ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка методики спрощення інтеграції з мікросервісами та сторонніми API у Node.js додатках»

на здобуття освітнього ступеня магістра
зі спеціальності 121 Інженерія програмного забезпечення
(код, найменування спеціальності)
освітньо-професійної програми «Інженерія програмного забезпечення»
(назва)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Іван ХОЛОД
(підпис)

Виконав: здобувач вищої освіти гр. ПДМ-62
Іван ХОЛОД

Керівник: Андрій БОНДАРЧУК
д.т.н., професор

Рецензент:
науковий ступінь, вчене звання
Ім'я, ПРІЗВИЩЕ

Київ 2024

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Магістр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
«_» _____ 2023 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Холоду Івану Петровичу

1. Тема кваліфікаційної роботи: «Розробка методики спрощення інтеграції з мікросервісами та сторонніми API у Node.js додатках»

керівник кваліфікаційної роботи _____ Андрій БОНДАРЧУК, д.т.н., професор,

затверджені наказом Державного університету інформаційно-комунікаційних технологій від « 19 » жовтня 2023 р. № 145.

2. Строк подання кваліфікаційної роботи « 29 » грудня 2023 р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, ефективна та надійна інтеграція мікросервісів та сторонніх API.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
   1. Аналіз функціональних особливостей Node.js.
   2. Розробка архітектури системи.

3. Аналіз та вибір технології розробки.
4. Проєктування функціоналу та модулів системи.
5. Розробка програмного забезпечення.

5. Перелік ілюстративного матеріалу: *презентація.*
   1. Задачі створення спрощеної інтеграції.
   2. Модель архітектури мікросервісів.
   3. Структура веб-додатку Node.js.
   4. Архітектура REST API.
   5. Тестування готового рішення.

6. Дата видачі завдання « 19 » жовтня 2023 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір науково-технічної літератури	19.10-05.11.23	
2	Аналіз функціональних особливостей Node js	06.11-12.11.23	
3	Розробка архітектури системи	13.11-19.11.23	
4	Аналіз та вибір технологій розробки	20.11-26.11.23	
5	Проєктування функціоналу та модулів системи	27.11-03.12.23	
6	Розробка програмного забезпечення	04.12-10.12.23	
7	Оформлення роботи та розробка демонстраційних матеріалів	11.12-20.12.23	
8	Здача роботи	21.12-29.12.23	

Здобувач вищої освіти _____ Іван ХОЛОД
(підпис)

Керівник
кваліфікаційної роботи _____ Андрій БОНДАРЧУК
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 94 стор., 1 табл., 49 рис., 36 джерел.

Метою кваліфікаційної роботи – є підвищення ефективності та надійності інтеграції з мікросервісами та сторонніми API у Node.js додатках шляхом розроблення та валідації методики.

Предметом дослідження – є процес інтеграції мікросервісів та сторонніх API в Node.js додатках з орієнтацією на визначення оптимальних методів оптимізації та спрощення цього процесу.

Об'єктом дослідження – є конкретні аспекти, що впливають на ефективність та надійність інтеграції мікросервісів та сторонніх API в Node.js додатках, включаючи розробку та валідацію методики для поліпшення цього процесу.

Короткий зміст роботи: У роботі було розроблено новаторську методику спрощення інтеграції з мікросервісами та сторонніми API для вдосконалення продуктивності та ефективності. Проаналізовано отримані результати тестування та ефективності методики з наданням рекомендацій щодо оптимізацій розробленого API.

КЛЮЧОВІ СЛОВА: API, РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, NODE.JS, PHP, ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, ВИСОКОПРОДУКТИВНІСТЬ, МІКРОСЕРВІСИ, ІНТЕГРАЦІЯ, ОПТИМІЗАЦІЯ, ШВИДКОДІЯ, ПРОДУКТИВНІСТЬ, ДІАГРАМИ ПОРІВНЯНЬ, ЕФЕКТИВНІСТЬ.

ABSTRACT

The text part of the qualification work for the master's degree: 94 pages, 1 table, 49 figures, 36 sources.

The purpose of the qualification work is to increase the efficiency and reliability of integration with microservices and third-party APIs in Node.js applications by developing and validating a methodology.

The subject of the research is the process of integrating microservices and third-party APIs in Node.js applications with a focus on determining the best methods for optimizing and simplifying this process.

The object of research - are specific aspects that affect the efficiency and reliability of the integration of microservices and third-party APIs in Node.js applications, including the development and validation of a methodology to improve this process.

Summary of work: In this paper, we developed an innovative methodology for simplifying integration with microservices and third-party APIs to improve performance and efficiency. The obtained results of testing and efficiency of the methodology are analyzed with recommendations for optimizing the developed API.

KEYWORDS: API, SOFTWARE DEVELOPMENT, NODE.JS, PHP, SOFTWARE TESTING, HIGH-PERFORMANCE, MICROSERVICES, INTEGRATION, OPTIMIZATION, SPEED, PERFORMANCE, COMPARISON CHARTS, EFFICIENCY.

ЗМІСТ

ВСТУП

Актуальність теми. В сучасному світі інтернет-технологій перед програмістами виникає важлива задача переносу продукту на різні платформи та створення додаткових клієнтів веб-сервісів. Оцінки експертів на 2017 рік свідчать, що близько 75% інтернет-трафіку припадатиме на мобільні пристрої, що вказує на зростання потреби у мобільних клієнтах інтернет-додатків та веб-сервісів [1]. У цьому контексті, використання API набуває ключового значення, дозволяючи розробникам використовувати готові блоки для збудови додатків та реалізації нових ідей.

Веб-додаток, який пропонує API розробникам, володіє великим потенціалом для розвитку. Користувачі можуть розширити можливості програми, знайти нові сфери застосування та інтеграції у власні веб-сервіси, що призведе до підвищення якості та відвідуваності цих сервісів [2]. Використання API також дозволяє створювати чітку структуру програм та розділяти програмну логіку від візуального представлення, що полегшує розробку та підтримку.

Дана робота спрямована на дослідження методів удосконалення розробки API для підвищення його швидкодії, надійності та забезпечення ефективного використання в умовах росту популярності мобільних пристроїв та розширення функціоналу веб-сервісів. Метою роботи є розробка методу, який не лише підвищить ефективність та захищеність API, але й спростить процес його розробки та інтеграції.

Зв'язок роботи з дослідницькими програмами, планами та темами.

Мета і завдання дослідження.

Метою дослідження – є підвищення ефективності та надійності інтеграції з мікросервісами та сторонніми API у Node.js додатках шляхом розроблення та валідації методики.

Для досягнення цієї мети вирішуються наступні наукові завдання.

1. Дослідити сучасні підходи та технології в галузі розробки API та інтеграції мікросервісів;

2. Розробити новаторську методику спрощення інтеграції з мікросервісами та сторонніми API для вдосконалення продуктивності та ефективності;
3. Провести тестування розробленого API та оцінити його продуктивність, використовуючи стандартні тестові набори та метрики ефективності;
4. Аналізувати отримані результати тестування та ефективності методики, надавати рекомендації щодо оптимізацій та подальшого розвитку розробленого API.

Предметом дослідження – є процес інтеграції мікросервісів та сторонніх API в Node.js додатках з орієнтацією на визначення оптимальних методів оптимізації та спрощення цього процесу.

Об'єктом дослідження – є конкретні аспекти, що впливають на ефективність та надійність інтеграції мікросервісів та сторонніх API в Node.js додатках, включаючи розробку та валідацію методики для поліпшення цього процесу.

Практичне значення отриманих результатів.

Розроблена методика для оптимізації розробки API застосовує сучасні технології та оптимізовані підходи. Результати тестування показали, що використання Node.js порівняно з PHP призвело до зниження часу виконання тестів на близько 50%. Це свідчить про значне підвищення продуктивності розробленого API. Крім того, методика спрощує інтеграцію з мікросервісами та сторонніми API, роблячи цей процес більш прозорим та зрозумілим для розробників.

Апробація отриманих результатів. Основні положення роботи були представлені та обговорені на наступних конференціях:

- Науково-практична конференція "Проблеми експлуатації та захисту інформаційно-комунікаційних систем", Київ, 2023.

1 АНАЛІЗ ФУНКЦІОНАЛЬНИХ ОСОБЛИВОСТЕЙ NODE.JS

1.1. Вступ до Node.js

Node.js є асинхронною керованою подіями виконавчою платформою JavaScript, яка має потужну, але компактну стандартну бібліотеку. Node.js Foundation, галузевий консорціум з відкритою моделлю управління, підтримує та підтримує її. Дві різні версії Node активно підтримуються: поточна (Current) і довгострокова підтримка (LTS, Long Term Support).

JavaScript стає однією з найважливіших мов у всіх галузях розробки програмного забезпечення з моменту появи Node.js у 2009 році. Ця зміна частково пов’язана з впровадженням специфікації ECMAScgipt 2015, яка усунула низку значних недоліків у попередніх версіях мови Node. Специфікація використовує JavaScript-ядро Google V8, засноване на шостій версії стандарту ECMAScgipt, яку іноді називають ES6 або ES2015. Ситуація також покращилася завдяки новаторським технологіям, таким як Node, React та Electrop, які дозволяють застосовувати JavaScript у всьому, від браузерів до серверів і платформ мобільних додатків. Найбільші підприємства поступово приймають JavaScript, а Microsoft навіть допомагає Node.

Таким чином, ми розглянемо технологію Node, її моделі управління подіями та причини, чому JavaScript стала чудовою мовою програмування загального призначення. Для початку ми розглянемо стандартний веб-додаток Node.

1.1.1 Огляд основних принципів Node.js

1) Типовий веб-додаток Node.

Взагалі кажучи, модель програмування з одним потоком є однією з сильних сторон Node та JavaScript. Хоча мовами програмування, що нещодавно з’явилися, такими як Rust і Co, намагаються надати безпечні інструменти паралельного програмування, Node працює з моделлю, яка використовується в браузері.

Програмні потоки (threads) є стандартним джерелом помилок. Браузерний код — це набір команд, які виконуються одна за одною, а не паралельно. Така модель не має сенсу для інтерфейсів користувача, оскільки люди не хочуть чекати швидких процесів, як-от звернення до файлів або даних по мережі. Для вирішення цієї проблеми браузери використовують події. Коли користувач клацає на кнопці, подія ініціюється і виконується функція, яка раніше була визначена, але ще не була виконана. Тим самим запобігають деяким проблемам багатопотокового програмування, таким як взаємні блокування ресурсів і стан гонки.

2) *Неблокуюче введення/виведення.*

Що це означає для програмування сервера? Подібно до інших виконавчих середовищ, у яких виникають повільність при запитах введення/виводу, таких як обробка дискових або мережевих операцій, Node.js намагається уникнути перешкод для виконання бізнес-логіки. Це досягається завдяки трьом ідеям, використовуваним у Node.js: події, асинхронні API та неблокуючий введення/виведення. З точки зору програміста Node.js, неблокуючий ввод/вивід є терміном нижчого рівня, що дозволяє програмі звертатися до мережних ресурсів, не зупиняючи виконання інших функцій. Після завершення мережної операції обробляється функція зворотного виклику для обробки результатів.

У веб-додатку Node.js для обробки замовлень використовується бібліотека веб-програмування Express (рис. 1.1). Клієнтський браузер надсилає запити, такі як придбання продуктів, перевірка стану запасів чи надсилання квитанції електронною поштою, і отримує відповідь у форматі JSON. Одночасно відбуваються інші процеси, такі як відправлення електронної пошти та оновлення бази даних. Цей код, хоч і написаний у стандартному імперативному стилі JavaScript, працює одночасно завдяки використанню неблокуючого введення/виведення.

Node.js використовує мережу для взаємодії з базою даних через бібліотеку libuv (рис. 1.1). Завдяки цій бібліотеці Node.js може здійснювати неблокуючі мережеві виклики операційної системи, що дозволяє виконувати мережеві операції без блокування. Навіть якщо ви використовуєте зручний JavaScript-синтаксис для

роботи з базою даних, такий як db.insert(query, err => {}), Node.js внутрішньо використовує оптимізовані операції з неблокуванням мережі.

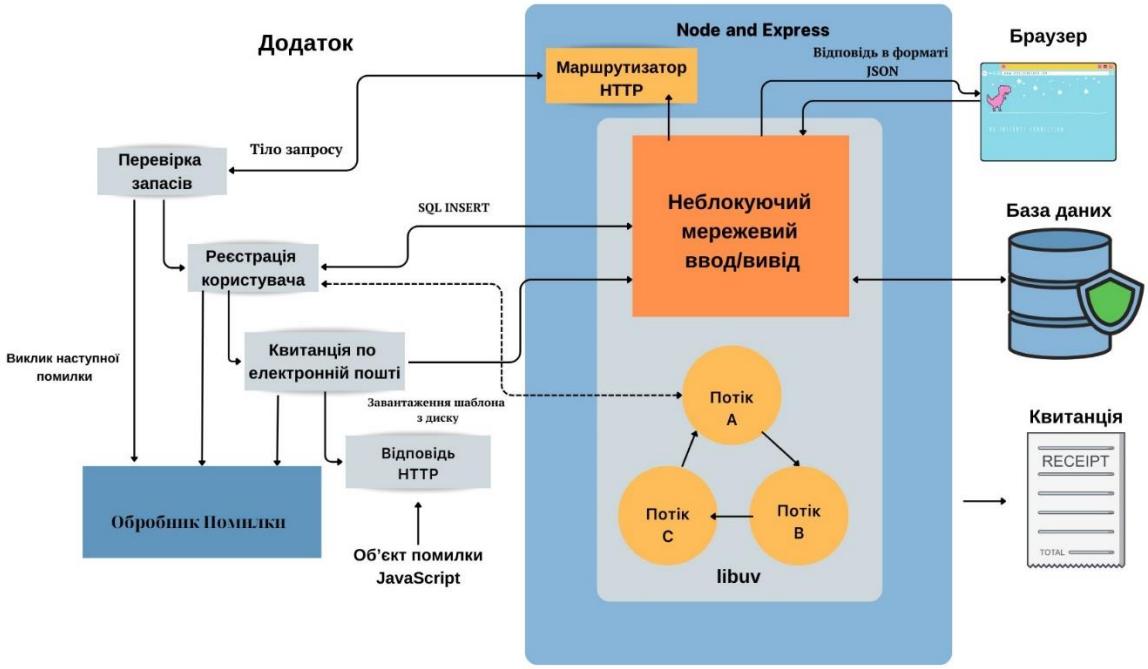


Рис. 1.1 Асинхронні та неблокуючі компоненти у програмах Node

Процес звернення до диска визначається приблизно так само, але, вражаюче, повний збіг відсутній. Пул потоків, що використовує libuv, створює враження використання неблокуючого виклику під час створення квитанції електронною поштою та читання шаблону повідомлення з диска. Хоча управління пулом потоків є відносно складним завданням, важливо осмислити команду для електронної пошти. Це значно простіше, ніж використання send('template.ejs', (err, html) => {}).

Головною перевагою використання асинхронних API з неблокуючими операціями введення/виведення є те, що Node.js може виконувати інші завдання під час виконання повільних процесів. Навіть якщо веб-додаток Node.js працює з одним потоком і процесом, він може обробляти одночасно багато підключень від тисяч користувачів сайту в будь-який момент. Для кращого розуміння цього механізму слід детальніше ознайомитися з циклом подій.

3) *Цикл подій.*

Отже, наш фокус може бути зосереджений на одному конкретному аспекті - обробці запитів браузера, як це вказано на рисунку 1.1. В цьому випадку програма використовує вбудований базовий модуль http для створення HTTP-сервера Node. Сервер використовує потоки, події та парсер HTTP-запитів Node з платформенного коду для ефективної обробки запитів. Тоді функція зворотного виклику у вашій програмі, яка була додана через бібліотеку веб-фреймворка Express, починає свою роботу. Ця функція викликає запит до бази даних, а додаток використовує HTTP для надсилання відповіді у форматі JSON. Весь цей процес включає щонайменше три неблокуючі мережеві дії: одну для запиту, одну для бази даних і одну для відповіді. Як Node планує і керує всіма цими неблокуючими мережевими операціями? Відповідь - це цикл подій. Зазначений набір мережевих операцій взаємодіє з циклом подій, як показано на рисунку 1.2.

Цикл подій пройшов кілька етапів та працює в одному напрямку, реалізуючи чергу типу "перший прийшов - перший вийшов" (FIFO). Спрощена послідовність основних фаз, які виконуються під час кожної ітерації циклу, представлена на рисунку 1.2. Таймери, заплановані функціями JavaScript setTimeout та setInterval, спочатку виконуються. Потім обробляються зворотні виклики введення/виведення, що відбувається, коли неблокуючий мережевий дзвінок повертає введення або виведення. Під час опитування нові події введення/виведення читаються, а зворотні виклики, заплановані функцією setImmediate, стартують наприкінці цього процесу. Це є особливим випадком, оскільки він дозволяє планувати негайне виконання зворотних дзвінків після поточних зворотних дзвінків введення/виведення, що вже перебувають у черзі. На важливо пам'ятати, що незважаючи на те, що Node використовує модель з одним потоком, він надає інструменти, що дозволяють писати ефективний та масштабований код.

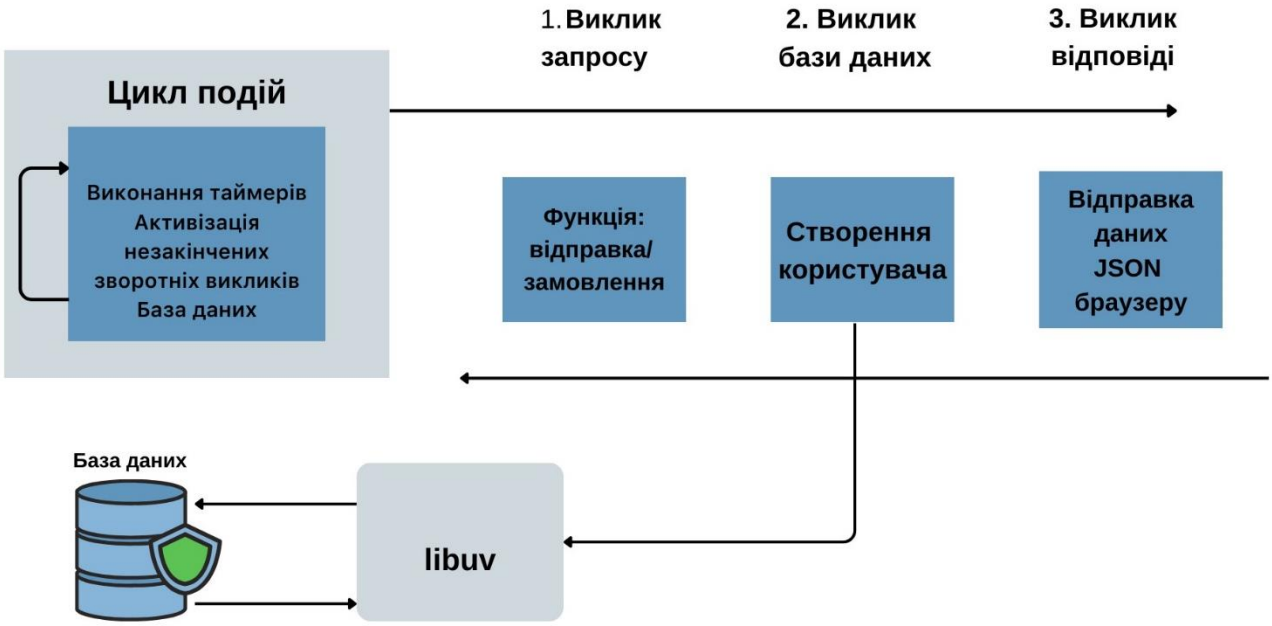


Рис. 1.2 Цикл подій

У прикладах використовуються "стрілкові" функції ES2015. Перш ніж продовжити, розглянемо деякі нові мовні можливості, що дозволяють писати більш ефективний код, оскільки Node підтримує численні функції JavaScript нового покоління.

Технологія Node базується на ядрі JavaScript V8, розробленому для браузера Google Chrome під назвою Chromium. За допомогою засобів оптимізації та прямої компіляції в машинний код V8 забезпечує високу швидкість виконання коду Node. Розділ 1.1.1 містить інформацію про додатковий вбудований компонент Node-libuv. У цьому розділі розглядаються операції введення та виведення; V8 включає в себе інтерпретацію та виконання коду JavaScript. Щоб використовувати libuv з V8, необхідно використовувати сполучний прошарок C++. На рисунку 1.3 зображено всі програмні компоненти, включаючи Node.

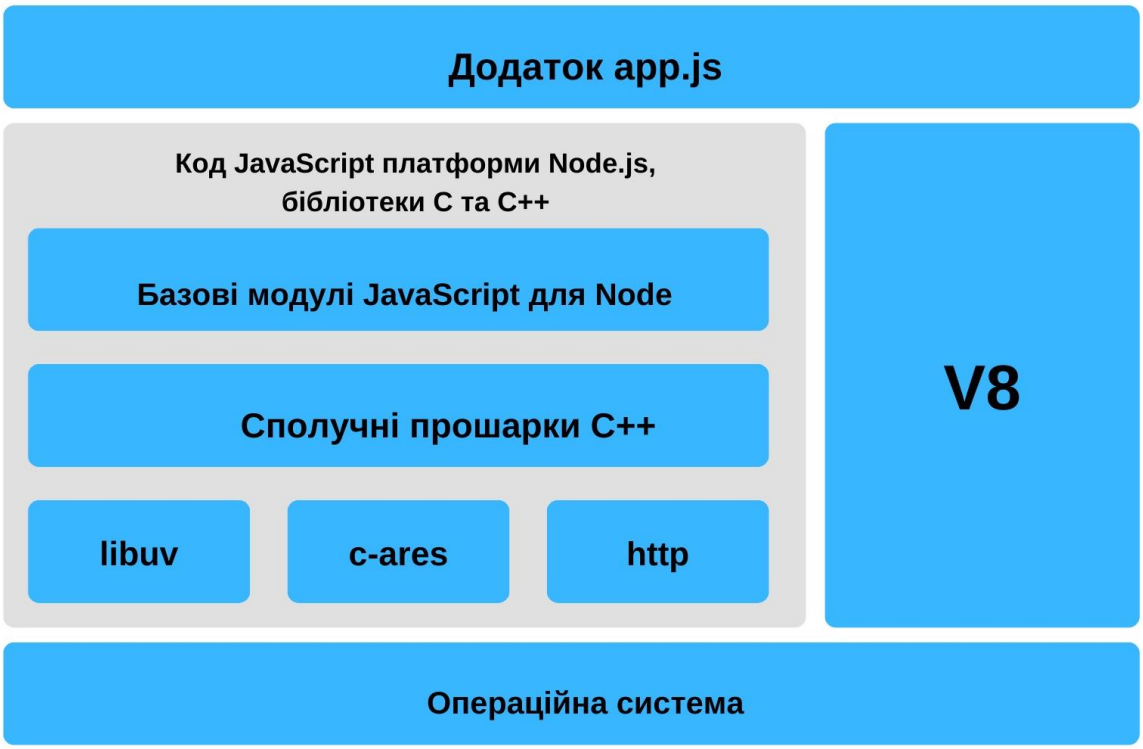


Рис . 1.3 Програмний стек Node

Таким чином, специфічні можливості JavaScript Node визначаються набором можливостей, що підтримуються 8. Функціональні групи відповідають за керування цією підтримкою.

4) *Графік випуску версій Node.*

Підтримка Node розділяється на три основні категорії: "поточна" (current), "щоденна" (nightly) і "довгострокова" (LTS). Для версій LTS надається 18-місячна та 12-місячна підтримка відповідно. Випуски Node керуються системою семантичного управління версіями (SemVer). У цій системі основний (major), додатковий (minor) і патч (patch) номери версії призначаються відповідно.

Наприклад, у версії 6.9.1 основний номер версії - 6, додатковий номер - 9, і номер патча - 1. Зміна основного номера версії може вказувати на те, що деякі API стали несумісними з попередніми версіями, і проекти повинні бути перевірені для нової версії Node.

В термінології Node, збільшення основного номера версії означає виходження нової поточної версії. Щоденні збірки автоматично створюються

кожні 24 години з останніми змінами, але, як правило, вони використовуються тільки для тестування нового функціоналу Node.

Вибір версії залежить від проекту та організації. LTS-версії надійно перевершують менш регулярні оновлення. Цей варіант ідеально підходить для великих компаній, для яких часті оновлення можуть викликати проблеми. Якщо важливі нововведення швидкості та функціональності, обирайте поточну версію.

1.1.2 Переваги асинхронності та подійно-орієнтованого підходу

Засоби асинхронного програмування.

Якщо ви коли-небудь займалися розробкою фронтенду, в якій події (наприклад, клацання мишею) ініціюють виконання логіки, то ви вже займалися асинхронним програмуванням. Асинхронне програмування на стороні сервера працює за тими ж принципами: відбуваються події, за якими спрацьовує логіка у відповідь. У світі Node для управління логікою відповідей використовуються дві популярні моделі: зворотні виклики та слухачі подій.

Зворотні дзвінки зазвичай визначають логіку для одноразових відповідей. Наприклад, при виконанні запиту до бази даних можна призначити функцію зворотного виклику, яка визначає, що потрібно зробити з результатами запиту. Функція зворотного дзвінка може вивести результати, виконати з ними певні обчислення або виконати іншу функцію зворотного дзвінка, використовуючи результати запиту як аргумент.

