ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка системи розподіленої доставки цифрового контенту на прикладі гри Minecraft»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Михайло СТРАТОВИЧ
(підпис)

Виконав: здобувач вищої освіти групи ПД-41

Михайло СТРАТОВИЧ

Керівник: Юрій АБРОСКІН
асистент кафедри ІПЗ

Рецензент: _____

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
« ____ » _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Стратовичу Михайлу Ярославовичу

1. Тема кваліфікаційної роботи: «Розробка системи розподіленої доставки цифрового контенту на прикладі гри Minecraft»
керівник кваліфікаційної роботи асистент кафедри ІПЗ Юрій Аброскін,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «24» лютого 2025 р. № 56.

2. Строк подання кваліфікаційної роботи «02» червня 2025 р.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про системи доставки цифрового контенту, принципи роботи CDN та хмарних сервісів; аналітична інформація щодо архітектурних підходів на базі AWS; концепції побудови інфраструктури як коду (IaC); аналіз існуючих рішень доставки ігрового контенту та вимог до систем з підтримкою модифікацій.

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

1. Аналіз підходів до розподіленої доставки цифрового контенту в ігрових системах.
2. Проєктування архітектури та компонентів системи доставки контенту.

3. Реалізація системи доставки цифрового контенту та клієнтського застосунку.
4. Тестування та оцінювання ефективності функціонування системи доставки контенту.

5. Перелік графічного матеріалу: *презентація*
   1. Аналіз аналогів.
   2. Вимоги до програмного забезпечення.
   3. Програмні засоби реалізації.
   4. Діаграма варіантів використання.
   5. Алгоритм роботи застосунку.
   6. Екранні форми.
   7. Апробація результатів дослідження

6. Дата видачі завдання «25» лютого 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	25.02–04.03.2025	
2	Аналіз та дослідження існуючих аналогів	05.03–13.03.2025	
3	Аналіз підходів до розподіленої доставки цифрового контенту в ігрових системах	14.03–20.03.2025	
4	Проєктування архітектури та компонентів системи доставки контенту	21.03–05.04.2025	
5	Реалізація системи доставки цифрового контенту та клієнтського застосунку	06.04–25.04.2025	
6	Тестування та оцінювання ефективності функціонування системи доставки контенту	26.04–28.04.2025	
7	Оформлення роботи: вступ, висновки, реферат	29.04–11.05.2025	
8	Розробка демонстраційних матеріалів	12.05–18.05.2025	
9	Попередній захист роботи	19.05–30.05.2025	

Здобувач вищої освіти _____ Михайло СТРАТОВИЧ
(підпис)

Керівник
кваліфікаційної роботи _____ Юрій АБРОСКІН
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 51 стор., 2 табл., 24 рис., 22 джерел.

Мета роботи – спрощення, масштабованість і підвищення гнучкості системи доставки цифрового контенту шляхом використання принципів хмарних технологій і мереж CDN.

Об'єкт дослідження – процес доставки цифрового контенту користувачам у розподілених системах на прикладі гри Minecraft.

Предмет дослідження – система автоматизованої доставки цифрового контенту.

Короткий зміст роботи: У кваліфікаційній роботі проаналізовано принципи побудови систем розподіленої доставки цифрового контенту в ігрових середовищах, зокрема на прикладі ігор з відкритою архітектурою, таких як Minecraft. Розглянуто технологічні підходи, засновані на використанні CDN, хмарної інфраструктури AWS та концепції інфраструктури як коду (IaC). Проведено аналіз існуючих рішень доставки ігрового контенту (Steam, Epic Games Launcher, CurseForge) та обґрунтовано переваги розробленої системи DUIKT. Реалізовано клієнтську частину на базі кросплатформенного застосунку та серверну логіку у без серверній архітектурі. Система підтримує автоматизовану доставку модифікацій, перевірку цілісності файлів та масштабованість за рахунок використання глобальної CDN. Проведено тестування основних функціональних можливостей та описано структуру інфраструктури, зібраної засобами IaC.

Сферою використання застосунку є розповсюдження ігрового контенту в іграх з модифікаціями, наприклад Minecraft.

КЛЮЧОВІ СЛОВА: РОЗПОДІЛЕНА ДОСТАВКА, ЦИФРОВИЙ КОНТЕНТ, CDN, MODPACK, MINECRAFT, AWS, ІНФРАСТРУКТУРА ЯК КОД, ІГРОВИЙ ЛАУНЧЕР

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

AWS – Amazon Web Services

S3 – Simple Storage Service (сервіс об’єктного зберігання AWS)

CDN – Content Delivery Network

CI/CD – Continuous Integration / Continuous Deployment

API – Application Programming Interface

IAM – Identity and Access Management

OAI – Origin Access Identity

DNS – Domain Name System

ACM – AWS Certificate Manager

UI – User Interface

JSON – JavaScript Object Notation

RSA – Rivest–Shamir–Adleman (алгоритм криптографічного підпису)

SHA256 – Secure Hash Algorithm 256-bit

ZIP – Архівований формат файлів

Lambda – AWS Lambda (безсерверні функції)

CD – Content Delivery

IDE – Integrated Development Environment

CDN – Content Delivery Network

TypeScript – Мова програмування з підтримкою типізації

Electron.js – Фреймворк для створення кросплатформних десктопних застосунків

Terraform – Інструмент для інфраструктури як коду (IaC)

Route 53 – DNS-сервіс від AWS

CloudFront – CDN-сервіс від AWS

GitHub Actions – Сервіс CI/CD для автоматизації робочих процесів

ВСТУП

Актуальність: У сучасній ігровій індустрії цифровий контент відіграє ключову роль у забезпеченні повноцінного ігрового досвіду. Особливо це стосується ігор з відкритою архітектурою, таких як Minecraft, де гравці активно використовують модифікації, додаткові ресурси та модпаки. Зростаюча популярність таких ігор призводить до постійного оновлення та поширення великої кількості даних, які мають бути доставлені до користувача швидко, безпечно та без зайвих технічних складнощів. Однак у більшості подібних ігор відсутні вбудовані механізми централізованого оновлення, що змушує користувачів завантажувати файли з різних сторонніх джерел. Це не лише ускладнює встановлення та підтримку модифікацій, але й створює ризики безпеки, конфлікти між версіями модів та негативно впливає на загальне враження від гри, що створює високий ризик втрати гравців [1, с. 1]. З розвитком хмарних технологій і концепції розподіленої доставки (CDN, AWS, IaC) з'являється можливість побудови надійних і масштабованих систем, які дозволяють забезпечити швидку, автоматизовану та безпечну доставку цифрового контенту до гравців незалежно від їхнього місцезнаходження, що особливо важливо в умовах високої конкуренції в індустрії та зростаючих очікувань користувачів [1, с. 1]. Таким чином, розробка подібних систем є актуальним і практично значущим завданням.

