СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Навчально-науковий інститут (факультет) інформаційних технологій та електроніки_

Кафедра __інформаційних технологій та програмування__

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної випускної роботи

освітній ступінь _бакалавр_
спеціальність 121 “Інженерія програмного забезпечення”
спеціалізація “Інженерія програмного забезпечення”

на тему “Мікросервіси в веб-застосунку з менеджменту вебсайтів”

Виконав: студент групи ІПЗ-21д _____ _Ю.Г. Ковальов_____
(підпис) (ініціали і прізвище)

Керівник _____ д.т.н., проф., Д.М. Марченко_
(підпис) (ініціали і прізвище)

Завідувач кафедри _____ д.т.н., проф., О. І. Захожай
(підпис) (ініціали і прізвище)

Рецензент_____ д.т.н., проф., В.О. Лифар

Київ - 2025

СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Навчально-науковий інститут (факультет) інформаційних технологій та електроніки
(повне найменування інституту, факультету)

Кафедра _інформаційних технологій та програмування____
(повна назва кафедри)

Освітній ступінь _бакалавр_
(бакалавр, магістр)

спеціальність _121 “Інженерія програмного забезпечення”__
(шифр і назва спеціальності)

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____

“____” _____20___року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ ВИПУСКНУ РОБОТУ СТУДЕНТУ
____Ковальов Юрій Григорійович__
(прізвище, ім’я, по батькові)

1. Тема роботи_Мікросервіси в веб-застосунку з менеджменту вебсайтів__
_____
_____
_____

Керівник роботи Марченко Дмитро Миколайович, д.т.н., проф.,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджений наказом університету від “27”травня_2025року № 67/15.15-С_

2. Строк подання студентом роботи__16.06.2025_____

3. Вихідні дані до роботи__Об’єктом даної роботи є розгляд, порівняння, та впровадження мікросервісної архітектру в веб застосунку з управління вебсайтами___
_____
_____
_____

4.Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)_____
_____
_____
_____
_____
_____

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

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання____01.04.2025___

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання кваліфікаційної випускної роботи	Строк виконання етапів	Примітка
1	Одержання завдання на виконання роботи	01.04.25	виконано
2	Укладання і погодження з керівником плану і етапів виконання роботи	06.04.25	виконано
3	Узагальнення даних літературних джерел, укладання розділу «Аналіз предметної галузі»	16.04.25	виконано
4	Аналіз шляхів виконання завдання. Вибір і погодження з керівником оптимального шляху	26.04.25	виконано
5	Укладання та тестування програмного продукту	26.05.25	виконано
6	Укладання, оформлення та погодження пояснювальної записки з керівником	14.06.25	виконано
7	Надання готової пояснювальної записки на кафедру	16.06.25	виконано
8	Укладання доповіді і презентації	17.06.25	виконано

Студент _____ Ю.Г. Ковальов
(підпис) (ініціали і прізвище)

Керівник роботи _____ Д.М. Марченко
(підпис) (ініціали і прізвище)

РЕФЕРАТ

Пояснювальна записка до кваліфікаціної роботи на тему “ Мікросервіси в веб-застосунку з менеджменту вебсайтів” містить 54 сторінки, 11 рисунків, 25 джерел.

Об’єкт дослідження - мікросервісна архітектура.

Предмет дослідження — технології мікросервісної архітектури на AWS.

Мета кваліфікаційної роботи — розробити та впровадити частину вебзастосунку з використанням мікросервісної архітектури на базі AWS.

Методи дослідження — аналіз і порівняння, дослідження джерел, проектування та програмна розробка.

Розроблений програмний продукт використовується у діючому веб застосунку з менеджменту веб сайтів. Для розробки програмного продукту використані підходи та інструменти які було обрано в ході дослідження і порівняння моноліту та мікросервісів. Головні переваги мікросервісів перемогли їх недоліки. Для розробки було обрану мову програмування Javascript (Node.js), Serverless framework та AWS Lambda, AWS API GateWay на Amazon Web Services.

ЗМІСТ

ВСТУП

Зростаюча кількість вебсайтів спонукає адміністраторів до пошуку зручної платформи з якої в декілька кліків можна виконувати дії для багатьох сайтів. Така платформа - “Сяйво”, створена спеціально для управління кількома WordPress-сайтами одночасно, йде набагато далі за простий інструмент оновлення. Платформа поєднує: єдиний інтерфейс для всіх проєктів; підтримку агентств та команд; моніторинг; гнучке масштабування; для мережі multisite і агентств. Завдяки цим можливостям це економить час, спрощує рутинні завдання й робить роботу з WordPress-проєктами значно ефективнішою. Це ідеальний інструмент для фрилансерів, агентств і команд, що підтримують багато клієнтів одночасно. Широкий функціонал платформи спонукає до застосування немонолітної архітектури. Платформа має безліч окремих функцій: керування оновленнями, резервне копіювання, моніторинг аптайму, безпека, звітність, інтеграції з API WordPress тощо. Мікросервіси дозволяють розділити ці функції на окремі незалежні компоненти, які можна масштабувати окремо — наприклад, якщо навантаження зростає лише на модуль бекапів. Падіння одного сервісу (наприклад, моніторингу аптайму) не вплине на роботу інших (наприклад, авторизації або безпеки). Це підвищує загальну стійкість системи до помилок.

Об’єктом дослідження цієї роботи є мікросервісна архітектура, зокрема на базі AWS Lambda functions.

Предметом дослідження виступає застосування мікросервісної архітектури для платформи з управління вебсайтами.

Метою роботи є розробка та обґрунтування доцільності впровадження мікросервісної архітектури в порівнянні з монолітної архітектурою.

Для досягнення цієї мети поставлено такі завдання:

- провести аналіз існуючих проблем на платформі та можливих напрямків оптимізація, рефакторінга;

- дослідити та проаналізувати принципи роботи мікросервісної архітектури на AWS;
- обґрунтувати доцільність застосування мікросервісної архітектури;
- розробити та впровадити мікросевісну архітектуру;
- визначити переваги та недоліки нової системиву порівнянні з попереднім рішеннями.

Практична новизна роботи полягає у застосуванні нової архітектури що підвищить стійкість, стабільність на розсередить навантаження з основого сервіса платформи на розсередженні мікросервіси.

Практичне значення одержаних результатів полягає у впровадженні, написанні, підтримці мікросервісів діючої веб платформи з управління вебсайтами.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Монолітна архітектура в вебзастосунку

Монолітна архітектура — це традиційний підхід до побудови веб-додатків, при якому вся функціональність (інтерфейс, логіка, обробка даних і доступ до бази даних) розміщується в єдиній кодовій базі. Цей стиль домінував до епохи мікросервісів і сьогодні залишається актуальним у багатьох проєктах через свою простоту та ефективність на ранніх етапах розробки [2, 6, 14, 15].
Основи монолітної архітектури.

Монолітна архітектура передбачає, що всі компоненти додатку — бізнес-логіка, доступ до даних, обробка запитів, авторизація, інтерфейс — реалізовані в єдиному модулі. Кодова база одна, застосунок компілюється або запускається як одне ціле, оновлення відбувається також одночасно для всієї системи. Це означає, що при зміні навіть одного модуля — наприклад, авторизації — необхідно повторно розгортати увесь застосунок [13, 14, 15].

Переважно, моноліт складається з трьох основних шарів: представлення (View/UI), логіки (Controller/Service) і даних (Model/ORM). У типовому фреймворку — наприклад, Django, Laravel або Spring Boot — ці шари взаємодіють напряму й синхронно [2, 6, 14, 15]. Схема цього виглядає так:

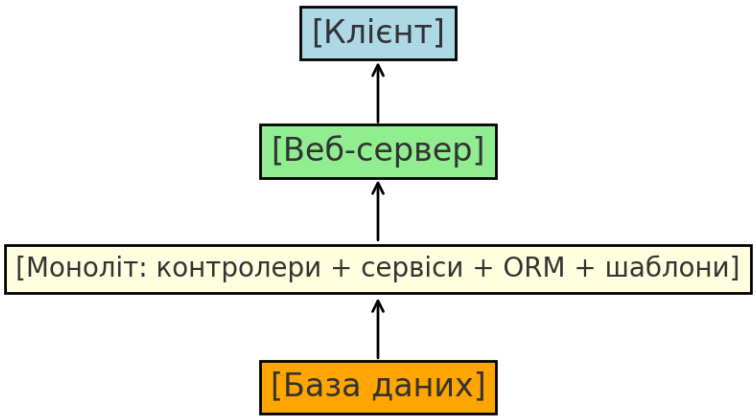

Рис. 1. Структура монолітної веб-архітектури.

Таке рішення дозволяє дуже швидко розгорнути MVP чи стартап-проєкт. Один сервер, один застосунок, мінімум DevOps-залежностей.

Переваги монолітної архітектури.

Швидкий запуск продукту — розробник може сконцентруватися на функціональності, а не на інфраструктурі.

Спрощене розгортання — вся система деплоїться як одне ціле, зазвичай на один сервер або інстанс.

Мінімальні вимоги до інфраструктури — на відміну від мікросервісів, не потрібні окремі CI/CD пайплайни, реєстри, балансувальники тощо.

Інтегроване середовище — всі частини програми розробляються у спільному середовищі, що полегшує відлагодження.

Вища продуктивність при низьких навантаженнях — оскільки всі виклики виконуються локально, без мережевої латентності [2, 6, 14, 15].

Недоліки монолітів.

Важка підтримка при зростанні — коли кодова база збільшується, зростає кількість залежностей і складність змін.

Слабка масштабованість — неможливо масштабувати лише частину функціональності, доводиться масштабувати увесь застосунок.

Тісне зчеплення — зміни в одному компоненті можуть спричинити побічні ефекти в іншому.

Повільне розгортання — кожна зміна вимагає тестування і деплою всієї системи.

Обмеження при використанні різних технологій — неможливо використовувати різні мови програмування або БД для різних модулів [2, 6, 13, 14, 15].

Приклади технологій:

- PHP: Laravel, Symfony
- Python: Django, Flask
- Java: Spring Boot
- Node.js: Express
- Ruby: Ruby on Rails

Ці фреймворки традиційно підтримують монолітний підхід, хоча більшість також допускає модульність чи окремі мікросервісні компоненти.

Модульний моноліт — сучасний підхід.

Окрему увагу слід звернути на концепцію модульного моноліту (Modular Monolith). Це коли велика система розбита на окремі домени чи модулі, але всі вони зберігаються в одній кодовій базі і розгортаються разом. Це дозволяє досягти певної незалежності модулів, зберігаючи при цьому переваги моноліту. Shopify — класичний приклад великої компанії, яка успішно масштабувала моноліт завдяки чіткій модульності (див. Scaling the Monolith) [2, 6, 13, 14, 15].

Коли обирати моноліт. Моноліт — чудовий вибір, якщо:

- ви створюєте MVP чи прототип;
- команда невелика (до 5–7 осіб);
- немає необхідності в високій масштабованості;
- важлива швидкість розробки;
- ви не маєте ресурсів на DevOps, моніторинг, оркестрацію.

Сценарії використання: GitLab, Netflix.

GitLab довгий час був монолітним застосунком на Ruby on Rails. Це дозволило команді швидко розвивати продукт, фокусуючись на швидкості релізів. З часом GitLab зіткнувся з проблемами масштабованості, але замість переходу до мікросервісів обрав модель "componentized monolith" — розділення на логічні блоки всередині однієї бази коду [2, 6, 13, 14, 15].

Netflix спочатку був класичним монолітом для онлайн DVD-сервісу. Зі збільшенням трафіку та складності, компанія перейшла на мікросервіси. Кожна функція (вхід, рекомендації, потокове відео) стала окремим сервісом. Цей перехід дав змогу масштабуватись до глобального рівня [2, 6, 13, 14, 15].

DevOps у монолітах. Монолітні застосунки мають спрощений DevOps-процес. Достатньо налаштувати один CI/CD пайплайн, єдина точка моніторингу, логування. Однак, великі моноліти потребують ретельного планування [14, 15]:

- деплой має бути атомарним і з тестами;
- бажано реалізовувати Blue-Green деплой чи Canary-реліз;

- логіка має бути ізольованою в модулях навіть без фізичного розділення сервісів.

Монолітні системи дозволяють ефективно використовувати класичні стратегії тестування. Оскільки вся логіка розміщується в одному застосунку, можна легко створити end-to-end, інтеграційні й юніт-тести без потреби в симуляції зовнішніх API.

Юніт-тести — перевірка логіки всередині модулів (наприклад, сервісів або моделей). Інтеграційні тести — тестування взаємодії між компонентами (наприклад, між контролерами і базою даних). E2E тести — автоматизоване тестування ключових сценаріїв користувача через інтерфейс.
Інструменти: PHPUnit (Laravel), PyTest (Django), RSpec (Rails), JUnit (Spring).

