НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Інститут прикладного системного аналізу

Кафедра системного проектування

«На правах рукопису»
УДК ____004.514_____

До захисту допущено:
Завідувач кафедри
____А.І. Петренко_____
«___»_____20__ р.

Магістерська дисертація

на здобуття ступеня магістра з комп'ютерних наук

за освітньо-професійною програмою «Інтелектуальні сервіс-орієнтовані розподілені обчислювання»

зі спеціальності 122 «Комп'ютерні науки»

на тему: «FaaS як шаблон архітектури додатків»

Виконав:
студент VI курсу, групи ДА-з91мп
Плотніков Сергій Олександрович _____

Керівник:
к. т. н., доцент
Гіоргізова-Гай Вікторія Шавлівна _____

Консультант з розробки стартап-проекту:
к. т. н., доцент
Гіоргізова-Гай Вікторія Шавлівна _____

Рецензент:
в.о. зав. каф. ММСА ІПСА,
к.т.н., доцент Тимощук Оксана Леонідівна _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент (-ка) _____

Київ – 2020 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Інститут прикладного системного аналізу

Кафедра системного проектування

Рівень вищої освіти – другий (магістерський)

Спеціальність – 122 «Комп'ютерні науки»

Освітньо-професійна програма «Інтелектуальні сервіс-орієнтовані розподілені обчислювання»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ А.І.Петренко

«_07__»__Липня_____2020__ р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Плотніков Сергій Олександрович

1. Тема дисертації «**FaaS як шаблон архітектури додатків**», науковий керівник дисертації к.т.н., доцент Гіоргізова-Гай В.Ш., затверджені наказом по університету від «_2__»__листопада_____ 2020_ р. № 3182-с_____

2. Термін подання студентом дисертації: 25 грудня

3. Об'єкт дослідження: шаблони архітектури FaaS додатків

4. Вихідні дані: Docker, хмарна система AWS, середовище розробки Jetbrains PHPStrom 2020.3, Node.js, фреймворк Angular.js

5. Перелік завдань, які потрібно розробити

1. Дослідити поняття Serverless і FaaS, їх характеристики, переваги і недоліки.
2. Ознайомитися з найпопулярнішими FaaS провайдерами, виконати їх порівняння і обрати один із них для подальшого дослідження.
3. Провести аналіз сучасних патернів FaaS дадатків. Проаналізувати застосуваня патернів у різних типах додатків.
4. Побудувати два додатки за допомогою FaaS архітектури і класичної мікросервісної архітектури.

5. Зробити висновки про переваги і недоліки використання даних архітектур для побудови додатків

6. Орієнтовний перелік графічного (ілюстративного) матеріалу: презентація на тему «FaaS як шаблон архітектури додатків»

7. Орієнтовний перелік публікацій

8. Консультанти розділів дисертації

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Реалізація стартап-проекту	Доцент, к.т.н. Гіоргізова-Гай В.Ш.		

9. Дата видачі завдання 7 липня 2020

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Отримання завдання	7 липня 2020	
2	Збір і ознайомлення з літературою	11 вересня 2020	
3	Дослідження предметної області	26 вересня 2020	
4	Аналіз існуючих шаблонів безсерверних додатків	10 жовтня 2020	
5	Реалізація додатку на основі безсерверної архітектури і мікросервісної, розміщеної у Docker контейнерах	1 листопада 2020	
6	Оцінка результатів і створення порівняльної характеристики результатів роботи додатку	18 листопада 2020	
7	Оформлення дипломної роботи	11 грудня 2020	

Студент Плотінков С.О.

Науковий керівник Гіоргізова-Гай В.Ш.

РЕФЕРАТ

на магістерську дисертацію

виконану на тему: «FaaS як шаблон архітектури додатків »

студентом: Плотніковим Сергієм Олександровичем

Робота складається зі вступу та п’яти розділів. Загальний обсяг роботи: 127 аркушів основного тексту, 60 ілюстрації, 24 таблиць. При підготовці використовувалась література з 26 джерел.

Актуальність. З появою хмарних провайдерів поступово відбувається перехід від монолітної архітектури додатків до класичної мікросервісної, що зумовлено потребами автоматичного масштабуванням і зручністю розробки. З появою FaaS сервісу від AWS у 2014році безсерверні обчислення вийшли в тренд у побудові додатків у хмарі. Завдяки таким можливостям, як повна відсутність в необхідності конфігурації інфраструктурного шару додатку, авто-масштабованість і оплата на вимогу роблять безсерверні обчислення привабливими як архітектура хмарних додатків.

Тому дослідження архітектурних шаблонів serverless додатків є актуальною темою, оскільки вибір правильної архітектури додатку є найголовнішою задачею при розробці.

Мета і завдання дослідження. Метою даної магістерської роботи є дослідження архітектурних патернів побудови FaaS додатків. А саме:

- аналіз і порівняння шаблонів побудови додатків і засобів їх реалізації від популярних FaaS провайдерів та вибір послуг одного з провайдерів для подальшого дослідження (було вибрано AWS);
- реалізація одного з найбільш розповсюджених безсерверних шаблонів від вибраного провайдера (було взято патерн FaaS “Simple Web Service”) і порівняння її за виділеними критеріями з аналогічною реалізацією у вигляді класичної мікросервісної серверної архітектури.

- узагальнення результатів досліджень і формулювання загальних рекомендацій щодо вибору послуг певного провайдера і архітектури побудови додатку в залежності від вимог до додатку.

Об'єкт дослідження — архітектурні шаблони, які використовуються для побудови serverless додатків.

Предмет дослідження — дві реалізації типового додатка у AWS: на FaaS архітектурі і на класичній мікросервісній архітектурі, з подальшим порівнянням їх перевагах, недоліків і визначенням умов найкращого застосування кожного рішення.

Для розробки додатків використовувались сервіси від хмарного провайдера AWS. FaaS архітектура була побудована на сервісіх AWS Lambda і API Gateway. Класична мікросервісна архітектура на Docker контейнерах була розміщена у сервісі Elastic Container Service з використанням Application Load Balancer.

Методи дослідження. Для вирішення поставлених задач в роботі було застосовано методи аналізу та синтезу, порівняння та узагальнення отриманих результатів.

Наукова новизна отриманих результатів роботи полягає у наступному:

1 Проведено порівняльний аналіз за виділеними критеріями відомих FaaS сервісів від хмарних провайдерів: AWS Lambda, Microsoft Azure Functions і Google Cloud Functions.
2 Запропоновано методи прискорення роботи додатків, розроблених на FaaS архітектурі у AWS.
3 Проведено порівняння реалізацій додатка на серверній і безсерверній архітектурі у AWS за виділеними критеріями. Визначені недоліки, переваги і умови найкращого застосування кожного варіанту.

Практична цінність. Отримані результати порівняння хмарних провайдерів можуть бути використані для вибору провайдера під час розробки додатку, в залежності від необхідних вимог. Також проаналізовані архітектурні шаблони можуть бути використані на початку моделювання системи в

залежності від типу додатку. Результати порівняння розроблених реалізацій додатка можуть бути використані в якості рекомендацій при виборі архітектури для реалізації додатка: переваги та недоліки безсерверної і класичної мікросервісної архітектури, для яких типів додатків краще підходить певна архітектура.

Ключові слова: FaaS, AWS Lambda, Elastic Container Service, мікросервісна архітектура, serverless.

ABSTRACT

for master's thesis of Serhii Oleksandrovych Plotnikov

on: «Faas as a pattern for architecture of applications»

The master's thesis consists of introduction and four sections. Total volume of work: 127 sheets of the main text, 60 illustrations, 24 tables. Literature from 26 different sourced was used in the preparation.

Relevance of the topic. With the advent of cloud providers, there is a gradual transition from a monolithic application architecture to a classic microservice, due to the need for automatic scaling and ease of development. With the advent of the FaaS service from AWS in 2014, serverless computing has become a trend in building applications in the cloud. With features such as a complete lack of application infrastructure layer configuration, auto-scalability, and on-demand payment make server-side computing attractive as a cloud application architecture.

Therefore, the study of architectural templates of serverless applications is a topical issue, because choosing the right application architecture is the most important task in development.

The purpose and objectives of the study. The purpose of this master's thesis is to study the architectural patterns of FaaS applications. Namely:

• analysis and comparison of application building templates and means of their implementation from popular FaaS providers and selection of services of one of the providers for further research (AWS was selected);

• implementation of one of the most common serverless templates from the selected provider (the FaaS "Simple Web Service" pattern was taken) and its comparison according to the selected criteria with a similar implementation in the form of a classic microservice server architecture.

• generalization of research results and formulation of general recommendations for the choice of services of a particular provider and the architecture of the application depending on the requirements for the application.

The object of research is architectural templates that are used to build serverless applications.

The subject of the research is two implementations of a typical application in AWS: on FaaS architecture and on classical microservice architecture, with further comparison of their advantages, disadvantages and determination of conditions of the best application of each solution.

Services from the cloud provider AWS were used to develop applications. The FaaS architecture was built on the AWS Lambda and Gateway API services. The classic microservice architecture on Docker containers was hosted in the Elastic Container Service using Application Load Balancer.

Research methods. Methods of analysis and synthesis, comparison and generalization of the obtained results were used to solve the set tasks.

The scientific novelty of the obtained results of work is the following:

1. A comparative analysis was performed according to the selected criteria of well-known FaaS services from cloud providers: AWS Lambda, Microsoft Azure Functions and Google Cloud Functions.

2. Methods of accelerating the work of applications developed on the FaaS architecture in AWS are proposed.

3. The comparison of implementations of the application on server and serverless architecture in AWS according to the selected criteria is carried out. The disadvantages, advantages and conditions of the best application of each option are identified.

Practical value. The results of the comparison of cloud providers can be used to select a provider during application development, depending on the required requirements. Also analyzed architectural templates can be used at the beginning of system modeling depending on the type of application. The results of comparing the developed implementations of the application can be used as recommendations when choosing an architecture for the implementation of the application: advantages and disadvantages of serverless and classical microservice architecture, for which types of applications a particular architecture is better suited.

Keywords: FaaS, AWS Lambda, Elastic Container Service, microservice architecture, serverless.

ЗМІСТ

ВСТУП

П'ятнадцять років тому більшість компаній несли повну відповідальність за свої серверні додатки, починаючи від спеціально розроблених додатків до конфігурації мережевих комутаторів та фаєрволів, від управління високо-доступними серверами баз даних до розгляду вимог потужності їх дата-центру. Але з’явились хмарні рішення. Хмарні рішення змінили наше мислення про додатки. Розробник може більше не думати про мережеве обладнання або складання щорічного капітального плану, які сервери компанії потрібно купити. Натомість компанії можуть орендувати віртуальні машини погодинно, передавати управління базами даних сторонньому провайдеру. Але все ще залишається одна проблема: розробник все ще мислить у поняттях серверів - дискретні компоненти, які потрібно розміщувати, забезпечувати залежностями, забезпечувати налаштовування, розгортання, ініціалізацію, моніторинг, управління, передислокацію та повторну ініціалізацію. Проблема в тому, що більшу частину часу розробника не цікавлять ці дії, єдине, що його турбує — щоб логіка додатку правильно виконувалась і дані були консистентні та безпечні. Чи можуть в цьому допомоги хмарні технології? Так, з’явились Serverless рішення. Serverless є відносно новим методом хмарних обчислень з тих пір, як Amazon вперше представив їх у 2014 році. Amazon, Microsoft, Google – усі ці компанії бачили значні переваги, які може принести безсерверна архітектура, і саме тому вони приєдналися до світу хмарних обчислень як постачальники хмарних послуг (Cloud Service Provider — CSP).

Використання Serverless архітектури значно допомагає розробникам більше зосередитися на своєму основному продукті. Таким чином, увага розробника зосереджується виключно на робочому процесі, розподіленій логіці та зовнішньому управлінню сховищами даних. Такі служби, як AWS Lambda, Google Cloud Functions та Microsoft Azure Functions, будуть займатися

фізичним обладнанням, операційною системою віртуальних машин, а також веб-сервером, тоді як розробникам потрібно буде турбуватися лише про одне - про свій код.

У даній роботі ми розглянемо, що насправді означає Serverless і FaaS, представимо їх переваги і обмеження та розглянемо еталонні рішення побудови FaaS архітектури від провайдерів.

Метою даної магістерської роботи є дослідження архітектурних шаблонів побудови FaaS додатків. На основі досліджень у практичній частині роботи, проведених на двох реалізаціях у AWS прототипу типового додатка (з використанням FaaS патерну “Simple Web Service” та аналогічного мікросервісного патерну), проведено аналіз_переваг і недоліків використання даних архітектур для побудови додатків.

1 ПОНЯТТЯ SERVERLESS І FAAS

1.1 Еволюція Serverless

Трансформація монолітної архітектури до мікросервісної архітектури, а згодом до безсерверної зумовлена необхідністю більшої швидкості і маштабованості. Таким чином, мікросервісна архітектура перетворилась в ключовий метод надання командам розробників гнучкості та інших переваг, таких як можливість доставки додатків з великою швидкістю, використовуючи IaaS та PaaS середовища. Концепція цієї ідеї полягає в розбитті монолітних додатків на менші сервіси, кожен із яких мав свою бізнес-логіку. У монолітній архітектурі одна несправна служба може поламати весь серверний додаток та всі служби, що працюють на ньому. При мікросервісній архітектурі кожна служба працює у своєму власному контейнері, і, отже, архітектори програм можуть самостійно розробляти, управляти та масштабувати ці служби. Мікросервіси можна масштабувати та розгортати окремо та писати різними мовами програмування. Але ключовим рішенням, з яким стикаються багато організацій при розгортанні мікросервісної архітектури, є вибір між середовищами IaaS та PaaS. Мікросервісна архітектура включає в себе управління вихідним кодом, сервер збірки, сховище коду, сховище зображень, диспетчер кластерів, планувальник контейнерів, динамічне знаходження сервісів, балансувальник навантаження програмного забезпечення та хмарний балансувальник навантаження. Також необхідна команда DevOps з досвідом для підтримки continuous delivery.

FaaS архітектура робить крок далі, роблячи додаток більш деталізованим до рівня функцій та подій. Відбувся перехід від монолітів до мікропослуг і функцій. FaaS також покращує недоліки моделі PaaS, такі як масштабування. Масштабування мікросервісу, розміщеного на PaaS, є складною задачею. Архітектура може мати елементи, які написані різними

мовами програмування, розгорнуті в декількох хмарах та локальних місцях, що працюють на різних контейнерах. Коли зростає навантаження на додаток, усі основні компоненти мають бути скоординовані для масштабування, або повинна бути наявна можливість для визначення, які окремі елементи потрібно масштабувати, щоб задовольнити навантаження. Навіть якщо PaaS додаток буде налаштовано на автоматичне масштабування, то не буде можливості зробити це на рівні окремих запитів, якщо не має інформації про тенденцію трафіку. Тож додаток FaaS набагато ефективніший, коли справа стосується витрат.

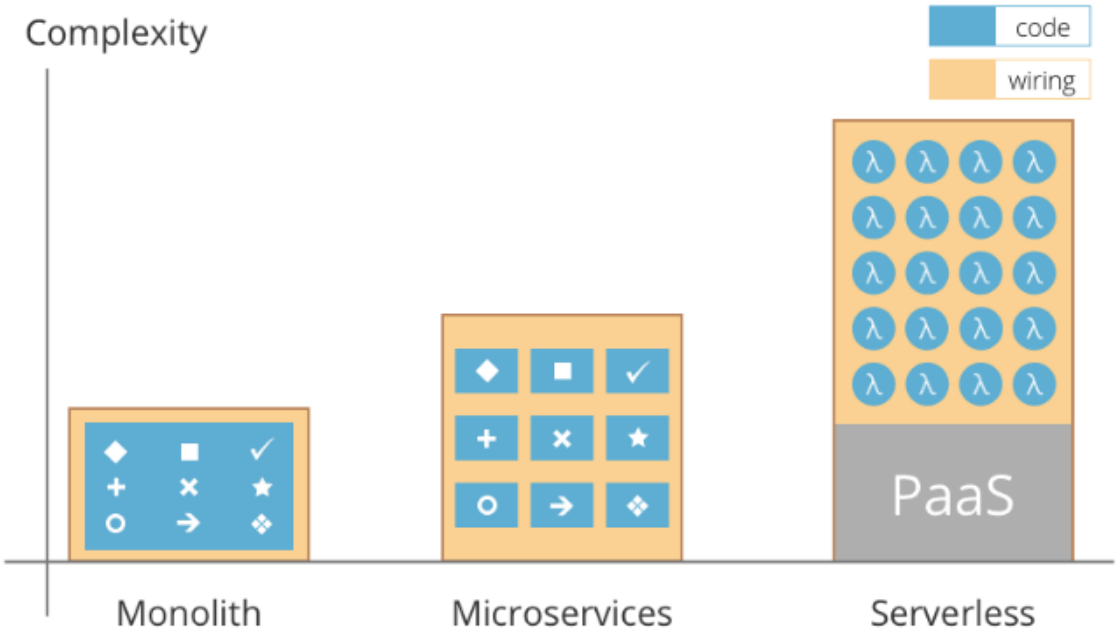


Рисунок 1.1 — Порівняння існуючих типів архітектур

Тим не менш, завжди буде місце для співіснування мікросервісної архітектури і FaaS, оскільки є певні задачі, які взагалі не можуть бути зроблені з функціями. Наприклад, API / мікросервіс завжди буде реагувати швидше, оскільки він може тримати зв’язки з базами даних та іншими сервісами відкритими та готовими. Більше того, тут слід зазначити ще одне: групуючи набір функцій за API gateway, створюється мікросервіс. Це показує, що обидва підходи можуть співіснувати [1].

1.2 Сервіси IaaS, PaaS, CasS, FaaS і SaaS

Однією з найбільших тенденцій в IT за останнє десятиліття — це бурхливий розвиток бізнес-моделі "як послуга", також відомої під абревіатурою XaaS. У моделі XaaS бізнес надають послуги та програми через Інтернет, а не встановлюють їх на локальних машинах [2].

До появи хмарних обчислень, сервери були однією з основних складових для розробників, коли мова заходила про створення додатків. Доводилось купувати або брати в оренду сервери, організовувати електропостачання, витратні матеріали, охолодження, підключення кабелів. І це все ще вимагало багато детального планування, величезної кількості часу і, головне, грошей. IaaS був ідеальним рішенням для зниження витрат, гнучкості та масштабованості з додаванням надійності та ідеальної архітектури.

Завдяки віртуальним машинам, без попередніх витрат і невеликої кількості зусиль, розробники змогли запустити сервери з обраною операційною системою. Більше того, загальна вартість витрат різко зменшилась [3].

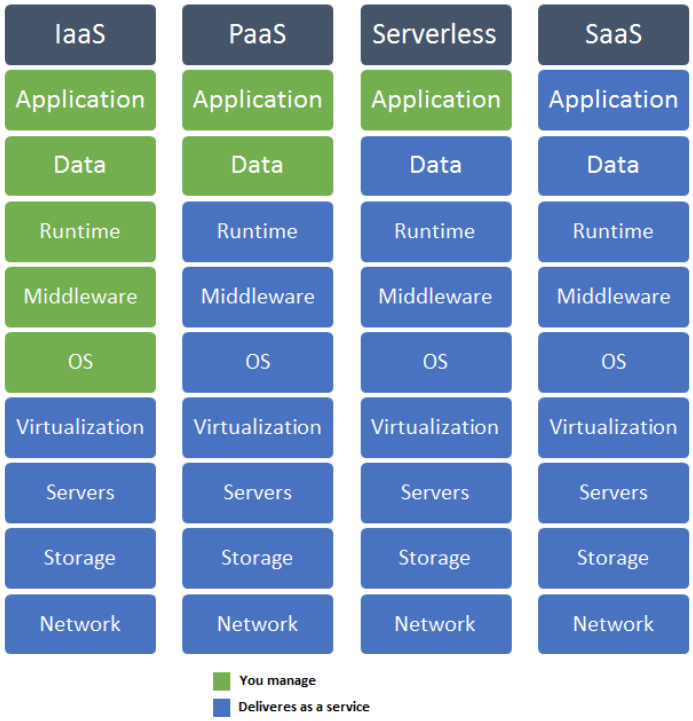


Рисунок 1.2 - Порівняння типів бізнес-моделей “як послуга”

1.2.1 Сервіс IaaS

Інфраструктура як послуга - це найнижчий рівень серед усіх хмарних моделей. Дана модель наведена на рис.1.2. Він надає велику потужність, але вимагає багато налаштувань. IaaS надає віртуальну машину, віртуальну мережу, яку розробники повинні підтримувати. Різниця між IaaS та наявністю фізичної серверної кімнати полягає в тому, що не потрібно купувати будь-які фізичні комп'ютери, і є можливість мати сервери в різних частинах світу. Середовище IaaS є альтернативою запуску приватного дата-центру. Сучасні компанії все частіше не хочуть інвестувати в сервери, сховища та мережі для приватних центрів обробки даних та управляти ними з кількох причин, зокрема, через вартість [4].

Початок роботи з IaaS має значні накладні витрати, але ці системи можна легко налаштувати, забезпечуючи адміністраторам детальний контроль над серверами, сховищами та мережевими послугами. Порівняно з іншими рішеннями, IaaS важче підтримувати, і для цього потрібен хороший інженер DevOps, який налаштовує віртуальні машини для ефективної та безпечної роботи, забезпечує доступ до комп'ютерів (віртуальних або на виділеному обладнанні), мереж та сховищ.

Приклади: Google Compute Engine, Amazon Web Services, Digital Ocean.

1.2.2 Сервіс CaaS

Це один крок вгору по сходах абстракції. Там, де платформи IaaS використовують віртуальні машини або оголене металеве обладнання як основні ресурси, платформи CaaS пакують програми та всі їх залежності в контейнери, більш легкі, ніж віртуальні машини. З цієї причини на одному хості можна розмістити більше контейнерів, ніж повноцінних віртуальних машин. CaaS - це підходяща платформа для розробників, які хочуть отримати більший контроль над задачами контейнерізації. Використання CaaS дозволяє

розробникам розгортати програми на контейнерах, не турбуючись про обмеження в оркестрації контейнерів, передбачені типовим PaaS, такими як підтримка мови або мультихмарний хостинг.

Приклади: AWS EC2 container service, Google Kubernetes, Docker Swarm.

1.2.3 Сервіс PaaS

Платформа як послуга - це простий спосіб розгортання програми з використанням певної технології (наприклад, Node.js, Ruby, PHP, Python, Java, .NET). Найпопулярнішими платформами є Salesforce Heroku, Google App Engine, AWS Elastic Beanstalk. Це інтегроване рішення для розробки та розгортання додатків, яке звільняє розробників від більшої частини забезпечення, налаштування та управління апаратними ресурсами. Розробники отримують вигоду від набагато більшої автоматизації за рахунок менш детального контролю.

Як правило, налаштовувати CI не потрібно. Достатньо просто натиснути коміт, він розпізнає, що додаток знаходиться в Node.js, і запустить npm install і npm start. Якщо програма написана на Ruby, вона буде запускати пакетну інсталяцію та подібні команди для інших середовищ.

Основний недолік полягає в тому, що ця модель не надто гнучка, оскільки неможливо встановити власні системні залежності (наприклад, від apt-get), і можливо використовувати лише одну з доступних технологій. Ще одним недоліком є те, що дані не захищені. Наприклад, якщо розробник використовує Heroku як Paas, а mLab - як DBaaS, не тільки mLab має доступ до його даних, але і Heroku , оскільки невідомо, який код фактично запускається на сервері [3].

1.2.4 Сервіс SaaS

Це ще одна хмарна модель, в якій провайдер не надає користувачам віртуальні комп’ютери або навіть мережі. Натомість він надає доступ до баз

даних або до будь-яких інших програмних сервісів, що в свою чергу дозволяє користувачам залишатися зосередженими на реальному використанні програмного забезпечення [5].

Програмне забезпечення як послуга надає готове програмне забезпечення, такі як бібліотеки NPM або Office365, але воно не вимагає від нас розгортання або обслуговування сервера. Простий приклад - це програма для розсилки, така як SparkPost або SendGrid. Все, що потрібно зробити, це надіслати HTTP-запит із адресою відправника електронної пошти, адресою одержувача, темою електронної пошти, вмістом електронної пошти тощо. З іншого боку, без такого роду інструменту розробникам довелося б налаштувати SMTP-сервер та масштабувати його у міру зростання кількості електронних листів.

Інші приклади: Google Apps (наприклад, Google Drive), DropBox і Slack - цими програмами може користуватися людина, але вони також мають великі можливості інтеграції.

1.2.5 Сервіс FaaS

FaaS навіть простіший, ніж PaaS. Як випливає з назви, він базується на функціях, які може ініціювати певна подія, тому це архітектура, заснована на подіях. Він відноситься до безсерверної архітектури. Розробник пише функцію і не повинен займатися такими задачами, як розгортання, ресурси сервера, масштабованість. FaaS можна автоматично масштабувати. Отже, плата йде за реальне споживання, а не за заявлені потреби у ресурсах.

Приклади таких сервісів - AWS Lambda, Google Cloud Functions, IBM Cloud Functions, Microsoft Azure Functions [3].

1.3 Поняття Serverless і FaaS

Serverless насправді охоплює цілий ряд технічних питань і технологій. Можна згрупувати ці ідеї у дві області: Backend-as-a-Service (BaaS) and Functions-as-a-Service (FaaS).

BaaS — це заміна компонентів на стороні сервера, які розробник кодує та управляє власноруч за допомогою готових послуг. Це поняття більше наближене до Software-as-a-Service (SaaS), ніж до екземплярів віртуальних машин та контейнерів. Послуги BaaS - це дистанційні компоненти загального призначення(тобто, не внутрішньо-процесні бібліотеки), які можливо включити у свої продукти за допомогою типової парадигми інтеграції - API.

BaaS став особливо популярним серед команд розробників, що розробляють мобільні додатки або SPA. Розробник може розраховувати на використання сторонніх сервісів для виконання задач, які в іншому випадку потрібно було робити самому. Наприклад, величезна екосистема хмарних баз даних - DynamoDB, Firebase, CosmosDB, служб аутентифікації - AWS Cognito, Auth0, Amazon RDS сервіс, Kinesis, тощо.

З іншої сторони Serverless розглядається як — Function-as-a-Service (FaaS) [6].

Serverless - це модель виконання хмарних обчислень, де провайдер хмарних послуг динамічно управляє розподілом та наданням серверів. Розробник не взаємодіє з віртуальними машинами або фізичними серверами: вони автоматично розгортаються у хмарі постачальниками. Безсерверна програма запускається в обчислювальних контейнерах без збереження стану, які ініціюються подіями, є тимчасовими (можуть тривати один виклик) і повністю управляються хмарним постачальником. Хмарні провайдери дбають про забезпечення, підтримку та масштабування безсерверної архітектури. Ціноутворення базується на кількості виконань, а не на заздалегідь куплених обчислювальних потужностях.

FaaS - це реалізація безсерверної архітектури, де інженери мають можливість розгорнути окрему функцію як частинку бізнес-логіки [7].

FaaS забезпечує абстракцію інфраструктури, що дозволяє розробникам більше зосереджуватися на виконанні коду без стану у відповідь на події**.**

Архітектура FaaS - це не що інше, як ефемерні контейнери, що виконують код. На відміну від традиційної мікросервісної архітектури, розробнику не потрібно турбуватися про балансувальник навантаження, екземпляри EC2 та масштабування. Розробники більше зосереджуються на коді, а не на інфраструктурі.

FaaS - це новий та вдосконалений спосіб створення та розгортання серверного програмного забезпечення, яке більше зосереджується на розгортанні окремих функцій або операцій.

Традиційний спосіб розгортання програмного забезпечення на стороні сервера - це почати з екземпляра хоста - наприклад, екземпляра віртуальної машини або контейнера. Потім розгортається додаток на цьому хості. У випадку, якщо хостом є віртуальна машина або контейнер, тоді програма є процесом операційної системи. Зазвичай додаток містить код для кількох взаємопов’язаних операцій, як показано на рис. 1.3.

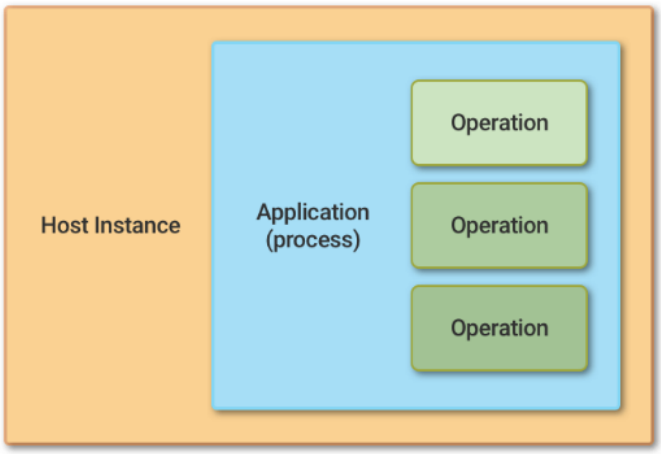


Рисунок 1.3 - Структура традиційного способу розгортання додатку

FaaS суттєво змінює цю модель розгортання. Виділяється з моделі як екземпляр хосту, так і процес додатку, а замість цього йде зосередження лише на операціях або функціях, що виражають логіку додатка. Ці функції завантажуються індивідуально на платформу FaaS, що постачається провайдером, і ця платформа виконує всю роботу, щоб запустити код від імені розробника, як зображено на рис. 1.4.

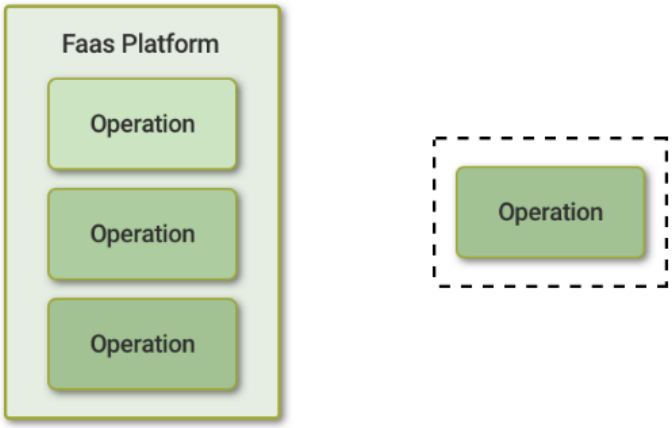


Рисунок 1.4 - Структура розгортання додатку на FaaS платформі

Однак ці функції не є постійно активними в процесі сервера. Вони знаходяться у простої до тих пір, поки їх не потрібно буде запускати, як це було б у традиційній системі. Натомість платформа FaaS налаштована таким чином, що вона реагує на певну подію для кожної операції. Коли ця подія відбувається, платформа постачальника ініціює функцію, а потім викликає її для ініціюючої події.

Як тільки функція завершить своє виконання, платформа FaaS може її вимкнути. В якості альтернативи, як оптимізація, вона може зберігати її деякий час, поки не з’явиться інша подія для обробки. Цей підхід - керований подіями. Більше того, постачальник FaaS також інтегрується з різними синхронними (HTTP API Gateway) та асинхронними (чергами повідомлень) джерелами подій, окрім надання платформи для розміщення та виконання коду [6].

Наразі AWS Lambda є однією з найпопулярніших реалізацій платформи Functions-as-a-Service. Serverless не означає, що серверів більше немає, це означає, що розробнику більше не потрібно ними перейматися.

BaaS і FaaS пов'язані за своїми експлуатаційними атрибутами (наприклад, відсутність управління ресурсами) і часто використовуються разом. Усі великі постачальники хмарних послуг мають « Serverless products», що включають продукти BaaS і FaaS [8].

Наприклад, AWS надає повний набір сервісів, які він контролює, для створення та запуску serverless додатків. Наприклад, AWS Lambda, Amazon API Gateway, Amazon DynamoDB, Amazon Simple Notification Serivice, тощо. Serverless додатки не вимагають забезпечення, обслуговування та адміністрування серверів для backend-компонентів, таких як обчислення, бази даних, зберігання, обробка потоків, черги повідомлень тощо. Розробникам також більше не потрібно турбуватися про забезпечення відмовостійкості та доступності програми. Натомість AWS обробляє усі ці можливості [9].

База даних BaaS Google Firebase має явну підтримку FaaS через Google Cloud Functions для Firebase.

Розглянемо на прикладі традиційну трьохрівневу, орієнтовану на клієнта систему з логікою на стороні сервера. Гарний приклад - типовий додаток для електронної комерції.

Рисунок 1.5 - Традиційна трьохрівнева система додатку зоомагазин

Традиційно архітектура буде виглядати приблизно так, як показано на рис. 1.5. Скажімо, серверна частина реалізована на PHP або Go, а клієнтом є компонент HTML та JS:

Більша частина логіки в системі, такі як аутентифікація, навігація по сторінках, пошук, транзакції - реалізована на сервері. З serverless архітектурою архітектура може виглядати приблизно так, як зображено на рис. 1.6:

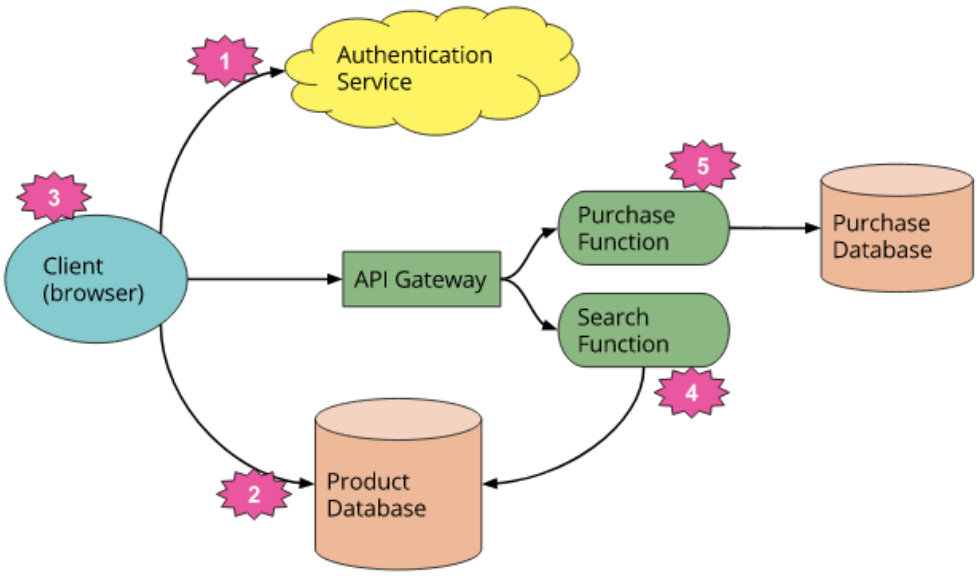


Рисунок 1.6 — Serverless архітектура

Це суттєво спрощена схема, але навіть тут можна спостерігаємо ряд суттєвих змін:

1. Відповідно до цієї архітектури аутентифікація та управління користувачами були видалені з серверної частини і тепер здійснюватиметься службою BaaS, такою як AWS Cognito або Auth0 . Ці служби можна викликати безпосередньо з клієнтського додатку для вирішення таких задач, які виникають перед користувачем,наприклад, реєстрація та аутентифікація. Завдяки керуванню користувачами та аутентифікації, які зараз обробляє BaaS, логіка спрощується.
2. Надано дозвіл клієнту на прямий доступ до підмножини бази даних (для списків продуктів), яка сама повністю розміщується стороннім провайдером (наприклад, Google Firebase або DynamoDB.)

