Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мережних технологій факультету інформатики

Кваліфікаційна робота

освітній ступінь — бакалавр

на тему: **«Побудова мережевого застосування з високою доступністю на хмарній платформі»**

Виконав: студент 4-го року
освітньої програми «Комп'ютерні науки»
спеціальності 122 «Комп'ютерні науки»
Росада Василь Васильович
Керівник: Черкасов Д.І.,
кандидат тех.наук, ст.викладач
Рецензент_____
Кваліфікаційна робота захищена з оцінкою _____
Секретар ЕК _____
«__»_____2023 р.

Київ 2023

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мережних технологій факультету інформатики

ЗАТВЕРДЖУЮ

Зав.кафедри мережних технологій,

проф, доктор фіз.-мат.наук

_____ Малашонок Г.І.

(підпис)

„____”_____2022 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на кваліфікаційну роботу

студенту 4 року навчання бакалаврської програми “Комп’ютерні науки”

Росаді Василю Васильовичу

на тему: «**Побудова мережевого застосування з високою доступністю на хмарній платформі»**

Зміст текстової частини до кваліфікаційної роботи:

Зміст
Анотація
Вступ
1. Огляд існуючих рішень
2. Проєктування структури власного рішення
3. Розробка серверного компонента застосунку
Висновки
Список використаних джерел
Додатки

Дата видачі: “___”_____2022 р.

Керівник: к.т.н. Черкасов Д.І. _____(підпис)

Завдання отримав: Росада В.В _____(підпис)

Тема: Побудова мережевого застосування з високою доступністю на хмарній платформі

Календарний план виконання роботи:

№ п/п	Назва етапу роботи	Термін виконання етапу	Примітка
1.	Отримання теми кваліфікаційної роботи	01.12.2022	
2.	Пошук і ознайомлення з джерелами	15.01.2023	
3.	Написання вступу	30.01.2023	
4.	Написання першого розділу	20.02.2023	
5.	Створення архітектури для застосунку	10.03.2023	
6.	Написання другого розділу	01.04.2023	
7.	Розробка серверного компонента застосунку	20.04.2023	
8.	Написання третього розділу та висновків	15.05.2023	
9.	Захист кваліфікаційної роботи	30.05.2023	

Студент _____

Керівник _____

“____”_____

Зміст

Анотація

В роботі на прикладі новинного ресурсу розглянуті підходи до побудови мережевого застосування з високою доступністю, яке призначене для використання користувачами з широкою географією. Базою для розробки обрано хмарну платформу AWS.

В рамках досліджень виконано аналіз технічних рішень і технологій, які можуть бути використані для виконання вимог.

Розробку системи виконано на структурному рівні, визначено перелік базових компонентів і продуктів, описані типові сценарії функціювання системи. Детально розглянуто розробку шлюза API та сценаріїв створення та перегляду новин.

Вступ

Сучасні вебзастосунки мають складну багатокомпонентну архітектуру, в якій кожен компонент відповідає за певну функціональність застосунку, наприклад: інтерфейс користувача, базу даних, частина бізнес-логіки й т.п. Ці компоненти можуть знаходитися на розподілених мережевих ресурсах і взаємодіють між собою через мережу. Такий підхід забезпечує такі переваги як масштабування, оптимальний розподіл навантаження, відмовостійкість, додавання нового функціоналу для застосування чи зміни наявного.

Доступність застосунку – здатність безперервно виконувати закладені функції, значною мірою впливає на формування позитивного чи негативного досвіду користувача. Методами підвищення доступності застосунку є дублювання (резервування) компонентів, балансування навантаження обчислювальних ресурсів, тощо. Для цього потрібно забезпечити надійність, стабільність, безпеку фізичної інфраструктури. У разі створення виділеної апаратної платформи для розгортання застосунку потрібно виділяти велику кількість матеріальних та людських ресурсів. Також потрібен і час на побудову подібної інфраструктури, яка матиме складнощі з масштабуванням, оскільки потребуватиме пропорційного збільшення закладених ресурсів.

Проте є альтернатива виділеній інфраструктурі – це хмарні платформи. Хмарні провайдери позбавляють користувачів клопоту щодо фізичної інфраструктури та надають її в зручному вигляді для користувача. Найважливіші переваги хмари:

- Ви платите лише за ті ресурси, які використовуєте (pay-as-you-go).
- Швидке розгортання, а також можливість гнучкого і швидкого масштабування, як в бік збільшення залучених ресурсів, так і в бік зменшення відповідно до реального навантаження.
- Хмарні платформи надають доступ до готових функціональних компонентів, які можуть бути використані для побудови застосування: DNS сервіси, сховища даних, віртуальні машини тощо.

- Сервіси хмари, які використовують для розгортання застосунку мають дуже велику надійність, яку складно досягти власними силами.

В результаті розробники мають можливість абстрагуватися від проблем підтримки власної інфраструктури та більше зосередитись на розробці функціональності продукту.

Сьогодні утримання власного дата-центру через повномасштабну війну, яка привела до руйнувань інфраструктури та відключенням електроенергії стало складнішим та несе більше ризиків. Тому перехід на хмарну платформу набув ще більшої актуальності. Наприклад, після початку повномасштабного вторгнення всі критичні реєстри держави були перенесені на хмарну платформу AWS.

Метою даної роботи є огляд можливостей хмарної платформи та їх використання для побудови сайту новин. З огляду на події у світі та в країні зросла роль новинних ресурсів. Такі ресурси містять багато мультимедійного контенту та аудиторію з усього світу, для якої важливо мати швидкий та надійний доступ до новин. Тому потрібно, щоб ресурс, який надає цей контент фізично знаходився ближче до користувачів.

Для новинних ресурсів характерна вибухова (не рівномірна) активність користувачів. Наприклад, таке відбувається при публікації «гарячої» новини. Щоб справитись з таким навантаженням, потрібно мати доступ до гнучкого й автоматизованого керування обсягом залучених ресурсів. Такі вимоги до застосування чудово узгоджуються з особливостями хмарних платформ.

1. Огляд існуючих рішень

В цьому розділі будуть розглянуті та порівняні сервіси хмарних провайдерів для побудови мережевого застосування з високою доступністю. Сьогодні лідерами на ринку хмарних технологій є Amazon Web Services (AWS), Google Cloud та Microsoft Azure. Популярні хмарні провайдери надають схожий спектр послуг. Тому достатньо розглянути одну платформу хмарних обчислень. Для цієї роботи вибрана платформа AWS.

1.1. Обчислювальні сервіси

AWS надає багато варіантів обчислювальних сервісів,[1] які можна поділити на такі типи послуг: [3]

1. Infrastructure as a Service (IaaS)
2. Container as a Service (CaaS)
3. Function as a Service (FaaS)

Infrastructure as a Service

AWS дозволяє створити, налаштувати та повноцінно використовувати віртуальну машину, яку називають Amazon Elastic Compute Cloud, або EC2 [2]. Ця послуга належить до типу послуг IaaS. Ця модель полягає в тому, що хмарний провайдер оперує фізичною інфраструктурою та гіпервізором, а всім іншим керує користувач.

Провайдер пропонує декілька способів нарахування оплати за екземпляри EC2. Деякі з них:

- On-Demand – плата нараховується за години або секунди в залежності від типу віртуальної машини.
- Reserved – оренда EC2 на рік або три роки. Якщо порівнювати з On-Demand дозволяє заощадити до 72%.

- Spot – використання тих вільних в інфраструктурі AWS віртуальних машин. Це дозволяє зекономити до 90%, проте, коли попит на EC2 зросте, то екземпляр можуть зупинити.

В порівнянні з іншими типами послуг, то при використанні IaaS відмовостійкість, яка забезпечує високу доступність, потребує більше ресурсів на налаштування та сильно залежить від проектування застосунку. Платформа надає сервіси для того, щоб забезпечити високу доступність. За допомогою сервісу Auto Scalling можна налаштувати автоматичне масштабування сервісу. Цей сервіс швидко запускає та призупиняє екземпляри в залежності від поточного рівня навантаження та вибраних налаштувань. В налаштування сервісу можна вказати мінімальну кількість екземплярів EC2, що одночасно працюють, що підвищує доступність.

Може виникнути ситуація, коли велика частина трафіка буде припадати на один екземпляр, через що він може бути перевантаженим, а інші екземпляри будуть мати багато вільних ресурсів. Для того, щоб цього уникнути і рівномірно розподіляти трафік серед запущених екземплярів EC2 можна використати сервіс Elastic Load Balancer.

Хорошою практикою є запуск екземплярів EC2 в різних зонах доступності (Availability zones). В географічному регіоні інфраструктура провайдера має декілька ізольованих локацій, які називаються зонами доступності. Є можливість, що з різних причин, наприклад через катаклізм, зона стає недоступною. В цьому випадку, якщо застосунок розгорнутий в декількох зонах доступності, то він буде доступним для користувачів.

Важливо, що параметри масштабування EC2 можна налаштовувати.

Переваги EC2:

1. Можливість вибору типу віртуальної машини який найкраще підійде для застосунку. Наприклад, D3 оптимізований для застосунків, які потребують високої пропускної здатності диска.

2. Високий рівень контролю, що дозволяє повністю налаштовувати середовище виконання застосунку.
3. Краща безпека внаслідок більшої ізоляції та можливості налаштування під власні вимоги безпеки.

Недоліки:

1. Потребує більше часу на налаштування та підтримку, ніж інші сервіси, що сповільнює розробку.
2. Споживає надлишкові обчислювальні ресурси для віртуалізації.
3. Складніше розгортання та оновлення застосунків

Container as a Service

Інший тип послуг це Container as a Service. Ці послуги дозволяють запускати Docker-контейнери, які містять в собі код програми та всі потрібні залежності для її запуску. Це дозволяє в середовищі, яке підтримує контейнери, запускати програму без налаштування середовища для її запуску. [4]

AWS пропонує різні сервіси для контейнерів [5]. Наприклад, для оркестрації контейнерів є Amazon Elastic Container Service (ECS) та Amazon Elastic Kubernetes Service (EKS). ECS краще інтегрований з хмарною платформою, а EKS дозволяє використовувати Kubernetes, яка є поширеною системою оркестрації контейнерів.

Для запуску контейнерів ці рішення можуть використовувати EC2 та Fargate. Fargate є безсерверним (serverless) рішенням. Це означає, що для запуску контейнерів не потрібно налаштовувати та конфігурувати сервери, як у випадку з EC2. Це дозволяє користувачу зосередити більше ресурсів на розробці застосунку.

Сервіси ECS та EKS надають чудові можливості для забезпечення високої доступності застосунку. Вони самі по собі забезпечують високу доступність, наприклад, автоматично відновлюють роботу контейнера, який через неполадки припинив свою роботу. Крім, того контейнери більш легковісні, ніж віртуальні машини, тому у випадку потреби запуску контейнера, це відбудеться швидше.

ECS та EKS так само, як і EC2 інтегруються з Elastic Load Balancer, а також дозволяють збільшувати доступність застосунку розгортаючи його в різних зонах доступності. Проте ці сервіси надають гнучкіші та зручніші за EC2 інструменти для налаштування масштабування. Це дозволяє забезпечити краще використання ресурсів при високій відмовостійкості. Також, при використанні цих сервісів оновлення застосунку чи його компонента можна зручно зробити без потреби зупинки його роботи.

Якщо система гнучкої оркестрації не потрібна, можна використати Elastic Beanstalk та App Runner. Ці сервіси для запуску контейнерів потребують менше контролю та налаштувань.

Переваги контейнерів:

1. Ефективніше використання обчислювальних ресурсів ніж у віртуальних машин.
2. Легко перенести в інше середовище виконання. Наприклад, з традиційного дата-центру в хмару.
3. Швидке та гнучке масштабування

Недоліки:

1. Менше контролю в порівнянні з EC2.
2. Менш безпечні, через гіршу ізольованість.

Function as a Service

FaaS, або serverless сервіси зручні тим, що користувачам не потрібно перейматись налаштуванням серверів, їх запуском, масштабуванням. Всі ці задачі за вас виконує хмарний провайдер. Для serverless обчислень AWS пропонує сервіс Lambda. [6]

З Lambda потрібно лише завантажити код і сервіс створить Lambda функцію. Для цієї функції можна вибрати пам’ять та виділену потужність процесора. Після цього до цієї функції можна прив’язати подію, що запустить її. Ця подія може бути створена сервісом AWS, наприклад шлюзом, базою даних чи

створена користувачем. Плата нараховується тільки за виконані функції та залежить від вибраних потужностей і того скільки виконується функція.

Сервіс забезпечує високу доступність Lambda функцій, без потреби додаткових налаштувань подібно до IaaS та CaaS. Наприклад, стандартно для Lambda функцій використовуються різні зони доступності.