Слухачі подій є зворотні виклики, пов'язані з концептуальною сутністю (подією). Скажімо, клацання кнопкою миші це подія, яка обробляється у браузері. А код ходу в сервері HTTP генерує подію request при видачі запиту HTTP. Ви можете прослухати подію request та додати логіку відповіді. У наступному прикладі функція handleRequest буде викликатись кожного разу, коли генерується подія request; Для цього виклик методу EventEmitter.prototype. on пов'язує слухача події із сервером: server.on('request', handleRequest)

Примірник HTTP-сервера в Node є прикладом генератора подій - класу (EventEmitter), який може використовуватися при успадкування і який додає

можливість генерування та обробки подій. Багато аспектів базової функціональності Node успадковуються від Event Emitter; Ви також можете створити свій генератор подій.

Отже, ми з'ясували, що логіка відповіді зазвичай організується в Node одним із двох способів. Тепер можна перейти безпосередньо до асинхронного програмування; будуть розглянуті такі теми:

• як обробляти одноразові події зі зворотними викликами,

• як реагувати на події, що повторюються, за допомогою слухачів подій;

• як подолати деякі проблеми асинхронного програмування.

Почнемо з одного із найпоширеніших способів організації асинхронного коду: функцій зворотного виклику.

Файл JSON (titles.json), наведений у лістингу 1.4, відформатовано у вигляді масиву із заголовками повідомлень.

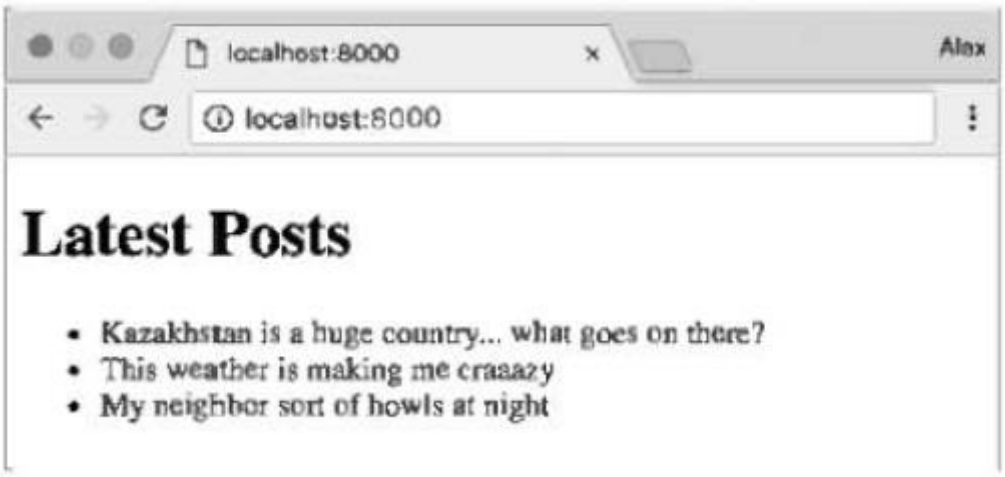


Рис. 1.4 HTML-відповідь від веб-сервера, який читає заголовки з файлу JSON та повертає результати у вигляді веб-сторінки

Виділимо характерні риси асинхронності та подійно-орієнтованого підходу в контексті розробки застосунків на платформі Node.js, так як, ці технології відіграють ключову роль у створенні ефективних, швидких та масштабованих програм:

1. Ефективність I/O операцій.

Однією з найбільших переваг Node.js є його здатність ефективно виконувати операції вводу/виводу (I/O) через асинхронний підхід. Коли ваш застосунок звертається до операцій, таких як читання з файлу чи відправлення запитів до бази даних, Node.js не блокує виконання інших завдань, дозволяючи досягти вражаючої продуктивності.

2. Масштабованість. Дізналися ви коли-небудь про те, як Node.js здатний обробляти тисячі одночасних з'єднань? Це досягається завдяки асинхронному підходу, який дозволяє обробляти багато з'єднань одночасно, уникнувши при цьому зайвих обмежень, які можуть виникнути при використанні традиційних, синхронних методів.

3. Швидкодія. Секрет швидкодії Node.js полягає в його здатності уникати блокувань та очікувань завершення операцій. Коли одне завдання чекає, Node.js просто переходить до виконання іншого, що забезпечує ефективне використання системних ресурсів та високу продуктивність.

4. Простота коду. Асинхронний та подійно-орієнтований підхід дозволяють писати менш складний та більш зрозумілий код. Вам не потрібно вдаватися в глибокі конструкції управління потоками – ви можете легко реагувати на події та виконувати відповідні дії, що спрощує розробку та управління кодом.

5. Розширюваність. Node.js підтримує модульну архітектуру та легко розширюється за рахунок використання модулів та пакетів. Ви можете легко інтегрувати різні бібліотеки та фреймворки, що робить ваш застосунок більш гнучким та розширюваним.

6. Єдність мови. Використання однієї мови програмування (JavaScript) і на клієнтській, і на серверній стороні полегшує розробку та забезпечує єдність коду. Ви можете використовувати ті самі знання та інструменти як на обох сторонах, що полегшує спільну роботу команд та розвиток проектів.

Отже, асинхронність та подійно-орієнтований підхід в Node.js грають ключову роль у створенні високопродуктивних та масштабованих застосунків. Ці технології допомагають розробникам ефективно використовувати ресурси, писати зрозумілий код та швидко реагувати на зміни в системі. Використання Node.js стає

справжньою вигодою для розробників, які стежать за високою продуктивністю та швидким розвитком своїх проектів.

1.2 Мікросервісна Архітектура

1.2.1 Переваги розробки монолітного дизайну

На початку процесу розробки додатки були відносно скромними, тому монолітна архітектура надавала їм більшу кількість переваг. Інтегроване середовище розробки (IDE) та інші інструменти розробки здебільшого орієнтовані на створення одного додатка. Приклад простоти внесення радикальних змін виникає тоді, коли можливо модифікувати як код, так і структуру бази даних, а потім сконструювати і розгорнути вихідні дані, які отримані в результаті.

Розробники змогли легко протестувати додаток, написавши наскрізні тести, які складалися з запуску програми, доступу до REST API та оцінки користувацького інтерфейсу за допомогою Selenium. Просто скопіювати WAR-файл на сервер, на якому вже встановлена копія Tomcat - це все, що потрібно розробнику для розгортання програми.

Застосунок працював з багатьма екземплярами програми, які були розміщені за балансувальником навантаження. Це дуже спростило масштабування програми.

Незважаючи на це, процес розробки, тестування та масштабування з часом значно ускладнився.

З іншого боку, багато розробників виявили, що ця стратегія має суттєвий недолік. Монолітні архітектури зазвичай є фундаментом, на якому будуються успішні програми. Як наслідок, команда розробників щоразу відповідає за реалізацію низки нових функцій, що призводить до збільшення кодової бази. Крім того, досягнення компанії з часом призвели до збільшення кількості програмістів, які працюють у компанії. Це не тільки прискорило темпи розширення кодової бази, але й призвело до збільшення обсягу адміністративної роботи, яку потрібно було виконувати.

Програма, яка колись була крихітною і простою, за час свого існування перетворилася на величезний моноліт, як показано на рисунку 1.2. Аналогічно, команда розробників, яка раніше була досить невеликою, сьогодні складається з багатьох скрам-команд, кожна з яких працює над окремим функціональним напрямком. Через те, що додаток переріс свою архітектуру, він вступив у монолітний період. Прогрес став нестерпно повільним і болючим. Використання гнучких підходів для розробки та розгортання коду наразі стало складним.

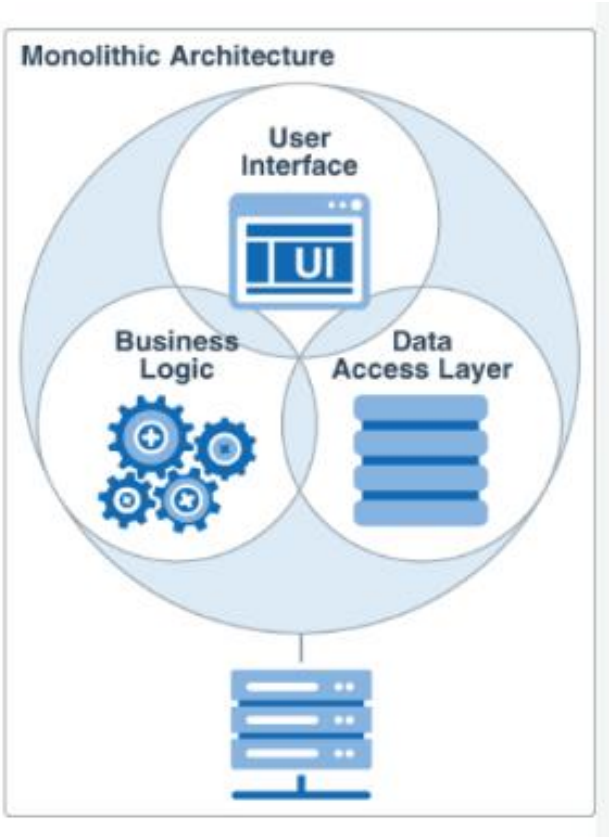


Рис. 1.5 Приклад монолітної архітектури

Всі зміни, внесені великою командою розробників, містяться в одному сховищі вихідного коду. Коли справа доходить до запуску коду у виробниче середовище, що вимагає ручного тестування, необхідно пройти довгий і складний шлях, перш ніж можна буде досягти збереження. Додаток збільшився в розмірі, складності та надійності, і його надзвичайно складно підтримувати. Надзвичайна складність програми є основною проблемою, яку необхідно вирішити. Осягнути її було б неможливо для одного розробника. Наслідком цього стало те, що процес

інтеграції нових функцій та виправлення помилок ставав дедалі складнішим і тривалішим. Дедлайни не були дотримані розробниками. Ситуація погіршується ще й тим, що складність, яка і так є непосильною, має тенденцію до експоненціального зростання. Якщо розробник має обмежене розуміння кодування, він не зможе вносити зміни належним чином. Кожна модифікація додає до коду ще один рівень складності, що значно ускладнює його подальше розуміння.

Ще одна проблема, яка виникає з додатком, полягає в тому, що він не є надійним. Як наслідок, середовище, в якому відбувається виробництво, часто порушується. Той факт, що програма настільки велика, ускладнює її належне тестування, що є однією з причин. Якщо тестування недостатнє, можна з упевненістю стверджувати, що в остаточну версію програми будуть внесені помилки. Програма набагато проблематичніша тим, що не має локалізації помилок, оскільки всі модулі виконуються в рамках одного процесу. У рідкісних випадках помилка в одному модулі, наприклад, витік пам'яті, може призвести до аварійного завершення роботи всіх екземплярів системи поспіль. Можливість того, що з вами зв'яжуться посеред ночі через несправність у промисловому середовищі, не є чимось, що особливо подобається розробникам. В результаті зниження доходів і довіри до організації, керівництво також незадоволене.

Останнє, з чим може зіткнутися команда, - це те, що архітектура, яка зумовила монолітну епоху, вимагає від них використання технологічного стеку, який стає все більш і більш застарілим. Поки це відбувається, розробникам складно переходити на нові фреймворки та мови програмування. Переробляти весь монолітний додаток, використовуючи нові технології, які нібито є кращими, було б небезпечною справою, яка до того ж коштувала б неймовірно дорого. Наслідком цього є те, що програмісти повинні працювати з тими інструментами, які вони обрали, коли вперше почали працювати над проектом. Як наслідок, їм часто доводиться підтримувати код, написаний з використанням застарілих технологій.

Оскільки фреймворк Spring продовжує вдосконалюватися з метою забезпечення зворотної сумісності, команда розробників теоретично повинна мати можливість оновитися до новіших версій фреймворку. На жаль, додаток

використовує версії фреймворку, які не сумісні з останніми випусками Spring. Їх оновлення ніколи не було пріоритетом для розробників. Як наслідок, значна частина коду розроблена з використанням технологій, які вже не використовуються. Програмісти також зацікавлені в експериментах з мовами, які не базуються на віртуальній машині Java (JVM), такими як Go Lang або Node.js. Монолітна архітектура, на жаль, не дозволяє розглядати цю альтернативу.

Отже, організації потрібно переходити до мікросервісного дизайну і рухатися вперед. Цікаво відзначити, що архітектура програмного забезпечення має дуже мало спільного з вимогами функціональних аспектів. Можна використовувати будь-яку архітектуру для того, щоб успішно реалізувати набір сценаріїв, які є функціональними вимогами до програми. Насправді, програми, які є успішними таким чином, зазвичай є великими і монолітними.

Само собою зрозуміло, що архітектура також важлива, оскільки вона відповідає за визначення так званих вимог до якості обслуговування, які також називають нефункціональними можливостями або атрибутами якості. Обслуговуваність, розширюваність і тестуємість - це три критерії якості, на які вплинуло розширення стандартних додатків. Ці атрибути є особливо важливими, оскільки вони впливають на швидкість доставки програмного забезпечення.

Дисциплінована команда, з одного боку, має можливість уповільнити процес сповзання в монолітне пекло з більшою швидкістю. Підтримувати модульність свого додатку - це те, що програмісти можуть робити старанно. На додаток до цього, вони здатні написати сценарій вичерпного автоматизованого тестування. З іншого боку, вони не зможуть обійти виклики, які притаманні величезним командам, що працюють над єдиною монолітною кодовою базою. Крім того, вони нічого не зможуть зробити зі стеком технологій, який постійно старіє. Єдиний варіант - відкладати неминуче якомога довше. Їм потрібно буде перейти на нову архітектуру, яка базується на мікросервісах, щоб вирватися з монолітної ери.

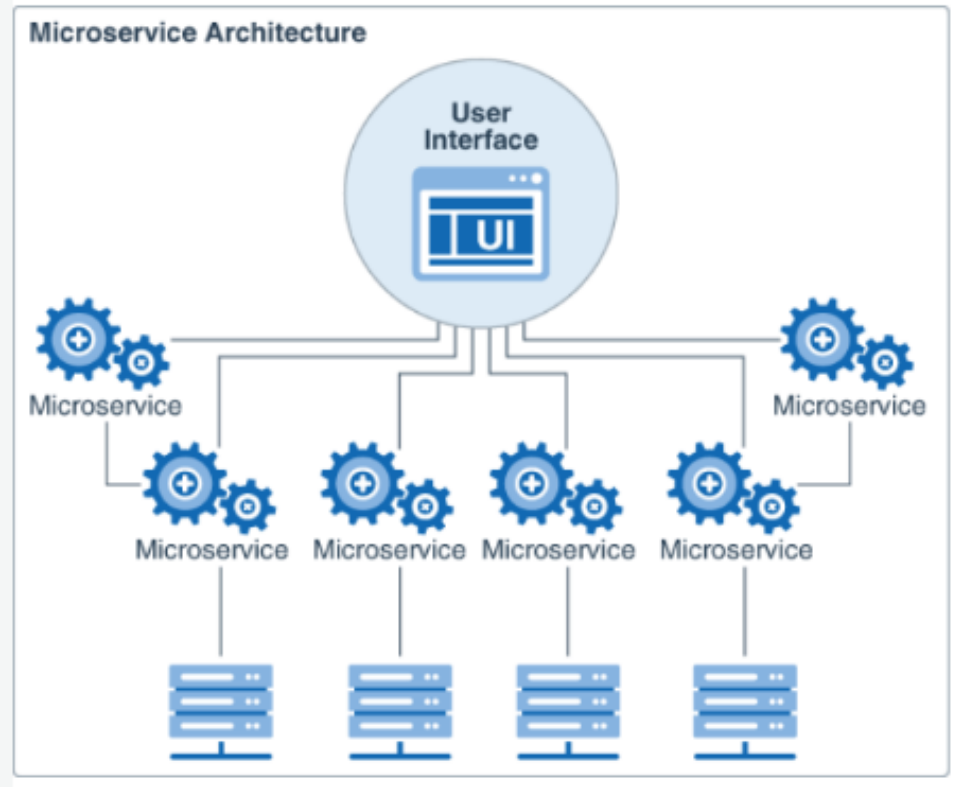


Рис. 1.6 Архітектура мікросервісу

У сучасному світі все більше фахівців погоджуються з тим, що при розробці великої та складної програми варто розглянути можливість використання мікросервісів. Але коли ми говоримо про мікросервіси, що саме ми маємо на увазі? Через те, що основний акцент робиться на об'ємі, сам термін не дає нам багато інформації. Архітектуру мікросервісів можна визначити нескінченною кількістю способів. Деякі люди сприймають назву сервісу в надто буквальному сенсі і стверджують, що він повинен бути надзвичайно маленьким, наприклад, складатися зі ста рядків коду. Дехто вважає, що розробка сервісу не повинна займати більше двох тижнів.

Після завершення аналізу ми можемо вважати мікросервісну архітектуру сервіс-орієнтованою архітектурою, яка складається з частин, що слабко пов'язані між собою і мають обмежений контекст.

Розробка великих, складних додатків значною мірою залежить від модульності як важливого компонента. Одна людина не в змозі осягнути всю складність сучасних додатків, оскільки вони занадто великі, щоб їх могла створити одна людина. Модулі, які створюються і розуміються різними людьми, повинні

використовуватися для розбиття додатків на складові частини. Монолітний проект характеризується наявністю модулів, які є комбінацією понять з мов програмування, таких як пакети на Java, і ресурсів, які беруть участь у процесі збірки, таких як JAR-файли. З іншого боку, дослідники та розробники виявили, що ця стратегія зазвичай не є особливо практичною. Додатки з монолітною архітектурою і тривалим життєвим циклом мають тенденцію ставати "неактуальними".

В рамках мікросервісної архітектури сервіс виступає одиницею модульності. API, які використовуються сервісами, виконують функцію непроникного бар'єру. API, на відміну від пакетів в Java, неможливо обійти, щоб отримати доступ до внутрішнього класу. Якщо розглядати це в довгостроковій перспективі, то це значно спрощує збереження модульності додатку. Використання сервісів як будівельних блоків пропонує додаткові переваги, такі як можливість запускати та розширювати будь-який окремий сервіс незалежно.

Мікросервісна архітектура відрізняється тим, що сервіси можуть з'єднуватися один з одним тільки через інтерфейси прикладного програмування (API). Можна досягти слабкого зв'язку, призначивши кожній службі окрему базу даних. Наприклад, у випадку онлайн-бізнесу, відділ замовлень і відділ обслуговування клієнтів можуть мати власні бази даних, кожна з яких містить таблиці для ЗАМОВЛЕНЬ і КЛІЄНТІВ відповідно. У будь-який момент в процесі розробки можна вносити зміни в структуру даних сервісу без необхідності узгодження з розробниками інших сервісів. Під час виконання програми сервіси відокремлені один від одного; наприклад, жоден з них не буде змушений чекати через те, що інший сервіс заблокував базу даних.

Модулі, про які ми говорили раніше, є аналогами великої кількості сервісів. Всі сервіси та їхні інтерфейси прикладного програмування (API) мають різні визначення, і в цьому полягає їхня відмінність. Розробка, тестування, розгортання та масштабування кожного з них може здійснюватися незалежно від інших. Архітектура сприяє збереженню модульності. Оминання інтерфейсу прикладного

програмування (API) сервісу не дозволяє розробнику отримати доступ до внутрішніх компонентів сервісу.

1.2.2 Визначення мікросервісів

Зокрема, функціональна декомпозиція є основною концепцією дизайну мікросервісів. Набір сервісів є кращим, ніж розробка одного величезного додатку, оскільки він є більш ефективним. Ідея про те, що архітектуру мікросервісів можна розглядати як форму функціональної декомпозиції, з одного боку, є досить практичною. З іншого боку, вона не дає вичерпних відповідей на ряд питань. Наприклад, який зв'язок між цим методом і більш загальним поняттям архітектури програмного забезпечення?

Архітектура програми - це її загальна структура, яка складається з різних компонентів, а також взаємозв'язків, що існують між цими компонентами. Якісні характеристики програми визначаються її архітектурою, саме тому архітектура має таке велике значення. Традиційно в архітектурі основна увага приділяється масштабованості, надійності та безпеці. Однак у сучасному світі можливість безпечного і своєчасного випуску коду також є життєво важливою і цінною характеристикою. Отже, мікросервісна архітектура - це стиль проектування, який полегшує обслуговування, тестування та розгортання додатків у простий спосіб.

1.2.3 Визначення архітектури ПЗ

Існує нескінченна кількість визначень архітектури програмного забезпечення. Нижче наведено приклад одного з найбільш точних визначень, які можна сформулювати: Термін "архітектура програмного забезпечення" відноситься до набору структур, які є важливими для обговорення обчислювальної системи. Ці структури включають програмні елементи, зв'язки між цими елементами та атрибути, які притаманні як елементам, так і зв'язкам.

Це визначення трохи абстрактне. З іншого боку, архітектура програми - це декомпозиція програми на складові частини (елементи) та взаємозв'язки, які існують між цими елементами. Процес декомпозиції є важливим з ряду причин.

Причина, чому це вигідно, полягає в тому, що це полегшує розподіл праці і знань, дозволяючи численним особам або командам, кожна з яких володіє потенційно вузькоспеціалізованими знаннями, ефективно співпрацювати над однією програмою. Крім того, він описує спосіб, у який частини програми взаємодіють одна з одною.

Якісні характеристики заявки визначаються поділом заявки на частини та взаємозв'язками, які існують між цими частинами.

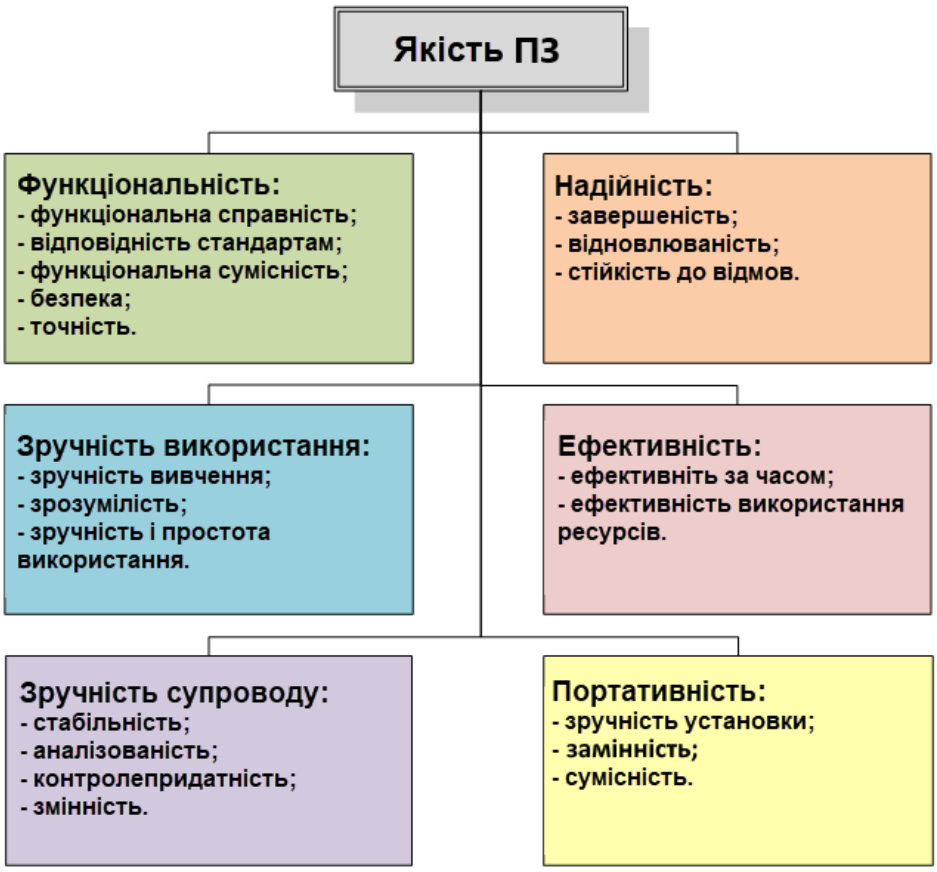


Рис. 1.7 Вимоги до якості програмного забезпечення

Передумови для додатків можна розділити на дві групи. Перший набір вимог складається з функціональних вимог, які пояснюють причину написання коду. Найчастіше вони представлені у вигляді користувацьких історій або варіантів використання. У цьому контексті архітектура не відіграє значної ролі. Функціональні вимоги можна виконати практично з будь-яким дизайном, навіть найнеефективнішим на даний момент.

Коли мова заходить про другу категорію потреб, які пов'язані з якістю обслуговування, архітектура одразу виходить на перший план. Це те, що зазвичай називають якісними характеристиками. Ці характеристики, які включають в себе масштабованість і надійність, визначаються протягом усього часу виконання. Крім того, вони включають опис аспектів розробки, до яких відносяться простота обслуговування, тестування і розгортання. Те, чи здатен додаток відповідати цим вимогам якості, визначається обраною архітектурою.

Загальний огляд багатьох архітектурних стилів.

Кожен архітектурний стиль - це сукупність дизайнерських рішень, які визначають конкретні архітектурні елементи та матеріали, що використовуються в будівництві. Програмне забезпечення можна розглядати через призму тієї ж концепції.

Сімейство систем, які принципово схожі за своєю структурною будовою, називається архітектурним стилем. Стиль, зокрема, визначає набір компонентів і з'єднань, які можуть бути використані в реалізації цього стилю, а також набір правил, які регулюють спосіб, у який ці компоненти і з'єднувачі можуть бути з'єднані.

