КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
факультет інформаційних технологій
кафедра програмних систем і технологій

На правах рукопису

УДК_____

ВИПУСКНА КВАЛІФІКАЦІЙНА МАГІСТЕРСЬКА РОБОТА

Тема: «Архітектура додатку аналізу динамічних даних Інтернету всього на основі мікросервісної архітектури»

Спеціальність 121 «Інженерія програмного забезпеченні»

Виконав студент

_____
(шифр (підпис) (дата) (розшифровка групи) підпису)

Науковий керівник

_____
(посада) (підпис) (дата) (розшифровка підпису)

Робота допущена до захисту
на засіданні кафедри протокол №__ від «__» _____ 2021
завідувач кафедри Програмних систем і технологій

_____

2021

Рішенням Екзаменаційної комісії
випускна кваліфікаційна робота студента

_____

захищена з оцінкою

_____

Голова Екзаменаційної комісії

Київський національний університет імені Тараса Шевченка
Факультет інформаційних технологій
Кафедра програмних систем і технологій
Спеціальність 6.050103 Програмна інженерія

ЗАТВЕРДЖУЮ:
Завідувач кафедри
програмних систем і технологій
_____(О.С.Бичков)
„___” _____ 20___р.

ЗАВДАННЯ
НА ВИПУСКНУ КВАЛІФІКАЦІЙНУ МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ
Козачку Павлу Анатолійовичу
(прізвище, ім’я, по батькові)

1. Тема випускної кваліфікаційної роботи Архітектура додатку аналізу динамічних даних Інтернету всього на основі мікросервісної архітектури
затверджені наказом вищого навчального закладу від „__”_____20__р. №_____

2. Строк здачі студентом закінченої роботи _____
3. Вихідні дані до роботи

Сервіс-орієнтована архітектура, що підтримує безсерверні обчислення
Розроблений прототип гібридної архітектури
Проведений аналіз розробленої архітектури

4. Зміст пояснювальної записки (перелік питань, що їх належить розробити)

1. Аналіз технологій для побудови прототипу даної архітектури
2. Опис принципів проектування розробленого прототипу
3. Встановлення взаємозв’язку між складовими вузлами прототипу

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

6. Консультанти з роботи із зазначенням розділів роботи, що їх стосуються

Розділ	Консультант	Підпис, дата	
		Завдання ви-дав	Завдання прийняв

7. Дата видачі завдання _____

Керівник _____ _____
(підпис) (розшифровка підпису)

Завдання прийняв до виконання _____ _____
(підпис) (розшифровка підпису)

КАЛЕНДАРНИЙ ПЛАН

Номер і назва етапів кваліфікаційної роботи	Термін виконання етапів роботи	Відмітка про виконання етапів роботи	Примітка
1. Постановка задачі	22.10.2020	Виконав	
2. Дослідження предметної області	27.11.2020	Виконав	
3. Проектування рішення	22.01.2021	Виконав	
3.1 Документування та обгрунтування запропонованого рішення	29.01.2021	Виконав	
3.2 Аналіз можливих вузьких місць	22.02.2021	Виконав	
4. Розробка архітектури для сервіс-орієнтованого додатку	28.02.2021	Виконав	
5. Розробка сервіс-орієнтованого додатку	10.03.2021	Виконав	
5. Розгортання архітектури в гібридному середовищі	19.03.2021	Виконав	
6. Проведення комплексного тестування отриманого рішення	20.03.2021	Виконав	
7. Оформлення і друк пояснювальної записки		Виконав	
8. Отримання рецензій		Виконав	
9. Отримання допуску до захисту		Виконав	

Студент – магістр _____П.А.Козачок_____
(підпис) (розшифровка підпису)

Керівник роботи _____В.Л. Шевченко _____
(підпис) (розшифровка підпису)

АНОТАЦІЯ

Випускна кваліфікаційна робота: 78 с., 27 рис., 17 табл., 20 джерел.

Тема: Архітектура аналізу динамічних даних інтернету всього на основі мікросервісної архітектури.

Об'єкт дослідження: інформаційна технологія, що дозволяє будувати додатки будь-якого рівня складності та типу, за допомогою наявних ресурсів, які повинні розподілятися відносно потреб кінцевих користувачів та навантаження на наявні вузли. Технологія забезпечує можливість інтегрування нових обчислювальних пристроїв в систему: як хмарних, так і Raspberry Pi 4, що виконує роль локального обслуговувача в межах локальної мережі.

Предмет дослідження: створення архітектури системи, що забезпечує її реконфігурування в залежності від потреби споживачів та її навантаження.

Мета роботи: є покращення гнучкості розгортання бізнес-логіки в системі з використанням концепції капілярної доставки сервісів «Fog and Edge» і Raspberry Pi 4 у якості сервера, що обслуговує користувача.

Результати дослідження: запропонована модель архітектури гібридного типу розгортання. Вдосконалено метод доставки бізнес-логіки за рахунок комбінування принципів капілярної доставки "Fog and Edge" та безсерверних обчислень. Розроблено прототип системи, що може обслуговувати користувача в рамках локальної мережі, та хмарного середовища.

Висновок: В результаті виконання кваліфікаційної роботи було побудовано прототип сервіс-орієнтованого додатку, що базується на принципах безсерверних обчислень та моделі гібридного типу розгортання. Було проведено ряд тестів для встановлення найоптимальніших сценаріїв взаємодії з локальними та хмарними вузлами.

Перелік ключових слів:

Fog and Edge обчислення, Raspberry Pi, Node.JS, гібридна інфраструктура, мікросервісна архітектура, хмарні обчислення, безсерверні обчислення.

АНОТАЦИЯ

Выпускная квалификационная работа: 78 с., 27 рис., 17 табл., 20 источ.

Тема: Архитектура для анализа динамических данных интернета всего на основе микросервисной архитектуры.

Объект исследования: информационная технология, позволяющая строить приложения любого уровня сложности и типа, с помощью имеющихся ресурсов, которые должны распределяться относительно потребностей конечных пользователей и нагрузки на имеющиеся узлы. Технология обеспечивает возможность интегрирования новых вычислительных устройств в систему: как облачных, так и Raspberry Pi 4, выполняющего роль обслуги в пределах локальной сети.

Предмет исследования: создание архитектуры системы, обеспечивающей ее ре-конфигурацию в зависимости от потребности потребителей и ее нагрузки.

Цель работы: является улучшение гибкости развертывания бизнес-логики в системе с использованием концепции капиллярной доставки сервисов «Fog and Edge» и Raspberry Pi 4 в качестве сервера, обслуживающего пользователя.

Результаты исследования: предложена модель архитектуры гибридного типа развертывания. Усовершенствован метод доставки бизнес-логики за счет комбинирования принципов капиллярной доставки "Fog and Edge" и безсерверных вычислений. Разработан прототип системы, которая может обслуживать в рамках локальной сети и облачного среды.

Вывод: В результате выполнения квалификационной работы был разработан прототип сервис-ориентированного приложения, которое базируется на принципах безсерверных вычислений и модели гибридного типа развертывания. Был проведен ряд тестов для установления оптимальных сценариев взаимодействия с локальными и облачными узлами.

Перечень ключевых слов:

Fog and Edge вычисления, Raspberry Pi, Node.JS, гибридная инфраструктура, микросервисная архитектура, облачные вычисления, безсерверные вычисления.

ANNOTATION

Masters research work: 78 pages, 27 pictures., 17 tables., 20 references.

Topic: The architecture of the application for the analysis of dynamic Internet of Everything data on the basis of micro-server architecture

Object of research: An informational technology that allows you to build applications of any level of complexity and type, using available resources, which must be distributed in order to the needs of end users and the load on existing nodes. The technology provides the ability to integrate new computing devices into the system: both cloud and Raspberry Pi 4, which acts as a service node within the local network.

Subject of research: creation of the architecture of the system, ensuring its reconfiguration depending on the needs of users and its load.

Goal is to improve the flexibility of deploying business logic in the system using the concept of capillary delivery of services "Fog and Edge" and Raspberry Pi 4 as the server which serving the user.

Results: An architecture model of a hybrid deployment type is proposed. The business logic delivery method has been improved by combining the principles of capillary delivery "Fog and Edge" and serverless computing. A prototype of a system has been developed that can serve a user within the local network and in the cloud.

Keyword list:

Fog and Edge computing, Raspberry Pi, Node.JS, Hybrid Infrastructure, Microservices Architecture, Cloud Computing, Serverless computing.

ЗМІСТ

ПЕРЕЛІК ОСНОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ

SOA - Service oriented Architecture - архітектурний шаблон програмного забезпечення, модульний підхід до розробки програмного забезпечення, заснований на використанні розподілених, слабко пов'язаних замінних компонентів, оснащених стандартизованими інтерфейсами для взаємодії за стандартизованими протоколами.

ORM - Object Relation Mapping - технологія програмування, яка зв'язує бази даних з концепціями об'єктно-орієнтованих мов програмування, створюючи «віртуальну об'єктну базу даних».

HTTP - широко поширений протокол передачі даних, спочатку призначений для передачі гіпертекстових документів

JSON - це текстовий формат обміну даними між комп'ютерами. JSON базується на тексті, який може бути прочитанний людиною.

JWT - це стандарт токена доступу на основі JSON, стандартизованого в RFC 7519.

CPU - функціональна частина комп'ютера, что призначила для інтерпретації команд.

IO - взаємодія між обробником інформації (наприклад, комп'ютер) і зовнішнім світом, який може представляти як людина, так і будь-яка інша система обробки інформації.

REST - Representational State Transfer - підхід до архітектури мережевих протоколів, які забезпечують доступ до інформаційних ресурсів.

SPA - це веб-застосунок чи веб-сайт, який вміщується на одній сторінці з метою забезпечити користувачу досвід близький до користування настільною програмою.

PWA - це веб додаток, створений з використанням певних технологій для досягнення заданих цільових показників.

ВСТУП

Актуальність роботи обумовлена практичною потребою підприємств розгорнути бізнес-логіку в гібридному середовищі виконання та обслуговувати якомога більше користувачів системи з найвищим рівнем утилізації обчислювальної потужності та найменшим часом відгуку на запит. Деякі види додатків, що обслуговують користувачів мають підвищені вимоги до часу відповіді, наявності обчислювальних ресурсів (наприклад, системи реагування, система доставки реклами, тощо).

Мета і задачі дослідження

Метою магістерської роботи є покращення гнучкості розгортання бізнес-логіки в системі з використанням концепції капілярної доставки сервісів «Fog and Edge» і Raspberry Pi 4 у якості сервера, що обслуговує користувача.

Об'єктом дослідження є - Інформаційна технологія, що дозволяє будувати додатки будь-якого рівня складності та типу, за допомогою наявних ресурсів, які повинні розподілятися відносно потреб кінцевих користувачів та навантаження на наявні вузли. Технологія забезпечує можливість інтегрування нових обчислювальних пристроїв в систему: як хмарних, так і Raspberry Pi 4, що виконує роль локального обслуговувача в межах локальної мережі.

Предметом дослідження є - створення архітектури системи, що забезпечує її ре-конфігурування в залежності від потреби споживачів та її навантаження.

Методи дослідження

При виконанні досліджень та прийнятті рішень використовувались такі **підходи**: прототипування, теоретичні дослідження, методи багатокритеріальної оптимізації, метод аналізу компромісів архітектури, математичне моделювання.

Наукова новизна отриманих результатів

Вдосконалено методи доставки бізнес-логіки на обчислювальні засоби підприємства, що знаходяться в межах хмарного середовища, або ж в локальній мережі. Була вдосконалена модель архітектури інформаційної системи, що може

сама себе конфігурувати та надає зручний спосіб інтеграції вже готових додатків до неї.

Практичне значення одержаних результатів

Методи, що представлені в цій кваліфікаційній роботі дозволяють доставляти бізнес-логіку з більшим показником еластичності, меншим часом очікування відповіді користувачем за рахунок використання вузлів, що представлені мікрокомп'ютерами такими Raspberry Pi та знаходяться в межах локальної мережі. Запропоноване рішення має в собі переваги хмарного середовища та локального розгортання за рахунок локальних вузлів, або ж хмарних обчислювальних ресурсів, що знаходяться в географічній близькості до кінцевого користувача.

Особистий внесок студента

Основним результатом є: проектування гібридної архітектури та розробка прототипу, що виконує задачу обслуговування користувача.

Апробація результатів випускної кваліфікаційної роботи

Результати дослідження були представлені та схвалені на міжнародній науково-технічній фаховій конференції CADSM 15, що відбулася з 26-го лютого по 1-е березня 2019 в с. Поляна, Закарпатська область. Результати дослідження було впроваджено компанією «Компанія технологія захисту». Результати досліджень впроваджені в навчальному процесі в дисципліні «Архітектура комп’ютера».

Результати дослідження було апробовано:

- 2017-18 навчальний рік - II місце. ХНУРЕ. Спеціальність «Автоматизація і комп’ютерно-інтегровані технології».
- 2018-19 навчальний рік - II місце ХНУРЕ. Спеціальність «Комп’ютерні науки».

Публікації

1. **Козачок П.А.**, Бражиненко М.Г., Шевченко В.Л. . Проблеми публікації-підписки у бездротових мережах для вузлів Інтернету всього / MSTIoE 2018-3 : East European Conference on Mathematical Foundations and Software Technology of Internet of Everything (21-22.12.2017, Kyiv - Київ). Зб. Матер. С.32.

2. **Kozachok P.**, Brazhenenko M., Shevchenko V., Tkachenko M. M2M Communication Protocol for Low Bandwith MEMS Sensor Networks. 2018 14-th International Conference Perspective Technologies and Methods in MEMS Design (MEMSTECH). Proceeding. - Polyana, April 18-22 pp.35-38. Scopus

3. **Kozachok P.**, Brazhenenko M., Byckov O., Shevchenko V., Petrivskiy V. Cloud Based Architecture Design of System of Systems 2019 CADSM 15-th - Polyana, February 26 – March 1 t.6 pp.19-24.

Структура та обсяг роботи

Робота викладена на 78 сторінках друкованого тексту, який складається із вступу, чотирьох розділів, висновків, списку використаних джерел (20 найменувань), 5 додатків. Робота містить 17 таблиць, 27 рисунків.

РОЗДІЛ 1
ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Розгортання бізнес логіки

За статистикою InnMind кожного року в світі відкривається понад 100 млн. стартапів, з них понад 1,35 мільйонів - це технологічні стартапи. Та лише 1% з них має можливість перерости в середній, або великий бізнес. Це може бути пов'язано з не достатком фінансування, не достатком експертизи при реалізації прототипу, або ж через ситуацію, коли розроблене рішення неможливо підтримувати в рамках конкретної інфраструктури, а перехід на іншу інфраструктуру не розглядається у зв'язку зі складністю реалізації.

Для того, щоб вирішити проблему розгортання бізнес-логіки [1] певного підприємства в хмарних середовищах необхідно ізолювати програмістів від конфігурації середовища виконання їхнього коду, а натомість надати рішення, яке могло би витримати високе навантаження, оптимально утилізувати наявні ресурси системи, надати гнучкість і зручність на етапі проектування та розробки.

З впровадженням штучного інтелекту у різні сфери все більше і більше виникає необхідність в якомога швидкому відгуку для користувача. До прикладу, онлайн аукціони можуть отримувати дуже велике навантаження від користувачів, які роблять свої ставки. Іншим прикладом може служити доставка реклами, коли користувач заходить на сторінку певного веб-ресурсу, різні алгоритми працюють над тим, щоб надати йому найбільш відповідну рекламу, якщо один рекламний провайдер не витримує правила надати відповідь за декілька мілісекунд, то він не може витримати конкуренцію з тими провайдерами алгоритми яких вписується в заданий час.

Також прикладом можуть слугувати системи реакції на основі датчиків, яки отримують інформацію з зовнішнього середовища. Якщо відповідь від керуючого вузлу буде отримано достатньо пізно, то відбудеться ситуація, коли система вчасно не зреагує на подразник, та станеться аварія на підприємстві.

З появою Інтернету речей (IoT) розробляються різні алгоритми штучного інтелекту (AI) різних обчислювальних складностей, які працюють на безперервно породжених потоках даних, що прямують з датчиків. Багато програм IoT, наприклад, для

автомобілів, що самостійно керуються, мають критичний до часу характер та вимагають низьку затримку зв'язку, швидкі обчислення та високу пропускну здатність. Для того, щоб задовольнити такі вимоги, хмарна централізована обчислювальна парадигма розвивалася у напрямку нових перспективних розподілених «Edge» і «Fog» обчислювальних тенденцій.

В рамках даної дипломної роботи ми розглядаємо одну з таких систем, які вимагають швидкої реакції на подразники, та для якої ми будемо проектувати рішення для розгортання архітектури.

В наш час є наступні загальні методи для розгортання бізнес-логіки підприємства хмарному середовищі:

1. Монолітні рішення - це рішення, коли уся необхідна інфраструктура знаходиться в рамках одної оперативної одиниці;
2. Мікросервісна архітектура (описана в пункті 1.2.1);
3. Безсерверні обчислення (описані в пункті 1.2.2).

1.2.1. Мікросервісна архітектура

Мікросервісная архітектура - архітектура програмного забезпечення, що базується на сервіс-орієнтованому принципі, набула поширення в середині 2010-х років у зв'язку з розвитком практик гнучкої розробки, орієнтована на взаємодію мікросервісів (невеликих, легко змінюваних і слабо пов'язаних модулів) [2].