Перевага моноліту полягає в тому, що система працює як єдина одиниця, тому безпеку можна централізовано забезпечити на рівні:

- аутентифікації та авторизації (JWT, OAuth2)
- захисту бази даних (ORM, SQL escaping)
- керування сесіями (CSRF protection, cookie flags)
- логування та аудиту (централізовані журнали)

Також важливо слідкувати за оновленнями залежностей, використовувати security headers, HTTPS та забезпечувати резервне копіювання даних.

1.2. Мікросервісна архітектура в веб-розробці

Мікросервісна архітектура (Microservices Architecture) — це стиль проєктування програмного забезпечення, у якому застосунок складається з невеликих, незалежних сервісів, кожен з яких виконує чітко визначену функцію. Ці сервіси спілкуються між собою за допомогою легких протоколів, зазвичай HTTP або через асинхронні повідомлення (наприклад, через Kafka чи RabbitMQ) [1, 3].

На відміну від моноліту, мікросервіси дають змогу командам розробників працювати незалежно над різними частинами системи, використовуючи різні мови програмування чи технології. Це дозволяє швидше впроваджувати зміни, масштабувати окремі компоненти й підвищити стійкість до збоїв [1].

Наприклад, в e-commerce системі окремі сервіси можуть обробляти замовлення, управляти товарами, обслуговувати кошик або авторизацію. У разі збою одного з них, інші можуть продовжувати роботу [3].

Мікросервіси зазвичай розгортаються за допомогою контейнерів (наприклад, Docker) та оркеструються через Kubernetes. CI/CD, автоматичне тестування, моніторинг та логування (ELK, Prometheus + Grafana) є критично важливими для управління складністю такої архітектури.

Основні переваги:

Масштабованість: Один із головних аргументів на користь мікросервісної архітектури — можливість горизонтального масштабування. Замість масштабування всього застосунку, як це відбувається у моноліті, мікросервіси дозволяють масштабувати лише ті компоненти, які дійсно потребують додаткових ресурсів. Наприклад, сервіс з оплатою може працювати під більшим навантаженням, ніж сервіс з реєстрацією користувачів [1,3].

Незалежність команд: Оскільки кожен мікросервіс — це окремий модуль, його розробка, тестування і деплоймент можуть бути ізольованими. Це означає, що кілька команд можуть працювати над різними сервісами одночасно, не заважаючи одна одній. Такий підхід знижує технічну заборгованість та дозволяє ефективно масштабувати організацію [1,4,5].

Швидке розгортання: Зміни у коді одного мікросервісу не потребують повторного деплою всього застосунку. Це дає змогу частіше релізити нові фічі або виправлення помилок, що є критично важливим для конкурентного бізнесу.
Гнучкість технологій: Оскільки сервіси можуть бути незалежними, для кожного з них можна обирати найбільш відповідні мову програмування, фреймворк, базу даних або інші технології. Наприклад, сервіс машинного навчання можна реалізувати на Python, а API — на Node.js або Go [1,4,5].

Стійкість до збоїв: Якщо один з мікросервісів виходить з ладу, система в цілому може залишатися працездатною. Завдяки правильній ізоляції, механізмам повторних спроб (retry), тайм-аутів і fallback-сценаріїв, можна забезпечити високий рівень надійності і доступності системи [1,4,5].
Простота оновлень і масштабування функцій: Мікросервіси дозволяють реалізовувати так звані канарейкові релізи (canary releases) або A/B-тестування,

даючи змогу безпечно перевіряти нові функції на частині користувачів до повноцінного розгортання [1].

Краще логування та моніторинг: Завдяки ізольованості сервісів можна точніше відслідковувати метрики, створювати більш детальні журнали подій та виявляти вузькі місця в роботі окремих компонентів.

Покращена безпека: Обмеження доступу до кожного сервісу окремо дозволяє будувати модель безпеки на рівні найменших повноважень (principle of least privilege). Кожен сервіс має лише ті права, які йому необхідні [1].

Основні виклики:

Складність інфраструктури: Кожен мікросервіс потрібно розгортати, масштабувати, моніторити й обслуговувати окремо. Це вимагає автоматизації та використання сучасних інструментів DevOps. Наприклад, інфраструктура на базі Kubernetes потребує не лише розгортання кластерів, але й налаштування CI/CD пайплайнів, спостереження, журналювання та управління конфігураціями [1,4,5].

Мережеві затримки і помилки: Мікросервіси взаємодіють між собою мережею, що завжди вносить затримку й підвищує ймовірність помилок — наприклад, timeout, dropped packets, або недоступність вузла. Потрібно впроваджувати retry-механізми, тайм-аути, circuit breakers (наприклад, Hystrix) та fallback-логіку, щоб забезпечити стабільну роботу системи.

Безпека міжсервісної взаємодії: У великій кількості точок входу й взаємодій зростає площа атаки. Сервіси повинні автентифікувати й авторизовувати один одного. Потрібні механізми типу mutual TLS, service mesh (наприклад, Istio) та шифрування даних у транзиті [1,4,5].

Моніторинг та трасування (distributed tracing): Ускладнена діагностика проблем. У моноліті все логується в одне місце. У мікросервісах запит може проходити через десятки сервісів. Для аналізу потрібні інструменти трасування (Jaeger, Zipkin), логування (ELK), агрегації метрик (Prometheus, Grafana) [1,4,5].

Управління версіями API: Коли сервіси змінюються незалежно, постає питання сумісності API. Потрібно або підтримувати кілька версій (versioning), або використовувати backward-compatible зміни та контрактне тестування (contract testing).

Тестування: Інтеграційні тести стають складнішими, адже необхідно імітувати чи піднімати залежні сервіси. Для цього використовують тестові стенди, Docker Compose або мок-сервіси.

Загальна когерентність системи: Мікросервіси ведуть до eventual consistency — не всі частини системи одразу отримують однакові дані. Це ускладнює логіку транзакцій. Замість класичних ACID-транзакцій потрібні SAGA-патерни або event-driven підходи.

Підвищення складності управління: Кількість сервісів може зрости до десятків або сотень. Потрібні системи управління залежностями, реєстрації сервісів, оновлень, політик доступу тощо. Це може спричинити "пекло конфігурацій" (configuration hell), якщо не використовувати інструменти централізованого керування.

Архітектура

Мікросервісна система зазвичай складається з наступних компонентів:

API Gateway — точка входу в систему, яка маршрутизує запити клієнтів до відповідних мікросервісів. Це центральний елемент, який може також відповідати за аутентифікацію, кешування, агрегацію відповідей з декількох сервісів, обмеження швидкості (rate limiting) і логування. Популярні рішення: Kong, NGINX, Istio, AWS API Gateway.

Сервіси (мікросервіси) — незалежні компоненти, кожен з яких реалізує окрему бізнес-функцію. Вони мають власні інтерфейси, логіку, конфігурацію та, найчастіше, власну базу даних. Наприклад: сервіс замовлень, сервіс оплати, сервіс авторизації.

Бази даних — кожен мікросервіс відповідає за свій сегмент даних і має окрему базу даних. Це дозволяє досягти високої ізоляції, уникнути проблем блокувань і конфліктів транзакцій. Підходи можуть бути як реляційними (PostgreSQL, MySQL), так і NoSQL (MongoDB, Redis, Cassandra).

Service Registry & Discovery — механізм для автоматичної реєстрації мікросервісів і знаходження їх адреси. Це важливо, коли сервіси динамічно змінюють IP-адреси. Інструменти: Netflix Eureka, Consul, Zookeeper.

Orchestrator — система управління життєвим циклом контейнерів. Вона автоматизує розгортання, масштабування, самовідновлення і оновлення мікросервісів. Найбільш популярний інструмент — Kubernetes, а також Nomad, OpenShift.

Event Bus / Message Broker — забезпечує асинхронну взаємодію між сервісами. Наприклад, коли сервіс замовлень публікує подію "замовлення створено", сервіс виставлення рахунків її підписується та реагує. Це дозволяє зменшити зв’язність і підвищити продуктивність. Основні інструменти: Apache Kafka, RabbitMQ, NATS, AWS SNS/SQS.

Configuration Server — централізоване зберігання конфігурацій усіх сервісів із можливістю оновлення без перезапуску. Приклад: Spring Cloud Config, Consul Key/Value Store.

Service Mesh — інфраструктурний рівень для прозорої взаємодії між сервісами, забезпечення TLS, політик маршрутизації, моніторингу та трасування. Приклад: Istio, Linkerd, Kuma. в систему, яка маршрутизує запити до відповідного сервісу

Сервіси (мікросервіси) — незалежні елементи, кожен з яких обслуговує конкретну бізнес-функцію

Бази даних — кожен сервіс має свою базу або сховище.

Service Registry & Discovery — визначення, де розгорнуто сервіси (наприклад, через Consul або Eureka)

Orchestrator — Kubernetes, Nomad або інші системи керування контейнерами

Event Bus / Message Broker — асинхронна взаємодія (Kafka, RabbitMQ, NATS)

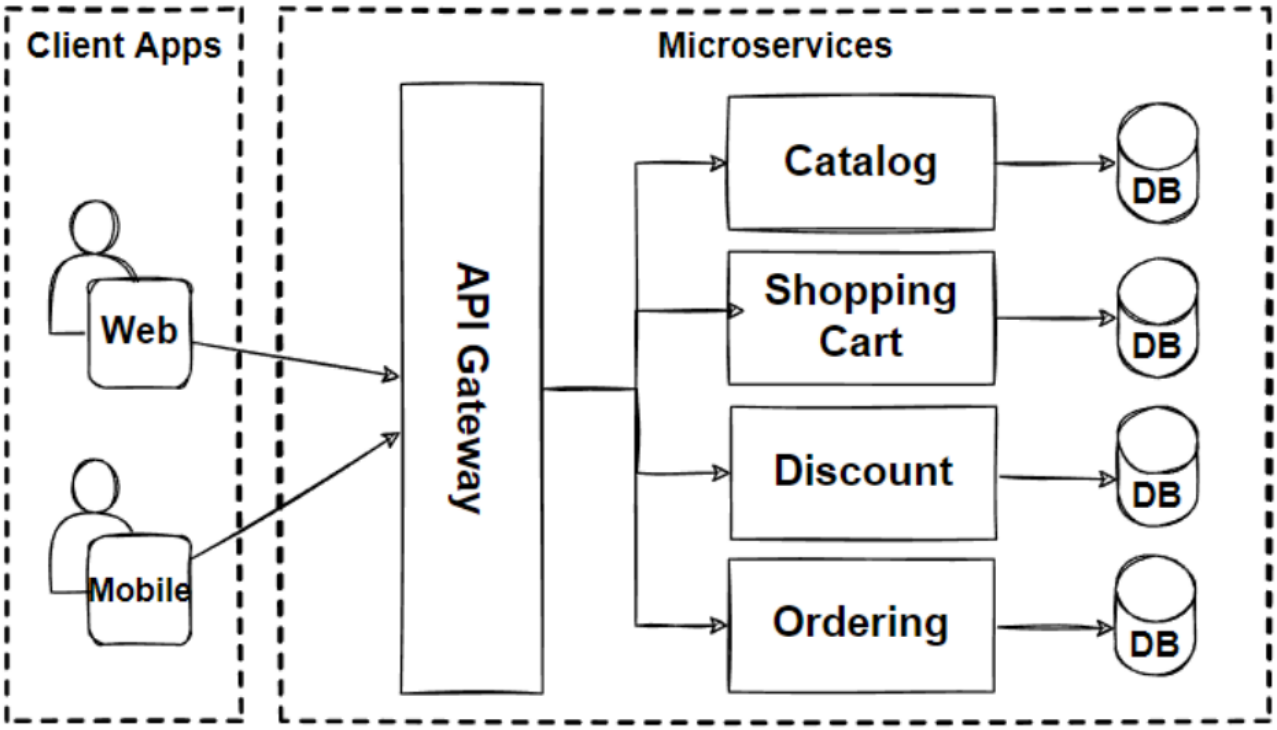

Рис. 3. Мікросервісна структура.

DevOps і Observability. DevOps-практики є невід'ємною частиною успішної реалізації мікросервісної архітектури. Оскільки система складається з великої кількості незалежних сервісів, автоматизація всіх етапів — від розробки до розгортання — є критично важливою [3].

CI/CD (Continuous Integration / Continuous Delivery): автоматизовані пайплайни забезпечують збирання, тестування та деплой кожного сервісу незалежно. Це дозволяє впроваджувати зміни швидше, з меншою кількістю помилок. Наприклад, GitLab CI/CD, GitHub Actions, Jenkins або Argo CD для Kubernetes надають засоби для такого процесу [1].

