МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: **«РОЗРОБКА ОСВІТНЬОЇ ПЛАТФОРМИ НА ОСНОВІ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ ІЗ ВИКОРИСТАННЯМ NODE.JS ТА ANGULAR»**

Виконала: студентка 2 курсу, групи 8.1213-іпз-1
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми інженерія програмного забезпечення
(назва освітньої програми)
А.О. Валяєва
(ініціали та прізвище)

Керівник доцент кафедри програмної інженерії, доцент, к.т.н. Лимаренко Ю.О.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент професор кафедри комп'ютерних наук, професор, д.т.н. Шило Г.М.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти магістр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2024 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТЦІ

Валяєвій Анні Олегівні
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка освітньої платформи на основі мікросервісної архітектури із використанням Node.js та Angular

керівник роботи Лимаренко Юлія Олексіївна, к.т.н., доцент
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 01 » березня 2024 року № 385-с

2. Строк подання студентом роботи 25.11.2024 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Постановка задачі.
2. Основні теоретичні відомості.
3. Проєктування та реалізація програмного забезпечення.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 07.03.2024 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	11.03.2024	
2.	Збір вихідних даних.	08.04.2024	
3.	Обробка методичних та теоретичних	06.05.2024	
	джерел.		
4.	Розробка першого та другого розділу.	30.08.2024	
5.	Розробка третього розділу.	25.10.2024	
6.	Оформлення та нормоконтроль	20.11.2024	
	кваліфікаційної роботи магістра.		
7.	Захист кваліфікаційної роботи.	11.12.2024	

Студент _____ А.О. Валяєва
(підпис) (ініціали та прізвище)

Керівник роботи _____ Ю.О. Лимаренко
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота магістра «Розробка освітньої платформи на основі мікросервісної архітектури із використанням Node.js та Angular»: 82 с., 20 рис., 30 джерел.

ANGULAR, CI/CD, DOCKER, GRPC, KUBERNETES, MICROFRONTENDS, MICROSERVICES, NODE.JS, POSTGRESQL, RABBITMQ, REDIS, REST API.

Об’єкт дослідження – мікросервісна архітектура, платформа Node.js, фреймворк Angular.

Мета роботи: розробка освітньої платформи.

Метод дослідження – методи збору та аналізу вимог до програмного забезпечення, методи моделювання та проєктування програмного забезпечення.

У кваліфікаційній роботі наведено характеристику предметної області, огляд існуючих систем з предметної області, аналіз мікросервісної архітектури та огляд засобів розробки, зокрема платформи Node.js та фреймворку Angular. На основі даних теоретичних відомостей сформовані вимоги до програмного продукту, спроєктовано архітектуру та створено програмну реалізацію проєкту освітньої платформи.

Результатом виконання кваліфікаційної роботи є додаток освітньої платформи, написаний на основі мікросервісної архітектури, використовуючи платформу Node.js та фреймворк Angular.

SUMMARY

Master’s qualifying paper “Development of an Educational Platform on the Basis of Microservices Architecture Using Node.js and Angular”: 82 pages, 20 figures, 30 references.

ANGULAR, CI/CD, DOCKER, GRPC, KUBERNETES, MICROFRONTENDS, MICROSERVICES, NODE.JS, POSTGRESQL, RABBITMQ, REDIS, REST API.

The object of the study is microservice architecture, Node.js platform, Angular framework.

The aim of the study is development of an educational platform.

The methods of research are collecting and analyzing software requirements, methods of modeling and designing software.

The qualification paper provides a description of the subject area, a review of existing systems in this field, an analysis of microservices architecture, and an overview of development tools, specifically the Node.js platform and Angular framework. On the basis of these theoretical information, requirements for the software product were defined, the architecture was designed, and a software implementation of an educational platform project was created.

The result of the qualification project is an educational platform application built using a microservices architecture, developed with the Node.js platform and Angular framework.

ЗМІСТ

Завдання на кваліфікаційну роботу……..…..2
Реферат 4
Summary 5
Вступ..... 9
1 Аналіз предметної області та об’єкту дослідження 12
1.1 Характеристика предметної області 12
1.2 Огляд існуючих освітніх платформ 14
1.2.1 Освітня платформа Prometheus..... 14
1.2.2 Освітня платформа Udemy..... 15
1.3 Аналіз мікросервісної архітектури..... 17
1.3.1 Переваги мікросервісної архітектури 18
1.3.2 Недоліки мікросервісної архітектури 19
1.3.3 Типи та засоби комунікації мікросервісів..... 20
1.3.4 Патерни та практики мікросервісної архітектури 22
1.3.5 Застосування принципів мікросервісної архітектури при побудові користувацьких інтерфейсів..... 24
1.4 Огляд засобів розробки 28
1.4.1 Огляд СУБД PostgreSQL..... 28
1.4.2 Огляд платформи Node.js..... 29
1.4.3 Огляд фреймворку NestJS..... 30
1.4.4 Огляд фреймворку Angular..... 32
1.5 Огляд засобів розгортання та управління інфраструктурою..... 37
1.5.1 Огляд інструменту контейнеризації Docker..... 37
1.5.2 Огляд інструменту оркестрації Kubernetes 38
1.5.3 Огляд хмарного сервісу DigitalOcean 41
1.5.4 Огляд CI/CD-інструменту GitHub Actions 42
1.6 Висновки з розділу 1..... 43

2 Проєктування освітньої платформи 45
2.1 Призначення розробки 45
2.2 Вимоги до програмного продукту 45
2.2.1 Функціональні вимоги 45
2.2.2 Технічні вимоги 47
2.3 Діаграма використання 47
2.4 Вимоги до користувацького інтерфейсу 49
2.5 Побудова мікросервісної архітектури 50
2.6 Проєктування баз даних 51
2.6.1 Проєктування бази даних сервісу Auth 51
2.6.2 Проєктування бази даних сервісу Courses 52
2.6.3 Проєктування бази даних сервісу Enrollments 54
2.7 Проєктування додатку користувацького інтерфейсу із застосуванням принципів мікросервісної архітектури 56
2.8 Висновки з розділу 2 58
3 Програмна реалізація 59
3.1 Розробка серверної частини 59
3.1.1 Вибір засобів розробки 59
3.1.2 Сервіс Auth 59
3.1.3 Сервіс Courses 61
3.1.4 Сервіс Enrollments 62
3.1.5 Сервіс Storage 63
3.1.6 Сервіс Payments 64
3.1.7 Сервіс Notifications 64
3.1.8 Сервіс API Gateway 65
3.2 Розробка користувацького інтерфейсу 66
3.3 Розгортання та управління інфраструктурою 73
3.3.1 Контейнеризація сервісів з Docker 73
3.3.2 Оркестрація контейнерів з Kubernetes 74
3.3.3 Впровадження процесів CI/CD 75

3.4 Висновки з розділу 3 77
Висновки 79
Перелік посилань 80

ВСТУП

Актуальність теми. Розвиток інформаційних технологій впливає на традиційні методи навчання, вимагаючи розробки нових підходів для забезпечення ефективного освітнього процесу. У зв’язку з цим актуальним стає питання розробки індивідуалізованих та інтерактивних платформ для навчання.

Доступність навчального матеріалу та гнучкість доступу для нього є важливими характеристиками онлайн-форми навчання. Освітня платформа відкриває можливості для навчання у будь-який час та з будь-якого місця, що робить її зручним інструментом для широкого кола користувачів. Завдяки розвитку інформаційних технологій стає можливим глобальний доступ до освіти стає можливим завдяки розвитку інформаційних технологій.

Індивідуалізація навчання стає все важливішою у контексті розвитку освіти. Освітня платформа може адаптуватися до індивідуальних потреб користувачів, створюючи персоналізовані навчальні програми, що сприяє кращому засвоєнню матеріалу.

Розвиток технологій відкриває нові можливості для ефективного навчання. Використання інтерактивних уроків стає все більш поширеним завдяки освітнім платформам, що забезпечує більш цікавий та ефективний процес навчання.

Також освітня платформа може бути ефективним інструментом для регулярного підвищення кваліфікації та розвитку навичок, що відповідає вимогам схильного до змін ринку праці.

Беручи до уваги описані можливості та переваги, яка може надавати освітня платформа, можна зробити висновок, що розробка такого програмного продукту є актуальною задачею.

Мікросервісна архітектура виникла відповідно до вимог до сучасних додатків, що мають бути гнучкими, масштабованими та стійкими. Основна

перевага полягає в складному масштабуванні, яке дозволяє ефективно використовувати ресурси та забезпечує високу продуктивність у сучасних програмних продуктах. Тому мікросервісна архітектура є потужним інструментом для створення програмних додатків, які відповідають вимогам сучасного інформаційного середовища.

Node.js став популярним інструментом для розробки серверної частини додатків. Використання Node.js у поєднанні з мікросервісною архітектурою дозволяє створювати додатки, які легко масштабуються під навантаження. В свою чергу, Angular є з популярним рішенням для побудови користувацьких інтерфейсів, що надає можливість створювати складні додатки з багатофункціональним інтерфейсом користувача.

Таким чином, розробка освітньої платформи на основі мікросервісної архітектури із використанням Node.js та Angular є актуальною задачею, яка має потенціал для підвищення доступності освіти та дослідження нових можливостей у цій сфері.

Мета і завдання дослідження. Мета і завдання дослідження полягають у вивчені засобів розробки, проєктуванні мікросервісної архітектури та у створенні програмної реалізації додатка освітньої платформи.

Об’єкт дослідження. Об’єктом дослідження є мікросервісна архітектура, платформа Node.js, фреймворк Angular.

Предмет дослідження. Предметом дослідження є проєктування мікросервісної архітектури та процес розробки освітньої платформи.

Методи дослідження. Для вирішення поставленої задачі використовуються наступні методи дослідження:

- аналіз існуючих рішень з предметної області;
- аналіз мікросервісної архітектури;
- аналіз засобів розробки;
- аналіз засобів розгортання та управління інфраструктурою.

Новизна одержаних результатів. Новизна одержаних результатів полягає у тому, що для розробки освітньої платформи була використана

мікросервісна архітектура, що дало змогу визначити переваги та недоліки даної архітектури.

Практичне значення одержаних результатів. Практичне значення одержаних результатів дослідження полягає у тому, що було розроблено додаток освітньої платформи, який дозволяє отримувати та надавати структуровані навчальні матеріали. Дана розробка є прикладом визначення переваг на недоліків застосування мікросервісної архітектури та використаних засобів розробки.

Апробація одержаних результатів. Результати дослідження були представлені на П’ятнадцятій Всеукраїнській, двадцять другій регіональній науковій конференції молодих дослідників «Актуальні проблеми математики та інформатики» [1].

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОБ'ЄКТУ ДОСЛІДЖЕННЯ

1.1 Характеристика предметної області

Освітні платформи – це онлайн-ресурси, які надають доступ до навчальних матеріалів та інструментів для навчання. Вони створюють зручне середовище для доступу до навчальних матеріалів, обміну знаннями та спільної роботи над навчальними завданнями. Освітні платформи забезпечують можливість для взаємодії між викладачами та студентами, сприяючи організації навчального процесу та розвитку компетенцій.

Основна мета освітніх платформ – забезпечення ефективного навчання шляхом зручної взаємодії між учасниками освітнього процесу. Вони надають можливість здійснювати пошук навчальних матеріалів, обирати курси за різними критеріями, такими як тематика, рівень складності, формат навчання тощо. Такі ресурси надають можливість отримувати освіту незалежно від місця проживання та часових обмежень.

Освітні платформи надають широкий спектр навчальних матеріалів, включаючи такі навчальні одиниці, як відеоуроки, інтерактивні завдання та тестування. Вони також забезпечують можливість індивідуалізованого навчання, де кожен студент може вибирати теми та темп навчання відповідно до своїх потреб та інтересів.

Такі освітні ресурси забезпечують інструменти для взаємодії між викладачами та студентами, включаючи можливість обговорення матеріалів, спільної роботи над завданнями та відстеження прогресу навчання. Вони створюють сприятливе середовище для активного навчання та обміну знаннями і досвідом.

Також освітні платформи можуть пропонувати платні курси або програми з розширеними можливостями. Такі навчальні програми можуть включати ряд переваг, таких як доступ до додаткових матеріалів, індивідуальну підтримку, більш глибокий розбір теми.

Освітні платформи надають можливість оцінювання та отримання зворотного зв’язку. Вони можуть надавати можливість проведення тестів, виконання завдань та оцінювання знань. Також вони можуть мати систему зворотного зв’язку, де студенти отримуватимуть поради та коментарі від викладачів щодо своїх досягнень.

Освітні платформи також можуть мати функціонал адаптивного навчання, який дозволяє налаштовувати навчальний процес під кожного студента окремо, враховуючи його поточні знання, навички та потреби. Це сприяє ефективнішому засвоєнню матеріалу та досягненню кращих результатів навчання.

Користувачі таких платформ мають свій особистий профіль, у якому вони можуть вказати коло своїх інтересів, досягнення, теми, які вивчають або викладають, а також отримувати рекомендації щодо курсів та матеріалів.

Також такі платформи можуть допомагати в організації розкладу занять, нагадувати про події та дедлайни, а також надавати можливість створення особистих розкладів для студентів.

Однією з важливих функцій освітніх платформ є можливість отримання сертифікатів після успішного завершення курсів або програм. Ці сертифікати є підтвердженням знань та навичок, отриманих студентом під час навчання на платформі.

Освітні платформи мають функціонал створення звітів, наприклад, про прогрес студентів, що можуть включати детальні оцінки за виконані завдання, результати тестів та інші форми оцінювання, а також аналіз активності користувачів на платформі, таких як час, витрачений на навчання, частота відвідувань та участь у форумах. Ці звіти можуть допомагати викладачам та адміністраторам платформи аналізувати прогрес студентів, ідентифікувати слабкі місця в навчанні та рекомендувати додаткові матеріали або курси для поліпшення результатів.

Отже, освітні платформи є ефективним інструментом для забезпечення доступної, якісної, інтерактивної та індивідуалізованої освіти. Створення

таких платформ сприяє розвитку навчальних можливостей та підвищенню рівня освіти.

1.2 Огляд існуючих освітніх платформ

1.2.1 Освітня платформа Prometheus

Проєкт Prometheus – це українська освітня платформа, яка пропонує широкий спектр онлайн-курсів та навчальних матеріалів. Проєкт був заснований у 2014 році з метою зробити якісну освіту доступною для всіх в Україні.

Платформа Prometheus відрізняється своєю місією зробити освіту доступною та ефективною для всіх, незалежно від їхнього рівня освіти чи географічного розташування. Широкий вибір курсів на різні теми, від базових до продвинутих, дозволяє користувачам розвивати свої здібності та знання в обраній галузі.

Кожен курс на платформі Prometheus має детальний опис, який включає в себе інформацію про зміст курсу, цілі навчання, методи викладання та очікувані результати. Інтерфейс перегляду навчального матеріалу курсу, наведений на рисунку 1.1. Користувачі також мають можливість читати відгуки та рейтинги інших учасників курсу, що допомагає їм зробити обґрунтований вибір.

Крім того, Prometheus активно співпрацює з викладачами та експертами з різних сфер, щоб забезпечити актуальність та якість навчального матеріалу. Такий підхід допомагає платформі залишатися в лідерах у сфері онлайн-освіти та задовольняти потреби навчання сучасного світу.

Загалом, освітня платформа Prometheus є цінним ресурсом для всіх, хто цікавиться вивченням нового або підвищенням свого рівня освіти. Безкоштовні онлайн-курси, платні курси та інші навчальні матеріали роблять його чудовим варіантом для навчання на будь-якому рівні.



Рисунок 1.1 – Інтерфейс платформи Prometheus

1.2.2 Освітня платформа Udemy

Освітня платформа Udemy – це онлайн-ресурс, який спеціалізується на наданні доступу до широкого спектру навчальних курсів та програм з різних галузей та предметів. Udemy надає можливість користувачам вибирати серед тисяч курсів, розроблених як незалежними експертами, так і викладачами з відомих університетів та організацій. Платформа дозволяє користувачам знаходити курси за різними критеріями, такими як тематика, рівень складності, мова викладання тощо.

Кожен курс на платформі Udemy має докладний опис, в якому вказуються теми, які будуть вивчатися під час курсу, які конкретні теми або модулі будуть охоплені, та які знання та навички учасники отримають після завершення. Також надається інформація про плановану тривалість курсу, яка може варіюватися від кількох годин до кількох місяців, залежно від обсягу та складності матеріалу.

Користувачі можуть переглядати відгуки та оцінки, залишені іншими учасниками, що розкривають їхні враження від курсу. Це дозволяє потенційним слухачам певного курсу отримати об’єктивну оцінку якості навчального матеріалу та ефективності викладання, що сприяє усвідомленому вибору та підвищує якість навчання на платформі.

Крім того, після успішного завершення курсів на платформі Udemy, користувачам надається можливість отримати сертифікати, що підтверджують опрацювання певного навчального матеріалу.

На платформі Udemy кожен користувач може створити свій особистий профіль, де зберігати історію участі в курсах, відстежувати прогрес навчання та зберігати власні налаштування. Крім того, користувачі можуть спілкуватися з викладачами та іншими студентами через та коментарі до курсів, що стимулює обмін досвідом та співпрацю між учасниками.

Крім можливості навчання для користувачів, платформа Udemy відкриває широкі можливості для викладачів, які можуть створювати та продавати власні курси. Це дає можливість експертам з різних галузей поділитися своїм знанням та досвідом з аудиторією по всьому світу. Викладачі мають можливість формування контенту курсу та вибір методів навчання. Такий підхід дозволяє стимулювати навчальну пропозицію платформи, забезпечуючи користувачам доступ до широкого спектру знань та навичок.

Платформа Udemy впровадила ініціативу з назвою «Udemy for Business», спрямовану на задоволення потреб корпоративних клієнтів. Ця ініціатива дозволяє компаніям отримувати доступ до широкого спектру навчальних матеріалів для підвищення кваліфікації своїх працівників. Завдяки цьому, компанії можуть створювати та налаштовувати персоналізовані програми навчання, адаптовані до своїх конкретних потреб та цілей.

У підсумку, платформа Udemy становить значну користь для широкого кола користувачів, надаючи доступ до різноманітних навчальних матеріалів з різних галузей. Завдяки детальному опису курсів, можливості перегляду відгуків та отримання сертифікатів, учасники можуть здобувати нові знання

та розвивати свої навички у зручний для них спосіб. Основний інтерфейс платформи, а саме взаємодія з навчальним матеріалом курсу, наведена на рисунку 1.2. Платформа сприяє саморозвитку, навчанню та професійному зростанню своїх користувачів, створюючи стимулююче та доступне середовище для навчання в будь-який час та з будь-якої точки світу.

