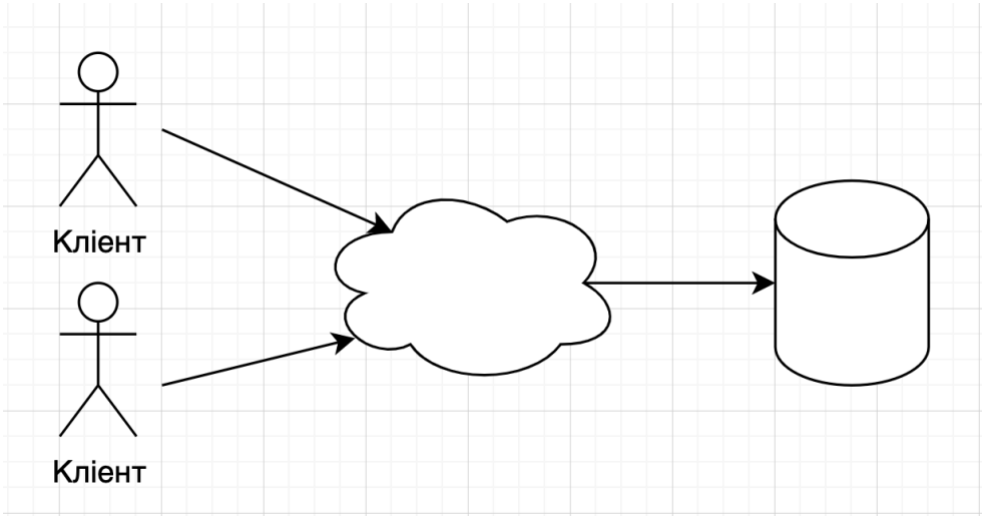

Рис 1.4 Клієнт-серверна архітектура

1.1.4 Модель-представлення-контролер

В більшості інтерактивних систем ми маємо задачу оновлення представлення даних. В таких системах потрібно виокремлювати функціональність користувацького інтерфейсу. Ця ідея і закладена в даному архітектурному підході – розподілення програми на певні компоненти. У даному випадку можна виділити [1]:

- Модель – модель для оперування на різних рівнях застосунку;
- Представлення – компонент відображення застосунку;
- Контролер – компонент, що відповідає управлінню та оновленню стану системи;

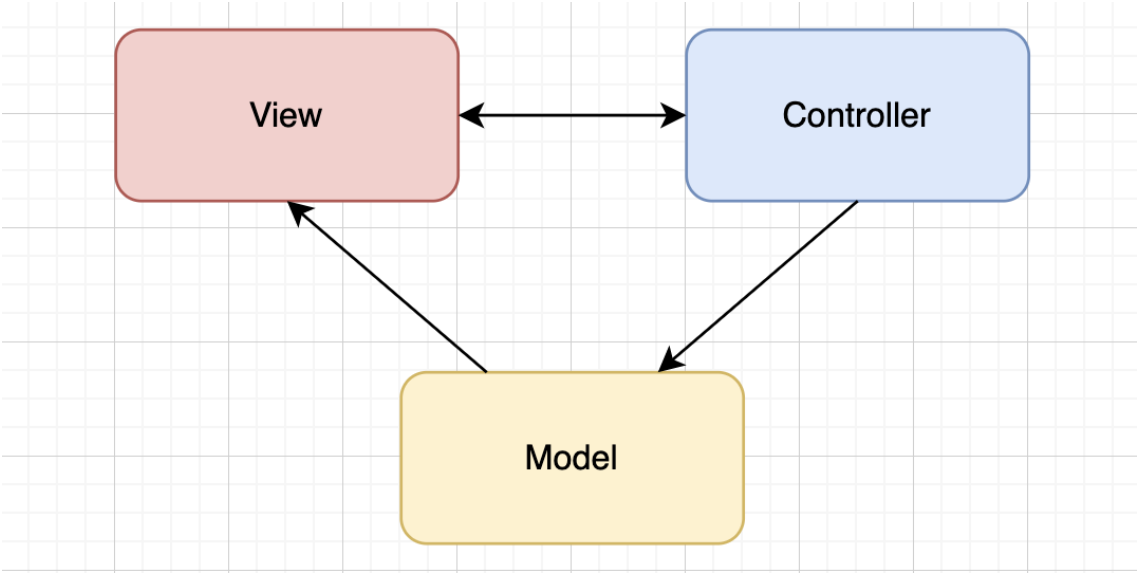

Рис 1.5 Модель-представлення-контролер

Така абстракція для доволі простих інтерфейсів можемо виглядати занадто складною. Подібна архітектура зазвичай використовується у мобільних застосунках або веб-сервісах [1].

1.1.5 Архітектура керована подіями

Системи з подібною архітектурою дозволяють обробляти асинхронні повідомлення, що асоціативні з певною подією. Така можливість обробки дозволяє гарно масштабувати подібні системи з збільшенням складності обробки подібних подій [9].