Канарейкові релізи, A/B тестування, Feature Toggles: мікросервіси дозволяють реалізовувати розгортання нових функцій поступово. Канарейковий реліз дозволяє запускати оновлення лише для частини користувачів, тоді як A/B тестування дає змогу перевіряти ефективність змін. Feature toggles дозволяють увімкнути чи вимкнути певну функціональність без зміни коду [17,19].

Моніторинг: для виявлення помилок, перевантаження або затримок необхідне постійне спостереження за всіма сервісами. Prometheus є стандартом для збору метрик, а Grafana — інструментом для їх візуалізації. Крім того,

застосовуються такі сервіси як Datadog, NewRelic, або Dynatrace для хмарних середовищ [1].

Трасування (Distributed Tracing): щоб зрозуміти, як запит проходить через десятки мікросервісів, використовують трасування — наприклад, Jaeger або Zipkin. Це дозволяє ідентифікувати вузькі місця в ланцюгу обробки запитів та покращити продуктивність системи [3].

Централізоване логування: логування з усіх сервісів агрегується в єдине сховище, зазвичай з використанням ELK-стека (Elasticsearch, Logstash, Kibana) або EFK (з Fluentd замість Logstash). Це дозволяє ефективно шукати, аналізувати і фільтрувати повідомлення про помилки або інформацію про активність користувачів [18, 19].

Алерти та автозцілення: система повинна сама повідомляти про проблеми. За допомогою Alertmanager можна налаштувати сповіщення у Slack, Email, Telegram. Також важливо мати механізми автоматичного відновлення — наприклад, перезапуск підозрілої інстанції або переміщення навантаження [17].

Інструменти Chaos Engineering: для перевірки стійкості системи можна використовувати практику інжектування збоїв (наприклад, за допомогою Chaos Monkey від Netflix). Це дозволяє гартувати систему до реальних збоїв [17].

DevOps і Observability в мікросервісній архітектурі — не просто технічна необхідність, а умова стабільності, безпеки й здатності швидко реагувати на зміни. Без якісної автоматизації та моніторингу велика кількість сервісів швидко перетвориться на некерований хаос [17].

Безпека

Важливою є Zero Trust Model — тобто навіть всередині мережі сервіси не мають довіри один до одного [16]. Обов'язково застосовуються:

TLS-з'єднання між сервісами

Ідентифікація за допомогою JWT, OAuth2

Контроль доступу (RBAC, ABAC)

Сканування образів, залежностей (Trivy, Snyk)

ВИСНОВОК ДО РОЗДІЛУ 1

Монолітна архітектура залишається актуальним підходом у веб-розробці, особливо для невеликих команд або проєктів, які потребують швидкого старту, простої інфраструктури та мінімального оверхеду. Її головною перевагою є простота — як у розгортанні, так і в розробці. Всі компоненти взаємодіють у межах одного процесу, що полегшує дебаг, тестування та забезпечення безпеки. Моноліт підходить для тих проєктів, які не передбачають частих змін, високого навантаження або великої кількості команд. Він також забезпечує кращу цілісність бізнес-логіки, адже всі модулі розташовані в одному кодовому просторі [1,2,3,1]9.

Проте зі зростанням команди, кількості функцій і вимог до масштабування, монолітна архітектура може стати вузьким місцем. Будь-яка зміна потребує повторного деплою всього застосунку, а помилка в одному модулі може призвести до збоїв усієї системи [1,2,3,5,9].

У сучасних умовах, коли масштабованість, незалежна розробка та часті релізи стають критичними, мікросервіси часто є кращим вибором. Але моноліт — це не архаїзм, а цілком обґрунтоване рішення для багатьох реальних сценаріїв. Іноді найкраща стратегія — почати з моноліту, а з часом перейти до мікросервісів у міру зростання вимог [1,3,14,16,18].

Мікросервісна архітектура є потужною відповіддю на сучасні виклики розробки масштабованих і динамічних веб-систем. Вона дозволяє розділити складні програмні продукти на керовані, автономні компоненти, які можуть розвиватися, масштабуватися й обслуговуватися незалежно одне від одного. Завдяки цьому компанії можуть швидше впроваджувати нові функції, адаптуватися до потреб ринку та знижувати ризики глобальних збоїв [1,4,5,6].

Втім, мікросервіси не є універсальним рішенням для всіх задач. Вони приносять із собою складність інфраструктури, вищі вимоги до DevOps-практик, необхідність ретельного моніторингу та безперервного тестування. Успіх мікросервісної архітектури напряму залежить від зрілості команди, культури автоматизації та чіткого технічного бачення [1,2,3,7,8].

РОЗДІЛ 2
ВИБІР ЗАСОБІВ ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

2.1. Порівняння монолітної та мікросервісної архітектури.

У веб-розробці вибір архітектурного підходу — критичне рішення, що впливає на ефективність розробки, масштабування, підтримку й адаптацію продукту до змін. Два найпоширеніші архітектурні підходи — моноліт і мікросервіси — мають суттєві відмінності, переваги та виклики. Нижче наведено докладне порівняння цих підходів у контексті реальних вимог сучасних веб-систем [1].

Основи архітектур. Монолітна архітектура — це підхід, за якого весь код додатку — від логіки до шаблонів і доступу до бази даних — розміщений в одному кодовому просторі. Додаток працює як єдине ціле. Він компілюється й розгортається як єдиний блок. Така структура дозволяє просту розробку, тестування та деплоймент [2]. Мікросервісна архітектура, навпаки, розбиває застосунок на набір незалежних сервісів, кожен з яких реалізує окрему бізнес-функцію та взаємодіє з іншими через мережу (зазвичай HTTP або messaging-брокери). Сервіси мають свою базу даних, незалежний життєвий цикл і технологічну свободу [3,6,9].

Історичний контекст. Моноліт був єдиним підходом на ранніх етапах розвитку веб. З виходом на сцену великих розподілених систем, компанії почали стикатися з обмеженнями масштабування. Це призвело до еволюції в бік SOA, а пізніше — до мікросервісів. Компанії на зразок Netflix, Amazon і Spotify впровадили мікросервісну архітектуру, щоб забезпечити гнучкість і масштабованість у відповідь на зростаючі навантаження [4].

Структура коду. Моноліт має централізовану кодову базу. Це спрощує навігацію, рефакторинг і перевірку цілісності логіки. Але з часом така база може перетворитися на "monolith of doom", де зміни однієї частини можуть неочікувано зламати іншу [5,7]. У мікросервісах кожен сервіс має свій репозиторій або модуль. Це дозволяє незалежний розвиток, використання

різних мов програмування, стандартів кодування. Однак це ускладнює загальну координацію та потребує сильного DevOps-супроводу [6,10].

Масштабування. Моноліт масштабується як єдине ціле. Якщо лише одна частина додатку потребує додаткових ресурсів — масштабувати доведеться всю систему [7,11]. Мікросервіси дозволяють горизонтальне масштабування окремих сервісів. Наприклад, у періоди високого навантаження на кошик чи платіжний сервіс можна підняти більше їхніх інстанцій, не чіпаючи решту системи [8,10].

Командна організація. Моноліт сприяє централізованій розробці. Це підходить для невеликих команд або проєктів на ранньому етапі [9].

Мікросервіси дозволяють організувати команди навколо сервісів. Кожна команда може мати повну відповідальність за свій сервіс: розробку, тестування, деплоймент і моніторинг. Це підвищує автономність і швидкість роботи [10].

Розгортання та CI/CD. Моноліт вимагає повторного розгортання всієї системи після змін. Це спрощує процес, але підвищує ризики збоїв. Потрібна синхронізація між розробниками [11]. Мікросервіси дозволяють деплой кожного сервісу незалежно. CI/CD-пайплайни налаштовуються окремо, можна застосовувати стратегії A/B тестування, канарейкові релізи. Але при цьому збільшується кількість пайплайнів і точок збоїв [12].

Надійність і відмовостійкість. Моноліт уразливий до збоїв — якщо впала одна частина, страждає вся система. Виняток — ретельно організовані внутрішні exception-handling механізми [13]. У мікросервісах кожен компонент працює ізольовано. Завдяки шаблонам Resilience (наприклад, Circuit Breaker, Retry, Bulkhead) система стає більш стійкою. Але і складність у тестуванні взаємодій зростає [14].

Тестування. У моноліті легко налаштовуються інтеграційні, юніт і E2E тести — все в одному контексті [1,5]. У мікросервісах потрібно створювати мок-сервіси, стейджинг середовища, інтеграційні тести між сервісами. Це складніше, але дозволяє протестувати систему ближче до реальної поведінки [16,18].

Безпека. Моноліт має централізовану безпеку, наприклад, через middleware, єдину систему сесій, аутентифікації та авторизації [17].

Мікросервіси вимагають впровадження міжсервісної безпеки: TLS, OAuth2, JWT, сервісної автентифікації, Zero Trust. Також можуть бути використані service mesh-інструменти типу Istio або Linkerd [18].

Інфраструктура. Монолітна архітектура не потребує складної інфраструктури. Розгортається одним контейнером або на одному сервері [19].

Мікросервіси потребують розвиненої інфраструктури: Kubernetes, моніторинг, логування, discovery-сервіси, централізовані конфігурації. Це підвищує поріг входу й вартість утримання [20].

Моніторинг і Observability. Моноліт моніториться як одна система. Усі логи, помилки й метрики зберігаються централізовано [1, 2,5].

У мікросервісах Observability — окрема дисципліна: агреговане логування (ELK, Loki), трасування (Jaeger, Zipkin), метрики (Prometheus, Grafana), алерти, Service Level Indicators (SLI), Service Level Objectives (SLO) [1,3,5].

Коли обирати моноліт:

- MVP або стартапи
- Невеликі команди
- Простий бізнес-процес
- Обмежені ресурси
- Потреба в швидкому старті [2, 3]

Коли обирати мікросервіси:

- Високе навантаження
- Велика кількість команд
- Часті оновлення продукту
- Потреба в масштабуванні окремих компонентів
- Багато різних бізнес-ліній у продукті [2, 4]

Обидва підходи мають місце в сучасній веб-розробці. Моноліт пропонує швидкість і простоту на ранніх етапах, а мікросервіси — гнучкість і масштабованість для великих, довготривалих продуктів. Головне — обрати архітектуру, що найкраще відповідає вимогам бізнесу, технічному рівню

команди та очікуваному навантаженню. Часто найкраща стратегія — почати з моноліту й перейти до мікросервісів у міру зростання [2, 5].

2.2. Вибір комбінованого підходу

У сучасному світі розробки програмного забезпечення важливо знаходити баланс між стабільністю, швидкістю розробки, масштабованістю і гнучкістю системи. Багато компаній стикаються з вибором архітектурного підходу: моноліт чи мікросервіси. Проте, все частіше зустрічається стратегія комбінування обох підходів для досягнення найкращих результатів. Ваш проєкт, який поєднує моноліт Laravel і мікросервіси на AWS Lambda з Node.js і Serverless Framework, є прекрасним прикладом такої гібридної архітектури. У цьому тексті ми детально розглянемо, чому цей підхід є дуже вигідним, розкриємо його переваги та практичні аспекти [1,3,4,7,9].

Баланс між стабільністю моноліту і гнучкістю мікросервісів. Монолітні системи, побудовані на Laravel, відомі своєю стабільністю, багатою функціональністю та відпрацьованою екосистемою. Laravel пропонує зручні інструменти для розробки, організації коду, тестування, маршрутизації, роботи з БД та кешем. Для бізнес-логіки, де важлива цілісність, а також для компонентів, які не часто змінюються, моноліт є природнім вибором [3].

AWS Lambda, навпаки, дає можливість розробляти та розгортати окремі функції як мікросервіси, які масштабуються незалежно, працюють за подіями (event-driven) та оплачується за фактичне використання. Це ідеально для функцій з непередбачуваним або змінним навантаженням, або для нових компонентів, які часто розвиваються [1].

Комбінуючи обидва підходи, ви зберігаєте надійність і простоту моноліту для ядра бізнесу, і водночас отримуєте гнучкість та швидкість розвитку, яку дають мікросервіси [1,4,6].

Прискорене впровадження нових функцій і масштабування. Моноліт, особливо Laravel, чудово підходить для швидкого прототипування і впровадження базових функціональностей. Ви можете швидко реалізувати API, інтерфейси, бізнес-логіку в одному проекті.