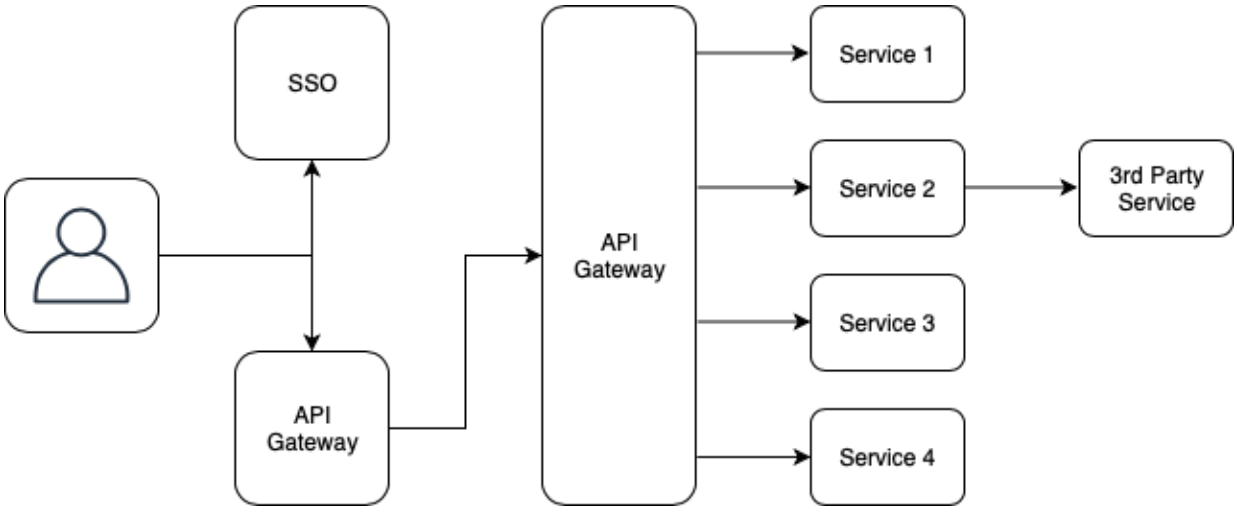


Рис.1.1. Приклад мікросервісної архітектури

Мікросервісна архітектура пропагує розробку маленьких, слабо пов'язаних і легко змінюваних модулів, що звуться мікросервісами. Такі модулі легше тримати в порядку, легше тестувати, легше вести версіонування. Мікросервісна архітектура використовується для розробки розподілених обчислювальних програмних систем. Для реалізації мікросервісної архітектури використовуються концепції віртуалізації, та технології контейнеризації [2].

В мікросервісній архітектурі системи будують з компонентів, які виконують елементарні функції і взаємодіють між собою за допомогою HTTP протоколу, на відміну від традиційних варіантів сервіс-орієнтованої архітектури де модулі можуть бути достатньо складними самі по собі, а взаємодія між ними часто покладається на стандартизовані великовагові протоколи (такі, як SOAP, XML-RPC).

Для того щоб було простіше додавати, або змінювати функції в системі в будь-який час — мікросервісна архітектура націлена на зменшення ступенів зчеплення і збільшення зв'язності за допомогою підвищення гранулярності модулів [2].

Властивості, характерні для мікросервісной архітектури:

1. Розраховується, що модулі будуть розроблені на скільки це можливо простими для покращення незалежності розгортання та заміни (оновлення) кожного з них;
2. Модулі організовані навколо функцій: мікросервіс по можливості виконує тільки одну досить елементарну функцію;
3. Для реалізації модулів можна використовувати різні мови програмування, фреймворки. Існує широкий вибір для розгортання модулів в різних середовищах віртуалізації або контейнеризації, в цілому модулі є незалежними від операційних систем;
4. Однорангові залежності між мікросервісами;
5. Філософія мікросервісів фактично копіює філософію Unix, згідно з якою кожна програма повинна «робити щось одне, і робити це добре» і взаємодіяти з іншими програмами простими засобами: мікросервіси мінімальні і призначаються для єдиної функції. Основні зміни в зв'язку з цим накладаються на організаційну культуру, яка повинна включати автома-

тизацію розробки і тестування, а також культуру проектування, від якої потрібно передбачати обхід колишніх помилок, виняток по можливості успадкованого коду (мікросервіси часто замінюють цілком, оскільки їх функції елементарні) [2].

Системи для керування контейнеризованими додатками - найбільш популярне середовище для виконання мікросервісів. Мікросервіси ізолюються в окремі контейнери, або невелику групу контейнерів, які стають доступні зовнішнім споживачам, або іншим мікросервісами через мережу. Для того, щоб забезпечити відмовостійкість, балансування навантаження, використовуються системи оркестрації, які займаються забезпеченням автоматизації, поновлення, розгортання мікросервісів [2].

Але використання мікросервісної архітектури призводять до наступного списку проблем:

1. Мережеві затримки: мікросервісна архітектура вимагає атомізувати модулі та налагодити їх взаємодію по мережі. Якщо мережа не в працездатному вигляді, або перенавантажена це може призвести до зменшення продуктивності системи;
2. Формати повідомлень: не існує єдиного стандарту для комунікації в рамках мікросервісної архітектури, зазвичай завдання стандартизації кладеться на плечі проектувальника конкретного рішення. Мікросервісну архітектуру відчутно важче діагностувати, та виправляти помилки бізнес-логіки;
3. Баланс навантаження і відмовостійкості;
4. Ускладнюється тестування і розробка.

1.2.2. Fog and Egde архітектура

З поширенням концепції інтернету речей для досягнення ефективної взаємодії з кінцем пристроями часом може не вистачати пропускної здатності мережі. До прикладу: якщо системі необхідно розпізнати предмет у відео - то всі дані, що стосуються цього фрагменту необхідно відправити на сервер для обробки інформації. Коли ж в межах локальної мережі це зробити набагато швидше, оскільки зазвичай в ло-

кальних мережах набагато більший пропускний канал. Також буває ситуація, коли необхідно обслуговувати користувача який знаходиться у русі, або немає можливості під'єднатися до локальної мережі.

Архітектура Fog and Edge пропонує використовувати капілярну архітектуру доставки та завантаження сервісів на різні обчислювальні вузли. Для цього використовуються Docker контейнери, що зберігаються у вигляді файлів в певному реєстрі контейнерів. Використовуючи загальний реєстр контейнерів завжди є змога завантажити необхідний для користувача контейнер на певний вузол який знаходиться в географічній близькості до нього. Рівень туману займається тим, що завантажує контейнери з реєстру та доставляє їх на пристрої, які знаходяться ближче до кінцевих користувачів. Таким чином можна використовувати невеликий мікрокомп'ютер на якому буде знаходитися спрощена модель бізнес-логіки, що дозволить давати відповідь користувачу, або можливо реалізувати певний пре-процесинг даних який згодом відправиться в хмарне середовище, але буде займати менший обсяг даних [3].

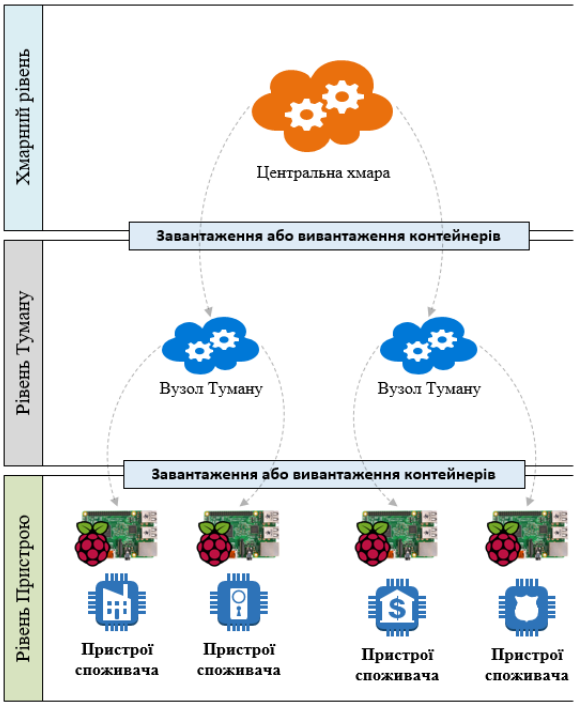


Рис.1.2. Капілярна архітектура

Edge node - обчислювальний пристрій, який має процесорну або багатопроцесорну бортову систему. Прикладами є акумуляторні машини, роботи, смартфони, Raspberry Pi або Arduino. У цьому контексті вузли туману можна розуміти як інфра-

структура хмарних обчислень, яка існує в тісній географічній близькості від кінцевих вузлів. З цією метою різні легкі операційні системи, такі як RancherOS або CoreOS, дозволяють перетворити навіть невеликі пристрої, такі як маршрутизатори, на хмарних постачальників, в цьому випадку вони вважаються обчислювальною інфраструктурою Fog [3].

Обчислювальна модель Edge / Fog - це парадигма обчислень, яка має на меті оптимізувати розумні додатки для розширення їх функціональних можливостей, близьких до географічного розташування пристроїв IoT, а не для далеких центрів обробки даних.

У розширеній Edge / Fog обчислювальній системі, сучасні підходи до розробки програмного забезпечення можуть бути використані для використання легких мікросервісів, упакованих у контейнери, для досягнення високого ступеня автоматизації, розгортання, еластичності та реконфігурації смарт-програм IoT під час виконання. З цією метою з'явилися різні технології управління контейнерами та оркестрування, включаючи Docker, Kubernetes, OpenShift Origin і Swarm. Іншими словами, як тільки ці технології поєднуються з мікросервісною архітектурою, можна досягти високого рівня гнучкості у розробці, розгортанні та переконфігурації додатків [3].

У Таб.2.1 представлено порівняння віртуалізації на базі контейнерів та віртуалізації на основі віртуальної машини.

Таб.2.1. Порівняння технологій віртуалізації і контейнеризації

Характеристика	**Контейнери**	**Віртуальні машини**
Вимоги	Рушій контейнерів напр. Docker	Гіпервізор напр. Xvisor
Вага	Легковісні	Важкі
Розмір	Невеликі	Великі
Час завантаження	Швидкі	Повільні

Технології віртуалізації на базі гіпервізора підтримують автономні віртуальні машини, які є незалежними та ізольованими від хост-машини. Кожен екземпляр вір-

туальної машини має свою власну операційну систему і набір бібліотек, і працює в середовищі емуляції гіпервізора. Це робить керовані віртуальні машини важкими. Для реалізації даної концепції використовуються контейнерні технології, що вважаються більш легкими. У порівнянні з віртуальними машинами, використання контейнерів не вимагає операційної системи для завантаження.

Віртуальні машини використовують багато ресурсів і як правило їх неможливо легко розгорнути на невеликих серверах або пристроях з обмежених ресурсами, таких як Raspberry Pi, або маршрутизатори. Контейнери ж можна використовувати гнучко і в таких рамках, оскільки їхня природа передбачає використання ресурсів основної ОС. Розгортання служб на базі контейнерів під час виконання може бути досягнуто швидше, ніж у віртуальних машинах. Завдяки цим перевагам різні контейнерні платформи віртуалізації, такі як Google Container Engine (GCE) і Amazon EC2 Container Service (ECS), стали альтернативами віртуалізації на основі гіпервізора [4].

Для реалізації даної архітектури використовують два типи застосунків:

- Мастер-сервери - Сервери які відповідають за реєстрацію кінцевих вузлів, виконують роль Service Discovery, тобто оголошення сервісів в рамках одного контуру;
- Сервери-виконавці - Сервери, що реєструються в реєстрі мастер-сервера та виконують прикладні задачі користувачів. Бувають двох типів: хмарні (ті, що розгортаються у хмарних середовищах) та вбудовані (ті, що розгортаються в мікрокомп’ютерах).

1.2.3. Безсерверні обчислення

Безсерверні обчислення (англ. Serverless computing) - стратегія організації платформних хмарних послуг, при якій хмара автоматично і динамічно управляє виділенням обчислювальних ресурсів в залежності від призначеної для користувача навантаження. Основне завдання такої стратегії - реалізація шаблону «функція як послуга», при якому для виконання кожного запиту (виклику функції) створюється окремий контейнер або віртуальна машина, яку знищують після виконання [5].

Користувачам даної методології не потрібно виділяти, налаштовувати сервери оскільки всі налаштування обчислювальних ресурсів, які необхідні для роботи коду керуються хмарою. В такому випадку запуск коду на виконання відбувається на вимогу користувача, або за певною подією. Деяка частина безсерверного коду може інтегруватися в додатки, що побудовані на мікросервісній архітектурі в якості окремих мікросервісів. Порівняння з технологіями контейнеризації та віртуалізації представлене в Таб.2.2.

Таб. 2.2. Порівняння попередніх технологій і безсерверних обчислень

Характеристика	**Контейнери**	**Віртуальні машини**	**Безсерверні обчислення**
Вимоги	Рушій контейнерів напр. Docker	Гіпервізор напр. Xvisor	Середовище виконання
Вага	Легковісні	Важкі	Надлегкі
Розмір	Невеликі	Великі	Найменші
Час завантаження	Швидкі	Повільні	Найшвидші

РОЗДІЛ 2
АПАРАТНО-ПРОГРАМНИЙ НАБІР ТЕХНОЛОГІЙ

2.1. Обране обладнання для реалізації Edge-вузлу та обґрунтування

Зазвичай Edge-вузол інтегрується напряму в інфраструктуру приміщення, для того, щоб бути в єдиній локальній мережі з користувачами, або сенсорами. Для реалізації даного проекту необхідно обрати пристрій, який матиме невеликий розмір, можливість під'єднання за допомогою RJ-45, та інтерфейс для під’єднання різного роду периферійних приладів, таких як веб-камера, зовнішній накопичувач даних, тощо.

Було розглянуто наступні можливі для цього пристрої:

1. Raspberry Pi 4;
2. Orange Pi One Plus;
3. Intel Quark (Edison);

Було надано перевагу Raspberry Pi 4, У зв'язку з наявністю експертизи з даним мікро-комп’ютером, та найкращим співвідношення ціни до можливостей. Характеристики представлені в Таб.2.1 та Таб.2.2.

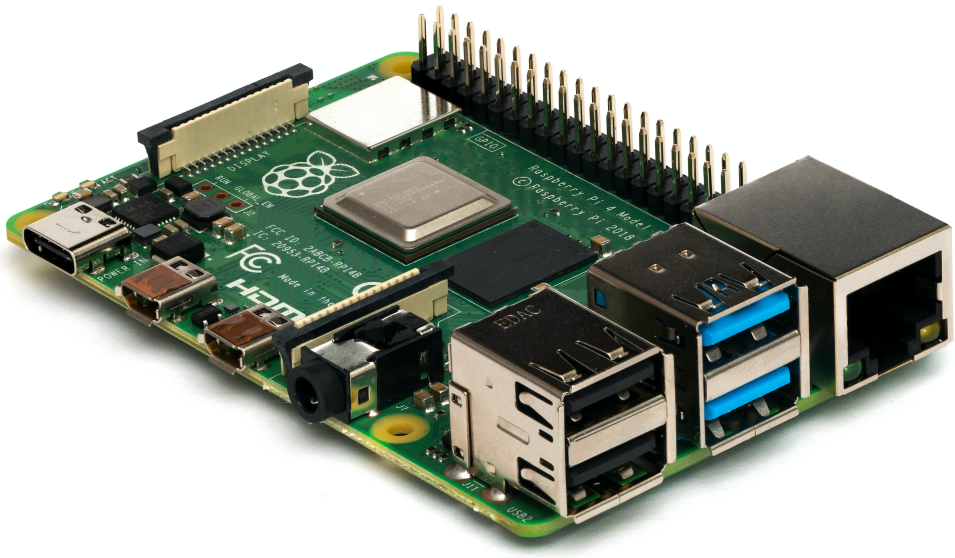


Рис.2.1. Raspberry Pi 4

Таб. 2.1. Характеристики Raspberry Pi 4

Характеристики RPi4	
Чіп	Broadcom BCM2711
Процесор	4 ядерний Cortex-A72 (ARM v8) 64-bit
Тактова частота	1.5 GHz
Графічний процесор	VideoCore VI
ОЗП	4096 MB
Пам’ять	microSD карта

Таб. 2.2. Інтерфейси вводу/виводу Raspberry Pi 4

Інтерфейси вводу-виводу	
USB 2.0	2
USB 3.0	2
Micro HDMI	2
3,5mm Jack	1
GPIO	40
Ethernet	1
Wifi	802.11b/g/n
Bluetooth	4.1

Raspberry Pi 4 підтримує велику кількість можливих ОС написаних на ядрі Unix, такі як: Raspbian, RISK OS, OSMC, Arch Linux, які можна завантажити і повністю безкоштовно встановити на пристрій [6].

Ціна Raspberry Pi 4 є оптимальною для використання в даному проекті і становить 35$ та набір системних компонентів Raspberry Pi 4 повністю задовільняє вимоги для реалізації проекту, а саме:

- Компактний розмір;

- Низьке енергоспоживання завдяки ARM архітектурі;
- Операційна система з підтримкою Docker.

Також до переваг даного пристрою необхідно віднести відсутність аналогів в даній ціновій категорії з подібним набором характеристик.

Для реалізації проекту було обрано такі інтерфейси передавання даних:

1. WiFi - загальновживана назва для стандарту IEEE 802.11 передачі цифрових потоків даних по радіоканалах. Поточні реалізації Wi-Fi дозволяють отримати швидкість передачі даних понад 100 Мбіт/с, при цьому користувачі можуть переміщуватися між точками доступу на території покриття мережі Wi-Fi.
2. RJ-45 — фізичний інтерфейс, який загалом використовується для з'єднання комп'ютерних мереж за допомогою звитої пари через мережевий комутатор, або при створенні мережі з двох комп'ютерів до один одного через мережеву карту.

2.1.2. Хмарний надавач послуг

Для реалізації хмарного середовища виконання ми провели аналіз найкращих сервісів, що надають послуги хмарних обчислень.

Було розглянуто наступні варіанти:

1. Amazon Elastic Compute Cloud
2. Google Compute Engine
3. DigitalOcean Droplets
4. Azure Virtual Machines
5. Linode

Ми надали перевагу рішенню від Amazon, оскільки у амазон найбільша кількість документації, найвищий відсоток робочого часу, є датацентри на всіх континентах, та є безкоштовний період для користування.