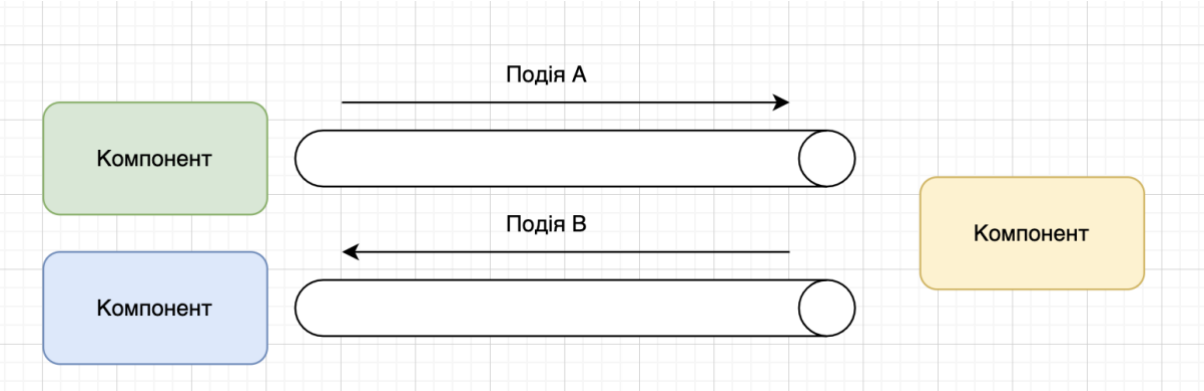

Рис. 1.6 Схема архітектури керованої подіями

Кожен з компонентів асинхронно обробляє нові події, котрі приходять з черги. Це дозволяє не втрачати нові події та гарантувати повноцінну обробку всіх сутностей. Така поведінка системи може погано відпрацьовувати при великих об’ємах даних або ж при відмові певного компоненту [9].

Для кращого розуміння роботи подібної архітектури можна розглянути систему, що керує банківськими переказами. Після початку певної транзакції платіж має певний стан “не виконаний”. Після обробки події від сторонньої платіжної системи ми отримаємо новий статус “зв’язок з банківською установою” і так далі.. Такий підхід доволі гнучко дозволяє повідомляти компоненти, що прослуховують канал передачі даних [9].

1.2 Порівняння монолітної та мікросервісної архітектури

В розрізі цього блоку ми порівняємо доволі популярні архітектурні підходи при розробці саме веб-застосунків по признаку розподіленого

виконання коду. Основна відмінність подібних систем полягає у їх реалізації, що відповідно впливає на низку інших характеристик.

Монолітна архітектура базується на принципі розташуванні системи на одній машині та в одному процесі. Зазвичай ця система базується на одному наборі технологій та мови програмування. Масштабування подібних систем можливе лише горизонтально завдяки запуску таких же серверів на декількох машинах [8].

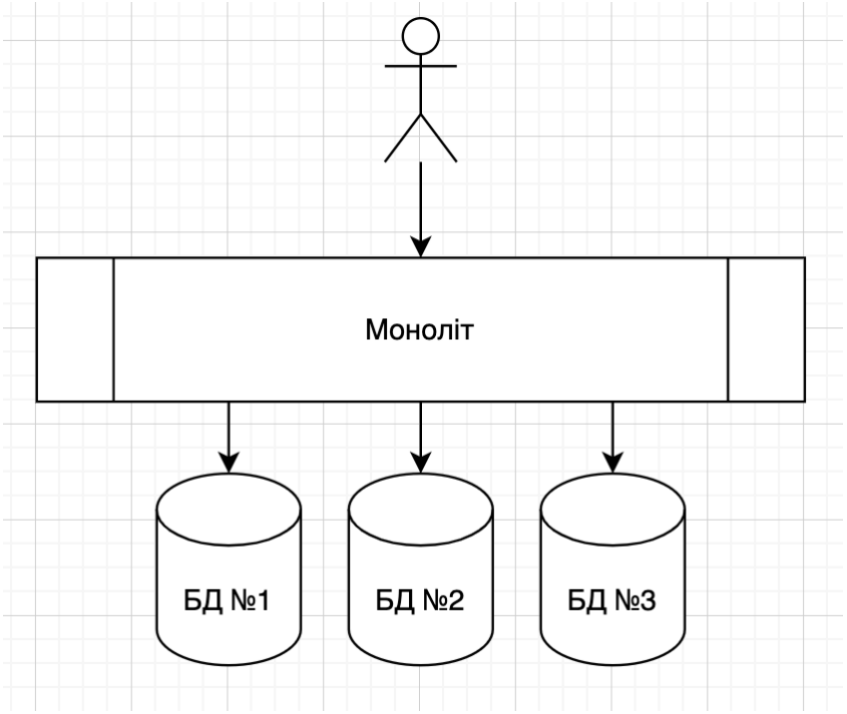

Рис 1.7 Схема монолітної архітектури

Мікросервісна архітектура базується як розподілена система. Кожна з атомарних частин, окремих програмних частин має свої бази даних. Через таку поведінку доступ до даних може відбуватися викликом ресурсів іншого сервісу.

