

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій

(повне найменування факультету)

Кафедра програмних засобів

(повне найменування кафедри)

Пояснювальна записка

до дипломного проекту (роботи)

магістр

(ступінь вищої освіти)

на тему АНАЛІЗ МЕТОДІВ ТА РЕАЛІЗАЦІЯ СУЧАСНОГО СТЕКУ

(назва теми)

ТЕХНОЛОГІЙ ДЛЯ ПІДВИЩЕННЯ ПРОДУКТИВНОСТІ ВЕБ САЙТІВ

ANALYSIS OF METHODS AND IMPLEMENTATION OF A MODERN

TECHNOLOGY STACK FOR INCREASING THE PRODUCTIVITY OF

WEBSITES

Виконав: студент(ка) 2 курсу, групи КНТ-223м

Спеціальності 122 Комп'ютерні науки

(код і найменування спеціальності)

Освітня програма (спеціалізація)

Комп'ютерні науки

ШТЕПА І.А.

(ПРИЗВИЩЕ та ініціали)

Керівник ТАБУНЩИК Г.В.

(ПРИЗВИЩЕ та ініціали)

Рецензент ШИТІКОВА О.В.

(ПРИЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
(повне найменування закладу вищої освіти)

Факультет комп'ютерних наук та технологій
Кафедра програмних засобів
Ступінь вищої освіти магістр
Спеціальність 122 Комп'ютерні науки
(код і найменування)
Освітня програма (спеціалізація) Системи штучного інтелекту
(назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗ, д.т.н, проф.
Сергій СУББОТІН
«_____» _____ 2024 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА

ШТЕПИ Івана Андрійовича

(ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Аналіз методів та реалізація сучасного стеку технологій для підвищення продуктивності веб сайтів. Analysis of Methods and Implementation of a Modern Technology Stack for Increasing the Productivity of Websites.

керівник проєкту (роботи) к.т.н., професор ТАБУНЩИК Галина Володимирівна,
(науковий ступінь, вчене звання, ПРИЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від "07" жовтня 2024 року № 398

2. Строк подання студентом проєкту (роботи) 3 грудня 2024 року

3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завдання.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз предметної області. 2. Налаштування CORS для безпечного доступу до API. 3 Використання PM2 для моніторингу та управління процесами на сервері. 4. Забезпечення безпеки та масштабованості веб-додатку. 5. Тестування веб-сайту.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів) Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРИЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4 Основна частина	ТАБУНЩИК Г.В., професор		
Нормоконтролер	КАМІНСЬКА Ж.К., асист.		

7. Дата видачі завдання «04» вересня 2024 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи та підбір літератури.	1-2 тижні	Завдання, ТЗ
2	Аналіз предметної області.	3-4 тижні	Розділ 1
3	Вибір програмного комплексу.	5-6 тижні	Розділ 2
4	Розробка алгоритму та написання програм.	7-8 тижні	Розділ 3
5	Проведення аналізу отриманих даних.	9 тиждень	Розділ 4
6	Оформлення пояснювальної записки та документів до неї.	10 тиждень	ПЗ, Додатки
7	Нормоконтроль та рецензування.	11 тиждень	
8	Захист роботи.	12 тиждень	

Студент(ка)

(підпис) Іван ШТЕПА
(Ім'я ПРІЗВИЩЕ)

Керівник проєкту (роботи)

Галина ТАБУНЩИК
(Ім'я ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи магістра:
129 с., 41 рис., 9 дод., 52 джерел.

ОПТИМІЗАЦІЯ ПІД НАВАНТАЖЕННЯ, NODE.JS, JAVASCRIPT,
MONGODB, CLUSTERS, ATLAS, BACKEND.

Мета дипломної роботи – підвищення продуктивності та масштабованості вебсайтів за рахунок використання сучасного стеку технологій.

Об’єкт дослідження – архітектура та інструменти оптимізації вебдодатків.

Предмет дослідження – сучасні технології серверної розробки, включаючи Node.js, MongoDB, Express.js, та інструменти оптимізації роботи серверних процесів.

Методи дослідження – аналіз архітектурних патернів, зокрема MVC-моделі, дослідження методів кластеризації процесів із використанням модуля cluster у Node.js, експериментальне тестування баз даних MongoDB та PostgreSQL, моделювання сценаріїв високих навантажень для оцінки ефективності системи, а також тестування алгоритмів балансування навантаження.

Результати. створено вебдодаток із MVC, кластеризацією та CORS; протестовано ефективність роботи серверів у режимі високих навантажень; доведено практичну придатність для навчальних платформ, електронної комерції та CRM-систем.

Висновки. реалізовано ефективний вебдодаток із використанням Node.js, MongoDB і PM2, підтверджено його продуктивність.

Галузь використання – навчальні платформи, електронна комерція, CRM-системи.

ABSTRACT

Explanatory note to the master's thesis: 129 p., 41 figures, 9 appendix, 52 sources.

LOAD OPTIMIZATION, NODE.JS, JAVASCRIPT, MONGODB, CLUSTERS, ATLAS, BACKEND.

The purpose of the thesis is to improve the performance and scalability of websites by using a modern technology stack.

The object of research is the architecture and tools for optimizing web applications.

The subject of the research is modern server-side development technologies, including Node.js, MongoDB, Express.js, and tools for optimizing server processes.

Research methods - analysis of architectural patterns, in particular MVC-models, research of methods of clustering processes using the cluster module in Node.js, experimental testing of MongoDB and PostgreSQL databases, modeling of high-load scenarios to assess system efficiency, as well as testing of load balancing algorithms.

Results. a web application with MVC, clustering and CORS has been created; server performance under high loads has been tested; practical applicability for learning platforms, e-commerce and CRM systems has been proven.

Conclusions. An effective web application using Node.js, MongoDB and PM2 has been implemented and its performance has been confirmed.

The field of application is educational platforms, e-commerce, CRM systems.

ЗМІСТ

	С.
Перелік скорочень та умовних познач.....	8
Вступ.....	9
1 Аналіз предметної області.....	10
1.1 Актуальність сучасних технологій для підвищення продуктивності вебсайтів.....	10
1.2 Сучасні методи оптимізації продуктивності вебсайтів.....	10
1.3 Порівняння інструментів і платформ.....	11
1.4 Аналіз трендів у веброботі.....	11
1.5 Постановка завдань дипломної роботи.....	12
2 Налаштування середовища розробника.....	14
2.1 Завантаження Node.JS.....	14
2.2 Завантаження Visual Code	14
2.3 Різниця Backend та Frontend.....	15
2.4 Що виконує Node JS.....	16
2.5 Встановлення інструментів NPM nodemon	19
2.6 Приклад сайту як демонстрація сучасного стеку технологій.....	20
2.7 Висновки до розділу 2	22
3 Вебсервер	23
3.1 Процес роботи вебсервера.....	23
3.2 Крос-доменний обмін ресурсами	24
3.3 Архітектурний патерн MVC	25
3.4 Контролер представлення моделі в Експрес	27
3.5 Проміжне програмне забезпечення	28
3.6 Висновки до розділу 3	29
4 Підвищення продуктивності Node	30
4.1 Створення простого блокуючого сервера.....	30
4.2 Функції блокування в реальному житті.....	32

4.3 Запуск декількох Node процесів	36
4.4 Модуль кластера Node	38
4.5 Кластеризація в дії	40
4.6 Максимізація продуктивності кластера	44
4.7 Балансування навантаження	47
4.8 Node Server продуктивність	49
4.9 PM2 інструмент	51
4.10 Висновки до розділу 4	52
5 Бази даних	54
5.1 Порівняння SQL та NoSQL баз даних.....	54
5.2 Налаштування MongoDB Atlas	56
5.3 Висновки до розділу 5	59
Висновки	61
Перелік джерел посилань	63
Додаток А Структура папки сервер	68
Додаток Б Структура папки клієнт.....	71
Додаток В Код проєкту структурований у стилі MVC (внутрішня частина)	74
Додаток Г Код проєкту структурований у стилі MVC (зовнішня частина).....	92
Додаток Д Приклад налаштування проміжного програмного забезпечення CORS	108
Додаток Е Використання PM2 для створення кластерів.....	112
Додаток Ж Налаштування MongoDB Atlas	115
Додаток И Приклад продуктивності	118
Додаток К Реалізація кластеризації.....	120
Додаток Л Слайди презентації.....	1203

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

API	– Application Programming Interface;
CORS	– Cross-Origin Resource Sharing;
CRUD	– Create, Read, Update, Delete;
DBMS	– Database Management System;
MVC	– Model-View-Controller;
PM2	– Process Manager 2;
REST	– Representational State Transfer;
SPA	– Single Page Application;
SSR	– Server-Side Rendering;

ВСТУП

Розробка інтерактивних рішень для багатьох аспектів життя стала можливою завдяки вебтехнологіям, які є невід'ємною складовою сучасного світу. Проблема розробки надійної, масштабованої та оптимальної архітектури стає все більш актуальною через зростання обсягу даних, бази користувачів та вимог до продуктивності вебдодатків.

Розробка інтерактивного вебсайту для симуляції запуску ракет є основною метою цієї дипломної роботи, і він побудований з використанням сучасного технологічного стеку, який включає Node.js, Express.js та MongoDB. Метою проекту є створення масштабованого та високопродуктивного рішення, яке гарантує стабільну роботу навіть під великими навантаженнями. Це досягається завдяки використанню складних методів організації коду на основі архітектури MVC та ефективному управлінню кластерами серверів за допомогою PM2 [1].

Налаштування CORS для безпечного доступу до API, використання Mongoose для організації взаємодії з базами даних, а також використання таких інструментів, як PM2 для моніторингу та управління процесами – ось деякі з важливих аспектів гарантування продуктивності та безпеки онлайн-додатків, які висвітлені в цій презентації.

Дослідження також включає вивчення архітектурних стратегій для включення складних даних і гарантування справжньої інтерактивності додатку. Для ілюстрації того, як працює система, в якості прикладу використовується інтерактивна симуляція запуску ракети, що дозволяє користувачам взаємодіяти з вебдодатком в режимі реального часу.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Актуальність сучасних технологій для підвищення продуктивності вебсайтів

У сучасному світі вебсайти є основним засобом взаємодії між компаніями, організаціями та користувачами. Щоденно зростає кількість вебдодатків, які потребують високої продуктивності, масштабованості та безпеки. За даними Google, швидкість завантаження сторінки є одним із ключових факторів, що впливає на поведінку користувачів: понад 53% користувачів мобільних пристроїв покидають сайт, якщо його завантаження займає більше ніж 3 секунди.

Враховуючи ці виклики, розробники орієнтуються на сучасні технології, які дозволяють забезпечити високу продуктивність вебдодатків, навіть у випадках великого навантаження. Впровадження ефективних методів оптимізації забезпечує не лише швидкість роботи вебсайту, але й сприяє підвищенню рейтингу у пошукових системах, покращенню взаємодії з користувачами та збільшенню конверсій.

1.2 Сучасні методи оптимізації продуктивності вебсайтів

Використання мережі доставки контенту. Content Delivery Network (CDN) – це глобально розподілена мережа серверів, яка дозволяє зберігати копії статичних ресурсів вебсайту ближче до кінцевих користувачів. Це значно зменшує час завантаження сторінок, оскільки запити не потрібно передавати на центральний сервер. Основні переваги CDN полягають в швидкості доставки контенту, зниженні навантаження на сервер, захисті від DDoS-атак через розподілену архітектуру.

Оптимізація зображень і ресурсів. Оптимізація зображень є одним із найпростіших і водночас найефективніших способів зменшення розміру вебсторінок. Інструменти, такі як TinyPNG, ImageOptim, та формати WebP,

допомагають знизити розмір файлів без втрати якості. Окрім цього, оптимізація JavaScript, CSS і використання мінімізації файлів значно пришвидшують рендеринг сторінок.

Асинхронне завантаження та кешування. Асинхронне завантаження дозволяє браузеру обробляти окремі частини сторінки незалежно одна від одної. Це забезпечує швидший доступ до видимої частини сторінки. Кешування даних як на стороні клієнта (Local Storage, Cache API), так і на стороні сервера (Redis, Memcached) мінімізує кількість повторних запитів, що пришвидшує роботу вебдодатків.

1.3 Порівняння інструментів і платформ

Платформи для backend. Node.js [2]. Завдяки асинхронній обробці запитів та рушію V8 від Google, Node.js є одним із найефективніших рішень для створення масштабованих серверів.

Deno. Новий конкурент Node.js, який усуває низку недоліків Node, таких як потреба в менеджері пакетів npm [3].

Python (Django, Flask). Хоча Python і є менш продуктивним у реальному часі, його переваги включають простоту і багатий набір бібліотек [4].

Бази даних. MongoDB. Документоорієнтована NoSQL база даних, що дозволяє обробляти великі обсяги неструктурованих даних. Ідеальна для динамічних додатків [5].

PostgreSQL. Реляційна база даних із потужною підтримкою транзакцій та складних запитів [6].

MySQL. Традиційна реляційна база даних, яка залишається популярною завдяки стабільності та підтримці широкого спектра інструментів [7].

1.4 Аналіз трендів у веброзробці

Зростання популярності SPA (Single Page Applications). Використання

фреймворків, таких як React, Angular та Vue.js, дозволяє створювати інтерактивні односторінкові додатки.

Server-Side Rendering (SSR) забезпечує швидке завантаження сторінок і покращує SEO, що важливо для комерційних проектів.

Мікросервісна архітектура дає змогу розробляти окремі компоненти додатку незалежно один від одного, що підвищує його гнучкість і масштабованість.

Автоматизація та CI/CD – інструменти, як-от Jenkins, GitLab CI/CD, допомагають у прискоренні процесу розгортання та тестування.

1.5 Постановка завдань дипломної роботи

У даному розділі проведено аналіз сучасних технологій для підвищення продуктивності вебсайтів.

Підхід, запропонований у даній роботі, базується на використанні Node.js, MongoDB та інших сучасних технологій, які дозволяють ефективно оптимізувати продуктивність вебсайтів. Вибрані інструменти забезпечують високу швидкодію, масштабованість та безпеку. Крім того, запропоновані методи відповідають сучасним трендам у веброзробці, таким як кластеризація серверів і побудова архітектури на основі мікросервісів. Тому, тема роботи є актуальною.

Отже, актуальною задачею є забезпечення високої продуктивності та масштабованості вебсайтів через використання сучасного технологічного стеку, зокрема Node.js, MongoDB, та інших інструментів. Це дозволяє ефективно управляти великими обсягами даних, забезпечувати високу швидкість завантаження та адаптивність до високих навантажень.

Мета роботи – створення вебдодатку з використанням сучасного стеку технологій для забезпечення високої продуктивності та масштабованості вебсайтів.

Для досягнення поставленої мети необхідно розв'язати наступні

завдання:

- дослідити існуючі алгоритми роботи серверів Node.js та їх оптимізацію для забезпечення високої продуктивності;
- розробити ефективні методи кластеризації та балансування навантаження;
- провести порівняльний аналіз баз даних MongoDB та PostgreSQL для вибору оптимальних рішень для різних типів даних;
- розробити вебдодаток з використанням Node.js, Express.js та MongoDB для забезпечення масштабованості та високої доступності;
- реалізувати налаштування CORS для безпечного доступу до API.

2 НАЛАШТУВАННЯ СЕРЕДОВИЩА РОЗРОБНИКА

2.1 Завантаження Node.JS [8]

Для того, щоб розпочати роботу з Node.js, вам потрібно його завантажити та встановити на свій комп'ютер. Node.js доступний для Windows, macOS та Linux. Щоб завантажити Node.js, відвідайте офіційний вебсайт nodejs.org і виберіть потрібну версію:

- Long-Term Support, або LTS – рекомендований для більшості користувачів, оскільки він стабільний і підтримується протягом тривалого періоду часу;
- Current – має нові функції, але може бути нестабільний. Це хороший вибір для розробників, які хочуть працювати з новими функціями Node.js.

Після завантаження файлу інсталяції потрібно запустити інсталятор і дотримуватися інструкцій майстра встановлення. Зазвичай під час інсталяції додається менеджер пакетів npm, який необхідний для роботи з бібліотеками та модулями Node.js.

Після завершення встановлення можна виконати команду, щоб перевірити, чи працює Node.js `node -v`.

2.2 Завантаження Visual Code

Завантажте та встановіть Visual Studio Code (VS Code) [9]. VS Code – це популярний текстовий редактор для розробників, який підтримує широкий спектр інструментів і мов програмування. Він доступний безкоштовно та працює на операційних системах Windows, macOS і Linux.

Для того, щоб завантажити VS Code, виконайте ці дії:

- перейдіть на офіційний вебсайт Visual Studio Code, який можна знайти за адресою code.visualstudio.com;
- виберіть потрібну версію операційної системи на головній сторінці: Windows macOS Linux у форматах deb, rpm і tar.gz;

- завантажте установчий файл.

Встановлення на Windows:

- запустіть інсталятор, який було завантажено;
- взяти до уваги умови ліцензії;
- визначте місце встановлення та необхідні компоненти. Для зручного запуску з командного рядка рекомендуємо додати VS Code до PATH;
- дочекайтеся завершення процесу після натискання «Install».

Після завершення встановлення відкрийте VS Code, щоб переконатися, що він працює. Для роботи з Node.js встановіть розширення Node.js Extension Pack. Цей пакет розширень додає підсвічування синтаксису, автодоповнення та інші корисні функції, що значно спрощує розробку.

2.3 Різниця Backend та Frontend

Різниця між frontend і backend [10]. Frontend і backend є двома основними компонентами розробки вебзастосунків. Хоча вони тісно пов'язані між собою, вони мають різні функції та обов'язки. Для створення ефективного застосування необхідно знати, як вони відрізняються.

Frontend (клієнтська частина) – це інтерфейс користувача, який бачить користувач. Користувач взаємодіє з програмою через передній сторінку.

Ключові елементи frontend це HTML, CSS та JavaScript – це мови програмування.

Можливі фреймворки React, Angular і Vue.JS

Відображення інтерфейсу (UI) є основним завданням.

Респонсивність – здатність адаптуватися до різних пристроїв забезпечити інтерактивність за допомогою інтерактивних меню, кнопок і модальних вікон. Прикладом є кнопка на вебсайті, яка змінює колір, коли ви натискаєте її.

Головні елементи backend: Node.js, Python, Java та PHP – це популярні мови програмування. MongoDB, PostgreSQL і MySQL – це основні бази даних. Основні обов'язки: обробка запитів клієнтів. Збереження, отримання та обробка

інформації з баз даних. Безпека (автентифікація, авторизація).

Як приклад можна зробити натискання кнопки дозволяє отримати список товарів із бази даних.

Node.js для backend Node.js – це платформа, яка використовується для створення серверної частини вебзастосунків. Вона дозволяє створювати високопродуктивні JavaScript-застосунки [11].

Чому Node.js є хорошим вибором для backend?

Для роботи в асинхронному режимі, обробка кількох запитів одночасно. Сьогоднішні технології включають GraphQL, MongoDB і WebSockets. Швидкість забезпечується рушієм V8. Можливість використовувати одну мову як для frontend, так і для backend JavaScript. У frontend і backend використовується API для взаємодії. API контролює, як клієнтська частина надсилає запити на сервер і отримує відповіді [12].

Користувач тисне на кнопку «Пошук». Frontend може надіслати запит на сервер за допомогою REST API або GraphQL. Backend обробляє запит, звертається до бази даних, наприклад MongoDB, і повертає результати [13].

У процесі отримання даних frontend відображає їх на екрані.

Щоб створити зручний і функціональний застосунок, frontend і backend працюють разом. Розуміння того, як вони відрізняються, дозволяє створювати ефективні плани та оптимізувати роботу. Створення сучасних вебрішень, які відповідають вимогам користувачів і бізнесу, легше з використанням інструментів, таких як Node.js, GraphQL і MongoDB.

2.4 Що виконує Node JS

Вхідні дані – це JavaScript-файл з міткою «JS», який містить інструкції Node.js. Цей файл містить серверний код, який можна використовувати для доступу до файлової системи, побудови HTTP-сервера, зв'язку з базами даних тощо.

Рушій JavaScript Node.js виконує JavaScript-код за допомогою потужного

движка V8, який також використовується браузером Google Chrome.

Код JavaScript перетворюється на швидкий машинний код - тобто зрозумілі комп'ютеру інструкції - за допомогою движка V8 [14].

API Node.js з наступними функціями може працювати за допомогою вбудованих API (інтерфейсів прикладного програмування), які пропонує Node.js: fs це доступ до файлової системи, http це створення серверів для HTTP; path маніпулювання шляхами до файлів, crypto це криптографічні процеси (шифрування, хешування).

Функціональність JavaScript значно розширюється завдяки цим API, які дозволяють взаємодіяти з ресурсами сервера та операційної системи.

Прив'язки для Node.js слугує частина «містком», що з'єднує бібліотеку libuv та API Node.js.

Функції прив'язок – це коли виклики перетворюють Node.js API в низькорівневі команди, які виконуються libuv.

Libuv – це бібліотека яка надає Node.js асинхронну, керовану подіями архітектуру. Вона керує читанням файлів та взаємодією з сокетом - прикладами операцій вводу/виводу (I/O) [15].

Неблокуючий режим обробки подій. Висновок щодо інтерфейсу, результати надсилаються користувачеві після обробки запитів і виконання відповідних інструкцій. Як приклада можна навести як веббраузер відображає HTML-сторінку а дані передаються до мобільного застосування.

На рис. 2.1 продемонстровано роботу Node.js і те, як він обробляє JavaScript-код для створення серверних застосунків.

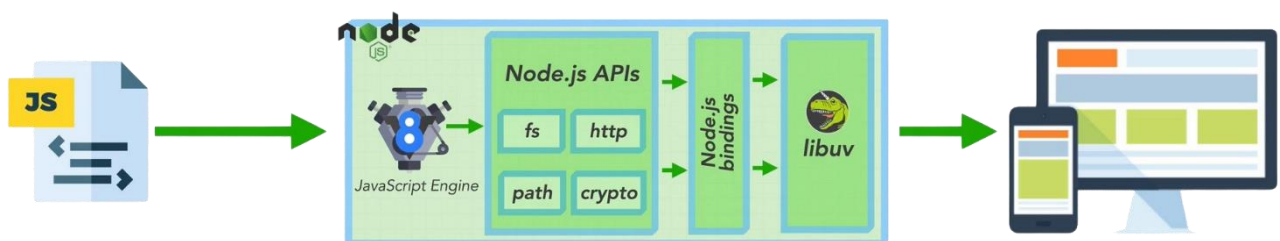


Рисунок 2.1 – Робота Node.js

Багатопоточність Node.js, на відміну від деяких інших серверних середовищ, Node.js не є багатопотоковим у звичайному розумінні. Скоріше, він використовує однопотокову, керовану подіями конструкцію, яка дозволяє ефективно керувати численними запитами без запуску нових потоків для кожного з них. Однак це не виключає використання багатьох потоків у Node.js.

