Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра Інформатики Факультету Інформатики

Розробка безсерверного веб застосунку з використанням сервіс-провайдера Amazon Web Services

Текстова частина до курсової роботи за спеціальністю
«Інженерія програмного забезпечення» - 121

Керівник курсової роботи
Борозенний С. О.
_____
(підпис)
“____”
_____ 2022 р.

Виконав студент БП ІПЗ-4
Колесник М. А.
“____”
_____ 2022 р.

Зміст

Вступ

Сучасні цифрові продукти мають бути швидко-доступні у багатьох регіонах, мати високий рівень захисту, а також механізми продовження безперебійної роботи при критичних помилках в системі. Невід'ємними елементами є також здатність до динамічного масштабування різних модулів системи: бази даних, бізнес логіки, сховищ статичних даних, черг із повідомленнями тощо. Незважаючи на велику кількість open-source (у відкритому доступі) інструментів та відкритих джерел з навчальною інформацією, розробка справді якісного програмного забезпечення і досі залишається неймовірно складним завданням. Саме тому мою увагу привернув підхід serverless.

Serverless (безсерверний)[1] — це сучасний підхід до розробки програмного забезпечення, у якому код застосунку виконується виключно на вимогу. У цій моделі відповідальність за виділення сервера та всіх необхідних ресурсів для виконання коду користувацької програми бере на себе сервіс-провайдер. Використання serverless підходу значно спрощує розробку якісного програмного забезпечення, а також пришвидшує time-to-market (проміжок часу від формування концепції продукту до його запуску на ринок).

Найпопулярнішим[2] сервіс-провайдером на сьогодні є платформа Amazon Web Services. Саме компанія Amazon зробила відомим serverless підхід до розробки програмного забезпечення із випуском свого FaaS (function as a service) сервісу AWS Lambda у 2014 році. Вже[3] у 2017 році ринок serverless був оцінений у 3 мільярди доларів США. На сьогодні, експерти прогнозують, що до 2025 року ринок безсерверних обчислень виросте до 21 мільярду доларів США.

У цій курсовій роботі я хочу дослідити переваги і недоліки serverless, проаналізувати ситуації, у яких даний підхід є найкращим інструментом для вирішення задач інженерів. На основі здобутих знань я реалізую власний serverless веб застосунок із використанням сервіс-провайдера Amazon Web Services.

1. **Безсерверні веб застосунки**

1.1. Що таке безсерверний веб застосунок?

Безсерверний (serverless) веб застосунок — це програма яка має мережевий інтерфейс і код якої виконується за рахунок безсерверних обчислень. Під мережевим інтерфейсом мається на увазі, що клієнти взаємодіють зі програмою через мережу інтернет. Безсерверні[4] обчислення (serverless computing) — це модель виконання коду у хмарі (in cloud), у якій сервіс-провайдер (компанія, яка надає послуги serverless computing), бере на себе повну відповідальність за керування серверами, на яких будуть здійснюватися обчислення користувацьких програм. Слово «безсерверний» не дуже чітко передає ідею моделі, адже код розробників виконується на фізичних машинах сервіс-провайдера.

1.2. Головна ідея моделі serverless

Головна ідея serverless полягає у тому, що інженери програмного забезпечення можуть фокусуватися лише на написанні коду — створенні бізнес логіки додатків, і не витрачати час на інфраструктуру — налаштування, підтримку, моніторинг і масштабування серверів. Ще одним важливим елементом serverless є її модель монетизації. Вона має назву pay-for-what-you-use, або ж скорочено — pay-per-use, у якій користувачі послуг платять виключно за використані ресурси сервіс-провайдера, які той витратив для виконання коду користувацької програми.

1.3. Історія виникнення

У 2006 році була створена перша комерційна serverless платформа Zimki[5], яка не зазнала успіху. У 2008 році Google випустив Google App Engine, платформу на якій можна було виконувати обчислення для одного з фреймворків мови Python. App Engine надавала можливість розробникам створювати функції, які могли працювати максимум шістдесят секунд і не мали можливості зберігати дані на диск. Ідея виділення тимчасових ресурсів для виконання атомарних операцій одразу набула популярності.

Справжньої популярності модель serverless набула завдяки Amazon Web Services Lambda, сервісу, який представила компанія Amazon у 2014 році. AWS Lambda це FaaS (function as a service) сервіс, який власне надав можливість розробникам створювати функції, код в яких виконувався на серверах Amazon. Згодом, у 2016 році компанія Google анонсувала Google Cloud Functions, а компанія Microsoft Azure Cloud Functions — FaaS сервіси, аналоги AWS Lambda.

На сьогоднішній день[6] Amazon, Google та Microsoft є найбільшими провайдерами serverless послуг у світі. Їхні FaaS сервіси дають можливість запускати код написаний на таких відомих і актуальних мовах програмування: Python, Node.js, Java, Go, C#. Сервіси також дозволяють розробникам створювати власні середовища виконання коду. Це означає, що код написаний на будь-якій мові програмування може виконуватися за рахунок безсерверних обчислень.

1.4. Головні переваги підходу

Головними перевагами serverless є: автоматичне масштабування сервісів, гнучка модель оплати послуг pay-per-use, а також розвинена екосистема інших сервісів на платформі провайдера, які легко інтегруються із serverless сервісами. Також, очевидною і водночас очевидною перевагою є повна відповідальність сервіс провайдера за контроль над інфраструктурою.

1.4.1. Автоматичне масштабування

Автоматичне масштабування сервісів робить можливим витримувати динамічне збільшення та зменшення навантаження на них. [7]Сервіс AWS Lambda дає можливість користувачам виконувати код функціях. [8]Сервіс здатний спокійно обслуговувати від одного до десятків тисяч запитів одночасно на кожну створену функцію. При збільшенні навантаження сервіс автоматично створює і запускає нові копії функцій, щоб задовольнити попит клієнтів. При зменшенні трафіку, сервіс сам видаляє зайві копії функцій.

Принцип автоматичного масштабування сервісів лежить в основі інших відомих serverless сервісів AWS таких як Amazon DynamoDB і Amazon Aurora. [9]Amazon DynamoDB — це сервіс який дає можливість створювати нереляційну базу DynamoDB. [10]Amazon Aurora — це сервіс який дає можливість створювати сервери реляційних баз даних такі як: MySQL та PostgreSQL. Обидва сервіси здатні динамічно збільшувати та зменшувати кількість активних підключень до серверу залежно від потреб клієнтів. Ці сервіси також реалізовані таким чином, що можуть масштабуватися нескінченно за рахунок реплікацій, шардінгу та інших внутрішніх підходів та алгоритмів AWS, про які розробнику навіть не потрібно знати. Головним завданням інженера, який працює з цими serverless сервісами баз даних стає коректне проектування схем даних: таблиць та документів.

На мою думку, автоматичне масштабування сервісів є найсильнішою стороною serverless. Воно дозволяє економити гроші на спеціалістах, швидше розробляти та оновлювати програмне забезпечення, а також спати спокійно у чорну п’ятницю. Іноді дуже сумно дізнаватися, що великі бізнеси не справляються з напливом трафіку у чорну п’ятницю і, як наслідок, втрачають гроші. Так[11] у США в 2020 році у день великих знижок 48 великих веб сайтів були частково недоступні через нездатність справитися із великим напливом запитів клієнтів. Серед сервісів були такі гіганти: PayPal, Apple, HP, Wallmart, Coinbase, Adidas та Nike.

1.4.2. Модель оплати послуг pay-per-use

Pay-per-use — означає, що оплата здійснюється виключно за активну роботу сервісів. У випадку з AWS Lambda[12], споживачі послуг платять виключно за час роботи функцій і к-сть витраченого об’єму оперативної пам’яті під час їх виконання. Після завершення розробки власного веб застосунку я самостійно розрахую приблизну вартість використання усіх задіяних в роботі застосунку сервісів. Це дасть мені можливість об’єктивно оцінити приблизну вартість утримання інфраструктури на AWS та економність serverless.

1.4.3. Екосистема сервісів

Наявність розвиненої екосистеми сервісів на платформі сервіс-провайдера дає можливість швидко і легко інтегрувати їх із serverless елементами системи. Гарним прикладом сервісів, які ідеально розширюють можливості AWS Lambda є: S3, SNS, SQS, Step Functions, RDS, Kinesis, API Gateway, Route 53, Aurora, DynamoDB тощо. На[13] сьогоднішній день платформа AWS налічує понад двісті сервісів. У мажах своєї курсової роботи усі елементи мого веб застосунку будуть складатися із сервісів AWS.

1.4.4 Відсутність потреби у контролі інфраструктури

Незважаючи на те, що безсерверні обчислення насправді відбуваються на серверах, розробникам ніколи не доводиться мати з ними справу напряму. Це дає велику перевагу бізнесу, якому більше не потрібно наймати DevOps інженерів, щоб розширити, оновити чи відремонтувати якусь частину інфраструктури. Ще однією перевагою стає ріст швидкості розробки застосунків.

1.5. Недоліки і проблеми

Незважаючи на велику кількість позитивних сторін, serverless має і значні недоліки. Найбільшим є vendor lock-in, cold starts, потреба ділити ресурси з іншими клієнтами платформи сервіс-провайдера.