Однак, коли певні функції починають потребувати значного масштабування (наприклад, обробка великих обсягів даних, інтеграція з зовнішніми сервісами, аналітика), варто винести їх у мікросервіси на AWS Lambda. Завдяки автоматичному масштабуванню Lambda ви уникаєте проблем перевантаження серверів моноліту, забезпечуючи при цьому високу доступність [1,3,7, 15].

Крім того, нові функції можна розробляти і розгортати автономно, не зачіпаючи стабільність основного моноліту. Це дозволяє знизити ризики та пришвидшити релізи [1,2,3].

Використання кращих технологій під кожне завдання. Мікросервіси дають свободу вибору технологій. Якщо в Laravel основна логіка написана на PHP, то для мікросервісів можна вибрати Node.js, Python, Go або будь-яку іншу мову, що найкраще підходить під конкретну задачу 12, 18].

Node.js і AWS Lambda — це ідеальне поєднання для подій, що вимагають асинхронної обробки, наприклад, роботи з API Gateway, обробки потоків, роботи з базами даних NoSQL, або інтеграції з іншими AWS-сервісами 1,3,4,5].

Таким чином, у вашому проекті ви використовуєте Laravel там, де це доцільно, і Lambda там, де потрібна висока продуктивність та масштабованість.

Економія ресурсів завдяки serverless-інфраструктурі. Один з головних плюсів AWS Lambda — оплата тільки за фактичне використання ресурсу, без потреби тримати сервери постійно включеними. Це особливо вигідно для функцій, які працюють не безперервно, а за подіями [1,2,5,6].

З монолітом ви орендуєте сервер або віртуальну машину, яка завжди працює. Поєднання дозволяє оптимізувати вартість — критичні, постійно використовувані частини на моноліті, а сплескові, нерегулярні або експериментальні — в Lambda [1,2,6].

Підвищення надійності системи. У моноліті збій одного компонента може вплинути на всю систему. Використання мікросервісів на Lambda ізолює критичні частини системи, зменшуючи вплив потенційних збоїв [19].

AWS Lambda автоматично перезапускає функції, має вбудовану масштабованість і моніторинг, що дозволяє швидко виявляти і усувати проблеми [13,15].

Це підвищує загальну відмовостійкість системи. Простота інтеграції з AWS-екосистемою. AWS пропонує широкий набір сервісів (S3, DynamoDB, SNS, SQS, API Gateway, CloudWatch), які можна легко інтегрувати з Lambda.
Це дає можливість швидко будувати розширені функції, наприклад, обробку завантажень, асинхронні черги, сповіщення, аналітику і т. Д [13,15].

Інтеграція моноліту з цими сервісами можлива через SDK AWS або REST API, а Lambda функції можуть працювати з ними більш природно, що спрощує архітектуру [12,15,16].

Кращий контроль над життєвим циклом релізів. Розділення системи на моноліт і мікросервіси дозволяє планувати релізи окремо: моноліт розгортається за класичним циклом, а Lambda функції можуть оновлюватись щохвилини через serverless framework. Це дозволяє уникнути великих "вікон" для релізів, зменшує ризики і дає змогу швидше реагувати на зміни ринку або потреби користувачів [12,14,15,17].

Полегшення переходу до cloud-native. Поєднання Laravel і AWS Lambda — це крок у бік cloud-native архітектури без повного рефакторингу існуючого коду. Ви зберігаєте стабільність і звичний розробницький процес для основної частини, і при цьому поступово інтегруєте serverless-функції, що відкриває двері для подальшої міграції та масштабування [6, 7, 13,15].

Підвищення продуктивності команд. Розподіл між монолітом і мікросервісами дозволяє команді паралельно працювати над різними частинами системи без конфліктів у коді [11].

Команди, що працюють над Lambda, можуть застосовувати інші практики, технології, CI/CD, не заважаючи монолітній команді. Це стимулює інновації і скорочує час розробки [10, 15].

Підтримка різних моделей масштабування. Моноліт масштабувати простіше за допомогою вертикального масштабування (потужніший сервер), але це має межі [1].

Lambda — це горизонтальне масштабування на рівні функцій з можливістю масштабувати окремі сервіси незалежно [14].

Поєднання дозволяє комбінувати ці підходи, забезпечуючи гнучкість і ефективність [14].

У сучасній хмарній розробці Serverless архітектура набирає дедалі більшої популярності, а AWS Lambda — один із провідних сервісів безсерверних обчислень. Вибір технологічного стека для розробки Lambda-функцій критично важливий для успіху проєкту. Node.js у поєднанні з Serverless Framework є одним з найпопулярніших і найефективніших рішень для розробки функцій Lambda. У цьому документі розглянемо, чому саме цей вибір є оптимальним, розберемо переваги Node.js і Serverless Framework у контексті AWS Lambda, наведемо практичні кейси, виклики та рекомендації [1] [2][3].

Вибір мови програмування. Чому саме Node.js?

Переваги Node.js для AWS Lambda.

Висока швидкість виконання. Node.js працює на рушії V8 (Google Chrome), що забезпечує високу продуктивність та швидкий старт функцій. Важливо для AWS Lambda, де час холодного старту критично впливає на продуктивність [4].

Асинхронність і подієвий цикл. Node.js створений для обробки асинхронних операцій — це ідеально підходить для event-driven архітектури Lambda, де функції часто працюють із мережею, базами даних, чергами [5].
Велика екосистема. NPM пропонує тисячі готових пакетів, які можна використовувати у Lambda-функціях. Це прискорює розробку та дозволяє повторно використовувати код [6].

Поширеність серед розробників. Node.js має величезну спільноту, що полегшує пошук фахівців, навчальні матеріали, приклади і підтримку [7].

Serverless Framework: що це і чому він потрібен.

Автоматизація розгортання. Serverless Framework забезпечує легкий опис інфраструктури та функцій у YAML-конфігурації, автоматично створюючи всі необхідні ресурси AWS [8].

Підтримка мультихмарності. Хоча фокус на AWS Lambda, Serverless Framework також підтримує інші cloud-провайдери — Azure, Google Cloud, що дає гнучкість у майбутньому [9].

Розвинуті плагіни та розширення. Велика кількість плагінів допомагає інтегрувати тестування, моніторинг, управління секретами, CI/CD [10].

Краща організація проекту. Чітка структура проекту і декларативне описання функцій спрощує роботу команд і масштабування [11].

ВИСНОВОК ДО РОЗДІЛУ 2

Поєднання монолітної архітектури та мікросервісів є гарним вибором, оскільки дає змогу отримати найкраще з обох світів. Моноліт забезпечує стабільність, простоту розробки та централізоване управління основною бізнес-логікою, що дозволяє швидко впроваджувати зміни та підтримувати цілісність системи. Водночас, мікросервіси дозволяють виокремити окремі функціональні блоки, що потребують гнучкості, масштабування чи особливих технологічних стеків, зменшуючи навантаження на основний застосунок та підвищуючи загальну стійкість системи [1,3,7,9,13].

Таке гібридне рішення допомагає уникнути складнощів масштабування великого моноліту, одночасно зберігаючи зручність та швидкість розробки у знайомому середовищі. Крім того, мікросервіси дозволяють командам працювати паралельно, використовуючи найкращі технології для конкретних задач, що підвищує ефективність розробки і дозволяє швидко адаптуватися до змін бізнес-вимог. Тому поєднання моноліту і мікросервісів — це збалансований підхід, що підвищує гнучкість, масштабованість і надійність сучасних веб-додатків [1,3,7,9,13].

Вибір Node.js у поєднанні з Serverless Framework для розробки AWS Lambda-функцій є надзвичайно ефективним та популярним рішенням, що надає численні переваги для створення сучасних безсерверних додатків. По-перше, Node.js базується на рушії V8, розробленому компанією Google для браузера Chrome, що забезпечує високу продуктивність та швидкість запуску коду. Це критично важливо в AWS Lambda, де холодний старт функцій впливає на час відповіді системи. Легкість та швидкість запуску дозволяють зменшити затримки й підвищити якість обслуговування користувачів. Крім того, асинхронна подієва модель Node.js чудово підходить для архітектури, орієнтованої на події, якою є AWS Lambda. Завдяки цьому можна ефективно обробляти великі обсяги паралельних запитів, взаємодіючи з мережею, базами даних і сторонніми сервісами без блокувань, що позитивно впливає на масштабованість і стабільність системи [1,3,7,9,13].

Наступною вагомою перевагою є велика екосистема Node.js та npm – найбільшого у світі пакетного менеджера. Ця екосистема надає тисячі готових бібліотек і модулів, які дозволяють прискорити розробку, повторно використовувати код і впроваджувати найкращі практики без необхідності створювати все з нуля. Широка спільнота розробників забезпечує регулярні оновлення, безпеку, підтримку та величезну кількість навчальних матеріалів. Serverless Framework, у свою чергу, спрощує автоматизацію процесів розгортання та управління інфраструктурою за допомогою декларативного опису функцій і ресурсів у YAML-конфігураціях. Це суттєво знижує кількість помилок, пришвидшує цикли розробки та релізів, а також полегшує масштабування проектів різного рівня складності [7,8,13].

Ще одна важлива перевага Serverless Framework полягає в його мультихмарній підтримці, що дає змогу легко переносити проекти між різними провайдерами або поєднувати декілька хмарних сервісів в одному додатку. Це забезпечує довгострокову гнучкість і захист від залежності від одного провайдера. Розвинена екосистема плагінів і розширень дозволяє інтегрувати практики CI/CD, моніторинг, управління секретами, що критично для створення надійних, стабільних і безпечних систем. Також інструменти для локальної емуляції Lambda-функцій полегшують розробникам тестування та налагодження, підвищуючи продуктивність роботи та якість коду [7,17,19].

Щодо масштабування, AWS Lambda підтримує автоматичне горизонтальне масштабування, що дозволяє обробляти будь-які пікові навантаження без додаткових налаштувань. Водночас, Serverless Framework забезпечує організований підхід до управління цими процесами, що важливо для підтримки стабільності та контролю якості. З економічної точки зору, Serverless-модель оплати "за використання" допомагає оптимізувати витрати, особливо в випадках нерівномірного навантаження або сезонних піків. Мінімальний розмір пакетів та оптимізований код на Node.js сприяють зменшенню часу виконання та, відповідно, зниженню вартості роботи функцій.
Безперечно, у такому підході є й виклики — наприклад, холодний старт функцій, обмеження часу виконання та пам’яті, складнощі з управлінням залежностями [7,18,19].

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ПРОЕКТУ

3.1. Опис архітектури проекту

Проект розділено на чотири окремі репозиторії, що забезпечує чіткий поділ відповідальностей і дозволяє масштабувати розробку окремих частин незалежно.

Монолітна частина.

Репозиторій містить Laravel-проект, який відповідає за основну бізнес-логіку, веб-інтерфейс, автентифікацію користувачів, роботу з базою даних і загальний контроль системи. Laravel забезпечує надійний і зрозумілий фундамент для централізованого управління.

Мікросервісна частина — 2 репозиторії Lambda-функцій.

Два окремі репозиторії містять AWS Lambda-функції, реалізовані за допомогою Serverless Framework на Node.js. Ці мікросервіси відповідають за різноманітні допоміжні та інтеграційні процеси:

- Взаємодія з AWS сервісами:

1. DynamoDB — зберігання і швидкий доступ до даних.
2. Simple Scalable Storage (S3) — для зберігання файлів та резервних копій.
3. CloudWatch — моніторинг і логування подій.
4. Simple Queue Service (SQS) — обробка асинхронних черг повідомлень.
5. Simple Email Service (SES) — надсилання електронної пошти.
6. Certificate Manager (ACM) — управління SSL-сертифікатами для захищеного зв’язку.
7. EventBridge — оркестрація подій між сервісами.

- Взаємодія зі сторонніми сервісами:

1. Uptime Monitor — контроль доступності сайтів.
2. Stripe — керування платежами та підписками.

- Двостороння взаємодія з плагіном: основна роль — комунікація між серверними функціями і встановленим плагіном на WordPress сайтах, що дозволяє обмінюватись даними в режимі реального часу, віддалено керувати станом сайту, запускати оновлення і отримувати звіти.

Репозиторій плагіна

Містить код плагіна для WordPress, який встановлюється на сайти клієнтів. Плагін відповідає за передачу даних, аутентифікацію запитів, взаємодію з API мікросервісів, і виконання команд, отриманих від Lambda функцій.

