МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ ЕЛЕКТРОНІКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП’ЮТЕРНИХ НАУК

Комплексна кваліфікаційна робота бакалавра

ІНФОРМАЦІЙНА СИСТЕМА КЕРУВАННЯ ПІЦЕРІЄЮ. ІНФОРМАЦІЙНЕ І ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ БЕЗСЕРВЕРНОГО ДОДАТКУ ДЛЯ ОНЛАЙН ОБРОБКИ ЗАМОВЛЕНЬ

Здобувач освіти гр. ІН – 82 Алла МАЛЮК

Науковий керівник,
кандидат технічних наук, доцент,
доцент кафедри комп’ютерних наук Ігор ШЕЛЕХОВ

Завідувач кафедри
доктор технічних наук, професор Анатолій ДОВБИШ

СУМИ 2022

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ ЕЛЕКТРОНІКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК

Затверджую_____
Зав. кафедрою Анатолій ДОВБИШ
"_____" _____2022 р.

ЗАВДАННЯ

до кваліфікаційної роботи

здобувача вищої освіти четвертого курсу, групи ІН-82 спеціальності «122 – Комп'ютерні науки» денної форми навчання Малюк Алли Сергіївни.

Тема: «ІНФОРМАЦІЙНА СИСТЕМА КЕРУВАННЯ ПІЦЕРІЄЮ. ІНФОРМАЦІЙНЕ І ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ БЕЗСЕРВЕРНОГО ДОДАТКУ ДЛЯ ОНЛАЙН ОБРОБКИ ЗАМОВЛЕНЬ»

Затверджена наказом по СумДУ
№_____ від _____2022 р.

Зміст пояснювальної записки: 1) аналітичний огляд сучасних безсерверних технологій; 2) Огляд постачальників та особливостей данної розробки; 3) Постановка задачі; 4)Вибір методу розв'язання задач; 5) Опис програмної реалізації та тестування.

Дата видачі завдання «_____»_____ 2022 р.
Керівник роботи _____Ігор ШЕЛЕХОВ
Завдання прийняв до виконання _____ Алла МАЛЮК

РЕФЕРАТ

Записка: 48 стор., 28 рис., 2 табл., 1 додаток, 26 джерел.

Об’єкт дослідження — Процес проєктування інформаційної системи керування піцерією на основі безсерверних технологій .

Мета роботи — Розробка та програмна реалізація інформаційної системи керування піцерією на основі безсерверних технологій за допомогою застосування моделі FaaS.

Методи дослідження — методи проєктування інформаційних систем, методи оптимізації структури баз даних, хмарні технології.

Результати — у роботі було проведено детальний аналіз технологій та постачальників безсерверних обчислень. На базі отриманих результатів бало розроблено та протестовано додаток, розроблений для піцерії на основі безсерверних обчислень використовуючи постачальника AWS Lambda.

БЕЗСЕРВЕРНІ ОБЧИСЛЕННЯ, AWS LAMBDA,
ФРЕЙМВОРК JASMINE, “ФУНКЦІЯ ЯК СЕРВІС”.

ЗМІСТ

ВСТУП

На сьогоднішній день веб-додатки увійшли до нас в розумінні, як річь, без якої не можна уявити сучасне життя, завдяки яким люди отримали чудові можливості для отримання нових знань, розваг, роботи тощо. Саме на них можна знайти неосяжну кількість корисної інформації, поспілкуватися с друзями та купити речі, які не доступні в нашій країні. В умовах пандемії зросла кількість використання веб-додатків для покупки продуктів харчування, готової їжі та одягу. Адже людина сидячи дома, може відкрити додаток, оглянути весь пропонований товар, оплатити його та отримати з доставкою додому. За допомогою цього ми витрачаємо меншу кількість часу, яку можливо використати з більшою користю ніж витратити його на дорогу до магазину, де можливо цього товару і не буде, адже там може не вистачити місця, щоб умістити весь товар. Отже, як ми бачимо додатки стають все більш популярними та актуальними в сьогоденні [1].

Щоб підтримувати працездатність та зберігання інформації власник магазину або ресторану мусить витратити велику кількість грошей, на розробку та підтримку зберігання інформації у подальшому. Данні про закази зберігаються на серверах але їх покупка, обслуговування, налаштування та оренда дорого коштує. Для вирішення цієї проблеми були винайдені безсерверні додатки [2], за допомогою яких зменшується витрата коштів та часу.

Разом із швидкістю поширення інформації зростає і необхідність дослідження та розробки все більш новітніх технологій для її відображення. Особливо це стосується даних, що змінюються або оновлюються з такою швидкістю, що єдиним можливим варіантом їх відображення стає відображення в режимі реального часу.

Тому абсолютно доцільним рішенням для оптимізації додатків може стати архітектура безсерверних обчислень, новітня стратегія організації гнучких хмарних додатків, що активно розвивається.

Тож в даній роботі буде побудований безсерверний API для піцерії, який буде розгорнутий використовуючи AWS Lambda за допомогою Claudia.

1 АНАЛІТИЧНИЙ ОГЛЯД

1.1 Сучасні онлайн сервіси для обробки заказів в умовах карантину

Пандемія короновірусної хвороби (COVID-19) спричинила зростання та падіння кількох галузей у всьому світі. Хоча онлайн сервіси вже давно набирали популярності серед населення.

Через відсутність або зменшену кількість традиційних каналів покупок, клієнти починаюсь все більше використовувати платформи електронної комерції, щоб захистити покупки та транзакції. Багато магазинів, кафе та ресторанів перейшли до Інтернету, щоб продовжувати обслуговування цільової аудиторії [1].

Незважаючи на те, що платформи електронної комерції були створенні та починали розвиватись до пандемії, їх ідентичність зміцнилася з початком карантину. Поки представники бачать великий попит на свій товар, доти буде зростати перехід на онлайн сервіси для обробки заказів. Також почала зростати кількість людей, які зацікавлені в відкриті власного бізнесу в Інтернеті. Як ми могли замітити такі тенденції зросту були ще до пандемії, але карантин дав велике прискорення. Одні з найпопулярніших продуктів на платформі це медичні товари, одяг та готові до вживання продукти харчування.

Перехід від традиційних методів покупок до більш сучасних підштовхнув онлайн платформи продавати ще більше товарів першої необхідності. Порівняно з тим що було раніше, нас чекає ще довгий шлях до того, як інтернет магазини стануть домінуючим каналом, але світ до цього рухається з великою швидкістю.

Кожен ресторан намагається покращити свій сервіс, щоб здобути більшу увагу серед населення та напливу користувачів. Щоб зробити це йому треба також покращити спосіб роботи не тільки самого магазину, а й з партнерськими каналами, наприклад доставкою.

Принцип онлайн продажів залишається тим самим і схожий на традиційний, але є одна відмінність це те, що ви можете зробити покупку де б ви не знаходилися.

1.2 Безсерверні обчислення

Деякі дослідники [2, 4] вважають, що безсерверні технології дозволяють прискорити розгортання додатків, зробити його більш гнучким, економічно вигідним та масштабованим.

Безсерверні обчислення [3] – це спосіб розгортання та виконання додатків у хмарній інфраструктурі з оплатою за фактичне використання, без оренди чи купівлі серверів. За планування, масштабування, балансування та моніторинг обчислювальних потужностей відповідає постачальник послуг безсерверних обчислень. Крім того, постачальник може розглядати ваші програми як функції.

Безсерверні обчислення відносяться до архітектури, в якій програми (або частини програм) виконуються на вимогу в спеціалізованих середовищах. Зазвичай вони розташовані в хмарі, хоча безсерверні обчислення також можуть працювати локально[23].

1.3 Моделі хмарних обчислень

Модель хмарних обчислень – це архітектура послуг, що пропонують постачальники хмарних платформ, результатом яких є виділення обчислювальних ресурсів для клієнтів

Спочатку розберемо альтернативні моделі хмарних обчислень для сучасних додатків та визначимо яку саме проблему вирішують безсерверні обчислення [6,8].

1.3.1 Сервери ручної конфігурації або «Інфраструктура як сервіс»

Найбільш базова модель розповсюдження обчислювальних потужностей. Зазвичай передбачає собою виділення клієнту віртуальної машини з доступом за SSH підключенням, що найближче імітує використання реального сервера як комп’ютера із встановленою операційною системою.

Дана модель іменується IaaS або «Інфраструктура як сервіс» оскільки хмарні платформи беруть на себе виключно обов’язки, пов’язані з

налаштуванням інфраструктури та позбавляє клієнтів необхідності налаштовувати власну інфраструктуру з закупівлею обладнання, його підключенням і так далі.

Переваги:

- Гнучке налаштування
- Можливість використання одного центру для всіх компонентів додатку
- Повна свобода у межах виділених обчислювальних потужностей

Недоліки:

- Необхідність повного ручного налаштування для кожного окремого сервера
- Ресурси виділяються тільки пакетами, передбаченими платформами, тобто якщо ваш додаток використовує менше обчислювальних потужностей ніж передбачено вашим тарифом на платформі, ви будете платити за повний пакет послуг
- Складнощі з масштабуванням додатків
- Необхідність залишати буфер з обчислювальних потужностей на випадок підвищеної навантаженості

Приклади сервісів:

- Droplets від Digital Ocean
- AWS EC2 від Amazon
- Compute Engine від Google
- Azure VM від Microsoft

1.3.2 Сервери ручної конфігурації або «Інфраструктура як сервіс»

Найбільш базова модель розповсюдження обчислювальних потужностей. Зазвичай передбачає собою виділення клієнту віртуальної машини з доступом за SSH підключенням, що найближче імітує використання реального сервера як комп’ютера із встановленою операційною системою.

Дана модель іменується IaaS або «Інфраструктура як сервіс» оскільки хмарні платформи беруть на себе виключно обов’язки, пов’язані з

налаштуванням інфраструктури та позбавляє клієнтів необхідності налаштовувати власну інфраструктуру з закупівлею обладнання, його підключенням і так далі.

Переваги:

- Гнучке налаштування
- Можливість використання одного центру для всіх компонентів додатку
- Повна свобода у межах виділених обчислювальних потужностей

Недоліки:

- Необхідність повного ручного налаштування для кожного окремого сервера
- Ресурси виділяються тільки пакетами, передбаченими платформами, тобто якщо ваш додаток використовує менше обчислювальних потужностей ніж передбачено вашим тарифом на платформі, ви будете платити за повний пакет послуг
- Складнощі з масштабуванням додатків
- Необхідність залишати буфер з обчислювальних потужностей на випадок підвищеної навантаженості

Приклади сервісів:

- Droplets від Digital Ocean
- AWS EC2 від Amazon
- Compute Engine від Google
- Azure VM від Microsoft

1.3.3 Безсерверні обчислення або «Функція як сервіс»

І врешті-решт, безсерверна архітектура розповсюдження хмарних потужностей надає клієнту можливість виконання коду за моделлю FaaS або «Функція як сервіс», тобто зникає навіть така абстракція як віртуальна машина, а кожна функція (що є одиницею безсерверного коду) виконується у власному окремому одноразовому контейнері, що базується на технологіях передбачених платформою [22].

Переваги:

- Відсутність налаштувань (все відбувається на стороні платформи)
- Гнучка цінова політика (Ви платите лише за той об’єм коду, що виконується)
- Гнучка конфігурація параметрів виконання коду
- Автоматичне масштабування
- Функція з помилками не призведе до повного збою програми

Недоліки:

- Необхідність писати нові чи оптимізовувати вже існуючі додатки саме під безсерверну архітектуру
- Можливі затримки у виконанні коду через автомасштабування
- Збільшення кількості функцій вимагає ретельнішої конфігурації

Приклади сервісів:

- AWS Lambda від Amazon
- Cloud Functions від Google
- Azure Functions від Microsoft
- Alibaba Functions від Alibaba Group
- OpenWhisk від IBM

1.4 Постачальники безсерверних рішень

Модель безхмарних обчислень спрямована саме на швидший старт проекту, адже час дуже цінна річ, а також одну з головних ролей грає також і використані кошти та вибір платформи, тож розглянемо одні з найпопулярніших сервісів.

1.4.1 AWS Lambda

AWS Lambda - це безсерверний обчислювальний сервіс від компанії Amazon, що запускає код у відповідь на події і автоматично керує обчислювальними ресурсами, необхідними для виконання цього коду. Сервіс був випущений у листопаді 2014 року. Та наразі є лідером серед платформ з моделлю «Функція як послуга».

Кожен екземпляр AWS Lambda — це контейнер, створений на AMI Amazon Linux, який має від 128 до 3008 МБ оперативної пам'яті (з кроком 64 МБ), 512 МБ віртуального сховища та часу виконання, значення якого можна налаштувати в діапазоні від 1 до 900 секунд (15 хвилин).

Для побудування АПІ на базі AWS Lambda використовується інтеграція з іншим сервісом на платформі AWS, а саме Amazon API Gateway, що власне обробляє HTTP або WS запити та викликає необхідні функції для отримання відповіді на запит.

1.4.2 Google Cloud Functions

Google Cloud Functions - це безсерверна платформа від компанії Google, що керується подіями. Платформа підключає ваш код до Google Cloud Platform, налаштовуючи тригери, що будуть давати відповіді на дії користувача та обробляти зміни програми.

У комбінації з іншим сервісом компанії Google, а саме Firebase, Cloud Functions можуть дуже швидко видавати повідомлення щодо оновлень даних у додатку.

1.4.3 Azure Functions

Azure Functions - це безсерверний обчислювальний сервіс від компанії Microsoft, що так точно як і попередньо розглянуті платформи, реагує на внутрішні події на платформі Azure, та обробляє їх.

Особливістю даного сервісу є тісна інтеграція з інструментами та сервісами Azure, пов’язаними з кібербезпекою, такими як: безпека мережі, захист даних, управління ризиками та ідентифікація.