1.5.1. Vendor lock-in

Користувачі платформ для serverless обчислень стають справжніми заручниками своїх сервіс-провайдерів. Кожен провайдер по своєму реалізовує сервіси та визначає цінову політику. Дуже часто програми написані під одного сервіс провайдера неможливо мігрувати на платформу іншого через особливості роботи різних модулів системи.

1.5.2. Cold starts

Великою перевагою serverless сервісів є їх можливість працювати на вимогу. В той період часу, коли сервіс не є потрібний, під нього не виділяються ресурси на серверах провайдера, і, відповідно, за нього не потрібно платити. Проте кожен раз коли сервіс запускається після сну виникає cold start (холодний запуск). Cold start — це перший довгий запуск сервісу. Сервіс AWS Lambda[14] витрачає 1 секунд, щоб почати роботу.

Проблему холодного старту можливо вирішити штучними викликами функцій з певним інтервалом. Проте коли функції є частиною складного процесу це стає практично неможливо.

1.5.3. Потреба ділити ресурси із іншими користувачами

Сервіс-провайдери задля того, щоб максимально ефективно використати власні ресурси, розділяють їх між багатьма користувачами. Ця проблема спільного використання техніки з іншими сторонами відома як «multitenancy». Multitenancy може вплинути на продуктивність роботи serverless застосунку. Використання спільних ресурсів може привести до проблеми втрати безпеки даних. Для вирішення цієї проблеми сервіс-провайдери пропонують ізольовані середовища за додаткову плату.

1.5.4. Складність аналізувати роботу функцій

Коли виконання бізнес логіки програми розбивається на декілька етапів, стає дуже важко аналізувати роботу serverless функцій. Для вирішення цієї проблеми інженери мають налаштовувати додаткові системи для збору та аналізу логів (logs).

1.6. Коли варто використовувати serverless підхід?

Розробники, і компанії, які хочуть скоротити час виходу на ринок і створювати гнучкі застосунки, які можна швидко розширювати та оновлювати, можуть отримати велику користь від використання безсерверних обчислень.

Використання serverless підходу сильно знизять витрати на застосунки, у яких спостерігається непостійне використання серверних ресурсів, особливо, якщо при цьому періоди пікового навантаження чергуються з часом низького рівню трафіку або ж повної відсутності запитів.

1.7. Актуальність serverless у 2022 році

З моменту, як компанія Amazon випустила сервіс Lambda у 2014 році, serverless підхід продовжує набувати все більшою популярності. Згідно[15] із статистки зі звіту компанії Datadog по стану serverless ринку за 2020 рік, більше 50% користувачів платформи AWS використовують сервіс Lambda. Ще одним цікавим фактом є те, шо 70% найбільших у світі компаній, які займаються розробкою програмного забезпечення, використовують serverless у своїх застосунках. Сьогодні можна сміливо сказати, що serverless вже не просто нішова технологія, а повноцінний і дуже конкурентний підхід до розробки програмного забезпечення.

2. **Планування розробки власного serverless застосунку**

2.1. Опис ідеї

Після аналізу сильних і слабких сторін serverless, а також ситуацій, коли справді варто використовувати підхід, я вирішив створити застосунок для менеджменту розробки цифрового продукту, на кшталт Atlassian Jira. [16]Atlassian Jira успішно допомагає більше 50 000 командам програмістів по всьому світу створювати завдання, відстежувати статус їх виконання, а також планувати майбутні етапи розробки цифрового продукту. Serverless чудово підходить для реалізації застосунку для менеджменту розробки цифрового продукту. Головні переваги підходу:

- Пік користування застосунком припадає на робочий час, тобто з 8:00 по 19:00, а це означає, що більшу частину дня користувачі не будуть використовувати систему.
- Serverless підхід дозволить швидко запустити проект з мінімальною кількістю розробників, адже абсолютно всі елементи системи будуть збудовані на основі сервісів AWS

2.2. Основні функції застосунку

Для мінімальної робочої версії потрібно реалізувати такі функції серверної частини застосунку:

- Створення, оновлення, видалення, отримання акаунту
- Створення, оновлення, видалення, отримання користувачів на акаунті
- Аутентифікація і авторизація користувачів
- Створення, оновлення, видалення, отримання проектів на акаунті
- Створення, оновлення, видалення, отримання завдань, які належать до проектів. Кожне завдання може мати виконавця

Після реалізації основних функцій, можна розширити систему такими можливостями:

- Створення, оновлення, видалення, отримання коментарів, які належать до завдань
- Створення, оновлення, видалення, отримання фіч, (features — логічне об’єднання завдань, які варто зробити разом за певний період часу) які належать проектам, і, яким належать завдання
- Створення, оновлення, видалення, отримання релізів, (release – версія оновлення програмного забезпечення) які належать проекту і, яким належать завдання

Додатково до серверної частини застосунку я створю веб сайт із документацією REST API ендпоїнтів (endpoints) згідно специфікації OpenAPI 3.0.

3. **Розробка serverless застосунку**

3.1. Створення акаунту на платформі AWS

Першим кроком у розробці будь-якого serverless застосунку є створення акаунту на платформі сервіс-провайдера. Результатом реєстрації в AWS є створення нового root (від слова корінь) акаунту. При роботі з AWS існує декілька важливих правил, які варто дотримуватися. Використання ресурсів в AWS коштує грошей. Тому найголовнішим правилом — є захист root акаунту. Друге найважливіше правило полягає у правильному наданні ролей користувачам та ресурсам. Користувачі і ресурси мають право робити тільки те, що потрібно і нічого більше. Задля того, щоб дотриматися цих правил я створю окремого користувача, якому і буду поетапно надавати доступи до ресурсів платформи. Таким чином я захищу root акаунт і максимально ізолюю доступ до ресурсів акаунту. Ще однією хорошою практикою вважається підключення багатофакторної автентифікація для root акаунту та усіх користувачів, що я також обов'язково зроблю.

3.2. Робота з AWS IAM. Створення нового користувача і надання йому прав доступу до ресурсів та сервісів

Перед створенням нового користувача мені потрібно було розібратися, як працює сервіс AWS IAM (Identity and Access Management). Root акаунт має повний доступ до всіх сервісів та ресурсів AWS в межах акаунту. Root акаунт може створювати користувачів і надавати їм права доступу до сервісів та ресурсів, використовуючи сервіс IAM. Ресурси це об'єкти у системі, які користувач створює, наприклад S3 bucket — це ресурс сервісу AWS S3. Сама дія користувача є action (дія). Акт створення S3 bucket-а і є прикладом action. Дозвіл на здійснення action надається через policy. Policies — це є правила, в яких визначаються дозволи на здійснення action відносно ресурсів та сервісів. Policy може дозволяти, або ж забороняти дії над ресурсами і сервісами. Користувачі можуть бути об'єднані у групи. До групи можна додавати Policy і таким чином поширювати їх на всіх користувачів групи. Останнім ключовим елементом IAM є

roles (ролі). Ресурси у системі також можуть комунікувати між собою. Для того, щоб дозволити AWS Lambda функції відправити файл на S3 bucket спочатку потрібно створити роль. До створеної role потрібно додати відповідні policies. Потім role можна прикріпити до раніше створеної AWS Lambda функції.

Спочатку я створив групу користувачів із назвою «developers» (рисунок 1). Потім створив нового користувача з ім'ям «maks.kolesnyk». До нової групи я додав набір policies (рисунок 2).

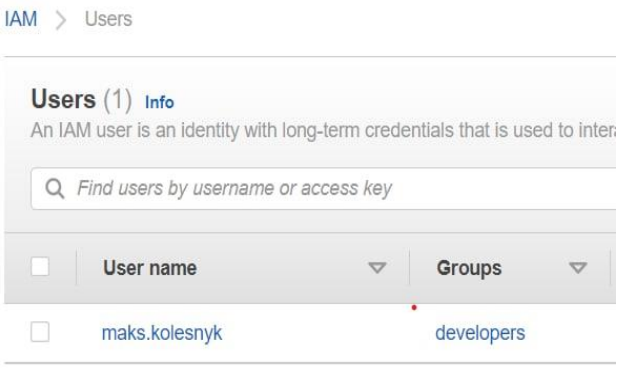

Рисунок 1– користувачі групи IAM

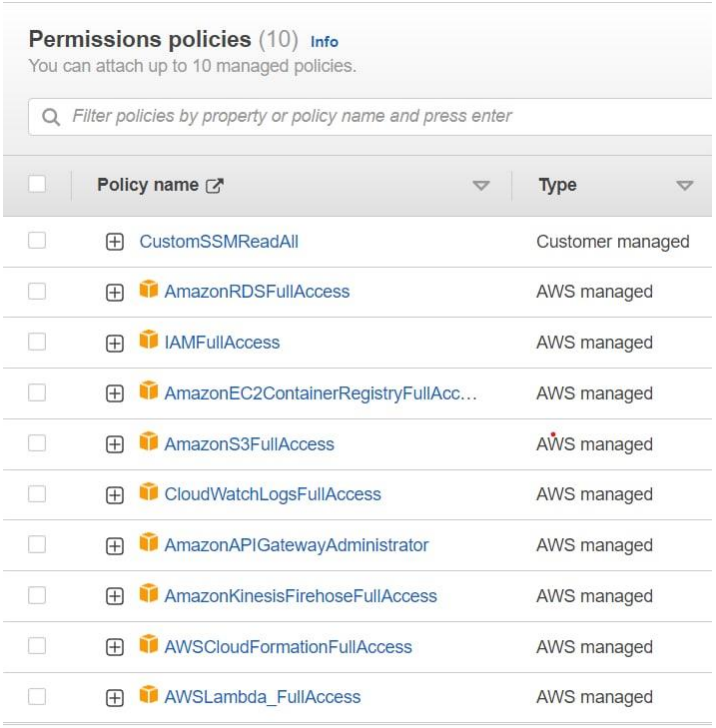

Рисунок 2 – правила в групі користувачів IAM

3.3. Налаштування роботи з сервером бази даних AWS RDS

Після описання функціональних вимог застосунку я зрозумів, що для мого проекту чудово підійде реляційна база даних. Я вирішив обрати єдиний безкоштовний варіант на платформі AWS і створив інстанс t2.micro MySQL серверу бази даних в сервісі AWS RDS (рисунок 3). Інстанс t2.micro має 1 CPU та 1 гігабайт оперативної пам'яті. В безкоштовний план також входить 20 гігабайтів пам'яті на SSD диску.