Amazon Elastic Compute Cloud (Amazon EC2) - це веб-сервіс, що надає безпечні масштабовані обчислювальні ресурси в хмарі. Він спрощує проведення обчислень в хмарі в масштабі всього Інтернету [7].

Веб-інтерфейс сервісу Amazon EC2 дозволяє отримати доступ до обчислювальних ресурсів і налаштувати їх з мінімальними зусиллями. Він надає користувачам повний контроль над обчислювальними ресурсами, а також перевірене обчислювальне середовище Amazon для роботи. Amazon EC2 дозволяє скоротити час, необхідний для створення і завантаження нових інстансів сервера, до декількох хвилин, і забезпечує можливість швидко масштабуватися в будь-якому напрямку з урахуванням мінливих вимог до обчислювальних ресурсів. Amazon EC2 змінює економічну складову процесу обчислень, надаючи можливість платити тільки за використовувані ресурси. Amazon EC2 дозволяє розробникам уникати поширених хибних сценаріїв і створювати відмовостійкі додатки [7].

2.2. Програмний набір

Для вибору програмного набору технологій ми керувалися наступними принципами: наявність експертизи в даній технології; наявність документації, та її повнота; наявність широкої спільноти розробників, що користуються даною технологією. До програмного набору входять як готові рішення, так і технології, що дозволяють будувати своє рішення за допомогою певної мови програмування.

Підчас дослідження кінцевої плаформи розгортання сервісів було встановлено наступні технології/програмні каркаси як:

- Node.JS + Sequelize ORM + fastify;
- Java (Spring) + Hibernate ORM + Tomcat;
- C# + Entity Framework + IIS.

Оскільки для розробки застосунків реального часу краще всього підходить подійно-орієнтований підхід, що дозволяє тримати високі навантаження (від 10 тис. запитів), то середовища виконання коду повинні мати в собі механізми асинхронного виконання.

У випадку стеку побудованому на основі Node.JS такий механізм забезпечує комбінація бібліотеки libuv та рушій V8 від Google [8].

У випадку стеку побудованому на основі C# такий механізм вбудований в можливості мови починаючи з версії 5.0.

У випадку стеку побудованому на основі Java такий механізм забезпечується фреймворком DataKernel.

Основною мовою програмування в проекті ми обрали JavaScript, оскільки ми розробляємо архітектуру, що може інтегруватися в будь-який проект, то краще використовувати найбільш популярні мови програмування. Для Java Script та існує велика кількість вже готових модулів, які можна використовувати в своїх проектах цілком безкоштовно. До таких модулів належать: fastify, sequelize, lodash.

Також ми виділили наступні причини, чому ми надали перевагу стеку з використанням Node.JS:

- Оскільки мова JavaScript інтерпретується, то немає необхідності робити збірку для конкретного пристрою. Код який написаний мовою JavaScript можливо виконати на різних платформах без внесення змін;
- Середовище Node.JS не потребує багато ресурсів та серверу додатків (IIS, Tomcat), що робить його привабливішим ніж розглянуті варіанти вище. Особливо актуально це стає при використанні обчислювальних пристроїв з обмеженими ресурсами, такими як Raspberry Pi;
- За допомогою мови JavaScript можливо реалізувати прототип у найбільш стислий термін. Такий прототип цілком можна використовувати в робочих умовах;
- За умови успішної реалізації прототипу, використовуючи можливості сучасних інтегрованих середовищ розробки, проект можливо перевести на TypeScript - мову програмування рівня підприємства в короткий термін.

Node або Node.js - програмна платформа, заснована на рушію V8 (здійснює трансляцію JavaScript в машинний код), що перетворює JavaScript з вузькоспеціалізованої мови в мову загального призначення. Node.js надає свій API, що написаний на C++, через який можна взаємодіяти з пристроями введення-виведення, підключати зовнішні (native) бібліотеки, що написані на інших технологіях, та надавати можливість інженерам взаємодіяти з ними, або реагувати на виклики з Java Script коду.

За допомогою цієї технології можна розробляти додатки для Web (express, koa, fastify), мобільних пристроїв (Phone Gap) і навіть настільні додатки (за допомогою node-gui, NW.js, AppJS, Electron для всіх популярних ОС). Також є можливість програмувати мікроконтролери (наприклад, tessel, espruino, iskraJS). В основі Node.js лежить асинхронне, подієво-орієнтоване програмування з неблокуючим введенням / виведенням[8].

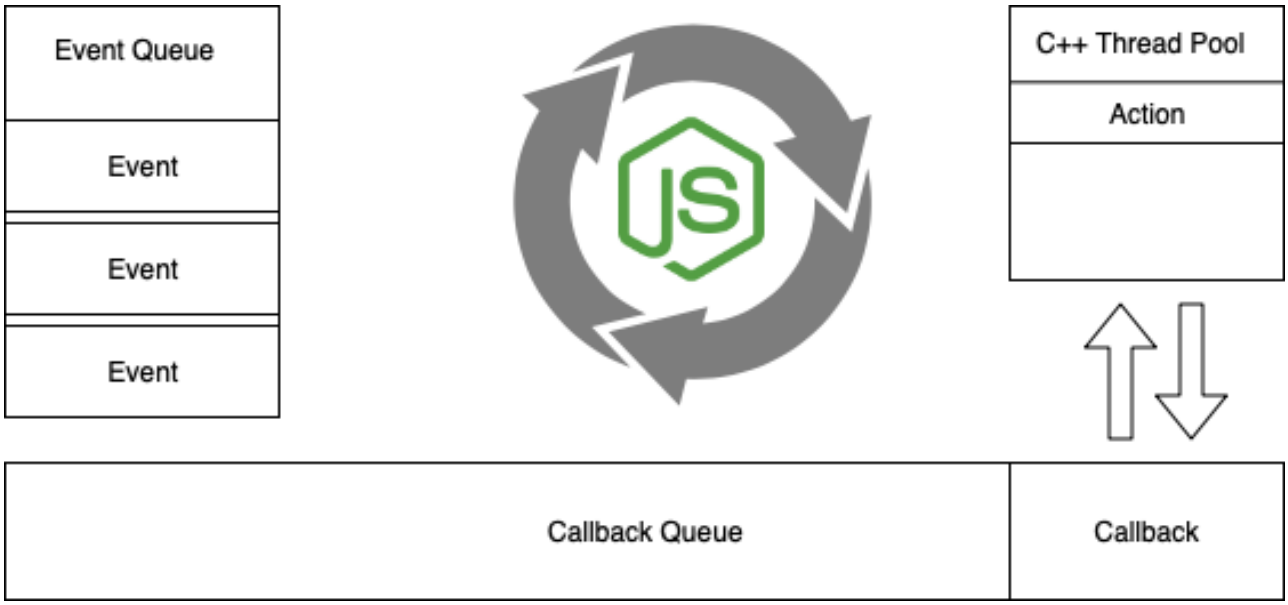


Рис.2.2. Архітектура Node.JS

Платформа окрім роботи із серверними скриптами для веб-запитів, також використовується для створення клієнтських та серверних програм.

Для забезпечення обробки великої кількості паралельних запитів у Node.js використовується асинхронна модель виконання коду, заснована на обробці подій в неблокуючому режимі та визначенні обробників зворотніх викликів. Всі системні виклики, що спричиняють блокування, виконуються всередині пулу потоків і потім, як і обробники сигналів, передають результат своєї роботи назад через неіменовані канали [8].

Оскільки Node.JS є монопоточним, та використовує принципи асинхроного програмування, то необхідно розуміти як з ним працювати коректно. У синхронному коді кожна операція чекає закінчення попередньої. Тому вся програма може «зави-

снути», якщо одна з команд виконується дуже довго. Асинхронний код прибирає операцію, яка блокує основний потік програми, так що основний потік не блокується, і програма може виконувати інші операції.

Всі важкі операції переносяться в сторонні процеси-воркери. За допомогою бібліотеки node-gyp можливо писати власні модулі які будуть виконуватися у окремому потоці, тим самим не навантажуючи основний потік. Для цього необхідно дотримуватися зазначеної структури модуля у якому будуть прописані всі методи життєвого циклу додатку [8].

Асинхронне програмування успішно вирішує безліч завдань. Одна з найважливіших — користувач може взаємодіяти з програмою поки виконується інше завдання [9].

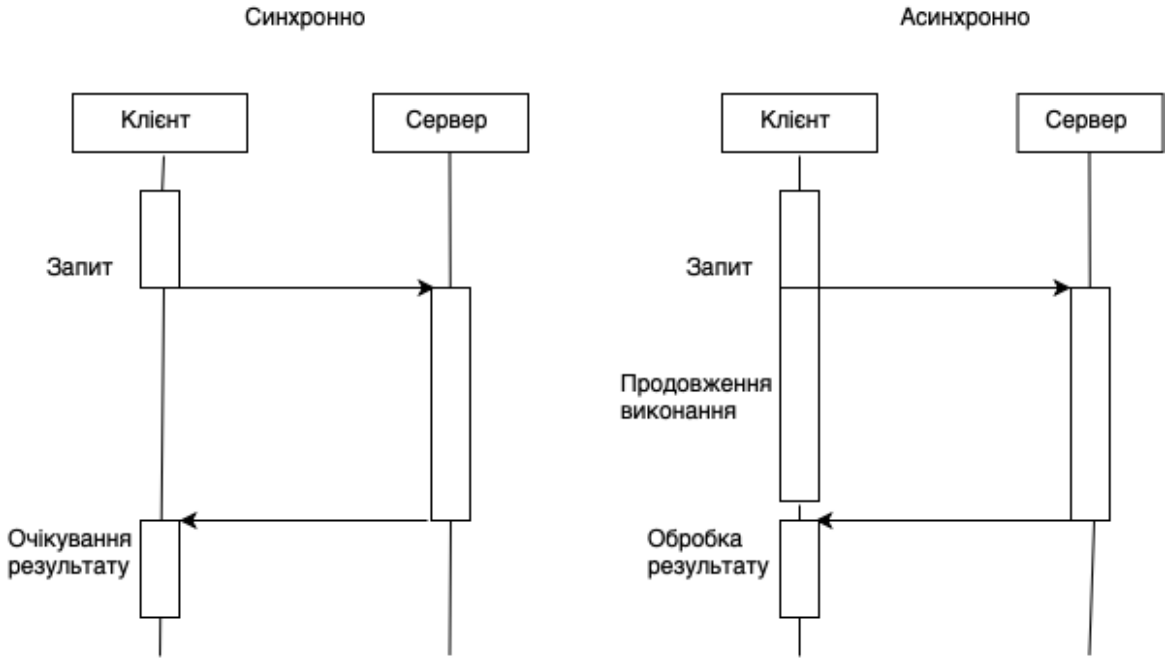


Рис.2.3. Різниця між синхронним і асинхронним виконанням операцій

2.3. React

React — відкрита JavaScript бібліотека для створення інтерфейсів користувача, яка покликана вирішувати проблеми часткового оновлення вмісту веб-сторінки, з якими стикаються в розробці односторінкових застосунків. Розробляється Facebook, Instagram і спільнотою індивідуальних розробників [10].

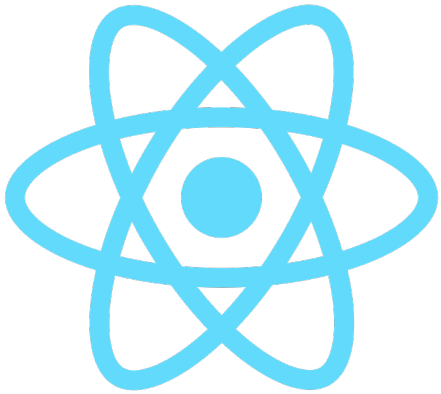


Рис.2.4. Бібліотека React

React дозволяє розробникам створювати великі веб-застосунки, які використовують дані, котрі змінюються з часом, без перезавантаження сторінки. Його мета полягає в тому, щоб бути швидким, простим, масштабованим. React обробляє тільки користувацький інтерфейс у застосунках. Це відповідає видові у шаблоні модель-вид-контролер (MVC), і може бути використане у поєднанні з іншими JavaScript бібліотеками або в великих фреймворках MVC, таких як AngularJS. Він також може бути використаний з SSR на основі надбудов, щоб піклуватися про частини без користувацького інтерфейсу побудови веб-застосунків. Як бібліотеку інтерфейсу користувача React найчастіше використовують разом з іншими бібліотеками, такими як Redux [10].

2.4. Amazon Linux AMI

Amazon Linux AMI - це підтримуваний і обслуговується образ Linux, що надається Amazon Web Services для використання в Amazon Elastic Compute Cloud

(Amazon EC2). Він призначений для забезпечення стабільної, безпечної і високопродуктивної середовища виконання програм, які потребують в Amazon EC2 [11].

Рис.2.4. Логотип Amazon Linux AMI

Образ підтримує можливості нових типів інстансів EC2 і містить пакети для простої інтеграції з AWS. AWS регулярно випускає оновлення безпеки і робочі виправлення для всіх інстансів, на яких використовується AMI Amazon Linux. AMI Amazon Linux надається користувачам Amazon EC2 безкоштовно [11].

2.5. RaspbianOS

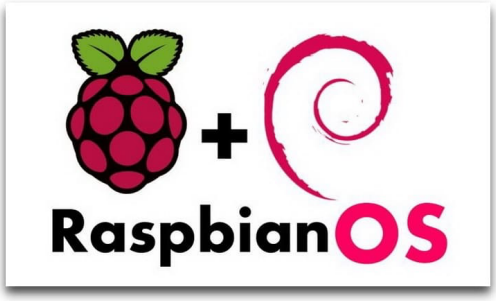


Рис.2.5. Логотип Raspbian OS

Raspbian [12] - це офіційна операційна система для Raspberry Pi, вона розроблена спеціально для цього пристрою і має все необхідне програмне забезпечення. Raspbian заснована на ARM версії Debian 8 Jessie і містить такі програми за замовчуванням - офісний пакет LibreOffice, веб-браузер, поштовий клієнт - Claws Mail, легке оточення робочого столу, а також інструменти для програмування.

РОЗДІЛ 3
ОГЛЯД ЗАПРОПОНОВАНОГО РІШЕННЯ

3.1. Інфраструктура системи

Ми розробили систему-прототип, що складається з двох сервісів: «Ядро системи» та «Обслуговувач користувача», та модулю, який може інтегруватися в додатки кінцевих користувачів (Рис.3.1). Ядро системи повинно бути розміщене на потужностях, що мають постійне підключення до мережі інтернет, та стабільну IP адресу. В той же час «обслуговувач» може бути розміщений на всіх платформах, що підтримують технологію Node.JS (виділений сервер, Raspberry Pi, віртуальна машина, контейнер). Повну діаграму запропонованої інфраструктури представлено в Додатку А.

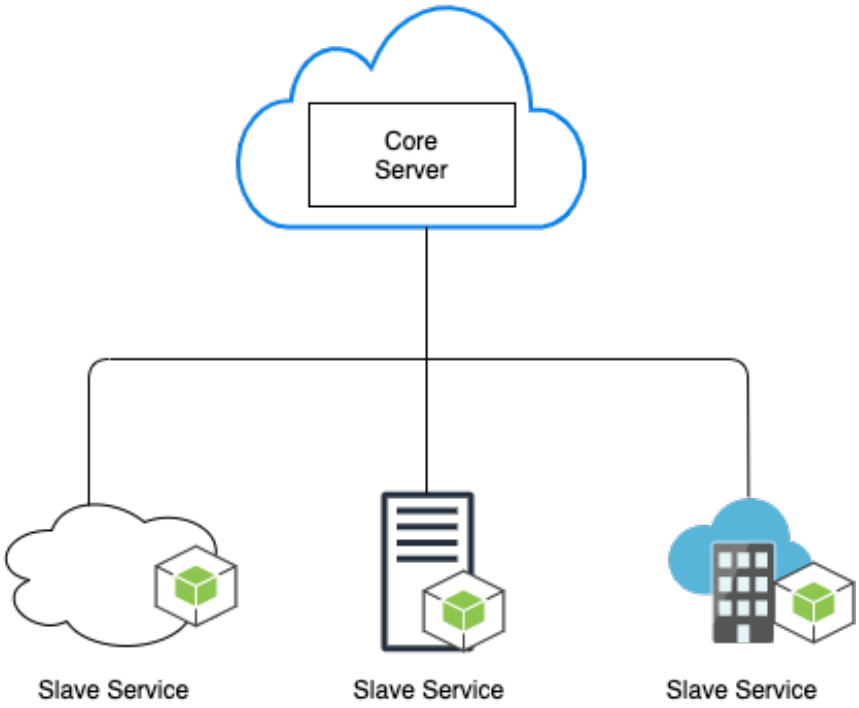


Рис.3.1. Високорівнева архітектура системи

На відміну від архітектури «Fog and Edge» бізнес логіка, що необхідна користувачу не запускається в окремому контейнері. Замість цього вся бізнес логіка переноситься в «обслуговувач користувача» разом з конфігурацією яка необхідна користувачу. Для того, щоб зменшити вірогідність втручання в безпеку системи, слід

обмежити технології, що можуть бути використані (драйвери БД, драйвери систем нотифікації, нативні розширення систем криптографії, тощо) до певного списку. Для формування даного списку, можна використати статистику пакетного менеджера npm.

Нижче у Таб.3.1. приведене порівняння підходів до написання сервісів та їх вимоги до апаратної частини.

Таб.3.1. Порівняння технологій віртуалізації і контейнеризації

Характеристика	**Безсерверні функції**	**Контейнери**	**Віртуальні машини**
Вимоги	Node.JS	Рушій контейнерів напр. Docker	Гіпервізор напр. Xvisor
Вага	Надлегкі	Легковісні	Важкі
Розмір	Найменший	Невеликі	Великі
Час завантаження	Моментальний	Швидкі	Повільні
Сервісні одиниці	Мільйони	Десятки	Десятки
Швидкодія на CPU intensive задачах	Повільна	Висока	Висока
Здатність до часткового завантаження функціональності	Так	Ні	Ні

Як ми можемо побачити з таблички приведеної вище - використання безсерверних функцій найоптимальніший спосіб утилізувати апаратні ресурси за умови легковажності конкретної операції. Для того щоб опрацьовувати великий об'єм інформації в одному пакеті, слід використовувати рішення які є найближчими до виділеного сервера. Притому безсерверні функції дозволяють відокремити набір функціональності використовуючи кожну функцію як окрему операцію, в той же час використовуючи мікросервісний підхід - доведеться завантажувати повноцінний сервіс.