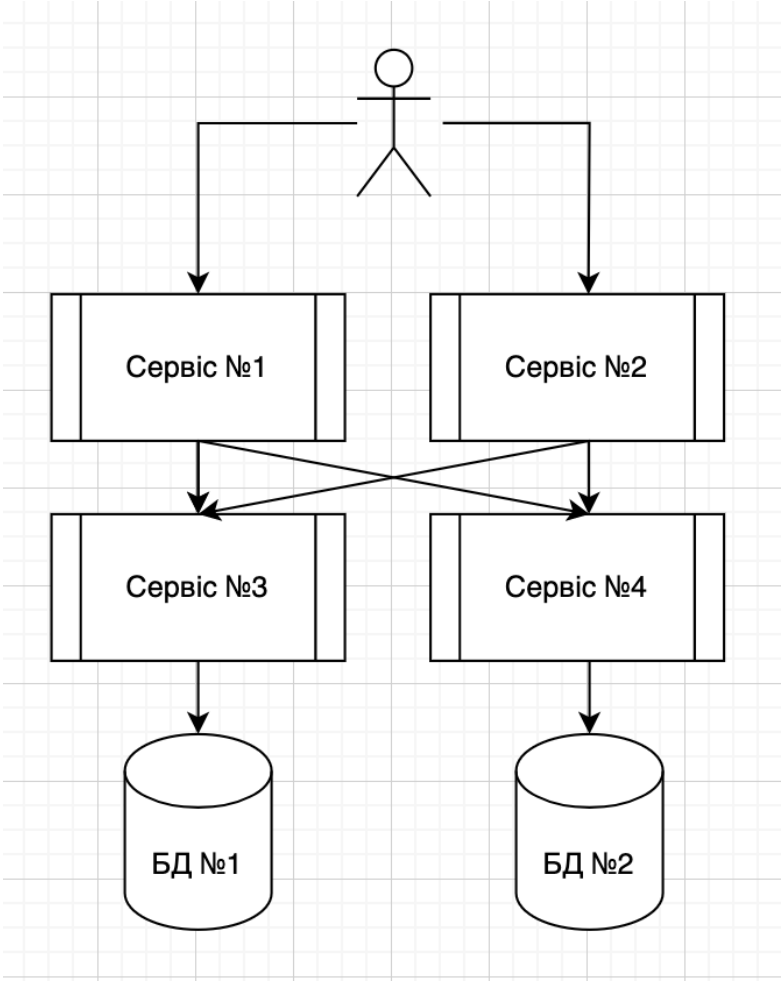

Рис 1.8 Схема мікросервісної архітектури

Монолітна архітектура базується на єдиній програмній системі, що дозволяє окремим компонентам системи доступатися до всіх ресурсів системи. Для мікросервісної архітектури підхід відрізняється, бо кожен сервіс має окрему кодову базу. При такому підході ми можемо встановлювати обмеження на доступ до певних ресурсів системи [2].

В сучасному світі для back-end застосунків важливий показник uptime (% часу доступності до системи) під час оновлення версії. У випадку з монолітною архітектурою, при будь-якій зміні хоча б одного сервісу потрібно буде перезапускати повністю всю систему. В епоху хмарних систем, подібне розгортання застосунку є досить тяжким для системи

оточення [10]. Через таке розгортання наш сервіс буде недоступним протягом розгортання, що безпосередньо впливає на показник uptime. В той же час, мікросервісна архітектура має можливість розгортати тільки змінені сервіси. Через це для хмарних рішень простіше розгорнути оновлення тільки певного сервісу. В такому випадку простіше контролювати стабільність системи та скоріше виявляти проблеми в роботі даної системи [2].

Для доволі популярних систем доволі важливим є можливість масштабування – здатність певної системи коректно справлятися з великим навантаженням користувачів. Розрізняють два типи масштабування [5]:

- Вертикальне – збільшення потужності комп’ютера або сервера, встановлення кращого CPU і тд.;
- Горизонтальне – доволі складне. Потрібні інфраструктурні зміні в коді системи для розміщення на окремих серверах;

При горизонтальному масштабуванні простіше за все використовувати розподілені системи, що мають змогу обробки запитів з декількох вузлів. Тому при горизонтальному масштабуванні мікросервісна архітектура має перевагу, бо кожен з мікросервісів можна розташувати на окремих серверах.

У розрізі швидкодії, мікросервісна архітектура має вагомий мінус, через затрати для передачі інформації по протоколам між сервісами.

1.3 Шаблон API Gateway

При збільшенні кількості мікросервісів потрібно звернути увагу на зручну маршрутизації запитів. Цю проблему призваний вирішити шаблон API Gateway.