Об'єктом дослідження є процес доставки цифрового контенту користувачам у розподілених системах на прикладі гри Minecraft.

Предметом дослідження система автоматизованої доставки цифрового контенту.

Метою роботи є спрощення, масштабованість і підвищення гнучкості системи доставки цифрового контенту шляхом використання принципів хмарних технологій і мереж CDN.

Методи дослідження: на першому етапі було проведено аналіз особливостей доставки цифрового контенту в ігровій індустрії, зокрема для ігор

з відкритою архітектурою на прикладі гри Minecraft. Досліджено наявні системи доставки контенту та офіційні рекомендації щодо побудови розподілених систем доставки цифрового контенту від AWS для формування вимог до власного рішення, з урахуванням гнучкості, масштабованості, безпеки та автоматизації. Далі було визначено архітектурний підхід на основі використання хмарної інфраструктури AWS, CDN (CloudFront), безсерверних функцій (Lambda) та сховищ (S3, DynamoDB), а також опису інфраструктури у вигляді коду (Terraform). Для клієнтської частини було обрано Electron.js та TypeScript як оптимальні інструменти для створення кросплатформеного десктопного застосунку. Під час розробки дослідження проводились ітеративно: шляхом побудови прототипів, тестування взаємодії компонентів, перевірки доставки контенту та підпису файлів. CI/CD-процес реалізовано за допомогою GitHub Actions, що дало змогу дослідити ефективність автоматизованого оновлення ресурсів. Усі технічні рішення були перевірені під час реалізації повнофункціональної системи та її подальшого тестування в умовах, наближених до реального використання.

Наукова новизна роботи полягає у створенні незалежної системи доставки цифрового контенту, яка не прив’язана до закритих ігрових платформ і надає повний контроль над процесом публікації, оновлення та розгортання ресурсів. Розроблене рішення реалізовано у вигляді окремого клієнтського застосунку з перевіркою автентичності файлів через цифровий підпис, що підвищує безпеку та довіру до вмісту. Система адаптована до потреб приватних ігрових серверів та незалежних розробників, які використовують сучасні підходи до автоматизації та інфраструктури як коду. Таким чином, робота пропонує нову архітектурну модель доставки контенту, що поєднує сучасні хмарні технології з відкритою модульністю для ігор, орієнтованих на активну спільноту.

Практична значущість результатів полягає в можливості використання розробленої системи для автоматизованої доставки ігрового контенту на приватні сервери та в спільнотах, що створюють та розповсюджують модифікації.

1 АНАЛІЗ ПІДХОДІВ ДО РОЗПОДІЛЕНОЇ ДОСТАВКИ ЦИФРОВОГО КОНТЕНТУ В ІГРОВИХ СИСТЕМАХ

1.1 Специфіка доставки цифрового контенту в ігровій індустрії

Цифровим контентом в ігровій індустрії називають різноманітні медіа та інформаційні ресурси (текстури, 3D-моделі, звуки, відео, сценарії, доповнення DLC, модифікації тощо), які необхідно доставити у клієнтський додаток гри для забезпечення геймплею. Більшість сучасних ігор вимагає передачі таких даних гравцю: ігри (консолі, ПК, мобільні) вимагають передачі різних ресурсів клієнту для забезпечення геймплею. Обсяги цих ресурсів можуть бути дуже великими — від сотень мегабайт для мобільних ігор до десятків гігабайт для ПК чи консолей. Тому для доставки контенту зазвичай застосовують розподілені мережі доставки контенту (CDN), які кешують ресурси на вузлах біля гравців, щоб мінімізувати затримки та прискорити завантаження [1, с. 1].

Специфіка доставки цифрового контенту полягає у вирішенні наступних проблем:

– Затримки (латентність): повільні завантаження або оновлення контенту помітно погіршують ігровий досвід; зокрема, для завантажень контенту метою є забезпечення достатньо гарного рівня продуктивності: користувач не розсердиться, якщо завантаження триває 10 хвилин замість 8, але буде дуже роздратований, якщо воно займає 6 годин [3]. Тому оптимізація доставки балансує між швидкістю та вартістю, часто використовуючи CDN та інші розподілені сховища.

– Безпека: сторонні модифікації можуть містити шкідливий або несумісний код. Часто трапляються випадки підміни файлів, коли шкідливий файл маскується під легітимний мод, але насправді містить троян, майнер або інші загрози безпеці. Для обмеження ризиків деякі платформи впроваджують багаторівневу модерацію: автоматичне сканування, перевірку модераторами і

звітність спільноти для фільтрації небезпечного контенту [4].

– Фрагментація версій та несумісність модів: Оскільки користувацький контент постійно оновлюється, не всі моди сумісні з поточною версією гри або між собою. Це створює додаткове навантаження на гравця, який повинен стежити за сумісністю, конфліктами залежностей та актуальністю пакунків модів [1, с. 4].

Особливо актуальними ці проблеми є для ігор з так званою відкритою архітектурою — тобто таких, що дозволяють стороннє розширення, інтеграцію власного контенту, зміну ігрових механік та елементів інтерфейсу. На відміну від закритих систем, відкриті ігри не мають жорсткої ізоляції від стороннього втручання і дозволяють активне втягнення спільноти в розвиток проєкту. Саме така архітектура стимулює створення так званого UGC (User-Generated Content) — модів, ресурс-паків, датапаків, скінів, лаунчерів, що не розробляються офіційним розробником гри, але часто становлять значну частину геймплейного досвіду [2, с. 1].

Кастомізація — це можливість змінювати вміст гри відповідно до вподобань гравця. Вона реалізується через моди (від modification) — сторонні доповнення, які змінюють або розширюють функціональність гри, додаючи нові об’єкти, механіки, сценарії. Модпаки (від modpacks) — це зазвичай готові збірки з десятків або сотень модів, перевірених на сумісність, які встановлюються одночасно. Наявність кастомного контенту часто є основною причиною повернення гравців у гру, навіть через роки після офіційного релізу.

