МІНІСТЕРСТВО ОСВІТИ НАУКИ УКРАЇНИ

ПРИВАТНЕ АКЦІОНЕРНЕ ТОВАРИСТВО
«ПРИВАТНИЙ ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД
«ЗАПОРІЗЬКИЙ ІНСТИТУТ ЕКОНОМІКИ
ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ»

Кафедра економічної кібернетики та інженерії програмного забезпечення

ДО ЗАХИСТУ ДОПУЩЕНА

Завідувач кафедри,
д.е.н., доц.

_____ С.І. Левицький

БАКАЛАВРСЬКА ДИПЛОМНА РОБОТА «РОЗРОБКА СЕРВЕРНОГО ФУНКЦІОНАЛУ ВЕБ-ЗАСТОСУНКІВ НА БАЗІ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ»

Виконав

ст. гр. ІПЗ – 217 _____ Д.В. Бєлоусов

Керівник

доц. _____ О.А. Жеребцов

Запоріжжя

2022

ПРИВАТНЕ АКЦІОНЕРНЕ ТОВАРИСТВО «ПРИВАТНИЙ ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД «ЗАПОРІЗЬКИЙ ІНСТИТУТ ЕКОНОМІКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ»

Кафедра економічної кібернетики та інженерії програмного забезпечення

ЗАТВЕРДЖУЮ

Завідувач кафедри,
д.е.н., доц.

_____ С.І. Левицький

17 січня 2022 р.

З А В Д А Н Н Я

НА БАКАЛАВРСЬКУ ДИПЛОМНУ РОБОТУ

студенту гр. ІПЗ-217 ,

спеціальності 121 - «Інженерія програмного забезпечення»

Бєлоусову Дмитру Вадимовичу

1.Тема: Розробка серверного функціоналу веб-застосунків на базі мікросервісної архітектури

затверджена наказом № 06.2 – 2-7-2 від 15 січня 2022 р.

2. Термін здачі студентом закінченої роботи: 18 червня 2022 р.

3. Перелік питань, що підлягають розробці:

1. Розглянути питання актуальності розробки застосунку.

2. Провести огляд галузі та аналітику проблеми і її рішень загалом.

3. Провести огляд та аналіз популярних аналогів, зробити висновки про вимоги до проекту.

4. Розглянути методи та способи створення веб-застосунків та обрати направлення розробки.

5. Провести огляд стеку технологій для розробки проекту та зробити вибір на підставі вимог.

6. Розробити проектування архітектури веб-застосунку.

7. Створити відповідний застосунок, спираючись на отримані дані.

8. Проаналізувати отримані результати.

9. Оформити звіт за результатами роботи

4. Календарний графік підготовки випускної роботи молодшого спеціаліста

№ етапу	Зміст	Терміни виконання	Готовність по графіку %, підпис керівника	Підпис керівника про повну готовність етапу, дата
1	Формулювання (корегування) теми випускної роботи молодшого спеціаліста та збір практичного матеріалу за темою випускної роботи	17.01.22-19.02.22		
2	**І атестація** І розділ випускної роботи молодшого спеціаліста	26.03.22-02.04.22		
3	**ІІ атестація** ІІ розділ випускної роботи молодшого спеціаліста	23.04.22-30.04.22		
4	**ІІІ атестація** ІІІ розділ випускної роботи молодшого спеціаліста, висновки та рекомендації, додатки, реферат	21.05.22-28.05.22		
5	Перевірка випускної роботи молодшого спеціаліста програмою «Антиплагіат»	25.05.22-18.06.22		
6	Доопрацювання випускної роботи молодшого спеціаліста, підготовка презентації, отримання відгуку керівника і рецензії	30.05.22-11.06.22		
7	**Попередній захист випускної роботи молодшого спеціаліста**	**14.06.22-18.06.22**		
8	Подача випускної роботи молодшого спеціаліста на кафедру	за 3 дні до захисту		
9	**Захист випускної роботи молодшого спеціаліста**	**20.06.22-25.06.22**		

Керівник _____ О.А. Жеребцов
(підпис) (ініціали, прізвище)

«____» _____ 2022 р.

Завдання отримав до виконання _____ Д.В. Бєлоусов
(підпис студента) (ініціали, прізвище)

«____» _____ 2022 р.

РЕФЕРАТ

Бакалаврська робота містить 97 сторінок, одна таблиця, 30 рисунків, один додаток, 15 бібліографічних посилань.

Метою роботи є розробка серверного функціоналу веб-застосунків на базі мікросервісної архітектури за допомогою сучасних веб-технологій.

Об’єктом дослідження є сучасна система розробки серверної частини додатку за допомогою мікросервісної архітектури.

Предметом дослідження є архітектурне рішення для зручної роботи з великими серверними додатками.

Проект реалізовано за допомогою таких засобів як NestJS, NodeJS та JavaScript. Здійснено проектування моделі предметної області, програмування сутностей та алгоритмів на базі серверної архітектури.

API сервера є легким у використанні за допомоги Swagger. Застосунок дозволяє заощаджувати час користувача за рахунок мінімального введення даних.

NEST.JS, TYPESCRIPT, MICROSERVICES, MONGODB, SWAGGER

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

Скорочення	Повна назва	Пояснення/переклад
API	Application Programming Interface	Прикладний програмний інтерфейс
CRUD	Create, Read, Update і Delete	Створення, зчитування, оновлення та видалення
CLI	Command-line interface	Текстовий інтерфейс користувача, в якому інструкції можна дати тільки введенням із клавіатури текстових рядків
GIT	Global Information Tracker	Система керування версіями з розподіленою архітектурою
HTTP	Hyper Text Transfer Protocol	Протокол передачі гіпертекстових документів – протокол передачі даних, що використовується в комп'ютерних мережах
IT	Information Technology	Система методів, процесів та способів використання обчислювальної техніки і систем зв'язку для створення, збору, передачі, пошуку, оброблення та поширення інформації з метою ефективної організації діяльності людей
JS	Java Script	Динамічна, об'єктно-орієнтована прототипна мова програмування; реалізація стандарту ECMAScript
TS	Type Script	мова програмування, представлена Microsoft восени 2012; позиціонується як засіб розробки вебзастосунків, що розширює можливості JavaScript
JSON	JavaScript Object Notation	Текстовий формат обміну даними на основі JavaScript
MongoDB		документо-орієнтована система керування базами даних з відкритим вихідним кодом, яка не потребує опису схеми таблиць.
NestJS		платформа для створення ефективних, масштабованих додатків Node.js на стороні сервера.
REST	Representational State Transfer	Архітектурний стиль взаємодії компонентів розподіленого застосування в мережі
SQL	Structured query language	Мова структурованих запитів
UI	User Interface	Користувацький інтерфейс

URL	Uniform Resource Locator	Уніфікований локатор ресурсів або адреса ресурсу
URI	Uniform Resource Locator	Уніфікований локатор ресурсів або адреса ресурсу
БД	База даних	Сукупність даних, що зберігаються відповідно до схеми даних, маніпуляція яких виконується відповідно до правил моделювання даних
ПІБ	Прізвище, ім'я, по батькові	
ПК	Персональний комп'ютер	Однокористувальницька, універсальна мікроЕОМ
СУБД	Система управління базами даних	Поєднання програмного забезпечення та лінгвістичних інструментів для загальних або спеціальних цілей, що забезпечують управління створенням та використанням баз даних
NOSQL	Not Only SQL	база даних, яка забезпечує механізм зберігання та видобування даних відмінний від підходу таблиць-відношень в реляційних базах даних.

ВСТУП

У наш час, коли у світі все активніше відбувається розвиток IT-технологій та розмір серверних додатків невпинно зростає, необхідно все більше часу витрачати на підтримку великих додатків для Ентерпрайз бізнесу. У величезних компаніях дуже складно контролювати роботу коли розміри коду перевищують мільйон рядків коду. Для вирішення цих проблем було знайдено унікальний спосіб розділяти обов’язки на великих проектах. Мікросервісна архітектура дозволяє розділяти розробку на частин проекту для того, щоб одночасно працювати великій кількості людей без зайвих обмежені які часто відбуваються на проектах типу «моноліт».

Однак, на даний час дана технологія в Україні є ще недостатньо розвиненою, а веб-застосунки у даній області мають недостатній функціонал та певні обмеження. Тому створення веб-застосунку з використанням мікросервісно є актуальною задачею розробки.

РОЗДІЛ 1
ТЕОРЕТИЧНА ЧАСТИНА

1.1 Актуальність теми

Архітектура має дуже важливий вплив на розрубку веб-додатку. В основному відрізняють два підходи, монолітну архітектуру та мікросервісну.

Вираз "монолітна архітектура" відразу асоціюється зі словом "моноліт". А монолітом ще з давніх-давен називають великий єдиний блок з каменю або бетону. Моноліт — це щось велике та єдине, що має спільну та потужну структуру. Моноліт - це сила і на віки.

У програмуванні «монолітна архітектура» також має на увазі наявність спільної та єдиної платформи, де сконцентровані всі компоненти однієї програми. Скільки б не нараховувалося подібних компонентів, усі вони уніфіковані і водночас управляються з однієї місця. У цьому вся і визначається сила «монолітних» додатків.

Багато сучасних стартапів вибирають саме монолітну архітектуру програми, тому що вона комфортна при роботі невеликими групами розробників. При її використанні всі компоненти програми взаємозв'язуються та взаємозамінні – це допомагає розвивати програму автономної та самодостатньої. Монолітна архітектура вважається традиційною та перевіреною при розробці додатків, але в той же час багато розробників вважають такий підхід у реалізації додатків старомодним і вже нікуди не придатним.

Щоб розуміти, чи підійде такий спосіб розробки чи ні, потрібно розглянути переваги і недоліки монолітної архітектури.

Переваги монолітної архітектури:

- Проста розробка та простий запуск програми. Через те, що вся розробка сконцентрована в одному місці, легко інтегрувати інструменти для полегшеної розробки, ті ж каталоги або бібліотеки. Плюс у разі потреби змінити елементи програми не потрібно вносити зміни окремо в різних місцях — все робиться в одному місці.
- Наскрізні проблеми практично відсутні. Велика кількість програм мають залежність від завдань, які здійснюються між компонентами програми: логи, обмеження швидкості, контрольні журнали і т. д. При монолітній архітектурі ці проблеми практично відсутні, тому що все сконцентровано в одному коді і все працює в одному додатку.
- Покращена продуктивність. Якщо враховувати, що програми були зібрані правильно, то той самий додаток при монолітній архітектурі буде працювати продуктивніше, ніж при мікросервісах. Це, знову ж таки, забезпечується єдиним кодом програми та роботою з «одного» місця.

Недоліки монолітної архітектури:

- Великий обсяг коду. Якщо продукт, що розробляється, досить великий і постійно масштабується, то згодом його код розростається до великих розмірів. Це ускладнює його розуміння та подальше обслуговування. Плюс може настати момент, коли код буде перевантажений і втратить від цього свою якість.
- Важко модернізується. Іноді потрібно додати до програми якусь нову «фішку». При монолітній архітектурі можна зіткнутися з багатьма перешкодами, щоб це реалізувати. Тому що в деяких випадках додати якусь фішку означає повністю переписати програму. А це довго та дорого.
- Гнучкість обмежена. З другого пункту випливає, що впровадження чогось нового — це ціла історія, і дуже часто це означає повторне розгортання програми, тому що вносити зміни потрібно на весь код. Ця ж ситуація стосується виправлення багів, додавання оновлень. Оновлення = переписана

заново програма. Тому за монолітної архітектури досить складно адаптувати вже працюючий додаток під свої потреби, тобто гнучкість «кульгає».

- Залежність між компонентами. З одного боку, це є перевагою, оскільки збільшує продуктивність. Але з іншого боку, якщо в якомусь компоненті програми буде помилка, це уповільнить або взагалі зупинить роботу всього додатка, а не одного компонента.

Через це можна прийти к висновку що для оптимізації роботи на великому підприємстві – використання «моноліту» не є оптимальним рішенням. Тому зараз більш популярним стає мікросервісна архітектура, яку розглянемо далі.

Поняття мікросервісної архітектури (Microservice Architecture) стає все популярнішим. Йдеться спосіб розробки софта шляхом об'єднання незалежно розгорнутих IT-сервисов. Давайте подивимося, які переваги має цей підхід.

1. Автономність та незалежність.

Мікросервісна архітектура дозволяє створювати незалежні крос-функціональні команди, націлені на вирішення конкретного бізнес-завдання. Ці команди самодостатні та максимально ефективні. Кожен елемент побудованої в такий спосіб IT-системи виконує потрібну функцію, тому може бути донесений до production незалежно від інших сервісів.

2. Можливість застосування різних технологій, мультиплатформенність.

Через мікросервіси можна поєднувати різні технології, вибираючи найкращі з можливих рішень. А використання стандартних протоколів взаємодії (HTTP-дзвінки через API, брокери повідомлень) дозволяє писати мікросервіси різними мовами програмування та використовувати різні технології зберігання даних.

Крім того, мікросервіси здатні функціонувати на будь-якому пристрої, в хмарних середовищах і в On-premise.

3. Хороша масштабованість.

Безумовно, йдеться про нову якість масштабованості, тому що в разі потреби вам не потрібно масштабувати всю систему і розбирати її вщент - достатньо буде внести зміни лише на конкретній ділянці IT-системи.

4. Стабільність та керованість.

Оскільки мікросервіси є незалежними одна від одної, стабільність системи підвищується. Збої та дефекти в одному мікросервісі не вплинуть на роботу інших, тому сама система функціонуватиме з мінімальними простоями.

Крім того, замість монолітного та надскладного масиву IT-системи ми отримаємо окремі компоненти з більш керованою архітектурою, де кожен елемент відповідатиме за свою функцію.

Залишається додати, що мікросервіси можна перепрофілювати інших завдань після початкового запуску, що забезпечить повторне використання.

1.2 Мікросервісна архітектура

Термін “Мікросервісна архітектура”, також відомий як мікросервіси, з’явився в середині 2010-х, щоб описати особливий архітектурний стиль розробки програмних додатків. Цей стиль розробки отримав розповсюдження в зв’язку з розвитком практик гнучкої розробки та DevOps. На даний момент мікросервісній архітектурі приділяється багато уваги: статті, блоги, дискусії в соціальних мережах і презентації на конференціях. Коли йдеться про забезпечення гнучкої розробки і доставки складних корпоративних додатків — такий спосіб розробки має значні переваги.

Якщо коротко, то архітектурний стиль мікросервісів — це підхід, коли єдиний додаток будується як сукупність невеликих, самодостатніх, незалежних, не тісно зв’язаних сервісів, що спілкуються між собою за допомогою легких механізмів як то HTTP, gRPC, AMQP. Ці сервіси побудовані навколо бізнес-потреб (кожен відповідальний за конкретний процес) та розгортаються незалежно з використанням повністю

автоматизованого середовища. Існує абсолютний мінімум централізованого управління цими сервісами. Самі по собі сервіси можуть бути написані на різних мовах і використовувати різні технології зберігання даних.

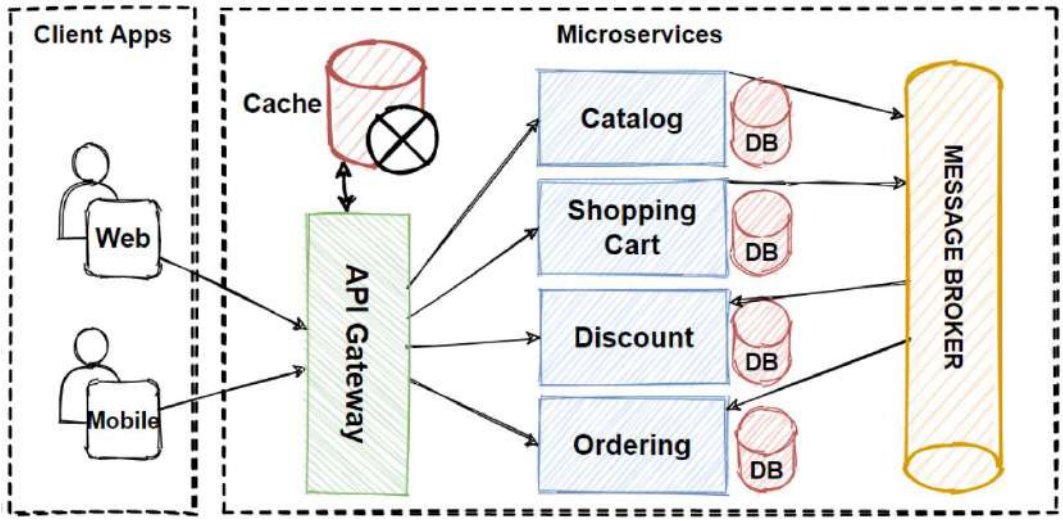

Рис. 1.1 - Схема роботи мікросервісної архітектури.

Причини використання.

Одна з причин використання мікросервісів полягає в тому, що компанії хочуть мати можливість швидко щось змінювати, щоб швидше реагувати на зміни бізнес-вимог, випереджати конкурентів. Мікросервіси допомагають розробникам доставляти зміни швидше, безпечніше і з більш високою якістю, тобто зберігати швидкість розвитку продукту, навіть коли той стає неосяжних розмірів. Адже не тісно зв’язані сервіси дають можливість проводити зміни з більшою частотою ітерацій мінімізуючи вплив змін на решту частин системи.

При всьому цьому не потрібно забувати що такий підхід додає додаткову складності проекту в цілому. Нам потрібні DevOps’и для моніторингу та управління, при цьому між ними і розробниками повинні бути тісні відносини і хороша взаємодія. При роботі з мікросервісами нам доводиться більше розгортати, ускладнюється система моніторингу, сильно розростається

кількість можливих збоїв. Тому в компанії дуже важлива сильна DevOps-культура.

Мікросервіси vs моноліт.

Для того щоб простіше розпочати знайомство з мікросервісною архітектурою потрібно порівняти цей стиль розробки з так званим монолітним стилем.

Моноліт будуються як єдине ціле. Будь які зміни, навіть самі невеликі, потребують перебудови та розгортання всього додатку. З часом стає складніше зберігати хорошу модульну структуру, зміни логіки одного модуля мають тенденцію впливати на код інших модулів. Монолітні програми також може бути важко масштабувати, коли різні модулі мають конфліктні вимоги до ресурсів. Наприклад, один модуль може реалізовувати логіку обробки зображень з інтенсивним використанням процесору. Інший модуль може бути вимогливим до використання оперативної пам’яті. Однак, оскільки ці модулі розгортаються разом, вам доведеться йти на компроміс з вибором апаратного забезпечення. Масштабувати доводиться всей додаток, навіть якщо це потрібно тільки для одного модуля цього додатку.

Мікросервіси на противагу — кожен мікросервіс розгортається окремо. Тож якщо ви змінюєте щось в одному з них, ви можете розгорнути ці зміни, не чіпаючи інших мікросервісів, які можуть продовжувати працювати.

Тому веб проекти частіше розробляються як окремі сервіси що можна розгортати та масштабувати окремо.

Але мікросервісна архітектура постійно піддається критиці с самого момента її формування, серед нових проблем, котрі виникають при її впровадженні відзначаються:

- мережеві затримки: якщо в модулях, що виконують кілька функцій, взаємодія локально, то мікросервісная архітектура накладає вимогу атомізації модулів і взаємодії їх по мережі;

- формати повідомлень: відсутність стандартизації та необхідність узгодження форматів обміну фактично для кожної пари взаємодіючих мікросервісів призводить як до потенційних помилок, так і складнощів налагодження;
- баланс навантаження і відмовостійкості
- складність операційної підтримки — потрібні грамотні DevOps-інженери, безперервне розгортання і автоматичний моніторинг. Без всього цього мікросервіси використовувати не слід.
- тестування мікросервісів може бути громіздко. Використовуючи моноліт, нам потрібно тільки запустити додаток на сервері і переконатися в зв'язку з базою даних. А в мікросервісах, кожен окремий сервіс повинен бути запущений перед тим, як почати тестування.

Переваги та недоліки мікросервісів. До переваг можемо віднести:

- Сервіси запускається швидше, що робить розробників більш продуктивними та прискорює розгортання
- Кожен сервіс може бути розгорнутий незалежно від інших. Тож якщо ви змінюєте щось в одному з них, ви можете розгорнути ці зміни, не чіпаючи інших мікросервісів, які можуть продовжувати працювати.
- Кожен сервіс може бути масштабований окремо.
- Баг в одному мікросервісі не підірве роботу системи. Неполадки у мікросервісі не повинні зламати весь додаток. Швидше за все, вони не вплинуть суттєво на роботу додатку, особливо великого.
- Усуваються будь-які довгострокові зобов'язання щодо технології. При розробці нового сервісу ви можете вибрати новий стек технологій. Будь-який сервіс у системі можна замінити. Його можна переписати з нуля в межах прийнятного часу та бюджету без необхідності перебудовувати всю систему.

- Мікросервіси, як правило, краще організовані, оскільки кожен мікросервіс має дуже специфічну роботу і не займається роботою інших сервісів. Тож потенційно легші для розуміння, підтримки і тестування.
- Також відокремлені сервіси легше перекомпонувати і переналаштувати, щоб виконувати задачі різних додатків (наприклад, обслуговувати веб-клієнтів і публічний API).