У Node.js існує цикл подій (Event Loop), який, як показано на зображенні в центрі, керує асинхронними операціями. Цикл подій працює в одному потоці та обробляє завдання, такі як HTTP-запити, таймери та зворотні виклики. Це гарантує безперешкодну роботу та розподіл складних завдань на фон.

Пул потоків (Thread Pool).

Node.js має пул потоків, керований бібліотекою libuv. Наприклад, він використовується для виконання деяких операцій, які не можна виконати асинхронно за допомогою операційних систем.

Операції з файловими системами.

Виконання криптографічних операцій, таких як шифрування стиснення або зберігання файлів. Зазвичай пул потоків має чотири потоки за замовчуванням. Однак ви можете змінити кількість потоків, використовуючи змінну середовища `UV_THREADPOOL_SIZE`.

Асинхронний ввід/вивід (Asynchronous I/O). Node.js використовує асинхронний ввід/вивід для таких задач, робота з мережею (Network); доступ до баз даних; виконання тривалих запитів без блокування основного потоку.

Worker Threads. В Node.js було додано підтримку Worker Threads, які дозволяють створювати окремі потоки JavaScript-коду. Це корисно для виконання важких обчислень; паралельного виконання завдань, які не можуть бути оброблені циклом подій [16].

На рис. 2.2 можна побачити цикл подій (Event Loop) – основний механізм обробки задач. Пул потоків (Thread Pool) – для виконання тривалих або блокуючих операцій. Асинхронний ввід/вивід (Asynchronous I/O) – ефективна робота з файловою системою та мережею.

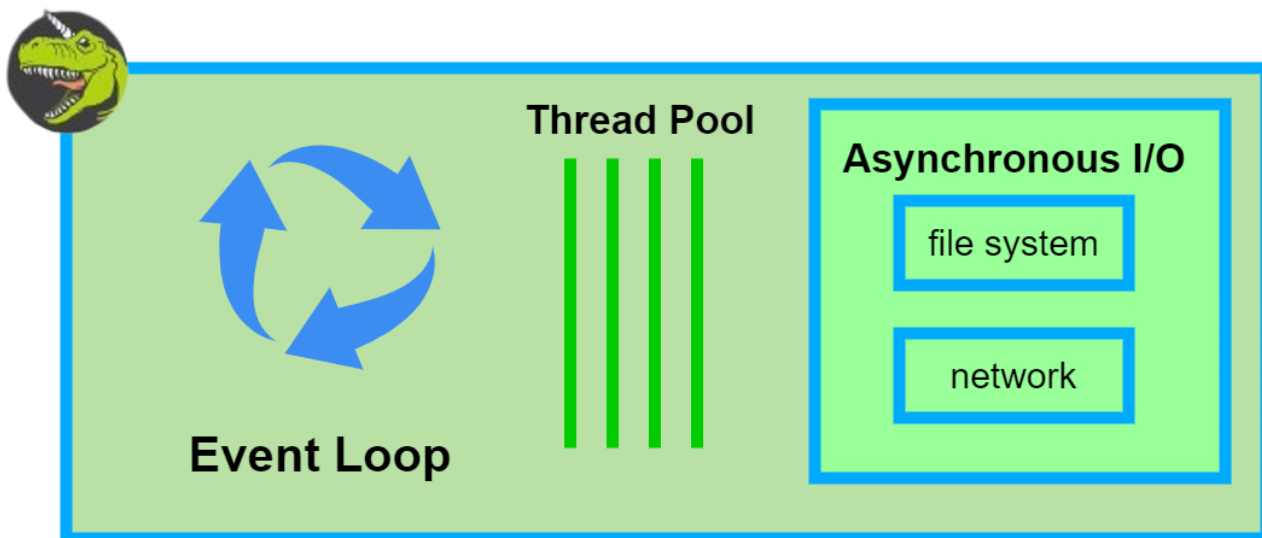


Рисунок 2.2 – Архітектура Node.js. Цикл подій, пул потоків і асинхронний ввід/вивід

2.5 Встановлення інструментів NPM nodemon

Nodemon – це утиліта, яка автоматично перезапускає Node.js додаток при внесенні змін у вихідний код. Це особливо корисно під час розробки, оскільки розробникам не потрібно вручну зупиняти та знову запускати сервер після кожного оновлення файлів. nodemon спрощує процес розробки та підвищує продуктивність.

Основні переваги є автоматичний моніторинг змін у файлах, перезапуск сервера без втручання розробника, підтримка різних типів файлів (JS, JSON, HTML тощо), зменшення рутинної роботи.

Щоб встановити nodemon, потрібно використовувати менеджер пакетів npm. Є два варіанти встановлення.

Глобальне встановлення дозволяє використовувати nodemon в будь-якому проекті без необхідності встановлювати його в кожному окремо [17].

```
npm install -g nodemon
```

Після встановлення можна перевірити версію nodemon, `nodemon --version`.

Використання у package.json. Додаємо скрипт для запуску серверу

```
{ "scripts": { "start": "node app.js", "dev": "nodemon app.js"
} }
```

Запуск через термінал. Замість звичайного `node app.js` використовуємо `nodemon app.js`.

Запуск через `npm scripts`. Якщо було додано `nodemon` у `package.json` як показано вище, запусить сервер у режимі розробки командою `npm run dev`.

Переваги використання `nodemon` під час роботи з `Node.js`.

Розробники можуть економити час, автоматизуючи рутинні завдання.

Можливість відстежувати зміни в будь-яких файлах, навіть якщо вони розташовані в різних директоріях.

У процесі створення динамічних вебдодатків забезпечує зручний робочий процес.

Розробка додатків `Node.js` стає легшою та ефективнішою з `nodemon`.

2.6 Приклад сайту як демонстрація сучасного стеку технологій

Прикладом є запуск симуляції старту ракет, обирати планети, придатні для колонізації людиною, та моніторити параметри космічної місії в режимі реального часу [18].

В умовах стрімкого розвитку вебтехнологій створення вебдодатків, які об'єднують складні алгоритми аналізу даних та інтерактивність користувацького інтерфейсу, є важливим завданням. У цій главі розглядається приклад вебдодатку, який симулює запуск ракети, обирає цільові планети, придатні для життя людини, та дозволяє відстежувати параметри місії в режимі реального часу.

Метою створення вебдодатку є демонстрація можливостей сучасного стеку технологій (`Node.js`, `Express`, `MongoDB`, `React.js`) для обробки великих обсягів даних, побудови серверної логіки та інтерактивного відображення даних у реальному часі. Основними функціональними можливостями додатку є аналіз даних про екзопланети на основі реального набору даних місії `Kepler`, визначення планет, придатних для життя, за допомогою заданих параметрів, симуляція запуску ракети до обраної планети, відображення інформації про хід

місії у реальному часі.

Аналіз набору даних місії Kepler. Для роботи з набором даних використовується алгоритм, реалізований на Node.js. Код аналізує файл `kepler_data.csv`, фільтрує планети за наступними параметрами:

- `koi_disposition` – планета має бути підтвердженою (CONFIRMED);
- `koi_insol` – отримувана кількість енергії від зірки має бути в межах 0.36–1.11 відносно Землі (зона, де можливе існування рідкої води);
- `koi_prad` – радіус планети має бути меншим за 1.6 радіуса Землі (умова для твердотільних планет).

Функція `isHabitablePlanet()` відповідає за перевірку цих умов.

Лістинг 1.1 ImageProcessing

```
function isHabitablePlanet(planet) {
  return planet['koi_disposition'] === 'CONFIRMED'
    && planet['koi_insol'] > 0.36 && planet['koi_insol'] <
1.11
    && planet['koi_prad'] < 1.6;
}
```

Серверна частина. Сервер реалізований на базі Node.js із використанням таких бібліотек Express.js для створення API, MongoDB для зберігання та доступу до даних про планети.

Сервер надає два основні API endpoints `GET /planets` – повертає список усіх планет, придатних для життя; `POST /launch` – приймає дані для запуску місії, включаючи вибір планети та інші параметри.

Клієнтська частина. Frontend реалізований на React.js. Інтерфейс відображає список планет, їхні характеристики та симулює запуск ракети.

Інтерактивний трекер місії дозволяє користувачам відстежувати хід польоту.

Реалізація симуляції. Симуляція запуску ракети включає вибір планети з придатних для життя, визначення параметрів запуску (дата, швидкість, витрати пального) та моделювання польоту в реальному часі з використанням вебсокетів

для оновлення даних на клієнті.

2.7 Висновки до розділу 2

В даному розділі було розглянуто налаштування середовища розробника для створення вебдодатків за допомогою Node.js. Описано процес завантаження та встановлення Node.js та Visual Studio Code, що є основними інструментами для розробки серверної частини вебдодатків. Розглянуто ключові концепції, як-от різниця між frontend і backend, а також роль Node.js як серверної платформи для обробки запитів у реальному часі за допомогою асинхронних механізмів.

Практична цінність полягає в демонстрації того, як ефективно налаштувати середовище для розробки серверної частини вебдодатків, що використовують сучасні технології. Ці інструменти дозволяють створювати масштабовані додатки, що можуть бути використані в різних галузях, від освітніх платформ до інтернет-магазинів і систем для обробки великих даних.

Подальші дослідження можуть включати вивчення більш складних методів оптимізації серверів і баз даних, дослідження нових фреймворків і бібліотек для розширення функціональних можливостей вебдодатків, а також тестування їх ефективності в умовах реального використання.

3 ВЕБСЕРВЕР

3.1 Процес роботи вебсервера

Вебсервер – це програмне або апаратне забезпечення, що відповідає за обробку HTTP-запитів, отриманих від клієнтів (зазвичай веббраузерів), та надсилає HTTP-відповіді, які зазвичай містять вебсторінки, зображення, або інші ресурси. Основна функція вебсервера – обробляти запити клієнтів і забезпечувати доставку контенту з сервера до клієнта [19].

З чого починається робота вебсервера. Коли користувач вводить URL-адресу в браузері (наприклад, `www.facebook.com`), браузер надсилає HTTP-запит до вебсервера це клієнтський запит. URL-адреса перетворюється в IP-адресу через DNS-сервер. У прикладі з рис. 3.3 DNS повертає IP-адресу 31.13.80.36 – DNS-запит. Браузер встановлює з'єднання з вебсервером за вказаною IP-адресою та надсилає HTTP-запит на порт (зазвичай порт 80 для HTTP або 443 для HTTPS) Обмін через HTTP/HTTPS.

Сервер обробляє цей запит і знаходить відповідний ресурс (наприклад, HTML-сторінку, JSON-дані чи інші файли).

У свою чергу ми отримаємо відповідь сервера, вебсервер надсилає відповідь клієнту у вигляді HTTP-відповіді, яка може містити HTML інтерактивний контент для рендерингу вебсторінки, JSON/XML, формати для передачі структурованих даних, текстові файли або інший вміст.

Основні компоненти вебсервера – це програмне забезпечення (наприклад, Apache, Nginx або IIS), операційна система (Linux, Windows Server тощо), а також порти, через які здійснюється обмін даними (наприклад, порт 80 для HTTP і порт 443 для HTTPS).

Вебсервери, такі як Apache, Nginx або IIS, використовуються для обробки запитів та керування ресурсами [20].

Сервер працює на операційній системі, яка надає базову інфраструктуру для роботи вебсервера (Linux, Windows Server тощо).

Вебсервер отримує та надсилає ресурси, збережені на ньому, такі як

HTML-файли, CSS, JavaScript, медіафайли або відповіді API. HTTP зазвичай використовує порт 80, а HTTPS – порт 443 [21].

На прикладі з рис. 3.3 користувач вводить адресу `www.facebook.com`. DNS-сервер перетворює цю адресу на IP (`31.13.80.36`). Браузер встановлює з'єднання з вебсервером за цим IP через HTTP/HTTPS. Вебсервер Facebook обробляє запит і надсилає у відповідь HTML-сторінку, яка відображається у браузері.

Вебсервери є ключовою складовою сучасного інтернету. Вони використовуються для розгортання вебсайтів та API, надання динамічного контенту, підтримки серверного рендерингу (рис. 3.1).



Рисунок 3.1 – Процес роботи вебсервера

3.2 Крос-доменний обмін ресурсами

Cross-Origin Resource Sharing (CORS), або крос-доменний обмін ресурсами – це механізм, який дозволяє серверу вирішувати, які домени можуть отримувати доступ до його ресурсів. Цей механізм створений для обходу обмежень політики одного походження (Same-Origin Policy), що діє у веббраузерах. Політика одного походження за замовчуванням забороняє вебдодаткам взаємодіяти з ресурсами, що знаходяться на інших доменах, якщо

вони не відповідають тому ж походженню (домену, протоколу та порту).

CORS дозволяє розробникам створювати додатки, які можуть звертатися до API чи даних з різних доменів, забезпечуючи безпеку таких запитів.

Розгляньмо приклад. Уявімо, що ви переглядаєте статтю про машинне навчання на Wikipedia. Відкривши консоль розробника (Developer Console) у браузері й оновивши сторінку, ви побачите, що для завантаження однієї статті браузер виконує численні запити. Один із цих запитів може бути до іншого домену, наприклад, Wikimedia.org, для завантаження зображення.

Щоб такий запит пройшов успішно, сервер Wikimedia має надіслати в заголовок відповіді CORS-заголовок Access-Control-Allow-Origin, який визначає, які домени можуть отримувати доступ до ресурсу.

Якщо сервер дозволяє доступ усім доменам, він поверне «Access-Control-Allow-Origin: *»

Якщо сервер хоче дозволити доступ лише певному домену, наприклад <https://google.com>, заголовок виглядатиме так:

«Access-Control-Allow-Origin: <https://google.com>»

Якщо заголовок Access-Control-Allow-Origin не вказаний, браузер за замовчуванням блокує доступ до ресурсу з інших доменів через політику одного походження [22].

CORS – це потужний інструмент, який дозволяє обмінюватися даними між різними доменами, але при цьому забезпечує контроль і безпеку. Дотримання рекомендацій з налаштування та використання CORS дозволить вам створювати безпечні й гнучкі вебдодатки.

3.3 Архітектурний патерн MVC

Model View Controller (MVC) – це архітектурний патерн, який визначає, як розділити код додатка на три компоненти залежно від їхньої функціональності, а саме Model (Модель), View (Подання), Controller (Контролер).

Окрім цих компонентів, є користувач, який взаємодіє із системою.

Робота MVC полягає у тому що користувач ініціює взаємодію, зазвичай через запити до контролера. Контролер обробляє ці запити та змінює дані в моделі. Модель представляє стан даних додатка, наприклад, дані з бази даних чи інші джерела. Після оновлення моделі подання реагує на ці зміни та оновлює відображення даних для користувача.

Компоненти MVC в Express.js: Controller, Model, View. У контексті Express.js, контролер – це функції або методи, які обробляють вхідні запити та формують відповіді. Контролери визначають логіку додатка отримують дані від користувача, взаємодіють із моделлю та повертають результати у вигляді подання чи API-відповіді.

Модель містить дані додатка та методи доступу до них. У нашому прикладі це може бути масив в оперативній пам'яті, але на практиці модель зазвичай пов'язана з базою даних (наприклад, MongoDB чи SQL). Крім того, модель трансформує дані з того вигляду, в якому вони зберігаються, у той, який потрібен додатку.

Подання відповідає за те, як дані з моделі будуть відображені користувачеві. Це може бути HTML-сторінка, графіки, діаграми або навіть JSON-відповідь через API. Наприклад, якщо ви створюєте REST API, поданням буде структура даних, яку ви повертаєте у відповідь на запити.

Архітектура MVC розділяє код на логічні частини, що значно полегшує його читання, оновлення та масштабування. Кожен компонент відповідає лише за одну конкретну функцію, тому внесення змін до одного з компонентів не зачіпає інші [23].

Окрім того, використання стандартних патернів, таких як MVC, робить код зрозумілим для інших розробників, які долучаться до проекту. Це також спрощує процес налагодження та покращення додатку (рис. 3.2).

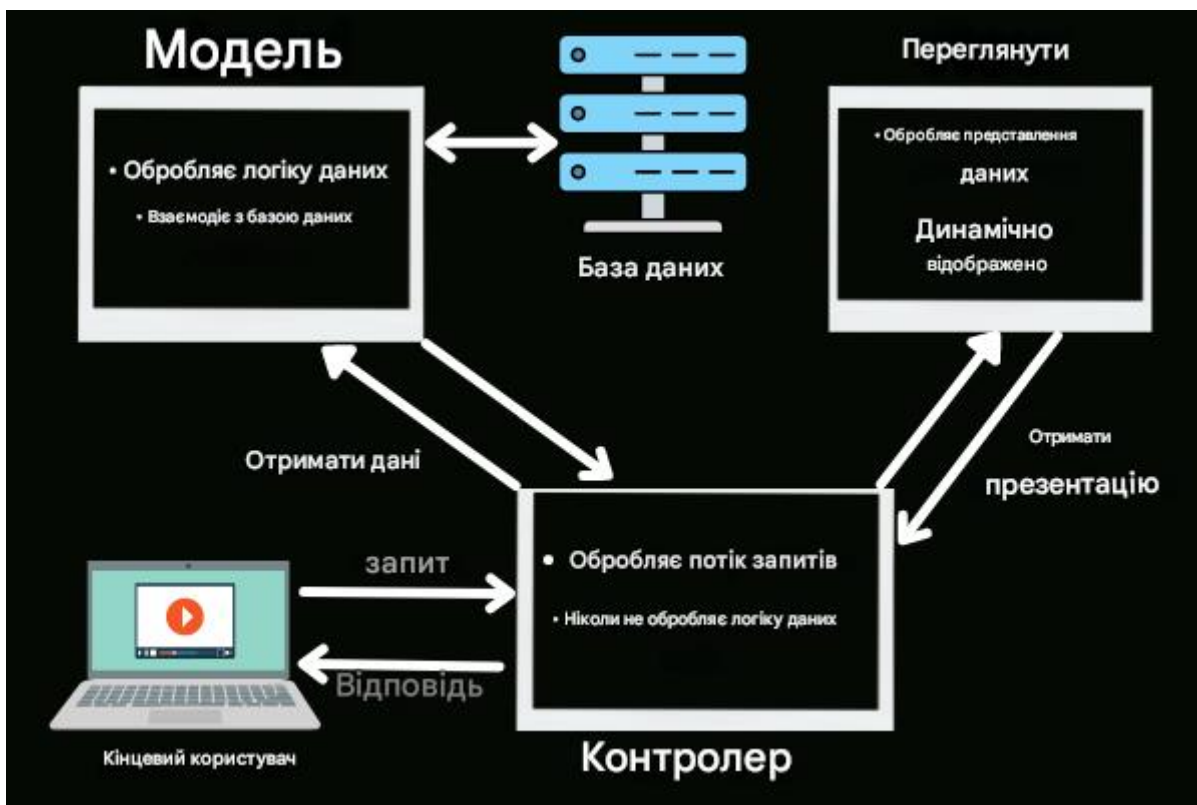


Рисунок 3.2 – Схема роботи архітектури MVC

3.4 Контролер представлення моделі в Експресс

Компоненти створеної структури відповідають архітектурі MVC [24]. Розроблені компоненти можна детально побачити у додатку В-Г.

Models файли відповідають за взаємодію з базою даних (MongoDB) і за бізнес-логіку.

У файлі `models/launches.model.js` містяться функції для завантаження, збереження, оновлення й отримання даних запусків. Використовує схему `launches.mongo.js` для підключення до колекції "launches" у MongoDB.

Файл `models/launches.mongo.js` містить схему Mongoose для моделі запусків.

У файлі `models/planets.model.js` містяться функції для завантаження даних із файлу CSV, перевірки населеності планет і збереження їх у базі даних. Використовує схему `planets.mongo.js`.

Файл `models/planets.mongo.js` містить схему Mongoose для моделі планет. Контролери (Controllers) містять логіку обробки HTTP-запитів,

взаємодіючи з моделями для доступу до даних.

Контролер `routes/launches/launches.controller.js` обробляє запити до `/launches`, включаючи отримання всіх запусків, додавання нового запуску та скасування запуску. Використовує функції з `launches.model.js`.

Маршрути (Routes) в `Express.js` зв'язують URL-адреси з відповідними методами контролера.

Маршрут `routes/launches/launches.router.js` пов'язує endpoints `/launches` з методами `httpGetAllLaunches`, `httpAddNewLaunch`, і `httpAbortLaunch`.

У `routes/planets/api.js` (фрагмент) наведено маршрутизація для планет і запусків.

Файли в каталозі `Services` – допоміжні функції, які використовуються в контролерах і моделях.

Файл `services/query.js` використовується для обчислення пагінації.

Роль кожного компонента в MVC.

Модель (Model) – забезпечує взаємодію з базою даних, використовуючи `MongoDB` (через `Mongoose`).

Контролер (Controller) – містить логіку обробки запитів (наприклад, валідацію даних і звернення до моделі).

Подання (View) – в цьому API-проекті відсутній компонент "View" у традиційному сенсі, оскільки відповіді повертаються у форматі JSON що є характерним для REST API [25].

3.5 Проміжне програмне забезпечення

Запити на перехресне походження (cross-origin requests) – це запити, які здійснюються з домену одного походження (origin) до іншого походження. Origin визначається як комбінація протоколу сайту, хоста та порту. Як приклад наведено `https://www.google.com:443` або `http://localhost:3000`.

У браузерах за замовчуванням заблоковані cross-origin запити як захисний механізм. Це означає, що запити, зроблені з одного сайту, не можуть

автоматично потрапляти на інший сервер, захищаючи дані користувачів.

Щоб дозволити певні запити, сервер має надіслати спеціальний заголовок Access-Control-Allow-Origin. Він може дозволити запити з будь-якого сайту (за допомогою «*»), або дозволити доступ лише з конкретного походження (вказавши точне значення). Практика whitelisting (список дозволених джерел) є найкращою практикою в плані безпеки, щоб контролювати доступ лише до певних ресурсів, яке можна побачити додатку Д [26].

