Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)
Факультет інформаційних технологій
(факультет)
Кафедра інформаційних технологій та комп'ютерної інженерії
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА

кваліфікаційної роботи ступеня ____магістра___
(бакалавра, спеціаліста, магістра)

студента _____Власенко Володимира Станіславовича_____
(ПІБ)
академічної групи _____123м-20-1_____
(шифр)
спеціальності _____123 «Комп'ютерна інженерія»_____
(код і назва спеціальності)
за освітньо-професійною програмою ___«Комп'ютерна інженерія»___
(офіційна назва)
на тему «Визначення структури комп'ютерної системи компанії з надання ріелторських послуг на основі оптимізації програмного забезпечення web-сервера»
(назва за наказом ректора)

Керівники	**Прізвище, ініціали**	**Оцінка за шкалою**		**Підпис**
		рейтинговою	**інституційною**	
кваліфікаційної роботи	доц. Ткаченко С.М.			
розділів:				
технічні вимоги до системи	доц. Ткаченко С.М.			
розроблення програмного забезпечення	доц. Ткаченко С.М.			

Рецензент				

Нормоконтролер	проф. Цвіркун Л.І.			

Дніпро
2022

ЗАТВЕРДЖЕНО:

завідувач кафедри
інформаційних технологій
та комп'ютерної інженерії
(повна назва)

_____ __Гнатушенко В.В._
(підпис) (прізвище, ініціали)
«___»_____ 202__ року

ЗАВДАННЯ
на кваліфікаційну роботу
ступеня _____магістр _____
(бакалавра, спеціаліста, магістра)

студенту _____Власенко В.С.____ **академічної групи** ____123м-20-1____
(прізвище та ініціали) (шифр)

спеціальності _____123 «Комп'ютерна інженерія»_____

за освітньою-професійною програмою _123 «Комп'ютерна інженерія»___
(офіційна назва)

на тему «Визначення структури комп'ютерної системи компанії з надання ріелторських послуг на основі оптимізації програмного забезпечення web-сервера»,

затверджену наказом ректора НТУ «Дніпровська політехніка» від 10.12.2021 р. №1036

Розділ	Зміст	Термін виконання
Стан питання та постановка завдання	На основі матеріалів виробничих практик, інших науково-технічних джерел сформулювати наукове завдання, конкретизувати предмет та мету досліджень	20.09.2021
Технічні вимоги до системи	Обґрунтувати теоретичну базу розв'язання наукового завдання, якому присвячено роботу	25.10.2021
Розроблення програмного забезпечення	Розробка програмного забезпечення	29.11.2021
Графічна частина	Графічні результати роботи подати у вигляді рисунків схем таблиць на 10 арк. формату А4.	05.01.2022

Завдання видано _____ доц. Ткаченко С.М.
(підпис керівника) (прізвище, ініціали)
Дата видачі 06 вересня 2021 р.

Дата подання до екзаменаційної комісії 10.01.2022 р.

Прийнято до виконання _____ ____Власенко В.С.__
(підпис студента) (прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка: 85 с., 34 рис., 5 табл., 4 додаток.

Мета: Визначення структури комп’ютерної системи компанії з надання ріелторських послуг на основі оптимізації програмного забезпечення web-сервера

У сучасний час використання web-додатків для задоволення бізнес цілей поширена практика. Для цілей компанії з надання ріелторських послуг необхідно було розробити додаток, що буде забезпечувати широкий спектр функціоналу. Однією з вимог до додатка була здатність працювати під певним навантаженням.

З метою оптимізації додатків було прийняте рішення застосувати до нього горизонтальне масштабування до web-сервера. Необхідно було розглянути основні засоби оптимізації, провести їх аналіз, здатність працювати як у пікових, так і стандартних навантаженнях та вибрати найбільш придатний.

Результати застосованих заходів оптимізації у вигляді таблиць, рисунків описані у пояснювальній записці та додатках.

JavaScript, NodeJS, V8, libuv, Docker, Cluster, Worker_thread, PM2, Haproxy, AWS, EC2, docker-compose, Autocannon, NPM.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

V8 – рушій від компанії Google, який використовується в браузері Google Chrome.

NodeJS – це програмна платформа, що використовує рушій V8 для запуску JavaScript у середовищі сервера.

Кластеризація - це метод перетворення кількох web-серверів у групу серверів, що діє як єдина система.

PM2 – сторонній менеджер процесів для web-серверів, що використовують NodeJS.

Контейнери – це абстракція на рівні програми, яка поєднує код та залежності.

ВСТУП

Використання додатків для організації та спрощення роботи з великими масивами даних для бізнесів різної спрямованості стає все більш поширеним у сучасному світі. Це зумовлено бажанням заощадити більше часу для роботи з одним клієнтом і тим самим збільшенням кількості нових споживачів з метою отримання більшої вигоди за рахунок більшої кількості споживачів. Виконання цього завдання лягає на додаткові комп'ютерні системи, які розробляються індивідуально відповідно до потреб будь-якої компанії з урахуванням усіх вимог до кінцевого продукту. Дані системи обробляють інформацію швидше, ніж із цим міг би впоратися співробітник, і на їхні плечі лягають такі завдання як:

- пошук інформації щодо заданих фільтрів;
- складні розрахунки;
- зберігання великих масивів даних;
- захист інформації;
- організація клієнтської бази.

Однак, використання додаткових комп'ютерних систем вимагає додаткових фінансових витрат, пов'язаних з розробкою програмного забезпечення, складом штату для адміністрування, управління та підтримки та доопрацювання нового функціоналу комп'ютерної системи, а також змістом інфраструктури, на якій працює дана комп'ютерна система. Якщо оцінка вартості розробки та підтримки комп'ютерної системи буде меншою, ніж очікуваний прибуток від економії на часі та кількості необхідного персоналу для ведення діяльності в будь-якій із сфер, компанія може використовувати додаткове програмне забезпечення для своїх цілей.

Мета роботи і завдання дослідження.

Метою представленої роботи є забезпечення необхідної швидкодії web-сервера компанії з надання ріелторських послуг на основі аналізу вимог зі сторони замовника та доступних інструментів та функціоналу.

Для досягнення поставленої мети необхідно вирішити такі завдання:

— Аналіз додатка та web-сервера компанії з надання ріелторських послуг;

— Побудова структурної схеми комп’ютерної системи;

— Вибір функціоналу та інструментів, що використовуються;

— Опис програмного забезпечення;

— Розробка інфраструктури для обраних засобів оптимізації;

— Дослідження розробленої моделі комп’ютерної системи за результатами досліджень.

Об’єкт дослідження: web-сервер компанії з надання ріелторських послуг

Предмет дослідження: структура моделі комп’ютерної системи та методи оцінювання швидкодії web-серверу компанії з надання ріелторських послуг.

Методи дослідження: для досягнення поставленої мети використовувалися інструменти з імітування навантаження на сервер.

Ідея роботи: Реалізація найбільш швидкодіючого засобу оптимізації web-серверу, що був реалізований за допомогою програмної платформи NodeJS.

Наукові положення:

1. Встановлено, що засоби оптимізації web-серверу, що призводять до майже безвідмовних відповідей під час великих навантажень можуть мати погану швидкодію під час невеликих навантажень.
2. Встановлено, що використання зовнішніх інструментів оптимізації має кращу швидкодію, ніж їх програмні аналоги.

1 СТАН ПИТАННЯ І ПОСТАНОВКА ЗАВДАННЯ

1.1 Характеристика галузі

Використання додатків для організації та спрощення роботи з великими масивами даних для бізнесів різної спрямованості стає все більш поширеним у сучасному світі. Це зумовлено бажанням заощадити більше часу для роботи з одним клієнтом і тим самим збільшенням кількості нових споживачів з метою отримання більшої вигоди за рахунок більшої кількості споживачів. Виконання цього завдання лягає на додаткові комп'ютерні системи, які розробляються індивідуально до потреб будь-якої компанії з урахуванням всіх вимог до реалізації та кінцевого продукту. Дані системи обробляють інформацію швидше, ніж із цим міг би впоратися співробітник, і на їхні плечі лягають такі завдання, як:

- пошук інформації щодо заданих фільтрів;
- складні розрахунки;
- зберігання великих масивів даних;
- захист інформації;
- організація клієнтської бази.

Однак, використання додаткових комп'ютерних систем вимагає додаткових фінансових витрат, пов'язаних з розробкою програмного забезпечення, складом штату для адміністрування, управління та підтримки та доопрацювання нового функціоналу комп'ютерної системи, а також змістом інфраструктури, на якій працює ця комп'ютерна система. Якщо оцінка вартості розробки та підтримки комп'ютерної системи буде меншою, ніж очікуваний прибуток від економії на часі та кількості необхідного персоналу для ведення діяльності в будь-якій сфері, компанія може використовувати додаткове програмне забезпечення для своїх цілей.

Існує велика різноманітність мов програмування, які можна використовувати для реалізації вебсервера компанії. Кожна з яких має певний набір переваг і недоліків, і з огляду на даний набір, визначається найбільш придатна або перелік з відповідних мов для реалізації необхідних цілей.

Серед найпоширеніших вимог можна назвати такі:

- можливість швидко виконувати паралельні обчислення, що потребують великої кількості процесорного часу;
- здатність швидко підключати та запускати необхідні модулі та бібліотеки;
- легковагість кінцевого коду програми;
- можливий час, витрачений на розробку програми;
- низька вартість розробки та підтримки інфраструктури комп'ютерної системи.

1.2 Характеристика об’єкта

Однією з мов програмування, які набирають популярності за рахунок великої кількості переваг перед іншими мовами є JavaScript. Донедавна ця мова програмування використовувалася для написання інтерфейсів і виконувала функції:

- з обробки серверних відповідей на графічному клієнті додатка з метою подальшого представлення їх для користувача у зручному для розуміння вигляді;
- реалізації інтерактивних анімацій для графічного клієнта програмного забезпечення;
- використання додаткового функціоналу доступного для браузерного вікна та документа.

Ця мова програмування стала популярною за рахунок її безальтернативності у використанні для розробки графічних клієнтів програм, які працюють у браузері. І за рахунок її великої популярності швидкість вивчення стає вищою, за рахунок більшої кількості інформації, що лежить у відкритому доступі, яку публікують більш кваліфіковані фахівці. Крім того, ентузіасти цієї мови займаються можливістю використання JavaScript не тільки для розробки браузерних графічних інтерфейсів, але й додатків для робочих комп'ютерів, мобільних додатків, web-серверів і навіть для програмування фізичних апаратів.

Використання JavaScript у середовищах, відмінних від браузерів, стає можливим завдяки адаптації компіляторів коду під необхідне оточення, чи то мобільний телефон, чи віддалений сервер.

З найбільш популярних движків, які інтерпретують і компілюють код, можна виділити наступні:

- V8 – двигун від компанії Google, який використовується в браузері Google Chrome, а також всіх інших браузерах на базі Chromium, наприклад Edge (оновлена версія Internet Explorer) та Samsung Internet. Він найчастіше використовується для адаптації використання JavaScript для розробки програмного забезпечення під персональні комп'ютери, наприклад бібліотеки CEF та Electron. А також для розробки програмних платформ NodeJS та Deno для запуску JavaScript на web-сервері.
- SpiderMonkey – двигун від компанії Mozilla, який використовується в браузері Firefox, а також в інших браузерах на його базі.
- JavaScriptCore – движок від компанії Apple для браузера Safari, який також використовується у браузерах на основі WebKit. Крім того цей двигун використовується в найбільш популярній програмній платформі для розробки мобільних додатків на JavaScript – React Native.

- Chakra – двигун від компанії Microsoft, який використовується в браузері Internet Explorer.
- Rhino – двигун Mozilla Foundation, написаний на Java з відкритим вихідним кодом.
- KJS - движок компанії KDE для веб-браузера Konqueror.
- Nashorn – двигун компанії Oracle Java Languages and Tool Group з відкритим вихідним кодом.
- JerryScript – легкий двигун для Інтернету речей.

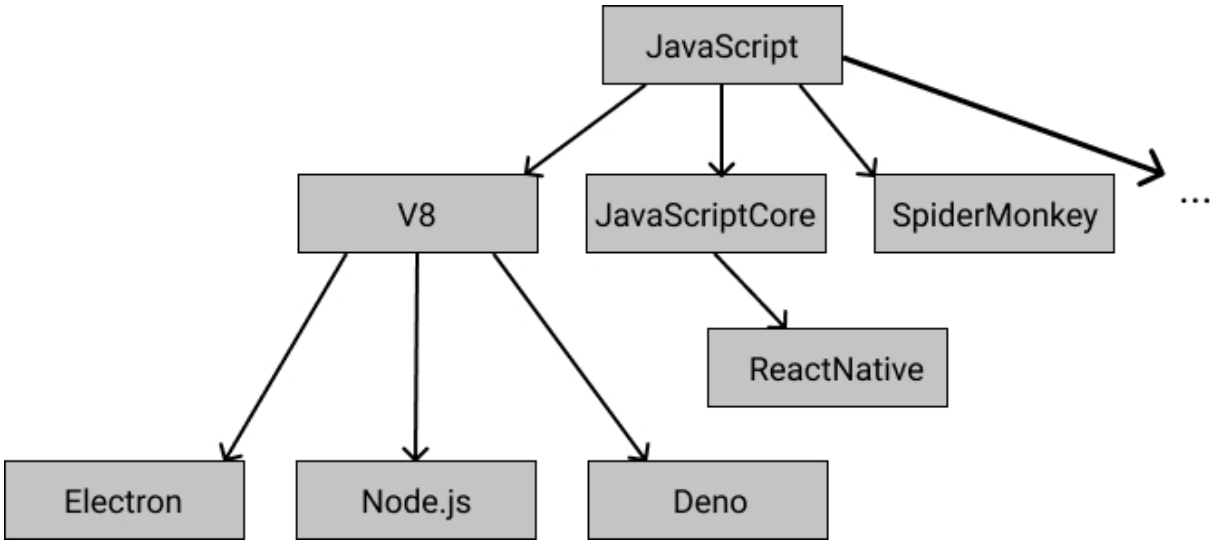


Рисунок 1.1 - Дерево найбільш розповсюджених рушіїв та середовищ, що їх використовують.

JavaScript вважається мовою програмування, що інтерпретується, проте сучасні компілятори, також звані двигунами, ще з часів першого - SpiderMonkey займалися інтерпретацією коду. Двигун V8, написаний на C++ в 2008, дозволив компілювати JavaScript, що значно збільшило продуктивність браузерів, які використовували цей компілятор. Перевага компіляції полягала в тому, що компілятор генерує машинний код, який система користувача виконує безпосередньо, тоді як інтерпретатор рядок за рядком проходив за програмою та виконував вихідний код. Оскільки компілятор працює над створенням машинного

коду, він застосовує оптимізацію. Як компіляція, так і оптимізація призводять до швидшого виконання коду, попри додатковий час, необхідний на етапі компіляції.

Сучасні рушії переслідують дві основні цілі:

– швидкий запуск програми інтерпретатора;
– швидке виконання компілятора.

Досягнення обох цілей починається з інтерпретатора. Паралельно двигун позначає часто виконувані частини коду як "Гарячий шлях" і передає їх компілятор разом з контекстною інформацією, зібраною під час виконання. Цей процес дозволяє компілятору адаптувати та оптимізувати код для поточного контексту. Крім того, V8 використовує компіляцію "на ходу", що дозволяє виконувати певний код, тільки в тому випадку, якщо він використовується.

Вбудоване кешування, є основним способом оптимізації в двигунах JavaScript. Інтерпретатор повинен виконати пошук, перш ніж зможе отримати доступ до властивості об'єкта. Ця властивість може бути частиною прототипу об'єкта, мати метод отримання або бути доступним через проксі. Пошук властивостей досить затратний за швидкістю виконання.

Механізм надає кожному об'єкту «тип», який він генерує під час виконання. V8 називає ці "типи", прихованими класами чи формами об'єктів. Щоб два об'єкти мали ту саму форму, обидва об'єкти повинні мати однакові властивості в одному порядку.

За допомогою форм об'єкта двигун знає місце у пам'яті кожної якості. Механізм жорстко кодує ці розташування у функції, що звертається до властивості, а вбудоване кешування виключає операції пошуку. Це дає величезне покращення продуктивності.

Двигун V8 написаний на C++, має відкритий вихідний код і він працює на різних операційних системах, Unix, Windows, Linux. Він постійно розвивається та підтримується. Використання C++ дозволило виділяти кілька потоків процесора під потреби.

Як він використовує внутрішні потоки:

- основний потік отримує код, компілює його і потім виконує;
- окремий потік для компіляції, так що основний потік може продовжувати виконання, поки перший оптимізує код;
- потік, що займається аналізом продуктивності, який повідомить середовище виконання, на які методи ми витрачаємо багато часу, щоб компілятор міг їх оптимізувати;
- кілька потоків для обробки збирача сміття.

Для складання сміття V8 використовує традиційний підхід, який полягає в тому, щоб помічати і потім по мітках очищати код, що не використовується. Фаза маркування повинна зупинити виконання JavaScript. Щоб контролювати витрати на складання сміття та зробити виконання стабільнішим, V8 використовує інкрементне маркування: замість обходу всієї купи, намагаючись помітити всі можливі об'єкти, він проходить лише частину купи, а потім відновлює нормальне виконання. Наступна зупинка збирача сміття буде продовжена з того місця, де зупинився попередній обхід купи. Це дозволяє робити дуже короткі паузи під час нормального виконання. Складальник сміття обробляється окремими потоками.

JavaScript - це однопоточна мова програмування, що означає, що вона має один стек викликів. Отже, він може виконувати одне процесорне завдання за один раз.

Стек викликів – це структура даних, яка здебільшого фіксує, де у програмі ми знаходимося. Якщо ми переходимо до функції, поміщаємо її у верхню частину

стека. Якщо ми повернемося з функції, то ми вискочимо з вершини стека. Це все, що може стек.

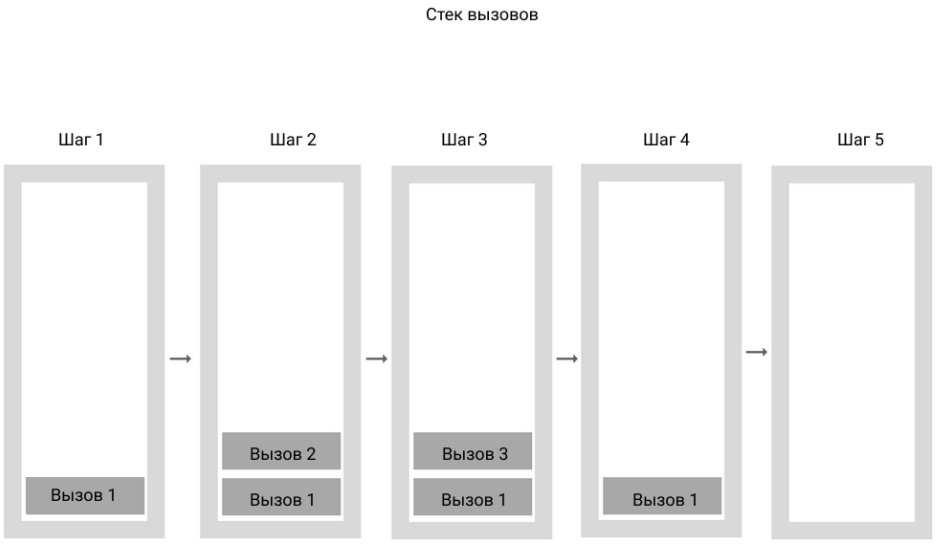


Рисунок 1.2 - схема роботи стеку викликів, що використовують рушії JavaScript.