1.5 Особливості безсерверних додатків

Сервери дозволяють користувачам спілкуватися з програмою та отримувати доступ до її бізнес-логіки, але керування серверами займає значний час і ресурси. Команди повинні підтримувати обладнання сервера, піклуватися про оновлення програмного забезпечення та безпеки та створювати резервні копії на випадок збою [5]. Прийнявши безсерверну архітектуру, розробники

можуть перекласти ці обов’язки на стороннього постачальника, дозволяючи їм зосередитися на написанні коду програми.

Однією з найпопулярніших безсерверних архітектур [6, 8] (рис. 1.1) є функція як послуга (FaaS). Розглянемо більш детальніше про цю архітектуру, адже ми будемо використовувати її у своєму проекті.

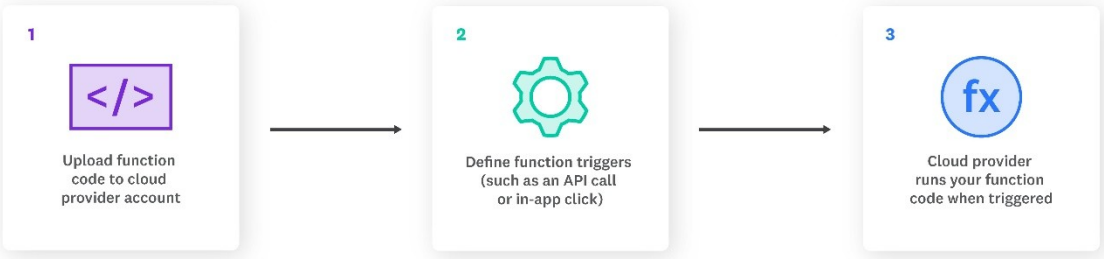


Рисунок 1.1 – Розподіл функцій в безсерверних додатках

Хоча безсерверна архітектура існує вже більше десяти років, Amazon представила першу популярну платформу FaaS, AWS Lambda, у 2014 році. Зараз більшість розробників все ще використовують AWS Lambda [7].

1.5.1 Основні переваги та недоліки безсерверних обчислень

Основні потенційні переваги безсерверних обчислень наступні [2, 12]:

- Безсерверні обчислення усувають потребу користувача підтримувати повне середовище операційної системи для виконання безсерверного коду (звідси термін безсерверний). Все, що їм потрібно зробити, – це завантажити код, налаштувати умови, які призводять до його запуску, а потім решту роботи виконуватиме безсерверна система;
- У безсерверних середовищах користувач платить лише за час, який фактично працює його код. Це більш вигідно, ніж традиційна модель ціноутворення для хмарних віртуальних машин (ВМ), де йому доводиться постійно платити за те, щоб віртуальні сервери працювали, навіть якщо вони деякий час простоюють;
- Безсерверний код виконується дуже швидко, що дозволяє легко розгортати більше екземплярів, якщо потрібно розширити конкретні

завдання. Таким чином, Serverless чудово підходить для ситуацій, де висока продуктивність або масштабованість є пріоритетом.

Теоретично переваги безсерверної системи зводяться до того, щоб зробити розгортання додатків швидшим, гнучкішим, економічно ефективним і масштабованим [4].

Однак деякі технології повністю позбавлені недоліків, а безсерверна архітектура має кілька недоліків, які потрібно буде враховувати [7, 9, 10]:

- Ніщо не є ідеальним. Безсерверна модель може демонструвати більшу затримку в певних сценаріях. Наприклад, коли вперше надходить запит користувача, платформі може знадобитися виконати холодний запуск. Розробники можуть подолати це, підтримуючи сервіси «теплими», що зберігає все попередньо завантаженим і готовим до виконання.
- Налагодження та моніторинг складніші, оскільки безсерверне середовище, за визначенням, не працює локально. Налагодження в мережі створює кілька проблем. Для покрокового коду та перевірки значень можуть знадобитися нові методи та інструменти. Окремі екземпляри безсерверних функцій можуть з’являтися і зникати залежно від того, як ними керує хост. Чим більша кількість функцій у програмі, тим більша проблема.
- Зараз ринок стикається з відсутністю незалежних стандартів, що створює ризик для використання безсерверної архітектури. Ще одна проблема — це блокування постачальника, що ускладнює заміну постачальника послуг компаніям і зменшує пул розробників, з яких організації можуть вибирати.
- Не можливо зробити безсерверну програму цілком безсерверною.
- Ще один недолік безсерверної моделі полягає в тому, що в більшості випадків вона не дозволяє повністю розгорнути програму. Запускати все в безсерверному середовищі – недоцільно чи економічно невигідно, тому ця модель передбачає спосіб запускати в ній певні частини додатку, яким

постійно потрібна висока продуктивність та значні обчислювальні ресурси.

- У переважній більшості безсерверні платформи є пропрієтарними, тобто мають власну специфіку, яка залежить від виробника. Така ситуація склалася, оскільки розробники приділяли мало уваги створенню стандартів для написання, розгортання та управління функціями, особливо в порівнянні з іншими хмарними технологіями, такими як контейнери

1.5.2 Коли і в яких проектах краще використовувати безсерверні тенології

Після огляду недоліків та переваг виникає питання коли і де використовувати безсерверні технології. Наприклад, якщо ви створюєте програму, яка інтенсивно використовує веб-сокети, безсерверні обчислення не підходять для реалізації. Застосування в AWS Lambda може працювати до 15 хвилин, після чого не зможе продовжувати отримувати повідомлення через веб-сокети.

Контейнери запускаються досить швидко, але не миттєво. При запуску контейнер може зайняти до кількох десятків мілісекунд, що може бути неприйнятним для деяких додатків.

Відсутність конфігурації є однією з основних переваг безсерверних обчислень, але ця перевага може бути серйозною перешкодою для деяких типів додатків. Якщо ви вирішили створити програму, яка вимагає конфігурації на системному рівні, вам краще подумати о можливісті використання традиційного підходу. AWS Lambda певною мірою підтримує налаштування; ви можете надати статичний двійковий файл і використовувати Node.js для його виклику, але в багатьох випадках це може призвести до великих витрат.

Ще одним важливим недоліком є так звана блокування постачальника. Самі функції не є великою проблемою, тому що є звичайні функції Node.js, але якщо вирішено реалізувати всю програму як безсерверну, то доведеться витратити багато часу з деякими сервісами. Однак це поширена проблема -

вона характерна не тільки для безсерверних обчислень, і його можна мінімізувати, вибравши гарну архітектуру для програми.

Незважаючи на ці проблеми, безсерверна архітектура надає багато рішень організаціям, які потребують оптимізації процесів і модернізації [9, 11].

1.6 Огляд фреймворків для тестування

Фреймворк — це програмна платформа, яка визначає структуру програмної системи; програмне забезпечення, яке полегшує розробку та інтеграцію різних компонентів великого програмного проекту.

1.6.1 Mocha

Широко використовується в Node.js. Він зосереджений на різних типах тестів, таких як модульне, інтеграційне та наскрізне тестування. він був розроблений для підтримки процесів тестування Node.js.

1.6.2 Jest

Створений розробниками Facebook, в основному зосереджений на модульному тестуванні. Він відомий як надійний фреймворк для тестування для React, хоча його також можна застосувати до інших бібліотек та фреймворків JavaScript.

1.6.3 Jasmine

Проста платформа для тестування JavaScript без DOM. Jasmine — це платформа для тестування JavaScript, орієнтована на поведінку. Він не покладається на браузери, DOM чи будь-який фреймворк JavaScript. Таким чином, він підходить для веб-сайтів, проектів Node.js або будь-де, де може працювати JavaScript.

1.6.4 Висновок

Mocha, Jest і Jasmine — популярні фреймворки з корисними спільнотами та роками розвитку. Загалом, Mocha та Jasmine краще підходять для тестування бек-енду, оскільки спочатку вони були створені для додатків Node; тому вони

мають більше внутрішніх інструментів та документації, ніж Jest. Що стосується інтерфейсу, вибір середовища тестування зазвичай залежить від зовнішнього середовища. Jasmine частіше використовується з Angular, а Jest був створений Facebook для використання з React.

1.7 Постановка задачі

Як показує проведений аналітичний огляд практична задача інформаційного аналізу і синтезу безсерверних додатків є актуальною для більшості сучасних підприємств, установ та організацій. Метою даної роботи є розробка безсервеного додатку для керування піцерією. При цьому основними завданнями роботи є:

1)

На основі розглянутої інформації буде розроблений безсерверний додаток для піцерії:

1) Створення інформаційної моделі безсерверного додатку
2) Проєктування та нормалізація бази даних безсерверного додатку
3) Вибір засобів для програмної реалізації безсерверного додатку
    а) Вибір системи керування базами даних
    б) Вибір бібліотеки для створення, структурування та розгортання безсерверного API
    в) Вибір фреймворку для тестування безсерверного додатку
4) Програмна реалізація безсерверного додатку
5) Тестування безсерверного додатку
    а) Перевірка виконання обробника GET
    б) Перевірка виконання обробника POST
    в) Перевірка виконання обробника PUT
    г) Перевірка виконання обробника DELETE
    д) Перевірка Бази даних
    е) Тестування за допомогою автотестів

-

2 ВИБІР МЕТОДУ РОЗВ'ЯЗАННЯ ЗАДАЧ

2.1 Інформаційна модель безсерверного додатку

На рис. 2.1 зображено дизайн типового трирівневого додатку [13] з окремими маршрутами для піци, замовлень та користувачів. У ньому також повинні бути точки входу для чат-ботів та обробника платежів. Всі маршрути повинні запускати деякі функції-обробники на рівні бізнес-логіки, а результати обробки - вирушати на рівень даних, базу даних та сховище файлів та зображень [11, 14, 16].

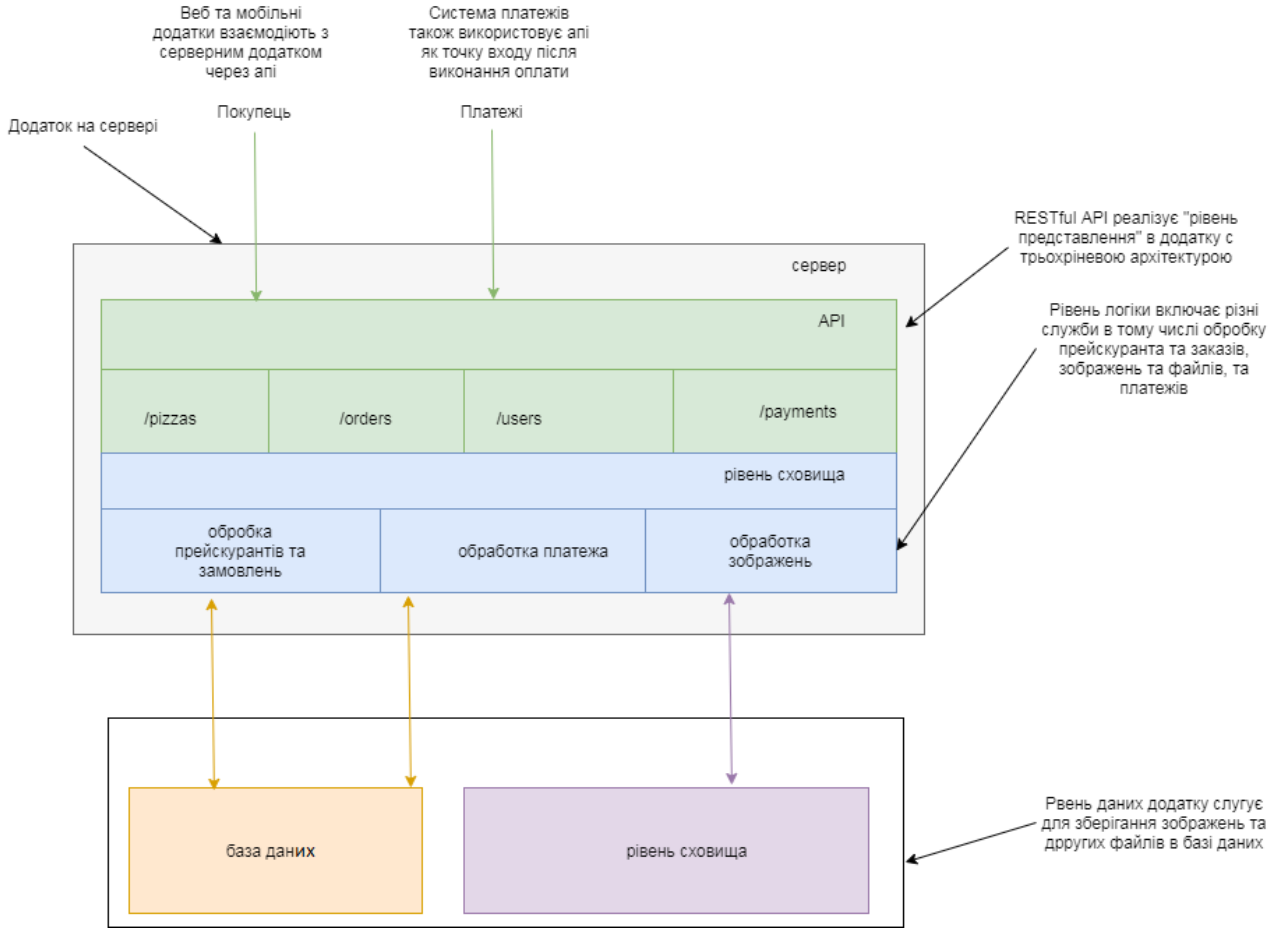


Рисунок 2.1 – Pizza API з типовим трирівневим дизайном

Цей підхід є ідеальним для невеликих додатків, у тому числі і для нашого Pizza API, принаймні до тих пір, поки кількість замовлень на піцу не зросте до певного рівня. Після цього вам потрібно буде масштабувати інфраструктуру [15]. Але, щоб масштабувати монолітну програму, необхідно відокремити шар

даних (щоб не копіювати базу даних для узгодження даних). Після цього додаток буде виглядати так, як показано на рис. 2.2. І в нас все одно залишається єдиний конгломерат із усіма маршрутами та бізнес-логікою. Таку програму можна реплікувати [15] (запускати додаткові копії, щоб збільшити пропускну спроможність), якщо у вас занадто багато користувачів, але в кожному екземплярі будуть присутні всі служби програми, незалежно від інтенсивності їх використання.