На основі певного архітектурного стилю ви можете визначити архітектурне представлення вашого додатку. Це пов'язано з тим, що стиль дає обмежену палітру елементів (компонентів) і зв'язків (з'єднувачів). Насправді, нерідко використовується поєднання багатьох архітектурних стилів. Як ілюстрація, монолітна архітектура - це стиль, який організовує представлення реалізації як єдиного компонента, який може бути виконаний і розгорнутий. При використанні мікросервісної архітектури додаток організовується як набір сервісів, які лише слабко пов'язані між собою.

Багаторівнева архітектура - це відомий архітектурний стиль, який існує вже давно. Він досягається шляхом класифікації компонентів програми на кілька рівнів. Кожному рівню відповідає певна ієрархія завдань. Також існує обмеження на існування залежностей між рівнями. Рівень може залежати від рівня, який знаходиться безпосередньо під ним (згідно з принципом суворого розбиття) або від будь-якого рівня, який лежить нижче нього.

З іншого боку, трирівнева архітектура, яка є окремим випадком багаторівневої архітектури, може бути застосована до логічного представлення. Багаторівнева архітектура може бути застосована до будь-якого з представлень, які були згадані раніше. Класи додатків розділені на три рівні або шари, а саме: рівень представлення, який включає код, що реалізує інтерфейс користувача або зовнішні API; рівень бізнес-логіки, який включає бізнес-логіку; і рівень зберігання даних, який реалізує логіку взаємодії з базою даних.

Існує кілька ключових проблем, пов'язаних з багаторівневою архітектурою, незважаючи на те, що вона є чудовим прикладом архітектурного стилю. По-перше, вона не враховує той факт, що додаток, швидше за все, буде викликатися більш ніж однією системою.

Однією з характеристик єдиного рівня зберігання даних є те, що він не враховує можливість того, що додаток буде взаємодіяти з багатьма базами даних.

З теоретичної точки зору, той факт, що рівень бізнес-логіки залежить від рівня зберігання даних, унеможливлює тестування бізнес-логіки незалежно від бази даних.

Крім того, багаторівнева архітектура призводить до спотворення залежностей у ретельно розробленому додатку. Репозиторій інтерфейсів або сам інтерфейс часто надається бізнес-логікою. Ці інтерфейси визначають способи, які можна використовувати для доступу до даних. Класи DAO, які відповідають за реалізацію інтерфейсів репозиторію, визначаються рівнем зберігання даних. Іншими словами, залежності є протилежністю того, що насправді визначає багаторівневий дизайн.

1.2.4 Переваги використання мікросервісної архітектури

Існує багато переваг, пов'язаних з дизайном мікросервісів, які спрямовані на управління складністю системи. Існує ряд переваг платформи, які пропонують мікросервіси в порівнянні з монолітними системами, які є громіздкими.

Мікросервіси на рівні платформи мають низку переваг, однією з яких є масштабованість, що дозволяє кожному сервісу масштабуватися незалежно відповідно до вимог. Це усуває вимогу нарощувати всю систему в монолітних системах, які характеризуються тим, що навіть незначний компонент може обмежити

ефективність. Додатковою перевагою є незалежне розгортання, яке дозволяє оновлювати та доставляти окремі модулі незалежно від інших компонентів, що робить процес розгортання більш доцільним. Завдяки тому, що мікросервіси складаються з різноманітних технологій, можна вибрати інструментарій, який найбільше підходить для конкретної роботи. Це включає в себе використання декількох видів баз даних.

Коли мова йде про переваги платформ, важливим є модульність, оскільки вона дозволяє підтримувати логічний поділ системи на модулі, а також забезпечує фізичну ізоляцію для захисту від порушень меж контексту. Крім того, мікросервіси відрізняються тим, що вони прості для розуміння і підтримки, а також дозволяють підбирати апаратне забезпечення для кожного окремого сервісу.

У разі виникнення поломки легко локалізувати проблему до конкретного мікросервісу, що позбавляє від необхідності перезапускати всю систему. Це одна з найвагоміших переваг системи. У порівнянні з типовими трирівневими додатками, кількість часу, необхідного для запуску та розгортання мікросервісів, може бути значно зменшена.

Мікросервіси дозволяють безперервне розгортання та автономне масштабування кожного сервісу, що є ще одним важливим аспектом, який слід враховувати. Це гарантує, що необхідну кількість екземплярів сервісу можна розгорнути без значної кількості комунікацій з іншими розробниками. Як наслідок, мікросервіси пропонують більшу гнучкість у порівнянні з монолітними системами і спрощують налагодження безперервної інтеграції та безперервної доставки.

1.2.5 Недоліки використання мікросервісів

Незважаючи на те, що мікросервісна архітектура має багато потенційних переваг, вона не позбавлена недоліків і труднощів, особливо з точки зору інтероперабельності та інтеграції сервісів. Враховуючи, що термін "мікросервісна архітектура" існує лише протягом дуже короткого періоду часу, однією з найбільш значущих проблем є відсутність стандартизації та загальних специфікацій.

Розробка мікросервісів пов'язана з низкою складних проблем, включаючи обробку запитів і розподіл цих запитів між сервісами в розподіленій системі. При взаємодії через виклик методів з використанням RPC також виникають проблеми, такі як врахування можливості затримок при віддалених викликах та управління помилками, що виникають на клієнті та в мережі.

Існує ряд проблем, які пов'язані з розробкою мікросервісних додатків. Однією з таких проблем є управління даними, що призводить до того, що організація транзакцій у розподіленій системі стає критично складною. Коли мова йде про величезні NoSQL бази даних і брокери повідомлень, використання розподілених транзакцій часто не є можливим. Особливо це стосується кількох ситуацій.

Побудова мікросервісів також може призвести до виникнення таких проблем, як збільшення використання ресурсів внаслідок вимоги, що кожен сервіс повинен мати власний контейнер і програмне середовище. Коли справа доходить до прийняття стандартних протоколів для обміну даними між сервісами, підвищена завантаженість мережі стає актуальною.

Процес тестування та моніторингу ускладнюється порівняно з монолітними програмами, проте нові програмні рішення можуть допомогти у вирішенні цих проблемних питань. Іншою важливою проблемою є збільшення складності рефакторингу, що особливо помітно при перенесенні логіки з одного сервісу на інший.

Перш ніж зупинитися на виборі, необхідно провести аналіз, оскільки вибір архітектури мікросервісу, зроблений без ретельного обмірковування, може призвести до хаотичного дизайну і краху всього проекту.

1.3 Вплив Node.js на дизайн мікросервісів

У світі сучасних технологій мікросервісна архітектура є надзвичайно актуальним підходом до розробки програмного забезпечення. У цьому дослідженні особлива увага приділяється ролі Node.js у створенні мікросервісів, яка визначається рядом важливих елементів.

1.3.1 Робота Node.js у створенні мікросервісів

Node.js, як виконавче середовище, засноване на подієвому та асинхронному програмуванні, має вирішальне значення для створення архітектури мікросервісів. Його функції підвищують продуктивність і швидкість розробки, оптимізуючи використання ресурсів системи. У Node.js є механізми подій і асинхронності, які дозволяють взаємодіяти з різними мікросервісами, що дозволяє створювати високомасштабовані, легко змінювані та динамічні системи. Досліджується, як використання Node.js впливає на взаємодію та структуру мікросервісів, надаючи їм гнучкість і простоту масштабування.

1.3.2 Ефективність і універсальність Node.js

Дослідження проводяться щодо ефективності та гнучкості Node.js в контексті побудови мікросервісів, оскільки він дуже поширений у розробці серверних додатків. JavaScript є основною мовою програмування для серверної та клієнтської частини, що полегшує розробку та підтримку коду. Це важлива частина успішної імплементації мікросервісної архітектури. Асинхронність Node.js дозволяє обробляти велику кількість запитів і забезпечує високу продуктивність мікросервісів. Дослідження також розглядає сценарії використання Node.js, які дозволяють максимізувати гнучкість і продуктивність при створенні та управлінні мікросервісами.

2 РОЗРОБКА АРХІТЕКТУРИ СИСТЕМИ

2.1 Планування архітектури програмного забезпечення

Для цього дослідження було обрано клієнт-серверну архітектуру, яка зазвичай поділяється на два різних типи: дворівневу та трирівневу. Тому, система буде розроблена з використанням трирівневої архітектури, яка включатиме наступні компоненти:

1. Клієнтський рівень – це частина програми, спеціально створена для взаємодії з користувачем через інтерфейс користувача (UI).

2. Сервер – охоплює сегмент програми, в якому міститься вся бізнес-логіка додатку. Клієнтський рівень має доступ до функцій сервера через точно визначений інтерфейс.

3. Рівень даних – відноситься до рівня взаємодії з базою даних (БД). Цей компонент системи надає серверу можливість підключення до бази даних.

Вибір цього методу гарантує наступні переваги:

- Підвищена безпека системи;

- Оптимальна масштабованість і гнучкість;

- Виняткова продуктивність.

На рисунку 2.1 зображено схематичне зображення клієнт-серверної архітектури з трьома з'єднаннями.

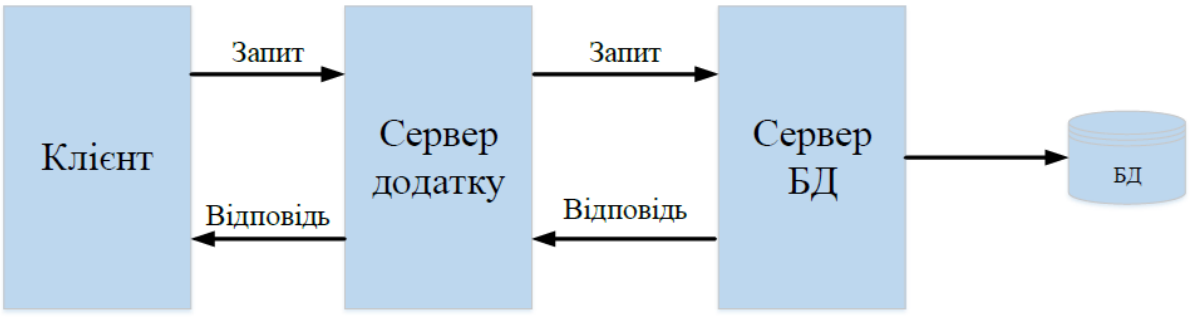


Рис. 2.1 Загальна схема трирівневої клієнт-серверної архітектури

Архітектура клієнт-сервер - це система, яка розподіляє робоче навантаження між серверами, що надають послуги, і клієнтами, які отримують послуги. Ці дві сторони спілкуються між собою за допомогою мережевих протоколів, як правило, HTTP. Архітектура клієнт-сервер відрізняється відсутністю дублювання коду серверної програми в клієнтських програмах. Крім того, сервер виконує всі обчислення і базову обробку даних, що призводить до зниження вимог до комп'ютерів, на яких працюють клієнтські програми. Крім того, сервер, як правило, має вищі заходи безпеки порівняно з клієнтськими програмами. На цьому рівні системи простіше встановити правила авторизації, щоб гарантувати, що тільки клієнти з необхідними привілеями матимуть доступ до даних.

Спрощення архітектури до клієнт-серверної моделі лежить в основі архітектурного стилю REST для розробки API. В результаті цього обмеження система набуває таких цінних якостей, як ефективність, простота, універсальність, розширюваність, налаштованість, переносимість і надійність.

Фундаментальним принципом, який лежить в основі цієї архітектури, є розділення вимог. Відокремлення вимог клієнтського інтерфейсу від вимог серверної частини, яка відповідає за зберігання даних, дозволяє зробити клієнтський програмний код більш портативним і легко адаптувати його до різних платформ. Водночас, оптимізація серверної частини значно підвищує масштабованість системи в цілому. Це дозволяє окремим компонентам розвиватися незалежно один від одного.

Система користувача побудована як трирівнева клієнт-серверна архітектура. Ця архітектура включає рівень представлення даних, який відповідає за інтерфейс користувача. Вона також включає рівень компонентів додатків, який діє як проміжне програмне забезпечення між сервером додатків і рівнем бази даних. Нарешті, є рівень управління ресурсами, який відповідає за доступ до бази даних.

Трирівневу структуру клієнт-серверу можна розширити до багаторівневої архітектури, додавши більше серверів. Кожен сервер буде пропонувати власні

послуги та використовувати послуги інших серверів на різних рівнях.

2.2 Вибір шаблону проектування системи

Система була розроблена з використанням архітектурного патерну MVC.

MVC – це фреймворк, який розділяє додаток на три окремі та автономні компоненти: рівень даних, користувацький інтерфейс та логіку додатку. Модифікуючи кожен з трьох компонентів незалежно, система отримує переваги з точки зору портативності та масштабованості.

Тому, розглянемо компоненти MVC більш детально:

- Модель – відповідає за надання даних для обробки та представлення, а також реагує на команди контролера, змінюючи власний стан.

- Представлення – відповідає за відображення даних, наданих моделлю користувачеві, і реагує на зміни в моделі.

- Контролер – реагує на дії користувача та інформує модель про необхідні зміни.

Основна концепція, що лежить в основі цієї структури, полягає у відокремленні бізнес-логіки (Model) від її візуального представлення (View). Відокремлення коду дозволяє застосовувати поняття повторного використання коду. Більше того, це виявляється дуже корисним у сценаріях, де користувачеві потрібен одночасний доступ до ідентичного матеріалу, але з різних налаштувань або точок зору. Крім того, до реалізації моделі можна приєднати багато представлень, не змінюючи її. Наприклад, дані можуть одночасно відображатися у таких форматах, як текст, графік, таблиця або гістограма. Модифікація моделі не вплине на саму модель. Крім того, можна змінити обробку дій користувача, таких як введення даних і жести миші, не вносячи жодних змін у реалізацію подання. Для цього достатньо використати новий контролер або модифікувати поточний. Бізнес-логіка та візуальне представлення програми можуть бути побудовані автономно одне від одного. Розробники логіки програми можуть не знати, яке саме представлення буде використовуватися.

Рисунок 2.2 ілюструє макет архітектурного патерну MVC.

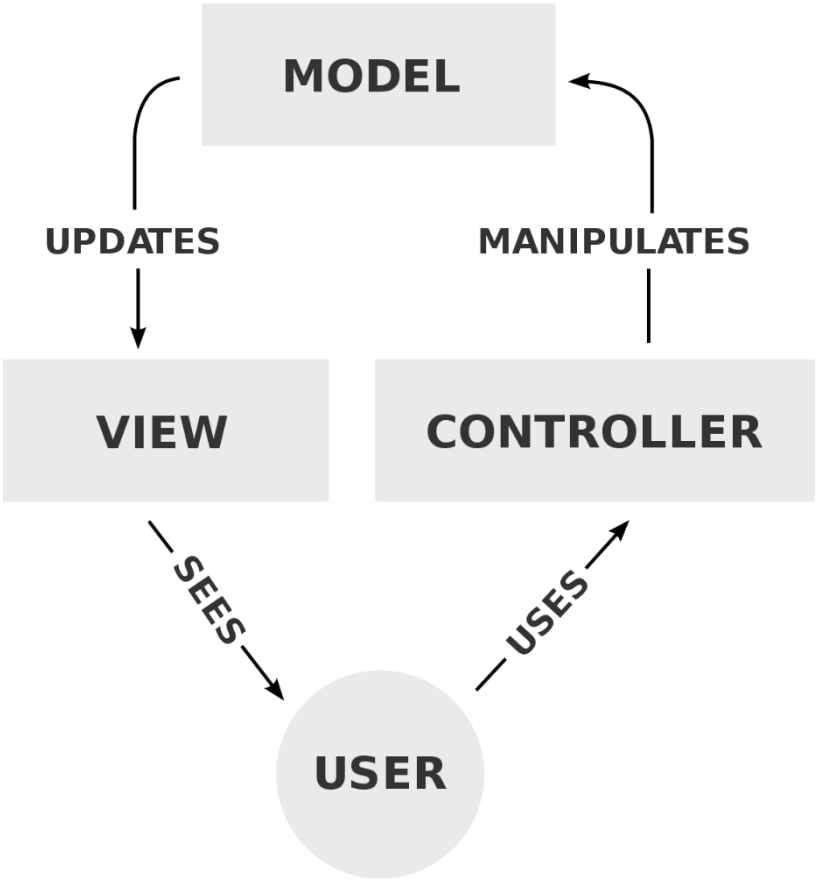


Рис. 2.2 Загальна схема MVC

Модель охоплює як дані, так і процедури, необхідні для маніпулювання ними, включаючи запити до бази даних і перевірку даних. Модель є агностичною до представлення; їй бракує можливості візуального відображення цих даних. Крім того, вона працює автономно від контролера і не має спільних точок взаємодії з користувачем. Це полегшує пошук і організацію даних.

Представлення відповідає за отримання необхідних даних з моделі та передачу їх користувачеві. Представлення не може впливати на стан моделі через прямий зв'язок. Крім того, подання не обробляє дані, введені користувачем.

Контролер полегшує обмін інформацією між користувачем і системою. Він визначає, які процедури слід виконати у відповідь на конкретну дію користувача. Для того, щоб виконати необхідну дію, він використовує модель і представлення.

Модель-Представлення-Контролер (MVC) є широко застосовуваним архітектурним дизайном і не має жорсткої реалізації. Не існує загальноприйнятого місця для бізнес-логіки додатку. Вона може знаходитися або в контролері, або в моделі. Деякі фреймворки однозначно визначають точне місце розташування логіки програми, тоді як інші не мають таких чітких вказівок. Місце для валідації користувацького вводу не є чітко визначеним. Якщо валідація є простою, її можна розмістити безпосередньо у вікні. Однак, як правило, вона залишається в межах контролера або моделі.

Схема Model-View-Controller може бути реалізована з використанням різних шаблонів проектування, в залежності від складності архітектурного рішення.

- "спостерігач" – це той, хто дивиться або спостерігає за чимось.

- "стратегія" – це план або підхід, розроблений для досягнення певної мети.

- "верстальник" – це той, хто створює макет або дизайн чого-небудь, наприклад, веб-сайту або журналу.

Прикладом патерну "верстальник" є розділення представлення і моделі шляхом побудови протоколу взаємодії між ними, який використовує "апарат подій". Термін "апарат подій" означає ідентифікацію певних умов, що виникають під час виконання програми, і подальше розповсюдження повідомлень зареєстрованим особам, відомим як підписники. Щоразу, коли відбувається модифікація внутрішніх даних моделі, запускається подія, яка сповіщає про це всі підписані на неї подання. Отримавши сповіщення, подання оперативно змінює свій статус.

Використовуючи шаблон "стратегія", подання автономно визначає контролер, відповідальний за обробку реакції користувача та полегшення комунікації з моделлю. Цей шаблон також має модифікацію, відому як "команда".

Основна мета шаблону "верстальник" – полегшити послідовну комунікацію з підлеглими елементами в межах складної ієрархії.

MVC-модель може існувати в активному або пасивному стані. У цьому випадку модель програми складається виключно з набору методів, які

використовуються для отримання даних. Контролер, з іншого боку, охоплює всю прикладну логіку програми.

В об'єктно-орієнтованому програмуванні переважає використання активної моделі MVC, де модель охоплює бізнес-логіку програми разом з методами доступу до СУБД. Крім того, моделі можуть містити в собі інші моделі. Водночас, контролери мають "тонку" відповідальність, яка включає в себе отримання запиту користувача, аналіз запиту та прийняття рішення про наступну дію системи на основі результатів аналізу (наприклад, перенаправлення запиту до інших елементів моделі).

Контролер функціонує виключно як посередник між різними компонентами інформаційної системи.

3 АНАЛІЗ ТА ВИБІР ТЕХНОЛОГІЙ РОЗРОБКИ

3.1 Node.js та JavaScript як мова програмування на стороні сервера

JavaScript – це дуже поширена мова програмування, яка щороку зазнає постійного зростання. JavaScript, спочатку була призначена для покращення інтерактивності веб-сайтів на стороні клієнта, зараз знайшла застосування як мова на стороні сервера. Згідно з дослідженням 2022 року, проведеним сайтом для розробки програмного забезпечення GitHub, JavaScript посідає перше місце за популярністю серед його учасників (рис. 3.1).

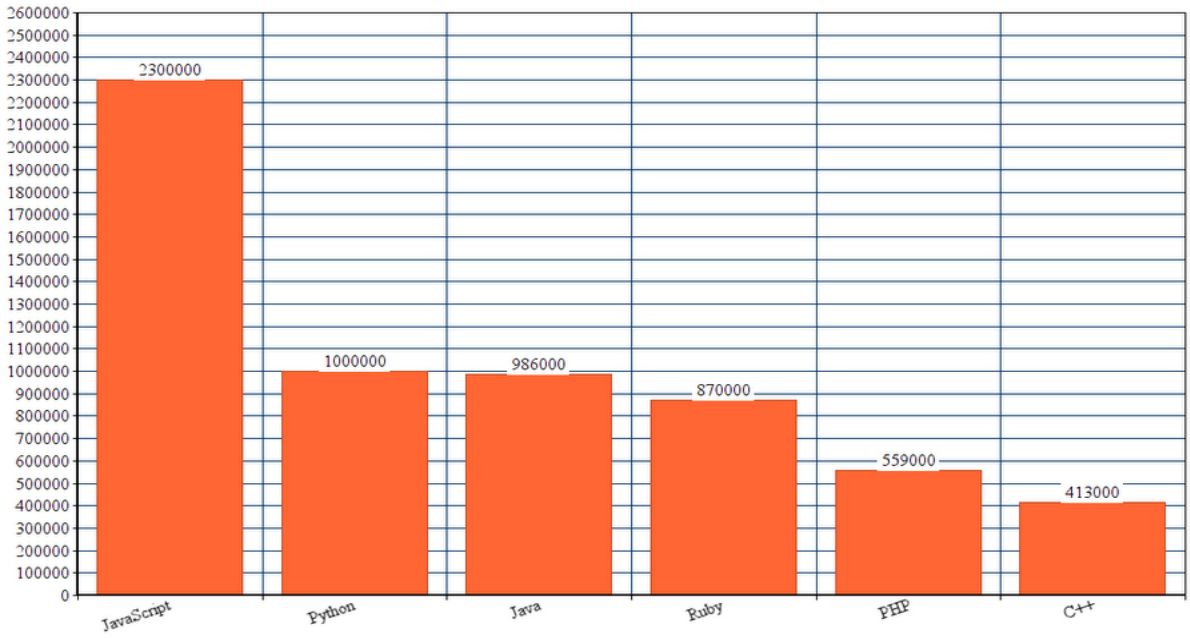


Рис. 3.1 Рейтинг мов програмування у 2022 році за версією GitHub

JavaScript підтримує різні парадигми програмування, такі як об'єктно-орієнтоване, функціональне та імперативне. Первинні структурні характеристики охоплюють наступні елементи:

- Динамічна типізація – це особливість мови програмування, коли тип змінної визначається під час виконання програми, а не її явного оголошення. При такому підході змінна пов'язується з типом не під час її визначення, а коли їй

присвоюється значення. Така ситуація має як переваги, так і недоліки. Зокрема, це спрощує процес написання скриптів, хоча водночас такий підхід часто призводить до виникнення незрозумілих помилок у коді, які складно виявити. Крім того, швидкість роботи програми може бути скомпрометована через зменшення часу обробки, що виділяється на динамічну перевірку типів змінних.

- Слабка типізація – дозволяє безперешкодно конвертувати змінні між різними типами даних. Як і динамічна типізація, ця функція має як переваги, так і недоліки. Цей метод значно підвищує адаптивність і універсальність коду, оскільки дозволяє автоматично перетворювати типи аргументів при виклику певної функції. З іншого боку, це може призвести до помилок і непередбачуваної поведінки програми, яку дуже складно відстежити.
- Керування пам'яттю, яке виконується автоматично – передбачає наявність окремої процедури, відомої як "збирач сміття", яка регулярно звільняє пам'ять, видаляючи об'єкти, що більше не потрібні програмі. У випадку ручного керування пам'яттю, програміст відповідає за активний нагляд за видаленням використаних об'єктів. Ці проблеми можуть призвести до суттєвих порушень функціональності програми, створення невирішених посилань та деградації пам'яті, що в кінцевому підсумку призведе до збоїв у роботі та значного зниження продуктивності.
- Прототипне програмування – це різновид об'єктно-орієнтованого програмування, де відсутнє поняття класу, а успадкування досягається шляхом копіювання прототипу, який є вже існуючим екземпляром об'єкта. Ця техніка дозволяє користувачеві обійти надмірну увагу до класифікації класів та їх взаємозв'язків, а натомість зосередитися на поведінці обмеженої кількості прикладів, які слугують фундаментальними об'єктами і згодом використовуються для генерації інших об'єктів.
- Підтримка першокласних об'єктів. Першокласні об'єкти - це сутності, які мають можливість повертатися, передаватися як параметри функцій, присвоюватися змінним тощо. Цей атрибут виникає завдяки використанню методології прототипування.

Node.js дозволяє використовувати JavaScript як серверну мову, функціонуючи як програмна платформа, що працює на движку V8, який перетворює JavaScript в машинний код. Ця платформа перетворює JavaScript, який спочатку був мовою програмування з вузькою спрямованістю, на універсальну мову, яку можна використовувати для широкого кола цілей. Node.js надає власний API пристроїв вводу/виводу, що дозволяє JavaScript взаємодіяти з цими пристроями та встановлювати зв'язки з бібліотеками, розробленими на різних мовах програмування.

Варіант використання Node.js для розробки API має багато помітних переваг. Важливим аспектом є реалізація неблокуючої моделі вводу та виводу. Рисунок 3.2 ілюструє принцип цієї архітектури.