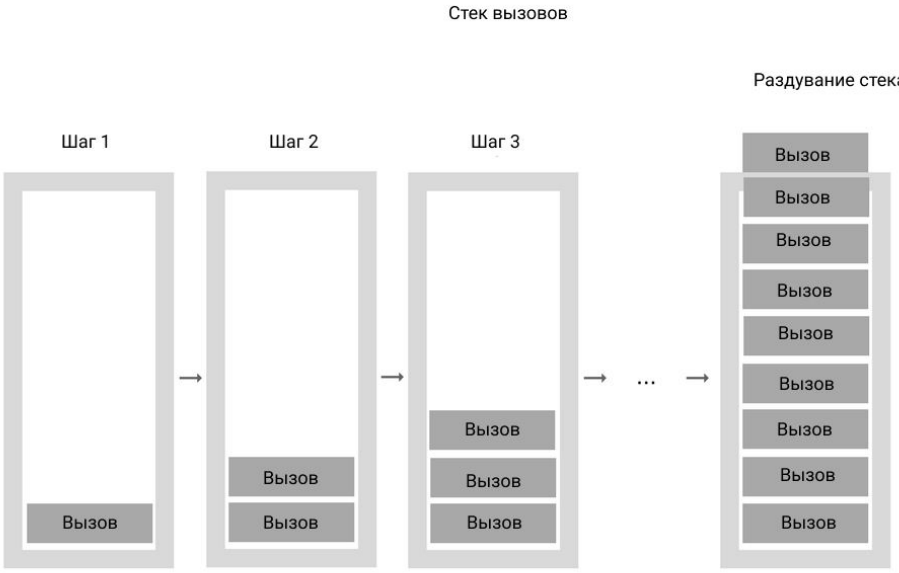


Рисунок 1.3 - схема роздування стеку у рушіях JavaScript.

"Роздування стека" - це відбувається при досягненні максимального розміру стека викликів. І це може статися досить легко, особливо, якщо ви використовуєте рекурсію без ретельного тестування коду.

Запуск коду в одному потоці може бути досить простим, оскільки вам не потрібно мати справу зі складними сценаріями, що виникають у багатопотокових середовищах, наприклад, з блокуванням. Але робота в одному потоці може стати серйозним обмеженням, якщо необхідно виконати кілька процесорних завдань поспіль.

Коли у програмі є виклики функцій у стеку викликів, обробка яких потребує дуже багато часу, наприклад, коли необхідно виконати якесь складне перетворення зображення за допомогою JavaScript в браузері. Проблема в тому, що хоча стеки викликів мають функції, які потрібно виконувати, потік, насправді, не може робити нічого іншого - він блокується. Це означає, що програма не може

виконуватися далі, вона не може запускати будь-який інший код, він просто зависає. І це створює проблеми, якщо є необхідно реалізувати додаток оптимізований, швидко працює додаток або додаток з красивими плавними інтерфейсами користувача.

І це не єдина проблема. Як тільки компілятор почне обробляти таку кількість завдань у стеку викликів, він може перестати відповідати на запити на тривалий час. І більшість браузерів вживають заходів, викликаючи помилку, запитуючи вас, чи хочете ви закрити веб-сторінку.

Для того щоб виконувати складні процесорні обчислення або повільний код, не блокуючи основний потік, в якому працює програма, необхідно використовувати асинхронні зворотні виклики, які обробляються двигуном трохи інакше. Він використовує вбудований функціонал C++, що дозволяє виділяти окремі потоки для асинхронних завдань та функцій зворотного виклику, що називається циклом подій.

Цикл подій виконує одне просте завдання – контролює стек викликів та чергу зворотних викликів. Якщо стек викликів порожній, цикл подій візьме першу подію з черги і відправить його до стек викликів, який ефективно запускає його. Функції зворотних викликів та таймаути є макрозавданнями, крім них існують мікрозавдання, також звані обіцянками. Дані мікрозавдання потрапляють в окремий цикл подій для мікрозавдань.

Макрозавдання запрограмовані таким чином, щоб компілятор міг гарантувати їхнє послідовне виконання при доступі з вашого внутрішнього двигуна. Мікрозадачі запрограмовані на дії, які повинні відбутися відразу після сценарію, що виконується в даний момент, наприклад виконання чогось асинхронного без підтримки штрафу, пов'язаного зі створенням нового макрозавдання. Ці мікрозавдання вставляються в чергу мікрозавдань, яка

обробляється після макрозавдань і наприкінці виконання кожного макрозавдання за умови, що основний потік не завантажений.

Таким чином, в ітерації циклу подій ми матимемо:

- перевірка, чи є завдання у черзі макрозавдань;
- якщо це так і це завдання виконується, дочекайтеся його завершення, перш ніж переходити до наступного кроку;
- потім запустіть всі мікрозавдання, які знаходяться в черзі мікрозавдань;

Якщо ми додаємо нові мікрозадачі під час виконання мікрозавдань, вони також виконуються.

Як наслідок цієї послідовності можна сказати, що два макрозавдання не можуть виконуватися один за одним, якщо між ними в хвості мікрозадач є елементи. Це дозволяє нам гарантувати, що такі зворотні виклики будуть асинхронними, навіть якщо мікрозавдання вже виконане.

У NodeJS, який працює на движку V8 за асинхронне виконання макрозавдань і мікрозавдань відповідає бібліотека libuv написана на C. Вона дозволяє звертатися до процесора та створювати паралельні потоки, в яких виконуються завдання із циклу подій. Це було реалізовано для розподілу навантаження з головного потоку з метою уникнення його блокування і, як наслідок, зниження швидкості обробки запитів web-сервером. Бібліотеку libuv можна налаштовувати, виділяючи допустиму кількість потоків, яка може додатково задіяти, але не більше, ніж дозволяють обчислювальні потужності, на яких запущено web-сервер.

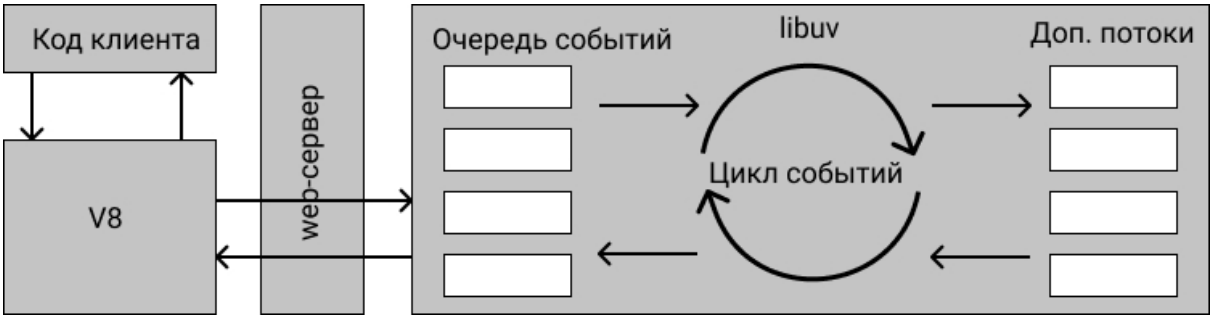


Рисунок 1.4 - схема роботи web-серверу з використанням NodeJS та бібліотеки libuv.

При проходженні циклу подій дана бібліотека вибирає події порядку черги і делегує виконання події на виділені нею потоки, у разі, якщо є вільні. Якщо вільних потоків процесора не буде, libuv чекатиме звільнення одного, на який вона зможе делегувати нову подію з циклу подій.

У NodeJS також реалізований вбудований функціонал для тестування, шифрування, хешування, відловлювання помилок, роботи з DNS, можливість використання HTTP/2, TLS/SSL, створення UDP сервера, робота з файловою системою, а також звернення до операційної системи та процесора сервера чи персонального комп'ютера. Крім даних бібліотек у NodeJS також реалізований функціонал з підключення своїх власних бібліотек написаних з використанням C або C++.

Однією з переваг використання NodeJS можна вважати можливість створення потоку трансляції даних. Його використовують для оптимізації та полегшення роботи web-сервера. Робота потоку трансляції даних полягає у послідовному читанні запису та введення.

Унікальність потоку трансляції даних полягає в тому, що замість програми, що зчитує файл в пам'ять відразу, як це робиться традиційним способом, потоки читають фрагменти даних частинами, обробляючи їх вміст, не

зберігаючи все в пам'яті. Це робить цей функціонал дійсно потужним при роботі з великими обсягами даних, наприклад, у тому випадку, якщо розмір файлу може бути більшим, ніж ваш вільний простір у пам'яті, що унеможливлює читання всього файлу в пам'ять для його обробки. Використання потоків обробки невеликих фрагментів даних дозволяє читати файли більшого розміру.

Потоки трансляції даних призначені не тільки для роботи з медіа або великими даними. Вони також дають нам можливість "компонувати" наш код. Проектування з урахуванням можливості комбінування означає, що кілька компонентів можуть бути об'єднані певним чином для отримання того самого результату. У NodeJS можна складати потужні фрагменти коду, передаючи дані в інші дрібніші фрагменти коду і з них, використовуючи трансляції.

Існує 4 види трансляцій:

- потік трансляції даних, доступний для запису, з якого ми можемо записувати дані;
- потік трансляції даних, доступний для читання, з яких можна читати дані;
- дуплексний потік трансляції даних реалізовує можливість читання і запису;
- потік трансляції даних з перетворення, за допомогою якого можна змінювати або перетворювати дані в міру їх запису та читання. Наприклад, у разі стиснення файлу, ви можете записувати стислі дані та читати розпаковані дані у файл та з файлу.

Таким чином можна сказати, що використання NodeJS для реалізації високонавантажених web-серверів має великий потенціал. Наприклад, веб-сервер для ріелторської фірми, в якій необхідно зберігати великі масиви даних з об'єктами нерухомості, а також мати можливість обробки даних масивів без великої втрати часу на виконання пошуку та вибірки.

1.3 Аналіз існуючих проблем використання програмних платформ для JavaScript.

Серед переваг використання NodeJS на web-сервері можна виділити такі:

- можливість використовувати додаткові потоки процесора для паралельної обробки кількох процесорних завдань, таких як шифрування, хешування, формування вибірки;
- можливість уникнути завантаження в оперативну пам'ять великих файлів, а також збільшена швидкість передачі даних та медіафайлів за допомогою потоків трансляції даних;
- простота і швидкість написання коду, оскільки можна використовувати одну команду для створення клієнтської частини програми, так і серверної, що дозволить заощадити додаткові фінанси компанії;
- висока популярність JavaScript, що дозволяє спільноті швидко допомагати один одному у вирішенні складних завдань.

Серед інших доступних програмних платформ для запуску JavaScript на сервері можна виділити ще одну, що стає все більш поширеною з кожним роком DenoJS.

DenoJS - це захищене середовище виконання для димнамічно типізованої мови JavaScript та статично типізованої похідної мови TypeScript, воно засноване на рушії JavaScript V8. Ця програмна платформа написана мовою програмування Rust. Крім того у DenoJS використовується Tokio, який є асинхронним середовищем виконання для Rust.

Основною метою створення DenoJS є досягнення продуктивного та захищеного середовища написання серверів, з метою забезпечення легкого досвіду для сучасних розробників. У порівнянні з NodeJS DenoJS є більш безпечний, оскільки NodeJS обробляє пакети та інші застарілі API-інтерфейси, які,

серед іншого, ніколи не зміняться. Розробники DenoJS хотіли зробити покращену версію NodeJS із сучасними інструментами JavaScript та API, сумісного із браузером та серверним середовищем.

Серед переваг DenoJS можна виділити наступні:

- Безпека, що додана у стандартний пакет DenoJS за замовчуванням. Захист працює в середовищі Sandbox, якщо це не дозволено спеціально, DenoJS не має доступу до файлів, середовища або мережі, чого немає у NodeJS. Для доступу до середовища або мережі необхідно явно додати маркери безпеки та дозволу під час виконання програми у середовищі виконання DenoJS. Якщо відповідні прапори не додані, при запуску програми буде видано помилку “PermissionsDenied”. Список прапорців безпеки: Allow-write: дозволити доступ до запису, Allow-read: дозволити доступ для читання, Allow-net: дозволити доступ до мережі, Allow-env: дозволити доступ до середовища, Allow-plugin: дозволити завантаження зовнішніх модулів, Allow-hrtime: дозволити вимір часу з високою роздільною здатністю, Allow-run: дозволити виконання підпроцесів, A: дозволити всі дозволи.
- Статично типізована похідна JavaScript TypeScript підтримується у стандартному пакеті DenoJS, оскільки у ньому присутній вбудований компілятор для TypeScript. Це дозволяє полегшити створення нових серверів, що використовують TypeScript.
- Створення єдиного файлу для виконання. DenoJS виступає у якості єдиного файлу без залежностей, що виконується. За аналогією з веб-браузером, ця програмна платформа може отримувати та імпортувати зовнішній код. У середовищі виконання DenoJS один файл може визначати складну поведінку без будь-яких інших інструментів. Враховуючи URL-адресу платформи DenoJS, вона запускається з використанням не більше ніж 15 МБ пам'яті.

– На відміну від NodeJS, у якому додаткові пакети, бібліотеки та фреймворки, що додаються за допомогою пакетного менеджеру NPM, цей пакетний менеджер додає пакети у окрему директорію, що містить залежності, а додані пакети теж можуть містити свої додаткові залежності, що робить програму, що написана на NodeJS більш повільною та важкою. DenoJS використовує децентралізовані пакети, що не використовують пакетний менеджер NPM. Це середовище виконання завантажує модулі через URL-адресу із використанням стандартного протоколу, який є сумісним із браузером. Завдяки цьому виконується імпорт модулів, з використанням цих посилань або шляхів до файлів з програми.
– DenoJS сумісний із браузером. Тобто програма написана з використанням цієї програмної платформи може бути запущена як у якості web-серверу, так і додатка у браузері. Це досягається завдяки використання стандартизованих API-інтерфейсів JavaScript для браузерів.
– DenoJS на відміну від NodeJS використовує більш сучасний та більш зручний синтаксис JavaScript EcmaScript.
– DenoJS має великий набір стандартних бібліотек, модулів і пакетів. Крім того, ця програмна платформа може імпортувати модулі з будь-якого місця в Інтернеті, наприклад з GitHub або особистого веб-сервера.
– DenoJS може використовувати HTTP ефективніше за NodeJS.

Таблиця 1.1 порівняння програмних платформ NodeJS та DenoJS

Функціонал	NodeJS	DenoJS
мова	JavaScript	JavaScript, Typescript
доставка модулів	вбудовані	через URL
захист	-	контроль доступу

рушій	V8	V8
трансляції	підтримуються	-
спільнота	велика	маленька

Однак середовище виконання JavaScript NodeJS, у свою чергу, також має перелік своїх переваг, серед них:

- NodeJS здатен масштабуватись без великих зусиль у горизонтальному та вертикальному напрямках за рахунок підключення додаткових ресурсів або написання коду більш адаптованого до циклу подій, що використовується у libuv.
- NodeJS працює у реальному часі. У разі створення web-серверів, у яких дані оновлюються дуже часто, використання NodeJS є більш ефективним за рахунок більш швидкої синхронізації. Крім того, цикл подій запобігає навантаженню HTTP з’єднання, пов'язаному з розробкою NodeJS.
- Усі асинхронні операції обробляються циклом подій, що доступний завдяки бібліотеці libuv, тому NodeJS швидко обробляє асинхронні операції.
- NodeJS легше вивчити,
- NodeJS дозволяє кешувати окремі модулі. Працює воно наступним чином, у разі запиту на початковий модуль, він кешується в пам'яті програми, що запобігає необхідності виконувати код зайвий раз.
- Використання потокової передачі даних у NodeJS дозволяє обробляти файл під час завантаження, що зменшує час його передачі.
- Корпоративна підтримка NodeJS стала дуже великою, оскільки за час його існування багато компаній з відомими на весь світ іменами почали використовувати цю програмну платформу, наприклад PayPal, Walmart, Microsoft, Yahoo та Netflix.

1.4 Мета і завдання дослідження

Метою представленої роботи є забезпечення здатності web-серверу компанії з ріелторських послуг працювати під достатньо великим навантаженням на основі аналізу результатів тестових навантажень на різні інфраструктурні рішення з оптимізації web-серверу, що будуть розглянуті у роботі.

Для досягнення мети роботи будуть розглянуті наступні питання:

- аналіз можливих варіантів оптимізації web-серверу;
- постановка цілей, що повинен вирішувати web-сервер;
- аналіз необхідних інструментів для реалізації оптимізації;
- розробка моделі комп'ютерної системи;
- дослідження розроблених рішень оптимізації

1.5 Визначення напрямків рішення поставлених завдань

Web-сервер компанії з ріелторських послуг може бути реалізований із використанням програмної платформи NodeJS. Він повинен відповідати за роботу з об'єктами нерухомості, користувачами, мати функціонал з адміністрування даних, та мати змогу, у разі потреби, додати нові функції. Web-сервер повинен справлятися з навантаженням 500 паралельних одночасних підключень.

1.6 Висновки до розділу

В розділі представлена характеристика галузі та об'єкта дослідження, розглянуто можливі засоби вирішення проблем, що у сучасний час потребує галузь. Розглянуті сильні та слабкі сторони об'єкту дослідження.

2 ТЕОРЕТИЧНИЙ РОЗДІЛ

2.1 Розрахунок комп’ютерної системи з використанням функціоналу зі створення кластерів.

Створимо класичний web-сервер, побудований на кластерах, з використовуванням вбудованого функціоналу NodeJS по створенню кластерів. У такому сервері повинна бути головна версія або майстер, та частина, що відповідає за копіювання головної версії на підпорядковану репліку сервера. Даний механізм дозволяє створити головний процес, що приймає усі запити і далі розподіляє їх між доступними копіями за допомогою алгоритму Round-robin. У нашому випадку ці копії обробляють запити на формування вибірок об’єктів нерухомості.

Кластеризація - це метод перетворення кількох web-серверів у групу серверів, що діє як єдина система. Ця група називається кластером. Вбудований функціонал з кластеризації використовується на кожному із серверів у групі. Кожен із серверів зберігає ту саму інформацію, і всі разом вони виконують адміністративні завдання, такі, як балансування навантаження, визначення збоїв вузлів та призначення функцій аварійного перемикання. Інший метод кластеризації, апаратна кластеризація, вимагає встановлення спеціалізованого обладнання на одному сервері, який управляє кластером.