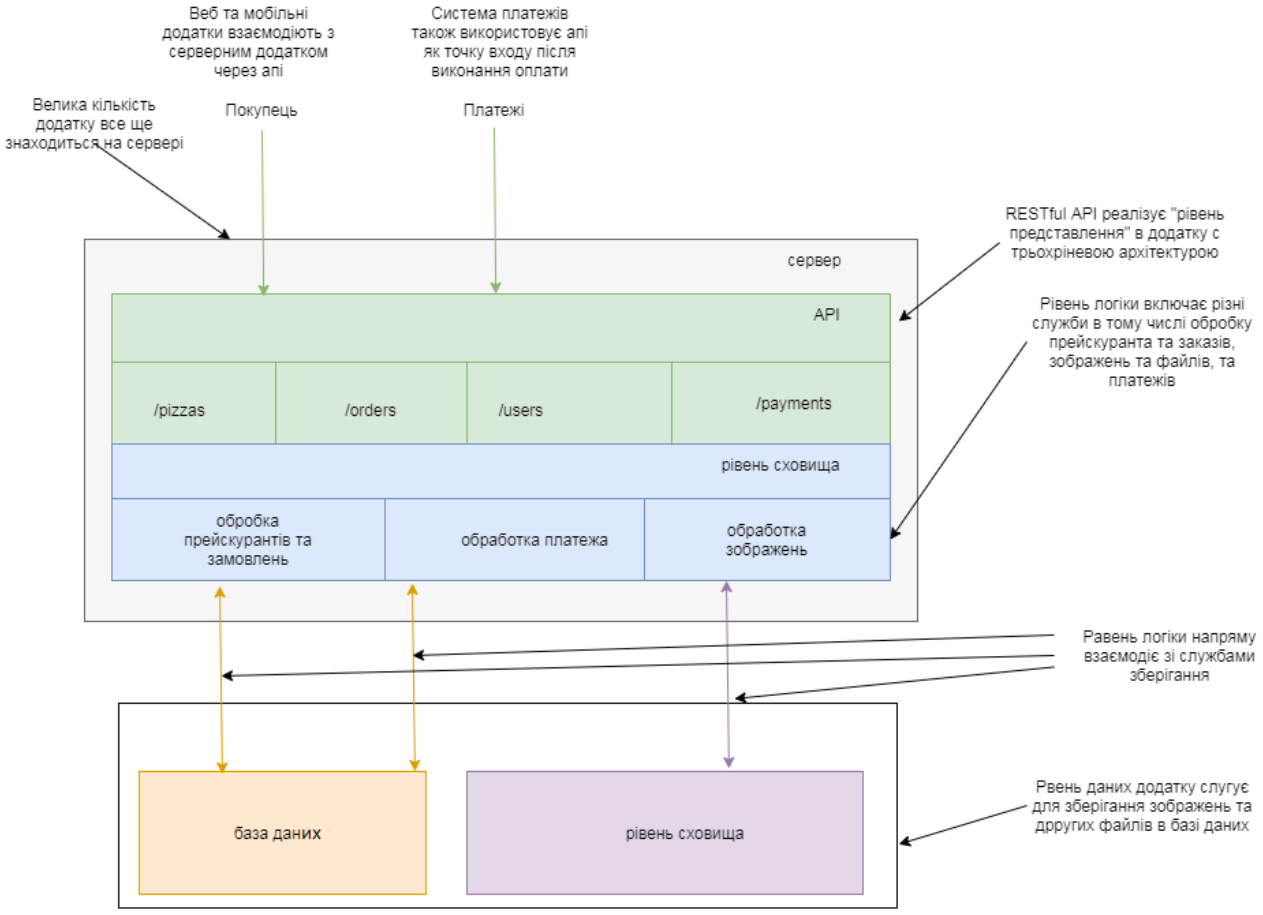


Рисунок 2.2 – Типова архітектура із зовнішньою базою даних та сховищем файлів

Нашому безсерверному Pizza API потрібна інфраструктура для роботи. Технологія безсерверних обчислень ще дуже молода, і зараз є лише кілька варіантів інфраструктури. Більшість варіантів належить великим постачальникам, оскільки для безсерверних обчислень потрібна велика та розвинена інфраструктура з підтримкою масштабування [16].

Ще одна перевага використання безсерверного додатка – рівень даних можно визивати безсерверну функцію з нашого додатка. Наприклад, коли загружається зображення піцци, є можливість запустити функцію-обробник, для зменшення розміру, якщо картинка завелика, та додавання малюнку до піцци.

Потоки обробки даних в безсерверному Pizza API можна побачити на рисунку 2.3.

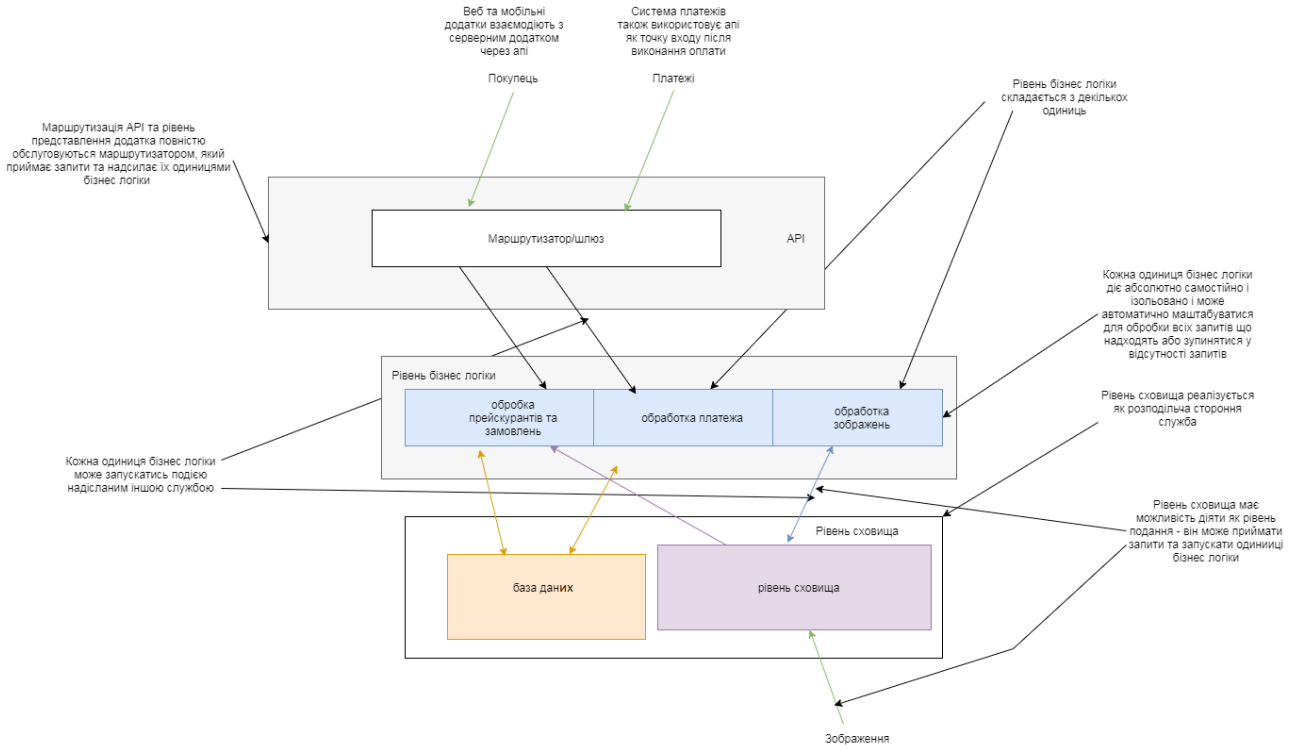


Рисунок 2.3 – Безсерверна реалізація Pizza API

З найвідоміших і найбільш сучасних інфраструктур ми будемо використовувати AWS Lambda [16].

На платформі Amazon слово безсерверні зазвичай безпосередньо пов'язане з AWS Lambda. Але для безсерверних програм, таких як Pizza API, AWS Lambda є лише одним із будівельних блоків. Щоб створити повноцінне додаток, часто потрібні інші служби, такі як служби зберігання, маршрутизації та бази даних. Перераховані всі необхідні служби, пропоновані платформою AWS [16]:

- Lambda – використовується для обчислень;
- API Gateway (шлюз API) – виконує маршрутизацію, приймаючи HTTP-запити та викликаючи інші служби залежно від маршруту;
- DynamoDB – автоматично масштабована база даних;
- Simple Storage Service (S3) – служба сховища, що реалізує абстракцію звичайного жорсткого диска та пропонує необмежений простір для зберігання.

Lambda – найважливіша частина структури нашого проекту. Lambda це безсерверний обчислювальний контейнер в AWS. Наприклад коли виникає подія така як HTTP-запит, цей контейнер викливає потрібну функцію і передає їй аргументи с даними з події, контекстом та функцію зворотнього виклику для відправлення відповіді.

AWS Lambda підтримує наступні мови програмування: Node.js, Python, Java(Java 8) та C#(.NET Core).

Під час реалізації архітектури, керованої подіями, надаємо службі, яка видає події, дозвіл на виклик нашої функції в політиці на основі ресурсів функції. Потім налаштовуємо цю службу для створення подій, які викликають нашу функцію.

Події – це данні, структуровані у форматі JSON. Структура JSON відрізняється залежно від служби, яка її видає, і типу події, але всі вони містять дані, необхідні функції для обробки події [21].

Залежно від служби виклик, керований подіями, може бути синхронним або асинхронним:

При синхронному виклику служба, яка видає подію, чекає відповіді від нашої функції. Ця служба визначає дані, які функція повинна повернути у відповідь. Служба керує політикою помилок, наприклад, чи потрібно повторювати помилки [21].

Для асинхронного виклику Lambda ставить подію в чергу, перш ніж передавати її нашії функції. Коли Lambda ставить подію в чергу, вона негайно надсилає відповідну відповідь службі, яка запустила подію. Після того, як

функція обробить подію, Lambda не повертає відповідь службі, яка запустила подію [21].

Функції Lambda мають деякі обмеження, наприклад, обмежені час виконання та обсягу доступної пам'яті. За замовчуванням функції виконується до трьох секунд, це означає, що функція роботи буде перервано по тайм-ауту, якщо вона спробує витратити більше часу обробку даних. Функція отримує 128 Мбайт ОЗУ, тобто вона може можливо надто багато складних обчислень[21].

Іншою важливою характеристикою лямбда-функцій є відсутність стану, тобто обчислюваний стан не зберігається між викликами.

Як показано на рис. 2.4, типовий порядок виконання лямбда-функцій виглядає так:

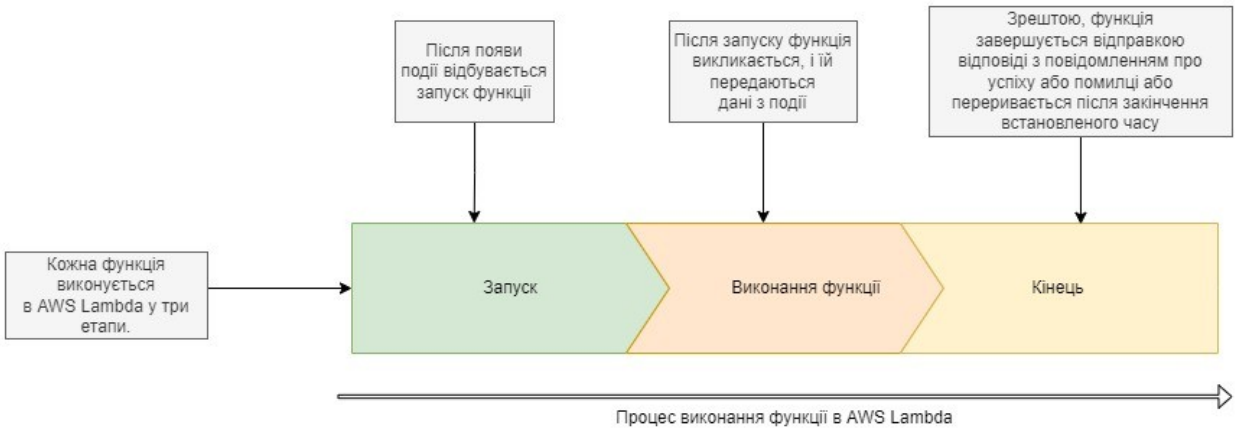


Рисунок 2.4 – Порядок виконання функції в AWS Lambda

Ще один важливий аспект, який може вплинути на нашу безсерверну роботу Pizza API – це затримка функції. Оскільки контейнерами з функціями лямбда керує постачальник, а не власник програми, немає способу дізнатися, чи буде подія оброблятися наявним контейнером або платформа створить нову. Якщо вам потрібно створити функцію перед виконанням функції і ініціалізувати новий контейнер, це може зайняти трохи більше часу - така ситуація називається холодним запуском, як показано на рис. 2.5. Час, необхідний для запуску нового контейнера, залежить від розміру програми та платформи, яка використовується для його запуску.