Провайдер автоматично масштабує Lambda функції, проте варто взяти до уваги, що є обмеження кількості функцій, які працюють одночасно. Це обмеження залежить від регіону і становить приблизно 1000 функцій, проте при потребі можна запросити у провайдера збільшення цього ліміту для себе. Якщо функція з якихось причин не змогла виконатись повністю, провайдер автоматично два рази спробує повторити її виконання.

У випадку коли Lambda функцію рідко використовують протягом якогось часу, то при її виклику можливі затримкі її виконання (cold starts) [7]. Для того, щоб цього уникнути можна зарезервувати потужності Lambda, але це потребує додаткової плати.

Lambda не підходить для довгого виконання, оскільки максимальний час виконання – 15 хвилин.

Переваги Lambda:

1. Плата тільки за виконаний код, що може бути вигідніше ніж інші сервіси. (pay-per-use)
2. Автоматичне масштабування.
3. Не потрібно витрачати ресурси на налаштовування інфраструктури

Недоліки:

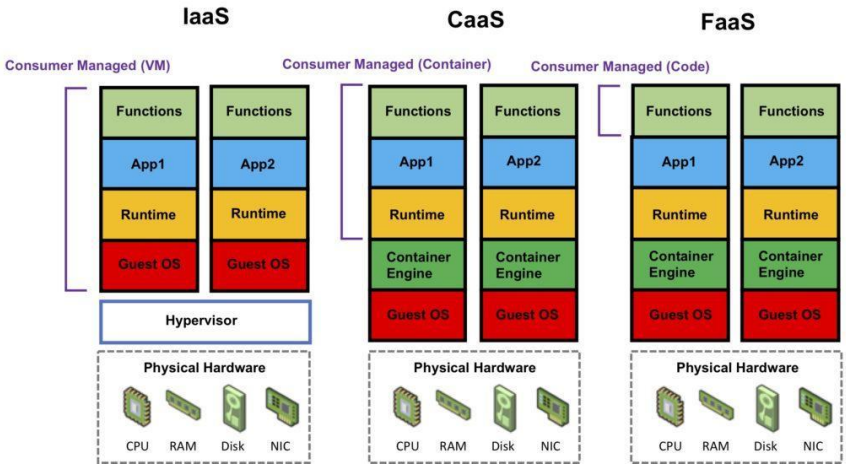
1. При нечастому використанні функції можливі затримки початку її виконання (cold start).
2. Обмежена кількість підтримуваних мов програмування.
3. Обмеження розміру коду та виділених обчислювальних потужностей для функції.

Вибір обчислювального сервісу

Типи сервісів мають різні особливості (Таблиця 1). Вибрати сервіс можна, наприклад, від потрібного рівню контролю над інфраструктурою (Рис 1.1). В залежності від ситуації це може бути, як перевагою, так і недоліком.

Таблиця 1.1. Порівняння IaaS, CaaS, FaaS

Особливості	IaaS (EC2)	CaaS (ECS)	FaaS (Lambda)
Рівень контролю	Високий	Частковий	Низький, контролюється провайдером
Оплата	Залежить від типу EC2 та часу виконання	Аналогічно до IaaS, або від вибраних vCPU та пам'яті та часу виконання	Залежить від часу виконання та кількості функцій
Середовище виконання	Залежить від вибраної ОС	Залежить від контейнера	Обмежено сервісом
Час виконання	Необмежений	Необмежений	Максимум 15 хвилин
Час запуску	Відносно довго	Відносно швидко	Швидко, проте можливі cold starts
Масштабування	Налаштовується	Налаштовується	Виконується сервісом без налаштувань
Доступність для застосунку	Залежить від проектування застосунку	Залежить від проектування застосунку	Залежить від провайдера
Очікувана вбудована доступність	99.95%	99.9%	99.99%



Рисинук 1.1. Рівень контролю, який надають різні типи сервісів [3]

Рішення може залежати від того наскільки важливим є портативність застосунку і можливість його запуску на інших платформах. В такому випадку контейнери підійдуть найкраще, а перенести serverless застосунок буде складніше.

Або від того, яке навантаження очікується. Наприклад, якщо навантаження має бути невеликим і застосунок має опрацьовувати невелику кількість запитів, то тоді доцільно з точки зору бюджету вибрати Lamdba сервіс. Проте, якщо критичним є те, щоб не холодних стартів, то це рішення може не підійти.

Існує безліч факторів спираючись на які можна вибрати сервіс. При потребі та можливості можна використати різні технології для одного застосунку. Це дозволить використати переваги та компенсувати недоліки сервісів. Наприклад, для постійних запитів можна використати віртуальні машини чи контейнери, а для епізодичних функцій можна використати Lambda.

1.2. Доступ по широкій географії

Для користувачів важливо, щоб вони могли отримати якісний доступ до вебзастосунку. Якщо користувачі розподілені по різних регіонах світу, то задача забезпечення якісного доступу стає складнішою. Бо у випадку, коли користувач розташований далеко від сервера, то затримка під час користування застосунком може бути відчутною для нього. AWS завдяки своїй інфраструктурі, яка

розміщена по всьому світу і використовує власне швидше за звичайне з'єднання, пропонує сервіси, які можуть розв'язати цю проблему.

Система доменних імен

Система доменних імен (DNS) ставить у відповідність домену (наприклад www.google.com) IP-адресу сервера. В AWS є високодоступний та масштабований DNS сервіс – Route 53 [9] (Рис 1.2).

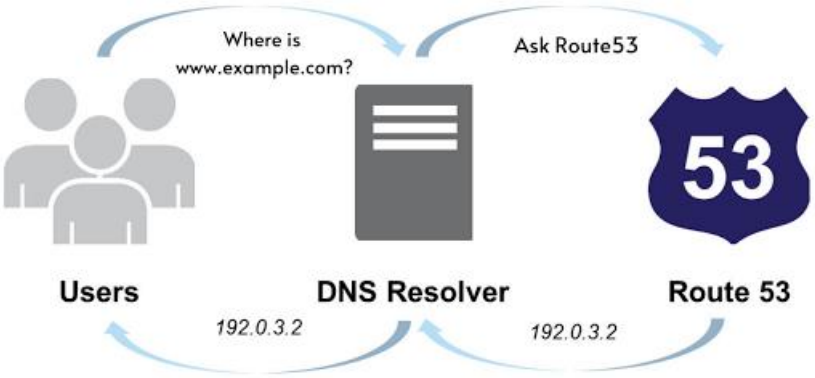


Рисунок 1. 2. Приклад запиту до Route 53

За допомогою цього сервісу можна реєструвати та керувати доменними іменами, а також перенести доменні імені з інших реєстраторів.

Для покращення доступу до застосунку по широкій географії Route 53 дозволяє розподіляти трафік до ресурсів, які розміщені в різних регіонах. При цьому він надає можливість розподіляти трафік за різними критеріями за допомогою політик маршрутизації. Деякі з політик маршрутизації:

- Weighted routing – дозволяє розподіляти трафік між багатьма ресурсами та налаштовувати те, як багато трафіку буде отримувати кожен ресурс.
- Failover routing – дозволяє визначити основний ресурс на який буде направлятись весь трафік, а також додаткові ресурси, які візьмуть на себе трафік, якщо основний ресурс перестане працювати.
- Geolocation routing – дозволяє визначити ресурси для трафіку, в залежності від того в якому географічному регіоні знаходяться користувачі.
- Geoproximity routing – також дозволяє розподіляти трафік в залежності від регіону користувача та сервера. Проте ця політика надає можливість

визначати розміри самих регіонів, що може знадобитись для більш рівномірного розподілу трафіку на сервери.

- Latency-based routing – розподіляє трафік між ресурсами за критерієм найнижчої затримки. Route 53 вимірює до якого ресурсу у користувача буде найнижча затримка і перенаправляє його трафік саме туди. Ця дозволяє найкраще розподілити трафік між ресурсами у різних регіонах, якщо стоїть задача максимально знизити затримку між застосунком і користувачем.

Route 53 також дозволяє перевіряти стан доступності ресурсу, наприклад вебсервера, за допомогою health checks, і у випадку якщо ресурс недоступний за допомогою політик маршрутизації оминати ресурси, які недоступні. Для випадків коли ресурс перестає працювати можна налаштувати сповіщення. [10]

Доступні типи health checks:

1. Моніторинг стану кінцевих точок, які зазначає користувач.
2. Health checks, які відстежують стани інших health checks. Це може бути корисно, наприклад, коли потрібно моніторити чи доступна мінімальна кількість здорових ресурсів.
3. Health checks, які відстежують попередження моніторингового сервісу AWS CloudWatch.

За допомогою цього функціоналу моніторингу стану ресурсів можна підвищити відмовостійкість всього застосунку.

Плата за користування Route 53 нараховується за:

- Кількість використаних hosted zones, що зберігають правила маршрутизації домену
- Кількість DNS запитів
- Щорічна плата за доменне ім’я.

Мережа доставки контенту

Мережа доставки контенту (CDN) представлена в AWS сервісом CloudFront. Цей сервіс швидко допомагає швидко надсилати контент користувачу (наприклад зображення, html, js, css файли). [11]

Спочатку розробнику потрібно мати сервер звідки CloudFront буде брати контент. Наприклад, це може бути сховище S3 чи http сервер запущений на EC2. При налаштуваннях CloudFront потрібно вказати джерело походження контенту.

Сервіс для своєї роботи використовує розміщені по всьому світу дата-центри, які в інфраструктурі AWS називаються edge локаціями. Коли користувач буде робити запит до застосунку до якого підключений CloudFront, запит направиться на найближчий до нього edge локації.

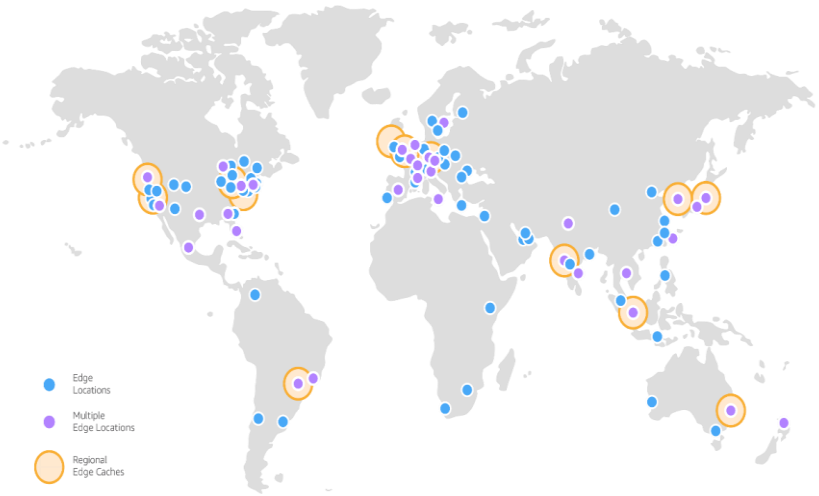


Рисунок 1.3. Карта edge локацій [12]

Якщо контент який потрібен користувачу вже там знаходиться, то контент відразу буде швидко доставлений користувачу. Якщо контенту там немає, то CloudFront робить запит вже до вказаного сервера, що буде навіть при цьому швидше, ніж прямий запит до сервера, оскільки інфраструктура AWS з'єднана між собою дуже швидкою приватною мережею. Потім CloudFront зберігає контент з сервера на edge локації та відправляє його користувачу. І якщо відбудеться наступний запит на цей контент, то він вже буде відразу відправлений з edge локації.

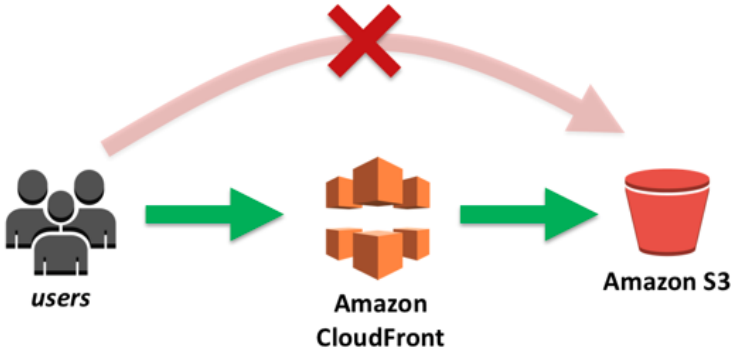


Рисунок 1.4. Запит користувачів до CloudFront

Завдяки цьому CloudFront дозволяє значно покращити доступ до даних застосунку для користувачів. Цей сервіс є високодоступним, якщо щось станеться з якоюсь edge локацією, то інші будуть доступні. За допомогою кешування контенту навантаження на сервер зменшиться, а також у разі недоступності сервера, кешований контент буде доступний для користувача. Це підвищує відмовостійкість самого застосунку.