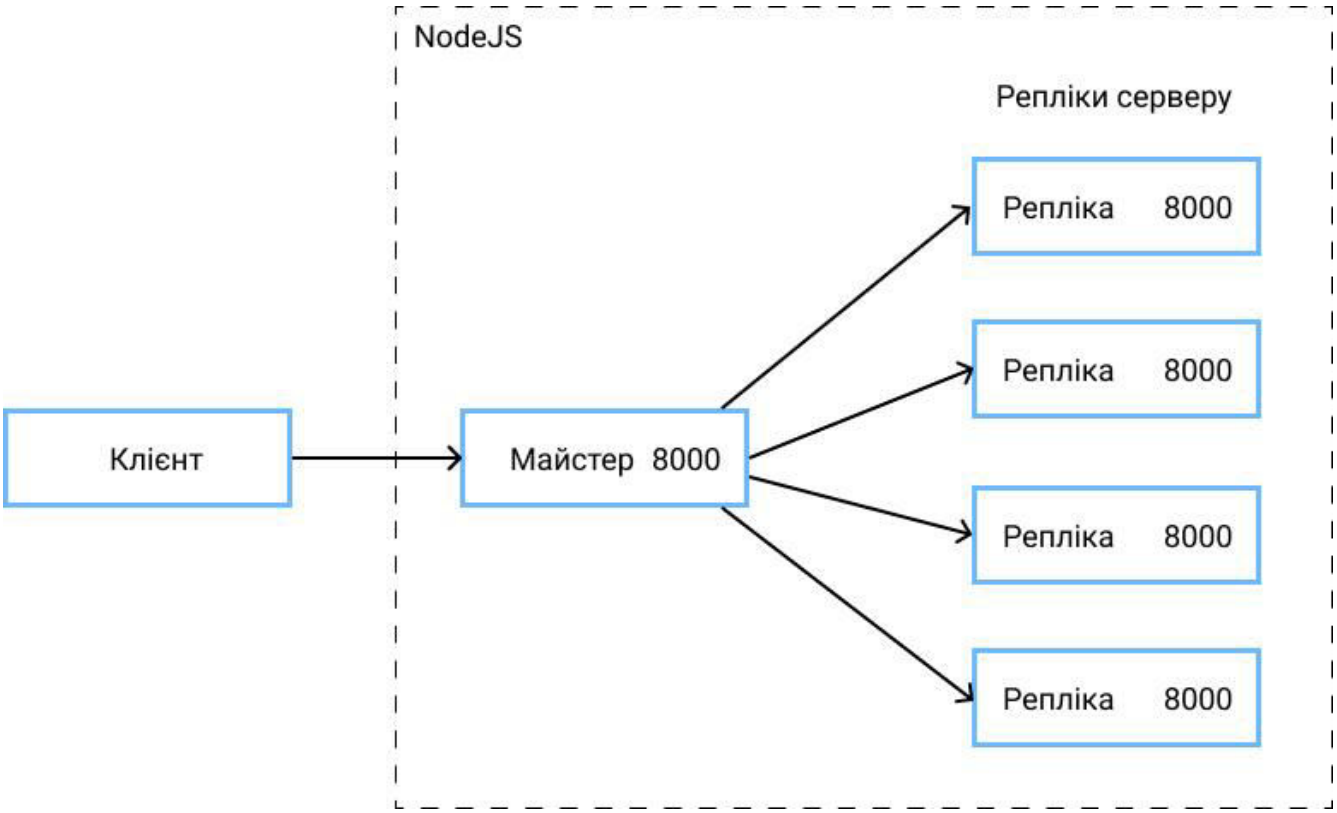


Рисунок 2.1 - архітектура web-серверу з використанням кластерів у середовищі виконання NodeJS.

Кластер web-серверів має здатність масштабуватись, оскільки сервери можуть бути легко додані або видалені з кластера в міру необхідності. Для цього не потрібно спеціалізоване обладнання, кластер, як правило, простіше і дешевше налаштовувати.

Round-robin — з усіх доступних алгоритмів розподілення навантаження найбільш поширений — він дозволяє надсилати кожен запит по черзі. Тобто у разі, якщо серверів чотири, запити направляються у порядку черги, спочатку на перший, потім на другий, після того на третій і у кінці циклу на четвертий. Наступний запит балансувальник навантаження віддасть знову першому, і так цикл буде оброблений заново. Перевагами цього алгоритму можна назвати простоту та ефективність. При цьому доступні копії серверів, що обробляють запити, можуть бути не зв’язаними між собою, через DNS цей алгоритм може надсилати запити між будь якими обчислювальними потужностями. Головна

проблема цього підходу - це нераціональне використання ресурсів. Тобто, навіть, якщо всі обчислювальні потужності однакові на всіх копіях серверу, реальне навантаження на кожній копії буде відрізнятися.

2.2 Розрахунок комп'ютерної системи з використанням менеджеру процесів PM2.

PM2 - менеджер процесів для web-серверів, що використовують NodeJS. У наш час використання PM2 вважається застарілою практикою, оскільки все частіше у розробці web-серверів використовується підхід контейнеризації, яка дозволяє більш гнучко налаштовувати інфраструктуру додатків. Цей інструмент все ще використовується для невеликих web-серверів, що виконують дуже вузьке коло задач, або додатків, що працюють на віртуальних машинах або сервера, що не використовують контейнеризацію. PM2 дуже схожий на механізм кластеризації, що вбудований у програмну платформу NodeJS, однак на відміну від цього функціоналу менеджер процесів не потребує адаптації коду під створення реплік, тобто він бере всю відповідальність за створення кластерів на себе. Це дозволяє відділити бізнеслогіку, що описана кодом від інфраструктурної частини описання запуску web-сервера, що дозволяє позбутися перенавантаження коду додатковою логікою, і, як наслідок, дозволяє полегшити розуміння кода розробниками, та розділити відповідальність за різні частини додатка між окремими членами команди розробників. PM2 також використовує для розподілу навантаження алгоритм Round-robin.

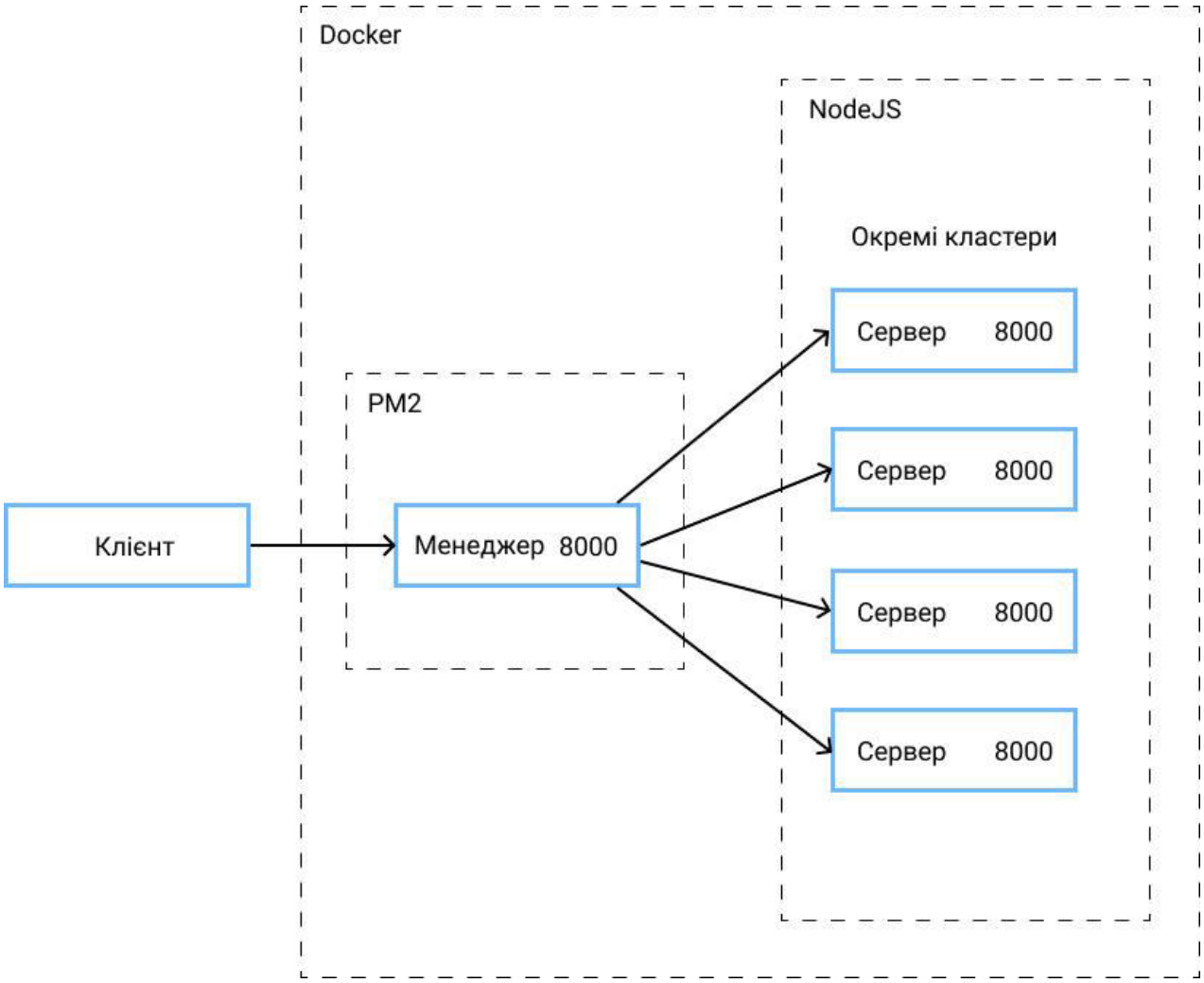


Рисунок 2.2 - архітектура web-серверу з використанням менеджеру процесів PM2.

Для використання PM2 необхідно створити окремий файл з екосистемою, що повинен називатись ecosystem.config.js або ecosystem.json.

У PM2 створені репліки будуть мати такий самий порт, що й сам менеджер процесів. Це реалізовано за допомогою внутрішнього механізму PM2, що дозволяє всередині репліки переписати дані про порт, у який згодом надсилаються запити через міжпроцесну комунікацію. Даний механізм достатньо вимогливий до процесору.

2.3 Розрахунок комп'ютерної системи з використанням функціоналу зі створення робочих потоків.

У відносно недавньому часі у програмну платформу NodeJS із версією 10.5.0 був доданий механізм зі створення робочих потоків, що виконують окремі задачі, до стабільної версії цей механізм довели у версії 12.0.0. Раніше для роботи з робочими потоками було необхідно паралельно скопіювати web-сервер у середовищі виконання NodeJS, створити у системи інше середовище виконання на NodeJS, що буде запускати інший web-сервер, та обмінюватися між ними повідомленнями через міжпроцесорну комунікацію самої системи, у якій працюють робочий поток з web-сервером.

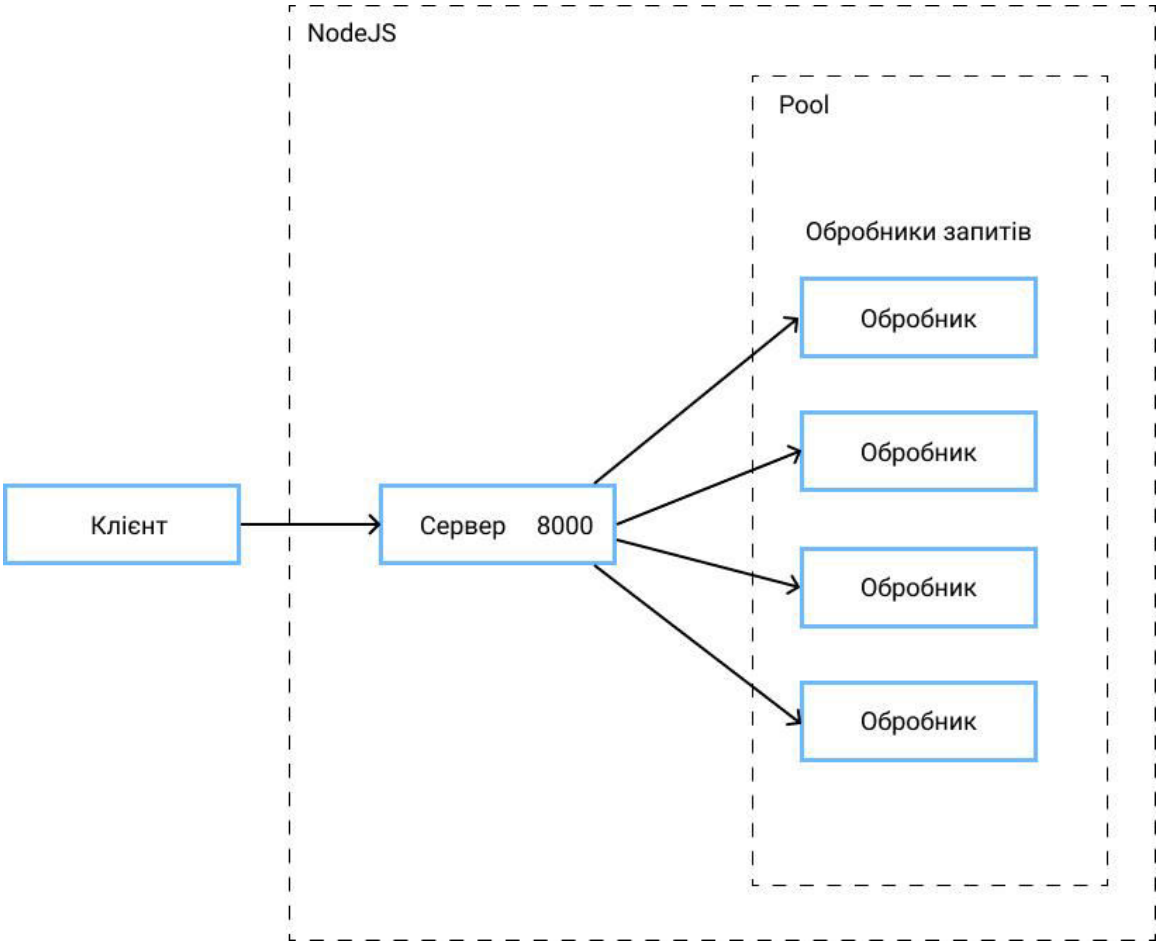


Рисунок 2.3 - архітектура web-серверу з використанням робочих потоків у середовищі виконання NodeJS.

Новий механізм, що відповідає за створення робочих потоків запускають маленьку, легковажну або обмежену копію web-сервера у середовищі виконання NodeJS, в котрих є свій ізольований рушій V8, своя окрема бібліотека libuv з індивідуальним циклом подій, і працює все ізольовано від інших потоків чи процесів (Рис 2.3).

Механізм робочих потоків дозволяє створити множину доступних потоків, що має назву множина обробників. Ці потоки з множини використовуються у разі надходження на сервер запиту, цей запит, згідно до з його цілі, сервер віддає на один з доступних з множини невеликих обробників. У головному потоці живе сервер, що відповідає за розподіл запитів. Головна різниця між двома попередніми варіантами оптимізації у тому, що у тих двох випадках у окремих потоках запускалися повноцінна важка копія web-серверу.

2.4 Розрахунок комп'ютерної системи з використанням функціонал зі створення робочих потоків і окремим зовнішнім балансувачем навантаження.

Окрім зазначеного вище прикладу web-сервера, що працює із використанням робочих потоків можна розглянути версию із використанням зовнішнього інструмента з балансування навантаження, що буде працювати на окремому порту. Це балансування буде розміщене на окремому порту, і буде, як у попередніх випадках за допомогою алгоритму розподілення навантаження розподіленої обчислювальної системи, розділяти отримані запити між репліками з множини обробників. Описані репліки будуть створені за допомогою механізму робочих потоків програмної платформи NodeJS. У цьому варіанті у множині будуть доступні невеликі обробники, що працюють у окремих робочих потоках.

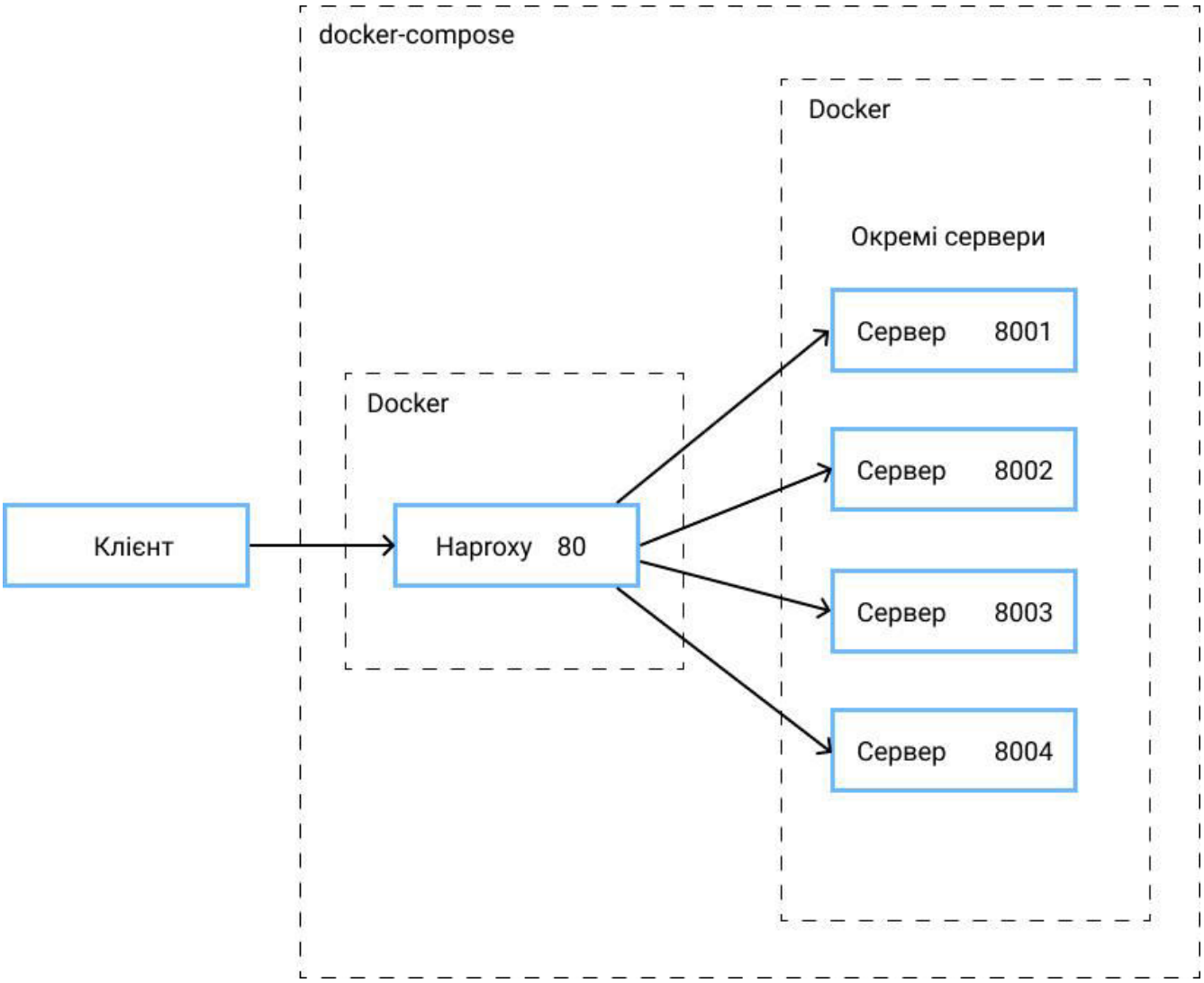


Рисунок 2.4 - архітектура web-серверу з використанням робочих потоків у середовищі виконання NodeJS та зовнішнім інструментом балансування навантаження.