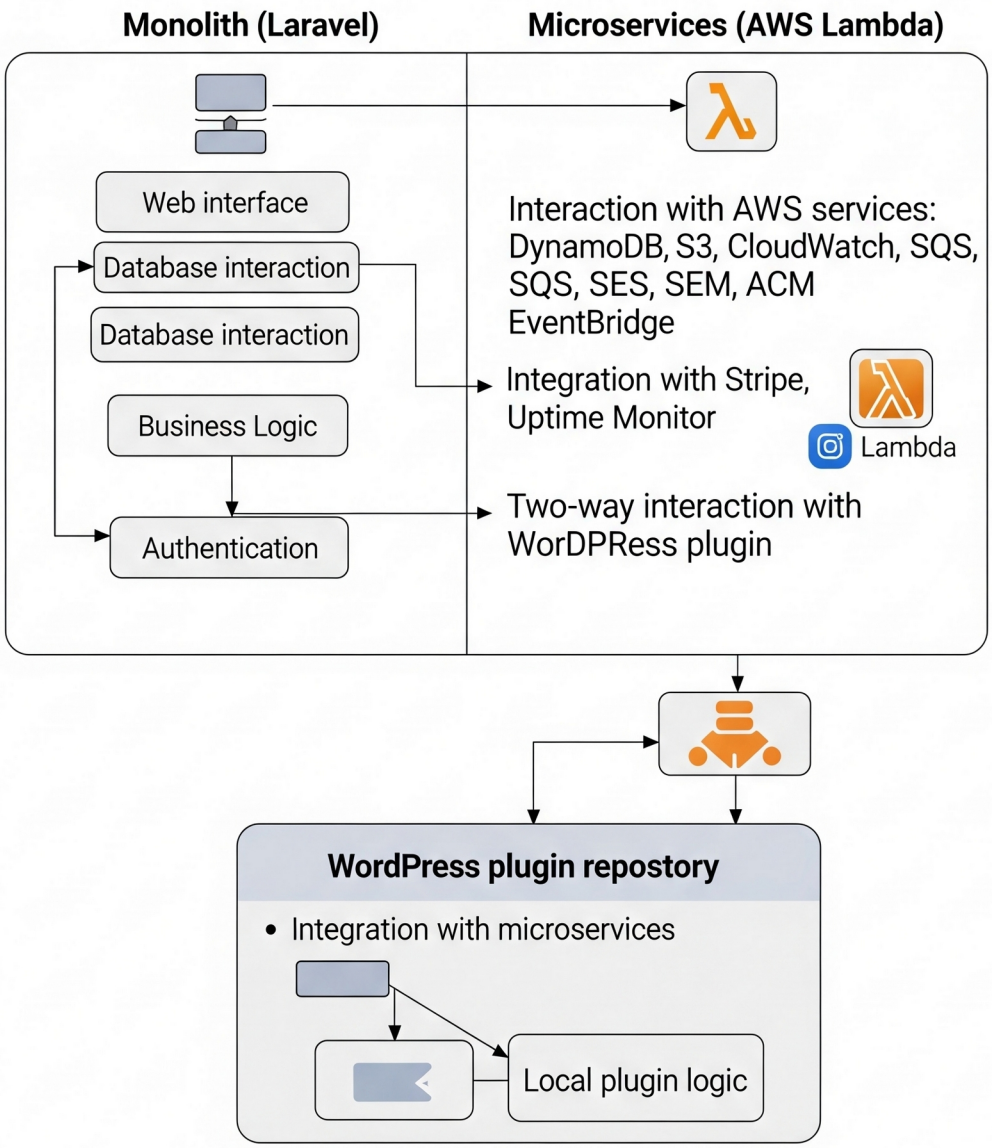

Рис. 4. Схема структури проекта

3.2. Структура serverless мікросервіса

Структура serverless мікросервіса складається з наступних елементів:

_json - JSON-файли, для даних, конфігурацій та тестів.

Config - ця папка містить файли конфігурації для різних середовищ або налаштувань програми. У контексті Serverless, тут зберігаються такі файли, як config.prod.json, на які посилається serverless.yml для динамічного завантаження змінних середовища та параметрів VPC.

Lib - основна частина проекту, містить основний код проекту.

node_modules - стандартна папка Node.js, яка містить усі залежності проекту, встановлені за допомогою менеджера пакетів (наприклад, npm або yarn).

Spec - ця папка зазвичай містить файли для тестування (специфікації тестів). Це може бути для юніт-тестів або інтеграційних тестів.

Gitignore - файл вказує Git, які файли та папки слід ігнорувати та не включати до репозиторію (наприклад, node_modules/, лог-файли, змінні середовища).

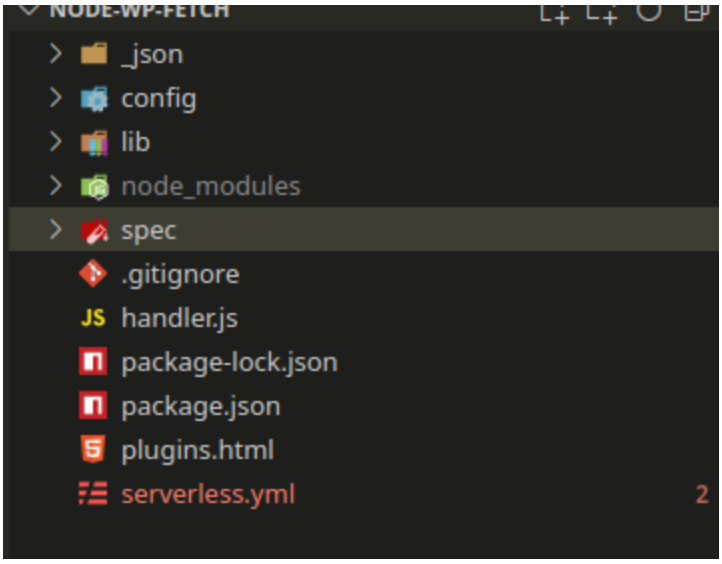

Рис. 5. Структура serverless мікросервіса

handler.js - це основний файл, який містить код функцій AWS Lambda. У файлі serverless.yml вище, багато функцій посилаються на handler.js як на свій обробник (наприклад, handler.connect, handler.getPlugins). Це означає, що всі ці функції експортуються з цього єдиного файлу.

package-lock.json - файл, який автоматично генерується npm (менеджером пакетів Node.js). Він фіксує точну версію кожної залежності проекту та її підзалежностей, забезпечуючи відтворювані збірки.

package.json - цей файл є маніфестом проекту Node.js. Він містить метадані проекту (назва, версія, опис), а також перелік його залежностей (які встановлюються в node_modules/) та скрипти для виконання завдань (наприклад, запуск тестів, збірка).

serverless.yml - файл конфігурації для фреймворку Serverless. Він визначає, як проект розгортається на AWS, включаючи функції Lambda, тригери API Gateway, бази даних DynamoDB, дозволи IAM та інші ресурси AWS, як ми бачили у попередньому аналізі. serverless.yml є основною конфігурацією для

бессерверного додатку , розгорнутого на AWS за допомогою фреймворку Serverless. Цей файл визначає інфраструктуру, функції AWS Lambda, їхні тригери та дозволи, необхідні для роботи сервісу.

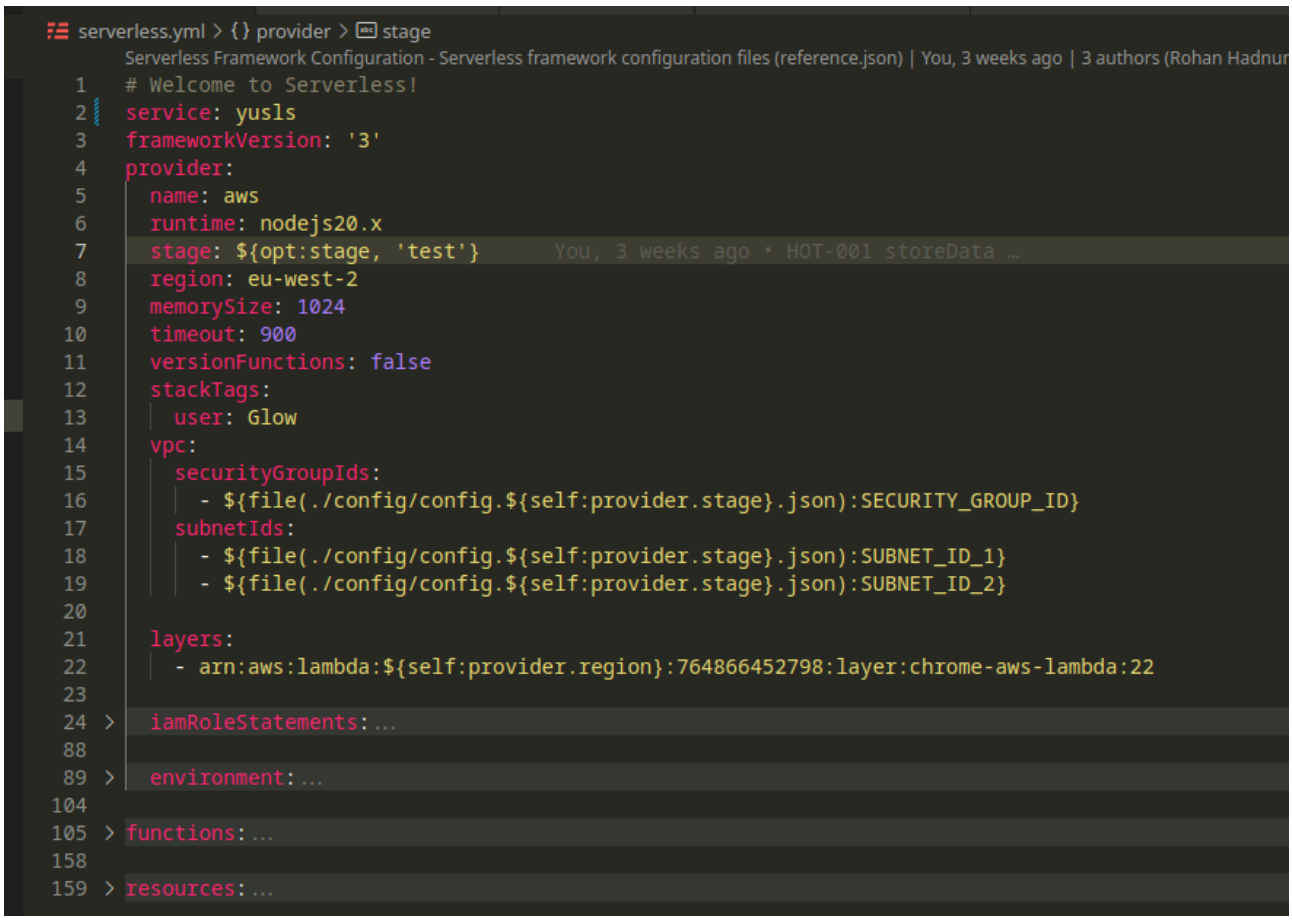

Рис. 6. Структура serverless.yml

Ми використовуємо Serverless Framework версії 3.

У секції provider описуємо провайдера хмарних послуг та загальні налаштування для всього сервісу. name: aws - для розгортання використовуємо Amazon Web Services (AWS). Вказуємо середовище виконання для функцій AWS Lambda - Node.js версії 20.x. stage: ${opt:stage, 'test'} - визначаємо стадію розгортання проекту. Значення може бути передано як опція командного рядка (наприклад, --stage dev). Якщо опція не вказана, за замовчуванням використовується 'test'. Це дозволяє підтримувати різні конфігурації для розробки, тестування та продакшну. region: eu-west-2 - вказуємо регіон AWS (Лондон), де розгортаємо всі ресурси сервісу. Встановлюємо обсяг пам'яті (1024 МБ) для кожної функції Lambda та timeout: 900 - максимальний час виконання (900 секунд або 15 хвилин) для функцій Lambda. Якщо функція виконується довше цього часу, вона буде примусово завершена.

У блоці vpc вказуємо, що функції Lambda будуть розгорнуті всередині Virtual Private Cloud (VPC). Це підвищує безпеку та дозволяє функціям отримувати доступ до приватних ресурсів у мережі AWS (наприклад, баз даних RDS).

Тут далі визначає ідентифікатори груп безпеки, до яких будуть належати функції Lambda. Групи безпеки контролюють мережевий трафік (вхідний/вихідний) до та від функцій. Значення ідентифікатора групи безпеки динамічно завантажується з файлу config.json, специфічного для поточної стадії розгортання. Також у vpc вказуємо ідентифікатори підмереж у VPC, де будуть розміщені функції Lambda. Розміщення у кількох підмережах (зазвичай у різних зонах доступності) покращує відмовостійкість.

У розділі layers додаємо шари до функцій Lambda. Шари використовуються для пакування залежностей, допоміжних бібліотек або кастомних рантаймів, які можуть бути спільними для багатьох функцій.