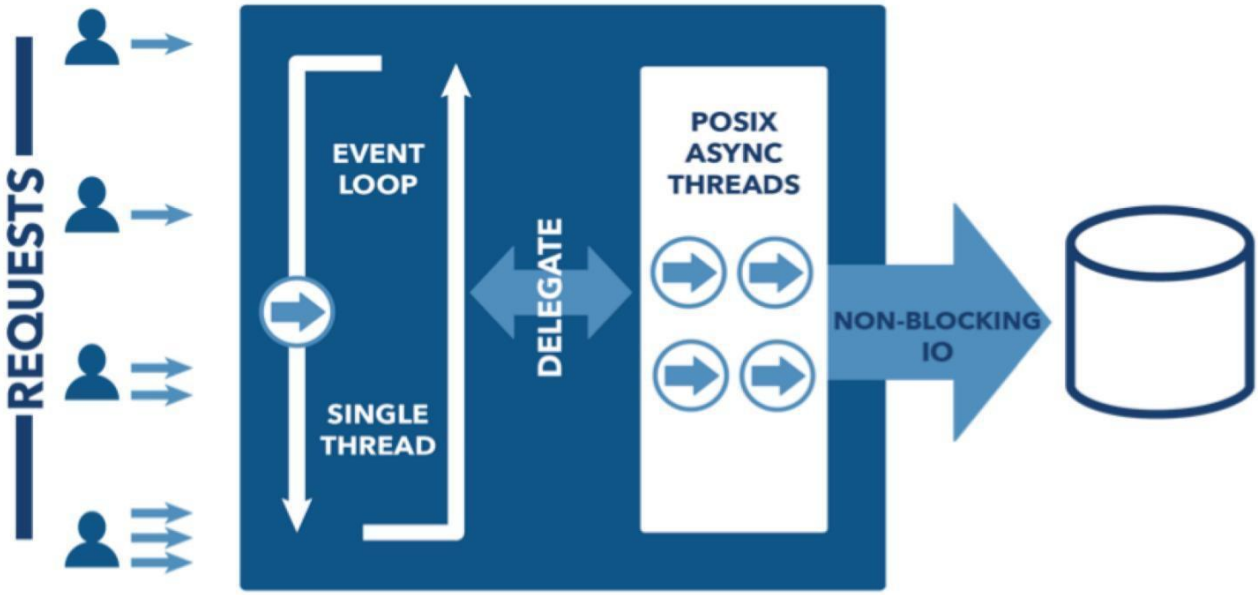


Рис. 3.2 Загальний принцип роботи моделі неблокуючого вводу/виводу

Ця архітектура дозволяє одночасно обробляти значну кількість з'єднань, що призводить до значного покращення масштабованості системи. Це контрастує з поширеною архітектурою, яка використовує паралельні потоки операційної системи. При використанні Node.js немає необхідності турбуватися про потенційне блокування процесів, оскільки функції не взаємодіють безпосередньо з пристроями вводу/виводу.

Тому, розглянемо асинхронний ввід та вивід більш детально, як показано на

рисунку 3.3. Node.js використовує одиночний потік, а цикл обробки подій бере на себе завдання керувати всіма асинхронними операціями вводу-виводу. Поведінка програми контролюється асинхронними викликами процедур зворотного виклику. Така стратегія дозволяє обійти проблеми, пов'язані зі створенням окремих потоків та їх подальшим блокуванням. Крім того, вона забезпечує суттєве покращення часу відгуку сервера та масштабованості системи.

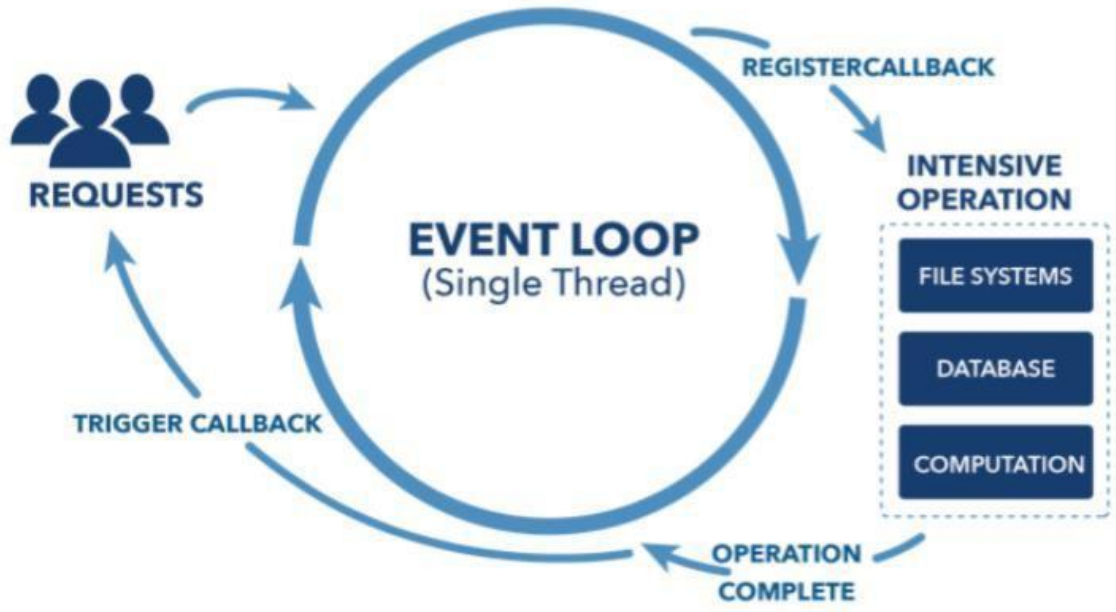


Рис. 3.3 Організація асинхронного вводу/виводу у Node.js

Використання JavaScript як мови на стороні сервера дозволяє значно підвищити швидкість виконання коду. Крім того, завдяки здатності движка V8 миттєво перетворювати JavaScript-код на машинний, Node.js виділяється як найбільш високошвидкісна платформа з усіх доступних на даний момент.

Завдяки своїй подієво-керованій архітектурі Node.js є оптимальним варіантом для побудови додатків реального часу. Завдяки тому, що і серверна, і клієнтська частини кодуються на JavaScript, процедура синхронізації між ними значно прискорюється, спрощується і стає зручнішою.

Node.js має не лише власні, притаманні лише їй функції, але й включає в себе менеджер пакетів node (npm). Основна концепція полягає в тому, щоб розробити компактний програмний модуль, який ефективно та швидко вирішує певну задачу.

Це дозволяє створювати великі проекти, збираючи ці окремі будівельні компоненти. Крім того, він дотримується принципів прозорості коду та повторного використання коду.

Найяскравішими є npm-модулі, які мають найвищий рівень популярності:

- Фреймворк Express – це надійний інструмент для веб-розробки. Він є переважаючою нормою для більшості сучасних Node.js додатків.

- Connect – це гнучкий фреймворк, який функціонує як HTTP-сервер у поєднанні з Node.js. Він пропонує колекцію ефективних плагінів як зв'язуюче програмне забезпечення.

- Socket.io та Socket.js – це серверні бібліотеки, що використовуються для обміну даними в режимі реального часу.

Node.js робить особливий акцент на співпраці з базами даних No-SQL, такими як Mongo DB. Ця стратегія дозволяє підвищити масштабованість і швидкість роботи продукту.

Node.js повністю використовує архітектуру програмування MVC (model-view-controller), яка широко визнана як загальноприйнятий стандарт.

Платформа, названа "node", підкреслює можливість побудови систем з використанням численних невеликих розподілених обчислювальних вузлів, які можуть спілкуватися та обмінюватися даними один з одним. Сервер Node.js завжди працюватиме на оптимальному рівні потужності, ніколи не перевищуючи межи необхідної. Основна суть архітектури Node полягає в її мінімалістичному характері. При цьому серверна частина додатку може бути розширена або зменшена в розмірах в залежності від вимог проекту.

Модульність, притаманна Node.js, дозволяє розробляти компактні додатки без необхідності підтримувати розгалужену інфраструктуру, значна частина якої зрештою залишиться невикористаною. Створюючи додаток на Node.js, розробник може самостійно визначати основні компоненти і, за необхідності, розширювати обсяг проекту.

3.2 Express.js фреймворк

Express.js – це фреймворк, призначений для створення веб-додатків на платформі Node.js. Він призначений для створення онлайн-додатків, в основному з акцентом на REST-API.

Express.js дозволяє швидко і без зайвих зусиль розробляти серверні веб-додатки. Основними атрибутами цього є мінімалізм, гнучкість, масштабованість та простота.

Ключовими особливостями Express.js є наступні:

- Комплексна підтримка різних функцій маршрутизації, включаючи саму маршрутизацію та окремі обробники для запитів post, get, put тощо.

- Автоматичне вилучення змінних з URL-адреси за допомогою обробки підстановочних знаків.

- Можливість роботи зі статичними файлами.

- Безшовна інтеграція з численними плагінами шаблонів.

- Ефективне кешування переглядів.

- Підтримка маршрутизації з урахуванням регістру.

- Впровадження механізму e-Tag.

- Миттєва швидкість розробки.

- Висока гнучкість фреймворку.

- Різноманітні методи побудови відповіді на запит.

Фреймворк Express.js пропонує простий, але надійний підхід до побудови маршрутів. На рисунку 3.4 показано приклад простого маршруту. Функція зворотного виклику активується при отриманні запиту для цього маршруту. Крім того, Express.js дозволяє створювати динамічні маршрути, які здатні динамічно адаптувати свої компоненти.

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

var app = express();

app.get('/', function(req, res) {
    res.setHeader('Content-Type', 'text/plain');
    res.end('You\'re in reception');
});

app.listen(8080);
```

Рис. 3.4 Приклад простого маршруту в Express.js

Включення методу e-Tag в Express.js значно підвищує продуктивність програми і спрощує операції кешування. E-Tag, також відомий як тег сутності, є компонентом протоколу HTTP. Він слугує механізмом, за допомогою якого HTTP проводить перевірку кешу і дозволяє клієнту робити запит на основі певних умов. Це значно зменшує обсяг даних, що передаються, і підвищує ефективність кешу, оскільки веб-серверу не потрібно надавати повну відповідь, якщо вміст ресурсу залишається незмінним.

Тому, розглянемо технологію e-Tag більш детально.

E-Tag – це унікальний ідентифікатор, який веб-сервер присвоює певній версії ресурсу, що знаходиться за певною URL-адресою. Коли вміст ресурсу за цією URL-адресою змінюється, присвоюється новий e-Tag. Використання е-тегів у такий спосіб нагадує використання відбитків пальців, які можна швидко порівняти, щоб визначити, чи є дві ітерації певного ресурсу однаковими.

Типові методи генерації е-мітки передбачають використання хеш-функції, стійкої до зіткнень, для обробки вмісту певного ресурсу, обчислення хеш-значення на основі часу останньої модифікації ресурсу або просто використання номера версії. Для того, щоб запобігти отриманню застарілих даних кешу, методи повинні гарантувати, що кожна електронна мітка має окреме значення. Невід'ємною перевагою Express.js є використання функцій проміжного програмного забезпечення для проміжного виконання.

Деякі поширені приклади проміжного програмного забезпечення включають:

- Стиснення – дозволяє стискати веб-сторінки у формат gzip для швидшої

передачі на сервер.

- Парсер файлів cookie – полегшує роботу з файлами cookie.

- Сесійний файл cookie – генерує інформацію про сесію, поки користувач залишається на сайті.

- Статичне обслуговування – витягує статичні файли, що зберігаються в каталозі сервера.

- Захист від CSRF (Cross-Site Request Forgery) – захищає інформацію на сервері від CSRF-атак.

Express.js полегшує створення веб-додатків на основі моделей, підтримуючи архітектуру "MVC". У цьому підході до розробки програмного забезпечення основна увага приділяється створенню моделей, які є абстрактним представленням програмного забезпечення. Ці моделі навмисно приховують певні деталі, щоб забезпечити спрощений опис решти функцій.

Модель кодує дані у спосіб, зрозумілий людині, і може бути проаналізована різними технологіями. Ця стратегія передбачає початкову розробку моделі додатку, яка не прив'язана до жодної конкретної технології. Потім ця модель перетворюється на модель, специфічну для платформи, на якій вона буде реалізована. Нарешті, специфічна для платформи модель перекладається у вихідний код.

3.3 База даних Mongo DB та модуль Mongoose.js

Mongo DB – використовує нову методологію побудови баз даних, яка усуває потребу в таблицях, схемах, SQL-запитах, зовнішніх ключах і різних інших характеристиках, які часто зустрічаються в об'єктно-реляційних базах даних. Mongo DB класифікується як база даних NoSQL. На відміну від реляційних баз даних, Mongo DB надає модель даних, орієнтовану на документи, що призводить до значного покращення швидкості та масштабованості.

База даних Mongo DB контролює колекції JSON-подібних документів, які

зберігаються у двійковому форматі за допомогою протоколу BSON (Binary JSON). Рисунок 3.5 ілюструє архітектуру Mongo DB.

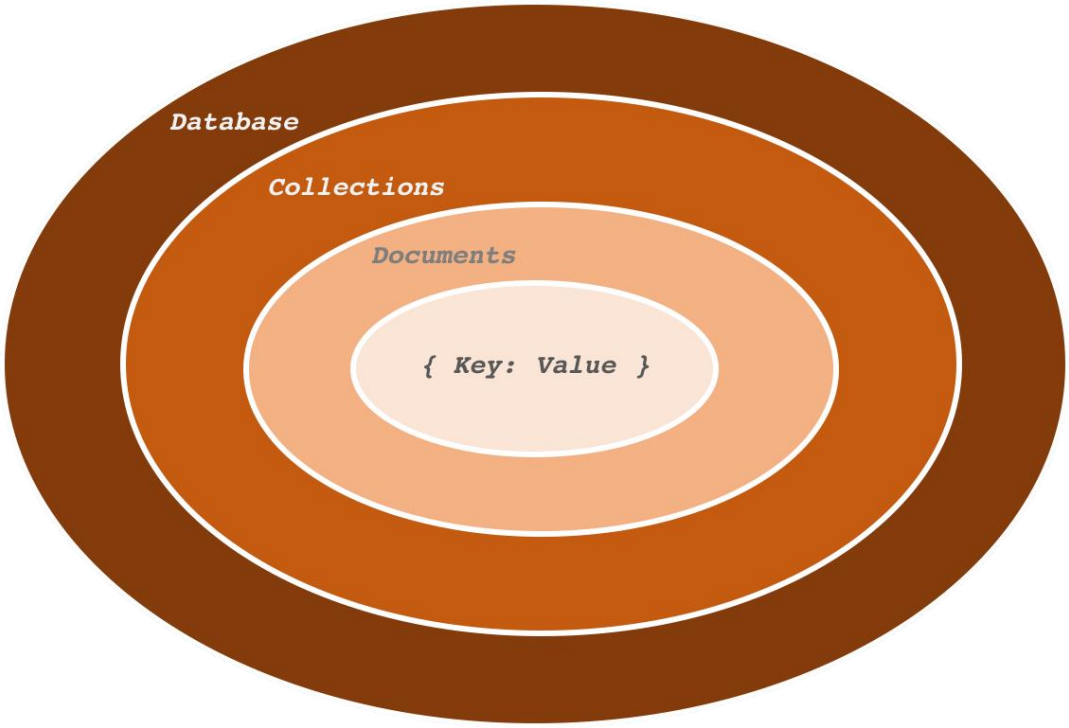


Рис. 3.5 Структура бази даних Mongo DB

BSON дозволяє прискорити маніпуляції з даними, сприяючи швидшому пошуку та обробці. Варто зазначити, що BSON займає більший обсяг пам'яті в порівнянні з JSON. Однак цей недолік повністю компенсується вищою продуктивністю обробки.

На відміну від таблиць у реляційних базах даних, Mongo DB використовує колекції. Реляційні бази даних обмежені вимогою, що таблиці можуть містити лише елементи з фіксованою та однорідною структурою, тоді як колекція може вміщувати об'єкти з різноманітною структурою та різними наборами характеристик.

Реляційні бази даних призначені для зберігання рядків, але Mongo DB спеціально розроблена для зберігання документів. Це дозволяє зберігати дані зі складною структурою. Документ можна уявити як сховище пар ключ-значення. Ключ - це базовий ідентифікатор, який пов'язаний з певним набором даних.

Кожен ключ має певне значення. На відміну від реляційних баз даних,

Mongo DB не вимагає, щоб поля мали значення. Якщо ключ не має значення, він просто пропускається в документі.

Mongo DB дозволяє виконувати спеціальні запити, які можуть отримувати певні поля документа і виконувати спеціальні методи JavaScript. Крім того, система надає допомогу в проведенні пошуку за регулярними виразами, і ви можете налаштувати запит для отримання випадкової вибірки значень.

Mongo DB дозволяє маніпулювати набором реплік. Набір реплік складається щонайменше з двох копій даних. Основна репліка обробляє всі операції читання і запису, тоді як допоміжні репліки гарантують, що копії даних залишаються актуальними. Якщо основна репліка вийде з ладу, інформація буде збережена. Крім того, допоміжні репліки можуть також функціонувати як джерело для операцій читання з сервера.

Mongo DB сумісна з надійним інструментом агрегації даних MapReduce. MapReduce – це фреймворк розподілених обчислень, здатний виконувати одночасні обчислення над великими обсягами даних. MapReduce складається з двох процесів. На етапі відображення вхідні дані проходять попередню обробку. Для цього первинний вузол отримує вхідні дані, розділяє їх на сегменти і передає їх підлеглим вузлам для попередньої обробки. На етапі редукції попередньо оброблені дані стискаються або об'єднуються. Головний вузол збирає відповіді від робочих вузлів і на їх основі генерує результат. Рисунок 3.6 демонструє приклад агрегації з використанням MapReduce.

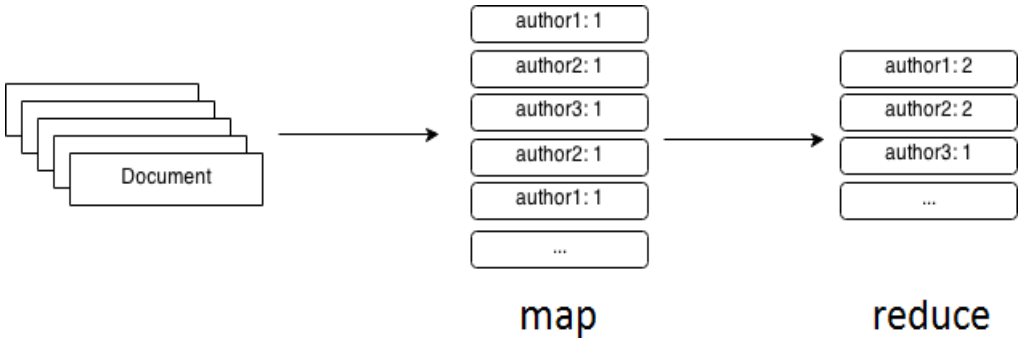


Рис. 3.6 Приклад використання парадигми MapReduce

Архітектура Mongo DB має дві відмінні риси:

1. Поняття "транзакція" не існує. Атомарність забезпечується лише на рівні документів. Отже, неможливо виконати часткове оновлення документа.

2. Поняття "ізоляція" не існує. Одночасні клієнти мають можливість змінювати дані, які були прочитані іншим клієнтом.

Система Mongo DB може складатися з декількох баз даних, розміщених на одному фізичному сервері. Функція Mongo DB дозволяє розподіляти багато баз даних на декількох фізичних серверах, полегшуючи безперебійну передачу даних і забезпечуючи їхню цілісність.

Mongo DB добре підходить для співпраці з Node.js. Мову JavaScript можна безпосередньо використовувати у запитах та агрегатних функціях і передавати до бази даних для виконання. Завдяки своїй архітектурі, Mongo DB пропонує широкі можливості для масштабування, швидкої продуктивності та доступності даних.

Mongoose.js – це спеціалізований модуль в Node.js, який забезпечує безперешкодну взаємодію з Mongo DB. Mongoose – це бібліотека ODM (Object Document Mapper) з унікальними можливостями. Вона дозволяє визначати об'єкти за допомогою строго типізованої схеми, яка відповідає документу Mongo DB.

Mongoose пропонує різноманітні можливості для створення та маніпулювання схемами. Наразі Mongoose підтримує вісім типів схем, які охоплюють рядок, число, дату, буфер, булевий, змішаний, об'єктний та масив.

Кожен вид має потенціал для виникнення та має наступні можливості:

- Встановити значення за замовчуванням.

- Визначити власну функцію перевірки даних.

- Вказати обов'язкове поле.

- Вказати get-функцію для модифікації даних перед тим, як вони будуть повернуті як об'єкт.

- Вказати get-функцію для модифікації даних перед збереженням їх до бази даних.

- Визначити регулярний вираз, щоб обмежити допустимі варіації даних під час процесу валідації.

- Визначити список, щоб задати набір допустимих рядків.

Змішаний тип даних дозволяє зберігати дані будь-якого типу, а отже, надає розробнику гнучкість. Однак, важливо проявляти стриманість у його використанні, оскільки перевірка цих даних і моніторинг їхніх характеристик, що змінюються, є обмеженими.

Mongoose має безліч функцій для запиту даних конкретної моделі.

На рисунку 3.7 показано ілюстративний фрагмент коду для проведення пошуку даних.

```
Book.find({
    title: /mvc/i
}).exec(function(err, books) {
    if (err) throw err;

    console.log(books);
});
```

Рис. 3.7 Приклад пошуку даних у Mongoose

3.4 Бібліотека Chai та фреймворк Mocha для тестування API

Щоб переконатися, що програма функціонує за призначенням, для перевірки використовується тестове програмне забезпечення. Рекомендується використовувати тестування не тільки після завершення роботи над продуктом, але й під час його розробки, щоб запобігти виникненню проблем, які можуть призвести до необхідності внесення змін у весь дизайн системи. Тестуючи нову функціональність, можна оперативно переконатися у відсутності конфліктів з раніше написаним кодом.

Mocha.js – дуже потужний і широко використовуваний інструмент. Mocha – це фреймворк JavaScript, призначений для тестування додатків. Mocha є компонентом управління пакетами npm в рамках фреймворку Node.js.

Первинні характеристики мокко зумовлюють наступні:

- Безшовна інтеграція асинхронного тестування, включаючи використання механізму "обіцянки";

- Забезпечення підтримки таймаутів асинхронного виконання;

- Гнучкість у використанні будь-якої бібліотеки тверджень.

Часто використовуються наступні бібліотеки тверджень:

- expect.js;

- should.js;

- better-assert;

- chai.

Mocha має можливість надавати звіти в декількох форматах, таких як Nyan, Landing Strip, Progress, Dot Matrix, JSON Stream, Spec, List, TAP, JSON, JSONCov, Min, HTMLCov, Doc, XUnit, Markdown, TeamCity та HTML.

Рисунок 3.8 ілюструє простий приклад тесту " Mocha ".

```
var assert = require("assert")
     describe('#indexOf()', function(){
         it('should return -1 when the value is not present', function()
             assert.equal(-1, [1,2,3].indexOf(5));
             assert.equal(-1, [1,2,3].indexOf(0));
}) })
```

Рис. 3.8 Приклад простого синхронного тесту Mocha

Коли справа доходить до тестування, Chai – це бібліотека BDD/TDD-тверджень, яку можна просто використовувати в поєднанні з будь-яким JavaScript-фреймворком без будь-яких труднощів. Для цілей дослідження буде використовуватися фреймворк Mocha.

Оскільки Chai пропонує кілька різних інтерфейсів, розробники можуть вибрати той, який найкраще відповідає їхнім потребам. Ланцюговий стиль BDD заохочує експресивну мову та стиль, який легко зрозуміти, тоді як стиль TDD, що стверджує, є більш синонімічним традиційному написанню.

У фундаментальній граматиці Mocha тести починаються з describe (), а сам тест знаходиться в it (), який розташований в центрі describe (). У ситуаціях, коли є значна кількість блоків, є можливість згрупувати їх разом, вклавши один в одного.

Концепція, яка лежить в основі асинхронних тестів Mocha, полягає в тому,

що ми передаємо додатковий параметр done (), який є функцією зворотного виклику, що виконується в разі успішного завершення тесту. Кожен окремий хук, такий як before, after і так далі, також може підтримувати асинхронну функціональність.

На рисунку 3.9 показано приклад асинхронного тесту.