Описаний варіант оптимізації дає змогу використовувати різноманітні алгоритми розподілення навантаження обчислювальної системи. Окрім найбільш поширеного Round-robin, можна виділити наступні:

– Weighted Round-robin або Static Round-robin, алгоритм схожий на звичний Round-robin, але він додатково бере до уваги динамічні ваги, що підтримуються для циклічного перебору, найменшого з'єднання та

узгодженості хешування. Це дозволяє змінювати ваги сервера на льоту з інтерфейсу командного рядка, або навіть агентом, що запущений на сервері;

- Least Connections, алгоритм в якому кожний наступний запит надсилається на web-сервер з найменшою кількістю підключень, що підтримуються. Також дуже ефективний та розповсюджений алгоритм розподілення навантаження, що є дуже ефективним та простим у реалізації. Це рішення дозволяє розподіляти навантаження на сервери з приблизно однаковими параметрами;
- Source IP hash, алгоритм, що хешує вихідну IP-адресу і ділить його на загальну вагу серверів, що працюють. Одна і та сама IP-адреса клієнта завжди досягає одного і того ж сервера, поки жоден сервер не виходить з ладу або не працює. Якщо результат хешування змінюється через зміну кількості працюючих серверів, клієнти перенаправляються на інший сервер. Цей алгоритм зазвичай використовується в режимі TCP, коли файли cookie неможливо встановити. За замовчуванням він також статичний;
- URI, алгоритм, що хешує або ліву частину URI, або весь URI і ділить хеш-значення на загальну вагу серверів, з множини доступних. Однаковий URI буде спрямовуватися завжди на один і той самий сервер до тих пір, поки хоча б один із web-серверів буде працювати або з’єднання буде відключене. Це також статичний алгоритм, який працює так само, як попередній алгоритм;
- URL Parameter, алгоритм, що можна використовувати лише на HTTP сервері, оскільки параметр URL відображається у рядку запиту кожного HTTP-запиту GET. Якщо знайдений параметр закінчується на знак рівності та значення, тоді значення хешується і поділяється на загальну вагу серверів, що працюють.

Крім можливості використання варіативних алгоритмів розподілу навантаження, цей варіант виконання дозволить створити таку кількість реплік кожного сервісу, яке буде необхідно виходячи з навантаження на кожен окремий сервіс. Наприклад, кожен окремий унікальний користувач при використанні клієнтського клієнта додатка з пошуку нерухомості у браузері на продаж або оренду буде переглядати інформацію про об'єкти нерухомості частіше, ніж інформацію про продавця та орендодавця кожного окремого об'єкта нерухомості. Крім того власник ріелторської фірми зацікавлений у тому, щоб користувач якнайшвидше отримував інформацію про масиви доступних об'єктів нерухомості, а також окремі об'єкти і вже на підставі отриманої інформації про об'єкти, користувач вже вирішував самостійно зацікавлений він у ньому чи ні. Даний фактор говорить про те, що швидкодія сервісу формування вибірок об'єктів нерухомості, а також пошуку інформації про окремі об'єкти грає також дуже важливу роль. Таким чином можна спрогнозувати, що навантаження на сервіс з формування вибірок об'єктів нерухомості, а також пошуку інформації про окремі об'єкти буде більше, ніж на сервіс пошуку інформації про користувача, а також швидкодія сервісу з формування вибірок об'єктів нерухомості, а також пошуку інформації про окремих об'єктів нерухомості об'єктах більш важливо, ніж сервісу пошуку інформації про користувача, сервісу з авторизації користувача, сервісу адміністрування програми та інших сервісів.

2.5 Висновки до розділу

У розділі розглянуті основні схеми оптимізації web-серверу компанії з ріелторських послуг, їх принцип роботи та їх відмінності. Вони полягають у:

- використанні вбудованого функціоналу програмної платформи NodeJS;
- використанні стороннього менеджеру процесів;

- використання вбудованого функціоналу програмної платформи NodeJS, що є більш новим модулем цієї програмної платформи;
- використання вбудованого функціоналу програмної платформи NodeJS, що є більш новим модулем цієї програмної платформи із використанням додаткового зовнішнього балансування навантаження.

– 3 РОЗДІЛ СИНТЕЗУ СИСТЕМИ КОНТРОЛЮ

3.1 Розробка схеми функціональної структури

Для проектування web-сервера ріелторської компанії потрібно визначити список необхідних задач, що мають бути реалізовані. Крім вже згаданого сервісу відповідального за здійснення пошуку об'єктів нерухомості та формування вибірок об'єктів нерухомості, які задовольняють вимогам пошуку, виділимо наступні:

- сервіс з пошуку інформації про користувачів, будь то користувач, який знаходиться в пошуку необхідного об'єкта нерухомості для купівлі або оренди, так і користувача, який є ріелтором, кожного окремого об'єкта нерухомості, який зацікавить клієнта;
- сервіс з управління правами користувачів, оскільки веб-додаток буде використовуватися як клієнтами, для пошуку об'єктів нерухомості, так і ріелторами для додавання нових об'єктів і редагування вже доданих виключно ним же. Крім того, необхідні користувачі з правами адміністратора, які зможуть виправити помилкові дані будь-якого з користувачів у будь-який момент;
- сервіс з авторизації користувачів, який відповідає за знаходження користувачів у системі, за допомогою якого перевірятиметься наявність, а також правильність використовуваних даних користувача для входу;
- обслуговування для пошуку оптимального маршруту до необхідного клієнту об'єкта нерухомості;
- сервіс для формування рейтингів для визначення найбільш затребуваних та найбільш популярних об'єктів нерухомості, а також для просування менш популярних.

Оскільки в веб-додатку компанії з надання ріелторських послуг також передбачається наявність користувачів з правами адміністратора, необхідно реалізувати додаткові сервіси по здійсненню управління додатком, крім управління користувачами, об'єктами нерухомості та редагуванням рейтингів об'єктів нерухомості:

– сервіс для створення, видалення та редагування фільтрів пошуку об'єктів нерухомості для знаходження найбільш оптимальних та придатних для клієнта;
– сервіс управління та контролю статусів платежів за орендовані чи придбані об'єкти нерухомості;
– сервіс для редагування окремих частин клієнта програми самими адміністраторами комп'ютерної системи підприємства з надання ріелторських послуг без залучення розробників.

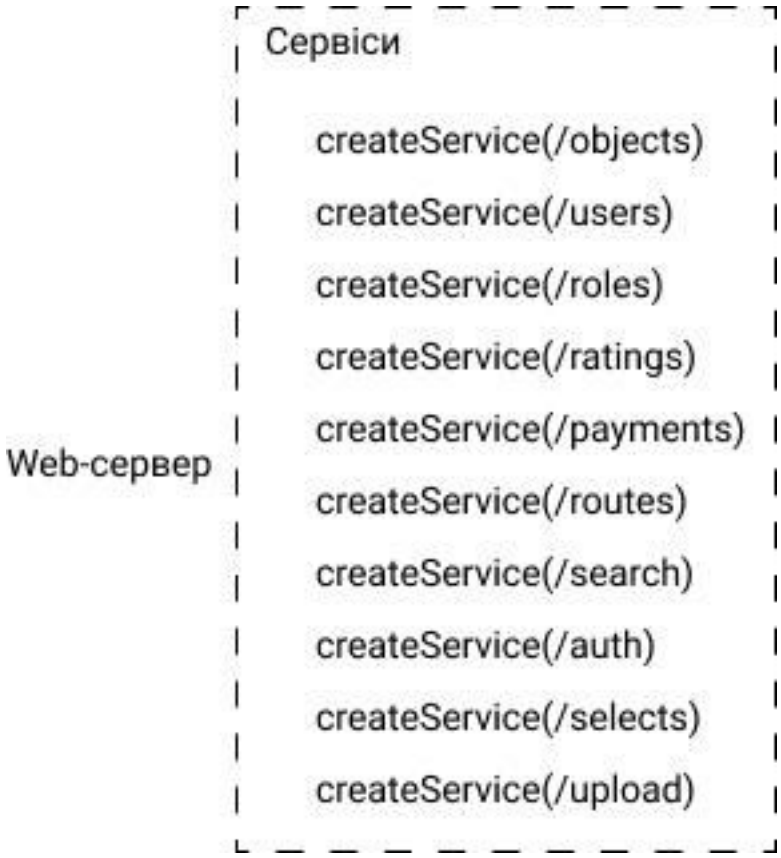


Рисунок 3.1 - перелік сервісів, що необхідні для реалізації web-сервера компанії з ріелторських послуг.

Склавши список всього функціоналу, який доступний для здійснення за допомогою використання веб-сервера компанії з надання ріелторських послуг, можна проаналізувати, які сервіси веб-сервера будуть використовуватися найчастіше, а які будуть рідше.

Оскільки в системі кількість користувачів з правами адміністратора менша, ніж кількість користувачів з правами клієнта та ріелтора, то навантаження на сервіси, призначені виключно для адміністраторів, буде найменшим. Сервіс для пошуку маршруту також використовується рідше за інших, оскільки клієнт захоче оглянути об'єкт тільки після того, як сам перегляне список доступних об'єктів нерухомості, що відповідають його вимогам, і вибере той, який йому найбільше підходить. Швидкість роботи сервісу з формування рейтингів на підставі переваг користувачів також не використовується часто, оскільки рейтинги оновлюються раз на кілька годин, і відповідно сервіс запускатиметься і оновлюватиме рейтинги з встановленою періодичністю. Сервіс з авторизації користувача також не відчуває великого навантаження, оскільки користувач входить в систему один раз, після чого залишається в ній на кілька днів, доки не вийде із системи самостійно або закінчиться термін дії його сесії. Сервіс з перевірки прав користувача відчуває таке ж навантаження, що, як і сервіс з авторизації користувачів, оскільки використовується щоразу, коли користувач авторизується.

3.2 Аналіз використаних інструментів

Для запуску web-серверу та змоги його тестування для визначення ефективності розглянутих методів оптимізації використовуємо контейнер, в якому необхідно зафіксувати кількість доступних потоків процесору, що будуть виділені для обчислення, у нашому випадку для усіх методів оптимізації оберемо однакову

кількість. Однак середовище виконання NodeJS не зможе визначити дане обмеження, що було встановлене інструментом контейнеризації Docker, тому воно буде визначати повністю всі доступні потоки на сервері, для цього необхідно власноруч обмежити максимальну кількість потоків для використання web-сервером.

Контейнери – це абстракція на рівні програми, яка поєднує код та залежності. Декілька контейнерів можуть працювати на одному сервері і спільно використовувати ядро операційної системи з іншими контейнерами, кожен з яких працює як ізольовані процеси в просторі користувача. Контейнери займають менше місця, ніж віртуальні машини, можуть обробляти більше додатків та вимагають менше віртуальних машин та операційних систем.

Використання контейнерів, за рахунок забезпечення ними контейнеризації коду з його усіма залежностями, дозволяє швидко та надійно переміщати його з однієї обчислювальної системи у іншу. Наприклад, це дозволяє зробити локальну розробку на персональному комп’ютері найбільш наближену до середовище, що використовується на сервері, на якому працює робоча версія програмного забезпечення. Контейнери мають образи, що є автономно виконуємим пакетом програмного забезпечення. Образ контейнера має у своєму складі усе необхідне для запуску програмного забезпечення:

- код;
- налаштоване середовище виконання;
- встановлені інструменти;
- системні бібліотеки;
- налаштунки програмного забезпечення.

Під час виконання образів контейнерів вони стають контейнерами. Ці контейнери можуть працювати як на системах Windows, так і Linux та Unix.

Контейнери ізолюють програмне забезпечення від його середовища та гарантують, що воно працює однаково, незважаючи на відмінності.

На відміну від віртуальної машини контейнер містить не всю операційну систему, а тільки її додатки, залежності і конфігурацію. Це робить контейнери Docker набагато легшим і швидшим, ніж звичайні віртуальні машини.

Розробка web-серверу із використанням NodeJS, що працює у середовищі контейнера пов’язана із дотриманням переліку практик, що мають характер рекомендації, але їх дотримання значно полегшить розробку програмного забезпечення, покращить швидкість його виконання у робочому середовищі, що використовується клієнтами та зробить контейнер більш безпечним та легким.

Образ для NodeJS у середовищі Docker досить великий і включає сотні вразливостей безпеки різного типу і ступеня серйозності. Оскільки поняття контейнеру має на увазі використання невеликих образів, оскільки це призведе до зменшення розміру програмного забезпечення на образі Docker, що зменшить потенційні вразливості, та прискорить процес створення образу. Крім того можна використовувати дайджест чи статичний хеш образу Docker з метою отримання детермінованих збірок базового образу. Виходячи з цього, необхідно використовувати невеликий образ NodeJS із версією, що доступна для використання та має довгострокову підтримку (LTS) для того щоб зменшити розмір контейнеру.

package.json

```
{
  "name": "charly",
  "description": "",
  "version": "1.0.0",
  "private": true,
  "main": "server",
  "author": {
    "email": "abc@gmail.com"
  },
  "bugs": {},
  "directories": {
    "lib": "server/",
    "test": "test/",
    "config": "config/"
  },
  "engines": {
    "node": "^14.0.0",
    "npm": ">= 3.0.0"
  },
  "scripts": {
    "test": "npm run lint && npm run jest",
    "build": "npx next build",
    "dev": "NODE_ENV=development && node server",
  },
  "dependencies": {
    "helmet": "^4.3.1",
    "lodash": "^4.17.20",
    "sequelize": "^6.3.5",
    "sequelize-cli": "^6.2.0",
  },
  "devDependencies": {
    "git-cz": "^3.2.1",
    "husky": "^3.0.5",
    "jest": "^26.6.3",
  },
}
```

Рисунок 3.2 - структура package.json файлу web-сервера на NodeJS.

Встановлювати лише ті залежності, що використовуються та необхідні для повноцінного функціонування web-серверу у production середовищі, яке

використовують звичайні користувачі сервісу в образі Docker для Node.js. Команда для встановлення залежностей, пакетів і бібліотек що використовуються як для production середовища так і середовища для розробки виглядає наступним чином: npm install. Для запобігання використання пакетів і залежностей, що необхідні для розробки програмного забезпечення у production версії web-серверу, що відповідають списку значень за ключем devDependencies (Рис. 3.2) необхідно використовувати наступну директиву: npm ci --only=production. Вона запобігає непотрібному ризику безпеці, що може бути порушена через пакети, що використовуються як залежність для розробки, а також надмірно завищує розмір образу.

Під час створення образа Docker для додатка реалізованого за допомогою NodeJS для production версії необхідно переконатися, що всі платформи та бібліотеки використовують оптимальні налаштування продуктивності та безпеки. Для того, щоб почати використовувати таку версію додатка необхідно додати наступну директиву у Dockerfile:

ENV NODE_ENV production

Деякі пакети, фреймворки, модулі та бібліотеки можуть включати оптимізовану конфігурацію або збірку, що працює у середовищі production. Тому для використання такої конфігурації необхідно присвоїти наступній змінній оточення NODE_ENV відповідне значення production.

Варто уникати використання контейнерів зі правами root. Контейнери мають слідувати принцип найменших привілеїв, що вважається давнім засобом контролю безпеки додатків та системи з перших днів існування Unix систем. Під час створення web-додатків на програмній платформі NodeJS слід дотримуватися цього правила під час запуску контейнерів. Це правило існує для того щоб оцінити чи зловмисник може скомпрометувати web-додаток за допомогою засобу, що

дозволяє вводити та виконувати команди, що будуть викликані з користувачем, якому належить процес додатка. Якщо цей процес має права доступу до всіх елементів системи, зловмисники можуть використовувати майже всі ресурси в контейнері, у тому числі спробу виходу з контейнера або підвищення привілеїв іншим користувачам.

Використовувати події, з метою безпечного завершення роботи web-додатка на програмній платформі NodeJS у контейнері Docker. Одна з найбільш поширених помилок пов'язана зі способом, за допомогою якого викликається головний процес. Наступні варіанти вважаються не підходящими для запуску процесу, і їх слід уникати:

- CMD "npm" "start"
- CMD ["node", "start"]
- CMD "node" "server.js"
- CMD "start-app.sh"

Інструментам оркестрації контейнерів, таким як Docker Swarm, Kubernetes або навіть самому движку Docker, потрібен засіб за допомогою якого надсилаються сигнали процесу в контейнері. У більшості випадків такі сигнали свідчать про завершення роботи програми: SIGTERM та SIGKILL. Процес може виконуватися паралельно, і у такому разі, немає впевненості у тому, що цей процес отримає ці сигнали.

Існує два способи вказівки директиви CMD у Dockerfile:

- нотація shell, у якій контейнер створює новий інтерпретатор оболонки, який виступає у ролі середовища для процесу. У такому випадку оболонка може неправильно пересилати сигнали вашому процесу;
- нотація execform, яка безпосередньо породжує процес, не вміщуючи його в оболонку. Він вказується за допомогою запису масиву, наприклад: CMD

[“npm”, “start”]. У такому випадку будь-які сигнали, надіслані в контейнер, надсилаються безпосередньо процесу.

Виходячи з цього для покращення виконання процесу Dockerfile запишемо директиву наступним чином:

CMD [“вузол”, “server.js”]

У такому випадку процес вузла нова оболонка не створюється, що гарантує, що цей процес отримає всі надіслані йому сигнали, без того, щоб він був огорнутий інтерпретатором оболонки.

Правильне завершення web-додатків на програмній платформі NodeJS. У разі отримання сигналів процесу, що завершують роботу додатків, необхідно, закривати всі процеси, створені за допомогою контейнеру правильним чином, не заважаючи користувачам.

Коли додаток NodeJS отримує сигнал переривання SIGINT, це провокує швидке завершення процесу, якщо, які-небудь обробники подій не налаштовані таким чином, щоб мати змогу обробляти його в іншому режимі. Це означає, що відкриті підключення до web-серверу, будуть негайно закриті, тому необхідно подбати про те, щоб такі з’єднання залишалися відкритими у разі раптового завершення процесу у середовищі виконання NodeJS.

Використання багатоетапних збірок контейнерів є хорошим засобом для переходження від простого, але потенційно неправильного Dockerfile до окремих етапів побудови контейнеру, з метою уникнення витоку конфіденційної інформації. Для цього є можливість використання більшого базового образу Docker для встановлення необхідних пакетів і модулів, та за необхідністю скомпілювати власні пакети npm, а потім скопіювати всі ці артефакти у невеликий образ.

Уникання непотрібних файлів у образах web-додатків на програмній платформі NodeJS. Docker має .dockerignore файл, який дозволяє уникати відправлення будь-яких файлів, що вказані у ньому демону Docker. Ось список файлів, щоб дати вам уявлення про те, що ви можете помістити у свій образ Docker, чого ми в ідеалі хотіли б уникати. До файлу .dockerignore варто додавати наступні види файлів:

- копії директорії із встановленими модулями node_modules;
- файли зі змінними оточення .env, aws.json;
- файли конфігурації локального середовища, оскільки це може призвести до того, що кеш образів стає недійсним.

Монтування секретів у збірку контейнерів, у разі необхідності конкретної конфігурації. Docker має функціонал, який називають секретами, цей механізм підходить для випадку, монтування секретів, наприклад коли потрібен .npmrc файл. Коли ми збираємо контейнер, необхідно вказати аргументи командного рядка, які визначають новий ідентифікатор секрету та посилаються на файл як джерело секрету. Потім необхідно додати прапори в директиву RUN для встановлення npm, яка монтує файл, на який вказує секретний ідентифікатор, в цільове розташування файлу .npmrc локального каталогу, в якому ми хочемо, щоб він був доступний. Після чого файл .npmrc монтується як секретний та ніколи не копіюється в образ Docker. У кінці необхідно додати файл.

Створений контейнер із робочою версією web-сервером на програмній платформі NodeJS необхідно запустити на комп’ютерній системі, що буде доступна максимальний час, оскільки необхідно щоб додатком користувалися клієнти з будь-якої точки країни. Для таких цілей необхідно використовувати сервіси, що дозволяють використовувати віддалені обчислювальні потужності, такі як, наприклад Amazon Elastic Compute Cloud (Amazon EC2)

Amazon EC2 дозволяє користувачам створювати додатки для автоматизації масштабування відповідно до мінливих потреб та періодів пікового навантаження, а також спрощує розгортання віртуальних серверів та управління сховищем, зменшуючи потребу в інвестиціях в обладнання та допомагаючи оптимізувати процеси розробки.