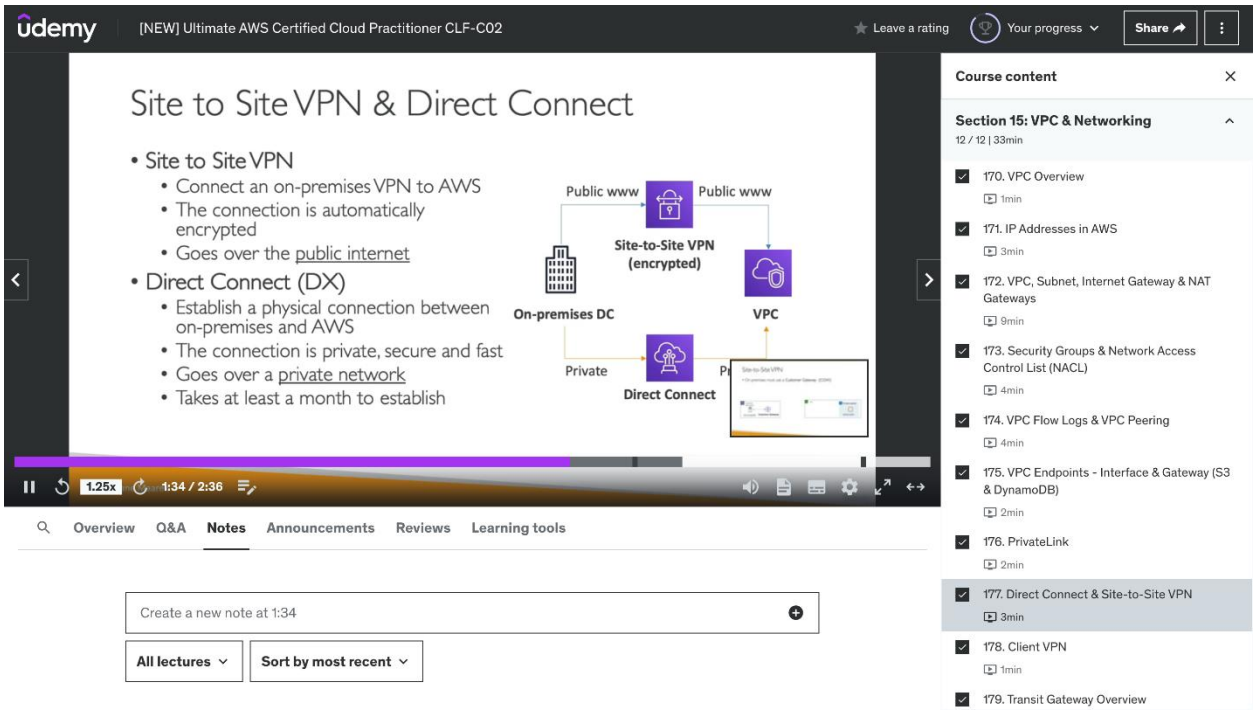


Рисунок 1.2 – Інтерфейс платформи Udemy

1.3 Аналіз мікросервісної архітектури

Мікросервісна архітектура (MSA) – це підхід до структурування програмного забезпечення, в якому додаток розбивається на невеликі, незалежні та самостійні компоненти, що називаються мікросервісами. Кожен мікросервіс виконує окрему функцію та взаємодіє з іншими мікросервісами через API.

Основні принципи MSA включають в себе розподілену розробку, автономність мікросервісів, незалежність розгортання та масштабування, а також використання легковагих протоколів для комунікації між сервісами.

Вибір мікросервісної архітектури вимагає ретельного зважування переваг та недоліків з метою забезпечення впровадження оптимального рішення для конкретного проєкту.

1.3.1 Переваги мікросервісної архітектури

Мікросервісна архітектура пропонує низку переваг, таких як незалежне масштабування, швидке оновлення, стійкість до відмов, гнучкість технологій та кращий користувацький досвід.

Одна з головних переваг мікросервісної архітектури полягає у здатності незалежно масштабувати та оновлювати окремі компоненти системи. Кожен сервіс може бути розгорнутий та масштабований незалежно від інших, що дозволяє оптимізувати використання ресурсів та ефективно реагувати на зміни в навантаженні. Мікросервіси дозволяють зменшити час між внесенням змін у код та його розгортанням в продуктивному середовищі. Кожен сервіс може бути розгорнутий незалежно, що дозволяє здійснювати швидкі оновлення та внесення змін без перерв у роботі системи в цілому.

У разі виникнення проблеми у одному сервісі, інші сервіси можуть продовжувати працювати. Це дозволяє підвищити стійкість до відмов та забезпечити неперервну доступність для користувачів.

Використання мікросервісної архітектури дає можливість використовувати різні технології та інструменти для кожного сервісу в залежності від його потреб та вимог. Це дає можливість використовувати найбільш ефективні інструменти для конкретних завдань та підвищує ефективність роботи [2].

Також вибір даної архітектури спрощує процес розробки та тестування. Команда розробки може поділятися на групи, які працюватимуть над окремим мікросервісом, не впливаючи на роботу інших. Це дозволяє паралельно вести розробку та тестування, що значно прискорює загальний процес.

У підсумку, мікросервісна архітектура надає переваги у розробці та роботі програмного забезпечення, сприяє швидкому впровадженню, масштабуванню та забезпечуючи гнучкість вибору технологічного стеку. Вона дозволяє підвищити стійкість системи, забезпечити ефективне використання ресурсів та створювати кращий користувацький досвід.

1.3.2 Недоліки мікросервісної архітектури

Головним недоліком мікросервісної архітектури є її складність, що виявляється у складності побудови правильної архітектури, налаштування розгортання, масштабування та моніторингу.

При використанні мікросервісної архітектури виникає складність налаштування розгортання та масштабування системи. Кожен сервіс потребує індивідуального розгортання та масштабування, що вимагає більших витрат ресурсів. Крім того, кожному сервісу необхідно відслідковувати навантаження та реагувати на зміни, що ускладнює процес управління інфраструктурою.

Один з недоліків полягає у складності управління. З ростом кількості сервісів у системі зростає складність їх координації та моніторингу. Це може призвести до складнощів у виявленні та виправленні помилок, а також у управлінні розгортанням нових версій сервісів.

Проблемою є складність у моніторингу та налагодженні системи. З ростом кількості сервісів стає складніше контролювати їх роботу та взаємодію. Необхідно використовувати потужні інструменти моніторингу та аналізу, щоб ефективно відслідковувати роботу системи та вчасно виявляти проблеми.

Недоліком мікросервісної архітектури є складність у розробці та тестуванні. Розробка та тестування мікросервісів може бути складнішою порівняно з монолітними додатками через їх розділену природу. Це вимагає додаткових зусиль у забезпеченні сумісності та інтеграції між сервісами.

Ще одним недоліком мікросервісної архітектури є складність реалізації транзакцій та забезпечення відповідності принципам ACID. У монолітних додатках транзакції можуть бути реалізованими на рівні баз даних. Однак у мікросервісній архітектурі, де кожен сервіс має свою власну базу даних, забезпечення атомарності транзакцій стає складнішою задачею. Виникають проблеми з узгодженістю даних між різними сервісами та забезпеченням консистентності під час виконання транзакцій. Ця складність може призвести до збоїв у відновленні даних або втрати консистентності, особливо в умовах високого навантаження.

Отже, мікросервісна архітектура супроводжується складністю у побудові, управлінні та моніторингу системи, а також у розробці та тестуванні сервісів.

1.3.3 Типи та засоби комунікації мікросервісів

Комунікація між мікросервісами супроводжується рядом проблем, зокрема, невизначеності доступності сервісів, можливих затримок передачі даних, втрати або дублювання пакетів, а також навантаженням на мережевий трафік. Забезпечення надійності, відмовостійкості та оптимізація мережевої взаємодії вимагають складних рішень, особливо при збільшенні кількості мікросервісів.

Взаємодія між мікросервісами може відбуватися за синхронними та асинхронними методами [3]. У синхронному способі спілкування клієнт виконує виклик до мікросервісу та чекає на отримання відповіді перед тим, як продовжити виконання своєї роботи. Це може призвести до блокування ресурсів, особливо у випадку, коли викликаний сервіс має велику затримку або недоступний. З іншого боку, у асинхронному способі клієнт продовжує виконання своєї роботи без очікування відповіді від мікросервісу. Це дозволяє уникнути блокування та оптимізувати використання ресурсів системи. У

залежності від вимог до системи та характеру взаємодії, обирають той чи інший метод, який найкраще відповідає потребам додатку.

Серед синхронних методів взаємодії мікросервісів можна виділити REST та gRPC [3], які набули широкого застосування.

REST (Representational State Transfer) – архітектурний стиль, який базується на принципах ресурсів, URI і HTTP протоколу [4]. Він є одним із найпоширеніших підходів до побудови вебсервісів і дозволяє звертатися до ресурсів за допомогою стандартних HTTP методів, таких як забезпечує простоту, швидкість та доступність, що робить його популярним в розробці мікросервісних додатків.

gRPC – одна з реалізацій RPC (Remote Procedure Call), розроблений компанією Google [5]. gRPC використовує протокол HTTP/2 для передачі даних, що забезпечує високу швидкість і ефективність мережевого взаємодії. Протокол HTTP/2 має ряд переваг перед його попередником HTTP/1.1, включаючи мультиплексування, яке дозволяє одночасно використовувати кілька потоків на одному з’єднанні, а також стискання заголовків, яке зменшує обсяг переданих даних і підвищує швидкість передачі [6]. Особливість gRPC полягає в можливості генерувати клієнтські та серверні частини з інтерфейсного опису служби за допомогою Protocol Buffers (protobuf) [7], що спрощує процес розробки та забезпечує структурованість у взаємодії між мікросервісами [8].

Одним із асинхронних методів взаємодії мікросервісів є меседжинг [9], який представлений такими інструментами, як RabbitMQ, ActiveMQ, AWS SQS, кожний з яких має свої особливості. Крім цього, існує метод стрімінгу, представлений, наприклад, Apache Kafka. Цей підхід передбачає постійний потік даних, що дозволяє обробляти їх після відправлення без очікування миттєвої відповіді.

RabbitMQ – брокер повідомлень з відкритим кодом, який використовується для асинхронної передачі повідомлень між різними компонентами програмного забезпечення [10]. Він реалізовує протокол AMQP

(Advanced Message Queuing Protocol), який забезпечує надійну, гнучку та масштабовану передачу повідомлень. RabbitMQ також надає багато функцій, таких як можливість маркування повідомлень, маршрутизація, управління чергами та моніторинг, що робить його популярним рішенням для реалізації мікросервісних архітектур та інших розподілених систем.

1.3.4 Патерни та практики мікросервісної архітектури

Існує ряд патернів та практик, які допомагають покращити роботу розподіленої системи, її розробку, впровадження та управління.

API-first підхід до розробки мікросервісної архітектури полягає в тому, що розробка починається з визначення API, які визначають спосіб взаємодії між компонентами системи, що забезпечує заздалегідь визначені формати даних та типи запитів та відповідей [11].

Decompose by Subdomain – патерн для переходу з монолітної системи на розподілену, який передбачає розбиття системи на окремі служби, що відповідають бізнес-піддоменам [11]. Під бізнес-піддоменом розуміють певну функціональну область або бізнес-кейс, який має чітко визначені межі та відповідальність.

API Gateway – патерн побудови розподілених систем, згідно якого створюється окремий мікросервіс, який виступає як проміжний рівень між клієнтом та набором мікросервісів. Він використовується для централізованого управління, контролю та захисту доступу до різних ресурсів через їх API. API Gateway дозволяє забезпечити єдиний доступ до різних сервісів, а також може виконувати функції маршрутизації трафіку, перетворення протоколів та форматів даних, автентифікації, авторизації, моніторингу та інших аспектів керування та безпеки.

Database per service – патерн побудови розподілених систем, який передбачає використання окремих баз даних для кожного мікросервісу в

системі [12]. Кожен сервіс має власну базу даних, яка містить лише дані, необхідні для виконання його зони відповідальності. Використання однієї бази даних кількома сервісами потенційно може призвести до анти-патерна Distributed Monolith. Сервіси мають використовувати виключно свою власну базу даних та взаємодіяти між собою тільки через API. Це дозволяє забезпечити автономність сервісів та зменшити залежності. Однак цей патерн також може ускладнити забезпечення узгодженості даних в системі. Оскільки кожен сервіс має свою власну базу даних, виникає проблема забезпечення консистентності даних між ними. Це може призвести до ситуацій, коли різні сервіси мають різні версії даних або неправильно відображають поточний стан системи.

Патерни, які використовуються для керування транзакціями в мікросервісах:

- Saga Pattern: використовується для керування послідовною послідовністю локальних транзакцій, кожна з яких відповідає за модифікацію даних в одному сервісі [11] (якщо виконання певної транзакції в середині саги виконалося з помилкою, виконується компенсуюча транзакція для скасування внесених змін);
- API Gateway-based Transactions: всі запити до мікросервісів проходять через API Gateway (Gateway може керувати транзакціями, використовуючи глобальні транзакції або керування координатором);
- Compensating Transaction Pattern: підходить для сценаріїв, де виправлення помилок можна здійснити шляхом виконання обернених дій (кожна транзакція має свою відповідну компенсуючу транзакцію для відміни змін, які були зроблені);
- Event-driven Transactions: сервіси реагують на події, які виникають в інших сервісах, і відповідно реагують;
- Distributed Transactions with Two-Phase Commit: цей підхід вимагає двофазового підтвердження (2PC), де є координатор транзакції, який керує підтвердженням або відміною транзакцій в усіх відповідних сервісах.

Service Registry/Discovery – патерн, що використовується для знаходження та комунікації між різними мікросервісами, які можуть бути динамічно масштабованими та розподіленими на різних серверах або контейнерах [11]. Service Registry – це централізований реєстр, де кожен мікросервіс реєструє свою адресу та метадані, Discovery – це процес знаходження доступних екземплярів сервісу. Цей патерн дозволяє забезпечити гнучкість та масштабованість мікросервісної архітектури, оскільки сервіси можуть додаватися або вилучатися динамічно, а всі залежності автоматично відновлюються і оновлюються без необхідності ручного втручання.

Circuit Breaker – патерн проєктування, який використовується для підвищення надійності та стійкості розподілених систем [11]. Основна ідея полягає в тому, щоб уникнути надмірного навантаження на сервіс або компонент системи, якщо він виявляється недоступним або несправним. Якщо певний мікросервіс недоступний, може виконуватися альтернативна логіка, наприклад повернення попередньо збережених даних. Патерн Circuit Breaker дозволяє системі здійснювати ефективне керування помилками та забезпечує зниження впливу недоступності або несправностей на загальну працездатність системи.

Патерни та практики мікросервісної архітектури допомагають розробляти гнучкі, масштабовані та надійні розподілені системи. Вибір певного патерну повинен бути прийнятий на основі ретельного аналізу поточної архітектури та вимог проєкту.

1.3.5 Застосування принципів мікросервісної архітектури при побудові користувацьких інтерфейсів

Microfrontends – підхід до розробки користувацьких інтерфейсів, де додаток розбивається на окремі незалежні модулі, які можуть бути розроблені, розгорнуті та масштабовані незалежно один від одного [13]. Кожен

мікрофронтенд відповідає за певну частину функціональності або відображення.

Основні концепції та переваги використання Microfrontends підходу в розробці клієнтських інтерфейсів:

– незалежність розробки та розгортання: кожен мікрофронтенд може бути розроблений та розгорнутий незалежно від інших, що дозволяє різним командам працювати паралельно [14];
– швидке впровадження змін: завдяки незалежному розгортанню мікрофронтендів, внесені зміни можуть бути швидко впроваджені без впливу на інші частини системи;
– гнучкість технологій: кожен мікрофронтенд може використовувати власні бібліотеки, що дозволяє використовувати найбільш доцільні інструменти для кожної окремої частини додатку.

Однак вибір Microfrontends підходу супроводжується певними ризиками та складнощами:

– збільшення складності: Microfrontends роблять архітектуру системи більш складною, що може ускладнити її розуміння та підтримку;
– ускладнення інфраструктури: використання Microfrontends може призвести до ускладнення інфраструктури розробки та впровадження, оскільки потрібно буде підтримувати інструменти для кожного окремого додатку;
– можливі проблеми з сумісністю технологій;
– складність тестування: тестування Microfrontends може бути складнішим через їхню розподілену природу, різноманіття технологій, залежність від інфраструктури, обмежену кількість спеціалізованих інструментів та необхідність у специфічних навичках;
– управління станом додатку: управління станом між різними мікрофронтендами може стати складнішим завданням, оскільки потрібно буде забезпечити синхронізацію та консистентність даних між ними.

Для вирішення задачі управління станом у мікрофронтендах можна виокремити наступні методи:

- використання локального сховища: об’єкт localStorage має можливість відстеження змін даних, що в ньому зберігаються;
- реалізація спільного API: підхід полягає у створені окремого мікрофронтенду, що відповідає за роботу з усіма запитами, та до якого звертаються інші мікрофронтенди;
- створення експортованих функцій: метод полягає у створені спільного модуля, який експортує функції для отримання або зміни спільного стану;
- використання кастомних подій браузера: метод передбачає визначення подій, які передають інформацію про зміни стану, та їх прослуховування іншими частинами додатку для відповідного оновлення стану;
- використання сторонніх бібліотек: існують декілька бібліотек для управління станом у мікрофронтендах, зокрема – бібліотеки Zustand та redux-micro-frontend.

Існує декілька підходів до впровадження мікрофронтендів, серед яких наступні:

- мікрофронтенди на базі IFrames: цей підхід передбачає вбудовування кожного мікрофронтенду в власний IFrame, з односторінковим застосунком (SPA) у ролі контейнера. SPA відповідає за відображення IFrame та координацію зв’язку між ними. Цей підхід дозволяє досягти максимальної незалежності між мікрофронтендами, оскільки кожен IFrame працює як самодостатня програма. Однак це також може призвести до збільшення складності та потенційних проблем з продуктивністю, оскільки кожен IFrame повинен бути завантажений та управляється окремо;
- інтеграція мікрофронтендів на клієнтській стороні: у цьому підході мікрофронтенди інтегруються в SPA на етапі збирання, а SPA відповідає

за відображення та оновлення інтерфейсу – такий підхід може забезпечити кращий користувацький досвід, однак це також може призвести до збільшення складності та збільшення обсягу коду, оскільки один окремий додаток має керувати всіма мікрофронтендами;

– мікрофронтенди на базі API-шлюзу: цей підхід передбачає використання API-шлюзу для маршрутизації запитів до відповідного сервісу мікрофронтендів. API-шлюз виступає як єдиний вхід для всіх запитів, що дозволяє мікрофронтендам залишатися незалежними та слабко зв’язаними (це може бути хорошим варіантом для додатків, які вимагають високого рівня модульності та гнучкості, але водночас може збільшити складність загальної системи).

Перед тим, як приймати рішення про впровадження Microfrontends, важливо ретельно оцінити ризики та переваги, а також врахувати специфічні потреби та вимоги проєкту. Ось кілька факторів, які слід врахувати перед прийняттям рішення:

– розмір та складність проєкту: використання Microfrontends виправдано для великих та складних проєктів, де важлива гнучкість та модульність;
– метод розробки: Microfrontends краще підходять для команд, які використовують agile-методології розробки та практикують постійне розгортання;
– навички та досвід команди: розробка, тестування та підтримка додатку з мікрофронтендів потребує знань додаткових інструментів;
– бізнес-задачі проєкту: Microfrontends можуть допомогти досягти таких цілей, як покращення швидкості виходу на ринок або підвищення продуктивності розробки.

У підсумку, Microfrontends дозволяють розділити проєкт користувацького інтерфейсу на невеликі незалежні частини, що полегшує розробку, розгортання та підтримку великих та складних вебдодатків, але його слід використовувати враховуючи додаткові ризики та складність.