Відмова від Docker контейнерів допомагає економити ресурси системи, в той же час дозволяючи обслуговувати набагато більшу кількість користувачів.

Оскільки використовується безсерверний підхід, ізоляція контексту виконання виконується на рівні кожної функції, це означає, що сусідні користувачі не можуть вплинути на хід виконання функції один одного.

Для комунікації всередині системи ми будемо використовувати HTTP (HyperText Transfer Protocol) - протокол прикладного рівня, за допомогою якого відбувається передача даних. На сьогоднішній день протокол HTTP використовується в Інтернеті практично скрізь з метою отримання інформації з веб-сайтів. Використання HTTP протоколу обумовлене його широкою розповсюдженістю. На відміну від інших протоколів комунікації цей використовується частіше в системах підприємств, тому якщо ми повинні забезпечити можливість використання нашої системи для будь-якого вже існуючого додатку, то це єдиний варіант.

Нашу систему-прототип ми спроектували з розрахунку на те, що вона сама себе буде конфігурувати в залежності від потреб кінцевих користувачів. Система буде розуміти як себе відконфігурувати враховуючи навантаженість на певну частину бізнес-логіки (інформацію вона буде брати з кількості запитів на конкретні ендпоінти), та враховуючи потреби користувачів, щодо бізнес-логіки, якою вони користуються.

Ми прийняли рішення використовувати API Gateway[5] для того, щоб спростити комунікацію клієнтському застосунку. При використанні такого підходу та сервісів авторизації - клієнтські за стосунки можуть взаємодіяти з різними сервісами як з одним. Тобто фактично API Gateway склеює ендпоінти в межах одного DNS домену.

Для того, щоб уникнути єдиної точки відмови необхідно користуватися принципом розподіленої сесії та дублювання важливих сервісів. Як ми можемо переконатися з високорівневої архітектури запропонованого прототипу, таким сервісом є ядро. Ми прийняли рішення, що цей сервіс буде запущено в декількох екземплярах та в різних дата-центрах, щоб забезпечити стабільність роботи і постійну доступність. Для того, щоб синхронізувати стан у цих екземплярах сервісу «Ядро» необхідно використовувати базу даних, що надає послуги підписки-

публікації. В якості даної бази даних ми обрали Redis, оскільки саме з цією базою даних у нас найбільше експертизи.

Для того, щоб мати можливість перенаправити користувача до робочого екземпляру сервісу «Ядро» було прийнято рішення використовувати NGINX якості балансувальника навантаження. Таким чином якщо в нас відбудеться ситуація, коли один із екземплярів сервісу буде відімкнено, то робота інших сервісів не постраждає.

3.2. Сервіс «Ядро»

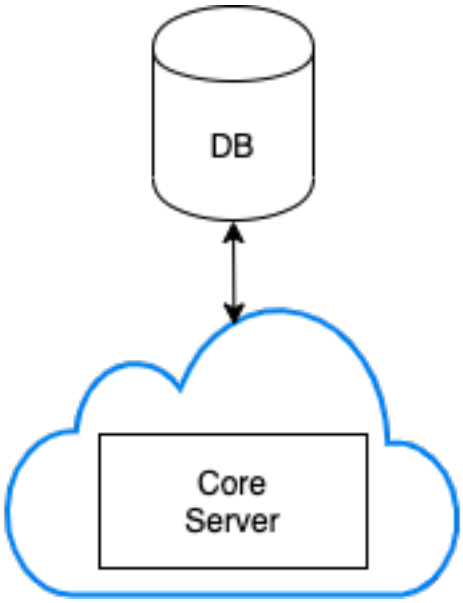


Рис.3.2. Високорівнева архітектура сервісу «Ядро»

Це головний сервіс системи який спілкується з сервісами «обслуговувачами», та взаємодіє з кінцевими користувачами, що завантажують свою бізнес-логіку до системи (Рис.3.2). Критично важливо, щоб вся бізнес логіка була написана в функціональному стилі. Для цього слід користуватися засобами Node.JS, та правильно зконфігурувати набір залежностей для власної системи. Сервіс підключений до бази даних, В якій він зберігає всю бізнес-логіку користувача. Для того, щоб система мала змогу розподілити бізнес логіку між серверами необхідно в JWT токен авторизації дописати ID функцій які він може викликати. Цей процес можна організувати на основі рольової системи, де кожній із ролей дозволено

викликати певні функції. Тому немає змісту завантажувати функції, які користувач навіть не зможе викликати на обчислювальний засіб.

Як вже було наведено вище, усі функції повинні бути чистими. Функція є чистою, якщо вона задовільняє двом умовам:

1. Функція, що викликається від одних і тих же аргументів, завжди повертає однакове значення;
2. Під час виконання функції не виникають побічні ефекти.

Слід пояснити перше правило - якщо викликається певна функція sum(2, 3), то очікується, що результат завжди буде дорівнювати 5.

Друге правило - не допускається використання побічних ефектів - є більш широким за своєю природою. Побічний ефект - це зміна чогось відмінного від функції, яка виконується в поточний момент. Зміна змінної поза функцією, виведення в консоль, виклик винятку, читання даних з файлу - все це приклади побічних ефектів, які позбавляють функцію чистоти. Якщо виклик функції не змінить нічого «зовні», то цю функцію можна використовувати в будь-якому сценарії. Це відкриває дорогу конкурентному програмування і багатопотоковим додатків [13].

Для комунікації між «ядром» та «обслуговувачем» ми обрали REST-протокол.

REST [2] - скорочення від Representational State Transfer, що можна перекласти як «передача репрезентативного стану». Це стиль проектування розподілених систем за допомогою обмежень. Центральною абстракцією в REST є ресурс. А головні обмеження виглядають так:

- Клієнт-серверна модель;
- Взаємодія без збереження стану;
- Логічний інтерфейс.

Ресурс - це представлення віртуального об'єкту, реального об’єкту, чи колекції об'єктів. Взагалі, ресурс може бути чим завгодно, розробник API вирішує що в нього буде ресурсом.

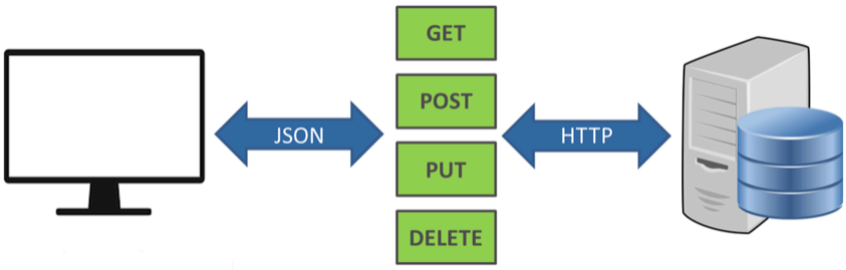


Рис.3.3. Відображення комунікації на основі REST протоколу

В REST кожен ресурс повинен бути унікальним. Тому ми присвоюємо їм ID. Для нашої системи /node/1 та /node/2 це два різних кінцевих вузла з ID 1 та 2. Також ресурси можуть бути вкладеними. Наприклад, URI клієнта 2 в першому кінцевому вузлі буде виглядати так /node/1/client/2. Повну діаграму запропонованої інфраструктури представлено в Додатку А.

3.2.1 Складові частини сервісу «Ядро»

«Ядро» складається з таких сервісів як:

1. **Проксі-сервіс** - відповідає за переспрямування клієнта на кінцевий вузол який було обрано як «найоптимальніший» за такими критеріями:
   - Географічна відстань між клієнтом та кінцевим вузлом;
   - Тип з’єднання кінцевого вузлу;
   - Тип кінцевого вузлу (хмара, мікрокомп’ютер).
2. **Сервіс кінцевих вузлів** - відповідає за реєстрацію, активацію кінцевих вузлів моніторинг активності зареєстрованих кінцевих вузлів. Сервіс під-’єднаний до роутера;
3. **Сервіс бази даних** - є наслідником інтерфейсу менеджера бази даних та реалізує фунціонал роботи з базою даних. Для прототипу було використано базу даних PostgreSQL.

Приклад коду сервісу ядра представлений в Додатку Б. Класова діаграма розробленого прототипу сервісу «Ядро» знаходиться в додатку Д.

У цьому сервісі використовуються такі періодичні задачі:

1. Відвантаження функцій - виконується щохвилини. Виконує завдання відвантаження цільових функцій з пам’яті «обслуговувачів». Працює тільки для кінцевих вузлів типу «мікрокомп’ютер». Потрібна для того, щоб звільнити завантажені кінцеві вузли до яких протягом останніх 10 хвилин не було надіслано жодного запиту. Після виконання завдання - виконується сповіщення кінцевих вузлів про приняте рішення про відвантаження цільового сервісу.
2. Скасування реєстрації кінцевих вузлів - виконується щохвилини. Виконує завдання видалення кінцевого вузлу зі списку активних вузлів. Потрібна для того, щоб видалити зі списку вузли, від яких не прийшло сповіщення про активність протягом останніх двох хвилин. Після виконання завдання - всі кінцеві вузли які було видалено мусять знову пройти процес реєстрації.

3.2.2 Опис таблиць бази даних

Для зручної розробки сервісу — необхідно забезпечити роботу з даними в термінах класів, а не таблиць даних, і, навпаки, перетворити терміни і дані класів в дані, придатні для зберігання в СКБД. Необхідно також забезпечити інтерфейс для CRUD-операцій над даними. Загалом, необхідно позбутися від необхідності писати SQL-код для взаємодії в СУБД, тому ми прийняли рішення використовувати методологію ORM;

ORM або Object-relational mapping — це технологія програмування, яка дозволяє перетворювати несумісні типи моделей в ООП, зокрема, між сховищем даних і об'єктами програмування. ORM використовується для спрощення процесу збереження об'єктів в реляційну базу даних і їх вилучення, при цьому ORM сама піклується про перетворення даних між двома несумісними станами. Для нас зручність використання ORM-інструментів значною мірою полягає в тому, що вони покладаються на метадані бази даних і об'єктів, так що об'єктам нічого не потрібно знати про структуру бази даних, а базі даних - нічого про те, як дані організовані в додатку. Також

потрібно відзначити, що ORM забезпечує повне розділення завдань в добре спроектованих додатках, при якому і база даних, і додаток можуть працювати з даними кожен у своїй вихідній формі.

Приклад класу даних можна побачити в Додатку В.

У таблицях 3.2 - 3.5 наведена детальна інформація про структуру таблиць бази даних. Варто відзначити, що таблиці було сформовано з розрахунку на принцип мінімальної достатності. В реальних проектах ці таблиці будуть відрізнятися.

Таб.3.2. Структура таблиці «Користувач»

Ім'я поля	**Тип поля**	**Опис поля**
id	SERIAL	Первинний ключ
Login	VARCHAR	Ім'я облікового запису для входу
Password	VARCHAR	Пароль для входу (шифрується за допомогою bcrypt)

Таб.3.3. Структура таблиці «Контекст»

Ім'я поля	**Тип поля**	**Опис поля**
Id	SERIAL	Первинний ключ
User_id	INTEGER	Посилання на користувача, який створив цей контекст
Name	VARCHAR	Ім'я контексту для відображення користувачеві
Value	VARCHAR	Значення контексту

Таб.3.4. Структура таблиці «Змінна середовища»

Ім'я поля	**Тип поля**	**Опис поля**
Id	SERIAL	Первинний ключ
User_id	INTEGER	Посилання на користувача, який створив цей контекст
Name	VARCHAR	Ім'я змінної середовища

Value	VARCHAR	Значення контексту

Таб.3.5. Структура таблиці «Змінна середовища»

Ім’я поля	**Тип поля**	**Опис поля**
Id	SERIAL	Первинний ключ
User_id	INTEGER	Посилання на користувача, який створив цей контекст
Name	VARCHAR	Ім’я змінної середовища для відображення користувачеві
Value	VARCHAR	Значення контексту
Route	VARCHAR	Посилання для виконання
Method	VARCHAR	Тип методу. Припустимі значення: GET, POST, PUT, DELETE
Context_id	INTEGER	Посилання на контекст виконання

Також для того, щоб забезпечити розподілену сесію використовується модель, що зображена в Таб.3.6., яка розміщується в базі даних, що знаходиться в пам’яті (Redis, MongoDB):

Таб.3.6. Структура моделі «Вузел»

Ім’я поля	**Тип поля**	**Опис поля**
Id	SERIAL	Первинний ключ
Ip	STRING	IP вузлу
Name	STRING	Ім’я вузлу
Type	STRING	Тип вузлу. Можливі значення: CLOUD, RPI
Last_update_timestamp	DATETIME	Посилання для виконання
Lambdas_list	INTEGER[]	Колекція ID функцій на цьому вузлі
RPS	INTEGER	Кількість запитів на секунду
Loaded_timestamp	DATETIME	Час завантаження даного вузла

3.2.3. Список використаних модулів

Таб.3.7. Використані модулі в сервісі «Обслуговувач»

Назва	**Версія**	**Опис**
axios	0.19.2	Бібліотека, що дозволяє робити запити по протоколу HTTP.
chalk	4.1.0	Бібліотека для форматування виводу. Використовується для того, щоб покращити вигляд журналу подій додатка.
cuid	2.1.8	Бібліотека для генерація випадкових стрічок.
fastify	2.7.0	Веб-сервер, що сумісний з інтерфейсом express.js, проте менш вибагливий до ресурсів системи.
gulp	3.4.1	Бібліотека для автоматизації рутинної роботи в проекті та налаштування конфігурації.
Sequelize	6.6.2	Бібліотека для взаємодії з базами даних, що повністю підтримує транзакції, методи повільної загрузки, валідації.
Redis	3.1.2	Бібліотека для взаємодії з базами даних, що зберігають дані в оперативній пам'яті у вигляді пари ключ-значення.
Moment.js	2.15.1	Бібліотека для роботи з датами.
Knex.js	0.95.0	Бібліотека для створення складних запитів до баз даних.
async	3.2.0	Бібліотека яка спрощує роботу з асинхронно виконуваним кодом.
lodash	Latest	Бібліотека що надає доступ до часто використовуваних алгоритмів.
Commander	7.2.0	Засіб для створення інтерфейсу командної стрічки. Дозволяє виконувати рутинні завдання.

3.3. Сервіс «Обслуговувач»

Цей сервіс використовується для того, щоб надавати послуги користувачеві. Він тримає комунікацію з ядром системи та завантажує в свою оперативну пам’ять бізнес-логіку користувача. За допомогою попередньої конфігурації він здатний виконувати функції з заданим контекстом.

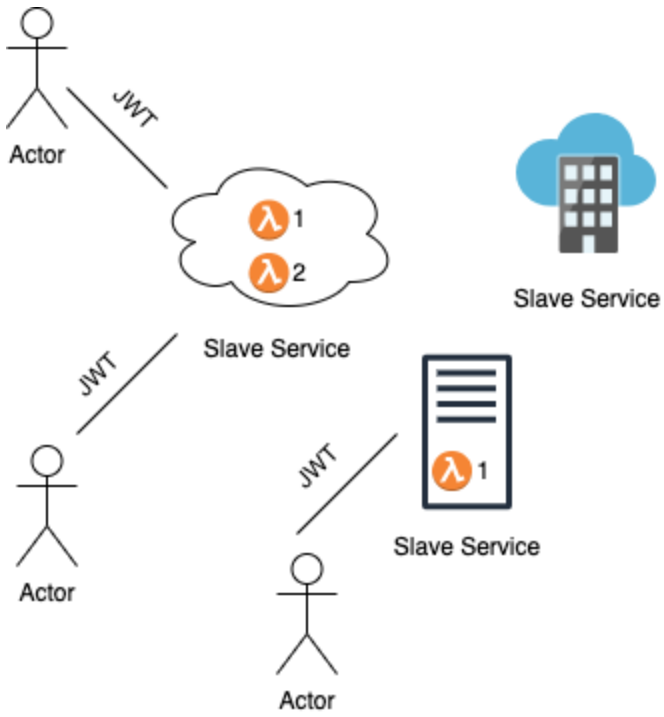


Рис.3.4. Комунікація між користувачами та «обслуговувачами»

Для того, щоб забезпечити максимальну продуктивність даного сервісу необхідно використовувати такі підходи як: багатопоточне програмування та асинхронна модель подій.

За збереження функцій пам'яті відповідає сервіс контейнер функції. Він відповідальний за інвалідацію, завантаження, вивантаження, облік функцій в сервісі.

Якщо на апаратному засобі більше ніж одне ядро центрального процесора, то відбувається розподіл навантаження на окремі потоки. Для цього, коли піднімається процес даного сервісу він піднімає один потік в якості контрольного центру, а на всі інші вільні потоки процесора (вони можуть сприйматися як віртуальні ядра) піднімає по потоку контейнеру функцій. Кожен контейнер функцій працює на

власному порті, щоб забезпечити можливість балансу навантаження зі сторони користувача.