Недоліки мікросервісів:

- Розробники повинні мати справу з додатковою складністю створення розподіленої системи.
- Складність розгортання. У виробництві існує також оперативна складність розгортання та управління системою, що складається з багатьох різних типів послуг.
- Коли ви будуєте нову архітектуру мікросервісу, ви, ймовірно, знайдете багато багатопланових проблем, яких ви не очікували під час розробки.

Мікросервіси як подолання складності.

Багато відомих компаній вирішили проблему моноліту, прийнявши архітектуру мікросервісів, замість того, щоб будувати єдиний монструозний моноліт. Серед них маємо Amazon, eBay, Walmart, Netflix, SoundCloud, Spotify, Twitter, Stripe, PayPal, Uber та Medium.

Впровадження Netflix було настільки успішним, що вони відкрили багато програмних засобів, з якими розроблялася їх архітектура мікросервісів. Сьогодні Netflix можна вважати першопрохідцем розвитку мікросервісів, і їхній підхід став об’єктом дослідження для багатьох інших компаній у всьому світі.

Створення складних додатків за своєю суттю є важко справою. Монолітна архітектура має сенс лише для простих, легких додатків. У кінцевому підсумку ви потрапите у світ болю, якщо ви будете використовувати його для складних програм. Мікросервісна архітектура,

незважаючи на недоліки та проблеми впровадження, є кращим вибором для складних продуктів що постійно розвиваються.

З власного досвіду роботи з мікросервісною архітектурою можу додати. Якщо моноліт розділити на окремі частини і не брати до уваги ключові принципи проектування архітектури, орієнтованої на мікросервіси, це може створити проблеми, а не елегантне рішення.

Мені як full stack розробнику, що цікавиться технологіями Docker та Kubernetes, розділяє принцип “Під кожну задачу свій інструмент”. Мікросервісна архітектура дуже імпонує.

1.3 Огляд фреймворків

Back-end - це серверна логіка веб-додатку. Тут реалізується робота зі сховищем даних, тому людина, що займається розробкою серверної логіки, повинена мати навички роботи з базами даних, а також досвід використання механізмів ORM. Саме від back-end розробника залежить продуктивність серверного коду, його масштабованість, безпека і раціональність.

Крім серверної логіки в сферу відповідальності back-end розробника входить налагодження та прототипування з використанням клієнтської частини програми. Це тягне за собою необхідність розуміння роботи стека протоколів TCP / IP, HTTP, REST / SOAP, принципів взаємодії браузера з веб-додатком.

Незважаючи на те, що сфера front-end традиційно вважається найбагатшою і різноманітною в плані технологій, back-end також має широкий спектр інструментів розробки. Крім канонічного PHP, міцні ніші зайняли Python з фреймворком Django, JavaScript і Node.js, Ruby та інші.

1.3.1 Django

Django - це програмний каркас з багатими можливостями, що підходить для розробки складних сайтів та веб-застосунків, написаний мовою програмування Python.

Django — фреймворк для веб-застосунків мовою Python. Один з основних принципів фреймворку - DRY (don't repeat yourself). Веб-системи на Django будуються з одного або кількох програм, які рекомендується робити відчужуваними та підключається. Це одна з помітних архітектурних відмінностей цього фреймворку від інших (наприклад, Ruby on Rails). Також, на відміну від багатьох інших фреймворків, обробники URL Django конфігуруються явно (за допомогою регулярних виразів), а не автоматично задаються зі структури контролерів.

Django проектувався для роботи під управлінням Apache (з модулем mod_python) та з використанням PostgreSQL як база даних. В даний час, окрім PostgreSQL, Django може працювати з іншими СУБД: MySQL (MariaDB), SQLite, Microsoft SQL Server, DB2, Firebird, SQL Anywhere та Oracle. Для роботи з базою даних Django використовує власний ORM, у якому модель даних описується класами Python, і генерується схема бази даних.

Архітектура Django схожа на «Модель-Уявлення-Контролер» (MVC). Контролер класичної моделі MVC приблизно відповідає рівню, який Django називається Представлення (View), а презентаційна логіка Представлення реалізується в Django рівнем Шаблонів (Templates). Через це рівневу архітектуру Django часто називають "Модель-Шаблон-Представлення" (MTV).

Спочатку розробка Django велася для забезпечення зручнішої роботи з ресурсами новин, що досить сильно позначилося на архітектурі: фреймворк надає ряд засобів, які допомагають у швидкій розробці веб-сайтів інформаційного характеру. Наприклад, розробнику не потрібно створювати контролери та сторінки для адміністративної частини сайту, у Django є

вбудований додаток для керування вмістом, який можна включити в будь-який сайт, зроблений на Django, і який може керувати відразу кількома сайтами на одному сервері. Адміністративна програма дозволяє створювати, змінювати та видаляти будь-які об'єкти наповнення сайту, протоколюючи всі вчинені дії, та надає інтерфейс для управління користувачами та групами (з пооб'єктним призначенням прав).

Веб-фреймворк Django використовується в таких великих та відомих сайтах, як Instagram, Disqus, Mozilla, The Washington Times, Pinterest, lamoda та інші.

Деякі можливості Django:

- ORM, API доступу до БД із підтримкою транзакцій
- вбудований інтерфейс адміністратора, з уже наявними перекладами на багато мов
- диспетчер URL на основі регулярних виразів
- розширювана система шаблонів з тегами та успадкуванням
- система кешування
- інтернаціоналізація
- архітектура, що підключається додатків, які можна встановлювати на будь-які Django-сайти
- «generic views» - шаблони функцій контролерів
- авторизація та аутентифікація, підключення зовнішніх модулів аутентифікації: LDAP, OpenID та інші.
- система фільтрів («middleware») для побудови додаткових обробників запитів, як, наприклад, включені в дистрибутив фільтри для кешування, стиснення, нормалізації URL та підтримки анонімних сесій
- бібліотека для роботи з формами (успадкування, побудова форм за існуючою моделлю БД)

- вбудована автоматична документація за тегами шаблонів та моделями даних, доступна через адміністративний додаток.

Деякі компоненти фреймворку між собою пов'язані слабко, тому їх можна просто замінювати на аналогічні. Але з деякими (наприклад, ORM) це зробити не дуже просто. Крім можливостей, вбудованих у ядро фреймворку, існують пакети, що розширюють його можливості.

На базі Django розроблено досить багато готових рішень, що розповсюджуються під вільною ліцензією, серед яких системи управління інтернет-магазинами, універсальні системи управління змістом, а також більш вузькоспрямовані проекти.

1.3.2 Ruby on Rails

Rails - це насамперед середовище розробки, яке чудово підходить для створення будь-якого типу веб-додатків: систем для управління веб-сайтами та платформ для ведення електронної торгівлі, програм для організації спільної роботи та для веб-сервісів для здійснення комунікації, для облікових та ERP -Систем, статистичних та аналітичних систем.

Ruby on Rails (RoR або Рейки) - це багаторівневий MVC-фреймворк для побудови веб-додатків, що використовують реляційні та NoSQL бази даних (наприклад, MySQL, MariaDB, PostgeSQL, MongoDB). Фреймворк написано мовою програмування Ruby. Rails підходить як для розробки звичайних сайтів, які мають бути реально швидкими, відмовостійкими та працюючими під високим навантаженням, так і для веб-додатків зі складною бізнес-логікою та динамічними web-інтерфейсами. Ruby on Rails є відкритим програмним забезпеченням та розповсюджується під ліцензією MIT.

Професійні розробники.

Варто відзначити той факт, що мовою програмування Ruby працюють в основному професіонали: поріг входження досить високий, тому програмісти в Ruby зазвичай приходять вже після декількох років роботи будь-якими

іншими мовами програмування (найчастіше зі світу PHP). Тому навіть початківець Ruby-програміст – це досвідчений веб-розробник з великим запасом знань та досвіду. Для мови Ruby найпопулярніший фреймворк – це Rails, понад 90% веб-додатків, написаних на Ruby, використовують саме Рейки.

Культура розробки на Ruby on Rails.

Основними принципами розробки на Rails є:

- Принцип DRY (Don't repeat yourself) – фреймворк надає механізми повторного використання програмного коду. Це дозволяє не лише мінімізувати дублювання коду, а й підвищити швидкість розробки.
- Принцип Convention over configuration — за замовчуванням у фреймворку використовуються численні конфігураційні угоди, типові для більшості додатків. Це дуже спрощує створення додатків, оскільки явна специфікація конфігурації потрібна лише у нестандартних випадках.
- Автоматизоване тестування - у складі RoR поставляються засоби для проведення повністю автоматичного модульного, інтеграційного та функціонального тестування, а ідеологія Ruby on Rails передбачає використання методів розробки через тестування (TDD - Test Driven Development). Все це робить розроблені програми реально надійними.

Розширюваність фреймворку Ruby on Rails.

Навколо Ruby on Rails склалася велика екосистема плагінів, що підключаються, з відкритим вихідним кодом («джемів», gems), які реалізують найбільш затребувані функції. «Джеми» бувають дуже різні: від низькорівневих, які відповідають за якийсь аспект внутрішньої роботи додатка, до високорівневих, що представляють собою окремі модулі для вирішення цілого спектру бізнес-завдань. Використання системи плагінів, що підключаються, багато в чому і спричинило високу популярність фреймворку — можливість вибірково підключати окремі компоненти та бібліотеки дуже

сильно прискорює розробку, а той факт, що використовувані розширення добре протестовані і налагоджуються роками, забезпечує надійність рішень, розроблених за допомогою такого підходу.

Міфи про мову Ruby та про фреймворк Ruby on Rails.

"Немає розробників". Міф. Розробники є. Звичайно, їх менше, ніж на PHP, але й середній рівень «на голову» вищий — дуже багато хто з тих, хто називає себе php-програмістом, насправді лише верстальники з поверхневими знаннями мови програмування, які не в змозі написати навіть саме простий веб-додаток. Якщо порівнювати Ruby з Java, то число розробників можна порівняти, а в порівнянні з .NET, Python і Perl - Ruby-розробників більше.

"Дуже дорого". Міф. Хороші веб-програмісти взагалі коштують дорого, незалежно від мови та платформи розробки. Рівень ЗП програміста на PHP і програміста на Ruby можна порівняти, якщо перший і другий може написати програму складніше «Hello, world!», працюють на фреймворках, знають ООП, парадигму MVC, а також мають досвід роботи у сфері понад 3 роки.

«Повільно» та «Немасштабуємо». Міфи. GitHub, Groupon, Basecamp, Twitter, Lenta.ru та ще багато проектів з багатотисячною відвідуваністю використовують Rails: працюють швидко, навантаження витримують і відмінно масштабуються.

1.3.3 NodeJS

Node.js - це open-source кроссплатформна виконавча середа для JavaScript, яка працює на серверах. З моменту випуску цієї платформи в 2009 році вона стала надзвичайно популярною і в наші дні грає дуже важливу роль в області веб-розробки. Якщо вважати показником популярності число зірок, які зібрав якийсь проект на GitHub, то Node.js, у якого більше 50000 зірок, це дуже і дуже популярний проект. Одним з ключових чинників, завдяки якому платформа Node.js стала настільки поширеною і популярною, є час її появи. Так, за кілька років до цього JavaScript почали вважати серйозним мовою.

Сталося це завдяки програмам Web 2.0, як Google Maps або Gmail, які продемонстрували світові можливості сучасних веб-технологій.

Платформа Node.js побудована на базі JavaScript двигуна V8 від Google, який використовується в браузері Google Chrome. Дана платформа, в основному, використовується для створення веб-серверів, проте сфера її застосування цим не обмежується.

Особливості Node.js:

- Швидкість. Однією з основних привабливих особливостей Node.js є швидкість. JavaScript-код, що виконується в середовищі Node.js, може бути в два рази швидше, ніж код, написаний на компільованих мовах, на зразок C або Java, і на порядки швидше інтерпретованих мов на зразок Python або Ruby.
- Простота. Платформа Node.js проста в освоєнні і використанні. Насправді, вона прямо-таки дуже проста, особливо це помітно в порівнянні з деякими іншими серверними платформами.
- JavaScript. У середовищі Node.js виконується код, написаний на JavaScript. Це означає, що мільйони фронтенд-розробників, які вже користуються JavaScript в браузері, можуть писати і серверний, і клієнтський код на одному і тому ж мовою програмування без необхідності вивчати абсолютно новий інструмент для переходу до серверної розробці.
- Двигун V8. В основі Node.js, крім інших рішень, лежить опенсорсний JavaScript-движок V8 від Google, який застосовується в браузері Google Chrome і в інших браузерах. Це означає, що Node.js користується напрацюваннями тисяч інженерів, які зробили середу виконання JavaScript Chrome неймовірно швидкої і продовжують працювати в напрямку вдосконалення V8.
- Асинхронність. У традиційних мовах програмування (C, Java, Python, PHP) всі інструкції, за замовчуванням, є блокуючими, якщо тільки розробник явно не подбає про асинхронність виконання коду. В результаті якщо, наприклад, в такому середовищі, зробити мережевий запит для завантаження

якогось JSON-коду, виконання потоку, з якого зроблено запит, буде призупинено до тих пір, поки не завершиться отримання і обробка відповіді. JavaScript значно спрощує написання асинхронного і неблокуючого коду з використанням єдиного потоку, функцій зворотного виклику (коллбеків) і підходу до розробки, заснованого на подіях. Кожен раз, коли нам потрібно виконати важку операцію, ми передаємо відповідним механізмом коллбек, який буде викликаний відразу після завершення цієї операції.

- Бібліотеки. Завдяки простоті і зручності роботи з менеджером пакетів для Node.js, який називається npm, екосистема Node.js прямо-таки процвітає. Зараз в реєстрі npm є понад півмільйона опенсорсний пакетів, які може вільно використовувати будь-який Node.js-розробник.

1.3.4 NestJS

Nest (NestJS) - це фреймворк для створення ефективних масштабованих Node.js серверних програм, створених з використанням TypeScript і повністю підтримуючи його.

NestJS залишається одним із улюблених фреймворків JavaScript для більшості backend-розробників. Він виводить масштабовані сервери Node.js на новий рівень. Nest.js багато в чому схожий на Angular.js, це не здасться дивним для тих, хто раніше працював над Angular.

Ключові особливості NestJS:

- Легко використовувати, вчитися та освоювати.
- Потужний інтерфейс командного рядка для підвищення продуктивності та спрощення розробки.
- Детальна та налагоджена документація.
- Відкритий вихідний код.
- Прості програми для модульного тестування.
- Створений для монолітів та мікросервісів.

На практиці, Node.js застосовують як веб-сервер через його подієво-орієнтовану модель і неблокуючу введення/виведення архітектури. Все це фундаментальні можливості, для яких у вашому проекті, потрібні надбудови вищого рівня, такі як фреймворк Nest.JS. Завдання фреймворку - зробити розробку простіше, швидше і структурованішою. Для Node.js є більше двох десятків популярних фреймворків, заснованих, як і Nest.JS, на стандартному Express.JS. Головна перевага Nest.JS у порівнянні з базовою платформою – чітка архітектура, що складається з трьох компонентів: контролерів, модулів та провайдерів. Завдяки їй додаток від початку розробки просто розділити на мікросервіси і працювати над кожним окремо з'єднуючи в єдину систему.

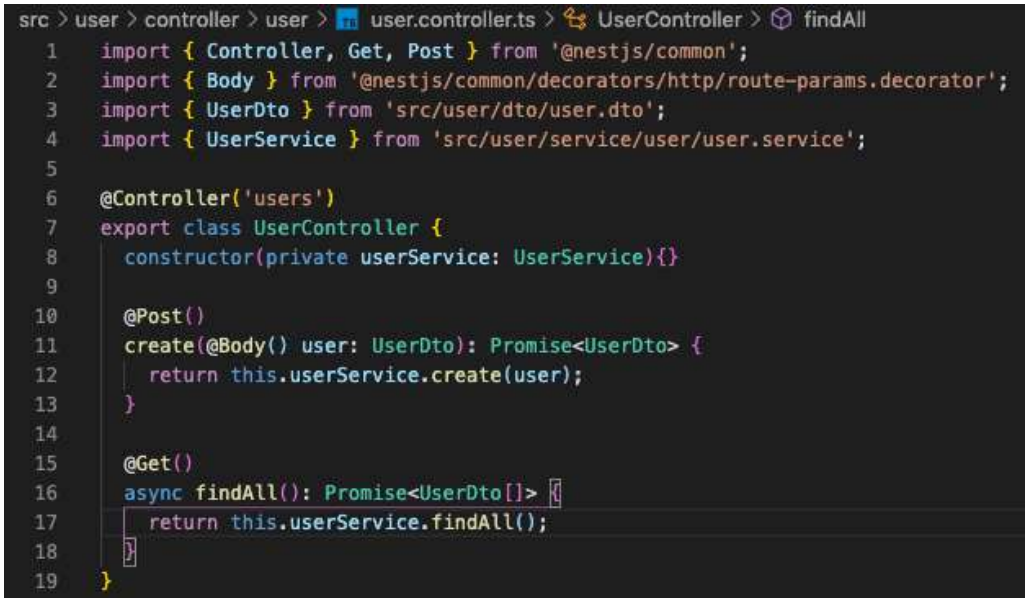

Рис 1.2 - Приклад контролера в Nest.JS

3 причини вибору NestJS для мого проекту:

- Універсальність та розширюваність. Nest.js – дає розробникам максимум свободи у використанні додаткових модулів. Він забезпечує високий рівень абстракції, який дозволяє використовувати API інших фреймворків, бібліотек та іншого, збираючи з модулів унікальну серверну програму будь-якого типу. Nest має відкритий вихідний код і практично безмежні можливості

масштабування. Зокрема вже є модулі для підключення баз даних PostgreSQL, MongoDB, MySQL та інтеграції технологій Caching, Mongoose, GraphQL, WebSockets та багатьох інших.

- Міцна основа та найкраще з нового. Nest.js побудований на принципах Express і будь-яке доповнення для цього фреймворку можна використовувати і в Нест, або взагалі забути про цю можливість, якщо ці модулі вам не потрібні. Nest.js - готовий каркас MVC-програми з коробки, написаний на TypeScript і підтримує JavaScript, а також безліч рішень для них. При цьому він не обмежується стандартними функціями, і дозволяє підключати всі найактуальніші JavaScript-рішення. Крім того, програми на Nest.js дуже просто тестувати, адже при всьому різноманітті можливостей система змушує використовувати строгу архітектуру, як у Angular. Вона ж відповідає за те, що ви не зіткнетеся з величезними витратами ресурсів на масштабування програми, коли це знадобиться - кожен мікросервіс можна доопрацьовувати окремо, не зупиняючи всю систему.
- Перспективи Зараз NestJS — фреймворк із найшвидшим зростанням популярності серед розроблених для NodeJS на TypeScript. Він подобається розробникам за можливість створювати програми з незвичайними функціями та втілювати оригінальні ідеї. Для нього вже написано багато модулів та прикладів розв'язання задач, які є у відкритому доступі та можуть стати у нагоді у вашому проекті. Така адаптивна екосистема і масштабованість - причини чому ви можете вибрати Nest для свого проекту, особливо якщо це стартап або програма з нестандартною бізнес-логікою.

NestJs – чудовий фреймворк для створення надійних API-інтерфейсів Node.js. Він надає масу функціональних можливостей, і якщо ви хочете зробити щось особливе, їхня документація просто чудова.

1.4 Висновок по розділу

Тема використання мікросервісної архітектури є дуже актуальною у наш час і користується великим попитом. Адже це має багато переваг, такі як:

- немає прив'язки до конкретних мов та технологій;
- проста інтеграція зі сторонніми рішеннями та можливість повторного використання;
- практично нескінченна масштабованість;
- простота обслуговування (управління серверною частиною в монолітному ПЗ завжди було у відповідальності власника ПЗ, а у випадку мікросервісів все лежить у компетенції хмарного провайдера);
- відмовостійкість;
- спрощена симетрична архітектура застосування замість ієрархічної (що властиво для монолітних продуктів) з одноранговими залежностями між компонентами;
- внесення правок без ризиків "обрушити" всю систему.

РОЗДІЛ 2
ЗАСОБИ ТА МЕТОДОЛОГІЇ РОЗРОБКИ ДОДАТКУ

2.1 Клієнт-серверна архітектура

2.1.1 Огляд поняття

Архітектура клієнтського сервера – це обчислювальна модель, в якій сервер розміщує, постачає та керує більшістю ресурсів і послуг, які споживає клієнт. Цей тип архітектури має один або кілька клієнтських комп'ютерів, під'єднаних до центрального сервера через мережу або Інтернет. Ця система спільно використовує обчислювальні ресурси. Архітектура клієнт/сервер також відома як модель мережевих обчислень або мережа клієнт/сервер, оскільки всі запити та послуги доставляються через мережу.