1.4 Огляд засобів розробки

1.4.1 Огляд СУБД PostgreSQL

PostgreSQL – це система управління реляційними базами даних з відкритим кодом [16]. Особливості PostgreSQL:

- відкритий код: код PostgreSQL доступний для загального перегляду і використання;
- сумісність з SQL: PostgreSQL підтримує стандарт SQL, а також додає розширення для реалізації додаткових можливостей, таких як розширене управління даними та специфічні функції;
- масштабованість: PostgreSQL може обробляти великі обсяги даних, підтримує різні типи індексів, які допомагають оптимізувати швидкість запитів та підтримує кластеризацію;
- транзакції: PostgreSQL гарантує надійність операцій з даними завдяки підтримці властивостей ACID – кожна транзакція є атомною, послідовною, ізольованою та довготривалою, що забезпечує цілісність даних;
- реплікація і резервне копіювання: PostgreSQL має можливості для реплікації даних, що дозволяє створювати резервні копії даних і забезпечувати високу доступність системи;
- складні типи даних: PostgreSQL підтримує складні типи даних, такі як JSON, XML, та навіть геопросторові дані через розширення PostGIS;
- налаштування: PostgreSQL дозволяє додавання модулів і плагінів – є можливість додавання нових типів даних, функцій, операторів і індексів, що дозволяє адаптувати базу даних до конкретних потреб;
- безпека: PostgreSQL пропонує розширені можливості безпеки, такі як підтримка SSL для шифрування з’єднань.

PostgreSQL використовується в різних галузях, від невеликих додатків до корпоративних систем, в тому числі, розглянута СУБД може використовуватися як сховище даних у мікросервісній архітектурі.

1.4.2 Огляд платформи Node.js

Node.js – це платформа для виконання JavaScript, яка дозволяє виконувати JavaScript-код на сервері [18]. Платформа побудована на двигуні JavaScript V8 від Google і надає можливість створювати швидкі та масштабовані серверні додатки. З використанням Node.js розробники можуть використовувати JavaScript для створення серверних програм, що включає обробку запитів, роботу з базами даних, взаємодію з мережевими протоколами і багато іншого.

Можливості Node.js можуть бути застосовані у розробці різноманітних додатків, включаючи:

- серверні API, які є швидкими та ефективними і мають мінімальне навантаження на ресурси сервера. Node.js добре підходить для обробки великої кількості одночасних з’єднань;
- вебдодатки: Node.js дозволяє створювати вебдодатки різної складності, екосистема Node.js містить бібліотек для їхньої розробки;
- додатки реального часу: Node.js добре підходить для створення додатків, які потребують частого та швидкого оновлення;
- додатки з мікросервісною архітектурою: Node.js може бути обрано як технологію для створення мікросервіса;
- інструменти командної рядка: Node.js може використовуватися для створення інструментів командної рядка, які автоматизують завдання або надають користувачам інтерфейс для взаємодії з серверними системами.

Це лише деякі з багатьох типів додатків, які можна розробити за допомогою Node.js. Node.js – це універсальна платформа, яку можна використовувати для створення широкого спектру програмних рішень.

Ключовим компонентом Node.js є libuv (Unicorn Velociraptor Library) – це кросплатформна бібліотека C з відкритим кодом, яка використовується для забезпечення асинхронного вводу-виводу (I/O) на основі циклу подій [18].

libuv надає асинхронні API для різних операцій вводу-виводу, що дозволяє Node.js та іншим платформам ефективно обробляти декілька запитів одночасно, не блокуючи основний потік. libuv підтримує широкий спектр операційних систем.

npm (Node Package Manager) – це стандартний менеджер пакетів для платформи Node.js [18]. Він використовується для управління пакетами, які можуть бути бібліотеками, фреймворками або іншими інструментами для розробки. npm дозволяє легко встановлювати, оновлювати та видаляти залежності проєктів. Враховуючи велику кількість пакетів, доступних у репозиторії пакетного менеджеру, він дозволяє швидко знаходити та використовувати готові рішення для своїх проєктів. npm дає можливість публікувати свої пакети. npm-пакет може бути приватним, доступними лише для власних проєктів, або публічними і доступним для загального використання. Підтримується версіювання пакетів: npm дає можливість вказувати конкретну версію пакету для встановлення та при публікації власного пакету, що забезпечує сумісність між проєктами, які посилаються на даний пакет як залежність.

Отже, використання платформи Node.js є гарним вибором для розробки серверної частини системи, в тому числі й для реалізації мікросервісної архітектури.

1.4.3 Огляд фреймворку NestJS

Екосистема платформи Node.js містить багато фреймворків для створення серверних додатків, кожен з яких має свої особливості та підходи до розробки, серед найпопулярніших – NestJS.

NestJS – це фреймворк, призначений для розробки масштабованих серверних додатків на платформі Node.js [19]. NestJS виділяється серед інших фреймворків для Node.js своєю чіткою модульною структурою, яка була розроблена на основі концепцій структури фреймворку Angular.

Модульна структура NestJS полягає в розбитті додатку на окремі, незалежні та керовані модулі. Кожен модуль відповідає за певну функціональність, що робить його чітко визначеним для розробки та тестування. Переваги модульної архітектури NestJS полягають в покращені організації коду, в легшому тестуванні, модифікації та повторному використанні модулів, що спрощує процес розширення та зміни кодової бази. Цей підхід також полегшує роботу команди над великими проєктами, оскільки різні частини команди можуть працювати над окремими модулями з меншою кількістю конфліктів.

Окрім модулів, NestJS пропонує чітку структуру інших компонентів, таких як контролери, сервіси, провайдери, проміжне програмне забезпечення, exception filters, pipes, guards, interceptors. Контролери відповідають за обробку HTTP-запитів та генерування відповідей, сервіси містять бізнес-логіку додатку, провайдери створюють та надають залежності. Разом ці компоненти забезпечують чітку та структуровану розробку серверних додатків.

NestJS реалізує ін’єкцію залежностей (DI) для спрощення та покращення процесу розробки. Цей принцип проєктування програмного забезпечення полягає в тому, що компоненти не створюють свої залежності самостійно, а отримують їх ззовні. DI знижує зв’язаність компонентів, спрощує модифікацію коду та полегшує написання модульних тестів. NestJS використовує декоратори для визначення та ін’єкції залежностей.

У NestJS використовується TypeScript – мова програмування, яка розширює можливості JavaScript шляхом додавання статичної типізації та інших функцій [17]. TypeScript використовується для створення додатків у NestJS через свою можливість визначати типи даних для змінних, параметрів функцій та інших об’єктів, що дозволяє виявляти помилки на етапі компіляції.

Nest CLI – це командно-рядковий інтерфейс (CLI), який полегшує роботу з NestJS фреймворком. Він дозволяє ініціалізувати NestJS проєкти, генерувати шаблони коду компонентів фреймворку, допомагає у локальній розробці та тестуванні.

NestJS надає можливості для роботи з мікросервісною архітектурою. Даний фреймворк надає зручні інструменти та підходи для побудови, керування та масштабування мікросервісних додатків. У NestJS мікросервіси створюються за допомогою пакету @nestjs/microservices та є окремими незалежними сервісами, що взаємодіють через мережу. Також NestJS підтримує інтеграцію з різними сервісами та протоколами, що часто використовуються при розробці розподілених систем, включаючи Redis, MQTT, NATS, RabbitMQ, Kafka і gRPC. Він забезпечує підтримку кожного з цих сервісів, що дозволяє створювати розподілені та масштабовані додатки з різноманітними вимогами до взаємодії та обміну даними.

NestJS дозволяє працювати з різними базами даних та підтримує інтеграцію з ORM (Object-Relational Mapping) та ODM (Object-Document Mapping) бібліотеками. До цих бібліотек належать TypeORM, MikroORM, Sequelize, Mongoose, Prisma [20] та інші.

NestJS дозволяє інтегрувати OpenAPI в свої додатки, що забезпечує автоматичне створення документації API на основі визначень маршрутів та схеми даних. Це дозволяє автоматично генерувати та оновлювати документацію API, що значно спрощує процес розробки та забезпечує актуальність документації.

Описані особливості фреймворку NestJS роблять його гарним вибором для розробки сучасних та масштабованих серверних рішень.

1.4.4 Огляд фреймворку Angular

Angular – фреймворк, розроблений компанією Google, який дозволяє створювати додатки для різних платформ, включаючи веб, мобільні пристрої та десктоп [21].

Фреймворк AngularJS був представлений в 2010 році. В 2016 році відбувся реліз повністю переписаного нового фреймворка під назвою Angular, побудованого на основі TypeScript.

Angular використовує архітектуру MVVM (Model-View-ViewModel). У цій архітектурі, Model представляє дані, з якими користувач взаємодіє, View – це представлення цих даних, а ViewModel виступає як посередник між Model і View. У компонентній архітектурі Angular кожен компонент виконує функції як View, так і ViewModel.

Структура Angular базується на розподіленні додатку на модулі та компоненти. Кожен Angular додаток складається з модулів, які відповідають за певну функціональність додатку. Модулі Angular дозволяють організувати кодову базу проєкту та використовувати інші функціональності з інших модулів. Компоненти є основною будівельною одиницею Angular додатків. У фреймворку Angular є багато інших абстракцій для різних задач, наприклад, сервіси, директиви, pipes, guards, interceptors. Сервіси відіграють важливу роль у розділенні логіки додатку на повторно використовувані та незалежні частині, які забезпечують функціональність та дані, доступні в усьому додатку, або, при певних налаштуваннях, на певному його рівні.

Dependency Injection (DI) – концепція, що дозволяє використовувати об’єкти без прямого їхнього створення, передаючи відповідальність за це окремому механізму – контейнеру DI. У Angular є вбудована система DI, яка надає простий та зручний спосіб керування залежностями в програмі. У Angular DI реалізовний за допомогою провайдера – конфігураційного об’єкта, який описує, який об’єкт необхідно створити та налаштування його створення. Провайдер може бути визначений у різних сутностях, таких як компонент, модуль або сервіс, і може бути використаний в будь-якому місці, де потрібна відповідна залежність. DI дозволяє ефективно управляти залежностями та полегшує тестування та рефакторинг коду.

RxJS (Reactive Extensions for JavaScript) – бібліотека для JavaScript, що полегшує роботу з потоками даних та подіями, ґрунтуючись на принципах реактивного програмування [22]. Вона надає інструменти для композиційної обробки потоків даних, використовуючи підхід функціонального програмування, де дані розглядаються як потоки подій, що піддаються обробці

за допомогою спеціальних операторів. В основі RxJS лежить шаблон Observable, який описує потік даних. Бібліотека використовує три ключові сутності: Observable, яке є джерелом асинхронних даних та генерує події; Observer, який є спостерігачем, що підписується на Observable для отримання та обробки подій; та Subscriber, який є конкретною реалізацією Observer і підписується на Observable та обробляє його події. Angular має вбудовану підтримку RxJS, що дозволяє розробникам використовувати спостережувані об’єкти та оператори RxJS безпосередньо в своїх Angular-проєктах.

State management (управління станом) – процес управління даними програми в процесі її виконання. Angular дозволяє реалізувати різні підходи до управління глобальним станом додатка, з найпопулярніших рішень з використанням сторонніх бібліотек – NgRx, NgXS, Akita, Elf та StateAdapt. Також управління станом додатка можна реалізувати за допомогою Angular-сервісів та RxJS, використовуючи різноманітні підходи.

NgRx – бібліотека для управління станом додатків в Angular. Вона надає засоби для ефективного керування складним станом додатку, забезпечуючи прозору та передбачувану модель управління даними [23]. Однією з ключових особливостей NgRx є використання RxJS. NgRx базується на шаблоні Redux, і включає такі ключові елементи як Store, Actions та Reducers. Store – централізований інструмент для зберігання стану додатку. Actions представляють собою об’єкти, які необхідні для тригеру зміни стану. Reducers – функції, які обробляють ці дії та оновлюють стан додатку. Інша важлива складова NgRx – це effects, які дозволяють обробляти асинхронні операції і їхній вплив на стан додатку. NgRx допомагає створювати Angular-проєкти, забезпечуючи чітке керування станом додатку.

Zone.js – бібліотека, яка відстежує та керує асинхронними операціями, в Angular вона використовується для реалізації механізму зон, що дозволяє встановлювати контекст виконання коду [21]. Шляхом використання зон Angular може контролювати асинхронні операції та визначати, у якому контексті вони виконуються, на основі чого в Angular реалізовано механізм виявлення змін.

Крім можливості створення SPA, Angular дозволяє розробляти додатки з SSR (Server-Side Rendering). SSR), що покращує SEO та зменшує час до першого відображення. Під час використання SSR код Angular-додатку виконується на сервері, а повністю сформований шаблон надсилається клієнту. Формування шаблону на сервері відбувається тільки при першому запиті, після нього додаток працює як SPA. Спочатку можливість розробляти додатки з SSR була надана Angular Universal – офіційного інструменту для реалізації SSR в Angular. З версії Angular 17 функція SSR була покращена та додана до самого фреймворку.

Angular надає широкий набір інструментів та підходів для тестування, що дозволяє забезпечити надійність та стабільність додатків. Angular надає можливість проводити різні види тестування:

– unit-тестування – тестування ізольованих одиниць коду (Angular надає функціональність для створення і запуску unit-тестів за допомогою фреймворків для тестування, наприклад, за замовчуванням використовується фреймворк Jasmine та ранер тестів Karma);
– інтеграційне тестування – тестування взаємодії між різними компонентами та модулями додатку (Angular дозволяє створювати інтеграційні тести, які перевіряють взаємодію компонентів та їхню відповідність очікуваній поведінці);
– end-to-end (E2E) тестування – тестування, яке моделює поведінку реального користувача, виконуючи взаємодію з додатком через його інтерфейс (Angular має вбудовану підтримку для E2E тестування за допомогою фреймворку Protractor).

Angular постійно оновлюється та розширюється командою розробників, що дозволяє користувачам отримувати нові функціональності та вдосконалення. Завдяки своїй широкій спільноті та вичерпній документації, Angular є популярним вибором для розробки додатків будь-якої складності.

Nx – набір інструментів для розробки на основі Node.js, який допомагає створювати та керувати монолітами та мультипроєктами на базі Angular,

React, NestJS та інших бібліотек. Основною функціональністю Nx є генерація проєктів для різних технологій за допомогою шаблонів та команд CLI. Nx автоматизує багато завдань розробки, таких як тестування, лінтування, публікація та розгортання. Nx також допомагає керувати залежностями між проєктами, роблячи код більш організованим та керованим. Цей інструмент може використовуватися як для розробки монолітних, так і мультипроєктних додатків. Крім того, Nx інтегрується з популярними IDE, що покращує досвід розробки. Nx – це потужна відкрита система збірки, яка надає інструменти та методи для підвищення продуктивності розробники, оптимізації CI та забезпечення якості кодової бази.

Module Federation – метод, за допомогою якого кодова база може бути розділена на модулі, які можуть бути розгорнутими незалежно, та який надає можливість використання модулів в додатках під час їхнього виконання [15]. Цей підхід дозволяє реалізувати Microfrontends. Він став поширеним після реалізації плагіна ModuleFederationPlugin у Webpack. Nx має підтримку Module Federation для Angular та React.

За допомогою Nx можна налаштувати монорепозиторій, що містить різні Angular-додатки та Angular-бібліотеки проєкту. Монорепозиторій – це підхід до керування кодовою базою, при якому всі додатки та бібліотеки розробляються в одному централізованому репозиторії. Кожен додаток в монорепозиторії Nx може мати власні налаштування, включаючи конфігурацію тестування, лінтування, збірки та розгортання. За допомогою Nx можливо керувати проєктами з численними Angular додатками та бібліотеками, зберігаючи при цьому структурованість кодової бази.

У підсумку, інструмент Nx з фреймворком Angular є гарним вибором для створення клієнтських інтерфейсів, в тому числі – для реалізації Microfrontends.

1.5 Огляд засобів розгортання та управління інфраструктурою

1.5.1 Огляд інструменту контейнеризації Docker

Docker – інструмент контейнеризації, що забезпечує спрощення розгортання, управління та масштабування програмних додатків за допомогою контейнерів [25].

Docker створює над програмним забезпеченням стандартизовані одиниці, відомі як контейнери. Кожен контейнер містить всі необхідні компоненти для роботи додатка, включаючи бібліотеки, системні інструменти, код і середовище виконання. Це дозволяє швидко розгортати та масштабувати програми в будь-якому середовищі.

Образи Docker є статичними шаблонами для створення контейнерів. Docker-образи які можна зберігати, поширювати та використовувати для створення нових контейнерів.

Volumes в Docker – це механізм, який дозволяє зберігати дані поза контейнерами, забезпечуючи їх стійкість та доступність навіть після видалення або перезапуску контейнера [25]. Volumes використовуються для зберігання даних, таких як бази даних, файли конфігурації, логи та інше. Підключення локальної директорії з кодом до контейнера використовується при локальній розробці, адже дозволяє вносити зміни до коду без необхідності збирати новий образ.

Dockerfile – послідовність команд, які Docker використовує для побудови образу [25]. Кожна команда в Dockerfile створює новий шар образу, який додає до попередніх шарів, перший з яких зазвичай є мінімальною інсталяцією Linux.

Docker Compose – інструмент, який дозволяє визначити та запустити багатоконтейнерні додатки за допомогою єдиного файлу конфігурації [25]. Якщо Dockerfile описує, як побудувати один контейнер, то Docker Compose дозволяє керувати цілим набором взаємопов’язаних контейнерів та визначати

залежності між ними. Крім того, Docker Compose автоматично створює мережі для взаємодії між контейнерами. Хоча Docker Compose не є повноцінним оркестратором, він дозволяє виконувати базові операції оркестрації, такі як масштабування, перезапуск і зупинка сервісів.

Реєстри Docker – це централізовані сховища, де зберігаються готові до використання контейнерні образи. Одним з найпопулярніших реєстрів є Docker Hub, який надає безкоштовний публічний реєстр та можливість створення приватних реєстрів.

Наразі Docker є найпопулярнішим інструментом для контейнеризації. Контейнеризація є зручною технологією при роботі з різними додатками, включаючи додатки з мікросервісною архітектурою, адже вона забезпечує ізоляцію, портативність та розгортання кожного мікросервісу в окремому контейнері.

1.5.2 Огляд інструменту оркестрації Kubernetes

Kubernetes – платформа з відкритим вихідним кодом, розроблена компанією Google, яка призначена для управління контейнеризованими додатками [26]. Kubernetes забезпечує автоматизацію розгортання, масштабування та управління контейнерами, спрощуючи роботу з ними.

Kubernetes підтримує роботу з тими інструментами контейнеризації, які підтримують інтерфейс CRI (Container Runtime Interface). CRI підтримують такі інструменти, як Docker, containerd, CRI-O, rkt.

Master Node – елемент Kubernetes , що керує кластером, управлінням станом і координацією компонентів, складається з таких компонентів:

- kube-apiserver: надає RESTful API для взаємодії з усіма іншими компонентами (він обробляє всі запити від клієнтів і забезпечує доступ до кластеру);
- kube-scheduler: відповідає за розподіл нових контейнерів по робочим вузлам, враховуючи стан ресурсів, політики та обмеження;

– kube-controller-manager: контролює стан кластера і забезпечує відповідність між бажаним станом системи та фактичним станом (він включає в себе контролери, такі як ReplicaSet Controller, Deployment Controller, Node Controller та інші);
– etcd: розподілене сховище типу «ключ-значення», яке зберігає конфігурацію та стан кластера.