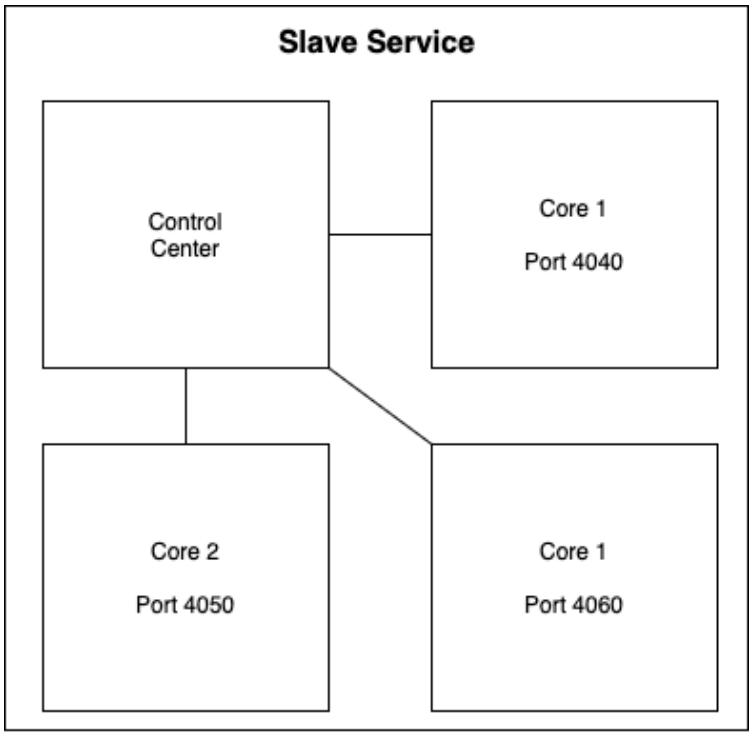


Рис.3.5. Внутрішня будова сервісу «Обслуговувач»

У випадку якщо система володіє лише одним ядром, то в одному потоці обробляється відвантаження / завантаження функцій до контейнеру функцій та обслуговування клієнтів. Повна класова діаграма розробленого прототипу сервісу «Обслуговувач» знаходиться в додатку Д.

Для зберігання інформації про підключення до різних сервісів рекомендовано використовувати **змінні оточення**. Для додавання змінних оточення користувач повинен вказати їх в спеціальному розділі користувацького інтерфейсу. Вони є доступними лише в контексті виконання функцій.

Контекст виконання функцій необхідний для того, щоб обмежити використання функціями сторонніх бібліотек (для роботи з файлової системою, базами даних тощо). Для створення контексту необхідно перейти у відповідний розділ користувацького інтерфейсу. Для того щоб повернути дані з контексту їх

необхідно завернути в об'єкт та присвоїти параметру **module.exports**. З контексту можна повертати як екземпляри третьосторонніх бібліотек так і певні константи.

Для того, щоб створити **функцію** необхідно перейти у відповідний розділ користувацького інтерфейсу та задати такі параметри як: ім'я функції, шлях, тип запиту, контекст виконання та ввести тіло самої функції. Рекомендовано використовувати контексти виконання для того щоб обмежити доступ до бібліотек для тих функцій де це не потрібно. Тим самим досягається принцип впровадження залежностей. Приклад коду та структура проекту сервісу «Обслуговувач» представлені в Додатку Б та Додатку В.

Для того, щоб надати можливість користувачу використовувати сторонні бібліотеки рекомендується притримуватись такого ходу дій:

1. Створити необхідні змінні оточення;
2. Створити контексти виконання функцій у яких будуть використовуватися змінні оточення, створюватися екземпляри доступу до об’єктів та повертатися з контексту;
3. Використати екземпляри з контексту в тілі функції.

3.3.1. Список використаних модулів

Таб.3.8. Використані модулі в сервісі «Обслуговувач»

Назва	**Версія**	**Опис**
axios	0.19.2	Бібліотека, що дозволяє робити запити по протоколу HTTP.
chalk	4.1.0	Бібліотека для форматування виводу.
cuid	2.1.8	Бібліотека для генерація випадкових стрічок.
fastify	2.7.0	Веб-сервер
gulp	3.4.1	Бібліотека для автоматизації рутинної роботи в проекті.

3.3.2. Процес реєстрації «обслуговувача» в системі

Для того, щоб включитися в роботу системи, кожен сервіс обслуговувача повинен пройти наступну процедуру:

1. Сервіс відправляє запит на реєстрацію у системі у якому він вказує інформацію про своє місце розташування (якщо є), IP- адресу;
2. Центр керування отримує запит на реєстрацію від кінцевого вузла, генерує йому унікальний ідентифікатор та повідомляє його про те, що вузол пройшов реєстрацію відправляючи йому JWT-код у якому міститься тип вузла та його ідентифікаційний номер;
3. Вузел записує цей ідентифікатор у пам'ять і з кожним наступним запитом відправляє його в ядро системи.

3.3.3. Процес обслуговування користувача в системі

Для того, щоб система могла обслуговувати користувача їй необхідно мати інформацію про даного користувача (список доступних функцій, тощо). Інформація береться з JWT токену, який користувач відправляє з кожним запитом до сервісу. Відносно даного токену узгоджується доступ до функцій для даного користувача та налаштовується роутінг для того щоб користувач міг користуватися даними функціями не вносячи зміни до власного API.

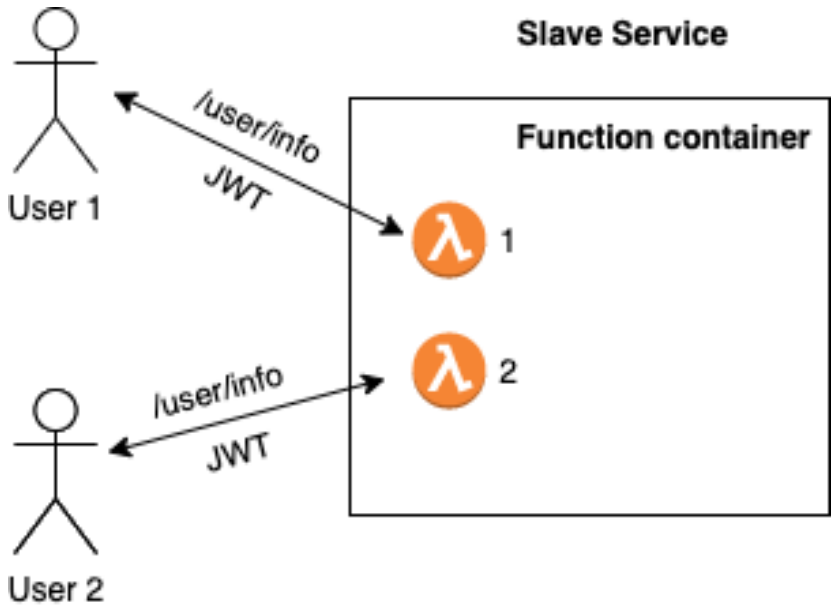


Рис.3.6. Роутинг різних користувачів за одною адресою

В цілях безпеки рекомендується змінювати токен протягом певного періоду. Для цього можна використати техніку **refresh token** [14].

3.3.4. Модель даних

Для того, щоб коректно працювати цьому сервісу необхідні наступні дані, що розміщені в моделі lambda. Дані розміщується в оперативній пам'яті вузлу, для доступу до них організовано спеціальні класи DTO.

Data Transfer Object (DTO) [15] - один з шаблонів проектування, використовується для передачі даних між підсистемами додатка. Оскільки вся логіка побудована на принципі сервісів, немає необхідності зберігати її в моделі даних.

Для розробки прототипу було використано принцип мінімальної достатності, що означає, що додаток оперує пише мінімально необхідною кількістю даних. Варто відзначити що в реальних проектах це структура даних буде іншою. Приклад оголошення моделі даних знаходиться у Додатку В. Тури моделі представлені в Таб.3.9 і Таб.3.10

Таб.3.9. Структура моделі «Функція»

Ім’я поля	**Тип поля**	**Опис поля**
Id	SERIAL	Первинний ключ
ctx	Object	Об'єкт, що містить контекст виконання функції
fn	STRING	Код функції, який будується для виконання

Таб.3.10. Структура моделі «Конфігурація»

Ім’я поля	**Тип поля**	**Опис поля**
master_url_connection_string	STRING	Підключення до ядра
geo	Object	Інформація про Гео локацію вузлу
type	STRING	Тип вузлу. Можливі значення: CLOUD, RPI.

3.4. Клієнтський модуль

Для того, щоб забезпечити зручність інтеграції нашої архітектури в вже розроблені додатки, ми вирішили розробити клієнтській модуль на мові програмування JavaScript. В обов'язки цього модуля входить:

1. Авторизація кінцевого користувача в системі;
2. Збір даних з головного сервера про наявність виконавців бізнес-логіки та їх навантаження;
3. Приховання комунікації з сервісами обслуговувачами за зручним фасадом;
4. Надання зручного API для розробників, щоб робити виклики до бізнес-логіки.

Нами було протестовано та встановлено, що розроблений модуль можливо інтегрувати в наступні технології:

- React
- Node.JS
- Electron
- node-gui
- PWA
- Angular

Також, під час розробки було знайдено, що для потреб кінцевих користувачів дуже зручною була би функція надання списку можливих дій в системі для конкретного користувача. Таким чином можна будувати логіку інтерфейсу відносно того, чи має певне користувач доступ до цієї дії. Це може бути реалізовано в вигляді заборони переходу на певні сторінки ресурсу, заборона натискати на певні кнопки, тощо. Таким чином кінцевий користувач матиме набагато більший рівень користувацького досвіду, скільки йому буде очевидно які дії для нього закриті, та це набагато приємніше ніж сповіщення про помилку. Всередині цього модуля перед виконанням кожної дії звіряється чи користувач може виконувати її.

Для того, щоб інтегруватися в різні системи ми розробили модель за допомогою принципів модулів Java Script ES6. Модулі можуть завантажувати один одного і

використовувати директиви export і import, щоб обмінюватися функціональністю, викликати функції одного модуля з іншого:

- **export** відзначає змінні і функції, які повинні бути доступні поза поточним модулем;
- **import** дозволяє імпортувати функціональність з інших модулів.

```
// файл module.js
export function sayHi(user) {
  return `Hello, ${user}!`;
}

// файл app.js
import {sayHi} from 'module';
```

Рис.3.7. Приклад використання ES6 модулів.

РОЗДІЛ 4
ОГЛЯД РЕЗУЛЬТАТІВ

4.1. Порівняння показників швидкодії

Для того, щоб зробити висновок про те, чи задовільняє розроблена система критерієм, які були задані при її проектуванні, ми провели ряд тестів. З отриманих даних можна буде зробити висновок про те наскільки прискорилася швидкість обробки даних, зменшився час відгуку на запит користувача, скільки часу було витрачено на передачу даних, тощо.

4.1.1. Перевірка швидкості відгуку на запит користувача

Перший тест, який ми провели, це тест на відгук на запит користувача. Для проведення цього тесту ми будемо використовувати вбудовану в Unix системи програму ping [16]. Тестувати будемо пакетами по 56 байт, в кожній серії тестувань ми будемо відправляти по 10 запитів на серію. Спочатку проведемо тест на «обслуговувачу», який знаходиться в віддаленому від користувача регіоні:

```
→  ~ ping 217.61.1.185
PING 217.61.1.185 (217.61.1.185): 56 data bytes
64 bytes from 217.61.1.185: icmp_seq=0 ttl=55 time=36.827 ms
64 bytes from 217.61.1.185: icmp_seq=1 ttl=55 time=38.270 ms
64 bytes from 217.61.1.185: icmp_seq=2 ttl=55 time=37.703 ms
64 bytes from 217.61.1.185: icmp_seq=3 ttl=55 time=38.107 ms
64 bytes from 217.61.1.185: icmp_seq=4 ttl=55 time=75.548 ms
64 bytes from 217.61.1.185: icmp_seq=5 ttl=55 time=37.512 ms
64 bytes from 217.61.1.185: icmp_seq=6 ttl=55 time=37.140 ms
64 bytes from 217.61.1.185: icmp_seq=7 ttl=55 time=37.220 ms
64 bytes from 217.61.1.185: icmp_seq=8 ttl=55 time=39.091 ms
64 bytes from 217.61.1.185: icmp_seq=9 ttl=55 time=40.050 ms
^C
--- 217.61.1.185 ping statistics ---
10 packets transmitted, 10 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 36.827/41.747/75.548/11.305 ms
```

Рис.4.1 Перевірка пінгу в віддаленому від користувача регіоні

Як ми можемо побачити з результатів приведених вище, часом відбуваються просадки до 75 мілісекунд, що може негативно вплинути на користувацький досвід в додатках які вимагають швидкої реакції з боку серверу. Варто зазначити, що комуні-

кція відбувається в межах глобальної мережі інтернет, що означає, що ця затримка може варіюватися.

Тепер проведемо тест на «обслуговувачу», який знаходиться в тому ж регіоні, що і користувач:

```
➜  ~ ping 172.67.15.177
PING 172.67.15.177 (172.67.15.177): 56 data bytes
64 bytes from 172.67.15.177: icmp_seq=0 ttl=58 time=17.314 ms
64 bytes from 172.67.15.177: icmp_seq=1 ttl=58 time=19.315 ms
64 bytes from 172.67.15.177: icmp_seq=2 ttl=58 time=18.855 ms
64 bytes from 172.67.15.177: icmp_seq=3 ttl=58 time=19.291 ms
64 bytes from 172.67.15.177: icmp_seq=4 ttl=58 time=20.398 ms
64 bytes from 172.67.15.177: icmp_seq=5 ttl=58 time=15.388 ms
64 bytes from 172.67.15.177: icmp_seq=6 ttl=58 time=19.816 ms
64 bytes from 172.67.15.177: icmp_seq=7 ttl=58 time=22.589 ms
64 bytes from 172.67.15.177: icmp_seq=8 ttl=58 time=20.420 ms
64 bytes from 172.67.15.177: icmp_seq=9 ttl=58 time=15.493 ms
^C
--- 172.67.15.177 ping statistics ---
10 packets transmitted, 10 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 15.388/18.888/22.589/2.140 ms
```

Рис.4.2. Перевірка пінгу в наближеному до користувача регіоні

З результатів приведених вище, ми можемо зробити висновок що відповідь на запити користувача відбувається швидше приблизно на 50%, та час очікування відповіді не перевищує 30 мілісекунд, що дозволяє використовувати цей обслуговувати в разі крайньої необхідності в додатках реального часу.

Тепер проведемо цей же самий тест на «обслуговувачу», що знаходиться в межах локальної мережі:

```
➜  ~ ping raspberrypi.local
PING raspberrypi.local (192.168.0.112): 56 data bytes
64 bytes from 192.168.0.112: icmp_seq=0 ttl=64 time=3.485 ms
64 bytes from 192.168.0.112: icmp_seq=1 ttl=64 time=3.516 ms
64 bytes from 192.168.0.112: icmp_seq=2 ttl=64 time=3.858 ms
64 bytes from 192.168.0.112: icmp_seq=3 ttl=64 time=3.982 ms
64 bytes from 192.168.0.112: icmp_seq=4 ttl=64 time=3.442 ms
64 bytes from 192.168.0.112: icmp_seq=5 ttl=64 time=3.912 ms
64 bytes from 192.168.0.112: icmp_seq=6 ttl=64 time=3.596 ms
64 bytes from 192.168.0.112: icmp_seq=7 ttl=64 time=3.405 ms
64 bytes from 192.168.0.112: icmp_seq=8 ttl=64 time=3.607 ms
64 bytes from 192.168.0.112: icmp_seq=9 ttl=64 time=3.989 ms
^C
--- raspberrypi.local ping statistics ---
10 packets transmitted, 10 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 3.405/3.697/3.989/1.422 ms
```

Рис.4.3. Перевірка пінгу в локальній мережі

Як ми можемо переконатися, час відповіді в локальні мережі мінімальний, та становить близько 3,5 мілісекунд, що є дуже хорошим результатом для додатків реального часу.

4.1.2. Перевірка пропускної здатності

Наступний тест який ми проведемо - тест на максимальну пропускну здатність. Для цього будемо використовувати програму iperf [17]. Спочатку проведемо тест на «обслуговувачу», що знаходиться в іншому регіоні від користувача:

```
------------------------------------------------------------
Client connecting to 217.61.1.185, TCP port 5001
TCP window size: 2.50 MByte (default)
------------------------------------------------------------
[  3] local 127.0.0.1 port 42200 connected with 217.61.1.185 port 5001
[ ID]  Interval       Transfer        Bandwidth
[  3]  0.0- 0.5 sec    4.6 MBytes     48.4 Mbits/sec
[  3]  0.5- 1.0 sec    5.0 MBytes     53.6 Mbits/sec
[  3]  1.0- 1.5 sec    4.2 MBytes     44.4 Mbits/sec
[  3]  1.5- 2.0 sec    5.0 MBytes     52.8 Mbits/sec
------------------------------------------------------------
[  3]  0.0- 2.0 sec   18.8 MBytes     49.8 Mbits/sec
```

Рис.4.4. Перевірка пропускної здатності в віддаленому регіоні

З тесту отриманого вище ми можемо зробити висновок, що середня швидкість передачі даних складає близько 50 мегабітів за секунду. Варто зазначити, що на пропускному здібність впливають також і непрямі фактори такі як: тип під'єднання клієнта до мережі (WiFi, Enthernet, GSM); якість обладнання, що надає доступ до інтернету; навантаження на провайдер даний момент часу; пропускна здібність мережевого інтерфейса клієнта, тощо. Тому ці результати варто вважати моментальними і опиратися на них ми можемо лише в сприятливих умовах.

Проведемо також тест на «обслуговувачу», що знаходиться в тому ж регіоні:

```
------------------------------------------------------------
Client connecting to 172.67.15.177, TCP port 5001
TCP window size: 2.50 MByte (default)
------------------------------------------------------------
[  3] local 127.0.0.1 port 42200 connected with 172.67.15.177 port 5001
[ ID]  Interval       Transfer        Bandwidth
[  3]  0.0- 0.5 sec    4.4 MBytes     46.0 Mbits/sec
[  3]  0.5- 1.0 sec    4.8 MBytes     51.0 Mbits/sec
[  3]  1.0- 1.5 sec    4.0 MBytes     42.2 Mbits/sec
[  3]  1.5- 2.0 sec    4.8 MBytes     50.3 Mbits/sec
------------------------------------------------------------
[  3]  0.0- 2.0 sec   18.8 MBytes     47.3 Mbits/sec
```

Рис.4.5. Перевірка пропускної здатності в ближньому регіоні

А з цього тесту ми можемо зрозуміти, що пропускна здатність приблизно однакова як і з «обслуговувачем», який знаходиться в іншому регіоні. Географічна наближеність до користувача дає нам можливість сподіватися на те, що пропускна здатність не повинна просісти так як може в випадку з «обслуговувачем», що знаходиться в іншому регіоні.