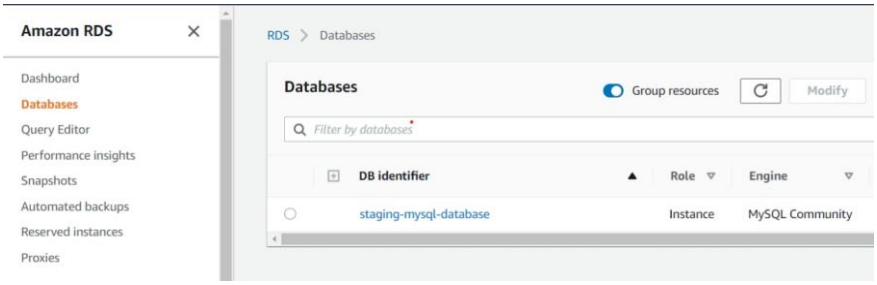

Рисунок 3 – інстанс бази даних MySQL

Сервіс[17] AWS RDS не тільки дозволяє створювати сервери баз даних. Він також робить приховану копію користувацького серверу, яка у разі критичної помилки на основному сервері буде обслуговувати запити клієнтів. AWS RDS також зберігає копію усіх збережених даних за останні 35 днів. Це дає можливість швидко відновлювати стан серверу у разі критичних помилок із записом даних. AWS RDS дає користувачу можливість вертикально масштабувати сервер бази даних у будь-який момент, а також створювати репліки бази даних на читання. На мою думку, весь цей перелік вбудованих функцій сервісу дає можливість не тільки швидко розробляти програмне забезпечення, а ще й економити чимало ресурсів на спеціалістах з баз даних.

Існує декілька підходів до налаштування доступу до серверу бази даних:

- Через AWS IAM. Усі actions, (авторизація, створення таблиць, та інші) які здійснює користувач чи сервіс визначаються за допомогою policies
- Через логін та пароль до серверу бази даних. В такому випадку користувачу надається повний доступ до серверу бази даних

Для з’єднання із сервером бази даних у своєму застосунку я користуватимусь логіном і паролем.

При розробці веб додатків дуже часто виникає проблема зберігання авторизаційних даних. Логін та пароль до серверу бази даних є прикладом інформації, яку варто тримати в безпечному і важкодоступному місці. Для таких потреб існує сервіс AWS System Manager. Одним з модулів цього сервісу є Parameter Store. Він[18] дозволяє зберігати key-value (ключ-значення) записи до яких можна отримувати доступ через систему AWS IAM.

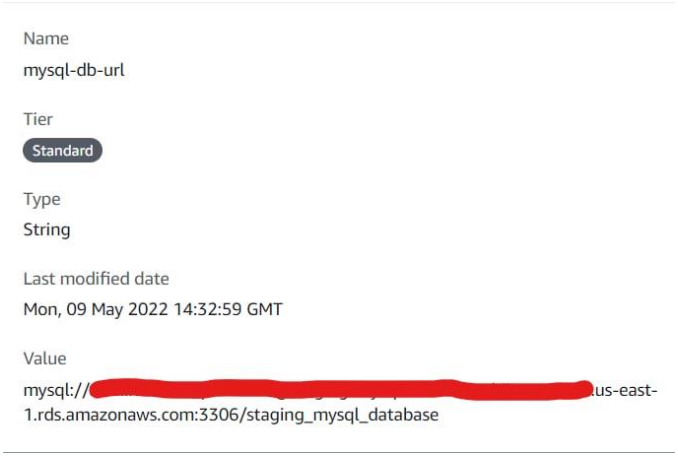

Рисунок 4 – запис в Parameter Store

Я вирішив скористатися Parameter Store і створив запис із ключем «mysql-db-url» і значенням «mysql://*/staging_mysql_database», де «*» містить в собі логін, пароль, а також url і порт інстансу сервера бази даних (рисунок 4).

3.3.1. Проектування схеми реляційної бази даних

Після успішного створення та запуску інстансу MySQL серверу бази даних я вирішив зробити схему таблиць реляційної бази даних. Для проектування схеми я використав безкоштовний сервіс [19]Diagrams.net.

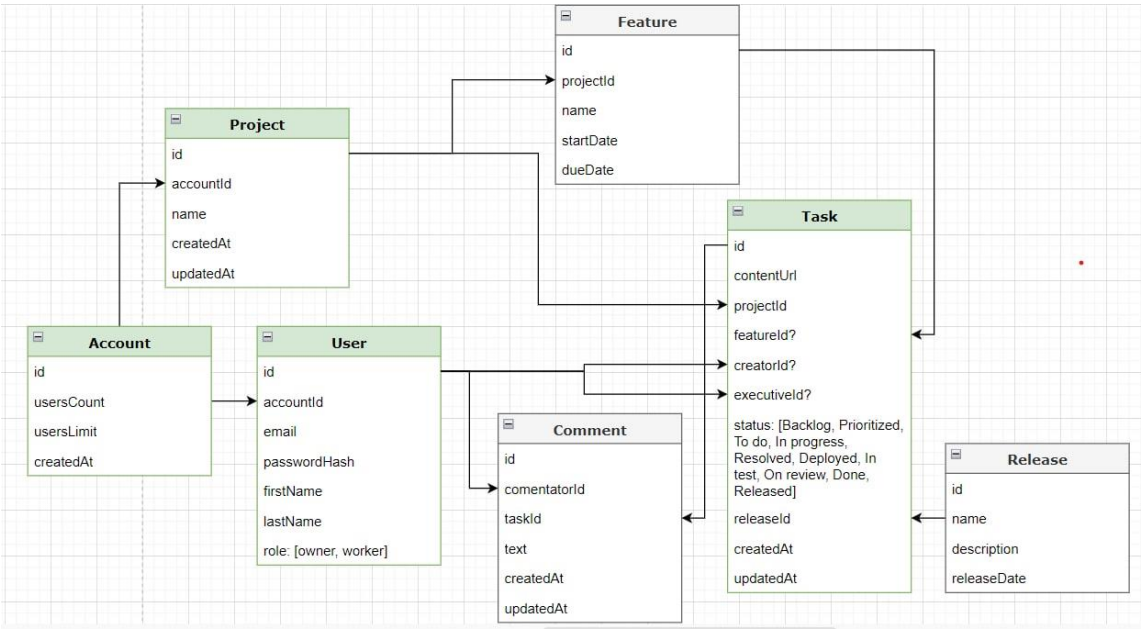

Рисунок 5 – схема таблиць реляційної бази даних

Після завершення проектування схеми таблиць (рисунок 5) я міг переходити до розробки застосунку.

3.4. Ініціалізація проекту

Перед тим як ініціалізувати проект і зробити перший коміт (commit) на GitHub, потрібно було обрати технологію, на якій буде написаний код застосунку. На момент написання курсової роботи AWS підтримує serverless розробку на таких мовах програмування: Python, Node.js, Java, Go, C#. 90%[20] усіх AWS Lambda функцій виконуються на Python та Node.js, з[21] яких 30% припадає на Node.js. Я обрав Node.js.

Node.js[22] — open-source (з відкритим кодом) платформа для виконання високопродуктивних мережевих застосунків, написаних мовою програмування JavaScript. Головною перевагою розробки на JavaScript є велика кількість open source бібліотек створених на JavaScript, а також неймовірна популярність даної мови програмування. На[23] сьогоднішній день існує більше 1300000 бібліотеку у менеджері пакетів npm (node package manager). Основні[24] архітектурні риси мови JavaScript: слабка динамічна типізація, автоматичний контроль пам’яті, прототипне наслідування і функції першого класу.

З мого досвіду роботи із мовою JavaScript мені не подобається лише слабка динамічна типізація. На щастя, у 2012 році компанія[25] Microsoft випустила мову програмування TypeScript. TypeScript відрізняється від JavaScript можливістю явного статичного визначення типів. Середовище виконання коду Node.js вміє працювати лише з JavaScript. Саме тому код написаний на TypeScript потрібно щоразу компілювати у JavaScript для виконання програми.

Визначивши мову програмування і середовище виконання коду я створив папку з проектом і базовим файлом package.json. Після цього я створив новий репозиторій на сайті [26]GitHub. Цей веб сервіс заснований на системі контроля версій Git. Він дає можливість поетапно завантажувати прогрес своєї роботи і, у разі критичних помилок повертатися на попередню версію проекту.