Worker Nodes – елементи Kubernetes, на яких виконуються контейнери. Кожен Worker Node складається з наступних компонентів:

– kubelet: забезпечує, щоб контейнеризовані додатки були запущені відповідно до специфікацій, отриманих від API Server, і підтримує їх стан відповідно до бажаного;
– kube-proxy: управляє мережею на Worker Node, забезпечуючи функції балансування навантаження і маршрутизації трафіку до контейнерів;
– container runtime: відповідає за управління контейнерами.

Pod є основною одиницею розгортання в Kubernetes і може містити один або кілька контейнерів, які мають спільні ресурси, такі як файлові системи та мережеві інтерфейси.

Service в Kubernetes є абстракцією, яка визначає логічний набір Pods і спосіб доступу до них. Service забезпечує доступ до контейнеризованих додатків у кластері, незалежно від змін у їхньому розташуванні чи стані. Також Service здійснює балансування навантаження між Pods, які відповідають за обробку запитів. Це дозволяє рівномірно розподілити навантаження між кількома репліками додатку.

Volumes у Kubernetes – об’єкти, які реалізують зберігання даних контейнерів та існують незалежно від життєвого циклу контейнера [26]. PersistentVolume – абстракція, що представляє фізичний обсяг зберігання, який доступний для використання в кластері Kubernetes. PV є ресурсом кластеру, який визначає конкретні параметри зберігання, такі як тип, розмір і доступні опції. PV може бути прив’язаний до різних типів зберігання, таких як локальні ресурси, NFS, або хмарні рішення (AWS EBS, Google Persistent Disk, Azure

Disk). PersistentVolumeClaim – це запит на обсяг зберігання. PVC дозволяє контейнерам запитувати обсяг зберігання без знання про конкретні деталі його реалізації. PVC запитує PV, відповідний його вимогам. Kubernetes автоматично знаходить доступний PV, який відповідає вимогам PVC. StorageClass у Kubernetes – ресурс, який визначає параметри для динамічного створення PersistentVolume, дозволяє визначати різні типи зберігання без необхідності вручну створювати PersistentVolume.

StatefulSet – тип об’єкту у Kubernetes, призначений для управління stateful додатками, такими як бази даних, які потребують зберігання даних та статичних мережевих ідентифікаторів [26]. Кожен Pod в StatefulSet може мати PersistentVolumeClaim, який вказує на власний PersistentVolume. StatefulSet забезпечує стабільність ідентифікаторів та порядок створення і видалення Podів, але не синхронізує дані між ними. Для досягнення узгодженості даних можна використовувати механізми реплікації бази даних, що використовується в певному StatefulSet, налаштувавши один з Pods як мастер.

ConfigMap та Secret – ресурси в Kubernetes, які використовуються для зберігання конфігураційних даних та секретів відповідно. Вони дозволяють відокремити конфігураційні дані та забезпечують безпечне зберігання даних.

Namespaces у Kubernetes є механізмом для логічного розподілу ресурсів у кластері. Вони дозволяють створювати ізольовані середовища для різних проєктів або команд, забезпечуючи розмежування ресурсів.

Ingress у Kubernetes – ресурс, який керує вхідним трафіком до кластера. Він забезпечує можливість маршрутизації трафіку до різних сервісів на основі доменних імен або URL-шляхів. Ingress Controller – контролер, який реалізує правила Ingress і забезпечує маршрутизацію трафіку. Це може бути один з таких контролерів, як NGINX Ingress Controller, Traefik або HAProxy.

YAML файли є основним інструментом для опису та конфігурації ресурсів у кластері. Вони використовуються для визначення характеристик і специфікацій таких компонентів, як Pods, Services, Deployments і ConfigMaps.

kubectl – інструмент командного рядка, який дозволяє взаємодіяти з кластером Kubernetes [26]. За допомогою нього можна застосувати YAML файли з конфігурацією ресурсів.

Helm – менеджер пакетів для Kubernetes, який спрощує процес розгортання, оновлення і управління додатками в кластері Kubernetes [26]. Helm Charts є наборами YAML-файлів, які описують ресурси Kubernetes, необхідні для розгортання додатку. Вони включають шаблони ресурсів, файли конфігурацій і залежності. Helm Repositories – сховища для зберігання і розподілу Charts.

Kubernetes надає можливості для моніторингу та логування кластера та його контейнерів через інтеграцію з такими інструментами, як: інструмент для збору та зберігання метрик Prometheus, інструмент для візуалізації даних Grafana і інструмент для пошуку і аналітики даних Elasticsearch.

1.5.3 Огляд хмарного сервісу DigitalOcean

DigitalOcean – платформа хмарних обчислень, що пропонує користувачам віртуальні сервери, сховища даних та мережеві ресурси для створення і розміщення вебдодатків. Основні послуги сервісу:

- Droplets – віртуальні сервери, які мають можливість гнучкого налаштування та масштабування;
- Volumes – додаткові сховища даних для Droplets, які дозволяють масштабувати дисковий простір та відокремлювати зберігання даних;
- Load Balancers реалізують балансування навантаження для розподілу трафіку між кількома Droplets, що допомагає підвищити надійність та масштабованість додатків;
- Container Registry – приватний реєстр, який дозволяє зберігати репозиторії Docker-образів;

- Kubernetes – сервіс для управління контейнеризованими додатками, сумісний зі стандартними інструментами Kubernetes, що дозволяє легко мігрувати існуючі кластери [27];
- Functions – безсерверна обчислювальна платформа, що надає можливість виконання коду без необхідності управління інфраструктурою;
- Databases – сервіс для управління базами даних, включаючи PostgreSQL, MySQL, MongoDB, Redis, OpenSearch (сервіс пропонує автоматизацію резервного копіювання, оновлень та масштабування);
- Spaces – S3-подібне хмарне сховище об’єктів, яке можна використовувати для зберігання великих обсягів даних.

DigitalOcean має інтерфейс командного рядка doctl, який надає можливість взаємодії з API DigitalOcean для створення і управління переліченими сервісами.

DigitalOcean є зручною платформою для управління хмарною інфраструктурою. Ряд розглянутих сервісів надає можливість для розгортання та управління інфраструктурою різноманітних додатків, включаючи додатки з мікросервісною архітектурою.

1.5.4 Огляд CI/CD-інструменту GitHub Actions

GitHub Actions – інструмент для автоматизації процесів розробки, який дозволяє створювати, налаштовувати і автоматизувати різноманітні процеси безпосередньо в репозиторіях GitHub [29].

Сервіс GitHub Actions дозволяє реалізувати CI/CD (безперервна інтеграція / безперервне розгортання) – набір практик, які автоматизують процес розробки, тестування та доставки програмного забезпечення. Цей підхід дозволяє прискорити доставку нових версій програмного продукту та знизити ризики, пов’язані з ручним втручанням в роботу інфраструктури.

Безперервна інтеграція може включати такі етапи, як: виконання тестів, аналіз кодової бази інструментами для перевірки дотримання стандартів та стилістичних рекомендацій, пошук вразливостей у залежностях, збірка коду та доставка нової збірки проєкту на віддалене сховище.

Безперервне розгортання автоматизує процес розгортання програмного забезпечення на різних середовищах.

CI/CD-процеси разом забезпечують безперервний цикл інтеграції і розгортання змін.

Робочі процеси GitHub Actions створюються і налаштовуються за допомогою файлів конфігурації у форматі YAML, які розміщені у директорії .github/workflows репозиторію [29]. Кожен процес (workflow) складається з списку завдань (jobs), які визначають окремі етапи автоматизації. GitHub Marketplace пропонує велику кількість рішень, які можна використовувати для автоматизації різних завдань, таких як: статичний аналіз коду, збірка артефактів та розгортання.

Ці можливості роблять GitHub Actions зручним інструментом для автоматизації розробки, інтеграції та розгортання, що дозволяє здійснювати управління автоматизованими процесами безпосередньо в середовищі GitHub. Розглянута платформа надає зручний і ефективний спосіб автоматизації численних аспектів розробки програмного забезпечення.

1.6 Висновки з розділу 1

У першому розділі наведено характеристику предметної області, проведено аналіз освітніх платформ, зокрема Prometheus та Udemy, визначено їхню функціональність, особливості та переваги.

Проведено аналіз мікросервісної архітектури, визначено її переваги та недоліки. Розглянуто типи та засоби комунікації між мікросервісами. Описано основні патерни та практики для реалізації мікросервісної архітектури.

Розглянуто застосування принципів мікросервісної архітектури при побудові клієнтських інтерфейсів.

Наведено огляд засобів розробки, зокрема СУБД PostgreSQL, платформи Node.js, фреймфорку NestJS та фреймворку Angular з їхніми екосистемами. Також описано такі засоби розгортання та управління інфраструктурою, як інструмент контейнеризації Docker, інструмент оркестрації Kubernetes, хмарний сервіс DigitalOcean та CI/CD-інструмент GitHub Actions.

У підсумку, аналіз предметної області та об’єкту дослідження надав підґрунтя для подальшого проєктування та розробки освітньої платформи з використанням мікросервісної архітектури та розглянутих засобів розробки.

2 ПРОЄКТУВАННЯ ОСВІТНЬОЇ ПЛАТФОРМИ

Перед початком створення програмної реалізації освітньої платформи необхідно визначити призначення розробки, зібрати функціональні та технічні вимоги до програмного продукту, спроєктувати користувацький інтерфейс та виконати побудову архітектури.

2.1 Призначення розробки

Розробка освітньої платформи має на меті створення інтерактивного та динамічного середовища для навчання, яке забезпечує користувачам доступ до навчальних матеріалів.

Основні завдання розробки включають в себе реалізацію автентифікації, функціоналу пошуку курсів, реєстрації на обраний курс та оплати, функціоналу зручного доступу та навігації навчальними матеріалами курсу та моніторинг їхнього опрацювання, реалізацію функціоналу створення нових курсів.

Загалом, розробка освітньої платформи спрямована на створення комплексного середовища для отримання та розміщення структурованих навчальних матеріалів.

2.2 Вимоги до програмного продукту

2.2.1 Функціональні вимоги

Програмний продукт, що розробляється, повинен відповідати наступним функціональним вимогам:

- платформа має надавати можливість реєстрації та автентифікації користувачів;
- користувач повинен мати можливість переглядати наявні на платформі курси та здійснювати пошук курсів за такими параметрами, як: назва курсу та його тематика;
- користувач повинен мати можливість переглядати списки своїх розпочатих, завершених та запланованих курсів;
- користувач повинен мати можливість зареєструватися на курс, оплативши його, якщо вибраний курс є платним;
- користувач повинен мати можливість переглядати навчальні матеріали курсу, на який він є зареєстрованим, та мати можливість здійснювати навігацію курсом;
- кожен курс має поділятися на модулі;
- модулі курсу мають складатися з уроків та тестувань;
- урок може бути у форматі відео або в текстовому форматі;
- уроки, опрацьовані користувачем, мають мати можливість бути виділеними як завершені;
- користувач має мати можливість відстежувати відсоток пройденого матеріалу курсу від його повної програми;
- після завершення курсу користувач повинен мати можливість згенерувати персоналізований сертифікат;
- у неавтентифікованих та автентифікованих користувачів має бути можливість перевірки валідності сертифікату;
- користувач з роллю автора повинен мати можливість створити новий курс, заповнити основну інформації курсу, додавати модулі, уроки та тестування;
- користувач має отримувати листи на електронну пошту при активації облікового запису та при оплаті курсу.

2.2.2 Технічні вимоги

Програмний продукт, що розробляється, повинен відповідати наступним технічним вимогам:

- серверна частина програми має бути реалізована на основі мікросервісної архітектури;
- мікросервіси мають взаємодіяти між собою за допомогою протоколу gRPC або через брокер повідомлень, такий як RabbitMQ;
- серверна частина має надавати єдину точку входу для клієнта, що має бути реалізована як REST API;
- клієнтський інтерфейс має бути реалізований як SPA;
- клієнтський інтерфейс має бути реалізований з використанням мікрофронтендів;
- система автентифікації та авторизації повинна бути реалізована з використанням JWT токенів, що містять інформацію про ролі користувача;
- на основі ролей та доступів користувача мають бути обмежені відповідні частини API мікросервісів та частини клієнтського інтерфейсу;
- має бути виконана контейнеризація та оркестрація контейнерів;
- кожен мікросервіс та мікрофронтенд повинен мати можливість бути розгорнутим незалежно;
- система має бути розгорнута в хмарному середовищі;
- має бути впроваджено CI/CD.

2.3 Діаграма використання

Діаграма прецедентів – один із типів UML-діаграм, яка відображає функціональність системи та її взаємодію з користувачами (акторами) через визначені функції (прецеденти).

Для проєкту освітньої платформи була розроблена діаграма прецедентів, зображена на рисунку 2.1. На діаграмі представлено три актори: користувач, автор та адміністратор. Кожен з цих акторів взаємодіє з системою, виконуючи визначені для нього дії.



Рисунок 2.1 – Діаграма використання

Користувач освітньої платформи має можливість виконувати основні функції системи: реєструватися, активувати обліковий запис, переглядати курси, реєструватися на курси, переглядати навчальні матеріали курсів та залишати відгуки до курсів, на які він є зареєстрованим, зберігати стан завершення уроків, додавати нотатки до уроків.

Автор має можливість виконувати всі прецеденти, визначені для користувача, а також створювати та редагувати курси.

Адміністратор має можливість перегляду зареєстрованих у системі користувачів, видалення користувачів, курсів та відгуків про курси та надання ролі автора користувачам.

Створена діаграма використання допомагає визначити та структурувати функції, які має надавати система, з точки зору взаємодії з користувачами, що мають різні ролі.

2.4 Вимоги до користувацького інтерфейсу

На основі функціональних вимог необхідно визначити вимоги до користувацького інтерфейсу освітньої платформи.

Серед основних частин інтерфейсу можна виділити:

- інтерфейс аутентифікації: надає користувачам спосіб входу та реєстрації на платформі, включаючи можливості відновлення пароля;
- інтерфейс профілю користувача: дозволяє користувачам переглядати та керувати своєю особистою інформацією, записами на курси, прогресом та налаштуваннями;
- інтерфейс огляду курсів: відображає список доступних курсів з ключовими даними, такими як назви курсів, описи, викладачі та відгуки, дозволяючи користувачам переглядати та обирати курси;
- інтерфейс окремого курсу: показує детальну інформацію про вибраний курс, включаючи уроки, матеріали, відстеження прогресу, генерування сертифікату про завершення курсу та доступ до відгуків до курсу та коментарів до уроків, надає можливість оплатити та розпочати курс;
- інтерфейс створення та редагування курсів: дозволяє авторам курсів створювати нові курси, додавати та налаштовувати модулі й уроки, а також додавати тести для перевірки знань (інтерфейс включає функції для редагування описів, структури курсу, матеріалів, тестових завдань та інших навчальних компонентів, що забезпечує гнучкість і зручність у розробці навчального контенту).

Визначені вимоги до користувацького інтерфейсу є основою для подальшої програмної реалізації, оскільки вони надають конкретні настанови щодо представлення програмного продукту, ґрунтуючись на функціональності, що має бути реалізована.

2.5 Побудова мікросервісної архітектури

Для побудови системи обрано мікросервісну архітектуру. Діаграму побудованої мікросервісної архітектури зображено на рисунку 2.2.

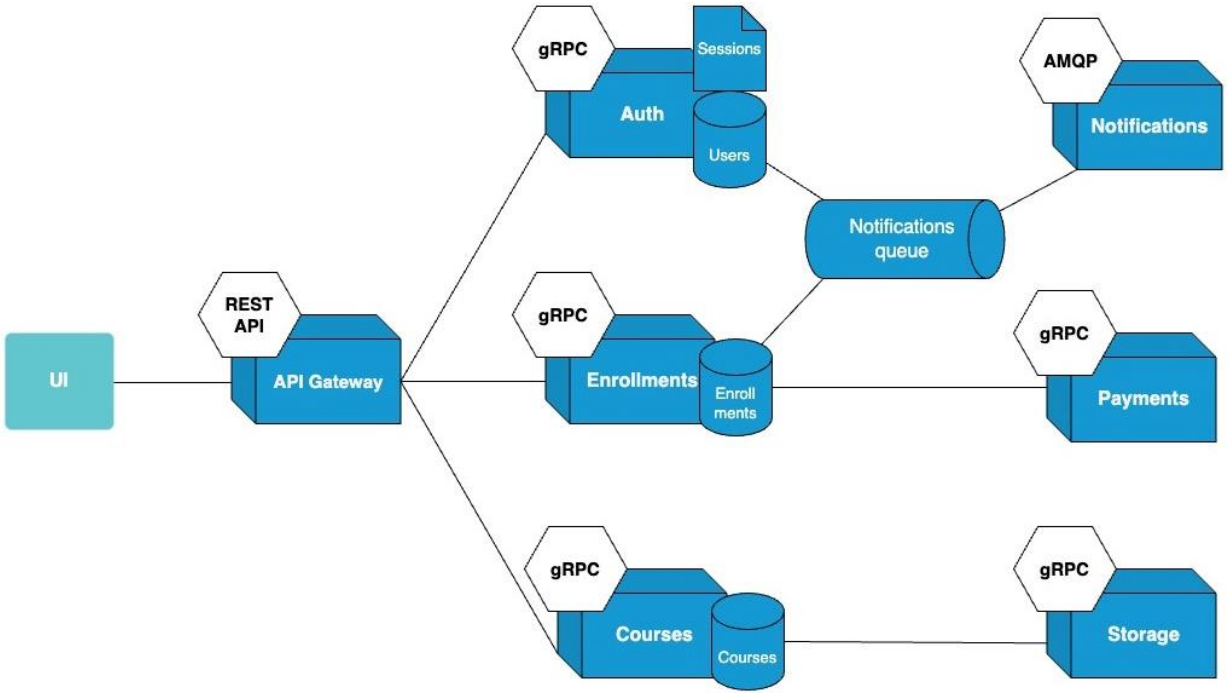


Рисунок 2.2 – Діаграма мікросервісної архітектури

Серверна частина системи поділена на наступні мікросервіси:

– сервіс Auth – зберігає дані облікових записів користувачів та генерує токени;
– сервіс Courses – зберігає загальну інформацію про курси та їхні навчальні матеріали;
– сервіс Enrollments – зберігає стан взаємодії користувача з курсами;

– сервіс Storage – зберігає файли на зовнішньому сховищі;
– сервіс Payments – здійснює платіжні операції;
– сервіс Notifications – надсилає повідомлення користувачу;
– сервіс API Gateway – єдина точка входу для клієнта.

Сервіси Auth, Courses та Enrollments мають окремі реляційні бази даних. Сервіс Auth також використовує in-memory сховище даних.

Сервіси Storage та Payments реалізують інтеграції з сторонніми сервісами. Сервіс Storage виконує інтеграцію з S3-сумісним сховищем. Сервіс Payments виконує інтеграцію з платіжним шлюзом.

Більшість сервісів взаємодіють між собою, використовуючи систему віддаленого виклику процедур gRPC Взаємодія з сервісом Notifications відбувається за протоколом AMQP з використанням брокера повідомлень.

Сервіс API Gateway є єдиною точкою входу для клієнтських запитів, сервіс надає REST API інтерфейс.

2.6 Проєктування баз даних

Мікросервіси Auth, Courses і Enrollments використовують окремі реляційні бази даних. Система реалізує патерн для побудови розподілених систем «Database per service», який передбачає використання окремих баз даних для кожного мікросервісу. Кожен сервіс має власну базу даних, що містить лише ті дані, які необхідні для виконання його функцій.