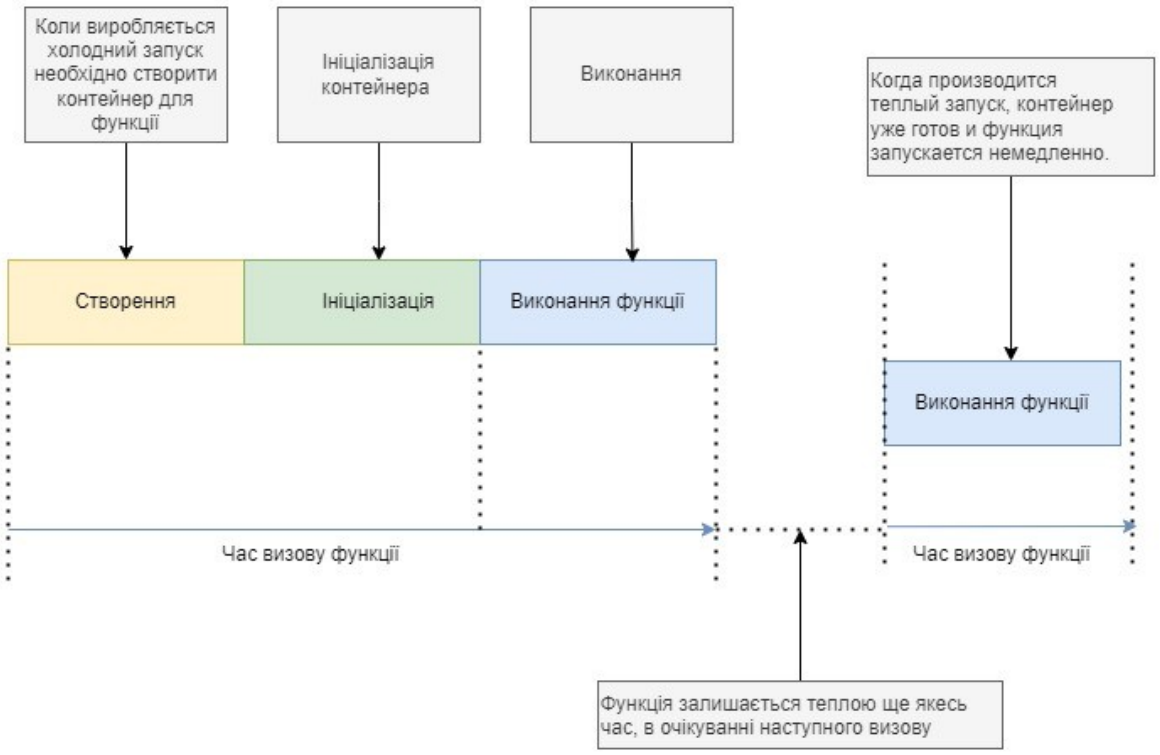


Рис. 2.5. Холодний та теплий запуск функції в AWS Lambda

2.3 Структура бази даних

Перед початком розробки слід з'ясувати, які дані повинні зберігатися. У нашому випадку елементарне замовлення піци визначається обраною піцою, адресою доставки та станом замовлення.

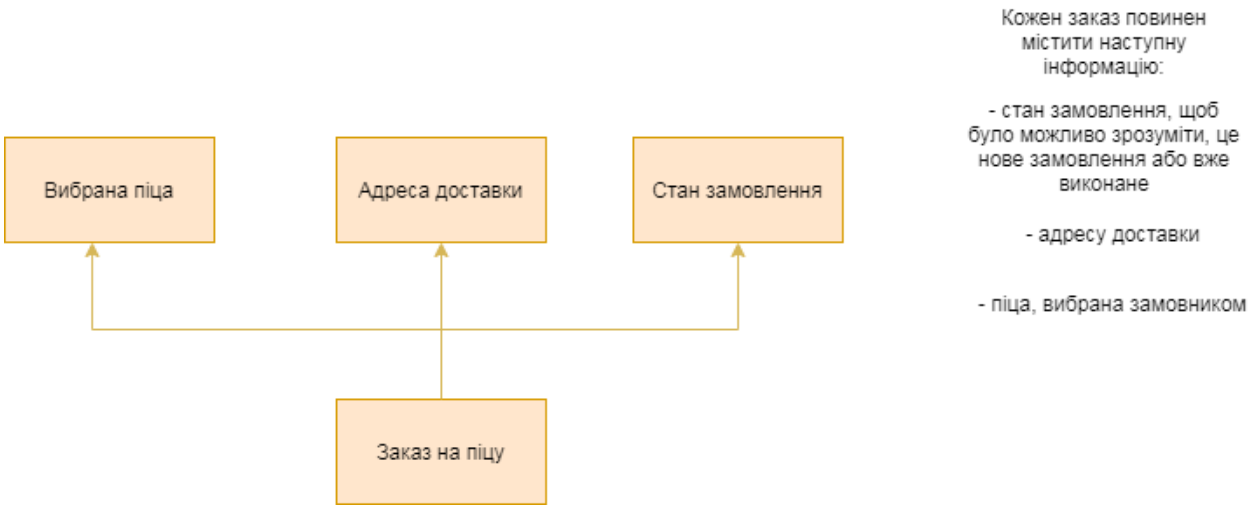


Рисунок 2.7 – Заказ піци

У традиційних додатках Node.js найчастіше використовують популярні бази даних такі як MongoDB, MySQL та PostgreSQL, але якщо поглянути на безсерверні обчислення, то кожен постачалник послуг має свої комбінаціїї систем зберігання даних.

DynamoDB [17-20] – це повноцінна, пропрієтарна служба баз даних NoSQL, пропонована Amazon у рамках портфеля послуг AWS. DynamoDB надає аналогічну модель даних і отримала свою назву від Dynamo – високодоступної структурованої системи зберігання пар ключ/значення.

DynamoDB зберігає дані у таблицях. Таблиця – це набір даних. Кожна таблиця містить кілька елементів. Елемент представляє щось єдине ціле і описується групою атрибутів. Елемент можна розглядати як об'єкт JSON, оскільки він має такі подібні характеристики:

- його ключі є унікальними;
- кількість атрибутів не обмежується;
- значення можуть бути різних типів, у тому числі числами, рядками та об'єктами.

Таблиця – це сховище моделі, представленої на рис. 2.7

Для зберігання замовлень використовуємо одну таблицю DynamoDB колекцією наших замовлень. Ми будемо отримувати замовлення через API та зберігати їх у таблиці DynamoDB. Кожне замовлення можна описати наступним набором характеристик:

- унікальний номер замовлення;
- обрана піца;
- адресу доставки;
- стан замовлення.

Ці параметри можна використовувати як ключі таблиці. Таблиця із замовленнями має виглядати, як показано в табл. 2.1.

Таблиця 2.1 – Структура замовлень у таблиці у DynamoDB

Номер замовлення	Стан замовлення	Піца	Адреса
1	очікує	Капрічоза	221б Бейкер
2	очікує	Наполетана	29 Васильківська

Найбільш простий спосіб взаємодіяти з нашою базою данних – використовувати клас DocumentClient, який працює асинхронно.

Клієнт документів спрощує роботу з елементами в amazon dynamodb, абстрагуючи поняття значень атрибутів. Ця абстракція анотує рідні типи javascript, які надаються як вхідні параметри, а також перетворює анотовані дані відповідей у власні типи javascript.

3 ІНФОРМАЦІЙНЕ І ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ БЕЗСЕРВЕРНОГО ДОДАТКУ

3.1 Вибір засобів для програмної реалізації

Розглянемо детальніше супутні технології, використані у процесі розробки проекту.

3.1.1 Вибір СУБД

При виборі підходящої бази даних було обрано найбільш рекомендовану Amazon DynamoDB.

Amazon DynamoDB – це безсерверна нереляційна транзакційна СУБД типу «ключ-значення», що розповсюджується на пропрієтарній платформі компанії Amazon, а саме на AWS.

Ключовою особливістю саме DynamoDB є її підхід до формування запитів, адже ключами у запитах можуть бути лише чітко визначені індекси, що надзвичайно прискорює швидкість обробки запитів, навіть на великих об’ємах даних. Таких підхід звичайно позбавляє можливості формування комплексних запитів, проте натомість надає клієнтам підвищену швидкість запитів у середовищі, яке не потребує налаштувань.

З продуктивністю обробки запитів також тісно пов’язана і цінова політика компанії Amazon щодо використання сервісу, адже сервіс має 2 тарифи з різною філософією:

- Тариф «Плати тільки за те, чим користуєшся» передбачає тарифікацію тільки в рамках використаних ресурсів. Причому, одиницями тарифікації є саме навантаження на системи обробки запитів, розподілене на зчитування та запис окремо. Даний тариф пропонується клієнтам, що заздалегідь не знають який об’єм трафіку очікується для додатку та який об’єм обчислювальних ресурсів їм знадобиться.
- Інший тариф навпаки створений для клієнтів з чітко прогнозованим об’ємом трафіку та тарифікується як і будь-яка інша послуга типу DBaaS, тобто база даних як сервіс, а саме у тарифікацію входять чітко виділені

обчислювальні ресурси та об’єм пам’яті для зберігання, що можуть бути масштабовані «на льоту».

Безсерверна суть Amazon DynamoDB проявляється в першу чергу в ексклюзивності на платформі розробників, тобто AWS, хоча за бажанням можна й встановити СУБД локально, компанія Amazon надає всі необхідні для цього інструменти, але при цьому попереджає, що всі версії системи, які розповсюджується поза межами платформи AWS є жорстко обмеженими по функціоналу та не є пристосованими для реального використання у продуктах, тобто є виключно інструментами для розробників.

Іншою суто безсерверною особливістю СУБД Amazon DynamoDB є тісна інтеграція з іншими безсерверними сервісами на платформі AWS, зокрема з основним провайдером безсерверних послуг – AWS Lambda. По-перше, будучи розташованими в рамках однієї платформи, Amazon DynamoDB може створювати тригери для функцій на Lambda як реакція на певні події всередині СУБД. По-друге, функції Lambda мають власну оптимізацію для програмної комунікації з DynamoDB, а режими доступу до ресурсів СУБД розглядаються напряму в конфігураційних файлах безсерверних додатків.

Отже, Amazon DynamoDB є СУБД для швидких запитів, тісної інтеграції з безсерверними додатками на платформі AWS та порівняно простої структури даних. Тобто, Amazon DynamoDB є ідеальним варіантом для додатку.

3.1.2 Використання бібліотеки Claudia.js

Claudia — це бібліотека Node.js, яка дозволяє легко розгортати проекти Node.js на AWS Lambda і API Gateway. Він автоматизує всі трудомісткі кроки розгортання та налаштування та запускає вашу програму.

Claudia побудована на основі AWS SDK для спрощення розробки. це не заміна AWS SDK або AWS CLI, а розширення, яке спрощує рішення деякі загальні завдання, такі як розгортання та налаштування.

Ось деякі з основних переваг Claudia:

- створення та модифікація функції однією командою (позбавляє від необхідності вручну архівувати програму, а потім завантажувати архів через інтерфейс користувача AWS Dashboard)
- позбавляється від операцій з шаблонами, дозволяючи зосередитися на більшому
- цікава робота, і зберігає налаштування проекту;
- спрощує версійність;
- легко навчитися - оволодіти ним, всього за кілька хвилин.

Claudia діє як інструмент командного рядка і дозволяє створювати і оновлення функцій з терміналу. Проте в екосистемі Claudia також є дві корисні бібліотеки Node.js: Claudia API Builder (дозволяє створювати API в API Gateway) і Claudia Bot Builder (дозволяє створювати чат-ботів для різних платформ обміну миттєвими повідомленнями).

На відміну від Claudia, який використовується на стороні клієнта і ніколи не розгорнуті в AWS, API Builder і Bot Builder завжди розгорнуті в AWS Лямбда (рисунок 2.6).

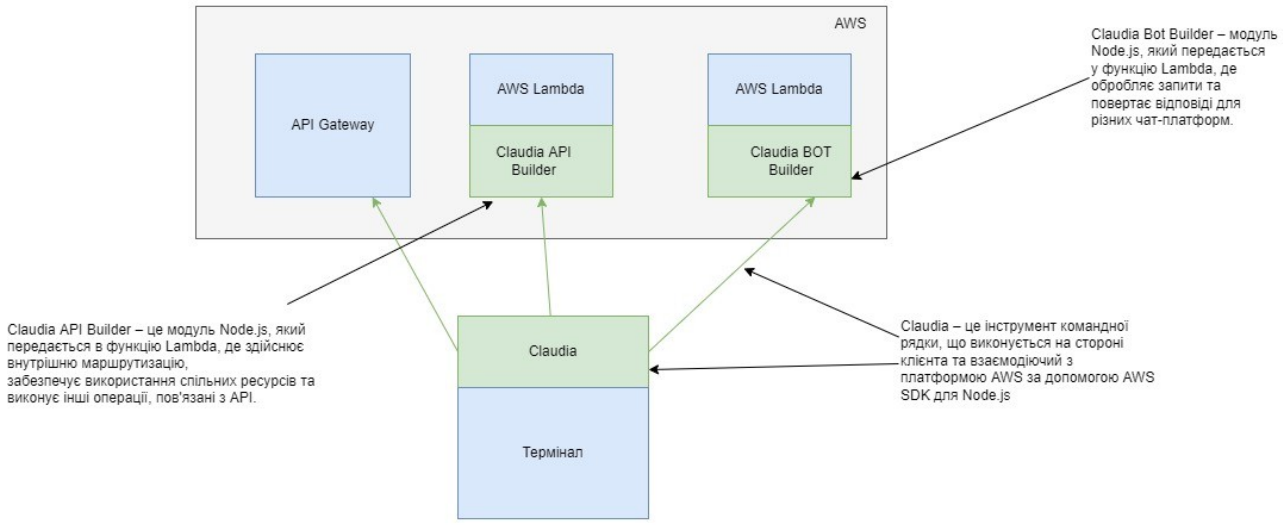


Рисунок 2.6 – зв'язки Claudia, API Builder та Bot Builder з платформою AWS

Тож, Claudia — це утиліта для розгортання, а не фреймворк. Це не абстрагує послуги AWS, але полегшує їх використання. На відміну від Seneca і

Serverless , Claudia не намагається змінити структуру чи спосіб її запуска проекта. Єдиною додатковою залежністю є додатковий API Builder, який полегшує веб-маршрутизацію runtime, але він був створений з прицілом на мінімалізм і автономію. Мікросервісні фреймворки мають багато хороших плагінів та розширень, які допомагають виконувати звичайні завдання, але Claudia свідомо зосереджується лише на розгортанні.