Після ініціалізації проекту, перш за все я вирішив налаштувати ESLint та Prettier. ESLint[27] — це інструмент статичного аналізу коду для виявлення невідповідностей із заданим шаблоном. Він допомагає покращувати якість написаного коду, а також стиль програмування. Prettier[28] — це бібліотека, яка форматує код згідно описаних правил. Комбінація ESLint та Prettier дає можливість розробнику і писати акуратний код, і автоматично його форматувати.

3.5. Розробка REST API ендпоїнтів

API (Application Programming Interface) — це програмне забезпечення, яке допомагає різним програма комунікувати один з одним. Коли ми заходимо на веб сайт, наш веб браузер комунікує із сервером через HTTP API. HTTP (Hyper-Text Transfer Protocol) — це протокол прикладного рівня для передачі довільних даних по мережі.

REST (Representational State Transfer) — це архітектурний стиль для проектування HTTP API. Інформація, яку отримує споживач REST API визначається унікальним ідентифікатором — url (Uniform Resource Locator). Так наприклад url, який повертає десяту сторінку першої книжки може мати такий вигляд: www.web-site.com/api/books/1/page/10. Важливим елементом REST є використання HTTP методів: GET, POST, PUT, PATCH, DELETE. Виклик www.web-

site.com/api/books/1/page/10 з методом DELETE може означати видалення десятої сторінки першої книжки, а з методом PATCH оновлення. Згідно протоколу HTTP, HTTP API може віддавати інформацію у довільному форматі: html, JSON, jpeg. Дані у моєму веб застосунку будуть передаватися у форматі JSON (JavaScript Object Notation) від клієнту на сервер, і, від серверу на клієнт. З форматом JSON особливо зручно працювати при розробці на JavaScript.

3.5.1. Розробка serverless REST API ендпоїнтів

Будь-який serverless REST API ендпоїнт на платформі сервіс-провайдера AWS складається із двох компонентів: API Gateway ендпоїнту і AWS Lambda функції. [29]Коли клієнт робить HTTP запит він напряму викликає ендпоїнт створений у сервісі API Gateway. Потім сервіс API Gateway викликає синхронно AWS Lambda функцію. Після отримання результату обчислення функції, API Gateway формує HTTP відповідь клієнту.

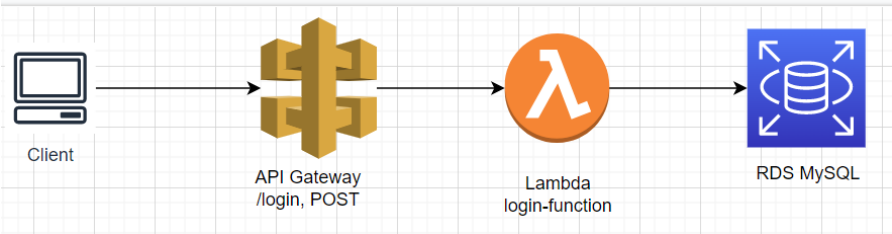

Рисунок 6 – діаграма запиту на вхід в систему

Процес створення API Gateway ендпоїнту і додавання до нього AWS Lambda функції займає дуже багато часу. Автоматизувати його допомагає технологія Serverless Framework. Станом[30] на 2020 рік, більше 80% компаній використовували Serverless Framework для розробки. Вона[31] дає можливість описувати всю потрібну інфраструктуру в спеціальному serverless.yaml файлі згідно спеціального шаблону визначеного фреймворком. Технологія автоматично перетворює інструкції описані у serverless.yaml файлі в елементи інфраструктури на платформі AWS такі як: Lambda функції, API Gateway ендпоїнти, ролі для ресурсів, підписки Lambda функцій на SQS черги чи SNS топіки.

Для того, щоб задеплоїти serverless REST API ендпоїнт з використанням Serverless Framework потрібно спочатку встановити утиліти serverless cli

(command line interface). [32]Після цього потрібно додати авторизаційні дані користувача платформи у конфігураційний файл утиліти. Це потрібно зробити, адже створення нових елементів буде відбуватися від імені користувача платформи.

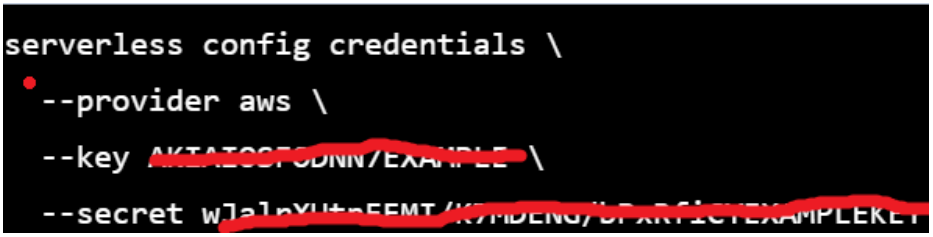

Рисунок 7 – приклад оновлення serverless config

Після того, як я додав авторизаційні дані мого користувача maks.kolesnyk (рисунок 7) мені потрібно було створити serverless.yaml файл, у якому потрібно описувати інфраструктуру.

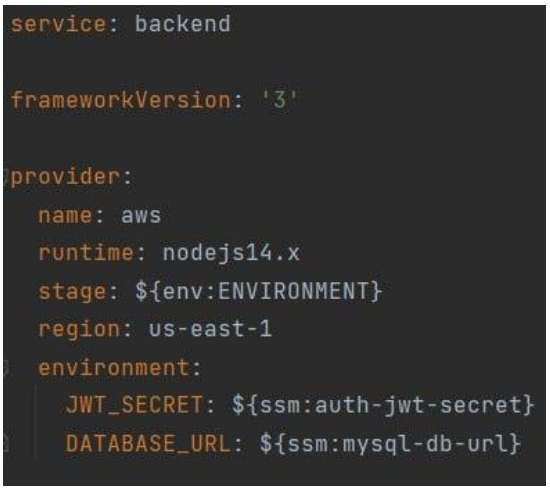

Рисунок 8 – уривок з serverless.yaml

У розділі provider (рисунок 8) я додав інформацію про середовище виконання мого коду — nodejs14.x, назву мого провайдера — aws, а також регіон на якому буде розгорнута інфраструктура. Вартий уваги також елемент environment. У ньому я вказав дві змінні: JWT_SECRET і DATABASE_URL. Вона тримають у собі посилання на записи в AWS Parameter Store. Під час збирання проекту Serverless Framework від імені авторизованого користувача зробить запит в Parameter Store, щоб дістати значення, які відповідають ключам: auth-jwt-secret і mysql-db-url. Ці значення будуть використовуватися для під'єднання до серверу бази даних і створення авторизаційного JWT API токену (token).

Рисунок 9 – уривок з serverless.yaml із функцією login

У розділі function файлу serverless.yaml я описав функцію аутентифікації login (рисунок 9). У визначення функції я додав: назву (name), розмір оперативної пам’яті, яка буде виділена для виконання функції (memorySize), максимальний час виконання (timeout), теги (tags), відносний шлях до папки з скомпільованим JavaScript кодом у якому міститься бізнес логіка (handler), а також події, (events) у яких буде задіяна функція. Перевагою опису інфраструктури у serverless.yaml файлі є декларативність написаного тексту. Навіть людині, яка ніколи не працювала із Serverless Framework легко здогадатися, що функція login буде задіяна HTTP ендпоїнтом із шляхом /auth/login та методом POST.

Щоб створити або оновити елементи інфраструктури Serverless Framework використовує сервіс AWS CloudFormation. [33]AWS CloudFormation дає можливість описувати елементи інфраструктури через код. Під час деплою коду на AWS, Serverless Framework зчитує інструкції описані в serverless.yaml і перетворює їх у CloudFormation темплейт (template) — документ у якому описані всі ресурси, які будуть створені або змінені на платформі сервіс-провайдера. На основі темплейту створюється CloudFormation stack, який містить у собі всі атомарні інструкції, які були здійсненні при створенні чи оновленні елементів інфраструктури.

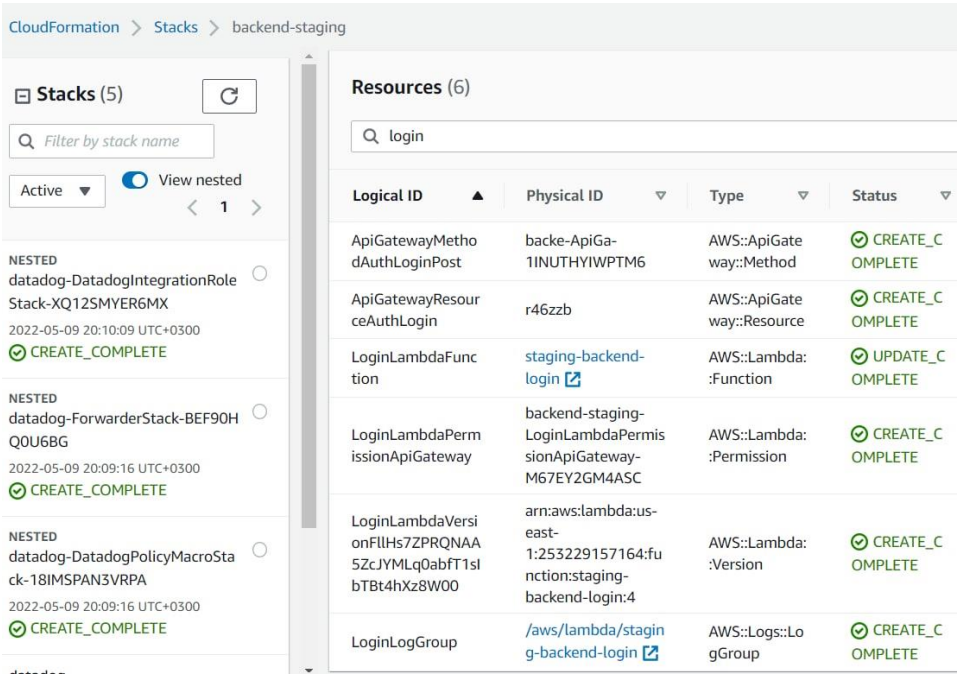

Рисунок 10 – CloudFormation стек

На рисунку 10 показні ресурси мого CloudFormation стеку, пов’язані із ендпоїнтом аутентифікації login. Тут чітко видно, що бути створені ендпоїнт API Gateway з ідентифікатором r46zzb, POST метод для нього, дозвіл на виклик Lambda функції, а також сама Lambda функція login разом із CloudWatch групою для логів.