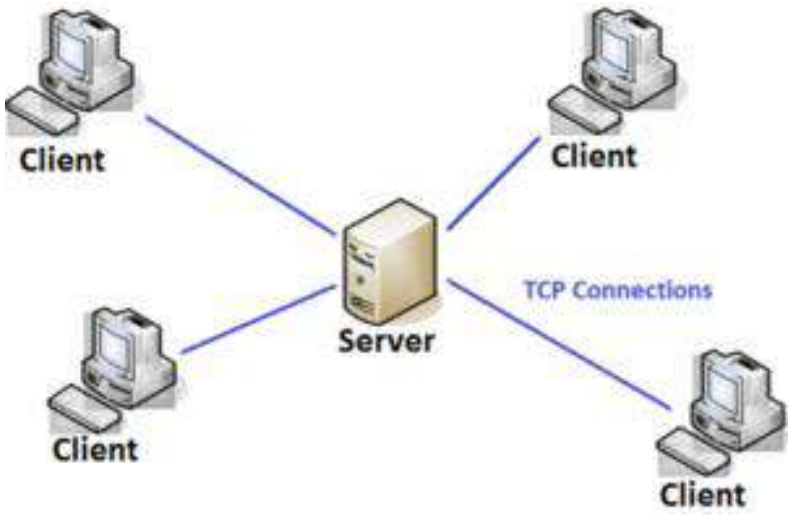

Рис. 2.1 - Схема роботи Веб-Сервера

Архітектура клієнт-сервер — це архітектура комп'ютерної мережі, в якій багато клієнтів (віддалені процесори) запитують і отримують послугу від централізованого сервера (хост комп'ютера). Клієнтські комп'ютери забезпечують інтерфейс, який дозволяє користувачеві комп'ютера запитувати

послуги сервера та відображати результати, які повертає сервер. Сервери чекають надходження запитів від клієнтів, а потім відповідають на них. В ідеалі сервер надає клієнтам стандартизований прозорий інтерфейс, щоб клієнтам не потрібно було знати про особливості системи (тобто апаратного та програмного забезпечення), яка надає послугу. Клієнти часто знаходяться на робочих станціях або на персональних комп'ютерах, тоді як сервери розташовані в інших місцях мережі, як правило, на більш потужних машинах. Ця обчислювальна модель особливо ефективна, коли клієнти і сервер мають різні завдання, які вони регулярно виконують. При обробці лікарняних даних, наприклад, на клієнтському комп'ютері може бути запущена прикладна програма для введення інформації про пацієнта, тоді як на серверному комп'ютері запущена інша програма, яка керує базою даних, в якій постійно зберігається інформація. Багато клієнтів можуть отримувати доступ до інформації сервера одночасно, і в той же час клієнтський комп'ютер може виконувати інші завдання, наприклад надсилати електронну пошту. Оскільки і клієнтські, і серверні комп'ютери вважаються інтелектуальними пристроями, модель клієнт-сервер повністю відрізняється від старої моделі «мейнфрейма», в якій централізований мейнфрейм виконував усі завдання для пов'язаних з ним «немих» терміналів.

Призначення архітектури клієнта/сервера.

Ми знаходимося в епоху, коли інформаційні технології відіграють важливу роль у бізнес-додатках, розглядаючись як область, яку організація має дуже інвестувати, щоб розширити можливості, доступні для конкуренції на світовому ринку. «Конкурентоспроможна глобальна економіка забезпечить застарілість і неясність для тих, хто не може або не хоче конкурувати» (Client/Server Architecture, 2011), згідно з цією заявою, організаціям необхідно підтримувати свої ринкові позиції шляхом реінжинірингу переважаючих організаційних структур і ділової практики, щоб досягти своїх бізнес-цілей. Коротше кажучи, це основна потреба розвиватися зі зміною технологічних

аспектів. Тому організації повинні пройти механізм отримання та обробки своїх корпоративних даних, щоб зробити бізнес-процедури більш ефективними, щоб досягти успіху або вижити на світовому ринку. Модель клієнт/сервер відкриває логічну перспективу розподіленої корпоративної обробки, де сервер обробляє та обробляє всі клієнтські запити. Це також можна розглядати як революційну віху в галузі обробки даних. «Обчислення клієнт/сервер є найефективнішим джерелом інструментів, які надають співробітникам повноваження та відповідальність». (Архітектура клієнта/сервера, 2011 р.) бізнес є силами, що створюють потребу в клієнт-серверних обчисленнях». (Архітектура клієнта/сервера, 2011 р.) Комп'ютерні обчислення клієнт/сервер мають величезний прогрес у комп'ютерній індустрії, залишаючи недоторканими будь-яку область чи куток. Часто гібридні навички необхідні для розробки програм клієнт-сервер, включаючи проектування баз даних, обробку транзакцій, комунікаційні навички, дизайн і розробку графічного інтерфейсу користувача тощо. Розширені програми вимагають досвіду розподілених об'єктів та компонентних інфраструктур. Найбільш поширеною сьогодні стратегією клієнт/сервер є реалізація локальної мережі ПК, оптимізована для використання груп/пакетів. Це в основному дало поріг для багатьох нових розподілених підприємств, оскільки усуває обчислення, орієнтовані на хост.

Характеристики архітектури клієнт-сервер:

- Клієнтські та серверні машини потребують різної кількості апаратних та програмних ресурсів.
- Клієнтські та серверні машини можуть належати різним постачальникам.
- Горизонтальна масштабованість (збільшення клієнтських машин) і вертикальна масштабованість (міграція на більш потужний сервер або на багатосерверне рішення)

- Клієнтська або серверна програма безпосередньо взаємодіє з протоколом транспортного рівня, щоб встановити зв’язок та надіслати чи отримати інформацію.
- Потім транспортний протокол використовує протоколи нижнього рівня для надсилання або отримання окремих повідомлень. Таким чином, комп’ютеру потрібен повний набір протоколів для роботи клієнта або сервера.
- Один комп’ютер серверного класу може надавати кілька послуг одночасно; для кожної служби потрібна окрема серверна програма.

Трирівнева архітектура клієнт-сервер.

Традиційна архітектура клієнт/сервер включає два рівні, рівень клієнта і рівень сервера. Інший поширений дизайн систем клієнт/сервер використовує три рівні:

- Клієнт, який взаємодіє з користувачем.
- Сервер додатків, який містить бізнес-логіку програми.
- Менеджер ресурсів, який зберігає дані.

Client-Server та Peer-to-Peer.

Основна відмінність між системами клієнт-сервер і одноранговими системами полягає в тому, що в архітектурі клієнт-сервер є призначені клієнти, які запитують послуги, і сервери, які надають послуги, але в однорангових системах однорангові партнери діють і як постачальники послуг, і як послуги. споживачів. Крім того, системи клієнт-сервер вимагають центрального файлового сервера, і їх впровадження дорожче, ніж однорангові системи. З іншого боку, у системі клієнт-сервер виділений файловий сервер забезпечує рівень доступу до клієнтів, забезпечуючи кращу безпеку, ніж однорангові системи, де безпекою займаються кінцеві користувачі. Крім того, однорангові мережі погіршують продуктивність зі збільшенням кількості вузлів, але системи клієнт-сервер стабілізуються і їх можна масштабувати

настільки, наскільки вам потрібно. Тому вибір одного над іншим залежить від середовища, яке вам потрібно впровадити.

На рисунку 3 зображена модель архітектури клієнт-сервер у двох різних режимах або типах. Існують різні типи архітектур, засновані на моделі клієнт-сервер, а саме. розподілений, рівноправний тощо.

- У розподіленій архітектурі одна або кілька виділених машин використовуються лише як сервер, а всі інші машини використовуються як клієнти. У цьому випадку клієнти можуть спілкуватися через сервер.
- У цьому режимі клієнт ініціює спілкування.
- Клієнт надсилає запит на сервер.
- Сервер відповідає або виконує певну послугу.
- У однорангової архітектурі кожен хост або екземпляр прикладної програми може функціонувати одночасно як клієнт і сервер. Обидва мають еквівалентні обов’язки та статус.
- У цьому режимі будь-який учасник може ініціювати спілкування.
- Будь-який пристрій може генерувати запит.
- Будь-який пристрій може дати відповідь.

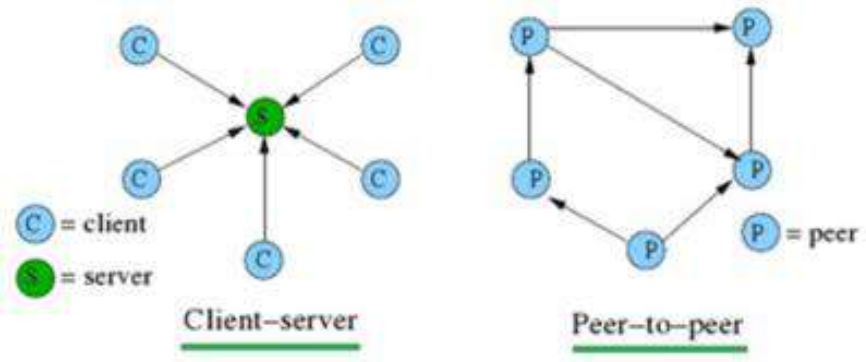

Рис. 2.2 - Візуалізація Client-Server та Peer-to-peer.

Переваги та недоліки архітектури клієнт-сервер.

Переваги: організації часто шукають можливості підтримувати послуги та якісну конкуренцію, щоб підтримувати свої позиції на ринку за допомогою

технологій, де модель клієнт/сервер робить ефективний вплив. Розгортання клієнт-серверних обчислень в організації позитивно підвищить продуктивність за рахунок використання економічно ефективних користувацьких інтерфейсів, розширеного зберігання даних, широких можливостей підключення та надійних додатків. При правильному застосуванні він здатний покращити організаційну поведінку за допомогою досвідченого працівника, який може маніпулювати даними та належним чином реагувати на помилки.

Покращений обмін даними: дані зберігаються звичайними бізнес-процесами та маніпулюються на сервері, доступні призначеним користувачам (клієнтам) через авторизований доступ. Використання мови структурованих запитів (SQL) підтримує відкритий доступ з усіх аспектів клієнта, а також прозорість мережевих служб показує, що подібні дані поширюються між користувачами.

Інтеграція сервісів: кожному клієнту надається можливість доступу до корпоративної інформації через інтерфейс робочого столу, що виключає необхідність входити в термінальний режим або інший процесор. Настільні інструменти, такі як електронні таблиці, презентації Power Point тощо, можна використовувати для роботи з корпоративними даними за допомогою баз даних і серверів додатків, що знаходяться в мережі, для отримання значущої інформації.

Спільні ресурси для різних платформ: програми, які використовуються для моделі клієнт/сервер, створюються незалежно від апаратної платформи або технічного досвіду відповідного програмного забезпечення (Operating System S/W), що забезпечує відкрите обчислювальне середовище, змушуючи користувачів отримувати послуги клієнтів і серверів. (база даних, додаток, сервери зв'язку).

Взаємодія з даними: усі інструменти розробки, які використовуються для клієнт-серверних програм, отримують доступ до серверного сервера баз

даних через SQL, стандартну мову визначення даних і доступу, корисну для послідовного керування корпоративними даними. Розширені продукти баз даних дозволяють користувачеві/додатку отримати об’єднане уявлення про корпоративні дані, розподілені на кількох платформах. Замість єдиної цільової платформи це забезпечує цілісність бази даних із можливістю виконувати оновлення в кількох місцях, забезпечуючи якісне ознайомлення та відновлення.

Можливість обробки даних, незважаючи на місце розташування: ми перебуваємо в епосі, яка зазнає трансформації машиноцентрованих систем у системи, орієнтовані на користувача. Машинно-центровані системи, такі як мейнфрейм, міні-мікропрограми, мали унікальні платформи доступу та функціональні ключі, параметри навігації, продуктивність та безпеку було видно. Через клієнт/сервер користувачі можуть безпосередньо входити в систему, незважаючи на розташування або технологію процесорів.

Простота обслуговування: оскільки архітектура клієнт/сервер є розподіленою моделлю, яка представляє розподілені обов’язки між незалежними комп’ютерами, інтегрованими в мережу, це є перевагою з точки зору обслуговування. Сервер легко замінити, відремонтувати, оновити та перемістити, поки клієнти залишаються без змін. Це неусвідомлення змін називається інкапсуляцією.

Безпека: сервери мають кращий контроль доступу та ресурсів, щоб гарантувати, що лише авторизовані клієнти можуть отримати доступ до даних або маніпулювати ними, а оновлення сервера ефективно адмініструються.

Недоліки (у порівнянні з одноранговими мережами).

Перевантажені сервери: коли є часті одночасні запити клієнта, сервери сильно перевантажуються, утворюючи затори. Але в мережі P2P додавання більшої кількості вузлів збільшить її пропускну здатність, оскільки вона розраховується як сума пропускної здатності кожного вузла в мережі.

Вплив централізованої архітектури: оскільки вона централізована, якщо критичний сервер виходить з ладу, клієнтські запити не виконуються. Тому клієнт/сервер не має надійності хорошої мережі P2P (ресурси розподілені між багатьма вузлами).

2.1.2. REST

REST, або REpresentational State Transfer — це архітектурний стиль для забезпечення стандартів між комп'ютерними системами в Інтернеті, що полегшує взаємодію систем між собою. REST-сумісні системи, які часто називають системами RESTful, характеризуються тим, що вони не мають стану та відокремлюють проблеми клієнта і сервера. Ми розглянемо, що означають ці терміни та чому вони є корисними характеристиками для служб в Інтернеті.

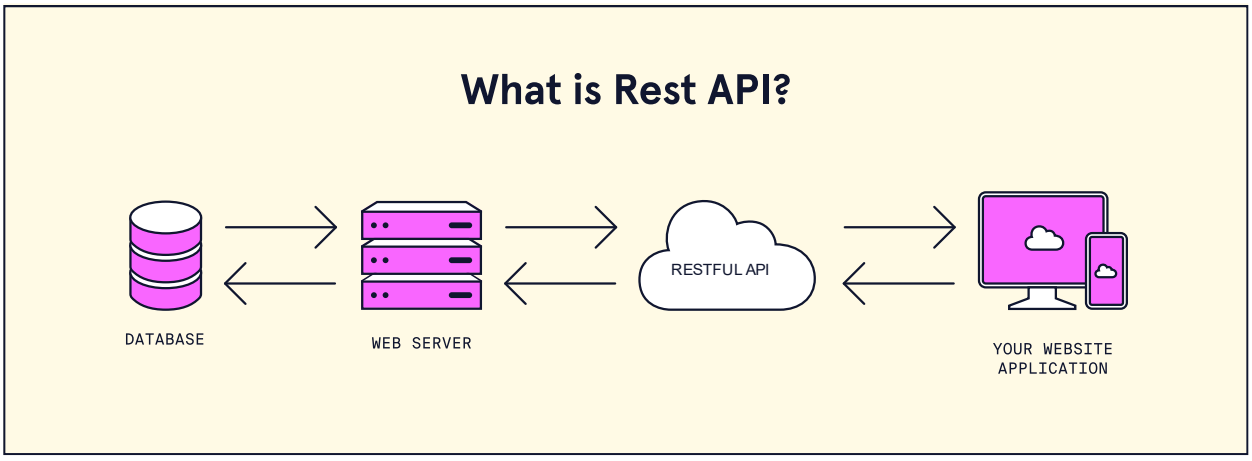

Рис. 2.3. Схема взаємодії Restful API.

Розділення клієнта і сервера.

В архітектурному стилі REST реалізацію клієнта та реалізацію сервера можна виконувати незалежно один від одного, не знаючи один про одного. Це означає, що код на стороні клієнта можна змінити в будь-який час, не впливаючи на роботу сервера, а код на стороні сервера можна змінити, не впливаючи на роботу клієнта.

Поки кожна сторона знає, який формат повідомлень надсилати іншій, їх можна зберігати модульними та окремими. Відокремлюючи проблеми інтерфейсу користувача від проблем зберігання даних, ми покращуємо гнучкість інтерфейсу на різних платформах і покращуємо масштабованість, спрощуючи компоненти сервера. Крім того, поділ дозволяє кожному компоненту розвиватися незалежно.

Використовуючи інтерфейс REST, різні клієнти звертаються до тих самих кінцевих точок REST, виконують однакові дії та отримують однакові відповіді.

Безстановий.

Системи, які дотримуються парадигми REST, не мають стану, що означає, що серверу не потрібно нічого знати про те, в якому стані знаходиться клієнт, і навпаки. Таким чином, і сервер, і клієнт можуть зрозуміти будь-яке отримане повідомлення, навіть не переглядаючи попередні повідомлення. Це обмеження без громадянства забезпечується використанням ресурсів, а не команд. Ресурси — це іменники Інтернету — вони описують будь-який об’єкт, документ чи річ, які вам може знадобитися зберігати чи надсилати іншим службам.

Оскільки системи REST взаємодіють за допомогою стандартних операцій над ресурсами, вони не покладаються на реалізацію інтерфейсів.

Ці обмеження допомагають додаткам RESTful досягти надійності, швидкої продуктивності та масштабованості як компонентів, якими можна керувати, оновлювати та використовувати їх повторно, не впливаючи на систему в цілому, навіть під час роботи системи.

Зв'язок між Клієнтом і Сервером.

В архітектурі REST клієнти надсилають запити на отримання або модифікацію ресурсів, а сервери надсилають відповіді на ці запити. Давайте подивимося на стандартні способи подання запитів і надсилання відповідей.

Оформлення запитів.

REST вимагає, щоб клієнт зробив запит до сервера, щоб отримати або змінити дані на сервері. Запит зазвичай складається з:

- метод HTTP, яке визначає, яку операцію виконувати;
- заголовок, який дозволяє клієнту передавати інформацію про запит;
- шлях до ресурсу;
- необов'язкове тіло повідомлення, що містить дані;

Методи HTTP.

Є 4 основних методи HTTP, які ми використовуємо в запитах для взаємодії з ресурсами в системі REST:

- GET — отримати певний ресурс (за ідентифікатором) або набір ресурсів;
- POST — створити новий ресурс;
- PUT — оновити певний ресурс (за ідентифікатором);
- DELETE — видалити певний ресурс за ідентифікатором;

2.1.3 GraphQL

GraphQL — це мова запитів і середовище виконання на стороні сервера для інтерфейсів програмування прикладних програм (API), що надає клієнтам саме ті дані, які вони запитують, і не більше того.

GraphQL розроблено, щоб зробити API швидкими, гнучкими та зручними для розробників. Його навіть можна розгорнути в інтегрованому середовищі розробки (IDE), відомому як GraphiQL. Як альтернатива REST, GraphQL дозволяє розробникам створювати запити, які витягують дані з кількох джерел даних за один виклик API.

Крім того, GraphQL дає розпорядникам API гнучкість додавати або забороняти поля, не впливаючи на наявні запити. Розробники можуть створювати API за допомогою будь-яких методів, які вони віддають перевагу, а специфікація GraphQL забезпечить їх роботу у передбачуваний спосіб для клієнтів.

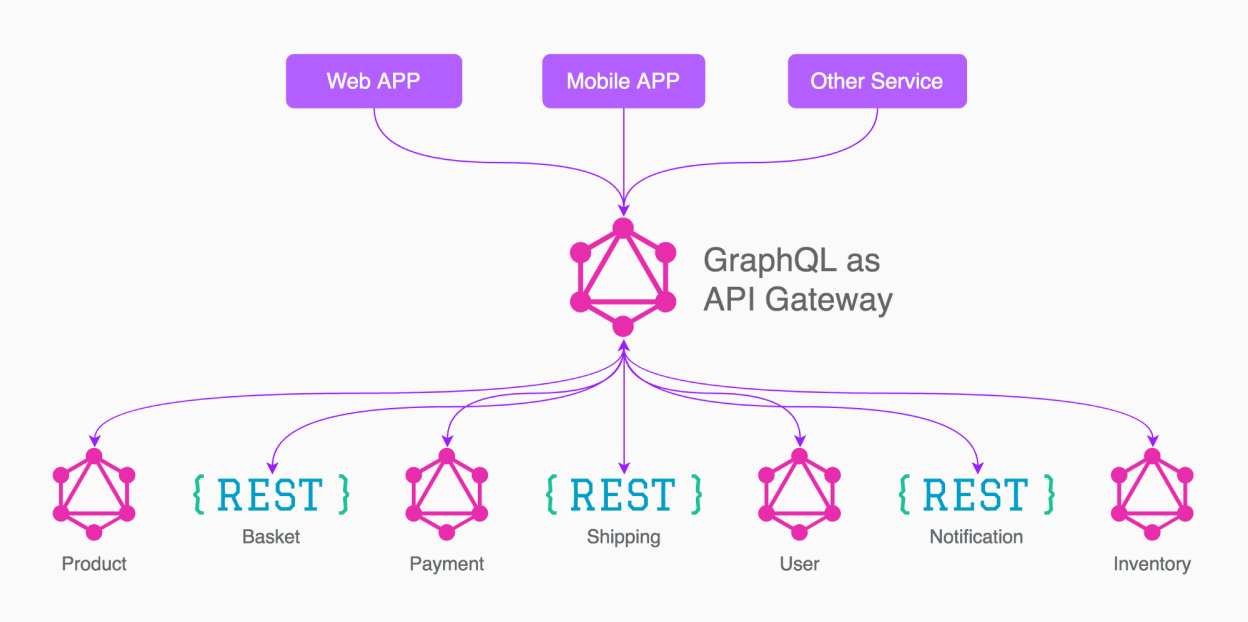

Рис. 2.4 - Візуалізація роботи GraphQL.