2.6.1 Проєктування бази даних сервісу Auth

База даних сервісу Auth зберігає дані облікових записів користувачів. ER-діаграма бази даних наведена на рисунку 2.3. База даних сервісу складається з 4 таблиць.

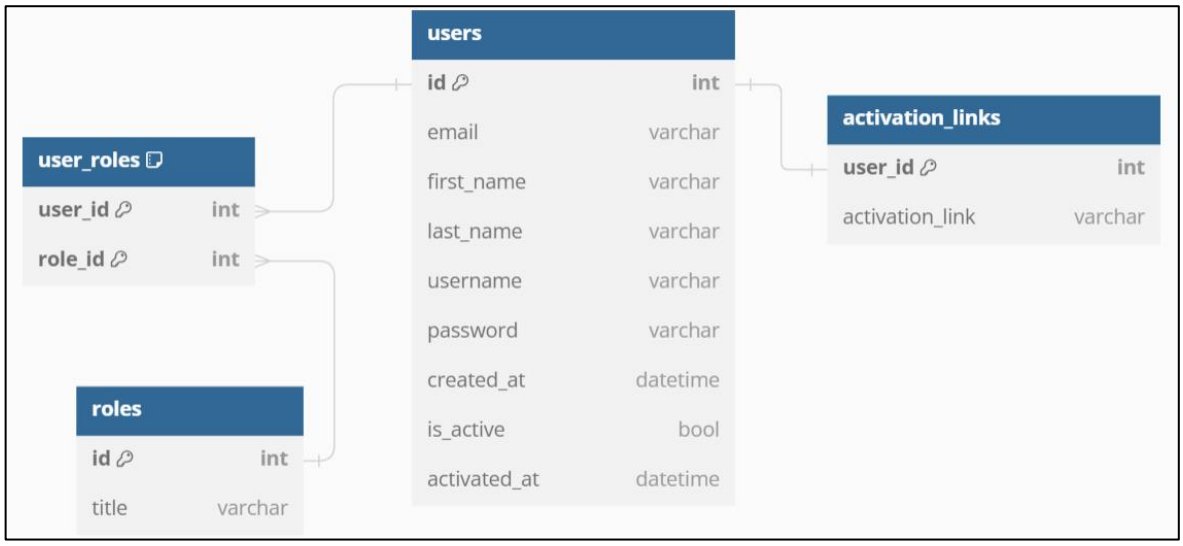


Рисунок 2.3 – ER-діаграма бази даних сервісу Auth

Таблиця «users» зберігає дані облікових даних користувачів. Таблиця має поле з унікальними значенням «email» та «username». Дані для активації облікового запису користувача винесені в окрему таблицю «activations_links», яка має один-до-одного відношення до таблиці «users».

Таблиця «roles» зберігає ролі, які можуть бути призначені користувачам. Для реалізації відношення багато-до-багатьох між користувачами та ролями виокремлено допоміжну таблицю «user_roles».

2.6.2 Проєктування бази даних сервісу Courses

База даних сервісу Courses зберігає загальну інформацію про курси та їхні навчальні матеріали. ER-діаграма бази даних наведена на рисунку 2.4. База даних сервісу складається з 13 таблиць.

Таблиця «courses» зберігає загальну інформацію про навчальні курси: назву, опис, автора, дату публікації та вартість.

Для класифікації курсів додані таблиці «categories» та «tags». Курс належить до однієї з категорій та може мати декілька тегів. Для реалізації відношення багато-до-багатьох між курсами та тегами курсів додано допоміжну таблицю «course_tags».

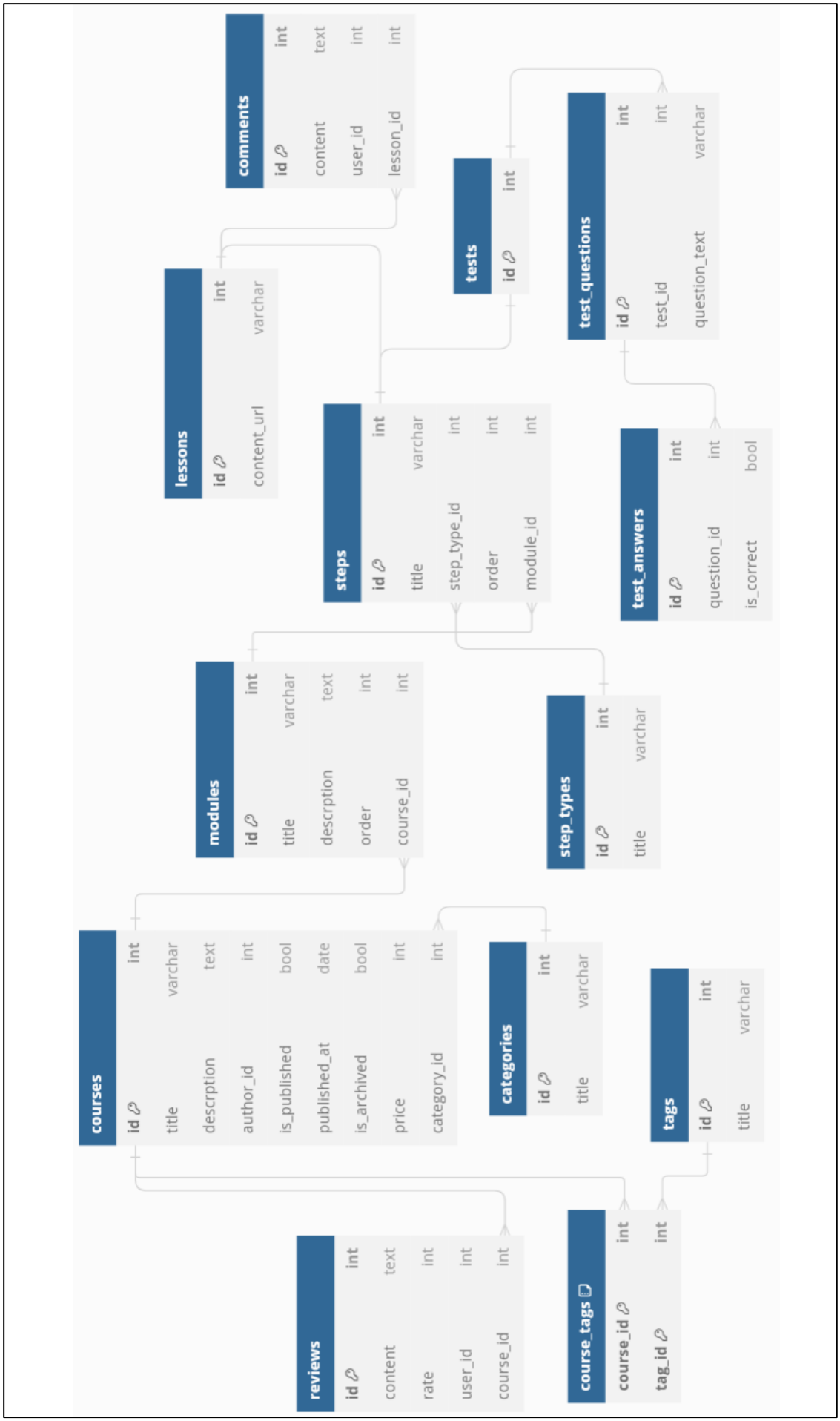


Рисунок 2.4 – ER-діаграма бази даних сервісу Courses

Таблиця «reviews» містить дані про загальнодоступні відгуки користувача до певного курсу. Відгук складається з оцінки курсу та текстової частини.

Курси поділяються на модулі, дані яких зберігаються у таблиці «modules». Таблиця містить поле «order» для зберігання порядку модулів курсу.

В свою чергу, модулі поділяються на кроки, дані яких зберігаються у таблиці «steps». Записи мають поле «order» для зберігання порядку уроків модуля.

Кожен крок модулю характеризується типом, посилаючись на запис таблиці «step_types». Кроком модуля може бути урок або тестування. Для їх зберігання відповідно створено таблиці «lessons» та «tests». Ці таблиці мають один-до-одного відношення до таблиці з загальними даними кроків модуля. Різні типи кроків мають різний тип зберігання та, як наслідок, потребують різних полів у таблиці. Додавання окремої таблиці для кожного типу кроку допомагає уникнути надлишкового зберігання null-значень у полях, що не використовуються певним типом кроку.

Для уроків передбачено поле «content_url», яке зберігає посилання на відео- або текстовий матеріал. Таблиця «comments» містить дані про загальнодоступні коментарі користувача до певного уроку.

Для зберігання тестувань додано таблиці «test_questions» та «test_answers».

2.6.3 Проєктування бази даних сервісу Enrollments

База даних сервісу Enrollments зберігає дані про стан взаємодії користувача з курсами. ER-діаграма бази даних наведена на рисунку 2.5. База даних сервісу складається з 4 таблиць.

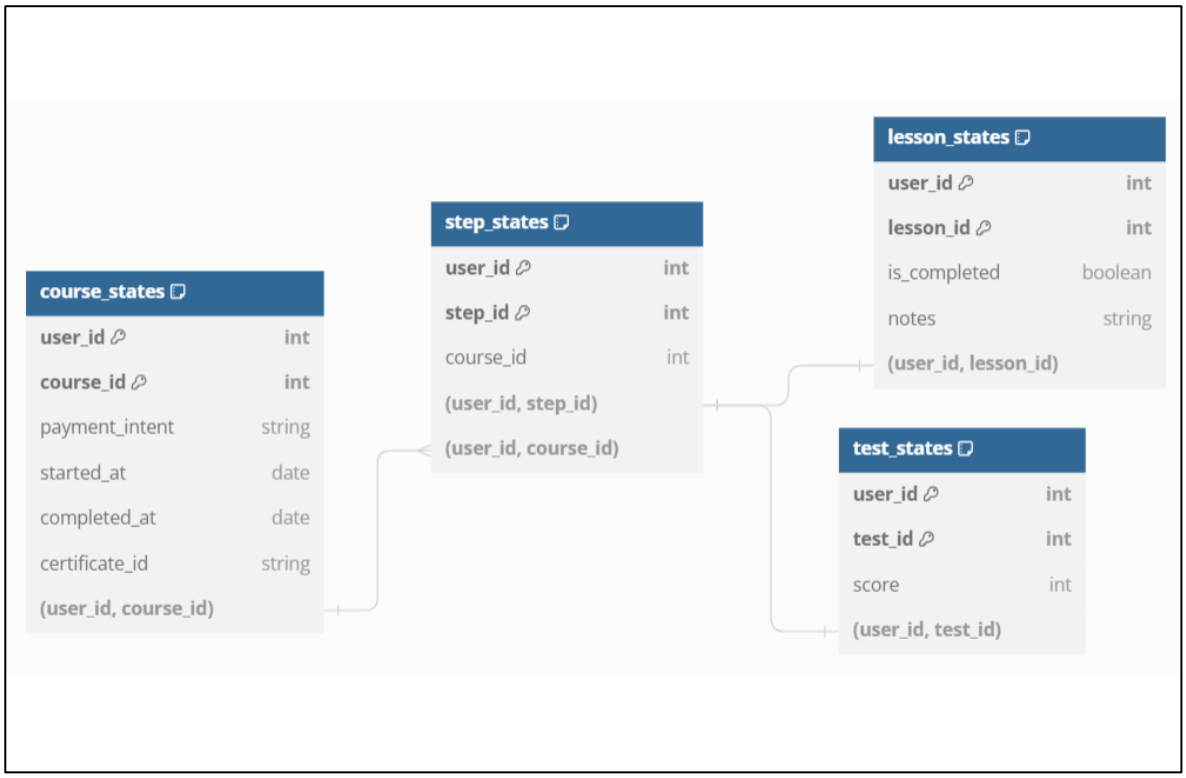


Рисунок 2.5 – ER-діаграма бази даних сервісу Enrollments

Таблиця «course_states» зберігає дані взаємодії користувача з курсом, а саме: дату реєстрації на курс, завершення курсу, дані про оплату, ідентифікатор сертифікату завершення курсу. Таблиця має складений первинний ключ, який складається з полів «user_id» та «courses_id».

Таблиця «step_states» містить дані взаємодії користувача з окремим кроком, яким може бути урок або тестування. Таблиця має складений первинний ключ, який складається з полів «user_id» та «step_id». Поля «user_id» та «course_id» утворюють зовнішній ключ для один-до-багатьох відношення з таблицею «course_states».

Виокремлені таблиці для зберігання взаємодій з двома типами кроків, які мають один-до-одного відношення до таблиці «step_states». Таблиця «lesson_states» зберігає дані про завершення певного уроку курсу та власні нотатки користувача до кожного окремого уроку. Таблиця «test_states» зберігає дані про проходження певного тестування.

2.7 Проєктування додатку користувацького інтерфейсу із застосуванням принципів мікросервісної архітектури

Для побудови додатку користувацького інтерфейсу вирішено використовувати архітектурний підхід Microfrontends.

Виокремлюючи функціональні частини системи та беручи до уваги вже спроєктовану мікросервісну архітектуру серверної частини, проєкт користувацького інтерфейсу поділено на наступні мікрофронтенди:

- додаток Host – основний додаток, який використовується як контейнер для інтеграції всіх інших додатків;
- додаток Auth – додаток, що відповідає за інтерфейс автентифікації та авторизації користувача;
- додаток Courses – містить функціональність, що забезпечує роботу автора курсу з загальною інформацією курсу та його навчальними матеріалами;
- додаток Enrollments – відповідальний за функціонал, що стосується роботи з курсом у ролі слухача курсу;
- додаток Admin – містить в собі функціональні можливості, що доступні тільки для користувачів, що мають серед ролей роль адміністратора.

Спільні компоненти, сервіси та інші сутності мікрофронтендів винесені у окрему бібліотеку. Головною метою створення бібліотеки є забезпечення повторного використання UI-елементів та логіки додатків. Створена бібліотека може бути розміщена на npm, що дозволить мікрофронтендам використовувати її як залежність.

Діаграму перелічених елементів додатку користувацького інтерфейсу зображено на рисунку 2.6.

Для впровадження підходу до розробки з використанням мікрофронтендів існує два методи структурування кодової бази проєкту: використання монорепозиторію або розподілених репозиторіїв. При роботі у розподілених репозиторіях виникає проблема з управлінням спільних

елементів проєктів, яка може бути вирішена створенням окремої бібліотеки. Як недолік, через окремо визначені залежності кожного мікрофронтенду, можуть виникнути проблеми з узгодженням версій бібліотек та синхронізацією їхніх оновлень. У розподілених репозиторіях кожен мікрофронтенд потребує визначення власного CI/CD, щоб мати можливість бути розгорнутим незалежно. При роботі з монорепозиторієм відсутня проблема з необхідністю управління спільними елементами проєктів та необхідність синхронізації залежностей проєктів. При використанні одного репозиторію для всіх мікрофронтендів виникає проблема з виконанням CI/CD для мікрофронтендів, які не мали змін, для вирішення якої необхідно застосувати інструменти для виявлення частин репозиторію, які були змінені. При роботі в монорепозиторії також існує ймовірність частих конфліктів, що може бути проблемою для команд, що працюють над окремими мікрофронтендами. Для впровадження обрано підхід з розподіленими репозиторіями.

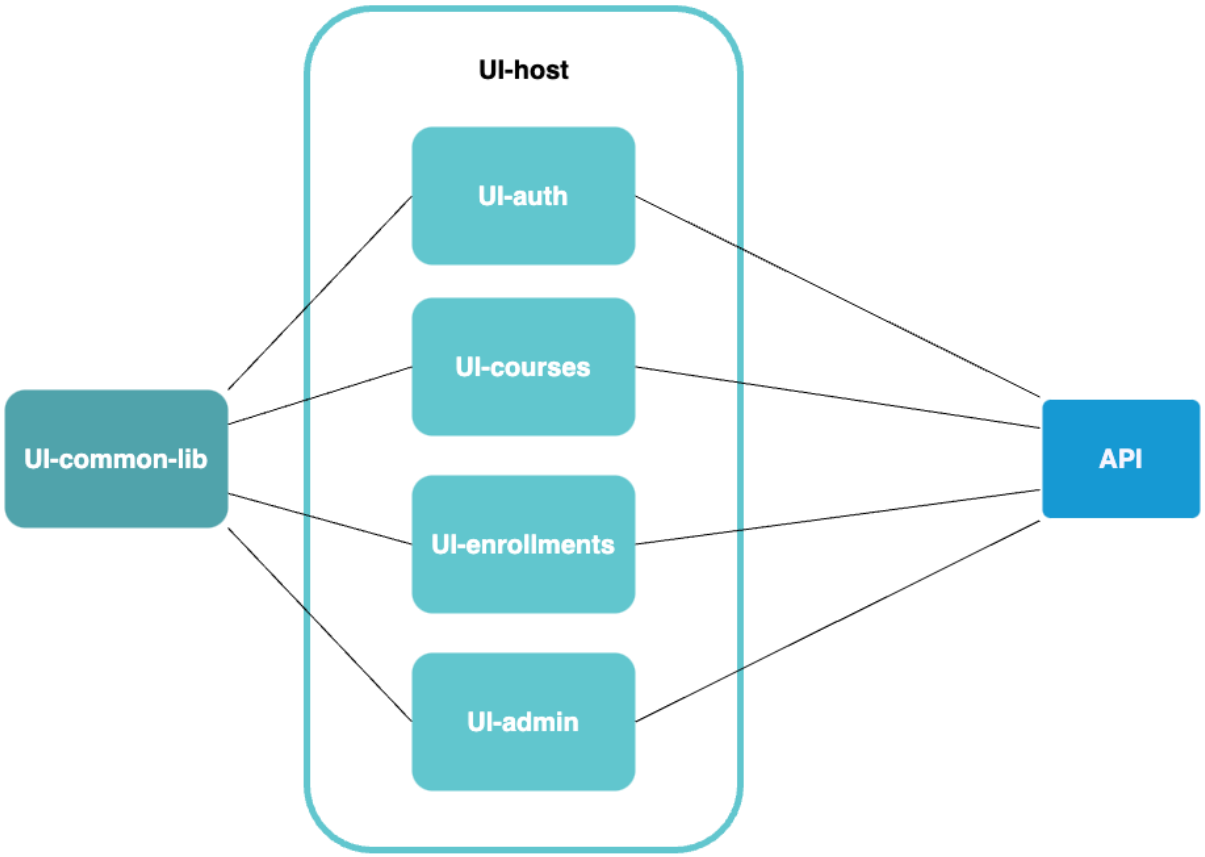


Рисунок 2.6 – Діаграма додатку користувацького інтерфейсу

Як спосіб реалізації мікрофронтендів обрано інструмент Module Federation – функціональність Webpack, що дозволяє динамічно завантажувати та використовувати модулі JavaScript з різних проєктів у рамках одного додатку.

Для синхронізації стану мікрофронтендів вирішено використовувати метод, який полягає у визначенні кастомних DOM-подій. Зокрема, синхронізація стану потрібна при роботі з автентифікацією та авторизацією: мікрофронтенд Host виконує інтеграцію з серверною частиною системи для автентифікації та авторизації, передаючи результат інтеграції іншим мікрофронтендам.

2.8 Висновки з розділу 2

У другому розділі визначено призначення розробки, сформовано функціональні та технічні вимоги до програмного продукту. Наведено діаграму використання. Виконано проєктування користувацького інтерфейсу.

Виконано побудову архітектури програмного продукту. Визначені мікросервіси системи, їхні функції та способи їхньої взаємодії. Спроєктовані бази даних сервісів. Використовуючи принципи мікросервісної архітектури, визначена структура додатку користувацького інтерфейсу та обраний метод її реалізації.