3. Деяка логіка, яка була на сервері Pet Store, тепер знаходиться в межах клієнта - наприклад, відстеження сеансу користувача, зчитування з бази даних і презентація у придатний для перегляду вигляд і т. д. У цьому випадку клієнт може виступати як SPA.
4. Кожна окрема операція може бути інкапсульована у функцію. Платформа FaaS, така як AWS Lambda, приймає кожну з цих функцій і запускає їх у паралельних «контейнерах», які можна контролювати та масштабувати окремо. У додатку “зоомагазин” (рис. 1.5) прикладом є "пошук". Замість того, щоб мати постійно запущений сервер, як це було в оригінальній архітектурі, натомість можливо реалізувати FaaS функції, які відповідають на запити HTTP через API gateway. Функція "пошук" клієнта та сервера зчитує дані з однієї бази даних про товари. Якщо буде вирішено використовувати AWS Lambda як платформу FaaS, то можливо перенести код пошуку з оригінального сервера Pet Store на нову функцію пошуку без будь-яких змін коду, оскільки Lambda підтримує багато мов програмування.
5. Нарешті, можливо замінити функцію "придбання" іншою окремою функцією FaaS, вирішивши зберегти її на стороні сервера з міркувань безпеки, а не повторно впроваджувати в клієнті. Він також розміщений через API gateway. Розбиття різних логічних вимог на окремо розгорнуті компоненти є дуже поширеним підходом при використанні FaaS.

У монолітній версії за усі задачі керування, роботу з потоком даних і безпеку відповідав єдиний додаток центрального сервера. У serverless версії немає центрального механізму, який би займався вирішенням цих задач. Кожен компонент відіграє більш архітектурно визначену роль [8].

1.4 Переваги FaaS

Розглянемо переваги, які надає FaaS.

1.4.1 Автоматичне масштабування і виділення ресурсів

За допомогою традиційних серверних архітектур (монолітної) розробник відповідає за низку заходів, що стосуються використання ресурсів додатків:

- Планування ресурсів - оцінка того, який тип і скільки ресурсів потрібно додатку. Наприклад, які типи серверів чи екземплярів найкраще підходять для різних його компонентів, і скільки екземплярів потрібно, щоб задовольнити очікуване навантаження.
- Розподіл ресурсів - відображення кожного компонента програми на ресурс, на якому він буде розгорнутий
- Надання ресурсів - процес придбання екземплярів машини та їх налаштування готового до розгортання
- Масштабування ресурсів - коригування всіх трьох попередніх дій на основі фактичного навантаження, яку має обробляти додаток у будь-який момент часу

Проблема полягає в тому, що всі ці налаштування потребують часу, і особливо планування ресурсів далеко не є точною діяльністю. Завжди потрібно буде коригувати тарифний план, і іноді це означає, що попередні ітерації резервування призведуть до марних зусиль або навіть марно витрачених ресурсів. Зазвичай для того, щоб уникнути цього, планується більше навантаження, ніж очікується, що призводить до надмірного забезпечення ресурсами. Це призводить до недостатньої ефективності - маємо більше ресурсів, ніж потрібно системі в будь-який час. Безсерверне управління повністю змінює управління ресурсами. Більше не потрібно планувати, розподіляти чи надавати екземпляри сервера або часто будь-яких інших ресурсів. Це пов’язано з тим, що автономне надання послуг без сервера. Іншими словами, розробники просто починають користуватися послугою, і вона зрозуміє, скільки і якого типу, ресурсів потрібно, і автоматично надає їх для потреб. Давайте розглянемо Serverless функції як приклад. Коли

створюється функція AWS Lambda, код прив’язується до події, додається певна логічна конфігурація, і все. Коли трапляється подія активації, платформа AWS Lambda автоматично створює екземпляр контейнера для виконання коду всередині і сама виділяє та надає хост для запуску контейнера. Розробник не бере участі в процесі.

Окрім автоматичного надання нового інстансу контейнера, безсерверні служби продовжуватимуть розподіляти та забезпечувати обробку навантаження у будь-який момент до двох, трьох та більше одиниць. Іншими словами, безсерверні послуги мають самостійне горизонтальне масштабування Звичайно, існують системи автоматичного масштабування, але перевага FaaS полягає в тому, що не потрібно виконувати будь-яку роботу з управління автоматичним масштабуванням - це повністю надається провайдером. Крім того, Serverless дозволяє автоматичне масштабування типів компонентів, які, як правило, було дуже важко самостійно масштабувати, наприклад, бази даних.

1.4.2 Зменшення вартості ресурсів

Як правило, коли розробник працює з програмами, то треба визначити, які і скільки базових хостів треба запускати. Наприклад, скільки RAM та CPU сервери баз даних потребують? Скільки різних інстансів потрібно для підтримки масштабування?

AWS Lambda може масштабуватися без будь-яких зусиль. На даний момент обмеження паралельного виконання функції за замовчуванням для лямбди становить 1000 на регіон. Тож лямбда створить 1000 одночасних екземплярів даної функції.

FaaS повинна виставляти рахунки на основі точного використання. Розробник платить лише коли функція насправді обробляє подію — тобто немає плати за час простою між запитами, за які стягується плата за сервер або контейнер. Оскільки розробник платить за 100 мс активного використання, це

також означає, що немає більше плати за 1000 лямбда-функцій, що працюють паралельно, порівняно із запуском послідовно. Плата буде зніматись тільки за те, як довго виконується наш код незалежно від того, як багато контейнерів запуститься. Наприклад, виклик 100 FaaS функцій в одному контейнері послідовно буде коштувати стільки ж само, як і при виклику функціїї FaaS 100 разів в 100 різних контейнерах, припускаючи, що загальний час виконання по всім подіям приблизно однаковий [6].

1.4.3 Відсутність відповідальності за backend інфраструктуру

Завдяки Serverless значно зменшується кількість різних технологій, за які розробник відповідаємо безпосередньо. Не маючи backend інфраструктури для управління, оскільки все залежить від провайдерів хмарних послуг, не має проблем із технічним обслуговуванням, які б могли виникнути, якби робота з сервером відбувалась самостійно. Це включає в себе оновлення, виправлення та слідкування за потенційними зломами.

1.4.4 Швидке розгортання і оновлення

Використовуючи FaaS інфраструктуру, немає необхідності завантажувати код на сервери або виконувати будь-яку конфігурацію бекенда, щоб задеплоїти робочу версію програми. Розробники можуть дуже швидко завантажувати фрагменти коду та релізити новий продукт. Вони можуть завантажувати код відразу або одну функцію за раз, оскільки додаток - це не один монолітний стек, а набір функцій, наданих постачальником.

Це також дозволяє швидко оновлювати, виправляти або додавати нові функції до програми. Не потрібно вносити зміни в цілу програму; натомість розробники можуть оновлювати програму по одній функції за раз [10].

1.5 Недоліки і обмеження FaaS

А тепер розглянемо недоліки і обмеження FaaS.

1.5.1 Стан

Функції FaaS мають суттєві обмеження, коли мова йде про локальний стан (пов'язаний з машиною/екземпляром), тобто дані, які зберігаються у змінних в пам'яті, або дані, які записуються на локальний диск. Не можна гарантувати, що стан зберігається під час декількох викликів, і, що ще важливіше, не можна вважати, що стан з одного виклику функції буде доступним для іншого виклику тієї самої функції. Тому функції FaaS часто описуються як stateless. Будь-який стан функції FaaS, який повинен бути постійним, повинен бути створений поза екземпляром функції FaaS.

1.5.2 Час виконання

Функції FaaS, як правило, обмежені тим, як довго може виконуватись кожний виклик функції. На даний момент «час очікування» для функції AWS Lambda для реагування на подію становить щонайбільше п’ять хвилин, перш ніж виклик функції буде припинений. Microsoft Azure та Google Cloud Functions мають подібні обмеження. Це означає, що певні класи довготривалих задач не підходять для функцій FaaS без зміни архітектури - можливо, доведеться створити декілька різних скоординованих функцій FaaS, тоді як у традиційному монолітному додатку може бути одна довготривала задача, яка виконує як координацію, так і виконання.

1.5.3 Затримка запуску та "холодний старт"

Платформа FaaS займає певний час, щоб ініціалізувати екземпляр функції перед кожною подією. Ця затримка запуску може істотно змінюватися,

навіть для однієї конкретної функції, залежно від великої кількості факторів, і може коливатися від декількох мілісекунд до декількох секунд.

Ініціалізація лямбда-функції може виконуватись як «теплий старт» - повторним використанням екземпляра функції Лямбда та її контейнера-хоста з попередньої події, або «холодний старт» - створенням нового екземпляра контейнера, запуском процесу хосту функції тощо. Не дивно, що, враховуючи час затримки запуску, саме ці холодні старти викликають найбільше проблем.

Затримка холодного запуску залежить від багатьох аспектів: мови, якою написаний додаток, кількості сторонніх бібліотек, кількості коду, конфігурації самого середовища функції Lambda, чи потрібно вам підключатися до ресурсів VPC тощо. Ці аспекти перебувають під контролем розробника, тому часто можна зменшити затримку запуску, спричинену холодним запуском.

Настільки ж важливим параметром, як тривалість холодного старту, є частота холодного старту. Наприклад, якщо функція обробляє 10 подій в секунду, причому кожна подія займає 50 мс на обробку, швидше за все, холодний запуск з’явиться з лямбдою кожні 100 000–200 000 подій. Якщо, з іншого боку, подія обробляється раз на годину, то, швидше за все, побачимо холодний старт для кожної події, оскільки Amazon вилучає неактивні екземпляри лямбди через кілька хвилин. Якщо AWS Lambda виконує обробку не менше однієї події в секунду, то більш ніж 99.99% подій будуть оброблятись теплим стартом. Знання цього допоможе зрозуміти, чи вплинуть холодні старти на додаток у сукупності, і чи не буде бажання виконати “keep-alive” своїх екземплярів функцій, щоб уникнути їх видалення.

1.5.4 Знижений загальний контроль

- Конфігурація — втрата повного контролю над конфігурацією FaaS
- Продуктивність коду додатку і FaaS платформи
- Безпека

1.5.5 Прив’язка до провайдера

Побудова безсерверних функцій на одній платформі може означати, що перехід на іншу буде складним. Можливо, потрібно буде переписати код, API, які існують на одній платформі, може не існувати на іншій, і для переходу з AWS на Azure або Google Cloud потрібно буде виділити значні ресурси.

Якщо існують плани інвестувати в безсерверну платформу, то необхідно переконатися, що постачальник, який розглядається, має весь необхідний функціонал [13].

1.5.6 Проблеми локального тестування

Кожного разу, коли serverless інстанс запускається, він створює нову версію себе, а це означає, що важко зібрати дані, необхідні для відладки та виправлення serverless функції.

Існують сторонні інструменти, які реєструють події serverless функцій, але без додавання додаткового інструменту відладка serverless функції може бути важкою задачею. Наприклад, AWS вимагає покрокової відладки (послідовної передачі), щоб з'ясувати, що йде не так. Відладка serverless функцій можлива, але це непросте завдання, і воно може зайняти багато часу та ресурсів [6].

1.6 Висновки

Процес побудови додатків зазнав суттєвих змін впродовж останніх 10 років. Монолітні системи мали в собі багато недоліків, починаючи від проблем масштабованості і закінчуючи тим, що одна несправна служба монолітного додатку виводила з ладу увесь додаток. Отже сталася трансформація побудови монолітної архітектури додатків до мікросервісної, а в останні роки — до безсерверної, яка зумовлена необхідністю кращої масштабованості, простотою розгортання додатків і швидкістю їх написання з подальшою їх підтримкою і впровадженням нового функціоналу. FaaS архітектура за рахунок ефективної масштабованості за потребою і передбачуваних витрат зайняла провідне місце у розробці додатків у хмарі. Зважаючи на розглянуті поняття FaaS і serverless, можна зробити висновок, що FaaS — це не що інше, як реалізація serverless архітектури, де провайдер хмарних послуг динамічно управляє розподілом і наданням серверів. Також були розглянуті FaaS властивості, які роблять написання і розгортання додатків набагато легшими, ніж на основі контейнерів. Звісно, FaaS, як і інша архітектурна модель, має певні недоліки, такі як збереження стану функцій, їх “холодний старт”, зниження загального контролю над системою і прив’язка до певного провайдера, але всі вони можуть бути вирішені в певній мірі, використовуючи певні практики і архітектурні шаблони, які будуть розглянуті у наступних розділах. Такі переваги FaaS, як самостійне автоматичне масштабування і розгортання, відсутність системного адміністрування і відсутність відповідальності за backend інфраструктуру виділяють FaaS системи зпроміж монолітних і мікросервісних і є надзвичайно цікавими для подальшого дослідження.

2 ПОРІВНЯННЯ ПРОПОЗИЦІЙ FAAS ПРОВАЙДЕРІВ

2.1 Сучасні FaaS провайдери

На сьогоднішній день існує три найпопулярніших FaaS провайдерів, які готові до роботи у продакшені і є загальнодоступними — це AWS з сервісом Lambda, Azure, який має свій сервіс Azure Functions і Google з serverless обчислювальним сервісом — Google Cloud Functions. AWS Lambda стала загальнодоступною у 2014 році, Azure Functions - наприкінці 2016 року, а Google Cloud Functions - зовсім недавно, у липні 2018 року. З такою великою різницею їх створення очевидно, що AWS Lambda домінує у serverless світі завдяки величезному ком'юніті розробників, надзвичайно великій інфраструктурі хмарних рішень та інтеграції з ними, широкому спектру задач, які можна вирішити за допомогою AWS Lambda. Azure Functions підтримують різноманітні підтримувані середовища виконання [13]. У даному розділі будуть детально розглянуті перераховані вище FaaS провайдери з їх детальним порівнянням.

2.1.1 AWS Lambda

Lambda — це обчислювальний сервіс, який дозволяє запускати код без виділення та управління серверами. Він забезпечує рівень хмарної логіки для нашого додатку. Лямбда функції можуть бути ініційовані різними подіями, які відбуваються в AWS або в сторонніх сервісах, які підтримуються. Вони дають змогу створювати реактивні, подієво-орієнтовані системи. Коли є декілька одночасних подій, на які слід відповісти, Lambda просто паралельно запускає декілька копій функції. Масштабованість лямбда-функцій відповідає робочому навантаженню. Таким чином, ймовірність наявності простою сервера або контейнера надзвичайно мала. Архітектури, які використовують лямбда-

функції призначені для зменшення втрат потужності. AWS Lambda запускає код на обчислювальній інфраструктурі високої доступності та виконує всі адміністрування обчислювальних ресурсів, включаючи обслуговування сервера та операційної системи, забезпечення пропускної здатності та автоматичне масштабування, моніторинг коду та логування. Все, що потрібно зробити, - це надати код однією з мов, які підтримує AWS Lambda [14]. Кожна лямбда-функція, яка створюється, містить код, який треба виконати, файл конфігурацій, який визначає, як виконується даний код, і, за бажанням, одне або більше джерел подій, які відслідковують і виявляють події та викликають дану функцію в міру їх виникнення.

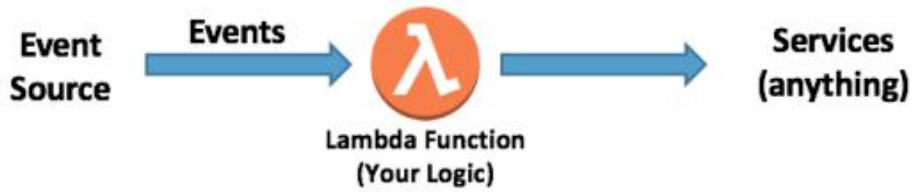


Рисунок 2.1 - Спрощена архітектура запуску лямбда-функції

Прикладом джерела події може бути API Gateway, який може викликати лямбда-функцію щоразу, коли метод API, створений за допомогою API Gateway, отримує запит HTTPs. Інший приклад - Amazon SNS, який може викликати лямбда-функцію в будь-який час, коли нове повідомлення розміщується в SNS topic. Lambda також надає сервіс API RESTful, який включає можливість безпосереднього виклику лямбда-функції. Існує можливість використовувати цей API для виконання коду безпосередньо, без налаштування інших джерел подій. Також можливо використовувати AWS Lambda для запуску коду у відповідь на події, такі як зміни даних у сегменті Amazon S3 або таблиці Amazon DynamoDB. Більше не потрібно писати будь-який код, щоб інтегрувати джерело події з лямбдою, керувати будь-якою

інфраструктурою, яка виявляє події та доставляє їх до нашої функції, або керувати масштабуванням нашої лямбда-функції відповідно до числа подій, які ініціюються. Розробник зосереджується на логіці свого додатку та налаштуванні джерела подій, які викликають роботу даної логіки.

Після налаштування джерела події для функції, код буде викликатись, коли подія ініціюється. Код може виконувати будь-яку бізнес-логіку, звертатись до зовнішніх веб-сервісів, робити інтеграцію з іншими службами AWS або будь-що інше, що потребує додаток.

Весь код поміщається у пакет — package. Package містить усі ассети, які розробник хоче мати у своєму розпорядженні локально після виконання коду. Пакет, як мінімум, включатиме код функції, яку служба Lambda буде виконувати, коли функція викликається. Однак він може містити й інші ресурси,на які код буде посилатися під час виконання, наприклад, додаткові файли, класи та бібліотеки, які код буде імпортувати, бінарні файли, які треба виконати, або файли конфігурації, на які код може посилатися під час виклику. Максимальний розмір пакету становить 50 МБ у стисненому вигляді та 250 МБ у розархівованому.

При написанні коду важливо розуміти, що код не може робити припущення про стан. Lambda повністю визначає, коли буде створений новий контейнер функцій та викликаний вперше. Контейнер може бути викликаним вперше з ряду причин. Наприклад, події, що викликають лямбда-функцію збільшуються в паралелізмі понад кількості контейнерів, раніше створених для функції, подія запускає лямбду, або, подія запускає лямбду вперше за тривалий проміжок часу і т. д. Лямбда відповідає за збільшення і зменшення контейнерів функцій для задоволення фактичного навантаження. Це означає, що код не може робити жодних припущень, що стан буде збережений від одного виклику до наступного. Однак кожен раз як функція-контейнер створюється і викликається, він залишається активним і доступним для наступних викликів принаймні декілька хвилин, перед тим як він завершить свою роботу. Якщо

наступні виклики відбуваються в контейнері, який вже був активним і викликався хоча б раз за останні декілька хвилин, то виклик виконується на “warm” контейнері, як показано на рис. 2.2. Коли відбувається виклик лямбда-функції, який вимагає початкового створення пакету функції і виклику вперше, то це виклик “cold” старту [15].

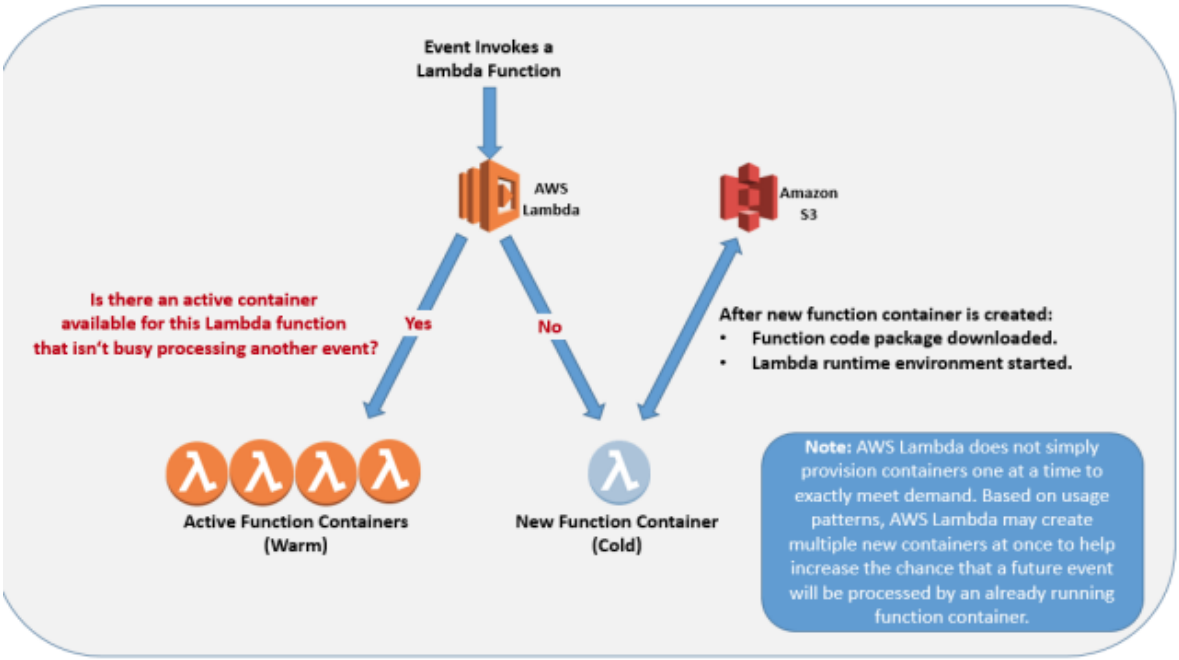


Рисунок 2.2 — Теплий і холодний старти лямбда-функції

2.1.2 Microsoft Azure Functions

Сервіс був запущений у 2016 році, щоб конкурувати з AWS Lambda. Azure Functions пропонує набір послуг, подібних до Amazon, з акцентом на сімейство мов та інструментів Microsoft. Azure Functions - це serverless обчислювальне рішення від Microsoft, яке дозволяє користувачу запускати код на вимогу. Користувачу не потрібно надавати або управляти інфраструктурою. Завдяки цьому розробники можуть легко запускати невеликі фрагменти коду, тобто функції в хмарі. Розробникам просто потрібно написати код, не турбуючись про базову інфраструктуру.

За допомогою Azure Functions хмарна інфраструктура забезпечує всі сучасні сервери, необхідні для забезпечення масштабованої роботи додатку. Функція ініціюється певним типом події. Підтримувані тригери включають відповідь на зміни даних, відповідь на повідомлення, запуск за розкладом або як результат запиту HTTP. Також функції можуть бути ініційовані за допомогою інших сервісів Microsoft, таких як Azure Event Hub і Azure Storage.

Azure Functions підтримують кілька мов у рантаймі. Розробляючи власний код для інтеграції, розробники можуть використовувати мову на власний вибір, наприклад .NET, Node.js, Java, PowerShell тощо. Що стосується хостингу, то можливо розміщувати свої функції Azure в будь-якому середовищі, наприклад, Kubernetes, Windows або середовищі Linux. Завдяки доступним прив'язками додатків, Azure Functions також відповідають моделі реактивного програмування і можуть використовуватися в безлічі випадків використання. Для розробки Azure Function можна або використовувати портал Azure або використовувати середовища розробки, такі як Microsoft Visual Studio або Visual Studio Code. Існує можливість поєднати можливості робочого процесу Logic Apps та виконання коду Azure Functions для розробки рішень інтеграції корпоративного рівня, не турбуючись про реалізацію інфраструктури. Крім того, можна розробити свій легковісний API за допомогою lightweight і скористатися можливостями API Managment, щоб надати доступ до API ззовні [16].

Окрім надання стандартних функцій FaaS, Azure Functions також надає такі переваги:

- Користувачі можуть створювати функції мовами на свій вибір, такими як C#, F# та JavaScript;
- Користувачі можуть додавати залежності за допомогою NuGet або NPM;
- Azure Functions дуже добре інтегровані з постачальниками OAuth, такими як Azure Active Directory, Facebook, Google і Twitter;

- Azure Functions дуже добре інтегровані з іншими сервісами Azure та SaaS послугами;
- Azure Functions можна створювати на порталі Azure або розгортати за допомогою конвеєрів безперервної інтеграції (CI) / безперервної доставки (CD) за допомогою служб GitHub або Azure DevOps

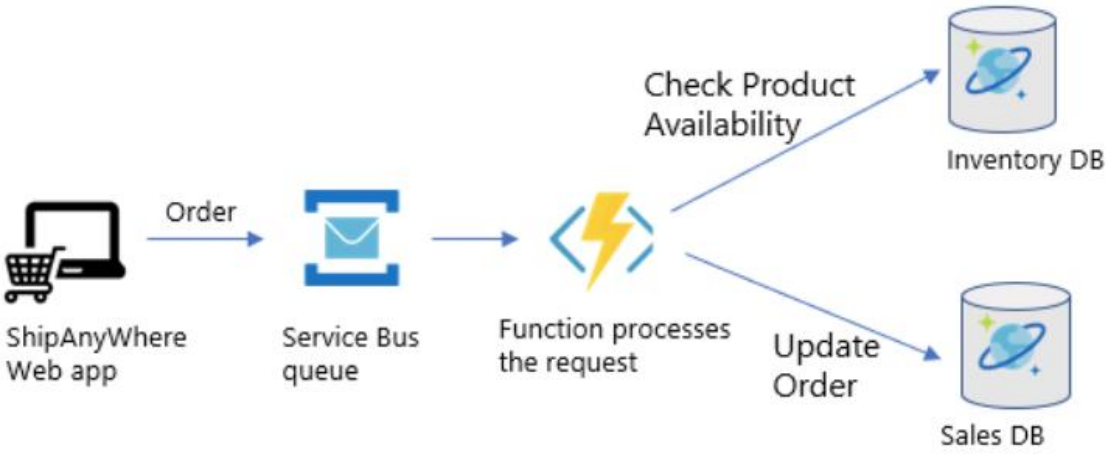


Рисунок 2.3 - Використання Microsoft Azure Functions

Durable Functions - це розширення Azure Functions. Вони дозволяють писати функції з відслідковуванням стану у безсерверному обчислювальному середовищі. Розширення дає змогу визначати робочі процеси, які відслідковують стан шляхом написання функцій оркестратора та сутностей з відслідковуванням стану шляхом написання функцій сутностей за допомогою моделі програмування Azure Functions. За лаштунками розширення керує станом, контрольно-пропускними пунктами та перезапускається, дозволяючи зосередитись на бізнес-логіці.

Завдяки Durable Functions можливо програмувати робочий процес та створювати екземпляри задач у послідовному або паралельному порядку, а також побудувати наблюдача або підтримати потік взаємодії з людиною. Також можливо зв’язати функції для контролю потоку. Розробник може використовувати сценарії Fan In — Fan Out, корелюючі події, гнучку

автоматизацію та тривалі процеси, а також схеми взаємодії з людьми, які важко застосувати лише за допомогою функцій або логічних програм.

2.1.3 Google Cloud Functions

Середовище виконання Google Cloud Functions забезпечує можливість безсерверного програмування поверх Google Cloud Platforms. Впровадження Google Cloud Functions, які є масштабованими та оплачуваними в міру використання FaaS, пропонує безліч варіантів використання у простішому середовищі, що полегшує розробникам запуск та масштабування коду в хмарі. Функції можуть бути написані на Python, Node.js або Go - тому розробникам, ймовірно, не потрібно буде вивчати щось нове для їх використання [17].

Google Cloud Functions легко вирішує багато проблем, з якими можна мати справу , наприклад:

- багато сервісів, які потребують додаткового управління;
- прогнозування трафіку для кожного сервісу
- ризик виходу з ладу сервісу під час сплеску навантаження
- переплата за невикористані потужності

Існує 2 типи Google Cloud Functions:

1. HTTP functions

HTTP functions в основному використовуються, коли є необхідність викликати функцію через запит HTTPs. Цей тип функції можна використовувати для веб-хуків або створення API. Хмарна функція абстрагує інформаційну інфраструктуру, тому не потрібно писати багато коду. HTTP тригер ініціюється з запиту, надісланого нашій функції, або може бути подією з подій Google Pub/Sub, або подією Firebase.

2. Background functions

Background functions спрацьовують, коли відбуваються певні хмарні події. Наприклад, коли файл завантажується в хмару, то функція може автоматично запускатися для обробки завантаження. Наразі фонові функції реагують на:

- Pub/Sub події
- Cloud storage
- Firebase події

Цей тип функцій дозволяє обробляти такі варіанти використання, як трансформація даних, обробка IoT повідомлень, перегляд логів і реагування на певні зміни в інфраструктурі.

Рисунок 2.4 - Робота Google Cloud Function

Cloud Services об’єднує в собі Google Cloud Platform та її різні сервіси, такі як Google Cloud Storage, Google Cloud Pub/Sub, Stackdriver, Cloud Datastore і так далі. Усі вони містять події, що відбуваються всередині. Наприклад, якщо у бакет завантажено новий об’єкт або видалено, або Cloud Pub/Sub отримує нове повідомлення, яке опубліковане за певною темою. На жаль не всі хмарні сервіси чи провайдери подій мають підтримку у Google Cloud Functions.

Коли подія, наприклад, завантаження об’єкту у бакет Cloud Storage, відбулась, то ініціюється подія, як зображено на рис. 2.5. Дані події, які пов’язані з нею, містять певну інформацію. Якщо хмарна функція налаштована на активацію цією подією, тоді відбувається виклик хмарної функції. Під час її виконання дані події передаються у функцію для визначення тригеру події і

отримання метаінформації про неї. Також функція має можливість викликати інші API. Це можуть бути API Google або зовнішні API. Після завершення виконання своєї логіки хмарна функція позначає,що вона закінчила обробку. Виникнення декількох подій одночасно призведе до декількох викликів хмарних функцій. За це відповідає інфраструктура Cloud Functions. Також не можна залежати від будь-якого стану, який був створений під час попереднього виклику функції. За необхідності існує можливість підтримувати стан поза цією структурою за допомогою якоїсь іншого сервісу, наприклад Shared Memcache [18].

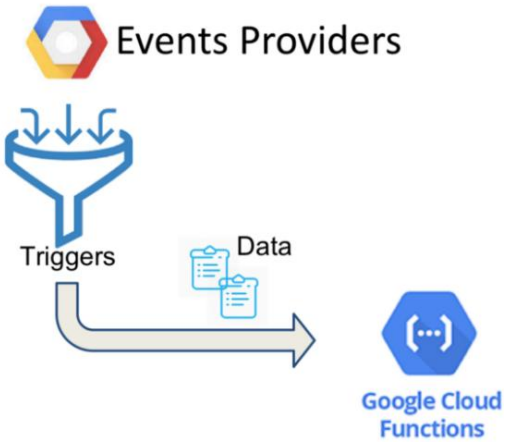


Рисунок 2.5 - Завантаження об’єкту до Cloud Storage

Перевагами Google Cloud Functions є:

1. Відсутність управління сервером

У Google Cloud Functions відсутнє управління сервером, що робить загальну архітектуру розробника більш гнучкою, до того ж створення додатку прискорюється.

2. Плата тільки за використання

Розробник платить тільки за час використання, тобто коли функція не активна, то плата не начислюється. Час виконання функції оплачується з точністю до найближчих 100 мілісекунд. Розподіл цін заснований на таких

факторах, як тривалість виконання нашої функції, скільки разів вона викликалась а також на кількості ресурсів, які ми надаємо для функцій.

3. Подієво-орієнтована архітектура
4. Автоматичне масштабуванням
5. Інтегровані сервіси

Ще однією великою перевагою Google Cloud Function є те, що вона повністю сумісна з платформою Firebase на додаток до Google Cloud Platform та Google Assistance. Firebase - це версія Google платформи для розробки мобільних доадтків, яка допомагає користувачам швидко розробляти та розвивати додаток. Firebase і Google Cloud Platform мають спільну інтеграцію, включаючи продукти, проекти, виставлення рахунків, контроль доступу, умови надання послуг та облікові записи для спрощеного використання.

2.2 Порівняльна характеристика пропозицій провайдерів

2.2.1 Підтримка мов програмування

Наразі AWS Lambda має вбудовану підтримку коду, написаного на Node.js, Python, Java (Java8 compatible), C#, F#, PowerShell, Go і Ruby.

Azure Functions мають підтримку для Node.js,, Python, Java, .NET Core та PowerShell Core. На відміну від AWS Lambda немає підтримки Go і Ruby.

За допомогою Google Cloud Functions виконання функції залежить від обраної мови. На даний момент він підтримує Node.js, Python, Java11 і Go.

2.2.2 Розгортання функцій

Загалом, для розгортання використовується Serverless framework, де розробник використовує serverless.yml файл для налаштування функцій або

внесення змін. Потім код функцій упаковується у Zip файли та надсилається на сервер. AWS Lambda оновлює кожну окрему функцію додатку, тоді як Azure Functions і Google Cloud Functions, як правило, аналізують serverless.yml через плагін і завантажують ресурси з різницею в порядку

AWS Lambda надає операції API, які можливо використовувати для створення та оновлення функцій Lambda за допомогою пакету розгортання, завантаженого у консоль у вигляді Zip файлу або відредагованого в самій консолі. AWS Lambda розгортає всі функції в середовищі Lambda на серверах, на яких працює Amazon Linux

Натомість Azure Functions надають безліч варіантів розгортання функції, таких як DropBox, Visual Studio Team Services, OneDrive, GitHub, Zip файли. Користувачі Azure Functions можуть розгортати код безпосередньо в службі Azure Functions. Але вони також можуть запускати програмне забезпечення всередині Docker контейнерів, що надає програмістам більше контролю над середовищем виконання. Функції Azure працюють із файлами Docker, які визначають середовище контейнера. Ці функції, упаковані всередині контейнерів Docker, також можуть бути розгорнуті на Kubernetes. Azure Functions також надає можливість розгортання функцій на серверах Windows або Linux. У більшості випадків головна операційна система не повинна мати значення. Однак, якщо у ваших безсерверних функцій є специфічний для ОС код або залежності, такі як мова програмування або бібліотека, яка працює лише на Linux, це важливий фактор [19].

Для розгортання Google Cloud Functions є можливість вибрати будь-який з наступних способів: CLI, Zip файл, вбудований веб-редактор, Cloud Storage або Cloud Storage Repository.

2.2.3 Управління залежностями

У AWS Lambda залежності упаковуються всередині самих пакетів розгортання. Рекомендації щодо створення пакету розгортання залежать від вибору мови. Існує можливість використовувати плагіни побудови, такі як Jenkins (для Node.js і Python) та Maven (Java) для створення пакетів.

Azure дозволяє включити файл package.json у каталог функцій і запустити npm install, як зазвичай, у проектах Node.js за допомогою консолі на порталі Azure. Цей процес подібний, незалежно від мови виконання. Також є можливість використовувати NuGet

Google Cloud Functions дозволяють управляти залежностями за допомогою пакетного менеджеру npm і виражаються у файлі метаданих, який називається package.json. Функція також має можливість завантажувати залежності, які оголошені в package.json, за допомогою npm.

2.2.4 Постійне сховище даних

У AWS Lambda функція повинна бути записана без будь-яких змінних. У той же час є можливість зберігати змінні в постійному сховищі, такі як S3, DynamoDB або будь-які інші хмарні сервіси зберігання.

У Azure Functions для зберігання постійних даних, які повинні бути постійними для багатьох екземплярів, є можливість використовувати Azure Blob Storage або Azure Table Storage.

Якщо є необхідність кешувати дані, наприклад, з'єднання з БД, то є можливість використовувати статичні змінні. Для функцій Google Cloud Functions, якщо є потреба поділитися станом між викликами функцій, функція повинна використовувати такі служби, як Cloud Datastore, Cloud Firestore, Cloud Storage або Cloud SQL для збереження даних.

2.2.5 Управління ідентифікацією і доступом

Для управління ідентифікацією і доступом система IAM надає рівень авторизації, який дозволяє здійснювати точне управління доступом для функцій. Наприклад, які дії функції можуть робити з цими ресурсами (доступ лише для читання чи лише для запису), і до яких областей вони мають доступ (конкретна функція чи цілий проект).

AWS Lambda надає можливість створювати свої власні політики IAM та додавати їх до своїх функцій Lambda. Це дозволяє отримати дозвіл для дій AWS Lambda API, користувачів, груп, ролей та ресурсів.

Azure Functions дозволяють керувати політиками функцій за допомогою управління доступом на основі ресурсів. Він підтримується в Subscription і ResourceGroup.

Ролі IAM Google Cloud Functions дозволяють перерахувати permissions, які містяться в кожній ролі. Ролі можуть надаватися користувачам у рамках як цілого проекту, так і в окремих функціях.

2.2.6 Джерела подій і їх типи

Це події користувача, які викликають serverless функцію.

AWS Lambda надає можливість виклику через HTTPS, визначену за допомогою REST API та кінцевою точкою, використовуючи API gateway.