Шляхом кешування та використання edge локацій CloudFront може допомогти в захисті від ddos атак. Також за замовчуванням для захисту від ddos атак сервіс використовує AWS Shield Standard. При потребі для додаткового захисту можна приєднати інші сервіси безпеки, які пропонує AWS. Наприклад, AWS Web Application Firewall (WAF), який допоможе контролювати доступ до застосунку та захиститись від різних вразливостей. [13]

Також CloudFront дозволяє використовувати Lambda@Edge. Це Lambda функції, які запускають на edge локаціях. За допомогою них можна підвищити швидкість виконання запиту, впроваджувати додатковий захист, змінювати контент відповідно до локації тощо.

Плата за використання CloudFront нараховується тільки за використані ресурси, тобто по моделі pay as you go.

Глобальний прискорювач.

Глобальний прискорювач, він же AWS Global Accelerator дозволяє пришвидшити з'єднання між застосунком та користувачем за допомогою прискорювачів (accelerator), що використовують глобальну мережеву інфраструктуру AWS. Чудово підходить для застосунків, які застосунків, які потребують з'єднання не через HTTP протокол. [14]

Прискорювач за замовчуванням надає 2 статичні Anycast iPv4 адреси, які будуть слугувати точками підключення до застосунку. Під час його налаштування можна визначити кінцеві ресурси на які буде перенаправлятись трафік після того, як він дійде до edge локації. Ці ресурси можуть бути в одному або декількох регіонах. Шлях трафіку визначається подібно до CloudFront, тобто спочатку запит направляється до найближчої edge локації, а потім вже у швидкій мережі AWS направляється на кінцевий ресурс.

Є два типи прискорювачів:

1. Standard accelerator – направляє трафік до оптимального кінцевого ресурсу. Оптимальність визначається станом ресурсу, розташуванням користувача і вагами, які присвоєні ресурсу. Кінцевими ресурсами можуть бути:
    - EC2
    - Elastic IP адреса
    - Мережевий балансувальник навантаження (Network Load Balancer)
    - Балансувальник навантаження програми (Application Load Balancer)
2. Custom accelerator – дозволяє направляти трафік багатьох користувачів до конкретного екземпляра EC2. Наприклад, це можна знадобитись, якщо в ігровому застосунку потрібно приєднати гравців до конкретного ігрового сервера. Для цього типу для кінцевих ресурсів доступні тільки підмережі Amazon Virtual Private Cloud (VPC) в яких є екземпляри EC2.

Сервіс допомагає зробити застосунок більш високодоступним, оскільки використовує глобальну мережеву інфраструктуру провайдера та направляє трафік на ресурси які працюють. Це досягається тим, що сервіс перевіряє стан всіх зазначених кінцевих ресурсів.

Як і CloudFront за замовчуванням використовує AWS Shield для захисту від ddos атак та дозволяє приєднати інші сервіси безпеки, наприклад WAF. [15]

AWS Global Accelerator чудово підходить для забезпечення якісного доступу по хорошій географії. Крім того, що цей сервіс зменшує затримки за швидкої мережі провайдера, він допомагає оптимально розподіляти по ресурсах, які знаходяться в різних регіонах. Ці переваги допомагають суттєво пришвидшити запити до застосунку для користувачів, які знаходяться в віддалених регіонах. AWS пропонує перевірити прискорення запиту до ресурсів в інших регіонах за допомогою сторінки [AWS Global Accelerator Speed Comparison.](#) Наприклад, на рисунку 1.5. швидкість передачі 100KB файлу до регіону North Virginia (us-east-1) на 27% відсотків більша, ніж за допомогою звичайного інтернет з'єднання.

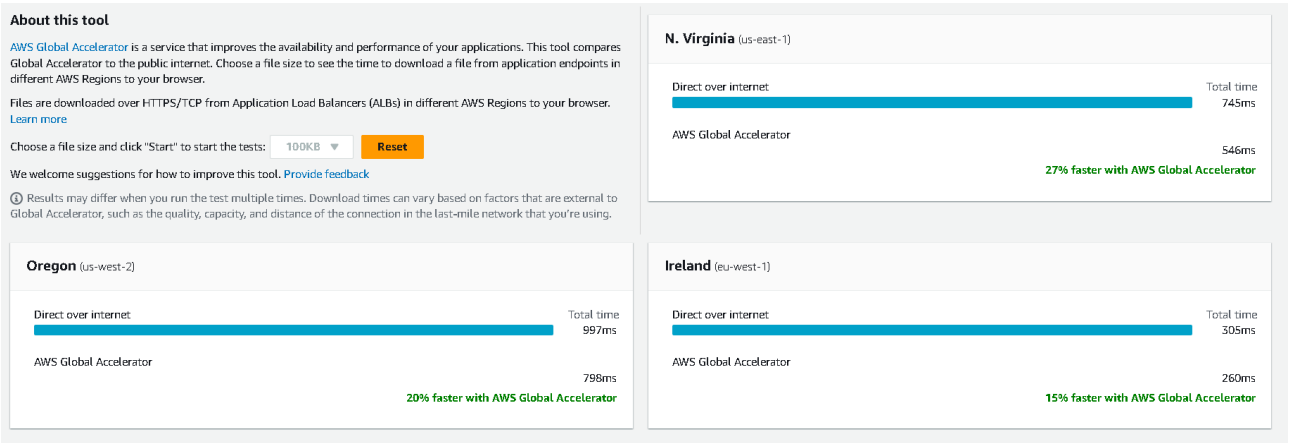


Рисунок 1.5. Перевірка швидкості Global Accelerator

Плата за використання сервісу нараховується за кількість прискорювачів, ціна за які фіксована та трафік який через них проходить.

Вибір рішення

Вибір між сервісами Route 53, CloudFront та Global Accelerator для забезпечення доступу по широкій географії залежить від застосунку та того, якого функціоналу він потребує.

У кожного сервісу є своє призначення, переваги та недоліки (Таблиця 2). Route 53 надає змогу керувати доменними іменами, вибирати потрібні політики маршрутизації та моніторити стан ресурсів. Це дозволяє автоматично перенаправляти трафік в оптимальний кінцевий ресурс, який працює. Проте цей сервіс не надає змоги кешувати дані та використовувати глобальну мережеву інфраструктуру провайдера для покращення передачі даних між користувачем та застосунком.

Таблиця 1.2. Порівняння Route 53, CloudFront, Global Accelerator

	Route 53	CloudFront	Global Accelerator
Керування доменними іменами	Присутнє	Відсутнє	Відсутнє
Маршрутизація трафіку до ресурсів в різних регіонах	За допомогою політик маршрутизації	Відсутня	За допомогою прискорювачів
Кешування	Відсутнє	на edge локаціях	Відсутнє
Захист від ddos	Відсутній	AWS Shield	AWS Shield
Прискорення передачі даних	Latency-based маршрутизація	Інфраструктура AWS та кешування	Інфраструктура AWS та оптимальна маршрутизація
Перевірка стану ресурсів	Присутня	Відсутня	Присутня
Підтримка Lambda@Edge	Відсутня	Присутня	Відсутня

CloudFront дозволяє тимчасово зберігати контент на edge локаціях та використовувати мережеву інфраструктуру AWS, використовує захист від ddos атак та інтегрується з іншими сервісами безпеки. Також він дозволяє виконувати

код прямо на edge локаціях за допомогою Lambda@Edge, що може ще сильніше пришвидшити застосунок. Проте CloudFront не дозволяє керувати доменними іменами та маршрутизацією та балансуванням трафіку, подібно до Route 53, та не підходить для застосувань, які використовують не HTTP протоколи.

Global Accelerator чудово підходить для застосунків, які використовують не HTTP протоколи. Він займається маршрутизацією до ресурсів, які знаходяться в різних регіонах і вибирає оптимальний маршрут для трафіку, який буде використовувати інфраструктуру провайдера. Сервіс, як і CloudFront використовує захист від ddos атак та інтегрується з іншими сервісами захисту. Проте він не кешує контент та не дозволяє виконувати код на edge локаціях. Він надає 2 статичні адреси, проте не надає керування доменними іменами, як Route 53.

Можна комбінувати ці сервіси між собою для компенсації недоліків. Наприклад, використати Route 53 та CloudFront. Route 53 можна використати для керування доменами та оптимальної маршрутизації, а CloudFront для прискорення трафіку та кешування.

2. Проєктування структури власного рішення

В цьому розділі буде розглянуто проєктування структури сайту новин для розгортання на платформі AWS. Базою цієї структури будуть обчислювальні платформи та сервіси для забезпечення доступу по широкій географії проаналізовані в минулому розділі.

2.1. Вибір обчислювальної платформи та сервісу для забезпечення доступу по широкій географії

Для вибору сервісів потрібно визначитись з основними критеріями, яким повинні відповідати вибрані рішення.

Критерії для вибору:

1. Для сайту новин характерне дуже швидке та велике зростання трафіку після публікації новин, які цікавлять велику аудиторію користувачів. Тому для комфорту користувачів сайт повинен без проблем справлятися з великим навантаженням та швидким його зростанням.
2. Затримки сайту для користувача мають бути мінімальними.
3. Висока доступність для аудиторії сайту новин, яка розподілена по всьому світу.
4. Контент сайту новин в основному статичний і добре кешується.
5. Оптимальна витрата грошей на інфраструктуру та часу спеціалістів на її налаштування та підтримку.

Для широкої географії користувачів в цьому випадку підійде Route 53 та CloudFront. Route 53 дозволить керувати доменним іменем та за допомогою політик маршрутизації, для зниження затримок, направляти трафік в оптимальні

кінцеві обчислювальні точки. CloudFront дозволить кешувати контент, в основному це будуть новини, що дозволить значно знизити навантаження на обчислювальні ресурси та пришвидшити його доставлення користувачам. Global Accelerator для сайту новин не підходить, оскільки не має кешування.

При виборі обчислювальної платформи краще використати переваги різних сервісів.

Платформи, які не мають обмежень часу роботи мають свої переваги. Вони можуть обробляти багато запитів користувача, без потреби створення під кожен запит нового екземпляра. Це дозволяє відповідати на запити користувача без затримок, які пов'язані з "холодним стартом", подібно до Lambda.

При великій кількості запитів, які будуть створювати значне постійне навантаження подібні платформи будуть дешевшими, якщо порівнювати з Lambda. Якщо буде сплеск навантаження, то ці платформи можуть досить швидко масштабуватись. Тому ці платформи чудово підходять для задач, яких багато і які вимагають мінімального часу на виконання.

Наприклад, для користувача сайту новин важливо, щоб контент для нього завантажувався швидко. У випадку з Lambda в цій задачі можуть бути затримки через холодний старт. Для подібних задач для сайту новин цілком підійде платформа EC2. Можна використати інші платформи, наприклад контейнерні, але це може бути дорожче.

Lambda функція для таких задач не підходить через холодний старт і ціну. Щоб уникнути холодного старту можна зарезервувати потужності Lambda функцій. Також є варіант «підігрівати» функції шляхом періодичного виклику, для того, щоб контейнери функцій не вимикались, що буде дешевше. Для прискорення також можна використати Lambda@Edge. Але всі ці опції можуть суттєво підвищити вартість цієї платформи.

Проте ця платформа є хорошим рішенням для тих задач, які виконуються періодично і час виконання яких невеликий, а також для тих, для яких "холодний старт" не є критичним. Наприклад, у випадку сайту новин такою задачею є публікація нового матеріалу. Використання Lambda для таких задач може

знизити навантаження платформу, яка постійно працює. Також це буде оптимально з точки зору вартості, оскільки кількість викликів Lambda функцій невелика.