3.6 Висновки до розділу 3

У третьому розділі було детально розглянуто роботу вебсервера, його основні функції та процеси, що забезпечують обробку HTTP-запитів. Вебсервер є невід'ємною частиною вебархітектури, забезпечуючи ефективну взаємодію між клієнтською частиною і сервером. Важливими аспектами роботи сервера є обробка запитів через порти, а також функціонування через протоколи HTTP/HTTPS, що дозволяє надавати різні ресурси та контент користувачам.

Практична цінність розділу полягає в розумінні принципів налаштування та оптимізації вебсервера, що дозволяє розробникам створювати більш стабільні та масштабовані вебдодатки. Ці знання важливі для розробки високопродуктивних і безпечних вебресурсів, здатних обробляти великі обсяги запитів.

Подальші дослідження можуть зосередитися на розширенні можливостей серверних архітектур, дослідженні нових методів кешування та оптимізації продуктивності вебсерверів при високих навантаженнях.

4 ПІДВИЩЕННЯ ПРОДУКТИВНОСТІ NODE

4.1 Створення простого блокуючого сервера

Перш ніж продемонструвати, як можна підвищити продуктивність, потрібно показати, як можна зіткнутися з проблемами продуктивності (рис. 4.1).

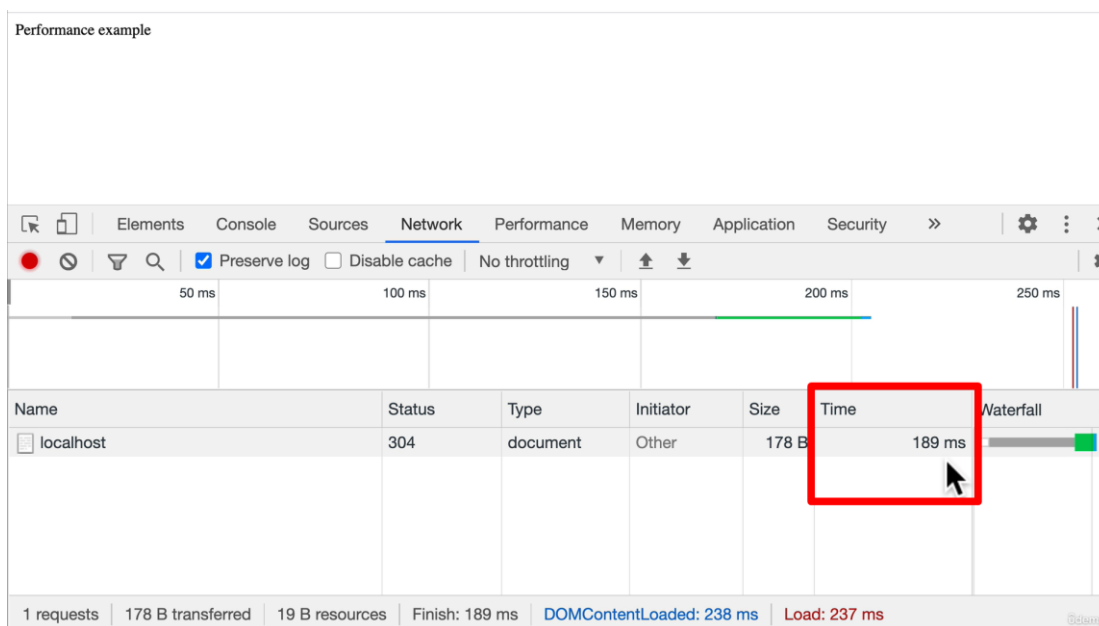


Рисунок 4.1 – Результат швидкодії відтворення вебсайту

Зазвичай цей запит займає менше 20 мілісекунд.

Однією з проблем у цьому розділі та при вимірюванні продуктивності в реальному світі є те, що точна поведінка залежить від умов вашого вимірювання.

Програми, які використовуються, швидкість вашого процесора, операційна система і навіть версія Node – все це може відігравати певну роль. Приклад коду продуктивності можна побачити у додатку 3.

Процесор і цикл обробки подій перебувають у стані високого навантаження.

За таких умов процес обробки працює на максимальній швидкості, яку може забезпечити процесор [27].

Цикл обробки подій зупиняється, доки не завершиться виконання функції затримки, що призводить до неможливості сервера обробляти нові запити. Якщо

у браузері в одній вкладці перейти до кінцевих точок таймера, а в іншій – до кореневих, це стане очевидним.

Незважаючи на те, що обробка сайту повиненна займати лише близько 20 мілісекунд, насправді це зайняло шість з половиною секунд, тому що навіть після перемикання на іншу вкладку і натиснення кнопки «Оновити», що зайняло близько двох з половиною секунд, кінцева точка таймера все ще виконувалася. Код блокування уповільнює роботу всього сервера.

Воно очікує коли перша вкладка оновиться і після цього йде оновлення другої вкладки (рис. 4.2 – 4.3).

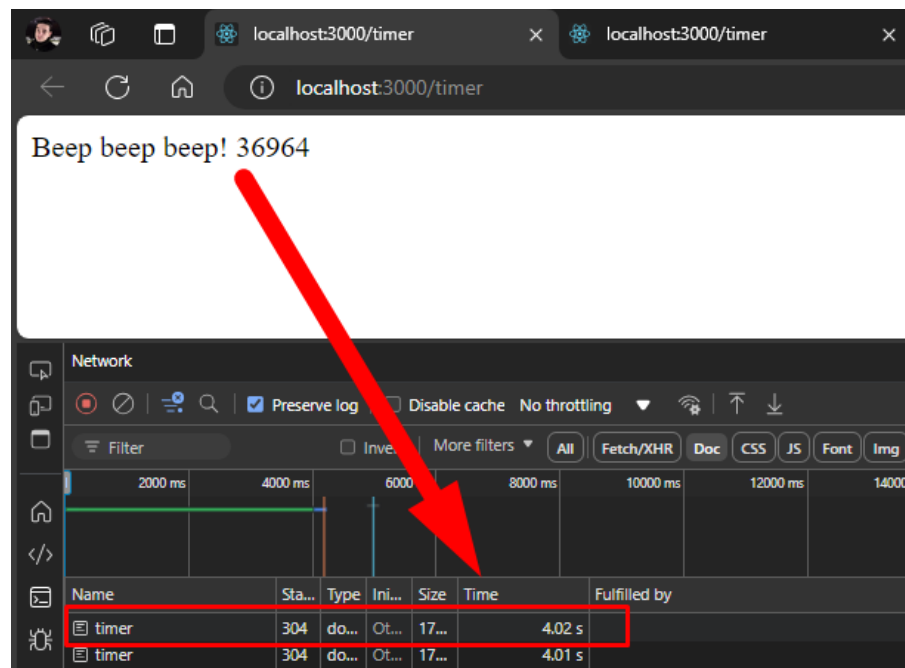


Рисунок 4.2 – Перший таймер відпрацював рівно 4 секунди

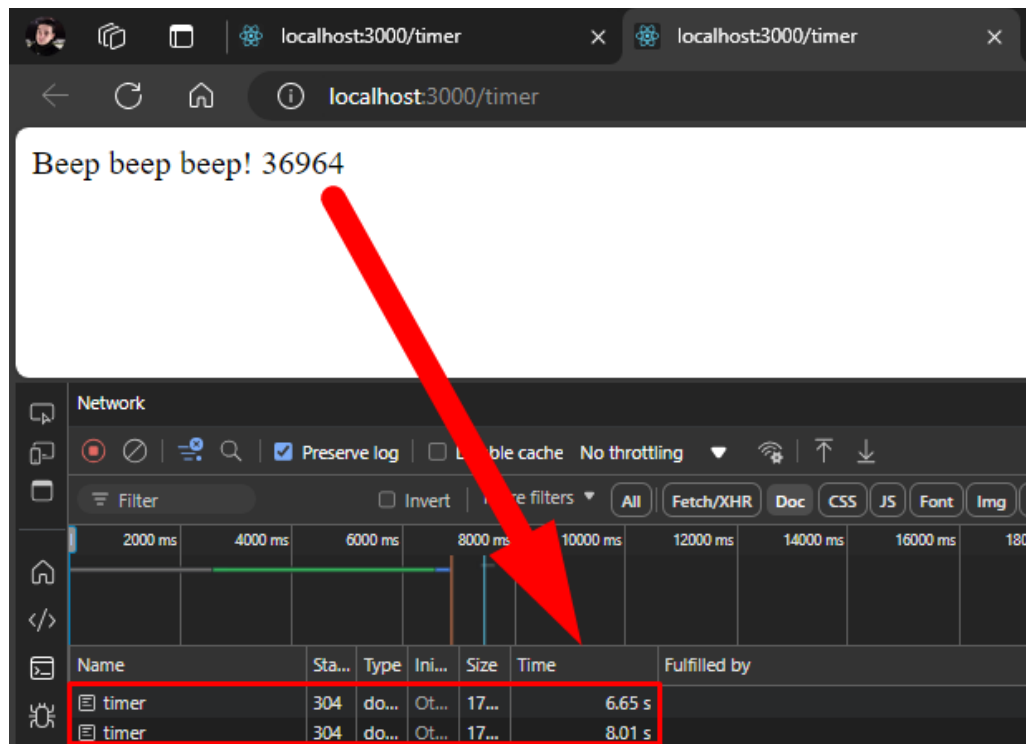


Рисунок 4.3 – Другий таймер відпрацював більше, ніж треба

4.2 Функції блокування в реальному житті

Розроблена функція затримки не дуже реалістична. Можна замінити її, наприклад, неблокуючим викликом для встановлення таймауту [28].

Які є приклади реальних функцій, що діють подібно до нашої функції затримки? Блокування циклу обробки подій (рис. 4.4).

```

5  function delay(duration) {
6    const startTime = Date.now();
7    while (Date.now() - startTime < duration) {
8      //event loop is blocked...
9    }
10 }

```

Рисунок 4.4 – Приклад простої затримки

Дві функції затримки, `JSON.stringify({})` функціонує там, де JSON, який

приймає stringify в об'єкті JavaScript, повертає його рядкове представлення.

А JSON.parsed() робить навпаки, де приймає об'єкта stringify і повертає відповідний об'єкт, тоді як stringify повертає його рядкове представлення (рис. 4.5).

```
12  app.get('/', (req, res) => {  
13    JSON.stringify({}) => "{}"  
14    JSON.parse("{}") => {}  
15    res.send('Performance example');  
16  });
```

Рисунок 4.5 – Функції затримки

Однак, якщо ваш сервер обробляє велику кількість запитів, це може призвести до збільшення часу їх обробки, особливо якщо кожен запит викликає функцію, таку як JSON.stringify. Це часто трапляється, наприклад, під час реєстрації об'єктів.

Якщо виклик цієї функції блокує сервер на 10 мілісекунд, а решта обробки запиту займає ще 10 мілісекунд, час відповіді на цей запит буде подвоєно. Більш важливим є те, що під час цих 10 мілісекунд сервер не зможе обробляти інші запити. Наступний запит не зможе бути оброблений до того, як завершиться обробка поточного запиту з використанням JSON.stringify. Тепер, деякі інші поширені функції, на які слід звернути увагу – це сортування масивів. Наприклад, використовуючи функцію сортування масиву, скажімо, є масив з п'яти чисел.

Якщо викликається sorts, блокується цикл обробки подій, поки ці числа сортуються для дуже великих масивів або для дуже великих об'єктів.

Ці функції можуть почати працювати дуже повільно. Також є певні функції у nodes, вбудовані в криптомодуль, який має функції, пов'язані з

криптографією та безпекою, які призначені для повільного виконання, щоб хакери довше не могли вгадати ваш пароль [29].

Наприклад, маємо функцію `crypto.pbkdf2(password, salt, iterations, keylen, digest, callback)`.

Так само, як і `crypto.scrypt(password, salt, keylen[, options], callback)`, це так звані функції виведення ключів, які використовуються для створення хешів паролів користувачів.

Щоб серверам не потрібно було зберігати паролі користувачів у вигляді звичайного тексту, а у вигляді хешу, який зазвичай виглядає як серія випадкових букв і цифр, можна, щоб використання цих хеш-функцій було дорогим і трудомістким, щоб хакери не могли перевіряти десятки тисяч або навіть мільйони паролів в секунду. Натомість їм доводиться виділяти значну частину своїх обчислювальних потужностей навіть для одного пароля.

Коли справа доходить до безпеки, маємо кумедну зворотну ситуацію, повільні функції блокування іноді є тим, що насправді потрібно, і це краще, ніж мати швидкі, легкі в обчисленні функції. Те, що ми робимо з функцією затримки, моделює найгірший випадок такої блокуючої поведінки, маючи функцію, яка повністю блокує цикл обробки подій на багато секунд.

За допомогою функції затримки моделюється найгірший випадок такої поведінки блокування, створюючи функцію, яка повністю блокує цикл обробки подій на багато секунд.

Зазвичай для реальних додатків ви хочете, щоб час відгуку не перевищував 100 або 200 мілісекунд.

Були проведені дослідження, які показують, як час відгуку сприймається людьми, коли мова йде про веб та додатки.

Ще в 1968 році було доведено, що близько 100 мілісекунд - це межа для користувачів, коли вони сприймають щось як миттєву реакцію [30].

Це те, що потрібно. І не довше однієї секунди. І цілком ймовірно, що потік думок користувача може перерватися – він може почати втрачати контекст того, що намагався зробити. Будь-які збільшення, що перевищують приблизно одну

секунду? Швидше за все, покаже прогресуюче зниження кількості користувачів, які залишаються на сайті (рис. 4.6).

Час реагування: 3 важливі обмеження



Якоб Нільсен
1 січня 1993

Ділитися

Зведення: Існує 3 основні часові обмеження (які визначаються перцептивними здібностями людини), які слід мати на увазі при оптимізації продуктивності вебу та додатків.

Уривок з глави 5 моєї книги «Інженерія юзабіліті» 1993 року:

Основні поради щодо часу реагування були приблизно однаковими протягом тридцяти років [Miller 1968; Card et al. 1991]:

- **0,1 секунди** – це приблизно межа для того, щоб користувач відчував, що система **реагує** миттєво, а це означає, що не потрібен спеціальний зворотний зв'язок, окрім як для відображення результату.
- **1,0 секунди** – це приблизно межа для того, щоб **потік думок користувача** залишався безперервним, навіть якщо користувач помітить затримку. Як правило, під час затримок більше 0,1 але менше 1,0 секунди спеціальний зворотний зв'язок не потрібен, але користувач втрачає відчуття роботи безпосередньо з даними.
- **10 секунд** – це приблизно межа для **утримання уваги користувача** на діалозі. У разі більш тривалих затримок користувачі захочуть виконувати інші завдання, чекаючи на завершення роботи комп'ютера, тому їм слід надати зворотний зв'язок із зазначенням, коли комп'ютер очікує на виконання. Зворотний зв'язок під час затримки особливо важливий, якщо час відповіді, ймовірно, буде сильно варіативним, оскільки тоді користувачі не знатимуть, чого очікувати.

Зазвичай час відповіді має бути якомога швидшим, але також комп'ютер може реагувати так швидко, що користувач не встигає за зворотним зв'язком. Наприклад, список прокрутки може рухатися настільки швидко, що користувач не зможе вчасно його зупинити, щоб потрібний елемент залишився

Рисунок 4.6 – Дослідження часу відгуку

73% користувачів втрачають інтерес до сайту.

Звичайно, це залежить від того, чим займається сайт. Якщо користувач очікує завантаження великого файлу або відео, він може бути трохи терплячішим, ніж якщо він очікує інформацію про акції та інвестиції в реальному часі. Але якщо користувач натиснув кнопку «Відправити» і протягом дев'яти секунд нічого не відбувається, він точно покине сайт.

Спінери та індикатори завантаження можуть допомогти. Але після певного моменту вони можуть почати дратувати.

Якщо у вас є інтернет-магазин, наприклад, ви можете використовувати

інформацію про те, скільки користувачів залишають ваш сайт на основі часу відгуку, щоб підрахувати, скільки доходу втрачає ваш магазин з кожною доданою мілісекундою.

Можна взяти середню суму грошей, яку користувачі витрачають на сайті в магазині, і помножити її на кількість користувачів, які втрачають інтерес, коли час завантаження сайту перевищує певну межу (рис. 4.7).

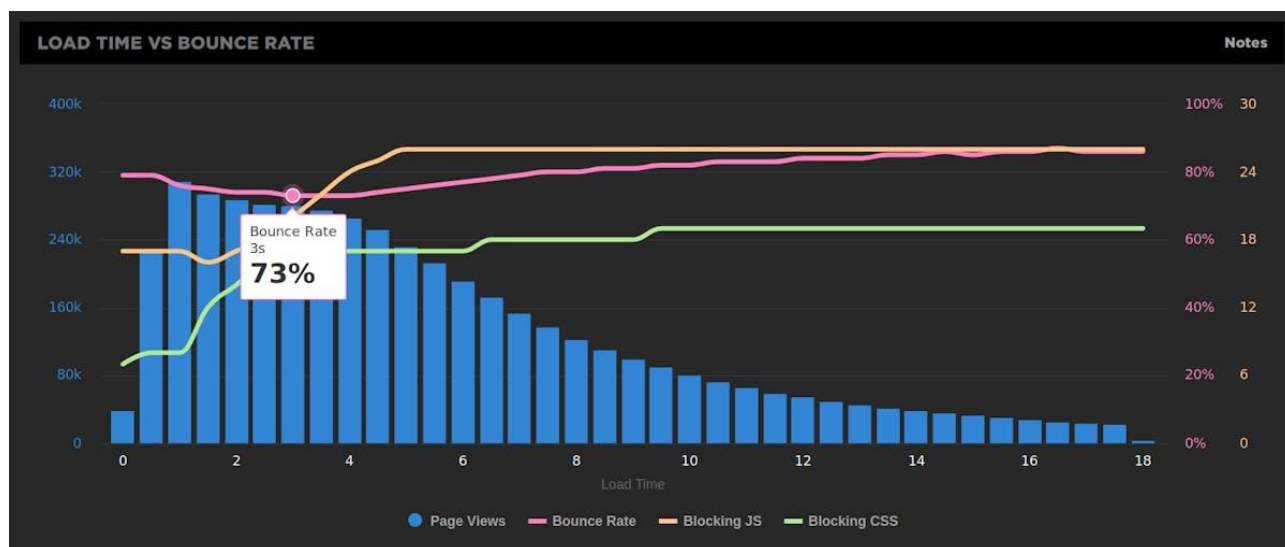


Рисунок 4.7 – Діаграми взаємодії. Кореляцію між ефективністю та залученістю користувачів [31]

Це лише кілька прикладів. Зрозуміло, що існують вагомі аргументи на користь високопродуктивних серверів і низького часу відгуку. Тепер ми знаємо, на які функції слід звертати увагу, і чому низький час відгуку важливий, навіть якщо наш цикл обробки подій заблоковано.

З'ясуємо, що ми можемо зробити, щоб покращити ситуацію.

4.3 Запуск декількох Node процесів

Що ми робимо в Node – це запускаємо кілька Node-процесів пліч-о-пліч, дозволяючи їм ділити роботу між собою, як команді, що працює разом над досягненням спільної мети.

З серверами робота, що виконується, розбивається на запити, що надходять на сервер.

Таким чином, замість того, щоб приймати кожен запит і обробляти його на одному сервері Node.js за допомогою одного нод-процесу, можна розподілити запити між декількома процесами Node.js.

Кожен з них відповідатиме на запит однаково.

Кожен з них містить копію серверного коду, який виконується паралельно пліч-о-пліч.

І тепер ці другі запити можуть бути оброблені другим процесом вузла, а третій запит може бути оброблений третім процесом вузла.

А якщо у нас більше запитів, ніж сервери, то кожен з цих процесів все ще може обробляти декілька запитів одночасно.

Важливо те, що тепер вони розподіляють навантаження порівну між собою [32].

Використання цієї техніки дозволяє однопоточним вузловим додаткам повною мірою використовувати всі процесори на комп'ютері.

Більшість комп'ютерів сьогодні мають декілька ядер процесора, які можуть виконувати код паралельно, не впливаючи на продуктивність коду, що виконується на будь-якому іншому ядрі процесора (рис. 4.8).



Рисунок 4.8 – Багатопоточність

4.4 Модуль кластера Node

Наш перший підхід до покращення продуктивності вузла – це використання вбудованого модуля кластера вузлів [33].

Кластерний модуль дозволяє створювати копії вашого вузлового процесу, кожна з яких паралельно виконує серверний код пліч-о-пліч.

Це працює наступним чином, коли введено тип вузла, сервер погоджується запустити вузол-додаток, створюється основний node-процес [34].

Node називає його головним процесом всередині кластерного модуля.

Ми маємо доступ до цієї функції, яка називається `fork`.

Кожного разу, коли запускається робоча функція в серверному файлі то вузол бере цей головний процес і створює його копію, яку називає робочим процесом. Можна викликати цю функцію `for`.

Однак, часто нам подобається створювати багато робочих процесів, приєднаних до одного головного процесу (рис. 4.9).

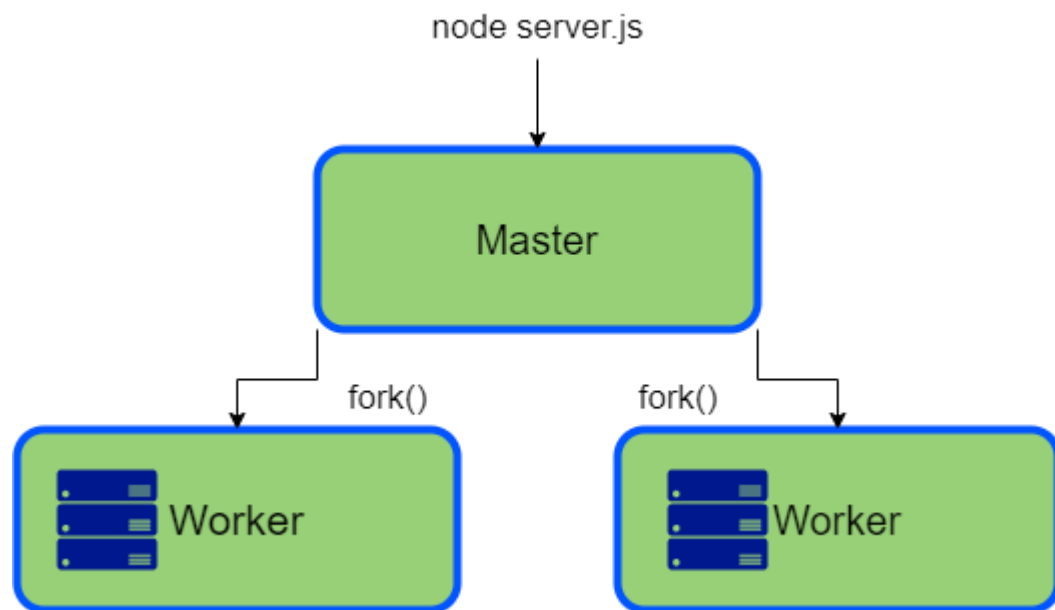


Рисунок 4.9 – Архітектура кластерного модуля Node.js, головний процес і робочі процеси

Саме ці робочі процеси виконують всю важку роботу з прийому HTTP

запитів, їх обробки та відповіді на них.