Виконаний етап проєктування системи освітньої платформи дає можливість почати розробку її програмної реалізації.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Розробка серверної частини

3.1.1 Вибір засобів розробки

Сервіси системи реалізовано з використанням платформи Node.js, мови TypeScript та фреймворку NestJS.

Для використання реляційних баз даних сервісами Auth, Courses та Enrollments обрано реляційну систему управління базами даних PostgreSQL та ORM Prisma. Як in-memory сховище для використання сервісом Auth обрано сховище типу «ключ-значення» Redis.

Для взаємодії з сервісом Notifications як брокер повідомлень, який використовує протокол AMQP, обрано RabbitMQ.

Для сервісу Payments у якості зовнішнього сервісу для обробки платежів обраний платіжний шлюз Stripe. Як S3-сумісне сховище для використання сервісом Storage обрано DigitalOcean Spaces.

3.1.2 Сервіс Auth

Сервіс Auth відповідає за аутентифікацію і авторизацію користувачів в системі. Він забезпечує перевірку ідентичності користувачів під час входу до додатку, а також визначає їхній доступ до ресурсів і функціоналу системи відповідно до збережених прав і ролей. Основними функціями сервісу є зберігання облікових записів користувачів, генерація пари токенів доступу та оновлення, зберігання виданих токенів оновлення.

Сервіс реалізує можливість виклику віддаленого методу реєстрації користувача, який очікує такі параметри, як електронна пошта та пароль. Проводиться перевірка унікальності електронної пошти, хешування пароля

зовнішнім пакетом bcrypt та генерування посилання для активації облікового запису.

Сервіс використовує СУБД PostgreSQL для зберігання і керування обліковими записами користувачів.

За замовчуванням обліковий запис користувача не є активованим. Сервіс Auth використовує інтерфейс сервісу Notifications для відправки електронного листа з попередньо згенерованим посиланням, при використанні якого змінюється відповідний флаг у записі таблиці облікових даних користувача.

Реалізовано метод авторизації користувача, який повертає пару згенерованих токенів доступу та оновлення. Access-токен використовується для авторизації запитів і для зберігання додаткової інформації про користувача, зокрема призначених йому ролей. Refresh-токен використовується для отримання нової пари токенів доступу та оновлення. Для access-токена задано термін дії 30 хвилин, для refresh-токена – 60 днів.

Токен доступу є JWT (JSON Web Token) токеном. JWT – стандарт для передачі даних у вигляді JSON об’єктів, які зашифровані за допомогою симетричного або асиметричного ключа. JWT складається з трьох частин: заголовку (вказує на тип токена і тип підпису, який використовується для захисту токена), корисного навантаження (заданий JSON із застосуванням стандарту кодування Base64) і підпис (створений за допомогою шифрування заголовку, корисного навантаження та заданого секрету вказаним методом).

При аутентифікації користувача у системі згенерований refresh-токен та визначений fingerprint клієнта зберігається у сховищі Redis, створюючи refresh-сесію. Refresh-токени зберігаються для контролю доступу і інвалідації викрадених токенів. Для можливості аутентифікації на кількох пристроях одночасно необхідно зберігати всі refresh-токени для кожного користувача. Кожен користувач може мати до 5 refresh-сесій одночасно.

В методі оновлення access-токена за переданим refresh-токеном виконується пошук відповідної йому refresh-сесії, перевірка терміну дії переданого refresh-токена та порівнюються поточний та збережений fingerprint клієнта. Метод оновлює обидва токени.

При запитах аутентифікації та оновлення access-токена сервіс визначає fingerprint клієнта на основі заголовку запиту User-Agent, використовуючи зовнішній пакет device-detector-js, який призначений для виявлення пристроїв, платформ, браузерів і їхніх версій.

Метод зміни паролю використовує інтерфейс сервісу Notifications для відправки електронного листа з попередньо згенерованим посиланням. Також даний метод видаляє всі refresh-сесії користувача.

Метод виходу з системи очікує refresh-токен та видаляє відповідну йому refresh-сесію користувача. Access-токен не зберігається сервером, тому має можливість бути видаленим тільки на клієнтській частині, залишаючись валідним до спливу терміну його дії.

Авторизація була реалізована використовуючи модель контролю доступу на основі ролей (Role-Based Access Control, RBAC). Ролі складаються з списку дозволів, які визначають, які дії може виконувати користувач. Користувачу може бути призначена одна або декілька ролей. Права доступу користувача визначаються на основі дозволів, що пов’язані з заданими йому ролями. RBAC групує користувачів за визначеним списком ролей та призначає дозволи на рівні ролей замість призначення дозволів кожному користувачеві.

Інші сервіси системи при викликах їхніх методів отримують згенерований сервісом Auth access-токен, самостійно валідують його підпис та отримують з його корисного навантаження дані при користувача, включаючи список його ролей.

3.1.3 Сервіс Courses

Сервіс Courses відповідає за зберігання загальної інформації про курси та їхні навчальні матеріали.

Основна сутність, якою оперує сервіс, є сутність навчального курсу. Курс поділяється на модулі. В свою чергу, модулі курсу поділяються на кроки,

які можуть бути уроками або тестуваннями. Для класифікації курсів додані сутності категорій та тегів. Курс належить до однієї з визначених категорій та може мати декілька тегів. Реалізовано функціонал пошуку курсів за категоріями, тегами, авторами та назвами.

Для всіх автентифікованих у систему користувачів доступні методи отримання загальної інформації про курс. Користувач, зареєстрований на певний курс, може отримати навчальні матеріали курсу, проходити тестування, а також додавати, редагувати та видаляти власні відгуки про курс та коментарі до уроків.

Для користувачів з роллю автора доступний метод додавання нового курсу. Для користувача, що є автором певного курсу, доступні методи додавання, редагування та видалення модулів та кроків цього курсу.

Для користувачів, які мають роль адміністратора, доступний функціонал додавання категорій та тегів курсів. Також адміністратори можуть видаляти будь-які курси, відгуки про курси та коментарі уроків.

Сервіс Courses звертається до сервісу Storage для завантажень навчальних матеріалів. Інтерфейс сервісу Courses використовує безпосередньо сервіс API Gateway.

3.1.4 Сервіс Enrollments

Сервіс Enrollments відповідає за взаємодію користувача з курсами.

Сервіс зберігає дані взаємодії користувача з певним навчальним курсом. Зберігається дата реєстрації користувача на курс, дата його завершення, дані про оплату, дані про виданий сертифікат завершення курсу.

Також зберігається дані про взаємодію користувача з певним уроком або тестуванням. Зберігається статус завершення уроків курсу, нотатки користувача до окремих уроків та бал тестування.

Для реалізації функції генерування сертифікату завершення курсу використовується бібліотека jspdf, яка дозволяє створювати PDF-документи.

Згенерований сертифікат містить попередньо створений ідентифікатор. Сервіс надає можливість провалідувати сертифікат за наданим ідентифікатором.

Сервіс Enrollments використовує інтерфейс сервісу Payments для оплати певного курсу та інтерфейс сервісу Notifications при отримані доступу до курсу. До сервісу Enrollments звертається безпосередньо сервіс API Gateway.

3.1.5 Сервіс Storage

Сервіс Storage відповідає за зберігання файлів на зовнішньому сховищі.

Сервіс виконує інтеграцію з S3-сумісним сховищем DigitalOcean Spaces. DigitalOcean Spaces сумісний з API сервісу Amazon S3, тому для керуванням створеного Space може бути використаний AWS SDK для Node.js.

Сервіс надає процедури віддаленого виклику для завантаження та видалення файлів.

Предметна область розробки потребує реалізації функціональності завантаження файлів великого розміру. При роботі з такими завантаженнями виникають проблеми з масштабованістю (оскільки традиційні серверні рішення можуть мати труднощі з одночасними завантаженнями великих файлів), надійністю (через вірогідність мережевих проблем при передачі файлів) та використанням пам’яті (оскільки зберігання великих файлів швидко вичерпує ємність пам’яті).

S3-сумісні сховища надають можливість використовувати прямі завантаження з додатку користувацького інтерфейсу. Сервіс Storage використовує AWS SDK для генерування попередньо підписаного та обмеженого в часі посилання, яке отримує клієнтський додаток. Клієнтський додаток використовує отримане посилання для безпосереднього завантаження файлу в S3- сумісне сховище.

У розробленій системі сервіс Storage використовується сервісом Courses при редагуванні матеріалів курсів.

3.1.6 Сервіс Payments

Сервіс Payments відповідає за здійснення платіжних операцій. Він забезпечує обробку і керування фінансовими транзакціями, включаючи прийом та обробку платежів.

Сервіс Payments використовує Stripe для обробки платіжних транзакцій. Stripe є платіжним шлюзом, який забезпечує просту і безпечну інтеграцію з платіжними системами, дозволяючи приймати, обробляти і керувати платежами через різні платіжні методи. Використання Stripe включає авторизацію і підтвердження платежів, обробку кредитних і дебетових карток, а також підтримку міжнародних платежів і додаткових функцій, таких як повторні платежі та підписки.

Сервіс надає gRPC виклик для проведення платіжної операції, який очікує як параметри дані про користувача, дані банківської картки, суму та валюту платежу.

Сервіс використовує зовнішній пакет stripe, який надає API для взаємодії з платіжним шлюзом Stripe. Проєкт було налаштовано з використанням ключа, який був згенерований у панелі керування Stripe для створеного облікового запису.

У розробленій системі сервіс Payments приймає запити на проведення платежів від сервісу Enrollments для оплати користувачем обраного курсу.

3.1.7 Сервіс Notifications

Сервіс Notifications відповідає за надсилання повідомлень. Реалізований метод надсилання повідомлень – повідомлення листом на електронну пошту.

Взаємодія з сервісом Notifications відбувається через брокер повідомлень RabbitMQ. Інші мікросервіси системи надсилають повідомлення до черги, що відповідає за певний тип нотифікацій. Сервіс Notifications читає повідомлення з черг RabbitMQ та обробляє їх.

Корисне навантаження повідомлення з черги «email_queue» містить у собі електронну пошту отримувача, тему та зміст листа.

Для обробки повідомлень з черги «email_queue» сервіс використовує зовнішній пакет nodemailer, який побудований на основі протоколу SMTP та надає API для надсилання електронних листів. Пакет nodemailer з використовується з сервісом Gmail, доступ до API якого отримано за допомогою протокола авторизації OAuth 2.0.

У розробленій системі інтерфейс сервісу Notifications використовується сервісом Auth при активації облікового запису та сервісом Enrollments при отримані доступу до курсу.

3.1.8 Сервіс API Gateway

Сервіс API Gateway є єдиною точкою входу для клієнтських запитів.

Одна з основних функцій даного сервісу полягає в тому, щоб приховати gRPC-взаємодію від користувацького інтерфейсу та надати RESTful інтерфейс, адже gRPC не може бути безпосередньо використано браузером. Основні рішення цієї проблеми – API Gateway – сервіс, який реалізує REST API на основі gRPC-сервісів, та gRPC-Web – інструмент, який дозволяє здійснювати виклики gRPC з браузера, використовуючи HTTP/1.1.

Сервіс API Gateway реалізує RESTful API для використання додатком користувацького інтерфейсу. Сервіс напряму взаємодіє з трьома основними сервісами системи – сервісами Auth, Courses та Enrollments, які реалізовані на основі gRPC.

Інтерфейс сервісу описаний у форматі специфікації OpenAPI. Фреймворк NestJS надає @nestjs/swagger пакет, який дозволяє генерувати OpenAPI специфікацію, використовуючи декоратори сутностей проєкту. Використовуючи інструмент Swagger, згенеровано документацію сервісу.

3.2 Розробка користувацького інтерфейсу

На етапі проєктування додатку користувацького інтерфейсу обрано використання архітектурного підходу Microfrontends та метод його реалізації за допомогою інструменту Module Federation. Module Federation впроваджений, використовуючи сторонній пакет @angular-architects/module-federation, який спрощує метод інтеграції плагіну Module Federation з Webpack у Angular-додатки.

Кожен мікрофронтенд додатку створений, використовуючи фреймворк Angular, мову TypeScript та супутні обраному фреймворку наступні залежності:

- RxJS – бібліотека для роботи з асинхронними даними [22];
- NgRx – бібліотека для управління станом додатку [23];
- PrimeNG – бібліотека UI-компонентів [24].

Використовуючи перелічені бібліотеки, реалізовані спроєктовані мікрофронтенди Host, Auth, Courses, Enrollments, Admin та бібліотека спільного використання. Для синхронізації стану мікрофронтендів, зокрема при роботі з автентифікацією та авторизацією, використано метод визначення кастомних DOM-подій.

Додаток Host є головним додатком для реалізації Module Federation. Даний мікрофронтенд відповідає за інтеграцію всіх інших додатків на основі конфігураційних даних. Також додаток Host забезпечує загальну структуру і навігацію системи, об’єднуючи окремі мікрофронтенди в єдиний інтерфейс.

Додаток Auth відповідає за автентифікацію та авторизацію користувачів. Інтерфейс створення облікового запису користувача наведено на рисунку 3.1. Також мікрофронтенд Auth містить функціонал роботи з даними облікового запису користувача (рис. 3.2).

Інтерфейс, що належить до функціоналу зміни паролю облікового запису, наведено на рисунку 3.3.

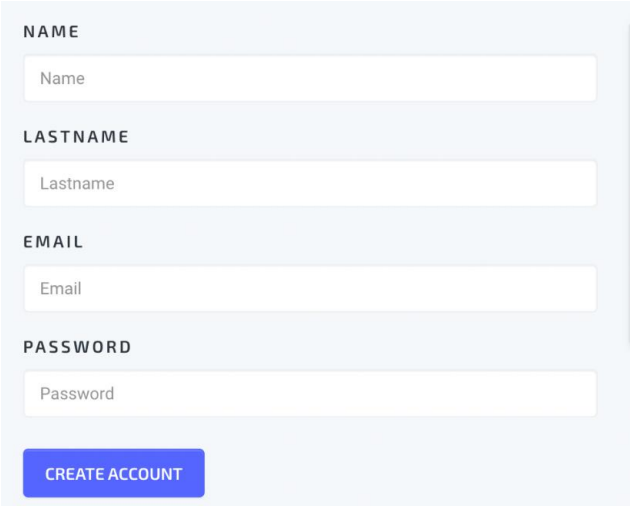


Рисунок 3.1 – Створення облікового запису користувача

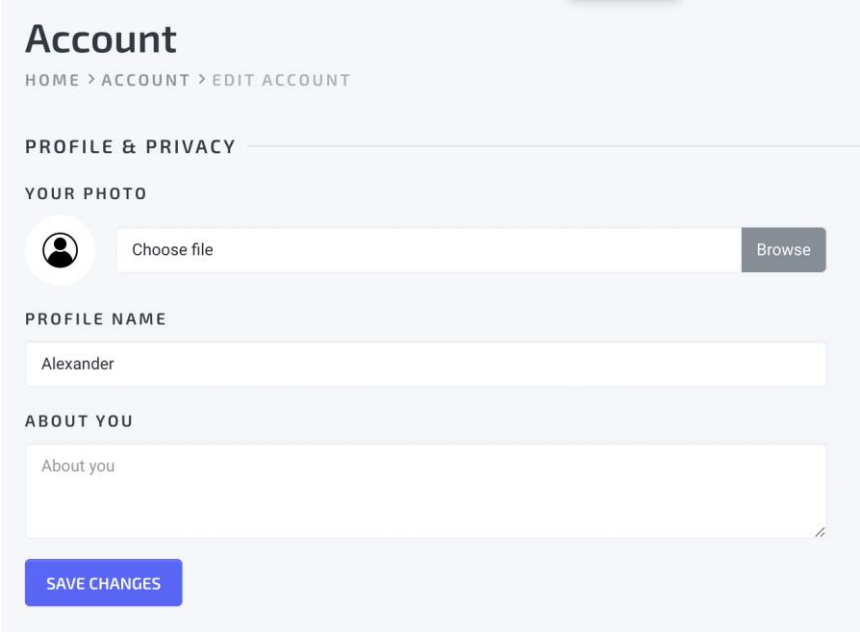


Рисунок 3.2 – Редагування профілю користувача

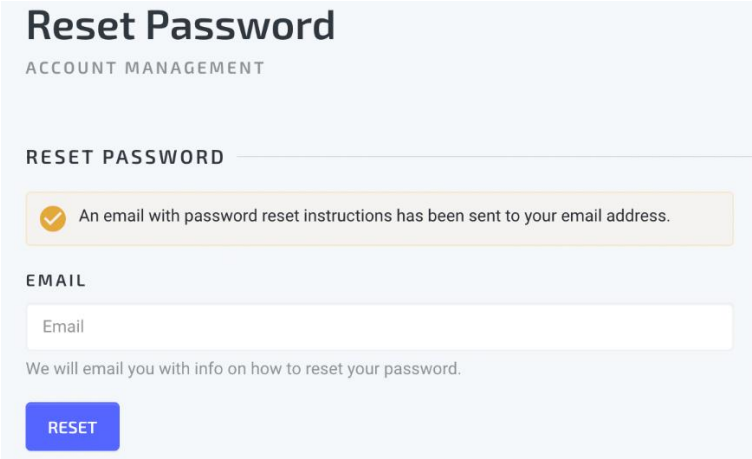


Рисунок 3.3 – Зміна паролю облікового запису

Мікрофронтенд Courses відповідає за відображення загального списку курсів з можливістю сортування та пошуку курсів за назвою, автором, категоріями, тегами (рис. 3.4). Крім того, додаток Courses містить відображення даних окремого курсу, в тому числі – відгуки про курс (рис. 3.5).