Крім цього, AWS Lambda підтримує безліч інших сервісів AWS, які можна налаштувати як джерело подій, наприклад S3 Storage, SNS, DynamoDB, Cloud Formation, тощо. Lambda function також можна викликати за допомогою AWS SDK, якщо вона має необхідний дозвіл на виклики.

Azure Functions має такі сервіси Microsoft, як Azure Cosmos DB, Timer, Queue Storage, Blob storage Service Bus Queue і Topic, Event Grid і Event Hub, . Також для виклику можна використовувати HTTP та Webhooks (від GitHub) та прив'язки.

Google Cloud Functions підтримує такий ініціатор подій, як HTTP, який потрібно вказати під час створення кінцевої точки функції. Крім цього, Cloud Storage та Cloud Pub/Sub можуть працювати як джерела подій. А також аутентифікація Firebase, Analytics для Firebase та Cloud Firestore.

2.2.7 Оркестрація

Додатки з FaaS архітектурою, в якій функції запускаються, виконуються та закінчуються за лічені мілісекунди, не зберігають стан. Кожна функція повністю незалежна, і дані не можуть бути збережені в контейнері, який видаляється, коли функція виконує своє завдання. Але існує можливість додати стан і оркестрацію у FaaS архітектуру.

AWS реалізує цей підхід за допомогою Step Functions. Цей модуль реєструє стан кожної функції, щоб його можна було використовувати в наступних функціях або для аналізу.

За допомогою Azure Functions є можливість оркеструвати та автоматизувати завдання за допомогою Azure Logic Apps, а також Durable functions. Azure Durable Functions -додають у код абстракції оркестовки робочих процесів. Вони поставляються з декількома шаблонами для поєднання декількох serverless функцій у довготривалі потоки з відстеженням стану.

Наразі Google Cloud Functions не підтримує інтеграцію з Cloud Composer.

2.2.8 Concurrency і час виконання

Concurrency відноситься до паралельної кількості виконань функції, які відбувається у будь-який момент часу. Можливо оцінити кількість паралельних виконань, але ця кількість залежить від типу джерела події. Є можливість оцінити кількість паралельних виконань, але ця кількість залежить від типу джерела події. Крім того, функції автоматично масштабуються на основі

швидкості вхідних запитів, але не всі ресурси в архітектурі можуть це зробити. Отже, паралельність також залежить від підтримки цих ресурсів.

AWS Lambda обмежує загальну кількість одночасних виконань функцій у певному регіоні до 1000. Крім того, максимальний час виконання функції становить 15 хвилин. Concurrency вимірюється на рівні акаунту. AWS Lambda завжди резервує окремий екземпляр для одного виконання. Кожне виконання має свій ексклюзивний пул пам'яті та процесорів. Таким чином продуктивність повністю передбачувана і стабільна.

В даний час функції Azure підтримують кілька функцій одночасно, за умови, що вони знаходяться в одному екземплярі програми, який не має обмежень. Кількість одночасних дій та виконання функцій оркестратора обмежується 10 кратним збільшенням кількості ядер на віртуальній машині. Як і AWS Lambda, також є обмеження на тривалість виконання у 5 хвилин, яке можна збільшити до 10 хвилин у рамках плану споживання. Преміму план надає необмежену тривалість виконання (60хвилин гарантовано). Для функцій, які були ініційовані HTTP, максимальний час очікування відповіді 230сек

Google Cloud Functions немає одночасних обмежень для виклику HTTP. Для інших типів викликів максимальна кількість одночасних запитів на функцію становить 1000. Однак максимальний час виконання, на відміну від інших постачальників послуг, за замовчуванням становить 60 секунд, які можна підвищити до 9 хвилин. Concurrency вимірюється на рівні проекту.

2.2.9 Масштабованість

AWS Lambda підтримує динамічну масштабованість у відповідь на збільшення трафіку. Однак на це поширюється обмеження одночасного виконання на рівні окремого акаунту. Для управління сплеском трафіку AWS Lambda негайно збільшить паралельно викоуччісь функції на заздалегідь визначену кількість, залежно від того, в якому регіоні він виконується. Якщо

немає заздалегідь визначеного обмеження, продовжиться збільшення кількості функцій на 500 на хвилину, поки навантаження не буде задоволене. AWS Lambda кращий за функції Azure для великих робочих навантажень. Крім того, ефекти затримки завантаження менш серйозні у випадку AWS Lambda, ніж функції Microsoft Azure. Оскільки код не вимагає збереження стану, AWS Lambda може створити необхідну кількість копій функцій без довгих процедур розгортання або затримок на налаштування. Принципових обмежень для масштабування функцій не існує. AWS Lambda динамічно розподіляє ресурси таким чином, щоб вони відповідали кількості вхідних подій.

На даний момент Azure Functions доступні на основі двох різних планів. По-перше, план споживання, який автоматично масштабує функцію, де час виконання функції закінчується через налаштований проміжок часу. По-друге, план обслуговування додатків, який запускає функції на виділених віртуальних машинах.

Google Cloud Functions пропонує автоматичне масштабування. Однак фонові функції масштабуються більш поступово, і це залежить від тривалості функцій; більш довші функції будуть масштабуватися трохи повільніше. Крім того, максимальна масштабованість залежить на обмеженнях швидкості.

2.2.10 Ціноутворення

Час виконання є прямо пропорційним середовищу виконання. Наприклад, розглянемо функцією, яка потребує великих обчислень і для виконання потребує декілька мілісекунд. Можна обрати менший обсяг пам'яті, оскільки це буде коштувати дешевше, але треба враховувати той факт, що чим менша машина, тим більше часу потрібно для виконання функції. Співвідношення вартості до обсягу оперативної пам’яті зображено у табл. 2.1.

AWS Lambda має рівень безкоштовного користування, у рамках якого він покриває перші 1 млн запитів функції та 400 000 ГБ-секунд на місяць. Після

цього потрібно буде заплатити $0,20 за 1 мільйон запитів або 0,0000002 доларів США за запит. Крім того, $0,00001667 за кожні використані ГБ-секунди.

Ціна за тривалість залежить від обсягу оперативної пам'яті, виділеної для функції. Є можливість виділити будь-який обсяг пам'яті для функції в діапазоні 128-3008 МБ з кроком 64 МБ.

Таблиця 2.1 - Співвідношеня вартості до обсягу виділеної пам’яті

Обсяг пам’яті (Мб)	Ціна за 100 мс (USD)
128	0.0000002083
512	0.0000008333
1024	0.0000016667
1536	0.0000025000
2048	0.0000033333
3008	0.0000048958

Рівень безкоштовного використання Azure Functions щомісяця покриває 1 мільйон запитів та 400 000 ГБ-секунд. Згодом треба оплатити $ 0,000016 / ГБ-сек і $ 0,20 за 1 мільйон виконання.

Вартість Google Cloud Functions залежить від часу виконання (GB-sec), кількості викликів і кількості ресурсів, які були надані для функції (Ghz-secs). Рівень безкоштовного використання включає перші 2 мільйони запитів і $0,40, якщо вийшли за ці рамки. Час обчислення вимірюється у двох одиницях: GB-secs і GHz-secs. Більше того, мережеві вихідні дані стягується фіксована плата в розмірі $0,12/Gb, причому перші 5 Gb надаються безкоштовно. Ця модель ціноутворення дещо відрізняється від решти провайдерів.

2.2.11 Продуктивність

- Concurrency і масштабованість

Для AWS 3328 МБ — це максимальний сукупний об’єм пам’яті, який можна розподілити між усіма екземплярами функцій на будь-якій віртуальній машині у AWS Lambda. AWS Lambda намагається розмістити новий екземпляр функції поверх існуючої віртуальної машини, щоб збільшити коефіцієнт використання пам'яті.

Azure Functions стверджує, що масштабує 200 інстансів для однієї функції Node.js, але на практиці це не вдається зробити.

За допомогою Google Cloud Functions є можливість запустити одночасно лише половину очікуваної кількості екземплярів, навіть для низького рівня паралельності. Решта запитів будуть в черзі.

 Холодний старт

AWS Lambda не має великої різниці між запуском існуючого та нового екземпляра при холодному старті. Фактично середній час холодного старту, становив лише 39 мс. Можливо передбачити стабільність і узгодженість, оскільки AWS Lambda резервує окрему пам'ять для кожного виконання, отже, існує унікальний пул циклів пам'яті для окремого екземпляра. Також є можливість активувати Provisioned Concurrency для кращого контролю продуктивності FaaS додатків. Коли ця функціональність активована, то функції знаходяться в ініціалізованому стані і готові до швидкого реагування.

Запуск екземплярів Azure займає більше часу, незважаючи на те, що їм навіть виділено 1,5 ГБ пам’яті. Середня затримку холодного старту приблизно 3640 мс. Microsoft Azure Functions дозволяють одночасно виконувати кілька одночасних розширень. Це призводить до зменшення загального часу продуктивності для інстансів, що потребують ресурсів. Premium plan дозволяє інстанси тримати завжди розігрітими, щоб запобігти холодному старту.

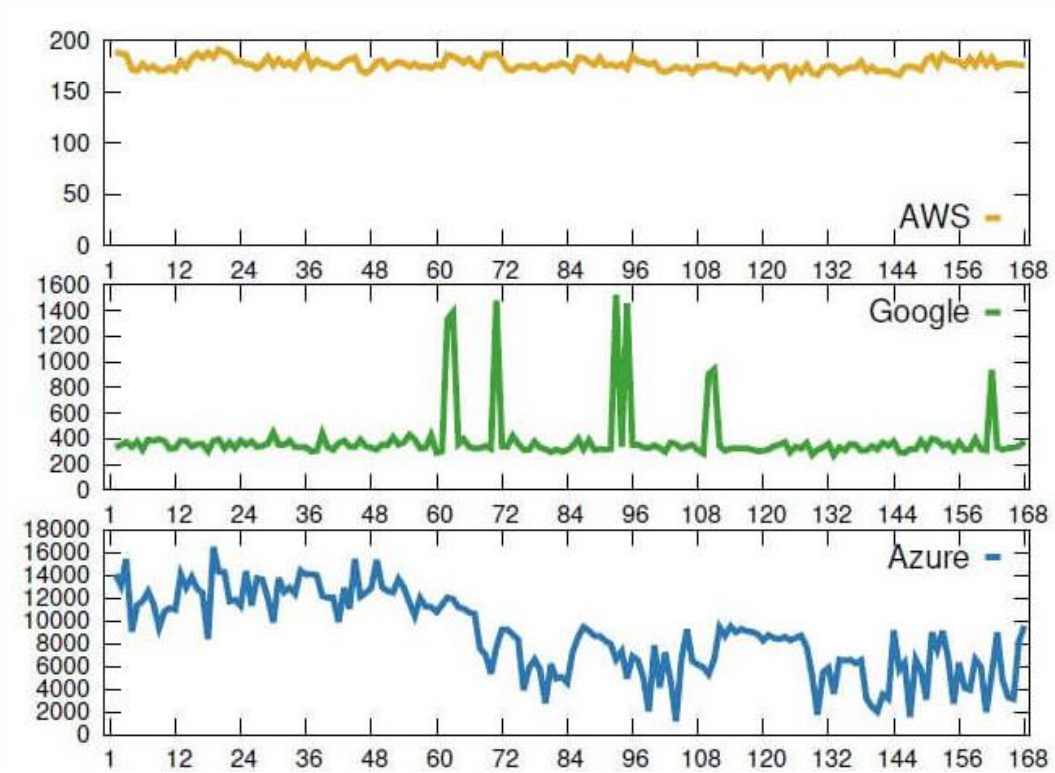


Рисунок 2.6 — Затримка холодного старту лямбда-функції

У Google Cloud Functions середня затримка холодного старту становить від 110 мс до 493 мс. Пам'ять розподіляється пропорційно процесору, але в Google розмір пам'яті більше впливає на затримку холодного запуску, ніж у AWS [13]. Як бачимо на рис. 2.6 FaaS сервіс від AWS має найкращі показники.

Максимальний час простою інстансу

Це максимальний час, протягом якого інстанс може залишатися в бездіяльності перед завершенням своєї роботи.

Для AWS екземпляр може залишатися активним протягом 27 хвилин. Примітно, що у 80% випадків інстанси закриваються через 26 хвилин. За допомогою функцій Azure не виявлено постійного простою. Для Google Cloud Functions час простою становить приблизно 120 хвилин. Навіть через 120 хвилин у 18% випадків інстанси залишаються активними.

Візьмемо в якості критеріїв порівняння розглянуті вище властивості сервісів і зробимо зведену таблицю порівняння послуг FaaS провайдерів (таблиця 2.2).

Таблиця 2.2 — Порівняння послуг FaaS провайдерів

Характеристика	AWS Lambda	Microsoft Azure Function	Google Cloud Function
Підтримувані мови	Node.js, Python, Java (Java8 compatible), C#, F#, PowerShell, Go і Ruby	для Node.js,, Python, Java, .NET Core та PowerShell Core	Node.js, Python, Java11 і Go
Розгортання	Zip, Serverless	DropBox, Visual Studio Team Services, OneDrive, GitHub, Zip	CLI, Zip, вбудований веб-редактор, Cloud Storage або Cloud Storage Repository
Джерела подій і їх типи	Api gateway, S3 Storage, Kenesis, SNS, CloudWatch Logs, Simple Email Service, DynamoDB, Cloud Formation, AWS SDK і т.д.	HTTP, Azure Cosmos DB, Timer, Queue Storage, Blob storage Service Bus Queue і Topic, Event Grid і Event Hub,	HTTP, Cloud Storage, Cloud Pub/Sub, Firebase

Таблиця 2.2 — Порівняння послуг FaaS провайдерів (продовження)

Характеристика	AWS Lambda	Microsoft Azure Function	Google Cloud Function
Моніторинг і логування	CloudWatch Log і CloudWatch	Application Insights	Stackdriver
Оркестрація	Step functions	Durable functions	Немає підтримки
Паралельні обчислення	1000 паралельних обчислень для акаунту для регіону	Необмежено(але залежить від тригерів подій)	Необмежено для HTTP викликів, 1000 для інших подій
Час виконання	15 хвилин	5хвилин, але може бути збільшено до 10хвилин. Преміум план надає необмежений час(60хвилин гарантовано)	1хвилина за замовчуванням, може бути збільшено до 9хвилин
Масштабованість	Автоматична	Автоматична або за метриками	Автоматична
Ціноутворення	1 млн запитів та 400000 ГБ-сек безкоштовно, $0,20 за 1 млн запитів $0,00001667 за кожні використані ГБ-сек	1 млн запитів та 400 000 ГБ-секунд безкоштовно. $0,000016/ ГБ-сек і $0,20 за 1 млн виконання.	2 млн безкоштовно, $0,40 за 1млн. мережеві вихідні дані - $0,12/Gb, 5 Gb безкоштовно

Таблиця 2.2 — Порівняння послуг FaaS провайдерів (продовження)

Характеристика	AWS Lambda	Microsoft Azure Function	Google Cloud Function
Максимальна кількість функцій	Необмежено	Необмежено	1000 функцій на проект
Управління залежностями	Deployment packages (Jenkins, Maven)	npm, NuGet	npm
Постійне сховище даних	S3, DynamoDB	Azure Blob Storage, Azure Table Storage, CosmosDB	Cloud Datastore, Cloud Firestore, Cloud Storage або Cloud SQL
IAM	IAM ролі	IAM ролі	IAM ролі

2.3 Висновки

У даному розділі були проаналізовані три найпопулярніші обчислювальні хмарні FaaS сервіси, такі як AWS Lambda, Azure Functions і Google Cloud Functions. Кожний сервіс має як свої переваги, так і недоліки, і відповідно кожний хмарний FaaS сервіс має свою область використання. AWS домінує у розвитку хмарних обчислень, маючи великий спектр різноманітних хмарних сервісів, випустив сервіс Lambda ще у 2014 році. Завдяки цьому AWS Lambda має найкращу взаємодію зі сторонніми BaaS сервісами від AWS, що значно виділяє його у порівнянні з рішеннями від Microsoft і Google. Широка взаємодія зі сторонніми сервісами надає безліч ініціаторів подій, які запускають лямбда-функцію. У свою чергу це надає змогу використовувати AWS Lambda майже для всіх типів додатків, починаючи від побудови API, написанні різноманітних Cron задач і стрімінгу даних у реальному часі, до побудови serverless чат-ботів і бекенду Інтернету речей. Сервіс CloudWatch, який займається логування у інфраструктурі AWS, логує кожний виклик лямбда-функції, що робить процес логування і відладки надзвичайно зручними і простими. Процес логування буде реалізований практично у 4 розділі. Процес розробки є надзвичайно зручним і надає можливість писати код як у консолі розробника AWS, так і за допомогою Sereverless фреймворку. Також AWS Lambda має найкращий час “холодного старту” лямбда-функцій у контейнерах, при цьому надаючи шляхи прискорення цього процесу за допомогою сторонніх сервісів, таких як Provisioned Concurrency. Управління доступом і ідентифікацією реалізований за допомогою політик сервісу IAM, що дозволяє легко налаштовувати ролі і правила. Політики IAM дозволяють контролювати доступ до ресурсів найпростішим засобом, що значно відрізняє AWS Lambda від Azure Functions, у якому управління доступом реалізоване на основі ресурсів. Azure Functions чудово підходять для розробки додатків на платформі

.NET, оскільки мають повну інтеграцію з усіма сервісами, які підтримують цю платформу. Google Cloud Functions є наймолодшим FaaS рішенням і на жаль, не має повного функціоналу, яким володіють сервіси від AWS і Microsoft.

Для того, щоб розробляти безсерверні додатки найбільш ефективно, мати найменший час затримки відгуку системи і використовувати ресурси з найменшими витратами, існує багато різноманітних архітектурних патернів, які нададуть найкращі рішення у побудові додатку. На основі порівняння FaaS сервісів для подальшого дослідження було обрано AWS Lambda Functions.

3 ШАБЛОНИ АРХІТЕКТУРИ FAAS ДОДАТКІВ В AWS

Serverless технології та архітектури можуть використовуватися для побудови цілих систем, створення ізольованих компонентів або реалізації конкретних детальних задач. Сфера використання FaaS дизайну велика, і одна з його переваг полягає в тому, що може використовуватися як для малих, так і для великих задач. Варто пам’ятати, що serverless - це не лише запуск коду в такому обчислювальному сервісі, як Lambda AWS. Йдеться також про використання сторонніх служб та API, щоб скоротити обсяг роботи, яку розробник повинен виконувати. Архітектурні патерни - це потужний спосіб використовувати найкращі практики, надійні рішення та загальні архітектурні бачення у побудові додатків.

На основі зробленого аналізу провайдерів у попередньому розділі для подальшого дослідження патернів serverless додатків було обрано AWS провайдер, для якого і будуть розглянуті архітектурні шаблони. Одним з найпопулярніших випадків використання serverless є створення RESTful API. У AWS лямбда-функції разом з API-gateway забезпечують зручний спосіб створення масштабованих ендпойнтів, які обробляють дані в режимі реального часу. Serverless backend-частина додатків є привабливою тим, що усуває більшу частину управління інфраструктурою, має деталізований та передбачуваний біліннг (особливо коли використовується безсерверна обчислювальна служба, така як Lambda), і може добре масштабуватися для забезпечення нерівномірного трафіку.

Комунікація в реальному часі в веб та мобільних додатках стає все більш поширенішою, і WebSockets - це чудовий спосіб реалізувати цю можливість. За допомогою WebSockets можна реалізувати функціональність чату, багатокористувацькі онлайн-ігри, тощо. API gateway має можливість

ініціалізувати та підтримувати з’єднання WebSocket з клієнтами, а потім запускати лямбда-функцію тільки тоді, коли повідомлення надходить від клієнта. Це дозволяє відповідати на запити без серверу. API Gateway робить усі налаштування за розробника.

За допомогою AWS Lambda можна створювати serverless внутрішні системи для обробки запитів API, пов'язаних з Інтернетом, мобільними пристроями, Інтернетом речей (IoT), а також сторонніх запитів API. Сервіси AWS Lambda і Amazon Kinesis дозволяють обробляти потокові дані для відстеження активності додатків, обробки послідовностей операцій, аналізу відвідуваності, очищення даних, створення метрик, фільтрації журналів, індексації, аналізу соціальних мереж, телеметрії і обліку даних пристроїв Інтернету речей в режимі реального часу.

Ці шаблони проектування є надзвичайно корисними для створення надійних, масштабованих та безпечних serverless додатків у хмарі. У даному розділі буде проведений аналіз існуючих патернів та способів їх використання.

3.1 Шаблон Simple Web Service

Simple Web Service є найбільш стандартним варіантом використання AWS Lambda як серверної служби. Він представляє логічний або доменний рівень багатошарової архітектури. Simple Web Service є найпопулярнішим варіантом використання serverless для побудови API, будь то RESTful API або GraphQL.

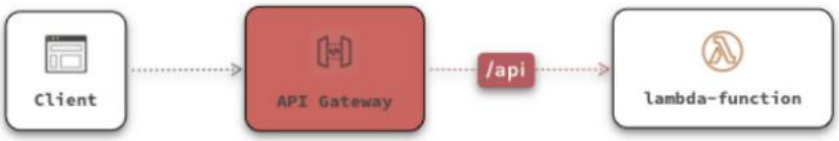


Рисунок 3.1 - Публічне API, яке надається через HTTP(s)

Для загальнодоступних API сервіс API gateway надає лямбда-функції через HTTPs, як зображено на рис. 3.1. Крім того API gateway може обробляти валідацію вхідних запитів, авторизацію, аутентифікацію, маршрутизацію та керування версіями.

Для найпростішої реалізації даного патерну, зображеного на рис. 3.2 можуть використовуватись такі компоненти, як AWS API Gateway, база даних DynamoDB, сховище S3 і Amazon Cognito User Pool.

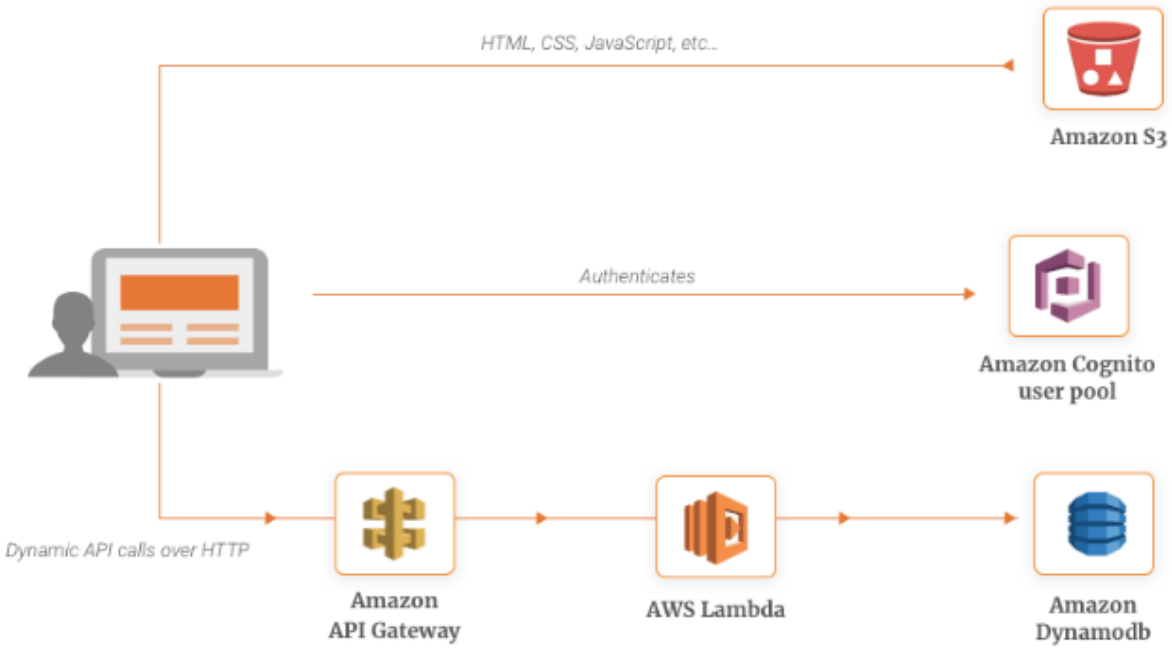


Рисунок 3.2 - Структура патерну Simple Web Service

Ці компоненти виконують наступні функції:

- JavaScript у браузері обмінюється даними з зовнішнім серверним API, який побудований за допомогою API Gateway та AWS Lambda.
- DynamoDB - це база даних NoSQL, яка використовується для зберігання даних за допомогою лямбда-функцій API інтерфейсу.
- Amazon S3 використовується для розміщення статичного контенту веб-сайту, такого як HTML, медіа-файли, CSS, JavaScript, який виконує роль інтерфейсу в браузері користувача.

- Amazon Cognito використовується для аутентифікації та управління користувачами за допомогою захищеного внутрішнього інтерфейсу API.

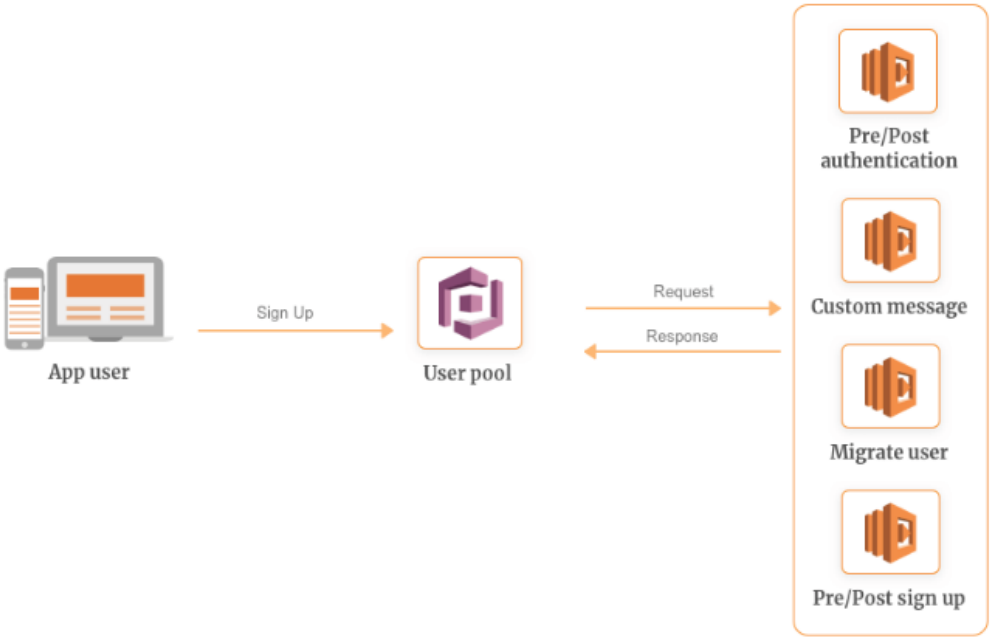


Рисунок 3.3 - Компоненти AWS Cognito

Amazon Cognito, зображений на рис. 3.3 при використанні з AWS Lambda надає можливість додавати перехоплювачі перед і після входу в систему для виконання власної логіки. Після створення лямбда-функції AWS є можливість активувати її на основі різних операцій пулу користувачів, таких як реєстрація користувача, підтвердження користувача, вхід в систему, тощо. Також є можливість процедуру аутентифікації налаштовувати за власним бажанням, наприклад, додати двухфакторну аутентифікацію

Нижче наведений список ініціюючих подій, на які можливо зробити хук за допомогою лямбда функції:

- Реєстрація, підтвердження реєстрації та вхід до системи
- До і після аутентифікації
- Користувацька аутентифікація
- Попередня генерація JWT токену

- Користувацьке повідомлення

3.2 Шаблон The Internal API

Internal API патерн — це веб-служба без зовнішнього інтерфейсу API gateway. Якщо створюється мікросервіс, до якого потрібно отримати доступ лише з інфраструктури AWS, то треба використовувати AWS SDK і отримати безпосередній доступ до HTTP API Lambda. HTTP-виклики всередині мікросервісів є стандартною (і часто необхідною) практикою.

Для внутрішніх API клієнти викликають лямбда-функції безпосередньо з клієнтської програми за допомогою AWS SDK.

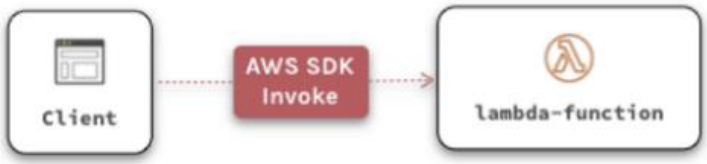


Рисунок 3.4 — Структура шаблону Internal API

AWS Lambda масштабується автоматично і має можливість обробляти коливання навантажень. Час виконання обмежений максимумом, дозволеним шлюзом API gateway (29 секунд) або обмеженням AWS Lambda у 15 хвилин, якщо його викликає SDK.

3.3 Шаблон The Scalable Webhook або Queue-based load leveling

Навантаження може бути непередбачуваним і деякі служби не можуть масштабуватися при періодичному великому навантаженні. Це може спричинити перевантаження, перезаповнення сервісів і їх вихід з ладу.

Створення черги між службами, яка діє як буфер, може вирішити ці проблеми. Повідомлення зберігаються і це дозволяє споживачам обробляти

навантаження у своєму власному темпі. Черга відокремлює завдання від сервісів, створюючи буфер, який вміщує запити на менш масштабовані серверні частини або сторонні служби. Незалежно від обсягу запитів навантаження на обробку визначається споживачами. Низький рівень паралелізму і розмір пакету можуть контролювати робоче навантаження.

Розглянемо приклад створення веб-хуку. У цьому випадку трафік часто може бути непередбачуваним. Це нормальна ситуація для лямбда-функції , але якщо також використовується "менш масштабований" серверний сервіс, такий як RDS (розподілена реляційна база від AWS), то можуть виникнути проблеми. Оскільки Lambda підтримує тригери SQS (сервіс AWS, який приймає черги повідомлень для збереження), то можливо регулювати робоче навантаження, ставлячи запити в чергу, а потім використовуючи регульовану (з низьким рівнем cuncurrency) функцію Lambda для роботи через дану чергу.

Тригери SQS для лямбда-функцій працюють коректно з регулюванням або throttling, тому більше не має потреби керувати власною політикою повторної обробки запитів.

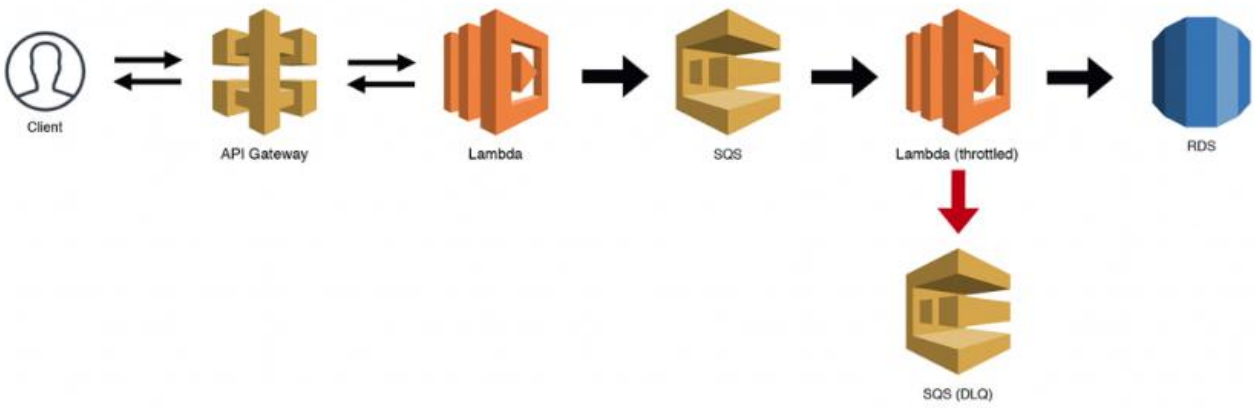


Рисунок 3.5 - Структура шаблону The Scalable Webhook

При використанні тригерів SQS з регульованими лямбда-функціями (throttling) необхідно обов’язково обробляти невірні повідомлення за допомогою Dead Letter Queues (DLQs). Але траба пам’ятати, що цей патерн не

приносить користі, коли сервіси очікують синхронну відповідь або мінімальну затримку.

У середовищі, де багато обробки даних, повідомлень та запитів, треба мінімізувати кількість функцій, які безпосередньо залежать від інших функцій, і використовувати замість цього шаблон “The Scalable Webhook”. Робоче навантаження з інтенсивним читанням не отримує вигоди від вирівнювання навантаження на основі черги головним чином тому, що треба отримати доступ до нижчого за рівнем ресурсу для отримання даних, необхідних споживачу. Найкраще цей шаблон підходить для ендпойнтів із інтенсивним записом.

3.4 Шаблон The State Machine

Існують певні задачі, коли serverless архітектура повинна забезпечити оркестрацію. AWS step functions надають найкращий спосіб організації оркестрації у безсерверних дадатках AWS. Кінцеві автомати підходять для координації декількох завдань і забезпечення їх належного виконання шляхом реалізації повторних спроб, таймерів очікування та відкатів. Однак вони є виключно асинхронними, що означає, що неможливо дочекатися результату крокової функції, а потім відповісти на синхронний запит.

AWS рекомендує використовувати step functions для оркестрації цілих робочих процесів, тобто координації декількох мікросервісів. Він не працює для сервісів, які повинні надавати синхронну відповідь клієнтам. Є можливість інкапсулювати крокові функції в мікросервісі, зменшуючи складність коду та додаючи відмовостійкість, але при цьому сервіси не зв’язуються.

Рисунок 3.6 — Шаблон State Machine

3.5 Шаблон The Router

Паттерн “State machine” є потужним, оскільки він надає прості інструменти для управління складністю, паралелізмом, обробкою помилок, тощо. Однак крокові функції не безкоштовні, і при використанні для всіх випадків призведе до величезних рахунків. Для менш складних оркестрацій, де переходи між станами не грають такої важливої ролі, можна обробляти їх, використовуючи шаблон Router.

У наведеному нижче прикладі асинхронний виклик лчмбда-функції визначає, який тип задачі слід використовувати для обробки запиту. По суті, це оператор “switch”, який може також додати деякий додатковий контекст та додаткові дані за необхідністю. Основна лямбда-функція викликає лише одне з трьох можливих задач. Асинхронні лямбди повинні мати DLQ, щоб ловити невдалі виклики для повторів.

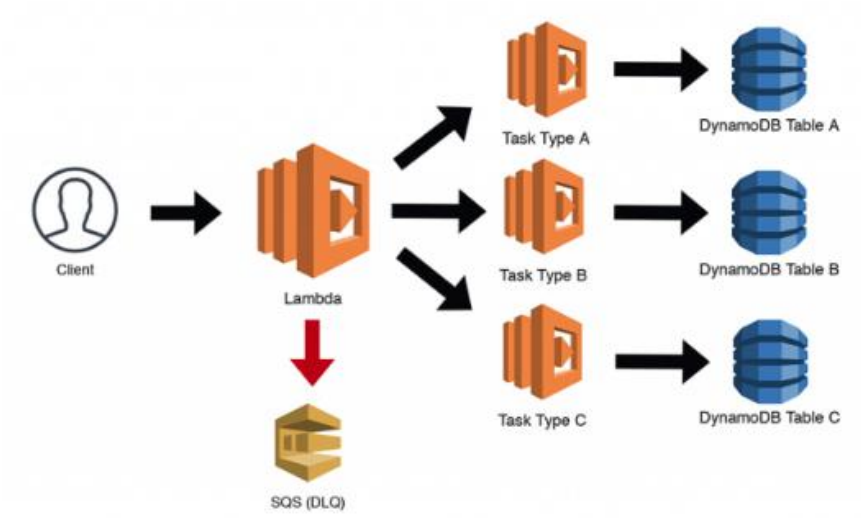


Рисунок 3.7 - Шаблон Router

3.6 Шаблон Aggregator

Накладні витрати на зв’язок між різними сервісами є загальною проблемою. Додаток повинен взаємодіяти з багатьма меншими сервісами, які мають велику кількість міжсервісних викликів.