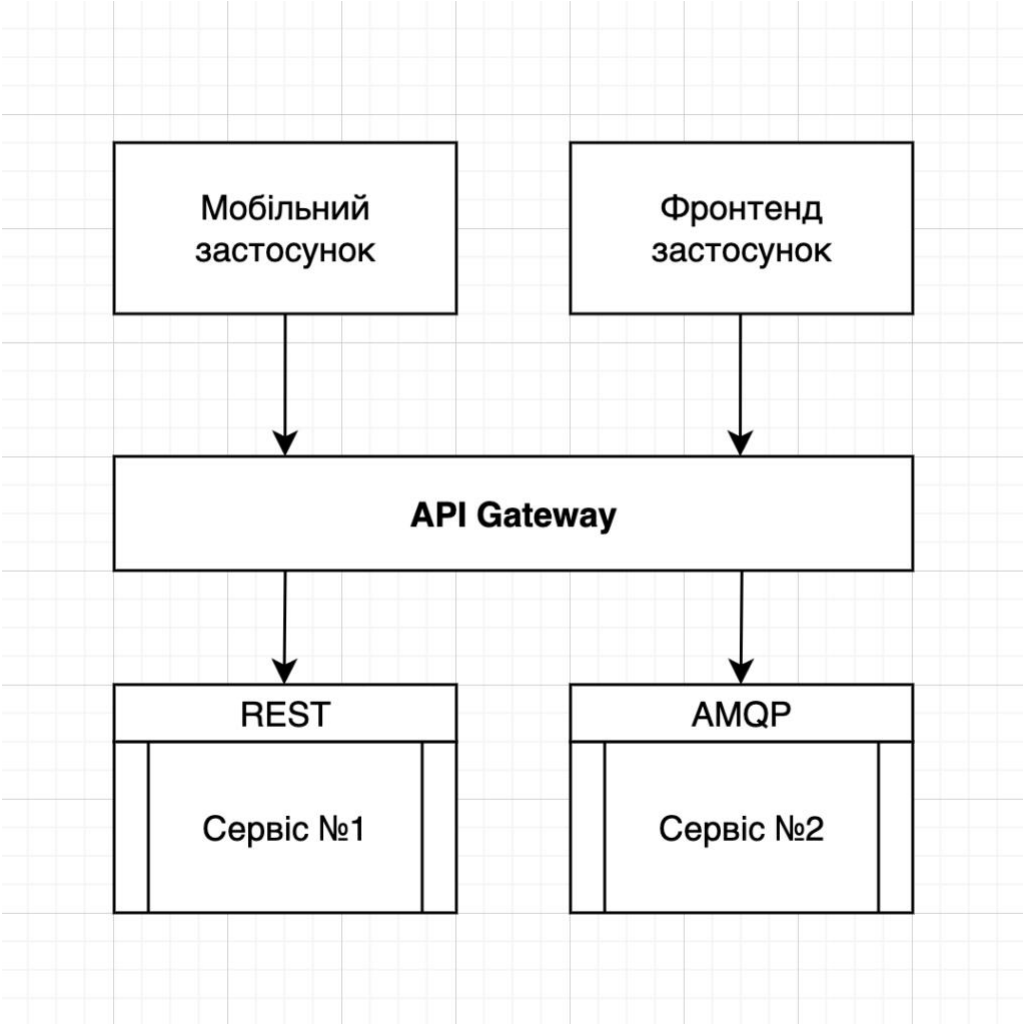

Рис 1.9 Схема шаблону API Gateway

Кожен з запитів клієнтів перенаправляється на відповідний сервіс. Ця єдина точка входу відповідає за безпеку застосунку або ж може відповідати за валідацію JWT токена та доступи до конкретних сервісів.

Оскільки, бізнес потреби клієнтів застосунку можуть з часом набувати все більшої логіки, то в результаті цей один API Gateway може стати монолітним. В таких випадках рекомендується використовувати окремі API Gateway для кожного з клієнтів.

Розглянемо переваги використання шаблону API Gateway:

- Зручно використовувати;

- Гнучке використання;
- Можемо кешувати запити;
- Можемо комбінувати відповіді з обраних сервісів;
- Можемо валідувати доступи користувачів;

Цей шаблон можна поставити у відповідність ООП-шаблон аналог – фасад. Він також дозволяє за допомогою протоколу керувати складною системою.

При багатьох перевагах, у цього шаблону є і недоліки:

- Незначне ускладнення системи додатковим сервісом;
- Потрібно кожного разу оновлювати шлюз API Gateway;

1.3.1 Виявлення сервісів (Service Discovery)

Через доволі гнучке оновлення компонентів мікросервісної системи потрібно вміло змінювати адресу нового образу сервісу. Для цього використовуються Service Discovery реєстри, що допомагають оновлювати адресу нового сервісу при будь-яких змінах. Тобто він буде відслідковувати всі зміни адрес сервісів та оновлювати кеш, що дозволить сильно зменшувати uptime системи.

1.3.2 Автоматичний вимикач (Circuit Breaker)

Доволі часто в мікросервісній архітектурі один сервіс буде відправляти додаткові запити до інших сервісів. У випадку помилки на одному з них це може зламати систему. Для вирішення цієї проблеми використовується патерн автоматичний вимикач. Він працює доволі просто, після певної кількості помилкових запитів вимикач буде повертати негативний результат задля економії ресурсів системи та зменшення відповіді від сервера.