2.2. Архітектура застосунку

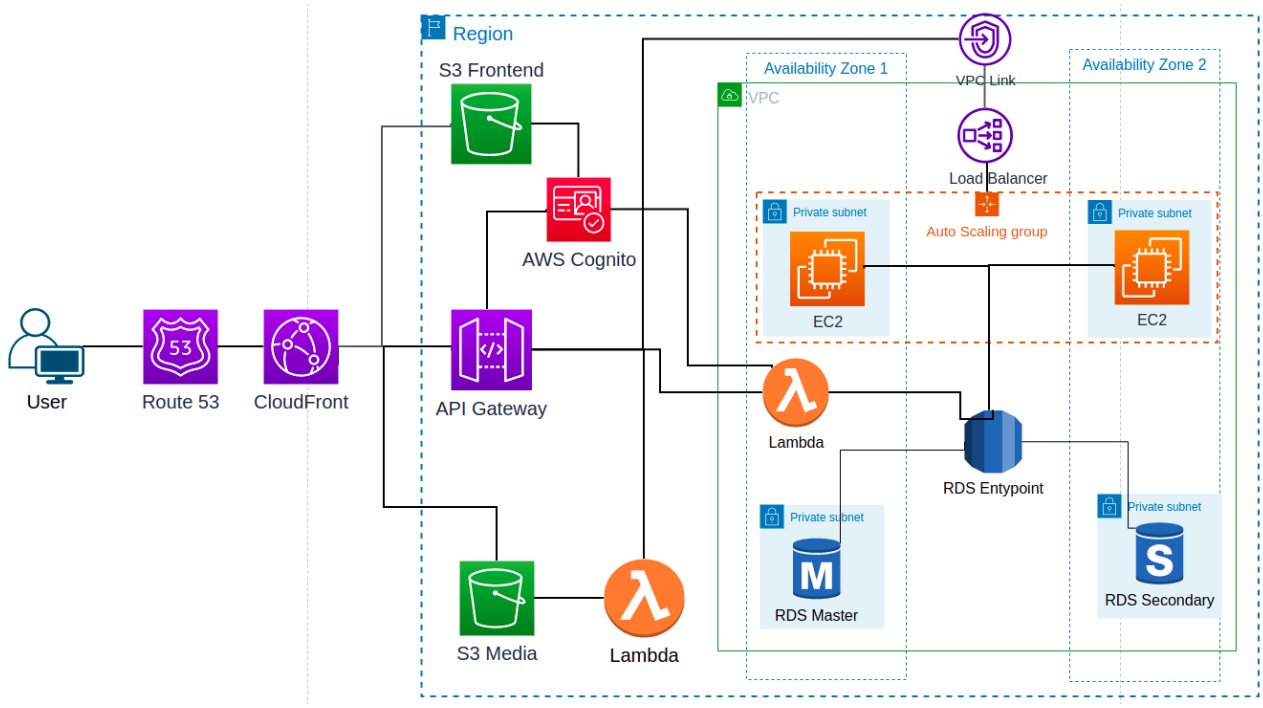


Рисунок 2.1. Архітектура застосунку

Route 53 направляє запити користувачів на CloudFront. Ці два сервіси глобальні, тому не розміщені в межах регіону. Всі інші компоненти знаходяться в одному регіоні. Якщо, те що потрібно користувачу присутньо в кеші CloudFront, то CloudFront повертає відповідь користувачу. В іншому випадку в залежності від запиту, CloudFront направить його відповідному компоненту. CloudFront кешує дані з таких компонентів:

1. Клієнтський рівень розміщений на S3
2. API Gateway, який є точкою входу до серверного рівня
3. S3 сховище для медіафайлів.

Сховище S3 (Amazon Simple Storage Service) дозволяє хостити статичний вебсайт за допомогою файлів клієнтського рівня (*.html, *.js, *.css) [16]. S3 зберігає дані, як мінімум в трьох зонах доступності в межах одного регі ону. Це

дозволяє досягти 99.99% доступності сервісу. Клієнтський компонент під час взаємодії з користувачем надсилає запити Cognito та серверному рівню через API Gateway.

Для реєстрації, входу, підтвердження пошти тощо клієнт надсилає запит Cognito. Цей сервіс виконує авторизацію, аутентифікацію та керування користувачами [17]. Його використання дозволяє замість написання власного рішення, використати надійне та безпечне рішення від хмарного провайдера. Сервіс теж забезпечує високу доступність використовуючи декілька зон доступності в межах регіону. Після створення нового користувача Cognito викличе Lambda функцію, яка додасть дані про користувача в базу даних.

Інші запити через Route 53 та CloudFront надсилаться клієнтом, або користувачем напряму, до API Gateway. Сервіс дозволяє створювати та керувати API серверної частини застосунку [18]. Він перенаправляє запит на потрібний обчислювальний сервіс. Також, якщо запит потребує перевірки доступу користувача, API Gateway зробить це за допомогою Cognito. CloudFront дозволяє керувати кешом, тому кешуватись будуть ті дані, які рідко, або не змінюються. Сервіс високодоступний і має доступність закладену провайдером на рівні 99.99%.

Бізнес-логіка виконується на платформах EC2 та Lambda. Екземпляри EC2 повинні бути розміщені в VPC (Virtual Private Cloud), що є логічно ізольованою віртуальною мережею. Для EC2 потрібно вибрати підмережу (subnet), яка є набором IP адрес у VPC. Кожна така підмережа знаходиться в певній зоні доступності.

Екземпляри EC2 розміщені у двох різних приватних підмережах, які належать різним зонам доступності. Це потрібно для підвищення доступності комонента системи, оскільки хмарний провайдер перекладає це на користувача й автоматично не запускає екземпляри EC2 у різних зонах доступності. Для ще більшої відмовостійкості можна збільшити кількість використаних зон доступності.

Для того, щоб справлятися з великим збільшенням на віртуальні машини навантаження для автоматичного масштабування використовується Auto scaling group. А для рівномірного розподілу навантаження по різних екземплярах EC2 залучений балансувальник навантаження. (Load balancer). Балансувальник має схему internal, що обмежує інтернет доступ до нього для підвищення безпеки. Балансувальник підключений до VPC Link, для того, щоб API Gateway міг перенаправляти запити до екземплярів EC2. Провайдер забезпечує 99.99% доступність роботи цих сервісів. EC2 взаємодіє з базою даних, яка теж розміщена в VPC.

Для рівня даних використовується сервіс баз даних RDS. Для застосунку використовується PostgreSQL. Щоб підвищити відмовостійкість рівня даних використовується Multi-Az розгортання. Два екземпляри бази даних розміщені в різних зонах доступності і автоматично синхронізуються між собою. Екземпляр secondary копіює стан екземпляру master, який здійснює операції запису. Якщо master стає недоступним, його замінює secondary. Схема бази даних розміщена в додатку А.

Lambda функції використовуються для періодичних задач. Lambda функції, які взаємодіють з базою даних, для того, щоб мати доступ до бази даних теж виконуються в VPC. Інші функції використовуються для взаємодії з сховищем S3 для медіафайлів. Як вже зазначалось в попередньому розділі, високу доступність Lambda функцій забезпечує провайдер, і вони не потребують додаткових дій для цього, як у випадку з EC2.

Використання відмовостійких сервісів AWS, разом з додатковим підвищенням відмовостійкості віртуальних машин, робить застосунок з використанням вищеописаної архітектури високодоступним та відмовостійким. Подібної надійності застосунку досягти на власних потужностях дуже складно, особливо, якщо враховувати використання глобальних сервісів Route 53 та CloudFront, який використовує інфраструктуру AWS розміщену по всьому світу.

Застосунок дозволяє неавторизованим користувачам:

- Переглядати відсортовані за датою публікації новини.

- Переглядати новини за певною категорією.
- Переглянути всі категорії новин
- Переглядати коментарі до новин, які залишили авторизовані користувачі.

Неавторизований користувач може зареєструватись При реєстрації йому потрібно вказати: пошту, ім’я та пароль. Для підтвердження реєстрації потрібно користувачу потрібно буде вказати код, надісланий Cognito, який прийде йому пошту.

Можливості зареєстрованого користувача:

- Змінити зображення свого профілю зі стандартного.
- Залишати коментарі до новин.
- Видаляти свої коментарі

Крім звичайних користувачів є редактори новин, які можуть:

- Публікувати новини
- Редагувати новини
- Видаляти новини
- Додавати категорії
- Редагувати категорії
- Видаляти категорії
- Видаляти коментарі користувачів

3. Розробка серверного компонента застосунку

3.1. API Gateway та авторизатор

Сервіс API Gateway дозволяє створити HTTP API, REST API, Websocket API. REST API більш гнучкий та пропонує більше можливостей ніж HTTP API, але дещо дорожчий. Для сайту новин буде створений REST API.

Таблиця. Опис API

Метод	Ресурс	Параметри	Опис	Платформа
GET	/articles	URL: page – номер сторінки page_size – розмір сторінки category – категорія asc_order – порядок сортування по даті	Повертає список новин відповідно до параметрів	EC2
GET	/articles/{id}	PATH: id новини	Повертає новину	EC2
POST	/articles	Body: стаття у форматі JSON Header: токен редактора	Публікує новину	Lambda
PUT	/articles/{id}	Body: стаття у форматі JSON PATH: id новини Header: токен редактора	Змінює новину	Lambda
DELETE	/articles/{id}	PATH: id новини	Видаляє новину	Lambda
GET	/categories		Повертає список категорій новин	EC2
POST	/categories	Body: категорія у форматі JSON Header: токен редактора	Додає категорію	Lambda
PUT	/categories	Body: категорія у форматі JSON Header: токен редактора	Редагує категорію	Lambda

DELETE	/categories/{id}	PATH: id категорії	Видаляє новину	Lambda
GET	/articles/{id}/comments	PATH: id новини URL: page – номер сторінки page_size – розмір сторінки asc_order – порядок сортування по даті	Повертає список коментарів до новини відповідно до параметрів	EC2
POST	/articles/{id}/comments	PATH: id новини Body: коментар у форматі JSON Header: токен користувача	Додає коментар до новини	EC2
DELETE	/comments/{id}	PATH: id коментаря Header: токен користувача, або редактора	Видаляє коментар	Lambda
POST	/images	Body: зображення Header: токен редактора	Додає зображення до сховища S3	Lambda
PUT	/users/pictures	Body: зображення Header: id користувача	Змінює зображення профілю користувача	Lambda
GET	/users/{id}	PATH: id користувача Header: токен користувача	Дозволяє переглядати користувачу свій профіль	EC2

При створенні REST API, крім створення нового API, можна клонувати вже з наявного, імпортувати з Swagger або Open API 3 або використати приклад AWS. Також можна вибрати тип кінцевої точки: регіональний, приватний, який доступний тільки в межах VPC та оптимізований по Edge локаціях і підключений до CloudFront.

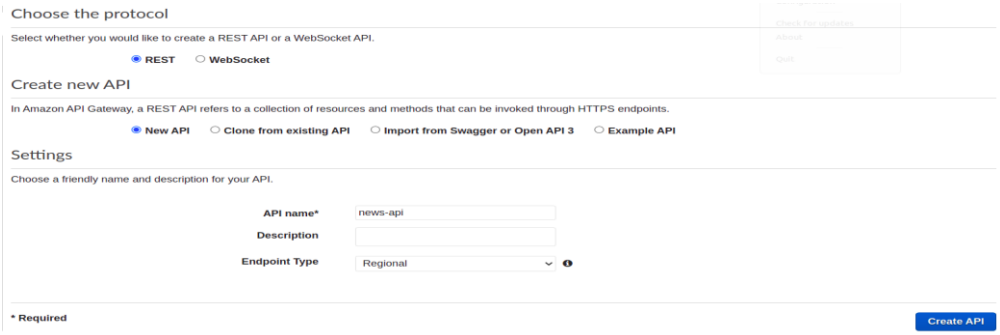


Рисунок 3.1. Створення REST API

В цьому випадку було вибрано створення нового API з назвою *news-api* та регіональним типом кінцевої точки. Потім API можна під'єднати до дистрибутива CloudFront.

REST API надає можливість створити авторизатори (Authorizers) для перевірки доступу користувача на здійснення запиту перед його направленням на сервер. Авторизатори можуть бути 2-х типів:

1. Lambda
2. Cognito

Cognito авторизатор перевіряє лише валідність токена доступу і не може перевірити чи належить користувач потрібній групі в пулі користувачів Cognito.

В нашому випадку редактори новин належать групі *Administrator*. Тому для створення авторизатора для редакторів буде використана Lambda функція, яка спочатку робить запит до Cognito для того, щоб верифікувати токен доступу, а потім парсить токен і перевіряє приналежність до групи *Administrator*.

Для написання Lambda функції використовується мова програмування GO та AWS SKD [22]. Оскільки GO не підтримується в редакторі коду на AWS, то потрібно його потрібно завантажити. Це можна зробити напряму, завантаживши відкомпільовану програму розміщену в zip архіві. Оскільки код Lambda функції (див. Додаток Б) використовує змінну середовища REGION, її потрібно додати в середовище виконання функції, що можна зробити в меню конфігурації. В цьому випадку це змінна має значення *eu-west-1*.