Кожен робочий процес містить весь код, необхідний для відповіді на будь-який запит на сервері, в той час як майстер відповідає лише за координацію створення цих робочих процесів за допомогою функції `fork`.

У цьому прикладі, де робоча функція була викликана двічі, у нас запущено три вузлових процеси.

Майстер, який ми створили самі, коли сказали Node виконати `server.js`, а потім два робочих процеси, кожен з яких був створений за допомогою функції `fork`, саме ці робочі процеси будуть приймати вхідні запити і обмінюватися ними, використовуючи вхідні запити і розподілятимуть їх, використовуючи так званий круговий підхід, коли перший запит надсилається першому робочому процесу.

Наступний запит – другому, оскільки у нас лише два процеси.

Третій запит повертається до першого працівника. Працівники по черзі відповідають на запити по мірі їх надходження.

І хоча не всі запити однакові, деякі з них можуть зайняти 15 секунд, тоді як інші – лише кілька мілісекунд.

В цілому, цей підхід найпростіший у реалізації, і він виявляється одним з найбільш справедливих способів розподілу роботи між двома працівниками, навіть у порівнянні з химерними підходами, які намагаються передбачити пріоритети і призначити різну важливість або вагу кожному запиту.

Обхід по колу – це хороший підхід.

Одне невелике застереження полягає в тому, що в Windows, через те, як Windows Managers керує процесами, вузол не дає жодних гарантій щодо використання циклічного підходу.

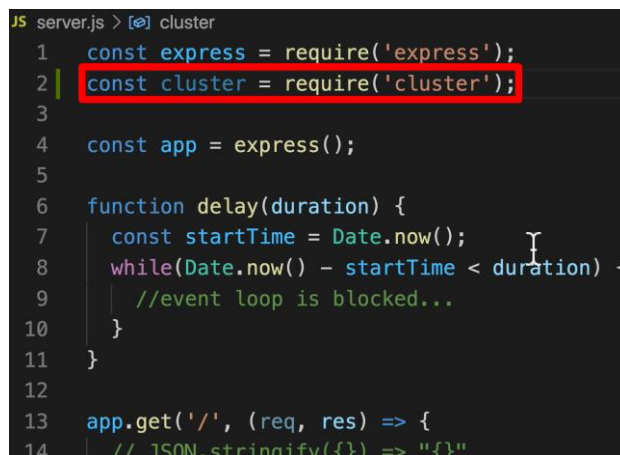
Замість того, щоб максимізувати продуктивність, він залишає можливість розподіляти завдання між робочими процесами [35].

До операційної системи Windows можна використовувати циклічний підхід.

4.5 Кластеризація в дії

Насамперед, для використання кластерного модуля нам потрібно задіяти вбудований кластерний модуль, присвоївши його константі, яка буде викликати кластер.

Тепер створюємо наш кластер процесів (рис.4.10).



```
server.js > [?] cluster
1  const express = require('express');
2  const cluster = require('cluster');
3
4  const app = express();
5
6  function delay(duration) {
7    const startTime = Date.now();
8    while(Date.now() - startTime < duration) {
9      //event loop is blocked...
10   }
11 }
12
13 app.get('/', (req, res) => {
14   // JSON.stringify({}) => '{}'
15 })
```

Рисунок 4.10 – Кластеризація

Коли ми пишемо node server.js, що ми називаємо master процесом, і саме з цього головного процесу ми можемо викликати нашу функцію розгалуження, щоб створити ці два робочі процеси. Робочі процеси, які ми розгалужуємо, виконують той самий код, що і в server.js.

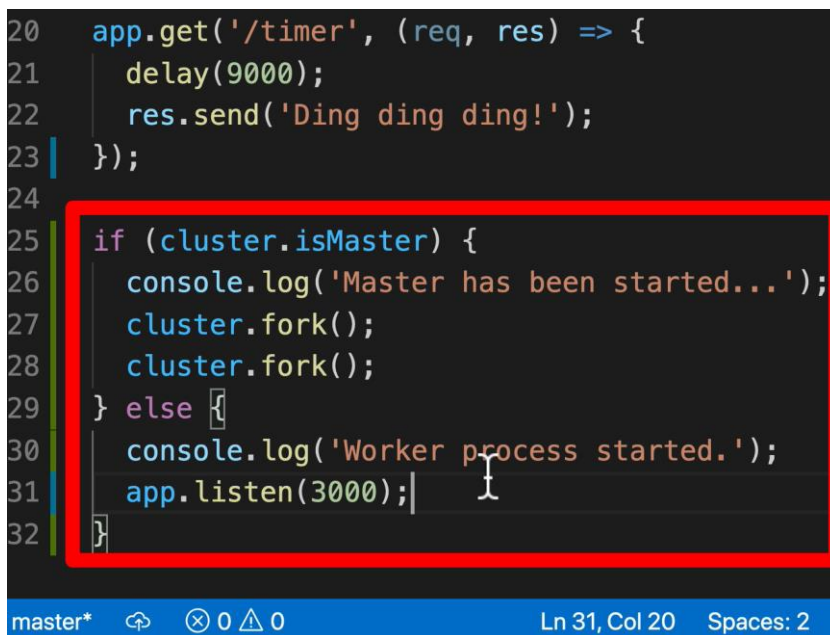
Ми можемо сказати, що якщо cluster.isMaster, то будь-який код всередині цього блоку буде виконано лише тільки тоді, коли наш файл server.js буде виконано як головний.

Запишемо це на консоль, щоб можна було побачити, що головний майстер було запущено. І все, що написано в умові else, буде працювати, коли його майстер буде помилковим, а код буде виконуватися як робочий процес.

Отже, побачимо в журналі консолі, що запущено робочий процес.

Зробити розгалуження на робочі процеси, які будуть запускати експрес-сервер. За допомогою двох маршрутів, які ми визначили тут. Отже, на рис.4.11 ми запускаємо вузловий сервер, який був головним [36]. А тепер ми хочемо

викликати `Cluster.Fork`. Кожен раз, коли ми викликаємо цю функцію з модуля кластера, ми створюємо `workers`. Ми можемо робити це скільки завгодно разів. Почнемо з двох (рис. 4.11).



```
20 app.get('/timer', (req, res) => {
21   delay(9000);
22   res.send('Ding ding ding!');
23 });
24
25 if (cluster.isMaster) {
26   console.log('Master has been started...');
27   cluster.fork();
28   cluster.fork();
29 } else {
30   console.log('Worker process started.');
```

The screenshot shows a code editor with a dark background. The code is written in a light blue font. A red rectangle highlights the conditional block starting from line 25. The status bar at the bottom indicates 'master*' and 'Ln 31, Col 20 Spaces: 2'.

Рисунок 4.11 – Ініціалізація кластеризації та запуск серверних процесів

Тільки як робочий процес ми хочемо слухати вхідні HTTP-запити і обробляти їх за допомогою наших експрес-маршрутів.

Отже, ми будемо викликати `app.listen`, коли ми працюємо як робочий процес, коли ми запускаємо HTTP-сервер і один з наших робочих процесів – незалежно від того, чи це вузли, вбудовані в `warebones` HTTP-сервера, чи це щось на кшталт експрес-вузлів – кластерний модуль розуміє функцію прослуховування.

Кожен з наших робочих процесів буде слухати на одному і тому ж порту 3000, а вузол HTTP-сервера знає, як розділити вхідні запити.

Сервер знає, що потрібно розділити вхідні запити, які надходять на порт 3000, між різними робочими процесами.

Кластеризація у Node.js дозволяє підвищити продуктивність серверів, розподіляючи навантаження між кількома процесами. Використання модуля `cluster` дозволяє створювати майстер-процес, який ініціалізує `workers` [37]. Ці

workers запускають сервер і обробляють запити паралельно, використовуючи той самий код.

Master process відрізняється від workers завдяки перевірці `cluster.isMaster`. У майстер-процесі викликається функція `fork`, яка створює необхідну кількість workers. Кожен worker слухає запити на одному й тому самому порту (наприклад, 3000), а Node.js автоматично розподіляє запити між ними. Для ідентифікації процесів можна використовувати унікальні ідентифікатори `process.pid`.

Кластеризація демонструє значні переваги. Наприклад, паралельне виконання довготривалих та швидких запитів без взаємного блокування. Проте, під час тестування необхідно відключати кеш браузера, щоб запити дійсно виконувались одночасно.

Результати показують, що різні запити обробляються окремими процесами, що суттєво підвищує швидкість відповіді сервера. Однак у реальних умовах важливо правильно налаштувати кластеризацію для досягнення оптимальної продуктивності.

Тому при тестуванні кластеризації серверів нам потрібно відключити кеш, щоб Chrome дійсно виконував запити.

Треба відключити кеш в обох вкладка.

Зробимо два запити в одній вкладці до нашої кінцевої точки таймера і дуже швидко, оновимо першу і другу вкладку.

У першій вкладці ми бачимо, що з'єднання очікує. Почекаємо дев'ять секунд, і ось ми маємо 200 відповідь, успішно. І в той самий час отримано відповідь на іншій вкладці. Обидва запити зайняли дев'ять секунд [38].

Вони виконувалися паралельно на комп'ютері, тому що він має два ядра процесора, і вони оброблялися в двох окремих процесах (рис. 4.12).

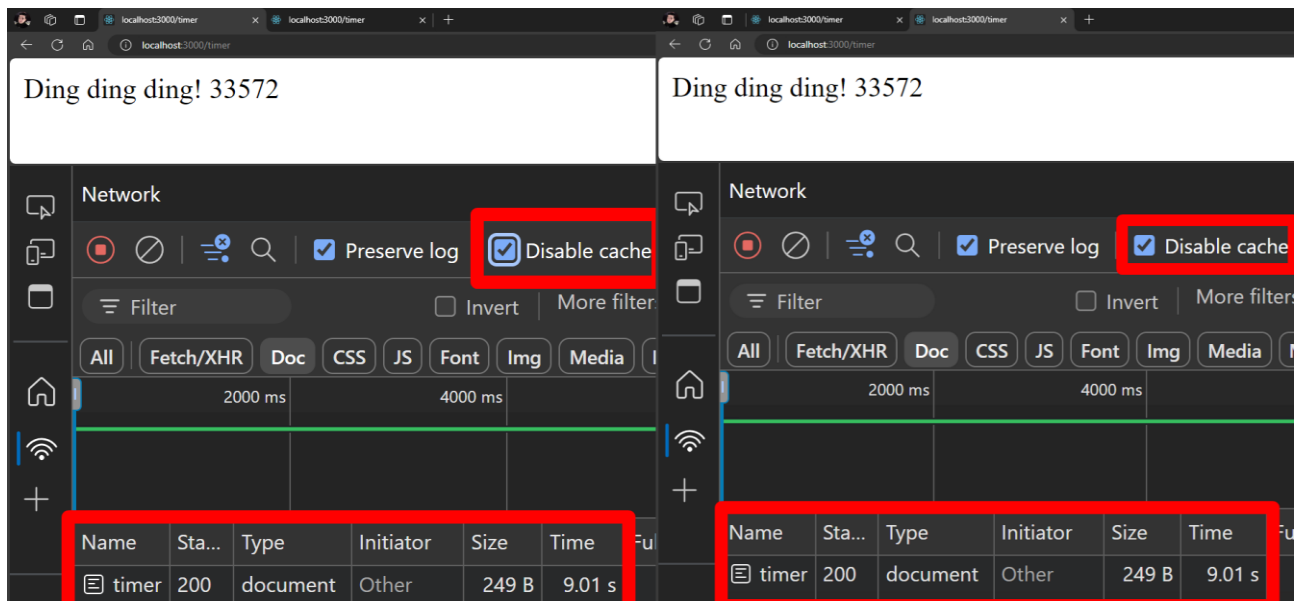


Рисунок 4.12– Паралельна обробка таймерів за допомогою двох фізичних ядер комп'ютера

Зробимо запит до кінцевої точки таймера в одній вкладці, а в іншій – до кореневого каталогу, і запит до кореневого каталогу повернувся майже миттєво, через 27 мілісекунд. Маршрут і точка тут не повинні чекати завершення таймера на першій вкладці. Це саме те, що необхідно. (рис. 4.13).

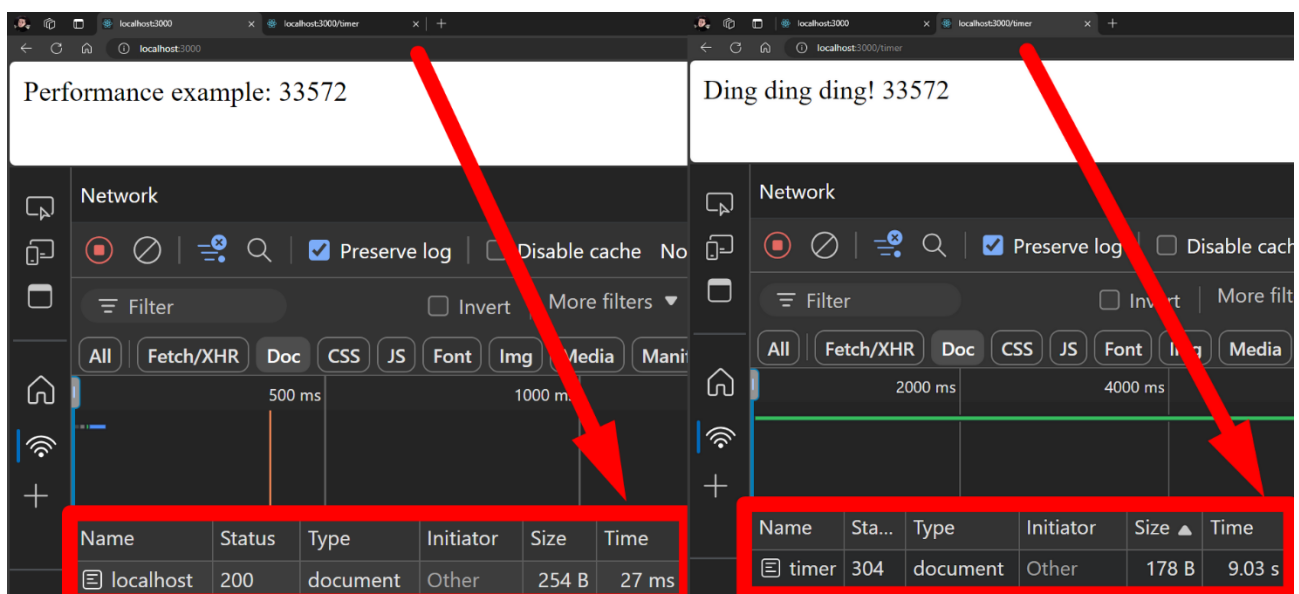


Рисунок 4.13– Порівняння root та timer швидкодії водночас

4.6 Максимізація продуктивності кластера

Але кластер має свої межі. Проведемо ще один маленький експеримент, продовжуючи приклад, якщо буде відкрито ще дві вкладки.

В кожній із вкладок відкрито мережеву консоль. Переконавшись, що цей кеш міток вимкнено, подивимось, що станеться, якщо буде перейдено до таймера та вказано на всіх вкладках одночасно. Запит повернувся через дев'ять секунд у першій вкладці, дев'ять секунд у другій.

На цій вкладці було перейдено до кореневої кінцевої точки замість таймера, і можна побачити, що навіть корені зайняли шість з половиною секунд.

Останній запит до таймера зайняв майже 16 секунд, майже вдвічі більше, ніж потрібно. Лише два запити були зроблені паралельно. Іншим двом довелося чекати, оскільки ці процеси вже були заблоковані.

Відповідаючи на перший запит, щоб таймер помітив ідентифікатори процесів, які було зареєстровано на першій вкладці, у нас є процес із цим ідентифікатором, що закінчується на двадцять дев'ять. Тут маємо три нуль для другого запиту та використовуємо наш циклічний підхід до розподілу роботи. Можна побачити, що третій запит повторно використав перший процес, а четвертий запит повторно використав другий процес (рис. 4.14).

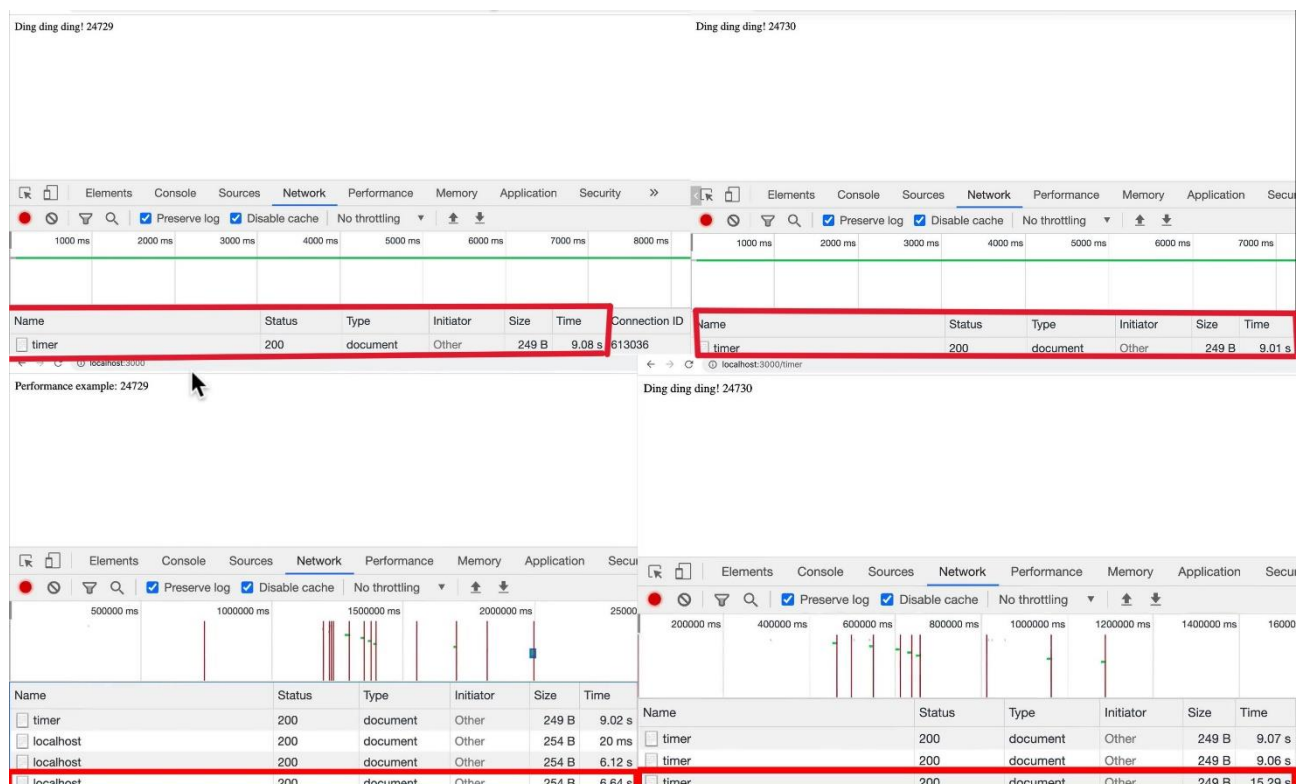


Рисунок 4.14 – Експеримент, тимчасовий недолік кластера

Використання кластеризації – це не те, що можна називати «панацеєю», те, що вирішує всі проблеми, просто застосовуючи той самий підхід.

Ми обмежені кількістю процесів, які можуть виконуватися паралельно. Наразі в нашому кластері ми спілкувалися з процесами, і ми могли обробляти лише два запити одночасно [39].

Це означає, що ми повинні продовжувати додавати ці хибні твердження, щоб створити більше робочих процесів.

Існує межа того, наскільки далеко може завести цей підхід. Розглянемо, що мається на увазі. Автоматизуємо створення форків. Буде використовуватися OS – модуль, який є ще одним вбудованим вузлом Node.

Ми збираємося використовувати операційну систему, щоб надати нам деяку інформацію, яка дозволить створити правильну кількість робочих процесів, які, працюють ефективно. Кожен процес має використовувати окремий процесор у комп'ютері.

Загалом ми хочемо обмежити силу, яку ми створюємо, кількістю логічних чи фізичних ядер у центральному процесорі (ЦП).

Фізичні ядра – це саме те. Це окремі процесори у комп'ютері, кожен з яких працює паралельно.

Логічні ядра дещо складніші. Вони використовують якусь химерну логіку, щоб дозволити вам запускати декілька потоків паралельно на одному фізичному ядрі, але лише в певних випадках і не так ефективно, як фізичне ядро може виконувати ваш код.

Щоб максимізувати продуктивність сервера в цілому, треба взяти ту кількість логічних ядер, яку ми можемо зробити, кажучи, що узгоджена кількість працівників дорівнює використанню нашого вбудованого модуля операційної системи OS, а не ЦП, що дає нам кількість логічних ядер. Для кожного з цих працівників (workers), що це дорівнює нулю.

Зараз ми лише створюємо обсяг роботи для наших процесів, який максимізує використання всіх ядер нашої серверної машини.

Це те, що необхідно в реальному світі. Перезапустимо сервер.

Маємо чотири фізичних ядра і вісім логічних. Тобто зараз є вісім логічних працівників.

Це означає, що в браузері можна робити до восьми запитів одночасно та отримувати підвищену продуктивність.

Чотири фізичні ядра повинні дозволити реакції повернутися до таймера приблизно через дев'ять секунд.