Налаштування EC2 включає створення образу машини Amazon (AMI), який включає операційну систему, додатки та конфігурації. Цей AMI завантажується в Amazon Simple Storage Service (S3) і реєструється в EC2, після чого користувачі можуть запускати віртуальні машини за необхідності. Серед переваг використання інстансів EC2 можна виділити наступні:

- великий вибір стандартних налаштувань та обчислювальних потужностей, вибрати підходящий інстанс можна за наступними змінними: розмір пам'яті, ЦП та розмір завантажувального розділу оптимізовані для вибраної ОС;
- EC2 може інтегруватися з великим різноманіттям інших сервісів, що представляє у використання компанія Amazon;
- наявність змоги адміністративного доступу до списку екземплярів інстансів, через які користувачі мають змогу зупиняти та запускати екземпляри, та можуть отримувати до історії записів у екземплярі EC2;
- функціонал який дозволяє розділяти та налаштовувати екземпляри у EC2 за тим чи вони залишаються закритими, чи доступні в Інтернеті. EC2 для реалізації цього механізму використовує Amazon Virtual Private Cloud (VPC), крім того підприємства можуть підключати свою захищену IT-інфраструктуру до ресурсів VPC.
- серед багатьох варіантів ціноутворення EC2 пропонує доступні погодинні ставки.

У Amazon EC2 є багато різноманітних варіації інстансів , що можуть задовольнити будь які потреби розглянемо перелік цих інстансів:

– екземпляри загального призначення, що забезпечують обчислювальні потужності у балансі, серед них пам'ять і мережеві ресурси і можуть бути використані для великого різноманіття загальних цілей. Вони підходять для web-серверів, віртуальних сховищ, тестування, розробки, розгортання баз даних, зберігання кешу та запуску контейнеризованих додатків. Існує перелік типи екземплярів цієї групи, що представлені у Табл. 3.1:

Табл 3.1 доступні екземпляри загального призначення з доступними їх конфігураціями

Тип	Потоків (к-ть)	Пам’ять (Гб)	Сховище	Пропускна спроможність мережі (Гбіт/с)	Пропускна спроможність EBS (Мбіт/с)
Mac	12	32	EBS	10	8000
T4g	2 - 8	0.5 - 32	EBS	5	2085 - 2780
T3	2 - 8	0.5 - 32	EBS	5	
T3a	2 - 8	0.5 - 32	EBS	5	
T2	1 - 8	0.5 - 32	EBS	невелика	
M6g	1 - 64	4 - 256	EBS / SSD	10 - 25	4750 - 19000
M6i	2 - 128	8 - 512	EBS	12.5 - 50	10000 - 40000
M6a	2 - 192	8 - 768	EBS	12.5 - 50	6600 - 40000

M5	2 - 96	8 - 384	EBS / SSD	10 - 25	4750 - 19000
M5a	2 - 96	8 - 384	EBS / SSD	10 - 20	2880 - 13570
M5n	2 - 96	8 - 384	EBS / SSD	25 - 100	4750 - 19000
M5zn	2 - 48	8 - 192	EBS	25 - 100	3170 - 19000
M4	2 - 64	8 - 256	EBS	до 25	450 - 10000
A1	1 - 16	2 - 32	EBS	10	

– екземпляри, що створені для обчислень, вони підходять для додатків, що створені для реалізації великих обчислень. У них використовуються процесори, що мають великий обчислювальний ресурс. Цілі які покривають дані екземпляри: високопродуктивні web-сервери, наукове моделювання, машинне навчання, обробка аналітики, високопродуктивні обчислення, кодування відео, ігри через мережу, моделювання. Існує перелік типи екземплярів цієї групи, що представлені у Табл. 3.2;

Табл 3.2 доступні екземпляри призначених для важких обчислень з доступними їх конфігураціями

Тип	Потоків (к-ть)	Пам’ять (Гб)	Сховище	Пропускна спроможність мережі (Гбіт/с)	Пропускна спроможність EBS (Мбіт/с)
C6i	2- 128	4 - 256	EBS	12.5 - 50	10 - 40
C6g	1 - 64	2 - 128	EBS / SSD	10 - 25	4750 - 19000

C6gn	1 - 64	2 - 128	EBS	25 - 100	9500 - 38000
C5	2 - 96	4 - 192	EBS / SSD	10 - 25	4750 - 19000
C5a	2 - 96	4 - 192	EBS / SSD	10 - 20	3170 - 9500
C5n	2 - 72	5.25 - 192	EBS	25 - 100	4750 - 19000
C4	2 - 36	3.75 - 60	EBS	до 10	500 - 4000

– екземпляри, що призначені для забезпечення високої продуктивності робочих навантажень, за допомоги виділення великого об’єму оперативної пам’яті, вони використовуються коли необхідно зберігати великі набори даних у пам'яті, наприклад: статичний часовий аналіз, моделювання, розгортання баз даних та кешів, зберігання аналітики. Існує перелік типи екземплярів цієї групи, що представлені у Табл. 3.3;

Табл 3.3 доступні екземпляри призначених для важких обчислень з доступними їх конфігураціями

Тип	Потоків (к-ть)	Пам’ять (Гб)	Сховище	Пропускна спроможність мережі (Гбіт/с)	Пропускна спроможність EBS (Мбіт/с)
R6g	1 - 64	8 - 512	EBS / SSD	10 - 25	4750 - 19000
R6i	2 - 128	16 - 1024	EBS	12.5 - 50	10000 - 40000
R5	2 - 96	16 -768	EBS / SSD	10 - 25	4750 - 19000

R5a	2 - 96	16 - 768	EBS / SSD	10 - 20	2880 - 13570
R5b	2 - 96	16 - 768	EBS	10 - 25	10000 - 60000
R5n	2 - 96	16 - 768	EBS / SSD	25 - 100	4750 - 19000
R4	2 - 64	15.25 - 488	EBS	10 - 25	
X2gd	1 - 64	16 - 1024	SSD	10 - 25	4750 - 19000
X2idn	64 - 128	1024 - 2048	SSD	50 - 100	40000 - 80000
X2iedn	4 - 128	128 - 4096	SSD	25 - 100	20000 - 80000
X2iezn	8 - 48	256 - 1536	EBS	25 - 100	3170 - 19000
X1e	4 - 8	122 - 3904	SSD	10 - 25	500 - 14000
X1	64 - 128	976 - 1952	SSD	10 - 25	7000 - 14000
HM	224 - 448	6144 - 24576	EBS	100	38000
z1d	2 - 48	16 - 384	SSD	10 - 25	

– екземпляри для використання у прискорених обчисленнях базуються на апаратних прискорювачах. Вони задовольняють наступному переліку цілей: обчислення чисел з плаваючою комою, фінансові обчислення, сейсмічний аналіз, обробка графіки або зіставлення шаблонів даних, розпізнавання голосу, машинне навчання або симуляції. Існує перелік типи екземплярів цієї групи, що представлені у Табл. 3.4;

Табл 3.4 доступні екземпляри призначених для прискорених обчислень з доступними їх конфігураціями

Тип	Потоків (к-ть)	Пам’ять (Гб)	Сховище	Пропускна спроможність мережі (Гбіт/с)	Пропускна спроможність EBS (Мбіт/с)
P4	96	1152	SSD	400	19000
P3	8 - 96	61 - 768	EBS / SSD	10 - 100	1500 - 19000
P2	4 - 64	61 - 732	EBS	10 - 25	
DL1	96	768	SSD	400	19000
Inf1	4 - 96	8 -192	EBS	25 - 100	4750 - 19000
G5	4 - 192	16 - 768	SSD	10 - 100	3500 - 19000
G5g	4 - 64	8 - 128	EBS	10 - 25	3500 - 19000
G4dn	4 - 96	16 - 384		25 - 100	
G4ad	4 - 256	16 - 256	EBS	10 - 25	3000 - 6000
G3	4 -64	30.5 - 488		10 - 25	
F1	8 - 64	122 - 976	SSD	10 - 25	
VT1	12 - 96	24 - 192	EBS	3.125 - 25	4750 - 25000

– екземпляри, що створені для здійснення функції сховища, вони призначені для тих випадків, коли потрібен високий послідовний доступ для читання та запису великого масиву даних у локальному сховищі. Вони реалізовані

таким чином, що здатні виконувати десятків тисяч довільних операцій введення-виводу. Існує перелік типи екземплярів цієї групи, що представлені у Табл. 3.5.

Табл 3.5 доступні екземпляри у ролі сховища з доступними їх конфігураціями

Тип	Потоків (к-ть)	Пам’ять (Гб)	Сховище	Пропускна спроможність мережі (Гбіт/с)	Пропускна спроможність EBS (Мбіт/с)
Im4gn	2 - 64	8 - 256	SSD	25 - 100	9500 - 38000
Is5gen	1 - 32	6 - 192	SSD	25 - 50	9500 - 19000
I4i	2 - 128	16 - 1024	SSD	10 - 75	10000 - 40000
I3	2 - 72	15.25 - 512	SSD	10 - 25	
I3en	2 - 96	16 - 768	SSD	25 - 100	
D2	4 - 36	30.5 - 244	HDD	до 10	
D3	4 - 32	32 - 256	HDD	15 - 25	850 - 5000
D3en	4 - 48	16 - 192	HDD	25 - 75	
H1	8 - 64	32 - 256	HDD	10 - 25	

Контейнеризований web-сервер компанії з ріелторських послуг, що запускається на екземплярі EC2, і на прикладі якого розглядаються засоби оптимізації NodeJS додатків необхідно протестувати на ефективність. Для цього використаємо Autocannon, що є інструментом для тестування HTTP/1.1, що

реалізований за допомогою NodeJS, з підтримкою конвеєрної обробки HTTP та HTTPS.

Autocannon має багато опцій, які дозволяють налаштовувати тестування web-серверу будь-яким необхідним чином. Наприклад: дозволяє налаштовувати кількість одночасних підключень, що використовується; кількість конвеєрних запитів, що використовуються; затримка запуску тестування; мінімальну кількість запитів, які потрібно зробити; кількість робочих потоків, які використовуються для запуску запитів; HTTP-метод, що використовується; максимальну кількість часу до очікування відповіді, у секундах; тіло запиту; кількість помилок для закінчення тестування; Максимальна кількість запитів, що спрямовується на одне підключення до сервера; заголовки HTTP-запиту; максимальна загальна кількість запитів до сервера; максимальна кількість запитів на секунду; мінімальну кількість запитів, яку необхідно зробити для кінця тестування; показувати коди стану підключення.ґ

Крім того цей інструмент має змогу задавати шлях до сокету домену Unix або Windows; час підготовки для початку тестування, щоб усі необхідні процеси запустилися, а зайві були закриті; виводити результат у формі таблиці або налаштованого JSON об’єкту, що виводиться до консолі, або записується у окремий файл з історією. Autocannon має змогу працювати без перерви для динамічного отримання стану web-серверу, аналізувати статус та тіло відповідей з web-сервера та порівнювати їх з бажаними, записуючи результати до історії.

Крім того цей інструмент має приємний функціонал зі встановлення розширення Server Name Indication (SNI) для захищеного TLS з’єднання, у разі якщо це необхідно.

Один з варіантів оптимізації, що базується на використанні зовнішнього інструмента, що реалізує механізм кластеризації web-сервера у середовищі

виконання NodeJS ззовні сервера та без ускладнення бізнес логіки кода, використовує інструмент PM2. Цей інструмент дуже зручний у використанні, оскільки має декілька додатків. Він дозволяє відстежувати статуси кластерів (Рис. 3.3), що запущені у даний час, ресурси, що вони використовують, конфігурувати кожний з кластерів окремо. Також цей інструмент дає змогу більш детально керувати кластерами окремо та відслідковувати їх статуси (Рис. 3.4).

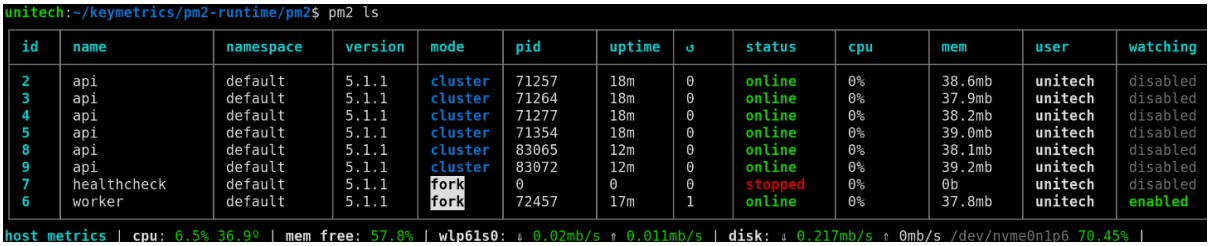


Рисунок 3.3 - Панель моніторингу кластерів у PM2.

PM2 має додатковий інструмент для більш детального та зручного моніторингу своїх додатків, що запущені за допомогою цього інструмента PM2 Plus, що працює у браузері. PM2 Plus представляє собою розширений інструмент моніторингу та діагностики запущених кластерів у режимі реального часу. Він надає функції як для захисту вашого поточного PM2, так і для моніторингу програм, що працюють на серверах. Він включає в себе відстеження проблем і виключень, звіти про розгортання, журнали в реальному часі, інтеграцію з електронною поштою і месенджерами, такими як Slack, моніторинг метрик і центр дій, що настроюється.

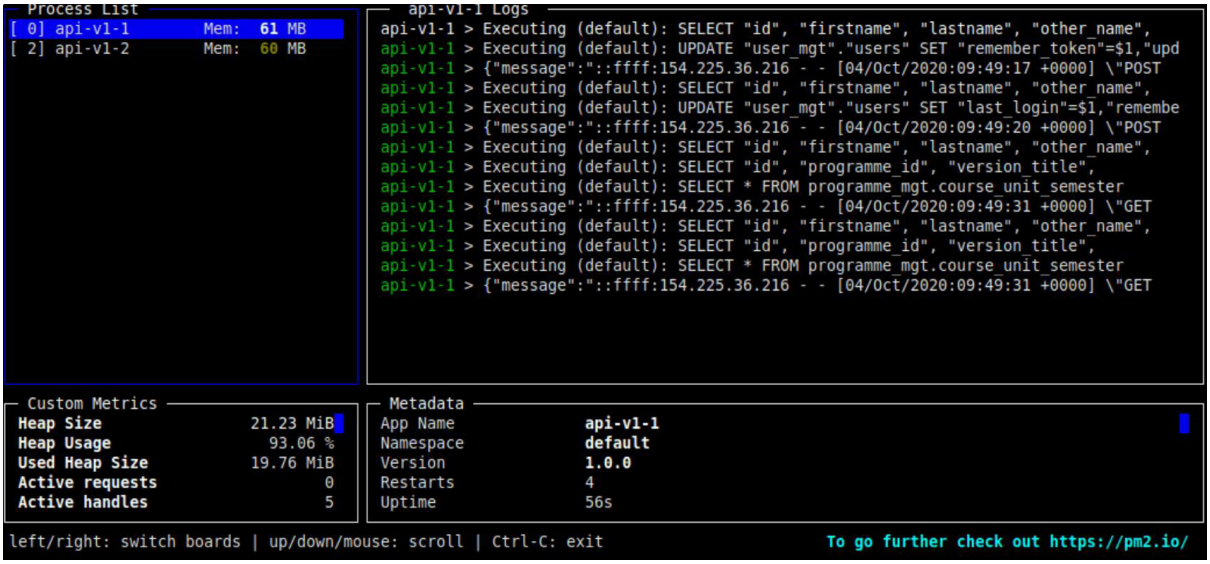


Рисунок 3.4 - Панель перегляду історії записів та статусу окремих кластерів, що створені за допомогою PM2.

Інструмент PM2 має велике різноманіття налаштунків, їх ключі, тип значення та логічний зміст вказано у наступному списку.

Загальні:

- name: (String) назва програми чи кластера;
- script: (String) шлях до скрипту який запускає web-сервер;
- cwd: (String) каталог, з якого буде запущено web-сервер;
- args: (String) усі аргументи, що необхідно передати до оточення web-серверу без використання командного рядка;
- interpreter: (String) шлях інтерпретатора;
- interpreter_args: (String) опції запуску, що передаються до інтерпретатора;
- exec_interpreter: (String) команда що запускає інтерпретатор.

Розширені налаштунки:

- instances: (Number) кількість екземплярів web-сервера, що запускаються;

- exec_mode: (String) режим для запуску, може мати значення "cluster" або "fork";
- watch: (Boolean або Array) опція для перегляду змін у екземплярі і перезавантаження кластеру у разі отримання змін;
- ignore_watch: (Array) список регулярних виразів, що вказують на імена файлів або директорій. що ігноруються у разі використання попередньої команди;
- max_memory_restart: (String) опція що вказує обсяг пам'яті, при досягненні якого кластер буде перезавантажено;
- env: (Object) змінні оточення env, що необхідні для роботи кластера;
- instance_var: (string) змінна оточення що вказує на ідентифікатор інстанса;
- treekill: (Boolean) опція, що відповідає за припинення дочірнього процесу.

Конфігурація записів у журнал історії:

- log_date_format: (String) формат дати у журналі;
- error_file: (String) шлях до файлу, у якому зберігаються помилки;
- out_file: (String) шлях до файлу журнала
- combine_logs: (Boolean) прибирає ідентифікатор процесу з записів у журналі;
- merge_logs: (Boolean) псевдонім для combine_logs;
- pid_file: (String) шлях до файлу pid.

Налаштунки роботи кластерів:

- min_uptime: (String) мінімальний час роботи кластера;
- listen_timeout: (Number) час до примусового перезавантаження у мілісекундах;

- kill_timeout: (Number) час у мілісекундах перед відправкою останнього сигналу SIGKILL;
- wait_ready: (Boolean) опція, що вказує на перезавантаження у разі отримання події process.send(‘ready’);
- max_restarts: (Number) кількість перезапусків кластеру до того як він буде вважатися помилковим і більше не буде перезапускатися;
- restart_delay: (Number) час очікування перед перезавантаженням;
- autorestart: (Boolean) вказує на те, чи буде кластер перезапущений у разі, якщо він вийшов з ладу, або виконався;
- vision: (Boolean) вказує PM2 на те чи запускатися з функцією Vision, що відповідає за метадані керування версіями;
- force: (Boolean) дозволяє запускати один і той самий скрипт кілька разів.

4 РОЗДІЛ РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Призначення і область застосування програми

Комп’ютерна система призначена для управління та зберігання даних об’єктів нерухомості. Для зберігання клієнтської бази з метою аналізу потреб клієнтів та аналітики їх цілей. Управлінням бази ріелторів, що працюють на компанії з надання ріелторських послуг з метою оцінки ефективності роботи кожного з працівників.

Для виконання аналізу потреб клієнтів з метою надання найбільш актуальної та підходящої інформації щодо об’єктів нерухомості, які можуть зацікавити клієнтів. Для формування рейтингів найбільш популярних та найбільш вигідних об’єктів нерухомості, з метою їх реалізації.

4.2 Обґрунтування технічних характеристик програми

4.2.1 Постановка завдання на розробку програми

Програмне забезпечення моделі комп’ютерної системи компанії з надання ріелторських послуг дозволить ефективно реалізувати процес продажу або здачі в оренду приміщень шляхом економії часу працівників, орієнтацією на зацікавлення клієнтів за найменший відрізок часу.