Для завантаження та управління контентом користувачі найчастіше використовують ігрові лаунчери — окремі програми, що дозволяють запускати гру з попередньо завантаженими модами, обирати версії, синхронізувати ресурси тощо. Лаунчер виконує функцію шлюзу між ігровим клієнтом та системою доставки контенту, й від його зручності безпосередньо залежить користувацький досвід.

У рамках даної роботи як приклад гри з відкритою архітектурою обрано Minecraft — одну з найпопулярніших ігор у світі, яка підтримує модифікації на

рівні ядра та має одну з найрозвинутіших мод-екосистем. Minecraft став середовищем для десятків тисяч модів і сотень модпаків, що створюються спільнотою гравців. Більшість сценаріїв використання модів у Minecraft реалізуються в одно користувацькому режимі, де гравець самостійно керує усіма файлами гри. У таких умовах проблема доставки контенту стає критично важливою: користувач очікує, що встановлення нового пакету модів буде автоматичним, безпечним і швидким. Саме це обумовлює актуальність створення окремої системи доставки цифрового контенту для подібних ігор.

1.2 Цільова аудиторія

Системи розподіленої доставки цифрового контенту мають широке застосування в ігровій індустрії, оскільки дозволяють забезпечити масштабовану, швидку та безпечну передачу ігрових ресурсів до клієнтів. Їх цільовою аудиторією є як кінцеві користувачі (гравці), так і професійна спільнота, що включає розробників ігор, авторів модифікацій, видавців, адміністраторів серверів та платформ. Гравці очікують швидкого завантаження контенту без затримок і складної ручної установки, а розробники — стабільної інфраструктури, яка дозволяє централізовано оновлювати вміст, перевіряти його цілісність і спрощувати поширення.

У межах даної кваліфікаційної роботи розглядається система доставки, орієнтована передусім на гравців одиночного режиму гри Minecraft. Автоматизація процесу встановлення модів і модпаків дозволяє уникнути типових помилок, пов’язаних із ручним копіюванням файлів, конфліктами версій або встановленням неавторизованих доповнень.

Крім гравців, цільову аудиторію також становлять розробники модифікацій, які прагнуть швидко опублікувати оновлення своїх проєктів і мати змогу використовувати сучасні інструменти автоматизації, зокрема CI/CD та GitHub. Для них важлива можливість централізованого, контрольованого поширення цифрового контенту без залучення сторонніх рішень або ручного

розгортання.

Також систему можуть використовувати адміністратори приватних серверів Minecraft, які готують клієнтські збірки для свого серверного середовища. Завдяки централізованій доставці модів та автоматизованому оновленню клієнтів вони можуть забезпечити синхронізацію між клієнтською та серверною конфігурацією, що знижує кількість технічних проблем при підключенні гравців до серверів.

1.3 Аналіз існуючих рішень та програмних платформ

1.3.1 Steam Client

Steam Client - це онлайн-сервіс компанії Valve для цифрової дистрибуції відеоігор, оновлень та супутніх сервісів. Клієнт Steam встановлюється на пристрій користувача і підключається до серверів Valve; через нього користувачі купують, завантажують і встановлюють ігри, отримують патчі та зберігають прогрес у хмарі. Способи доставки контенту: Контент доставляється через мережу контент-серверів Steam по всьому світу – після покупки гра автоматично завантажується клієнтом [9]. Steam використовує власну систему Digital Rights Management (DRM) і автооновлення: після випуску оновлення клієнт фоново завантажує його і оновлює файли гри. Окрім офіційних ігор, Steam підтримує і розповсюдження модифікацій через вбудовану Steam Workshop – центральний хаб користувацького контенту, де гравці можуть публікувати і завантажувати моди безпосередньо в ігри. Для кожної гри реалізація Workshop може відрізнятися: наприклад, в одних (Team Fortress 2) це користувацькі предмети, а в інших (Skyrim) – повноцінні моди, які встановлюються шляхом підписки гравця на них [6].

Рис. 1.1 Сервіс Steam Workshop

Переваги:

– Steam пропонує єдину екосистему для покупки ігор, їх автоматичного оновлення та збереження, а також спілкування у спільноті гравців – все в одному клієнті.

– Для багатьох ігор Steam інтегрував Steam Workshop, що дозволяє гравцям ділитися модами та встановлювати їх одним кліком [6]. Це значно спрощує поширення користувацького контенту у порівнянні з ручною інсталяцією модів – платформа сама завантажує потрібні файли та оновлює модифікації разом з грою.

– Інфраструктура Valve побудована таким чином, щоб стабільно обслуговувати мільйони одночасно активних користувачів. Steam демонструє високу ефективність у поширенні великих ігор і оновлень – навіть у випадках, коли розмір гри перевищує десятки гігабайт. Завдяки використанню розподіленої мережі доставки контенту (CDN), користувачі з різних регіонів отримують швидкий доступ до ігор. Додатково, автоматизована система оновлень та інтеграція з хмарними сервісами (такими як Steam Cloud) дозволяє зберігати прогрес і налаштування користувача, що підвищує зручність і комфорт від користування платформою.

Недоліки:

– Моди з Steam Workshop доступні лише власникам гри у Steam – якщо гру придбано поза Steam, до цього контенту доступ обмежено. Valve фактично контролює розповсюдження модів: наприклад, сервіс перевіряє наявність ліцензії гри перед дозволом завантажити мод, що унеможливлює просте поширення модифікацій поза платформою. Це створює бар'єри для відкритих спільнот: гравці версій гри з інших платформ залишаються без модів із Workshop.

– Не всі популярні «відкриті» до моддингу ігри присутні в Steam. Наприклад, Minecraft офіційно розповсюджується окремо і не підтримується в Steam, тому всю екосистему модів Minecraft спільноті доводиться розвивати поза Steam.

– Розробники модифікацій вимушені підлаштовуватись під правила Steam. Існують ліміти на розмір файлів модів і кількість одночасних підписок, які можуть стримувати масштабні проєкти. Вміст модів проходить модерацію на відповідність політикам контенту Valve. Таким чином, створювачі модів мало контролюють поширення свого контенту на платформі.

1.3.2 Epic Games Launcher

Epic Games Launcher (EGS) – це цифровий магазин та лаунчер від Epic Games. Архітектурно він схожий на Steam: окрема програма-лаунчер, через яку користувач авторизується у своєму обліковому записі Epic і завантажує придбані ігри з серверів Epic. Доставка ігор здійснюється шляхом прямого завантаження з Epic Games Store: після покупки гра додається до бібліотеки і доступна для завантаження через лаунчер. Epic Games Launcher використовує хмарну інфраструктуру Epic для масштабного розповсюдження контенту [7]. EGS зовсім не підтримує кастомізацію та розповсюдження модифікацій для ігор.