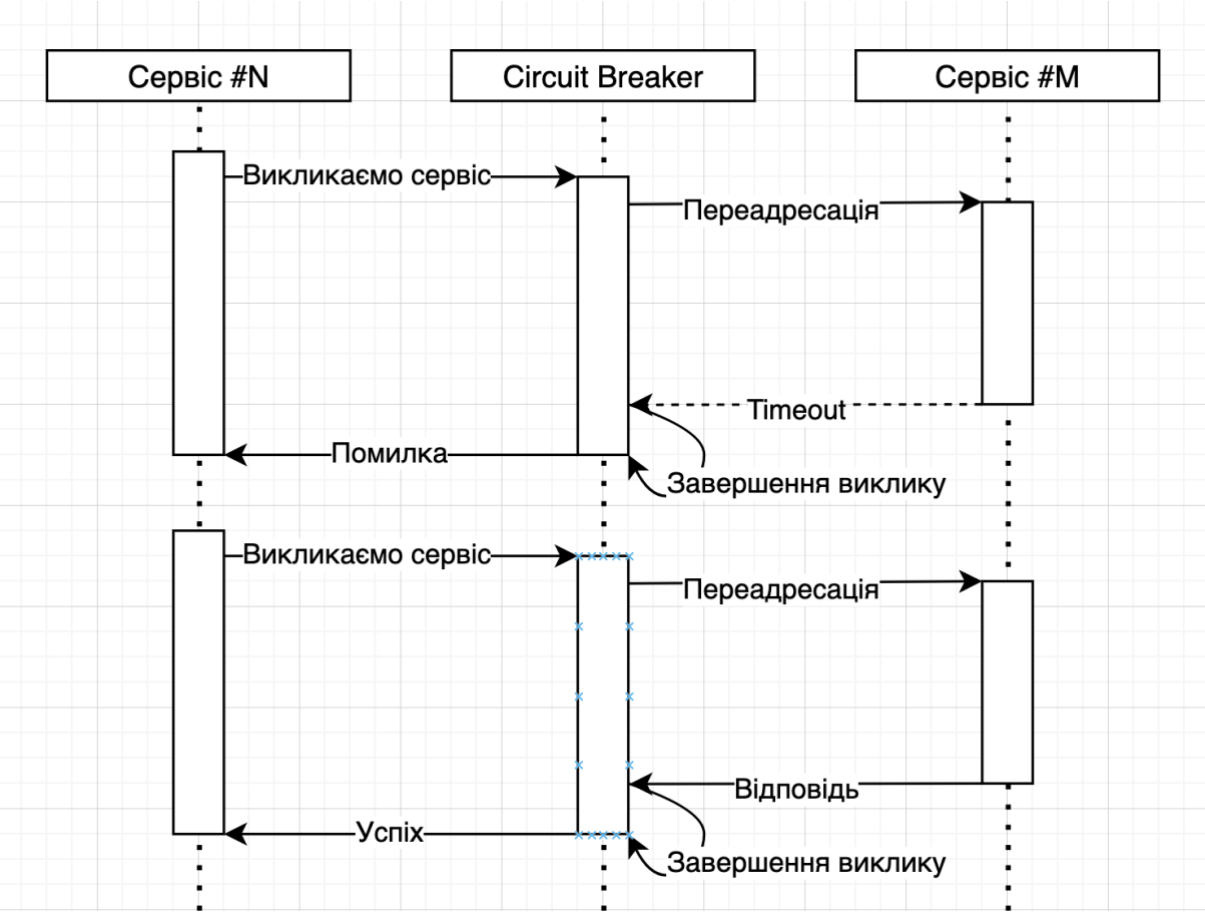

Рис 2.1 Схема автоматичного вимикача

Висновки

У ході аналізу та дослідження архітектурних патернів проектування систем було розглянуто найбільш популярні та актуальні. Під час аналіз ми виділили певні сценарії їх використання, що дозволить більш точково та правильно підбирати архітектурні рішення під кожну з ситуацій.

Розділ 2. Проектування та розробка системи

2.1 Ознайомлення з фреймворком NestJS

В розрізі даної курсової роботи вирішено використовувати фреймворк NestJS. На даному прикладі використання краще розглянемо MVC архітектуру та розглянемо підхід до розробки системи з мікросервісною архітектурою.

NestJS слідкує парадигмі дизайну “convention over configuration”. Даний фреймворк пропонує використовувати певні інструменти та створювати системи певним чином. NestJS одразу пропонує певну структуру проєкту. Також базовою мовою для NestJS є TypeScript, що гарно підходить для доволі великих команд, бо має строгу типізацію [6].

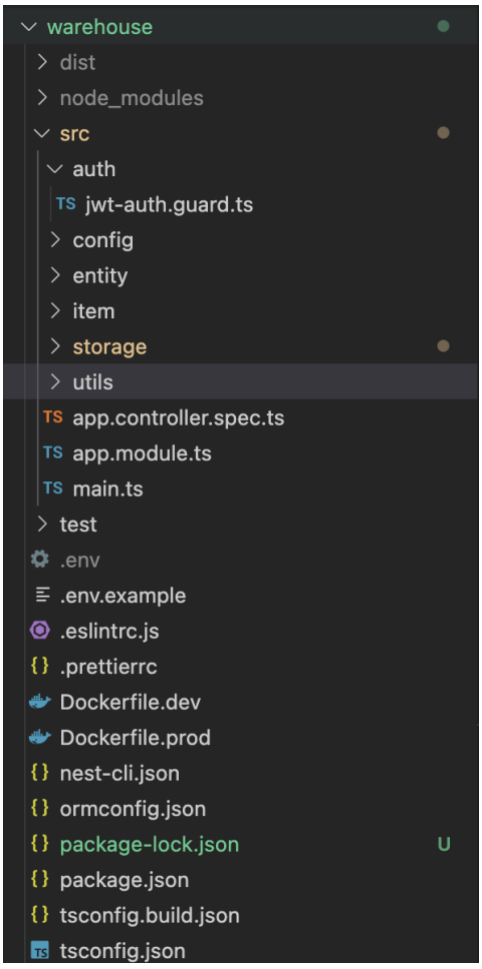

Рис 2.1 Структура проєкту на базі NestJS

З плюсів даного фреймворку можна виділити:

- Певні патерни розробки;
- Доволі просто розібратися з найкращими практиками використання фреймворку;
- Доволі просте розгортання початкового проєкту;
- Найбільш популярні сценарії використання реалізовані з старту;
- Можливість використовувати Express.js або Fastify;

З негативних моментів фреймворку NestJS можна виділити:

- Важкість системи
- Швидкодія

2.2 Моделювання схеми системи

Для розробки системи обрано тематику управління процесами закладу. Ця система має мати можливість керування замовленнями та складськими записами задля повного контролю. Система буде базуватися на 2ох мікросервісах, з виокремленою авторизацією та аунтефікацією. Такий розподіл здійснено за звичайною схемой виокремлення сервісу для керування користувачами.

Система має розподілення по ролям, а саме: ADMIN, SUPER_ADMIN, USER. У теоретичній частині ми розглядали шаблон проектування системи API Gateway. Одним з функцій даного фасаду є можливість управління доступами до певних сервісів за ролями авторизованих користувачів. У розрізі цієї системи недоцільно використовувати даний шаблон проєктування, бо наша система буде розбиватися тільки на 2 мікросервіси. У такому випадку використання даного шаблону принесе тільки додаткові затрати по імплементації подібного рішення.

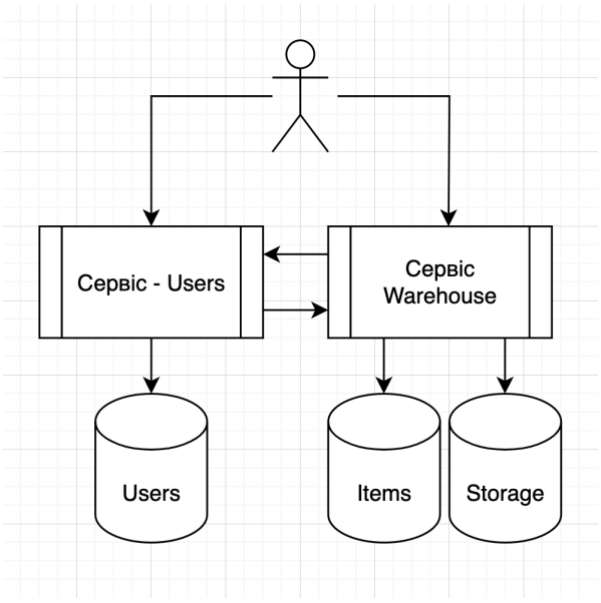

Рис 2.2 Схема системи без використання шаблону API Gateway

У розрізі ситуації коли ми використовуємо схему системи без використання шаблону API Gateway, сервіс warehouse буде напряму відправляти запити по перевірці статусу авторизації та ролі користувача [7].