Тож чотири, що залишилися, можуть побачити певну ефективність десь посередині [40].

Загалом було зроблено вісім запитів. Переконаємося, що вимкнено кеш.

Знову приблизно дев'ять секунд, Це свідчить про ефективність логічних ядер на комп'ютері.

Було максимізовано продуктивність сервера залежно від кількості ядер ЦП у комп'ютері. Код можна побачити у додатку И (рис. 4.15).

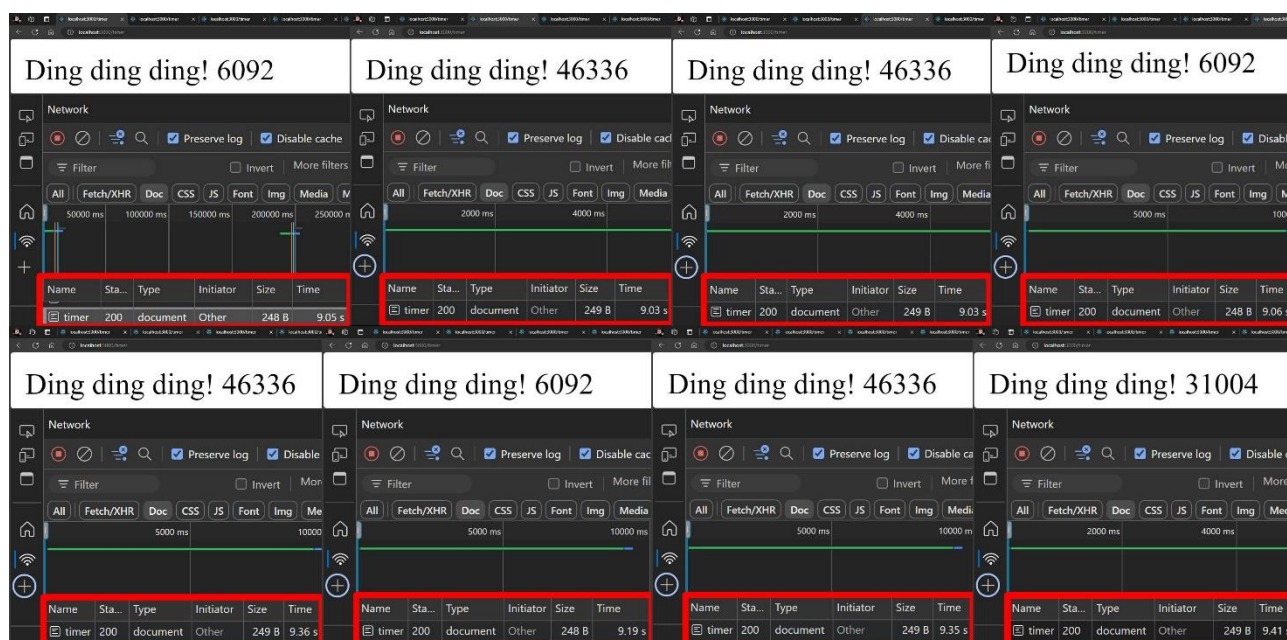


Рисунок 4.15– Паралельна обробка 8 вкладок водночас за 9 секунд

4.7 Балансування навантаження

Кругова система – це лише одна стратегія, яка використовується для так званого балансування навантаження. Але це надзвичайно важлива тема для створення backend.

Балансування навантаження – це спосіб розподілу набору завдань на набір ресурсів, наприклад, розділення запитів, які будуть оброблятися якими процесами.

Якщо є кластер робочих процесів для сервера, коли ми говоримо про балансування навантаження, використовується так званий балансувальник навантаження, щоб визначити, як, наприклад, запити мають розподілятися між різними процесами, які їх обробляють.

Балансувальник навантаження – це те, що приймає запити від користувачів і розподіляє їх таким чином, що відповідальність за обробку цих запитів розподіляється між багатьма різними процесами або потенційно різними програмами чи серверами [41].

Наприклад, ви можете мати два сервери, дві різні машини, на кожній з яких працює набір процесів, які можуть обробляти запити.

Отже, ви балансуєте навантаження запитів як через можливі різні сервери, так і через занадто багато різних процесів усередині цих серверів.

Балансування навантаження застосовується, коли паралельно запускаються кілька серверів або процесів, кожен з яких обробляє запит одного типу (рис. 4.16).

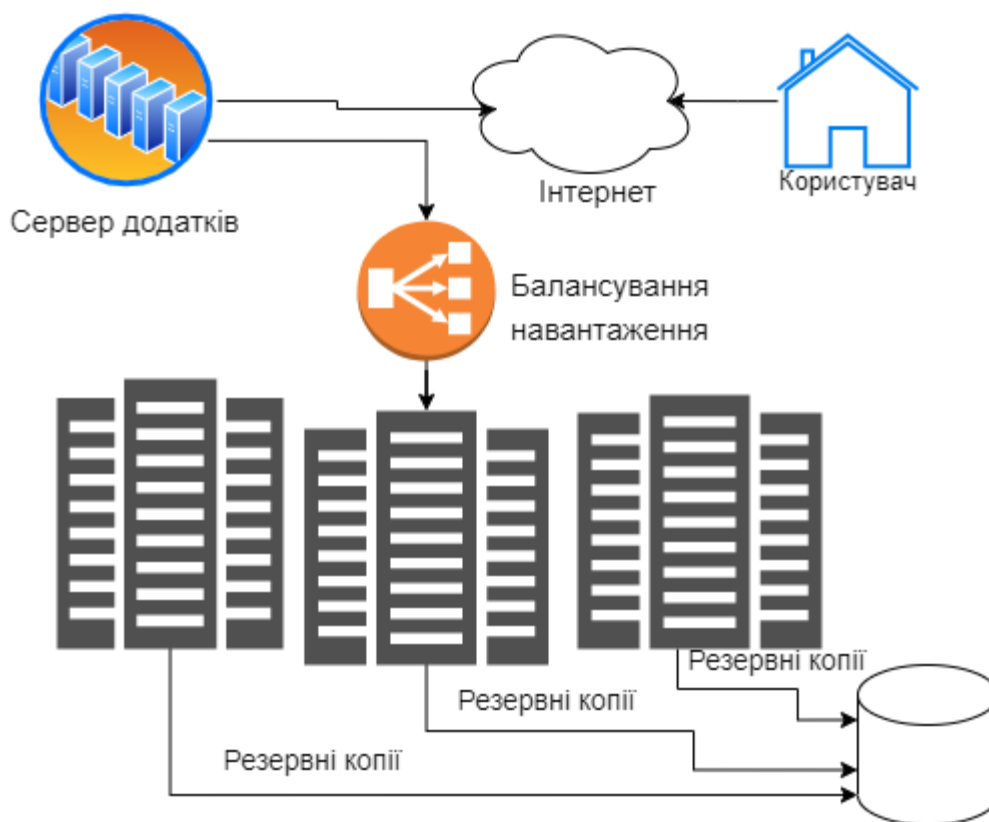


Рисунок 4.16 – Інфраструктура з балансуванням навантаження для MediaWiki

Як можна побачити у прикладі, використовуючи той самий набір можливих маршрутів, можна почути, що про балансування навантаження говорять у контексті так званого горизонтального масштабування [42].

Горизонтальне масштабування – це химерний термін для того, що ми робили в нашому кластері, а не те, що називається вертикальним масштабуванням, де додається більше швидкості процесу з одним вузлом.

Наш сервер з одним вузлом, наприклад, замінивши наш центральний процесор (ЦП) на швидший – це вертикальне масштабування.

Для горизонтального масштабування натомість береться наш сервер, і він

не обов'язково має бути надшвидким або великим, але ми розширюємо або масштабуємо наш додаток, щоб швидше обробляти більше запитів, додаючи більше серверів або, у випадку нашого кластера, більше процесів вузлів.

Це горизонтальне масштабування та балансування навантаження, у двох словах, дві найпоширеніші стратегії для розподілу запитів, які використовуються балансувальниками навантаження. У тому випадку, якщо ми не маємо попередньої інформації про те, скільки часу триватиме виконання запиту, що зазвичай буває, коли у нас є сервери з багатьма різними типами запитів, які іноді можуть займати більше часу, а іноді набагато менше.

Або коли у нас немає такої жорсткої передбачуваності, два основні підходи для балансування навантаження є круговими, з якими ми зараз маємо певний досвід. В основному просто випадковий розподіл запитів [43].

Кожен новий запит випадковим чином призначається одному з процесів.

Ці прості алгоритми для визначення того, який процес має обробити запит, насправді є найефективнішими, коли відсутня точна інформація про кількість часу, необхідного для виконання запиту.

Підводячи підсумок, у Node можна використовувати кластерний модуль для балансування навантаження запитів, які надходять на FTP-сервери нашого Node. Модуль кластера використовує циклічний підхід, щоб визначити, який процес оброблятиме ці вхідні запити.

4.8 Node Server продуктивність

Сервери Node.js зазвичай отримують запити, обробляють їх на циклі подій (event loop) та надсилають відповідь клієнту. Всі ці операції виконуються на одному потоці. Це означає, що Node.js може виконувати тільки одну операцію JavaScript-коду за раз, обробляючи лише один запит у будь-який момент часу.

Node.js дуже ефективно справляється з керуванням великою кількістю запитів завдяки передачі важких операцій (наприклад, введення/виведення) до пулу потоків (thread pool) операційної системи. Таким чином, JavaScript-код та

цикл подій залишаються "неблокуючими".

Наприклад, коли сервер обробляє запит, що вимагає читання або запису файлів, ця операція передається до пулу потоків. У той час JavaScript-код продовжує виконуватись, не блокуючи інші запити.

Node.js ідеально підходить для роботи з неблокуючими операціями, такими як обробка запитів до зовнішніх серверів чи інших систем. Однак виникають труднощі, коли потрібно виконувати складні та ресурсомісні обчислення, які блокують цикл подій. Такі операції не стосуються стандартних операцій введення/виведення, які Node.js обробляє ефективно.

Коли постає завдання виконувати важкі обчислення на JavaScript, який працює в одному потоці, це може значно знизити продуктивність сервера. Особливо це помітно на серверах, що працюють на машинах з багатоядерними процесорами.

Щоб краще зрозуміти ці виклики, створимо простий приклад для демонстрації таких проблем. Після цього розглянемо кілька підходів для покращення продуктивності серверів Node.js.

На рисунку 4.17 можна побачити обробку запитів клієнтів.

На зображенні клієнт виконує запит. Сервери (показані як стоси серверів із червоними індикаторами) отримують ці запити. Оскільки Node.js працює на основі циклу подій (event loop), він приймає запити й ефективно керує ними за допомогою своїх механізмів.

Зелене поле з різними іконками (наприклад, двигун V8, синій значок "обертання" та динозавр) представляє механізми, які Node.js використовує для роботи V8 Engine (двигун), основний двигун, який виконує JavaScript-код у Node.js.

Цикл подій (Event Loop), механізм, який приймає запити, передає "важкі" завдання до пулу потоків (thread pool) або операційної системи та повертає відповіді клієнтам.

Libuv (ілюстровано динозавром), бібліотека, яка дозволяє Node.js виконувати неблокуючі операції, передаючи задачі, пов'язані з

введенням/виведенням, до пулу потоків.

Node.js передає складні або довготривалі задачі, наприклад, читання файлів чи запити до бази даних, до пулу потоків або операційної системи. Це дозволяє основному JavaScript-коду не блокувати обробку інших запитів (рис. 4.17).



Рисунок 4.17 – Обробка запитів клієнта

4.9 PM2 інструмент

PM2 – це чудовий інструмент для керування Node.js-серверами, особливо в кластерному режимі. Він створений на основі модуля cluster і надає додаткові функціональні можливості, які полегшують роботу з серверами у продакшн-середовищі.

PM2 працює на базі модуля cluster. Це означає, що всі концепції та функціонал модуля cluster застосовуються і тут. Однак PM2 додає багато корисних функцій для керування кластерами.

PM2 дозволяє легко управляти процесами у кластері [44].

Автоматичний рестарт процесів, якщо код змінюється – PM2 автоматично перезапустить кластерні процеси. Це аналогічно тому, як працює nodemon, але для багатопоточних процесів.

Якщо один із процесів завершиться через помилку, PM2 може автоматично перезапустити його.

"Graceful Shutdowns" (коректне завершення роботи) – PM2 забезпечує

плавне завершення роботи процесів [45].

PM2 пропонує багатий набір функцій для моніторингу.

Збір логів із кожного процесу та керування ними.

PM2 популярний не лише в проєктах на Node.js. Завдяки своїй потужності його застосовують у багатьох інших середовищах.

PM2 доступний у реєстрі npm, і його можна легко встановити та використовувати як інструмент для керування Node.js-серверами [46].

Використання PM2 для створення кластерів можна побачити у додатку Е.

4.10 Висновки до розділу 4

У цій главі було розглянуто ключові аспекти підвищення продуктивності Node.js через використання кластеризації, балансування навантаження та додаткових інструментів, таких як PM2.

Проблеми продуктивності.

Блокуючі функції та їх вплив на час відгуку сервера були проаналізовані на прикладі простого серверного додатку. Було показано, що блокування циклу обробки подій може значно знизити ефективність обробки запитів.

Кластеризація.

Використання модуля cluster дозволяє запускати кілька процесів Node.js, які працюють паралельно, розподіляючи навантаження між фізичними та логічними ядрами процесора. Це дає можливість ефективніше використовувати ресурси системи, зокрема багатоядерні процесори, для одночасної обробки запитів.

Балансування навантаження.

Було описано стратегічні підходи до розподілу запитів, такі як кругова система та горизонтальне масштабування. Ці методи допомагають зменшити час відгуку сервера та забезпечити стабільну роботу навіть за високого навантаження.

Інструмент PM2.

PM2 був відзначений як зручний інструмент для управління кластерними процесами в продакшн-середовищі. Його функції автоматичного перезапуску, моніторингу та зручного керування роблять його важливим елементом у роботі з Node.js.

Таким чином, кластеризація та балансування навантаження є ефективними рішеннями для покращення продуктивності серверів Node.js. Однак вони мають свої межі, які необхідно враховувати під час розробки високонавантажених додатків. Використання додаткових інструментів, таких як PM2, дозволяє автоматизувати процеси та спростити управління серверними ресурсами.

5 БАЗИ ДАНИХ

Бази даних є ключовим компонентом сучасних систем зберігання та обробки даних. Вони поділяються на дві основні категорії, SQL (Sequel) та NoSQL бази даних. Ці підходи мають суттєві відмінності у структурі, моделі даних та методах використання, і вибір між ними залежить від специфіки задачі. У цьому розділі ми порівняємо SQL та NoSQL бази даних, зокрема, розглянемо PostgreSQL (SQL) та MongoDB (NoSQL) як приклади цих типів.

5.1 Порівняння SQL та NoSQL баз даних

SQL-бази даних слідуєть реляційній моделі даних, у якій інформація організована у вигляді таблиць з рядками та стовпцями. Кожна таблиця має чітко визначену структуру, а зв'язки між таблицями забезпечуються за допомогою ключів (первинних і зовнішніх). Це дозволяє ефективно моделювати складні зв'язки між даними [47].

На противагу цьому, NoSQL-бази, такі як MongoDB, використовують документну модель. Дані зберігаються у вигляді документів, які нагадують JSON-формат. Кожен документ містить усі пов'язані дані для конкретного об'єкта, наприклад, інформацію про користувача чи продукт. Це робить NoSQL бази більш гнучкими у роботі з неструктурованими чи слабо структурованими даними [48].

Організація даних. У PostgreSQL дані розділяються на таблиці, які взаємодіють через відношення. Наприклад, таблиця "Друзі" може містити посилання на таблицю "Міста", щоб визначити, де живе кожен друг. Це сприяє нормалізації даних – процесу усунення дублювання.

У MongoDB жодна нормалізація не потрібна, усі дані про друга та його місто можуть бути збережені в одному документі. Це знижує кількість запитів до бази та покращує швидкість читання, але потенційно збільшує об'єм даних через дублювання.

Запити до бази даних. PostgreSQL використовує мову запитів SQL, яка є стандартом для всіх реляційних баз. Вона дозволяє виконувати складні операції, як-от об'єднання таблиць (JOIN), із чітко визначеними правилами [48].

У MongoDB запити будуються на власному синтаксисі, спеціально розробленому для роботи з документами. Цей підхід забезпечує простоту роботи із вкладеними структурами даних, але може бути менш зрозумілим для користувачів, які звикли до SQL.

Масштабування. PostgreSQL зазвичай масштабується вертикально, тобто шляхом збільшення потужності одного сервера. Водночас сучасні версії PostgreSQL впроваджують можливості горизонтального масштабування, яке є стандартним для MongoDB.

MongoDB розроблена для горизонтального масштабування "з коробки". Це означає, що дані можуть бути розподілені між кількома серверами, що особливо зручно для великих обсягів даних та високих навантажень.

PostgreSQL підтримує широкий набір типів даних, включаючи числові, текстові, часові та навіть географічні. MongoDB більш гнучка в цьому плані, оскільки документи можуть містити довільні структури даних, але це може ускладнити перевірку їхньої коректності.

На рис. 5.1 представлено ключові відмінності між SQL та NoSQL базами даних. SQL бази орієнтовані на структуровані дані з чіткими взаємозв'язками, тоді як NoSQL пропонують більшу гнучкість у роботі з неструктурованими даними.

	MONGODB	POSTGRES
Тип	ДОКУМЕНТ	РЕЛЯЦІЙНИЙ
ОРГАНІЗОВАНО	КОЛЕКЦІЇ	ТАБЛИЦІ
МОВА ЗАПИТУ	NOSQL	SQL
МАСШТАБУВАННЯ	ГОРИЗОНТАЛЬНЕ	ВЕРТИКАЛЬНЕ

Рисунок 5.1 – Порівняння SQL та NoSQL

Рис. 5.2 ілюструє різницю у підходах до організації даних у PostgreSQL та MongoDB. У PostgreSQL дані розподілені між таблицями, тоді як у MongoDB все зберігається в одному документі. Цей контраст демонструє, як різні моделі баз даних підходять для різних задач.

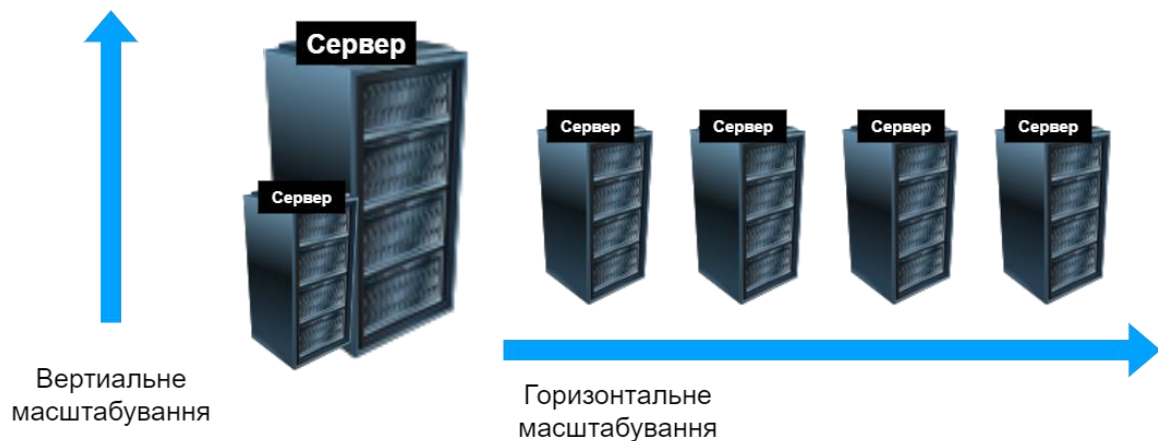


Рисунок 5.2 – Підходи організації даних SQL та NoSQL

5.2 Налаштування MongoDB Atlas

MongoDB Atlas – це хмарна платформа, що дозволяє створювати та керувати базами даних MongoDB без потреби встановлювати їх локально на комп'ютері. Цей розділ пояснює процес налаштування MongoDB Atlas і створення кластеру, який стане основою для роботи з базою даних у хмарному середовищі.

MongoDB Atlas є безкоштовним для використання у базовій конфігурації, що робить його чудовим вибором для навчальних і невеликих проєктів. Основною перевагою Atlas є легкість у створенні масштабованих кластерів, які включають автоматичне резервне копіювання, шифрування та інструменти для моніторингу [49].

Треба зареєструватись та перейти до вкладки Atlas, обрати проєкти (Projects) і натиснути новий проєкт (New Project) (рис. 5.3) [50-52].

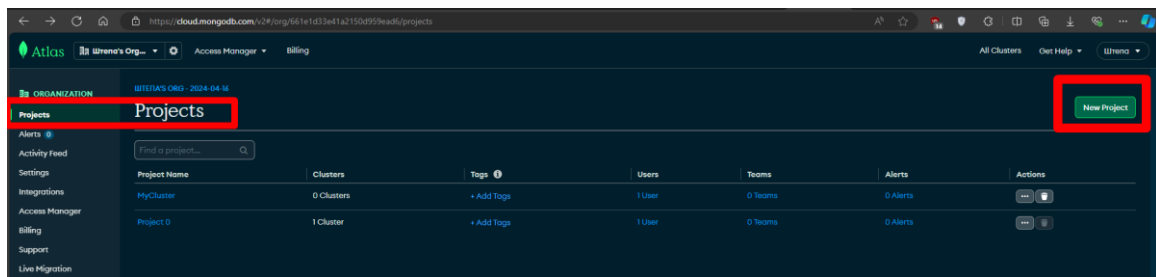


Рисунок 5.3 – Створення проекту в Mongo Atlas

Даємо назву проекту (рис. 5.4).