```
describe('User#save()', function(){
    it('should save without error', function(done){
         var user = new User('Luna');
         user.save(function(){
            done();
         });
    })
})
```

Рис. 3.9 Приклад асинхронного тесту Mocha

У цьому прикладі відображено використання асинхронних методів тестування, які дозволяють ефективно та гнучко перевіряти асинхронний код. Такий підхід є важливим у випадках, коли операції виконуються паралельно чи мають затримки у виконанні, а тестування повинно коректно взаємодіяти з асинхронними аспектами програмного забезпечення.

4 ПРОЄКТУВАННЯ ФУНКЦІОНАЛУ ТА МОДУЛІВ СИСТЕМИ

4.1 Проектування функціоналу системи

Враховуючи, що дослідження передбачає створення REST-API, необхідно забезпечити відповідність функціоналу системи всім принципам REST-архітектури.

Реалізація системи повинна відповідати архітектурі клієнт-сервер, що складається з трьох різних рівнів. Клієнт повинен бути захищений від якомога більшої кількості інформації про технологію та програмну логіку на стороні сервера, наскільки це можливо. Зокрема, клієнт не повинен знати про тонкощі обробки запитів, конкретну СУБД, яка використовується для зберігання даних, назви таблиць і полів у базі даних, а також про те, чи взаємодіє клієнт безпосередньо з сервером, чи через проміжний вузол. Сегментація проекту на окремі компоненти сприятиме належній розробці, забезпечить масштабованість у разі потреби та дозволить модифікувати код без надмірності.

Кожне системне повідомлення повинно мати характеристики, що не потребують пояснень. Кожен запит повинен містити достатньо службової інформації, щоб сервер міг надати вичерпну відповідь без потреби в додатковій інформації. Аналогічно, відповідь сервера повинна повністю задовольняти запит, щоб уникнути подальших запитів і мінімізувати навантаження на сервер. Цього можна досягти шляхом точного аналізу даних, наданих користувачем, та ефективної інтеграції і виконання бізнес-логіки.

Протокол взаємодії між клієнтом і сервером має бути побудований за принципом "без стану". Зокрема, сервер не зберігає жодної інформації про стан клієнта в проміжках між клієнтськими запитами. Відповідальність за це завдання делеговано клієнтській програмі. Найважливіші дані можуть

зберігатися на пристрої клієнта у вигляді сесії та файлів cookie. Під час обробки запитів вважається, що клієнт перебуває в тимчасовому стані. Стан сесії зазвичай зберігається на стороні клієнта. Однак у випадках, коли необхідно підтримувати постійний стан, наприклад, для цілей автентифікації, інформація про сеанс може бути надіслана зовнішньому сервісу, наприклад, сервісу бази даних. Цей метод дозволяє прискорити роботу сервера, масштабувати систему і збільшити пропускну здатність клієнта без шкоди для продуктивності.

Система повинна мати можливість зберігати та отримувати запити з кешу. Це може виконуватися як на стороні клієнта, так і на проміжних вузлах системи. Крім того, важливо, щоб у відповідях сервера чітко вказувалося, явно чи неявно, чи є вони в кеші. Це має вирішальне значення для уникнення можливості надсилання клієнтам неточних даних. Впровадження кешування може підвищити ефективність і масштабованість системи, а також прискорити роботу за рахунок усунення непотрібних комунікацій між клієнтом і сервером.

Кожен системний ресурс повинен мати унікальний ідентифікатор. Дизайн REST-серверів передбачає таку необхідність. Цей метод дозволяє кожному сервісу розвиватися автономно. Перспективи користувача принципово відрізняються від ресурсів, до яких він отримує доступ. Отже, користувач не має можливості безпосередньо взаємодіяти з ресурсами. Однак, володіючи представленням ресурсу разом з його метаданими, він має достатньо інформації, щоб вносити зміни або видаляти ресурси.

Крім того, включення унікальної ідентичності для кожного ресурсу відкриває широкі можливості для покращення зв'язності системи та використання гіперпосилань для встановлення зв'язків між ресурсами. Це значно покращить користувацькі інтерфейси, спростивши навігацію клієнта і полегшивши доступ до взаємопов'язаної інформації з меншими зусиллями.

API має включати механізм обробки помилок. Щоб уникнути плутанини для користувача, рекомендується використовувати стандартні

коди помилок HTTP, а не спеціальні коди. Важливо, щоб описи помилок були стислими та вичерпними, що дозволить користувачеві швидко зрозуміти конкретну природу допущеної помилки. Однак важливо, щоб пояснення помилки було стислим і не містило зайвих деталей, щоб запобігти плутанині користувача.

API повинен надавати інформацію в універсально зрозумілому і сумісному форматі для обробки на всіх платформах. Наразі буде використовуватися формат JSON (JavaScript Object Notation). Можливість створювати та маніпулювати даними у форматі JSON доступна в більшості комп'ютерних мов за допомогою вже існуючого коду, незалежно від конкретної мови програмування, що використовується. JSON краще підходить для серіалізації складних структур даних порівняно з XML через свою стислість. XML дуже багатослівний і включає безліч сторонніх даних, що призводить до значного збільшення розміру повідомлення і, відповідно, до зниження швидкості передачі даних.

Хоча JSON має багато переваг, бувають випадки, коли необхідно отримати дані у форматі XML, особливо коли певні сервіси вже налаштовані на використання XML. Отже, API повинен бути сумісним з MediaType. MediaType або MIME-тип – це спосіб вказати серверу бажаний формат для отримання даних користувачем. Використовується ідентифікатор, що складається з двох частин, наприклад, "application/json".

Щоб гарантувати високий рівень безпеки системи та захистити обмін даними, необхідно, щоб система використовувала SSL (Secure Socket Layer), криптографічний протокол, який використовує асиметричну криптографію для аутентифікації ключів, симетричне шифрування для забезпечення конфіденційності та коди автентифікації для збереження цілісності повідомлень. За відсутності цього протоколу авторизація та автентифікація не мають значення.

Система повинна включати функції фільтрації, сортування та підкачки даних. Включення цих функцій до стандартного функціоналу має вирішальне

значення для полегшення навантаження на розробників, які використовують API, усуваючи необхідність для них самостійно реалізовувати ці можливості.

Система повинна забезпечувати процедури автентифікації та авторизації користувачів. Завдяки створенню окремого токену стає можливим перевірити особу користувача та визначити конкретні ресурси, до яких він має право доступу. Аутентифікація буде реалізована за допомогою технології JWT (JSON Web Token). JWT, що розшифровується як JSON Web Token, є загальноприйнятим стандартом для генерації маркерів доступу у форматі JSON. Сервер генерує ці токени, підписує їх за допомогою секретного ключа, а потім надсилає клієнту. Клієнт може використовувати отриманий токен для підтвердження своєї особи та отримання дозволу на доступ до захищених ресурсів сервера. Токени JWT можуть зберігати додаткову інформацію про користувача, яка може бути використана для підвищення продуктивності системи, поряд з підписом.

4.2 Проектування шару функцій проміжного виконання

Реалізувати ідею проміжного програмного забезпечення – вдалося завдяки використанню програмної платформи Node.js та фреймворку побудови Express.js. Одним з найважливіших аспектів цієї стратегії є те, що запит до кожного ресурсу обробляється не однією дією контролера, а цілим стеком сконструйованих функцій. У циклі "запит-відповідь" ці функції мають доступ до об'єкта запиту, об'єкта відповіді та наступної проміжної функції обробки. Крім того, вони мають доступ до об'єкта відповіді. Окрім можливості виконання будь-якого коду, модифікації об'єктів запиту та відповіді, дострокового завершення циклу запиту-відповіді та виклику наступної функції-обробника з проміжного стеку, кожна з цих функцій має можливість виконувати наступні додаткові дії.

Стек посередницьких функцій розбивається на фундаментальні принципи, які проілюстровані на Рисунку 4.1.

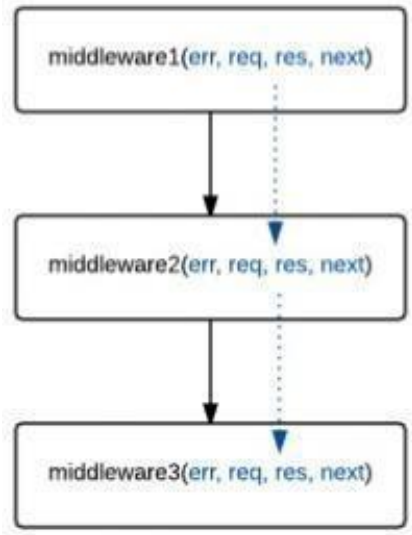


Рис. 4.1 Загальна концепція проміжного стека

У функції проміжного програмного забезпечення є три параметри:

- res, який є параметром відповіді HTTP;
- req, який є HTTP-параметром запиту;
- next, який є аргументом зворотного виклику наступної проміжної функції-обробника.

У випадку, якщо поточна проміжна функція обробки не є кінцевою функцією в циклі запит-відповідь, необхідно, щоб вона викликала функцію next () для передачі керування наступній функції в проміжному стеку.

Елементи проміжного стеку можна зв'язати між собою за допомогою функції next (). Існує два різних способи її застосування:

1. Почніть з передачі керування наступній функції у черзі. Коли це відбувається, функція next () викликається без параметрів.

2. Сповістити про це проміжну функцію стеку, яка спеціалізується на обробці помилок, і передати управління цій функції, яка відповідає за обробку помилок. У цьому конкретному випадку об'єкт помилки надсилається до функції у вигляді аргументу.

Переміщення вниз по проміжному стеку буде завершено, якщо не було викликано метод next (). Це означає, що рух буде завершено.

Крім того, функція може мати параметр err на додаток до трьох

параметрів, які були розглянуті раніше. З метою виконання обробки помилок, ця функція буде вважатися проміжною ланкою в результаті цього результату.

У Express.js доступні різноманітні функції проміжної обробки, зокрема наступні:

- проміжні функції прикладного рівня;
- функції проміжного рівня маршрутизації;
- проміжний обробник для керування обробкою помилок;
- вбудовані функції проміжної обробки;
- проміжні функції обробки сторонніх постачальників програмного забезпечення;
- проміжні функції обробки від сторонніх постачальників програмного забезпечення були включені.

Можно значно підвищити швидкість роботи системи, а також її продуктивність, побудувавши цілий стек функцій, які є проміжними функціями. Крім того, в результаті цього значно підвищиться рівень безпеки програми. Кожна проміжна функція обробки виконуватиме свою конкретну функцію, але при цьому матиме достатньо повноважень, щоб зупинити виконання запиту або змінити відповідь сервера, якщо вважатиме за потрібне це зробити. Послідовний виклик не дозволить вам обійти будь-які перевірки, проте, у випадку виявлення проблеми, програма буде миттєво перенаправлена до відповідної функції. У цьому випадку непотрібні перевірки не будуть виконуватися, що призведе до скорочення часу, необхідного для виконання запиту та його обробки.

Буде реалізовано додатковий рівень проміжних функцій обробки, які включатимуть автентифікацію, авторизацію, захист від міжсайтових атак, перевірку механізму CORS та інші сервіси. На рисунку 4.2 зображено структуру проміжного рівня, який присутній в системі.

Рис. 4.2 Рівень додаткових функцій обробки

4.3 Проектування загальної структури системи та розробка маршрутів

Це дослідження має на меті продемонструвати реалізацію додаткового рівня проміжних операцій обробки шляхом створення сервісу для ведення нотаток. Проект передбачає створення RESTful-API, що забезпечує реєстрацію та авторизацію користувачів, а також можливість читати, створювати, змінювати та видаляти нотатки, серед інших функцій.

Програмне забезпечення складатиметься з наступних модулів:

- Mongoose - для маніпуляцій з базами даних;
- Passport - для аутентифікації користувача;
- Body-parser - для ефективної обробки даних запиту.

Розробимо визначені "кінцеві точки" для нашого API. Визначимо адреси залучених сторін, окреслимо вхідні параметри, пояснимо логіку обробки та вкажемо потенційні вихідні результати.

URL-адресою для реєстрації нового користувача буде /api/register. HTTP надсилатиме запит за допомогою методу POST, оскільки він передає окремий набір даних і викликає зміну стану сервера. Після отримання такого запиту отримані дані будуть перевірені на достовірність, і буде зроблено спробу зареєструвати нового користувача, якщо дані є справжніми. Згодом буде згенеровано JSON Web Token (JWT), який слугуватиме засобом аутентифікації та авторизації користувача для доступу до певних сайтів. У випадку, якщо отримані дані не є справжніми,

користувачеві буде надіслано повідомлення, яке містить код помилки та вичерпне пояснення помилки.

Кінцева точка авторизації користувача буде розташована за адресою /api/login. Запит буде надіслано методом POST, щоб включити облікові дані користувача і викликати зміну статусу. Після отримання запиту дані пройдуть ретельну перевірку на достовірність. Згодом логін або електронна пошта користувача будуть запитані в базі даних. Якщо такий користувач існує, отриманий пароль буде порівняно з паролем, що зберігається в базі даних. Якщо надані ім'я користувача та пароль є точними, користувач отримає відповідь з кодом статусу 200 OK і буде перенаправлений на сторінку особистих нотаток. В іншому випадку користувачеві буде надіслано повідомлення з кодом і детальним поясненням помилки.

Для створення нотаток буде використовуватися кінцева точка /api/notes. Запит буде передано до програми передачі технологій малого бізнесу (HTTP) за допомогою механізму POST. Сервер очікує на отримання як заголовка, так і тіла ноти. Перед подальшою обробкою надіслані дані пройдуть валідацію. Після успішної валідації в базі даних буде створено новий запис. Буде згенеровано автоматичну позначку часу. Згодом користувач отримає повідомлення про успішне виконання запиту.

Доступ до шляху зміни нотатки буде здійснюватися за URL-адресою /api/notes/:noteId. HTTP виконає запит за допомогою методу PUT, замінивши таким чином всі наявні подання ресурсу на дані, надані у запиті. Перш ніж продовжити, система перевірить авторизацію користувача і з'ясує, чи має він відповідні дозволи на модифікацію нотатки. Параметри запиту включатимуть ідентифікатор нотатки, яку потрібно оновити, а також новий заголовок і текст нотатки. Ідентифікатор буде використано для вилучення нотатки з бази даних і модифікації її полів з урахуванням нових даних, наданих у запиті. Мітка часу створення нотатки залишиться незмінною, тоді як поле, що містить останню мітку часу редагування, буде змінено.

URI для доступу до всіх нотаток, що належать авторизованому користувачеві, буде /api/notes, а для виконання запиту буде використовуватися метод HTTP GET. Спочатку буде перевірено дійсність JWT-токену користувача. Якщо токен автентифіковано, всі нотатки, створені користувачем, який увійшов до системи, будуть завантажені з бази даних. Якщо в параметрах запиту явно не вказано кількість результатів на сторінці, буде використано значення за замовчуванням. Якщо це значення менше, ніж кількість записів у базі даних, у заголовку пересилається адреса наступного запиту, який необхідно виконати, щоб отримати наступний сегмент даних. Отже, розробники, які використовують цей API, будуть звільнені від необхідності самостійно реалізовувати метод пагінації.

Маршрут /api/notes/:noteId слугуватиме кінцевою точкою для видалення нотатки. Однак для виконання запиту буде використано метод HTTP DELETE. Видалення є небезпечним процесом, який потребує пильної уваги та ретельних перевірок. На початковому етапі обробки запиту система перевірить, чи користувач має право та відповідні привілеї для видалення нотатки. В іншому випадку користувачеві буде надіслано повідомлення з кодом і детальним поясненням помилки. Якщо користувач має достатні повноваження, буде зроблено запит до бази даних щодо нотатки, пов'язаної з її унікальним ідентифікатором (ID), і після цього її буде видалено.

Шлях для доступу до конкретної нотатки буде мати вигляд /api/notes/:noteId, а запит буде виконано за допомогою методу HTTP-GET. Після отримання запиту буде зроблено запит до бази даних для пошуку нотатки за ідентифікатором, вказаним у параметрах запиту. Якщо автентифікація користувача буде успішно підтверджена і нотатка з наданим ідентифікатором буде доступна, вона буде передана користувачеві у відповідь. В іншому випадку користувачеві буде показано повідомлення про помилку.

5 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

5.1 Розробка діаграм стану системи

Спочатку необхідно створити семантичне зображення додатку. Проаналізуємо та складемо вичерпний перелік багатьох категорій даних, які клієнтський додаток може захотіти отримати за допомогою запропонованого сервісу. Семантичний опис надає повне уявлення про значення даних у додатку та прояснює процеси, що відбуваються в додатку. Розробка зручного API для клієнтів є нашою першочерговою метою, тому дуже важливо надавати пріоритет перспективі клієнтської програми, а не сервісу.

Основні семантичні дескриптори нашого сервісу включають

- id - окремий ідентифікатор для кожного запису в системі;
- text - зміст кожної нотатки;
- updatedAt - дата останнього редагування кожної нотатки;
- createdAt - дата створення для кожної нотатки;
- title - назва, призначена для кожної нотатки;
- authorId - ідентифікатор користувача, відповідального за створення нотатки.

Перерахувавши всі елементи програми, можемо перейти до наступного етапу - побудови діаграми станів для нашого API. Кожен блок на діаграмі представляє потенційний стан, який є документом, що містить один або декілька семантичних дескрипторів, зазначених вище. Стрілками позначені переходи між станами. Ці переходи викликаються HTTP-запитами.

У нашій ситуації клієнтська програма може виконувати такі функції, як пошук, перегляд, створення, редагування та видалення нотаток. Більшість цих дій використовують значення стану для передачі даних між клієнтом і сервером. Наприклад, процес створення нової нотатки вимагає від користувача введення назви, тексту та змінних стану createdAt.

Побудована діаграма станів програми зображена на рисунку 5.1.

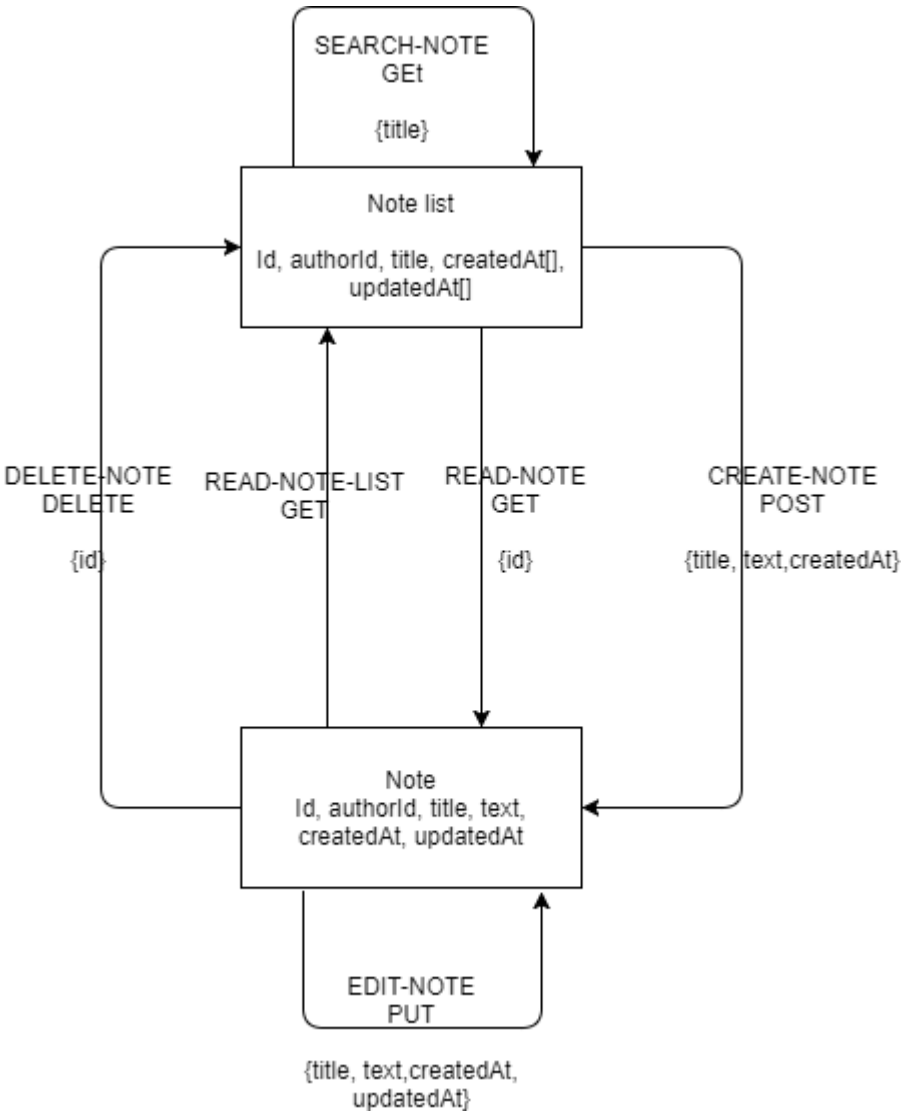


Рис. 5.1 Діаграма стану процесу

Всі дії, зображені на діаграмі, є семантичними дескрипторами, оскільки вони надають опис значення або семантики операцій, що виконуються сервером.

- edit-note. Змінити певну примітку;

- read-note-list. Отримати всі нотатки, створені користувачем;

- read-note. Отримати конкретну примітку;

- delete-note. Видалити примітку;

- create-note. Створити нову нотатку;

- search-note. знайти нотатку на основі її назви.

При створенні цієї діаграми стає простіше зрозуміти дані та дії, які можуть знадобитися клієнту для взаємодії з нашою службою. Крім того, це полегшує передачу бажаних станів від клієнта до сервісу.

Наступним етапом у процесі розробки є досягнення гармонізації назв. Наразі назви, які використовуються в цьому дослідженні для опису речей, не мають жодного значення, окрім конкретного завдання, над яким ми працюємо. Вони слугують лише для позначення дій або фрагментів даних, які клієнти будуть використовувати під час взаємодії з нашим сервісом. Дотримання угод про імена передбачає їх узгодження із загальновживаними словами за допомогою таких ресурсів, як schema.org, microformats.org, Dublin Core та IANA Link Relation Values. Ці ресурси надають колекції чітко визначених і широко використовуваних імен. Використання імен з цих ресурсів значно підвищує корисність нашого API, оскільки забезпечує однакове розуміння для всіх розробників. Після встановлення відповідності, подальші результати будуть наступними:

- id замінено на термін "identifier";
- title замінено на термін "name";
- createdAt замінено на термін "dateCreated";
- updatedAt замінено на термін "dateModified";
- read-note-list замінено на термін "collection";
- read-note замінено на термін "item";
- edit-note замінено на термін "edit";
- delete-note замінено на термін "delete";
- create-note замінено на термін "create-item";
- search-note замінено на термін "search".

Рис. 5.2 Заміна імен ресурсів

Крім того, для кожного переходу з одного стану програми в інший, замість того, щоб надавати конкретну техніку HTTP, ми визначаємо, чи є перехід безпечним і чи здатна дія призвести до того ж результату незалежно від того, скільки разів вона буде виконана. Ідемпотентні дії – це дії, які можна повторювати, не викликаючи жодних непередбачуваних побічних наслідків. Наприклад, HTTP PUT-запит вважається ідемпотентним, оскільки сервер повинен використовувати значення стану, надані клієнтом, для того, щоб змінити будь-які існуючі значення

в цільовому ресурсі. У той же час, метод HTTP POST не є ідемпотентним, оскільки він призначений для додавання отриманих значень до існуючих ресурсів, а не для їх заміни.

Переглянемо діаграму станів нашого додатку, щоб узгодити імена та оцінити безпечність і недієздатність переходів. Остаточний результат проілюстровано на рисунку 5.3.

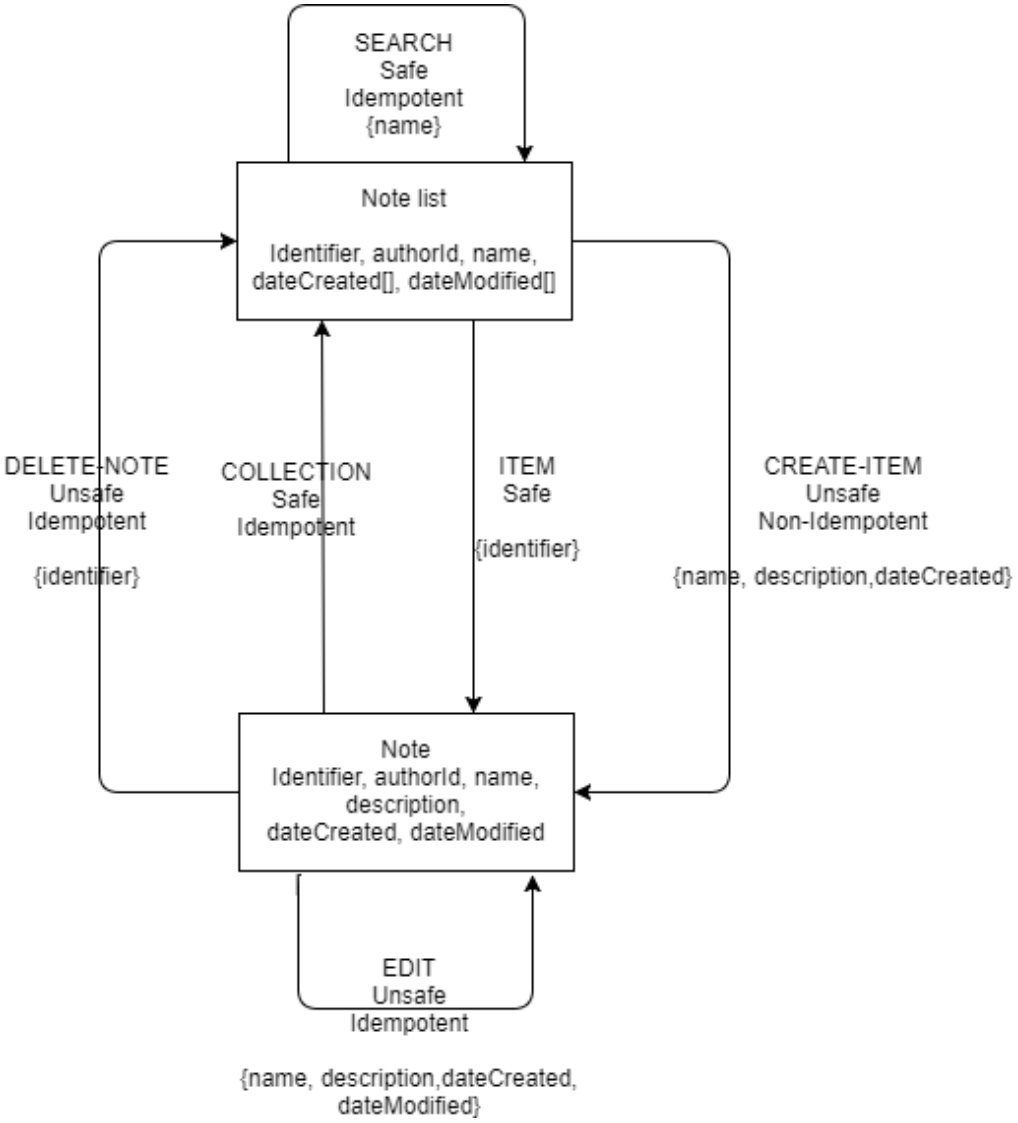


Рис. 5.3 Діаграма станів з відповідністю імен

Використання цього методу не тільки дозволяє зрозуміти функціональність, яка може знадобитися для клієнтських додатків, але й значно підвищує зручність використання інтерфейсу прикладного програмування (API) сервісу за рахунок консолідації назв станів і дій компонентів.

5.2 Проектування бази даних

Розробимо схему бази даних для нашого додатку. База даних буде зберігати дані користувачів та інформацію про їхні нотатки.

У випадку використання реляційної бази даних необхідно було б створити окремі таблиці для користувачів і нотаток, а потім з'єднати їх за допомогою зовнішніх ключів. Однак Mongo DB надає нам можливість зберігати дані різних типів і форматів в одній колекції. Отже, можна консолідувати дані користувача та всі його нотатки в єдину колекцію. Це розширить функціональність пошуку в базі даних і позбавить від необхідності виконання операцій JOIN при побудові запитів до бази даних.

Колекція міститиме такі атрибути, що характеризують користувачів: чіткий ідентифікатор, рядок, що позначає ім'я користувача, пароль, пов'язаний з користувачем, та адресу електронної пошти користувача. Кожне з цих полів є обов'язковим. Пароль буде зберігатися в зашифрованому форматі, оскільки він пройде через алгоритм хешування перед тим, як буде збережений в базі даних.

Ідентифікатор буде автоматично створено Mongo DB відповідно до визначення ідентифікатора об'єкта. Цей ідентифікатор складається з дванадцяти байт і включає наступні компоненти:

- Трибайтовий лічильник, який починається з випадково призначеного значення.

- Двобайтовий ідентифікатор процесу.

- Трибайтовий ідентифікатор пристрою.

- Чотирибайтове значення, що представляє кількість секунд, які минули з початку ери UNIX.

Тому немає необхідності турбуватися про створення ідентифікатора об'єкта бази даних і гарантії його унікальності та безпеки.

Об'єкт нотатки характеризується ідентифікатором, ім'ям, представленим у вигляді рядка, і вмістом нотатки, який також є рядком. Як і у випадку з користувачем, немає необхідності турбуватися про ідентифікацію. Крім того,

об'єкт нотатки міститиме два поля, призначені для зберігання точного часу і дати створення та останньої модифікації або оновлення нотатки. Mongo DB автоматично створить, заповнить і відредагує їх. Вам просто потрібно увімкнути позначку мітки часу.

На рисунку 5.4 показано остаточне представлення бази даних програми.

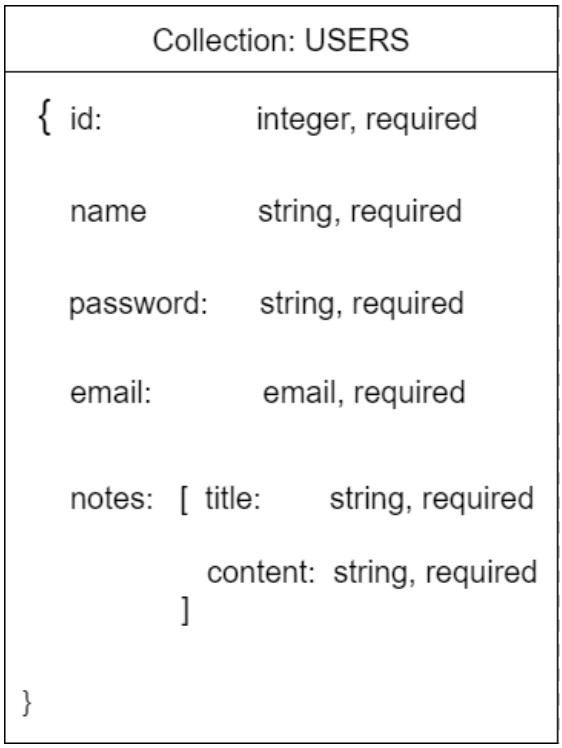


Рис. 5.4 Схема бази даних API

Для зберігання пароля в безпечному форматі використовуються хешування і соління. Ці криптографічні хеш-функції приймають на вхід фрагмент інформації і повертають рядок, який представляє інформацію, що була передана їм. Крім того, цей метод, як правило, не є незворотнім, а це означає, що "розхешувати" стрічку, яка була зашифрована таким чином, надзвичайно складно. Існує процес, відомий як соління, який передбачає додавання певних випадкових чисел до вхідних даних хеш-функції. Незважаючи на те, що ви можете зламати метод хешування, ви не зможете розшифрувати стрічку, якщо не знаєте значень солі, які були додані до функції. Це ще більше посилює безпеку і захищеність хешу.

Для хешування паролів наше програмне забезпечення використовує

модуль npm bcrypt, який використовує адаптивний метод генерації криптографічних ключів.

5.3 Розробка структури системи та кінцевих точок API

Отже, архітектурна структура Модель-Вигляд-Контролер (MVC) використовується нашою системою. До моделі буде включено схему бази даних для зберігання даних користувача та нотаток. Крім того, модель включатиме деяку бізнес-логіку, яка буде відповідати за обробку даних перед їх внесенням до бази даних та за пошук цих даних.

На додаток до додаткового шару проміжних функцій виконання, контролер буде місцем, де зберігається основна логіка програми. Сюди входять функції, які відповідають за управління маршрутами API. Сервер буде лише пропонувати необхідні ресурси у зручному для клієнтських програм форматі, а функція презентації буде делегована клієнтським програмам.

Загальну організацію папок і файлів, пов'язаних з проектом, показано на Рисунку 5.5.

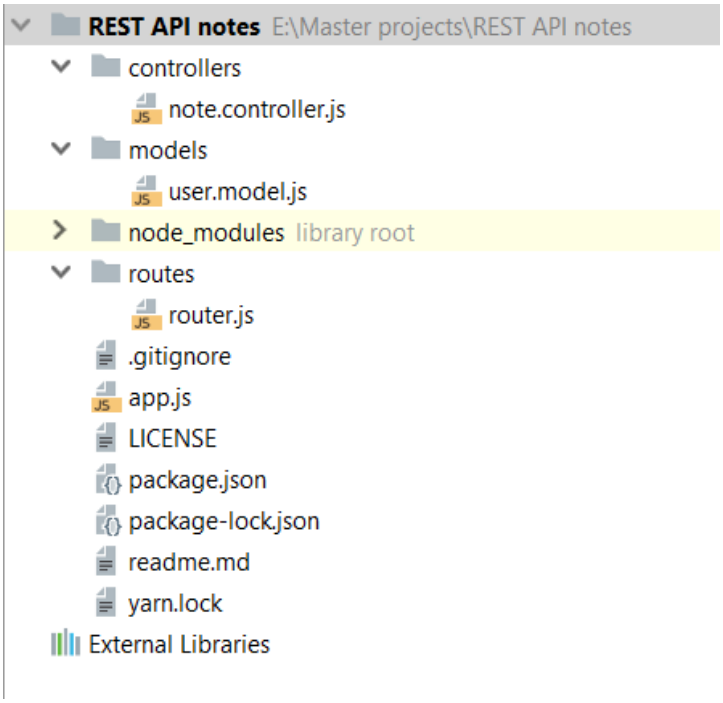


Рис. 5.5 Структура каталогів та файлів проекту

Файл app.js слугує точкою входу в програму. Він пов'язує разом усі необхідні пакети та бібліотеки, включаючи:

- express.js – є програмним фреймворком для створення та виконання додатків;

- Модуль "body-parser" – використовується для розбору вхідних запитів.

- Модуль "mongoose" – використовується для моделювання об'єктних даних, що полегшує взаємодію з базою даних Mongo DB.

- Модуль "bcrypt.js" – використовується для хешування та збереження паролів.

- Модуль "express session" – використовується для управління сесіями.

- Модуль "connect-mongo " – дозволяє зберігати сесії в базі даних. З іншого боку, "nodemon" - це компактний модуль, який негайно перезавантажує сервер при виявленні змін.

Рисунок 5.6 ілюструє зв'язок між вищезгаданими зовнішніми компонентами та нашою системою.