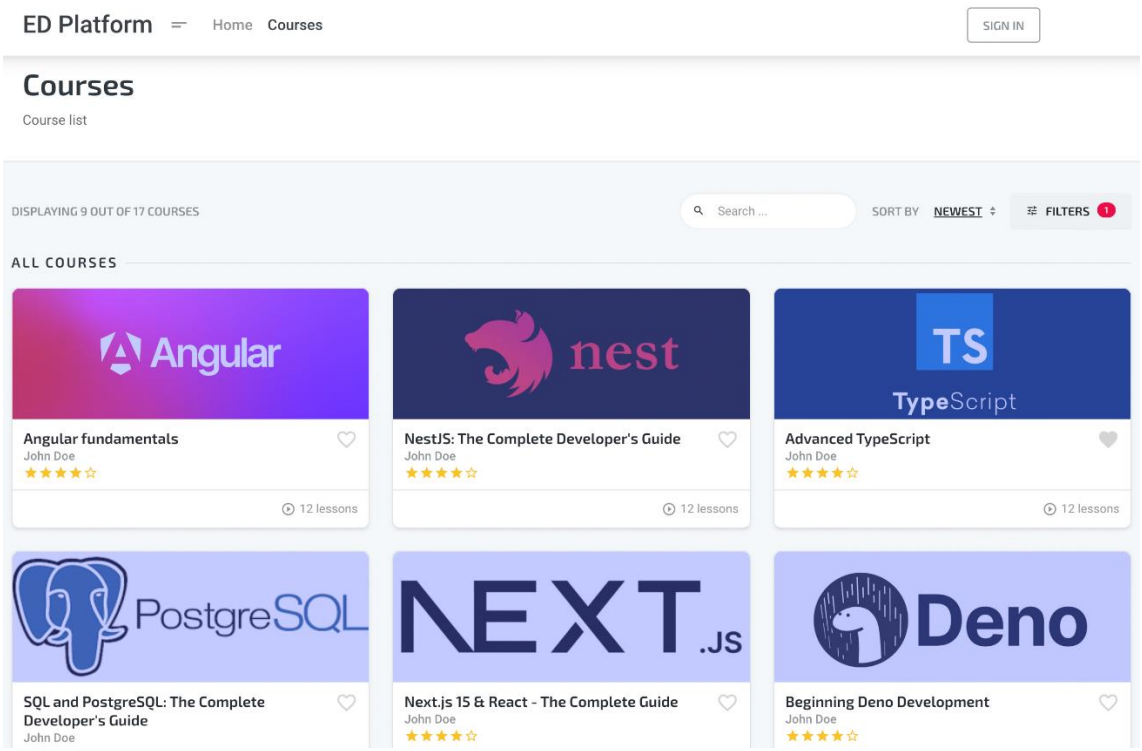


Рисунок 3.4 – Пошук курсів

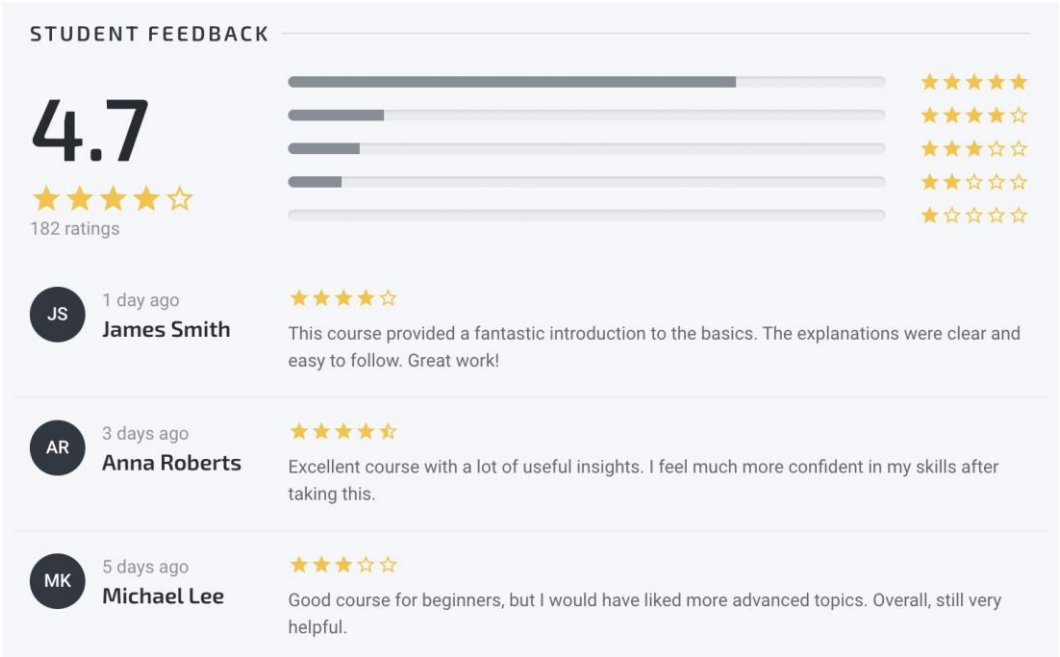


Рисунок 3.5 – Відгуки про курс

Також у мікрофронтенді Courses зосереджено функціонал, що дозволяє автору працювати над загальною інформацією про курс та його наповненням. Додаток має функціонал створення нового курсу, редагування загальної інформації про курс, додавання, редагування та видалення модулів, уроків та тестів курсу (рис. 3.6–3.7).

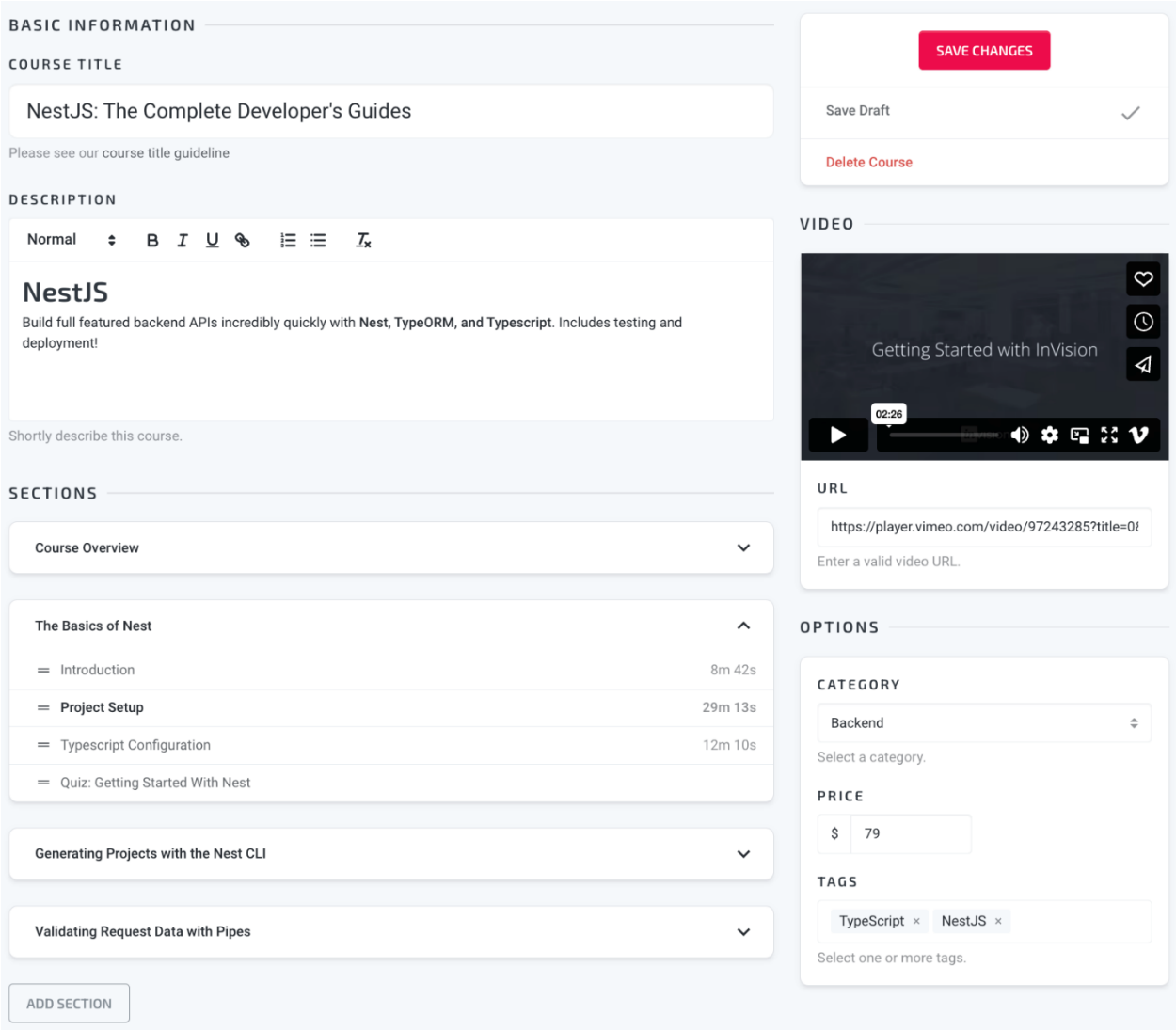


Рисунок 3.6 – Редагування матеріалу курсу

До мікрофронтенду Enrollments належить інтерфейс взаємодії слухача курсу з навчальними матеріалами (рис. 3.8–3.10). Даний мікрофронтенд також відповідальний за реєстрацію на курс, оплату курсу (рис. 3.11), завантаження сертифікату про завершення курсу.