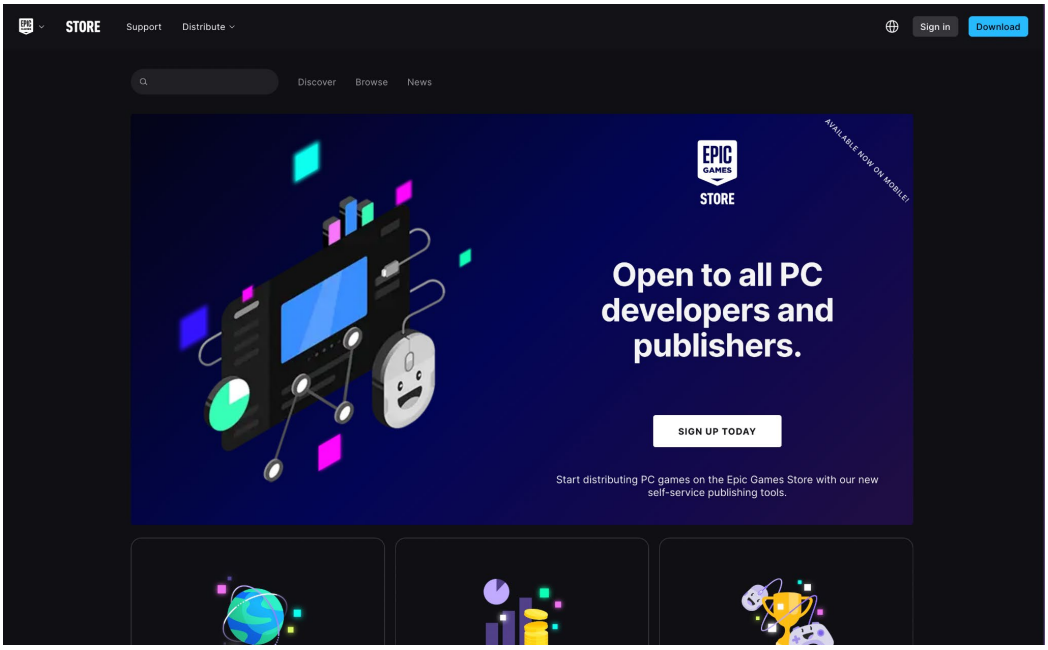


Рис. 1.2 Платформа Epic Games Launcher

Переваги:

– Інтерфейс лаунчера доволі легкий і швидкий, без надлишку сторонніх функцій. Відсутність зайвих сервісів спрощує користування – вся увага на старті була зосереджена на встановленні та запуску ігор.

– Epic Games Store бере лише 12% від прибутку з продажу ігор, до того ж скасовує роялті за використання Unreal Engine для ігор, випущених в EGS [7]. Це приваблює видавців та стимулює вихід багатьох проєктів ексклюзивно на цій платформі.

– Завдяки досвіду з Fortnite, Epic змогла побудувати інфраструктуру, здатну ефективно обробляти великі обсяги трафіку. BuildPatchTool дозволяє створювати пакети оновлень і заливати їх на сервер, а Sandbox-механізм дає змогу одночасно підтримувати кілька версій гри. Для Unreal Engine є плагіни, скрипти й API, які полегшують інтеграцію з CI/CD-процесами [7].

Недоліки:

– Система є закритою та недостатньо гнучкою: розробники змушені використовувати лише офіційні інструменти без можливості підключення сторонніх CDN або гнучкого налаштування структури оновлень.

– Відсутня підтримка користувацьких модифікацій. Epic Games не

пропонує власного аналога Steam Workshop, тому модифікації доводиться встановлювати вручну, якщо гра це дозволяє.

– Інтеграція з CI/CD-процесами обмежена через відсутність універсального командного інструменту, подібного до SteamCMD: розгортання оновлень переважно орієнтоване на Unreal Engine та внутрішні сервіси Epic Games.

1.3.3 CurseForge

CurseForge – це спеціалізована платформа для розповсюдження користувацького ігрового контенту: модифікацій, плагінів, аддонів тощо. На відміну від Steam і Epic, CurseForge не продає повні ігри, а зосереджена саме на модах. Архітектурно платформа складається з веб-сайту з базою даних проєктів та окремого клієнта для зручного керування модами на комп'ютері. Клієнт дозволяє автоматично завантажувати, оновлювати та застосовувати моди до встановлених ігор, відстежуючи сумісність версій. Основним механізмом є власна CDN-мережа CurseForge, що автоматично розповсюджує модифікації, а для розробників доступний спеціалізований інструментарій — Cloud Cooking, REST API, white-label UI та CI/CD-інтеграція [4].