3.1.3 Використання фреймворку для тестування Jasmine

Jasmine — це платформа для тестування JavaScript з відкритим кодом. Він розроблений для роботи на будь-якій платформі з підтримкою JavaScript, не заважаючи ні програмі, ні IDE, і має легкий для читання синтаксис [24].

Він не залежить від інших JavaScript фреймворків і не вимагає об’єктної моделі документа (DOM), тому його можна використовувати для тестування сценаріїв браузера та додатків Node.js. Jasmine має прозорий і очевидний синтаксис, який спрощує тестування [24].

Специфікації – це назва тестів в цьому фреймворкі. Вони визначаються викликом глобальної функції Jasmine it, яка, як і опис, приймає рядок та функцію. Рядок – це ім'я специфікації, а функція – це специфікація або тест. Специфікація містить одне або кілька очікувань, які перевіряють стан коду. Очікування в Jasmine - це твердження, яке або істинне, або хибне. Специфікація з усіма справжніми очікуваннями є прохідною специфікацією. Специфікація з одним або декількома помилковими очікуваннями є невдалою специфікацією.

3.2 Короткий опис програмної реалізації

Зробимо таблицю, де буде короткий опис функцій та методів, що використовувалися у нашому проекті. (див. Таблицю 3.1)

Таблиця 3.1 – Опис програмної реалізації

№	Назва класу/методу	Короткий опис
1	Function createOrder()	Функція для створення замовлення в Pizza-Api, яку потім використовує наш API
2	Function deleteOrder()	Функція для видалення замовлення в Pizza-Api, яку потім використовує наш API
3	Function generatePresignedUrl()	Функція для генерування посилання на корзину в Pizza-Api, яка створенна для сховища малюнків
4	Function getOrders()	Функція для отримання замовленнь в Pizza-Api, яку потім використовує наш API
5	Function getPizzas()	Функція для отримання піцц Pizza-Api, яку потім використовує наш API
6	Function updateDeliveryStatus()	Функція для оновлення статусу замовленнь в Pizza-Api, яку потім використовує наш API
7	function updateOrder()	Функція для оновлення замовлення в Pizza-Api, яку потім використовує наш API
8	module.exports	Експорт функції-обробника.
9	api.registerAuthorizer	API для авторизації в Amazon AWS
10	api.get	Get response в API
11	api.post	Post response в API
12	api.put	Put response в API
13	api.delete	Delete response в API

3.3 Практична реалізація

Першим із методів які були реалізовані це метод GET, якиий повертав нам весь список піцц. Для того щоб створити цей метод було потрібно налаштувати обліковий запис AWS та створити файл з налаштуваннями у нашому проекті, встановити Node.js та диспечер пакетів NPM, встановити Claudia за допомогою NPM як глобальну залежність.

Після ініціалізації додатка Node.js та встановлення claudia-api-builder за допомогою диспечера пакетів NPM, реалізуємо точку входа до нашого додатку. Створимо файл з назвою api.js та додамо метод GET з claudia-api-builder, який отримує два аргументи шлях та фунцію-обробник.

Обробник GET у нас виглядатиме на даному етапі так, як показано на рисунку 3.1.

```
'use strict'
const Api = require('claudia-api-builder') //підключення модуля claudia-api-builder
const api = new Api() //створення екземпляру claudia-api-builder

api.get('/pizzas', () => {   //визначення маршруту та обробника
 return [   //повернення списку піцц
  'Capricciosa',
  'Quattro Formaggi',
  'Napoletana',
  'Margherita'
 ]
})

module.exports = api  //експорт екземпляру claudia-api-builder
```

Рисунок 3.1 – Створення обробника GET

Щоб розгорнути API застосуємо бібліотеку Claudia за допомогою команди ‘claudia create’ в якій передаємо як аргументи регіон та точку входу нашого проекту. Після виконання команди у відповіді ми повинні побачити корисну інформацію про розгорнуту функцію та наш API(див. Рисунок 3.2).

```
PS C:\Users\Alla.maliuk\WebstormProjects\Maliuk_Diplom_test\pizza-api> claudia create --region eu-central-1 --api-module api
packaging files npm install -q --no-audit --production

validating package     npm dedupe -q --no-package-lock
npm WARN deprecated socks@1.1.10: If using 2.x branch, please upgrade to at least 2.1.6 to avoid a serious bug with socket data flow and an import issue introduced in 2.1.0
npm WARN deprecated querystring@0.2.0: The querystring API is considered Legacy. new code should use the URLSearchParams API instead.
npm WARN deprecated uuid@2.0.3: Please upgrade  to version 7 or higher.  Older versions may use Math.random() in certain circumstances, which is known to be problematic.  See https://v
8.dev/blog/math-random for details.
npm WARN deprecated uuid@3.3.2: Please upgrade  to version 7 or higher.  Older versions may use Math.random() in certain circumstances, which is known to be problematic.  See https://v
saving configuration
{
  "lambda": {
    "role": "pizza-api-executor",
    "name": "pizza-api",
    "region": "eu-central-1"
  },
  "api": {
    "id": "ora9hui3nf",
    "module": "api",
    "url": "https://ora9hui3nf.execute-api.eu-central-1.amazonaws.com/latest"
  }
}
PS C:\Users\Alla.maliuk\WebstormProjects\Maliuk_Diplom_test\pizza-api>
```

Рисунок 3.2 – Виконання команди ‘claudia create’

Використовуючи URL яку передала у відповіть на наш запит, ми зможемо побачити як працює наш обробника GET. Для цього потрібно відкрити будь-який веб-браузер та в адресний рядок вставити URL, додавши до неї наш

створенний раніше шлях. Після відкриття посилання бачимо відповідь (див. Рисунок 3.3)

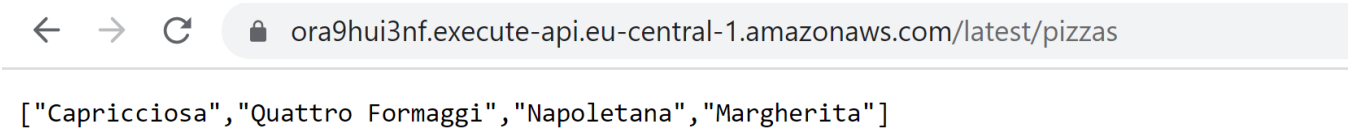


Рисунок 3.3 – Відповідь обробника GET за допомогою URL

Після додавання каталогу додаткових обробників, статичніх данних таких як список піцц з ідентифікаторами ми отримаємо нову структуру нашого проекту (див. Рисунок 3.4).

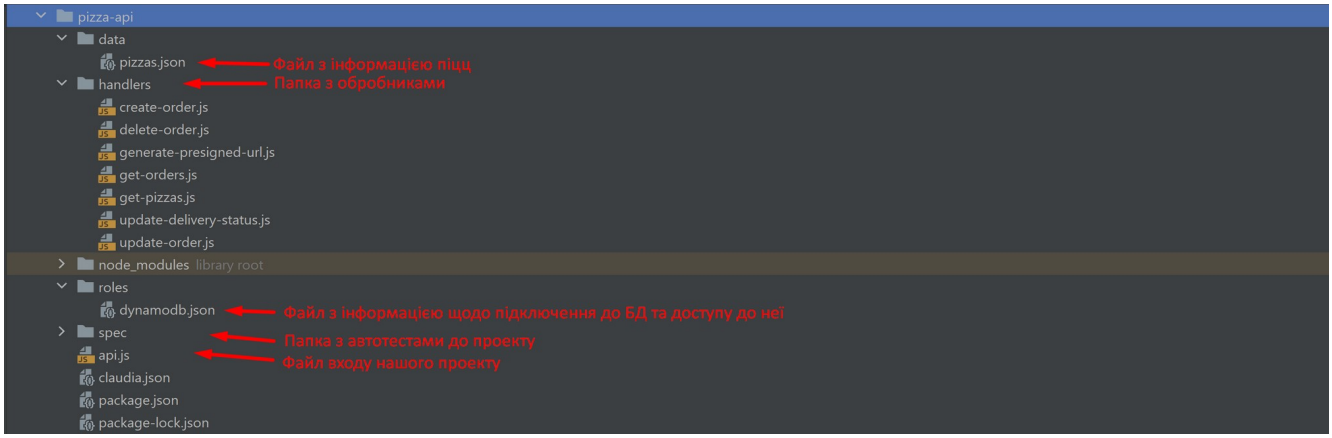


Рисунок 3.4 – Структура нашого проекту.

Тож маємо ророблений API з використанням AWS Lambda, API Gateway та Claudia API Builder, підключений до безсерверної бази даних DynamoDB. Також, була реалізована аутентифікація та авторизація за допомогою AWS Cognito.

3.4 Тестування

Фінальною частиною розробки є ручне та автоматизоване тестування. Як було сказано в аналітичному огляді для автоматизованного тестування було використано фреймворк Jasmine.

План тестування:

- Перевірка виконання обробника GET (повернення списку піцц та окремої за ідентифікатором який існує та якого не існує у списку піцц);
- Перевірка виконання обробника POST(успішна та не успішна ситуація);
- Перевірка виконання обробника PUT;
- Перевірка виконання обробника DELETE;
- Перевірка Бази данних(повернення таблиці з замовленнями та додавання замовлення);
- Тестування за домопогою автотестів.

3.4.1 Перевірка виконання обробника GET

Перевіримо оборник GET за допомогою URL, яка повертається при розгорнутанні API за допомогою бібліотеки Claudia. Щоб краще перевірити наше API використаємо «позитивний» та «негативний» спосіб тестування.

Для повернення списку піцц візмемо наш URL та додамо обробник GET Який не використовує ідентифікатори (див. Рисунок 3.5).

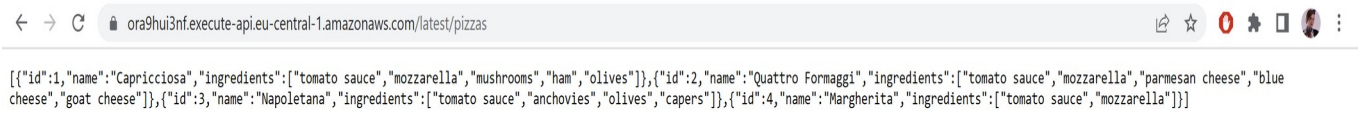


Рисунок 3.5 – Обробник GET – повернення списку піцц.

«Позитивне» тестування для повернення піцци за ідентифікатором (див. Рисунок 3.6).

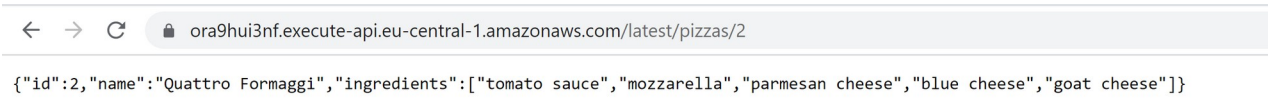


Рисунок 3.6 – Обробник GET – повернення піцци за іденифікатором.

«Негативне» тестування для повернення піцци за не існуючим ідентифікатором (див. Рисунок 3.7).

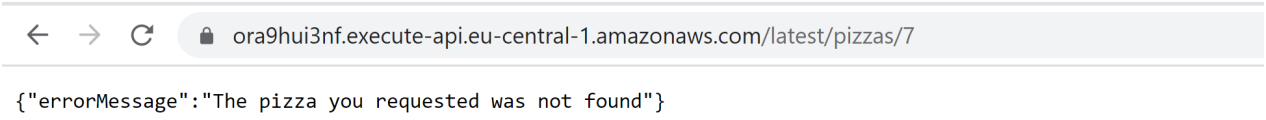


Рисунок 3.6 – Обробник GET – повернення піцци за не існуючим іденифікатором.

3.4.2 Перевірка виконання обробника POST

Перевіримо обробник POST /orders за допомогою бібліотеки libcurl, а саме командою curl.

Curl — це кросплатформна утиліта командного рядка, яка дозволяє вам взаємодіяти з багатьма різними серверами через багато різних протоколів із синтаксисом URL [27].