Патерн використовує API gateway для об’єднання кількох окремих запитів в один запит. Цей шаблон корисний, коли клієнт повинен здійснити кілька викликів до різних backend підсистем для виконання операції.

Сервіс централізує запити клієнтів, щоб зменшити вплив комунікаційних накладних витрат. Він розділяє та робить запити до нижчерівневних сервісів, збирає та зберігає відповіді по мірі надходження, агрегує їх та повертає клієнту як єдину відповідь.

Лямбда-функція на схемі нижче робить три синхронні виклики на три окремі мікросервіси. Вважається, що кожний мікросервіс буде використовувати щось на зразок паттерну “Internal API” і повертатиме дані клієнту. Наведені нижче мікросервіси також можуть бути зовнішніми службами, як сторонні API. Потім лямбда-функція агрегує всі відповіді та повертає комбіновану відповідь клієнту з іншого боку API-gateway.

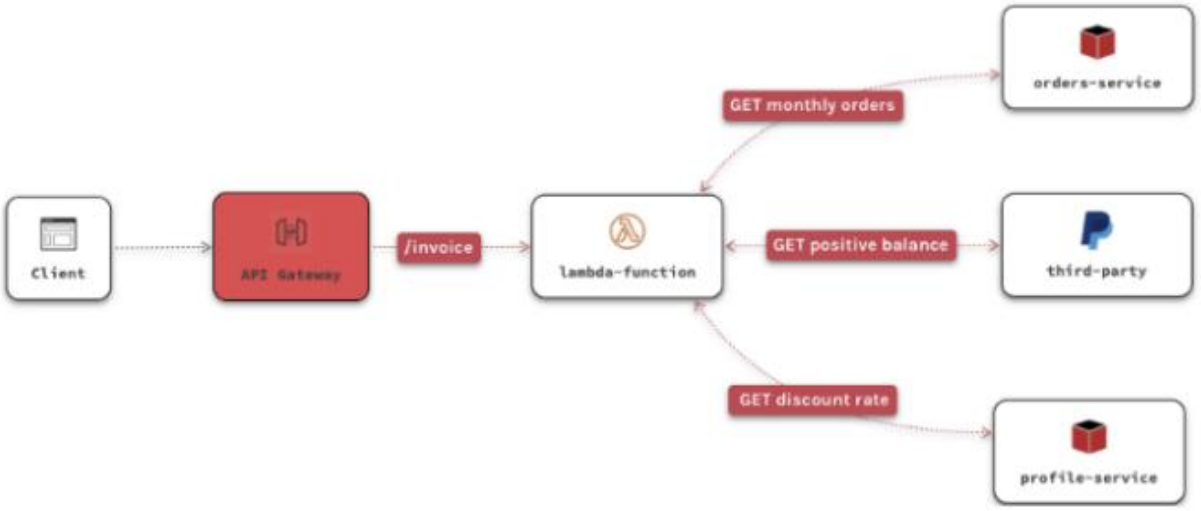


Рисунок 3.8 - Структура шаблону Aggregator

Також натомість мікросервісів можуть бути інші лямбда-функції. Microsoft називає цей шаблон “Gateway Aggregation”. Він може бути реалізований як служба з певною бізнес-логікою, яка здатна кешувати відповіді та знає, що робити у разі збою сервісів.

3.7 Шаблон Publisher/Subscriber

Сервіси у додатку зазвичай не можуть залишатися ізольованими. Платформа розростається, і кількість сервісів швидко збільшується. Система потребує сервіси для взаємодії, але при цьому взаємозалежності не повинні створюватись. Патерн “Publisher/Subscriber”, структура якого зображена на рис. 3.9, постачає повідомлення для багатьох зацікавлених споживачів асинхронно, не прив’язуючи сервіси-відправники до сервісів-споживачів. Події постачаються через канал у вигляді повідомлень. Зацікавлені сервіси-споживачі слухають події, підписуючись на ці канали. Асинхронний обмін повідомленнями - це ефективний спосіб відокремити сервісів-відправників від сервісів-споживачів та уникнути блокування сервіса-відправника, щоб дочекатися відповіді. Він дозволяє сервісам повідомляти про події інші сервіси-споживачі не створюючи взаємозалежності.

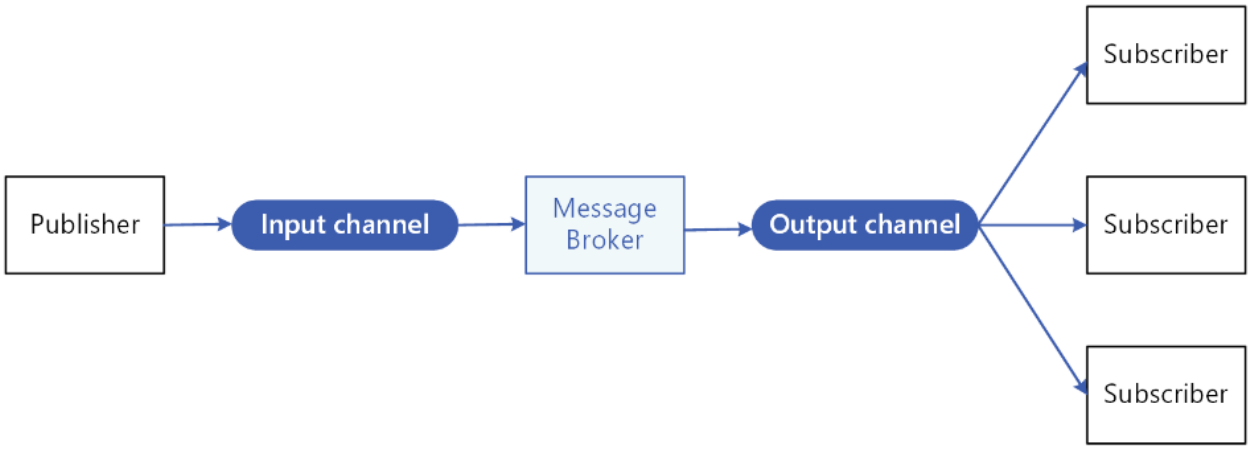


Рисунок 3.9 — Структура шаблону Publisher/Subscriber

Патерн надає такі переваги:

1. Розділяє підсистеми, які обмінюються даними. Підсистемами можна керувати незалежно.
2. Підвищує масштабованість та покращує швидкість відгуку відправника. Відправник може швидко відправити одне повідомлення на вхідний канал, а потім продовжити своє подальше виконання.
3. Підвищує надійність. Асинхронний обмін повідомленнями допомагає програмам продовжувати безперебійно працювати при підвищених навантаженнях та ефективніше обробляти періодичні збої.
4. Забезпечує простішу інтеграцію між системами, що використовують різні платформи, мови програмування або комунікаційні протоколи, а також між локальними системами та програмами, що працюють у хмарі.

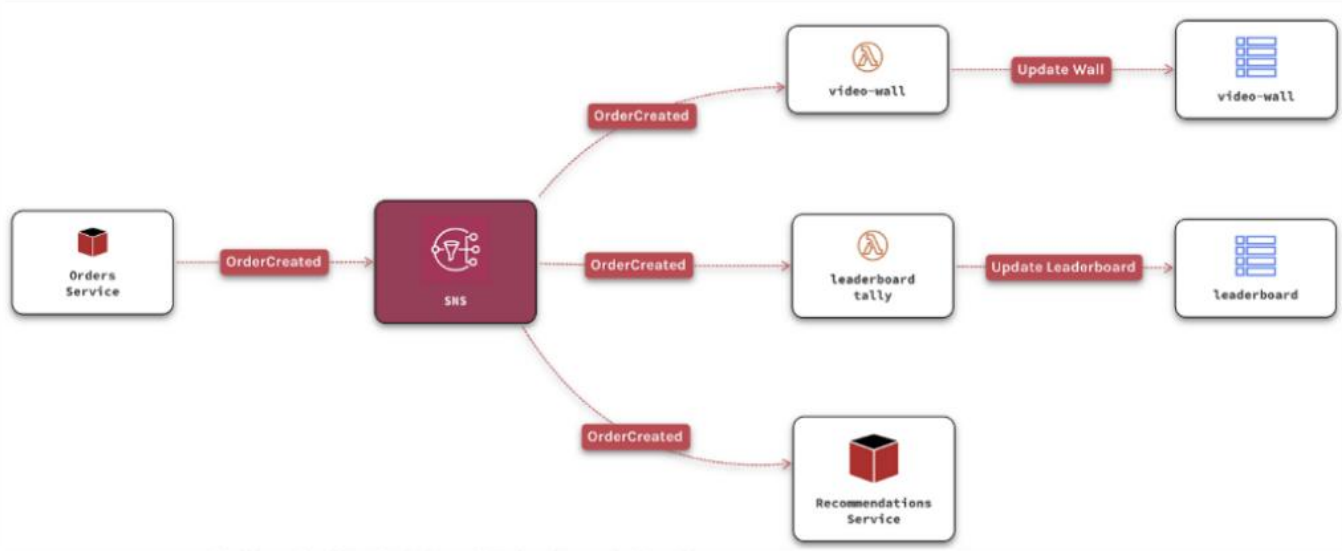


Рисунок 3.10 — Структура додатку, який використовує шаблон Pub/Sub

На рисунку 3.10 сервіс Orders опублікує подію, коли замовлення створюється в мобільному додатку. Лямбда-функція VideoWall прослуховує події OrderCreate. Вона прийме це замовлення, розіб'є елементи замовлення та оновить сторінку. Лямбда-функція leaderboard отримає ту саму подію та оновить підсумки. Сервіс рекомендацій буде відстежувати зроблені закази. Сервіси не пов'язані між собою. Вони працюють разом, спостерігаючи і реагуючи на події. Коли в систему додадуться нові сервіси та функції, то вони можуть підписатися, отримувати події та розроблятися незалежно.

Паттерн Publisher/Subscriber чудово підходить для архітектур, керованих подіями. Існує безліч різних варіантів обміну повідомленнями: SNS, Kinesis, Kafka, Pulsar тощо. Ці служби обміну повідомленнями займаються побудовою інфраструктурної частини pub/sub, але враховуючи асинхронний характер обміну повідомленнями, впорядкування повідомлень, дублювання, термін дії, ідемпотентність та послідовність повинні бути враховані при реалізації. Цей патерн не можна застосовувати, коли додаток потребує взаємодії в реальному часі.

Цікавим прикладом використання цього патерну є отримання інфраструктурних сповіщень у вигляді повідомлення Slack. Щоразу, коли

спрацьовує сигнал сервісу CloudWatch, він надсилає повідомлення до SNS topic. Отримавши повідомлення, SNS topic викликає лямбда-функцію, яка викличе API Slack для надсилання повідомлення до каналу Slack.

3.8 Шаблон Strangler

З часом інструменти розробки, технології хостингу та навіть системні архітектури, на яких системи були побудовані, можуть ставати дедалі все більш застарілими. У міру додавання нових функцій та функціональних можливостей складність цих програм може різко зрости, ускладнюючи їх підтримку або додавання нових функцій.

Повна заміна складної системи може бути непростою задачею. Часто потребується поступовий перехід, який зображено на рис. 3.11, на нову систему, зберігаючи при цьому стару систему для обробки функцій, які ще не перенесені. Однак запуск двох окремих версій програми означає, що клієнти повинні знати, де розташовані певні функції. Кожного разу при міграції функцій або сервісу, клієнти потрібні оновлюватись, щоб вони вказували на нове місце.

Рішенням цієї проблеми може бути поступова заміна певних елементів функціональності новими функціями та сервісами. Треба створити фасад, який перехоплює запити, що надходять у застарілу backend-систему. Фасад направляє ці запити або до застарілої програми, або до нових функцій/сервісів. Існуючі функції можна переносити на нову систему поступово, а споживачі можуть продовжувати використовувати той самий інтерфейс, не знаючи про те, що відбулася якась міграція.

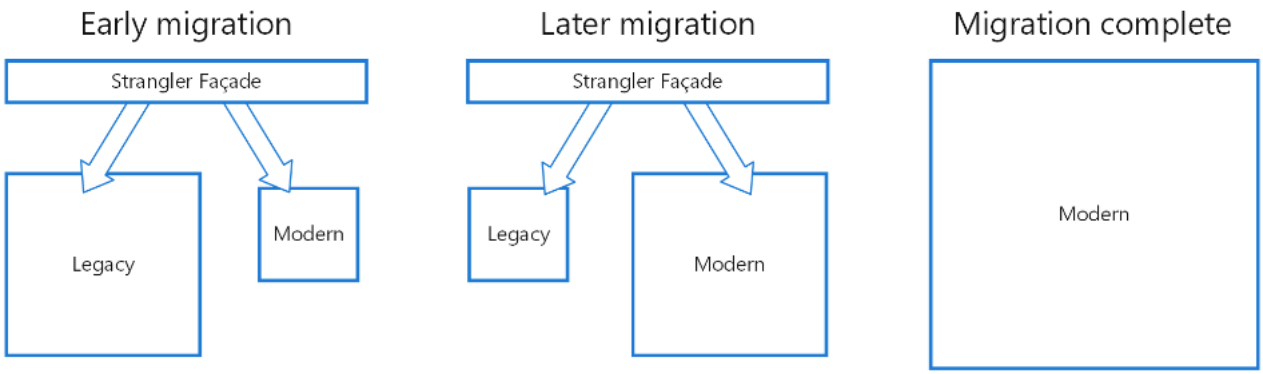


Рисунок 3.11 - Перехід з legacy коду до нової кодової бази

Цей патерн допомагає мінімізувати ризик міграції та розподілити зусилля з розробки і часу. Розробники продовжуватимуть створювати нові функції та переносити функції, які вже існують у нові сервіси. Коли всі необхідні функціональні можливості будуть переписані із застарілої монолітної системи, то її можна буде вивести з експлуатації.

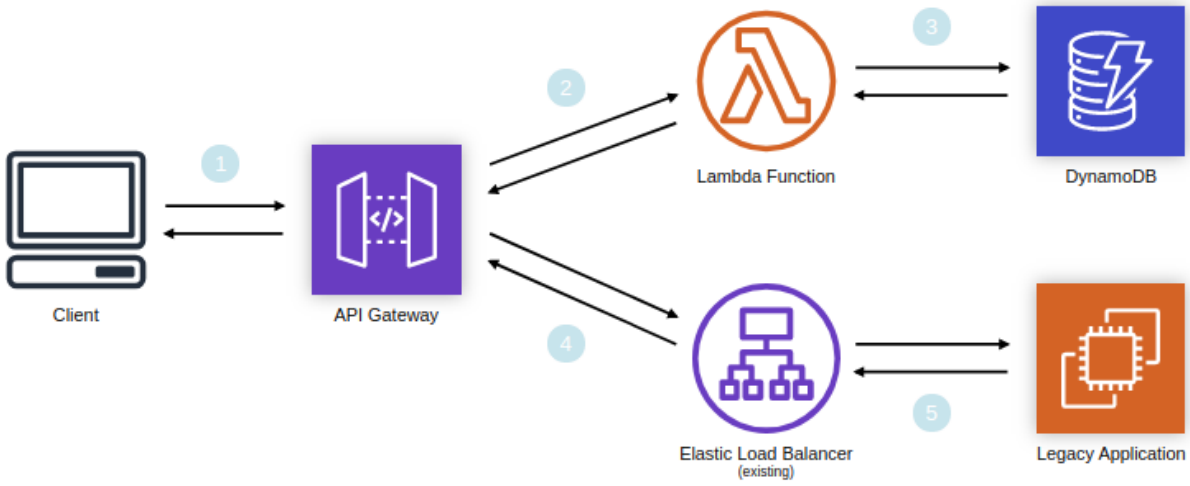


Рисунок 3.12 — Патерн Strangler

Зазвичай можна створити кастомний “Strangler facade” для маршрутизації наших запитів, але API-gateway може це зробити автоматично, використовуючи “AWS Service Integrations” та “HTTP Integrations”. Наприклад, існуючий API (з інтерфейсом Elastic Load Balancer) може маршрутизувати через

API-gateway за допомогою інтеграції “HTTP”. Існує можливість використовувати усі запити за замовчуванням для застарілого API, а потім направляти конкретні маршрути до безсерверного сервісу у міру додавання.

3.9 Шаблон Pipes and filters

Додаток може виконувати різноманітні завдання різної складності з інформацією, яку він обробляє. Простий, але негнучкий підхід до реалізації додатку полягає у виконанні цієї обробки як монолітного модуля. Однак такий підхід, ймовірно, зменшить можливості для рефакторингу коду, його оптимізації або повторного використання, якщо частини тієї самої обробки потрібні в іншому місці додатку. Малюнок (рис. 3.12) ілюструє проблеми з обробкою даних з використанням монолітного підходу. Додаток отримує та обробляє дані з двох джерел. Дані з кожного джерела обробляються окремим модулем, який виконує ряд завдань для перетворення цих даних, перед передачею результату бізнес-логіці програми.

Деякі задачі, які виконують монолітні модулі, функціонально дуже схожі, але модулі були розроблені окремо. Код, який імплементує задачу, тісно пов’язаний з модулем і не дає можливості повторного використання або масштабованості. Однак завдання обробки, що виконуються кожним модулем, або вимоги до розгортання кожного завдання можуть змінюватися в міру оновлення бізнес-вимог. Деякі задачі можуть вимагати інтенсивні обчислювальні ресурси і можуть отримати вигоду від роботи на потужному обладнанні, тоді як інші можуть не вимагати цього. Крім того, в майбутньому може знадобитися додаткова обробка або зміна порядку обробки завдань,. Потрібне рішення, яке вирішує ці проблеми та збільшує можливості повторного використання коду.

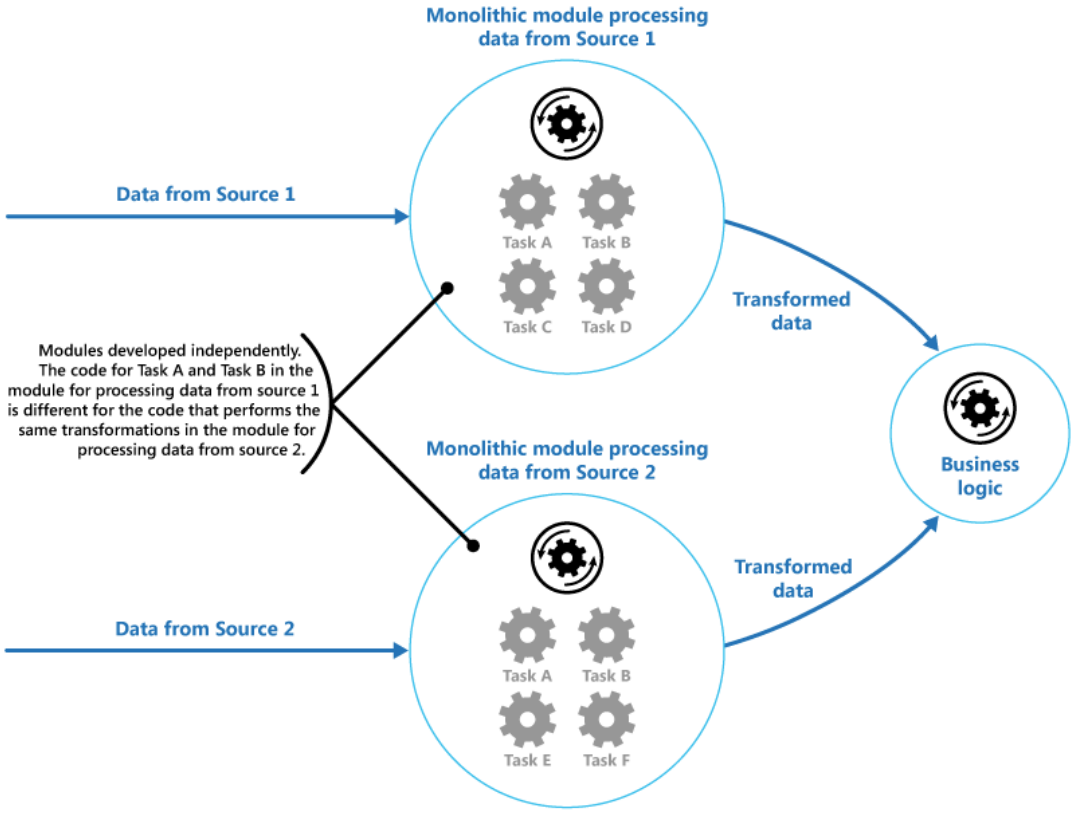


Рисунок 3.12 — Обробка даних у монолітній архітектурі

Для вирішення цієї проблеми потрібно розбити обробку, необхідну для кожного потоку, на набір окремих компонентів (або фільтрів), кожен з яких виконує одне завдання. Стандартизуючи формат даних, які отримує та надсилає кожен компонент, ці фільтри можна об'єднати в конвеєр або pipeline. Це допомагає уникнути дублювання коду та полегшує видалення, заміну або інтеграцію додаткових компонентів, якщо вимоги до обробки змінюються. На наступному малюнку (рис. 3.13) показано рішення, реалізоване за допомогою peipes та filters.

Час, необхідний для обробки одного запиту, залежить від швидкості найповільнішого фільтра в конвеєрі. Один або кілька фільтрів можуть бути вузьким місцем у системі, особливо якщо у потоці з певного джерела даних з’являється велика кількість запитів. Ключова перевага конвеєрної структури полягає в тому, що вона забезпечує можливості для паралельного виконання

екземплярів повільних фільтрів, що дозволяє системі розподіляти навантаження та покращувати пропускну здатність.

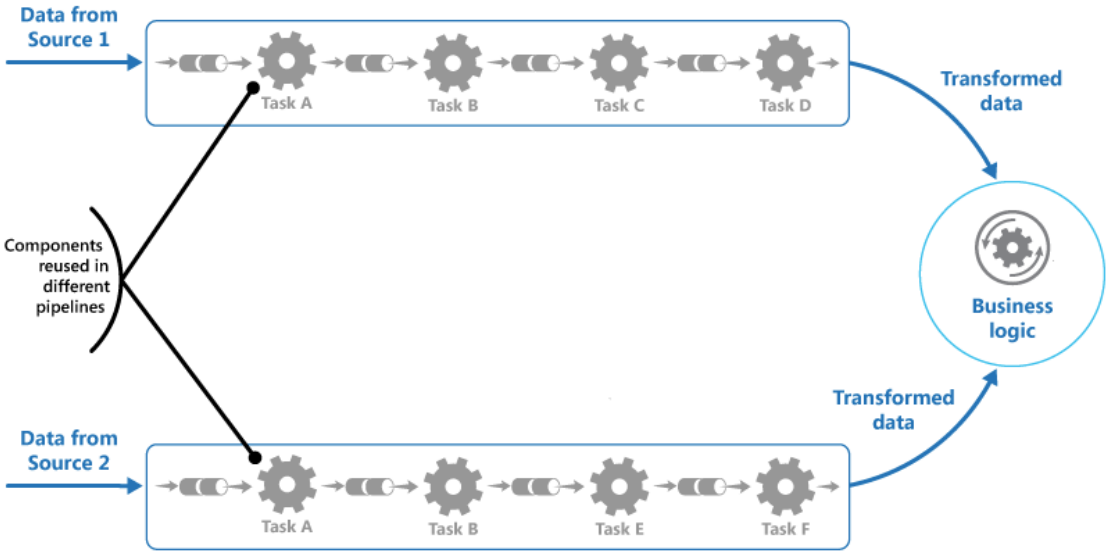


Рисунок 3.13 — Використання шаблону pipes and filters

Фільтри, які складають конвеєр, можуть працювати на різних машинах, що дозволяє їх масштабувати незалежно. Фільтр, який потребує більше обчислювальних ресурсів, може працювати на високопродуктивному обладнанні, тоді як інші менш вимогливі фільтри можуть розміщуватися на менш дорогих інстансах. Фільтри навіть не повинні знаходитися в одному центрі обробки даних або географічному розташуванні, що дозволяє кожному елементу конвеєра працювати в середовищі, близькому до необхідних ресурсів. На рис. 3.14 наведено приклад, застосований до конвеєру для даних із джерела 1.

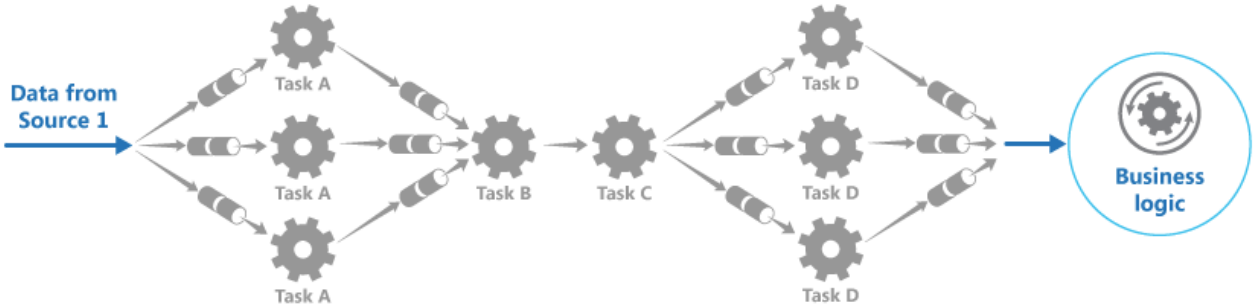


Рисунок 3.14 — Використання шаблону pipes and filters для джерела 1

Якщо вхід і вихід фільтра структуровані як потік, можна виконати обробку для кожного фільтра паралельно. Перший фільтр у конвеєрі може розпочати свою роботу та вивести свої результати, які передаються безпосередньо наступному фільтру в послідовності, перш ніж перший фільтр завершить свою роботу.

Ще однією перевагою є відмовостійкість, яку може надати ця модель. Якщо фільтр виходить з ладу або машина, на якій він працює, недоступна, конвеєр може перепланувати роботу, яку виконував фільтр, і направити цю роботу на інший екземпляр компонента. Відмова одного фільтра не обов'язково призводить до виходу з ладу всього трубопроводу. Використання шаблону «pipes and filters» разом із шаблоном компенсаційної транзакції є альтернативним підходом до реалізації розподілених транзакцій. Розподілена транзакція може бути розбита на окремі компенсовані завдання, кожне з яких може бути реалізована за допомогою фільтра, який також реалізує шаблон компенсаційної транзакції. Фільтри в конвеєрі можуть бути реалізовані як окремі розміщені завдання, що працюють близько до даних, які вони підтримують.

Патерн “pipes and filters” дозволяє розбити задачу, яка виконує складну обробку, на серію окремих елементів, які можна використовувати окремо. Це дозволяє покращити продуктивність, масштабованість і можливість повторного використання, дозволяючи незалежно розгортати і масштабувати елементи задачі, які виконують обробку.

Для реалізації цього патерну можна використовувати послідовність черг повідомлень, щоб забезпечити інфраструктуру, необхідну для реалізації конвеєра. Початкова черга повідомлень отримує необроблені повідомлення. Компонент, реалізований як завдання фільтра, прослуховує повідомлення в цій черзі, виконує свою роботу, а потім відправляє перетворене повідомлення в наступну чергу в послідовності. Інше завдання фільтра може прослуховувати повідомлення в цій черзі, обробляти їх, розміщувати результати в іншій черзі і

так далі, поки повністю перетворені дані не з’являться у остаточному повідомленні в черзі. На рис. 3.15 показано реалізацію конвеєру з використанням черг повідомлень.

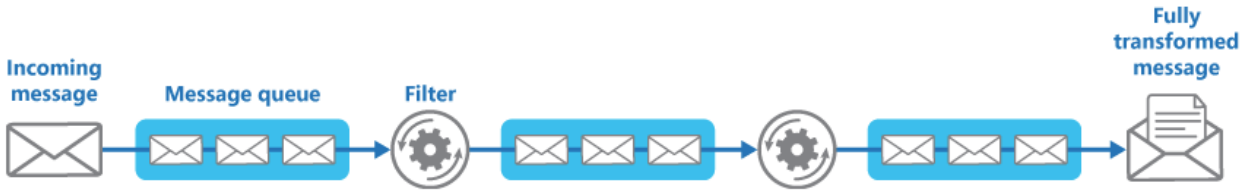


Рисунок 3.15 - Конвеєр з чергами повідомлень

Компоненти, призначені для перетворення даних, традиційно називаються фільтрами, тоді як коннектори, що передають дані від одного компонента до іншого, називаються pipes або труба. Безсерверна архітектура добре підходить для такого патерну. Рекомендується писати кожну лямбда-функцію як деталізований сервіс або завдання з урахуванням принципу єдиної відповідальності. Вхідні та вихідні дані повинні бути чітко визначені (тобто повинен бути чіткий інтерфейс) та мінімізувати будь-які побічні ефекти. Дотримання цієї поради дозволить створити функції, які можна повторно використовувати в конвеєрах та ширше у безсерверній системі.

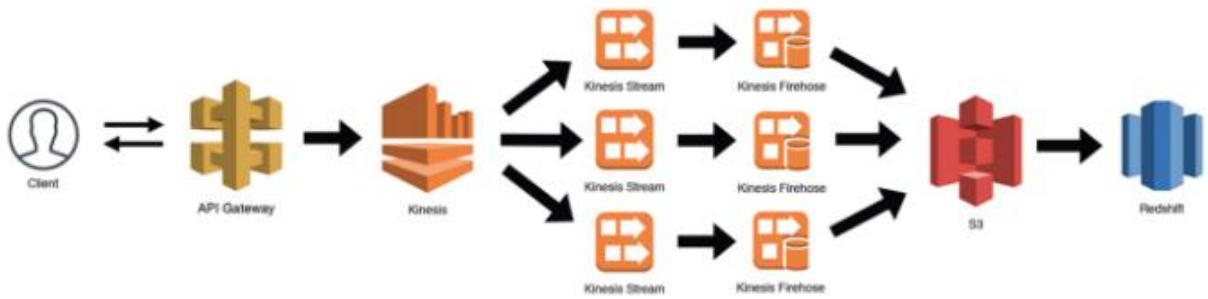


Рисунок 3.16 - Шаблон pipes and filters з сервісами AWS

3.10 Шаблон Fan-out/Fan-in

Лямбда-функції обмежуються 15 хвилинами загального часу виконання, тому великі завдання та інші інтенсивні за часом процеси можуть легко перевищити це обмеження. Використовуючи стратегію “поділяй і володарюй” можливо цю проблему пом’якшити. Щоб обійти це обмеження, обробка повинна бути розподілена між різними лямбда-воркерами. Використовується одна лямбда-функція, яка розділяє весь процес обробки на низку менших компонентів. Кожний обробник буде оброблювати задачу асинхронно і зберігати підмножину результатів у загальний репозиторій. Кінцевий результат можливо зібрати докупи іншим процесом або зробити запит стосовно нього до самого репозиторію.

Fan-out and Fan-in паттерн відноситься до поділу задачі на підзадачі, одночасного виконання декількох функцій з наступним агрегуванням результату. Це 2 патерни, які використовуються разом. Повідомлення Fan-out патерну надсилаються обробникам, кожний з яких отримує розділену підзадачу. Fan-in збирає результат з усіх робочих процесів, об’єднує його і надсилає повідомлення, яке сигналізує про те, що задача виконана. Завдання можуть надходити з будь-яких можливих тригерів AWS Lambda, як синхронних, так і асинхронних. Це включає традиційні виклики API або інтеграцію з API Gateway, DynamoDB Streams тощо.

В AWS Fan-out шаблон, як правило, реалізується з використанням SNS topics, які дозволяють викликати декількох підписників при додаванні нового повідомлення до теми. Візьмемо для прикладу S3. Коли до bucket-у додається новий файл, S3 може викликати одну лямбда-функцію з інформацією про файл. Але що, якщо потрібно одночасно викликати дві, три або більше лямбда-функцій? Замість того, щоб викликати лямбда-функції з одним повідомленням, можливо налаштувати S3 для відправлення повідомлення в тему SNS, щоб одночасно викликати всі функції, як показано на рис. 3.17, які є підписаними на

дану тему. Це ефективний спосіб створювати керовані подіями архітектури та паралельно виконувати операції.

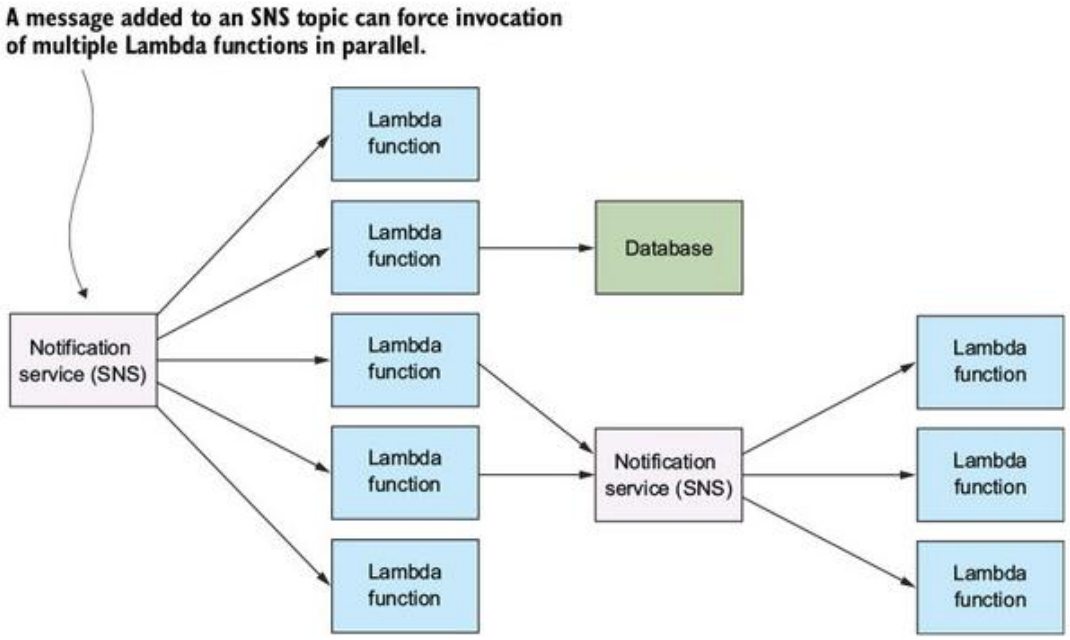


Рисунок 3.17 — Одночасний виклик лямбда-функцій

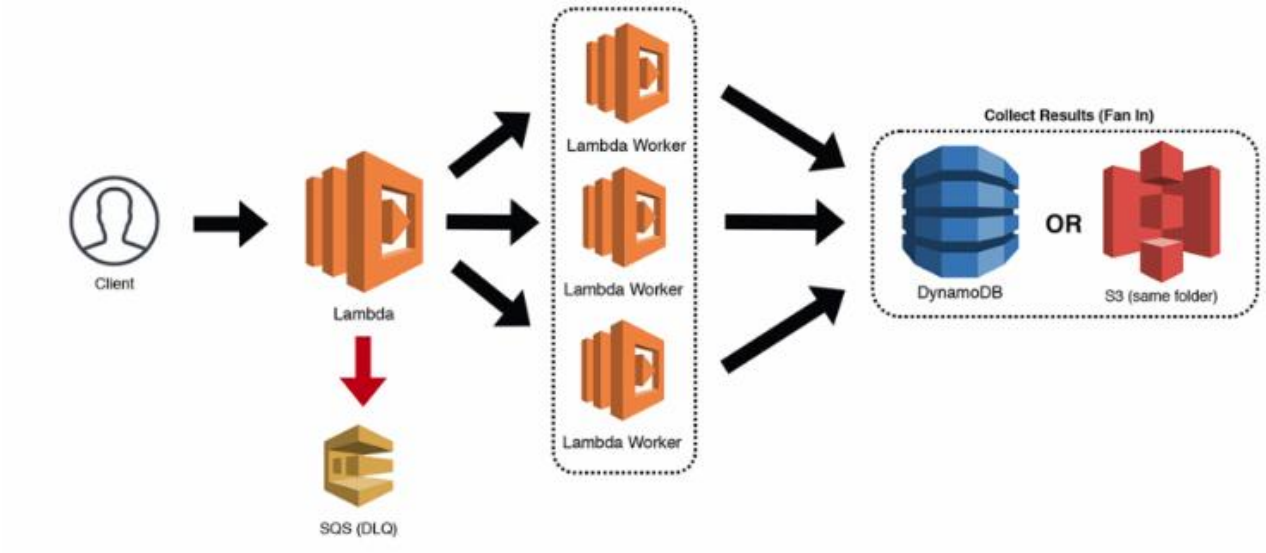


Рисунок 3.18 — Шаблон Fan-out/Fan-in

Шаблон, зображений на рис. 3.18 є корисним, коли потрібно одночасно викликати кілька лямбда-функцій. Тема SNS спробує повторити спробу

викликати лямбда-функції, якщо не вдається доставити повідомлення або якщо функцію не вдається виконати. Крім того, цей паттерн можна використовувати не лише для виклику декількох лямбда-функцій. Теми SNS підтримують інших підписників, такі як електронна пошта та SQS. Додаючи нове повідомлення до теми, можна одночасно викликати лямбда-функції, надіслати електронне повідомлення або надсилати повідомлення в чергу SQS одночасно [26].