ARN arn:aws:lambda:${self:provider.region}:764866452798:layer:chrome-aws-lambda:22 вказує на шар chrome-aws-lambda, версії 22, який, містить Chromium для використання в бессерверному середовищі (наприклад, для генерації PDF, скріншотів або веб-скрапінгу). Використання шару допомагає зменшити розмір розгорнутого пакета функції та потенційно прискорити "холодний старт".

Розділ iamRoleStatements визначає дозволи (політики IAM), які будуть призначені ролі Lambda, створеній для цього сервісу. Ці дозволи дозволяють вашим функціям взаємодіяти з іншими сервісами AWS (наприклад, DynamoDB, S3, SQS).

Розділ environment містить змінні середовища, які будуть доступні всім функціям Lambda під час виконання. Як правило, це конфігураційні параметри, такі як імена таблиць, ключі API, URL-адреси, які можуть відрізнятися для різних середовищ.

Розділ functions визначає кожну функцію AWS Lambda, яка є частиною цього сервісу. Тут вказується шлях до коду функції (handler), її тригери (наприклад, HTTP-запити через API Gateway, події S3, події SQS) та будь-які специфічні для функції налаштування.

Розділ resources дозволяє Serverless Frameworky створювати або керувати додатковими ресурсами AWS, які не є безпосередньо функціями Lambda або API Gateway. Це можуть бути бази даних DynamoDB, черги SQS, бакети S3, ролі IAM та інші компоненти інфраструктури, необхідні для роботи сервісу.

Рис. 7. Структура розділку iamRoleStatements.

Розділ iamRoleStatements визначає дозволи (права доступу) для IAM-ролі, яка буде призначена функціям AWS Lambda. Ці дозволи дозволяють функціям взаємодіяти з іншими сервісами AWS.

Дозволи для SQS (Simple Queue Service):

Effect "Allow" дозволяє наступні дії:

sqs:SendMessage - дозволяє надсилати повідомлення в чергу SQS.

sqs:GetQueueUrl - дозволяє отримати URL черги SQS.

sqs:ReceiveMessage - дозволяє отримувати повідомлення з черги SQS.

sqs:DeleteMessage - дозволяє видаляти повідомлення з черги SQS.

sqs:GetQueueAttributes - дозволяє отримувати атрибути черги SQS.

sqs:ListQueues - дозволяє перелічувати всі черги SQS.

Resource "arn:aws:sqs:${self:provider.region}:337968008125:${file(./config/config.${self:provider.stage}.json):QUEUE_NAME}" тут вказуємо що дії дозволені для конкретної черги SQS, ім'я якої динамічно береться з файлу конфігурації.

Дозволи для DynamoDB:

dynamodb:PutItem - дозволяє додавати нові елементи в таблицю DynamoDB.

dynamodb:BatchWriteItem - дозволяє записувати кілька елементів в одну або кілька таблиць DynamoDB.

dynamodb:GetItem - дозволяє отримувати один елемент з таблиці DynamoDB за його первинним ключем.

dynamodb:DeleteItem - дозволяє видаляти елемент з таблиці DynamoDB.

dynamodb:UpdateItem - дозволяє оновлювати існуючий елемент у таблиці DynamoDB.

dynamodb:Query - дозволяє виконувати операції запиту до таблиць або індексів DynamoDB.

QueryDataByOwners - дія дозволена для конкретного індексу QueryDataByOwners таблиці G_DATA_TABLE_NAME. Ці дії дозволені для кількох таблиць DynamoDB, імена яких динамічно завантажуються з файлу конфігурації.

Розділ environment визначає змінні середовища, які будуть доступні для всіх функцій AWS Lambda в цьому Serverless-сервісі.

Кожна змінна середовища завантажує своє значення динамічно з файлу конфігурації, специфічного для поточної стадії розгортання (./config/config.${self:provider.stage}.json). Це забезпечує гнучкість і дозволяє мати різні значення для розробки, тестування та продакшну без зміни коду.

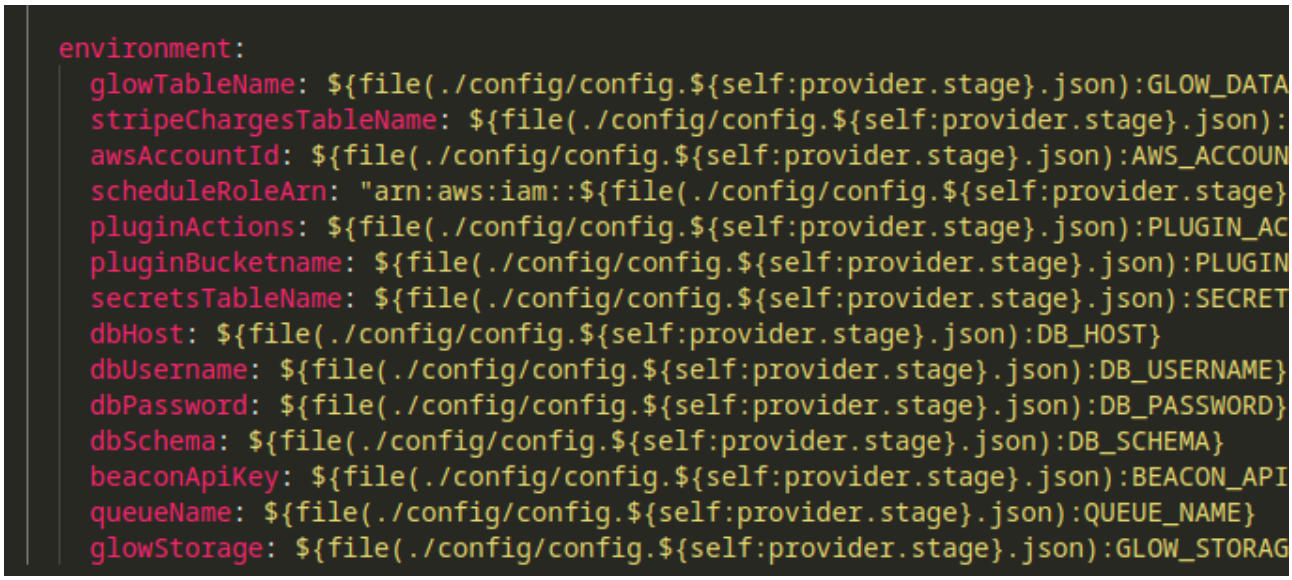

Рис. 7. Структура розділу environment.

glowTableName: це назва таблиці DynamoDB, що використовується для зберігання основних даних додатку "Glow".

stripeChargesTableName: Назва таблиці DynamoDB, яка, використовується для зберігання інформації про транзакції (платежі) зі Stripe.

awsAccountId: Ідентифікатор облікового запису AWS, в якому розгортаються ресурси.

scheduleRoleArn: ARN (Amazon Resource Name) IAM-ролі, яка використовується для планування завдань або активації функцій за розкладом.

pluginActions: це змінна, яка містить список дозволених дій або конфігурацій, пов'язаних з плагінами.

pluginBucketname: Назва бакета S3, пов'язаного з плагінами, для їх зберігання або інших операцій.

secretsTableName: Назва таблиці DynamoDB, яка використовується для зберігання секретів та конфіденційних даних.

dbHost: Хост (адреса) реляційної бази даних , до якої функції Lambda будуть підключатися.

dbUsername: Ім'я користувача для підключення до бази даних.

dbPassword: Пароль для підключення до бази даних.

dbSchema: Назва схеми або бази даних, до якої буде підключатися додаток.

beaconApiKey: API-ключ для "Beacon" сервісу.

queueName: Назва черги SQS, яка використовується для обміну повідомленнями між компонентами системи.
glowStorage: Назва бакета S3, який, використовується для загального зберігання даних або файлів, пов'язаних із додатком.

Розділ functions визначає функції AWS Lambda, які є частиною бессерверного сервісу glow-data-api, та їхні конфігурації. Наприклад, функція getPlugins має такий опис:

```
  getPlugins:
    handler: handler.getPlugins
    events:
      - http:
          path: /wp/{secretId}/plugins
          method: get
          request:
            parameters:
              paths:
                secretId: true
          private: true
```

```yaml
functions:
  wsconnect:
    handler: handler.connect
  wsdisconnect:
    handler: handler.disconnect
  wsdefault:
    handler: handler.default
  wssend:
    handler: handler.send
    events:
      - http:
          path: /ws/send
          method: post
          private: true
  deleteCoreErrors:
    handler: handler.deleteCoreErrors
  getPlugins:
    handler: handler.getPlugins
    events:
      - http:
          path: /wp/{secretId}/plugins
          method: get
          request:
            parameters:
              paths:
                secretId: true
          private: true
  getThemes:
    handler: handler.getThemes
    events:
      - http:
          path: /wp/{secretId}/themes
          method: get
          request:
            parameters:
              paths:
                secretId: true
          private: true
  getCore:
    handler: handler.getCore
    events:
      - http:
          path: /wp/{secretId}/core
          method: get
          request:
            parameters:
              paths:
                secretId: true
          private: true
  checkCredentials:
    handler: handler.checkCredentials
    events:
      - http:
          path: /wp/credentials/{secretId}
          method: get
          request:
            parameters:
              paths:
                secretId: false
          private: true
```

Рис. 8. Структура розділу functions

3.3 Опис AWS Lamda функції.

Детальний опис функції getPlugins.

Назва функції: getPlugins - це унікальна назва функції Lambda в рамках даного Serverless-сервісу. Вона використовується фреймворком Serverless для ідентифікації та керування цією конкретною функцією в AWS.

Обробник (Handler): handler.getPlugins - цей параметр вказує на те, який код буде виконуватися, коли функція getPlugins буде викликана.

handler означає, що файл з кодом функції називається handler.js (або handler.ts, якщо використовується TypeScript).

.getPlugins означає, що всередині цього файлу handler.js має бути експортована функція з назвою getPlugins (наприклад, module.exports.getPlugins = async (event) => { ... }; для Node.js). Таким чином, коли AWS Lambda отримує запит на виконання getPlugins, вона запустить код, що міститься у функції getPlugins у файлі handler.js.

Події (Events) - цей розділ визначає, які події AWS (або зовнішні тригери) будуть викликати функцію getPlugins. У даному випадку використовується один тип події: HTTP-запит.

Тип події: http - це означає, що функція буде інтегрована з Amazon API Gateway, який діятиме як "вхідна точка" для HTTP-запитів, перенаправляючи їх до цієї Lambda-функції.

path: /wp/{secretId}/plugins - це URL-шлях (ендпоінт) в API Gateway, за яким буде доступна ця функція.

/wp/ вказує на те, що цей ендпоінт пов'язаний з функціоналом WordPress.

{secretId} є параметром шляху (path parameter). Це означає, що частина URL буде динамічною і представлятиме собою ідентифікатор (secretId). Наприклад, запит на /wp/abc123xyz/plugins передасть значення abc123xyz як secretId до вашої Lambda-функції.

method: get - визначає HTTP-метод, який повинен бути використаний для доступу до цього ендпоінту. У даному випадку це GET, що зазвичай використовується для отримання (читання) даних.

Request - цей блок визначає очікувані параметри запиту.

Parameters: paths: secretId: true - вказує, що параметр шляху secretId є обов'язковим. Якщо запит не містить значення для secretId у шляху, API Gateway поверне помилку, перш ніж передати запит до Lambda-функції.

Значення цього параметра буде доступне в об'єкті event всередині вашої Lambda-функції (наприклад, event.pathParameters.secretId).

private: true - параметр є ключовим для безпеки. Він означає, що API Gateway вимагатиме API-ключ для доступу до цього ендпоінту. Запити без дійсного API-ключа або з недійсним ключем будуть відхилені API Gateway, не досягаючи Lambda-функції. Це забезпечує захист ендпоінту від несанкціонованого доступу.