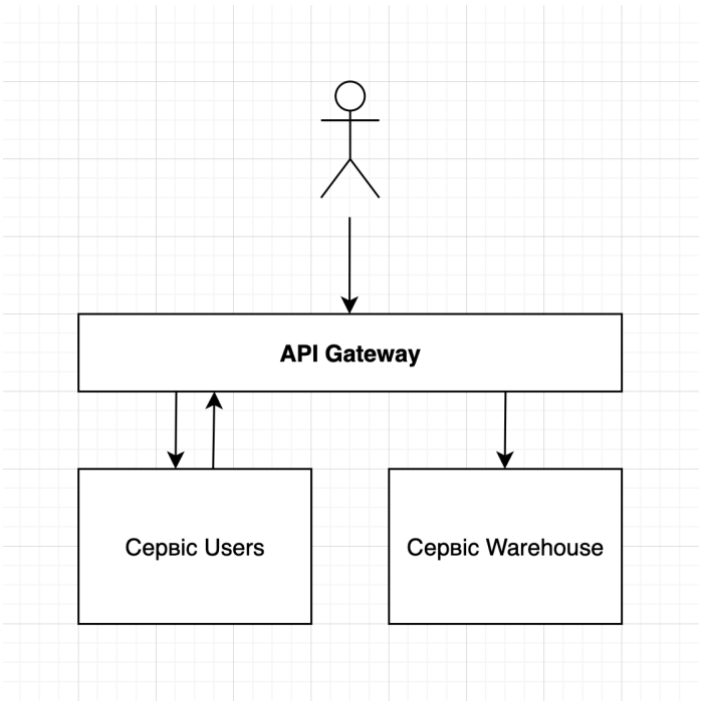

Рис 2.3 Схема системи з використанням API Gateway

При використанні схеми системи з використанням API Gateway, даний фасад відправляє запит до сервісу Users. В залежності від статусу або ж ролі даного користувача, Gateway переадресовую запит на відповідний сервіс або ж повертає певну помилку 403 – Forbidden.

2.3 Архітектура застосунку

Розглянемо сутності, які доступні в NestJS з пустого проєкту та розглянемо їх використання [6].

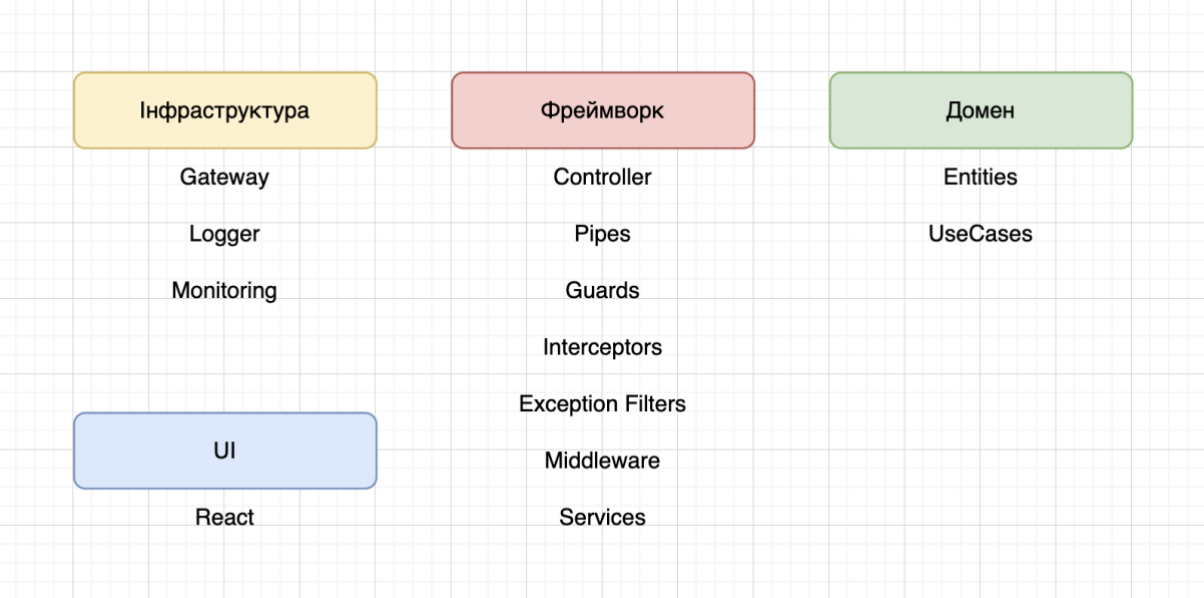

Рис 2.4 Схема архітектури застосунку

- **Controller.** Мета контролера — отримувати конкретні запити до програми. Механізм маршрутизації контролює, який контролер отримує які запити. Часто кожен контролер має більше одного маршруту, і різні маршрути можуть виконувати різні дії.

```
@Controller('user')
export class UserController {
  constructor(private readonly userService: UserService) {}
  @Post()
  async createUser(@Body() dto: UserDto) {
    const resp = await this.userService.create(dto);
```

```
  return { id: resp.id };
}
```