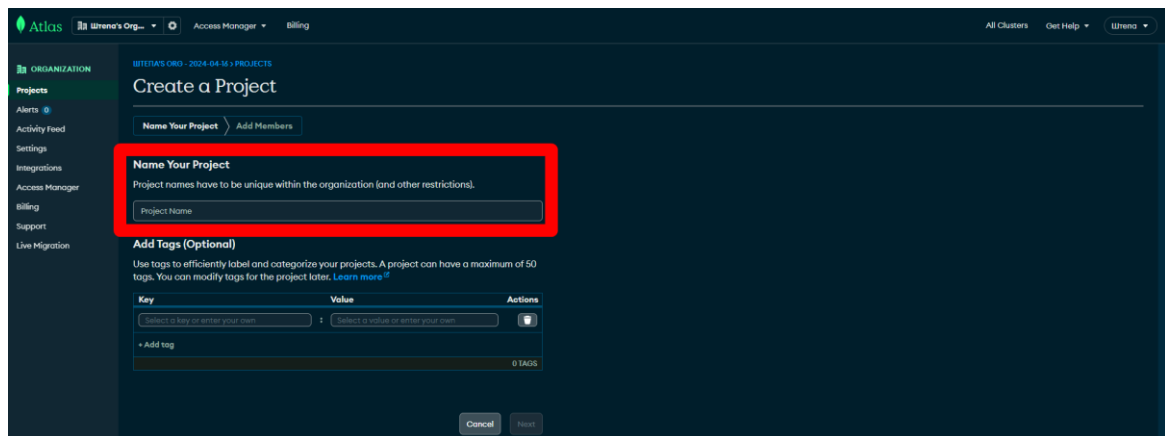


Рисунок 5.4 – Назва проекту

В новоствореному проєкті натискаємо зелену кнопку Create, зверху буде написаний текст «Create a cluster» (рис. 5.5).

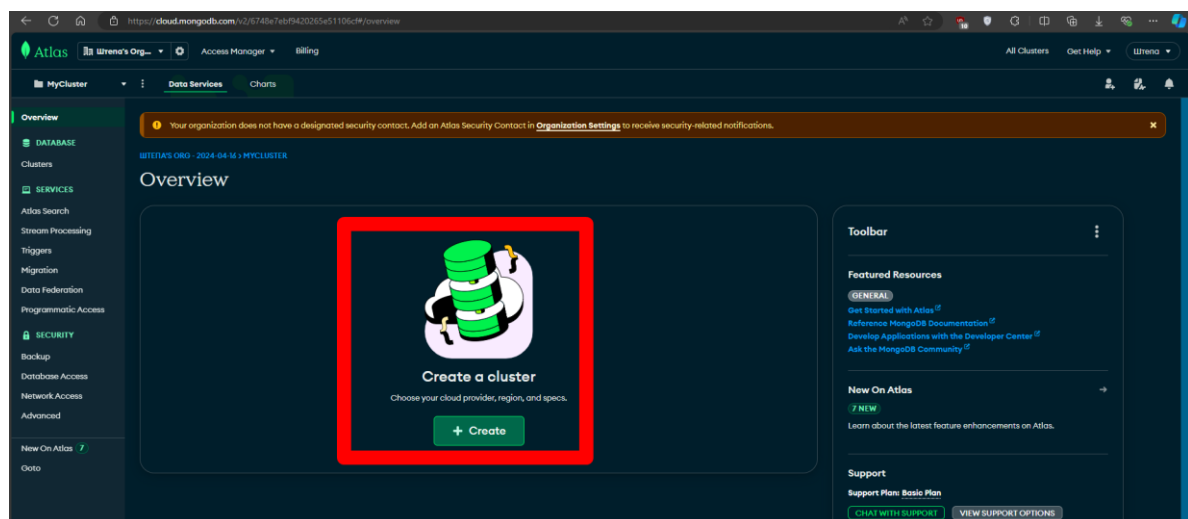


Рисунок 5.5 – Створення кластеру

Вказуємо безкоштовний тариф. Пишемо назву кластера. Обираємо провайдера. Обираємо регіон сервера бази даних і натискаємо зелену кнопку «Create Deployment» (рис. 5.6).

Рисунок 5.6 – Вибір опцій кластера

Далі переходимо у вкладку Network Access, натискаємо зелену кнопку «Add IP Address», натискаємо сіру кнопку «Allow Access from Anywhere» і підтверджуємо IP адресу натиснувши зелену кнопку Confirm (рис. 5.7).

Рисунок 5.7 – Додавання IP адресу

Переходимо до вкладки «Overview», натискаємо зелену кнопку «Connect», реєструємо користувача БД, треба запам'ятати ім'я користувача та пароль. Тепер потрібно скопіювати посилання MongoDB замінивши у посиланні <db_password> на пароль користувача

Детальне налаштування можна побачити у додатку Ж (рис. 5.8).

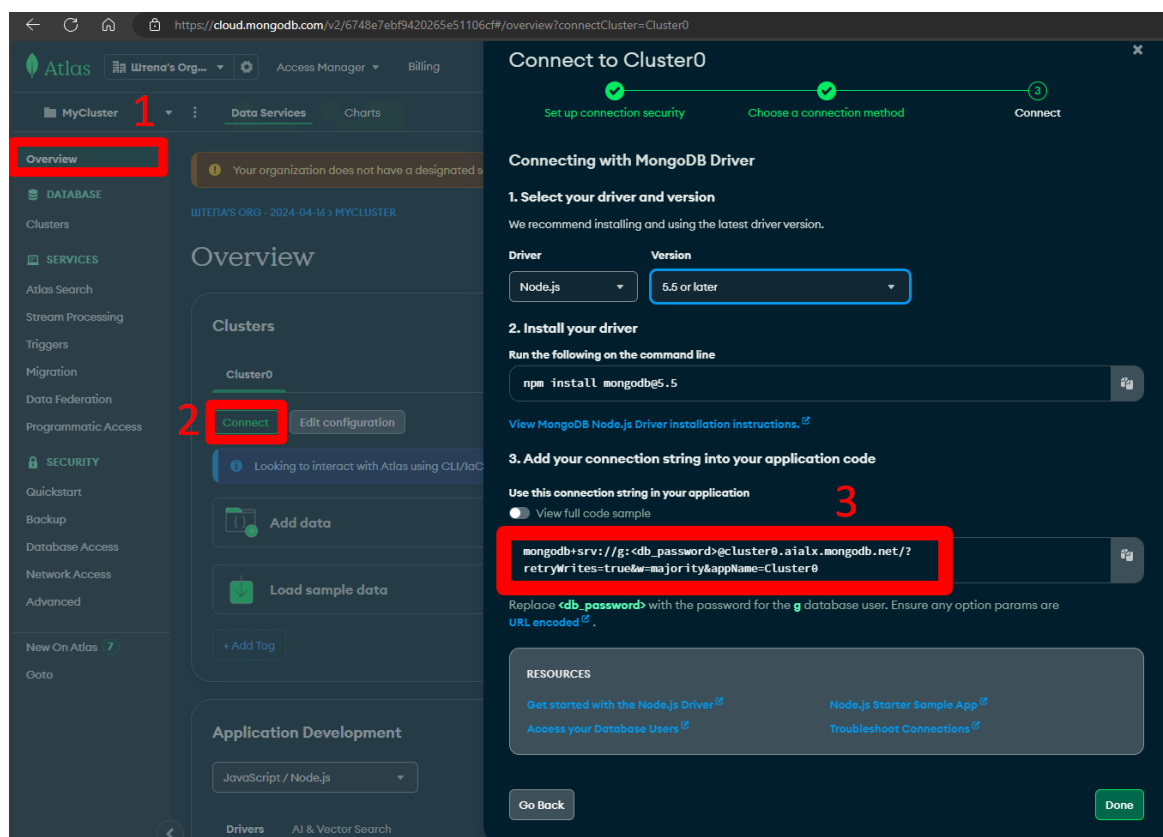


Рисунок 5.8 – Під'єднання до бази даних MongoDB

5.3 Висновки до розділу 5

У п'ятому розділі були розглянуті ключові аспекти використання баз даних, зокрема порівняння SQL та NoSQL систем, а також практичні аспекти налаштування та використання MongoDB. Розглянувши PostgreSQL і MongoDB, було з'ясовано основні відмінності між реляційними і документними базами даних, що впливає на вибір технології залежно від специфіки задачі.

SQL бази даних, такі як PostgreSQL, ефективно працюють з структурованими даними, забезпечуючи чітке розмежування інформації через

таблиці та зв'язки між ними, що сприяє нормалізації і зниженню дублювання. З іншого боку, NoSQL бази даних, зокрема MongoDB, пропонують більшу гнучкість для роботи з неструктурованими та слабо структурованими даними, що дозволяє зберігати всю необхідну інформацію в одному документі, зменшуючи необхідність у складних запитах та оптимізуючи швидкість читання.

Також розглянуто процес налаштування MongoDB Atlas, що дозволяє легко створювати масштабовані бази даних у хмарному середовищі, з можливістю автоматичного резервного копіювання та шифрування. MongoDB Atlas є зручним інструментом для розробників, які працюють з великими даними і потребують високої доступності та масштабованості.

Подальші дослідження можуть бути спрямовані на оптимізацію використання цих баз даних у великих масштабах, розробку нових підходів до управління даними та покращення процесів масштабування та резервного копіювання.

ВИСНОВКИ

У цій роботі було розглянуто створення кластеру бази даних MongoDB Atlas, а також налаштування підключення до нього через додаток на Node.js. Використання MongoDB Atlas є важливим кроком у розвитку сучасних додатків, які потребують масштабованих, надійних і безпечних баз даних. У ході роботи було детально описано процес створення кластеру, конфігурацію доступу до нього, а також налаштування користувачів із застосуванням принципу найменших привілеїв.

Розглянутий підхід до хмарних баз даних має ряд переваг.

Простота налаштування. MongoDB Atlas надає зручний інтерфейс, що дозволяє створити кластер всього за декілька хвилин.

Безпека. Завдяки вбудованим механізмам шифрування даних і можливості обмеження доступу за IP-адресами, забезпечується високий рівень захисту інформації.

Масштабованість. Використання кластерів із кількома вузлами дозволяє ефективно обробляти запити та забезпечує стійкість до відмов.

Доступність. Безкоштовний тарифний план є ідеальним вибором для навчальних цілей, тестування або невеликих проєктів.

У практичній частині було наочно продемонстровано,

Процес створення кластеру MongoDB, вибір хмарного провайдера та регіону.

Налаштування користувача з обмеженими правами доступу для дотримання принципу найменших привілеїв.

Додавання IP-адрес для доступу до бази даних.

Отримання URI для підключення до бази даних через додаток на Node.js.

Дослідження підтвердило, що MongoDB Atlas є зручним і ефективним інструментом для організації хмарних баз даних. Завдяки цьому сервісу розробники можуть зосередитися на розробці функціоналу додатку, делегуючи адміністрування бази даних надійному хмарному провайдеру.

Отримані знання та навички можуть бути використані для створення сучасних веб та мобільних додатків, які вимагають роботи з великими обсягами даних, високої швидкості обробки запитів та гнучкості у масштабуванні. Це дозволяє стверджувати, що хмарні бази даних, такі як MongoDB Atlas, є ключовим компонентом у розвитку ІТ-індустрії та відкривають нові можливості для ефективної розробки програмного забезпечення.

Практична цінність роботи полягає в тому, що створений вебдодаток із застосуванням сучасного технологічного стеку (Node.js, Express.js, MongoDB) демонструє ефективність в умовах високих навантажень. Запропоновані рішення можуть бути використані в різних галузях, таких як освітні платформи, електронна комерція, CRM-системи, та інших системах, які потребують швидкої обробки даних і масштабованості.

Подальші дослідження будуть спрямовані на:

- розширення можливостей кластеризації серверів та оптимізацію використання багатоядерних процесорів;
- дослідження нових алгоритмів балансування навантаження для підвищення ефективності обробки запитів;
- вивчення сучасних технологій у сфері серверного рендерингу (SSR) та інтеграції з хмарними обчисленнями;
- тестування розроблених рішень у реальних умовах з підвищеними вимогами до безпеки та продуктивності.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. NASA Exoplanet Archive [Electronic resource] – Access mode: <https://exoplanetarchive.ipac.caltech.edu/cgi-bin/TblView/nph-tblView?app=ExoTbIs&config=cumulative>.
2. Top Node.js Backend Frameworks in 2025 [Electronic resource] – Access mode: <https://flatirons.com/blog/top-nodejs-backend-frameworks-2024/>.
3. Deno [Electronic resource] – Access mode: <https://deno.com/>.
4. Django vs Flask [Electronic resource] – Access mode: <https://www.greengeeks.com/blog/django-vs-flask-python-framework/>.
5. MongoDB [Electronic resource] – Access mode: <https://www.mongodb.com/>.
6. PostgreSQL [Electronic resource] – Access mode: <https://www.postgresql.org/>.
7. MySQL [Electronic resource] – Access mode: <https://www.mysql.com/>.
8. Download Node.js® [Electronic resource] – Access mode: <https://nodejs.org/en/download/prebuilt-installer>.
9. Visual Studio Code [Electronic resource] – Access mode: <https://code.visualstudio.com/download>.
10. Backend та Frontend [Electronic resource] – Access mode: <https://www.geeksforgeeks.org/frontend-vs-backend/>.
11. Навіщо розуміти backend-частину та чому варто обрати Node.js [Electronic resource] – Access mode: <https://robotdreams.cc/uk/blog/241-zachem-ponimat-backend-chast-i-pochemu-stoit-vybrat-node-js>
12. Стратегії асинхронної обробки для середовища Node.js [Electronic resource] – Access mode: <https://peerdh.com/uk/blogs/programming-insights/asynchronous-processing-strategies-for-node-js-middleware>
13. How to build a blazing fast GraphQL API with Node.js, MongoDB and Fastify [Electronic resource] – Access mode: <https://www.freecodecamp.org/news/how-to-build-a-blazing-fast-graphql-api-with->

node-js-mongodb-and-fastify-77fd5acd2998/.

14. Node.js Introduction [Electronic resource] – Access mode: https://www.w3schools.com/nodejs/nodejs_intro.asp.

15. Thiru, Event Loops in Node.js: A Deep Dive into LibUV, Thread Pool, and Event Instances [Electronic resource] / Thiru – Access mode: https://medium.com/@thiru_73177/event-loops-in-node-js-a-deep-dive-into-libuv-thread-pool-and-event-instances-4054d9c6fde8.

16. NodeJS – Threads and The Event Loop [Electronic resource] – Access mode: <https://www.zartis.com/nodejs-threads-and-the-event-loop/>.

17. NPM nodemon [Electronic resource] – Access mode: <https://www.npmjs.com/package/nodemon>.

18. Kepler Exoplanet Archive [Electronic resource] – Access mode: https://exoplanetarchive.ipac.caltech.edu/cgi-bin/TblView/nph-tblView?app=ExoTbls&config=PS&constraint=default_flag=1&constraint=disc_facility+like+%27%25TESS%25%27.

19. HTTP request methods [Electronic resource] – Access mode: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods>.

20. HTTP response status codes [Electronic resource] – Access mode: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>.

21. What is an API (Application Programming Interface)? [Electronic resource] – Access mode: <https://www.mulesoft.com/api/what-is-an-api>.

22. Cross-Origin Resource Sharing (CORS) [Electronic resource] – Access mode: <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>.

23. MVC Framework Introduction [Electronic resource] – Access mode: <https://www.geeksforgeeks.org/mvc-framework-introduction/>.

24. MVC в Node.js з MongoDB [Electronic resource] – Access mode: <https://github.com/ghrabla/Express-JS-MongoDB-MVC-Rest-API>

25. Побудова RESTful API з Node.js і MongoDB [Electronic resource] – Access mode: <https://restful-api-node-typescript.books.dalenguyen.me/en/latest/>

26. CORS [Electronic resource] – Access mode:

<https://www.npmjs.com/package/cors>.

27. Performance example [Electronic resource] – Access mode: <https://gitlab.com/Ivan-hot/performance-example>.

28. Асинхронні аналоги для уникнення блокування [Electronic resource] – Access mode: <https://github.com/tim-kos/ajson>

29. Crypto [Electronic resource] – <https://nodejs.org/api/crypto.html>.

30. Nielsen, J. Response Times: The 3 Important Limits [Electronic resource] / J. Nielsen – Access mode: <https://www.nngroup.com/articles/response-times-3-important-limits/>.

31. Everts, T. Engagement charts: See correlations between performance and user engagement [Electronic resource] / T. Everts – Access mode: <https://www.speedcurve.com/blog/web-performance-monitoring-user-engagement/>.

32. Node.js Cluster Module [Electronic resource] – Access mode: <https://nodejs.org/api/cluster.html#cluster>

33. How to Create a Node.js Cluster for Speeding Up Your Apps [Electronic resource] – Access mode: <https://www.sitepoint.com/how-to-create-a-node-js-cluster-for-speeding-up-your-apps/>

34. Cluster [Electronic resource] – Access mode: <https://nodejs.org/api/cluster.html>.

35. Scaling Node.js Applications Using Clustering [Electronic resource] – Access mode: <https://shape.host/resources/scaling-node-js-applications-using-clustering-a-comprehensive-guide>

36. Why Node.js clustering is key for optimized applications [Electronic resource] – Access mode: <https://kinsta.com/blog/node-js-clustering/>

37. Clustering [Electronic resource] – Access mode: <https://betterstack.com/community/guides/scaling-nodejs/node-clustering/>

38. Node.js Cluster Process Module [Electronic resource] – Access mode: https://www.w3schools.com/nodejs/ref_cluster.asp

39. How To Scale Node.js Applications with Clustering [Electronic resource] – Access mode: <https://www.digitalocean.com/community/tutorials/how-to-scale->

node-js-applications-with-clustering

40. Node.js cluster Module: Node.js Clustering for Horizontal Scaling [Electronic resource] – Access mode: <https://devcrud.com/node-js-cluster-module-node-js-clustering-for-horizontal-scaling/>

41. Load balancing (computing) [Electronic resource] – Access mode: [https://en.wikipedia.org/wiki/Load_balancing_\(computing\)](https://en.wikipedia.org/wiki/Load_balancing_(computing)).

42. Scaling Nodejs Applications with Load Balancers and Proxies [Electronic resource] – Access mode: <https://moldstud.com/articles/p-scaling-nodejs-applications-with-load-balancers-and-proxies>

43. Maximize Node.js Performance with Load Balancing and Clustering Techniques [Electronic resource] – Access mode: <https://codezup.com/node-js-load-balancing-clustering/>

44. PM2 Tool [Electronic resource] – Access mode: <https://pm2.keymetrics.io/docs/usage/quick-start/>.

45. PM2 NPM [Electronic resource] – Access mode: <https://www.npmjs.com/package/pm2>.

46. Application development resources [Electronic resource] – Access mode: <https://cloud.google.com/architecture/application-development>.

47. Порівняльний аналіз баз даних SQL та NOSQL / [О. В. Романюк, А. В. Денисюк, А. В. Марущак та ін.] // Інформаційно-комп'ютерні технології – 2021 (ІКТ-2021): XII Міжнародна науково-технічна конференція, Житомир, 1-3 квітня 2021: тези доповідей. – Житомир: Житомирська політехніка, 2021. – С. 3-4.

48. Огороднік, М. О. Переваги та недоліки реляційних та нереляційних баз даних / М. О. Огороднік, О. В. Зелінська // Прикладні аспекти сучасних міждисциплінарних досліджень: І Всеукраїнська науково-практична конференція, Вінниця, 26 листопада 2021 р.: тези доповідей. – Вінниця: ДонНУ імені Василя Стуса, 2021. – С. 106–109.

49. What is NoSQL? [Electronic resource] – Access mode: <https://www.mongodb.com/resources/basics/databases/nosql-explained>.

50. Setting up MongoDB Atlas [Electronic resource] – Access mode:
<https://www.mongodb.com/docs/atlas/getting-started/?msockid=31b14dca944a63603dd359a0955862fa>.