Наступний тест проведемо з обслуговуючим який знаходиться в локальні мережі:

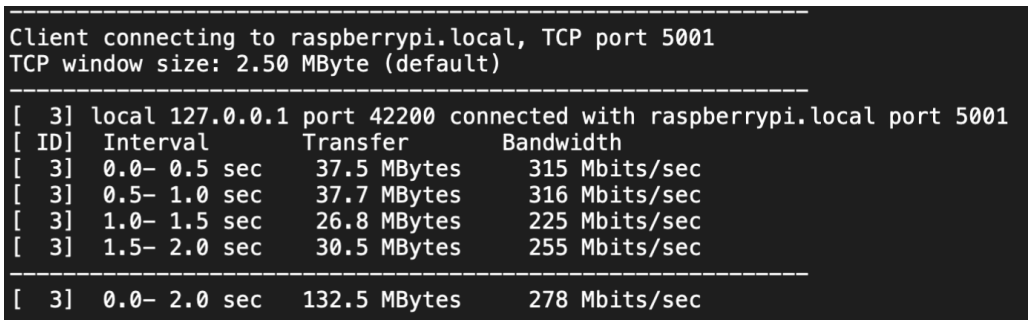


Рис.4.6. Перевірка пропускної здатності в локальній мережі

З результатів даного тексту ми можемо зрозуміти що пропускна здібність такого варіанту приблизно в 5 раз більше, ніж з «обслуговувачами», що знаходяться в глобальні мережі. Це дає нам можливість зробити твердження, якщо нам потрібно підтримувати додатки реального часу, то може бути цікаво робити певні обрахування на «обслуговувачу», що знаходиться в безпосередній близькості до користувача.

4.1.3. Перевірка часу виконання завдання

Останній тест, який ми проведемо, це тест на опрацювання CPU-intensive завдань. 8 завдання полягає у тому, що необхідно обробити набір даних розміром 100 мегабайт шифром Цезаря. Ми заміряли час необхідний для виконання такого завдання для кожного із видів обчислювальних пристроїв. Script даного тексту знаходиться в Додатку В. Зрозуміло, що в реальному житті характеристики обчислювальних засобів можуть варіюватися, для тесту ми зафіксуємо показники продуктивності для Raspberry Pi, та хмарних надавачів обчислювальної потужності. Детальні характеристики зображені у Таб.4.1.

Таб.4.1. Порівняння характеристик обчислювальних засобів

Характеристика	**Raspberry Pi 4**	**Cloud Service (EC2)**
Розмір ОЗП	4 ГБ	2 ГБ
Кількість ядер	4	2
Кількість циклів процесора за секунду	1.5 Ghz	2 Ghz

Будемо вважати, що обчислювальні засоби «обслуговувачів», що знаходяться в хмарному середовищі мають однакові характеристики.

Ми провели по 3 теста на кожному із типів обчислювальних засобів, та отримали такі середні результати:

- Для виконання завдання Raspberry Pi знадобилося 35 секунд;
- Для виконання завдання хмарному на давачів послуг знадобилося 23 секунди.

4.1.2. Аналіз отриманих результатів

Для зручності зведемо усі результати в єдину табличку Таб.4.2:

Таб.4.2. Результати проведених тестів

Вид тесту	Raspberry Pi в локальній мережі	EC2 в наближеному регіоні	EC2 в віддаленому регіоні
Відгук на запит	3.7 мс	18 мс	41 мс
Пропускна здібність	278 Мбіт/с	47 Мбіт/с	49 Мбіт/с
Виконання завдання	35 с	23 с	23 с

Для того, щоб розрахувати швидкість отримання результату одного завдання використаємо формулу (4.1):

$$t_{exec} = t_{ping} + t_{packet} + t_{exec(packet)} \qquad (4.1),$$

де t_{ping} - час, що необхідний для початку передачі даних; t_{packet} - час доставки одного пакета даних; $t_{exec(packet)}$ - час на виконання одного пакет. Для того, щоб спростити

обчислення ми припускаємо, що показники є константами, та не враховуємо можливу затримку яку створює обладнання (затримка мережевої карти, тощо).

Тепер розрахуємо час на виконання одного завдання для кожного виду «обслуговувачів» за допомогою даних приведених вище:

- Час що необхідний на виконання одного завдання для Raspberry Pi в локальні мережі становить 37.9c.
- Час що необхідно для виконання одного завдання для EC2 в наближеному регіоні становить 40c.

За умови того, що на Raspberry Pi буде виконуватися такий же самий код, що і в хмарному середовищі перевага незначна на CPU-intensive завданнях. Проте якщо замінити RaspberryPi на рішення, що буде настільки ж продуктивним як обслуговувати чи в хмарних середовищах, то таке значення буде 25c, що на 37% швидше ніж виконувати це завдання хмарному середовищі. Тому, якщо нам необхідно використовувати Raspberry Pi, то є зміст в тому, щоб спростити модель розрахунків для такого типу задач конкретно для мікро комп'ютера, тоді можна буде збільшити швидкість виконання завдання і зменшити час очікування відповіді.

Проте, якщо нам необхідно опрацьовувати IO-intensive завдання (це коли на відміну від великого об'єму даних - велика кількість запитів на невеликих завданнях) - Raspberry Pi покаже себе набагато краще.

Враховуючи те, що з'єднання вокальний мережі набагато надійніший ніж з'єднання в глобальній мережі інтернет, стає очевидно, що використовувати обчислювальні вузли які знаходяться в безпосередній близькості до користувачів в переважній більшості випадків вигідніше. В той же час, коли необхідно виконати запит користувача, що знаходиться у русі - використання обчислювальної потужності в найближчному регіоні до цього користувача дозволить значно підвищити рівень користувацького досвіду.

Також з вище наведених даних ми зробили висновок, що важкі завдання для процесора краще відправляти на хмарне середовище, ключову роль в часі очікування відіграє саме час виконання завдання. Для того, щоб не перенавантажувати хмарні середовища можна використовувати локальні вузли для первинної обробки даних.

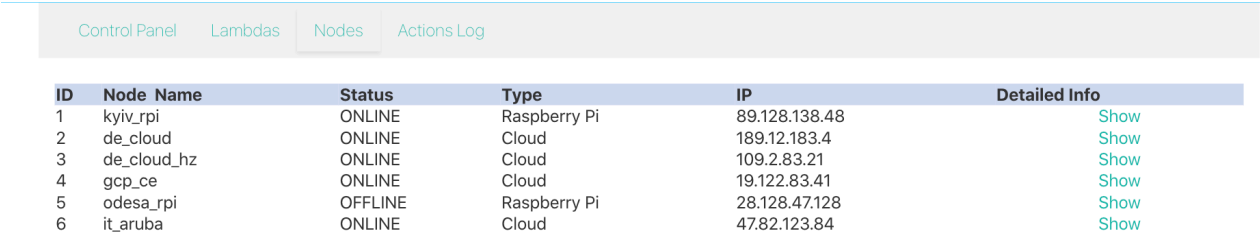
4.2. Інформаційна система для організації бізнес-логіки

Ми розробили інформаційну систему для організації бізнес-логіки, щоб надати керівникам додатків організовувати доступ до бізнес логіки, контролювати ресурси та навантаження, керувати функціями додатку. Це зроблено у вигляді веб додатку, до якого можна отримати доступ як через публічну інтернет мережу, так і при необхідності через приватну віртуальну мережу. Список використаних модулів розміщено в Таб.4.3.

Таб.4.3. Використані модулі в системі

Назва	**Версія**	**Опис**
React	16.3.1	JavaScript бібліотека, що дозволяє створювати складні користувацькі інтерфейси на основі реактивного зв’язування даних.
react-bootstrap	4.5.0	Набір готових компонентів для створення користувацького інтерфейсу.
react-dom	16.3.1	Бібліотека для роботи з DOM.
react-leaflet	2.7.0	Готові компоненти для створення інтерактивних географічних карт.
react-scripts	3.4.1	Бібліотека для автоматизації рутинної роботи в проекті та налаштування конфігурації.

4.2.1. Список вузлів



ID	Node Name	Status	Type	IP	Detailed Info
1	kyiv_rpi	ONLINE	Raspberry Pi	89.128.138.48	Show
2	de_cloud	ONLINE	Cloud	189.12.183.4	Show
3	de_cloud_hz	ONLINE	Cloud	109.2.83.21	Show
4	gcp_ce	ONLINE	Cloud	19.122.83.41	Show
5	odesa_rpi	OFFLINE	Raspberry Pi	28.128.47.128	Show
6	it_aruba	ONLINE	Cloud	47.82.123.84	Show

Рис.4.7. Таблиця списку вузлів обслуговувача

На цій сторінці (зображена на Рис.4.7.) можна переглянути загальні відомості з приводу підключених вузлів «обслуговувачів». Відомості доступні за такими полями даних як: ім’я, статус, тип, IP-адреса. Для того щоб провести сортування за значенням необхідно натиснути на назву колонки. При кліку на посилання в останній колонці відбувається перехід на сторінку з детальною інформацію щодо даного «обслуговувача».

Варто зазначити, що корисною є функція сортування по статусу, так можна виявити, що в системі не працює певна кількість сервісів, та при необхідності їх перезавантажити.

4.2.2. Детальна інформація про вузел

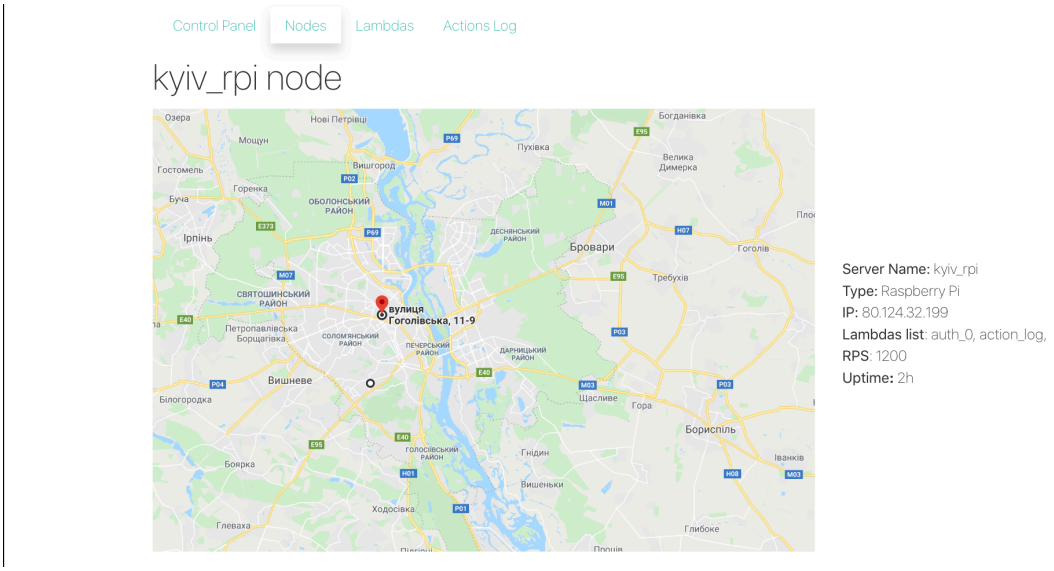


Рис.4.8. Детальна інформація про вузел

На даній сторінці можна переглянути:

1. Розташування обслуговуватися на карті;
2. Ім'я серверу;
3. Тип серверу;
4. IP адреса;
5. Набір функцій, які на нього завантаженні;
6. Кількість запитів за секунду;
7. Час активності.

Особливо корисною ця сторінка (зображена на Рис.4.8.) буде для адміністратора додатку, оскільки на ній знаходиться актуальна інформація про стан системи. Адміністратор може на основі приведених вище даних зробити висновок про потенційне розширення кількості обчислювальних засобів в певному регіоні, про те, що певні пристрої необхідно продіагностувати.

4.2.3. Керування змінними оточення

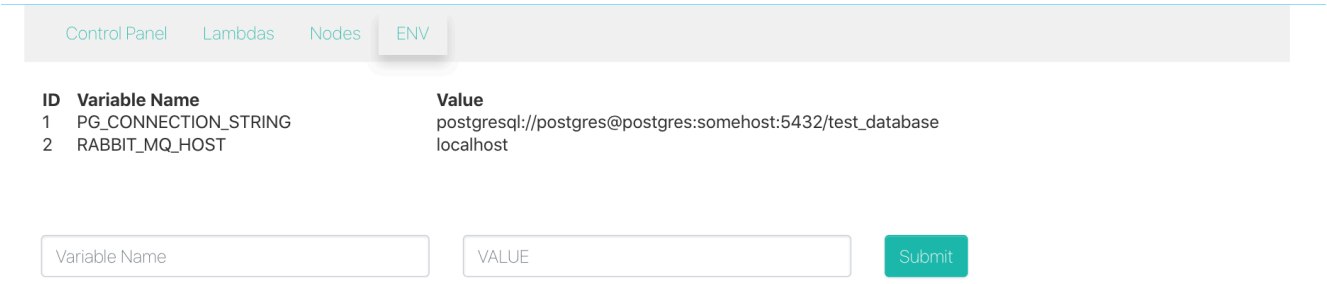


Рис.4.9. Інтерфейс керування змінними оточення

На цій сторінці (зображена на Рис.4.9.) можна додати, відредагувати та переглянути змінні оточення. Для того, щоб додати нову змінну оточення слід списати назву в поле для назви та значення даної змінної після чого натиснути на кнопку «зберегти». Редагування змінної відбувається після подвійного натискання

на її значення. Для того, щоб відсортувати дані за значенням необхідно натиснути на назву колонки.

Змінні потрапляють до контексту за принципом як є в вигляді стрічки. Тому якщо для конфігурації певної бібліотеки необхідно використовувати числове значення його потрібно перед тим заздалегідь привести до типу який необхідний.

Для кращої оптимізації швидкодії також слід використовувати змінну **NODE_ENV** зі значенням «**production**», оскільки це загальноприйнятий підхід на який орієнтуються розробники різних бібліотек, драйверів тощо.

4.4. Створення контексту виконання

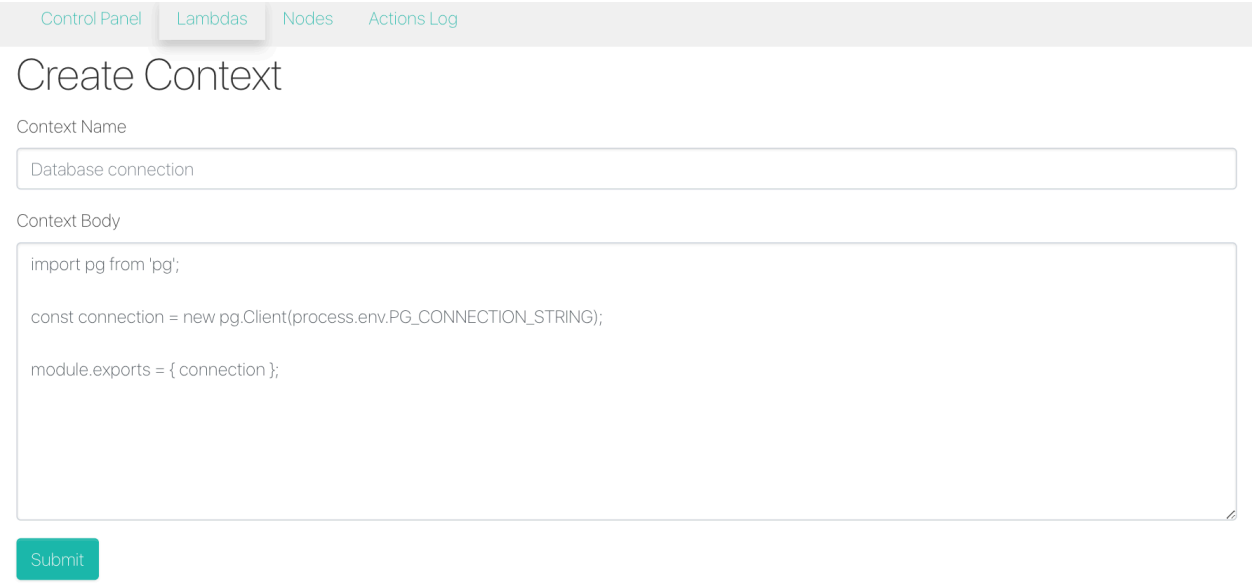


Рис.4.10. Інтерфейс створення контексту

На даній сторінці (зображена на Рис.4.10.) можна створити контекст виконання функції. Для цього необхідно задати йому ім'я та саме тіло контекста. В тілі контекста можна проводити ініціалізацію модулів, виводити конфігурацію з вхідних даних, розраховувати значення констант на які будуть опиратися кінцеві функції. Для виведення даних в функцію необхідно перевизначити атрибут **module.exports** новим об’єктом. В функції можна буде отримати доступ до контексту за допомогою вхідного параметру **ctx**.

Для оптимізації швидкості виконання бізнес-логіки визначення контексту не повинно бути важким, оскільки даний контекст виконуються перед кожною функцію яку викликає користувач.

Для оптимізації безпеки не слід повертати з контексту виконання змінні оточення та використовувати їх в тілі функцій [1]. Якщо виникає така необхідність - необхідно створити об’єкт, який буде виконувати певну бізнес-логіку та повернути його з контексту виконання.