Схеми, резольвери та інші поширені терміни GraphQL.

Розробники API використовують GraphQL для створення схеми для опису всіх можливих даних, які клієнти можуть запитувати через цю службу.

Схема GraphQL складається з типів об'єктів, які визначають, який тип об'єкта ви можете запитати та які поля він має.

Коли надходять запити, GraphQL перевіряє запити на відповідність схемі. Потім GraphQL виконує перевірені запити.

Розробник API приєднує кожне поле в схемі до функції, яка називається резольвером. Під час виконання викликається резольвер для отримання значення.

Окрім визначення та перевірки синтаксису для запитів API (викладеного в репозиторії graphql-spec), GraphQL залишає більшість інших рішень на розробку API. GraphQL не надає жодних вказівок щодо того, як зберігати дані або яку мову програмування використовувати — розробники можуть використовувати PHP (graphql-php), Scala (Sangria), Python (Graphene Python), Ruby (graphql-ruby), JavaScript (graphql. js) та багато іншого. GraphQL не пропонує жодних вимог до мережі, авторизації або розбиття на сторінки.

З точки зору клієнта, найпоширенішими операціями GraphQL, ймовірно, є запити та мутації. Якби ми думали про них з точки зору моделі створення, читання, оновлення та видалення (CRUD), запит був би еквівалентним прочитанню. Усі інші (створення, оновлення та видалення) обробляються мутаціями.

Переваги та недоліки GraphQL у корпоративних середовищах.

Переваги:

- Схема GraphQL встановлює єдине джерело істини в додатку GraphQL. Він пропонує організації спосіб об’єднати весь свій API.
- Виклики GraphQL обробляються за один цикл. Клієнти отримують те, що вимагають, без переплати.
- Чітко визначені типи даних зменшують непорозуміння між клієнтом і сервером.
- GraphQL є інтроспективним. Клієнт може запросити список доступних типів даних. Це ідеально підходить для автоматичного створення документації.
- GraphQL дозволяє програмному API розвиватися без порушення існуючих запитів.
- Доступно багато розширень GraphQL з відкритим кодом, які пропонують функції, недоступні в REST API.
- GraphQL не визначає конкретну архітектуру програми. Він може бути введений поверх існуючого API REST і може працювати з існуючими інструментами керування API.

Недоліки:

- GraphQL представляє криву навчання для розробників, знайомих з REST API.
- GraphQL перекладає більшу частину роботи запиту даних на серверну сторону, що додає складності для розробників серверів.

- Залежно від того, як це реалізовано, GraphQL може вимагати інших стратегій керування API, ніж REST API, особливо якщо враховувати обмеження швидкості та ціни.
- Кешування є більш складним, ніж з REST.
- Супроводжувачі API мають додаткове завдання написати схему GraphQL, яка підтримується.

GraphQL і відкритий код.

GraphQL був розроблений компанією Facebook, яка вперше почала використовувати його для мобільних додатків у 2012 році. Специфікація GraphQL була відкрита у 2015 році. Зараз вона контролюється GraphQL Foundation.

Існує безліч проектів з відкритим кодом, які включають GraphQL. Наведений нижче список не є вичерпним, але включає проекти, розроблені для полегшення прийняття GraphQL:

- Apollo, платформа GraphQL, яка включає в себе клієнтську бібліотеку (Apollo Client) і серверну базу (Apollo Server).
- Offix, автономний клієнт, який дозволяє виконувати мутації та запити GraphQL, навіть коли програма недоступна.
- Graphback, клієнт командного рядка для створення серверів Node.js з підтримкою GraphQL.
- OpenAPI-to-GraphQL, інтерфейс командного рядка та бібліотека для перекладу API, описаних у специфікаціях OpenAPI або Swagger, у GraphQL.

2.2 Веб-сервер

Веб-сервер — це програмне та апаратне забезпечення, яке використовує HTTP (Hypertext Transfer Protocol) та інші протоколи для відповіді на

клієнтські запити, зроблені через всесвітню мережу. Основним завданням веб-сервера є відображення вмісту веб-сайту шляхом зберігання, обробки та доставки веб-сторінок користувачам. Крім HTTP, веб-сервери також підтримують SMTP (Протокол передачі пошти) і FTP (Протокол передачі файлів), які використовуються для електронної пошти, передачі файлів і зберігання.

Апаратне забезпечення веб-сервера підключено до Інтернету і дозволяє обмінюватися даними з іншими підключеними пристроями, тоді як програмне забезпечення веб-сервера контролює, як користувач отримує доступ до розміщених файлів. Процес веб-сервера є прикладом моделі клієнт/сервер. Усі комп'ютери, на яких розміщуються веб-сайти, повинні мати програмне забезпечення веб-сервера.

Веб-сервери використовуються у веб-хостингу або розміщенні даних для веб-сайтів і веб-додатків - або веб-програм.

Як працюють веб-сервери?

Доступ до програмного забезпечення веб-сервера здійснюється через доменні імена веб-сайтів і забезпечує доставку вмісту сайту користувачеві, який запитує. Програмна сторона також складається з кількох компонентів, принаймні HTTP-сервер. HTTP-сервер може розуміти HTTP та URL-адреси. Як апаратне забезпечення, веб-сервер — це комп'ютер, на якому зберігається програмне забезпечення веб-сервера та інші файли, пов'язані з веб-сайтом, такі як документи HTML, зображення та файли JavaScript.

Коли веб-переглядачу, наприклад Google Chrome або Firefox, потрібен файл, розміщений на веб-сервері, браузер запитує файл за допомогою HTTP. Коли запит отримує веб-сервер, HTTP-сервер прийме запит, знайде вміст і надішле його назад до браузера через HTTP.

Точніше, коли браузер запитує сторінку з веб-сервера, процес буде виконуватися в ряді кроків. Спочатку людина вказує URL-адресу в адресному рядку веб-браузера. Потім веб-браузер отримає IP-адресу доменного імені -

або переклавши URL-адресу через DNS (систему доменних імен), або здійснивши пошук у його кеші. Це перенесе браузер на веб-сервер. Потім браузер запитає певний файл із веб-сервера за допомогою HTTP-запиту. Веб-сервер відповість, надіславши браузеру запитувану сторінку, знову ж таки, через HTTP. Якщо запитувана сторінка не існує або щось піде не так, веб-сервер відповість повідомленням про помилку. Після цього браузер зможе відобразити веб-сторінку.

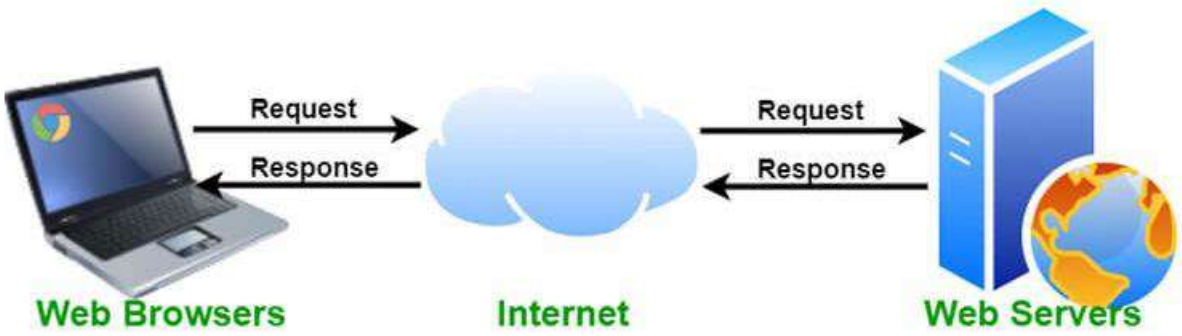

Рис. 2.5. Візуалізація клієнт-сервер.

Приклади використання веб-сервера.

Веб-сервери часто є частиною більшого пакету програм, пов'язаних з Інтернетом та інтранет, які використовуються для:

- відправка та отримання електронних листів;
- завантаження запитів на файли протоколу передачі файлів (FTP);
- створення та публікація веб-сторінок.

Багато базових веб-серверів також підтримують сценарії на стороні сервера, які використовуються для використання сценаріїв на веб-сервері, які можуть налаштувати відповідь клієнту. Скрипти на стороні сервера виконуються на серверній машині і зазвичай мають широкий набір функцій, що включає доступ до бази даних. Процес створення сценаріїв на стороні сервера також використовуватиме Active Server Pages (ASP), Hypertext

Preprocessor (PHP) та інші мови сценаріїв. Цей процес також дозволяє динамічно створювати документи HTML.

Динамічні та статичні веб-сервери.

Веб-сервер можна використовувати для обслуговування статичного або динамічного вмісту. Статичний — це вміст, який відображається як є, тоді як динамічний вміст можна оновлювати та змінювати. Статичний веб-сервер складатиметься з комп'ютера та програмного забезпечення HTTP. Він вважається статичним, оскільки сервер надсилатиме розміщені файли до браузера як є.

Динамічні веб-браузери складатимуться з веб-сервера та іншого програмного забезпечення, такого як сервер додатків і база даних. Він вважається динамічним, оскільки сервер додатків можна використовувати для оновлення будь-яких розміщених файлів перед їх відправкою у браузер. Веб-сервер може генерувати вміст, коли його запитують з бази даних. Хоча цей процес є більш гнучким, він також є більш складним.

Поширене та найкраще програмне забезпечення для веб-серверів на ринку.

Існує ряд поширених веб-серверів, серед яких:

- HTTP-сервер Apache. Розроблений Apache Software Foundation, це безкоштовний веб-сервер з відкритим вихідним кодом для Windows, Mac OS X, Unix, Linux, Solaris та інших операційних систем; для цього потрібна ліцензія Apache.
- Microsoft Internet Information Services (IIS). Розроблено Microsoft для платформ Microsoft; він не є відкритим, але широко використовується.
- Nginx. Популярний веб-сервер з відкритим вихідним кодом для адміністраторів через його легке використання ресурсів і масштабованість. Завдяки своїй архітектурі, що керується подіями, він може обробляти багато одночасних сеансів. Nginx також можна використовувати як проксі-сервер і балансувальник навантаження.

- Lighttpd. Безкоштовний веб-сервер із операційною системою FreeBSD. Він вважається швидким і безпечним, але споживає менше енергії ЦП.
- Веб-сервер Sun Java System. Безкоштовний веб-сервер від Sun Microsystems, який може працювати на Windows, Linux та Unix. Він добре обладнаний для роботи із середніми та великими веб-сайтами.

Провідні веб-сервери включають Apache, Microsoft's Internet Information Services (IIS) і Nginx - виражений механізм X. Інші веб-сервери включають сервер NetWare від Novell, веб-сервер Google (GWS) і сімейство серверів IBM Domino.

При виборі веб-сервера слід враховувати, наскільки добре він працює з операційною системою та іншими серверами; його здатність обробляти програмування на стороні сервера; характеристики безпеки; а також інструменти для видавництва, пошукової системи та створення сайтів, які постачаються з ним. Веб-сервери також можуть мати різні конфігурації та встановлені значення за замовчуванням. Для створення високої продуктивності допоможе веб-сервер, висока пропускна здатність і низька затримка.

Практика безпеки веб-сервера.

Існує багато методів безпеки, які люди можуть встановити для використання веб-сервера, які можуть зробити безпечнішим. Кілька прикладів практик безпеки можуть включати такі процеси, як:

- зворотний проксі, який призначений для приховування внутрішнього сервера і виступає в якості посередника для трафіку, що надходить на внутрішньому сервері;
- обмеження доступу за допомогою таких процесів, як обмеження доступу веб-хосту до інфраструктурних машин або використання Secure Socket Shell (SSH);
- підтримувати веб-сервери виправленими та оновленими, щоб гарантувати, що веб-сервер не схильний до вразливостей;

- моніторинг мережі, щоб переконатися у відсутності будь-якої чи несанкціонованої діяльності;
- використання брандмауера та SSL як брандмауери може контролювати HTTP-трафік, а рівень безпечних сокетів (SSL) може допомогти зберегти дані в безпеці.

2.3 Вибір інструментарію

2.3.1 Frontend: Swagger

Swagger — це набір правил, специфікацій та інструментів з відкритим вихідним кодом для розробки та опису API RESTful. Фреймворк Swagger дозволяє розробникам створювати інтерактивну, машинну та зрозумілу документацію API.

Специфікації API зазвичай містять таку інформацію, як підтримувані операції, параметри та виходи, вимоги до авторизації, доступні кінцеві точки та необхідні ліцензії. Swagger може автоматично генерувати цю інформацію з вихідного коду, попросивши API повернути файл документації зі своїх анотацій.

Swagger допомагає користувачам створювати, документувати, тестувати та використовувати веб-сервіси RESTful. Його можна використовувати як з підходом до розробки API зверху вниз, так і знизу вгору. У методі «зверху вниз» або методом «дизайн-перш» Swagger можна використовувати для розробки API перед написанням будь-якого коду. У методі «знизу-вгору» або метод спочатку коду, Swagger бере код, написаний для API, і створює документацію.

Компоненти Swagger.

Swagger надає різноманітні інструменти з відкритим кодом для API, зокрема:

- Swagger Editor – це дає змогу розробникам писати документацію для нових API, а також редагувати наявні. Редактор на основі браузера візуально відображає специфікації OpenAPI, обробляє помилки та надає зворотній зв’язок у реальному часі.
- Swagger Codegen – це дає розробникам можливість генерувати код клієнтської бібліотеки та пакети SDK для різних платформ.
- Swagger User Interface – це повністю настроюваний інструмент, який допомагає інженерам створювати документацію для різних платформ. Його можна розмістити в будь-якому середовищі.
- Swagger Inspector – це інструмент для тестування документації API. API можна легко перевіряти без обмежень, а результати автоматично зберігаються та доступні в хмарі.

Переваги Swagger.

На додаток до своєї мети стандартизації та спрощення практики API, Swagger має кілька додаткових переваг:

- Він має дружній інтерфейс користувача, який відображає план API.
- Документація зрозуміла як для розробників, так і для нерозробників, таких як клієнти чи керівники проектів.
- Технічні характеристики читаються людиною та машиною.
- Створює інтерактивну документацію, яку можна легко перевірити.
- Підтримує створення бібліотек API більш ніж 40 мовами.
- Формат прийнятний у JSON та YAML, щоб легше редагувати.
- Допомагає автоматизувати процеси, пов’язані з API.

Що таке Swagger Editor?

Swagger Editor – це інструмент, який допомагає нам перевіряти наш дизайн API в режимі реального часу, він перевіряє дизайн на відповідність

специфікації OAS Open API і забезпечує візуальний зворотний зв’язок на льоту.

Інструмент редактора можна запускати будь-де, локально або в Інтернеті. Забезпечує миттєвий зворотній зв’язок щодо дизайну API, вказує, чи помилки не обробляються належним чином або є якісь проблеми з синтаксисом.

Він також має інтелектуальні функції автозаповнення, які дозволяють нам писати код швидше. Його легко налаштувати, а також дозволяє розробникам створювати заглушки сервера для API для швидшої розробки.

Завдяки таким інструментам, як Swagger Editor, розробники мають уявлення в реальному часі про те, як розвивається дизайн API. Отримавши миттєву відповідь від заглушок. Це також допомагає нам проаналізувати, як сторонній розробник буде взаємодіяти з API.

Що таке SwaggerHub?

SwaggerHub — це платформа для проектування та документації для розробки API з відкритим API.

Це полегшує керування різними командами та проектами, створюючи папки з різними API та рівнями дозволів для кращої організації. Інформацією можна поділитися з уповноваженим керівництвом бізнесу та зацікавленими сторонами. Це допомагає розробникам працювати разом із зацікавленими сторонами, вносити нові зміни, переглядати зміни, об’єднувати матеріали. Постійно спілкуйтеся та відстежуйте проблеми.

SwaggerHub пропонує загальні моделі дизайну, які можна зберігати у спеціалізованих магазинах, які називаються доменами, і на які можна посилатися та повторно використовувати в коді. Під час написання серверного коду наші API взаємодіють з кількома іншими службами на серверній частині. Замість того, щоб запускати такі екземпляри, SwaggerHub дозволяє нам висміювати ці API, що сприяє швидшій розробці.

Історія Swagger.

Проект Swagger API був створений у 2011 році Тоні Тамом під час розробки інструментів для веб-сайту словника Wordnik. Фреймворк був розроблений для полегшення автоматизації API та його документації. Потім проект був зроблений з відкритим кодом, де він набув популярності серед компаній і розробників.

У 2015 році компанія, яка підтримувала Swagger, SmartBear Software, допомогла заснувати ініціативу OpenAPI, організацію, яку спонсорує Linux Foundation. Через рік Swagger було перейменовано в специфікацію OpenAPI і переміщено в нове сховище GitHub.

Наразі Swagger є найбільшою платформою для розробки API зі спільною мовою. Однак кілька альтернативних фреймворків, які набули популярності, включають RAML, APIBlueprint і Summation.

Функції, що сприяють розробці API.

Під час написання API є так багато речей, які потрібно зрозуміти, як-от належна обробка помилок, модульність коду, дотримання протоколів та інше. Swagger надає нам інструменти для швидкого кодування наших API, піклуючись про всі ці речі. Використання swagger — це найшвидший шлях до прототипу API, а потім до коду, який можна розгортати у виробництві.

Інструмент з відкритим кодом Swagger CodeGen генерує шаблонний код під час написання API. Дозволити розробнику зосередитися на бізнес-логіці, а не витрачати час на написання синтаксичних матеріалів.

CodeGen піклується про очевидний сантехнічний та стандартний код. Так само, як проект Spring Boot робить із програмою на основі Spring.

Створення документації API.

Після завершення написання коду. Ще одне колосальне, виснажливе завдання для розробників — написання документації для свого коду. У мене голова болить, коли я про це думаю.

Swagger дозволяє нам звільнитися, створюючи та підтримуючи документи API для нас. Економить масу часу!! Крім того, нам не потрібно повертатися та змінювати документацію щоразу, коли змінюється код.

Усі документи автоматично оновлюються Swagger. Ми також можемо створювати різні версії API відповідно до наших вимог і примх. Ми також можемо налаштувати Swagger для створення документації для існуючих API.

Тестування API за допомогою Swagger.

OpenAPI Spec допомагає нам виконувати тестування функціональності, продуктивності та безпеки нашого API. Платформа ReadyAPI допомагає поступово запускати тести API.

За допомогою інспектора Swagger розробники можуть перевірити запит і відповідь API, щоб переконатися, що вони працюють належним чином. Інструмент також допомагає в регресійному тестуванні після серйозних змін коду.

Swagger має функції, що дозволяють автоматичне тестування API, імітацію API, створення навантажувальних тестів. Стрес-тестування API. Симуляція кутових випадків, таких як пікові умови руху, взаємодія з іноземними даними.

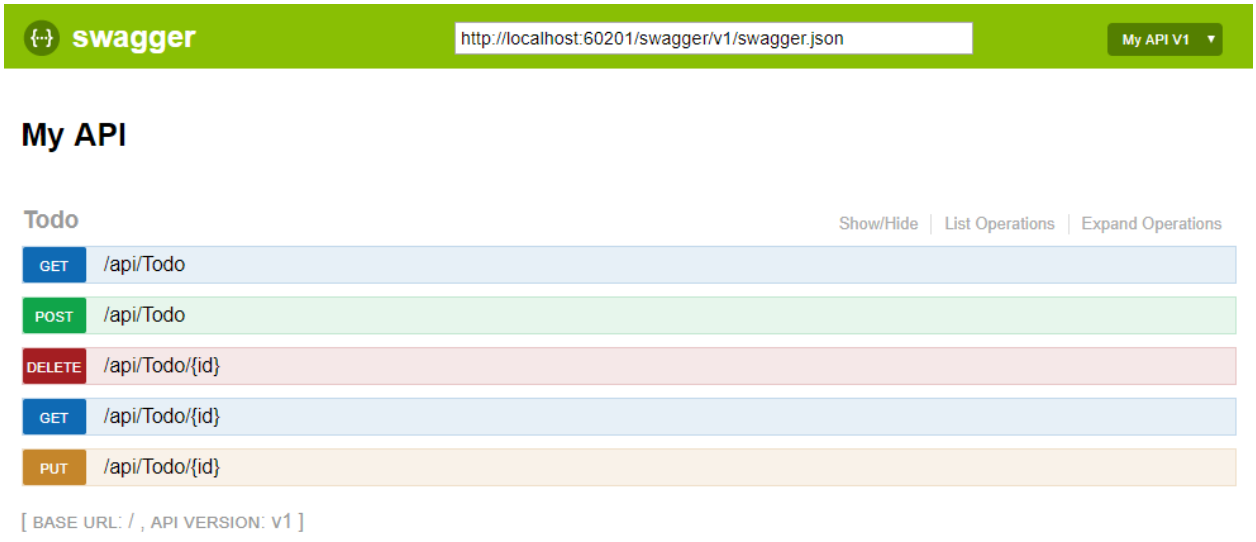

Рис. 2.6 - Інтерфейс Swagger.