51. What is Least Privilege? [Electronic resource] – Access mode:
<https://www.cyberark.com/what-is/least-privilege/>.

52. Mongoose [Electronic resource] – Access mode:
<https://mongoosejs.com/>.

ДОДАТОК А
Структура папки сервер

C:\Users\VanoLaptop\OneDrive\diploma_\Section-Diploma-Launch-Rockets-on-Inhabit-Planets-Project\server> tree -I 'node_modules'

tree -I 'node_modules' ігнорує велику структуру папки node_modules

C:.

```

|——server
file —— .env
file —— package-lock.json
file —— package.json
|——data
file——kepler_data.csv
|——public
file —— asset-manifest.json
file —— favicon.png
file —— index.html
file —— manifest.json
file —— robots.txt
| |——img
    file ——background-large.jpg
    file ——background-medium.jpg
    file ——background.jpg
    file ——glow.png
| |——sound
    file ——abort.mp3
    file ——click.mp3
    file ——deploy.mp3
    file ——success.mp3
    file ——typing.mp3
    file ——warning.mp3
| |——static
| |——js

```

```
file ——— main.dc55bdba.js
file ——— main.dc55bdba.js.LICENSE.txt
file ——— main.dc55bdba.js.map

└──src
    ├──models
    |   file ——— launches.model.js
    |   file ——— launches.mongo.js
    |   file ——— planets.model.js
    |   file ——— planets.mongo.js
    ├──routes
    |   file ——— api.js
    |   ├──launches
    |   |   file ——— launches.controller.js
    |   |   file ——— launches.router.js
    |   |   file ——— launches.test.js
    |   ├──planets
    |   |   file ——— planets.controller.js
    |   |   file ——— planets.router.js
    └──services
        file ——— app.js
        file ——— server.js
```

ДОДАТОК Б
Структура папки клієнт

C:\Users\VanoLaptop\OneDrive\diploma_\Section-Diploma-Launch-Rockets-on-Inhabit-Planets-Project\client> tree -I 'node_modules'

tree -I 'node_modules' ігнорує велику структуру папки node_modules

C:.

```

|——server
file —— .gitignore
file —— package-lock.json
file —— package.json
file —— README.md
|——build
|  |——img
      file —— background-large.jpg
      file —— background-medium.jpg
      file —— background.jpg
      file —— glow.png
|  |——sound
      file —— abort.mp3
      file —— click.mp3
      file —— deploy.mp3
      file —— success.mp3
      file —— typing.mp3
      file —— warning.mp3
|  |——static
|  |——js
      file —— main.bccfce63.js
      file —— main.bccfce63.js.LICENSE.txt
      file —— main.bccfce63.js.map
|——public
|  |——img
      file —— background-large.jpg

```

- file — background-medium.jpg
- file — background.jpg
- file — glow.png
- | | — sound
 - file — abort.mp3
 - file — click.mp3
 - file — deploy.mp3
 - file — success.mp3
 - file — typing.mp3
 - file — warning.mp3
- | — src
 - | — components
 - file — Centered.js
 - file — Clickable.js
 - file — Footer.js
 - file — Header.js
 - | — hooks
 - file — requests.js
 - file — useLaunches.js
 - file — usePlanets.js
 - | — pages
 - file — usePlanets.js
 - file — usePlanets.js
 - file — usePlanets.js

ДОДАТОК В**Код проєкту структурований у стилі MVC (внутрішня частина)**

Папка models

Файл launches.model.js

```

const axios = require('axios');

const launchesDatabase = require('./launches.mongo');
const planets = require('./planets.mongo');

const DEFAULT_FLIGHT_NUMBER = 100;

const SPACEX_API_URL =
'https://api.spacexdata.com/v4/launches/query';

async function populateLaunches() {
  console.log('Downloading launch data...');
  const response = await axios.post(SPACEX_API_URL, {
    query: {},
    options: {
      pagination: false,
      populate: [
        {
          path: 'rocket',
          select: {
            name: 1
          }
        },
        {
          path: 'payloads',
          select: {
            'customers': 1
          }
        }
      ]
    }
  });
}

```

```

    if (response.status !== 200) {
      console.log('Problem downloading launch data');
      throw new Error('Launch data download failed');
    }

    const launchDocs = response.data.docs;
    for (const launchDoc of launchDocs) {
      const payloads = launchDoc['payloads'];
      const customers = payloads.flatMap((payload) => {
        return payload['customers'];
      });

      const launch = {
        flightNumber: launchDoc['flight_number'],
        mission: launchDoc['name'],
        rocket: launchDoc['rocket']['name'],
        launchDate: launchDoc['date_local'],
        upcoming: launchDoc['upcoming'],
        success: launchDoc['success'],
        customers,
      };

      console.log(`${launch.flightNumber} ${launch.mission}`);

      await saveLaunch(launch);
    }
  }

  async function loadLaunchData() {
    const firstLaunch = await findLaunch({
      flightNumber: 1,
      rocket: 'Falcon 1',
      mission: 'FalconSat',
    });
    if (firstLaunch) {

```

```

        console.log('Launch data already loaded!');
    } else {
        await populateLaunches();
    }
}

async function findLaunch(filter) {
    return await launchesDatabase.findOne(filter);
}

async function existsLaunchWithId(launchId) {
    return await findLaunch({
        flightNumber: launchId,
    });
}

async function getLatestFlightNumber() {
    const latestLaunch = await launchesDatabase
        .findOne()
        .sort('-flightNumber');

    if (!latestLaunch) {
        return DEFAULT_FLIGHT_NUMBER;
    }

    return latestLaunch.flightNumber;
}

async function getAllLaunches(skip, limit) {
    return await launchesDatabase
        .find({}, { '_id': 0, '___v': 0 })
        .sort({ flightNumber: 1 })
        .skip(skip)
        .limit(limit);
}

```

```

async function saveLaunch(launch) {
  await launchesDatabase.findOneAndUpdate({
    flightNumber: launch.flightNumber,
  }, launch, {
    upsert: true,
  });
}

async function scheduleNewLaunch(launch) {
  const planet = await planets.findOne({
    keplerName: launch.target,
  });

  if (!planet) {
    throw new Error('No matching planet found');
  }

  const newFlightNumber = await getLatestFlightNumber() + 1;

  const newLaunch = Object.assign(launch, {
    success: true,
    upcoming: true,
    customers: ['Zero to Mastery', RocketLaunch],
    flightNumber: newFlightNumber,
  });

  await saveLaunch(newLaunch);
}

async function abortLaunchById(launchId) {
  const aborted = await launchesDatabase.updateOne({
    flightNumber: launchId,
  }, {
    upcoming: false,
    success: false,
  });
}

```

```

    return aborted.modifiedCount === 1;
}

```

```

module.exports = {
  loadLaunchData,
  existsLaunchWithId,
  getAllLaunches,
  scheduleNewLaunch,
  abortLaunchById,
};

```

Файл launches.mongo.js

```

const mongoose = require('mongoose');

const launchesSchema = new mongoose.Schema({
  flightNumber: {
    type: Number,
    required: true,
  },
  launchDate: {
    type: Date,
    required: true,
  },
  mission: {
    type: String,
    required: true,
  },
  rocket: {
    type: String,
    required: true,
  },
  target: {
    type: String,

```

```

    },
    customers: [ String ],
    upcoming: {
      type: Boolean,
      required: true,
    },
    success: {
      type: Boolean,
      required: true,
      default: true,
    },
  },
});

```

```

// Connects launchesSchema with the "launches" collection
module.exports = mongoose.model('Launch', launchesSchema);

```

Файл planets.model.js

```

const fs = require('fs');
const path = require('path');
const { parse } = require('csv-parse');

const planets = require('./planets.mongo');

function isHabitablePlanet(planet) {
  return planet['koi_disposition'] === 'CONFIRMED'
    && planet['koi_insol'] > 0.36 && planet['koi_insol'] <
1.11    && planet['koi_prad'] < 1.6;
}

function loadPlanetsData() {
  return new Promise((resolve, reject) => {
    fs.createReadStream(path.join(__dirname, '..', '..',
'data', 'kepler_data.csv'))

```

```

        .pipe(parse({
            comment: '#',
            columns: true,
        }))
        .on('data', async (data) => {
            if (isHabitablePlanet(data)) {
                savePlanet(data);
            }
        })
        .on('error', (err) => {
            console.log(err);
            reject(err);
        })
        .on('end', async () => {
            const countPlanetsFound = (await
getAllPlanets()).length;
            console.log(`${countPlanetsFound} habitable planets
found!`);
            resolve();
        });
    });
}

async function getAllPlanets() {
    return await planets.find({}, {
        '_id': 0, '__v': 0,
    });
}

async function savePlanet(planet) {
    try {
        await planets.updateOne({
            keplerName: planet.kepler_name,
        }, {
            keplerName: planet.kepler_name,
        }, {

```

```

        upsert: true,
      });
    } catch(err) {
      console.error(`Could not save planet ${err}`);
    }
  }
}

module.exports = {
  loadPlanetsData,
  getAllPlanets,
};

```

Файли planets.mongo.js

```

const mongoose = require('mongoose');

const planetsSchema = new mongoose.Schema({
  keplerName: {
    type: String,
    required: true,
  }
});

// Connects planetsSchema with 'planets' collection
module.exports = mongoose.model('Planet', planetsSchema);

```

Директорія routes/launches

Файл launches.controller.js

```

const {
  getAllLaunches,
  scheduleNewLaunch,
  existsLaunchWithId,
  abortLaunchById,
} = require('../models/launches.model');

```

```

const {
  getPagination,
} = require('../services/query');

async function httpGetAllLaunches(req, res) {
  const { skip, limit } = getPagination(req.query);
  const launches = await getAllLaunches(skip, limit);
  return res.status(200).json(launches);
}

async function httpAddNewLaunch(req, res) {
  const launch = req.body;

  if (!launch.mission || !launch.rocket || !launch.launchDate
    || !launch.target) {
    return res.status(400).json({
      error: 'Missing required launch property',
    });
  }

  launch.launchDate = new Date(launch.launchDate);
  if (isNaN(launch.launchDate)) {
    return res.status(400).json({
      error: 'Invalid launch date',
    });
  }

  await scheduleNewLaunch(launch);
  return res.status(201).json(launch);
}

async function httpAbortLaunch(req, res) {
  const launchId = Number(req.params.id);

  const existsLaunch = await existsLaunchWithId(launchId);

```

```

    if (!existsLaunch) {
        return res.status(404).json({
            error: 'Launch not found',
        });
    }

    const aborted = await abortLaunchById(launchId);
    if (!aborted) {
        return res.status(400).json({
            error: 'Launch not aborted',
        });
    }

    return res.status(200).json({
        ok: true,
    });
}

module.exports = {
    httpGetAllLaunches,
    httpAddNewLaunch,
    httpAbortLaunch,
};

```

Файл launches.router.js

```

const express = require('express');
const {
    httpGetAllLaunches,
    httpAddNewLaunch,
    httpAbortLaunch,
} = require('./launches.controller');

const launchesRouter = express.Router();

```

```

launchesRouter.get('/', httpGetAllLaunches);
launchesRouter.post('/', httpAddNewLaunch);
launchesRouter.delete('/:id', httpAbortLaunch);

```

```

module.exports = launchesRouter;

```

Файл launches.test.js

```

const request = require('supertest');
const app = require('../..//app');
const {
  mongoConnect,
  mongoDisconnect,
} = require('../..//services/mongo');
const {
  loadPlanetsData,
} = require('../..//models/planets.model');

describe('Launches API', () => {
  beforeAll(async () => {
    await mongoConnect();
    await loadPlanetsData();
  });

  afterAll(async () => {
    await mongoDisconnect();
  });

  describe('Test GET /launches', () => {
    test('It should respond with 200 success', async () => {
      const response = await request(app)
        .get('/v1/launches')
        .expect('Content-Type', /json/)
        .expect(200);
    });
  });
});

```

```

});

describe('Test POST /launch', () => {
  const completeLaunchData = {
    mission: 'USS Enterprise',
    rocket: 'NCC 1701-D',
    target: 'Kepler-62 f',
    launchDate: 'January 4, 2028',
  };

  const launchDataWithoutDate = {
    mission: 'USS Enterprise',
    rocket: 'NCC 1701-D',
    target: 'Kepler-62 f',
  };

  const launchDataWithInvalidDate = {
    mission: 'USS Enterprise',
    rocket: 'NCC 1701-D',
    target: 'Kepler-62 f',
    launchDate: 'zoot',
  };

  test('It should respond with 201 created', async () => {
    const response = await request(app)
      .post('/v1/launches')
      .send(completeLaunchData)
      .expect('Content-Type', /json/)
      .expect(201);

    const requestDate = new
Date(completeLaunchData.launchDate).valueOf();

    const responseDate = new
Date(response.body.launchDate).valueOf();
    expect(responseDate).toBe(requestDate);
  });
});

```

```

expect(response.body).toMatchObject(launchDataWithoutDate);
    });

    test('It should catch missing required properties', async
    () => {
        const response = await request(app)
            .post('/v1/launches')
            .send(launchDataWithoutDate)
            .expect('Content-Type', /json/)
            .expect(400);

        expect(response.body).toStrictEqual({
            error: 'Missing required launch property',
        });
    });

    test('It should catch invalid dates', async () => {
        const response = await request(app)
            .post('/v1/launches')
            .send(launchDataWithInvalidDate)
            .expect('Content-Type', /json/)
            .expect(400);

        expect(response.body).toStrictEqual({
            error: 'Invalid launch date',
        });
    });
});

```

Директорія routes/planets

Файл api.js

```
const express = require('express');
```

```

const planetsRouter = require('./planets/planets.router');
const launchesRouter = require('./launches/launches.router');

const api = express.Router();

api.use('/planets', planetsRouter);
api.use('/launches', launchesRouter);

module.exports = api;

```

Папка services

Файл mongo.js

```

const mongoose = require('mongoose');

require('dotenv').config();

// Update below to match your own MongoDB connection string.
const MONGO_URL = process.env.MONGO_URL;

mongoose.connection.once('open', () => {
  console.log('MongoDB connection ready!');
});

mongoose.connection.on('error', (err) => {
  console.error(err);
});

async function mongoConnect() {
  await mongoose.connect(MONGO_URL);
}

async function mongoDisconnect() {
  await mongoose.disconnect();
}

```

```

}

module.exports = {
  mongoConnect,
  mongoDisconnect,
}

```

Файл query.js

```

// src/services/query.js

function getPagination(query) {

  const limit = Math.abs(query.limit) || 0;
  const page = Math.abs(query.page) || 1;
  const skip = (page - 1) * limit;

  return {
    skip,
    limit,
  };
}

module.exports = {
  getPagination,
};

```

Папка src

Файл app.js

```

const path = require('path');
const express = require('express');
const cors = require('cors');
const morgan = require('morgan');

```

```

const api = require('./routes/api');

const app = express();

app.use(cors({
  origin: 'http://localhost:3000',
}));
app.use(morgan('combined'));

app.use(express.json());
app.use(express.static(path.join(__dirname, '..',
'public')));

app.use('/v1', api);

app.get('/*', (req, res) => {
  res.sendFile(path.join(__dirname, '..', 'public',
'index.html'));
});

module.exports = app;

```

Файл server.js

```

const http = require('http');

require('dotenv').config();

const app = require('./app');
const { mongoConnect } = require('./services/mongo');
const { loadPlanetsData } = require('./models/planets.model');
const { loadLaunchData } = require('./models/launches.model');

const PORT = process.env.PORT || 8000;

```

```
const server = http.createServer(app);

async function startServer() {
  await mongoConnect();
  await loadPlanetsData();
  await loadLaunchData();

  server.listen(PORT, () => {
    console.log(`Listening on port ${PORT}...`);
  });}
```

ДОДАТОК Г**Код проєкту структурований у стилі MVC (зовнішня частина)**

Файл Centered.js

```
import { withStyles } from "arwes";

const styles = () => ({
  root: {
    margin: "0 auto",
    maxWidth: 800,
  },
  "@media (max-width: 800px)": {
    root: {
      margin: "0 12px",
    }
  }
});

const Centered = props => {
  const {
    classes,
    className,
    children,
    ...rest
  } = props;
  return (
    <div      className={` ${classes.root}      ${className}`}
    {...rest}>
      {children}
    </div>
  );
};

export default withStyles(styles)(Centered);
```

Файл Clickable.js

```
import { withSounds } from "arwes";
```

```

const Clickable = props => {
  const {
    children,
    sounds,
    onClick,
    ...rest
  } = props;

  const clickWithSound = (e) => {
    sounds.click && sounds.click.play();
    onClick && onClick(e);
  };

  return (
    <span {...rest} onClick={clickWithSound}>
      {children}
    </span>
  );
};

export default withSounds()(Clickable);

```

Footer.js

```

import { Footer as ArwesFooter, Paragraph } from "arwes";
import Centered from "../Centered";

const Footer = () => {
  return <ArwesFooter animate>
    <Centered>
      <Paragraph style={{ fontSize: 14, margin: "10px 0" }}>
        This is not an official site and is not affiliated with NASA
        or SpaceX in any way. For educational purposes only.
      </Paragraph>
    </Centered>
  </ArwesFooter>

```

```

    </ArwesFooter>
  };

  export default Footer;

```

Header.js

```

import {
  Logo,
  Words,
  Header as ArwesHeader,
  Highlight,
  withStyles,
} from "arwes";
import { Link } from "react-router-dom";
import Clickable from "../Clickable";
import Centered from "../Centered";

const styles = theme => ({
  root: {
    display: "flex",
    flexDirection: "row",
    lineHeight: "80px",
  },
  logo: {
    display: "inherit",
    marginTop: "15px",
  },
  nav: {
    display: "inherit",
  },
  banner: {
    display: "inherit",
    fontWeight: "bold",
    marginLeft: "10px",
  },
});

```

```

    marginRight: "15px",
    fontSize: 28,
  },
  clickable: {
    fontSize: 21,
    "& i": {
      marginRight: theme.padding / 2,
      fontSize: 24,
    },
  },
  link: {
    color: theme.color.content,
    textDecoration: "none",
  },
  button: {
    padding: [0, theme.padding / 2],
  },
"@media (max-width: 800px)": {
  logo: {
    display: "none",
  },
  img: {
    display: "none",
  },
  banner: {
    display: "none",
  },
  button: {
    padding: [0, 8],
  },
  clickable: {
    fontSize: 16,
  }
},
});

```

```
const Header = props => {  
  const { classes, onNav, ...rest } = props;  
  return <ArwesHeader animate>  
    <Centered className={classes.root} {...rest}>  
      
    <Logo      animate      size={50}      className={classes.logo}
layer="header" />
    <Words animate className={classes.banner}>
        Mission Control
    </Words>
    <nav className={` ${classes.nav} `}>
        <Clickable className={classes.clickable} onClick={onNav}>
            <Highlight      className={classes.button}      animate
layer="header">
                <Link className={classes.link} to="/launch">
                    <i      className="material-
icons">check_circle_outline</i>Launch
                </Link>
            </Highlight>
        </Clickable>
        <Clickable className={classes.clickable} onClick={onNav}>
            <Highlight      className={classes.button}      animate
layer="header">
                <Link className={classes.link} to="/upcoming">
                    <i className="material-icons">update</i>Upcoming</Link>
                </Highlight>
            </Clickable>
        <Clickable className={classes.clickable} onClick={onNav}>
            <Highlight      className={classes.button}      animate
layer="header">
                <Link className={classes.link} to="/history">
                    <i className="material-icons">history</i>History</Link>
                </Highlight>
            </Clickable>
        </nav>
    </Centered>
</ArwesHeader>

```

```
};
export default withStyles(styles)(Header);
```

requests.js

```
const API_URL = 'v1';

// Load planets and return as JSON.
async function httpGetPlanets() {
  const response = await fetch(`${API_URL}/planets`);
  return await response.json();
}

// Load launches, sort by flight number, and return as JSON.
async function httpGetLaunches() {
  const response = await fetch(`${API_URL}/launches`);
  const fetchedLaunches = await response.json();
  return fetchedLaunches.sort((a, b) => {
    return a.flightNumber - b.flightNumber;
  });
}

// Submit given launch data to launch system.
async function httpSubmitLaunch(launch) {
  try {
    return await fetch(`${API_URL}/launches`, {
      method: "post",
      headers: {
        "Content-Type": "application/json",
      },
```

```

        body: JSON.stringify(launch),
    });
} catch(err) {
    return {
        ok: false,
    };
}
}

// Delete launch with given ID.
async function httpAbortLaunch(id) {
    try {
        return await fetch(`${API_URL}/launches/${id}`, {
            method: "delete",
        });
    } catch(err) {
        console.log(err);
        return {
            ok: false,
        };
    }
}

export {
    httpGetPlanets,
    httpGetLaunches,
    httpSubmitLaunch,
    httpAbortLaunch,
};

```

Файл useLaunches.js

```
import { useCallback, useEffect, useState } from "react";

import {
  httpGetLaunches,
  httpSubmitLaunch,
  httpAbortLaunch,
} from '../requests';

function useLaunches(onSuccessSound, onAbortSound, onFailureSound)
{
  const [launches, saveLaunches] = useState([]);
  const [isPendingLaunch, setPendingLaunch] = useState(false);

  const getLaunches = useCallback(async () => {
    const fetchedLaunches = await httpGetLaunches();
    saveLaunches(fetchedLaunches);
  }, []);

  useEffect(() => {
    getLaunches();
  }, [getLaunches]);

  const submitLaunch = useCallback(async (e) => {
    e.preventDefault();
    setPendingLaunch(true);
    const data = new FormData(e.target);
    const launchDate = new Date(data.get("launch-day"));
    const mission = data.get("mission-name");
    const rocket = data.get("rocket-name");
```

```

const target = data.get("planets-selector");
const response = await httpSubmitLaunch({
  launchDate,
  mission,
  rocket,
  target,
});

const success = response.ok;
if (success) {
  getLaunches();
  setTimeout(() => {
    setPendingLaunch(false);
    onSuccessSound();
  }, 800);
} else {
  onFailureSound();
}
}, [getLaunches, onSuccessSound, onFailureSound]);

const abortLaunch = useCallback(async (id) => {
  const response = await httpAbortLaunch(id);

  const success = response.ok;
  if (success) {
    getLaunches();
    onAbortSound();
  } else {
    onFailureSound();
  }
}

```

```

    }, [getLaunches, onAbortSound, onFailureSound]);

    return {
      launches,
      isPendingLaunch,
      submitLaunch,
      abortLaunch,
    };
  }
}
export default useLaunches;

```

Файл usePlanets.js

```

import { useCallback, useEffect, useState } from "react";
import { httpGetPlanets } from "../requests";

function usePlanets() {
  const [planets, savePlanets] = useState([]);

  const getPlanets = useCallback(async () => {
    const fetchedPlanets = await httpGetPlanets();
    savePlanets(fetchedPlanets);
  }, []);

  useEffect(() => {
    getPlanets();
  }, [getPlanets]);
}

```

```

    return planets;
  }

  export default usePlanets;

```

Файл App.js

```

import {
  BrowserRouter as Router,
} from "react-router-dom";
import {
  Arwes,
  SoundsProvider,
  ThemeProvider,
  createSounds,
  createTheme,
} from "arwes";

import AppLayout from "../pages/AppLayout";

import { theme, resources, sounds } from "../settings";

const App = () => {
  return <ThemeProvider theme={createTheme(theme)}>
    <SoundsProvider sounds={createSounds(sounds)}>
      <Arwes      animate      background={resources.background.large}
      pattern={resources.pattern}>
        {anim => (
          <Router>
            <AppLayout show={anim.entered} />

```

```

        </Router>
      ) }
    </Arwes>
  </SoundsProvider>
</ThemeProvider>;
};

export default App;

```

Файл index.js

```

import React from "react";
import ReactDOM from "react-dom";
import App from "./App";

ReactDOM.render(
  <App />,
  document.getElementById("root")
);

```

Файл settings.js

```

const resources = {
  background: {
    small: "/img/background-small.jpg",
    medium: "/img/background-medium.jpg",
    large: "/img/background-large.jpg",
  },
  pattern: "/img/glow.png",
};

```

```
const sounds = {
  shared: {
    volume: 0.5,
  },
  players: {
    click: {
      sound: { src: ["/sound/click.mp3"] },
      settings: { oneAtATime: true }
    },
    typing: {
      sound: { src: ["/sound/typing.mp3"] },
      settings: { oneAtATime: true }
    },
    deploy: {
      sound: { src: ["/sound/deploy.mp3"] },
      settings: { oneAtATime: true }
    },
    success: {
      sound: {
        src: ["/sound/success.mp3"],
        volume: 0.2,
      },
      settings: { oneAtATime: true }
    },
    abort: {
      sound: { src: ["/sound/abort.mp3"] },
      settings: { oneAtATime: true }
    },
    warning: {
      sound: { src: ["/sound/warning.mp3"] },
```

```
        settings: { oneAtATime: true }
      },
    }
  };

const theme = {
  color: {
    content: "#alecfb",
  },
  padding: 20,
  responsive: {
    small: 600,
    medium: 800,
    large: 1200
  },
  typography: {
    headerFontFamily: '"Titillium Web", "sans-serif"',
  },
};

export {
  resources,
  sounds,
  theme,
};
```

ДОДАТОК Д**Приклад налаштування проміжного програмного забезпечення CORS**

Додавання CORS Middleware у NestJS

У NestJS, налаштування CORS (Cross-Origin Resource Sharing) є дуже простим завдяки вбудованій підтримці. Ось як це можна зробити.