4.5. Список функцій

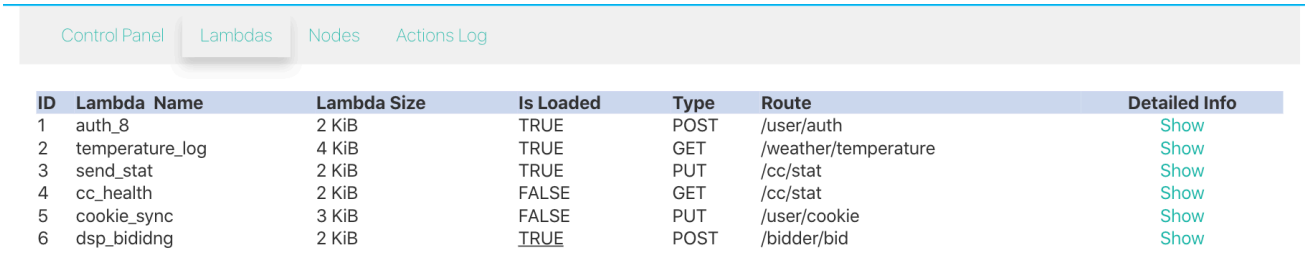


Рис.4.11. Таблиця загальних відомостей про функції

На даній сторінці (зображена на Рис.4.11.) можна переглянути відомості щодо функцій зареєстрованих системи. Табличка на цій сторінці поводить себе аналогічно сторінці перегляду обслуговуючів. При кліку на посилання в останній колонці відбувається перехід на сторінку з детальною інформацію щодо даної функції.

4.6. Створення контексту виконання

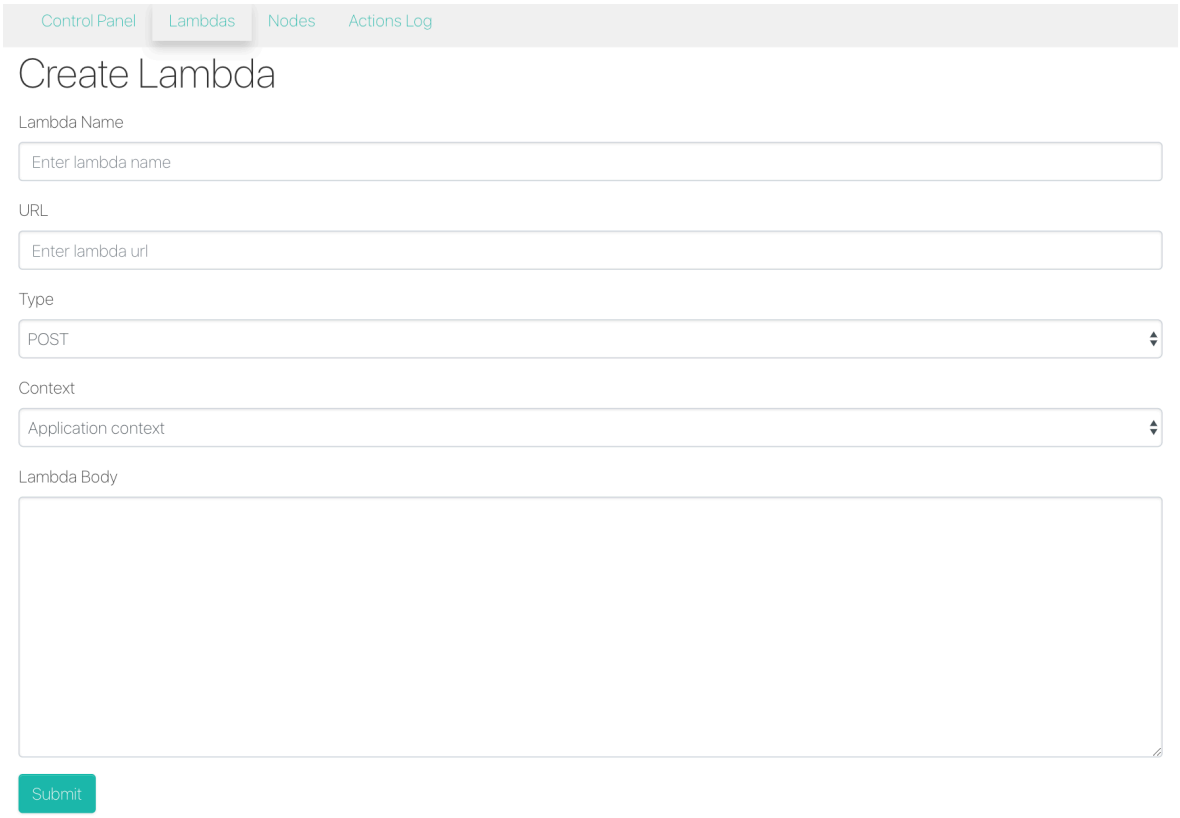


Рис.4.12. Інтерфейс створення функції

На даній сторінці (зображена на Рис.4.12.) користувачу пропонується створити функцію. Для цього йому необхідно ввести ім'я функції, її шлях, тип запиту, обрати контекст виконання функції. В відповідне поле необхідно ввести тіло функції починаючи з перевизначення параметру **module.exports**.

На вхід функція приймає такі параметри:

1. headers - HTTP хедери користувача;
2. params - URL параметри, позначені двокрапкою (/user/:id);
3. attributes - URL - аттрибути;
4. body - Тіло запиту для POST, PUT, DELETE, PATCH запитів;
5. user - дані про корисувача, які взяті з його JWT токену;
6. ctx - контекст виконання функції;

ВИСНОВКИ

1. Під час роботи над кваліфікаційною роботою студент дослідив предметно область, список готових рішень, що використовуються в даній предметні області, встановив проблему вирішення якої описано в даній роботі;
2. Було встановлено, що мікросервісна архітектура не задовільняє вимогам додатків реального часу, оскільки мікро сервіси і є великими одиницями, практично немає можливості утримувати бізнес-логіку у вигляді мікросервісів на одному вузлі, особливо для обчислювальних пристроїв з низькою продуктивністю;
3. Було досліджено концепцію Fog and Edge обчислень, та взято принцип капілярної доставки сервісів. В подальшому розроблене рішення доставляє по цьому принципу бізнес-логіку на різні вузли, але робить це не в вигляді контейнерів, а в вигляді вихідного коду, чим відрізняється від даної архітектури;
4. Було розглянуто концепцію безсерверних обчислень, та встановлено, що це найоптимальніший спосіб зберігання бізнес-логіки в випадку, коли нам необхідно утилізувати наявні обчислювальні ресурси в оптимальному обсязі;
5. Було проведено дослідження, мета якого — встановити найоптимальніший варіант обладнання, що можна розмістити у локальні мережі поряд з користувачем, та встановлено, що RaspberryPi 4 є найоптимальнішим рішенням через розвинуту спільноту цього мікро комп'ютера, його ціну, та наявність експертизи у студента;
6. Студент провів дослідження, мета якого — становити найоптимальніший варіант хмарного на давача послуг. Було розглянуто декілька пропозицій на ринку, та встановлено що Amazon EC2 є найоптимальнішим рішенням у зв'язку з наявністю експертизи, наявністю безкоштовного періоду користування, найрозвинутішою підтримкою з боку спільноти;
7. Було обрано програмні технології, що будуть використовуватися при створенні прототипу. Було досліджено різні альтернативні технологічні стеки, та обрано Node.JS з точки зору наявності експертизи, та вимог до обладнання. Також було обрано фреймворк для будування користувацького інтерфейсу, та операційні системи для обчислювальних засобів;
8. Було розроблено концепцію архітектури, що складається з 3 основних складових: сервісу «ядро», сервісу «обслуговувач», клієнтського модуля;

9. Було описано принципи проектування складових прототипу, що реалізує модель архітектури, описано модель даних, та надано список використаних моделей даних для кожної складової прототипу;
10. Було описано принципи взаємодії між складовими розробленого прототипу;
11. Розроблено та описано концепцію, якій повинен відповідати клієнтській модуль, встановлено ключові моменти для реалізації та розроблено даний модуль. Після розробки модуля було проведено тестування та досліджено у який набір технологій можливо інтегрувати даний модуль;
12. Було проведено комплексний тест розробленого прототипу такими параметрами: швидкість відповіді на запит, ширина пропускної здатності до каналу комунікації (між кінцевим користувачем, та сервісом «обслуговувач»), час виконання завдання для різних типів обчислювальних засобів.
13. Встановлено, що найкращий час відгуку знаходиться в пристроях, що розміщені в межах локальної мережі, гірший результат в обчислювальних засобах, що знаходяться в регіоні користувача, але доступ до нього в рамках глобальної мережі інтернет. Найгірший результат показав обчислювальний засіб, що знаходиться в віддаленому від користувача регіоні;
14. Встановлено, що найкращий показник пропускної здатності знаходиться в рамках локальної мережі. Ширина каналу до хмарних «обслуговувачів» не сильно відрізнялася в залежності від регіону. Проте ширина пропускної здатності може варіюватися від великого набору різних факторів.
15. Встановлено, що найкращий час виконання завдання знаходиться в хмарних «обслуговувачах». Якщо є необхідність використовувати низькопродуктивний вузол, постає необхідність в спрощені моделі обчислювання для нього;
16. Було розраховано загальний час виконання одного комплексного завдання в рамках локальної мережі з мікро комп'ютером, та в рамках глобальної мережі з хмарними давачами послуг. Встановлено, що при використанні CPU-intensive завдань краще відпрацьовують хмарні обчислювальні засоби, проте, якщо є необхідність в IO-intensive завданнях, то краще їх виконувати в рамках локальної мережі;

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Бізнес-логіка [Електронний ресурс]. Режим доступу: (https://uk.wikipedia.org/wiki/Бізнес-логіка);
2. M.Fowler. Microservices [Електронний ресурс]. Режим доступу: (https://martinfowler.com/articles/microservices.html);
3. Fog and Edge Computing: Principles and Paradigms. ISBN-10: 1119524989. December 2018;
4. Cloud Computing Solutions Architect: A Hands-On Approach: A Competency-based Textbook for Universities and a Guide for AWS Cloud Certification and Beyond **ISBN-10:** 0996025596. July 2019;
5. Haines, Steven. Serverless computing with AWS Lambda, Part 1 (англ.), JavaWorld. [Електронний ресурс]. Режим доступу: (https://www.infoworld.com/article/3210726/serverless-computing-with-aws-lambda.html);
6. Raspberry Pi 4 model b [Електронний ресурс]. Режим доступу: (https://www.raspberrypi.org/products/raspberry-pi-4-model-b/);
7. Practical Amazon EC2, SQS, Kinesis, and S3: A Hands-On Approach to AWS 1st ed. Edition, Kindle Edition ISBN-10: 1484228405, January 2017;
8. Node.JS [Електронний ресурс]. Режим доступу: (https://nodejs.org/uk/about/);
9. Асинхронне програмування [Електронний ресурс]. Режим доступу: (https://uk.wikipedia.org/wiki/Асинхронне_програмування);
10. React [Електронний ресурс]. Режим доступу: (https://uk.wikipedia.org/wiki/React);
11. Amazon Linux AMI [Електронний ресурс]. Режим доступу: (https://aws.amazon.com/ru/amazon-linux-ami/);
12. Raspbian [Електронний ресурс]. Режим доступу https://en.wikipedia.org/wiki/Raspberry_Pi_OS);
13. What Is a Pure Function in JavaScript [Електронний ресурс]. Режим доступу (https://www.freecodecamp.org/news/what-is-a-pure-function-in-javascript-acb887375dfe/);

14. Understanding Refresh Tokens [Електронний ресурс]. Режим доступу (https://auth0.com/learn/refresh-tokens/);
15. Data Transfer Object [Електронний ресурс]. Режим доступу https://uk.wikipedia.org/wiki/DTO);
16. Програма Ping [Електронний ресурс]. Режим доступу (https://uk.wikipedia.org/wiki/Ping);
17. Програма Iperf [Електронний ресурс]. Режим доступу (https://en.wikipedia.org/wiki/Iperf);
18. Бражиненко М.Г., Козачок П.А., Шевченко В.Л. . Проблеми публікації-підписки у бездротових мережах для вузлів Інтернету всього / MSTIoE 2018-3 : East European Conference on Mathematical Foundations and Software Technology of Internet of Everything (21-22.12.2017, Kyiv - Київ). Зб. Матер. С.32;
19. Brazhenenko M., Kozachok P., Shevchenko V., Tkachenko M. M2M Communication Protocol for Low Bandwith MEMS Sensor Networks. 2018 14-th International Conference Perspective Technologies and Methods in MEMS Design (MEMSTECH). Proceeding. - Polyana, April 18-22 pp.35-38. Scopus;
20. Maksym Brazhenenko, Pavlo Kozachok, Volodymyr Petrivskyi, Oleksiy Bychkov, Victor Shevchenko Cloud Based Architecture Design of System of Systems CADSM 2019, 15th International Conference on the Experience of Designing and Application of CAD Systems (CADSM), February 26 – March 2, 2019, Polyana-Svalyava (Zakarpattya), UKRAINE, Lviv Polytechnic National University, UKRAINE, Lodz University of Technology, POLAND, IEEE Ukraine Section, IEEE Ukraine Section (West), MTT/ED/AP/EP/SSC Societies Joint Chapter, Part Number: CFP19508-USB, ISBN: 978-1-7281-0053-1 pp.19-23. очікується Scopus, IEEE.

Додаток А. Діаграма запропонованої архітектури

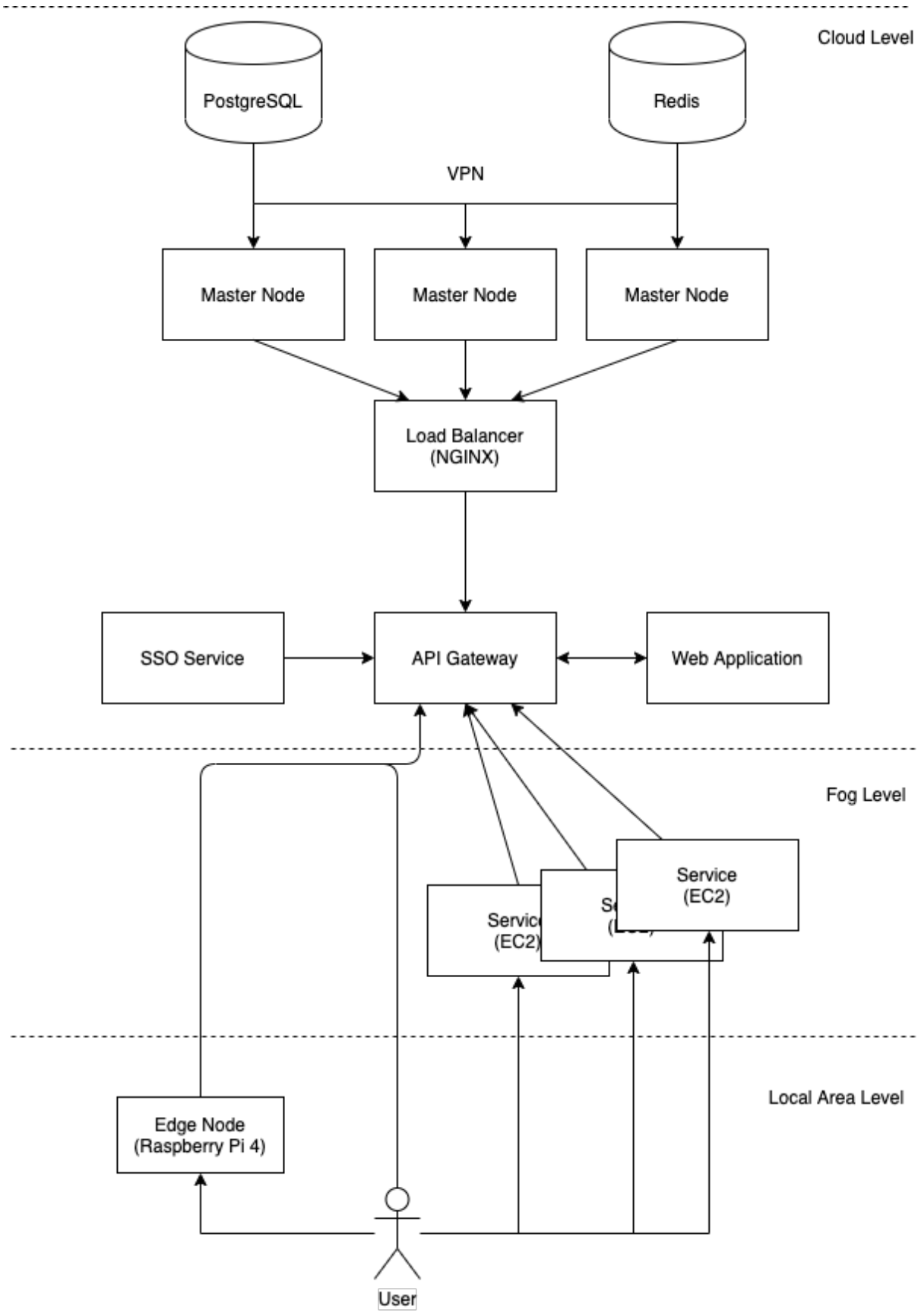


Рис.А.1. Діаграма запропонованої архітектури

Додаток Б. Структура файлів проекту прототипу

Нижче приведена структура файлів розробленої інформаційної системи:

```
├── fe
│   ├── README.md
│   ├── package.json
│   ├── public
│   │   ├── favicon.ico
│   │   ├── index.html
│   │   ├── logo192.png
│   │   ├── logo512.png
│   │   ├── manifest.json
│   │   └── robots.txt
│   ├── src
│   │   ├── App.css
│   │   ├── App.js
│   │   ├── components
│   │   │   ├── pages
│   │   │   │   ├── Contexts
│   │   │   │   ├── CreateContext
│   │   │   │   ├── CreateLambda
│   │   │   │   ├── ENVs
│   │   │   │   ├── EditContext
│   │   │   │   ├── EditLambda
│   │   │   │   ├── Lambdas
│   │   │   │   ├── Node
│   │   │   │   │   ├── page.js
│   │   │   │   │   └── styles.module.css
│   │   │   │   └── Nodes
│   │   │   └── shared
│   │   │       ├── ContextForm.js
│   │   │       └── LambdaForm.js
│   │   ├── index.js
│   │   ├── lib
│   │   ├── models
│   │   │   ├── Contexts.js
│   │   │   ├── ENVs.js
│   │   │   ├── Lambdas.js
│   │   │   └── Nodes.js
│   │   └── serviceWorker.js
│   └── yarn.lock
```