Реалізація комп’ютерної системи починається з аналізу потреб компанії з надання ріелторських послуг та формування списку необхідних сервісів, що потребує ця компанія. На основі цього списку формування переліку програмних сервісів, що розділені за логічним призначенням та метою реалізації.

4.2.2 Опис алгоритму і функціонування програми

Для забезпечення роботи комп’ютерної системи компанії з ріелторських послуг, за метою збільшити клієнтську базу та аналізу потреб клієнтів розроблено програму.

Программа завантажується на віртуальний сервер EC2 від компанії Amazon Web Services.

Встановлення необхідних інструментів для повної безпомилкової роботи програми у середовищі віртуального серверу. Встановлення переліку необхідних бібліотек та фреймворків у середовищі контейнеру, що був створений за допомогою програмного забезпечення Docker, без котрих робота web-серверу не буде здійснюватися, з метою подальшого запуску, у середовищі цього контейнеру web-серверу компанії з надання ріелторських послуг.

Після запуску проводиться перевірка навантаженням за допомогою інструмента Autocannon і здатність обробляти запити, та формувати відповіді за найменший час.

4.2.3 Опис і обґрунтування вибору методу організації вхідних та вихідних даних

В роботі розроблений перелік архітектурних варіантів запуску web-серверу компанії з надання ріелторських послуг з метою обрання найбільш швидкодіючого.

Для реалізації web-серверу була обрана програмна платформа Node.Js що використовує рушій V8 для інтерпретації та подальшої компіляції JavaScript.

Вхідними даними виступають архітектурні рішення з їх програмною та інфраструктурною реалізацією для запуску web-серверу. Вихідними даними

виступають метрики швидкості обробки запитів та формування відповідей web-сервером.

4.2.4 Опис і обґрунтування вибору та складу технічних і програмних засобів

У роботі для реалізації web-серверу компанії з надання ріелторських послуг обрано віртуальний сервер EC2 зі встановленим на ньому Linux дистрибутиву Ubuntu 20.04.

Середовище розробки web-серверу Node.Js 16.13.1 що дозволяє мову JavaScript, що інтерпретується та компілюється лише у середовищі браузера у середовищі операційної системи за допомогою рушія V8.

Web-сервер запускається і працює у середовищі контейнера, у якому встановлені всі необхідні бібліотеки та фреймворки для роботи web-серверу.

У роботі розглянуто чотири архітектурні рішення, котрі дозволяють запускати web-сервер паралельно за допомогою вбудованих інструментів Node.Js, сторонніх процесних менеджерів та балансувачів навантаження.

4.3 Опис розробленої програми

4.3.1 Загальні відомості

Описуваний web-сервер є текстової реалізацією моделі комп’ютерної системи, що використовуються компанією з ріелторських послуг.

Web-сервер реалізований на мові JavaScript.

Web-сервер запускається і виконується в середовищі контейнера створеного за допомогою програмного забезпечення Docker, із використанням програмної платформи Node.Js.

Web-сервер призначений для управління, зберігання та обробки даних об’єктів нерухомості, зберігання клієнтської бази, зберігання бази працівників, тобто ріелторів, що працюють у компанії, формування рейтингів об’єктів на базі їх популярності та аналіз уподобань клієнтів.

4.3.2 Функціональне призначення

Web-сервер є функціонально завершеним.

Web-сервер призначений для виконання потреб компанії з ріелторських послуг, та запускається і виконується за допомогою програмної платформи Node.JS. Web-сервер забезпечує реалізацію наступних функцій системи:

- зміну параметрів джерела повідомлень;
- зміну налаштувань серверу обробки повідомлень;
- формування потоку повідомлень при моделюванні;
- збереження результатів експериментів.

4.3.3 Опис логічної структури

Логічна структура програми представлена у вигляді моделі.

Опис роботи програми

Web-сервер запускається згідно до одного з чотирьох варіантів, що були розроблені з метою оптимізації швидкості обробки запитів. Оточення для запуску і роботи у ньому web-серверу підібрано таким чином, щоб виділені обчислювальні потужності серверу були однаковими для всіх схем оптимізації та таким чином, та було найбільш підходящим для всіх засобів оптимізації. Доступні обчислювальні потужності процессору web-сервера обмежені за допомогою інструмента контейнеризації Docker. Після запуску web-серверу у контейнері виконується тестування здатності web-серверу обробляти критичну кількість запитів та

здатність формувати відповіді за максимальний допустимий відрізок часу за допомогою інструмента Autocannon.

Серед вибраних варіантів оптимізації були вибрані наступні:

- з використання вбудованого функціоналу кластеризації у програмній платформі Node.JS;
- з використанням стороннього функціоналу управління процесами PM2;
- з використанням вбудованого функціоналу запуску обробників у окремих потоках, що доступний у програмній платформі Node.JS;
- з використанням зовнішнього балансувальника навантаження у комбінації з вбудованим функціоналом запуску обробників у окремих потоках, що доступний у програмній платформі Node.JS;

4.3.4 Використані технічні засоби

Для роботи web-серверу використовувався сервер з наступними технічними характеристиками: процесор 4.8 ГГц, пропускна здатність якого була зменшена для тестових запусків web-серверу до 5 потоків; 16 Гбайт оперативної пам’яті; операційна система Linux із дистрибутивом Ubuntu 20.04; інструмент контейнеризації Docker; програмна платформа Node.JS; інструмент для проведення тестування навантаження Autocannon.

4.3.5 Вхідні дані

У якості вхідних даних виступають наступні web-сервери, що працюють під тестовим навантаженням п’ятсот паралельних з’єднань протягом двадцяти секунд:

- web-сервер, що працює з використання вбудованого функціоналу кластеризації у програмній платформі Node.JS;

– web-сервер, що працює з використанням стороннього функціоналу управління процесами PM2;
– web-сервер, що працює з використанням вбудованого функціоналу запуску обробників у окремих потоках, що доступний у програмній платформі Node.JS;
– web-сервер, що працює з використанням зовнішнього балансувальника навантаження у комбінації з вбудованим функціоналом запуску обробників у окремих потоках, що доступний у програмній платформі Node.JS;

4.3.6 Вихідні дані

У якості вихідних даних виступають дві таблиці: одна для затримки запиту, а інша для обсягу запиту:

– у таблиці затримки вказано час запиту з відсотком;
– у таблиці обсягу запитів зазначено кількість надісланих запитів та кількість завантажених байтів.

4.4 Очікувані техніко-економічні показники

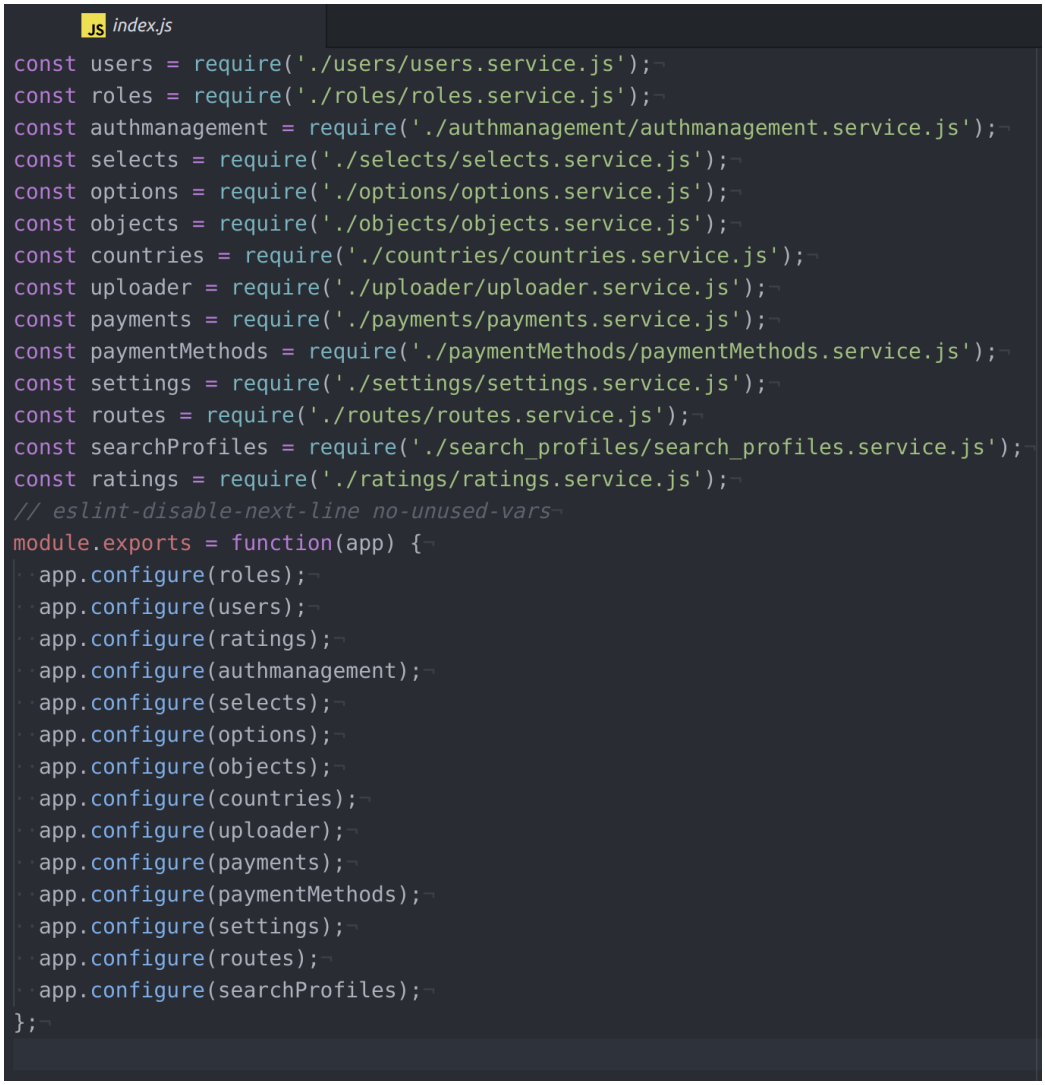
Проведення запусків web-серверу з різними засобами оптимізації дозволить на основі аналізу результатів здатності web-серверу обробляти запити та формувати відповіді під час високого навантаження на функціонал з формування вибірок об’єктів нерухомості та пошуку інформації про окремі об’єкти нерухомості обрати найбільш підходящий для використання у робочій версії web-серверу компанії з ріелторських послуг.

З економічної точки зору, розроблений web-сервер дозволить збільшити клієнтську базу компанії з ріелторських послуг, витрачати менше часу на роботу з клієнтом та пошук підходящого варіанту об’єкта нерухомості, що задовольняє потреби клієнта, зменшити кількість ріелторів, що працюють у компанії.

5 ЕКСПЕРИМЕНТАЛЬНИЙ РОЗДІЛ

5.1 Розрахунок та моделювання комп’ютерної системи з використанням функціонала кластеризації.

Web-сервер, що використовується компанією з ріелторських послуг має у своєму складі перелік різноманітних сервісів, що відповідають за повноцінне функціонування програми. На рисунку (Рис. 5.1) зображений повний перелік сервісів, що відповідають за роботу web-сервера з об’єктами нерухомості, користувачами та їх правами доступу, рейтингами, налаштуванням фільтрів пошуку об’єктів нерухомості, авторизацією користувачів, доступними країнами, платіжних системами та платежами, що були створені з їх допомогою та складанням маршруту на карті.



index.js

```
const users = require('./users/users.service.js');
const roles = require('./roles/roles.service.js');
const authmanagement = require('./authmanagement/authmanagement.service.js');
const selects = require('./selects/selects.service.js');
const options = require('./options/options.service.js');
const objects = require('./objects/objects.service.js');
const countries = require('./countries/countries.service.js');
const uploader = require('./uploader/uploader.service.js');
const payments = require('./payments/payments.service.js');
const paymentMethods = require('./paymentMethods/paymentMethods.service.js');
const settings = require('./settings/settings.service.js');
const routes = require('./routes/routes.service.js');
const searchProfiles = require('./search_profiles/search_profiles.service.js');
const ratings = require('./ratings/ratings.service.js');
// eslint-disable-next-line no-unused-vars
module.exports = function(app) {
  app.configure(roles);
  app.configure(users);
  app.configure(ratings);
  app.configure(authmanagement);
  app.configure(selects);
  app.configure(options);
  app.configure(objects);
  app.configure(countries);
  app.configure(uploader);
  app.configure(payments);
  app.configure(paymentMethods);
  app.configure(settings);
  app.configure(routes);
  app.configure(searchProfiles);
};
```

Рисунок 5.1 - налаштування повного переліку сервісів, що входять у склад web-серверу компанії з ріелторських послуг.

Для створення кожного окремого сервіса необхідно реалізувати шаблон, що буде описувати кожний сервіс згідно до моделі даних, яка включає в себе налаштування схеми даних окремих таблиць бази даних, що зберігається у змінній “Model”, та етапів, обробки що проходить запит перед формуванням відповідного запиту у базу даних, та етапів обробки відповіді з бази даних з подальшою формування відповіддю web-сервера. За ці етапи відповідає змінна “hooks”, повний код створення сервісу представлений на рисунку (Рис. 5.2).

coresService.js

```
const createService = require('feathers-sequelize');
const assign = require('lodash/assign');

module.exports = (options, app, hooks, mixin = {}) => {
  const { name, Model } = options;
  let service = {};

  if (Model) {
    service = createService(options);
  }
  app.use(name, assign(service, mixin));

  return assign(app.service(name).hooks(hooks), {
    appliedHooks: hooks,
  });
};
```

Рисунок 5.2 - код створення сервісів, що забезпечують роботу web-сервера компанії з ріелторських послуг

Для запуску web-сервера з використанням функціоналу відповідального за створення кластерів, що надається програмною платформою NodeJS необхідно додати робочу версію програми до екземпляру Amazon EC2. Зробити це можна досить легко, за допомогою системи контролю версій Git, він дозволяє створити репозиторій із кодом, що буде доступний усім, або у разі використання приватних репозиторієв лише обмеженому колу користувачів. Для цього необхідно згенерувати пару ключів, приватний та публічний, приватний залишиться на екземплярі, а публічний додати до налаштунків. Після цього необхідно додати користувача, що має згенеровану пару ключів до групи користувачів, що мають доступ до репозиторія з кодом та за допомогою команди git clone завантажити

проект на екземпляр Amazon EC2 для подальшого здійснення перевірки здатності працювати під необхідним навантаженням.

Після того як репозиторій був завантажений на віддалений інстанс Amazon EC2 необхідно запустити docker-compose із конфігурацією, що описана у docker-compose.yml файлі (Рис. 5.3).

docker-compose.yml

```yaml
version: '2.4'
services:
  server:
    build:
      context: ./server
      dockerfile: ./Dockerfile
    environment:
      NODE_ENV: production
    image: cluster_server
    container_name: cluster_server
    restart: always
    ports:
      - "80:8000"
    cpus: '5'
```

Рисунок 5.3 - docker-compose.yml файл із конфігурацією web-сервера із використанням кластерів.

Даний контейнер з web-сервером використовує оптимізоване оточення production, працює встановлює оптимізовану та стабільну версію програмної платформи NodeJS node:12-alpine, та працює на 8000 порті і ретранслюється на стандартний порт для HTTP протоколу по замовчуванням 80 (Рис. 5.4).

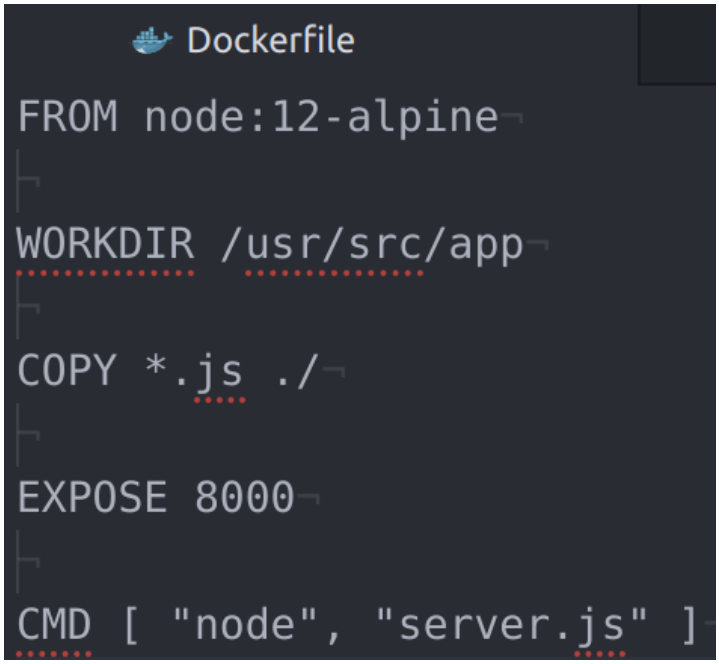


Рисунок 5.4 - Dockerfile web-сервера, що працює із використанням функціоналу кластерів.

```
Creating network "cluster_default" with the default driver
Building server
Sending build context to Docker daemon  6.144kB
Step 1/5 : FROM node:12-alpine
12-alpine: Pulling from library/node
97518928ae5f: Pull complete
58ab2943ea3a: Pull complete
da5d3df7401d: Pull complete
7384cfecbf77: Pull complete
Digest: sha256:8fad09b7620b2bc715cbba92e3313c64a797e453f560118576f1740a44584d5d
Status: Downloaded newer image for node:12-alpine
 ---> 106bb94759ad
Step 2/5 : WORKDIR /usr/src/app
 ---> Running in 12b60e66616f
Removing intermediate container 12b60e66616f
 ---> e6ecd7d38222
Step 3/5 : COPY *.js ./
 ---> eea8411ded2c
Step 4/5 : EXPOSE 8000
 ---> Running in 4305e4930cd1
Removing intermediate container 4305e4930cd1
 ---> dd319e2edc24
Step 5/5 : CMD [ "node", "server.js" ]
 ---> Running in c9de5c204a10
Removing intermediate container c9de5c204a10
 ---> d85338eb7fd8
Successfully built d85338eb7fd8
Successfully tagged cluster_server:latest
```

Рисунок 5.5 - Шаги збирання контейнеру web-сервера з використанням кластерів.

Після запуску команду docker-compose up -d контейнер починає збиратися та згідно до описаних шагів у Dockerfile (Рис. 5.5) проходити один за одним (Рис. 5.5).

Після того як контейнер буде зібраний у ньому запуститься web-сервер. У ньому будуть працювати всі функції та сервіси, що були описані у третьому розділі, та які забезпечують функціонування додатка. Найбільш важливий сервіс, що повинен витримувати найбільше навантаження - це сервіс відповідальний за роботу з масивами об’єктів нерухомості, тому проведемо перевірку оптимізації на прикладі цього сервісу. Коли процес буде працювати, відтворимо перевірку навантаженням за допомогою інструмента Autocannon. Для цього використаємо команду:

autocannon -c 500 -d 20 http://charly.de/objects

Ця команда відкриває п'ятсот паралельний з’єднань із сервером, що будуть відкритими протягом двадцяти секунд на адресу http://charly.de/objects. Сервіс, що відповідає за роботу з об’єктами нерухомості розташований за шляхом /objects.

```
Running 20s test @ http://charly.de/objects
500 connections
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	94 ms	355 ms	494 ms	2627 ms	382.95 ms	613.79 ms	9999 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	474	474	525	543	521.4	19.45	474
Bytes/Sec	59.7 kB	59.7 kB	66.2 kB	68.5 kB	65.7 kB	2.46 kB	59.7 kB

```
Req/Bytes counts sampled once per second.