3.11 Висновки

Патерни, розглянуті вище, повинні стати відправною точкою при розробці serverless додатків. Перед початком розробки serverless додатку треба визначити, що повинна робити кожна лямбда-функція, які дані вона повинна приймати, з якими сервісами повинна взаємодіяти і яка її зона відповідальності. Як і з терміном serverless не існує формального узгодженого визначення того, із чого повинна складатися serverless система. Однак serverless архітектура повинна задовольняти наступні стандарти.

1. Сервіси повинні мати свої власні приватні дані.
2. Serverless мікросервіси повинні незалежно розгортатися. Вони повинні бути повністю незалежними та автономним. Цілком припустимо, що вони можуть залежати від інших сервісів, але ці залежності повинні повністю базуватися на чітко визначених каналах зв'язку між ними.
3. Використання можливої консистентності. Реплікація та денормалізація даних є основними принципами serverless архітектури. Той факт, що сервіс А потребує деякі дані сервісу Б не означає, що їх слід поєднувати. Дані можуть взаємодіяти в реальному часі через синхронний зв'язок, якщо це можливо, або можуть бути репліковані між сервісами.
4. Використання асинхронних робочих навантажень. AWS Lambda виставляє рахунок за кожні 100 мс використаного часу обробки. Якщо очікується завершення інших процесів, то йде оплата за це очикування. Якщо є можливість, слід обрати хореографію замість оркестрації і надати змогу задачам працювати у фоновому режимі. Для більш складних оркестрацій слід використовувати Step functions. Також оркестрація має ще один негативний сторонній наслідок, який полягає у сильному зв’язуванні сервісів, що значно ускладнює координацію змін і порушує один з головних принципів архітектури — loose coupling. Звісно, хореографія також має свої недоліки. По-перше, асинхронність означає,

що немає можливості надати одразу відповідь, тому що існує тільки наявність запуску процесу. Також необхідно відслідковувати події і контролювати весь процес, коли йде обмін між сервісами.

5. Функції повинні залишатися невеликими, але цінними
6. Коли очікується відповідь від декількох функцій перед тим, як відбувається перехід до виконання наступної, слід використовувати концепцію кінцевих автоматів, які реалізовані у AWS за допомогою Step Functions. Вони управляють поведінкою декількох функцій, дозволяючи зберігати стан між викликами або переходами. Вони дають гарний результат для складних ланцюгів викликів функцій, в першу чергу для виконання паралельних подій і можливості агрегувати результати.

Поєднання мікросервісного підходу з serverless не тільки надає нам переваги мікросервісних архітектур, але також значно знижує проблеми масштабування, вимоги до управління інфраструктурою і в кінцевому підсумку може заощадити гроші за рахунок скорочення надлишкового використання ресурсів. Це потребує планування, а інколи і переосмислення звичного підходу побудови додатків, але в кінці кінців serverless архітектура надасть гнучкість у міру зростання додатку.

4 ПОРІВННЯ СЕРВЕРНОЇ І БЕЗЗСЕРВЕРНОЇ РЕАЛІЗАЦІЇ ДОДАТКА В AWS

У даному розділі буде побудовано дві реалізації одного додатку на основі контейнерів і на безсерверній архітектрі з їх наступним порівнянням в ціновому сегменті, по швидкодії і простоті їх побудови. Serverless додаток буде побудований з використанням патерну Simple WEB Service. Додаток буде складатися з Serverless API і SPA, реалізований на Angular. Реалізація на основі контейнерів буде мати аналогічну структуру і функціональні можливості і буде розміщена у хмарному сервісі AWS Elastic Container Service.

4.1 Serverless реалізація додатку

Для побудови додатку було обрано паттерн Simple WEB Service. Додаток складається з клієнтської і серверної частини. Клієнтська частина реалізована як SPA (однострінковий web-додаток, який для забезпечення роботи завантажує весь необхідний код разом із завантаженням самої сторінки) на Angular. Серверна частина розроблена за допомогою FaaS і BaaS сервісів AWS. Додаток має назву “Compare yourself”. Він має функціонал аутентифікації користувачів, зміни паролю, а також реєстрації з підтвердженням за допомогою електронної пошти, який розроблений за допомогою сервісу AWS Cognito. Користувач має можливість ввести про себе дані, такі як зріст, вік і заробітня плата, і порівняти себе по одному з обраних параметрів з іншими користувачами. Уся бізнес-логіка розташовується у лямбда-функціях і для взаємодії з SPA використовується єдина агрегована точка входу - API Gateway. Дані користувача зберігаються у сховищі DynamoDB. Клієнтська частина, яка представлена SPA зберігається у S3. Також для покращення продуктивності використовується сервіс кешування

CloudFront і сервіс Route53 для визначення домену. Усе логування додатку відбувається за допомогою CloudWatch.

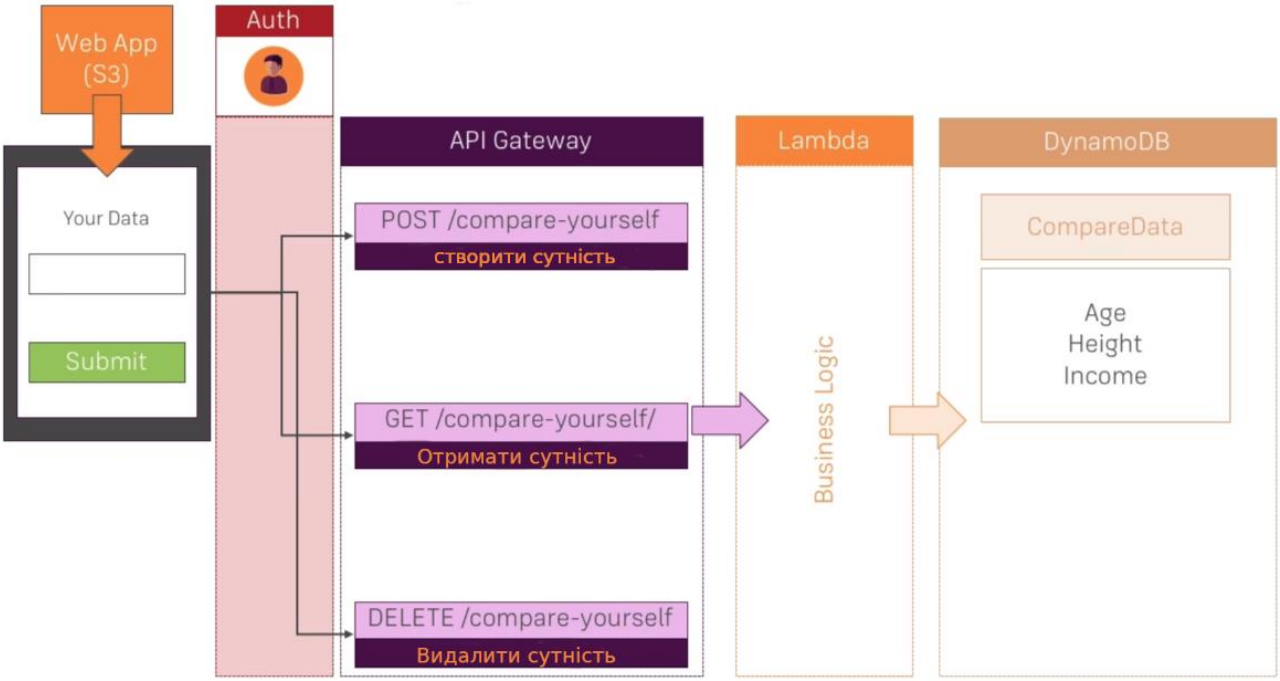


Рисунок 4.1 — Функціональна схема Serverless додатку

Розглянемо детальніше на рис. 4.2, які AWS сервіси використовуються для побудови Serverless додатку і яке їх призначення.

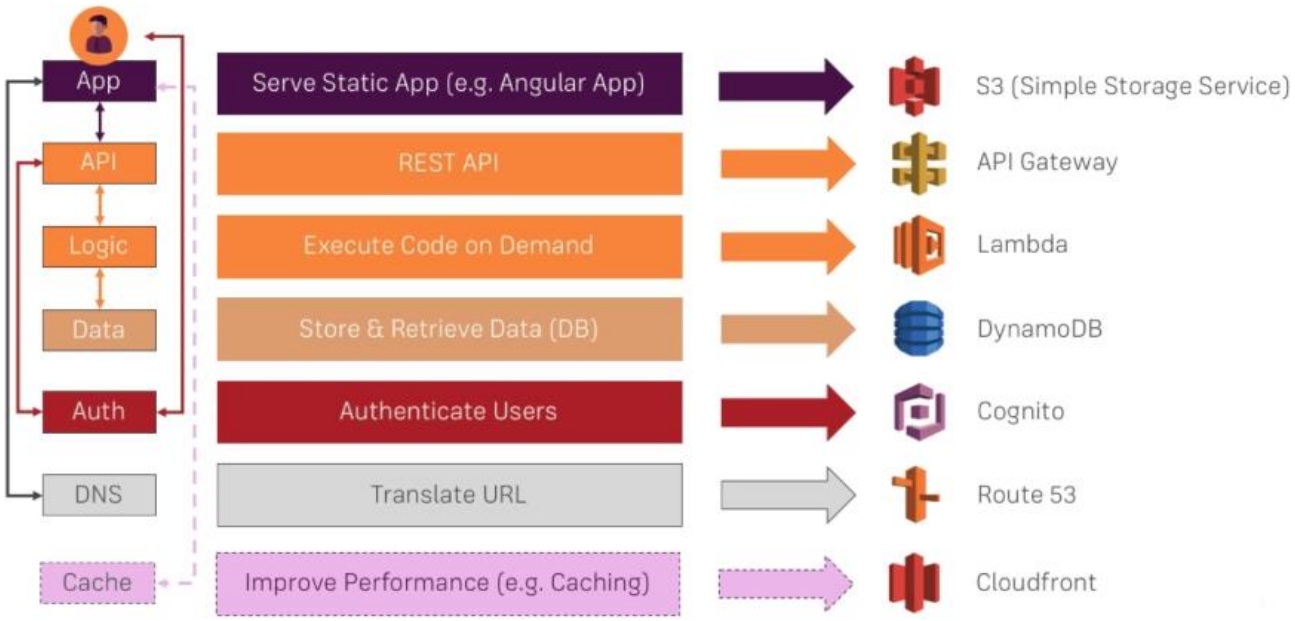


Рисунок 4.2 - Призначення AWS сервісів

Зазвичай для розгортання статичної клієнтської частини SPA використовуються окремі сервери, які потребують певного обслуговування, такого як масштабування, адміністрування, запуск і перезапуск при оновленні, а також безпека. AWS надає можливість розгортати статичні додатки за допомогою сервісу S3 у режимі статичного хостингу.

4.1.1 Послуга AWS S3

Це послуга Amazon Simple Object Storage. Це надійна, масштабована та швидка, безпечна система зберігання, яка доступна через веб-інтерфейс для завантаження будь-якої кількості даних з будь-якої точки світу.

Статичний хостинг веб-сайтів, архівування даних та доставка програмного забезпечення - це кілька загальних сценаріїв, коли S3 буде ідеальним інструментом. S3 надає можливість легко завантажувати дані, використовуючи AWS SDK. S3 також підтримує велику кількість популярних мов програмування, тому інтегрувати S3 доволі легко в існуючий стек.

У S3 файли зберігаються у структурах під назвою bucket. Buckets схожі на директорії на комп'ютері. Кожний bucket має свою унікальну назву, яку можна використовувати лише один раз, тобто ця назва є унікальною. Це корисно для унікальної ідентифікації ресурсів та для статичного хостингу веб-сайтів з доменними іменами.

Немає обмежень щодо кількості файлів, які можна зберігати у bucket-і. Також надаються додаткові функції, такі як контроль версій та політики. S3 також є об'єктною службою зберігання, що означає, що S3 розглядає кожен файл як об'єкт. Кожен об’єкт може мати власні метадані, що включають ім’я, розмір, дату та іншу інформацію. Amazon S3 - надійний інструмент для виконання усіх вимог до сховищ веб чи мобільних додатків. Завдяки ціноутворення за запитом та масштабованості S3 — найкраще хмарне рішення для зберігання даних як для малого, так і для великого бізнесу.

Статичний додаток складається з HTML, CSS і JS файлів. Серверний код відсутній, тому S3 — це найкраще місце для хостингу даного SPA додатку, тому що не потрібно жодних налаштувань середовища схову.

SPA додаток повинен взаємодіяти з API, наприклад, для зберігання або отримання певних даних. Для цих потреб використовується REST API, реалізоване за допомогою AWS API Gateway.

4.1.2 Послуга API Gateway

Це програмний компонент, популярний у світі мікросервісів, який є ключовою частиною HTTP-орієнтованої serverless архітектури.

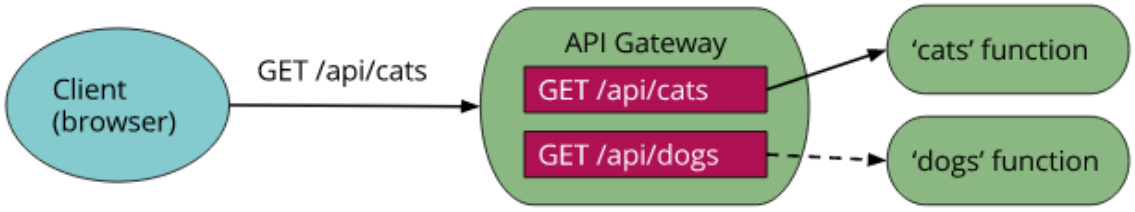


Рисунок 4.3 - API Gateway

Основна задача API Gateway - це виконувати задачі HTTP-серверу, де маршрути та кінцеві точки визначаються у конфігурації, і кожен маршрут пов’язаний з ресурсом для обробки цього маршруту. У випадку serverless архітектури обробником, як правило, є FaaS функція, але може бути будь-який інший сервіс. Коли API Gateway отримує запит, він знаходить конфігурацію маршрутизації, що відповідає запиту, а у випадку маршруту, що підтримується FaaS, він викличе відповідну функцію FaaS з передачею вхідного запиту. Крім того Api Gateway, зображений на рис. 4.3 трансформує запит HTTP у об'єкт JSON. Функція FaaS виконує свою логіку і повертає результат назад до API

Gateway, який, у свою чергу, перетворює отриманий результат у HTTP відповідь, яка повертається клієнту.

Amazon Web Services мають власний шлюз, а інші постачальники пропонують подібні можливості. Шлюз API Amazon - це послуга BaaS, оскільки вона є зовнішньою службою, яка налаштовується розробником, але її не потрібно запускати чи розгортати самостійно. Крім запитів на маршрутизацію, API Gateway також має можливість виконувати аутентифікацію і авторизацію, перевірку вхідних даних, регулювання користувачів, відображення коду відповіді тощо. [8]

4.1.3 Послуга DynamoDB

Наступною частиною додатку є AWS Lambda, яка будет виконувати бізнес-логіку додатку за потребою. Вона була детально розглянута у попередніх розділах. Для зберігання даних використовується NoSQL база даних DynamoDB.

Це документо-орієнтована база даних “ключ-значення” з однозначним часом відгуку у мілісекундах у будь-якому масштабі. Це повністю керована надійна база даних із вбудованими можливостями захисту, резервного копіювання та відновлення. Ключове слово, яке асоціюється з DynamoDB, полягає в тому, що це база даних NoSQL. Це означає, що вона не використовує традиційну мову запитів SQL, що використовується в реляційних базах даних а також не містить відношень. Структура DynamoDB полягає у зменшенні складності між таблицями шляхом консолідації об'єктів у загальну колекцію або "безсхемну" таблицю в базі даних NoSQL. Після цього ці об’єкти групуються разом на основі загальних тем, які відповідають умовам загальних запитів програми, які встановив розробник. Також важливо пам’ятати, що DynamoDB - це безсхемна база даних, в якій елементи можуть мати різні набори атрибутів. DynamoDB може обробляти більше 10 трильйонів запитів на

день і може підтримувати піки понад 20 млн. запитів в секунду, що робить його однією з найкращих служб AWS для програм, які потребують доступу до даних із низькою затримкою в будь-якому масштабі. Оскільки швидкість є суттєвою і з технічної точки зору потрібні більш короткі кроки ініціалізації при збереженні високого рівня безпеки, DynamoDB ідеально підходить, оскільки доступ здійснюється через HTTP, а служба використовує AWS IAM. Ці два елементи означають, що база даних завжди захищена, а запити швидко аутентифікуються та перевіряються без необхідності складної конфігурації мережі, наприклад, розділення мережі.

4.1.4 Послуга AWS Cognito

Аутентифікація виконується за допомогою AWS сервісу Cognito, який дозволяє легко створити user pools для дозволу користувачам проходити реєстрацію і дозволяти їм використовувати сервіси додатку. Amazon Cognito надає можливість легко додавати користувачів для реєстрації та входу у мобільні додатки та веб-програми. User pools в Amazon Cognito - це повністю керований каталог користувачів, який може масштабуватися до сотень мільйонів користувачів, тому не доведеться турбуватися про створення, захист та масштабування рішення для управління користувачами та аутентифікації. AWS Cognito має дві основні сутності: user pools і identity pools.

User pools є постачальником ідентифікаційних даних. Він може використовуватися для аутентифікації користувачів у мобільному додатку, на веб-сайті та керування користувачами. Identity pools використовуються, щоб дозволити користувачам надавати доступ до ресурсів AWS, таких як DynamoDB, S3 та інших. Крім того, Amazon Cognito дозволяє синхронізувати дані на декількох пристроях, щоб у користувача не було проблем при переході між ними або після купівлі нового пристрою. Додаток може зберігати дані на пристроях користувача, що надасть можливість працювати з ними навіть у

автономному режимі, а після відновлення підключення до Інтернету виконує автоматичну синхронізацію даних.

4.1.5 Послуга Route 53

При хостингу клієнтської частини у S3 для використання власного URL використовується AWS сервіс Route53, який дозволяє реєструвати і конфігурувати власний домен, і коли надходить запит до нього, то відбувається завантаження WEB-сторінки з S3.

Це високо-доступний та масштабований веб-сервіс хмарної системи доменних імен (DNS). Він спрямовує кінцевого користувача до Інтернету шляхом перетворення імені, наприклад iasa.com, в IP-адресу, наприклад 10.6.107.31. Route53 добре відповідає стандарту IPv6.

Route53 використовується не тільки для системи доменних імен, але також для реєстрації доменного імені, перевірки працездатності та маршрутизації інтернет-трафіку до ресурсу для документа.

4.1.6 Послуга CloudFront

Для покращення продуктивності веб-додатку буде використовуватись сервіс кешування, який AWS надає як Cloudfront, який копіює статичні файли до різних дата-центрів по всьому світу, що дозволяє користувачам отримувати доступ до WEB-сторінок швидше

CDN розшифровується як Content Delivery Network. Це дуже велика кількість серверів кешування, які розташовані по всьому світу. Він містить контент, який зберігається на вихідних серверах, і спрямовує користувачів до найкращого місця, щоб вони могли переглядати контент, що зберігається в кеш-пам’яті. Вміст може бути статичним або динамічним за своєю суттю. CDN збільшує масштабованість і продуктивність додатків.

Amazon CloudFront - це глобальна мережа доставки контенту з величезною потужністю та масштабами. Він оптимізований для продуктивності та масштабованості. Також вбудовані функції безпеки, і є можливість налаштувати їх для оптимального обслуговування. Користувач контролює послугу і може вносити зміни на льоту. Він включає звіти в режимі реального часу, що надає можливість контролювати продуктивність та вносити зміни до програми або способу взаємодії CDN з додатком. Він оптимізований для статичних та динамічних об'єктів та доставки відео.

4.1.7 Створення API Gateway

На сайті AWS оберемо сервіс API Gateway і за допомогою консолі розробника створимо перший компонент додатку. У діалоговому вікні оберемо “REST API” як тип нашого API і натиснемо кнопку “Build”

У новому діалоговому вікні необхідно обрати “New API”, яке дозволяє створити нове API з нуля, і ввести назву створюваного API.

Після цього отримуємо нове API і переходимо у редактор API Gateway. У цьому редакторі визначаються усі HTTP методи, які реагують на вхідні запити, а також усі ресурси, які додаток використовує.

Створимо перший ресурс (URL-сегмент) “/compare”. Для цього натискаємо на кнопку “Actions” і обираємо “Create Resource”. Вводимо назву ресурсу і встановлюємо прапорець “Enable API Gateway CORS” у встановлене положення, як зображено на рис. 4.4. CORS забороняє робити запити від SPA-додатку до ссерверного коду без певних заголовків. Серверний код повинен відправити заголовки клієнту, які дадуть йому змогу робити запити; в іншому випадку браузер не дозволить зробити запит від клієнта до сервера. Для цього браузери роблять preflight-запити до певних кінцевих точок і потребують відповіді від них. Для обробки preflight-запитів потрібно надати OPTION

ендпойнт, який відправить відповідь на цей запит у виглядів певних заголовків, які проінформують браузер, що можна надсилати запити до цих ресурсів.

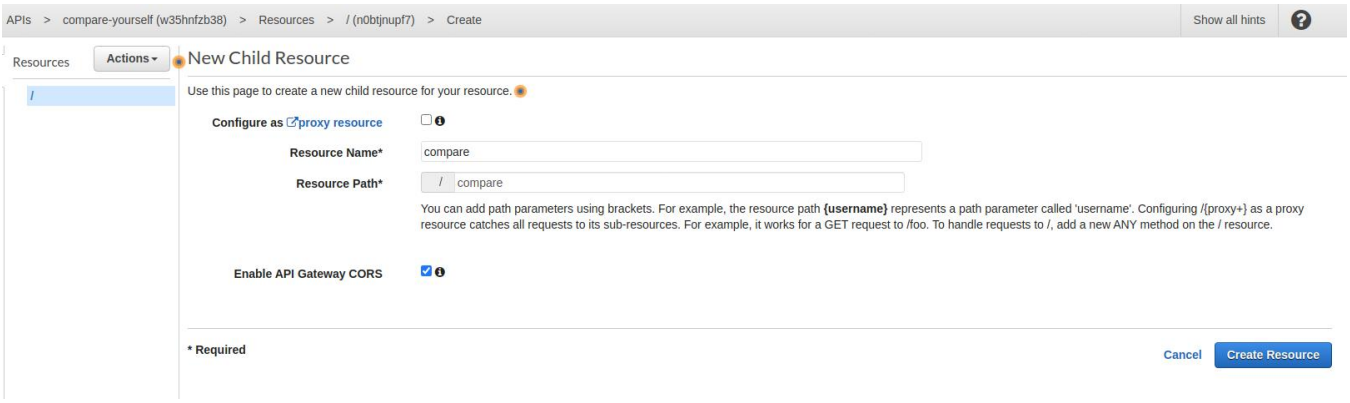


Рисунок 4.4 — Створення ресурсу “/compare”

Створимо POST запит для даного ресурсу для збереження даних користувача, де обробником бізнес-логіки буде виступати лямбда-функція. Для початку створимо відповідну лямбда-функцію. Для цього переходимо до сервісу Lambda і, використовуючи модальне вікно, створюємо лямбда-функцію. У діалоговому вікні треба ввести назву лямбда-функції і обрати мову, на якій вона буде написана. У нашому випадку це буде Node.js 12.x. Процес створення зображено на рис. 4.5.

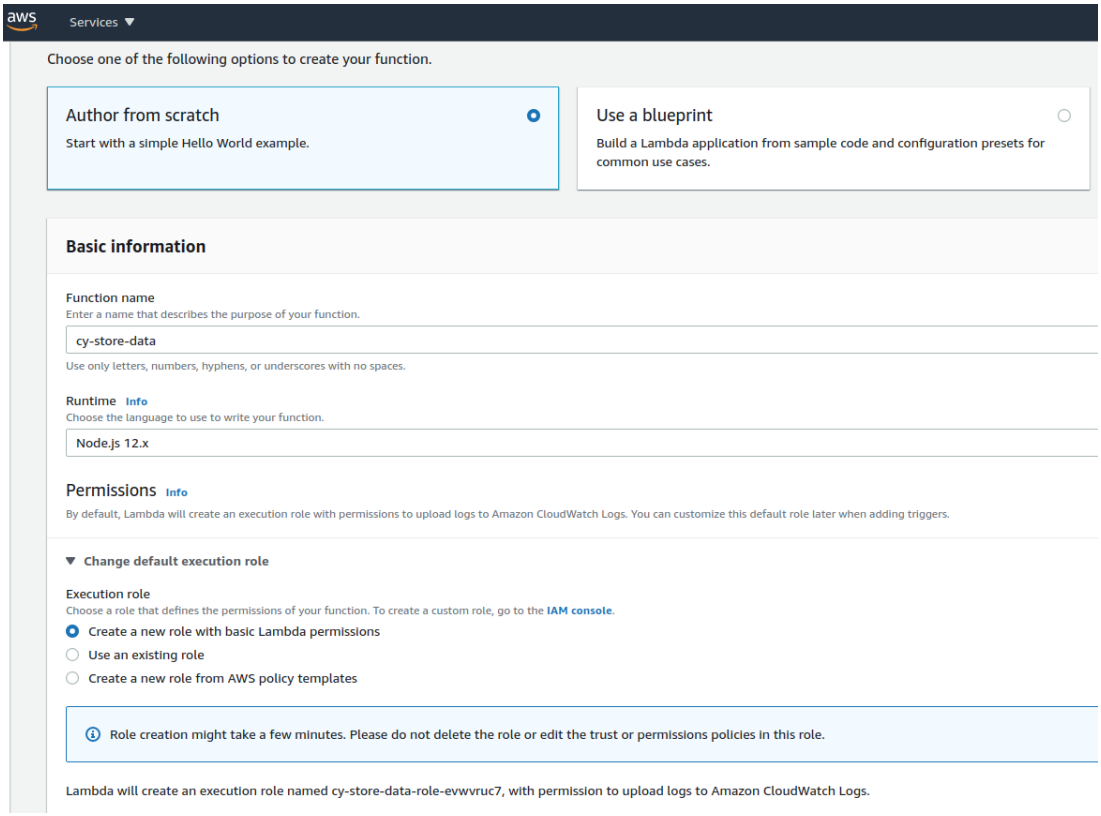


Рисунок 4.5 — Створення лямбда-функці

Після створення пустої лямбда-функції, треба її приєднати до API Gateway. Для цього натискаємо на “Actions”, обираємо “Create Method”. У діалоговому вікні, зображеному на рис. 4.6, обираємо інтеграційний тип як “Lambda Function” і підключаємо створену лямбда-функцію “cy-store-data”. Ця лямбда-функція буде викликатись, коли буде надходити POST запит на ресурс “/compare”.

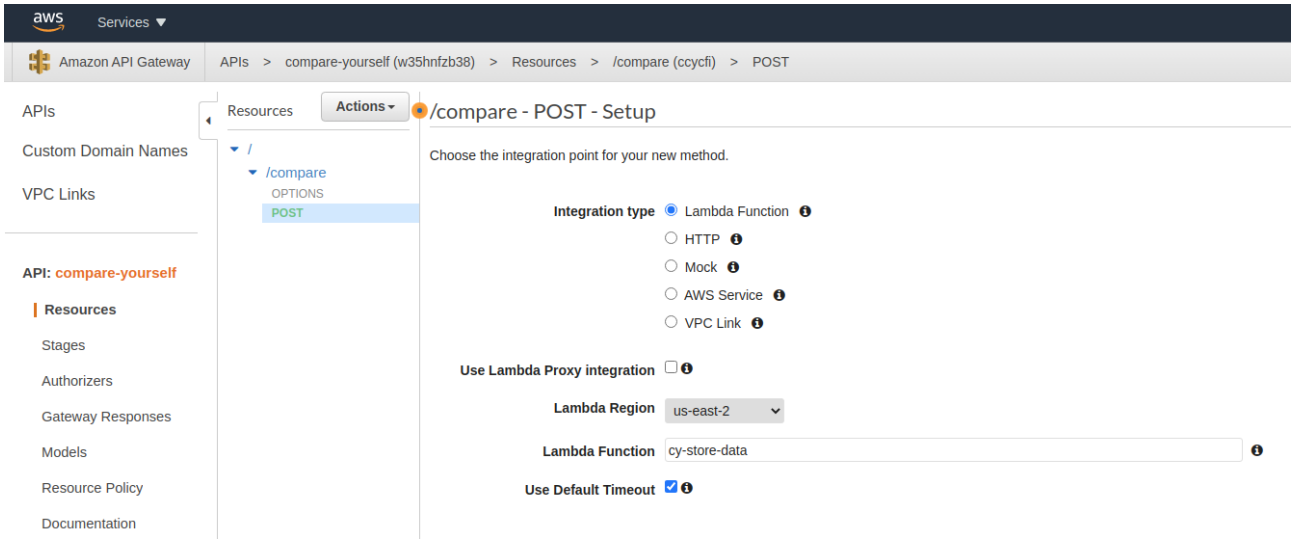


Рисунок 4.6 - Поєднання API Gateway і лямбда-функції

Додамо підтримку CORS для даного ресурсу для всіх методів. Для цього натискаємо “Actions” і обираємо “Enable CORS”. У даному діалоговому вікні обираємо, для яких методів дозволити CORS і які заголовки можна передавати з клієнта.

Опишемо об’єкт вхідного запиту, який повинен надходити на API Gateway. Для цього треба створити модель “compareData”, яка буде визначати, які параметри запит повинен містити, а саме - age, height і income.

Переходимо до секції Models і натискаємо на “Create”. Дамо назви для моделі і її content-type, а також опишемо модель, як зображено на рис. 4.7.

Після створення моделі можна використати її для валідації вхідних даних запиту. Для цього переходимо до відповідного POST методу ресурсу “/compare” і встановимо створену модель “CompareData” у “Request Body”, а також налаштуємо “Request Validator” на “validate body”.

Далі налаштовуємо інтеграційний запит, який отримує дані від запиту клієнта до ресурсу і трансформує його у певну зручну структуру, яку отримує лямбда-функція. Додаємо “body mapping templates” до POST методу, який крім параметрів запиту, отримує “userId” з JWT-токену, який передається у заголовку “Authorization”.

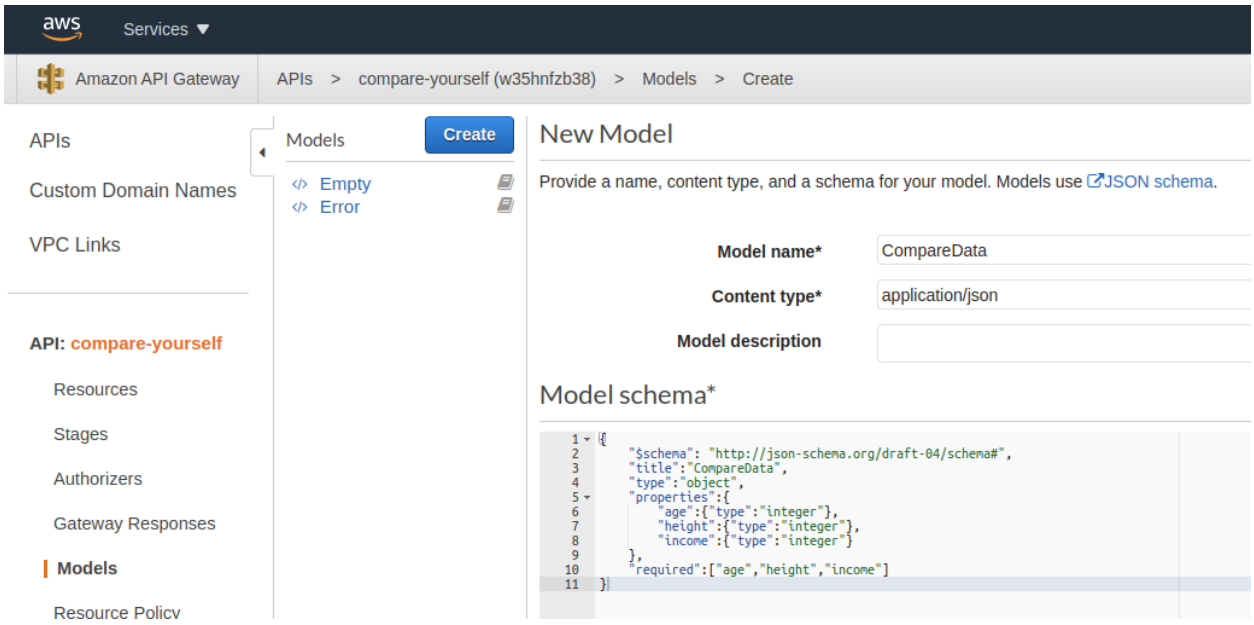


Рисунок 4.7 - Створення моделі CompareData

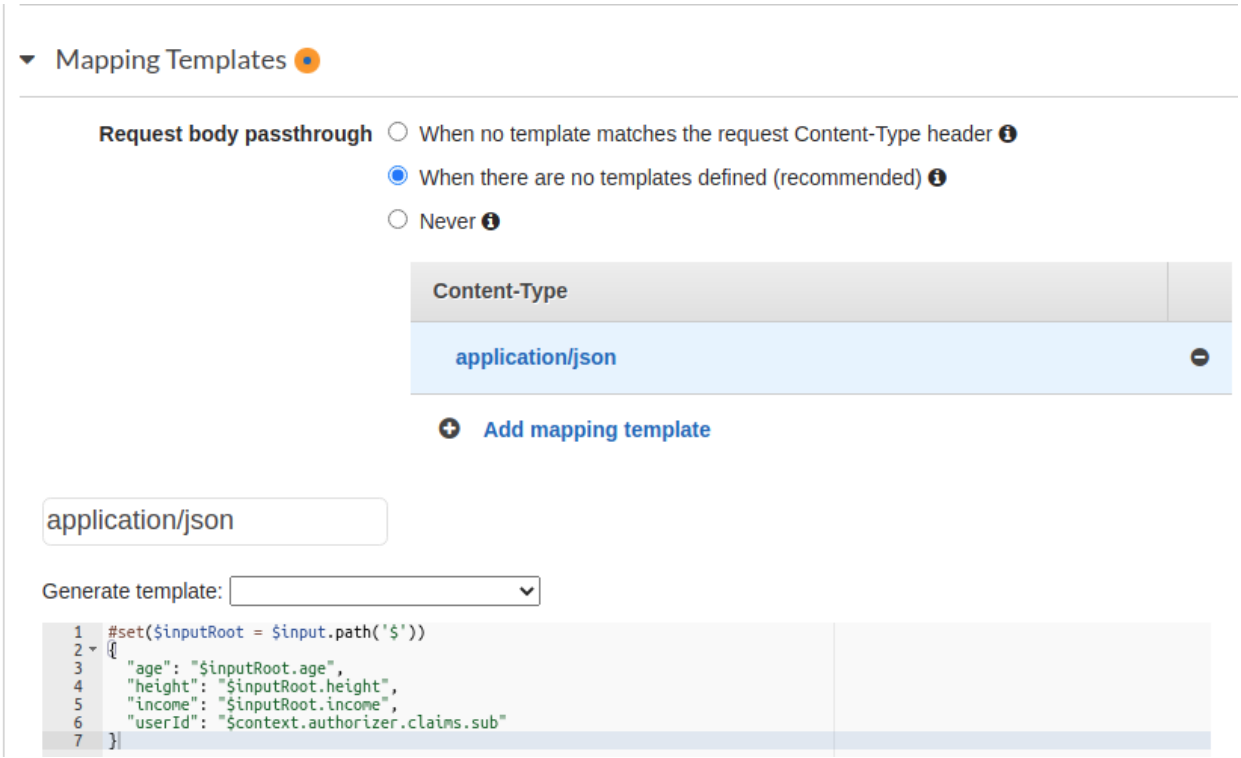


Рисунок 4.8 — Налаштування інтеграційного запиту

За допомогою інтеграційного запиту, зображеному на рис. 4.8 лямбда-функція відокремлюється від API Gateway і повністю контролюються параметри, що надходять до лямбди і що вона віддає назад до API Gateway.