1. Інсталяція пакета cors

Для використання CORS ми додамо middleware. У випадку з NestJS, ми використовуємо добре відомий npm-пакет cors.

Виконайте команду в терміналі

```
npm install cors
```

2. Додавання CORS Middleware

Наступний крок – інтегрувати CORS у ваш сервер NestJS. Додайте наступний код у файл main.ts

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import * as cors from 'cors';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  // Використання CORS Middleware
  app.use(
    cors({
      origin: 'http://localhost:3000', // Дозволяє запити
      тільки з цього походження
      methods: 'GET,HEAD,PUT,PATCH,POST,DELETE', // Дозволені
      HTTP-методи
      credentials: true, // Дозволяє надсилати куки
    }),
  );

  await app.listen(3000);} bootstrap();
```

3. Використання вбудованої підтримки CORS

NestJS також має власну підтримку CORS. Активуємо її простішим способом.

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  // Увімкнення CORS через опції
  app.enableCors({
    origin: 'http://localhost:3000', // Дозволяє тільки цей
    клієнт
    methods: 'GET,HEAD,PUT,PATCH,POST,DELETE',
    credentials: true,
  });

  await app.listen(3000);
}
bootstrap();
```

4. Можливості whitelist.

Якщо ваш API використовується декількома клієнтами з різними походженнями, ви можете динамічно налаштувати whitelist. Наприклад

```
const allowedOrigins = ['http://localhost:3000',
'http://example.com'];

app.enableCors({
  origin: (origin, callback) => {
    if (!origin || allowedOrigins.includes(origin)) {
      callback(null, true);
    } else {
      callback(new Error('Not allowed by CORS'));
    }
  },
});
```

5. Перевірка роботи

Після налаштування запустіть сервер, використовуючи команду

```
npm run start
```

Потім відкрийте ваш клієнт (наприклад, <http://localhost:3000>) та спробуйте зробити запит до сервера. У вкладці Network у браузері ви побачите, що заголовок Access-Control-Allow-Origin було встановлено коректно.

ДОДАТОК Е**Використання РМ2 для створення кластерів**

PM2 (Process Manager 2) – це інструмент для керування процесами в Node.js, який спрощує створення та керування кластерами серверів. Він забезпечує гнучкість, продуктивність і простоту у використанні.

Щоб встановити PM2, достатньо виконати в терміналі команду

```
npm install pm2
```

Краще додати PM2 до ваших залежностей у package.json, щоб зробити його частиною проєкту. Це не dev-залежність, адже PM2 використовується для продакшн-серверів.

Для зручності можна також встановити PM2 глобально

```
npm install -g pm2
```

Після встановлення PM2 можна використовувати його для запуску серверів. Наприклад, щоб запустити сервер, виконайте

```
pm2 start server.js
```

Це запускає ваш сервер як окремий процес. Ви можете переглядати статус процесу за допомогою `pm2 list` або `pm2 status`.

Щоб зупинити процес, виконайте

```
pm2 stop <ім'я або ID процесу>.
```

Для перезапуску

```
pm2 restart <ім'я або ID процесу>
```

Щоб повністю видалити процес із менеджера

```
pm2 delete <ім'я або ID процесу>
```

Використання кластерного режиму PM2.

PM2 підтримує кластерний режим, що дозволяє автоматично створювати кластер процесів без явного використання модуля `cluster` у коді.

Замість ручного управління процесами через `cluster.fork()`, можна дозволити PM2 робити це автоматично. Вам більше не потрібно перевіряти `isMaster` або писати логіку для створення `workers`. Достатньо залишити лише код для обробки запитів.

```
const http = require('http');

const server = http.createServer((req, res) => {
  res.end('Hello, World!');
});

server.listen(3000);
```

Щоб запустити кластер, виконайте команду

```
pm2 start server.js -i <кількість процесів>
```

-i дозволяє вказати кількість workers.

Ви можете використати max, щоб створити стільки процесів, скільки підтримує CPU вашого комп'ютера

```
pm2 start server.js -i max
```

Усі процеси можна переглянути за допомогою pm2 list. Кожен процес буде мати свій ID, статус, використання CPU та пам'яті.

Для перегляду реальних логів сервера використовуйте команду:

```
pm2 logs
```

PM2 зберігає історичні логи, які можна переглянути за допомогою:

```
pm2 logs --lines 200
```

Логи можуть бути збережені у файли, а також налаштовані на автоматичну ротацію, щоб уникнути створення занадто великих файлів.

PM2 значно спрощує керування процесами Node.js у продакшн-середовищі: перезапуск процесів після помилок чи змін у коді, перегляд статусу процесів та метрик, збереження та обробка логів, оптимізація використання ресурсів CPU.

ДОДАТОК Ж
Налаштування MongoDB Atlas

Інструкція з налаштування MongoDB Atlas.

Реєстрація облікового запису на MongoDB Atlas

Перейдіть на сайт MongoDB Atlas. Натисніть "Sign Up" для створення нового облікового запису. Заповніть форму реєстрації або увійдіть через обліковий запис Google. Після реєстрації вас буде перенаправлено на сторінку керування проектами.

Створення нового проекту та кластеру.

У верхньому правому куті натисніть "New Project", щоб створити новий проєкт. Назвіть свій проєкт і натисніть "Create Project". У вибраному проєкті натисніть "Build a Cluster". На сторінці створення кластеру оберіть Shared Cluster (безкоштовний рівень). Виберіть хмарного провайдера (наприклад, AWS, Google Cloud або Azure) та найближчий до вас регіон. Натисніть "Create Cluster". Зауважу що Безкоштовний рівень кластеру (M0) включає 512 МБ пам'яті, що ідеально підходить для навчальних і невеликих проєктів.

Налаштування доступу до бази даних.

Перейдіть до вкладки "Database Access" і натисніть "Add New Database User". Введіть ім'я користувача. Виберіть метод аутентифікації (пароль). Згенеруйте пароль і збережіть його, адже він знадобиться для підключення. Виберіть права доступу: "Read and Write to any Database" (читання та запис). Натисніть "Add User".

Далі перейдіть у вкладку "Network Access". Натисніть "Add IP Address". Для розробки оберіть "Allow Access from Anywhere" (це додає 0.0.0.0/0, що дозволяє доступ з будь-якої IP-адреси). Застосуйте зміни, натиснувши "Confirm".

Підключення до кластеру. Після створення кластеру натисніть "Connect" під ім'ям кластеру. Оберіть опцію "Connect your application". У розділі "Driver" оберіть Node.js (версія 3.6 або новіша). Скопіюйте згенерований URI-з'єднання. Він матиме вигляд [29]

```
mongodb+srv://<username>:<password>@<cluster-name>.mongodb.net/<database>?retryWrites=true&w=majority.
```

Замініть <username> і <password> на ваші дані.

Замість <database> вкажіть ім'я вашої бази даних.

Встановіть бібліотеку MongoDB для Node.js

`npm install mongodb.`

Підключіться до вашої бази даних у коді Node.js

```
const { MongoClient } = require('mongodb');

const uri = "mongodb+srv://<username>:<password>@<cluster-
name>.mongodb.net/<database>?retryWrites=true&w=majority";

const client = new MongoClient(uri);

async function run() {
  try {
    await client.connect();
    console.log("Підключення до MongoDB успішно!");
    const database = client.db("<database>");
    const collection = database.collection("<collection>");
    // Виконуйте операції з базою даних тут
  } finally {
    await client.close();
  }
}

run().catch(console.error);
```

ДОДАТОК И
Приклад продуктивності

Файл server.js

```
const express = require('express');

const app = express();

function delay(duration) {
  const startTime = Date.now();
  while (Date.now() - startTime < duration) {
    //event loop is blocked...
  }
}

app.get('/', (req, res) => {
  res.send(`Performance example: ${process.pid}`);
});

app.get('/timer', (req, res) => {
  delay(4000);
  res.send(`Beep beep beep! ${process.pid}`);
});

console.log('Running server.js...');
console.log('Worker process started.');
```

```
app.listen(3000);
```

Опис коду. Процесор і цикл обробки подій тут дуже, дуже зайняті. Обертання за таких умов відбувається так швидко, як тільки може наш процесор.

Цикл обробки `while` подій не продовжиться, поки не повернеться наша функція затримки, а це означає, що наш сервер не зможе відповідати на будь-які запити.

ДОДАТОК К
Реалізація кластеризації

Файл main.js

```

const express = require('express');
const cluster = require('cluster');
const os = require('os');

const app = express();

// Функція затримки за допомогою setTimeout (неблокуюча)
function delay(duration) {
  return new Promise((resolve) => setTimeout(resolve,
duration));
}

app.get('/', (req, res) => {
  res.send(`Performance example: ${process.pid}`);
});

app.get('/timer', async (req, res) => {
  await delay(9000); // Неблокуюча затримка на 9 секунд
  res.send(`Ding ding ding! ${process.pid}`);
});

console.log('Running server.js...');
if (cluster.isMaster) {
  console.log('Master process started...');

  // Динамічне створення workers відповідно до кількості ядер
  const numCPUs = os.cpus().length;
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('exit', (worker) => {
    console.log(`Worker      ${worker.process.pid}      exited.`

```

```

Starting a new worker...`);
    cluster.fork();    //  Перезапуск  worker,  якщо  він
завершується
    });
  } else {
    console.log(`Worker process started: ${process.pid}`);
    app.listen(3000, () => {
      console.log(`Worker ${process.pid} is listening on port
3000`);
    });
  }
}

```

Опис коду. Створення сервер на основі Node.js з використанням Express і кластерного модуля для покращення продуктивності. Він ілюструє, як головний процес (master) створює два робочих процеси (worker), що виконують серверний код паралельно.

На сервері є два основні маршрути.

Кореневий маршрут (/) – відповідає швидким текстовим повідомленням із зазначенням ідентифікатора процесу, що обробляє запит.

Маршрут таймера (/timer) – штучно блокує цикл обробки подій на заданий час (9 секунд), після чого надсилає відповідь із вказівкою ідентифікатора процесу.

При запуску сервер перевіряє, чи є процес головним. Якщо так, він створює дві копії (форки) робочих процесів. Кожен робочий процес окремо запускає сервер, слухаючи на порту 3000, і готовий обробляти запити паралельно, що дозволяє краще використовувати багатоядерність процесора.

ДОДАТОК Л
Слайди презентації

Національний університет «Запорізька політехніка»

Кафедра програмних засобів

АНАЛІЗ МЕТОДІВ ТА РЕАЛІЗАЦІЯ СУЧАСНОГО СТЕКУ ТЕХНОЛОГІЙ ДЛЯ
ПІДВИЩЕННЯ ПРОДУКТИВНОСТІ ВЕБ САЙТІВ

Дипломна кваліфікаційна робота

ANALYSIS OF METHODS AND IMPLEMENTATION OF A MODERN
TECHNOLOGY STACK FOR INCREASING THE PRODUCTIVITY OF WEBSITES

Виконав:

ст. гр. КНТ-223м

І.А. ШТЕПА

Керівник:

професор, к.т.н.

Г.В. ТАБУНЩИК

1

Рисунок К.1 – Слайд №1

Об'єкт та предмет дослідження, мета роботи

- **Об'єкт дослідження**

- Архітектура та інструменти для оптимізації вебдодатків.

- **Предмет дослідження**

Сучасні серверні технології, такі як Node.js, MongoDB, Express.js.

- **Мета роботи**

Підвищення продуктивності та масштабованості вебсайтів шляхом використання сучасного технологічного стеку.

2

Рисунок К.2 – Слайд №2

Використані технології

- **Node.js** - Асинхронна серверна платформа.
- **MongoDB** - Документоорієнтована база даних.
- **Express.js** - Фреймворк для побудови вебдодатків.
- **PM2** - Інструмент для моніторингу процесів.
- **MVC-архітектура** - Організація коду для зручності масштабування.

Рисунок К.3 – Слайд №3

Налаштування середовища

1. Завантаження Node.js та встановлення Visual Studio Code.
2. Налаштування NPM та використання nodemon.
3. Вибір стеку технологій (Frontend – React.js, Backend – Node.js).
4. Використання MongoDB Atlas для хмарного зберігання даних.

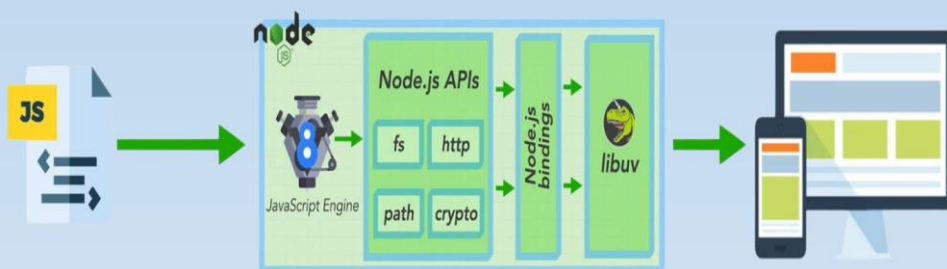


Рисунок 1 – Робота Node.js

Рисунок К.4 – Слайд №4

Архітектура додатку

• MVC-патерн:

- Model це робота з базами даних MongoDB.
- View це відображення даних через React.js.
- Controller це логіка взаємодії між компонентами.

• Кластеризація серверів для підвищення продуктивності.

• CORS для безпечного обміну ресурсами.

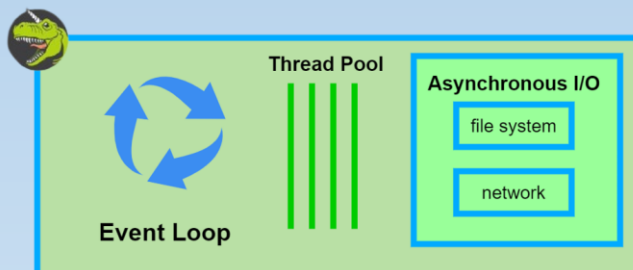


Рисунок 2 – Архітектура Node.js. Цикл подій, пул потоків і асинхронний ввід/вивід

5

Рисунок К.5 – Слайд №5

Актуальність теми

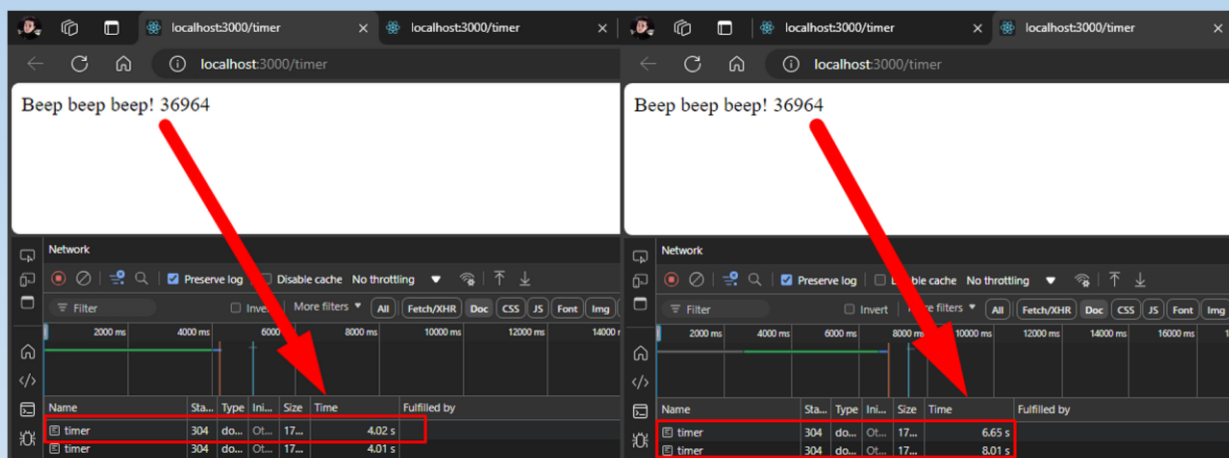


Рисунок 3 – Різниця часу у відпрацюванні сторінки водночас

6

Рисунок К.6 – Слайд №6

Мета роботи

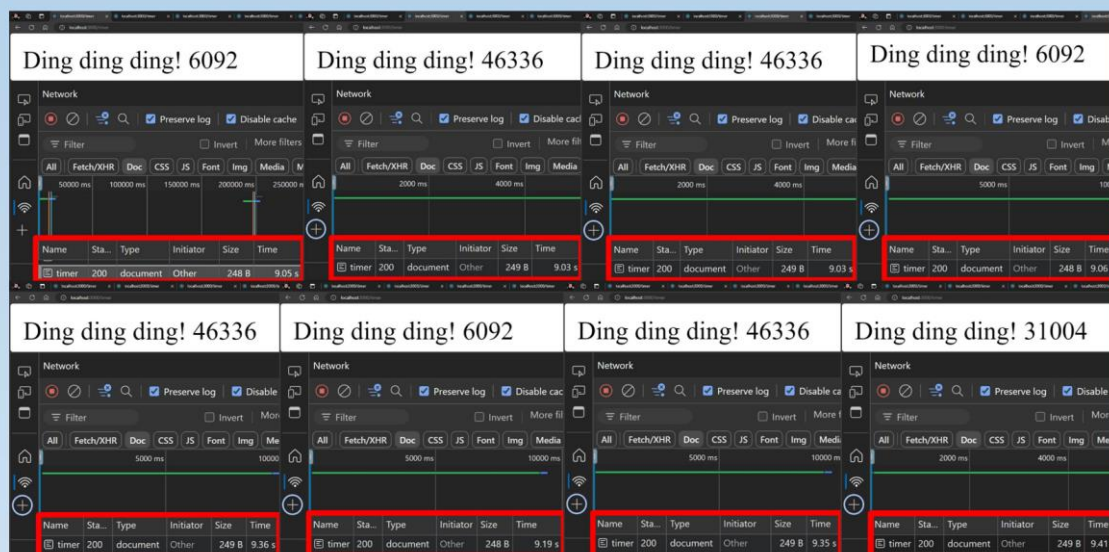


Рисунок 4 – Паралельна обробка 8 вкладок водночас за найбільше, менше 9 секунд

7

Рисунок К.7 – Слайд №7

Балансування навантаження

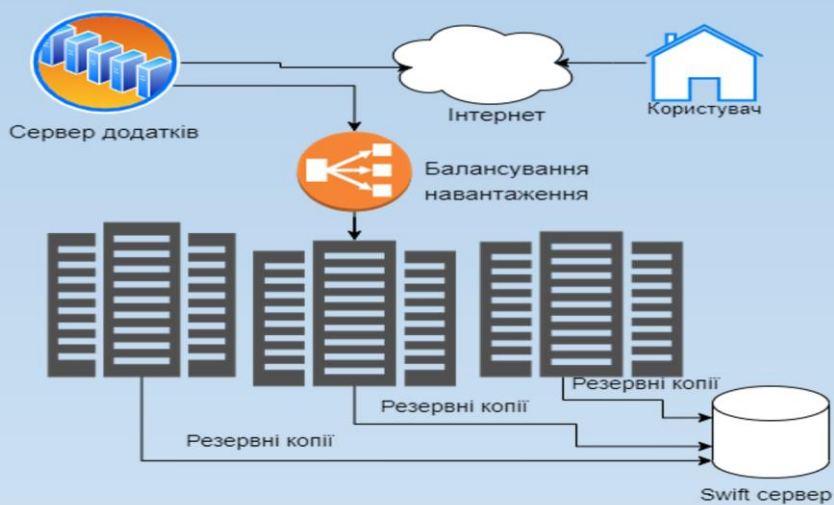


Рисунок 5 – Інфраструктура з балансуванням навантаження для вебсайту

8

Рисунок К.8 – Слайд №8

Node Server продуктивність



Рисунок 6 – Обробка запитів клієнта

9

Рисунок К.9 – Слайд №9

Бази даних

	MONGODB	POSTGRES
Тип	ДОКУМЕНТ	РЕЛЯЦІЙНИЙ
ОРГАНІЗОВАНО	КОЛЕКЦІЇ	ТАБЛИЦІ
МОВА ЗАПИТУ	NOSQL	SQL
МАСШТАБУВАННЯ	ГОРИЗОНТАЛЬНЕ	ВЕРТИКАЛЬНЕ

Рисунок 7 – Порівняння SQL та NoSQL

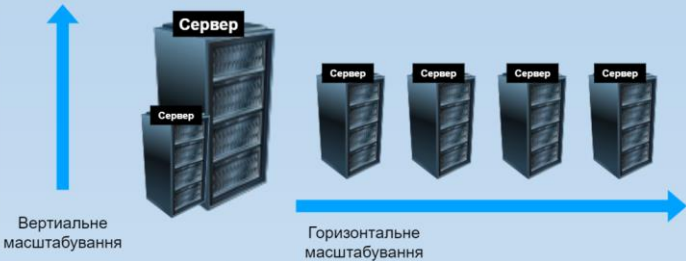


Рисунок 8 – Підходи організації даних SQL та NoSQL

10

Рисунок К.10 – Слайд №10

Результати тестування

- Продуктивність сервера під навантаженням збільшена на 35% завдяки кластеризації.
- Час відповіді скоротився з 500 мс до 150 мс, завдяки кластеризації та інструменту PM2 .
- Надійність системи підтверджена під час моделювання високих навантажень.

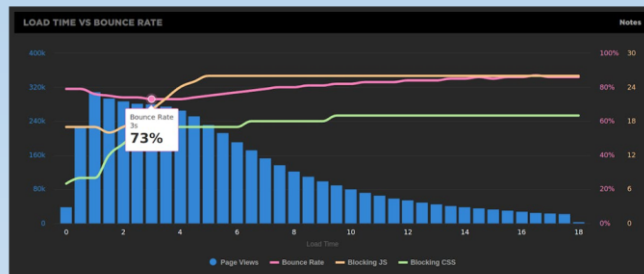


Рисунок 9 – Діаграми взаємодії. Кореляцію між ефективністю та залученістю користувачів

11

Рисунок К.11 – Слайд №11

Висновки

- Реалізовано сучасний стек технологій (Node.js, MongoDB, Express.js).
- Забезпечено масштабованість і безпеку вебдодатків.
- Система показала високу продуктивність під великим навантаженням.
- Можливе використання у навчальних платформах, CRM-системах, електронній комерції.

12

Рисунок К.12 – Слайд №12