3.5.2. Написання бізнес логіки в AWS Lambda функціях

У сфері розробки програмного забезпечення бізнес-логікою або domain логікою (логікою предметної області) вважається частина застосунку, у якій кодом описані реальні правила бізнесу, які має здійснити програма, щоб виконати запит клієнта. У моєму веб додатку бізнес логікою ендпоїнту автентифікації login є такий набір правил:

- Щоб увійти в систему користувач повинен надати email та пароль
- Email повинен бути валідний
- Пароль повинен містити мінімум 8 символів
- Користувач із наданим email повинен існувати в базі даних
- Наданий пароль користувача повинен відповідати паролю збереженому в базі даних

- В результаті успішного виконання всіх правил бізнес логіки сервер повинен сформувати унікальний токен авторизації для подальших дій користувача у системі і відправити його клієнту

3.5.2.1. Валідація та трансформація вхідних даних

Як я зазначав раніше, мій веб застосунок буде працювати із даними у форматі JSON. HTTP дає можливість надсилати дані у різних форматах. Саме тому першим кроком розробки ендпоїнту є валідація формату вхідних даних. З цим завданням мені допоможе впоратися бібліотека Middy.

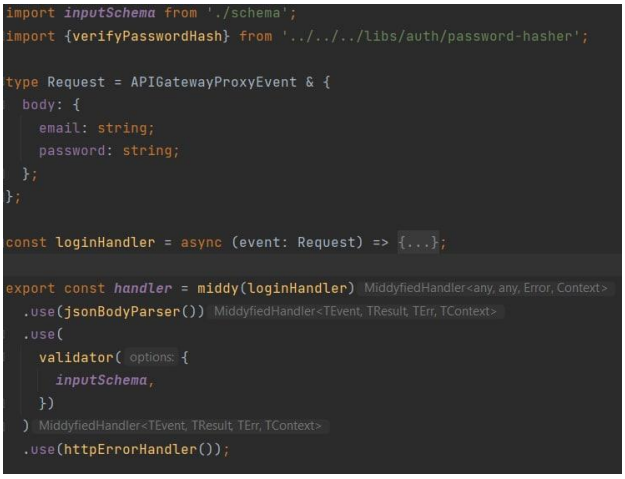

Рисунок 11 – функція login

Middy — це JavaScript бібліотека для створення функцій проміжної обробки (middleware functions), яка розроблена спеціально для обробки івентів (events), які API Gateway передає як параметр у Lambda функцію, в якій і описана бізнес логіка. На зображенні показаний стиснутий код Lambda функції який відповідає за автентифікацію користувача. У коді описані дві JavaScript змінні: handler та loginHandler (рисунок 11). Кожна змінна в собі тримає функції. При вхідному запиті API Gateway викликає змінну handler, яка є функцією middy. Функція middy приймає як аргумент змінну loginHandler. Функція middy створює і повертає нову функцію, у якій loginHandler буде виконуватися після викликів функцій проміжної обробки: jsonBobyParser та validator, і до httpErrorHandler. Таким чином middy робить змінну loginHandler також функцією проміжної обробки.

Функція проміжної обробки jsonBobyParser є частиною бібліотеки Middy. Вона і відповідає за валідацію значення, яке передається в event.body. Якщо значення має правильний JSON формат, функція перетворює його в JavaScript об’єкт і викликає наступну функцію проміжної обробки з мутованим значення в event.body. Якщо вхідне значення з event.body має невалідний JSON формат, jsonBobyParser кидає помилку зі статусом 400 із відповідним повідомленням про помилку читання JSON файлу.

Зловити помилку і правильно її відформатовати перед поверненням сервісу API Gateway під час виконання ланцюжку функцій проміжної обробки допомагає httpErrorHandler. Ця функція буквально ловить усі помилки, які можуть виникнути при роботі функції middy.

Для того щоб провалідувати вхідні дані, які передав користувач я використав validator. Функція validator також належить бібліотеці Middy, але збудована на основі іншої бібліотеки — ajv. Бібліотека[34] ajv вміє перевіряти вхідний JavaScript об’єкт за правилами визначеними у іншому JavaScript об’єкті — схемі (рисунок 12) по якій буде здійснена валідація першого об’єкта. На зображенні (рисунок) видно мою схему валідації вхідних параметрів для ендпоїнту аутентифікації. У ній вказано, що вхідний об’єкт має містити body, body має містити email та password, email мати правильний формат, а пароль має містити від 8 до 30 символів.

```
export default {
  type: 'object',
  properties: {
    body: {
      type: 'object',
      properties: {
        email: {type: 'email'},
        password: {type: 'string', minLength: 8, maxLength: 30},
      },
      required: ['email', 'password'],
    },
  },
};
```

Рисунок 12 – схема валідації вхідних параметрів функції login

3.5.2.2. Використання Prisma ORM для роботи з сервером бази даних

Після валідації вхідних даних надісланих користувачем потрібно перевірити, чи користувач з наданим email існує в базі даних. Щоб самостійно не писати код для з'єднання з базою даних і пошуку користувачів у таблиці я вирішив скористатися ORM.

ORM (Object-Relational Mapping) дають можливість працювати з реляційними базами даних через об'єкти класів. Для свого застосунку я обрав Prisma ORM.

```
const loginHandler = async (event: Request) => {
  const {email, password} = event.body;
  logger.info( message: `Received input`,  meta: {body: event.body});
  const user = await prisma.user.findUnique({where: {email: email}});

  if (!user) {
    throw createHttpError(401, 'Invalid credentials');
  }
```

Рисунок 13 – уривок з функції login

На рисунку 13 видно використання Prisma ORM для пошуку користувача у таблиці user за email адресою із вхідних даних.

Головною причиною вибору Prisma ORM серед інших відомих аналогів таких як: Sequelize чи Type ORM, був новий підхід до генерації ORM класів.

```
model User {
  id              String      @id
  email           String      @unique
  passwordHash    String
  firstName       String
  lastName        String
  role            UserRole
  createdAt       DateTime    @default(now())
  updatedAt       DateTime    @default(now())

  account         Account     @relation(fields: [accountId], references: [id])
  accountId       String

  createdTasks    Task[]      @relation("creator")
  tasksToExecute  Task[]      @relation("executive")
  comments        Comment[]
}
enum UserRole {
  owner
  worker
}
```

Рисунок 14 – модель User із файлу schema.prisma

Для того, щоб згенерувати ORM класи потрібно описати у спеціальному форматі файл schema.prisma (рисунок 13). На основі файлу будуть автоматично згенеровані ORM класи з інтерфейсом для здійснення практично будь-яких SQL запитів.