4.1.8 Налаштування DynamoDB

Додаємо новий сервіс DynamoDB. Для цього обираємо його зі списку усіх сервісів AWS і натискаємо “Create”. Вводимо назву таблиці і “Partition Key”

Далі необхідно створити permissions, тобто дозволити лямбда-функції взаємодіяти з DynamoDB, а саме, зберігати дані, видаляти і отримувати. Для початку необхідно створити власний policy на додавання елементів до таблиці. Для цього переходимо на вкладку Policy і натискаємо кнопку “Create”.

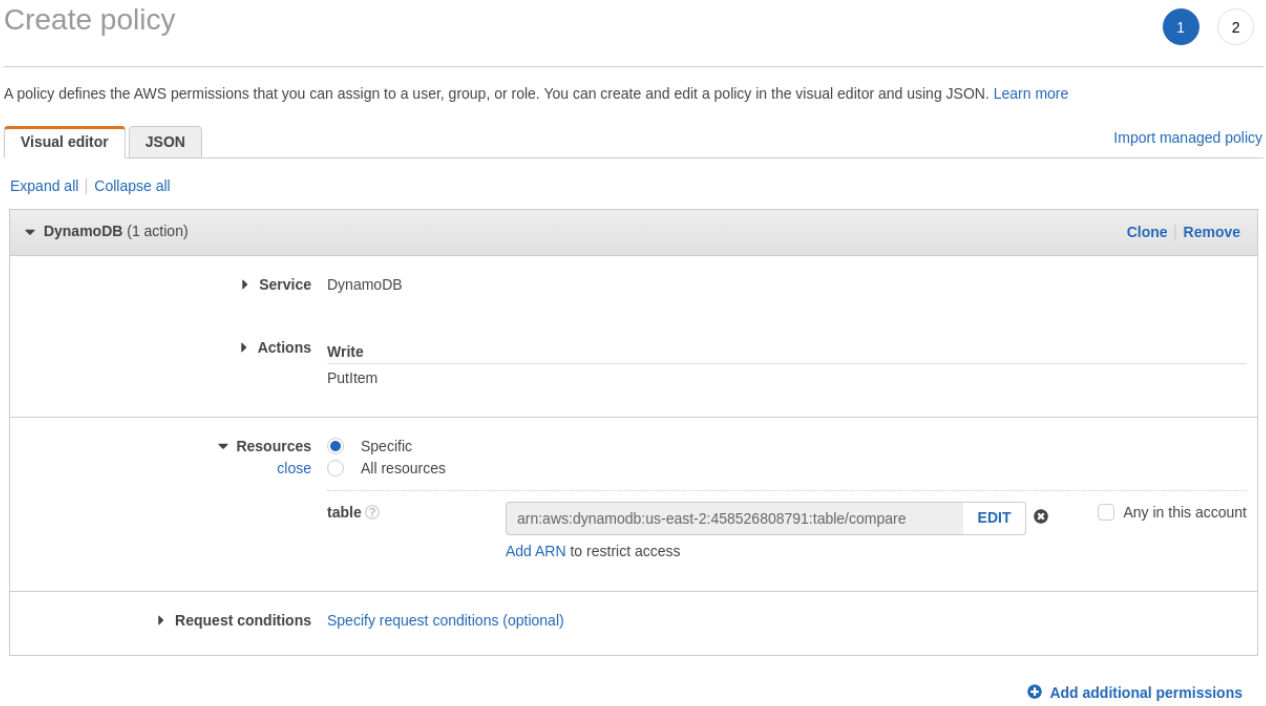


Рисунок 4.9 - Створення permissions на запис до бази

У діалоговому вікні, зображеному на рис. 4.9 визначаємо назву сервісу — DynamoDB, які дії можна виконувати над даним сервісом - “PutItem” і обираємо відповідну таблицю. Після цього додаємо створену policy до лямбда-функції, яка зберігає дані користувача.

4.1.9 Інтеграція аутентифікації за допомогою AWS Cognito

Для цього перейдемо до секції усіх сервісів і оберемо з проміж них Cognito.

Після цього створимо User Pool. Для цього дамо йому назву “compare-yourself” s натискаємо на “Step through settings” для подальшого налаштування пулу.

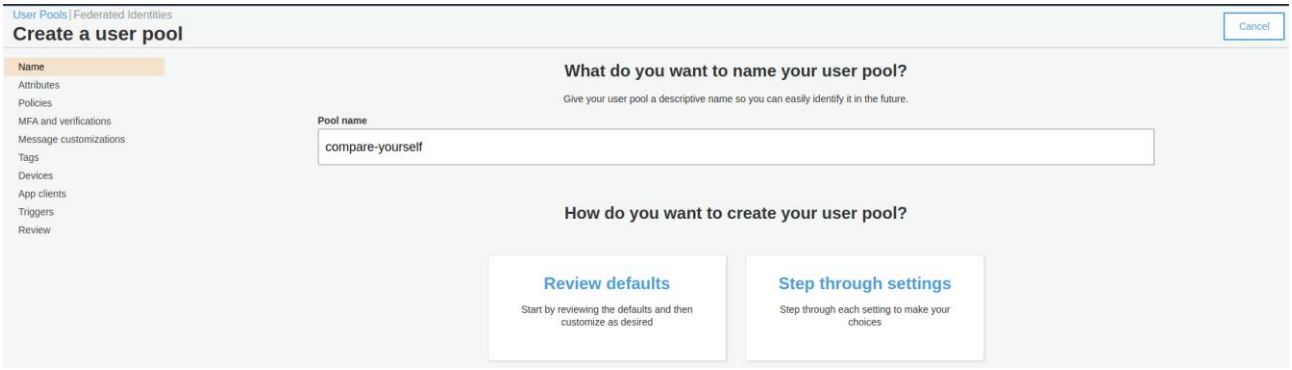


Рисунок 4.10 — Створення User Pool

У діалоговому вікні, зображеному на рис. 4.10 обираємо, що процес логіну буде використовувати email користувача або його username. Далі налаштовуємо параметри для паролю, яка його довжина і які символи мін має містити. Також при налаштуванні пулу є можливість підключити двох-факторну аутентифікацію, налаштування шаблону листів для скидання паролю, верифікацію акаунту, логування, відслідковування, з яких пристроїв заходив користувач, тощо.

Після цього необхідно приєднати SPA до Cognito. Для цього необхідно додати назву SPA у секції “App clients” і налаштувати тривалість дії токенів, якими будуть обмінюватись SPA і AWS Cognito.

Також необхідно налаштувати API Gateway на використання Cognito. Для цього у створеному API Gateway переходимо до секції “Authorizers”, зображеної на рис. 4.11 і створюємо новий. Даємо назву цьому авторайзеру, обираємо тип “Cognito” і створений до цього User Pool.

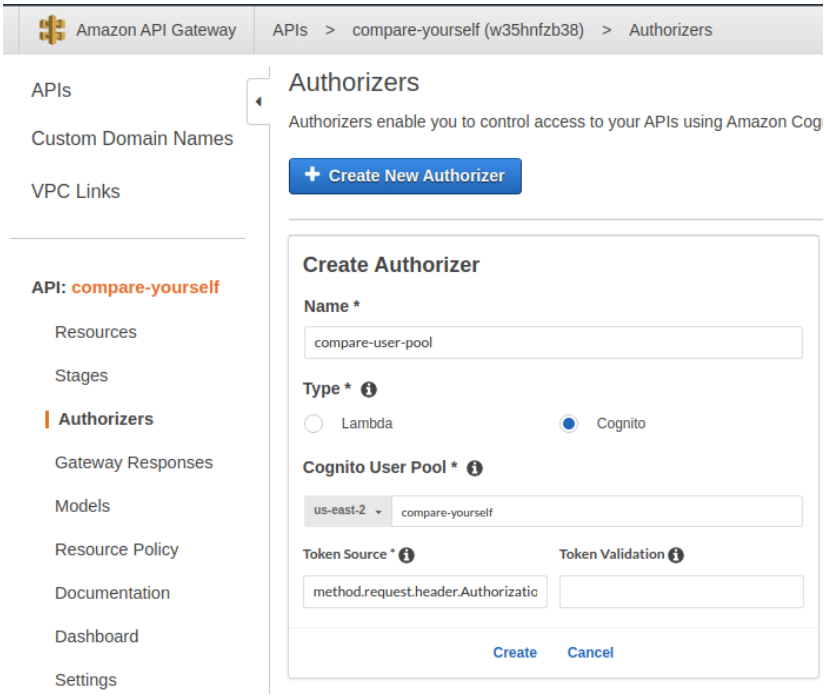


Рисунок 4.11 — Створення нового Authorizer

Після того, як новий авторайзер створений, необхідно його додати до методу ресурсу. Для цього обираємо POST метод і встановлюємо йому Authorization, який був створений на попередньому кроці.

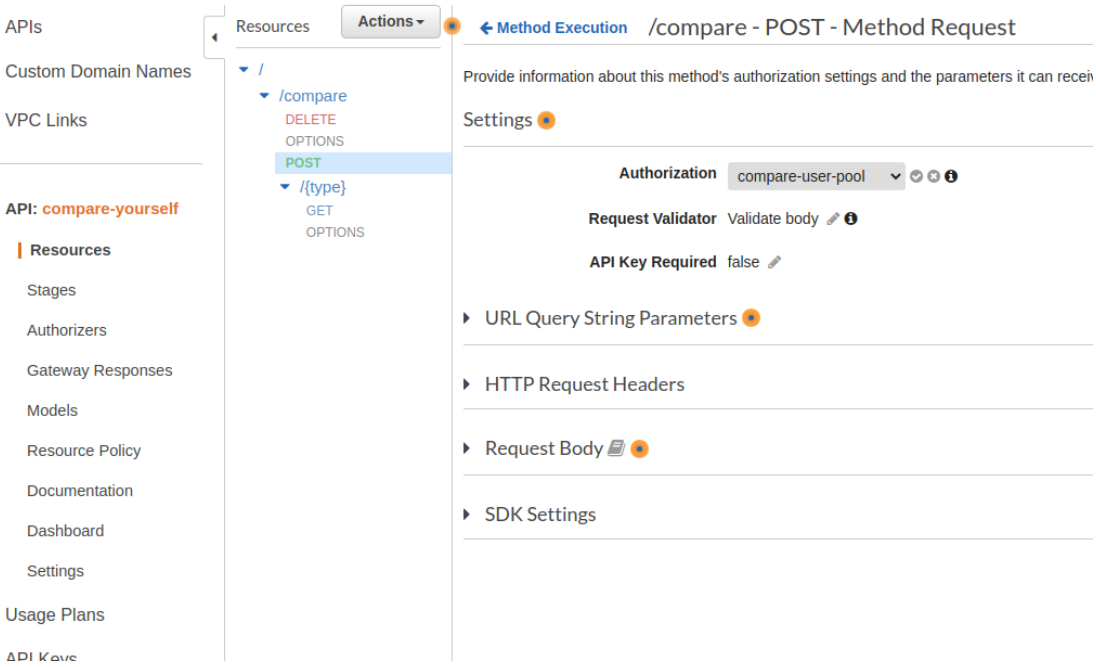


Рисунок 4.12 — Додавання authorizer до POST методу

Після реєстрації користувач додається до User Pool.

4.1.10 Завантаження SPA додатку до S3

Для цього створимо новий S3 bucket. Переходимо до сервісу S3 і натискаємо “Create new”. Необхідно ввести унікальну назву для нього і обрати регіон.

Після цього необхідно збудувати SPA додаток для продакшену за допомогою команди “npm run build --prod” і завантажити додаток у консолі розробника. Результат завантаження зображено на рис. 4.13

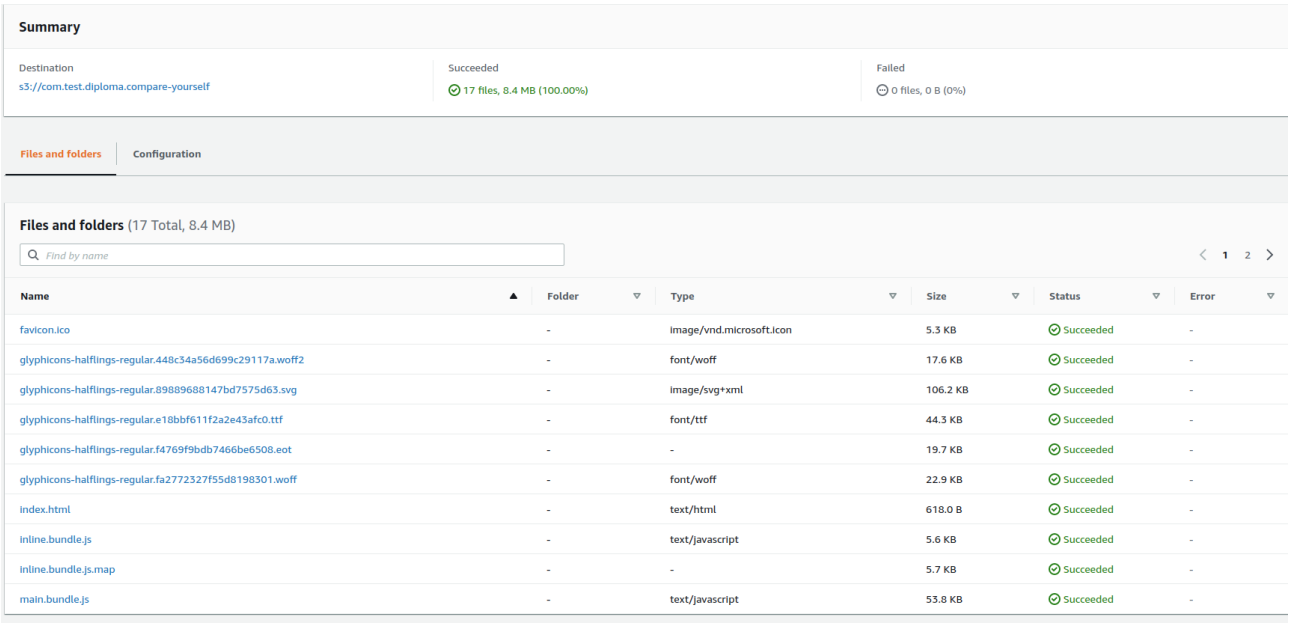


Рисунок 4.13 - Результат завантаження файлів до S3

Після завантаження необхідно надати дозвіл на зчитування файлів з цього bucket анонімним користувачам і перевести його у режим статичного сайту. У консолі розробника створимо новий policy і надамо дозвіл на зчитування.

Також переводимо його у режим статичного сайту, де обираємо стартову сторінку і сторінку для відображення помилок.

Для логування усіх подій у SPA додатку був створений ще один додатковий bucket, який зберігає всі логи. Для логування роботи лямбда-функцій і API Gateway використовується AWS Cloudwatch.

Для цього необхідно обрати сервіс CloudWatch, перейти до розділу “Log Groups” і створити нову групу. Для створення групи необхідно ввести ARN лямбда-функції і назву групи.

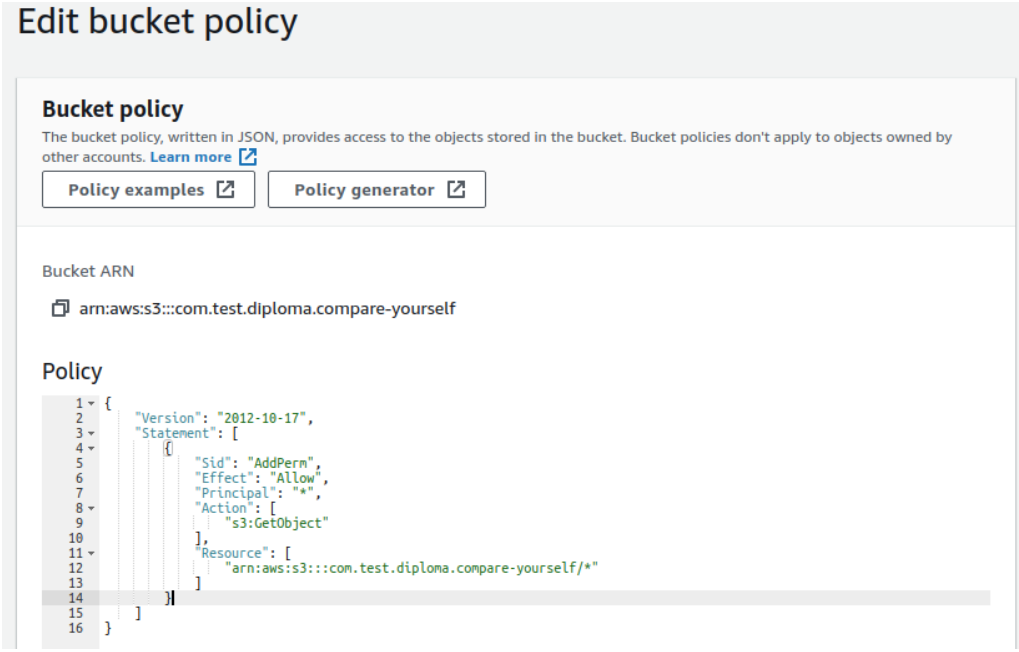


Рисунок 4.14 — Створення policy

4.2 Реалізація додатку на основі контейнерів

Серверна частина складається з 3 мікросервісів. Вони містять в собі бізнес логіку створення, видалення і зчитування даних користувача відповідно. Дані мікросервіси розміщуються у контейнерах, які завантажуються до AWS Elastic Container Service. Клієнтська частина зберігається у S3, а для зберігання даних використовується сервіс DynamoDB. Для аутентифікації і кешування використані ті ж самі сервіси, що й для serverless додатку. Функціональну схему показано на рис. 4.15.

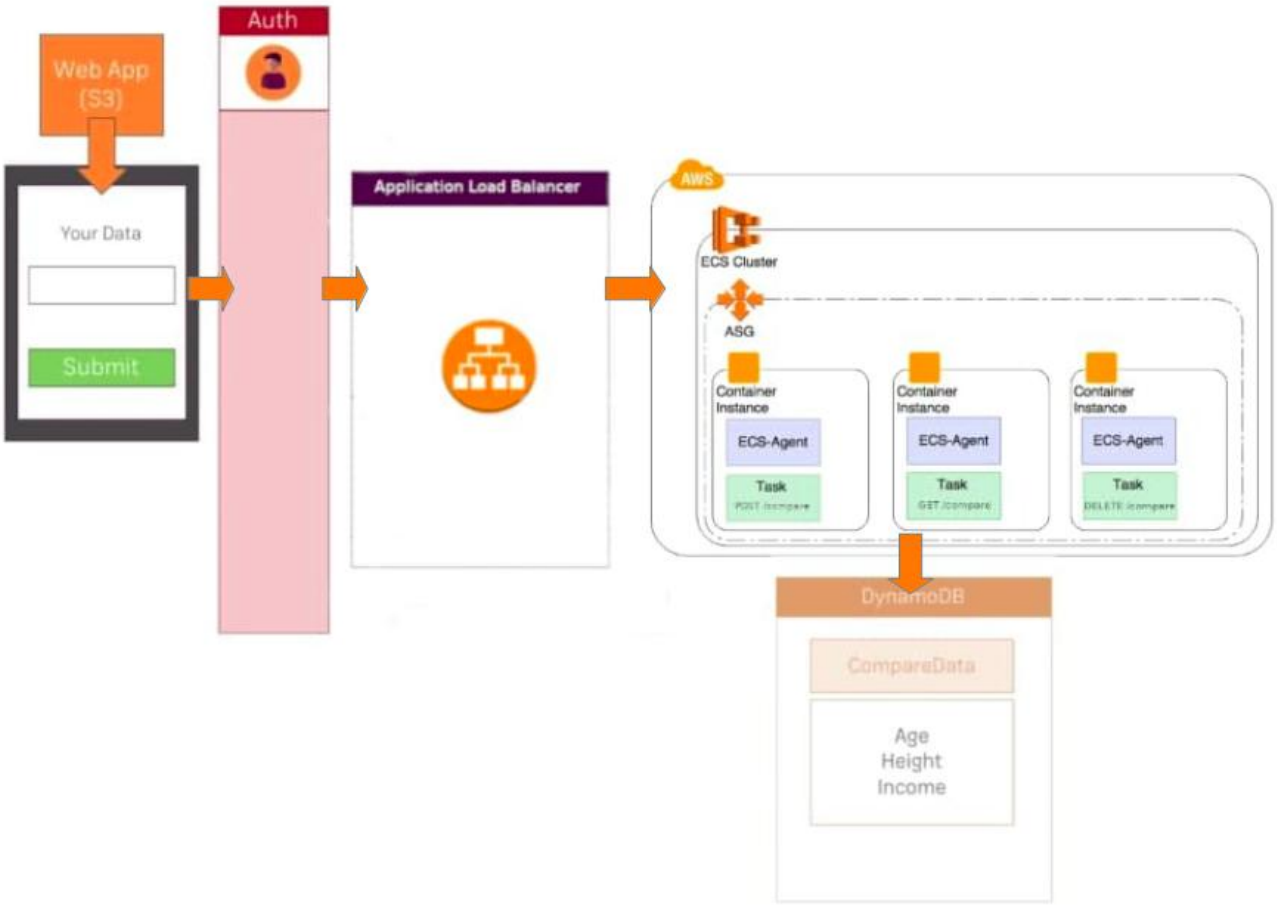


Рисунок 4.15 - Класична мікросервісна архітектура

Amazon Elastic Container Service — це повністю керований сервіс організації контейнерів. Це високо-масштабований, швидкий сервіс управління контейнерами, який значно спрощує запуск, зупинку та керування контейнерами Docker у кластері, де Docker Container - це стандартна одиниця програмного забезпечення, яка пакує код та всі його залежності, завдяки чому програма запускається швидко та надійно з одного обчислювального середовища в інше. Він дозволяє розмістити кластер у безсерверній інфраструктурі, якою керує Amazon ECS, запустивши служби або завдання, використовуючи тип запуску Fargate. Також для більшого контролю можна розміщувати завдання на кластері екземплярів Amazon Elastic Compute Cloud (Amazon EC2), якими керує розробник, використовуючи тип запуску EC2.

Тип запуску Amazon ECS визначає тип інфраструктури, на якій розміщуються завдання та сервіси. Amazon ECS проводить масштабування, моніторинг, дозволяє запускати та зупиняти програми на основі контейнерів за допомогою простих викликів API, також дозволяє отримувати стан кластера за допомогою централізованої служби та надає доступ до багатьох звичних функцій Amazon EC2. Amazon ECS також дозволяє планування з розміщення контейнерів у створеному кластері на основі потреб у ресурсах, політик ізоляції та вимог щодо доступності. Amazon ECS позбавляє розробника від необхідності керувати власними системами управління кластером та конфігурацією або турбуватися про масштабування інфраструктури управління. Він динамічно масштабує кластер докер-контейнерів між екземплярами EC2 і регіонами на основі метрик CloudWatch або балансувальника навантаження.

Розглянемо детальніше тип запуску EC2. Тип запуску EC2 дозволяє запускати контейнерні програми у кластері екземплярів Amazon EC2, якими керує розробник. Amazon ECS використовує Docker image у визначеннях завдань (Task definitions) для запуску контейнерів у кластері віртуальних машин Amazon EC2 з попередньо встановленим Docker. Task Definitions є обов’язковим для запуску будь-якого додатка на Amazon ECS. Єдине, за що відповідає розробник — це визначення завдання, за все інше відповідає Amazon ECS. Формат визначення завдання створюється у вигляді текстового файлу або у форматі JSON, що описує один або кілька контейнерів, це композиція Dockerfile і docker-compose.yml. Тобто, task - це логічне групування контейнерів, що працюють разом на одному і тому самому екземплярі віртуальної машини. Обмеження створення контейнерів становить максимум 10, що утворюють програму.

Task definitions визначає такі параметри додатку, як Docker image, на основі якого буде створений контейнер у завданні; скільки використовувати CPU та оперативної пам’яті для кожного завданням або кожним контейнером у межах завдання; тип запуску, який визначає інфраструктуру, на якій

розміщуються завдання; конфігурація логування для завдань; команда, яку контейнер повинен запускати при його запуску.

На рис. 4.16 показано загальну архітектуру:

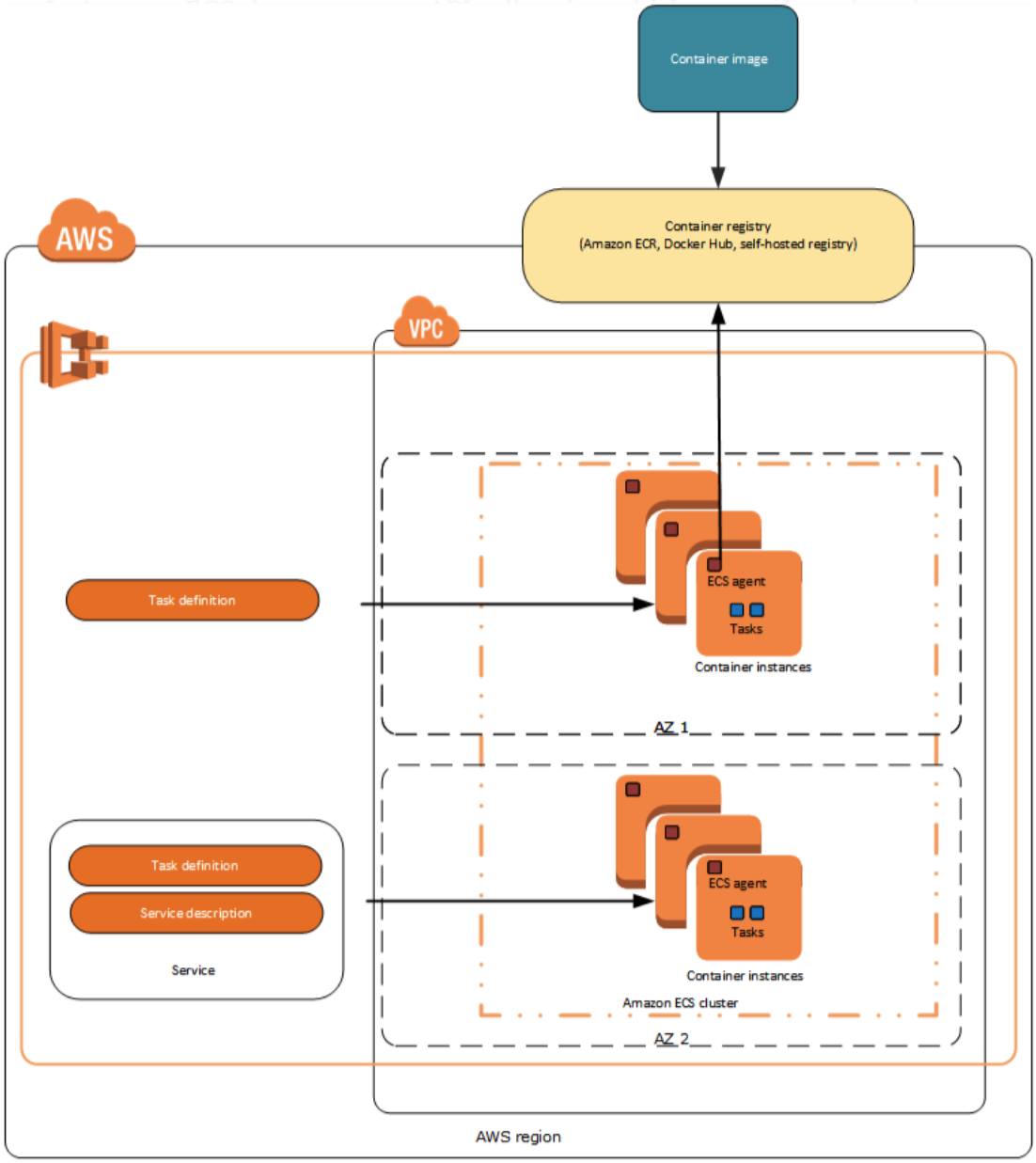


Рисунок 4.16 - Архітектура ECS

Можна визначити декілька контейнерів у визначенні завдання. Програма може охоплювати декілька визначень завдань, об’єднуючи пов’язані контейнери у власні визначення завдань, кожне з яких представляє один компонент.

Сервіс дозволяє запускати задану кількість завдань конкретного task definitions на кластері, і всі ці завдання або контейнери відстежуються на наявність збоїв і автоматично перезапускаються планувальником послуг. Сервіс завдань Amazon ECS відповідає за розміщення завдань у кластері

Кластер - це логічне групування екземплярів EC2, яке ECS використовує для запуску контейнерів. При використанні типу запуску EC2 кластери представляють собою групу екземплярів докер-контейнерів, що працюють на декількох серверах. Екземпляр контейнера Amazon ECS - це екземпляр Amazon EC2, на якому запущений агент контейнера Amazon ECS. Amazon ECS завантажує зображення контейнерів із вказаного розробником реєстру та запускає ці зображення у кластері. Кожен кластер є специфічним для конкретного образу докера.

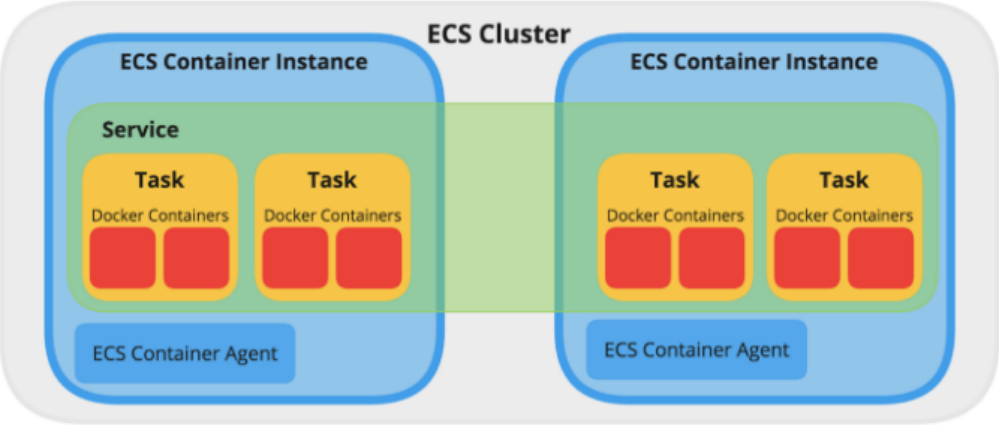


Рисунок 4.17 — ECS кластер

На рис. 4.17 видно, що кластер є групою екземплярів контейнерів ECS. Amazon ECS виконує такі задачі, як планування, обслуговування, обробка запитів на масштабування. Можна побачити, що є 4 запущені задачі або докер контейнери. Вони є частиною сервісу ECS. Сервіс та завдання охоплюють 2 екземпляри контейнера. На даній схемі не визначені task definitions, оскільки task є просто “екземпляром” task definition.

Агент-контейнер, зображений на рис. 4.18 працює на кожному ресурсі інфраструктури в кластері Amazon ECS. Він надсилає інформацію про поточні запущені завдання та використання ресурсів Amazon ECS, а також запускає та зупиняє завдання кожного разу, коли отримує запит від Amazon ECS.

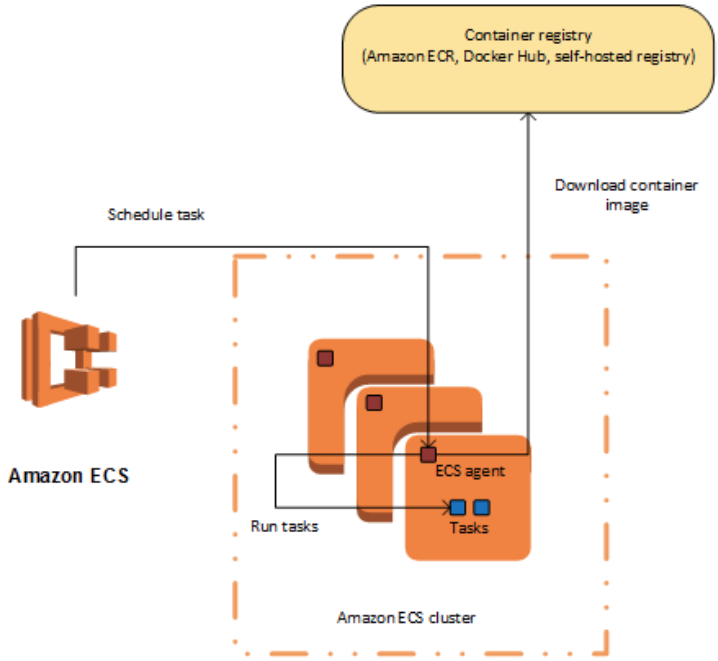


Рисунок 4.18 — Робота агент-контейнеру

Реєстр контейнерів Amazon Elastic або Amazon ECR - це керована служба реєстру AWS Docker, яка є безпечною, масштабованою та надійною. Amazon ECR підтримує приватні сховища Docker з дозволами на основі ресурсів за допомогою IAM, щоб певні користувачі або завдання мали доступ до сховищ та зображень. Розробники можуть використовувати CLI Docker для переміщення, витягування та управління зображеннями [25].

Група автоматичного масштабування може подбати про масштабування ресурсів кластера, наприклад про кількість екземплярів EC2, доступних у кластері. Існує можливість приєднати балансувальник навантаження до групи авто-масштабування та визначити політики для автоматичного масштабування на основі перевірок стану балансувальника навантаження. Існує 2 типи

балансувальника — Elastic load balancer і Application load balancer. Application load може динамічно відображати порти. Він і буде використаний при розробці додатку. Створимо Dockerfile, зображений на рис. 4.19

```
FROM node:14-buster

WORKDIR /app
ADD . /app

RUN npm install

EXPOSE 3001
```

Рисунок 4.19 - Dockerfile

Створимо власний репозиторій. Для нього встановимо видимість як приватний і дамо назву “compare_yourself”.