```
1  var express = require('express');
2  var app = express();
3  var bodyParser = require('body-parser');
4  var mongoose = require('mongoose');
5  var session = require('express-session');
6  var MongoStore = require('connect-mongo')(session);
7
```

Рис. 5.6 Підключення зовнішніх модулів

Крім того, як показано на рисунку 5.7, цей файл налаштовує функціонал роботи з сесіями та встановлює з’єднання з базою даних Mongo DB.

```
mongoose.connect('mongodb://localhost/testForAuth');
var db = mongoose.connection;
db.on('error', console.error.bind(console, 'connection error:'));
db.once('open', function () {
    // we're connected!
});
app.use(session({
    secret: 'work hard',
    resave: true,
    saveUninitialized: false,
    store: new MongoStore({mongooseConnection: db})
}));
```

Рис. 5.7 Підключення до Mongo DB

App.js відповідає за підключення файлу до кінцевих точок API і запуск HTTP-сервера, тобто сервера, на якому буде виконуватися система.

Всі маршрути API записані у файлі, який називається router.js. На рисунку 5.8 зображено кінцеві точки API, які були розроблені. Ці кінцеві точки було створено у попередньому розділі.

```
app.post('/login', notes.login);
app.post('/reg', notes.reg);
app.get('/logout', notes.logout);
app.get('/profile', notes.getUserProfile);
app.post('/create', notes.createNote);
app.get('/notes', notes.getAllNotes);
app.get('/note', notes.findNote);
app.put('/edit', notes.editNote);
app.delete('/delete', notes.deleteNote);
```

Рис. 5.8 Кінцеві точки API

На даний момент для кожного маршруту визначено метод HTTP, який буде використовуватися для виконання кожного маршруту системи, адресу кінцевої точки та функцію-обробник. Програмний контролер відповідатиме за реалізацію всіх процедур-обробників.

5.4 Робота з моделлю даних та авторизація

Файл user.model.js містить як компонент моделі даних, так і пов'язану з ним бізнес-логіку. В одному з попередніх пунктів ми розробили структуру майбутньої бази даних і дійшли висновку, що користувацькі нотатки будуть включені до тієї ж колекції, що й користувацькі дані. Це спростить пошук інформації, яку шукають користувачі, і усуне необхідність створювати різні таблиці та встановлювати зв'язки між людьми. Щоб втілити такий план у життя, ми спочатку визначаємо поля колекції "Користувачі", які включають ім'я користувача, електронну пошту та пароль, а потім робимо всі елементи обов'язковими для заповнення. Крім того, ми гарантуємо, що поля імені користувача та електронної пошти є ексклюзивними, щоб запобігти виникненню будь-яких конфліктів. На додаток до цього, ми визначимо поле нотаток, яке буде використовуватися для зберігання колекції користувацьких нотаток. Завдяки використанню Mongo DB можна зберігати дані різних типів і структур в одному документі. З огляду на це, ми визначимо колекцію нотаток, включивши поля заголовка і вмісту як частини цього масиву. Нижче наведено схему даних, яку ми отримаємо в результаті цього (Рисунок 5.9).

```
var NoteSchema = mongoose.Schema({
    title: String,
    content: String
}, {
    timestamps: true
});

var UserSchema = new mongoose.Schema({
  email: {
    type: String,
    unique: true,
    required: true,
    trim: true
  },
  username: {
    type: String,
    unique: true,
    required: true,
    trim: true
  },
  password: {
    type: String,
    required: true
  },
    notes: [NoteSchema]
});
```

Рис. 5.9 Схема даних моделі

Як було зазначено раніше, Mongo DB автоматично генерує поля ідентифікаторів, тому нам не потрібно турбуватися про це питання. Завдяки тому, що параметр timestamps має значення true, Mongo DB автоматично створюватиме і заповнюватиме поле createdAt щоразу, коли створюватиметься нова нотатка. Крім того, поле updatedAt буде створюватися і оновлюватися щоразу, коли нотатку буде змінено.

Хешування виконується над паролем перед тим, як він буде збережений у базі даних. Це робиться перед тим, як пароль буде збережено. Додатковий рівень функцій проміжного програмного забезпечення може включати цю функцію як один із своїх компонентів. Для самого процесу хешування використовується функція hash() модуля npm bcrypt. Ця функція також виконує соління, яке полягає у додаванні випадкових значень до пароля протягом декількох ітерацій для того, щоб зробити пароль більш безпечним. Як показано на рисунку 5.10, представлено код функції хешування.

```
54  UserSchema.pre('save', function (next) {
55    var user = this;
56    bcrypt.hash(user.password, 10, function (err, hash) {
57      if (err) {
58        return next(err);
59      }
60      user.password = hash;
61      next();
62    })
63  });
```

Рис. 5.10 Хешування пароля перед збереженням у базі даних

Після завершення цієї функції, яка є функцією проміжного рівня, викликається функція next (), яка є наступною функцією проміжного стеку. У разі виникнення помилки об'єкт помилки буде передано функції, яка була спеціально створена для обробки помилок. Ця функція візьме на себе відповідальність за ситуацію.

Аутентифікація користувача здійснюється за допомогою функції authenticate, код якої показано на рисунку 5.11.

```
33  UserSchema.statics.authenticate = function (email, password, callback) {
34    User.findOne({ email: email })
35      .exec(function (err, user) {
36        if (err) {
37          return callback(err)
38        } else if (!user) {
39          var err = new Error('User not found.');
40          err.status = 401;
41          return callback(err);
42        }
43        bcrypt.compare(password, user.password, function (err, result) {
44          if (result === true) {
45            return callback(null, user);
46          } else {
47            return callback();
48          }
49        })
50      });
51  }
```

Рис. 5.11 Функція автентифікації користувача

Адреса електронної пошти та пароль користувача, а також функція зворотного виклику, якій буде передано управління після виконання функції, є трьома параметрами, які приймає функція. У разі виникнення помилки, функція

зворотного виклику отримає об'єкт помилки, який містить код помилки, а також повідомлення. Вся інформація про користувача буде надіслана до функції зворотного виклику, якщо адреса електронної пошти та пароль, які були введені, ідентичні тим, що зберігаються в базі даних.

5.5 Обробка системних маршрутів та розробка додаткового шару проміжних функцій виконання

Основна логіка роботи програми знаходиться у файлі note.controller.js. Важлива роль проміжного виконання полягає у перевірці прав користувача на доступ або модифікацію ресурсів. У випадку, якщо користувач не має прав, йому буде показано повідомлення про помилку, що містить код помилки 401 "неавторизований". Або ж управління буде передано наступній функції в стеку.

Функція реєстрації нового користувача перевіряє логін, електронну пошту та пароль на правильність. Після цього в базі даних створюється новий користувач за допомогою функції User.create. Після успішного створення система поверне ідентифікатор нового користувача і збереже його в даних сеансу. У разі виникнення проблеми, користувач отримає повідомлення про помилку за допомогою наданого коду:

Статус відповіді встановлюється в 500.Transmit({повідомлення: "Під час реєстрації сталася помилка."})

Кінцева точка /login обробляється функцією login (), яка перевіряє правильність адреси електронної пошти та пароля користувача. Потім вона передає керування функції-посереднику authenticate (), яка підтверджує особу користувача. Код методу login () показано на рисунку 5.12.

Процес створення нової нотатки керується у функції create (). Спочатку управління передається проміжній стековій функції, яка перевіряє авторизацію користувача та його здатність генерувати нотатки. Якщо ця умова виконана, перевіряються параметри запиту. Після цього робиться спроба згенерувати нову нотатку за допомогою методу User.findOneAndUpdate (). Після цього користувачеві

надсилається повідомлення про успішне виконання дії або повідомлення про помилку, якщо виникають якісь проблеми.

```
exports.login = function (req, res, next) {
    if (req.body.email &&
        req.body.password) {

        User.authenticate(req.body.email, req.body.password, function (error, user) {
            if (error || !user) {
                var err = new Error('Wrong email or password.');
                err.status = 401;
                return next(err);
            } else {
                req.session.userId = user._id;
                return res.send(user._id);
            }
        });
    } else {
        res.status(500).send({message: "Some error ocuured while signing in."})
    }
};
```

Рис. 5.12 Функція обробки входу на сайт

Функція getNotes () обробляє кінцеву точку API /notes/ і отримує всі нотатки, що належать поточному користувачеві. Спочатку керування передається проміжній стековій функції, яка перевіряє авторизацію користувача. Якщо відповідь позитивна, виконується запит до бази даних, після чого користувачеві надається масив, що містить всі його нотатки. Або ж користувачеві надається повідомлення з кодом і детальним поясненням помилки, що виникла.

Щоб змінити створену користувачем нотатку, необхідно скористатися функцією editNote (). Кінцева точка /edit/ виконується системою. Спочатку керування передається проміжній функції проміжного програмного забезпечення, яка перевіряє авторизацію користувача. Якщо користувач авторизований, управління повертається. Таким чином, функція editNote() перевіряє правильність параметрів, наданих користувачем, і робить запит до бази даних, щоб знайти примітку, яка змінюється, тим самим оновлюючи відповідний документ у базі даних. Після успішного завершення операції користувач отримає змінену примітку. У разі невдачі буде відображено повідомлення про помилку, що містить код помилки та відповідний текст.

Кінцева точка /delete/ API виконується у функції deleteNote(). Як і в попередніх випадках, керування спочатку передається проміжній стековій функції, яка перевіряє авторизацію користувача. Після цього в базі даних виконується операція пошуку потрібної нотатки за її ідентифікатором, після чого вона видаляється. Користувач отримує повідомлення або про успішне видалення, або про проблему з авторизацією, або про відсутність нотатки з наданим ідентифікатором.

Для того, щоб ефективно керувати помилковим маршрутом, ми повинні включити додаткову функцію в наш додатковий стек. Ця функція буде викликатися у випадку, якщо користувач спробує пройти невірним шляхом. Йому буде доставлено повідомлення з кодом 404, що вказує на неіснування такого маршруту. Код функції зображено на рисунку 5.13.

```
app.use(function(req, res) {
  res.status(404).send({url: req.originalUrl + ' not found'})
});
```

Рис. 5.13 Функція обробки неправильного маршруту

У веб-додатках одна з найпоширеніших уразливостей відома як csrf - це різновид атаки на користувача веб-сайту, що використовує недоліки в протоколі HTTP. У випадку, коли можлива жертва атаки потрапляє на сторінку, яка була створена зловмисником, від її імені надсилається запит на інший сервер з метою виконання шкідливої дії (наприклад, переказу грошей). Для цього сервер, на який надсилається запит, повинен мати можливість аутентифікувати передбачувану жертву.

Використання сеансу, дані якого зберігаються в базі даних, а не на стороні клієнта, було вирішено як засіб захисту від csrf-атак. Клієнт не зможе дізнатися секретне слово, яке дозволить йому отримати доступ до сесії, якщо буде зроблено цей крок. На рисунку 5.14 показано код, який міститься у функції проміжного виконання. Ця функція відповідає за реалізацію конфігурації та

збереження даних сеансу в базі даних.

```
app.use(session({
    secret: 'work hard',
    resave: true,
    saveUninitialized: false,
    store: new MongoStore({mongooseConnection: db})
}));
```

Рис. 5.14 Налаштування сесії для збереження в базі даних

Взаємне використання ресурсів різного походження, також відоме як CORS, є важливим компонентом маршрутизації мережевого трафіку, який вирішує, чи будуть дозволені асинхронні запити, що надходять з інших доменів, чи ні. У зв'язку з тим, що ми перебуваємо в процесі створення загальнодоступного API, який буде задовольняти вимоги будь-якого клієнтського додатку, нам необхідно належним чином налаштувати механізм CORS. За допомогою npm-модуля cors можна налаштувати безліч параметрів. Ці параметри включають адреси, з яких буде доступний наш API, дозволені типи запитів і дозволені заголовки. На рисунку 5.15 зображено конфігурацію механізму CORS у робочому стані.