11k requests in 20.08s, 1.31 MB read
477 errors (477 timeouts)
```

Рисунок 5.6 - Вихідні результати, таблиця затримок (зверху) та таблиця обсягу запитів (знизу).

Проаналізувавши результати (Рис. 5.6), можна зробити висновки, що інструмент Autocannon використовував середню кількість запитів у секунду, що дорівнює 521.4 та за 20 секунд зробив 11 тисяч запитів. Маємо 477 помилок, з яких усі були через таймаути, що вказують на те, що 477 запитів чекали відповіді більше 10 секунд, а середня затримка відповіді 382.95 мілісекунд. Більш детальна таблиця затримок (Рис. 5.7), що описує відсоток навантаження на сервер та відповідну йому затримку у мілісекундах.

Req/Bytes counts sampled once per second.

Percentile	Latency (ms)
0.001	156
0.01	156
0.1	180
1	316
2.5	627
10	825
25	860
50	895
75	936
90	960
97.5	1003
99	1051
99.9	7249
99.99	9300
99.999	9575

11k requests in 20.07s, 1.36 MB read
5 errors (5 timeouts)

Рисунок 5.7 - детальна таблиця затримок, згенерована на базі результатів тестування навантаження.

5.2 Розрахунок та моделювання комп’ютерної системи з використанням функціонала менеджеру процесів PM2.

За аналогією з прикладом, що використовував механізм кластеризації, репозиторій web-серверу із використанням менеджеру процесів PM2 завантажемо

на екземпляр Amazon EC2 за допомогою системи контролю версій Git. Після цього запустимо docker-compose up -d, що дозволить демонізувати контейнер із web-сервером. Після запуску цієї команди контейнер шаг за шагом (Рис. 5.11) буде зібраний та готовий для використання

Рисунок 5.8 - docker-compose.yml файл із конфігурацією web-сервера із використанням менеджеру процесів PM2.

Даний контейнер з web-сервером використовує оптимізоване оточення production, працює встановлює оптимізовану та стабільну версію програмної платформи NodeJS node:12-alpine, та працює на 8000 порті і ретранслюється на стандартний порт для HTTP протоколу по замовчуванням 80 (Рис. 5.10). Цей контейнер важить більше, ніж у прикладі з кластеризацією, оскільки у ньому додатково глобально встановлюється інструмент PM2. Для коректної роботи

менеджеру процесів необхідно додати файл, у якому буде сконфігурована екосистема, у якій буде працювати web-сервер, для цього у файл ecosystem.config.js додамо конфігурацію, вказану на рисунку (Рис. 5.9)

ecosystem.config.js

```
module.exports = [{
    script: 'server.js',
    name: 'app',
    exec_mode: 'cluster',
    instances: 4,
    env_production: {
        NODE_ENV: 'production'
    },
    env_development: {
        NODE_ENV: 'development'
    },
    treekill: true,
    watch: false,
    source_map_support: true,
    log_date_format: 'YYYY-MM-DD HH:mm Z',
    combine_logs: true,
    log_type: 'json',
    listen_timeout: 100000,
    kill_timeout: 5000,
    wait_ready: false,
    restart_delay: 4000,
    autorestart: true,
}]
```

Рисунок 5.9 - Файл конфігурації екосистеми менеджеру процесів PM2.

Dockerfile

```
FROM node:12-alpine

WORKDIR /usr/src/app

COPY *.js ./

EXPOSE 8000

RUN npm install pm2 -g

CMD ["pm2-runtime", "ecosystem.config.js"]
```

Рисунок 5.10 - Dockerfile web-сервера, що працює із використанням менеджеру процесів PM2.

```
Creating network "pm2_default" with the default driver
Building server
Sending build context to Docker daemon  6.656kB
Step 1/6 : FROM node:12-alpine
 ---> 106bb94759ad
Step 2/6 : WORKDIR /usr/src/app
 ---> Using cache
 ---> e6ecd7d38222
Step 3/6 : COPY *.js ./
 ---> 2a7c04747082
Step 4/6 : EXPOSE 8000
 ---> Running in 370344cec7dd
Removing intermediate container 370344cec7dd
 ---> 7220a0a7ee5b
Step 5/6 : RUN npm install pm2 -g
 ---> Running in f4ac7272024e
```

```
+ pm2@5.1.2
added 180 packages from 200 contributors in 18.039s
Removing intermediate container f4ac7272024e
 ---> 6e2658b01bb2
Step 6/6 : CMD ["pm2-runtime", "ecosystem.config.js"]
 ---> Running in b6c2b15771cc
Removing intermediate container b6c2b15771cc
 ---> 2093a21e0bc2
Successfully built 2093a21e0bc2
Successfully tagged pm2_server:latest
WARNING: Image for service server was built because it did not already exist.
pose up --build`.
Creating pm2_server ... done
```

Рисунок 5.11 - Шаги збирання контейнеру web-сервера з використанням менеджеру процесів PM2.

Після того як контейнер буде зібраний у ньому запуститься web-сервер. У цьому випадку ми також розглянемо здатність витримувати навантаження сервісу, відповідального за об’єкти нерухомості. За допомогою інструмента Autocannon використаємо таку саму команду з таким самим навантаженням:

autocannon -c 500 -d 20 http://charly.de/objects

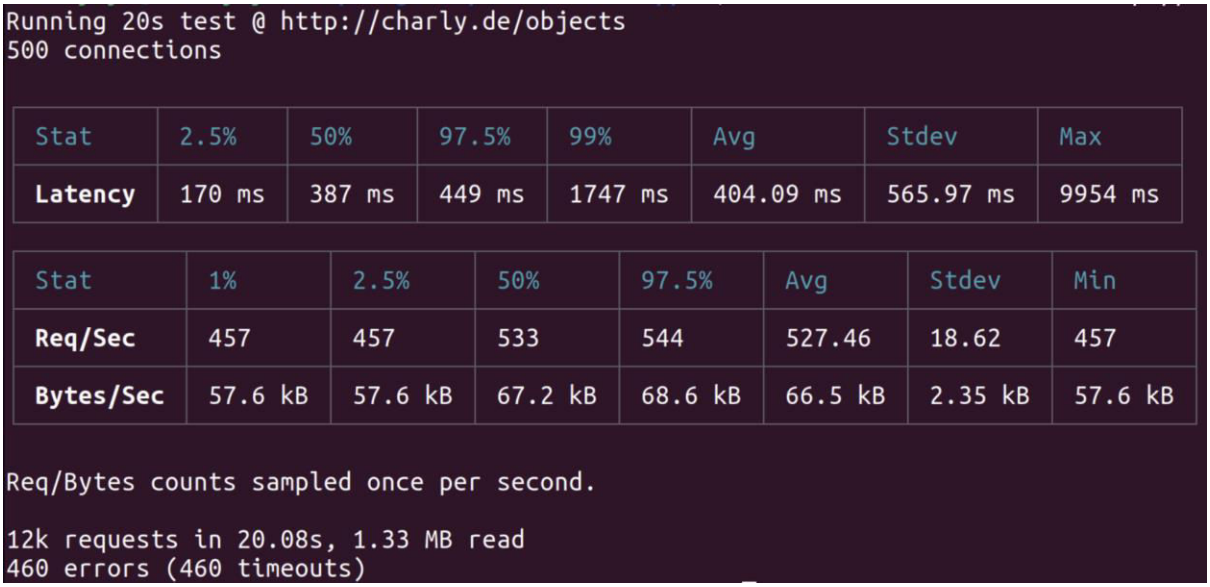


Рисунок 5.12 - Вихідні результати, таблиця затримок (зверху) та таблиця обсягу запитів (знизу).

Проаналізувавши результати (Рис. 5.12), можна зробити висновки, що інструмент Autocannon використовував середню кількість запитів у секунду, що дорівнює 527.46 та за 20 секунд зробив 12 тисяч запитів. Маємо 460 помилок, з яких усі були через таймаути, що вказують на те, що 460 запитів чекали відповіді більше 10 секунд, а середня затримка відповіді 404.09 мілісекунд. Більш детальна таблиця затримок (Рис. 5.13), що описує відсоток навантаження на сервер та відповідну йому затримку у мілісекундах.

Req/Bytes counts sampled once per second.

Percentile	Latency (ms)
0.001	154
0.01	154
0.1	181
1	344
2.5	632
10	812
25	864
50	898
75	962
90	1006
97.5	1062
99	1088
99.9	1097
99.99	1937
99.999	2055

11k requests in 20.07s, 1.37 MB read

Рисунок 5.13 - детальна таблиця затримок, згенерована на базі результатів тестування навантаження.

5.3 Розрахунок та моделювання комп'ютерної системи з використанням функціонала робочих потоків.

Використання робочих потоків має схожі етапи запуску, репозиторій web-серверу із використанням робочих потоків завантажемо на екземпляр Amazon EC2 за допомогою системи контролю версій Git. Після цього запустимо docker-

compose up -d. Після запуску цієї команди контейнер згідно до шагів (Рис. 5.16) буде зібраний та готовий для використання

docker-compose.yml

```yaml
version: '2.4'
services:
  server:
    build:
      context: ./server
      dockerfile: ./Dockerfile
    environment:
      NODE_ENV: production
    image: worker_thread_no_balancer_server
    container_name: worker_thread_no_balancer_server
    restart: always
    ports:
      - "80:8000"
    cpus: '5'
```

Рисунок 5.14 - docker-compose.yml файл із конфігурацією web-сервера із використанням робочих потоків.

Даний контейнер з web-сервером використовує оптимізоване оточення production, працює встановлює оптимізовану та стабільну версію програмної платформи NodeJS node:12-alpine, та працює на 8000 порті і ретранслюється на стандартний порт для HTTP протоколу по замовчуванням 80 (Рис. 5.14).

Dockerfile

```
FROM node:12-alpine

WORKDIR /usr/src/app

COPY *.js ./

COPY package*.json ./

RUN npm install

EXPOSE 8000

CMD [ "node", "server.js" ]
```

Рисунок 5.15 - Dockerfile web-сервера, що працює із використанням робочих потоків.

```
volodymyr@volodymyrPC:~/Projects/cluster-test/worker_thread_no_balancer$ docker-compose up -d
Creating network "worker_thread_no_balancer_default" with the default driver
Building server
Sending build context to Docker daemon  2.935MB
Step 1/7 : FROM node:12-alpine
 ---> 106bb94759ad
Step 2/7 : WORKDIR /usr/src/app
 ---> Using cache
 ---> e6ecd7d38222
Step 3/7 : COPY *.js ./
 ---> ee2a9401c963
Step 4/7 : COPY package*.json ./
 ---> b2b68a4866e8
Step 5/7 : RUN npm install
 ---> Running in c3eb854c1f7c
npm WARN cluster_bench@1.0.0 No description
npm WARN cluster_bench@1.0.0 No repository field.

added 1 package from 1 contributor and audited 1 package in 1.627s
found 0 vulnerabilities

Removing intermediate container c3eb854c1f7c
 ---> bdff503d166e
Step 6/7 : EXPOSE 8000
 ---> Running in 7a2d7fd46fa7
Removing intermediate container 7a2d7fd46fa7
 ---> 02bab3027808
Step 7/7 : CMD [ "node", "server.js" ]
 ---> Running in e7da3582e987
Removing intermediate container e7da3582e987
 ---> 48eff8c8c1d6
Successfully built 48eff8c8c1d6
Successfully tagged worker_thread_no_balancer_server:latest
WARNING: Image for service server was built because it did not already exist. To rebuild this i
pose up --build`.
Creating worker_thread_no_balancer_server ... done
```

Рисунок 5.16 - Шаги збирання контейнеру web-сервера з використанням робочих потоків.

Після того як контейнер буде зібраний у ньому запуститься web-сервер, що працює із використанням механізму робочих потоків. У цьому випадку ми також розглянемо здатність витримувати навантаження сервісу, відповідального за об’єкти нерухомості. За допомогою інструмента Autocannon використаємо таку саму команду з таким самим навантаженням:

autocannon -c 500 -d 20 http://charly.de/objects

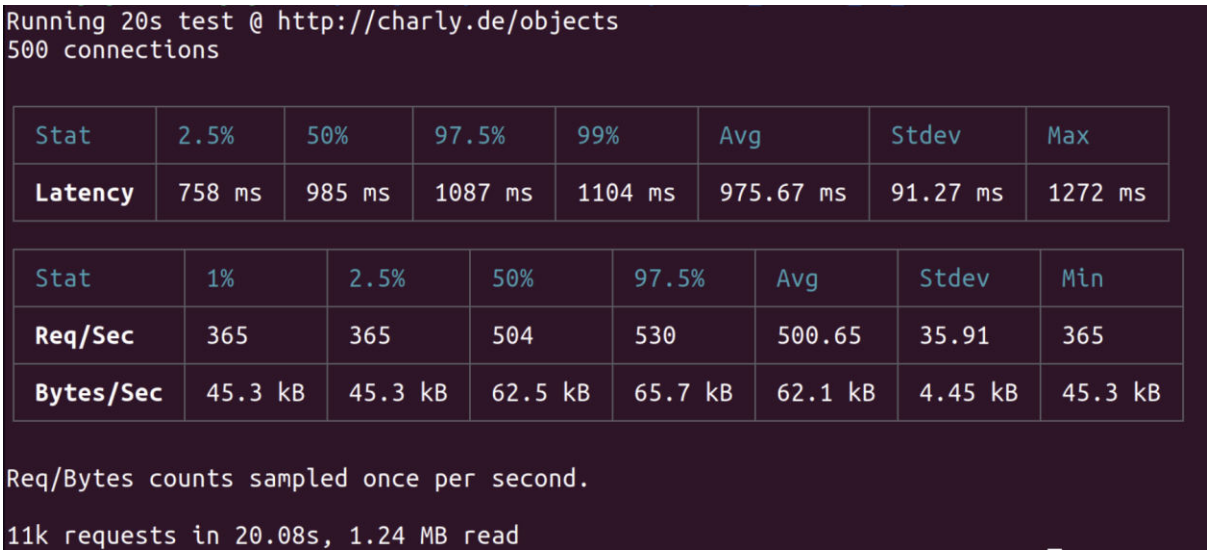
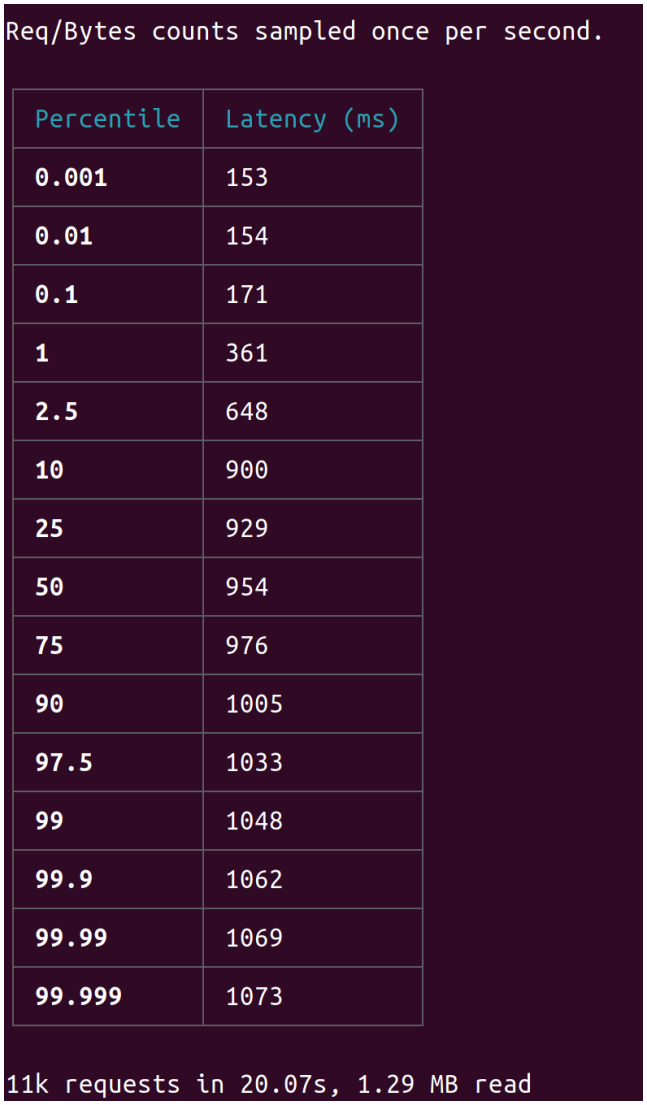


Рисунок 5.17 - Вихідні результати, таблиця затримок (зверху) та таблиця обсягу запитів (знизу).

Проаналізувавши результати (Рис. 5.17), можна зробити висновки, що інструмент Autocannon використовував середню кількість запитів у секунду, що дорівнює 500.65 та за 20 секунд зробив 11 тисяч запитів. Маємо 0 помилок, а середня затримка відповіді 975.67 мілісекунд. Більш детальна таблиця затримок (Рис. 5.18), що описує відсоток навантаження на сервер та відповідну йому затримку у мілісекундах свідчить про те, що відповіді з сервера навіть у пікових навантаженнях приходили у час дуже близький до середнього. Максимальний час відповіді за період тестових навантажень складав 1272 мілісекунди.


Req/Bytes counts sampled once per second.

Percentile	Latency (ms)
0.001	153
0.01	154
0.1	171
1	361
2.5	648
10	900
25	929
50	954
75	976
90	1005
97.5	1033
99	1048
99.9	1062
99.99	1069
99.999	1073

11k requests in 20.07s, 1.29 MB read

Рисунок 5.18 - детальна таблиця затримок, згенерована на базі результатів тестування навантаження.

5.4 Розрахунок та моделювання комп'ютерної системи з використанням функціонала робочих потоків та використанням зовнішнього інструмента відповідального за балансування навантаження.

Використання зовнішнього банасувача навантаження, що відтворений за допомогою інструмента Haproxy потребує додаткової конфігурації контейнеру з балансувачем. Але незважаючи на це, репозиторій з web-сервером також завантажується на екземпляр Amazon EC2 за допомогою системи контролю версій

Git, і запускається за допомогою команди docker-compose up -d. Після запуску цієї команди контейнер згідно до шагів (Рис. 5.21) буде зібраний та готовий для використання

docker-compose.yml

```yaml
version: '2.4'
services:
  server:
    build:
      context: ./server
      dockerfile: ./Dockerfile
    environment:
      NODE_ENV: production
    image: worker_thread_external_balancer_server_2
    container_name: worker_thread_external_balancer_server_2
    restart: always
    ports:
      - "8001:8001"
      - "8002:8002"
      - "8003:8003"
      - "8004:8004"
    cpus: '4'
  haproxy:
    build:
      context: ./haproxy
      dockerfile: ./Dockerfile
    image: worker_thread_external_balancer_haproxy
    container_name: worker_thread_external_balancer_haproxy
    depends_on:
      - server
    ports:
      - "80:80"
    restart: always
    cpus: '1'
```

Рисунок 5.19 - docker-compose.yml файл із конфігурацією web-сервера із використанням робочих потоків та зовнішнього проксі балансувальника Haproxy.

Даний контейнер з web-сервером використовує оптимізоване оточення production, працює встановлює оптимізовану та стабільну версію програмної платформи NodeJS node:12-alpine, та працює на 8001, 8002, 8003, 8004 портах, та на які з 80 порта проксуються запити за допомогою алгоритму розподілення навантаження Round-robin, що реалізований за допомогою інструмента Haproxy (Рис. 5.19).

Dockerfile

```
FROM node:12-alpine

WORKDIR /usr/src/app

COPY *.js ./

EXPOSE 8001

EXPOSE 8002

EXPOSE 8003

EXPOSE 8004

CMD [ "node", "server.js" ]
```

Dockerfile

```
FROM haproxy

COPY haproxy.cfg /usr/local/etc/haproxy/haproxy.cfg
```

Рисунок 5.20 - Dockerfile web-сервера, що працює із використанням робочих потоків (зверху) та Haproxy проксі-сервера (знизу).