Нижче приведена структура файлів розробленого сервісу «Ядро»:

```
├── master_node
│   ├── app.mjs
│   ├── configs
│   │   └── index.mjs
│   ├── models
│   │   ├── context.model.mjs
│   │   ├── env.model.mjs
│   │   ├── lambda.model.mjs
│   │   ├── node.model.mjs
│   │   └── user.model.mjs
│   ├── repositories
│   │   ├── context.repository.mjs
│   │   ├── env.repository.mjs
│   │   ├── lambda.repository.mjs
│   │   └── user.repository.mjs
│   ├── services
│   │   ├── lambda.factory.mjs
│   │   ├── nodes.service.mjs
│   │   ├── nodes_api.service.mjs
│   │   └── user_api.service.mjs
│   └── src
│       └── master_node.application.mjs
└── package.json
```

Нижче приведена структура файлів розробленого сервісу «Обслуговувач»:

```
├── service_node
│   ├── app.mjs
│   ├── configs
│   │   └── index.mjs
│   ├── models
│   │   └── lambda.mjs
│   ├── services
│   │   ├── connection.service.mjs
│   │   ├── lambdas_container.service.mjs
│   │   ├── runtime.service.js
│   │   └── server.service.mjs
│   └── src
│       └── service_node.application.mjs
├── services
│   └── model_loader.service.mjs
└── yarn.lock
```

Додаток В. Приклади коду розробленого прототипу

Приклад початкової точки виконання сервісу:

```
const startProcess = async () => {
  const app = new MasterNodeApplication(configs)
  await app.start()

  process.on('SIGINT', app.stop.bind(app));
  process.on('SIGTERM', app.stop.bind(app));
  process.on('uncaughtException', app.stop.bind(app));
}

startProcess()
  .then(() => {
    console.log(`MasterNodeApplication has been started`)
  })
  .catch(console.error)
```

Приклад головного класу сервісу:

```
export default class MasterNodeApplication extends EventEmitter {
  constructor(config) {
    super()
    this._envRepository = new ENVRepository()
    this._contextRepository = new ContextRepository()
    this._lambdaRepository = new LambdaRepository(this._contextRepository)
    this._lambdaFactory = new LambdaFactory(this._envRepository, this._context-
Repository)
    // реєстрація, оновлення, лямбди
    this._nodesService = new NodesService(config.nodesService)
    // прийом заявок на реєстрацію, переспрямування, доставка лямбд
    this._nodesAPIService = new NodesAPIService(config.nodesRPCService, this._nodes-
Service, this._lambdaRepository, this._lambdaFactory)
    // обслуговування користувача
    this._userAPIService = new UserAPIService(
      config.userAPIService,
      this._envRepository,
      this._contextRepository,
      this._lambdaRepository,
      this._nodesService
```

```
    )
  }

  async start() {
    await Promise.all([
      this._nodesAPIService.start(),
      this._nodesService.start(),
      this._userAPIService.start()
    ])
  }

  async stop() {
    await Promise.all([
      this._nodesAPIService.stop(),
      this._nodesService.stop(),
      this._userAPIService.stop()
    ])

    console.log(`${chalk.yellow(this.constructor.name)} has been stopped`)
  }
}
```

Приклад оформлення класу моделі:

```
export default class ENVModel {
  constructor({ userID, id, name, value }) {
    this.userID = userID
    this.id = id
    this.name = name
    this.value = value
  }
}
```

Приклад оформлення класу репозиторії:

```
export default class ContextRepository {
  constructor() {
    this._modelLoaderService = new ModelLoaderService(
      `${process.cwd()}/sample_data/contexts.csv`,
      ['userID', 'id', 'name', 'value']
    )
```

```
    this._contexts = this._modelLoaderService.load().map(context => new Context-
Model(context))
  }

  _storeContexts() {
    this._modelLoaderService.save(this._contexts)
  }

  addContext(userID, { name, value }) {
    const context = new ContextModel({ id: uuid(), userID, name, value })
    const id = this._contexts.push(context) - 1
    this._storeContexts()
    return this._contexts[id]
  }

  getContext(userID, id) {
    const context = this._contexts.find(({ id: ctxID }) => ctxID === id)
    if (!context) throw new Error(`${this.constructor.name}: no such context!`)
    return new ContextModel(context)
  }

  removeContext(userID, id) {
    this._contexts = this._contexts.filter(({ id: ctxID, userID: ctxUserID }) =>
ctxID !== id && userID === ctxUserID)
    this._storeContexts()
    return 'ok'
  }

  editContext(userID, id, { name, value }) {
    const context = this.getContext(userID, id)
    context.name = name
    context.value = value
    this.removeContext(userID, id)
    this._contexts.push(context)
    this._storeContexts()
    return context
  }

  getByUser(userID) {
    return this._contexts.filter(({ userID: id }) => id === userID)
  }
}
```

Приклад оформлення класу-будівельника функцій:

```
export default class LambdaFactory {
  /**
   * @param {ENVRepository} envRepository
   * @param {ContextRepository} contextRepository
   */
  constructor(envRepository, contextRepository) {
    this._envRepository = envRepository
    this._contextRepository = contextRepository

    this.constructLambda = this.constructLambda.bind(this)
  }

  /**
   * @param {LambdaModel} lambda
   */
  constructLambda(lambda) {
    const { userID, contextID } = lambda
    const envs = this._envRepository.getByUser(userID)
    const context = this._contextRepository.getContext(userID, contextID)
    envs.forEach(({ name, value }) => {
      context.value = context.value.replace(name, `"${value}"`)
    })

    lambda.context = context.value;

    return lambda
  }
}
```

Сценарій для отримання часу виконання CPU-intensive завдань

```
var caesarShift = function (str, amount) {
  if (amount < 0) {
    return caesarShift(str, amount + 26);
  }

  var output = "";

  for (var i = 0; i < str.length; i++) {
    var c = str[i];
    if (c.match(/[a-z]/i)) {
      var code = str.charCodeAt(i);
      if (code >= 65 && code <= 90) {
        c = String.fromCharCode(((code - 65 + amount) % 26) + 65);
      }
      else if (code >= 97 && code <= 122) {
        c = String.fromCharCode(((code - 97 + amount) % 26) + 97);
      }
    }
    output += c;
  }

  return output;
};

const benchmark = (data) => {
  const startTime = +new Date();
  const chars = data.split('');
  const a = chars.map(c => caesarShift(c, 1))
  const endTime = +new Date();
  return endTime - startTime;
}
```

Додаток Д. Класові діаграми розробленого прототипу

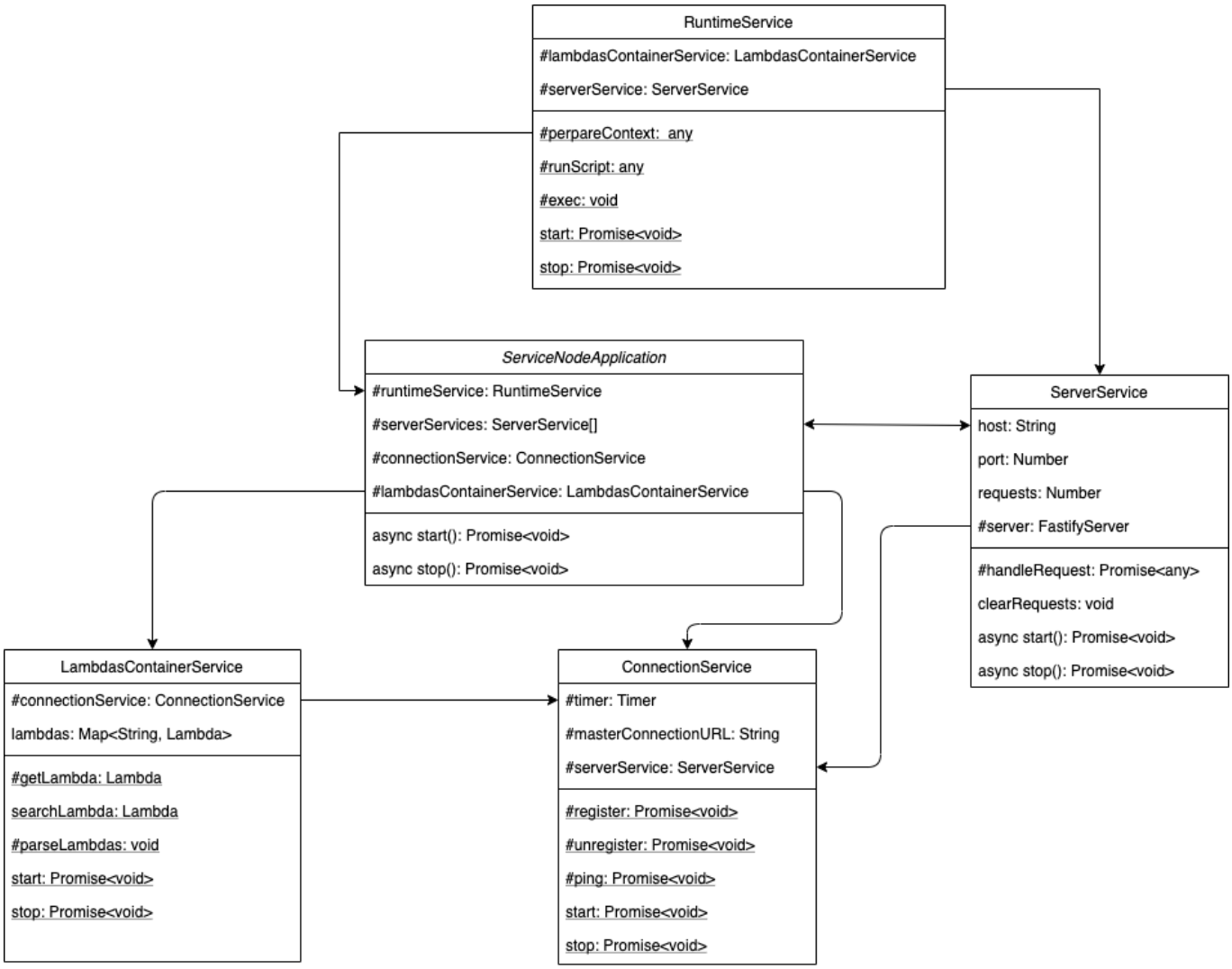


Рис.Д.1. Класова діаграма сервісу «Обслуговувач»

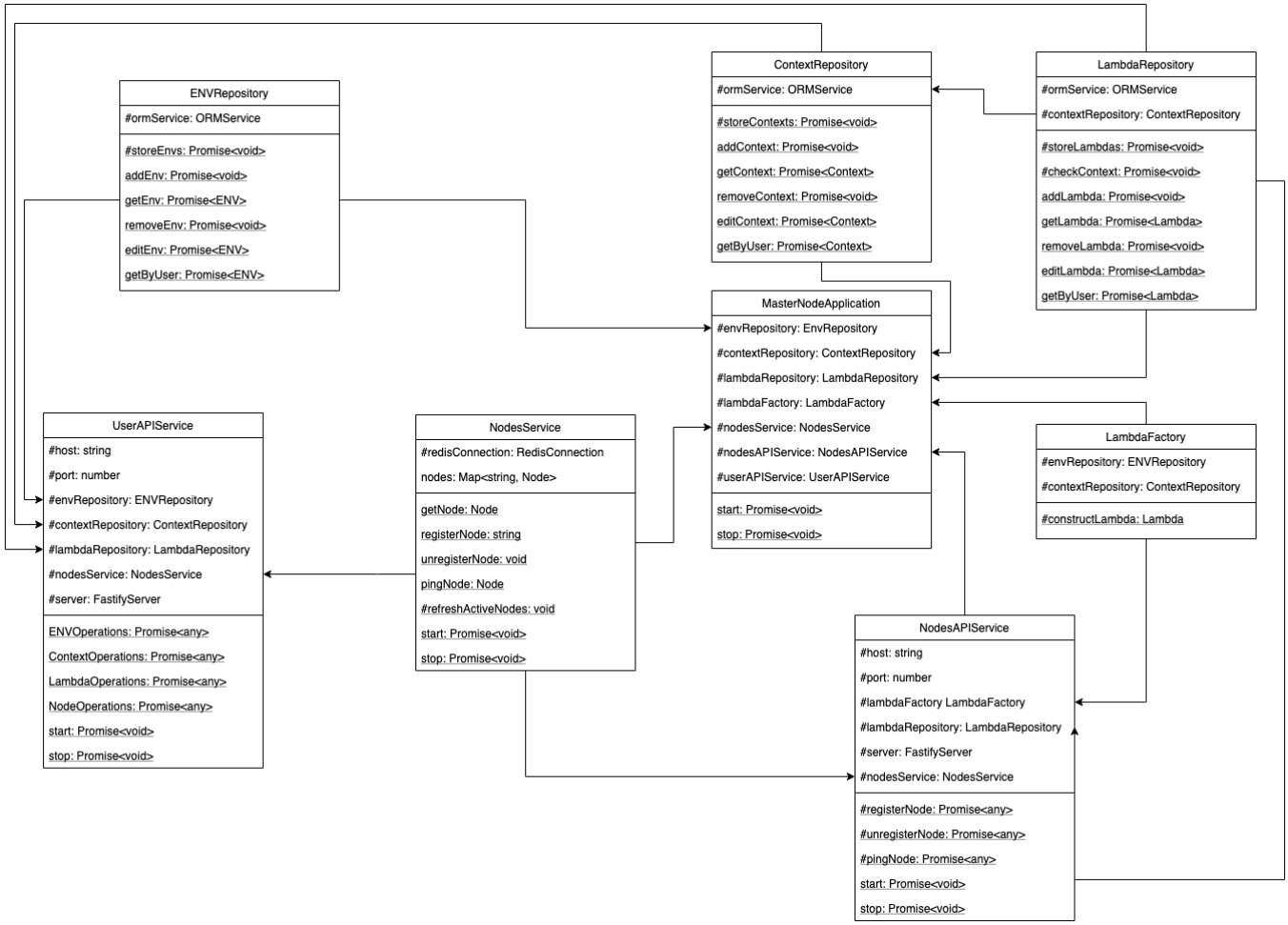


Рис.Д.2. Класова діаграма сервісу «Ядро»

Додаток Е. Процес розробки прототипу та апробації

Рис.Е.1. Процес розробки прототипу додатку

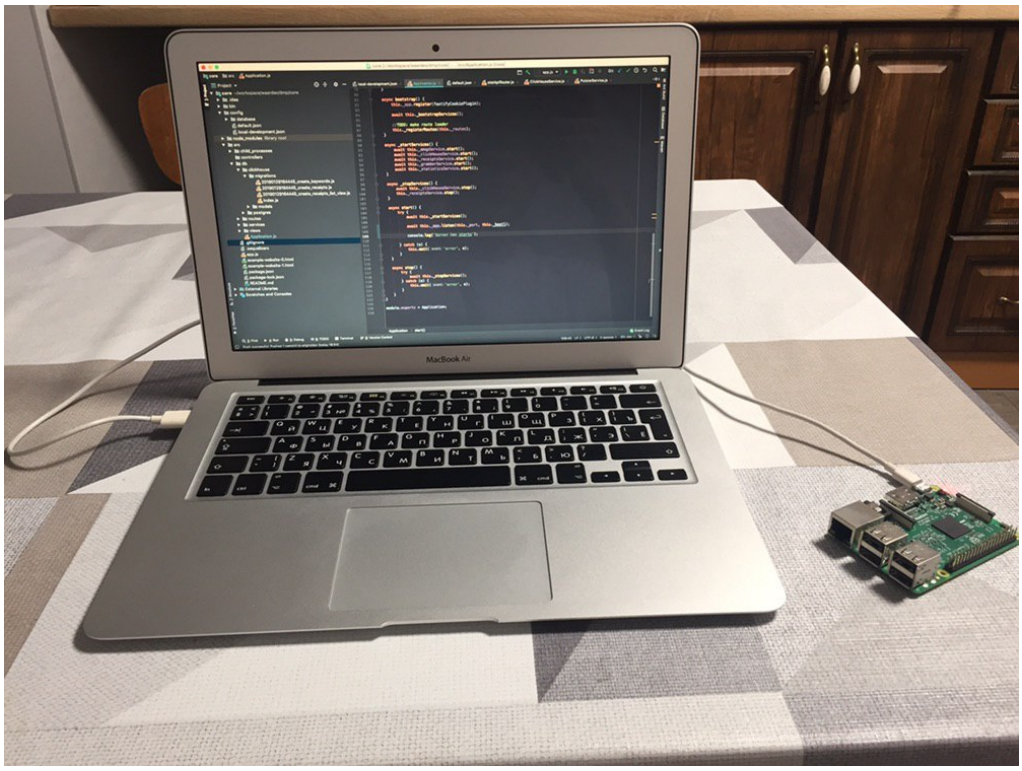


Рис.Е.2. Процес відлагодження прототипу додатку

Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

ДИПЛОМ

нагороджуються

Браженко Максим Геннадійович
Козачок Павло Анатолійович

Київський національний університет ім. Т. Шевченка

за **II** місце

у II турі Всеукраїнського конкурсу студентських наукових робіт 2017/2018 навчального року зі спеціальності ***"Автоматизація та комп'ютерно-інтегровані технології"***

Голова галузевої конкурсної комісії
25 квітня 2018 р.

І.В. Рубан

Рис.Е.3. Апробація. Диплом про заняте 2 місце ХНУРЕ, 2018

Рис.Е.4. Апробація. Диплом про заняте 2 місце ХНУРЕ, 2019