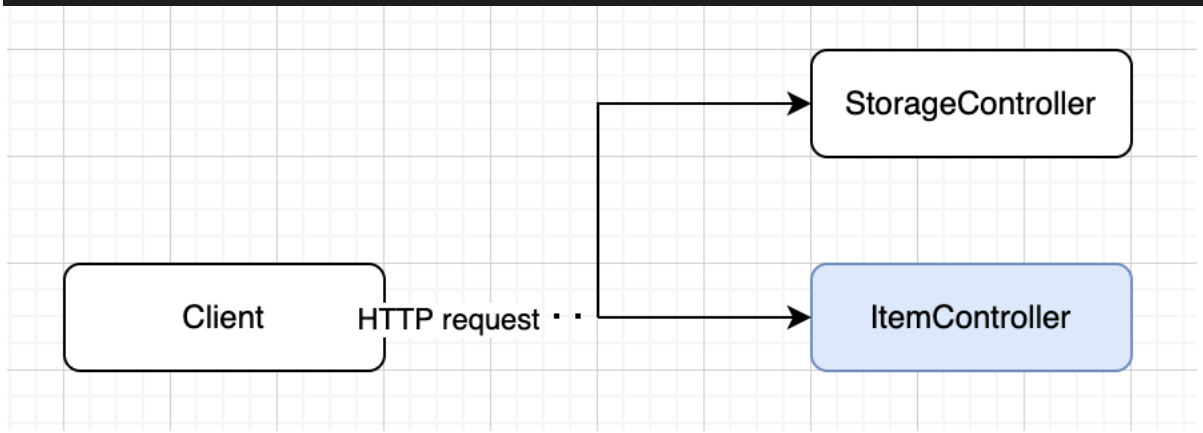

Рис 2.5 Схема комунікації з контролером

- **Provider** – є основним поняттям у Nest. Багато базових класів Nest можуть розглядатися як provider – сервіс, репозиторії, фабрики, помічники тощо. Основна ідея провайдера полягає в тому, що його можна ввести як залежність; це означає, що об'єкти можуть створювати різні відносини один з одним, а функцію "підключення" екземплярів об'єктів можна значною мірою делегувати системі середовища виконання Nest. Цей підхід називається інверсією контролю залежностей IoC. Схожий підхід використовується в Spring Boot.

```
@Injectable()
export class UserService {
  private readonly logger = new Logger(UserService.name)

  constructor(
    @InjectRepository(User) private readonly repo: Repository<User>,
  ) {}
}
```

- **Module.** Кожна програма має принаймні один модуль, кореневий модуль. Кореневий модуль є відправною точкою, яку Nest

використовує для побудови графа залежностей програми — внутрішньої структури даних.

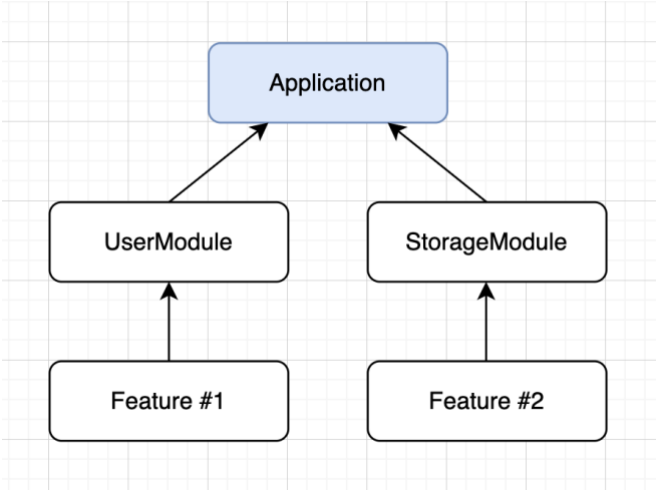

Рис 2.6 Схема архітектури модулів

```
@Module({
  imports: [
    TypeOrmModule.forRoot(configService.getTypeOrmConfig()),
    UserModule,
    AuthModule,
  ],
})
export class AppModule {}
```

- **Interceptors** мають набір корисних можливостей, які натхненні технікою аспектно-орієнтованого програмування (AOP). Вони дають можливість: прив’язувати додаткову логіку до/після виконання методу перетворити результат, повернутий функцією трансформувати виняток, викликаний з функції розширити поведінку основної функції повністю замінити функцію залежно від конкретних умов (наприклад, для кешування).

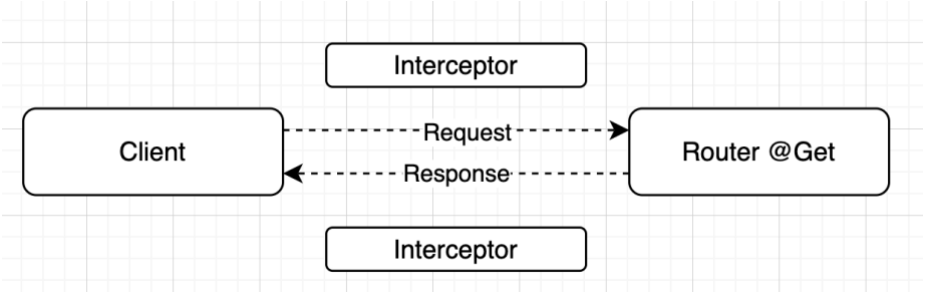

Рис 2.7 Схема взаємодії інтерсепторів

- **Guards** мають єдину відповідальність. Вони визначають, чи буде даний запит оброблений обробником маршруту чи ні, залежно від певних умов (наприклад, дозволів, ролей, списків керування доступом тощо), присутніх під час виконання. Це часто використовують для реалізації авторизації.

```
@Injectable()
export class JwtAuthGuard extends AuthGuard('jwt') {
  constructor(private reflector: Reflector) {
    super();
  }

  canActivate(context: ExecutionContext) {

    return super.canActivate(context);
  }
}
```