Моніторинг API у виробництві.

Після того, як код розгорнутий у виробництві, його потрібно відстежувати на предмет послідовної поведінки. За допомогою Swagger ми можемо контролювати доступність, швидкість і функціональність наших API у виробництві.

Це допомагає нам відстежувати поведінку API, забезпечуючи плавне виконання та генерує сповіщення, якщо виявить щось не так. За допомогою API Swagger ми можемо програмно керувати процесом моніторингу API.

Що таке OpenAPI?

OpenAPI є глобальним стандартом для написання RESTful API. Це як специфікація, яка дозволяє розробникам по всій планеті стандартизувати дизайн своїх API. Крім того, під час написання REST API з нуля дотримуйтесь усіх правил безпеки, керування версіями, обробки помилок та інших найкращих практик. І не тільки на основі, навіть існуючі API можна налаштувати, щоб вони відповідали глобальному стандарту.

Спочатку OpenAPI був відомий як специфікація Swagger. Swagger придумав найкращі методи створення API, а потім ці найкращі методи стали специфікацією OpenAPI.

Такі інструменти, як SwaggerHub, допомагають розробникам створювати API у браузерному редакторі, що відповідає стандартам і має повний контроль над процесом проектування.

За допомогою таких інструментів, як Swagger Inspector, ми також можемо створювати власні специфікації API та передавати їх іншим командам в організації.

Отже, щоразу, коли створюється нова версія API. SwaggerHub перевіряє API на відповідність стандартам OpenAPI і перераховує будь-які серйозні відхилення від цього. Таким чином, невідповідності в дизайні виявляють і виправляють на ранніх стадіях.

Файл Open API містить всю специфікацію вашого API. Це допомагає розробникам повністю описати свій API, наприклад, перелік доступних кінцевих точок і операцій на кожній кінцевій точці. Параметри, що входять до методів, і відповідь від методу. Способи аутентифікації, метадані, такі як ліцензія, умови використання тощо.

Різниця між Swagger і Open API?

OpenAPI — це специфікація, а Swagger — реалізація специфікації. Так само, як JPA - це специфікація, а Hibernate - це реалізація.

Swagger надає інструменти для реалізації специфікації OpenAPI. Сьогодні OpenAPI береться на озброєння великими фанатами в галузі, одночасно сприяючи цьому, розвиваючи процес розробки API.

2.3.2 СУБД: MongoDB

MongoDB — програма керування базами даних NoSQL з відкритим вихідним кодом. NoSQL використовується як альтернатива традиційним реляційним базам даних. Бази даних NoSQL досить корисні для роботи з великими наборами розподілених даних. MongoDB — це інструмент, який може керувати документально-орієнтованою інформацією, зберігати або отримувати інформацію.

MongoDB підтримує різні форми даних. Це одна з багатьох технологій нереляційних баз даних, які виникли в середині 2000-х під прапором NoSQL – зазвичай для використання в програмах великих даних та інших роботах з обробки даних, які погано вписуються в жорстку реляційну модель. Замість використання таблиць і рядків, як у реляційних базах даних, архітектура MongoDB складається з колекцій і документів.

Організації можуть використовувати Mongo DB для спеціальних запитів, індексування, балансування навантаження, агрегації, виконання JavaScript на стороні сервера та інших функцій.

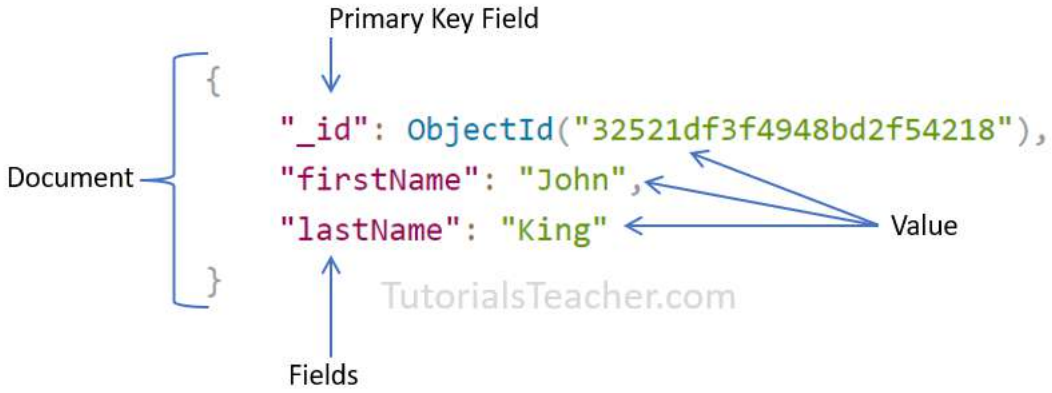

Рис. 2.7 - Структура документа в MongoDB.

MongoDB використовує записи, які складаються з документів, які містять структуру даних, що складається з пар полів і значень. Документи є основною одиницею даних у MongoDB. Документи подібні до нотації об'єктів JavaScript, але використовують варіант під назвою Binary JSON (BSON). Перевага використання BSON полягає в тому, що він вміщує більше типів даних. Поля в цих документах подібні до стовпців у реляційній базі даних. Містяться значення можуть бути різних типів даних, включаючи інші документи, масиви та масиви документів, відповідно до посібника користувача MongoDB. Документи також включатимуть первинний ключ як унікальний ідентифікатор.

Набори документів називаються колекціями, які функціонують як еквівалент таблиць реляційної бази даних. Колекції можуть містити будь-які типи даних, але обмеження полягає в тому, що дані колекції не можуть бути розподілені між різними базами даних.

Оболонка mongo є стандартним компонентом відкритих дистрибутивів MongoDB. Після встановлення MongoDB користувачі підключають оболонку mongo до своїх запущених екземплярів MongoDB. Оболонка mongo діє як інтерактивний інтерфейс JavaScript для MongoDB, що дозволяє користувачам запитувати й оновлювати дані, а також виконувати адміністративні операції.

Двійкове представлення JSON-подібних документів забезпечується форматом зберігання документів і обміну даними BSON. Автоматичне розподілення — це ще одна ключова функція, яка дозволяє розподілити дані в колекції MongoDB між кількома системами для горизонтальної масштабованості, оскільки обсяги даних і вимоги до пропускної здатності зростають.

СУБД NoSQL використовує єдину головну архітектуру для узгодженості даних із вторинними базами даних, які зберігають копії первинної бази даних. Операції автоматично реплікуються в ці вторинні бази даних для автоматичного перемикання збоїв.

Плюси і мінуси MongoDB.

Як і інші бази даних NoSQL, MongoDB не вимагає попередньо визначених схем. Він зберігає будь-які типи даних. Це дає користувачам можливість створювати будь-яку кількість полів у документі, полегшуючи масштабування баз даних MongoDB порівняно з реляційними базами даних.

Однією з переваг використання документів є те, що ці об’єкти відображаються на рідні типи даних у ряді мов програмування. Крім того, наявність вбудованих документів зменшує потребу в об’єднанні бази даних, що може зменшити витрати.

Основною функцією MongoDB є його горизонтальна масштабованість, що робить його корисною базою даних для компаній, що працюють із додатками для великих даних. Крім того, шардінг дозволяє базі даних розподіляти дані між кластером машин. Нові версії MongoDB також підтримують створення зон даних на основі ключа шарда.

MongoDB підтримує ряд механізмів зберігання даних і надає підключаються API механізмів зберігання, які дозволяють третім сторонам розробляти власні механізми зберігання для MongoDB.

СУБД також має вбудовані можливості агрегації, які дозволяють користувачам запускати код MapReduce безпосередньо в базі даних, а не

виконувати MapReduce на Hadoop. MongoDB також включає власну файлову систему під назвою GridFS, подібну до розподіленої файлової системи Hadoop (HDFS). Використання файлової системи переважно для зберігання файлів, розмір яких перевищує ліміт BSON у 16 МБ на документ. Ці подібності дозволяють використовувати MongoDB замість Hadoop, хоча програмне забезпечення бази даних інтегрується з Hadoop, Spark та іншими фреймворками обробки даних.

Хоча MongoDB має деякі цінні переваги, у нього також є деякі недоліки. Завдяки стратегії автоматичного перемикання збоїв користувач налаштовує лише один головний вузол у кластері MongoDB. Якщо головний вузол виходить з ладу, інший вузол автоматично перетворюється на новий головний. Цей перемикач обіцяє безперервність, але це не миттєво – це може зайняти до хвилини. Для порівняння, база даних Cassandra NoSQL підтримує кілька головних вузлів, тож якщо один головний вузол виходить з ладу, інший буде готовий для високодоступної інфраструктури бази даних.

Єдиний головний вузол MongoDB також обмежує швидкість запису даних у базу даних. Запис даних має бути записаний на головному вузлі, а запис нової інформації в базу даних обмежено потужністю цього головного вузла.

Інша потенційна проблема полягає в тому, що MongoDB не забезпечує повну посилальну цілісність через використання обмежень зовнішнього ключа, що може вплинути на узгодженість даних. Крім того, аутентифікація користувачів не ввімкнена за замовчуванням у базах даних MongoDB, що свідчить про популярність технології серед розробників. Однак зловмисники атакували велику кількість незахищених систем MongoDB, що призвело до додавання параметра за замовчуванням, який блокує мережеві з’єднання з базами даних, якщо вони не були налаштовані адміністратором бази даних.

Платформи MongoDB.

MongoDB доступний у спільнотах і комерційних версіях через постачальника MongoDB Inc. MongoDB Community Edition є випуском із відкритим кодом, а MongoDB Enterprise Server надає додаткові функції безпеки, механізм зберігання в пам'яті, функції адміністрування та аутентифікації, а також можливості моніторингу через Ops Manager.

Графічний інтерфейс користувача (GUI) під назвою MongoDB Compass дає користувачам можливість працювати зі структурою документа, виконувати запити, індексувати дані тощо. Конектор MongoDB для BI дозволяє користувачам підключати базу даних NoSQL до своїх інструментів бізнес-аналітики для візуалізації даних і створення звітів за допомогою SQL-запитів.

Наслідуючи інших постачальників баз даних NoSQL, MongoDB Inc. запустила хмарну базу даних як сервіс під назвою MongoDB Atlas у 2016 році. Atlas працює на AWS, Microsoft Azure та Google Cloud Platform. Пізніше MongoDB випустила платформу під назвою Stitch для розробки додатків на MongoDB Atlas з планами розширити її на локальні бази даних.

Компанія також додала підтримку транзакцій ACID з кількома документами як частину MongoDB 4.0 у 2018 році. Відповідність властивостям ACID — атомарності, узгодженості, ізоляції та довговічності — у кількох документах розширює типи операційних навантажень, які MongoDB може обробляти з гарантованою точність і надійність.

Історія MongoDB.

MongoDB був створений Дуайтом Меррiманом та Еліотом Горовіцем, які зіткнулися з проблемами розвитку та масштабованості традиційних підходів до реляційних баз даних під час створення веб-додатків у DoubleClick, онлайн-рекламній компанії, яка зараз належить Google Inc. Назва бази даних походить від слово humongous представляє ідею підтримки великих обсягів даних.

Мерріман і Горовіц допомогли створити 10Gen Inc. у 2007 році для комерціалізації MongoDB і пов'язаного з ним програмного забезпечення. У 2013 році компанія була перейменована в MongoDB Inc. і стала публічною в жовтні 2017 року під символом MDB.

СУБД була випущена як програмне забезпечення з відкритим вихідним кодом у 2009 році і доступна на умовах версії 3.0 загальної публічної ліцензії GNU Affero Foundation Free Software Foundation, на додаток до комерційних ліцензій, пропонованих MongoDB Inc.

Такі організації, як страхова компанія MetLife, використовували MongoDB для програм обслуговування клієнтів, в той час як інші веб-сайти, такі як Craigslist, використовували його для архівування даних, а фізична лабораторія CERN використовувала його для агрегації та виявлення даних. Крім того, The New York Times використовувала MongoDB для підтримки програми для створення форм для надсилання фотографій.

2.3.3 Docker

Docker — це програмна платформа з відкритим вихідним кодом для створення, розгортання та керування віртуалізованими контейнерами програм у загальній операційній системі (ОС) з екосистемою суміжних інструментів. Технологія контейнерів Docker дебютувала в 2013 році; Компанія Docker Inc. була створена для підтримки комерційної версії програмного забезпечення для керування контейнерами і була основним спонсором версії з відкритим вихідним кодом. Mirantis придбала бізнес Docker Enterprise у листопаді 2019 року.

Як працює Docker.

Docker пакує, забезпечує та запускає контейнери. Контейнерна технологія доступна через операційну систему: контейнер упаковує службу або функцію програми з усіма бібліотеками, файлами конфігурації, залежностями та іншими необхідними частинами та параметрами для роботи.

Кожен контейнер використовує послуги однієї базової операційної системи. Образи Docker містять усі залежності, необхідні для виконання коду всередині контейнера, тому контейнери, які переміщуються між середовищами Docker з однією ОС, працюють без змін.

Docker використовує ізоляцію ресурсів у ядрі ОС для запуску кількох контейнерів в одній ОС. Це відрізняється від віртуальних машин (VM), які інкапсулюють цілу ОС із виконуваним кодом поверх абстрактного рівня фізичних апаратних ресурсів.

Docker було створено для роботи на платформі Linux, але було розширено, щоб запропонувати більшу підтримку операційних систем, що не належать до Linux, включаючи Microsoft Windows і Apple OS X. Доступні версії Docker для Amazon Web Services (AWS) і Microsoft Azure.

Компоненти та інструменти Docker.

Docker Community Edition є відкритим вихідним кодом, а Docker Enterprise Edition — це комерційна версія, пропонована Docker Inc. Docker складається з різних компонентів та інструментів, які допомагають створювати, перевіряти та керувати контейнерами.

Docker Engine — це базова технологія, яка обробляє завдання та робочі процеси, пов’язані зі створенням контейнерних додатків. Движок створює процес-демона на стороні сервера, який містить зображення, контейнери, мережі та томи зберігання. Демон також надає клієнтський інтерфейс командного рядка (CLI), щоб користувачі могли взаємодіяти з демоном через програмний інтерфейс програми Docker. Контейнери, створені Docker, називаються Dockerfiles. Файли Docker Compose визначають склад компонентів у контейнері Docker.

Docker Hub — це інструмент програмного забезпечення як послуги, який дозволяє користувачам публікувати контейнерні додатки та ділитися ними через загальну бібліотеку. Сервіс рекламує понад 100 000 загальнодоступних програм, а також державні та приватні реєстри контейнерів.

Подібно до Hub, Trusted Registry — це сховище з додатковим рівнем контролю та права власності на зберігання та розповсюдження зображень контейнера.

Режим Docker Swarm в Docker Engine підтримує балансування кластерного навантаження для Docker. Ресурси кількох хостів Docker об’єднані разом, щоб діяти як один, що дозволяє користувачам швидко масштабувати розгортання контейнерів на кількох хостах.

Universal Control Plane — це веб-інтерфейс уніфікованого кластера та керування додатками.

Compose — це інструмент для налаштування служб багатоконтейнерних додатків, перегляду статусів контейнерів, виводу журналу потоків і запуску процесів з одним екземпляром.

Content Trust — це інструмент безпеки для перевірки цілісності віддалених реєстрів Docker за допомогою підписів користувачів і тегів зображень.

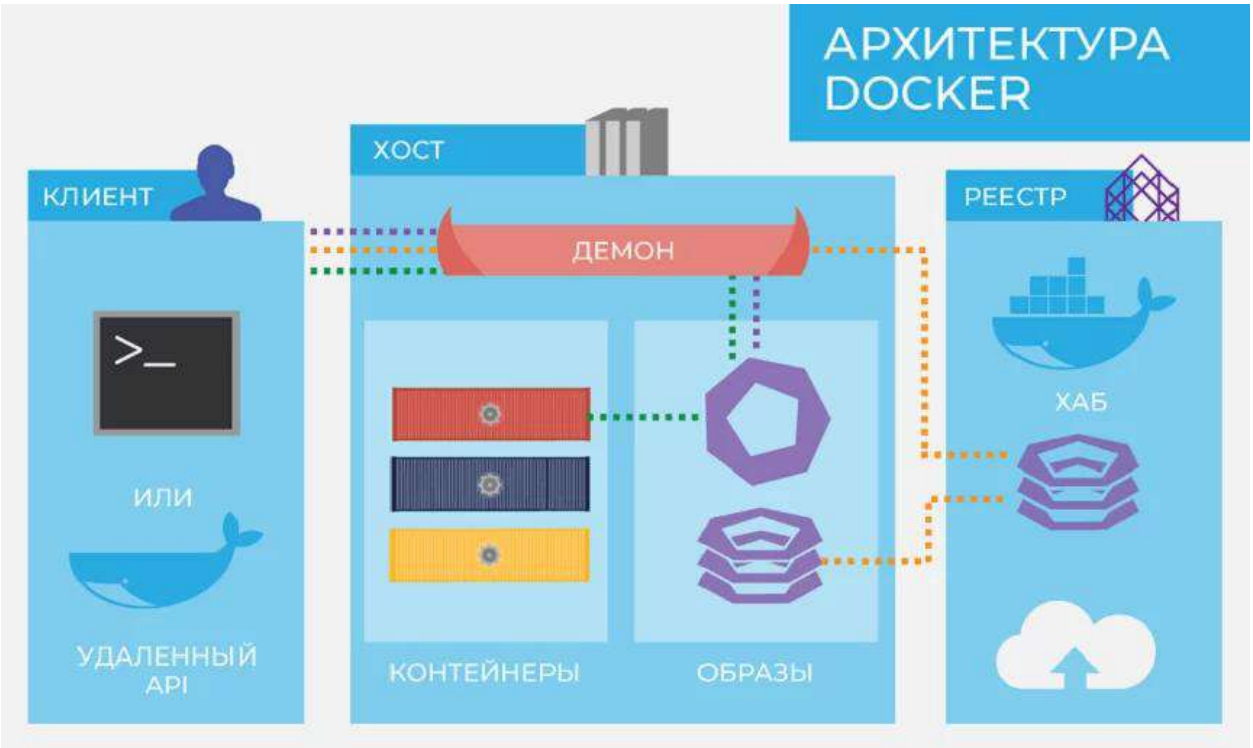

Рис. 2.8 - Візуалізація інфраструктури Docker.

Версії Docker та основні функції.

Docker Enterprise 1.13, випущений у січні 2017 року, додав зворотну сумісність для інтерфейсу командного рядка (CLI) для роботи зі старими демонами Docker, а також кілька команд очищення для ефективнішого керування дисковим простором і даними, а також деякі виправлення помилок та безпеки. Інші вдосконалення Docker Enterprise у 2017 році включали вбудовану підтримку Kubernetes для оркестровки контейнерів, на додаток до режиму Docker Swarm; і підтримка мейнфрейма IBM і Windows Server 2016, щоб користувачі могли запускати змішані кластери та програми в кількох операційних системах.

Docker Enterprise Edition 2.0, випущений у квітні 2018 року, містив підтримку кількох ОС і кількох хмар для гібридних середовищ.

Docker Enterprise 3.0, остання версія від липня 2019 року, додала синьо-зелені оновлення кластерів контейнерів і можливість створювати багатосервісні програми на основі контейнерів, які запускаються з будь-якого середовища. Інші нові функції включають:

Docker Desktop Enterprise, який дозволяє розробникам розгортати програми в середовищі, що відповідає Kubernetes, з автоматизованою інтеграцією конвеєра та централізованим управлінням IT;

Docker Applications, набір інструментів підвищення продуктивності для розробників;

Служба Docker Kubernetes, яка автоматизує керування та масштабування програм на основі Kubernetes, а також забезпечує безпеку, контроль доступу та автоматизоване керування життєвим циклом; і

Docker Enterprise as a Service, повноцінний керований корпоративний контейнерний сервіс.

Переваги та недоліки Docker.

Docker з'явився як де-факто стандартна платформа для швидкого створення, створення, розгортання, масштабування та контролю контейнерів

на всіх хостах Docker. Docker забезпечує високий ступінь портативності, щоб користувачі могли реєструвати та ділитися контейнерами на різних хостах у приватних і публічних середовищах. Переваги Docker включають ефективну розробку додатків, менше використання ресурсів і швидше розгортання порівняно з віртуальними машинами.

Існують також потенційні проблеми з Docker. Величезною кількістю контейнерів, можливих на підприємстві, може бути важко ефективно керувати. Оскільки використання контейнерів еволюціонує від детального віртуального хостингу до оркестровки компонентів і ресурсів програми, поширення та взаємозв'язок компонентних додатків, які можуть включати сотні ефемерних контейнерів, є серйозною перешкодою.

В останні роки Docker був замінений Kubernetes для оркестровки контейнерів; однак більшість пропозицій Kubernetes насправді запускають Docker поза лаштунками.

Безпека Docker.

Історично постійна проблема з контейнерами - і Docker, як синонімічний розширення - це безпека. Незважаючи на чудову логічну ізоляцію, контейнери все ще використовують операційну систему хоста. Атака або збій у базовій операційній системі можуть потенційно поставити під загрозу всі контейнери, що працюють на ОС. Уразливості можуть включати доступ та авторизацію, зображення контейнерів та мережевий трафік між контейнерами. Образи Docker можуть зберігати root-доступ до хоста за замовчуванням, хоча це часто переноситься з пакетів сторонніх постачальників.