Prisma ORM в себе включає також інструмент для створення міграцій Prisma CLI (command line interface). Команда CLI «prisma migrate» на основі файлу schema.prisma генерує sql файли (рисунок 14), які потім запускає на сервері бази даних.

```
-- CreateTable
CREATE TABLE `User` (
    `id` VARCHAR(191) NOT NULL,
    `accountId` VARCHAR(191) NOT NULL,
    `email` VARCHAR(191) NOT NULL,
    `passwordHash` VARCHAR(191) NOT NULL,
    `firstName` VARCHAR(191) NOT NULL,
    `lastName` VARCHAR(191) NOT NULL,
    `role` ENUM('owner', 'worker') NOT NULL,
    `createdAt` VARCHAR(191) NOT NULL,

    UNIQUE INDEX `User_email_key`(`email`),
    PRIMARY KEY (`id`)
) DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

Рисунок 15 – частина файлу міграції

Ще одним приємним доповненням до Prisma ORM є Prisma studio (рисунок 15). Це автоматично згенерований веб сайт, який дає можливість оглянути поточний стан таблиць на сервері бази даних і навіть зробити елементарні SQL операції вибірки, створення, видалення і оновлення записів.

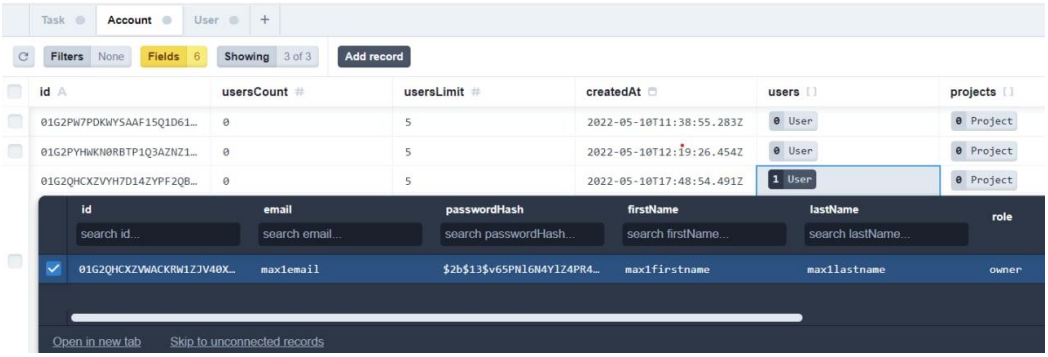

Рисунок 16 – Prisma studio в роботі

3.5.2.3. Тестування бізнес логіки

Обов’язковим елементом будь-якого програмного забезпечення є баги (bugs). Баги це непередбачувані проблеми, які виникають під час виконання програми. Відомий філософ 17 століття Рене Декарт колись придумав гарно фразу: «Cogito ergo sum» (з лат. — «Я мислю, отже існую»). У світі розробки можна скласти схожу фразу, але про помилки: «Я пишу код, отже породжую

баги». У книжці Steve McConnell “Code Complete” описав вартість фіксу багів (усунення помилки) на різних етапах розробки програмного забезпечення. Згідно його розрахунків, ціна виправлення помилки на стадії production (стадія на якій клієнти користуються кодом) у 15 разів вища ніж на стадії розробки. Саме тому будь-яке програмне забезпечення варто тестувати. Існує два основних види тестувань: інтеграційні і юніт тести. Під час написання коду програми можна і потрібно використовувати юніт тестування. Ідея юніт тестування полягає перевірці окремих частин коду програми. Тестуючи код в процесі його написання програміст не тільки по факту перевіряє поведінку програми, а й автоматично покращує якість коду. Найпопулярнішим[35] інструментами для юніт тестування JavaScript коду є фреймворк Jest. Його головні перевага полягають у простоті використання і швидкості інтеграції у проекти.

На рисунку 16 показаний мій приклад тестування функції handler. Я викликаю функцію з валідними параметрами, і очікую, що функція prisma.user.findUnique буде викликана з email адресою з event.body.

```
it( name: 'should call findUnique with input email from input params', fn: async () => {
  const event: any = {
    body: {email: 'maxym@ukma.edu.ua', password: '12345678'},
  };

  await handler(event, {} as any, {} as any);

  expect(prisma.user.findUnique).toHaveBeenCalledWith( params: {where: {email: 'maxym@ukma.edu.ua'}});
```

Рисунок 17 – unit тест виклику функції

3.5.2.4 Авторизація

Більшість ендпоїнтів у моєму додатку повинні мати повинні мати авторизацію. Виключеннями є логін, та створення нового акаунту. Для роботи з проектами, користувачами та завданнями запити користувачів повинні проходити перевірку. Для цього я скористався технологією JWT (JSON web token) а також API Gateway Lambda authorizer.

JWT[36] — це JSON об’єкт, який складається із трьох частин: заголовка (header), даних (payload) і зашифрованого підпису. Зашифрований підпис

формується і валідується на основі секретного слова, про яке може знати тільки сервер під час створення а валідації підпису. Кожен раз коли користувач аутентифікується в системі він отримує token. Цей token він використовує при всіх подальших запитах до ресурсів, які вимагають token для авторизації.

API Gateway Lambda authorizer — це спеціальна фіча сервісу API Gateway, яка дозволяє до існуючого ендпоїнту додати функцію попередньої обробки у вигляді AWS Lambda функції, у якій і буде міститися логіка валідації даних користувача. Я реалізував власну функцію авторизації і потім додав її через serverless.yaml до всіх захищених ендпоїнтів.

На рисунку 17 показаний опис функції createUser у serverless.yaml файлі. Поле authorizer містить в собі назву (name) функції авторизації authorize (рисунок 18), звідки буде братися токен авторизації (identitySouce), а також тип авторизації (type).

```yaml
createUser:
  handler: dist/src/users/create-user/handler.handler
  name: ${self:provider.stage}-${self:service}-createUser
  tags:
    name: createUser
  timeout: 15
  memorySize: 128
  events:
    - http:
        path: /users
        method: post
        authorizer:
          name: authorize
          identitySource: method.request.header.Authorization
          type: token
```

Рисунок 18 – функція createUser

```yaml
authorize:
  handler: dist/src/auth/authorize/handler.handler
  name: ${self:provider.stage}-${self:service}-authorize
  tags:
    name: authorize
  timeout: 10
  memorySize: 128
```

Рисунок 19 – функція authorize

3.5.2.5. Логування Lambda функцій

Одним із найбільших недоліків роботи із serverless застосунками вважається складність аналізувати виконання коду. Для вирішення цієї проблеми

можна логувати різні етапи роботи кожного ендпоїнту. В першу чергу я додав виклики функції стандартного виводу у всіх місцях, де здійснюються асинхронні операції (рисунок 19). Додатково я зробив логування вхідних параметрів, які передає користувач, а також базової інформації про авторизованого користувача, який здійснює запит: accountId, userId та role (рисунок 19). Для логування я використав бібліотеку Winston[37].

```
const createProjectHandler = async (event: Request) => {
  const {accountId, userId, role} = event.requestContext.authorizer;
  setDefaultLoggerMeta( meta: {accountId, userId, role});

  const {name} = event.body;
  logger.info( message: `Received input`,  meta: {body: event.body});

  await validate(accountId, name);