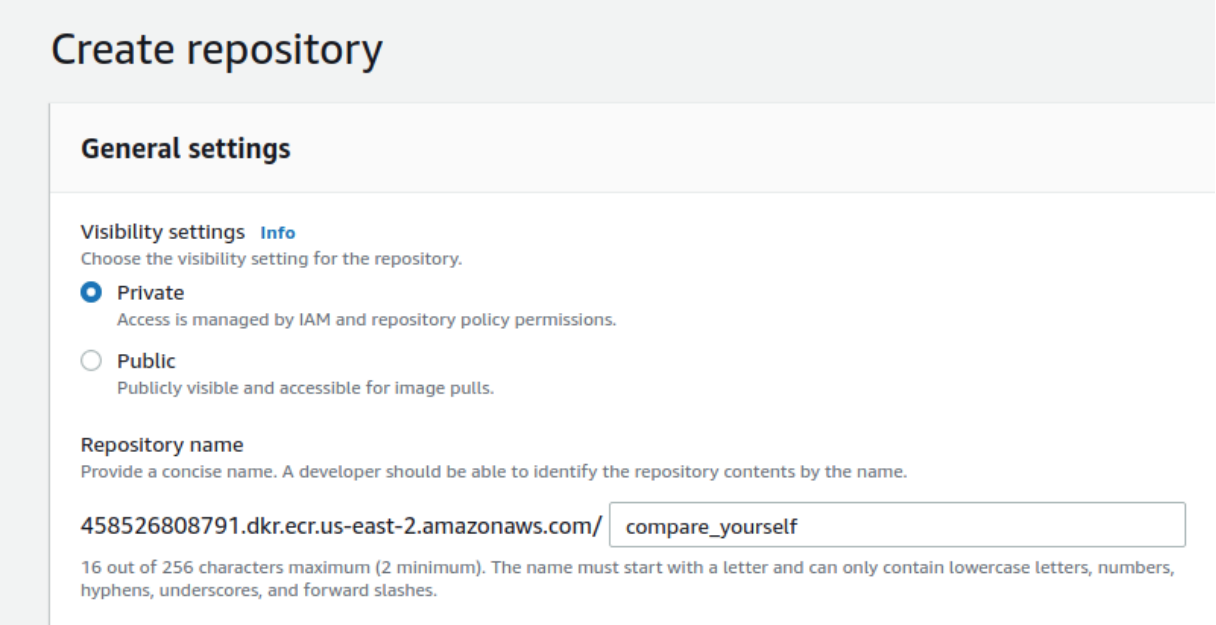


Рисунок 4.20 — Створення нового репозиторію

Після цього необхідного створити image описаного вище Dockerfile за допомогою команди “docker build -t назва_мікросервісу . “ для кожного мікросервісу.

Після створення image необхідно поставити йому тег і завантажити дані image до створеного AWS репозиторію.

docker tag compare_yourself:latest 458526808791.dkr.ecr.us-east-2.amazonaws.com/compare_yourself:latest

docker push 458526808791.dkr.ecr.us-east-2.amazonaws.com/compare_yourself:latest

Після завантаження образів мікросервісів необхідно створити task definitions, як зображено на рис. 4.21. Переходимо до розділу Task Definitions і створюємо новий. Оберемо тип запуску EC2. Також необхідно дати назву даного task definition, обрати тип мережі як Bridge і додати контейнер. Для цього необхідно ввести назву контейнера, image URL, з якого буде створений даний контейнер. Також необхідно встановити ліміт на оперативну пам’ять і встановити відображення портів і змінні середовища.

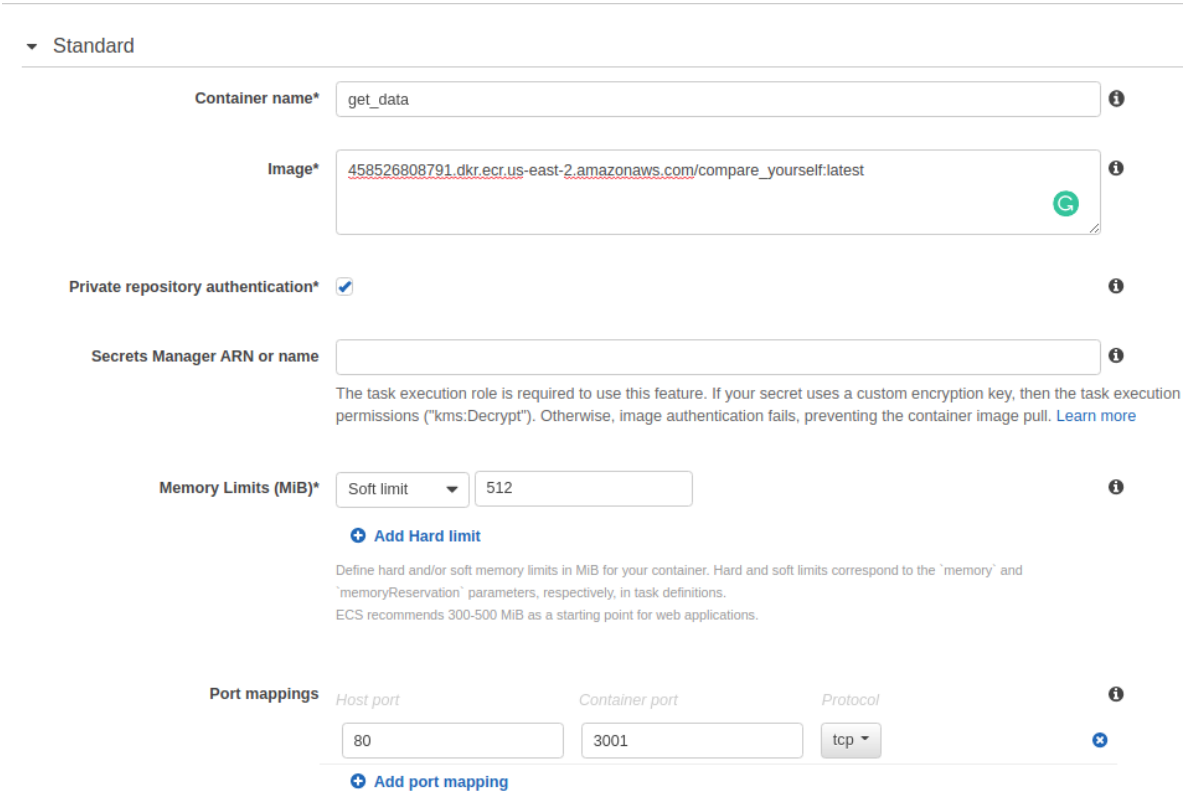


Рисунок 4.21 — Створення task definitions

Аналогічно створюємо ще 2 task definitions для store_data і delete_data контейнерів. Тепер необхідно створити кластер. Переходимо до розділу Cluster і створюємо новий. Обираємо тип EC2 Linux + Networking, який також створить такі ресурси, як VPC і Auto Scaling group with Linux AMI.

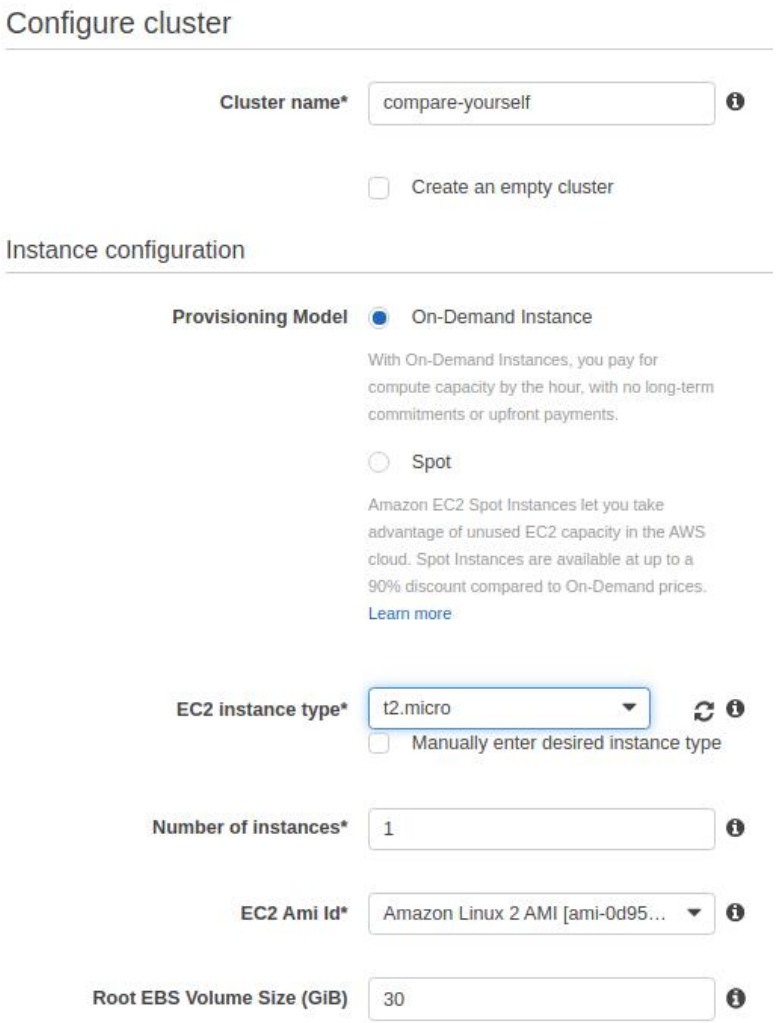


Рисунок 4.22 - Створення кластеру

Після створення кластеру, зображеного на рис. 4.22 і опису завдань, необхідно створити сервіси для запуску даних завдань. Створимо сервіс для task definitions “get_data”. Для цього перейдемо до відповідного розділу, оберемо тип запуску як EC2, беремо кластер, у якому даний таск буде запущений і оберемо кількість для запуску, як показано на рис. 4.23.

Також необхідно додамо Application Load Balancer і налаштувати авто-скейлінг, де можна встановити максимальне і мінімальне значення даних завдань.

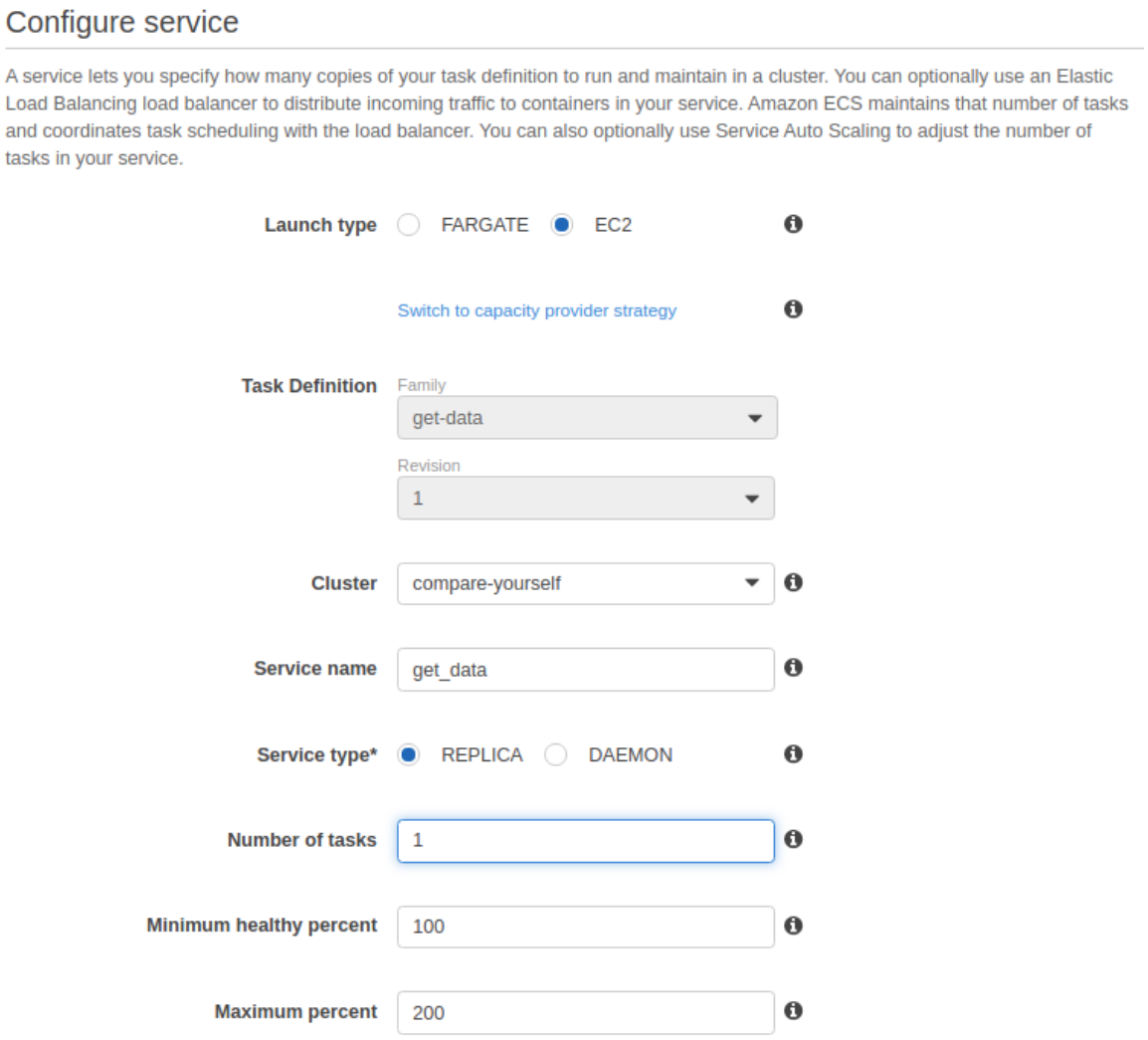


Рисунок 4.23 — Конфігурування сервісу

4.3 Результати роботи програми

На головній сторінці додатку, зображеній на рис. 4.24 для неаутентифікованих користувачів з’являються 2 посилання на вхід і реєстрацію нового користувача. Для реєстрації необхідно ввести такі дані про користувача, як і’мя користувача, його email, ввести пароль і підтвердити його.

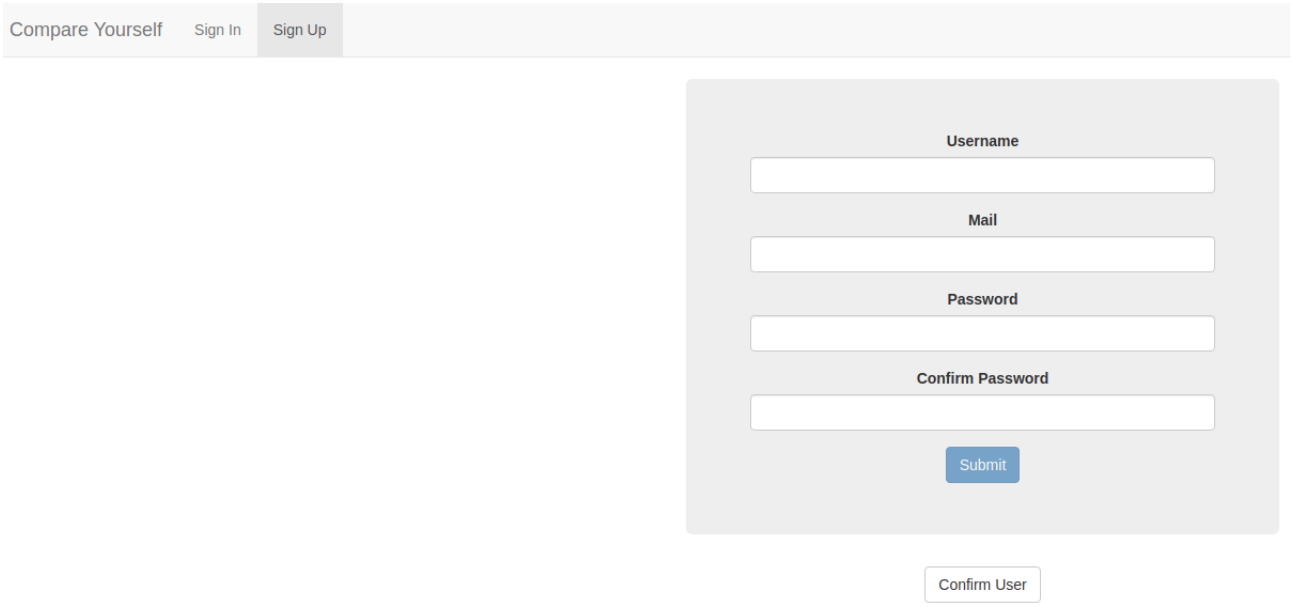


Рисунок 4.24 -Головна сторінка додатку для неаутентифікованих користувачів

Після вводу даних необхідно натиснути на кнопку Submit, як зображено на рис. 4.25

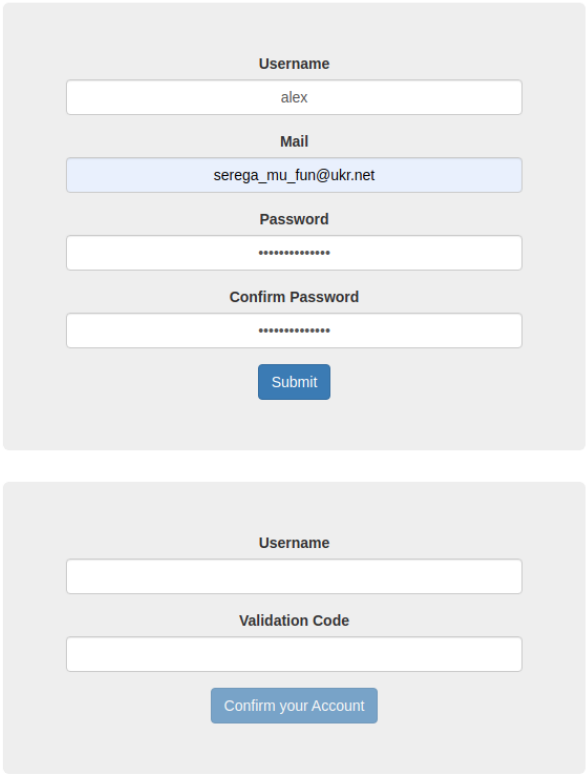


Рисунок 4.25 — Реєстрація

На адресу введеної електронної пошти буде надіслано код підтвердження даної електронної пошти. Отриманий код разом з ім’ям користувача необхідно ввести у відповідні поля і натиснути кнопку “Confirm your account”. Після цього відбудеться перенаправлення на головну сторінку додатку, де є можливість ввести власні дані або отримати їх, якщо вони були вже додані. Після вводу нових даних про користувача збережені дані будуть відображені на екрані, як на рис. 4.26.

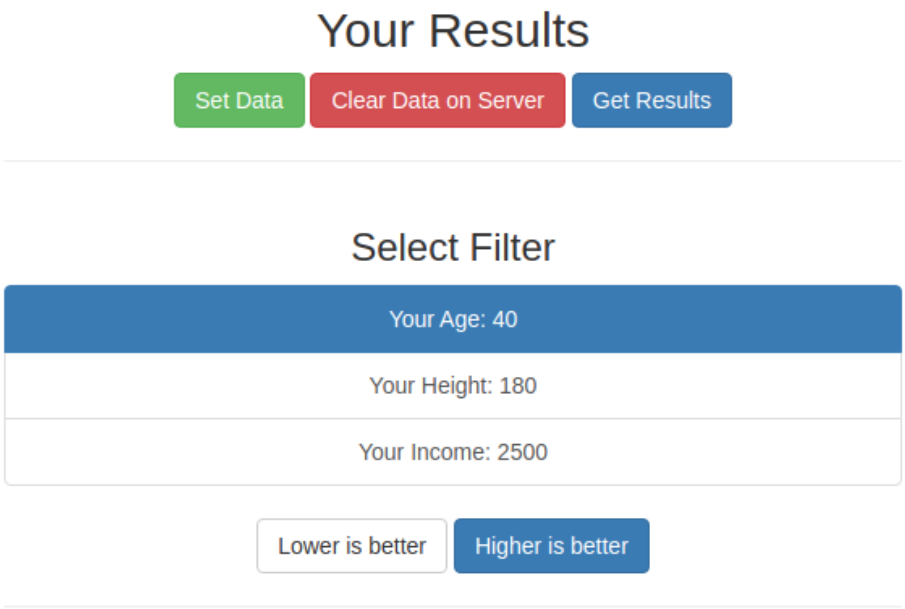


Рисунок 4.26 - Результат додавання даних користувача

Після цього можливо встановити нові дані за допомогою кнопки “Set Data”, видалити дані, натиснувши на “Clear Data on Server”, або отримати дані про всіх користувачів додатку за допомогою кнопки “Get Results”.

Отримавши всіх користувачів, можна зробити фільтрацію по одному з трьох параметрів, таких як вік, зріст і заробітна плата. Червоним кольором підкреслюються записи, які не відповідають критеріям фільтру, зеленим — які відповідають. Результат роботи зображено на рис. 4.27.

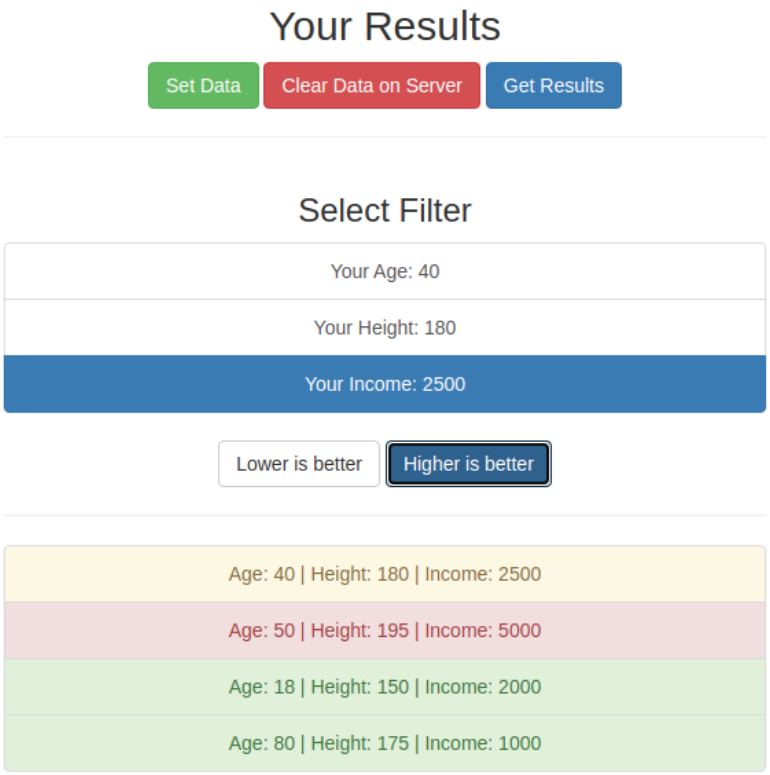


Рисунок 4.27 — Порівняння даних користувача з іншими даними за фільтром “Higher is better”

4.4 Порівняння розроблених додатків

У даній роботі був програмно реалізований паттерн для Serverless додатків - Simple Web Service. Для його реалізації були використані такі AWS сервіси: API Gateway, Lambda, S3, CloufFront, CloudWathc, Route53 і DynamoDB. Для збільшення швидкості додатку, були проведені наступні операції.

Лямбда-функція розгортається в екземплярі EC2 на кожний виклик заново. При холодному старті отримання даних усіх користувачів займало приблизно 1.1045 сек. Для пришвидшення виконання лямбда-функції і запиту в цілому було використано підключення AWS SDK і створення об’єктів сторонніх ресурсів, таких як DynamoDB і Cognito було винесено за межі обробника лямбда-функції, як зображено на рис. 4.28.

Ці зміни дозволили не створювати екземпляри об’єктів DynamoDB і CognitoIdentityServiceProvider при кожному виклику цієї лямбда-функції, і як наслідок пришвидшили виконання запиту на 0.1сек.

```
1  const AWS = require('aws-sdk');
2  const dynamoDB = new AWS.DynamoDB({
3    region: 'us-east-2',
4    apiVersion: '2012-08-10'
5  });
6
7  const cisp = new AWS.CognitoIdentityServiceProvider({
8    apiVersion: '2016-04-18'
9  });
10
11 exports.handler = (event, context, callback) => {
12   const accessToken = event.accessToken;
13   const type = event.type;
14
15   if (type === 'all') {
16     const params = {
17       TableName: 'compare'
18     };
19
```

Рисунок 4.28 — Ініціалізація ресурсів перед запуском лямбда-функції

Наступне місце для прискорення додатку — це клієнтська частина. Середній час рендерингу SPA-додатку — 450мс. Був доданий сервіс AWS CloudFront для кешування усіх статичних файлів SPA додатку, які розміщаються у сервісі S3. Додаток розміщується у регіоні Ohio. За допомогою цього сервісу усі HTML, CSS і JS файли були закешовані у всіх вузлах AWS по всьому світу, що значно прискорило швидкість доступу до SPA додатку для користувачів, і як наслідок рендеринг клієнтської частини. Тепер користувачам з України не потрібно робити запит в США для отримання клієнтської частини, тому що тепер статична частина розміщується також у дата-центрі в Польші. Відображення клієнтської частини виконується швидше ніж за 200мс.

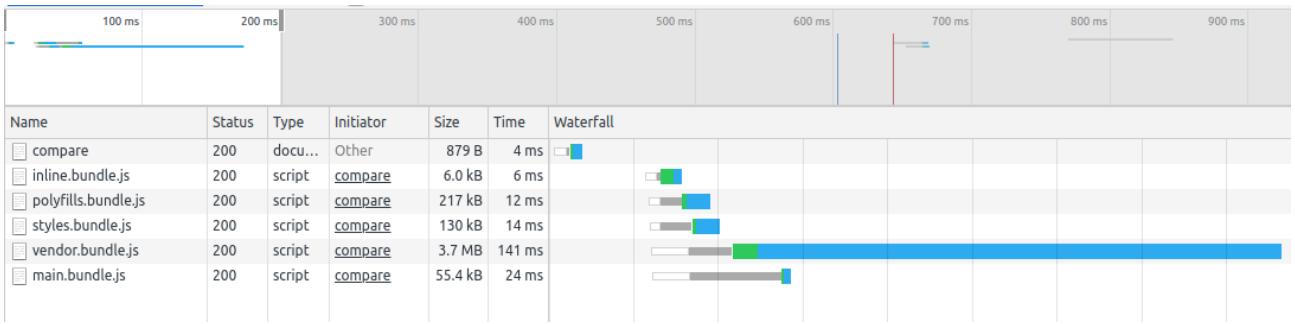


Рисунок 4.29 — Час завантаження SPA

Для подолання проблеми холодного старту лямбда-функції існує можливість підключити Provisioned Concurrency, але це потребує додаткової плати. Наприклад, якщо для функції було виділено 256Мб пам’яті, кількість паралельних викликів встановлена на 100, тривалість виконання функції приблизно 1сек і Provisioned Concurrency включений на 1місяць, то це буде коштувати 270доларів. У даному додатку для розігрівання лямбда-функцій було використано пакет “lambda-warning” і CloudWatch Event для пінгування лямбда-функцій.

```
const AWS = require('aws-sdk');
const dynamoDB = new AWS.DynamoDB({
  region: 'us-east-2',
  apiVersion: '2012-08-10'
});

const warmer = require('lambda-warmer')

const cisp = new AWS.CognitoIdentityServiceProvider({
  apiVersion: '2016-04-18'
});

exports.handler = async (event, context, callback) => {

  if (await warmer(event)) return 'warmed'
```

Рисунок 4.30 — Розігрівання лямбда-функції

Запуск розігрітих лямбда-функцій пришвидшив виконання запиту до 0.3546 — 0.614 сек. Для додатку, розміщеного в ECS середній час виконання запиту був приблизно 0.33-0.4 сек. Розрахуємо приблизну вартість використання усіх сервісів додатку.

1. API Gateway

У випадку 1млн запитів на день і розміром кешу у 6.1Gb місячна вартість буде дорівнювати 252.47 USD. У випадку високо-навантаженого

додатку в щоденною кількістю запитів у розмірі 10млн вартість буде дорівнювати 1210.6 USD.

2. DynamoDB

При виділенні 100GB пам’яті для збереження даних і його базовими налаштуваннями для операцій запису і зчитування даних, пікових навантажень і бекапом на 100GB обійдеться у 276.23USD у місяць.

3. S3

Для розміщення клієнтської частини необхідний один bucket, який обійдеться у 25USD на місяць.

4. Lambda

Виділимо для кожної лямбда-функції 512Mb оперативної пам’яті. Час виконання кожної функції приблизно 300мс. Розрахуємо приблизний кількість викликів функцій на місяць: при невеликому навантаженні в 1млн запитів на день і підігріванням лямбда-функцій обійдеться у 79.53USD,а у випадку великого навантаження у 10млн запитів на день — 843.63USD.

5. AWS Cognito

Ціна за додавання 1 користувача до User Pool-у дорівнює 0.0055USD, коли кількість користувачів у пулі не перевищує 100тис. Тому вартість за 100тис користувачів = 275USD.

Як бачимо, у випадку 1 млн запиту на день за місяць необхідно буде залпатити приблизно 908USD, а у випадку великого навантаження у 10млн запитів — 2630USD.

Також були розроблені 3 мікросервіси за допомогою Node.js і розгорнуті у AWS Elastic Container Service, який працює на серверах Linux. Клієнтська частина також зберігається у S3.

AWS Elastic Container Service включає в себе віртуальні машини EC2, які працюють постійно і Application Load Balancer дла балансування навантаження.

Розрахуємо вартість використання додатку, розгорнутого у Elastic Container Service. Як і для Serverless додатку використовувались такі AWS сервіси, як DynamoDB і S3 і ціна для них незмінна, і дорівнює 276.23USD і 25USD відповідно.

Розрахуємо вартість використання EC2. У якості операційної системи обрано Linux, 4 ядра CPU, 16Gb оперативної пам'яті і ssd на 30Gb. Для звичайного навантаження вистачить 3 інстансів EC2, щj сумарно буде коштувати 303USD. Для пікових навантажень можна збільшити кількість інстансів до 10, що буде коштувати 1012USD.

Також необхідно розрахувати вартість використання Elastic Load Balancer, а саме Application Load Balancer, яка дорівнює приблизно 58USD в місяць.

Як бачимо, для непікового навантаження вартість розгортання додатку у Elastic Beanstalk обійдеться у 937USD, а для пікового навантаження - 1646USD.

Провівши аналіз вартості використання двох додатків, можна зробити висновки, що для високо-навантажених додатків краще використовувати Elastic Container Service, тому що вартість буде майже в 2 рази дешевша. Для додатків з невеликим навантаженням у 1млн запитів на день вигідно використовувати Serverless рішення, якщо відмовитись від сервісу AWS Cognito і реалізувати систему аутентифікації власноруч, або якщо кількість унікальних користувачів не перевищую 50тис,тому що у цьому випадку AWS надає цю послугу безкоштовно.

Можна зробити висновок, що Serverless додатки не дуже добре підходять для високо-навантажених систем з малим часом затримки. Також треба уникати використання Serverless архітектури, коли потребується постійне з'єднання з лямбда-функції зі сторонніми сервісами. Завдяки тестам продуктивності можна точно налаштувати автоматичне масштабування для обробки навантаження. Наприклад, установити 5 інстансів EC2 і збільшувати їх

кількість, коли трафік збільшується. Щоб успішно обробляти навантаження у 1000 запитів в секнду, достатньо запустити 10 інстансів EC2(у порівнянні з 800-1000 одночасних лямбда-функцій).

Що стосується зручності написання і розгортання додатку, у хмарі, то простіше писати Serverless додатки, тому що розробник не буде піклуватися про інфраструктуру і налаштування, а лише реалізовувати бізнес-логіку за допомогою лямбда-функцій. Для реалізації додатку на основі контейнерів необхідно прописувати конфігураційні файли для контейнерів, і запускати їх за допомогою сервісів, які також необхідно конфігурувати самостійно.

4.5 Висновки

У даному розділі було представлено реалізації додатку, використовуючи безсерверну архітектуру і архітектуру на основі контейнерів. В обох випадках використовувались сервіси хмарного провайдера AWS. Для реалізації serverless додатку було обрано патерн “Simple Web Service”, який описує схему побудови REST API. Даний патерн доволі легко реалізується за допомогою сторонніх сервісів AWS.

Для розміщення додатку на основі контейнерів було використано AWS сервіс Elastic Container Service. Elastic Container Service - це спроба компанії Amazon інтегруватися в ринок управління контейнерів, де зараз існують Kubernetes, Mesos/Marathon, Docker Swarm. Однак, на відміну від них Amazon надає сервіс з API, таким чином найбільш близький аналог це Google Container Engine (Kubernetes-as-a-serivce). Варто відзначити, що сам ECS безкоштовний, а плата знімається тільки за екземпляри EC2 і балансувальник навантаження.

Було проведене порівняння у ціновому сегменті за допомогою сервісу AWS “калькулятор цін”. Порівняння показало, що для невеликих і середніх додатків з невеликим навантаженістю serverless додаток дешевше, ніж його аналог на основі контейнерів. У свою чергу при рості навантаження значно кращі результати показує додаток, розміщений у ECS. Щодо швидкодії, то ECS додаток також у середньому показує кращі результати, ніж Serverless додаток, навіть після його модифікації у вигляді розігрівання лямбда-функцій.

Що стосується архітектури, то serverless додаток значно легше розгортати і це потребує менше налаштувань, ніж ECS додаток. Розробник не переймається інфраструктурними питаннями, а лише зосереджений на написанні логіки у лямбда-функціях. ECS додаток також масштабує автоматично контейнери, але потребує певних налаштувань задач, які він буде розгортати у кластері. Логування і моніторинг також є більш гнучкими для serverless додатку з простішим інтерфейсом налаштування.

5 РОЗРОБКА СТАРТАП-ПРОЕКТУ

5.1 Опис ідеї проекту

Стартап-проект “FaaS як шаблон архітектури додатків” описано у Таблиці. 5.1.

Таблиця 5.1 - суть ідеї стартапу

Зміст ідеї	Напрямки застосування	Переваги для користувача
Використання FaaS архітектури для побудови додатків у хмарному середовищі	Розробники додатків, розмішених у хмарному середовищі	Простота написання додатків, відсутність конфігурації, зручне розгортання додатків
	Компанії, які розробляють додатки для замовників	Відсутність підтримки інфраструктурного рівня додатку, зменшена вартість, автоматичне масштабування

Даний стартап-проект буде цікавим як для розробників програмного забезпечення, так і для IT-компаній на початковому етапі проектування або побудови додатку з наступним його розгортанням у хмарі. Також концепція стартапу може бути використана при модифікації додатку і його переходу на новий хмарний провайдер або архітектуру.

Проведемо аналіз потенційних техніко-економічних переваг даної ідеї, порівнюючи з пропозиціями, які надають конкуренти. Попереднім колом конкурентів на ринку будемо вважати мікросервісну архітектуру, розміщену у хмарі і монолітну, яка розміщається як у хмарі, так і на власних серверах.

Таблиця 5.2 - Нейтральні, слабкі та сильні характеристики ідеї застосунку

№ п\п	Техніко-економічні характеристики	Потенційні конкуренти				W слабка сторона	N нейтральна сторона	S сильна сторона
		Мій проект	К-т 1	К-т 2	К-т 3			
1.	Архітектура	FaaS	Мікросервісна	Моноліт на	Монолітна		+	
2.	Спосіб розгортання	Хмара	Хмара	Хмара	Власний дата-центр			+
3.	Вартість використання	Сердня	Середня	Середня	Висока		+	
4.	Можливість авто-масштбування	Підтримується	Пітримується	Пітримується	Відсутня			+
5.	Необхідність конфігурації	Мінімальна	Середня	Середня	Висока			+

У Таблиці 5.2 наведений перелік таких характеристик, як спосіб розгортання, тип архітектури, вартість використання, можливість авто-масштабування додатку і необхідність його конфігурації.

5.2 Технологічний аудит проекту

Розглянемо технології, які потребує ідея проекту для її реалізації.

Таблиця 5.3 - технологічний аудит проекту

№ п\п	Ідея проекту	Технології	Наявність технології	Доступність технології
1	FaaS як шаблон побудови хмарних додатків	Хмарний провайдер AWS	Наявна	Платна
2		Фреймворк Angular.js	Наявна	Безкоштовна
3		Node.js	Наявна	Безкоштовна
4		Фреймворк Serverless	Наявна	Безкоштовна
5		Середовище розробки PHPStorm 2020.3	Наявна	Платна

FaaS додаток буде розроблено за допомогою Node,js з використанням фреймворку Serverless з його наступним розміщенням у хмарному провайдері AWS. Для розробки архітектури додатку необхідні такі сервіси AWS, як AWS Lambda, DynamoDB, Cognito, CloudFront, CloudWatch і Api Gateway.