Docker регулярно додає покращення безпеки до платформи Docker, такі як сканування зображень, впровадження безпечних вузлів, ідентифікація криптографічних вузлів, сегментація кластерів і безпечне розповсюдження секретів. Керування секретами Docker також існує в Kubernetes, а також у D2iQ, CISOfy Lynis і HashiCorp Vault. Різні інструменти сканування безпеки контейнерів з'явилися від Aqua, Twistlock, NeuVector та інших.

Деякі організації запускають контейнери на віртуальній машині, хоча для контейнерів не потрібні віртуальні машини – це не вирішує вектор проблеми спільних ресурсів, але пом’якшує потенційний вплив недоліку безпеки. Іншою альтернативою є використання низькопрофільних або «мікро» ВМ, які не вимагають таких же накладних витрат, як типова ВМ; наприклад, gVisor, Kata Containers і Amazon Firecracker. Перш за все, найпоширенішим і рекомендованим кроком для забезпечення безпеки контейнерів є не піддавати доступ до Інтернету хостів контейнерів, а використовувати лише зображення контейнерів із відомих джерел.

Безпека також була основним пунктом продажу альтернатив Docker, зокрема rkt CoreOS. Однак Docker зробив багато кроків для покращення своїх параметрів безпеки, в той же час імпульс для цих альтернатив контейнерів згас.

Альтернативи Docker, екосистема та стандартизація.

Існують інструменти сторонніх розробників, які працюють з Docker для таких завдань, як керування контейнерами та кластеризація. Екосистема Docker включає в себе поєднання відкритих і запатентованих технологій, таких як Kubernetes з відкритим кодом, власне пакування Kubernetes OpenShift від Red Hat і розповсюдження Kubernetes від Canonical, яке називають «чистим» Kubernetes. Docker конкурує з власними контейнерами додатків, такими як VMware vApp та інструментами абстракції інфраструктури, включаючи Chef.

Docker — не єдина доступна контейнерна платформа, але вона займає переважну більшість ринку контейнерів. CoreOS rkt, вимовляється як ракета, відомий своєю безпекою з підтримкою SELinux і надійним керуванням платформою. Red Hat (зараз належить IBM) придбала CoreOS і інтегрувала його функціональні можливості в архітектуру OpenShift. Однак rkt зараз є архівним проектом у Cloud Native Computing Foundation.

Інші основні контейнерні платформи включають LXD, який є від Canonical (і його версії Ubuntu Linux), і OpenVZ, найстарішої із системних контейнерних платформ, спочатку розробленої Virtuozzo. OpenVZ поєднує невеликий розмір і високу швидкість стандартних контейнерів з додатковою безпекою абстрактного рівня ОС.

Docker також відігравав провідну роль в ініціативі щодо більш формальної стандартизації упаковки та розповсюдження контейнерів: Open Container Initiative, створеної для підтримки загального формату контейнера та середовища виконання. Серед інших учасників — понад 40 інших постачальників контейнерної індустрії, включаючи CoreOS, AWS, Intel, Red Hat і Virtuozzo.

Нарешті, Windows Server 2019 і Windows 10 пропонують пряму підтримку контейнерів за допомогою функції контейнера Windows на основі технології Docker.

Історія компанії Docker.

Docker був вперше випущений як платформа з відкритим кодом у березні 2013 року під назвою dotCloud. Docker Engine 1.0 був запущений у 2014 році. У 2016 році Docker інтегрував свою групову оркестрацію з Docker Engine у версії 1.12. Більш широка мета Docker полягала в тому, щоб розвинути свій бізнес із використанням контейнера як сервісу, але врешті-решт ці плани наздогнав розвиток Kubernetes.

Docker Enterprise був представлений у березні 2017 року, і компанія також передала свою утиліту для виконання контейнерів для контейнерів Cloud Native Computing Foundation.

У листопаді 2019 року Mirantis придбала продукти Docker та IP навколо Docker Engine - Enterprise, Docker Trusted Registry, Docker Universal Control Plane і Docker CLI, а також комерційний продукт Docker Swarm. Спочатку Мірантіс вказав, що зосередиться на Kubernetes і зрештою припинить підтримку Docker Swarm, але пізніше підтвердив свій намір підтримувати та

розробляти нові функції для нього. Решта компанії Docker Inc. тепер зосереджена на Docker Desktop, а також на Docker Hub.

2.3.4 Среда розробки WebStorm

WebStorm — сучасна екосистема JavaScript. Він включає в себе інтелектуальне завершення коду, виявлення помилок «на льоту», потужну навігацію та рефакторинг для JavaScript, TypeScript, мов таблиць стилів та всіх популярних фреймворків.

Функції WebStorm:

- Інтелектуальна допомога з кодуванням – WebStorm надає вам розумну допомогу в кодуванні для JavaScript і мов, скомпільованих у JavaScript, Node.js, HTML і CSS. Насолоджуйтесь завершенням коду, потужними функціями навігації, виявленням помилок «на льоту» та рефакторингом для всіх цих мов.
- Сучасні фреймворки - WebStorm надає розширену допомогу в кодуванні для Angular, React, Vue.js і Meteor. Насолоджуйтесь підтримкою React Native, PhoneGap, Cordova та Ionic для мобільної розробки та розробляйте для сервера за допомогою Node.js.
- Розумний редактор — IDE аналізує ваш проект, щоб забезпечити найкращі результати завершення коду для всіх підтримуваних мов. Сотні вбудованих інспекцій повідомляють про будь-які можливі проблеми прямо під час введення тексту та пропонують варіанти швидкого вирішення.Налагодження, відстеження та тестування – WebStorm надає потужні вбудовані інструменти для налагодження, тестування та відстеження ваших програм на стороні клієнта та Node.js. Завдяки мінімальній необхідній конфігурації та продуманій інтеграції в IDE ці завдання значно легші з WebStorm.

- Налагодження – WebStorm надає вбудований налагоджувач для коду на стороні клієнта (який працює з Chrome) і програм Node.js. Розмістіть точки зупинки, перегляньте код та оцініть вирази – і все це, не виходячи з IDE.
- Тестування. Виконуйте тестування з легкістю, оскільки WebStorm інтегрується з Karma test Runner, Mocha, Jest і Protractor. Виконуйте та налагоджуйте тести прямо в IDE, переглядайте результати в красивому візуальному форматі та перейдіть до коду тесту.
- Відстеження та профілювання – WebStorm містить spy-js, вбудований інструмент, який допомагає відстежувати код JavaScript. Дослідіть, як файли пов’язані з викликами функцій, і ефективно визначте будь-які можливі вузькі місця.
- Повна інтеграція інструментів – WebStorm інтегрується з популярними інструментами командного рядка для веб-розробки, надаючи вам продуктивний, спрощений досвід розробки без використання командного рядка.
- Інструменти для створення – насолоджуйтесь простим уніфікованим інтерфейсом користувача для виконання завдань Grunt, Gulp або npm прямо з IDE. Усі завдання перераховані у спеціальному вікні інструментів, тому просто двічі клацніть назву завдання, щоб запустити його.
- Інструменти якості коду – на додаток до сотень власних перевірок WebStorm, він може запускати ESLint, JSCS, TSLint, Stylelint, JSHint або JSLint для вашого коду та висвітлювати будь-які проблеми на льоту, прямо в редакторі.
- Шаблони проектів. Розпочинайте нові проекти з екрана привітання, використовуючи популярні шаблони проектів, як-от Express або Web starter kit, і отримайте доступ до ще більшої кількості генераторів проектів завдяки інтеграції з Yeoman.

- Функції IDE - WebStorm побудовано на основі відкритої платформи IntelliJ. Насолоджуйтесь тонко налаштованим, але дуже настроюваним досвідом, який він надає, щоб відповідати вашому робочому процесу розробки.
- VCS – WebStorm надає уніфікований інтерфейс користувача для роботи з багатьма популярними системами контролю версій, забезпечуючи послідовну роботу користувачів у git, GitHub, SVN, Mercurial та Perforce.
- Локальна історія. Незалежно від того, використовуєте ви VCS чи ні, локальна історія може бути справжньою заощадженням коду. У будь-який момент ви можете переглянути історію певного файлу чи каталогу та повернутися до будь-якої з попередніх версій.
- Налаштування - WebStorm надзвичайно настроюється. Налаштуйте його так, щоб він ідеально відповідав вашому стилю кодування, від ярликів, шрифтів і візуальних тем до вікон інструментів і макета редактора.
-

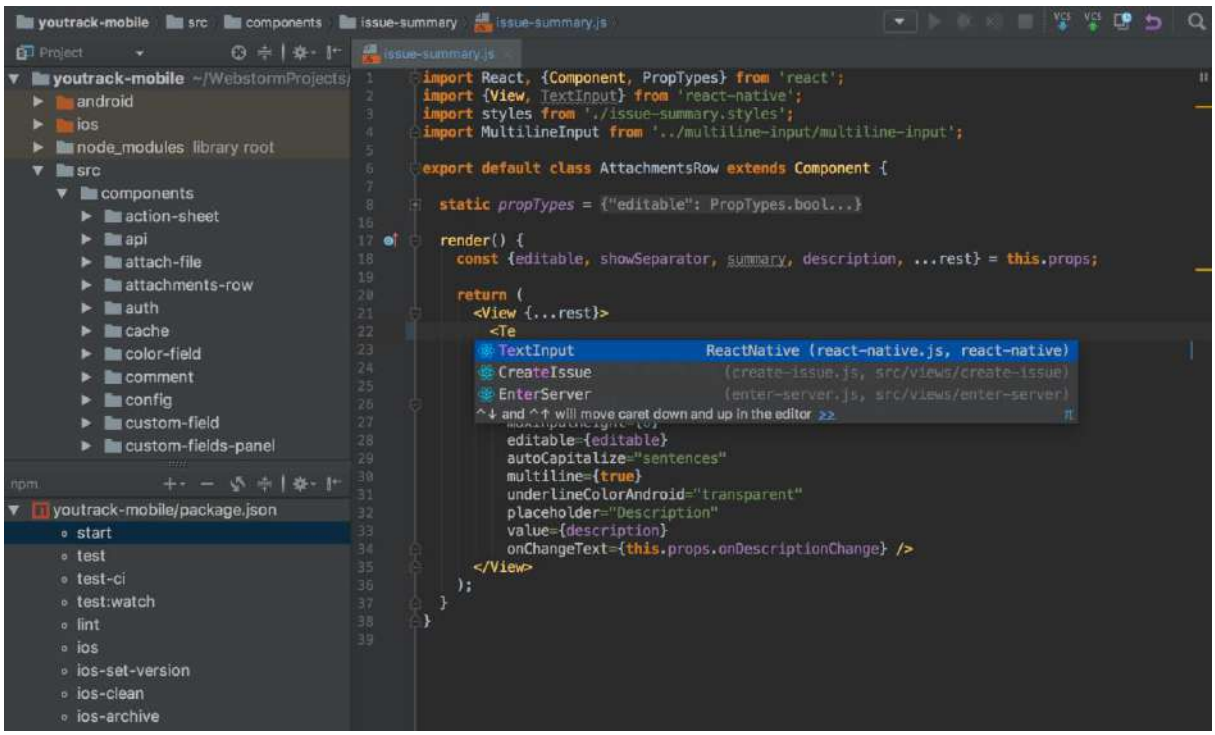

Рис. 2.9 - Інтерфейс WebStorm.

2.4 Висновок по розділу

Враховуючи усе вище зазначене було обрано для створення веб-серверу наступні технології:

- Swagger для легкого користування API.
- Nest.JS для побудови мікро-серверної архітектури на зручній мові програмування TypeScript.
- MondoDB як сучасне рішення для збереження даних.
- Docker для зручного розгортання веб-сервера на локальному обладнані.
- WebStorm як швидкий та зручний інструмент для написання коду.

РОЗДІЛ 3
ПРОГРАМНИЙ МОДУЛЬ

3.1 Розробка бази даних

Розробка бази даних починається з визначення полів для всіх сутностей системи. Було створено дві таблиці: таблиця User, яка зберігає інформацію про користувача та таблиця Task, яка зберігає задачі користувачів.

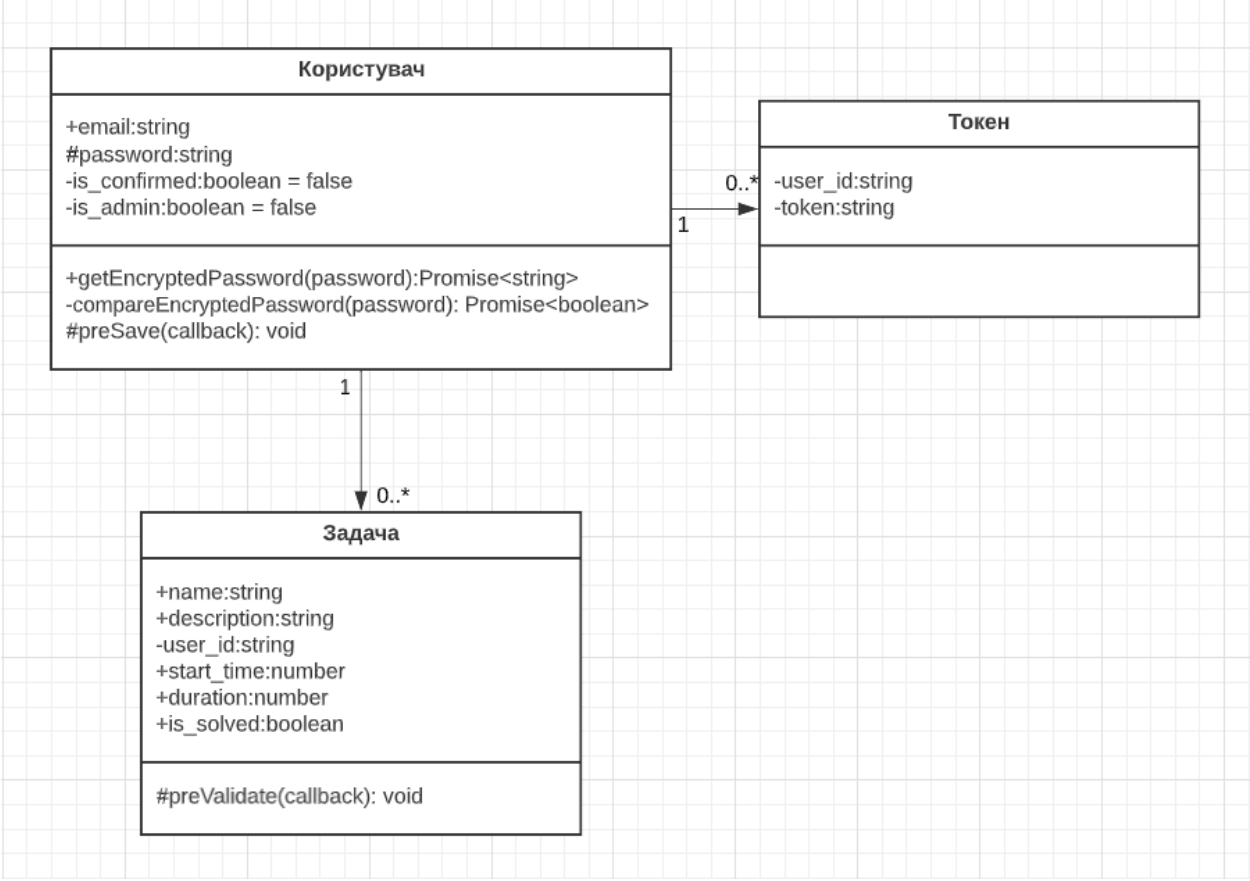

Рис. 3.1 - Діаграма класів.

Таблиця користувачів має унікальний ключ (ID), поштову адресу, пароль та поле яке визначає чи підтвердив користувач свій акаунт на пошті. Також модель користувача має додаткові функції “getEncryptedPassword” та “compareEncryptedPassword” що виконують функції додаткової безпеки,

шифрують та дешифрують пароль користувача відповідно. “preSave” це функція «хук» що викликається перед збереженням даних користувача та змінює його пароль та результат виклика функції “ getEncryptedPassword”.

Реалізація “preSave” хука:

```
UserSchema.pre('save', async function (next) {
        if (!this.isModified('password')) {
                return next();
        }
        this.password = await this.getEncryptedPassword(this.password);
        next();
});
```

Коли зберігаємо користувача, перевіряємо чи були зміни в полі пароля та хешируємо його якщо такі зміни були.

Таблиця задач має унікальний ключ (ID), назву, опис, дату початку задачі, тривалість, позначку чи задача є зробленою та унікальний ключ користувача к котрому ця задача відноситься. Такий зв’язок має назву «один до багатьох». Також модель задачі має “preValidate” хук який викликається до збереження даних моделі для того щоб додатково провалідувати задачу.

Реалізація “preValidate” хука:

```
TaskSchema.pre('validate', function (next) {
  const self = this as ITask;
  if (this.isModified('user_id') && self.created_at) {
    this.invalidate('user_id', 'The field value can not be updated');
  }
  next();
});
```

Перевіряємо чи була задача вже створена та забороняємо модифікувати власника задачі.

Як помічник роботи з базою даних, використовується npm пакет «mongoose», який має багато функцій для роботи з базою даних які перекладаються на мову NOSQL, що суттєво спрощує розробку веб-сервера.

Приклад роботи з базою даних за допомогою «mongoose»:

```
import { Model } from 'mongoose';
constructor(
  @InjectModel('User') private readonly userModel: Model<IUser>
) {}
public async searchUserById(id: string): Promise<IUser> {
  return this.userModel.findById(id).exec();
}
```

Функція findById приймає ключ як обов’язковий аргумент та повертає користувача з таким ключом якщо той присутній у базі даних.

Таким чином сервер працює з базою даних без взаємодії з неї напряму, що робить такий зв’язок більш абстрактним та дозволяє з легкістю змінити СУБД без потреби адаптувати код під нову систему.

3.2 Проектування системи

На рисунку 3.2 зображено діаграму прецедентів веб-сервера. Діаграма містить два актора: Користувач та адміністратор.

Користувач має обмежений функціонал, що складається з можливості увійти до системи, переглянути свої задачі та вийти з системи.

Адміністратор має той самий функціонал що й користувач окрім того що може підтвердити користувача та створити задачу для користувача.

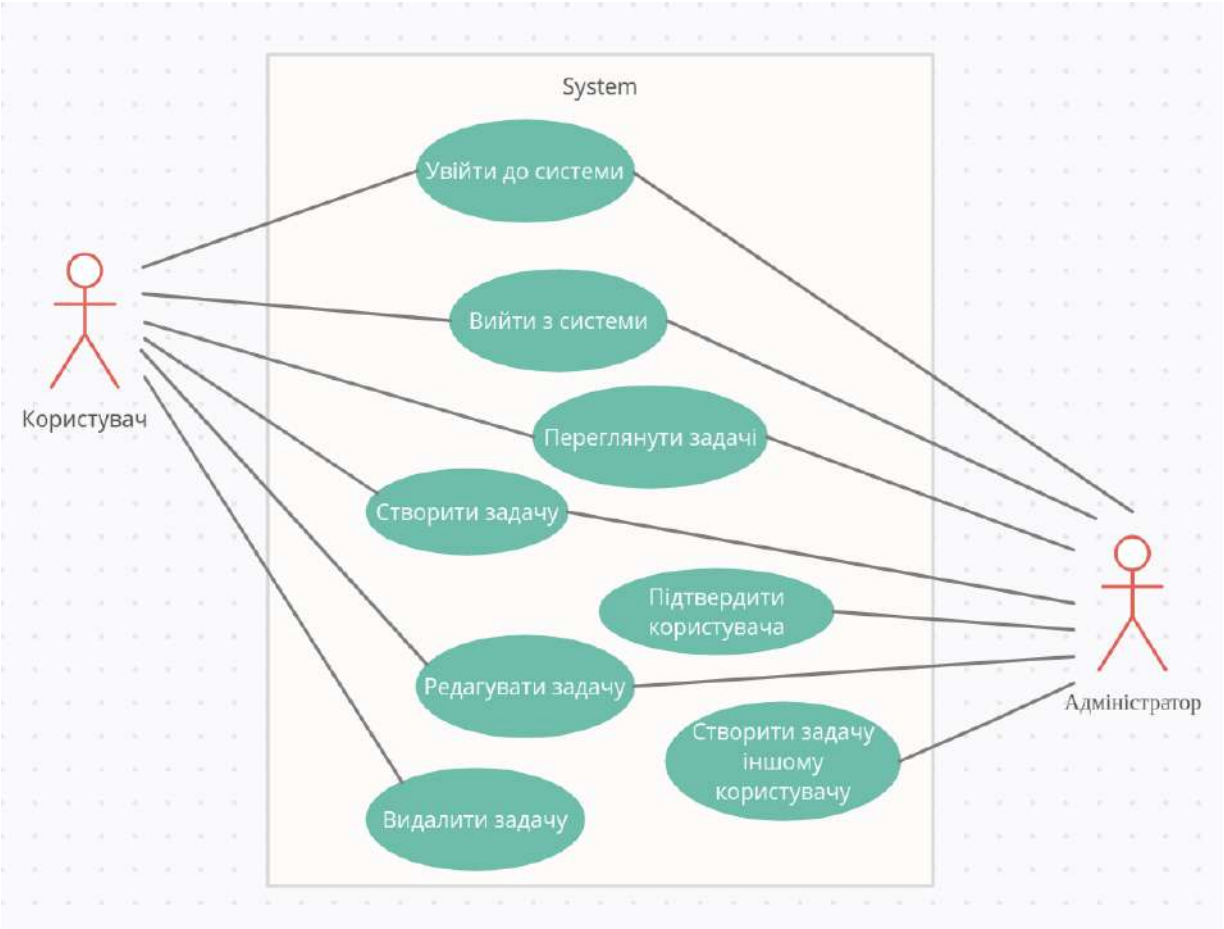

Рис. 3.2. Діаграма прецедентів.