  const project = await prisma.project.create({
    data: {
      id: ulid(),
      account: {connect: {id: accountId}},
      name,
    },
  });
  logger.info( message: `Created project`);
```

Рисунок 20 – частина функції createProjectHandler

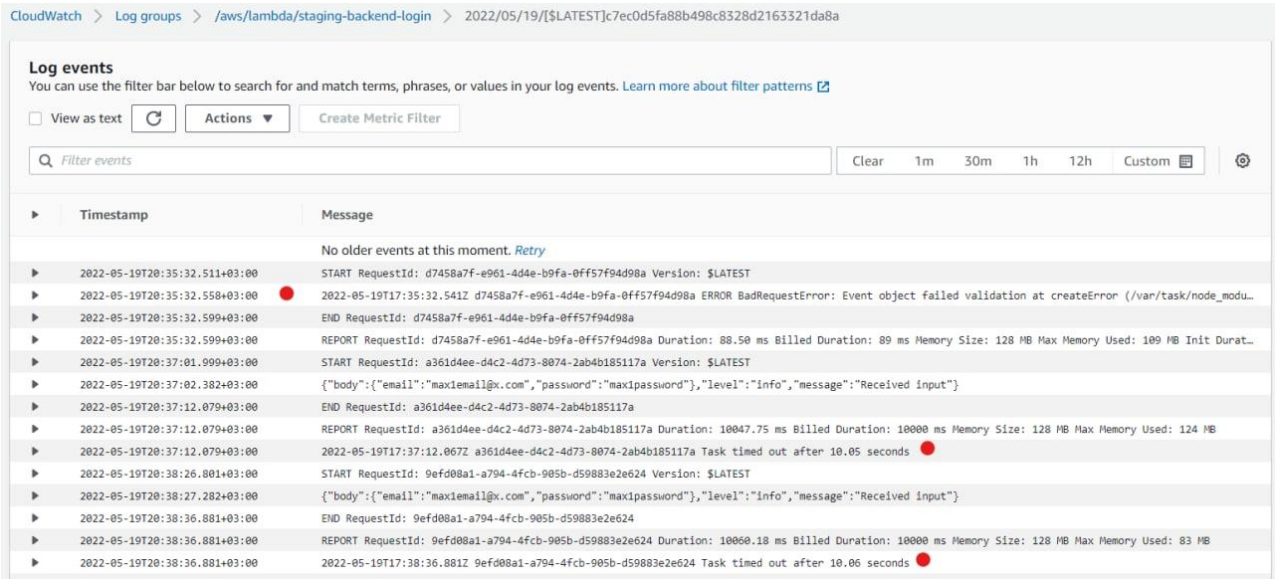

Рисунок 21– логи з функції login у сервісі CloudWatch

Serverless Framework автоматично створює log групу в сервісі AWS CloudWatch для кожної функції, а також надає функціям права робити записи в log групах. Після цих операцій, увесь стандартний вивід із AWS Lambda

функцій починає зберігатися у відповідних log групах. Це дає можливість аналізувати функції у сервісі AWS CloudWatch.

На рисунку 20, я відмітив червоними точками логи з помилками у моїй Lamba функції login. CloudWatch і справді дуже допомагає аналізувати результати виконання Lambda функцій. [38]Ще однієї перевагою збереження логів на CloudWatch є можливість сервісу нескінченно розширюватися. Логи у сервісі також можуть зберігатися впродовж необмеженого періоду часу.

Для мого застосунку функцій CloudWatch з головою вистачало для моніторингу роботи системи, але мені закортіло вивести аналіз логів на вищий рівень. Головною проблемою сервісу CloudWatch є неможливість аналізувати декілька груп логів одночасно. Саме тому я інтегрував AWS CloudWatch із стороннім сервісом для моніторингу і аналізу логів — Datadog. Datadog дає можливість створювати власні дашборди (dashboards) (рисунок 22) пов’язані абсолютно із будь-яким сервісом чи ресурсом на платформі AWS. Він також дозволяє зручно фільтрувати і переглядати логи (рисунок 21). Можливості для аналізу і фільтрації даних у сервісі Datadog значно перевищують можливості AWS CloudWatch. У[39] більшості випадках Datadog використовують виключно компанії із великомасштабними застосунками. У межах курсової роботи я додав його з інтересу і для досвіду.

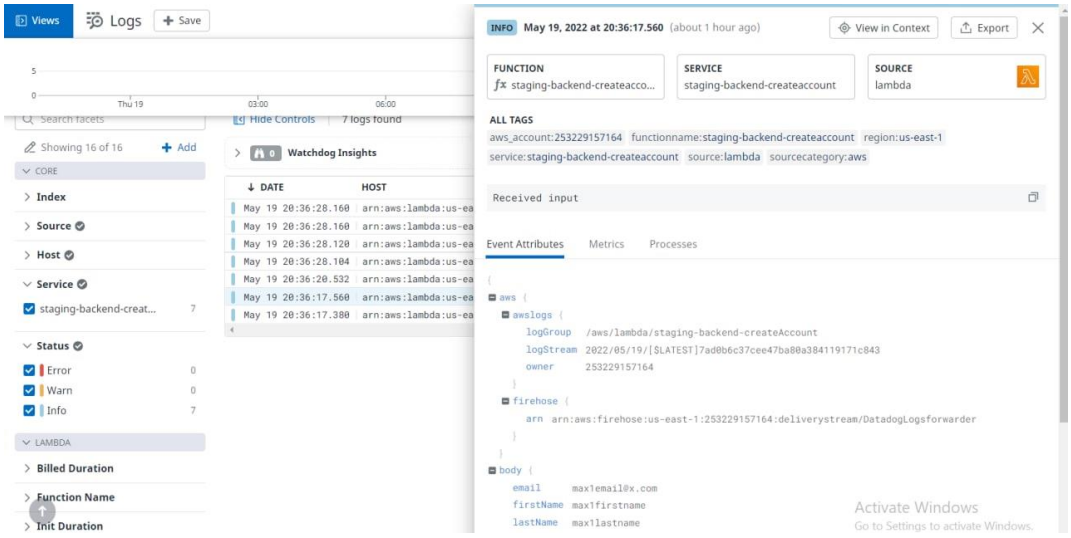

Рисунок 22 – логи з функції login у сервісі Datadog

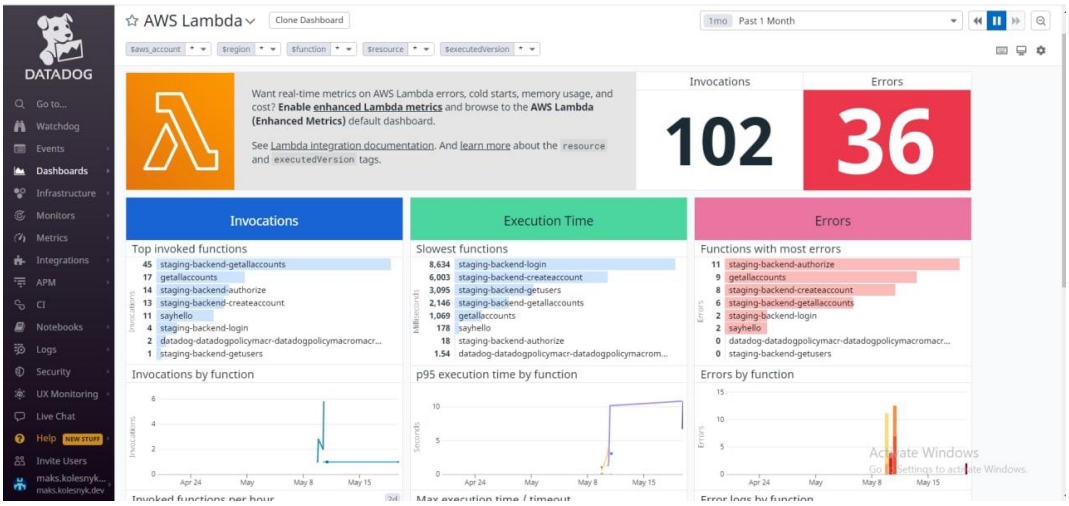

Рисунок 23 – дашборд сервісу AWS Lambda на сайті Datadog

Мене дуже вразив сервіс Datadog. Незважаючи на те, що мій веб застосунок дуже маленький, я дізнався багато цікавої інформації про роботу своїх Lambda функцій із автоматично-сформованого дашборду для всього сервісу Lambda на моєму AWS акаунті. Також читання логів стало набагато зручніше у порівнянні з аналізом груп логів в AWS CloudWatch.

3.6. Розробка документації проекту

Для свого serverless веб додатку я створив документацію згідно OAS (OpenAPI Specification) третьої версії. OAS[40] — це найпопулярніший фреймворк для опису REST API. Вся документація — це один суцільний файл у JSON або YAML (Yet Another Markup Language) форматі, усі елементи якого описані згідно OpenAPI специфікації.

```yaml
openapi: 3.0.0
info:
  title: White-tea API documentation
  version: 0.0.1
servers:
  - url: https://9b359ezcwb.execute-api.us-east-1.amazonaws.com
    description: Staging server

components:
  securitySchemes:
    auth_token:
      description: JWT token required for consuming API. Token valid for 100 days
      type: apiKey
      in: header
      name: Authorization
```

Рисунок 24 – уривок з файлу openapi.yaml

В openapi.yaml (рисунок 23) файлі я описав всі REST API ендпоїнти свого веб застосунку. Щоб спростити перегляд документації я вирішив скористатися бібліотекою Redocly. Redocly[41] здатна генерувати красивий html (рисунок 25) файл з вхідного YAML файлу із документацією описаною згідно OpenAPI специфікації.

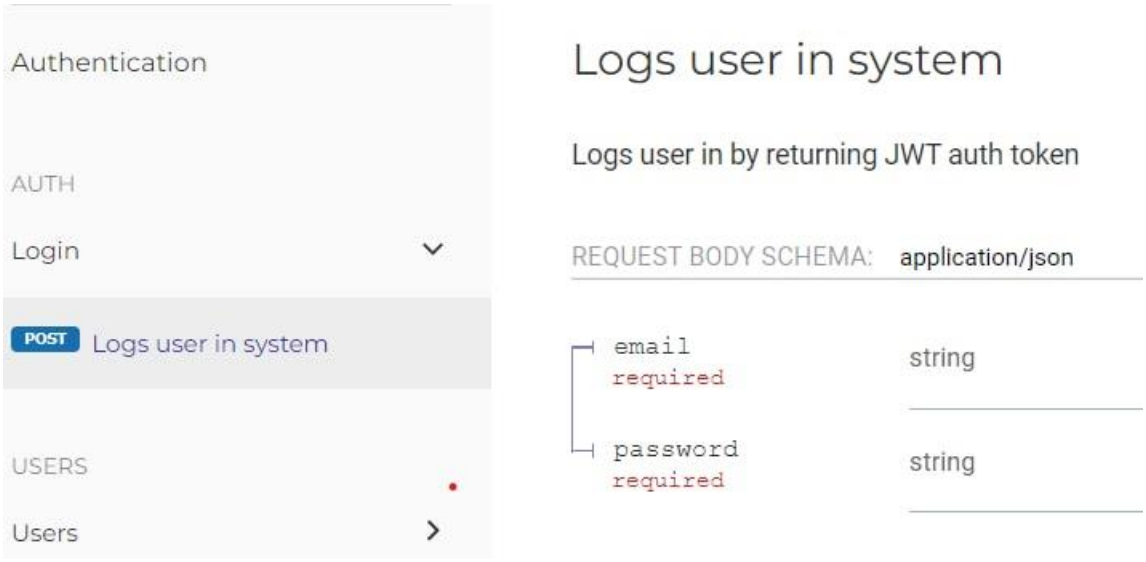

Рисунок 25 – згенерований html з документацією

Для згенерованого html файлу я створив окремий бакет (bucket) в сервісі S3 під назвою «backend-api-docs». Згенерований html файл я помістив у новостворений бакет, і захостив файл публічно, як статичний веб-сайт. Я виніс документацію у окремий GitHub проект для майбутнього налаштування автоматичного процесу деплою.

3.7. Автоматизація деплою проектів

До цього моменту, для деплою веб застосунку я використовував Serverless Framework, а для сайту документації власноруч завантажував оновлений файл в AWS бакет. Щоб автоматизувати цей процес я інтегрував сервіс CircleCI в мої репозиторії на GitHub. CircleCI[42] моніторить GitHub і запускає білди (builds) для кожного нового коміту (commit). Для налаштування CircleCI в своїх репозиторіях я створив файл .circleci/config.yml у якому, згідно специфікації описав інструкції, які мають бути виконані для деплою проектів на платформу AWS. На сайті CircleCI у налаштуваннях проектів я вказав авторизаційні дані

мого користувача maks.kolesnyk. CircleCI від його імені буде доступатися до ресурсів AWS.