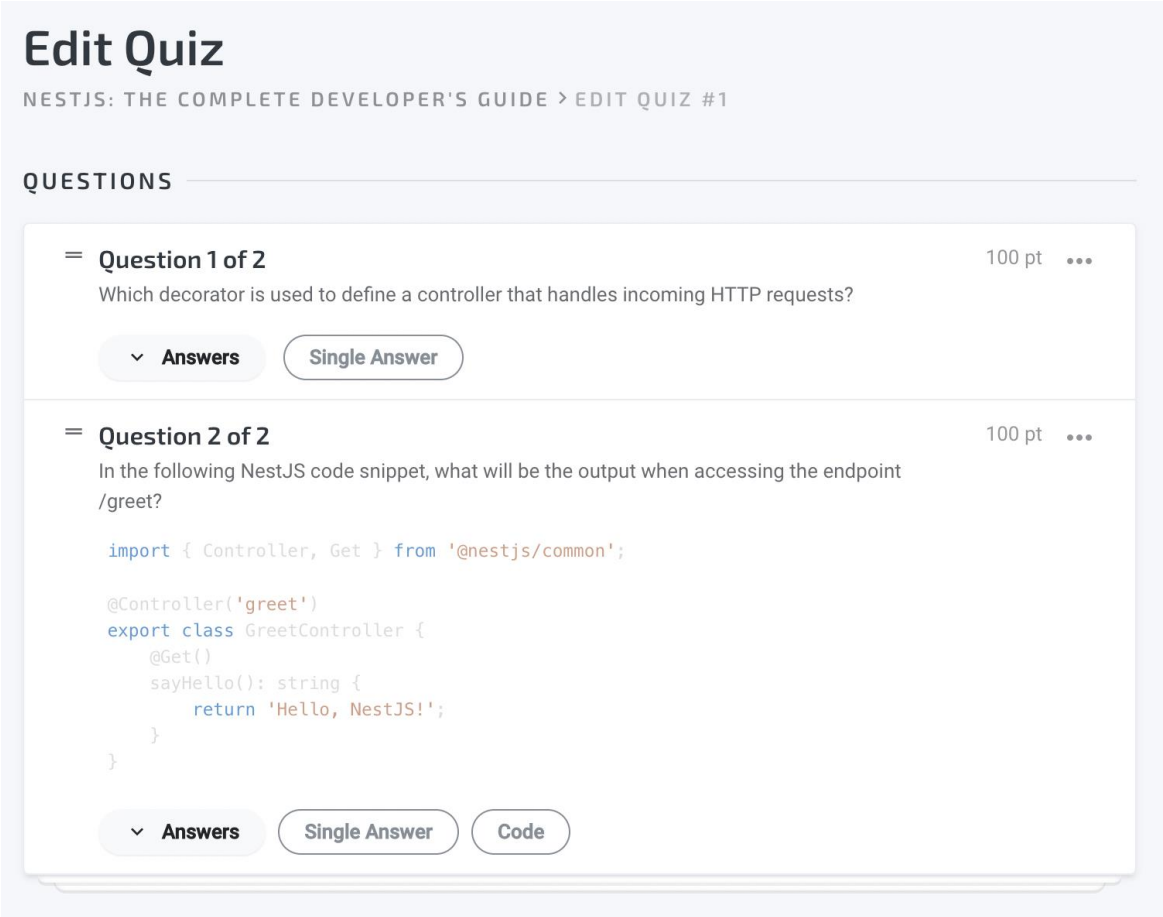


Рисунок 3.7 – Редагування тестування

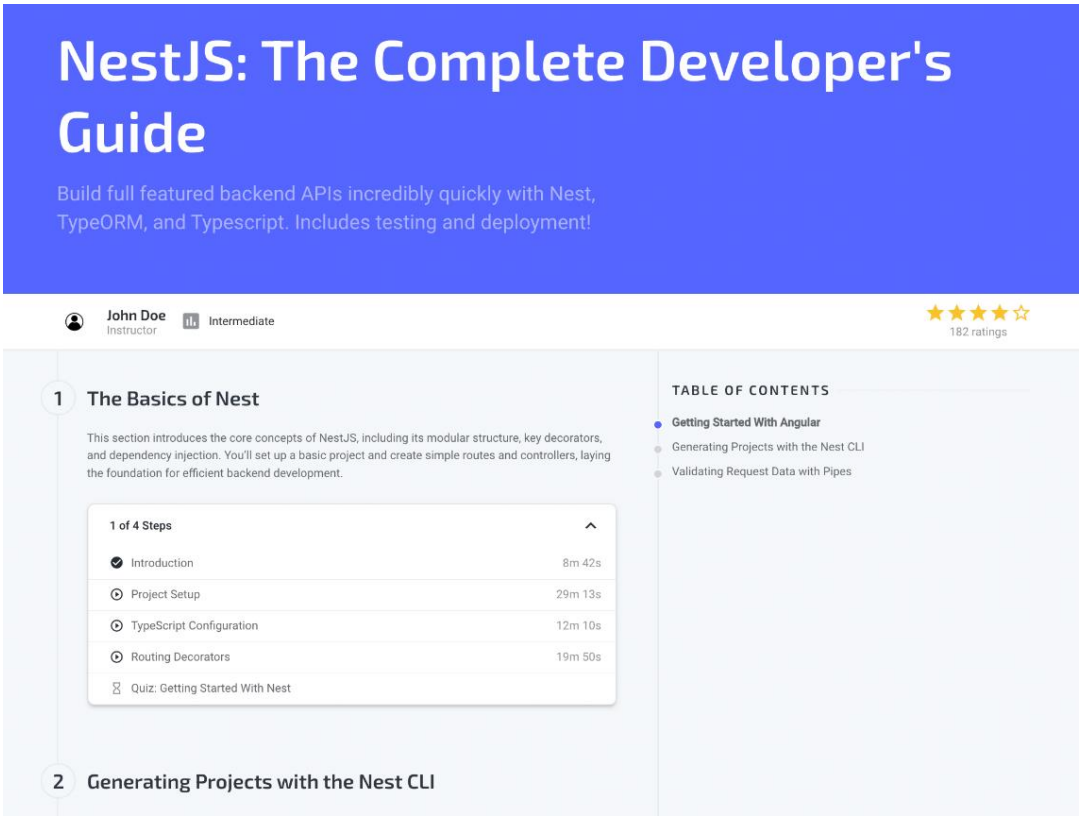


Рисунок 3.8 – Навчальні матеріали курсу

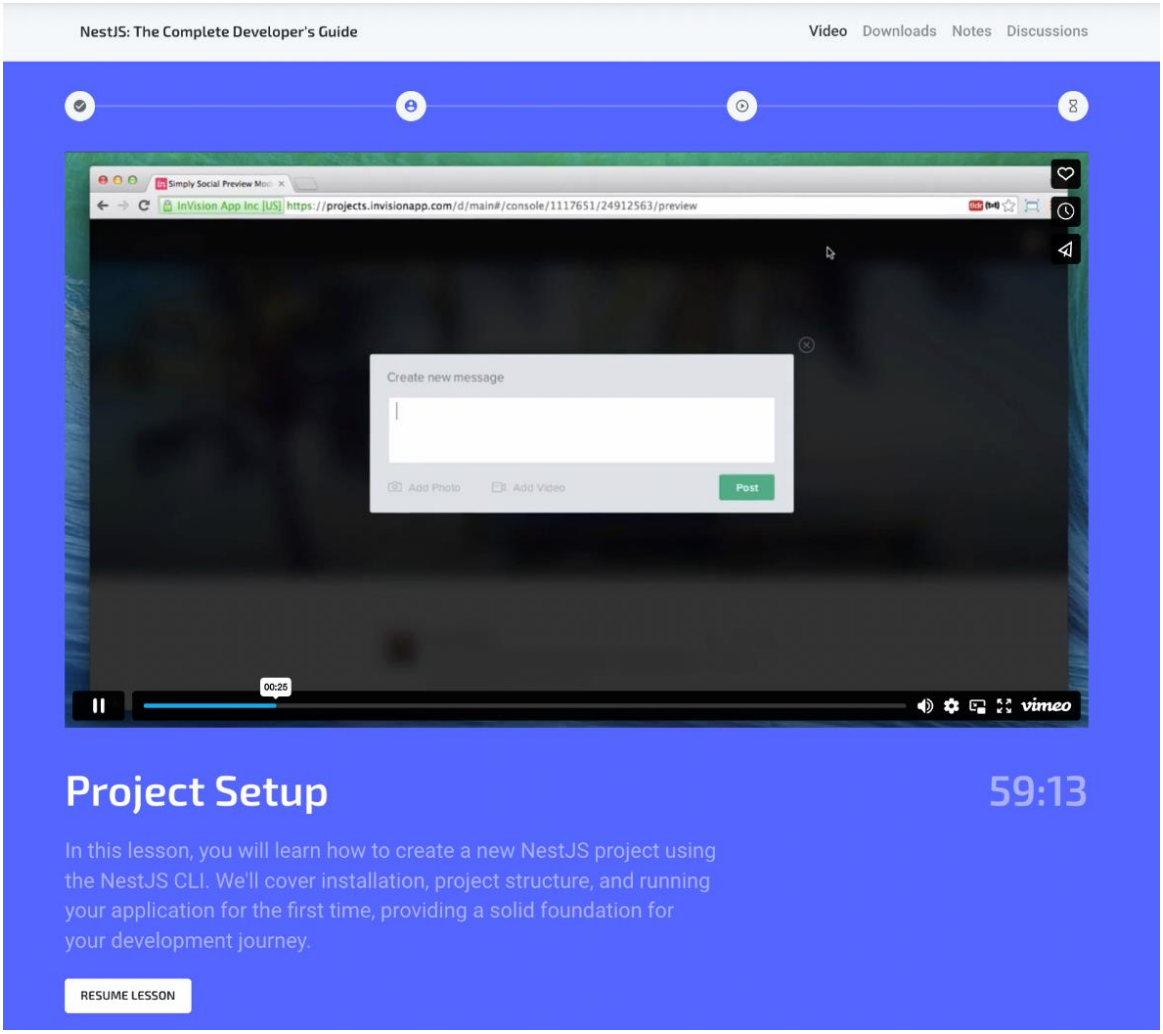


Рисунок 3.9 – Відеоурок курсу

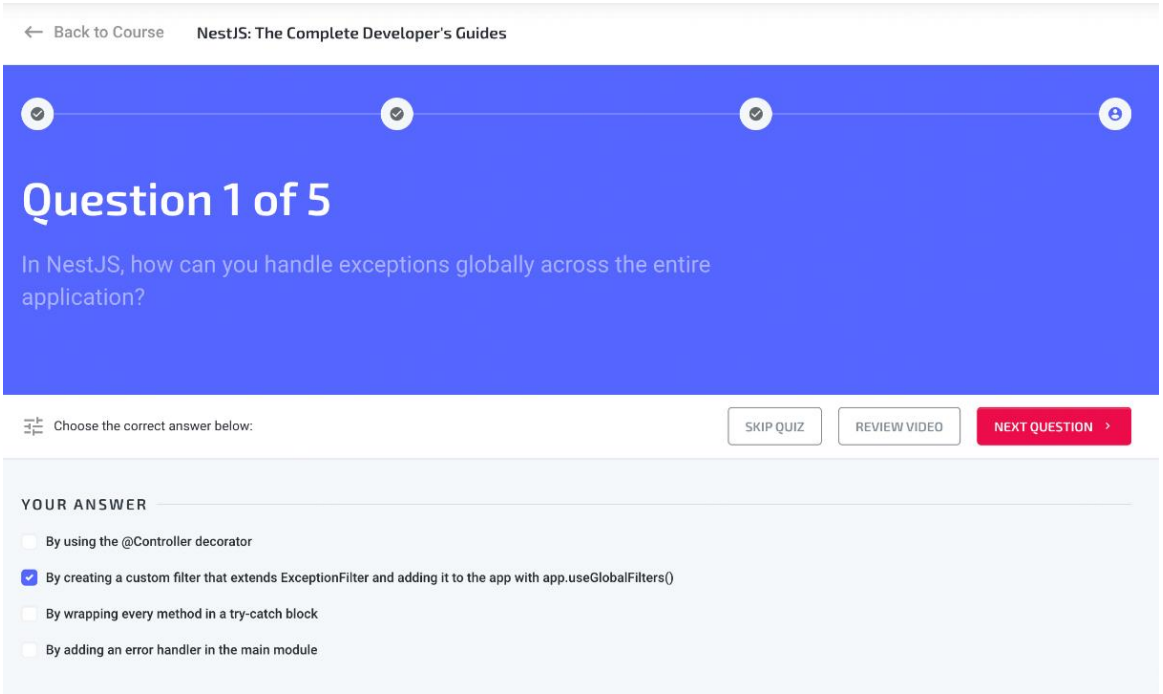


Рисунок 3.10 – Тестування курсу

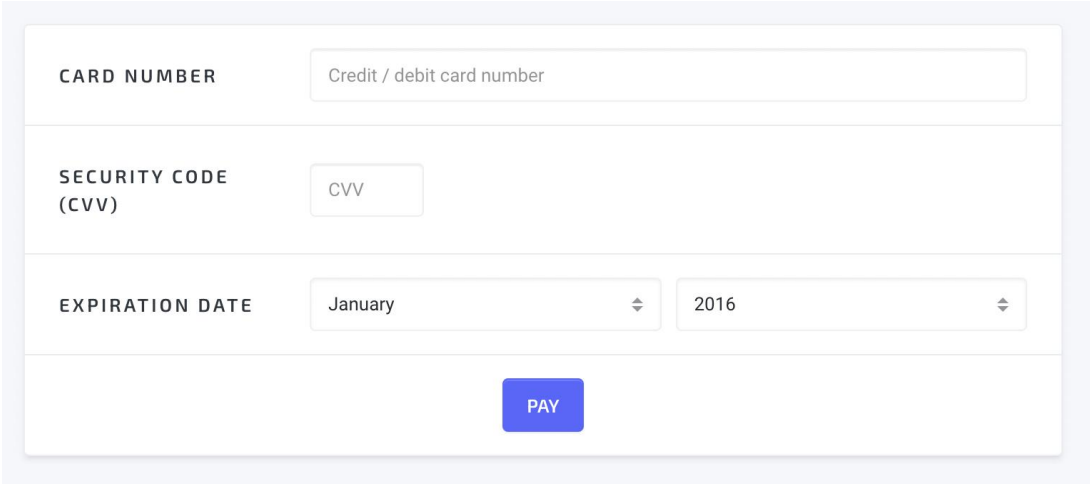


Рисунок 3.11 – Заповнення даних платіжної картки

В мікрофронтенді Admin зосереджено функціональні можливості, доступні для користувачів з роллю адміністратора. На рисунку 3.12 наведено інтерфейс для роботи з обліковими записами, також мікрофронтенд містить інтерфейси для роботи з відгуками до курсів, коментарями до уроків, категоріями та тегами курсів та ролями користувачів..

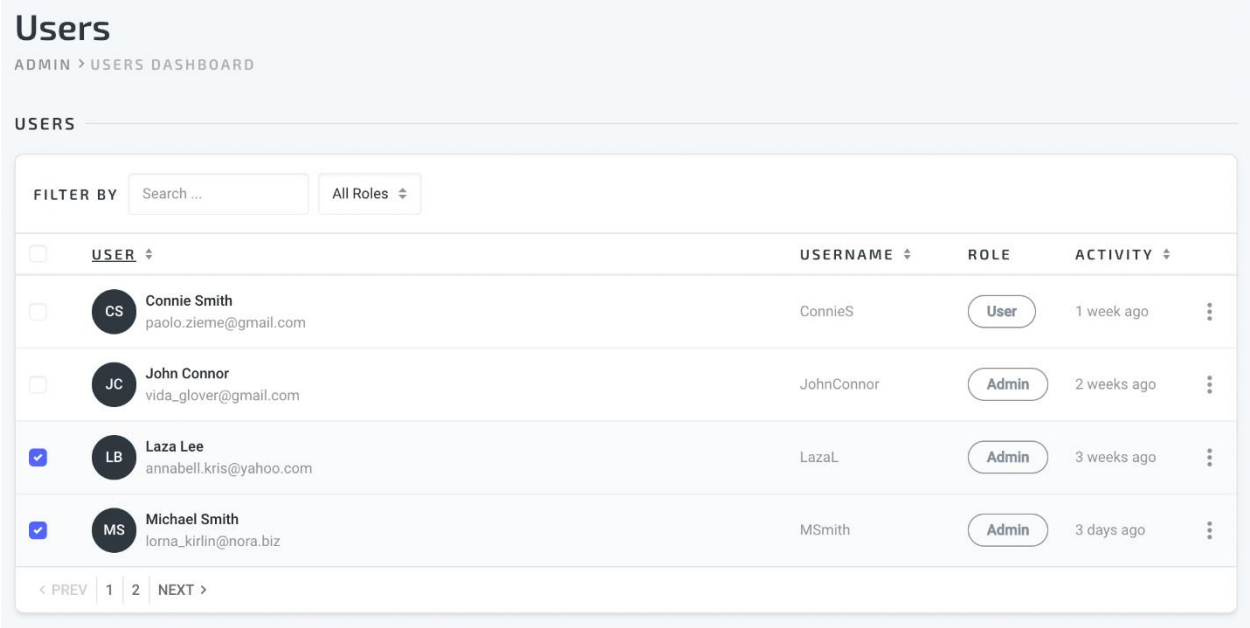


Рисунок 3.12 – Адміністрування облікових записів

Реалізована Angular-бібліотека, що містить спільні компоненти інтерфейсу та повторювану функціональність. Створена Angular-бібліотека розміщена на npm, проєкти мікрофронтендів використовують її як залежність.

3.3 Розгортання та управління інфраструктурою

Контейнеризацію кожного мікросервіса системи виконано за допомогою інструмента Docker. Оркестрацію контейнерів організовано, використовуючи інструмент Kubernetes.

Як хмарний сервіс використано DigitalOcean. Для організації CI/CD обрано інструмент для автоматизації процесів розробки GitHub Actions.

3.3.1 Контейнеризація сервісів з Docker

Використовуючи інструмент Docker, виконана контейнеризація мікросервісів проєкту.

Кожен мікросервіс проєкту має окремий Dockerfile, що описує процес збірки та налаштування контейнера для конкретного сервісу.

Для налаштування локального середовища розробки використано Docker Compose. Конфігураційний файл docker-compose.yml визначає налаштування необхідних контейнерів.

Деякі налаштування сервісів винесені у змінні оточення, що оголошенні у .env файлі:

- налаштування портів;
- ім’я контейнерів, які мають як залежність інші контейнери;
- параметри підключення до бази даних;
- ключі API для доступу до сторонніх сервісів та інтеграцій.

Необхідні для налаштування локального середовища розробки файли docker-compose.yml та .env винесені у окремий репозиторій.

Крім безпосередньо контейнерів мікросервісів проєкту, задані налаштування наступних контейнерів, необхідних для роботи цих сервісів:

- окремих баз даних СУБД PostgreSQL для сервісів Auth, Courses та Enrollments;

– сховища Redis для сервісу Auth;
– брокера повідомлень RabbitMQ для взаємодії з сервісом Notifications.

Визначено томи для зберігання даних вказаних баз даних та брокера повідомлень, які дозволяють зберігати інформацію між перезапусками контейнерів.

Також визначено томи для локальної розробки, що забезпечують можливість внесення змін у кодову базу мікросервісу без необхідності перезапуску відповідного йому контейнера.

За замовчуванням Dockerfile описані для робочого середовища. У файлі docker-compose.yml відбувається адаптація Dockerfile під потреби локального середовища. Зокрема, для сервісів серверної частини змінюється команда, яка виконується на початку роботи контейнерів.

Для сервісів клієнтського проєкту, що є Angular-додатками, описані окремі файли налаштувань Dockerfile та Dockerfile.dev. Dockerfile виконує збірку проєкту та використовує образ вебсерверу NGINX. Dockerfile.dev містить команду, що використовує сервер, наданий Angular CLI – Webpack Dev Server, який призначений для роботи у локальному середовищі розробки.

Для зберігання створених Docker-образів було обрано сервіс Container Registry хмарної платформи DigitalOcean.

3.3.2 Оркестрація контейнерів з Kubernetes

Кожен мікросервіс має окремий ресурс Deployment, який відповідає за розгортання, оновлення та відновлення у разі завершення роботи реплік контейнерів. Конфігурації Deployment визначають необхідну кількість реплік для кожного мікросервісу.

Для кожного мікросервісу також визначено ресурс Service, який забезпечує доступ до реплік контейнерів. Service також виконує функцію балансування навантаження, розподіляючи запити між всіма репліками мікросервісу. Конфігурації ресурсів Service описано з використанням типу

ClusterIP. Сервіс типу ClusterIP не надає зовнішнього доступу, обмежуючи комунікацію лише внутрішньою мережею кластера.

Налаштовано ресурс Ingress для управління зовнішнім доступом до API Gateway мікросервісу. Налаштовано Ingress Controller, який відповідає за обробку вхідного трафіку і маршрутизацію запитів до відповідних сервісів у кластері. Як Ingress Controller використано NGINX Ingress Controller.

Налаштовано підтримку HTTPS. Для автоматизації управління сертифікатами SSL/TLS використано інструмент Cert Manager. Cert Manager спрощує процес отримання, встановлення та автоматичного оновлення сертифікатів. Виконано налаштування Cert Manager для інтеграції з Let's Encrypt. Let's Encrypt є безкоштовним CA (сертифікаційним центром), який надає SSL/TLS сертифікати.

Контейнеризація додатків зі станом супроводжується необхідністю вирішення питань управління станом, масштабування, зберігання та відновлення даних. Для контейнерів, які мають стан, а саме: реляційної СУБД PostgreSQL, сховища типу «ключ-значення» Redis та брокера повідомлень RabbitMQ, створені ресурси Kubernetes типу StatefulSet.

Для локальної роботи над кластером Kubernetes використовувався Minikube. Minikube створює кластер, який складається з однієї ноди, яка виконує функції як мастер-ноди, так і робочої ноди. Це дозволяє тестувати конфігураційні файли Kubernetes в локальному середовищі.

Для розгортання і управління кластером Kubernetes у хмарному середовищі використано сервіс DigitalOcean Kubernetes, який інтегрується з іншими сервісами DigitalOcean, такими як балансувальники навантаження та реєстр контейнерних образів [27].

3.3.3 Впровадження процесів CI/CD

Кодова база кожного мікросервісу та мікрофронтенду розміщена у окремому репозиторії. В кожному з репозиторіїв виначено окремі процеси

CI/CD. Такий підхід реалізує можливість незалежного розгортання кожного сервісу серверної та клієнтської частин.

Впроваджені процеси CI (безперервної інтеграції) складаються з наступних етапів:

– виконання unit-тестів мікросервісу;
– побудова нової версії Docker-образу;
– розміщення Docker-образу на віддаленому реєстрі.

Частина CD (безперервного розгортання) полягає у розгортанні нової версії Docker-образу мікросервіса у хмарному середовищі [28].

Реалізація CI/CD виконана, використовуючи інструмент для автоматизації процесів розробки GitHub Actions.

У директорії .github/workflows кожного репозиторію задана конфігурація workflow у форматі YAML [30]. Кожен workflow репозиторію складається з трьох jobs.

Job «test» та job «build-and-push» реалізують CI, виконуючи unit-тести, збірку нової версії Docker-образу, інтеграцію з DigitalOcean Container Registry для розміщення створеного Docker-образу.

Job «deploy» реалізує CD, виконуючи інтеграцію з кластером Kubernetes, створеного на хмарній платформі DigitalOcean. Job складається з наступних кроків:

– встановлення інструменту doclt – DigitalOcean CLI;
– встановлення інструменту kubectl – Kubernetes CLI;
– збереження конфігураційного файлу Kubernetes для кластеру, розгорнутого на платформі DigitalOcean;
– оновлення ресурсів Kubernetes, які пов’язані із зібраним на етапі CI Docker-образом.

CI/CD-пайплайн бібліотеки користувацького інтерфейсу складається з кроків виконання модульних тестів, виконання побудови бібліотеки та розміщення її нової версії на npm.

Для опису кроків CI/CD-пайплайну було використано інструменти з GitHub Marketplace.

CI/CD-пайплайн використовує дані, збережені за допомогою сервіса GitHub Secrets, що дозволяє уникнути зберігання у репозиторії таких даних, як API ключі, токени доступу до хмарного сервісу, параметри підключення до баз даних [30]. Щоб уникнути зберігання ресурсів Kubernetes Secrets в репозиторії, створення цих ресурсів відбується під час роботи CI/CD-пайплайну. Створення ресурсів Kubernetes Secrets винесено у Bash-скрипти.

Так як всі конфігурації Kubernetes розміщені в окремому репозиторії, цей репозиторій теж має окремий пайплайн, який застосовує визначені в ньому конфігурації до кластеру.

3.4 Висновки з розділу 3

У третьому розділі обрано засоби розробки та зовнішні сервіси, які дозволяють реалізувати побудовану архітектуру.

Описано реалізацію мікросервісів серверної частини, використовучи платформу Node.js, мову TypeScript та фреймворк NestJS.

Наведено опис роботи з сховищами даних – реляційною СУБД PostgreSQL та сховищем типу «ключ-значення» Redis.

Описані виконані інтеграції із стороніми сервісами – платіжним шлюзом Stripe та S3-сумісним сховищем DigitalOcean Spaces.

Описано реалізацію інтерфейсів мікросервісів з використанням архітектурного стилю REST API, системи віддаленого виклику процедур gRPC та брокера повідомлень RabbitMQ.

Наведено опис впровадження підходу Microfrontends для реалізації додатку користувацького інтерфейсу. Описано реалізацію Angular-додатків та Angular-бібліотеки.

Виконано контейнеризацію мікросервісів та оркестрацію контейнерів інструментами Docker та Kubernetes. Впроваджено CI/CD, використовуючи Github Actions з інтеграцією з хмарним сервісом DigitalOcean.

Результатом роботи над даним розділом є закінчена програмна реалізація проєкту освітньої платформи, яка відповідає визначеним вимогам та реалізована, дотримуючись побудованої архітектури.

ВИСНОВКИ

Результатом виконання кваліфікаційної роботи є реалізований проєкт освітньої платформи. Основне призначення платформи полягає в організації та наданні доступу до структурованих навчальних матеріалів.

У кваліфікаційній роботі наведено характеристику предметної області, проведено аналіз існучих освітніх платформ. Проведено аналіз мікросервісної архітектури, визначено її особливості, переваги та недоліки. Виконано аналіз основних принципів побудови мікросервісної архітектури. Розглянуто застосування принципів мікросервісної архітектури при побудові клієнтських інтерфейсів. Наведено огляд засобів розробки, зокрема платформи Node.js та фреймворку Angular.

Виконано проєктування системи освітньої платформи: визначено призначення розробки, сформовано функціональні та технічні вимоги, розроблено діаграму використання, виконано проєктування користувацького інтерфейсу, побудовано мікросервісну архітектуру.

Програмна реалізація проєкту оствітньої платформи виконана за допомогою платформи Node.js та фреймворку Angular. Розроблений додаток відповідає меті роботи та визначеним функціональним та технічним вимогам.

ПЕРЕЛІК ПОСИЛАНЬ

1. Валяєва А. О., Лимаренко Ю. О. Розробка освітньої платформи на основі мікросервісної архітектури із використанням Node.js та Angular. *Актуальні проблеми математики та інформатики*: Збірка тез доповідей П’ятнадцятої Всеукраїнської, двадцять другої регіональної наукової конференції молодих дослідників. Запоріжжя: ЗНУ, 2024. С. 17–18.
2. Microservices architecture design. URL: https://learn.microsoft.com/en-us/azure/architecture/microservices (дата звернення: 05.05.2024).
3. Communication in a microservice architecture. URL: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture (дата звернення: 05.05.2024).
4. RESTful web API design. URL: https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design (дата звернення: 10.05.2024).
5. gRPC documentation. URL: https://grpc.io/docs (дата звернення: 15.05.2024).
6. What’s the Difference Between gRPC and REST. URL: https://aws.amazon.com/compare/the-difference-between-grpc-and-rest (дата звернення: 15.05.2024).
7. Performance best practices with gRPC. URL: https://learn.microsoft.com/en-us/aspnet/core/grpc/performance (дата звернення: 15.05.2024).
8. Protocol Buffers documentation. URL: https://protobuf.dev (дата звернення: 15.05.2024).
9. Asynchronous message-based communication. URL: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/asynchronous-message-based-communication (дата звернення: 20.05.2024).
10. RabbitMQ documentation. URL: https://www.rabbitmq.com (дата звернення: 20.05.2024).

11. Design patterns for microservices. URL: https://learn.microsoft.com/en-us/azure/architecture/microservices/design/patterns (дата звернення: 22.05.2024).
12. Pattern: Database per service. URL: https://microservices.io/patterns/data/database-per-service (дата звернення: 22.05.2024).
13. Micro Frontend Architecture. URL: https://nx.dev/concepts/module-federation/micro-frontend-architecture (дата звернення: 20.05.2024).
14. Micro frontends. URL: https://microfrontends.com/ (дата звернення: 20.05.2024).
15. Module Federation documentation. URL: https://module-federation.io (дата звернення: 22.05.2024).
16. PostgreSQL documentation. URL: https://www.postgresql.org/docs (дата звернення: 02.09.2024).
17. TypeScript documentation. URL: https://www.typescriptlang.org (дата звернення: 10.09.2024).
18. Node.js documentation. URL: https://nodejs.org/dist/latest-v20.x/docs/api (дата звернення: 17.09.2024).
19. NestJS documentation. URL: https://docs.nestjs.com (дата звернення: 17.09.2024).
20. Prisma documentation. URL: https://www.prisma.io (дата звернення: 17.09.2024).
21. Angular documentation. URL: https://angular.dev (дата звернення: 29.09.2024).
22. RxJS documentation. URL: https://rxjs.dev (дата звернення: 29.09.2024).
23. NgRx documentation. URL: https://ngrx.io (дата звернення: 29.09.2024).
24. PrimeNG documentation. URL: https://primeng.org (дата звернення: 29.09.2024).
25. Docker documentation. URL: https://docs.docker.com (дата звернення: 10.10.2024).

26. Kubernetes documentation. URL: https://kubernetes.io/docs/home (дата звернення: 17.10.2024).
27. DigitalOcean Kubernetes: Infrastructure Best Practices. URL: https://www.digitalocean.com/community/developer-center/digitalocean-kubernetes-infrastructure-best-practices (дата звернення: 17.10.2024).
28. CI/CD for microservices architectures. URL: https://learn.microsoft.com/en-us/azure/architecture/microservices/ci-cd (дата звернення: 19.10.2024).
29. GitHub Actions documentation. URL: https://docs.github.com/en/actions (дата звернення: 20.10.2024).
30. Integrating with GitHub Actions – CI/CD pipeline to deploy a Web App to Amazon EC2. URL: https://aws.amazon.com/blogs/devops/integrating-with-github-actions-ci-cd-pipeline-to-deploy-a-web-app-to-amazon-ec2 (дата звернення: 20.10.2024).