```
app.use(cors({
    origin: ["http://localhost:3001"],
    methods: ["GET", "POST", "PUT"],
    allowedHeaders: ["Content-Type", "Authorization"]
}));
```

Рис. 5.15 Проміжна функція для налаштування механізму CORS

Тому можна використовувати підхід стиснення відповідей JSON, а також статичних файлів у формат GZIP, щоб зробити запити простішими та ефективнішими. Цей формат можна зрозуміти, і більшість сучасних браузерів конвертують його автоматично. Як наслідок, це призведе до суттєвого прискорення роботи системи. Створіть додаток, який виконуватиме функцію проміжного виконання, і скористайтеся для цього модулем стиснення npm. Для

стиснення всіх відповідей сервера JSON використовуйте функцію compression(). Слід звернути увагу на те, що ця функція має бути розташована майже на самому верху проміжного стека, щоб гарантувати, що всі відповіді на запити будуть обов'язково стиснуті.

5.6 Тестування системи

Для того, щоб перевірити правильність роботи системи, ми проведемо тести на кінцевих точках API та проаналізуємо результати. Для цього ми будемо використовувати POSTMAN, надійний HTTP-клієнт, який дозволяє створювати HTTP-запити, створювати параметри в декількох форматах, створювати заголовки запитів, перевіряти запити, керувати файлами cookie, а також отримувати та аналізувати результати запитів.

Розглянемо функціональність кінцевої точки /reg, яка відповідає за реєстрацію нового користувача. Для цього згенеруємо POST-запит за адресою http://localhost:3000/reg і додамо параметри запиту у форматі application/json. У параметрах вкажемо адресу поштової скриньки, ім'я користувача та пароль (рис. 5.16).

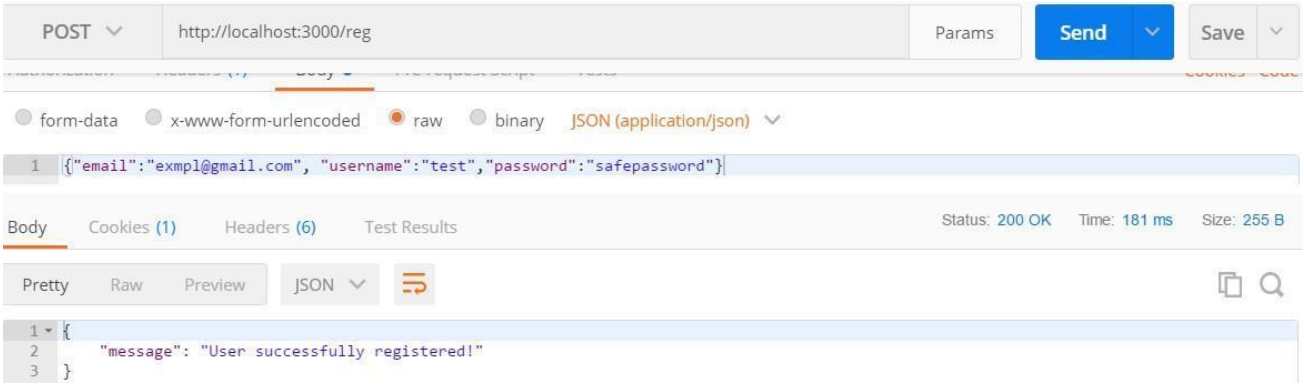


Рис. 5.16 Тестування функції створення нового користувача

Крім того, ми маємо можливість вивчати заголовки результатів запитів, які включають такі важливі атрибути, як e-Tag, що вказує на те, чи був запит кешований, і Content-Type, що описує формат відповіді. Створення нового

користувача було виконано без жодних проблем, як видно з поточного стану справ. Повторне надсилання запиту з ідентичними параметрами призведе до появи повідомлення про те, що користувач із зазначеною адресою вже існує в системі.

Для того, щоб перевірити здатність API обробляти помилкові маршрути, ми ініціюємо POST-запит до HTTP за адресою http://localhost:3000/test. Оскільки в нашому API відсутня ця кінцева точка, користувач повинен отримати повідомлення про помилку, що містить код статусу 404, який вказує на те, що запитувану сторінку не було знайдено. На рисунку 5.17 показано результат виконання запиту. Запити з неправильними маршрутами обробляються коректно.

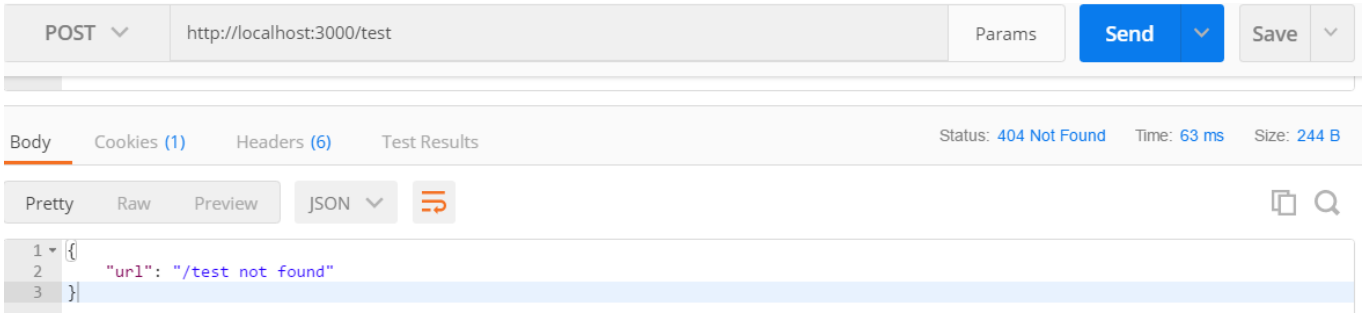


Рис. 5.17 Результат запиту до API з некоректним маршрутом

Ми згенеруємо HTTP-запит з методом POST і маршрутом http://localhost:3000/login, щоб перевірити, чи може користувач увійти на сайт. Для того, щоб протестувати відповідь від API, ми почнемо з того, що вкажемо в параметрах запиту помилковий пароль. Як наслідок, ми дізнаємося, що адреса електронної пошти або пароль, які ми ввели, були невірними, разом з кодом помилки 401, що означає "не авторизований" (рис. 5.18). Після цього надішлемо запит ще раз, цього разу гарантуючи, що аргументи містять правильний пароль. У відповідь ми отримаємо повідомлення про те, що вхід на сайт відбувся успішно, а повідомлення міститиме код 200, що означає "OK" (рис. 5.19).

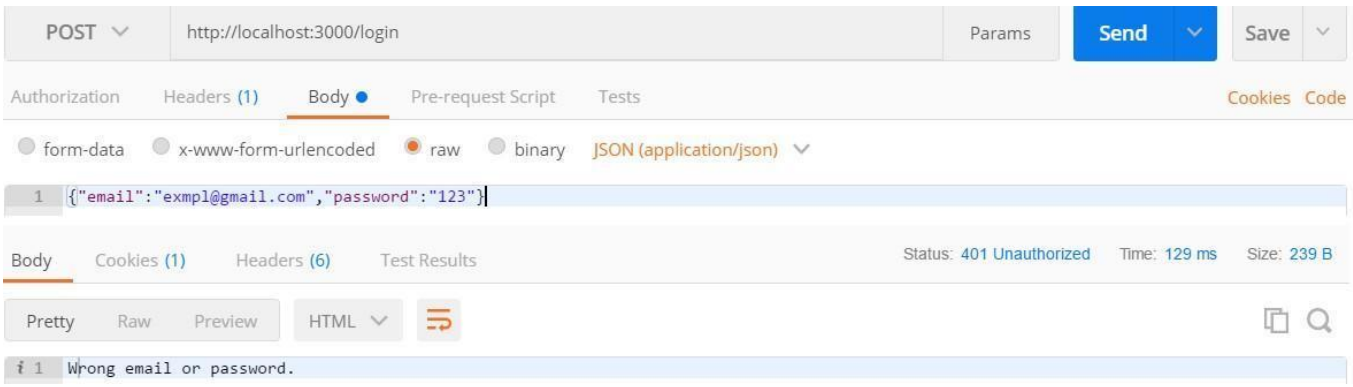


Рис. 5.18 Імітація спроби авторизації з неправильним паролем

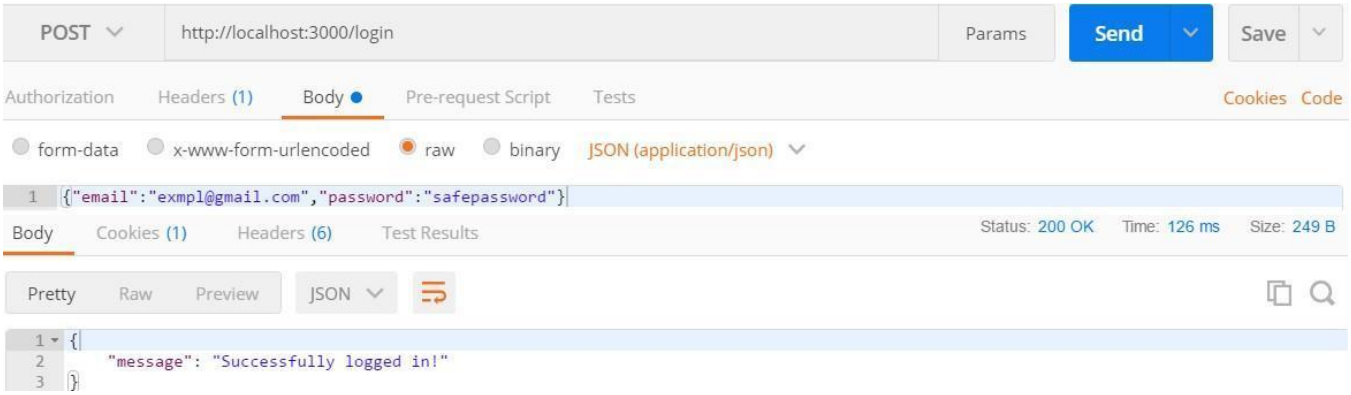


Рис. 5.19 Результат успішного виконання запиту на авторизацію

Ми згенеруємо POST-запит до HTTP за наступним маршрутом: http://localhost:3000/create. Це дозволить нам перевірити, що нова нотатка була створена. Включіть назву нової нотатки, а також її текст в параметри запиту, і переконайтеся, що ви використовуєте специфікацію application/json. Очікується, що в результаті виконання запиту ми отримаємо новостворену нотатку. Оскільки відповідь на запит одразу надає інформацію, яка може знадобитися користувачеві, цей формат відповіді відповідає принципам REST API. Користувачеві не потрібно робити додатковий запит, оскільки інформація вже міститься у відповіді. На рисунку 5.20 зображено результат тесту продуктивності.

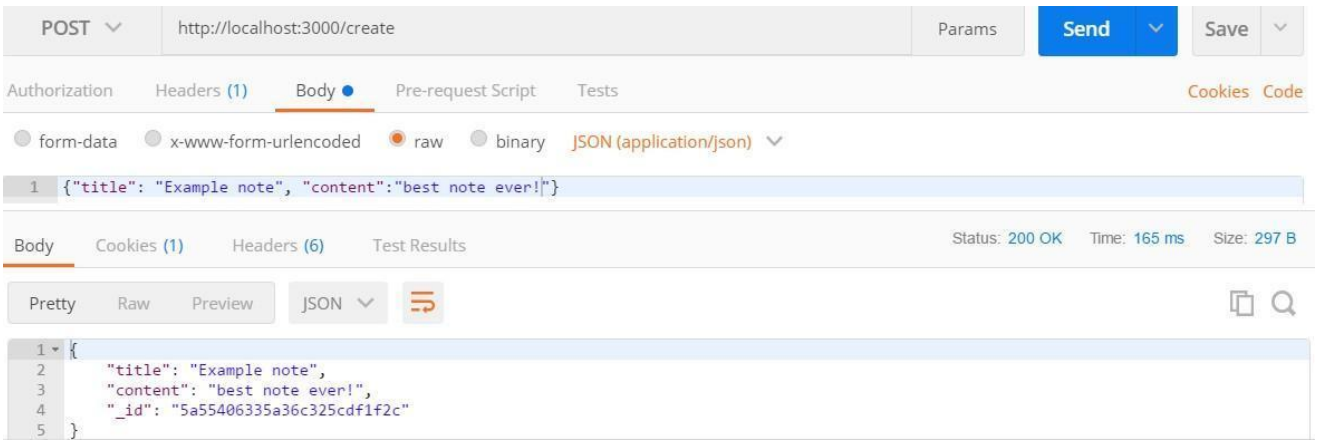


Рис. 5.20 Тестування створення нової нотатки

Щоб визначити, чи працює інтерфейс прикладного програмування (API) належним чином з точки зору підтвердження повноважень відправника запиту на виконання певних дій, давайте перевіримо. Вийдіть з веб-сайту за наступним URL-адресом: http://localhost:3000/logout. Після цього спробуйте створити нову нотатку, не маючи на це повноважень. Наслідком цього стане повідомлення про помилку з кодом 401 "not authorized" і текстом "Not authorized!" Отже, неавторизований користувач не зможе виконати жодних дій за допомогою інтерфейсу прикладного програмування (API). На рисунку 5.21 зображено результат тестування продуктивності.

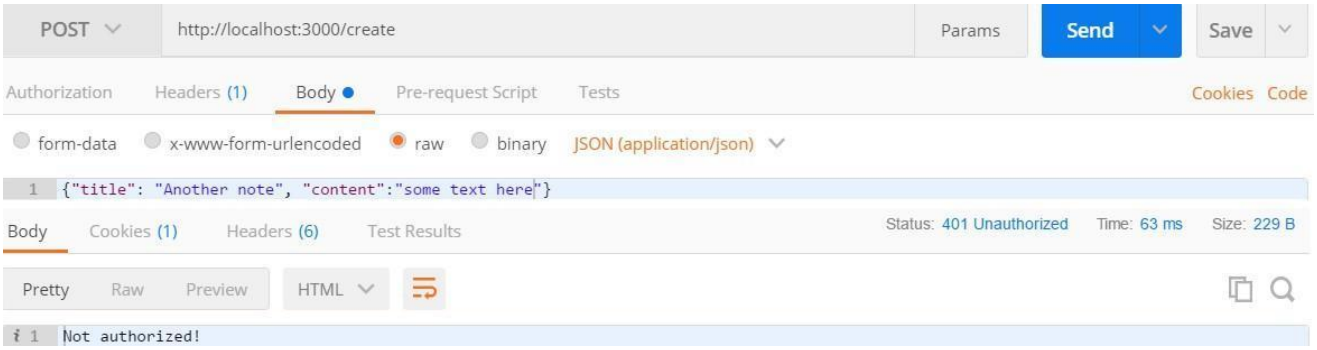


Рис. 5.21 Моделювання спроби несанкціонованого доступу

Ми створимо HTTP-запит, використовуючи метод GET і маршрут http://localhost:3000/notes, щоб протестувати функціональність отримання всіх нотаток користувача. Завдяки цьому ми отримаємо всі нотатки користувача, які були створені у форматі JSON (рис. 5.22).

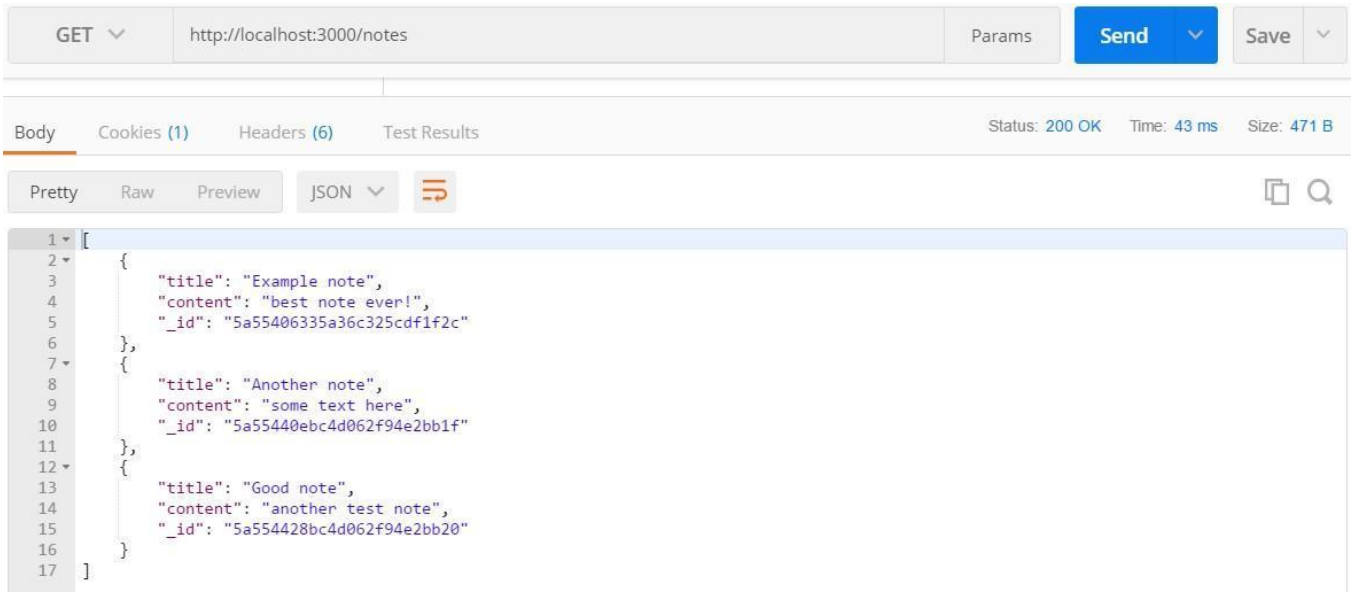


Рис. 5.22 Отримання всіх нотаток користувача

Використовуючи клієнт POSTMAN HTTP, ми провели тести, щоб переконатися, що наш інтерфейс прикладного програмування (API) функціонує належним чином. У той же час, вкрай важливо проводити тестування самого програмного коду, оскільки це дозволяє виявити помилки і некоректну поведінку програми на етапі розробки. Для цього використовуються модульні тести. Бібліотека тверджень Chai та фреймворк тестування Mocha - це те, що ми використовуємо в нашій ситуації. Процес розробки програми значно полегшується завдяки автоматизованим тестам, які також виявляють несправності та помилкову поведінку її частин на ранній стадії. Це призводить до значного покращення як кількості, так і якості процесу налагодження.

У багатьох сучасних проектах використовується процес тестування, відомий як Behavior Driven Development (BDD). Створення тесту можна здійснити за наступним алгоритмом:

1. Складіть специфікацію, яка описує основні можливості системи, що розглядається.

2. Наразі триває розробка початкової реалізації.

3. Проводяться тести та виправляються виявлені недоліки.

4. Специфікація розширюється, а потім відбувається повторення пунктів 3 і 4 до тих пір, поки не буде розроблена необхідна функціональність.

Хорошим прикладом тестів для кінцевої точки API /notes, яка повертає всі користувацькі нотатки, є той, який ми розглянемо тут. Приклад частини тестового коду показано на рисунку 5.23.

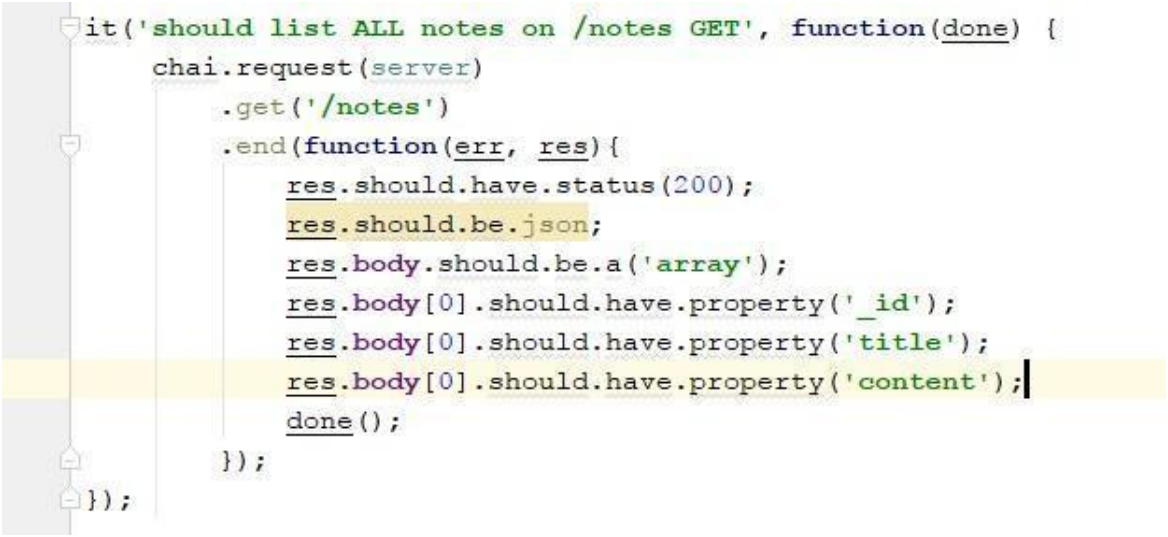


Рис. 5.23 Тестування кінцевого пункту /notes

Спочатку блок it містить опис того, що функція повинна виконувати. Для наших цілей результатом має бути компіляція всіх нотаток користувача. Рекомендується виконувати запит за допомогою техніки GET і напрямку HTTP /notes. Опис вимог до результату запиту надається за допомогою ключового слова end. Ці вимоги включають наступне:

- статус відповіді повинен бути 200 - "OK" Той факт, що на сервері не виникло помилки, свідчить про те, що повинні бути виконані наступні умови: формат відповіді повинен бути JSON;

- тіло відповіді на запит має бути масивом;

- кожен елемент масиву повинен містити атрибути _id, title та content. Тест буде вважатися неуспішним, якщо хоча б одна з умов не буде виконана.

У випадку, якщо ми виконаємо тест часу виконання, ми переконаємося, що обробка маршруту /notes задовольняє усім вимогам, які було задано.

Додаткове тестування.

У рамках нашого дослідження високо важливо чітко визначити версії використовуваних API. Ми використовуємо PHP 7.1 та Node.js 20.9.0 для

забезпечення стабільності та оптимальної продуктивності. Ця інформація дозволяє нам точно визначити середовище виконання та забезпечити правильну сумісність між компонентами проекту. Версії API вказані у технічних відомостях дослідження та грають ключову роль у забезпеченні прозорості і стабільності нашого розробленого рішення.

Рис. 5.24 Ілюстрація PHP версії 7.1

Рис. 5.25 Ілюстрація Node.js версії 20.9.0

Проілюструємо фрагменту коду JUnit, в якому реалізовано тестування одного з тестів – тест авторизації.

```
test.js
process.env.NODE_ENV = 'test';

const chai = require('chai'),
    chaiHttp = require('chai-http'),
    expect = chai.expect;
(fs = require('fs')), (server = require('./app'));

chai.use(chaiHttp);

describe('Users API', () : void  => {
    before(() : void  => {
        const TEST_DATA : {1: {...}, 2: {...}}  = {
            1: {
                id: 1,
                login: 'login1',
                password: 'password1',
            },
            2: {
                id: 2,
                login: 'login2',
                password: 'password2',
            },
        };

        fs.writeFileSync(
            'data-test.json',
            JSON.stringify(TEST_DATA)
        );
    });

    it('authorizeUser', (done) : void  => {
        chai.request(server)
            .get('/api/users')
            .end((err, res) : void  => {
                expect(err).to.be.null;
                expect(res).to.have.status(200);
                expect(res.body).to.haveOwnProperty('data');