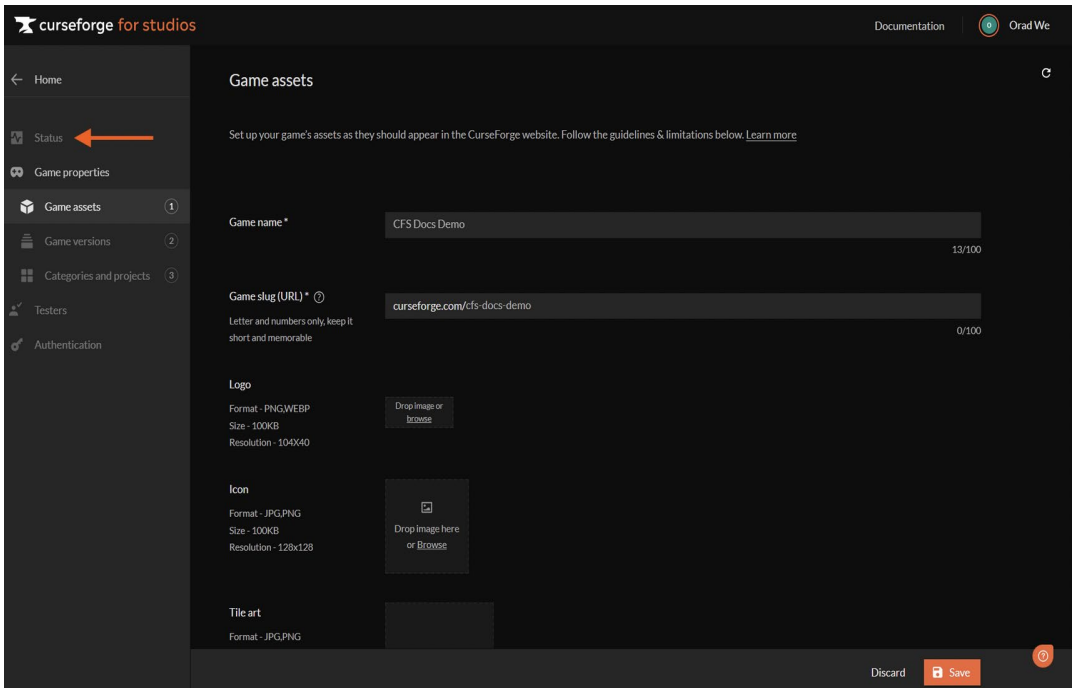


Рис. 1.3 Платформа CurseForge

Переваги:

– CurseForge створена з урахуванням потреб моддингових спільнот, тому надає гнучкі можливості для поширення кастомного контенту. Величезна кількість модифікацій доступна в межах одного ресурсу, що спрощує користувачам пошук потрібних доповнень [4].

– Платформа CurseForge не прив’язана до конкретного дистриб’ютора ігор – моди з неї можуть використовувати всі гравці незалежно від того, де куплена гра. Це особливо важливо для ігор, що не мають власних офіційних «майстерень».

– Фірмовий клієнт CurseForge пропонує зручний інтерфейс для встановлення модів: користувач може буквально в кілька кліків додати мод до гри.

Недоліки:

– Оскільки CurseForge не є повноцінним ігровим лаунчером, користувачам все одно потрібні інші платформи для самої гри. Це означає, що екосистема розділена: гра запускається через свій власний лаунчер, а моди керуються окремо через CurseForge.

– CurseForge не є універсальним рішенням для всіх ігор. Його використання можливе лише для тих проєктів, які офіційно підтримують модифікації і де розробники (або спільнота) готові докласти зусиль до інтеграції CurseForge у процес розповсюдження модів.

– CurseForge адмініструється компанією Overwolf, яка контролює серверну інфраструктуру, оновлення платформи, механізми доставки контенту та інші технічні аспекти. Це означає, що розробники гри не мають повного контролю над модифікаціями, які публікуються та поширюються через CurseForge [4].

В таблиці 1.1 зведено результати аналізу існуючих систем розподіленої доставки цифрового контенту для ігор та їх порівняння.

Таблиця 1.1

Результати аналізу систем для розподіленої доставки цифрового контенту для ігор та їх порівняння

Система	Основне призначення	Архітектура / Технології	Гнучкість і кастомізація	Метод доставки та кешування	Підтримка модів	Міжплатформність
Steam Client	Доставка ігор, оновлень та DLC через інтегровану CDN-платформу.	Моноліт з додатковими модулями, сервери Valve.	Мінімальна, закрита екосистема.	Власна CDN-мережа Steam з глобальними кеш-серверами (Steam Content Servers).	Часткова.	Так.
Epic Games Launcher	Доставка офіційного ігрового контенту (ігри, оновлення, івенти) через власну інфраструктуру.	Монолітна структура лаунчера з вбудованими службами Epic.	Низька, кастомізація обмежена.	Власна CDN-інфраструктура Epic (Epic CDN) із розподіленим кешуванням.	Відсутня.	Так.
CurseForge	Доставка модів, аддонів та користувацького контенту для відкритих ігор (Minecraft тощо).	Модульна інфраструктура з API, white-label UI, cloud cooking, CDN-платформа від Overwolf	Висока: публічний API, кастомізація, інтеграція в інші ігри.	Хмарна інфраструктура CurseForge із кешуванням на CDN, підтримкою модпаки і версій.	Повна.	Так.

1.4 Огляд IT-засобів, що використовуються у подібних рішеннях

Системи доставки цифрового контенту широко використовують CDN (Content Delivery Network) – розподілену мережу серверів, що розташовані по всьому світу. CDN кешує статичний контент ближче до кінцевих користувачів, зменшуючи затримки та навантаження на початкові сервери. Завдяки географічно розосередженим PoP (points of presence), CDN скорочує відстань, яку має пройти трафік, що забезпечує більш швидку доставку великих файлів (ігрових ресурсів, оновлень тощо) та покращує доступність сервісу [11].

Наприклад, на успішних ігрових платформах (Steam, Epic Games тощо) використовуються одночасно декілька CDN: Valve задіює Akamai разом з іншими CDN для розподілу трафіку [9]. Компанія Amazon для власних сервісів віддає перевагу CloudFront.

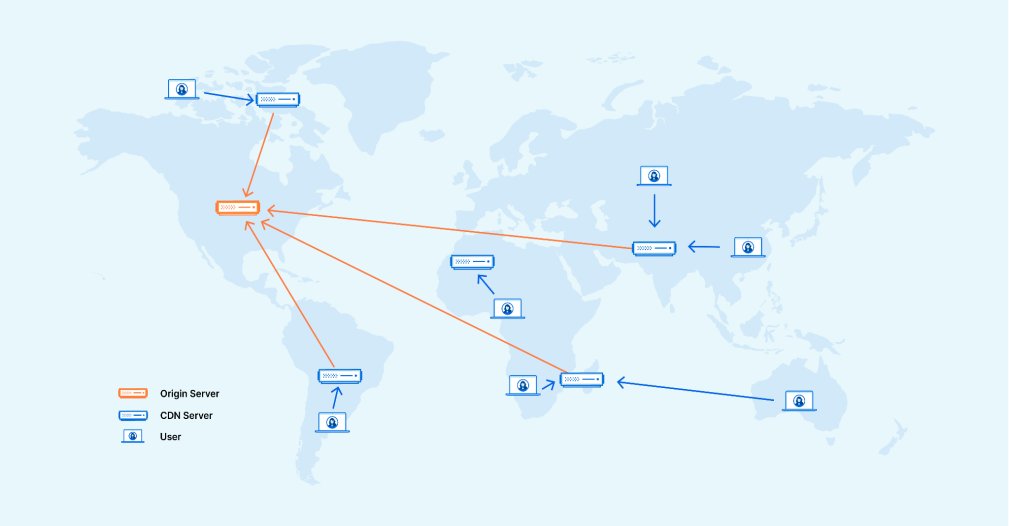


Рис. 1.4 Географічна схема розподіленої доставки цифрового контенту через CDN [11]

Для великих систем доставки контенту критичною є автоматизація управління інфраструктурою. Infrastructure as Code (IaC) дає змогу описувати всю хмарну інфраструктуру в конфігураційних скриптах (наприклад, Terraform або AWS CloudFormation). Це забезпечує відтворюваність, швидке

масштабування і уникнення ручних помилок: сервіси впроваджуються автоматично одним натисканням, конфігурації зберігаються у системі керування версіями і розгортаються однаково у різних середовищах. Іншими словами, IaC-підхід дозволяє динамічно додавати чи видаляти ресурси під час піків навантаження, полегшує аудит змін та підвищує безпеку інфраструктури.