Обробник POST /orders route з помилковою ситуацією (див. Рисунок 3.7)

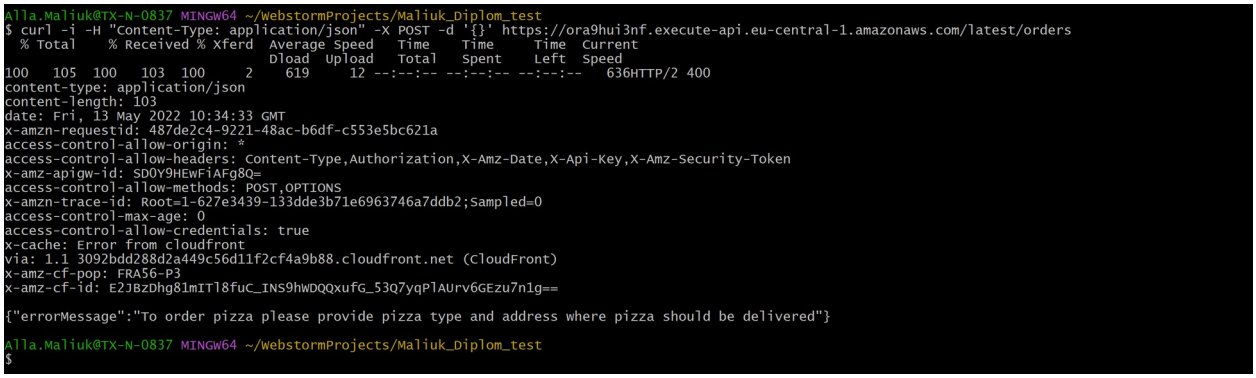


Рисунок 3.7 – Обробник POST – помилкова ситуація

Обробник POST /orders route з успішною ситуацією (див. Рисунок 3.8)

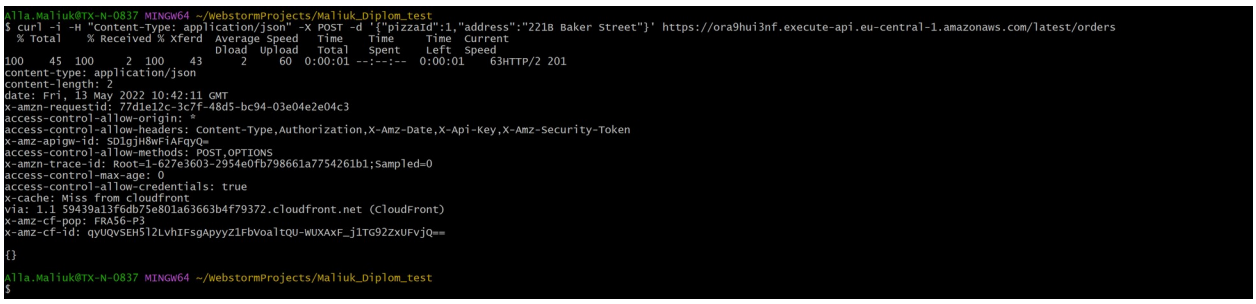


Рисунок 3.8 – Обробник POST – успішна ситуація

3.4.3 Перевірка виконання обробника PUT

Перевіримо обробник PUT за допомогою бібліотеки libcurl, а саме командою curl. Обробник PUT створений для оновлення замовлення. Перевірку цього обробнику можна подивитися на рисунку 3.9.

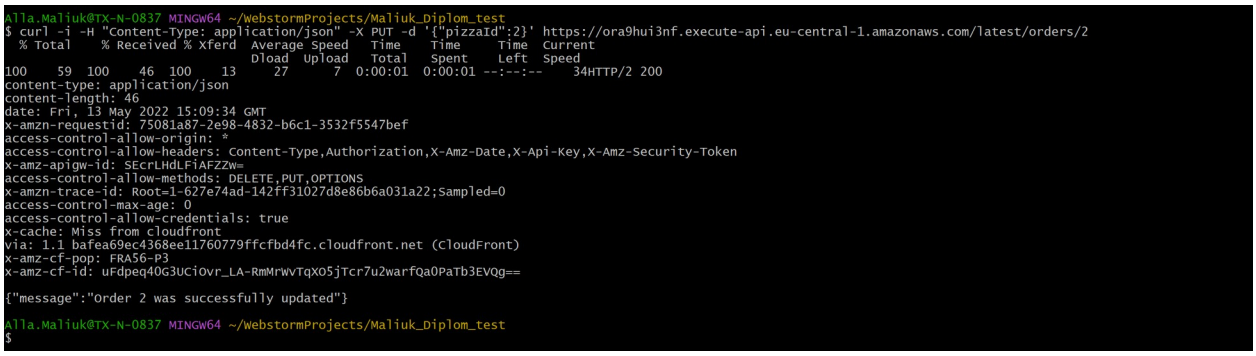


Рисунок 3.9 – Обробник PUT

3.4.4 Перевірка виконання обробника DELETE

Перевіримо обробник DELETE за допомогою бібліотеки libcurl, а саме командою curl. При успішному виконанні повинен повернутися код запиту 200 з пустим тілом запиту. (див. Рисунок 3.10).

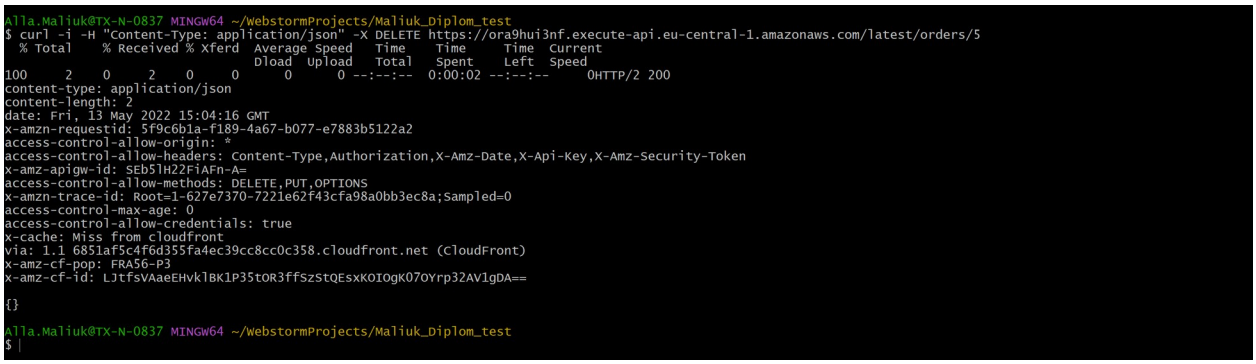


Рисунок 3.10 – Обробник DELETE

3.4.5 Перевірка Бази даних

Для перевірки, що замовлення додаються до таблиці Бази данних потрібно спочатку відправити запит POST, тоді зможемо відслідити кількість записаних рядків. Для перевірки використовуємо функцію scan (див. Рисунок 3.11).

Рисунок 3.11 – Перевірка Бази данних

Додамо нове замовлення (див. Рисунок 3.12) та перевіримо що воно також додалося до БД (див. Рисунок 3.13).

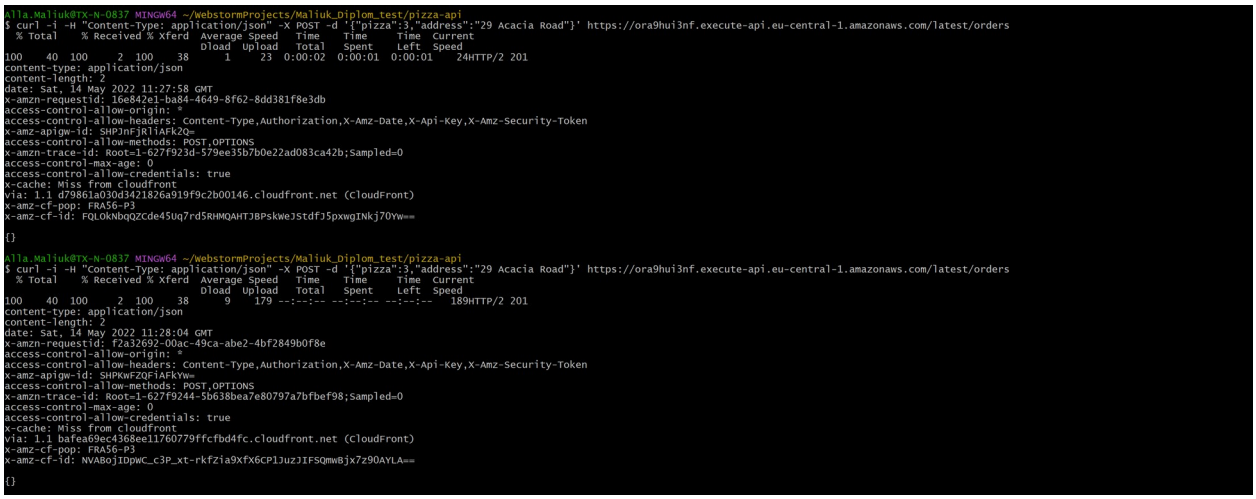


Рисунок 3.12 – Додавання нових замовлень

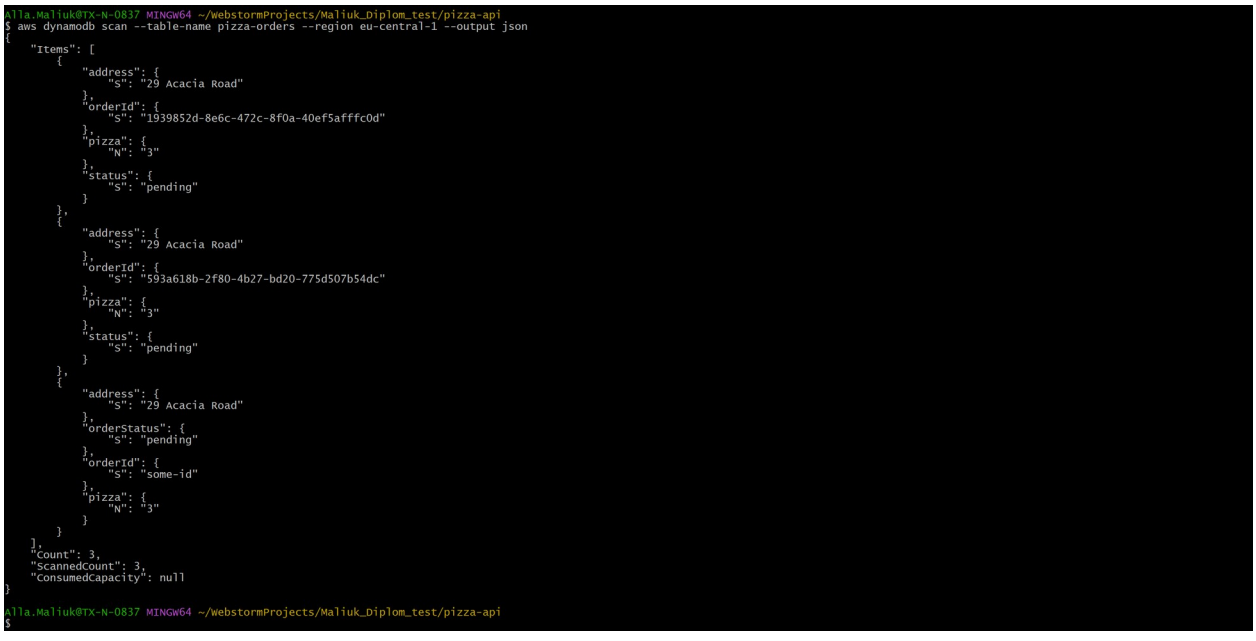


Рисунок 3.13 – Перевірка нових замовлень в БД

3.4.6 Тестування за допомогою автотестів

Як ми бачили на рисуноку 3.4 всі автоматизовані тести зберігаються у каталозі spec. В цій папці каталог support з конфігурацією для сценарію запуска Jasmine, а також каталог handlers, де зберігаються саме тести.

В файлі jasmin.json зберігається налаштування для сценарію запуска (див. Рисунок 3.14).

```
{
  "spec_dir": "spec", //шлях до папки з специфікаціями відносно кореного фолдеру
  "spec_files": [
    "**/*[sS]pec.js"  //назва специфікації
  ]
}
```

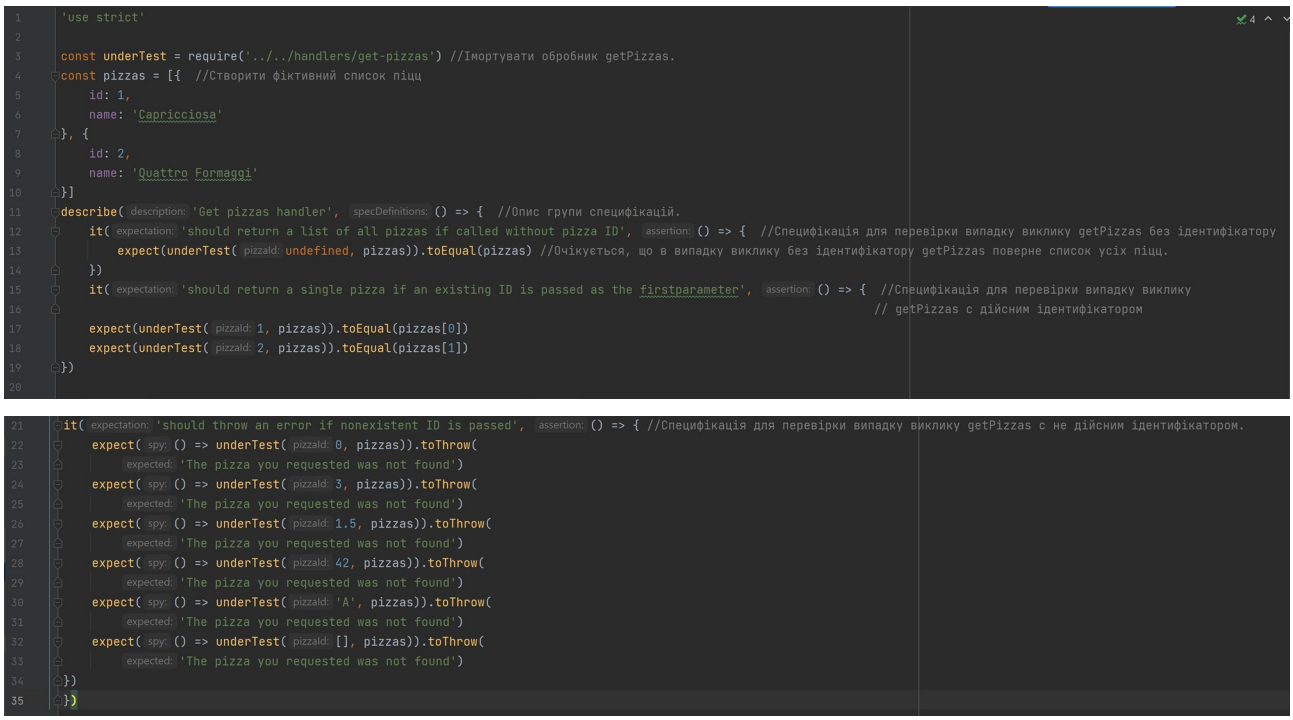
Рисунок 3.14 – конфігурація Jasmine

Для того щоб автотести запускалися, потрібно створити сценарії для запуску (див. Рисунок 3.15).

```
'use strict'

const SpecReporter = require('jasmine-spec-reporter').SpecReporter  //Імпорт бібліотеки SpecReporter
const Jasmine = require('jasmine') //Імпорт бібліотеки jasmine
const jrunner = new Jasmine() //Створення экземпляру Jasmine.
let filter  //Створення змінни filter

process.argv.slice(2).forEach(option => {  //Отримання всіх аргуметів командного рядку, окрім двох перших, и виконати цикл за ними.
 if (option === 'full') {  //Якщо отриманий аргумент full, замінити генератор звіту за замовчуванням генератором звітів специфікації
   jrunner.configureDefaultReporter( options: { print() {} })
   jasmine.getEnv().addReporter(new SpecReporter())
 }
 if (option.match( regexp: '^filter='))  //Якщо отриманий аргумент filter, зберегти значення фильтру в змінній filter
     filter = option.match( regexp: '^filter=(.*)')[1]
})

jrunner.loadConfigFile()  //Завантижити конфигурацію з файла jasmine.json.
jrunner.execute( files: undefined, filter) //Запустити тестування, передавши отримані фильтри.
```