```
Creating network "worker_thread_external_balancer_default" with the default driver
Building server
Sending build context to Docker daemon  6.656kB
Step 1/8 : FROM node:12-alpine
 ---> 106bb94759ad
Step 2/8 : WORKDIR /usr/src/app
 ---> Using cache
 ---> e6ecd7d38222
Step 3/8 : COPY *.js ./
 ---> 2850f91134e5
Step 4/8 : EXPOSE 8001
 ---> Running in 1ec894bb4df6
Removing intermediate container 1ec894bb4df6
 ---> b143b310be5c
Step 5/8 : EXPOSE 8002
 ---> Running in b03b3a5c9a4e
Removing intermediate container b03b3a5c9a4e
 ---> c7b1c80bfebb
Step 6/8 : EXPOSE 8003
 ---> Running in 84dc3408659a
Removing intermediate container 84dc3408659a
 ---> 7ee74fd6428e
Step 7/8 : EXPOSE 8004
 ---> Running in 3c409e6f0670
Removing intermediate container 3c409e6f0670
 ---> 9795dd8e8eba
Step 8/8 : CMD [ "node", "server.js" ]
 ---> Running in ce3d1eb4b3f1
Removing intermediate container ce3d1eb4b3f1
 ---> 32b190de8164
Successfully built 32b190de8164
Successfully tagged worker_thread_external_balancer_server_2:latest
```

```
Building haproxy
Sending build context to Docker daemon  3.584kB
Step 1/2 : FROM haproxy
latest: Pulling from library/haproxy
a2abf6c4d29d: Pull complete
6d06b1621367: Pull complete
41acba0a6a4d: Pull complete
2edb97a2e8e6: Pull complete
Digest: sha256:78819cc9c600464c97822be5903db18be24fce9733c76d1ccb8c8863ceb00f35
Status: Downloaded newer image for haproxy:latest
 ---> 575a5788d81a
Step 2/2 : COPY haproxy.cfg /usr/local/etc/haproxy/haproxy.cfg
 ---> 11d0a7079f20
Successfully built 11d0a7079f20
Successfully tagged worker_thread_external_balancer_haproxy:latest
```

Рисунок 5.21 - Шаги збирання контейнеру web-сервера з використанням робочих потоків (зверху) та проксі-сервера (знизу).

Після того як контейнери будуть зібрані у одному з них запуститься web-сервер, що працює із використанням механізму робочих потоків, а у другому проксі-сервер з механізмом розподілення навантаження. У цьому випадку ми також розглянемо здатність витримувати навантаження сервісу, відповідального за об’єкти нерухомості. За допомогою інструмента Autocannon використаємо команду з таким самим навантаженням:

autocannon -c 500 -d 20 http://charly.de/objects

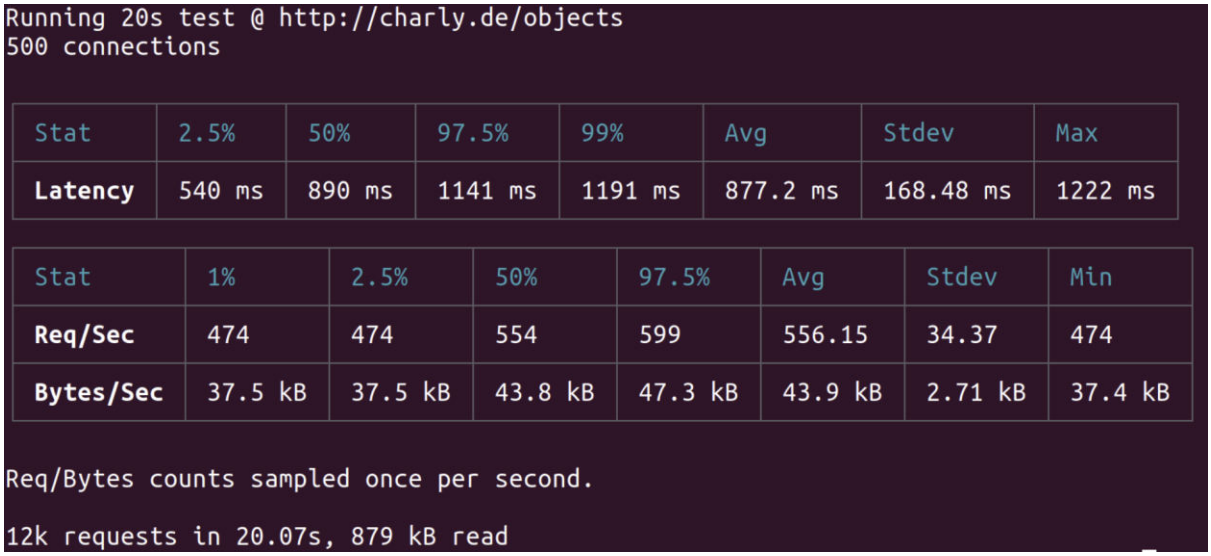


Рисунок 5.22 - Вихідні результати, таблиця затримок (зверху) та таблиця обсягу запитів (знизу).

Проаналізувавши результати (Рис. 5.22), можна зробити висновки, що інструмент Autocannon використовував середню кількість запитів у секунду, що дорівнює 556.15 та за 20 секунд зробив 12 тисяч запитів. Маємо 0 помилок, а середня затримка відповіді 877.2 мілісекунд. Більш детальна таблиця затримок (Рис. 5.22), що описує відсоток навантаження на сервер та відповідну йому затримку у мілісекундах свідчить про те, що відповіді з сервера по аналогії з варіантом використання робочих потоків без зовнішнього проксі-серверу наближається до середнього та складав 1222 мілісекунди. Але на відміну від затримки відповіді web-серверу що працював із використанням робочих потоків, поточний варіант реалізації має меншу затримку відповіді на менших рівнях завантаженості web-серверу.

Req/Bytes counts sampled once per second.

Percentile	Latency (ms)
0.001	171
0.01	171
0.1	186
1	366
2.5	534
10	592
25	758
50	901
75	974
90	1032
97.5	1088
99	1173
99.9	1199
99.99	1200
99.999	1200

12k requests in 20.07s, 904 kB read

Рисунок 5.23 - детальна таблиця затримок, згенерована на базі результатів тестування навантаження.

haproxy.cfg

```
global
    log /dev/log local0
    log localhost local1 notice
    maxconn 10000
    daemon

defaults
    log global
    mode http
    option httplog
    option dontlognull
    retries 3
    timeout connect 5000
    timeout client 50000
    timeout server 50000

frontend http-in
    bind *:80
    default_backend webservers

backend webservers
    stats enable
    stats auth admin:admin
    stats uri /haproxy?stats
    balance roundrobin
    option httpchk
    option forwardfor
    option http-server-close
    server node1 host.docker.internal:8001
    server node2 host.docker.internal:8002
    server node3 host.docker.internal:8003
    server node4 host.docker.internal:8004
```

Рисунок 5.24 - конфігурація проксі-сервера Haproxy, що відповідає за розподілення навантаження.

5.5 Висновки до розділу

У розділі було реалізовано усі засоби оптимізації web-серверу компанії з ріелторських послуг, запущені, та після впевненості, що вся інфраструктура працює, проведенні тестування окремих засобів оптимізації навантаженням, у максимально наближених один до одного умовах.

ВИСНОВКИ

Кваліфікаційна робота є завершеною науковою роботою, в якій вирішена науково-практична задача. У даній роботі було сформовано цілі, що потребували вирішення для компанії з надання ріелторських послуг за допомогою інтеграції у її роботу web-додатку. Основна вимога, що полягала, у здатності web-сервера працювати під великим навантаженням була реалізована за допомогою застосування доступних засобів оптимізації.

Розглянуті засоби оптимізації полягали як у використанні вбудованого функціоналу кластеризації та декомпозиції програмної платформи, що застосовувалась, так і у використанні сторонніх менеджерів процесів та інструментів розподілення навантаження між серверами.

У результаті аналізу практичної частини було вирішено використовувати засіб оптимізації web-сервера, що полягав у використанні стороннього менеджеру процесів, оскільки результати тестів пропускної здатності сервера показали задовільний результат затримки відповіді. Також затримка відповіді під час невеликих навантажень була однією з найменших. Крім того використання менеджеру процесів дозволяє використовувати великий спектр додаткових метрик та налаштувань, що полегшує підтримку web-сервера.

Практичну цінність дана робота може мати для тих, хто потребує оптимізувати web-сервер, що планується до розробки, або вже існуючий.

Перелік посилань

1. Пауэрс Ш. Изучаем Node. Переходим на сторону сервера 2017. – 304 с.
2. Майк Кантелон, Марк Хартер, Натан Райлих, TJ Головайчук, Алекс Янг «Node.js в действии» 2018 р. 432 с.
3. Документація Node [Електронний ресурс] - https://nodejs.org/en/docs/
4. Документація V8 [Електроний ресурс] - https://v8.dev/docs
5. Документація Docker [Електронний ресурс] - https://www.docker.com/
6. Документація AWS [Електронний ресурс] - https://aws.amazon.com/
7. Документація PM2 [Електронний ресурс] - https://pm2.keymetrics.io/docs/
8. Документація Autocannon [Електронний ресурс] - https://www.npmjs.com/package/autocannon
9. Документація DenoJS [Електронний ресурс] - https://doc.deno.land/
10. Load Balancing with HAProxy [Електронний ресурс] - https://blog.stackpath.com/load-balancing-haproxy/
11. Best practices to containerize Node.js web applications with Docker [Електронний ресурс] - https://snyk.io/blog/10-best-practices-to-containerize-nodejs-web-applications-with-docker/

ДОДАТОК А

Текст програми мовою JavaScript із використанням кластерів

Програмне забезпечення комп’ютерної системи web-сервера компанії з надання ріелторських послуг

Національний технічний університет

«Дніпровська політехніка»

ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ КОМП'ЮТЕРНОЇ СИСТЕМИ WEB-СЕРВЕРА КОМПАНІЇ З НАДАННЯ РІЕЛТОРСЬКИХ ПОСЛУГ

Текст програми мовою JavaScript із використанням кластерів

804.02070743.22002-01

Аркушів 4

2022

АНОТАЦІЯ

У документі наведено текст програми та інфраструктурної конфігурації, розробленої в кваліфікаційній роботі.

Розроблена програма та інфраструктурна конфігурація є реалізацією оптимізації web-сервера компанії з надання ріелторських послуг для реалізації підприємницької діяльності. Оптимізація виконувалась горизонтально із використанням функціоналу кластерів.

Програма написана на мові JavaScript із використанням програмної платформи NodeJS.

ЗМІСТ

Текст інфраструктурних файлів

```
// docker-compose.yml
version: '2.4'
services:
  server:
    build:
      context: ./server
      dockerfile: ./Dockerfile
    environment:
      NODE_ENV: production
    image: cluster_server
    container_name: cluster_server
    restart: always
    ports:
      - "80:8000"
    cpus: '5'


// Dockerfile
FROM node:12-alpine

WORKDIR /usr/src/app

COPY *.js ./

EXPOSE 8000

CMD [ "node", "server.js" ]
```

Текст програми на мові JavaScript

```
// server.js

const cluster = require('cluster');
const http = require('http');
const numCPUs = 5;
const app = require('./app');

const host = process.env.HOST || '0.0.0.0';
const port = process.env.PORT || 8000;

const start = async function startServer() {
   // Cluster
   if (cluster.isMaster) {
      console.log(`Master ${process.pid} is running.`);

      // Run cluster.fork based on numCPUs
      for (let i = 1; i < numCPUs; i += 1) {
         cluster.fork();
      }

      cluster.on('exit', (worker, code, signal) => {
         console.log(`A worker with ID ${worker.process.pid} died.`);
      })
   } else {
      const router = function requestRouter(request, response) {
         app(request, response);
      }

      const server = http.createServer(router);
      server.listen(port, host, () => {
         console.log(`Node.js Standard Library HTTP server running on port: ${port}`);
      })
   }
}

start();
```

ДОДАТОК Б

Текст програми мовою JavaScript із використанням менеджеру процесів

Програмне забезпечення комп’ютерної системи web-сервера компанії з надання ріелторських послуг

Національний технічний університет

«Дніпровська політехніка»

ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ КОМП'ЮТЕРНОЇ СИСТЕМИ WEB-СЕРВЕРА КОМПАНІЇ З НАДАННЯ РІЕЛТОРСЬКИХ ПОСЛУГ

Текст програми мовою JavaScript із використанням менеджеру процесів

804.02070743.22002-01

Аркушів 4

2022

АНОТАЦІЯ

У документі наведено текст програми та інфраструктурної конфігурації, розробленої в кваліфікаційній роботі.

Розроблена програма та інфраструктурна конфігурація є реалізацією оптимізації web-сервера компанії з надання ріелторських послуг для реалізації підприємницької діяльності. Оптимізація реалізована горизонтально з використанням стороннього менеджеру процесів.

Програма написана на мові JavaScript із використанням програмної платформи NodeJS.

ЗМІСТ

Текст інфраструктурних файлів

```
// docker-compose.yml
version: '2.4'
services:
  server:
    build:
      context: ./server
      dockerfile: ./Dockerfile
    environment:
      NODE_ENV: production
    image: pm2_server
    container_name: pm2_server
    restart: always
    ports:
      - "80:8000"
    cpus: '5'
```

```
// Dockerfile
FROM node:12-alpine

WORKDIR /usr/src/app

COPY *.js ./

EXPOSE 8000

RUN npm install pm2 -g

CMD ["pm2-runtime", "ecosystem.config.js"]
```

Текст програми на мові JavaScript

```
// server.js

const cluster = require('cluster');
const http = require('http');
const app = require('./app');

const host = process.env.HOST || '0.0.0.0';
const port = process.env.PORT || 8000;

const router = function requestRouter(request, response) {
   app(request, response);
}

const server = http.createServer(router);
server.listen(port, host, () => {
   console.log(`Node.js Standard Library HTTP server running on port: ${port}`);
})
```

ДОДАТОК В

Текст програми мовою JavaScript із використанням робочих потоків

Програмне забезпечення комп’ютерної системи web-сервера компанії з надання ріелторських послуг

Національний технічний університет

«Дніпровська політехніка»

ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ КОМП'ЮТЕРНОЇ СИСТЕМИ WEB-СЕРВЕРА КОМПАНІЇ З НАДАННЯ РІЕЛТОРСЬКИХ ПОСЛУГ

Текст програми мовою JavaScript із використанням робочих потоків

804.02070743.22002-01

Аркушів 4

2022

АНОТАЦІЯ

У документі наведено текст програми та інфраструктурної конфігурації, розробленої в кваліфікаційній роботі.

Розроблена програма та інфраструктурна конфігурація є реалізацією оптимізації web-сервера компанії з надання ріелторських послуг для реалізації підприємницької діяльності. Оптимізація виконана горизонтально з використанням ункціоналу робоих потоків.

Програма написана на мові JavaScript із використанням програмної платформи NodeJS.

ЗМІСТ

Текст інфраструктурних файлів

```
// docker-compose.yml
version: '2.4'
services:
  server:
    build:
      context: ./server
      dockerfile: ./Dockerfile
    environment:
      NODE_ENV: production
    image: worker_thread_no_balancer_server
    container_name: worker_thread_no_balancer_server
    restart: always
    ports:
      - "80:8000"
    cpus: '5'
```

```
// Dockerfile
FROM node:12-alpine

WORKDIR /usr/src/app

COPY *.js ./

COPY package*.json ./

RUN npm install

EXPOSE 8000

CMD [ "node", "server.js" ]
```

Текст програми на мові JavaScript

```
// server.js

const http = require('http');
const { StaticPool } = require('node-worker-threads-pool');
const numCPUs = 5;

const host = process.env.HOST || '0.0.0.0';
const port = process.env.PORT || 8000;

const start = function startServer() {
   const filePath = './worker.js';
   const pool = new StaticPool({
      size: numCPUs - 1,
      task: filePath,
      workerData: {},
   });

   const router = async function requestRouter(request, response) {
      try {
         const result = await pool.exec(request, response);
         response.end(result);
      } catch (e) {
         console.log(e);
         process.exit(0);
      }
   }

   const server = http.createServer(router);
   server.listen(port, host, () => {
      console.log(`Node.js Standard Library HTTP server running on port: ${port}`)
   })
}

start();
```

ДОДАТОК Г

Текст програми мовою JavaScript із використанням кластерів і зовнішнього балансування навантаження

Програмне забезпечення комп’ютерної системи web-сервера компанії з надання ріелторських послуг

Національний технічний університет

«Дніпровська політехніка»

ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ КОМП'ЮТЕРНОЇ СИСТЕМИ WEB-СЕРВЕРА КОМПАНІЇ З НАДАННЯ РІЕЛТОРСЬКИХ ПОСЛУГ

Текст програми мовою JavaScript із використанням кластерів і зовнішнього балансування навантаження

804.02070743.22002-01

Аркушів 5

2022

АНОТАЦІЯ

У документі наведено текст програми та інфраструктурної конфігурації, розробленої в кваліфікаційній роботі.

Розроблена програма та інфраструктурна конфігурація є реалізацією оптимізації web-сервера компанії з надання ріелторських послуг для реалізації підприємницької діяльності. Оптимізація виконана горизонтально з використанням функціоналу робочих потоків та зовнішнього інструменту з балансування навантаження.

Програма написана на мові JavaScript із використанням програмної платформи NodeJS.

ЗМІСТ

Текст інфраструктурних файлів

```
// docker-compose.yml
version: '2.4'
services:
  server:
    build:
      context: ./server
      dockerfile: ./Dockerfile
    environment:
      NODE_ENV: production
    image: worker_thread_external_balancer_server_2
    container_name: worker_thread_external_balancer_server_2
    restart: always
    ports:
      - "8001:8001"
      - "8002:8002"
      - "8003:8003"
      - "8004:8004"
    cpus: '4'
  haproxy:
    build:
      context: ./haproxy
      dockerfile: ./Dockerfile
    image: worker_thread_external_balancer_haproxy
    container_name: worker_thread_external_balancer_haproxy
    depends_on:
      - server
    ports:
      - "80:80"
    restart: always
    cpus: '1'


// haproxy/Dockerfile
FROM haproxy

COPY haproxy.cfg /usr/local/etc/haproxy/haproxy.cfg
// server/Dockerfile
FROM node:12-alpine

WORKDIR /usr/src/app
```

```
COPY *.js ./

EXPOSE 8001

EXPOSE 8002

EXPOSE 8003

EXPOSE 8004

CMD [ "node", "server.js" ]
```

Текст програми на мові JavaScript

```
// server/server.js

const {Worker} = require('worker_threads');
const numCPUs = 4;
const host = process.env.HOST || '0.0.0.0';
const port = process.env.PORT || 8000;

const start = function startServer() {
   for (let i = 1; i<=numCPUs; i++) {
      new Worker('./worker.js', {workerData: {host, port: port + i}});
   }
}

start()


// server/worker.js
const { parentPort, workerData } = require('worker_threads')
const http = require('http');
const app = require('./app')

const server = http.createServer((req, res) => {
   app(req, res);
})
server.listen(workerData.port, workerData.host, () => {
   console.log(`Node.js Standard Library HTTP server running on port:
${workerData.port}`)
})
```