```
workflows:
  test_deploy:
    jobs:
      - test
      - deploy:
          requires:
            - test
          filters:
            branches:
              only:
                - develop
```

Рисунок 26 – workflow

На рисунку 26 мій workflow «test_deploy», який складається із двох джоб (jobs): «test» і «deploy». Джоби відпрацьовують по черзі, коли з’являються нові коміти на гілці (branch) «develop» у моєму репозиторії на GitHub.

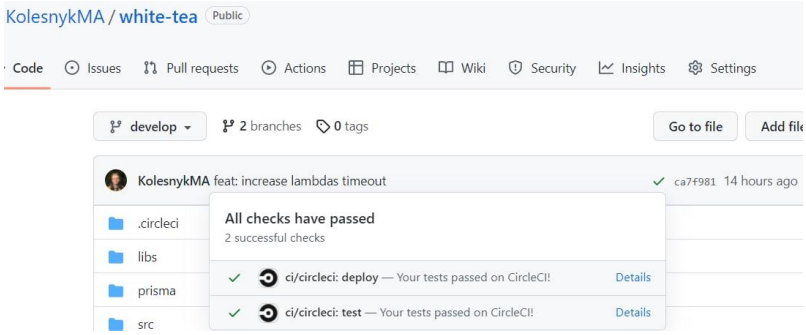

Рисунок 27 – робота workflow на гілці develop

Для проекту з документацією я налаштував аналогічний workflow. Автоматизація деплою принесла мені багато задоволення.

4. Розрахунок вартості утримання веб застосунку

Після реалізації мінімального робочого проекту я вирішив розрахувати вартість утримання застосунку у production середовищі. Я здійснив розрахунок для 100 акаунтів, на кожному з яких працює по 10 користувачів. Я очікую, що користувачі щодня роблять 100 REST API. В місяць моя система буде отримувати 3100000 запитів. Веб застосунок буде задеплоїний в одному регіоні. Для розрахунків я використав[43] офіційний калькулятор із платформи AWS.

- API Gateway. За 3100000 HTTP API запитів вартість сервісу складатиме 10.85 USD.
- AWS Lambda. Якщо функції будуть виконуватися з середнім часом 3 секунди і займатимуть до 128 мегабайтів оперативної пам’яті під час виконання, вартість складатиме 13.13 USD.
- AWS S3. На кожному акаунті буде створюватися приблизно 200 завдань в місяць. Кожне завдання містить в собі текстову частину, документи та картинки. До завдань можна додавати коментарі, які також можуть містити вкладення. Із запасом можна взяти 100 гігабайтів на кожен акаунт в місяць, а також 2000000 запитів до S3 на читання і запис файлів. Загальна вартість сервісу складатиме 13.37 USD.
- AWS CloudWatch. Середній розмір одного повідомлення 128 байтів. В середньому на запит потрібно 7 логів. Цt виходить 21000000 логів на місяць з 3100000 запитів, які займають 2.68 гігабайтів. Загальна вартість складатиме 3.78 USD.
- AWS RDS. Для обслуговування 1000 користувачів можна почати з одного інстансу бази даних db.t4g.micro. Він має 2 CPU та 1 гігабайт оперативної пам’яті. Його вартість на місяць складає 23.36 USD. За кожні 30 гігабайтів на SSD потрібно буде також сплачувати 6.90 USD в місяць.

В результаті підрахунків, загальна приблизна вартість обслуговування 1000 користувачів мого веб застосунку складатиме приблизно 70 USD в місяць у production середовищі. Звісно, це лише приблизна вартість утримання інфраструктури на AWS. Проте, якщо врахувати ту кількість плюсів, які дають

використані сервіси, навіть 700 USD у місяць були б копійками у порівнянні із зарплатами програмістів, котрі власноруч мали б проектувати, розширяти та утримувати аналогічну інфраструктуру.

Висновок

Під час написання курсової роботи я розібрався із перевагами та недоліками serverless. Проаналізував ситуації, у яких даний підхід варто використовувати для розробки програмного забезпечення. На основі набутих знань я реалізував власний serverless веб застосунок для менеджменту процесу розробки цифрового продукту. Для створення застосунку я використав комбінацію найактуальніших технологій доступних на сьогодні і вибудував свій власний підхід до розробки serverless додатків. Я також добре познайомився із принципами роботи платформи Amazon Web Services. Додатково до реалізації застосунку я також розрахував приблизну вартість його утримання у production середовищі. Цим я на власному досвіді переконався у економності підходу.

Безперечно, serverless підхід до реалізації застосунків є дуже потужним, якщо розуміти, як і коли його застосовувати. Я щиро задоволений набутим досвідом та отриманими знаннями від написання цієї роботи.

Список використаних джерел

[1] An Introduction to Dew Computing: Definition, Concept and Implications // Ray, Partha Pratim (2018) - https://ieeexplore.ieee.org/document/8114187
[2] Top 5 Serverless Platforms in 2022 // Artem Arkhipov (01.01.2021) https://www.techmagic.co/blog/top-5-serverless-platforms-in-2020/
[3] Serverless architecture market outlook // https://www.alliedmarketresearch.com/serverless-architecture-market#:~:text=The%20global%20serverless%20architecture%20market,27.8%25%20from%202018%20to%202025.
[4] An Introduction to Dew Computing: Definition, Concept and Implications // Ray, Partha Pratim (2018) - https://ieeexplore.ieee.org/document/8114187
[5] PaaS Primer: What is platform as a service and why does it matter? // Brandon Butler (2013) - https://www.networkworld.com/article/2163430/paas-primer--what-is-platform-as-a-service-and-why-does-it-matter-.html
[6] Top 5 Serverless Platforms in 2022 // Artem Arkhipov (01.01.2021) https://www.techmagic.co/blog/top-5-serverless-platforms-in-2020/
[7] AWS Lambda https://aws.amazon.com/lambda/?nc1=h_ls
[8] AWS Lambda Documentation, Lambda function scaling https://docs.aws.amazon.com/lambda/latest/dg/invocation-scaling.html
[9] AWS DynamoDB https://aws.amazon.com/dynamodb/?nc1=h_ls
[10] AWS Aurora https://aws.amazon.com/rds/aurora/?nc1=h_ls
[11] Black Friday and Cyber Monday outages and other issues angered customers on these 48 retail sites https://www.fastcompany.com/90580744/black-friday-and-cyber-monday-outages-angered-customers-on-these-48-retail-sites
[12] AWS Pricing https://aws.amazon.com/pricing/?nc1=h_ls
[13] Cloud computing with AWS https://aws.amazon.com/what-is-aws/
[14] Operating Lambda: Performance optimization – Part 1 https://aws.amazon.com/ru/blogs/compute/operating-lambda-performance-optimization-part-1/#:~:text=According%20to%20an%20analysis%20of,ms%20to%20over%201%20second.
[15] The state of serverless https://www.datadoghq.com/state-of-serverless-2020/
[16] Jira software https://www.atlassian.com/ru/software/jira
[17] AWS RDS https://aws.amazon.com/rds/?nc1=h_ls
[18] AWS Systems Manager Parameter Store https://docs.aws.amazon.com/systems-manager/latest/userguide/systems-manager-parameter-store.html
[19] Diagrams https://app.diagrams.net/
[20] Serverless most popular programming languages // Taavi Rahemaji (2020) https://dashbird.io/blog/serverless-most-popular-programming-languages/#:~:text=Just%20like%20Python%20and%20Nodejs,serverless%20monitoring%20tool%20out%20there.
[21] State of the serverless https://www.datadoghq.com/state-of-serverless/
[22] Node.js https://uk.wikipedia.org/wiki/Node.js

[23] So long, and thanks for all the packages! // Nassri, Ahmad (14 April 2020) https://en.wikipedia.org/wiki/Npm_(software)#:~:text=Over%201.3%20million%20packages%20are%20available%20in%20the%20main%20npm%20registry.
[24] JavaScript https://en.wikipedia.org/wiki/JavaScript
[25] TypeScript https://en.wikipedia.org/wiki/TypeScript
[26] GitHub https://en.wikipedia.org/wiki/GitHub
[27] Getting Started with ESLint https://eslint.org/docs/user-guide/getting-started
[28] Prettier.io https://prettier.io/
[29] Using AWS Lambda with Amazon API Gateway https://docs.aws.amazon.com/lambda/latest/dg/services-apigateway.html
[30] State of serverless https://www.datadoghq.com/state-of-serverless/
[31] Serverless Framework Documentation https://www.serverless.com/framework/docs
[32] Serverless Framework Documentation, AWS credentials https://www.serverless.com/framework/docs/providers/aws/guide/credentials
[33] AWS Cloudformation https://aws.amazon.com/cloudformation/?nc1=h_ls
[34] Ajv https://ajv.js.org/
[35] What Is the Best Unit Testing Framework for JavaScript? // By Testim (2021) https://www.testim.io/blog/best-unit-testing-framework-for-javascript/#:~:text=Playwright-,Jest,VueJS%2C%20NodeJS%2C%20and%20others.
[36] JSON Web Tokenhttps://ru.wikipedia.org/wiki/JSON_Web_Token#%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0
[37] Winston https://github.com/winstonjs/winston
[38] Amazon CloudWatch FAQs https://aws.amazon.com/ru/cloudwatch/faqs/#:~:text=You%20can%20store%20your%20log,Log%20Group%20at%20any%20time.
[39] What is Datadog? https://docs.microsoft.com/en-us/azure/partner-solutions/datadog/overview
[40] What is OpenAPI? https://www.openapis.org/faq
[41] Redocly https://redocly.com/
[42] CircleCI https://en.wikipedia.org/wiki/CircleCI
[43] AWS Pricing Calculator https://calculator.aws/#/createCalculator