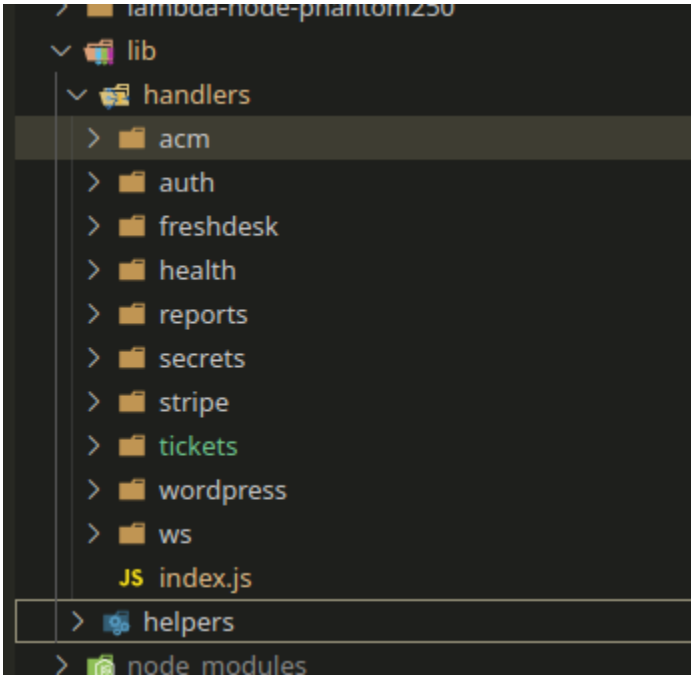

Рис. 9. Структура lib

Структура lib.

lib - коренева папка, містить основний вихідний код програми.

handlers - основна папка для обробників функцій Lambda. Її внутрішня структура розбита за функціональними областями, що є гарною практикою для великих бессерверних проектів. Кожна підпапка всередині handlers/ відповідає за певний набір пов'язаних функцій:

acm - містить код обробників для функцій, пов'язаних з AWS Certificate Manager (ACM), таких як запит сертифікатів або їх прив'язка до балансувальників навантаження (як ми бачили у serverless.yml).

Auth - Містить обробники для функцій, пов'язаних з автентифікацією, наприклад, генерація підписів (/auth/signature ендпоінт).

Freshdesk - містить обробники для інтеграції з Freshdesk, що є системою підтримки клієнтів.

Health - обробники для функцій перевірки стану або "здоров'я" сервісів (/health/{secretId} ендпоінт).

Reports - містить обробники для функцій, пов'язаних зі створенням та отриманням звітів в форматі pdf.

Secrets - обробники для керування "секретами" та обліковими даними (наприклад, створення, оновлення, перевірка відповідності секретів WordPress).

Stripe - містить обробники для функцій, пов'язаних з інтеграцією платіжної системи Stripe (керування платежами, цінами, підписками).

Tickets - обробники для функціоналу системи тікетів (створення, оновлення, відповіді на тікети, отримання списків тікетів).

wordpress/: Містить обробники для функцій, пов'язаних з WordPress (отримання інформації про плагіни, теми, ядро, керування оновленнями та секретами WordPress).

Ws - Обробники для функцій WebSocket (з'єднання, відключення, надсилання повідомлень).

index.js - головний файл обробника в lib/handlers/. Він виступає як центральна точка експорту для всіх Lambda-функцій. Тобто, всі обробники, на які посилаються у serverless.yml (наприклад, handler.getPlugins, handler.connect), насправді експортуються з цього файлу index.js, який, у свою чергу, імпортує функції з підпапок (наприклад, acm/, auth/ тощо).

helpers/: Ця папка, містить допоміжні функції, які можуть використовуватися кількома обробниками. Це місце для функцій, що виконують загальні завдання, такі як форматування даних, валідація, взаємодія з AWS SDK тощо.

3.4. Приклад AWS Lambda функції

Розглянемо одну з AWS Lambda функцій, наприклад, фукнкція attachCertificate.js, яка є обробником для приєднання SSL-сертифіката до балансувальника навантаження.

```
// Імпорт необхідних клієнтів AWS SDK v3
// ElasticLoadBalancingV2Client та AddListenerCertificatesCommand використовуються для взаємодії з Application Load Balancer (ALB)
```

```
const { ElasticLoadBalancingV2Client, AddListenerCertificatesCommand, } = require("@aws-sdk/client-elastic-load-balancing-v2");
// ACMClient та ListCertificatesCommand використовуються для взаємодії з AWS Certificate Manager (ACM)
const { ACMClient, ListCertificatesCommand } = require("@aws-sdk/client-acm");
// Імпорт допоміжних функцій, ймовірно, для валідації вхідних даних
const { validate } = require('./helpers');
// Отримання ARN (Amazon Resource Name) слухача ALB зі змінних середовища
// Ця змінна має бути налаштована в serverless.yml або іншому конфігураційному файлі.
const { albListenerArn } = process.env;
// Основна функція-обробник AWS Lambda
// Вона асинхронна, оскільки виконує мережеві запити до сервісів AWS
module.exports = async event => {
  // Розбір JSON-рядка з тіла запиту HTTP в об'єкт JavaScript
  const body = JSON.parse(event.body);
  // Ініціалізація клієнтів AWS SDK для ACM та ELBv2
  const acm = new ACMClient({});
  const elbV2 = new ElasticLoadBalancingV2Client();
  // Валідація вхідних даних (тіла запиту)
  try {
    validate(body); // Виклик допоміжної функції для валідації
  } catch (err) {
    // Обробка помилок валідації
    if (err instanceof Error) {
      // Якщо це стандартна помилка JavaScript, логуємо її
      console.error(err);
      // Повертаємо відповідь з кодом 500 (Internal Server Error)
      return {
        statusCode: 500,
          body: JSON.stringify({ message: 'An error occured when attempting to validate request' // Загальне повідомлення про помилку валідації
        })
      }
    } else {
// Якщо помилка не є об'єктом Error (наприклад, масив повідомлень про помилки від валідатора). Повертаємо відповідь з кодом 400 (Bad Request)
      return {
        statusCode: 400,
        body: JSON.stringify({ messages: err }) // Повідомлення про помилки валідації
```

```
        }
      }
    }
    // Пошук сертифіката ACM за доменним ім'ям
    let certificate;
    try {
      // Отримання списку всіх сертифікатів ACM в поточному регіоні (до 1000 елементів)
            certificates = ( await acm.send( new ListCertificatesCommand({MaxItems: 1000}) ) ).CertificateSummaryList;
    } catch (err) {
      // Обробка помилки при отриманні списку сертифікатів
      console.error(err);
      return {
        statusCode: 500,
     body: JSON.stringify({ message: 'An error occured when searching for certificates' })
      };
    }
    console.log('certifictaes count:', certificates.length);
    try {
// Фільтрація знайдених сертифікатів для пошуку того, що відповідає 'body.domain'
      certificate = certificates.filter(cert => cert.DomainName === body.domain)[0];
    } catch (err) {
      console.error(err);
      return {
        statusCode: 500,
      body: JSON.stringify({ message: 'An error occured when searching for certificates' }) // Повторне повідомлення, що може бути уточнено
      };
    }
    // Перевірка, чи знайдено сертифікат та чи має він ARN
    if (certificate && certificate.CertificateArn) {
      try {
            // Надсилання команди до ELBv2 для додавання сертифіката до слухача балансувальника навантаження
        await elbV2.send( new AddListenerCertificatesCommand({
          Certificates: [ // Масив сертифікатів для додавання
         {CertificateArn: certificate.CertificateArn // ARN знайденого сертифіката
            }
          ],
```

```
                ListenerArn: albListenerArn // ARN слухача ALB, отриманий зі змінних середовища
            }) );
        } catch (err) {
            // Обробка помилки, якщо не вдалося приєднати сертифікат
            console.error(err);
            return {
                statusCode: 500,
                body: JSON.stringify({ message: 'An error occured when attaching the certificate to the load balancer' })
            }
        }
    } else {
        // Якщо сертифікат не знайдено
        return {
            statusCode: 404,
                    body: JSON.stringify({ message: `Unable to find certificate with domain: ${body.domain}` })
        }
    }
    return {
        statusCode: 200
    }
};
```

3.5 Опис AWS API Gateway

AWS API Gateway складається з наступних частин.

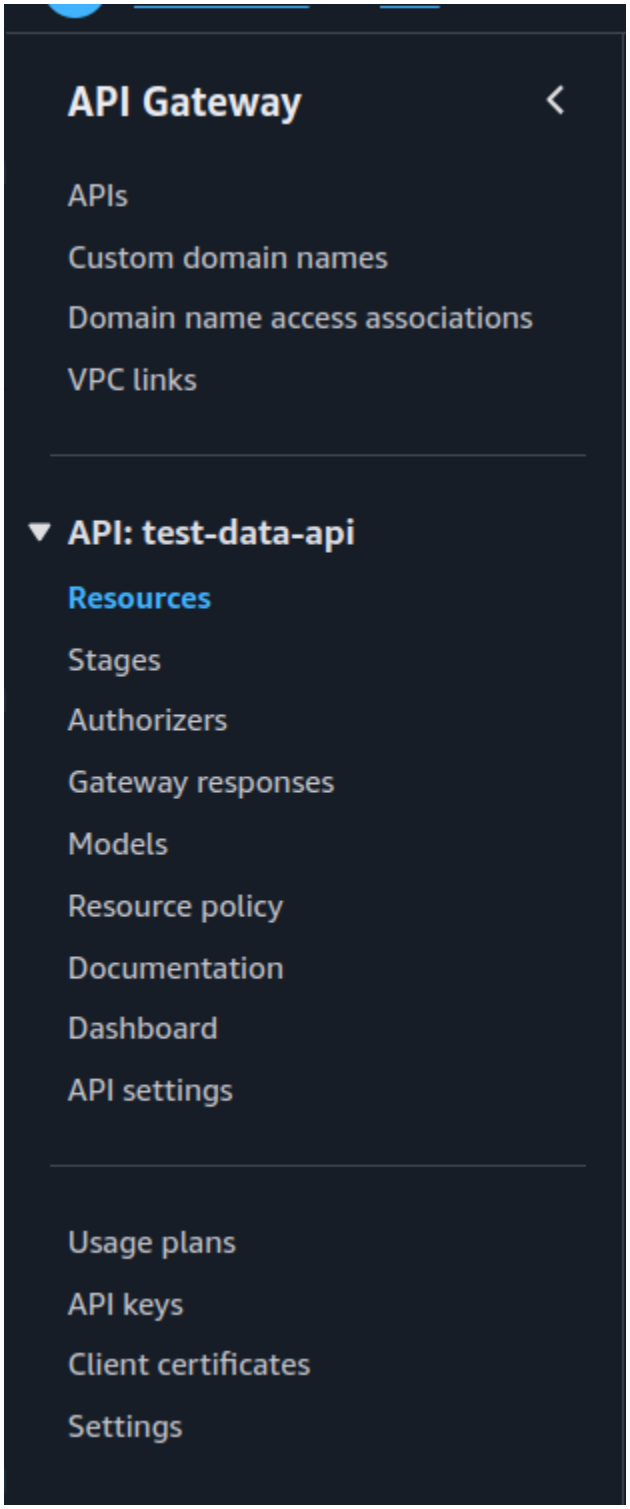

Рис. 10 AWS API Gateway

Resources (Ресурси) - найважливіший розділ для REST API. Тут ви визначаєте шляхи (наприклад, /users, /products), HTTP-методи (GET, POST, PUT, DELETE) та їх інтеграції (наприклад, з Lambda функціями, HTTP-ендпоінтами). Це фактична структура вашого API.

Stages (Стадії) - дозволяє створювати та керувати різними версіями вашого API (наприклад, dev, staging, prod). Кожна стадія може мати власні налаштування, розгортання та URL-адреси.

Authorizers (Авторизатори)- налаштовання механізмів авторизації для API, такі як Lambda Authorizers (Custom Authorizers) або IAM ролі/політики.

Gateway responses (Відповіді шлюзу) - дозволяє налаштовувати відповіді API Gateway на помилки або інші події до того, як запит досягне бекенд-інтеграції (наприклад, 4xx або 5xx помилки).

Models (Моделі) - визначає схеми даних (наприклад, у форматі JSON Schema) для тіл запитів та відповідей. Це допомагає валідувати дані та генерувати SDK.

Resource policy (Політика ресурсів) - дозволяє контролювати доступ до вашого API на основі IP-адрес, VPC, джерел тощо.

Documentation (Документація): Дозволяє створювати та керувати вбудованою документацією для API.

Dashboard (Панель моніторингу) - надає метрики та графіки моніторингу для вашого API, включаючи кількість запитів, помилки, затримки.

API settings (Налаштування API) - загальні налаштування для поточного API, такі як увімкнення кешування, ведення журналів (логінгу) та метрик.

Загальні налаштування безпеки та використання:

Usage plans (Плани використання) - дозволяє створювати та керувати планами використання для API, включаючи обмеження (throttling) та квоти для API-ключів.

API keys (API-ключі): Тут ми генеруємо та керуємо API-ключами, які використовуються для автентифікації клієнтів до наших приватних API-ендпоінтів.

Client certificates (Клієнтські сертифікати) - використовуються для двосторонньої TLS-аутентифікації, де API Gateway перевіряє сертифікат клієнта.

Settings (Налаштування) - загальні налаштування API Gateway на рівні облікового запису.