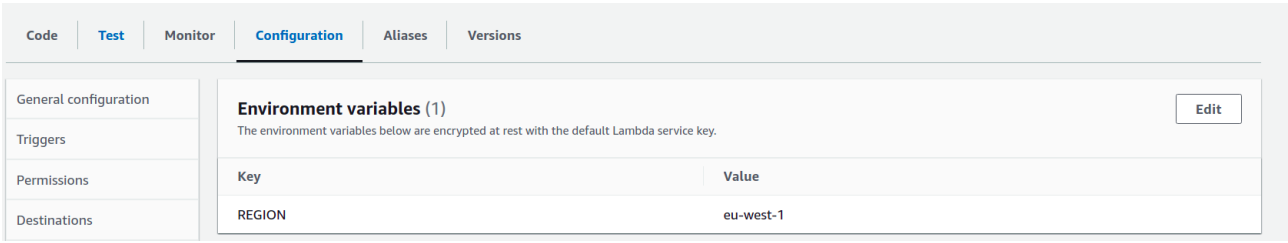


Рис 3.2. Додана змінна середовища

Після створення відповідної Lambda функції створюється авторизатор який її використовує. При його створенні вказується назва *AdminAuth*, тип Lambda, регіон виконання eu-west-1 (Ірландія), Lambda Event Payload – Token. Token Source (джерело токену), позначає заголовок в якому він розміщений і має значення Authorization. Також увімкнене кешування, яке буде працювати протягом 300 секунд. Token Validation дозволяє перевірити токен за допомогою регулярних виразів і не використовується. Lambda Invoke Role опціональний параметр, який дозволяє вказати роль, з відповідними дозволами, для виклику функції. Якщо вона не вказана, то для створення авторизатора, потрібно буде надати API доступ на виклик Lamba.

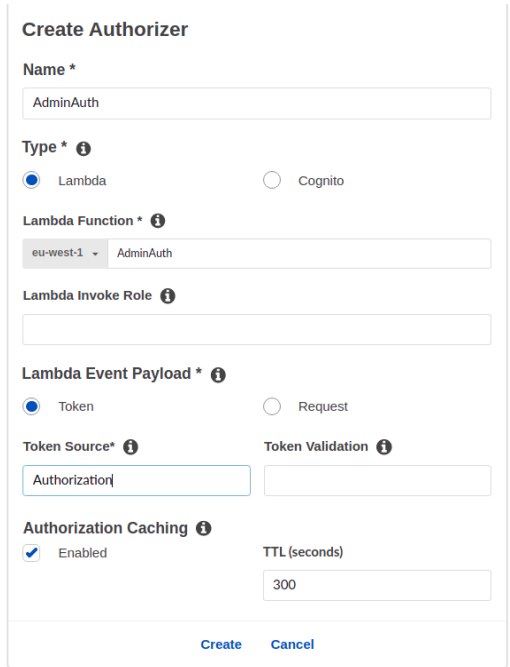


Рисунок 3.3. Створення авторизатора

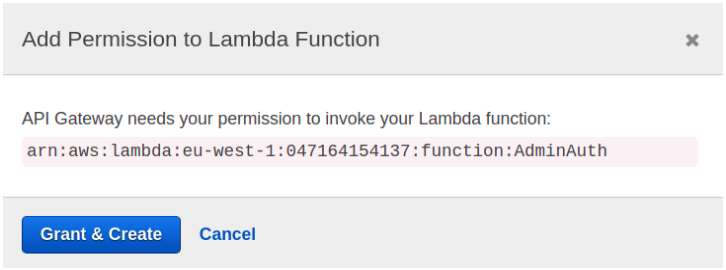


Рис 3.4. Надання API Gateway дозволу на виклик функції

Після створення авторизатора його можна протестувати:

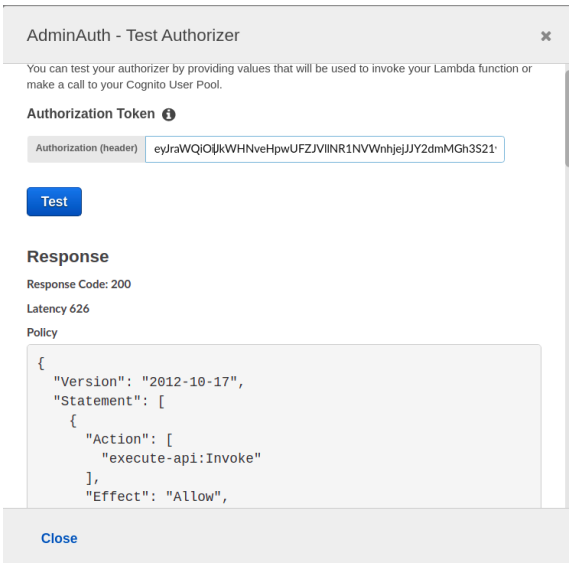


Рисунок 3.5. Тест створеного авторизатора

Для авторизації звичайних користувачів, які не є редакторами замість використання Lambda авторизатора, достатньо створити Cognito авторизатор.

3.2. Структура серверних компонентів

Бекенд застосунку розділений на 3-и рівні за функціональним призначенням:

1. Handler – рівень, який працює з HTTP запитами, які можуть надходити до EC2 або Lambda функцій. Цей рівень парсить дані запиту і перевіряє їх правильність. Після цього передає їх компоненту рівня Service, після виконання якого, повертає HTTP відповідь користувачу.
2. Service – рівень, який виконує бізнес-логіку. Він викликається компонентом Handler рівня, обробляє дані, валідує їх, при потребі

викликає компонент Repository. Після виконання бізнес-логіки, повертає відповідь компонента Handler.

3. Repository – рівень, який взаємодіє з базою даних. Викликається компонентом Service, якому після роботи з базою даних повертає відповідь.

Цей підхід полегшує підтримку, зміну та розширення кодової бази.

Кожен рівень застосунку має загалом п’ять компонентів, по одному для кожної сутності: article, category, user, comment та для медіа контенту.

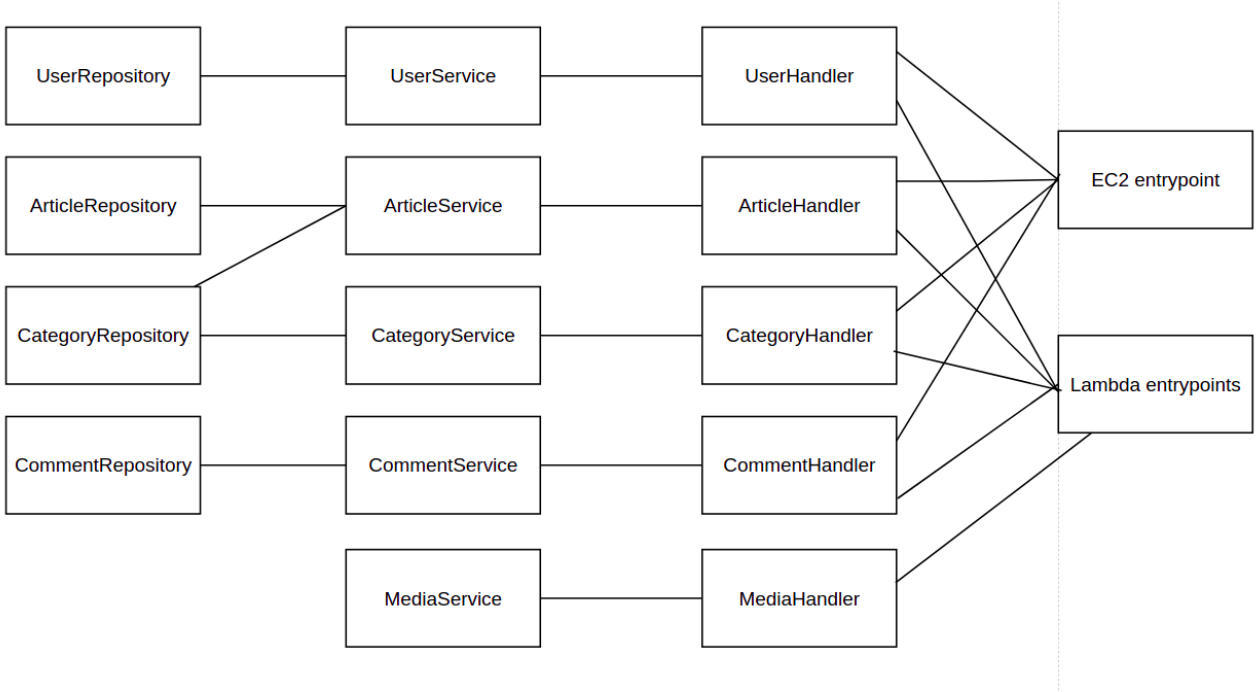


Рисунок 3.6. Схема взаємодії компонентів бекенду

Компоненти сутності взаємодіють переважно з компонентами цією ж сутності. Виключенням є ArticleService, котрий взаємодіє з CategoryRepository. Також для медіаконтенту не використовується Repostiory, оскільки для взаємодії з S3 використовується AWS SDK.

Кожен компонент має свої методи. До прикладу інтерфейси компонентів сутності Article написані мовою GO з використанням фреймворку Gin:

ArticleRepository:

```
type ArticleRepository interface {
    InsertArticle(article model.Article, categoryID int) (*model.Article, error)
    UpdateArticle(article model.Article, categoryID int) (*model.Article, error)
    DeleteArticle(articleID int64) error
    GetArticle(articleID int64) (*model.Article, error)
    GetArticles(categoryID int, param model.GetArticlesParams) ([]model.Article, error)
}
```

ArticleService:

```
type ArticleService interface {
    CreateArticle(requestParam model.CreateArticleRequest, editorID string) (*model.Article, error)
    UpdateArticle(article model.Article) (*model.Article, error)
    DeleteArticle(articleID int64) error
    GetArticle(articleID int64) (*model.Article, error)
    GetArticles(params model.GetArticlesParams) ([]model.Article, error)
}
```

ArticleHandler:

```
type ArticleHandler interface {
    CreateArticle(c *gin.Context)
    UpdateArticle(c *gin.Context)
    DeleteArticle(c *gin.Context)
    GetArticle(c *gin.Context)
    GetArticles(c *gin.Context)
}
```

Повний перелік інтерфейсів компонентів знаходиться в Додатку Б

3.3. Приклад детальної розробки

Більш детально серверний рівень розглянемо на прикладі додавання новин. Для функціонала додавання новин використана обчислювальна платформа Lambda. Цей вибір зумовлений тим, що час виконання цих функцій недовгий, затримки пов’язані з холодним стартом допустимі та додавання новин буде відбуватись відносно нечасто.

Метод *CreateArticle* компонента *ArticleHandler* для додавання новин приймає від функції JSON об’єкт:

```
{
   "title": "title",
   "category": "category",
   "content": "content"
```

```
}
```

Для того, щоб новина додалась, всі поля повинні бути не пустими. Після того, як поля були перевірені з JSON об’єкта створюється об’єкт типу:

```
type CreateArticleRequest struct {
    Title    string `json:"title" binding:"required"`
    Category string `json:"category" binding:"required"`
    Content  string `json:"content" binding:"required"`
}
```

Після чого, він разом з ID редактора, котрий взятий з токену розміщеному в заголовку авторизації та передається в метод *CreateArticle* компонента *ArticleService*. Там генерується дата публікації та з *CreateArticleRequest* об’єкта створюється об’єкт типу:

```
type Article struct {
    ArticleID       int64     `json:"article_id"`
    Title           string    `json:"title"`
    Category        string    `json:"category"`
    Content         string    `json:"content"`
    EditorID        string    `json:"editor_id"`
    PublicationDate time.Time `json:"publication_date"`
}
```

А також перевіряє наявність отриманої категорії в таблиці *categories* та отримує ID категорії за допомогою методу *GetArticleIDByName* компонента *ArticleRepository*. Після цього новина додається в базу даних компонентом *ArticleRepository,* який повертає об’єкт типу *Article* вже з генерованим ID. Цей об’єкт з *ArticleService* передається *ArticleHandler,* котрий повертає відповідь користувачу. У випадку, якщо на якомусь з кроків виникає помилка, вона теж повертається користувачу. (детальний код в додатку Г)

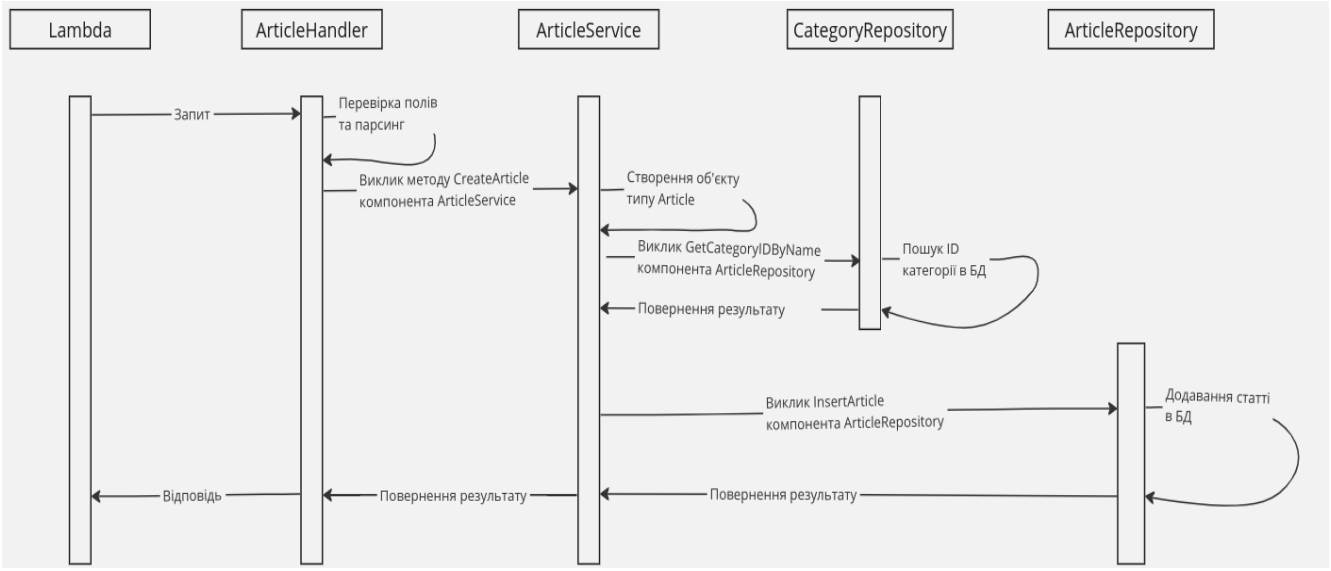


Рисунок 3.7. Діаграма успішного опрацювання запиту

Для того, щоб Lambda функція мала доступ до бази даних, вона повинна виконуватись в межах VPC. Для того, щоб під'єднати Lambda функцію до VPC, вона повинна мати дозволи для виконання в межах VPC. Для цього AWS вже має готову політику доступу *AWSLambdaVPCAccessExecutionRole*, яку можна використати для Lambda функції.