Рисунок 3.15 – сценарій запуску

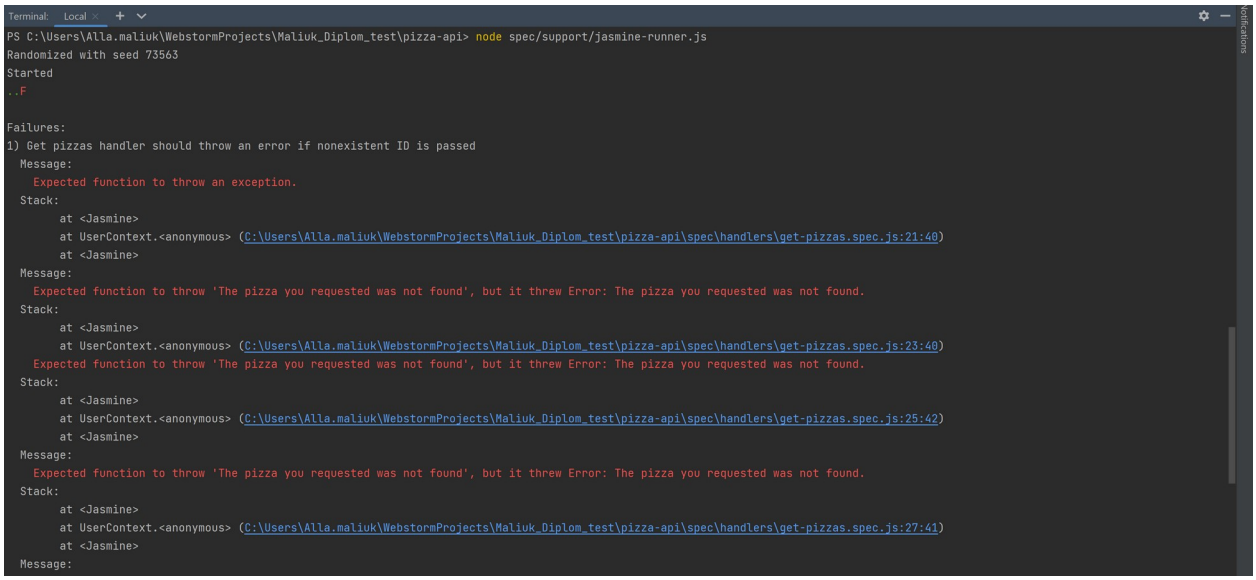
Створимо модульний тест для перевірки обробника GET (див. Рисунок 3.16).



```
'use strict'

const underTest = require('../../handlers/get-pizzas') //Імортувати обробник getPizzas.
const pizzas = [{  //Створити фіктивний список піцц
    id: 1,
    name: 'Capricciosa'
}, {
    id: 2,
    name: 'Quattro Formaggi'
}]
describe( description: 'Get pizzas handler', specDefinitions: () => {  //Опис групи специфікацій.
    it( expectation: 'should return a list of all pizzas if called without pizza ID', assertion: () => {  //Специфікація для перевірки випадку виклику getPizzas без ідентифікатору
        expect(underTest( pizzaId: undefined, pizzas)).toEqual(pizzas) //Очікується, що в випадку виклику без ідентифікатору getPizzas поверне список усіх піцц.
    })
    it( expectation: 'should return a single pizza if an existing ID is passed as the firstparameter', assertion: () => {  //Специфікація для перевірки випадку виклику
                                                                                                                          // getPizzas с дійсним ідентифікатором
    expect(underTest( pizzaId: 1, pizzas)).toEqual(pizzas[0])
    expect(underTest( pizzaId: 2, pizzas)).toEqual(pizzas[1])
})

it( expectation: 'should throw an error if nonexistent ID is passed', assertion: () => { //Специфікація для перевірки випадку виклику getPizzas с не дійсним ідентифікатором.
    expect( spy: () => underTest( pizzaId: 0, pizzas)).toThrow(
        expected: 'The pizza you requested was not found')
    expect( spy: () => underTest( pizzaId: 3, pizzas)).toThrow(
        expected: 'The pizza you requested was not found')
    expect( spy: () => underTest( pizzaId: 1.5, pizzas)).toThrow(
        expected: 'The pizza you requested was not found')
    expect( spy: () => underTest( pizzaId: 42, pizzas)).toThrow(
        expected: 'The pizza you requested was not found')
    expect( spy: () => underTest( pizzaId: 'A', pizzas)).toThrow(
        expected: 'The pizza you requested was not found')
    expect( spy: () => underTest( pizzaId: [], pizzas)).toThrow(
        expected: 'The pizza you requested was not found')
})
})
```

Рисунок 3.16 – сценарій для перевірки обробника GET

Запустивши цей тест отримаємо негативний результат (див. Рисунок 3.17), з огляду на помилку можна зрозуміти, що отримавши 0 як ідентифікатор, наш обробник повертає всеодно список піцц, а не помилку, як повинно буди при позитивному результаті тесту. Помилка виникла тому що в JS 0 це хибне значення, тому потрібно змінити умову в обробнику з того, що показане на рис. 3.18 на те, що показано на рис. 3.19. Після зміни умови та перезапуску теста ми маємо успішний результат.



```
Terminal: Local
PS C:\Users\Alla.maliuk\WebstormProjects\Maliuk_Diplom_test\pizza-api> node spec/support/jasmine-runner.js
Randomized with seed 73563
Started
..F

Failures:
1) Get pizzas handler should throw an error if nonexistent ID is passed
  Message:
    Expected function to throw an exception.
  Stack:
        at <Jasmine>
        at UserContext.<anonymous> (C:\Users\Alla.maliuk\WebstormProjects\Maliuk_Diplom_test\pizza-api\spec\handlers\get-pizzas.spec.js:21:40)
        at <Jasmine>
  Message:
    Expected function to throw 'The pizza you requested was not found', but it threw Error: The pizza you requested was not found.
  Stack:
        at <Jasmine>
        at UserContext.<anonymous> (C:\Users\Alla.maliuk\WebstormProjects\Maliuk_Diplom_test\pizza-api\spec\handlers\get-pizzas.spec.js:23:40)
    Expected function to throw 'The pizza you requested was not found', but it threw Error: The pizza you requested was not found.
  Stack:
        at <Jasmine>
        at UserContext.<anonymous> (C:\Users\Alla.maliuk\WebstormProjects\Maliuk_Diplom_test\pizza-api\spec\handlers\get-pizzas.spec.js:25:42)
        at <Jasmine>
  Message:
    Expected function to throw 'The pizza you requested was not found', but it threw Error: The pizza you requested was not found.
  Stack:
        at <Jasmine>
        at UserContext.<anonymous> (C:\Users\Alla.maliuk\WebstormProjects\Maliuk_Diplom_test\pizza-api\spec\handlers\get-pizzas.spec.js:27:41)
        at <Jasmine>
  Message:
```

Рисунок 3.17 – Негативний результат запуску тесту

```
4   function getPizzas(pizzaId, pizzas = listOfPizzas) {
5       if (!pizzaId)
6           return pizzas
7       const pizza = pizzas.find((pizza) => {
8           return pizza.id == pizzaId
9       })
```

Рисунок 3.18 – Хибна умова

```
4   function getPizzas(pizzaId, pizzas = listOfPizzas) {
5       if (typeof pizzaId === 'undefined')
6           return pizzas
7       const pizza = pizzas.find((pizza) => {
8           return pizza.id == pizzaId
9       })
```

Рисунок 3.19 – Коректна умова

У проекті були розроблені тести обробників, а також тест для перевірки зберігання замовлення в таблиці БД. Інтеграційні тести можуть підключатися до чинних служб AWS, тому що ціна замала для безсерверних обчислень. Також при тестуванні було досягнуто висноку, що при проектуванні безсерверних функцій важливо оцінити на майбутнє простоту тестування цієї функції.

ВИСНОВКИ

Отже, як ми зрозуміли, веб-додатки дуже тісно існують с життям сучасного населення. Кожен хоч раз користувався ними. В умовах пандемії кількість додатків зростає все швидше та швидше. Принцип онлайн продажів залишається тим самим і схожий на традиційний, але є одна відмінність це те, що ви можете зробити покупку де б ви не знаходилися.

В даній роботі було досліджено що таке безсерверні додатки. Як вони реалізовуються, їх недоліки та переваги. Теоретично переваги безсерверної системи зводяться до того, щоб зробити розгортання додатків швидшим, гнучкішим, економічно ефективним і масштабованим.

У ході огляду можливостей безсерверних обчислень, був розглянутий детальний порівняльний аналіз постачальників безсерверних платформ, з чого був зроблений висновок про перевагу платформи AWS Lambda від компанії Amazon над іншими безсерверними платформами, актуальними на час написання роботи.

Кожна без виключення реалізація додатків має свої недоліки, тому за допомогою якого створювати свій власний, залишається на вибір кожного.

Також було розглянуто інформаційну модель безсерверного додатку та за допомогою її прийшли до висновку, щоб створити повноцінний додаток, часто потрібні інші служби, такі як служби зберігання, маршрутизації та бази даних.

Тож, беручи до уваги всі наведені дані та розроблений приклад інтеграції двох технологій, можна зробити висновок, що наразі екосистема платформи AWS при використанні безсерверного провайдера AWS Lambda є підходящою для створення додатків, що відображають веб-контент в режимі реального часу.

СПИСОК ЛІТЕРАТУРИ

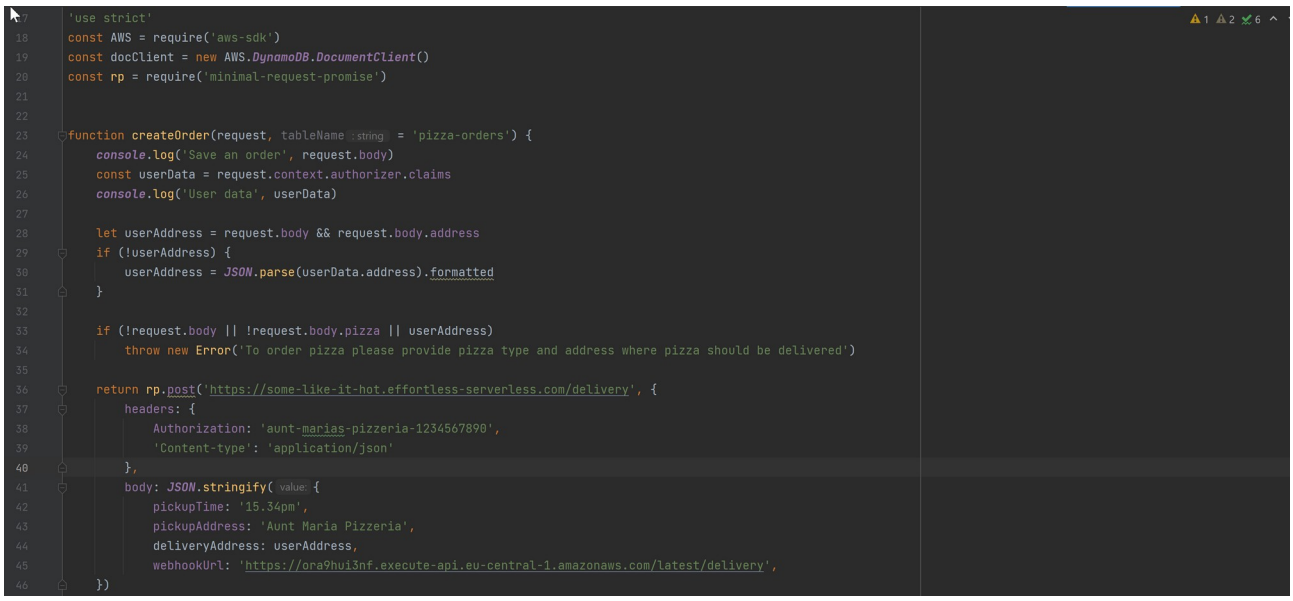
1. How online selling is thriving in the new normal [Електронний ресурс] Режим доступу до ресурсу: https://www.bworldonline.com/how-online-selling-is-thriving-in-the-new-normal/
2. 10 Advantages (and Disadvantages) of Serverless Architecture [Електронний ресурс] Режим доступу до ресурсу: https://anexinet.com/blog/10-advantages-and-disadvantages-of-serverless-architecture/
3. Serverless Architecture Overview [Електронний ресурс] Режим доступу до ресурсу: https://www.datadoghq.com/knowledge-center/serverless-architecture/
4. Añel Juan A. et al. Cloud and Serverless Computing for Scientists: A Primer / Juan A. Añel, Diego P. Montes, Javier Rodeiro Iglesias // Springer, 2020. – 94 p.
5. Calles Miguel A. Serverless Security: Understand, Assess, and Implement Secure and Reliable Applications in AWS, Microsoft Azure, and Google Cloud. – Apress Media LLC, 2020. – 364 p.
6. Chowhan Kuldeep. Hands-On Serverless Computing. – Packt Publishing, 2018. – 350 p.
7. Dabit Nader. Full Stack Serverless : Modern Application Development with React, AWS, and GraphQL. – O'Reilly Media, 2020. – 184 p.
8. Gurturk Cagatay. Building Serverless Architectures. – Packt Publishing, 2017. – 274 p.
9. Kanikathottu H. Serverless Programming Cookbook : Practical solutions to building serverless applications using Java and AWS. – Packt Publishing, 2019. – 490 p.
10. Katzer Jason. Learning Serverless: Design, Develop, and Deploy with Confidence. – O’Reilly Media, 2021. – 233 p.
11. Salehi Mohsen Amini, Li Xiangbo. Multimedia Cloud Computing Systems. – Springer, 2021. – 198 p.