2.4 Керування ролями

Для доступу до певний методів контролерів, NestJS пропонує використовувати guard для обробки ролей користувача при запитах. Визначемо список можливих ролей для нашого проєкту.

```
export enum Permission {
  REGULAR_USER = 0,
  ORG_ADMIN = 1,
  SUPER_ADMIN = 2
```

```
}
```

Для зручного маркування ролей додамо кастомні декоратори до методів:

- @Access(Permission.REGULAR_USER, …) – доступ до списку певних ролей;
- @Public() – доступ до всіх визначених ролей;

```
export const ROLE_LIST = 'roleList';

export const Access = (...arr: Permission[]) => SetMetadata(ROLE_LIST, arr);
export const Public = () =>
  SetMetadata(ROLE_LIST, [
    Permission.REGULAR_USER,
    Permission.ORG_ADMIN,
    Permission.SUPER_ADMIN,
  ]);
```

Через додаткову залежність у вигляді Reflector можемо отримати значення ролей для певної функції контролера на котрій виконується запит. Базуючись на переліку цих ролей, ми можемо вміло фільтрувати запит.

```
@Injectable()
export class RolesGuard implements CanActivate {
  constructor(private reflector: Reflector) {}

  canActivate(context: ExecutionContext): boolean {
    const roles = this.reflector.get<Permission[]>(ROLE_LIST, context.getHandler());
    if (!roles) return true;

    const request = context.switchToHttp().getRequest();
    const user = request.user;

    return roles.includes(user?.permissionLevel);
  }
```

```
}
```

Висновки

У ході даної роботи було змодельовано архітектурну модель веб-застосунку відповідно до певної тематики та бізнес задач. Результатом даної роботи є застосунок на базі NestJS. Також було розглянуто основні атомарні архітектурні сутності даного фреймворку, що дозволило доволі гнучко адаптувати потреби бізнес задачі під певне технічне рішення.

Список використаних джерел

1. John Deacon. Model-View-Controller (MVC) architecture. Academia. 1995. P. 1–5. Mario Villamizar.
2. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. Columbia. 2015. Mark Richards.
3. Software architecture patterns. Sebastopol, CA : O'Reilly Media, 2015. 345 p.
4. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions / Gregor Hohpe, Bobby Woolf – Addison-Wesley, 2004 – 736 p.
5. Moskal, V., & Yeromina, N. (2021). METHODS OF DEVELOPING WEB APPLICATIONS. Збірник наукових праць SCIENTIA. URL: https://ojs.ukrlogos.in.ua/index.php/scientia/article/view/15870
6. NestJS documentation. NestJS - progressive Node.js framework. URL: https://docs.nestjs.com/.
7. David Jaramillo. Leveraging microservices architecture by using Docker technology. Norfolk, VA.
8. Newman S. Monolith to microservices : навчальний посібник. 2015.
9. Brenda M. Michelson. Event-Driven architecture overview. Elemental links research. 2011. P. 3–6.
10. Paul D.Manuel. A data-centric design for n-tier architecture. Information sciences. 2003. No. 150. P. 195–206.
11. J. T. Zhao. Management of API gateway based on micro-service architecture. No. 1087.

Додатки

Додаток А roles.guard.ts

```
import { Injectable, CanActivate, ExecutionContext } from '@nestjs/common';
import { Reflector } from '@nestjs/core';
import { ROLE_LIST } from 'src/utils/jwt';
import { Permission } from 'src/utils/permission-levels';

@Injectable()
export class RolesGuard implements CanActivate {
  constructor(private reflector: Reflector) {}

  canActivate(context: ExecutionContext): boolean {
    const roles = this.reflector.get<Permission[]>(ROLE_LIST, context.getHandler());
    if (!roles) return true;

    const request = context.switchToHttp().getRequest();
    const user = request.user;

    return roles.includes(user?.permissionLevel);
  }
}
```

Додаток Б – storage.controller.ts

```
import { Body, Controller, HttpException, HttpStatus, Param, Patch, Post, Req } from '@nestjs/common';
import { Access, Public } from 'src/utils/jwt';
import { Permission } from 'src/utils/permission-levels';
import { Storage } from './storage.entity';
import { StorageService } from './storage.service';

@Controller('storage')
export class StorageController {
    constructor(private readonly storageService: StorageService) {}
```

```
  @Public()
  async getAll() {
    return this.storageService.getAll()
  }

  @Public()
  async getById(@Param('id') id: string) {
     return this.storageService.getById(id)
  }

  @Access(Permission.ORG_ADMIN, Permission.SUPER_ADMIN)
  async update(@Body('storage') newStorage: Storage) {
     return this.storageService.update(newStorage);
  }

}
```

Додаток В – storage.service.ts

```
import { BadRequestException, HttpException, HttpStatus, Injectable, Logger } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Access, Public } from 'src/utils/jwt';
import { Permission } from 'src/utils/permission-levels';
import { Repository } from 'typeorm';
import { Storage } from './storage.entity';

@Injectable()
export class StorageService {
  private readonly logger = new Logger(StorageService.name);

  constructor(
    @InjectRepository(Storage) private readonly repo: Repository<Storage>,
  ) {}

  async getAll(){
    try {
      return this.repo.find();
```

```
  } catch (e) {
    this.logger.warn(e.detail);
    throw new BadRequestException('Error occurred while getting all storages');
  }
 }

 async getById(id: string){
  try {
    return this.repo.findOne(id);
  } catch (e) {
    this.logger.warn(e.detail);
    throw new BadRequestException('Error occurred while getting storage by id');
  }
 }

 async update(storage: Storage): Promise<Storage> {
  try {
    return this.repo.save(storage);
  } catch (e) {
    this.logger.warn(e.detail);
    throw new BadRequestException('Error occurred while updating a storage');
  }
 }
}
```