Об’єктні сховища застосовуються для зберігання самих ігрових файлів і артефактів. До прикладу, Amazon S3 – масштабовне хмарне сховище об’єктів – використовується багатьма ігровими студіями для розміщення графіки, аудіо, моделей тощо. У разі застосунку «Mojo Melee» від Mystic Moose студія обрала S3 як масштабований сховище для всіх ігрових ресурсів, а доставку контенту прискорює за допомогою CloudFront [13]. Аналогічно Google Cloud Storage – кероване об’єктне сховище з можливістю зберігати будь-який обсяг даних, а Azure Blob Storage – масштабовне і безпечне сховище об’єктів для хмарних навантажень, – використовуються для тих самих цілей у мультиклауд-середовищах.

Для зберігання метаданих (параметрів файлів, налаштувань доставки, інформації про користувачів тощо) у системах доставки звичайно застосовують бази даних. Це можуть бути реляційні СУБД на кшталт PostgreSQL, а також NoSQL-розподілені сховища. Наприклад, Amazon DynamoDB (NoSQL key-value) підходить для геймінгових застосунків завдяки низькій латентності та автоматичному масштабуванню. Медійні та ігрові компанії використовують DynamoDB як індекс метаданих контенту або таблицю станів гри, що дозволяє витримувати навантаження високого рівня. У згаданому кейсі Mystic Moose DynamoDB використовують саме для менеджменту метаданих гри [13].

Сучасні системи доставки реалізують на різних технологіях. Наприклад, бекенд часто пишуть на TypeScript/Node.js завдяки однорідності стеку і великій екосистемі пакетів [17, с. 13 - 15]. Python використовують для компонентів API та обробки даних (напр. через Flask, FastAPI). За критичних вимог до продуктивності застосовують Go (відомо, що у Riot Games більшість інфраструктурних сервісів написані саме на Go) [15]. Rust популярний для

продуктивних і безпечних рішень на боці сервера або на краю мережі (наприклад, Cloudflare активно використовує Rust в своєму edge-комп'ютингу). Всі ці інструменти та мови дають гнучкість при розробці, забезпечують високу безпеку та масштабованість системи, а також дозволяють значною мірою автоматизувати її побудову і супровід (IaC, CI/CD-процеси тощо).

1.5 Формулювання об'єкта, предмета, мети та технічних завдань дослідження

У цьому дослідженні розглядається створення гнучкої та автоматизованої архітектури доставки на прикладі гри Minecraft, яка має широку спільноту та активно використовує систему модифікацій. У процесі наукового аналізу було визначено об'єкт, предмет, мету та основні технічні завдання дослідження, що обумовлюють його логіку та структуру.

Об'єктом дослідження є процес доставки цифрового контенту користувачам у розподілених системах на прикладі гри Minecraft.

Предметом дослідження система автоматизованої доставки цифрового контенту.

Метою роботи є спрощення, масштабованість і підвищення гнучкості системи доставки цифрового контенту шляхом використання принципів хмарних технологій і мереж CDN.

Основні технічні завдання:

– спроєктувати масштабовану хмарну інфраструктуру доставки;

– реалізувати механізми CI/CD для завантаження та оновлення контенту;

– розробити серверну частину для динамічного формування посилань, автентифікації та перевірки цілісності;

– створити клієнтський застосунок для отримання контенту та перевірки підписів файлів;

– запровадити механізми контролю доступу та безпеки;

2 ПРОЄКТУВАННЯ АРХІТЕКТУРИ ТА КОМПОНЕНТІВ СИСТЕМИ ДОСТАВКИ КОНТЕНТУ

2.1 Моделювання функціональних і нефункціональних вимог до системи

На основі аналізу предметної галузі, цільової аудиторії, існуючих платформ доставки цифрового контенту та рекомендацій щодо побудови таких систем, сформульовано перелік задач, які має вирішувати розроблювана система. Основною метою є створення гнучкого, масштабованого та безпечного середовища для автоматизованої доставки модифікацій і ресурсів у грі з відкритою архітектурою (наприклад, Minecraft). Система має забезпечити зручне розповсюдження цифрових активів без залежності від закритих екосистем.

Сформульовано наступні вимоги:

– Клієнтська частина має автоматично ініціювати завантаження актуальної версії контенту.

– Контент має доставлятися з урахуванням географічного розташування користувача через CDN.

– Система повинна перевіряти цілісність файлів за допомогою цифрових підписів, запобігаючи підміні або пошкодженню даних.

– Нові версії контенту мають автоматично розміщуватися у хмарному сховищі за допомогою CI/CD-процесів.

– Усі завантаження повинні проходити автентифікацію через захищені канали та бути доступними лише авторизованим клієнтам.

– Висока швидкість і стабільність завантажень для користувачів з різних регіонів.

– Надійна робота при пікових навантаженнях (наприклад, масове оновлення клієнтів).

– Вся передача даних здійснюється через HTTPS, з обмеженням доступу

до об'єктного сховища через CloudFront OAI.

– Контент має бути реплікований між зонами доступності, щоб забезпечити відмовостійкість і уникнення втрати даних.

Для кращої візуалізації сценаріїв використання розроблюваної системи створено діаграму варіантів використання, що наведена на рисунку 2.1.