12. Seroter R. (ed.) Serverless Computing. – NY: InfoQ, 2017. – 43 p.
13. Stigler Maddie. Beginning Serverless Computing: Developing with Amazon Web Services, Microsoft Azure, and Google Cloud. – Apress, 2017. – 199 p.
14. Teller Swizec. Serverless Handbook for frontend engineers: Dive into modern backend. Understand any backend. – Independently published, 2021. – 301 p.
15. Zambrano Brian. Serverless Design Patterns and Best Practices: Build, secure, and deploy enterprise ready serverless applications with AWS to improve developer productivity. – Packt Publishing, 2018. – 260 p.
16. Zickert F. React-Architect: Full-Stack React App Development and Serverless Deployment. – React-Architect, 2020. – 191 p.
17. DeBrie Alex. The DynamoDB Book. – Independently published, 2020. – 448 p.
18. Strauch Christof. NoSQL Databases. – Stuttgart Media University, 2017. – 149 p.
19. Altoros Engineering Team. The NoSQL Technical Comparison Report. – Sunnyvale, CA: Altoros, 2017. – 52 p.
20. Ganesh Chandra Deka. NoSQL: Database for Storage and Retrieval of Data in Cloud. – CRC, 2017. – 470 p.
21. Using AWS Lambda with other services. [Електронний ресурс] Режим доступу до ресурсу: https://docs.aws.amazon.com/lambda/latest/dg/lambda-services.html
22. Науменко Т. Безсерверна технологія (Functions as a service) для створення хмарних мікросервісних додатків/ Т. Науменко // Науковий журнал «Комп’ютерно-інтегровані технології: освіта, наука, виробництво» - 2018 - №33 – с. 25-30
23. AWS Lambda [Електронний ресурс] – Режим доступу до ресурсу https://aws.amazon.com/ru/lambda/?c=ser&sec=srv
24. Jasmine [Електронний ресурс] – Режим доступу до ресурсу https://jasmine.github.io/
25. Mocha & Jest [Електронний ресурс] – Режим доступу до ресурсу https://dev.to/heroku/comparing-the-top-3-javascript-testing-frameworks-2cco

26. Curl [Електронний ресурс] – Режим доступу до ресурсу https://www.hostinger.com.ua/rukovodstva/chto-takoe-curl

ДОДАТОК

Create-order.js



```
'use strict'
const AWS = require('aws-sdk')
const docClient = new AWS.DynamoDB.DocumentClient()
const rp = require('minimal-request-promise')


function createOrder(request, tableName :string = 'pizza-orders') {
    console.log('Save an order', request.body)
    const userData = request.context.authorizer.claims
    console.log('User data', userData)

    let userAddress = request.body && request.body.address
    if (!userAddress) {
        userAddress = JSON.parse(userData.address).formatted
    }

    if (!request.body || !request.body.pizza || userAddress)
        throw new Error('To order pizza please provide pizza type and address where pizza should be delivered')

    return rp.post('https://some-like-it-hot.effortless-serverless.com/delivery', {
        headers: {
            Authorization: 'aunt-marias-pizzeria-1234567890',
            'Content-type': 'application/json'
        },
        body: JSON.stringify( value: {
            pickupTime: '15.34pm',
            pickupAddress: 'Aunt Maria Pizzeria',
            deliveryAddress: userAddress,
            webhookUrl: 'https://ora9hui3nf.execute-api.eu-central-1.amazonaws.com/latest/delivery',
    })
```

```
    })
})
.then(rawResponse => JSON.parse(rawResponse.body))
        .then(response => {
            return docClient.put( params: {
                TableName: 'pizza-orders',
                Item: {
                    cognitoUsername: userAddress['cognito:username'],
                    orderId: response.deliveryId,
                    pizza: request.body.pizza,
                    address: userAddress,
                    orderStatus: 'pending'
                }
            }).promise()
        })
        .then(res => {
            console.log('Order is saved!', res)
            return res
        })
        .catch(saveError => {
            console.log(`Oops, order is not saved :(`, saveError)
            throw saveError
        })
}
module.exports = createOrder
```

Delete-order.js

```
'use strict'
const AWS = require('aws-sdk')
const docClient = new AWS.DynamoDB.DocumentClient()
const rp = require('minimal-request-promise')

function deleteOrder(orderId, userData) {
    return docClient.get( params: {
        TableName: 'pizza-orders',
        Key: {
            orderId: orderId
        }
    }).promise() Promise<PromiseResult<GetItemOutput, AWSError>>
        .then(result => result.Item) Promise<AttributeMap>
        .then(item => {
            if (item.cognitoUsername !== userData['cognito:username'])
                throw new Error('Order is not owned by your user')
            if (item.orderStatus !== 'pending')
                throw new Error('Order status is not pending')
            return rp.delete(`https://some-like-it-hot.effortless-serverless.com/delivery/${orderId}` , {
                headers: {
                    Authorization: 'aunt-marias-pizzeria-1234567890',
                    'Content-type': 'application/json'
                }
            })
        }) Promise<any>
        .then(() => {
            return docClient.delete( params: {
                TableName: 'pizza-orders',
                Key: {
                    orderId: orderId
                }
            }).promise()
        })
}
module.exports = deleteOrder
```

Get-orders.js

```
const AWS = require('aws-sdk')
const docClient = new AWS.DynamoDB.DocumentClient()

function getOrders(orderId) {
    if (typeof orderId === 'undefined')
        return docClient.scan( params: {
            TableName: 'pizza-orders'
        }).promise()
            .then(result => result.Items)

    return docClient.get( params: {
        TableName: 'pizza-orders',
        Key: {
            orderId: orderId
        }
    }).promise()
        .then(result => result.Item)
}
module.exports = getOrders
```

Get-pizzas.js

```
const listOfPizzas = require('../data/pizzas.json')

function getPizzas(pizzaId, pizzas = listOfPizzas) {
    if (typeof pizzaId === 'undefined')
        return pizzas
    const pizza = pizzas.find((pizza) => {
        return pizza.id == pizzaId
    })
    if (pizza)
        return pizza
    throw new Error('The pizza you requested was not found')
}
module.exports = getPizzas
```

Update-delivery-status.js

```
'use strict'
const AWS = require('aws-sdk')
const docClient = new AWS.DynamoDB.DocumentClient()

module.exports = function updateDeliveryStatus(request) {
    if (!request.deliveryId || !request.status)
        throw new Error('Status and delivery ID are required')
    return docClient.update( params: {
        TableName: 'pizza-orders',
        Key: {
            orderId: request.deliveryId
        },
        AttributeUpdates: {
            deliveryStatus: {
                Action: 'PUT',
                Value: request.status
            }
        }
    }).promise()
        .then(() => {
            return {}
        })
}
```

dynamodb.json

```
{
 "Version": "2012-10-17",
"Statement": [
{
"Action": [
 "dynamodb:Scan",
 "dynamodb:DeleteItem",
 "dynamodb:GetItem",
 "dynamodb:PutItem",
 "dynamodb:UpdateItem"
],
 "Effect": "Allow",
 "Resource": "*"
}
]
}
```

Get-pizzas.spec.js

```
'use strict'

const underTest = require('../../handlers/get-pizzas') //Імортувати обробник getPizzas.
const pizzas = [{  //Створити фіктивний список піцц
    id: 1,
    name: 'Capricciosa'
}, {
    id: 2,
    name: 'Quattro Formaggi'
}]
describe( description: 'Get pizzas handler', specDefinitions: () => {  //Опис групи специфікацій.
    it( expectation: 'should return a list of all pizzas if called without pizza ID', assertion: () => {  //Специфікація для перевірки випадку виклику getPizzas без ідентифікатору
        expect(underTest( pizzaId: undefined, pizzas)).toEqual(pizzas) //Очікується, що в випадку виклику без ідентифікатору getPizzas поверне список усіх піцц.
    })
    it( expectation: 'should return a single pizza if an existing ID is passed as the firstparameter', assertion: () => {  //Специфікація для перевірки випадку виклику
                                                                                                                          // getPizzas с дійсним ідентифікатором
    expect(underTest( pizzaId: 1, pizzas)).toEqual(pizzas[0])
    expect(underTest( pizzaId: 2, pizzas)).toEqual(pizzas[1])
})

it( expectation: 'should throw an error if nonexistent ID is passed', assertion: () => { //Специфікація для перевірки випадку виклику getPizzas с не дійсним ідентифікатором.
    expect( spy: () => underTest( pizzaId: 0, pizzas)).toThrow(
        expected: 'The pizza you requested was not found')
    expect( spy: () => underTest( pizzaId: 3, pizzas)).toThrow(
        expected: 'The pizza you requested was not found')
    expect( spy: () => underTest( pizzaId: 1.5, pizzas)).toThrow(
        expected: 'The pizza you requested was not found')
    expect( spy: () => underTest( pizzaId: 42, pizzas)).toThrow(
        expected: 'The pizza you requested was not found')
    expect( spy: () => underTest( pizzaId: 'A', pizzas)).toThrow(
        expected: 'The pizza you requested was not found')
    expect( spy: () => underTest( pizzaId: [], pizzas)).toThrow(
        expected: 'The pizza you requested was not found')
})
})
```

Jasmine.json

```
{
  "spec_dir": "spec",
  "spec_files": [
    "**/*[sS]pec.js"
  ]
}
```

Jasmine-runner.js

```
'use strict'

const SpecReporter = require('jasmine-spec-reporter').SpecReporter  //Імпорт бібліотеки SpecReporter
const Jasmine = require('jasmine') //Імпорт бібліотеки jasmine
const jrunner = new Jasmine() //Створення экземпляру Jasmine.
let filter  //Створення змінни filter

process.argv.slice(2).forEach(option => {  //Отримання всіх аргуметів командного рядку, окрім двох перших, и виконати цикл за ними.
 if (option === 'full') {  //Якщо отриманий аргумент full, замінити генератор звіту за замовчуванням генератором звітів специфікації
   jrunner.configureDefaultReporter( options: { print() {} })
   jasmine.getEnv().addReporter(new SpecReporter())
 }
 if (option.match( regexp: '^filter='))  //Якщо отриманий аргумент filter, зберегти значення фильтру в змінній filter
     filter = option.match( regexp: '^filter=(.*)')[1]
})

jrunner.loadConfigFile()  //Завантижити конфигурацію з файла jasmine.json.
jrunner.execute( files: undefined, filter) //Запустити тестування, передавши отримані фильтри.
```

Api.js

```
const Api = require('claudia-api-builder')
const api = new Api()
const getSignedUrl = require('./handlers/generate-presigned-url.js')
const getPizzas = require('./handlers/get-pizzas')  //Импортировать обработчик get-pizzas из каталога handlers
const createOrder = require('./handlers/create-order')  //Импортировать обработчик create-order из каталога handlers
const updateOrder = require('./handlers/update-order') //Импортировать обработчик update-order из каталога handlers
const deleteOrder = require('./handlers/delete-order') //Импортировать обработчик delete-order из каталога handlers.

api.registerAuthorizer('userAuthentication', {
 providerARNs: ['arn:aws:cognito-idp:eu-central-1:964327314239:userpool/eu-central-1_HmriURQi9']
})

// Определение маршрутов
api.get('/', () => 'Welcome to Pizza API')

api.get('/pizzas', () => {
 return getPizzas()
})
api.get('/pizzas/{id}', (request) => {
 return getPizzas(request.pathParams.id)
}, {
 error: 404
})

api.post('/orders', (request) => {
 return createOrder(request)
}, {
 success: 201,
 error: 400,
 cognitoAuthorizer: 'userAuthentication'
```

```
})

api.put('/orders/{id}', (request) => {
 return updateOrder(request.pathParams.id, request.body)
}, {
 error: 400,
 cognitoAuthorizer: 'userAuthentication'
})

api.delete('/orders/{id}', (request) => {
 return deleteOrder(request.pathParams.id, request.context.authorizer.claims)
}, {
 error: 400,
 cognitoAuthorizer: 'userAuthentication'
})
```

```
api.post('delivery', (request) => {
 return updateDeliveryStatus(request.body)
}, {
 success: 200,
 error: 400,
 cognitoAuthorizer: 'userAuthentication'
})

api.get('upload-url', (request) => { //Добавить новый маршрут GET
     return getSignedUrl()  //Вызвать обработчик getSignedURL.
   },
   { error: 400 },  //В случае ошибки вернуть HTTP-код 400.
   { cognitoAuthorizer: 'userAuthentication' })  //Потребовать авторизацию для этого нового маршрута.

module.exports = api
```

api.spec.js

```
'use strict'
const underTest = require('../api')
describe( description: 'API',  specDefinitions: () => {
    [
        {
            path: '',
            methods: ['GET']
        }, {
        path: 'pizzas',
        methods: ['GET']
    }, {
        path: 'orders',
        methods: ['POST']
}, {
    path: 'orders/{id}',
    methods: ['PUT', 'DELETE']
}, {
    path: 'delivery',
    methods: ['POST']
}, {
    path: 'upload-url',
    methods: ['GET']
}
].forEach(route => {
    it( expectation: `should setup /${route.path} route`,  assertion: () => {
        expect(Object.keys(underTest.apiConfig().routes[route.path])).
        toEqual(route.methods)
    })
})
})
```