                done();
            });
    });
```

Рис. 5.26 Фрагмент коду JUnit, в якому реалізовано тестування авторизації

Успішне виконання цього тесту є ключовим кроком у забезпеченні правильної роботи авторизаційного механізму нашої бази даних. Нижче подані

знімки екрану, які демонструють успішне виконання тесту авторизації в обох середовищах.

Рис. 5.27 Тест авторизації на PHP

Рис. 5.28 Тест авторизації на Node.js

Тому, вищезазначені рисунки – демонструють, що обидва середовища вдало виконали тест авторизації, підтверджуючи правильність нашого коду та правильну конфігурацію авторизаційного процесу в PHP та Node.js.

На наступних рисунках будуть представлені результати виконання різноманітних тестів, вони охоплюють ключові аспекти функціональності нашої БД та визначають її стабільність та ефективність в різних сценаріях використання. Результати чітко відображають час виконання кожного тесту, що дозволяє нам ефективно визначити та вирішити можливні проблеми або оптимізаційні можливості.

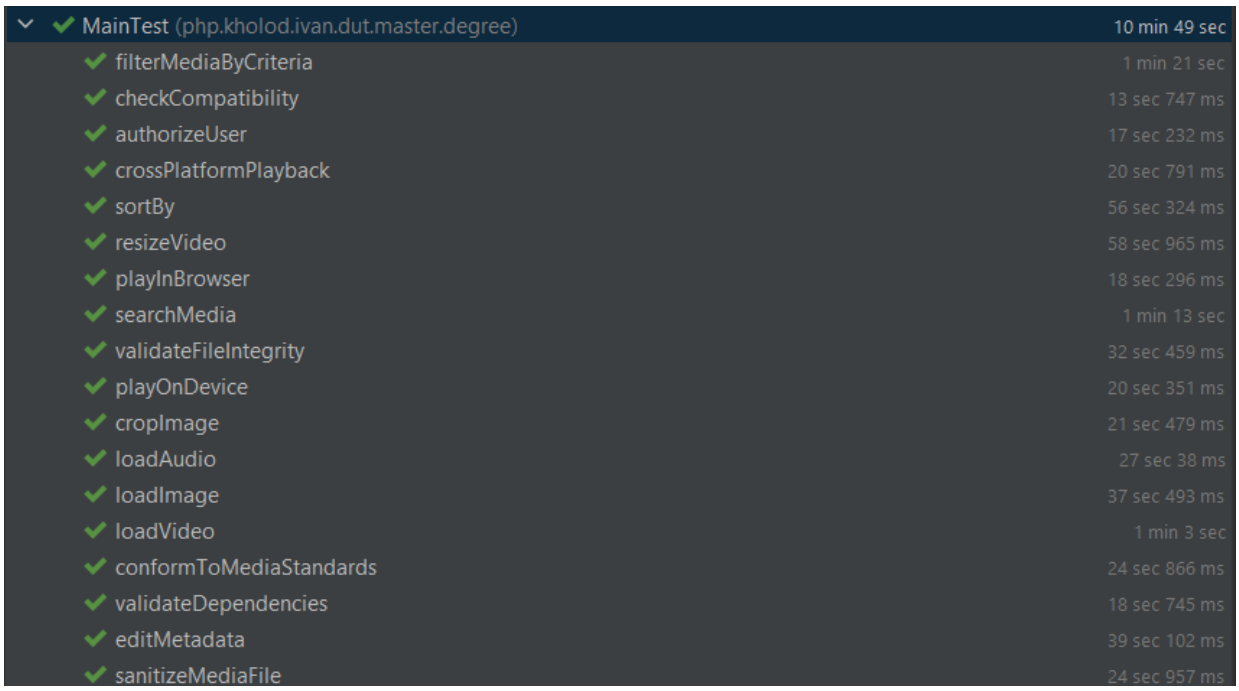


Рис. 5.29 Візуалізація результатів тестів до PHP

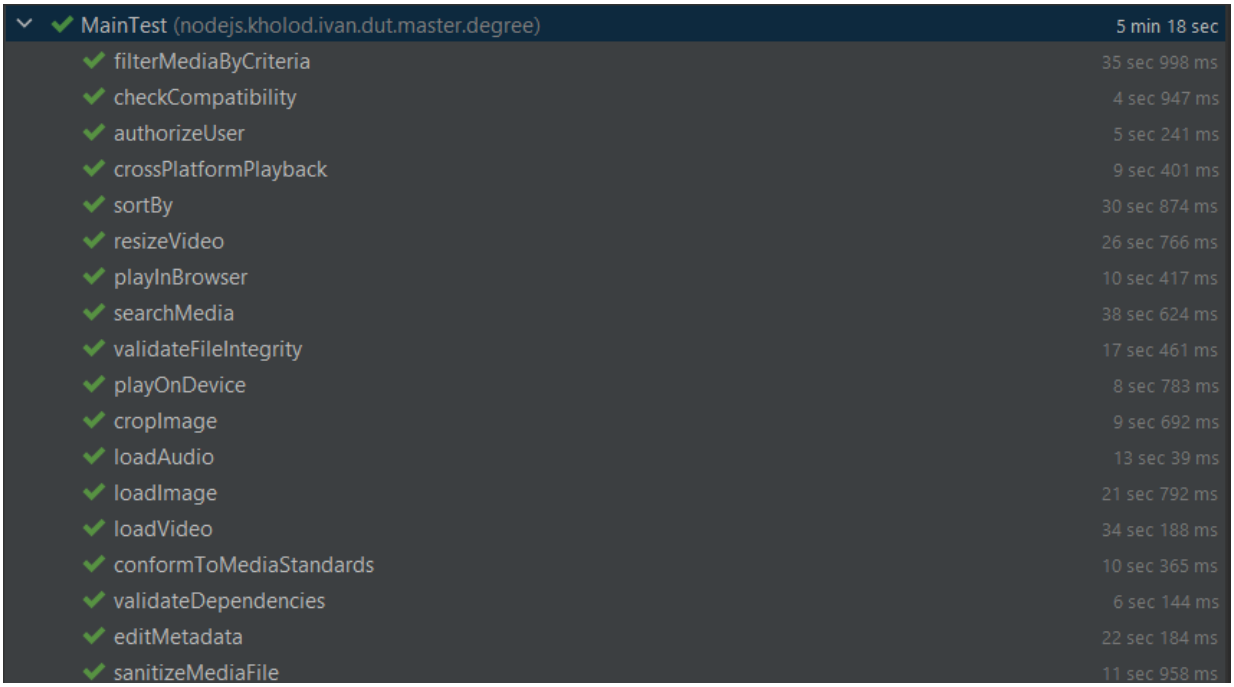


Рис. 5.30 Візуалізація результатів тестів до Node.js

Отримавши успішні результати тестів для обох API, наступним етапом нашого дослідження є створення докладної таблиці, яка відобразить час виконання кожного тесту на PHP та Node.js. Ця таблиця буде включати в себе конкретні значення часу, витраченого на виконання кожного тесту в обох середовищах. Додатково до основної інформації, ми плануємо додати стовбці із відсотковою

різницею у швидкості виконання між PHP та Node.js. Ці відсотки допоможуть краще розуміти, наскільки одне середовище є ефективніше за інше.

Крім того, у таблиці буде відображено, в скільки разів одне середовище перевершує чи поступається іншому в часі виконання тестів. Це важливий аспект, оскільки дозволить точніше визначити, в яких саме аспектах одне API виявилося більш швидкодійним, а в яких - менш ефективним.

Ця таблиця порівняння буде слугувати важливим інструментом для візуалізації та аналізу ефективності обох API. Вона не лише дозволить нам конкретно оцінити різницю у часі виконання кожного тесту, але і виявить потенційні напрямки для подальших оптимізацій у розробці API. Такий підхід допомагає усвідомити переваги та визначити можливості для подальшого вдосконалення та оптимізації обраного середовища в залежності від конкретних потреб проекту.

Таблиця 5.1

Порівняння часу виконання методів тестування між PHP та Node.js

Назва методу тестування	Час виконання PHP	Час виконання Nodejs	Відсоток різниці	Рази
MainTest	10 min 49 sec	5 min 18 sec	51.71%	2.04
filterMediaByCriteria	1 min 21 sec	35 sec 998 ms	55.55 %	2.25
checkCompatibility	13 sec 747 ms	4 sec 947 ms	63.59 %	2.77
authorizeUser	17 sec 232 ms	5 sec 241 ms	69.95 %	3.28
crossPlatformPlayback	20 sec 791 ms	9 sec 401 ms	54.85 %	2.21
sortBy	56 sec 324 ms	30 sec 874 ms	45.18 %	1.82
resizeVideo	58 sec 965 ms	26 sec 766 ms	54.58 %	2.21

Закінчення таблиці 5.1

playlnBrowser	18 sec 296 ms	10 sec 417 ms	43.28 %	1.75
searchMedia	1 min 13 sec	38 sec 624 ms	47.06 %	1.88
validateFilelntegrity	32 sec 459 ms	17 sec 461 ms	46.22 %	1.86
playOnDevice	20 sec 351 ms	8 sec 783 ms	56.99 %	2.31
croplmage	21 sec 479 ms	9 sec 692 ms	54.54 %	2.2
loadAudio	27 sec 38 ms	13 sec 39 ms	51.03 %	2.06
loadlmage	37 sec 493 ms	21 sec 792 ms	41.97 %	1.7
loadVideo	1 min 3 sec	34 sec 188 ms	46.34 %	1.84
conformToMediaStandards	24 sec 866 ms	10 sec 365 ms	58.15 %	2.27
validateDependencies	18 sec 745 ms	6 sec 144 ms	67.39 %	3.06
editMetadata	39 sec 102 ms	22 sec 184 ms	43.5 %	1.77
sanitizeMediaFile	24 sec 957 ms	11 sec 958 ms	51.95 %	2.08

З метою визначення зрозумілих переваг API, розробленого на Node.js порівняно з версією, написаною на PHP 7.1, було прийнято рішення використовувати отримані результати тестування для побудови інформативної діаграми. Ця діаграма надасть наглядне візуальне відображення різниці у часі виконання методів тестування обох API. Через використання діаграми, з’явиться можливість чітко виділити та порівняти різницю у швидкості виконання кожного методу тестування між PHP та Node.js. Цей графічний підхід дозволить набагато легше сприймати та аналізувати важливість й ефективність Node.js API,

підкреслюючи його переваги у вигляді графічного представлення порівнянь в часі виконання тестів.

Рис. 5.31 Діаграма порівняння швидкості виконання тестів

Отримані результати наукового дослідження, що стосується ефективності та новизни нашого підходу до розробки API, визначили та підкреслили важливий аспект нашої методики. Порівняльний аналіз між версією API на PHP 7.1 та Node.js 20.9.0 виявив, що розроблений нами метод працює значно швидше.

В рамках кваліфікаційної роботи, де основний акцент робився на детальному вивченні та впровадженні принципів спрощення інтеграції з мікросервісами та сторонніми API в Node.js додатках, отримані результати дали чітке підтвердження ефективності нашої методики. Реалізація API на Node.js показала приблизно 50% зменшення часу виконання порівняно з PHP 7.1, що вказує на вдосконалення продуктивності та визначає наш підхід як перспективний у розробці програмного забезпечення.

ВИСНОВКИ

Проаналізовано результати дослідження, спрямованого на оптимізацію розробки API для підвищення його швидкодії, надійності та ефективного використання в умовах зростання популярності мобільних пристроїв та розширення функціоналу веб-сервісів. Визначено основну мету дослідження – розробку методу, що не лише підвищить ефективність та безпеку API, але й спростить процес його розробки та інтеграції.

Досліджено та впроваджено високопродуктивний метод розробки API в кваліфікаційній роботі. Цей метод сприяє підвищенню продуктивності та швидкості роботи API, а також підвищенню рівня захищеності системи. Розроблено API для веб-додатку управління замітками для ілюстрації ефективності методу.

На основі вибору технологій та інструментів (JavaScript, Node.js, Express.js, MongoDB, Chai, Mocha) розроблено та протестовано API. Обрана мова програмування JavaScript спрощує розробку та сприяє високій швидкодії. Node.js, як першопрохідець у використанні мови JavaScript для серверних додатків, забезпечує велику масштабованість та ефективність. Express.js дозволяє легко розробляти REST API та забезпечує високий рівень безпеки та функціональності.

Вибір бази даних MongoDB дозволяє покращити масштабованість та використовує метод агрегації даних MapReduce. Тестування проведено з використанням бібліотек Chai та фреймворку Mocha. Отримані результати підтвердили ефективність розробленого методу.

Отже, дослідження визначило ефективність та новизну підходу до розробки API, що підтверджено порівняльним аналізом між версіями на PHP 7.1 та Node.js 20.9.0. Розроблений метод працює набагато швидше, зменшуючи час виконання методів тестування на приблизно 50%. Це робить наш підхід перспективним для розробки програмного забезпечення та відзначає його у контексті використання мікросервісів та сторонніх API в Node.js додатках.

ПЕРЕЛІК ПОСИЛАНЬ

1. Johnson, R. C. (2018). "Node.js Microservices: Building Scalable and Maintainable Server-Side Systems." Boston: Pearson Education.

2. Литвиненко, І. П. (2017). "Методи оптимізації інтеграції мікросервісів у великих корпоративних системах." Харків: Видавництво ХНУ ім. В. Н. Каразіна.

3. White, S. L. (2017). "Optimizing Microservices Integration in Large Enterprise Systems." London: Manning Publications.

4. Петренко, Н. М. (2016). "Аналіз інтеграції з мікросервісами в екосистемі Node.js." Одеса: Видавництво ОНУ ім. І. І. Мечникова.

5. Taylor, K. R. (2016). "Analysis of Microservices Integration in Node.js Ecosystem." San Francisco: O'Reilly Media.

6. Anderson, A. B. (2019). "Security Aspects in Microservices Integration with Node.js Applications." Berlin: Springer.

7. Rodriguez, D. C. (2018). "The Role of Node.js in Microservices and External API Interaction." Amsterdam: Apress.

8. Тимченко, О. С. (2018). "Роль Node.js у взаємодії з мікросервісами та зовнішніми API." Київ: Видавництво КНУ.

9. Martin, N. S. (2017). "Strategies for Interaction with Third-Party APIs in Modern Web Systems." New York: McGraw-Hill Education.

10. Ковальчук, О. І. (2019). "Стратегії взаємодії з сторонніми API у сучасних веб-додатках." Дніпро: Видавництво ДНУ.

11. Carter, M. M. (2016). "Optimizing Microservices Integration: Case Studies of Large-Scale Web Applications." San Francisco: Wiley.

12. Григорчук, С. С. (2018). "Вплив архітектурних рішень на ефективність інтеграції мікросервісів у веб-додатках." Львів: Видавництво ЛНУ ім. І. Франка.

13. Turner, D. V. (2015). "Architectural Aspects of Microservices Integration in Web Applications." London: Prentice Hall.

14. Кузнецов, Д. О. (2017). "Патерни взаємодії зі сторонніми API в сучасних веб-системах." Київ: Видавництво КПІ.

15. Peterson, O. O. (2019). "Microservices Patterns: Designing Scalable Architecture for Node.js Applications." Berlin: Addison-Wesley.

16. Іванова, М. М. (2016). "Оптимізація інтеграції з мікросервісами на прикладі розробки великих веб-додатків." Одеса: Видавництво ОНУ.

17. Cooper, E. P. (2018). "Node.js in Microservices: Best Practices for Efficient Integration." San Francisco: O'Reilly Media.

18. Шевченко, Д. В. (2017). "Архітектурні аспекти інтеграції мікросервісів у веб-застосунках." Харків: Видавництво ХНУ ім. В. Н. Каразіна.

19. Reed, L. M. (2017). "Modern Approaches to Microservices Architecture on the Node.js Platform." Amsterdam: Apress.

20. Кравченко, О. О. (2019). "Стратегії оптимізації взаємодії зі сторонніми API в середовищі Node.js." Дніпро: Видавництво ДНУ.

21. Wilson, H. J. (2016). "Standardization of Microservices Integration with Third-Party APIs in Contemporary Web Applications." New York: Springer.

22. Гончаренко, Л. М. (2018). "Сучасні підходи до розробки мікросервісних архітектур на базі Node.js." Запоріжжя: Видавництво ЗНУ.

23. Baker, R. J. (2015). "Efficient Methods for Interacting with Microservices on the Node.js Platform." London: Wiley.

24. Сидоренко, О. П. (2016). "Оптимізація взаємодії з мікросервісами у великих веб-проектах." Львів: Видавництво ЛНУ ім. І. Франка.

25. Clark, M. A. (2019). "Simplifying Microservices Integration in Node.js Applications: Practical Insights." Berlin: Pearson Education.

26. Козак, І. І. (2017). "Стандартизація інтеграції з мікросервісами та сторонніми API в сучасних веб-додатках." Київ: Видавництво КПІ.

27. Hill, S. P. (2018). "Impact of Architectural Decisions on Microservices Integration in Web Applications." San Francisco: McGraw-Hill Education.

28. Біловол, В. В. (2019). "Аспекти безпеки при інтеграції з мікросервісами у Node.js додатках." Запоріжжя: Видавництво ЗНУ.

29. Garcia, A. O. (2017). "Strategies for Interaction with Third-Party APIs: Lessons from Node.js Developers." Amsterdam: Addison-Wesley.

30. Попов, С. С. (2017). "Вплив архітектурних рішень на ефективність інтеграції мікросервісів у веб-додатках." Одеса: Видавництво ОНУ ім. І. І. Мечникова.

31. Mitchell, P. Q. (2016). "Security Considerations in Microservices Integration with Node.js." New York: Apress.

32. Головченко, А. О. (2016). "Стратегії взаємодії зі сторонніми API в сучасних веб-системах." Харків: Видавництво ХНУ ім. В. Н. Каразіна.

33. Sanders, T. R. (2015). "Role of Node.js in Microservices and External API Communication: A Comprehensive Guide." London: Prentice Hall.

34. Ткаченко, О. В. (2019). "Аспекти безпеки при інтеграції з мікросервісами у Node.js додатках." Запоріжжя: Видавництво ЗНУ.

35. Baker, R. J. (2015). "Efficient Methods for Interacting with Microservices on the Node.js Platform." London: Wiley.

36. Сидоренко, О. П. (2016). "Оптимізація взаємодії з мікросервісами у великих веб-проектах." Львів: Видавництво ЛНУ ім. І. Франка.

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

(Презентація)

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО - КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Магістерськаробота

«Розробка методики спрощення інтеграції з мікросервісами та стороннімиAPI у Node.jsдодатках»

Виконав: студент групи ПДМ-62 ХолодІван Петрович

Керівник: д.т.н., проф., професор кафедри ІПЗ Бондарчук А.П.

Київ - 2023

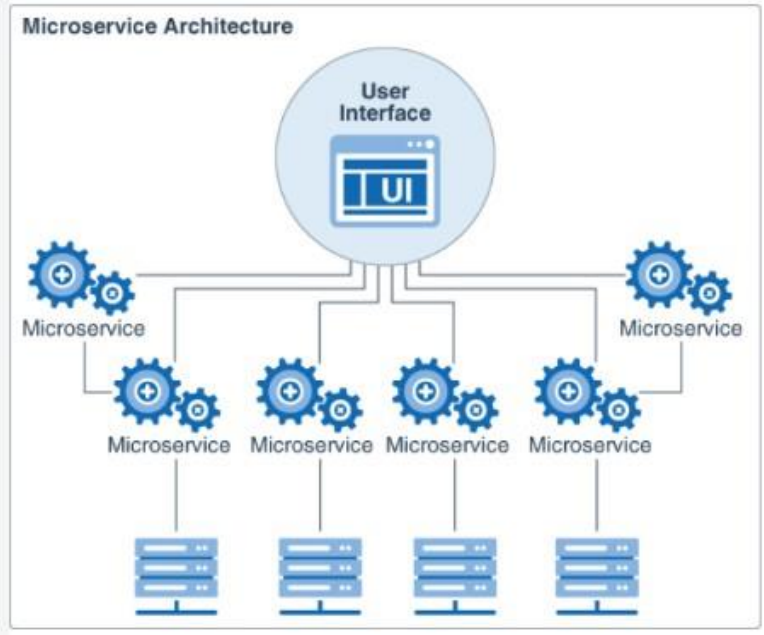
МЕТА, ОБ’ЄКТА ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи – підвищення ефективності та надійності інтеграції з мікросервісами та сторонніми API у Node.js додатках шляхом розроблення та валідації методики.

Об’єкт дослідження – конкретні аспекти, що впливають на ефективність та надійність інтеграції мікросервісів та сторонніх API в Node.js додатках, включаючи розробку та валідацію методики для поліпшення цього процесу..

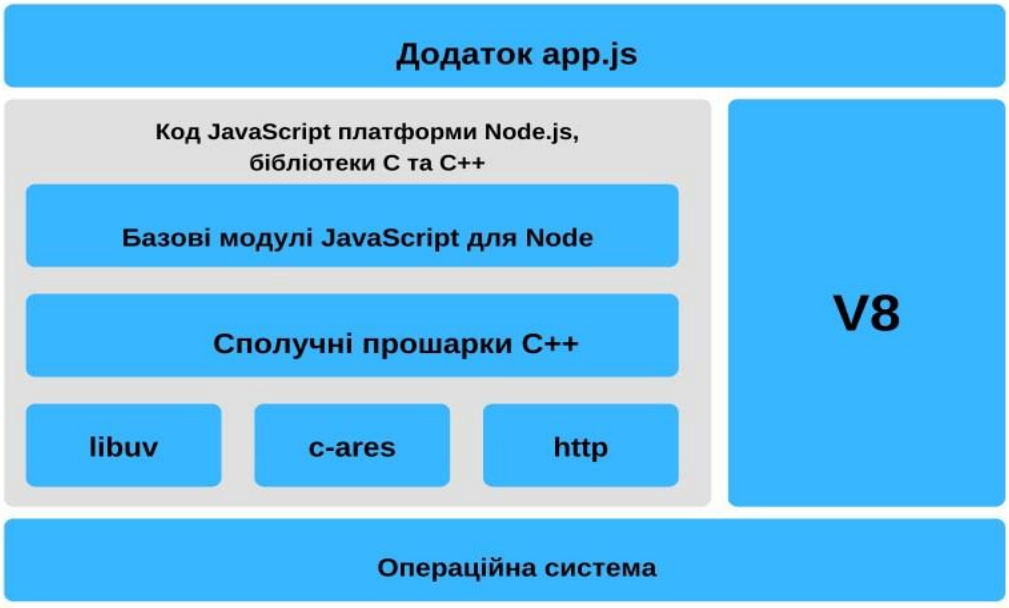
Предмет дослідження – процес інтеграції мікросервісів та сторонніх API в Node.js додатках з орієнтацією на визначення оптимальних методів оптимізації та спрощення цього процесу.

Модельархітектури мікросервісів



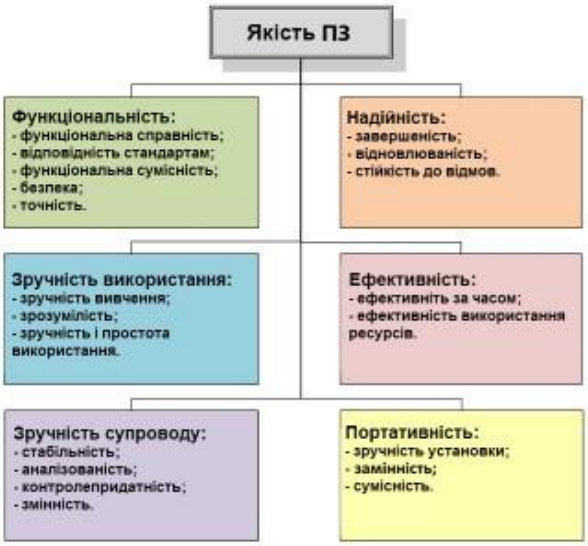
3

Структура веб-додаткуNode.js



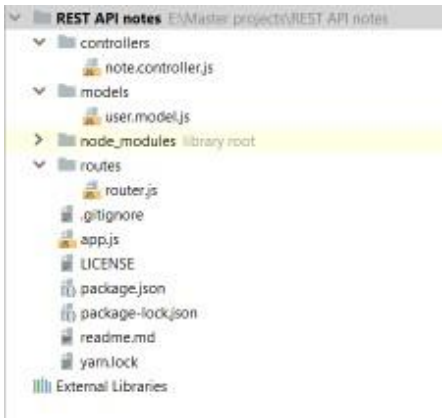
4

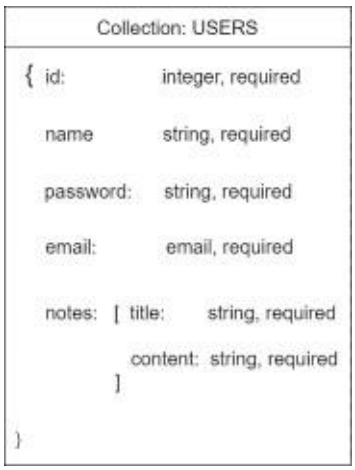
Характеристики якості програмного забезпечення



5

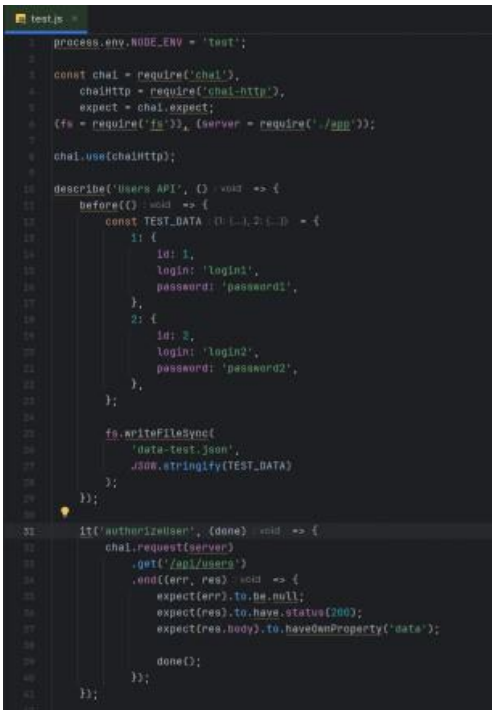
Як працює REST API





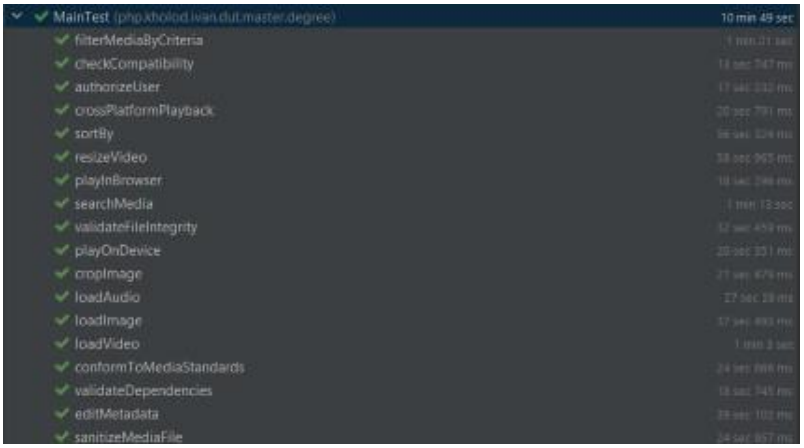
6

Приклад одного із тестів(authorizeUser)



7

Успішне проходження тестів



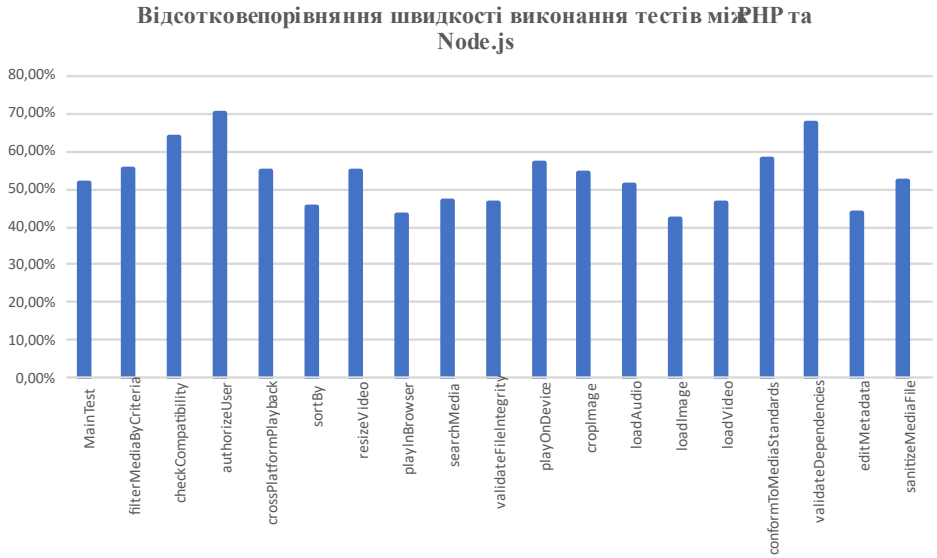
8

Результати тестування основних функцій медіабібліотеки

Назва методу тестування	Час виконання PHP	Час виконання Node.js	Відсоток різниці	Рази
MainTest	10 min 49 sec	5 min 18 sec	51.71%	2.04
filterMediaByCriteria	1 min 21 sec	35 sec 998 m	55.55%	2.25
checkCompatibility	13 sec 747 m	4 sec 947 m	63.59%	2.77
authorizeUser	17 sec 232 m	5 sec 241 m	69.95%	3.28
crossPlatformPlayback	20 sec 791 m	9 sec 401 m	54.85%	2.21
sortBy	56 sec 324 m	30 sec 874 m	45.18%	1.82
resizeVideo	58 sec 965 m	26 sec 766 m	54.58%	2.21
playInBrowser	18 sec 296 m	10 sec 417 m	43.28%	1.75
searchMedia	1 min 13 sec	38 sec 624 m	47.06%	1.88
validateFileIntegrity	32 sec 459 m	17 sec 461 m	46.22 %	1.86
playOnDevice	20 sec 351 m	8 sec 783 m	56.99 %	2.31
cropImage	21 sec 479 m	9 sec 692 m	54.54 %	2.2
loadAudio	27 sec 38 m	13 sec 39 m	51.03 %	2.06
loadImage	37 sec 493 m	21 sec 792 m	41.97 %	1.7
loadVideo	1 min 3 sec	34 sec 188 m	46.34 %	1.84
conformToMediaStandards	24 sec 866 m	10 sec 365 m	58.15 %	2.27
validateDependencies	18 sec 745 m	6 sec 144 m	67.39 %	3.06
editMetadata	39 sec 102 m	22 sec 184 m	43.5 %	1.77
sanitizeMediaFile	24 sec 957 m	11 sec 958 m	51.95 %	2.08

9

Підвищення якості опрацювання за допомогою інтеграції



10

ВИСНОВКИ

В даній кваліфікаційній роботі була запропонована та детально досліджена методика спрощення інтеграції з мікросервісами та сторонніми API в Node.js додатках. Основний акцент роботи робився на використанні єдиного API для взаємодії з цими ресурсами, що мав на меті покращити узгодженість та ефективність інтеграції.

В ході дослідження було проведено аналіз взаємодії Node.js зі сторонніми API, ідентифікацію потенційних обмежень та можливостей оптимізації цього процесу, а також розроблено новий метод інтеграції, який враховує специфіку мікросервісної архітектури та спрощує розробку.

Також, була розроблена модель Node.js додатку, в якому успішно використано розроблену методику . Для визначення ефективності та стабільності нового методу в реальних умовах використання було проведено систематичне тестування.

Загальна мета дослідження – полегшити та оптимізувати процес інтеграції з мікросервісами та сторонніми API в Node.js додатках, а також покращити узгодженість цього процесу за допомогою впровадження єдиного API, була досягнута. Отримані результати підтверджують ефективність нового методу і вказують на його перспективність для подальшого вдосконалення та використання в сучасних веб-додатках.

ПУБЛІКАЦІЇ ТА АПРОБАЦІЯ РОБОТИ

Статті:

1. Холод І. П., Розробка методики спрощення інтеграції з мікросервісами та сторонніми API в Node.js додатках. — Київ: ДУІКТ. Зв’язок. №6 2023.

Тези доповідей:

1. Холод І.П. Розробка методики спрощення інтеграції з мікросервісами та сторонніми API в Node.js додатках // Науково-практична конференція “Проблеми експлуатації та захисту інформаційно комунікаційних систем - 2023” -
Київ: НАУ, 2023 с.117
2. Бондарчук А.П., Холод І.П. Методи передачі стану в JavaScript // IV науково-практична конференція “Проблеми комп’ютерної інженерії” – Київ: ДУІКТ, 2023.