Більш детальний опис системи розташовано на рисунку 3.3.

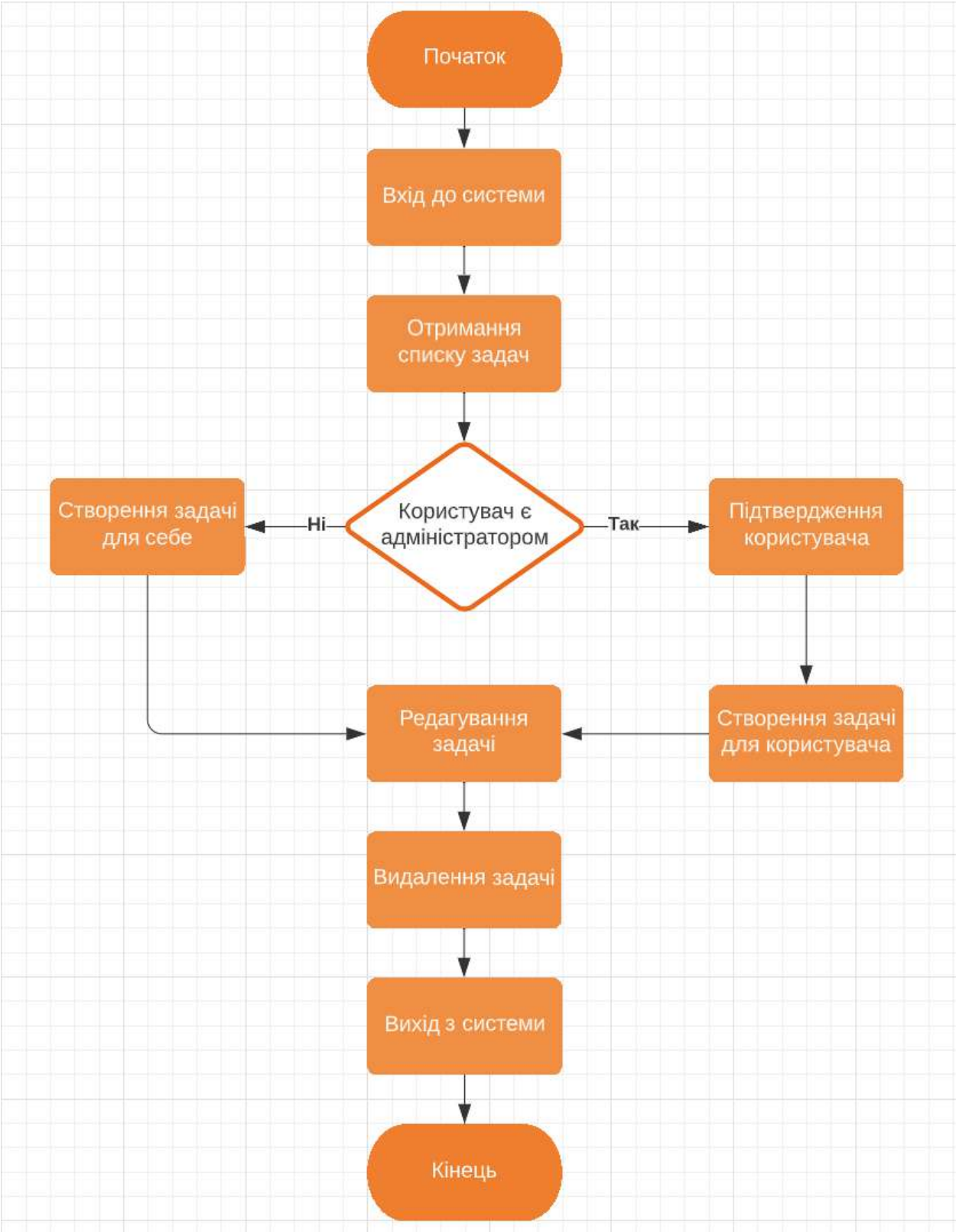

Рис. 3.3 - Діаграма активності.

3.3 Програмування системи

Сервер складається з чотирьох мікросервісів:

- Permission: відповідає за перевірку прав користувача;
- Task: відповідає за роботу з задачами (їх створення, редагування та видалення);
- Token: відповідає за генерацію та валідацію тимчасових токенів;
- User: відповідає за роботу з користувачами (їх створення, редагування та видалення);

Також проект має директорію db, яка використовується СУБД для збереження тимчасових даних, та gateway який відповідає за Front-end частину проекту.

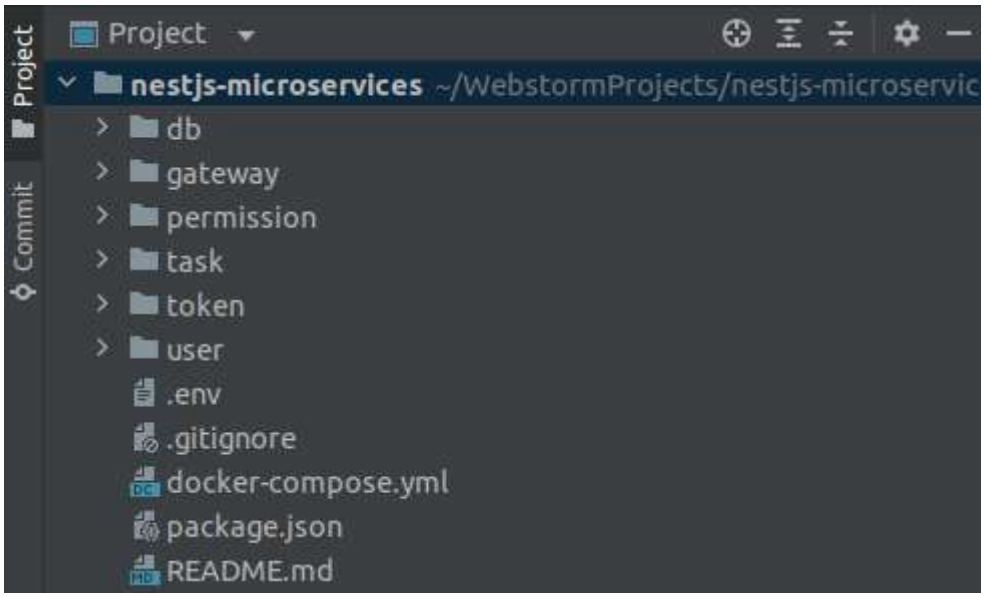

Рис. 3.4 - Каталог проекту.

Файл .env відповідає за збереження глобальних змінних проекту та має вигляд текстового файлу.

MONGO_DSN=mongodb://testAdmin1:testAdmin1@localhost:27017/testdb

MONGO_DATABASE=testdb

MONGO_USER=testUser1

MONGO_PASSWORD=testUser1

MONGO_ROOT_USER=testAdmin1

MONGO_ROOT_PASSWORD=testAdmin1

API_GATEWAY_PORT=8000

TASK_SERVICE_PORT=8001

TASK_SERVICE_HOST=localhost

TOKEN_SERVICE_PORT=8002

TOKEN_SERVICE_HOST=localhost

USER_SERVICE_PORT=8003

USER_SERVICE_HOST=localhost

MAILER_SERVICE_PORT=8004

MAILER_SERVICE_HOST=localhost

PERMISSION_SERVICE_PORT=8005

PERMISSION_SERVICE_HOST=localhost

BASE_URI=http://localhost

Структура написання .env файлів це КЛЮЧ=ЗНАЧЕННЯ та групування їх з нового рядка.

.gitignore це файл який спрощує роботу з системою контроля версій GIT та має перелік назв файлів які потрібно проігнорувати та не додавати до репозиторію GIT.

Приклад .gitignore в моєму проекті:

dist // скомпільований код проекту

node_modules // модулі npm

logs // логі проекту

.env // глобальні зміни

db // тимчасові файли бази даних

Розглянемо структуру мікросервіса на прикладі мікросервіса user. Мікросервіс має директорію dist, яка зберігає скомпільований код, директорію node_modules, яка зберігає велику кількість бібліотек та інших залежностей,

файлі конфігі такі як nest-cli.json, package.json, tsconfig.json та основну директорію src.

Src директорія має вхідний файл main.ts, контролер user.controller.ts, модуль user.module.ts та директорії interfaces, schemas та services.

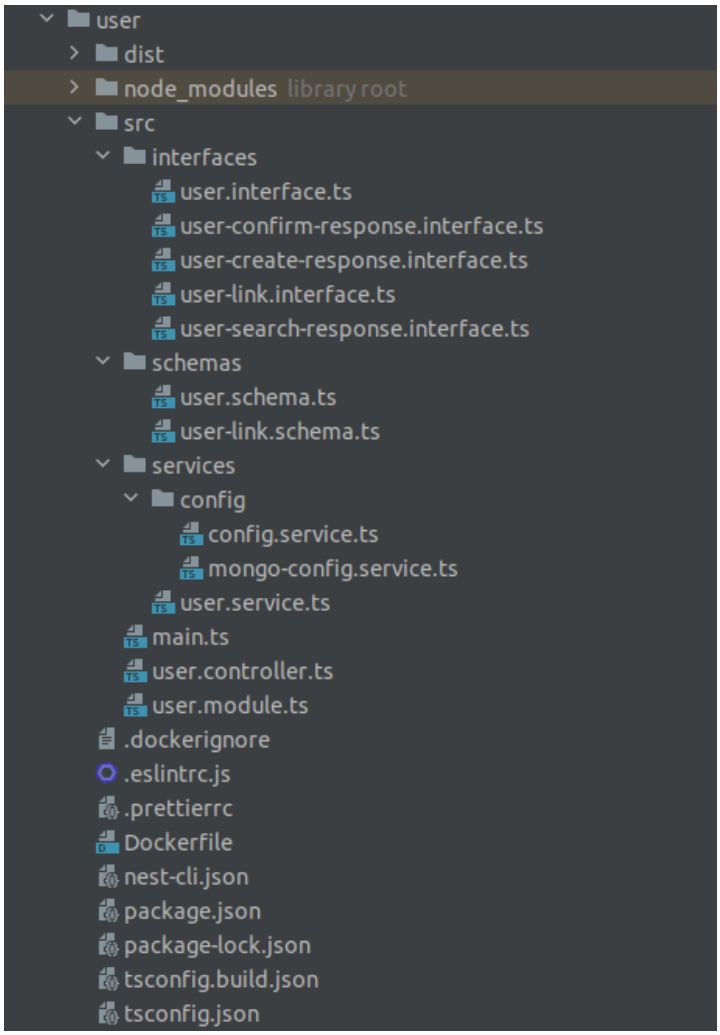

Рисунок 3.5 - Структура мікросервіса user.

Main.ts це файл з якого починається робота мікросервіса, він має такий вигляд:

```
import { NestFactory } from '@nestjs/core';
import { UserModule } from './user.module';
import { Transport, TcpOptions } from '@nestjs/microservices';
import { ConfigService } from './services/config/config.service';
```

```
async function bootstrap() {
  const app = await NestFactory.createMicroservice(UserModule, {
    transport: Transport.TCP,
    options: {
      host: '0.0.0.0',
      port: new ConfigService().get('port'),
    },
  } as TcpOptions);
  await app.listenAsync();
}
bootstrap();
```

Спочатку імпортуються класи фреймворка далі конфігурується та запускається функція bootstrap. В ній викликається функція createMicroservice яка приймає такі налаштування як спосіб зв’язку мікросервіса (transport), хост та порт на якому його буде розташовано. У кінці викликається функція listenAsync яка запускає мікросервіс.

Далі розглянемо файл-модуль user.module.ts.

```
import { Module } from '@nestjs/common';
import { MongooseModule } from '@nestjs/mongoose';
import { UserController } from './user.controller';
import { UserService } from './services/user.service';
import { MongoConfigService } from './services/config/mongo-config.service';
import { ConfigService } from './services/config/config.service';
import { UserSchema } from './schemas/user.schema';
@Module({
  imports: [
    MongooseModule.forRootAsync({
      useClass: MongoConfigService,
    }),
```

```
    MongooseModule.forFeature([
      {
        name: 'User',
        schema: UserSchema,
        collection: 'users',
      },
    ]),
  ],
  controllers: [UserController],
  providers: [
    UserService,
    ConfigService,
  ],
})
export class UserModule {}
```

Створюється пустий клас UserModule який налаштовується директивою @Module. Розглянемо параметри директиви:

- Imports: це перелік імпортованих модулів які використовуються в цьому модулі. В цьому прикладі використовується тільки модуль MongooseModule який суттєво спрощує роботу з базою даних;
- Controllers: це перелік контролерів які обробляють запити на мікросервіс;
- Providers: це перелік сервісів та інших допоміжних класів модуля;

Далі розглянемо файл user.controller.ts:

```
import { Controller, HttpStatus } from '@nestjs/common';
import { MessagePattern } from '@nestjs/microservices';
import { UserService } from './services/user.service';
import { IUser } from './interfaces/user.interface';
import { IUserCreateResponse } from './interfaces/user-create-response.interface';
```

```
import { IUserSearchResponse } from './interfaces/user-search-response.interface';
@Controller('user')
export class UserController {
  constructor(private readonly userService: UserService) {}
  @MessagePattern('user_search_by_credentials')
  public async searchUserByCredentials(searchParams: {
    email: string;
    password: string;
  }): Promise<IUserSearchResponse> {
    let result: IUserSearchResponse;
    if (searchParams.email && searchParams.password) {
      const user = await this.userService.searchUser({
        email: searchParams.email,
      });
      if (user && user[0]) {
        if (await user[0].compareEncryptedPassword(searchParams.password)) {
          result = {
            status: HttpStatus.OK,
            message: 'user_search_by_credentials_success',
            user: user[0],
          };
        } else {
          result = {
            status: HttpStatus.NOT_FOUND,
            message: 'user_search_by_credentials_not_match',
            user: null,
          };
        }
      } else {
```

```
      result = {
        status: HttpStatus.NOT_FOUND,
        message: 'user_search_by_credentials_not_found',
        user: null,
      };
    }
  } else {
    result = {
      status: HttpStatus.NOT_FOUND,
      message: 'user_search_by_credentials_not_found',
      user: null,
    };
  }
  return result;
}
@MessagePattern('user_get_by_id')
public async getUserById(id: string): Promise<IUserSearchResponse> {
  let result: IUserSearchResponse;
  if (id) {
    const user = await this.userService.searchUserById(id);
    if (user) {
      result = {
        status: HttpStatus.OK,
        message: 'user_get_by_id_success',
        user,
      };
    } else {
      result = {
        status: HttpStatus.NOT_FOUND,
```

```
        message: 'user_get_by_id_not_found',
        user: null,
      };
    }
  } else {
    result = {
      status: HttpStatus.BAD_REQUEST,
      message: 'user_get_by_id_bad_request',
      user: null,
    };
  }
  return result;
}
@MessagePattern('user_create')
public async createUser(userParams: IUser): Promise<IUserCreateResponse> {
  let result: IUserCreateResponse;
  if (userParams) {
    const usersWithEmail = await this.userService.searchUser({
      email: userParams.email,
    });
    if (usersWithEmail && usersWithEmail.length > 0) {
      result = {
        status: HttpStatus.CONFLICT,
        message: 'user_create_conflict',
        user: null,
        errors: {
          email: {
            message: 'Email already exists',
            path: 'email',
```

```
        },
      },
    };
  } else {
    try {
      userParams.is_confirmed = false;
      const createdUser = await this.userService.createUser(userParams);
      delete createdUser.password;
      result = {
        status: HttpStatus.CREATED,
        message: 'user_create_success',
        user: createdUser,
        errors: null,
      };
    } catch (e) {
      result = {
        status: HttpStatus.PRECONDITION_FAILED,
        message: 'user_create_precondition_failed',
        user: null,
        errors: e.errors,
      };
    }
  }
} else {
  result = {
    status: HttpStatus.BAD_REQUEST,
    message: 'user_create_bad_request',
    user: null,
    errors: null,
```

```
    };
   }
   return result;
  }
}
```

Створюється клас UserController який позначається директивою @Controller. Клас імпортує сервіс UserService та має декілька функцій які позначаються директивою @MessagePattern. Ця директива є важливою частиною мікросервісної архітектури, вона приймає ключ по якому буде викликатися з інших мікросервісів.

Розглянемо функції контролера:

- searchUserByCredentials: ця функція дозволяє шукати користувача використовуючи його поштову адресу та пароль;
- getUserById: ця функція дозволяє шукати користувача використовуючи його унікальний ідентифікатор ID;
- createUser: ця функція дозволяє створити користувача;
- Розглянемо файл user.service.ts:

```
import { Injectable } from '@nestjs/common';
import { InjectModel } from '@nestjs/mongoose';
import { Model } from 'mongoose';
import { IUser } from '../interfaces/user.interface';
@Injectable()
export class UserService {
 constructor(@InjectModel('User') private readonly userModel: Model<IUser>) {}
 public async searchUser(params: { email: string }): Promise<IUser[]> {
  return this.userModel.find(params).exec();
 }
 public async searchUserById(id: string): Promise<IUser> {
  return this.userModel.findById(id).exec();
```

```
  }
  public async updateUserById(
    id: string,
    userParams: { is_confirmed: boolean },
  ): Promise<IUser> {
    return this.userModel.updateOne({ _id: id }, userParams).exec();
  }
  public async createUser(user: IUser): Promise<IUser> {
    const userModel = new this.userModel(user);
    return await userModel.save();
  }
}
```

Створюється клас UserService з директивою @Injectable яка помічає цей клас як провайдер з подальшою змогою до імпортування. В конструкторі класа імпортується модель бази даних User за допомогою директиви @InjectModel. Всі функції цього сервіса є проміжним пунктом між контролером та базою даних. Всі вони викликаються в контролері та взаємодіють з базою даних. Таким чином вся бізнес логіка програми скривається від контролера що задає гарний архітектурний тон проекту.

Далі розглянемо структуру інтерфейсу на прикладі файлу user.interface.ts

```
import { Document } from 'mongoose';
export interface IUser extends Document {
  id?: string;
  email: string;
  password: string;
  is_confirmed: boolean;
  compareEncryptedPassword: (password: string) => boolean;
  getEncryptedPassword: (password: string) => string;
```

```
}
```

Створюється та експортується інтерфейс IUser який розширюється від класу Document бібліотеці mongoose. Інтерфейс описує поля та функції моделі.

Далі розглянемо структуру схеми на прикладі файлу user.schema.ts

```
import * as mongoose from 'mongoose';
import * as bcrypt from 'bcrypt';
const SALT_ROUNDS = 10;
function transformValue(doc, ret: { [key: string]: any }) {
  delete ret._id;
  delete ret.password;
}
export interface IUserSchema extends mongoose.Document {
  email: string;
  password: string;
  is_confirmed: boolean;
  comparePassword: (password: string) => Promise<boolean>;
  getEncryptedPassword: (password: string) => Promise<string>;
}
export const UserSchema = new mongoose.Schema<IUserSchema>(
  {
    email: {
      type: String,
      required: [true, 'Email can not be empty'],
      match: [
        /^(([^<>()\[\]\\.,;:\s@"]+(\.[^<>()\[\]\\.,;:\s@"]+)*)|(".+"))@((\[[0-9]{1,3}\.[0-
9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\])|(([a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))$/,
        'Email should be valid',
      ],
    },
```

```
    is_confirmed: {
      type: Boolean,
      required: [true, 'Confirmed can not be empty'],
    },
    password: {
      type: String,
      required: [true, 'Password can not be empty'],
      minlength: [6, 'Password should include at least 6 chars'],
    },
  },
  {
    toObject: {
      virtuals: true,
      versionKey: false,
      transform: transformValue,
    },
    toJSON: {
      virtuals: true,
      versionKey: false,
      transform: transformValue,
    },
  },
);
UserSchema.methods.getEncryptedPassword = (
  password: string,
): Promise<string> => {
  return bcrypt.hash(String(password), SALT_ROUNDS);
};
UserSchema.methods.compareEncryptedPassword = function (password: string) {
```

```
  return bcrypt.compare(password, this.password);
};
UserSchema.pre('save', async function (next) {
  if (!this.isModified('password')) {
    return next();
  }
  this.password = await this.getEncryptedPassword(this.password);
  next();
});
```

Якщо інтерфейс описує лише вигляд моделі, то схема описує всю реалізацію полів та функцій. Розглянемо аргументи поля:

- type: показує якого типу є це поле;
- required: показує чи є це поле обов’язковим та яке повідомлення повинна повертати база даних якщо поле відсутнє підчас збереження моделі;
- match: приймає аргумент типу RegExp який валідує значення;
- minlength: мінімальне значення поля;
- maxlength: максимальне значення поля;

3.4 Інструкція з експлуатації

3.4.1 Інструкція для користувача

При вході на сайт у вас з’являться всі присутні точки виклику API.