5.3 Аналіз ринкових можливостей запуску стартап-проекту

Проведемо попередню характеристику потенційного ринку у Табл.5.4:

Таблиця 5.4 - Базова характеристика ринку стартапу

№ п\п	Показники ринку	Характеристика
1	Кількість головних гравців, од	3
2	Динаміка ринку (якісна оцінка)	Щорічне зростання
3	Наявність обмежень для входу	Існує певна кількість архітектур, які себе

		зарекомендували
4	Специфічні вимоги до стандартизації та сертифікації	Відсутні
5	Середня норма рентабельності в галузі (або по ринку), %	52%

Визначемо групи потенційних клієнтів стартап-проекту, які наведені у табл. 5.5.

Таблиця 5.5 - Характеристика потенційних клієнтів

№ п\п	Потреба, яка створює ринок	Цільова аудиторія	Відмінності у поведінці різних цільових груп	Вимоги споживачів до продукту
1	Узагальнена концепція, яка описує FaaS архітектуру побудови додатків у хмарних системах	IT-компанії і розробники, які займаються побудовою додатків з нуля	Відсутні	Архітектура повинна задовольняти вимогам додатку, таким як авто-маштабування, надійність, передбачуване ціноутворення, простота розробки

Після того, як були визначені потенційні групи клієнтів, опишемо фактори загрози, які наведені у табл. 5.6.

Таблиця 5.6 - Аналіз загроз

№ п\п	Фактор	Зміст загрози	Можлива реакція компанії
1	Існування конкурентів	Існує багато популярних архітектурних шаблонів розробки хмарних додатків	Постійний пошук засобів вдосконалення безсерверних архітекутрних підходів побудови додаіків
2	Зростання вартості використання сервісів від хмарних провайдерів	З часом хмарін провайдери можуть підіймати ціну на певні сервіси або видаляти їх, що вплине на популярність і доцільність використання архітектури	Модифікація патернів з урахуванням ціни і доступності сервісів хмарних провайдерів

Проаналізувавши загрози, розглянемо у табл. 5.7 фактори можливостей.

Таблиця 5.7 - Фактори можливостей

№ п\п	Фактор	Зміст можливості	Можлива реакція компанії
1	Зростання кількості клієнтів, які переходять на FaaS	При переході на безсерверну архітектуру клієнти будуть потребувати архітектурні	Виставити ціну на дану концепцію меншу за середню на світовому ринку

	архітектуру	патерни	
2	Поява нових клінтів, які планують створювати валсний стартап	Необхідність знаходження нових рішень з розвитку архітектури	Залучення нових архітекторів для розробки нових архітектур

Далі проведемо ступеневий аналіз існуючої конкуренції, який наведено в табл. 5.8.

Таблиця 5.8 - Ступеневий аналіз конкуренції на ринку

Назва характеристики	Особливість конкурентного середовища	В чому проявляється характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
Тип	Олігополія	Існує усього 3найголовніші типи архітектури: монолітна і класична мікросервісна на основі контенерів	Необхідно приділяти увагу до впровадження нових ідей для конкурентноспособності
Рівень конкурентної боротьби	Міжнародний	У всьому світі створюються нові типи архітектурних шаблонів	Створювати описання архітектурних патернів з урахуванням підтримки інших мов

Галузева ознака	Внутрішньо-галузева	Використовується додатками, які обирають безсерверну архітектуру розміщення у хмарі	Слідкувати за впровадженням нових сервісів, що надають хмарні провайдери
Вид товарів	Товарно-видова	Проста концепція створення програмного забезпечення	Всі конкуренти мають одну мету – надати архітектурне рішення, яке буде зручним і дешевим
Характер конкурентних переваг	Нецінова	Ринок потребує нові архітектурні рішення	Залучати архітекторів до розробки нових шаблонів
Інтенсивність	Марочна	Бренд має значення	Розкручувати бренд свого стартапу

Таблиця 5.9 – Аналіз конкуренції в галузі за М. Портером

Складові аналізу	Прямі конкуренти в галузі	Потенційні конкуренти	Фактор сили постачальників	Фактор сили споживачів	Фактор загроз з боку товарів-замінників
	Монолітна і класична мікросервісна	Змішані архітектури	Наявні	Побажання розробників	Відсутні
Висновки	На даний	Існують	Постачаль	Розробника	Фактор

	момент існує 2 архітектури, які є конкурентними: монолітна і класична мікросервісна	змішані архітектури, які можуть використовувати переваги обох концепцій	ники хмарних сервісів виставляють ціни на використання	м важлива простота і швидкість розробки, а також відсутність потреби роботи з інфраструктурою	загроз відсутній

Таблиця 5.10 - Обгрунтування факторів конкурентоспроможності

№ п\п	Фактор конкурентоспроможності	Обґрунтування (наведення чинників, що роблять фактор для порівняння конкурентних проектів начущим)
1	Створення концепції розробки хмарних додатків	Архітектурні патерни є еталоним рішенням, використовуюче найкращі практики для побудови додатків
2	Гнучка інтеграція з хмарними провайдерами	Хмарні провайдери забезпечують усю роботу з інфраструктурним рівнем додатку, розробник тільки створює бізнес-логіку

Таблиця 5.11 - Порівняльний аналіз сильних та слабких сторін проекту

№ п\п	Фактор конкурентоспроможності	Бали 1-20	Рейтинг товарів конкурентів у порівнянні з нашим						
			-3	-2	-1	0	+1	+2	+3
1	Створення концепції розробки хмарних додатків	18		+					
2	Гнучка інтеграція з хмарними провайдерами	15				+			

Таблиця 5.12 - SWOT-аналіз стартапу

S	Створення концепції розробки хмарних додатків, гнучка інтеграція з хмарними провайдерами	Платні сервіси AWS для створення FaaS додатку	W
O	Зростання кількості клієнтів, які переходять на FaaS архітектуру	Існування конкурентів, зростання вартості використання сервісів від хмарних провайдерів	T

Таблиця 5.13 - Альтернативи ринкового впровадження стартап-проекту

№ п\п	Альтернатива ринкової поведінки	Ймовірність отримання ресурсів	Строки реалізації
1	Написання статей про стартап з їх розміщенням на тематичні стайти	Середня	місяць
2	Виступ на втематичних конференціях	Висока	2-3 місяців

5.4 Розроблення ринкової стратегії проекту

Таблиця 5.14 - Вибір цільових груп потенційних користувачів

№ п\п	Опис профілю цільової групи потенційних клієнтів	Готовність споживачів сприйняти продукт	Орієнтовний попит в межах цільової групи	Інтенсивність конкуренції в сегменті	Простота входу в сегмент
1	Існуючі IT-компанії	Не всі готові	Більшість компаній вже використовують певні ахітектурні патерни	Висока	Надзвичайно складно
2	Нові клієнти або стартап-проекти	Готові	Потребують нові архітектурні рішення для побудови додатку	Висока	Існує помірна складність
Головною цільовою групою обираємо нових клієнтів, які тільки починають розробляти додаток, який буде розміщуватись у хмарному середовищі.					

Таблиця 5.15 - Визначення базової стратегії розвитку

№ п\п	Обрана альтернатива розвитку проекту	Стратегія охоплення ринку	Ключові конкурентоспроможні позиції відповідно до обраної альтернативи	Базова стратегія розвитку
1	Розвиток проекту буде здійснюватись за рахунку реклами на спеціалізованих ресурсах і виступи на конференціях з devops.	Реклама, яка буде зосереджена на нових клієнтах — розробниках додатків	Презентаціях архітектурних патернів на тематичних конференціях	Слідкувати за сервісами, які надають хмарні провайдери, які можуть бути впроваджені в архітектурні патерни

Таблиця 4.16 - Визначення базової стратегії конкурентної поведінки

№ п\п	«Першопроходець» на ринку	Агресивний пошук нових споживачів	Копіювання основних характеристик в конкурент	Стратегія конкурентної поведінки
1	Ні	Так	Відсутнє	Пропонування ефективних швидких архітектурних рішень побудови додатків

Таблиця 4.17 - Визначення стратегії позиціонування

№ п\п	Вимоги до товару цільової аудиторії	Базова стратегія розвитку	Ключові конкурентоспроможні позиції власного стартапу	Вибір асоціацій, які мають сформувати комплексну позицію власного проекту (три ключових)
1	Відсутність додаткової конфігурації системи, авто-масштабованість, низька вартість у використанні, структурованість	Залучення архітекторів для розробки нових архітектурних рішень	Низька вартість хмарних сервісів, необхідних для реалізації FaaS архітектури	Авто-масштабування, простота розробки

5.5 Розроблення маркетингової програми стартап-проекту

Таблиця 4.18 - Визначення ключових переваг концепції потенційного товару

№ п\п	Потреба	Вигода, яку пропонує продукт	Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити)
1	Авто-	Архітектура	Повна відсутність у

	масштабування	передбачає використання сервісів хмарних провайдерів, які забезпечують авто-масштабування	налаштуванні масштабування компонентів додатку
2	Дешева вартість сервісів хмарного провайдера, які необхідні для реалізації архітектури	Сервіси підібрані таким чином, щоб максимально зменшити вартість розміщення готового додатку у хмарі	Зменшення витрат на сервіси хмарних провайдерів

Таблиця 4.19 - Трирівнева модель товару

Рівні товару	Сутність та складові	
1 Товар за задумом	Концепція, яка покращує архітектуру додатку і зменшує вартість його розміщення у хмарі	
2 Товар у реальному виконанні	Властивості/характеристики	Час на реалізацію
	1 Патерн Strangler	2 тижні
	2 Патерн Publisher/Subscriber	6 тижнів
	Якість: архітектура розроблена за кращими стандартами	
	Пакування: шаблон можна використовувати, придбавши ліцензію	
3. Товар із підкріпленням	Проста концепція архітектури	
	Консультації з її використання	
Товар буде мати захист від копіювання за рахунок патенту		

Таблиця 4.20 - Визначення меж встановлення ціни

№ п\п	Рівень цін на товари-замінники, грн	Рівень цін на товари-аналоги, грн	Рівень доходів цільової групи споживачів, грн	Верхня та нижня межі встановлення ціни на товар/послугу, грн
1	Інформація відсутня	Інформація відсутня	Інформація відсутня	500-1000 $

Таблиця 4.21 - Формування системи збуту

№ п\п	Специфіка закупівельної поведінки цільових клієнтів	Функції збуту, які має виконувати постачальник товару	Глибина каналу збуту	Оптимальна система збуту
1	Використання шаблонів для побудови додатків	Консультації	Без посередників	Змішана

Таблиця 4.22 - Концепція маркетингових комунікацій

№ п\п	Специфіка поведінки цільових клієнтів	Канали комунікацій, якими користуються цільові клієнти	Ключові позиції, обрані для позиціонування	Завдання рекламного повідомлення	Концепція рекламного звернення
1	Розгортання	Інтернет	Авто-	Показ	Огляд

	додатків у хмарних провайдерах		масштабування, використовує бюджетні сервіси хмарних провайдерів	переваги FaaS архітектури	базових сервісів, які потребує архітектура для реалізації

5.6 Висновки

Під час розробки даного стартап-проекту, який є концепцією архітектурного FaaS патерну “Simple Web Service” було проведене порівняння з конкурентними рішеннями, які існують на ринку, а також проаналізовані як сильні сторони, так і слабкі. Існує багато конкурентних рішень, але незважаючи на це, розроблена концепція шаблону має чимало переваг, серед яких можна визначити спосіб розгортання додатку, який реалізує даний патерн, наявність можливості авто-масштабування, відсутність додаткових налаштувань конфігурації, дешева вартість сервісів, які потребує реалізація даного шаблону. В рамках дослідження було визначено основну цільову аудиторію, для якої розробляється концепція даного шаблону — це розробники програмного забезпечення і стартап-проекти, які потребуються нове архітектурне рішення при проектуванні свого додатку. Також для засобів, які необхідні для побудови продукту, відбувся технічний аудит. Для просування стартапу необхідний канал комунікації з потенційними клієнтами , в якості якого було обрано просування через публікації в електронних ресурсах і презентація в тематичних конференціях.

Завдяки виконанню даної розробки були отримані навичок у проектуванні і побудові стартап-проекту, визначення переваг і недоліків ідеї, знаходження цільової аудиторії і просування продукту.

ВИСНОВКИ

У даній магістерській роботі була досліджена тема — FaaS як шаблон архітектури додатків. На початку дослідження були розглянуті причини переходу побудови архітектури додатків від монолітної до мікросервісної, а згодом і безсерверної, а саме - необхідність автоматичної масштабованості, простота розгортання додатків і швидкість їх написання з подальшою підтримкою і впровадженням нового функціоналу. З огляду на це, безсерверна архітектура була обрана для подальшого дослідження. Далі були проаналізовані три найпопулярніші обчислювальні FaaS сервіси від хмарних провайдерів, такі як AWS Lambda, Azure Functions і Google Cloud Functions. Кожний сервіс має як свої переваги, так і недоліки, у зв'язку з чим кожний хмарний FaaS сервіс має свою область використання. Завдяки найкращій взаємодії зі сторонніми BaaS сервісами AWS, AWS Lambda значно виділяється у порівнянні з аналогічними рішеннями від Microsoft і Google. Широка інтеграція зі сторонніми сервісами, дешева і передбачувана плата за використання разом з найшвидшим часом роботи лямбда-функцій виділяють сервіс Lambda від провайдера AWS як найкращий поміж усіх FaaS сервісів, який і було обрано для подальших досліджень. Була розглянута проблема "холодного старту" лямбда-функцій і запропоновано два способи пришвидшення запуску функцій: за допомогою платного сервісу Provisioned Concurrency, який тримає віртуальні машини для запуску лямбда-функцій у "розігрітому" стані, і за допомогою пінгування лямбда-функції сервісом CloudWatch.

Окрім цього були розглянуті типові архітектурні патерни для побудови FaaS додатків на основі AWS провайдера. Існують шаблони усіх типових рішень, які застосовуються і для мікросервісної архітектури на основі контейнерів, такі як API Gateway, CQRS, Aggregator. Circuit Breaker, Strangler. Архітектура побудови Serverless додатків подібна до побудови на основі контейнерів. На основі дослідженого матераілу можна зробити висновок, що

хмарні провайдери надають можливість просто розбивати логіку мікросервісів на невеликі лямбда-функції, при цьому не надаючи суттєвих архітектурних змін для розробника. Розробник, який має певний досвід розробки мікросервісної архітектури не матиме жодних проблем при розробці архітектури для FaaS додатків.

У рамках подальшого дослідження були розглянуті 2 архітектури побудови додатку – serverless і мікросервісна на основі Docker контейнерів. Для цього використовувася архітектурний патерн “Simple Web Service”. API Serverless додатку було створене за допомогою BaaS сервісу API Gateway і FaaS сервісу AWS Lambda. Контейнери мікросервісного додатку розміщувалися у AWS Elastic Container Service і балансувалсь за допомогою Application Load Balancer. Після створення додатків було проведено аналіз вартості їх розміщення у хмарному провайдері AWS і швидкодії. З отриманих результатів було зроблено висновки, що у ціновому сегменті безсерверні архітектури краще підходять для малих і середніх додатків з невеликою навантаженістю, не більше 1млн. Запитів. У випадку високо-навантажених ентерпрайз додатків розміщення мікросервісів у контейнерах ECS буде дешевше приблизно в 2 рази. Також архітектура на лямбда-функціях має більший час затримки навіть при віикористанні “розігрітих” контейнерів для запуск функцій за рахунок API Gateway, який трансформує запит клієнта у вхідіні параметри лямбда-функції.

Serverless архітектура має перевагу у простоті її розгортання і налаштування, оскільки усі інфраструктурні питання з запуску і масштабування вирішує AWS Lambda, а розробник фокусується тільки на розробці бізнес-логіки додатку. Створювати serverless додатки можливо навіть у консолі розробника AWS. ECS повністю займається керуванням контейнерів, але при цьому вимагає певних налаштувань, таких як створення контейнерів, які запускаються у кластері, і створення сервісів, які запускають дані контейнери. З іншої сторони це надає розробнику більше можливостей для налаштувань додатку.

ПЕРЕЛІК ПОСИЛАНЬ

1. Solanki J. Evolution of Serverless: Monolithic-Microservices-FaaS [Електронний ресурс] / Jignesh Solanki. – 2017. – Режим доступу до ресурсу: https://dev.to/jignesh_simform/evolution-of-serverless-monolithic-microservices-faas-3hdp.
2. Another As A Service? What is the Difference Between IaaS, CaaS, PaaS, and FaaS? [Електронний ресурс]. – 2020. – Режим доступу до ресурсу: https://blog.iron.io/what-is-the-difference-between-iaas-caas-paas-and-faas/.
3. Solanki J. Serverless Architecture: A Comprehensive Guide [Електронний ресурс] / Jignesh Solanki. – 2017. – Режим доступу до ресурсу: https://www.simform.com/serverless-architecture-guide/.
4. Cloud: IaaS vs PaaS vs SaaS vs DaaS vs FaaS vs DBaaS [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://brainhub.eu/blog/cloud-architecture-saas-faas-xaas/.
5. What Is FaaS? (Function As A Service) [Електронний ресурс]. – 2018. – Режим доступу до ресурсу: https://dashbird.io/blog/what-is-faas-function-as-a-service/.
6. Roberts M. What is Serverless? Understanding the Latest Advances in Cloud and Service-Based Architecture / M. Roberts, J. Chapin., 2017. – 56 с. – (O'Reilly). – (First Edition).
7. Bashir F. What is Serverless Architecture? What are its Pros and Cons? [Електронний ресурс] / Faizan Bashir. – 2018. – Режим доступу до ресурсу: https://hackernoon.com/what-is-serverless-architecture-what-are-its-pros-and-cons-cc4b804022e9.
8. Roberts M. Serverless Architectures [Електронний ресурс] / Mike Roberts. – 2018. – Режим доступу до ресурсу: https://martinfowler.com/articles/serverless.html#WhatIsServerless.

9. Serverless [Електронний ресурс] – Режим доступу до ресурсу: https://aws.amazon.com/serverless/?nc1=h_ls.

10. Why use Serverless Computing? | Pros and Cons of Serverless [Електронний ресурс] – Режим доступу до ресурсу: https://www.cloudflare.com/learning/serverless/why-use-serverless/.

11. Feoktistov I. Introduction to Serverless Architecture in Cloud-based Applications [Електронний ресурс] / Ihor Feoktistov – Режим доступу до ресурсу: https://relevant.software/blog/serverless-architecture/.

12. Roberts M. Learning Lambda — Part 1 [Електронний ресурс] / Mike Roberts. – 2017. – Режим доступу до ресурсу: https://blog.symphonia.io/posts/2017-01-19_learning-lambda-part-1.

13. Akiwatkar R. AWS Lambda vs Azure Functions vs Google Cloud Functions: Comparing Serverless Providers [Електронний ресурс] / Rohit Akiwatkar. – 2019. – Режим доступу до ресурсу: https://www.simform.com/aws-lambda-vs-azure-functions-vs-google-functions/.

14. What is AWS Lambda? [Електронний ресурс] – Режим доступу до ресурсу: https://docs.aws.amazon.com/lambda/latest/dg/welcome.html.

15. Baird A. Serverless Architectures with AWS Lambda: Overview and Best Practices [Електронний ресурс] / Andrew Baird. – 2017. – Режим доступу до ресурсу: https://aws.amazon.com/blogs/architecture/serverless-architectures-with-aws-lambda-overview-and-best-practices/.

16. Abhishek K. Serverless Integration Design Patterns with Azure / K. Abhishek, M. Srinivasa. – Birmingham, UK, 2019. – 464 с. – (Packt).

17. Hornay R. Google Cloud Functions: An Introduction [Електронний ресурс] / Rachel Hornay. – 2020. – Режим доступу до ресурсу: https://www.bmc.com/blogs/google-cloud-functions/.

18. Irani R. Google Cloud Functions Tutorial : What is Google Cloud Functions? [Електронний ресурс] / Romin Irani. – 2018. – Режим доступу до ресурсу:

https://medium.com/@iromin/google-cloud-functions-tutorial-what-is-google-cloud-functions-8796fa07fc7a.

19. Tozzi C. How do I compare AWS Lambda vs. Azure Functions for serverless? [Електронний ресурс] / Chris Tozzi. – 2020. – Режим доступу до ресурсу: https://searchcloudcomputing.techtarget.com/answer/How-do-I-compare-AWS-Lambda-vs-Azure-Functions-for-serverless.

20. Daly J. Serverless Microservice Patterns for AWS [Електронний ресурс] / Jeremy Daly. – 2020. – Режим доступу до ресурсу: https://www.jeremydaly.com/serverless-microservice-patterns-for-aws/#.

21. Romero E. Serverless Architectural Patterns [Електронний ресурс] / Eduardo Romero. – 2019. – Режим доступу до ресурсу: https://medium.com/@eduardoromero/serverless-architectural-patterns-261d873020#c6

22. Patterson S. Learn AWS Serverless Computing / Scott Patterson., 2019. – 382 с. – (1st edition (December 24, 2019)).

23. Sbarski P. Serverless Architectures on AWS: With examples using AWS Lambda / Peter Sbarski., 2017. – 376 с. – (1st Edition).

24. Kumar A. Serverless Integration Design Patterns with Azure / A. Kumar, S. Mahendrakar., 2019. – 494 с.

25. Ranadheer raju D. ECS-(Amazon Elastic Container Service) [Електронний ресурс] / Ranadheer raju D. – 2020. – Режим доступу до ресурсу: https://medium.com/@ranadheerraju11/ecs-amazon-elastic-container-service-ebbb4be7b7dd.

26. Richardson C. Microservices Patterns / Chris Richardson., 2019. – 490 с.

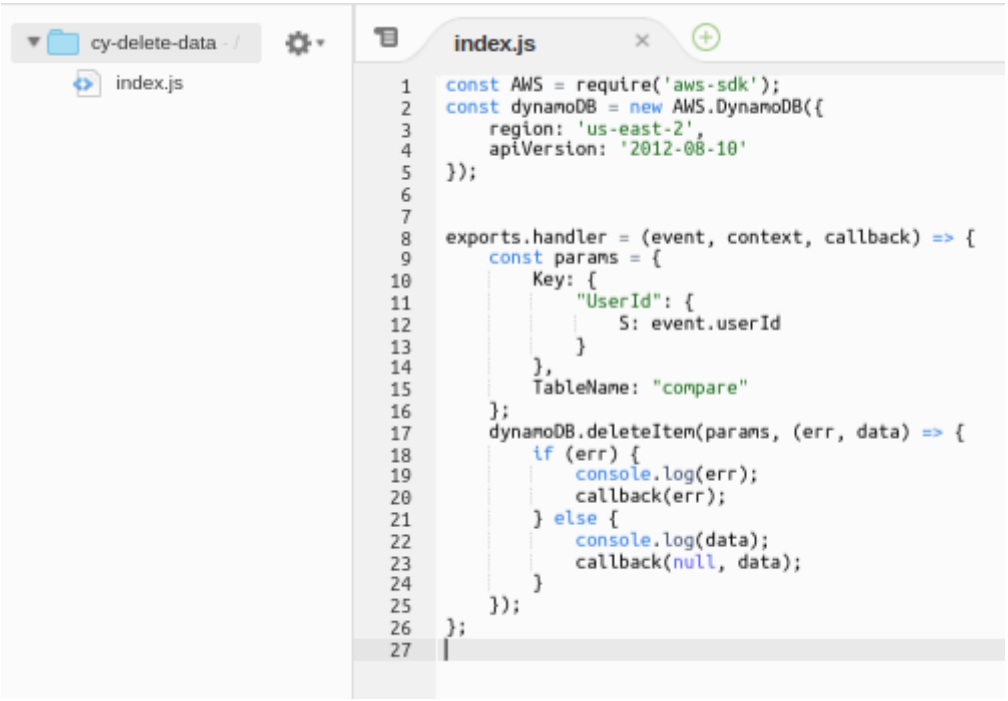
ДОДАТОК А

Реалізація API Serverless додатку

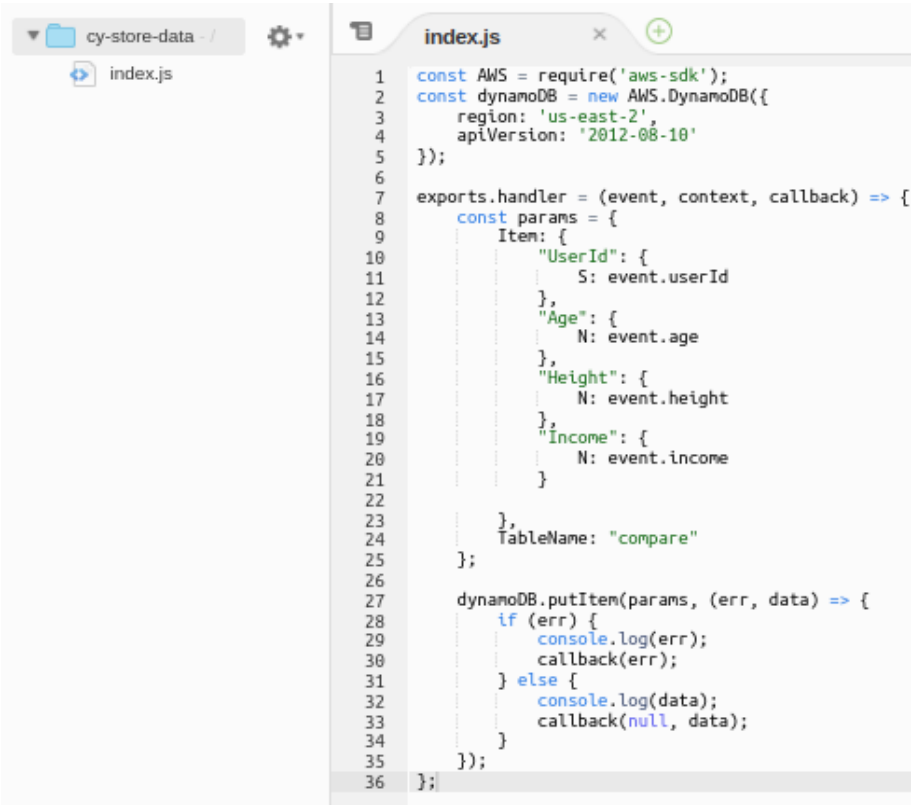
Текст програми

Листів 5

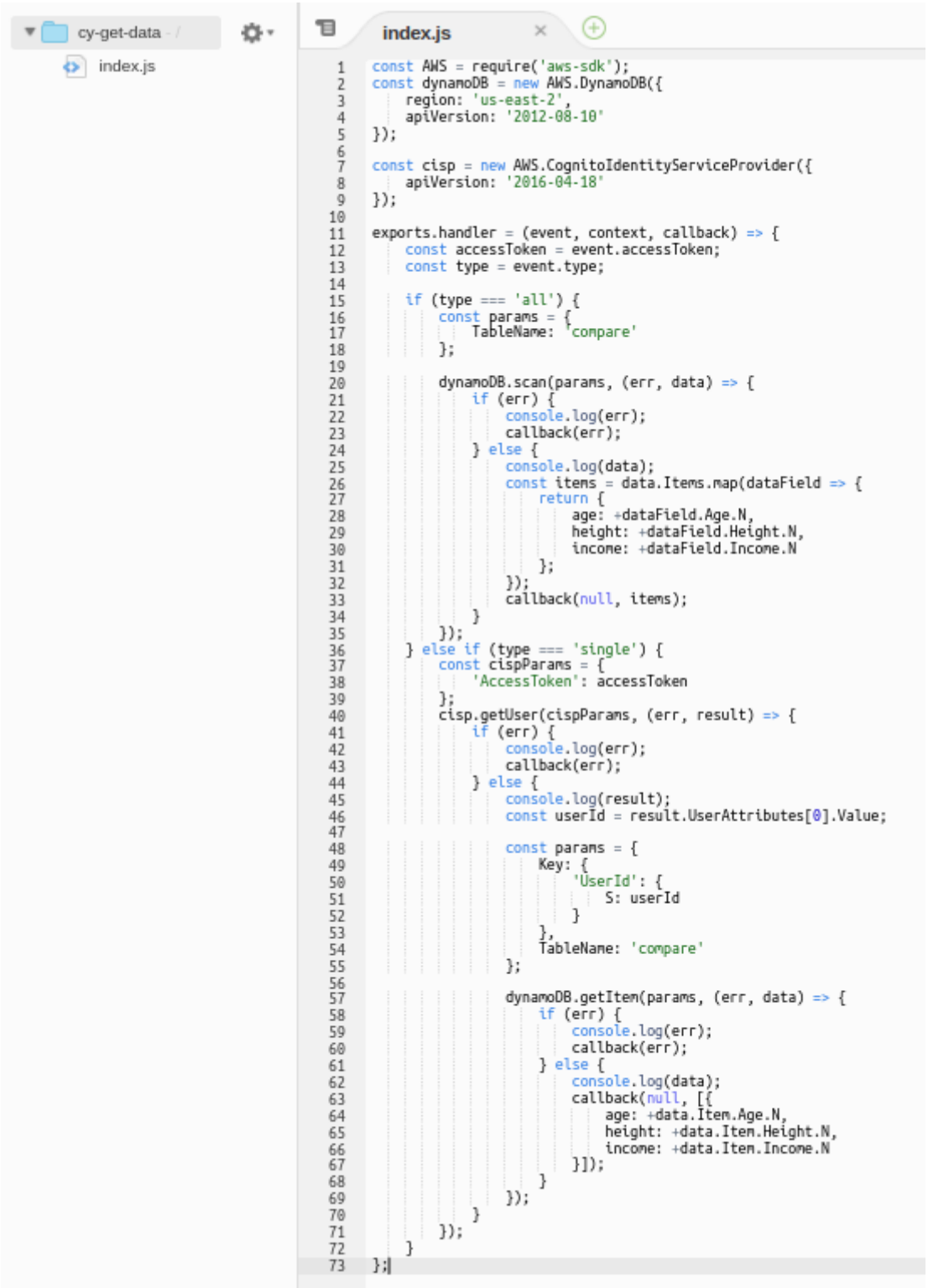
Лямбда-функція видалення даних користувача



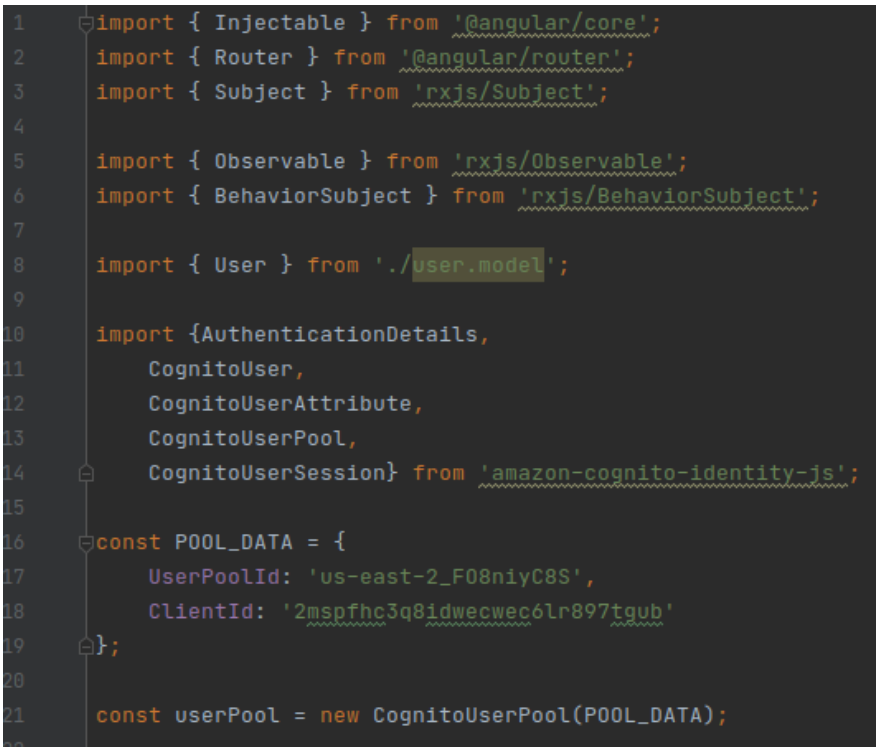
Лямбда-функція збереження даних користувача



Лямбда-функція отримання даних користувача



Сервіс аутентифікації



```
import { Injectable } from '@angular/core';
import { Router } from '@angular/router';
import { Subject } from 'rxjs/Subject';

import { Observable } from 'rxjs/Observable';
import { BehaviorSubject } from 'rxjs/BehaviorSubject';

import { User } from './user.model';

import {AuthenticationDetails,
    CognitoUser,
    CognitoUserAttribute,
    CognitoUserPool,
    CognitoUserSession} from 'amazon-cognito-identity-js';

const POOL_DATA = {
    UserPoolId: 'us-east-2_FO8niyC8S',
    ClientId: '2mspfhc3q8idwecwec6lr897tgub'
};

const userPool = new CognitoUserPool(POOL_DATA);
```

```
@Injectable()
export class AuthService {
    authIsLoading = new BehaviorSubject<boolean>(false);
    authDidFail = new BehaviorSubject<boolean>(false);
    authStatusChanged = new Subject<boolean>;
    registeredUser: CognitoUser;
    constructor(private router: Router) {}
signUp(username: string, email: string, password: string): void {
    this.authIsLoading.next(true);
    const user: User = {
    username: username,
    email: email,
    password: password
};
const attributeList: CognitoUserAttribute[] = [];
const emailAttribute = {
    Name: 'email',
    Value: user.email
};
attributeList.push(new CognitoUserAttribute(emailAttribute));
userPool.signUp(user.username, user.password, attributeList, null, (err, result) => {
    if (err) {
        this.authDidFail.next(true);
        this.authIsLoading.next(false);
        return;
    }
    this.authDidFail.next(false);
    this.authIsLoading.next(false);

    this.registeredUser = result.user;
});
}
```

```
55    confirmUser(username: string, code: string) {
56        this.authIsLoading.next(true);
57        const userData = {
58            Username: username,
59            Pool: userPool
60        };
61        const cognitoUser = new CognitoUser(userData);
62        cognitoUser.confirmRegistration(code, true, (err, result) => {
63            if (err) {
64                this.authDidFail.next(true);
65                this.authIsLoading.next(false);
66                return;
67            } else {
68                this.authDidFail.next(false);
69                this.authIsLoading.next(false);
70                this.router.navigate(['/']);
71            }
72        });
73    }
```

```
74      signIn(username: string, password: string): void {
75          this.authIsLoading.next(true);
76          const authData = {
77              Username: username,
78              Password: password
79          };
80          const authDetails = new AuthenticationDetails(authData);
81          const userData = {
82              Username: username,
83              Pool: userPool
84          };
85          const cognitoUser = new CognitoUser(userData);
86          const that = this;
87          cognitoUser.authenticateUser(authDetails, {
88              onSuccess(result: CognitoUserSession) {
89                  that.authStatusChanged.next(true);
90                  that.authDidFail.next(false);
91                  that.authIsLoading.next(false);
92                  console.log(result);
93              },
94              onFailure(err) {
95                  that.authDidFail.next(true);
96                  that.authIsLoading.next(false);
97                  console.log(err);
98              }
99          });
00          this.authStatusChanged.next(true);
01          return;
02      }
```

```
103     getAuthenticatedUser() {
104         return userPool.getCurrentUser();
105     }
106     logout() {
107         this.getAuthenticatedUser().signOut();
108         this.authStatusChanged.next(false);
109     }
110     isAuthenticated(): Observable<boolean> {
111         const user = this.getAuthenticatedUser();
112         const obs = Observable.create((observer) => {
113             if (!user) {
114                 observer.next(false);
115             } else {
116                 user.getSession((err, session) => {
117                     if (err) {
118                         observer.next(false);
119                     } else {
120                         if (session.isValid()) {
121                             observer.next(true);
122                         } else {
123                             observer.next(false);
124                         }
125                     }
126                 });
127             }
128             observer.complete();
129         });
130         return obs;
131     }
132     initAuth() {
133         this.isAuthenticated().subscribe(
134             (auth) => this.authStatusChanged.next(auth)
135         );
136     }
```