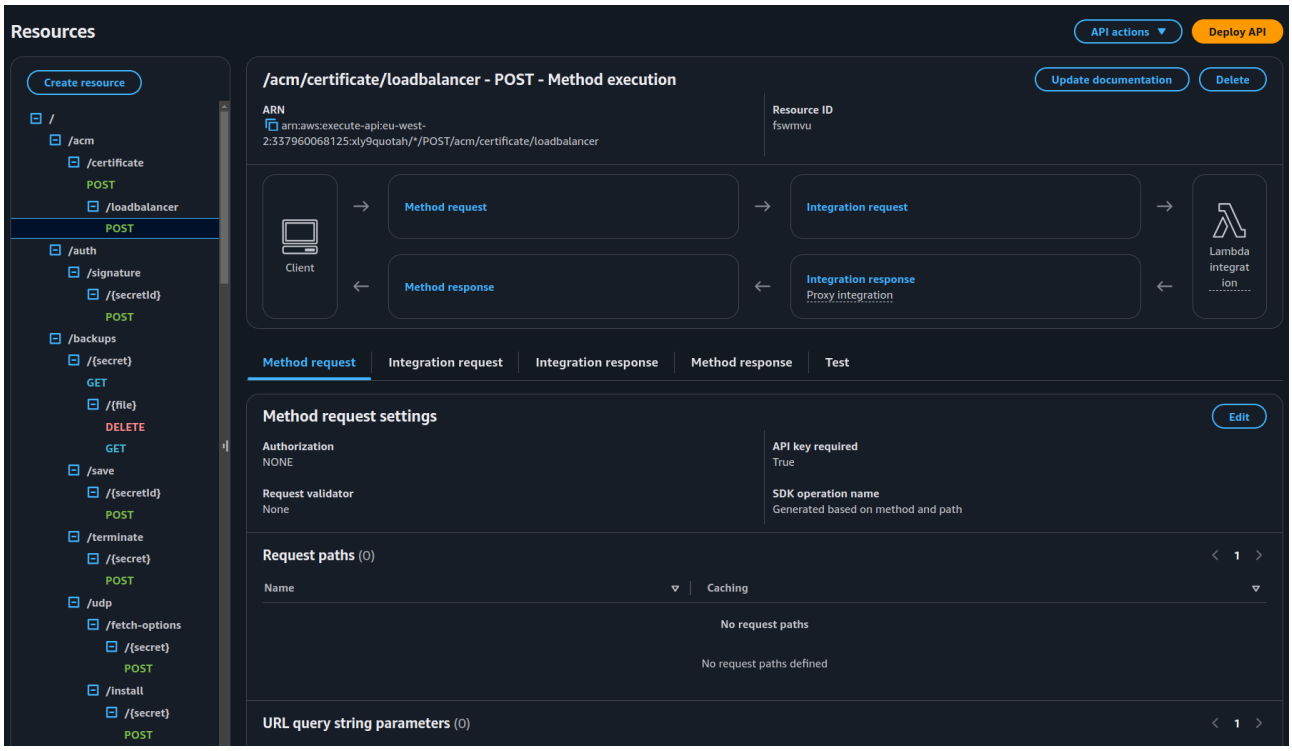

Рис. 11. Method Execution Flow

Блок "Потік виконання методу" (Method Execution Flow):

Method request (Запит методу) - представляє етап обробки запиту API Gateway. Тут відбувається валідація, авторизація, перетворення заголовків тощо.

Integration request (Запит інтеграції) - це етап, на якому API Gateway готує запит для бекенд-інтеграції. Тут можуть відбуватися перетворення тіла запиту, мапінг параметрів тощо.

Lambda integration (Інтеграція Lambda) - редставляє інтеграцію API Gateway з AWS Lambda функцією, яка є бекендом для цього ендпоінту.

Integration response (Відповідь інтеграції) - етап, на якому API Gateway отримує відповідь від бекенду (Lambda) і, можливо, виконує подальші перетворення перед поверненням її клієнту.

Proxy integration (Проксі-інтеграція) - API Gateway передає весь запит (заголовки, тіло, параметри) до Lambda функції "як є" і повертає всю відповідь від Lambda "як є" клієнту, без додаткових перетворень.

Method response (Відповідь методу) - представляє остаточну відповідь, сформовану API Gateway, яка надсилається клієнту.

ВИСНОВОК ДО РОЗДІЛУ 3

Частину проекту з управління вебсайтами реалізовано як мікросервіси на базі Serverless framework, javascript та AWS Lambda і AWS API Gateway. Це забезпечує масштабованість, відмовостійкість та економічну ефективність, оскільки ресурси споживаються лише під час фактичного виконання коду. Використання Node.js 20.x як середовища виконання вказує на сучасний стек технологій. Основна структура мікросервісів описана в файлі serverless.yml та детально розглянута в цьому розділі. Розроблені Lambda функції виконують широкий спектр функцій, що охоплюють кілька доменних областей:

Управління WordPress: Отримання інформації про ядро, плагіни та теми, перевірка облікових даних та керування оновленнями.

Резервне Копіювання: Збереження, завантаження, видалення та керування параметрами резервних копій.

Система Тікетів: Створення, оновлення, відповіді та перегляд тікетів підтримки, включаючи обробку вхідних електронних листів.

Платіжна Інтеграція (Stripe): Керування платежами, цінами та підписками.

Управління Секретами та Автентифікація: Створення, оновлення та перевірка "секретів" та генерація підписів.

Інфраструктурні Операції: Автоматизація керування SSL-сертифікатами через AWS Certificate Manager (ACM) та їх приєднання до Application Load Balancer (ALB).

WebSockets: Підтримка двостороннього зв'язку.

Проект активно використовує різні сервіси AWS:

DynamoDB: Як основна база даних для зберігання різноманітних даних (тікети, секрети, підписи, Stripe-транзакції). Широке використання глобальних вторинних індексів (GSI) та режиму PAY_PER_REQUEST вказує на оптимізацію для гнучких запитів та змінних робочих навантажень. Наявність TimeToLiveSpecification для SignaturesTable свідчить про ефективне керування життєвим циклом даних.

S3: Для зберігання файлів, таких як вкладення тікетів, звіти, резервні копії та вхідні електронні листи.

SQS: Для асинхронної обробки подій, зокрема підписок Stripe, з використанням Dead-Letter Queue (DLQ) для забезпечення надійності.

ACM та ELBv2: Для керування SSL/TLS сертифікатами та їх інтеграції з балансувальниками навантаження, що є критично важливим для HTTPS-трафіку.

Secrets Manager: Для безпечного зберігання та доступу до конфіденційних облікових даних.

EventBridge: Для планування завдань або реагування на події.

EC2 (через VPC): Функції Lambda розгорнуті у VPC, що забезпечує мережеву ізоляцію та можливість доступу до приватних ресурсів (наприклад, реляційної БД, якщо така використовується). Дозволи ec2:DescribeInstances та керування мережевими інтерфейсами підтверджують це.

ВИСНОВКИ РОБОТИ

В даній роботі розглянуто основні переваги та недоліки монолітної та мікросервісної архітектури в веб проектах, та застосовано комбінований підхід. Монолітна частина написана на Laravel та мікросервіси на Javascript на основі Serverless framework, AWS Lambda, AWS API GateWay. До основних переваг мікросервісного підходу можна віднести:

Відмовостійкість та Ізоляція Помилок (Resilience & Fault Isolation). Якщо один мікросервіс (наприклад, функція, що відповідає за Stripe-інтеграцію) виходить з ладу, це не обов'язково впливає на роботу інших мікросервісів (наприклад, функцій WordPress або тікетів). Помилки локалізуються, запобігаючи каскадним збоям усієї системи.

Масштабованість (Scalability). Індивідуальне масштабування: Кожна функція Lambda (мікросервіс) може масштабуватися незалежно від інших. Наприклад, функції обробки тікетів можуть відчувати пікові навантаження, тоді як функції резервного копіювання можуть використовуватися рідше. Завдяки мікросервісам AWS Lambda автоматично масштабує лише ті функції, які дійсно цього потребують, не впливаючи на інші частини системи.

Еластичність: AWS Lambda надає еластичність "з коробки", що дозволяє проекту ефективно реагувати на коливання попиту без ручного втручання або надлишкового резервування ресурсів.

Швидкість Розробки та Розгортання (Faster Development & Deployment). Автономні команди: Менші, сфокусовані команди можуть працювати над окремими мікросервісами незалежно, використовуючи власні технології та процеси, що прискорює розробку.

Незалежні розгортання: Зміни в одному мікросервісі (Lambda-функції) можуть бути розгорнуті без необхідності перерозгортати або зупиняти всю програму. Це зменшує ризик збоїв при розгортанні та дозволяє частіше випускати оновлення. Наприклад, оновлення функції createTicket не потребуватиме перерозгортання getPlugins.

Гнучкість Технологій (Technology Flexibility). Хоча цей проект переважно використовує Node.js 20.x, мікросервісна архітектура дозволяє використовувати різні технології (мови програмування, фреймворки, бази даних) для різних

мікросервісів. Це дає можливість обирати найкращий інструмент для конкретного завдання.

Незважаючи на значні переваги, мікросервісна архітектура також має певні недоліки та виклики.

Зростання складності. Мікросервіси – це розподілені системи. Замість одного моноліту, який працює в одному процесі, ви маєте безліч окремих сервісів, що взаємодіють між собою через мережу. Це ускладнює розробку, тестування, моніторинг та розгортання.

Міжсервісна комунікація: Виникає потреба в ефективних механізмах міжсервісної комунікації (API Gateway для HTTP, SQS для асинхронних повідомлень, WebSockets). Управління цими взаємодіями, їх версіонування та забезпечення сумісності додає складності.

Вартість. Ціна на AWS Lambda: Хоча Lambda позиціонується як економічно ефективна, вартість може зрости для дуже інтенсивних робочих навантажень. Модель "плати за використання" означає, що ви платите за кількість запитів та тривалість виконання. Для функцій з високою частотою викликів або довгим часом виконання (наприклад, checkCredentials з таймаутом 900 секунд), загальні витрати можуть бути значними. Навантаження на мережу (VPC): Розміщення функцій Lambda у VPC тягне за собою додаткові витрати, пов'язані з мережевими інтерфейсами (ENIs), які AWS створює та керує для кожної функції.

"Холодний старт" (Cold Starts). Затримка першого виконання Lambda-функції після періоду неактивності. Це відбувається тому, що AWS потрібно завантажити код функції, ініціалізувати середовище виконання (Node.js) та встановити мережеві з'єднання. Холодні старти приводять до помітних затримок для кінцевого користувача, особливо для рідко використовуваних функцій. Хоча Node.js зазвичай має менші холодні старти порівняно з іншими рантаймами (наприклад, Java), для чутливих до затримок операцій це все одно може бути проблемою. Проєкт має багато окремих функцій, що збільшує ймовірність холодних стартів для кожної з них.

У підсумку, хоча мікросервісна архітектура на AWS Lambda надає чудові можливості для масштабованості та гнучкості, вона вимагає значних інвестицій у складність, моніторинг, відлагодження та управління витратами.

СПИСОК ДЖЕРЕЛ

1. Newman, S. “Building Microservices,” O’Reilly Media, 2015.
2. Fowler, M. “Monolith First,” martinfowler.com/articles/monolith-first.html
3. Richardson, C. “Microservices Patterns,” Manning Publications, 2018.
4. Dragoni, N. et al. “Microservices: Yesterday, Today, and Tomorrow,” Springer, 2017.
5. Pautasso, C. et al. “Microservices in Practice,” IEEE Software, 2017.
6. Sam Newman. Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. O’Reilly, 2019.
7. Mark Richards. Software Architecture Patterns. O’Reilly, 2015.
8. Vaughn Vernon. Implementing Domain-Driven Design. Addison-Wesley, 2013.
9. Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, 2004.
10. Martin Fowler. Patterns of Enterprise Application Architecture. Addison-Wesley, 2002.
11. Neal Ford et al. Fundamentals of Software Architecture. O’Reilly, 2020.
12. Martin Fowler — "Monolith First", https://martinfowler.com/bliki/MonolithFirst.html
13. ThoughtWorks — Why You Should Start With a Monolith, https://www.thoughtworks.com/insights/blog/why-you-should-start-monolith
14. RedHat Developers — Monolithic vs. Microservices Architecture, https://developers.redhat.com/articles/monolithic-vs-microservices
15. Microsoft Learn — Monolithic Architecture Style, https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/monolithic
16. Martin Fowler — Microservices, https://martinfowler.com/articles/microservices.html
17. Chris Richardson — Microservices.io, https://microservices.io
18. AWS Architecture Blog — Microservices on AWS, https://aws.amazon.com/microservices/

19. Microsoft Learn — Microservices Architecture, https://learn.microsoft.com/en-us/dotnet/architecture/microservices/