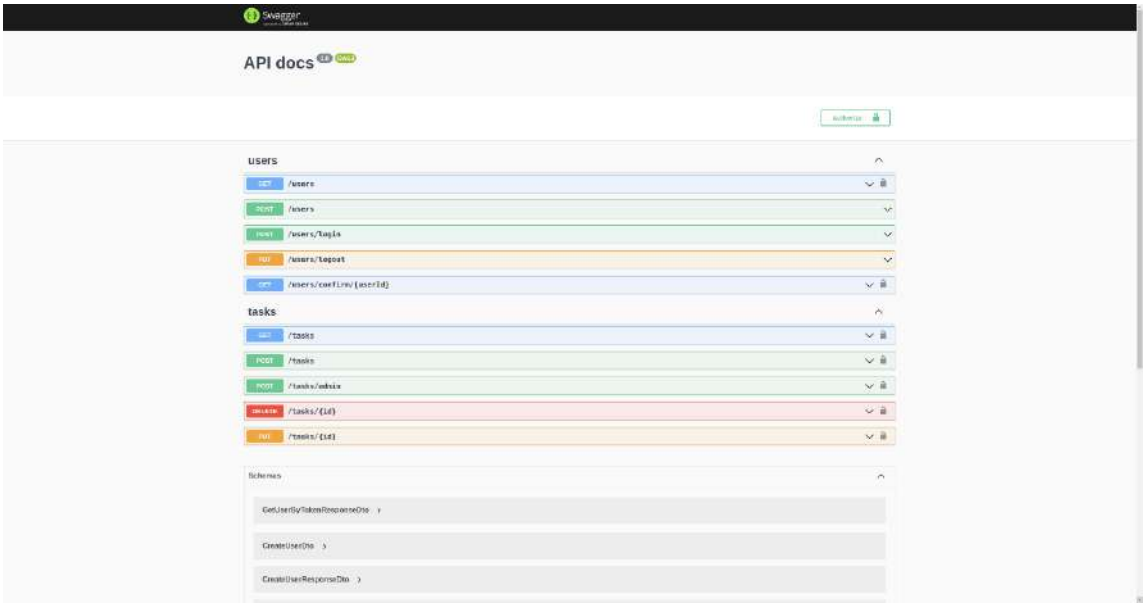

Рис. 3.6 - Перелік точок виклику API.

Як видно, точки поділяються на ті які відносяться до роботи над користувачами і ті які відносяться до роботи над задачами. Почнемо зі створення користувача натиснувши на точку POST /user.

Рис. 3.7 - Форма створення користувача.

Вводимо у форму поштову адресу і бажаний пароль для користувача та натискаємо на кнопку Execute. Після цього Swagger формує та виконує запит на сервер.

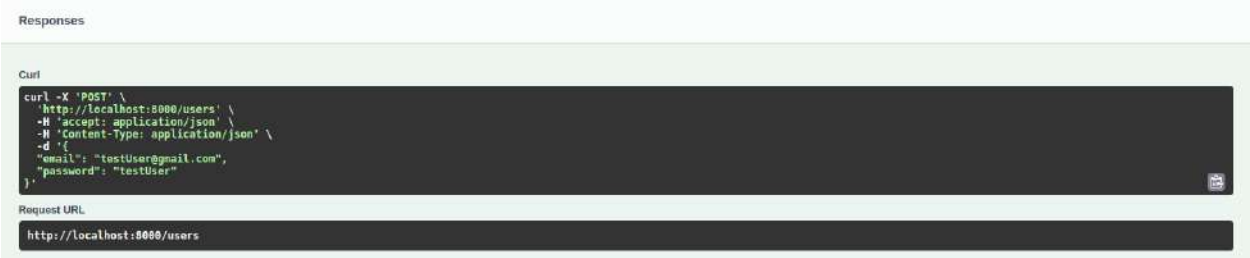

Рис. 3.8 - Сформований запит.

Після запиту ми отримуємо відповідь в який мається поле токену, це поле варто скопіювати для входу до системи.

Рис. 3.9 - Відповідь від сервера.

Для того щоб увійти до системи використовуючи Swagger, потрібно використати токен який повертає сервер. Для цього натискаємо кнопку Authorize нагорі сайту.

Рис. 3.10 - Кнопка Authorize.

Після натискання на кнопку відкривається модальне вікно яке містить поле для вводу токена. Вводимо скопійований токен у поле на натискаємо кнопку Authorize.

Рис. 3.11 - Модальне вікно авторизації.

Тепер можна використовувати точки які потребують авторизації (вони позначаються замком). Створимо для користувача нову задачу.

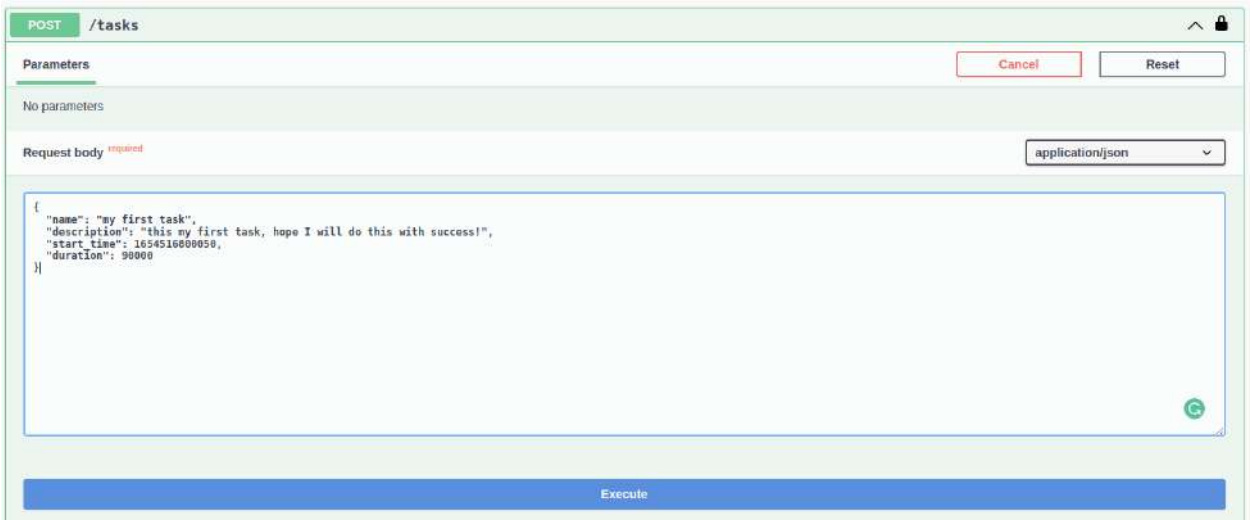

Рис. 3.12 - Форма створення задачі.

Заповнюємо ім’я та опис задачі, початок та тривалість задачі залишаємо як є. Натискаємо Execute та дивимось на відповідь від сервера.

Рис. 3.13 - Відповідь на створення задачі.

Тепер цю задачу можна редагувати, наприклад, позначити її як виконану. Для цього відкриємо форму редагування задачі та заповнимо за нашою потребою.

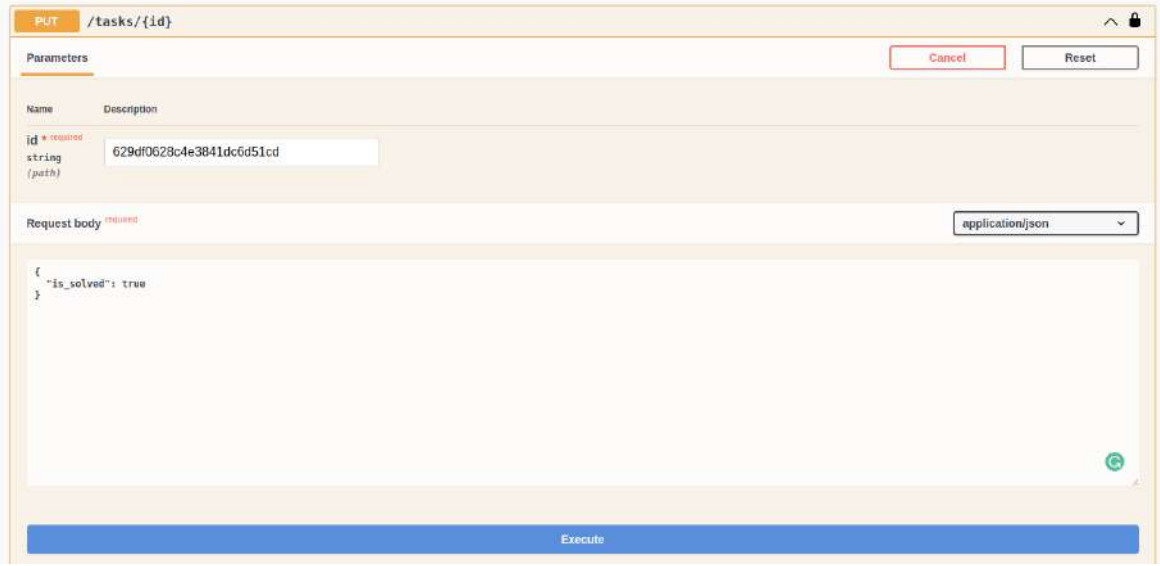

Рис. 3.14 - Форма редагування задачі.

Натискаємо Execute та перевіряємо відповідь від сервера.

Рис. 3.15 - Відповідь на редагування задачі.

Тепер задачу можна видалити, для цього відкриваємо форму видалення задачі на вводимо її унікальний ключ ID.

Рис. 3.16 - Форма видалення задачі.

Натискаємо кнопку Execute та перевіряємо відповідь.

Рис. 3.17 - Відповідь на видалення задачі.

Тепер можна вийти з системи натиснувши кнопку Authorize яку ми натискали для того щоб увійти до системи.

Рис. 3.18 - Модальне вікно в авторизованому стані.

Натискаємо кнопку Logout для того щоб стерти токен та закінчити сесію користувача.

3.4.2 Інструкція для адміністратора

На відміну від звичайного користувача, адміністратор має змогу підтвердити користувача, видалити користувача та створити задачу користувачу.

Для того щоб підтвердити користувача, потрібно відкрити належну форму, ввести унікальний ключ ID користувача та натиснути кнопку Execute.

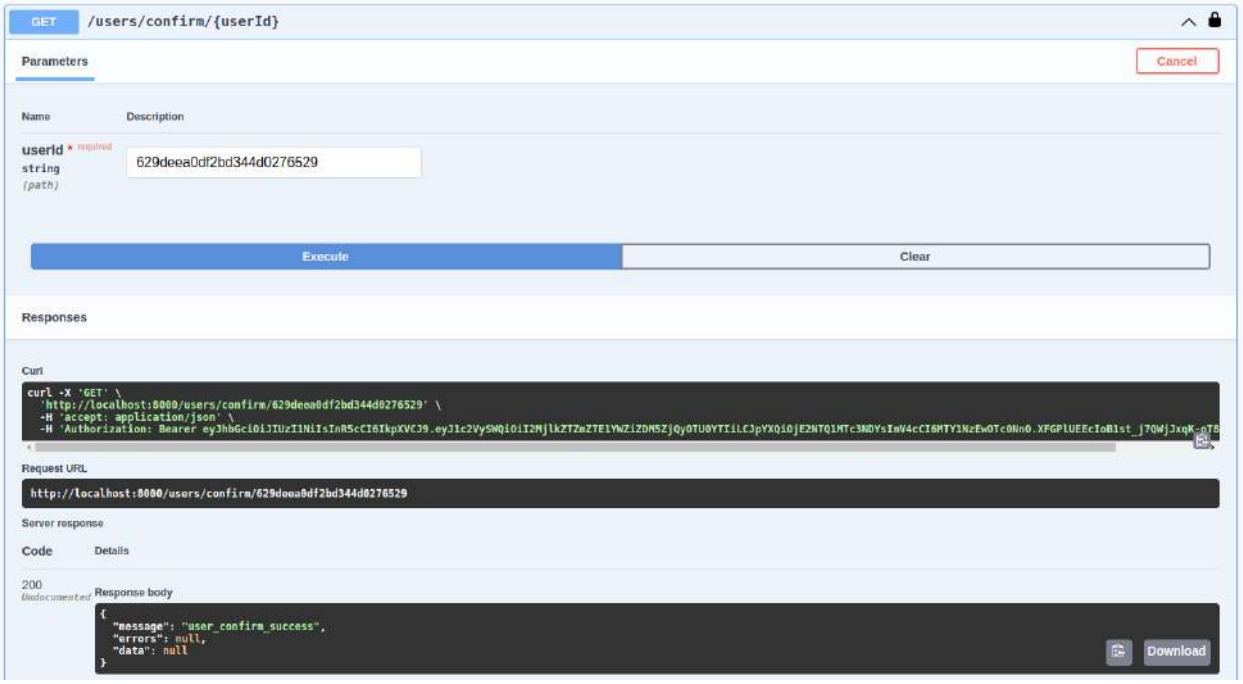

Рис. 3.19 - Форма підтвердження користувача.

Тепер можна створити задачу для існуючого користувача відкривши форму яка позначається шляхом /admin.

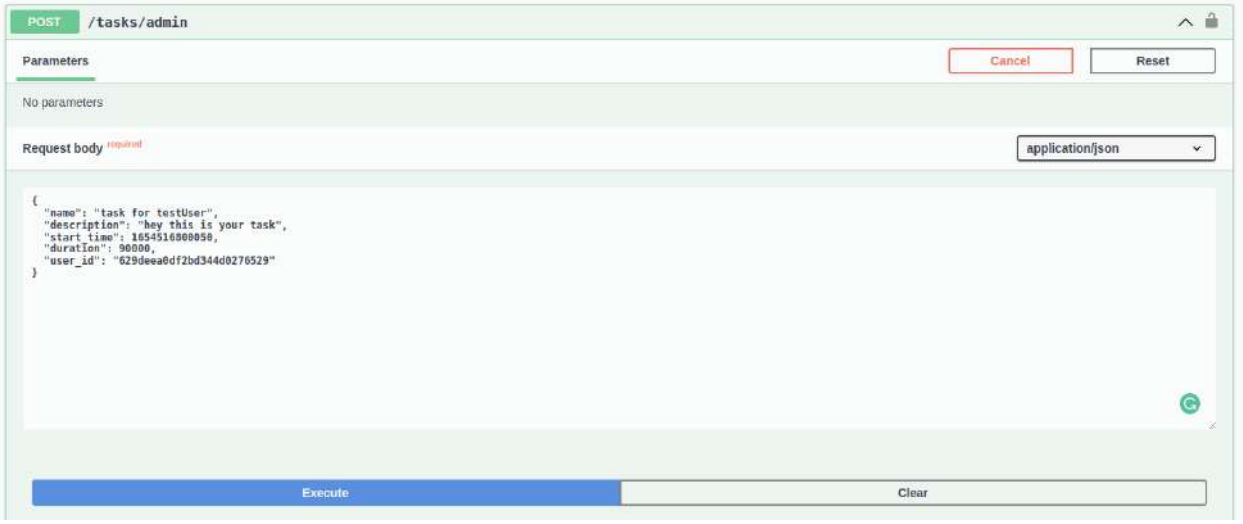

Рис. 3.20 - Форма створення задачі для користувача.

Заповнюємо форму, додаємо унікальний ключ ID користувача та натискаємо кнопку Execute.

3.5 Висновок по розділу

Було запроектовано веб-сервер використовуючи архітектурне рішення з розбиття веб-сервера на мікросервіси. Проект має чотири окремих мікросервіса:

- Сервіс керування користувачами.
- Сервіс керування задачами.
- Сервіс контролю доступів.
- Сервіс роботи з тимчасовими токенами.

Розроблений веб-сервер має функції створення користувачів, їх підтвердження, керування їхніми правами та робота над задачами, їх створення, редагування та видалення.

ВИСНОВКИ

В час коли створення невеликих хмарних сервісів набирає популярність, веб-сервер на базі мікросервісів є вдалим архітектурним рішенням. Саму тому темою моєї роботи став веб-сервер на базі мікросервісної архітектури, який має функціонал:

- Створення та авторизація користувачів.
- Керування правами користувачів.
- Створення, редагування та видалення задач.
- Підтвердження користувачів адміністратором.

При розробці системи було використано фреймворк NestJS на мові програмування TypeScript для створення та під’єднання мікросервісів. Front-end сайту був створений з допомогою Swagger. Системою управління базами даних було обрана MongoDB.

Було ході роботи було розроблено чотири мікросервіси:

- Сервіс користувачів.
- Сервіс задач.
- Сервіс доступів.
- Сервіс токенів.

ПЕРЕЛІК ПОСИЛАНЬ

1. Кристиан, Хорсдал Мікросервіси на платформі NestJS. К.: Пітер, 2018. - 462 с.
2. Ньюмен, Сэм Створення мікросервісів / Сэм Ньюмен. - М.: Питер, 2015. - 182 с.
3. Francia MongoDB and NestJS / Francia. - Москва: Огни, 2012. - 570 с.
4. O'Higgins MongoDB and Python / O'Higgins. - Москва: Мир, 2011. - 463 с.
5. Федоров, А.Г. JavaScript для всех / А.Г. Федоров. - М.: Компьютер-пресс, 1998. - 384 с.
6. Alexander, J. Vincent JavaScript Developer's Dictionary / Alexander J. Vincent. - Москва: Наука, 1978. - 943 с.
7. Danny, Goodman JavaScript Bible, 4th Edition / Danny Goodman. - Москва: Огни, 2002. - 603 с.
8. Don, Gosselin JavaScript - Introductory, Second Edition / Don Gosselin. - Москва: Огни, 2009. - 182 с.
9. Daniel, Koch Mastering TypeScript / Daniel Koch. - М.: Книга по Требованию, 2006. - 404 с.
10. Documentation | NestJS – A progressive Node.js framework. [Електронний ресурс] – Режим доступу: www. URL: https://docs.nestjs.com/ (дата звернення: 20.12.2021)
11. Swagger Documentation. [Електронний ресурс] – Режим доступу: www. URL: https://swagger.io/docs/ (дата звернення: 21.02.2022)
12. MongoDB Documentation. [Електронний ресурс] – Режим доступу: www. URL: https://www.mongodb.com/docs/ (дата звернення: 16.02.2022)
13. TypeScript: The starting point for learning TypeScript. [Електронний ресурс] – Режим доступу: www. URL: https://www.typescriptlang.org/docs/ (дата звернення: 23.12.2022)

14. RFC 793 – Transmission Control Protocol. [Електронний ресурс] – Режим доступу: www. URL: https://datatracker.ietf.org/doc/html/rfc793 (дата звернення: 20.12.2021)

15. Мікросервіси (Microservices). [Електронний ресурс] – Режим доступу: www. URL: https://habr.com/ru/post/249183/ (дата звернення: 19.12.2021)

ДОДАТОК А

Вихідний код програми

authorization.decorator.ts

```
import { SetMetadata } from '@nestjs/common';
export const Authorization = (secured: boolean)
=>
  SetMetadata('secured', secured);
```

permission.decorator.ts

```
import { SetMetadata } from '@nestjs/common';
export const Permission = (permission: string)
=>
  SetMetadata('permission', permission);
```

authorized-request.interface.ts

```
import { IUser } from '../user/user.interface';
export interface IAuthorizedRequest extends
Request {
  user?: IUser;
}
```

create-task.dto.ts

```
import { ApiProperty } from '@nestjs/swagger';
export class CreateTaskDto {
  @ApiProperty({ example: 'test task' })
  name: string;
  @ApiProperty({ example: 'test task description'
})
  description: string;
  @ApiProperty({ example: +new Date() })
  start_time: number;
  @ApiProperty({ example: 90000 })
  duration: number;
}
```

create-task-admin.dto.ts

```
import { ApiProperty } from '@nestjs/swagger';
export class CreateTaskAdminDto {
  @ApiProperty({ example: 'test task' })
  name: string;
  @ApiProperty({ example: 'test task description'
})
  description: string;
  @ApiProperty({ example: +new Date() })
  start_time: number;
  @ApiProperty({ example: 90000 })
  duration: number;
  @ApiProperty({ example:
'629224c922fb9861b2ad7cd4' })
  user_id: string;
}
```

create-task-response.dto.ts

```
import { ApiProperty } from '@nestjs/swagger';
import { ITask } from '../task.interface';
export class CreateTaskResponseDto {
  @ApiProperty({ example:
'task_create_success' })
  message: string;
  @ApiProperty({
    example: {
      task: {
        notification_id: null,
        name: 'test task',
        description: 'test task description',
        start_time: +new Date(),
```