AWSLambdaVPCAccessExecutionRole

Provides minimum permissions for a Lambda function to execute while accessing a resource within a VPC - create, describe, delete network interfaces and write permissions to CloudWatch Logs.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents",
                "ec2:CreateNetworkInterface",
                "ec2:DescribeNetworkInterfaces",
                "ec2:DeleteNetworkInterface",
                "ec2:AssignPrivateIpAddresses",
                "ec2:UnassignPrivateIpAddresses"
            ],
            "Resource": "*"
        }
    ]
}
```

Рисунок 3.8. Політика доступу *AWSLambdaVPCAccessExecutionRole*

Після цього Lambda функція використовується для створення методу *POST* для ресурсу *articles* в REST API. При налаштуванні використовується "Lambda Proxy Integration" для того, щоб в Lambda функцію дані запиту передавались в тому ж форматі, в якому їх отримав API Gateway. Також встановлюється значення Custom Timeout в 5 секунд, замість стандартного значення 29 секунд. Після створення в Method Request встановлюється раніше створений авторизатор для редакторів.

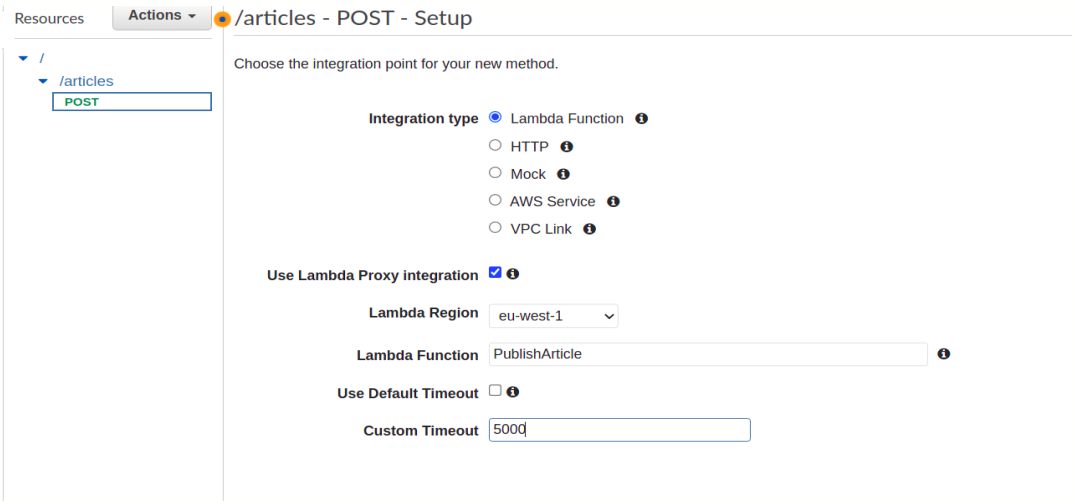


Рисунок 3.9. Додавання Lambda функції в REST API.

Після цього API можна розгорнути та робити запити до сервера:

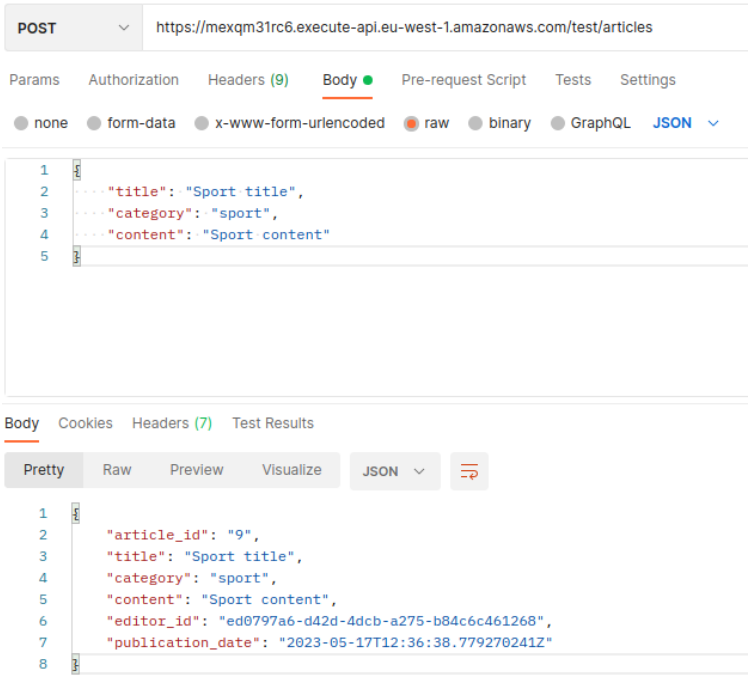


Рисунок 3.10. Демонстрація успішного запиту для створення новини

Якщо додавання новин виконується Lambda функцією, то запит на отримання списку новин виконується платформою EC2. Цей вибір зумовлений тим, що користувачі дуже часто переглядають список новин і при цьому важлива мінімальна затримка на повернення результату. До того ж це відбувається часто і якщо використовувати Lambda, то це може бути суттєво дорожче EC2.

Щоб надати користувачу гнучкість для перегляду списку новин запит може приймати 4-и параметри:

- page – номер сторінки для перегляду новин, починається з 0
- page_size – кількість новин в сторінці
- category – категорія новин, які будуть повернуті
- asc_order – порядок сортування новин по даті публікації. Якщо *asc_odred=false*, то новини будуть повертатись від новіших до старіших.

Коли *ArticleHandler* обробляє запит методом *GetArticles* ці параметри парсяться в об’єкт типу:

```
type GetArticlesParams struct {
    Category       string `form:"category"`
    Page           int    `form:"page"`
    PageSize       int    `form:"page_size" binding:"lte=100"`
    AscendingOrder bool   `form:"asc_order"`
}
```

При цьому, якщо page_size понад 100, користувачу повернеться помилка. Розмір сторінки обмежений, щоб не перезавантажувати сервер запитами, де розмір сторінки занадто великий. Після цього ці параметри передаються в *ArticleService*. Якщо користувач вказав категорію в параметрах, то її наявність перевіряється подібно до того, як це було при додаванні новини. Також, якщо користувач не вказав розмір сторінки, то встановлюється 20 елементів. Після цього викликається метод *GetArticles* компонента *ArticleRepository,* який відповідно до параметрів, які передав користувач робить запит до бази даних і повертає список новин.

Для того, щоб інтегрувати екземпляри EC2, які знаходяться в приватній мережі і на які розподіляє трафік внутрішній балансувальник, з API Gateway створюється VPC Link. Після цього при додаванні методу в API Gateway вибирається тип інтеграції VPC Link.

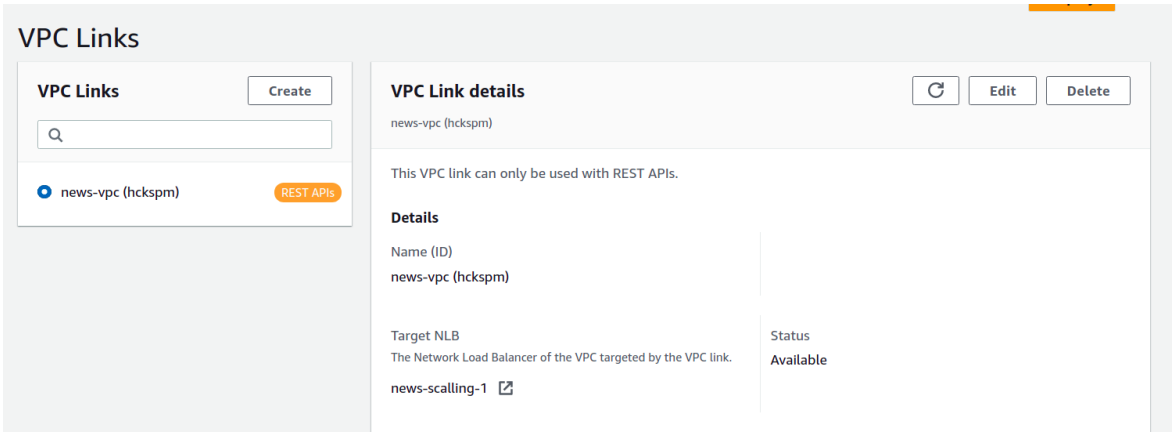


Рисунок 3.11. Створений VPC Link

Після цього можна робити запити на отримання списку новин:

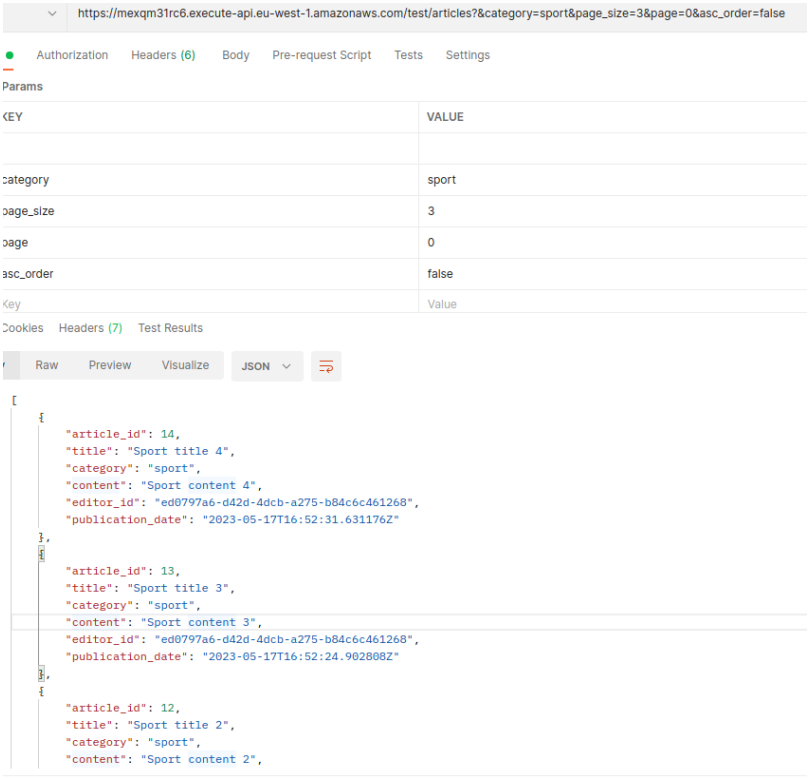


Рисунок 3.12. Приклад успішного запиту отримання списку новин

Висновки

В результаті виконання роботи були розглянуті можливості хмарної платформи AWS для забезпечення високодоступності застосунку. Зокрема були оглянуті обчислювальні платформи та рішення платформи для забезпечення доступу по широкій географії.

З обчислювальних платформ були розглянуті та порівняні між собою віртуальні машини EC2, контейнерні платформи та безсерверні Lambda функції.

Серед рішень для забезпечення доступу по широкій географії, які пропонує AWS були розглянуті та порівняні Route 53, CloudFront та Global Accelerator.

Проаналізована й обґрунтована можливість повної реалізації застосунку на хмарній платформі з виконанням всіх вимог до відмовостійкості та функціоналу.

Була побудована високодоступна багатокомпонентна архітектура для сайту новин з використанням сервісів: EC2, Lambda, Route 53, CloudFront, Cognito, API Gateway, RDS та S3.

Для сайту новин відповідно до його функціонала була спроєктована PostgreSQL база даних.

Детально описано API серверного рівня застосунку та розглянуто процес створення REST API за допомогою сервісу API Gateway та створення авторизатора адміністраторів для нього.

Розглянуто програмну структуру серверного рівня для сайту новин та детально описано розробку, підключення до REST API та демонстрація роботи деяких елементів серверного рівня.

Список використаних джерел

1. Overview of Amazon Web Services: AWS Whitepaper [Електронний ресурс] – Режим доступу до ресурсу: https://docs.aws.amazon.com/pdfs/whitepapers/latest/aws-overview/aws-overview.pdf
2. Amazon Elastic Compute Cloud: User Guide for Linux Instances [Електронний ресурс] – Режим доступу до ресурсу: https://docs.aws.amazon.com/pdfs/AWSEC2/latest/UserGuide/ec2-ug.pdf
3. Cloud Computing Service Models – IaaS, PaaS, SaaS [Електронний ресурс] – Режим доступу до ресурсу: https://digitalcloud.training/cloud-computing-service-models-iaas-paas-saas/
4. ECS vs EC2 vs Lambda [Електронний ресурс] – Режим доступу до ресурсу: https://digitalcloud.training/ecs-vs-ec2-vs-lambda/
5. What's The Difference Between Containers And Virtual Machines? [Електронний ресурс] – Режим доступу до ресурсу: https://aws.amazon.com/compare/the-difference-between-containers-and-virtual-machines/
6. Docker on AWS: AWS Whitepaper [Електронний ресурс] – Режим доступу до ресурсу: https://docs.aws.amazon.com/pdfs/whitepapers/latest/docker-on-aws/docker-on-aws.pdf
7. AWS Lambda: Developer Guide [Електронний ресурс] – Режим доступу до ресурсу: https://docs.aws.amazon.com/pdfs/lambda/latest/dg/lambda-dg.pdf
8. What is AWS Lambda? [Електронний ресурс] – Режим доступу до ресурсу: https://www.serverless.com/aws-lambda/
9. Reliability Pillar: AWS Well-Architected Framework [Електронний ресурс] – Режим доступу до ресурсу:

https://docs.aws.amazon.com/pdfs/wellarchitected/latest/reliability-pillar/wellarchitected-reliability-pillar.pdf

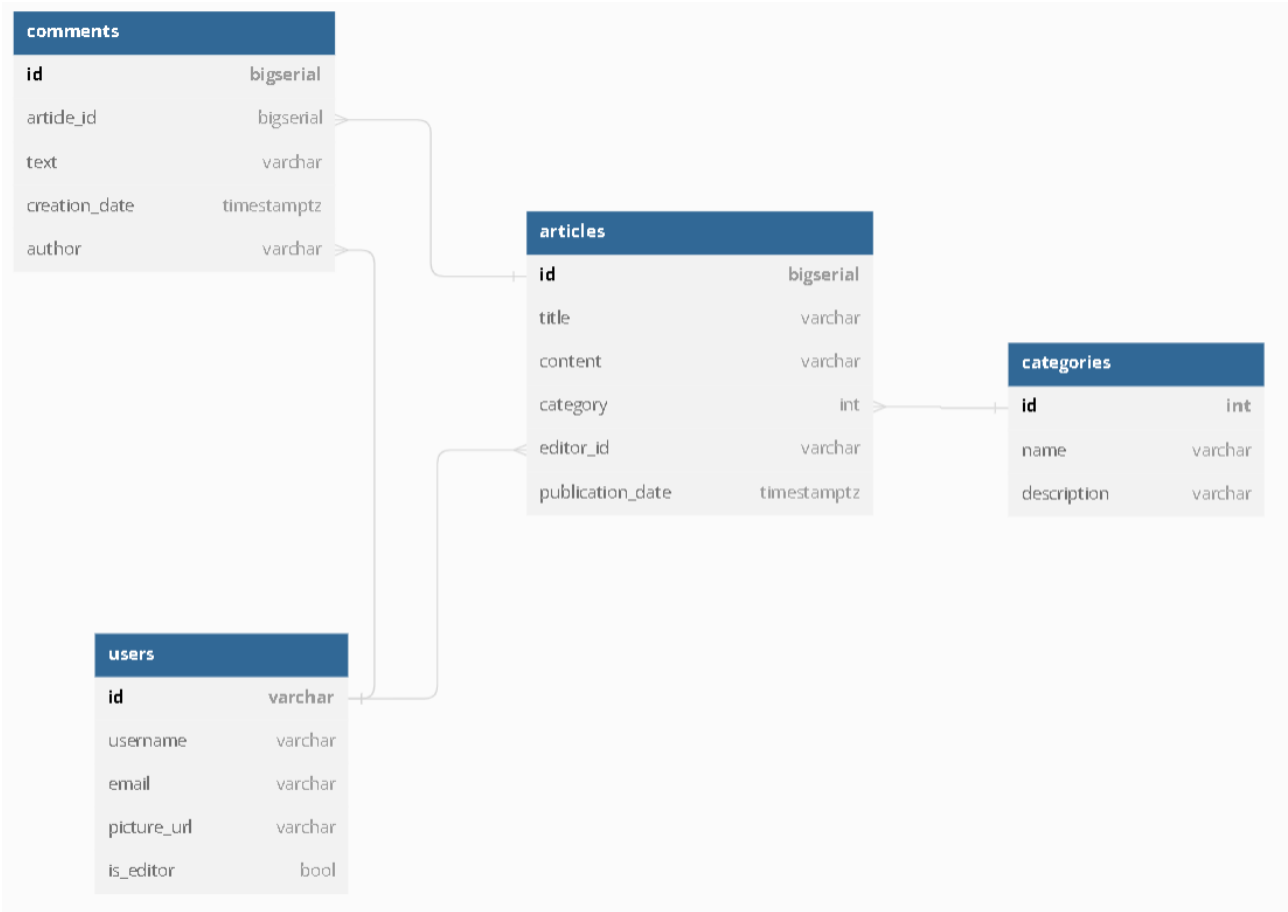
10. What is Route 53 and What are its functionalities? [Електронний ресурс] – Режим доступу до ресурсу: https://www.novelvista.com/blogs/cloud-and-aws/what-is-route-53-and-what-are-its-functionalities
11. Amazon Route 53: Developer Guide [Електронний ресурс] – Режим доступу до ресурсу: https://docs.aws.amazon.com/pdfs/Route53/latest/DeveloperGuide/route53-dg.pdf
12. What is Amazon CloudFront and How Does It Work? [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/mindful-engineering/today-we-will-learn-about-cloudfront-690bf3a8819a
13. Amazon CloudFront Key Features [Електронний ресурс] – Режим доступу до ресурсу: https://aws.amazon.com/cloudfront/features/?nc=sn&loc=2&whats-new-cloudfront.sort-by=item.additionalFields.postDateTime&whats-new-cloudfront.sort-order=desc
14. Amazon CloudFront: Developer Guide [Електронний ресурс] – Режим доступу до ресурсу: https://docs.aws.amazon.com/pdfs/AmazonCloudFront/latest/DeveloperGuide/AmazonCloudFront_DevGuide.pdf
15. AWS Global Accelerator: Developer Guide [Електронний ресурс] – Режим доступу до ресурсу: https://docs.aws.amazon.com/pdfs/global-accelerator/latest/dg/global-accelerator-guide.pdf
16. Amazon Web Services – Global Accelerator [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/amazon-web-services-global-accelerator/

17. AWS CloudFront vs Global Accelerator [Електронний ресурс] – Режим доступу до ресурсу: https://jayendrapatil.com/aws-cloudfront-vs-global-accelerator/
18. Amazon Simple Storage Service User Guide [Електронний ресурс] – Режим доступу до ресурсу: https://docs.aws.amazon.com/pdfs/AmazonS3/latest/userguide/s3-userguide.pdf
19. Amazon Cognito Developer Guid [Електронний ресурс] – Режим доступу до ресурсу: https://docs.aws.amazon.com/pdfs/cognito/latest/developerguide/cognito-dg.pdf
20. Amazon API Gateway Developer Guide [Електронний ресурс] – Режим доступу до ресурсу: https://docs.aws.amazon.com/pdfs/apigateway/latest/developerguide/apigateway-dg.pdf
21. Amazon Relational Database Service User Guide [Електронний ресурс] – Режим доступу до ресурсу: https://docs.aws.amazon.com/pdfs/AmazonRDS/latest/UserGuide/rds-ug.pdf
22. AWS SDK for Go v2 [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/aws/aws-sdk-go-v2

Додатки

Додаток А

Схема бази даних застосунку



Додаток Б

Код Lamba функції для авторизатора редакторів:

```
package main

import (
	"context"
	"fmt"
	"os"
	"strings"

	"github.com/aws/aws-lambda-go/events"
	"github.com/aws/aws-lambda-go/lambda"
	"github.com/aws/aws-sdk-go/aws"
	"github.com/aws/aws-sdk-go/aws/session"
	"github.com/aws/aws-sdk-go/service/cognitoidentityprovider"
	"github.com/dgrijalva/jwt-go"
)

const (
	allow = "Allow"
	deny  = "Deny"
)

func main() {
	lambda.Start(authorizerHandler)
}

func authorizerHandler(ctx context.Context, req
events.APIGatewayCustomAuthorizerRequest)
(events.APIGatewayCustomAuthorizerResponse, error) {
	token := req.AuthorizationToken
	// Створення сесії AWS
	sess := session.Must(session.NewSession(&aws.Config{
		Region: aws.String(os.Getenv("REGION")),
	}))
	// Створення клієнта для взаємодії з Cognito
	cognitoClient := cognitoidentityprovider.New(sess)
	// Параметри для запиту до Cognito, який приймає AccessToken
	getUserInput := &cognitoidentityprovider.GetUserInput{
		AccessToken: aws.String(token),
	}
	// Валідація токену за допомогою GetUser, оскільки метод який це робить
напряму недоступний в SDK
	_, err := cognitoClient.GetUser(getUserInput)
	if err != nil {
		// Відмова в доступі, якщо токен невалідний
		return generatePolicy(deny, req.MethodArn), nil
	}
	// Перевірка приналежності до групи Administrator в токені
	isAdmin := isAdmin(token)
	// Надання чи відмова в доступі в залежності від значення isAdmin
	return generatePolicy(getEffect(isAdmin), req.MethodArn), nil
}
```

```
// Перевірка того, чи входить користувач в групу адміністраторів за допомогою
токену
func isAdmin(tokenString string) bool {
	token, _ := jwt.Parse(tokenString, func(token *jwt.Token) (interface{},
error) {
		return []byte(""), nil
	})
	claims, _ := token.Claims.(jwt.MapClaims)
	return strings.Contains(fmt.Sprintf("%v", claims["cognito:groups"]),
"Administrator")
}

func getEffect(isAdmin bool) string {
	if isAdmin {
		return allow
	}
	return deny
}
// Функція генерації відповіді для авторизатора
func generatePolicy(effect, resource string)
events.APIGatewayCustomAuthorizerResponse {
	return events.APIGatewayCustomAuthorizerResponse{
		PolicyDocument: events.APIGatewayCustomAuthorizerPolicy{
			Version: "2012-10-17",
			Statement: []events.IAMPolicyStatement{
				{
					Action:   []string{"execute-api:Invoke"},
					Effect:   effect,
					Resource: []string{resource},
				},
			},
		},
	}
}
```

Додаток В

Повний список інтерфейсів застосунку:

Repository:

```go
type UserRepository interface {
    InsertUser(user model.User) error
    UpdateUserPicture(userID string, pictureURL string) error
    GetUser(userID string) (*model.User, error)
}

type ArticleRepository interface {
    InsertArticle(article model.Article, categoryID int) (*model.Article, error)
    UpdateArticle(article model.Article, categoryID int) (*model.Article, error)
    DeleteArticle(articleID int64) error
    GetArticle(articleID int64) (*model.Article, error)
    GetArticles(categoryID int, param model.GetArticlesParams) ([]model.Article, error)
}

type CategoryRepository interface {
    InsertCategory(category model.Category) (*model.Category, error)
    UpdateCategory(category model.Category) (*model.Category, error)
    DeleteCategory(categoryID int) error
    GetCategoryByID(categoryID int) (*model.Category, error)
    GetCategories() ([]model.Category, error)
    GetCategoryIDByName(name string) (int, error)
}

type CommentRepository interface {
    InsertComment(comment model.Comment) (*model.Comment, error)
    DeleteComment(commentID int64) error
    GetUserComments(articleID int64, params model.GetCommentsParams) ([]model.UserComment, error)
}
```

Service:

```
type UserService interface {
    CreateUser(user model.User) error
    UpdateUserPicture(userID string, pictureURL string) error
    GetUser(userID string) (*model.User, error)
}

type ArticleService interface {
    CreateArticle(requestParam model.CreateArticleRequest, editorID string) (*model.Article, error)
    UpdateArticle(article model.Article) (*model.Article, error)
    DeleteArticle(articleID int64) error
    GetArticle(articleID int64) (model.Article, error)
    GetArticles(params model.GetArticlesParams) ([]model.Article, error)
}

type CategoryService interface {
    CreateCategory(request model.CreateCategoryRequest) (*model.Category, error)
    UpdateCategory(category model.Category) (*model.Category, error)
    DeleteCategory(categoryID string) error
    GetCategory(categoryID int) (*model.Category, error)
}

type CommentsService interface {
    CreateComment(request model.CreateCommentRequest) error
    DeleteComment(commentID int64, userID string) error
    GetCommentsByArticle(articleID int, params model.GetCommentsParams)
}

type MediaService interface {
    UploadImage(image *os.File) (string, error)
    DeleteImage(url string) error
}
```

Handler:

```go
type UserHandler interface {
    CreateUser(c *gin.Context)
    UpdateUserPicture(c *gin.Context)
    GetUser(c *gin.Context)
}

type ArticleHandler interface {
    CreateArticle(c *gin.Context)
    UpdateArticle(c *gin.Context)
    DeleteArticle(c *gin.Context)
    GetArticle(c *gin.Context)
    GetArticles(c *gin.Context)
}

type CategoryHandler interface {
    CreateCategory(c *gin.Context)
    UpdateCategory(c *gin.Context)
    DeleteCategory(c *gin.Context)
    GetCategory(c *gin.Context)
    GetCategories(c *gin.Context)
}

type CommentHandler interface {
    CreateComment(c *gin.Context)
    DeleteComment(c *gin.Context)
    GetCommentsByArticle(c *gin.Context)
}

type MediaHandler interface {
    UploadImage(c *gin.Context)
    DeleteImage(c *gin.Context)
}
```

Додаток Г

Код для додавання новин:

repository/article.go

```
import (
	"database/sql"
	"fmt"
	"news/model"
)

// Структурра, що реалізує інтерфейс ArticleRepository
type PSQLArticleRepository struct {
	db *sql.DB
}

// Конструктор
func NewPSQLArticleRepository(db *sql.DB) *PSQLArticleRepository {
	return &PSQLArticleRepository{
		db: db,
	}
}

// Метод для додавання статті в SQL базу даних
func (r *PSQLArticleRepository) InsertArticle(a model.Article, categoryID int)
(*model.Article, error) {
	// Текс SQL запиту, який додає статтю та повертає її id
	insertArticleQuery := `INSERT INTO articles (
				title, category, content, editor_id,
publication_date
				) VALUES ($1, $2, $3, $4, $5)
				RETURNING id`
	// Додавання статті та встановлення ArticleID
	err := r.db.QueryRow(insertArticleQuery, a.Title, categoryID, a.Content,
a.EditorID, a.PublicationDate).
		Scan(&a.ArticleID)
	return &a, err
}
```

...

service/article.go

```
import (
	"log"
	"news/model"
	"news/repository"
	"time"
)

// Структура, яка імплементує інтерфейс ArticleService
// Складається з агрегованих структур, які імплементують інтерфейси
// ArticleRepository та CategoryRepository
type ArticleServiceStruct struct {
	articleRepo  repository.ArticleRepository
	categoryRepo repository.CategoryRepository
}

// Конструктор
func NewArticleServiceStruct(articleRepo repository.ArticleRepository,
	categoryRepo repository.CategoryRepository) *ArticleServiceStruct {
	return &ArticleServiceStruct{
```

```
            articleRepo:  articleRepo,
            categoryRepo: categoryRepo,
      }
}

// Метод для створення новин
func (s *ArticleServiceStruct) CreateArticle(
      requestParam model.CreateArticleRequest,
      editorID string) (*model.Article, error) {
      // Створення структури Article
      article := articleRequestToArticle(requestParam, editorID)

      // Перевірка того, чи є отримана категорія статті в базі даних
      // і повернення ID категорії
      categoryID, err := s.categoryRepo.GetCategoryIDByName(article.Category)
      if err != nil { // Повернення помилки, якщо вона виникла
            return nil, err
      }
      // Додавання статті в БД і повернення результату виконання InsertArticle
      return s.articleRepo.InsertArticle(article, categoryID)
}

// Допоміжна функція, яка створює структуру Article з CreateArticleRequest та
editorID
func articleRequestToArticle(request model.CreateArticleRequest, editorID
string) model.Article {
      return model.Article{
            Title:           request.Title,
            Category:        request.Category,
            Content:         request.Content,
            EditorID:        editorID,
            PublicationDate: time.Now(),
      }
}
```

handler/article.go

```
// Структура, яка імплементує інтерфейс ArticleHandler
// Складається структури, яка імплементує інтерфейс
// ArticleService
type ArticleHandlerStruct struct {
      articleServ service.ArticleService
}

// Конструктор для створення ArticleHandlerStruct
func NewArticleHandlerStruct(articleServ service.ArticleService)
*ArticleHandlerStruct {
      return &ArticleHandlerStruct{articleServ: articleServ}
}

// Метод для створення статті з отриманого запиту
func (h *ArticleHandlerStruct) CreateArticle(c *gin.Context) {
      var articleRequest model.CreateArticleRequest
      // Парсинг тіла з JSON в структуру CreateArticleRequest та перевірка
      // на те, чи заповнені поля
      err := c.BindJSON(&articleRequest)
      // Надсилання StatusBadRequest та помилки при невалідності тіла запиту
      if err != nil {
            sendError(c, http.StatusBadRequest, err.Error())
            return
      }
      // Отримання ID редактора з токену
      authHeader := c.GetHeader("Authorization")
      editorID := getEditorID(authHeader)
```

```
	// Виклик ArticleService для створення статті
	article, err := h.articleServ.CreateArticle(articleRequest, editorID)
	// Надсилання помилки у разі її виникнення
	if err != nil {
		if errors.Is(err, sql.ErrNoRows) {
			// Якщо помилка пов'язана з тим, що категорії не існує,
відправляється StatusBadRequest
			categoryError := fmt.Sprintf("category %s doesn`t exist",
articleRequest.Category)
			sendError(c, http.StatusBadRequest, categoryError)
		} else {
			// В іншому випадку StatusInternalServerError
			sendError(c, http.StatusInternalServerError, err.Error())
		}
		return
	}
	// Надсилання StatusOK відповіді та доданої статті у JSON форматі
	c.JSON(http.StatusOK, article)
}

// Допоміжна функція для отримання ID редактора з токену
func getEditorID(auth string) string {
	token, _ := jwt.Parse(auth, func(token *jwt.Token) (interface{}, error) {
		return []byte(""), nil
	})
	claims, _ := token.Claims.(jwt.MapClaims)
	return fmt.Sprintf("%s", claims["sub"])
}
```

cmd/create-article/main.go

```
var ginLambda *ginadapter.GinLambda

func init() {
	// Створення gin роутеру
	r := gin.Default()
	// Підключення до бази даних
	db, err := sql.Open("postgres", os.Getenv("DB_CONNECTION_STRING"))
	if err != nil {
		panic(err)
	}
	// Ініціалізація ArticleHandlerStruct
	articleHandler := app.InitArticleHandler(db)
	// Додавання CreateArticle до роутеру
	r.POST("/articles", articleHandler.CreateArticle)
	// Створення ginLambda функції з роутеру
	ginLambda = ginadapter.New(r)
}

func Handler(ctx context.Context, req events.APIGatewayProxyRequest)
(events.APIGatewayProxyResponse, error) {
	// Використання ginLambda проксі для взаємодії роутера gin з
APIGatewayProxyRequest
	return ginLambda.ProxyWithContext(ctx, req)
}

func main() {
	// Старт lambda функції
	lambda.Start(Handler)
}
```

Додаток Д

Код для отримання списку новин:

repository/article.go

```
// Метод для отриммання списку новин з бази даних за параметрами
func (r *PSQLArticleRepository) GetArticles(categoryID int, param
model.GetArticlesParams) ([]model.Article, error) {
	// створення запиту та параметрів для нього
	getArticlesQuery, queryParams := buildQueryAndParams(categoryID, param)
	// виконання запиту
	rows, err := r.db.Query(getArticlesQuery, queryParams...)
	if err != nil {
		return nil, err
	}
	// оголошення змінної для зберігання списку новин
	articles := []model.Article{}
	for rows.Next() {
		var a model.Article
		// заповнення новини даними з бази даних
		if err := rows.Scan(&a.ArticleID, &a.Title, &a.Category, &a.Content,
&a.EditorID, &a.PublicationDate); err != nil {
			return nil, err
		}
		// додавання новини в список
		articles = append(articles, a)
	}
	// закриття читання новин
	if err := rows.Close(); err != nil {
		return nil, err
	}
	// перевірка чи виникла помилка під час ітерації по рядкам
	if err := rows.Err(); err != nil {
		return nil, err
	}
	// повернення результату
	return articles, nil
}

// Метод побудови запиту для БД та створення списку параметрів для запиту
func buildQueryAndParams(categoryID int, param model.GetArticlesParams) (string,
[]interface{}) {
	// Основні елементи для запиту
	queryParams := []interface{}{
		param.PageSize, param.PageSize * param.Page,
	}
	// Основа запиту
	getArticlesQuery := `SELECT A.id, A.title, C.name, A.content, A.editor_id,
a.publication_date
                                    FROM articles A INNER JOIN categories C ON
A.category = C.id`
	sortOrder := ""
	if param.AscendingOrder {
		sortOrder = "ASC"
	} else {
		sortOrder = "DESC"
	}
	// Якщо користувач вказав котегорію
	if categoryID != -1 {
		// вставка категорії першим елементом в список параметрів
		queryParams = append([]interface{}{categoryID}, queryParams...)
```

```
            // формування кінцевого запиту разом з пошуком по категорії та
сортуванням
            getArticlesQuery = getArticlesQuery + fmt.Sprintf(`
                                    WHERE A.category = $1
                                    ORDER BY A.publication_date %s
                                    LIMIT $2
                                    OFFSET $3`, sortOrder)
      } else {
            // формування кінцевого запиту з сортуванням
            getArticlesQuery = getArticlesQuery + fmt.Sprintf(`
                                    ORDER BY A.publication_date %s
                                    LIMIT $1
                                    OFFSET $2`, sortOrder)
      }
      return getArticlesQuery, queryParams
}
```

service/article.go

```
// Отримання списку новин за параметрами
func (s *ArticleServiceStruct) GetArticles(params model.GetArticlesParams)
([]model.Article, error) {
      categoryID := -1
      // Якщо користувач не вказав розмір сторінки, встановлюється значення 20
      if params.PageSize == 0 {
            params.PageSize = 20
      }
      log.Printf("params: %v\n", params)
      if params.Category != "" {
            // Перевірка того, чи є отримана категорія новини в базі даних
            // і повернення ID категорії
            databaseID, err :=
s.categoryRepo.GetCategoryIDByName(params.Category)
            if err != nil { // Повернення помилки, якщо вона виникла
                  return nil, err
            }
            categoryID = databaseID
      }
      // повернення результату виконання методу компонента ArticleRepository
      return s.articleRepo.GetArticles(categoryID, params)
}
```

handler/article.go

```
// Отримання списку новин
func (h *ArticleHandlerStruct) GetArticles(c *gin.Context) {
      var getArticlesParams model.GetArticlesParams
      // Парсинг тіла параметрів в структуру GetArticlesParams та їх перевірка
      err := c.BindQuery(&getArticlesParams)
      // Надсилання StatusBadRequest та помилки при невалідності параметрів
запиту
      if err != nil {
            sendError(c, http.StatusBadRequest, err.Error())
            return
      }
      //// Виклик ArticleService для отримання новин
      articles, err := h.articleServ.GetArticles(getArticlesParams)
      // Надсилання помилки у разі її виникнення
      if err != nil {
            if errors.Is(err, sql.ErrNoRows) {
                  // Якщо помилка пов'язана з тим, що категорії не існує,
відправляється StatusBadRequest
                  categoryError := fmt.Sprintf("category %s doesn`t exist",
getArticlesParams.Category)
```

```
                sendError(c, http.StatusBadRequest, categoryError)
            } else {
                // В іншому випадку StatusInternalServerError
                sendError(c, http.StatusInternalServerError, err.Error())
            }
            return
        }
        // Надсилання новин користувачу в JSON форматі
        c.JSON(http.StatusOK, articles)
}
```