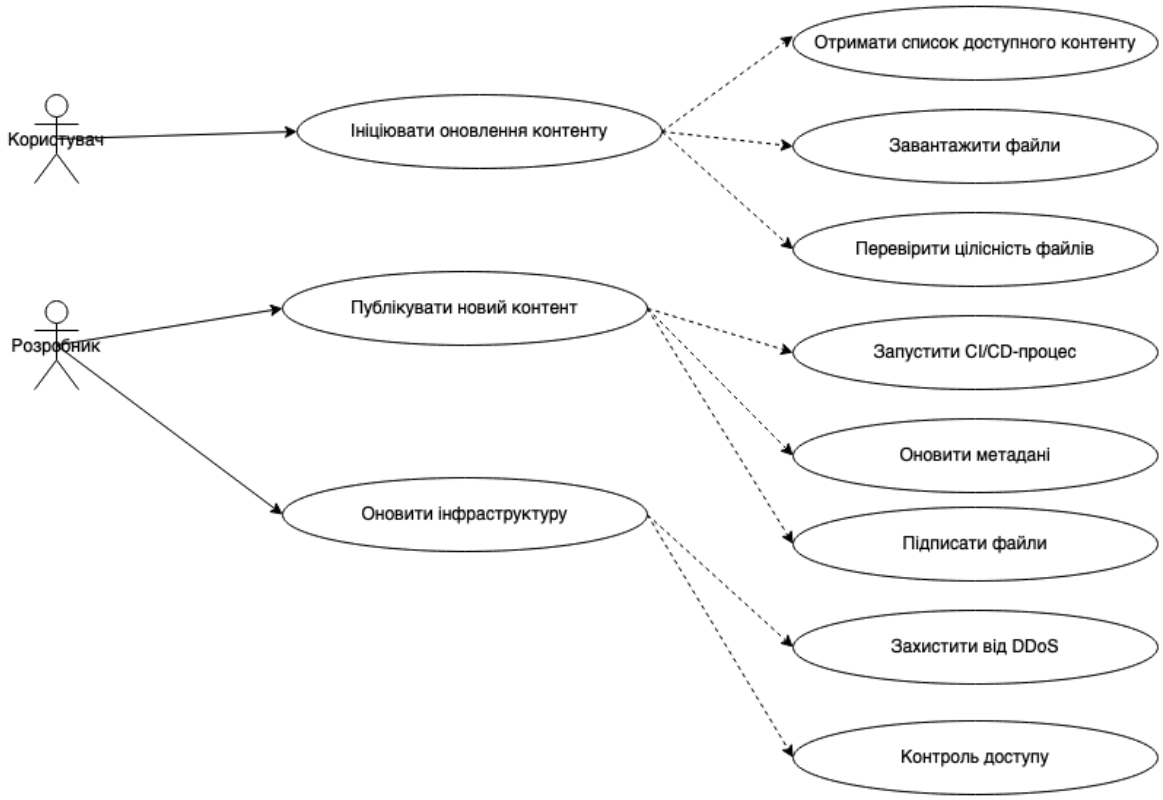


Рис. 2.1 Діаграма варіантів використання

На діаграмі виділено двох основних акторів — користувач та розробник. Користувач взаємодіє з клієнтським застосунком, ініціюючи оновлення контенту, отримуючи список доступних ресурсів, завантажуючи файли та перевіряючи їхню цілісність. Розробник, у свою чергу, публікує новий контент, запускає CI/CD-процеси, оновлює метадані, підписує файли, здійснює контроль доступу та відповідає за оновлення інфраструктури та безпеку (захист від DDoS тощо).

2.2 Проєктування архітектури системи доставки цифрового контенту

Відповідно до діаграми на рисунку 2.2, розроблена архітектура доставки цифрового контенту поєднує декілька сервісів AWS для забезпечення швидкого, масштабованого та безпечного доведення даних до користувача. Клієнтський додаток надсилає запит на спеціальний API, реалізований за допомогою Amazon API Gateway, який викликає безсерверну функцію AWS Lambda. Ця функція обробляє запит та формує оптимальний список URL-адрес для завантаження потрібного контенту через мережу доставки Amazon CloudFront, що використовує Amazon S3 як основне сховище.

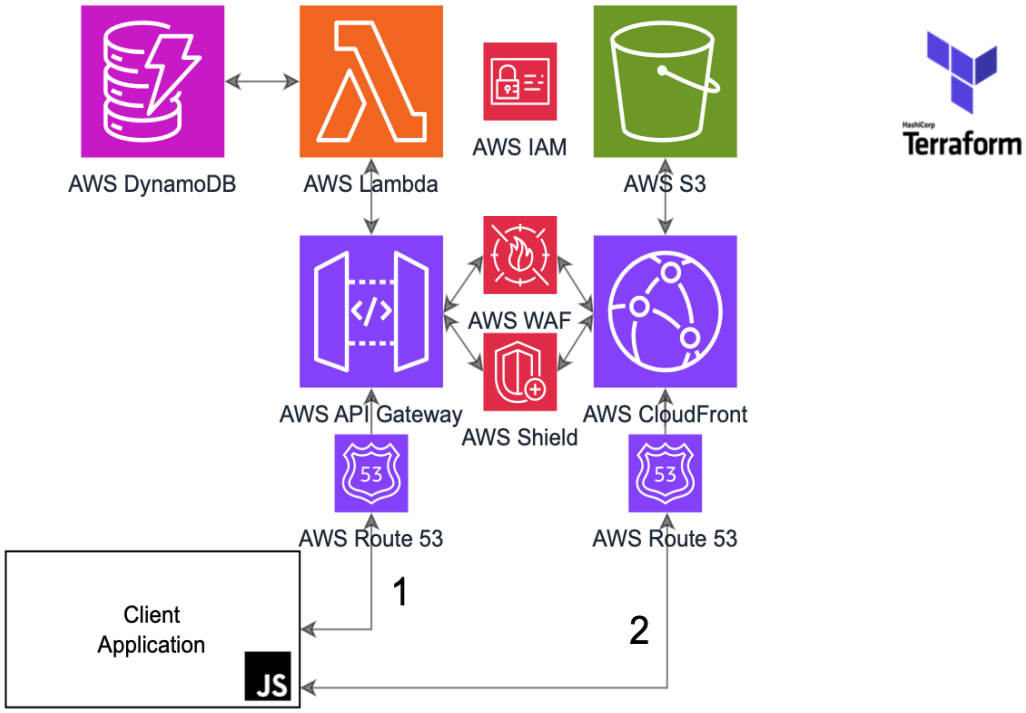


Рис. 2.2 Архітектура доставки цифрового контенту користувачу

Разом із посиланнями клієнту також передаються публічні криптографічні ключі, збережені у базі даних Amazon DynamoDB. Ці ключі використовуються для перевірки цифрових підписів файлів контенту, що завантажуються клієнтським застосунком. Перевірка підпису відбувається безпосередньо на боці користувача — клієнт застосовує отриманий відкритий

ключ до кожного з отриманих файлів і переконується, що підпис відповідає вмісту. Таким чином, будь-яке стороннє втручання у файли — зловмисна модифікація, пошкодження або заміна — буде негайно виявлено ще до того, як файл буде використано. Такий підхід гарантує не лише цілісність, а й автентичність отриманих даних, оскільки користувач може впевнено знати, що контент дійсно був створений і підписаний уповноваженою стороною, а також не зазнав змін під час доставки через мережу. Це особливо важливо для систем, де модифікація навіть одного байта може призвести до порушення цілісності або компрометації безпеки.

Таблиця 2.1

Структура таблиці для збереження метаданих в DynamoDB

Назва атрибуту	Тип
id	String
author	String
createdAt	String
description	String
privateKey	String
publicKey	String

Amazon Route 53 відіграє роль системи доменних імен (DNS) для цієї архітектури. Вона маршрутизує запити користувачів на найближчі географічно розташовані вузли системи, використовуючи записи типу Alias для інтеграції з CloudFront та API Gateway. Завдяки високій надійності Route 53 забезпечує швидкий і безвідмовний DNS-резолв і підтримує різні політики маршрутизації (наприклад, на основі затримки), що сприяє мінімізації часу відгуку користувача [18, с. 151 - 152].

Amazon CloudFront відповідає за глобальну доставку та кешування контенту. Це мережева служба CDN (Content Delivery Network), яка має понад 600 вузлів по всьому світу і автоматично спрямовує користувацькі запити до

найближчого регіонального вузла. Така архітектура суттєво знижує затримки під час передавання даних до кінцевого користувача, оскільки популярний контент роздається з кешу на периферійних серверах, а не щоразу з центрального сховища. Якщо необхідний файл уже закешовано на найближчому вузлі CloudFront, він доставляється миттєво; у протилежному випадку CloudFront автоматично завантажує його з походження (Amazon S3), кешує та віддає клієнту. Важливо, що CloudFront інтегрується з іншими сервісами AWS для підвищення продуктивності та безпеки: трафік між CloudFront і бекенд-сервісами (S3, API Gateway) передається високошвидкісною мережею AWS, а за потреби CloudFront може термінувати HTTPS-з'єднання на периферії, розвантажуючи бекенди. Крім того, використання CloudFront у зв'язці з S3 забезпечує економічну ефективність: дані, які передаються з S3 до CloudFront, не тарифікуються як вихідний трафік S3, що знижує загальні витрати на інфраструктуру [1, с. 7].

Amazon S3 використовується як основне сховище цифрового контенту (origin) завдяки своїй масштабованості та надійності. S3 гарантовано забезпечує 99,99% доступності даних і зберігає об'єкти з надлишковою резервізацією (durability 99.999999999%) у щонайменше трьох зонах доступності всередині регіону [1, с. 7]. Така архітектура сховища означає, що контент лишається доступним навіть у випадку відмови цілої дата-центру чи пристроїв зберігання. Використання S3 як незалежного шару зберігання дає змогу абстрагуватися від конкретного CDN-провайдера: в перспективі цей самий вміст S3 може паралельно розповсюджуватися через кілька CDN, забезпечуючи мульти хмарність та відмовостійкість рішення. В межах поточної архітектури S3 слугує єдиним джерелом істини для всіх файлів контенту, а CloudFront звертається до S3 лише у разі кеш-промаху. Для захисту вмісту використовується механізм Origin Access Identity (OAI) у CloudFront: S3-бакет налаштований як приватний, і прямий доступ до нього з Інтернету заборонений — натомість CloudFront виступає проксі з перевіреною особою. OAI гарантує, що користувачі не можуть оминути CDN і звернутися до S3 безпосередньо, тим самим усі запити проходять

через мережу CloudFront із застосуванням її контролю доступу [1, с. 14].

Amazon API Gateway реалізує рівень прикладного програмного інтерфейсу (API) для запитів, пов’язаних із отриманням контенту. Цей сервіс надає повністю керовану інфраструктуру для публікації та масштабування API будь-якого навантаження. API Gateway виступає “вхідними дверима” до бекенд-логіки: він приймає зовнішні запити (наприклад, від ігрового клієнта) та делегує їх опрацювання сервісам на бекенді. В даній архітектурі API Gateway інтегрований з AWS Lambda, тобто кожен HTTP-запит викликає відповідну Lambda-функцію. API Gateway автоматично обробляє всі аспекти трафіку, включно з управлінням потоками, автентифікацією запитів, підтримкою CORS та моніторингом, дозволяючи опрацьовувати сотні тисяч одночасних викликів без потреби ручного масштабування інфраструктури. Відсутність необхідності в адмініструванні серверів чи балансуванні навантаження суттєво спрощує підтримку API, а оплата лише за фактичні виклики робить рішення економічно вигідним. API Gateway також легко інтегрується з CloudFront для забезпечення мінімальної затримки і географічно оптимізованого доступу до API, за необхідності термінуючи з’єднання ближче до користувача.

AWS Lambda відповідає за виконання бізнес-логіки доставки контенту у безсерверному середовищі. Lambda-функція, написана для даної системи, виконує такі завдання: перевіряє параметри запиту та права доступу, звертається до бази даних для отримання необхідних метаданих (наприклад, списку файлів або ключів) та генерує у відповіді список URL-адрес на контент. За потреби, функція може здійснювати додаткові дії — наприклад, формувати підписані URL або одноразові токени доступу до вмісту. Використання Lambda гарантує автоматичне масштабування обчислювального шару: AWS прозоро запускає необхідну кількість ізольованих копій функції для обробки будь-якого обсягу запитів, забезпечуючи стабільну продуктивність без затримок. Lambda тісно інтегрується з іншими сервісами — зокрема, вона має прямий доступ до DynamoDB через AWS SDK, що дозволяє швидко отримувати чи оновлювати записи у таблиці під час опрацювання запиту. Важливо підкреслити, що обрана

без серверна архітектура (API Gateway + Lambda) підвищує надійність системи: відсутність виділених серверів означає, що відмова окремого вузла не вплине на доступність сервісу, а AWS автоматично виконує повтори або запустить функцію в іншій зоні у разі збоїв [22, с. 172 - 173].

Amazon DynamoDB використовується як високопродуктивна NoSQL база даних для зберігання метаданих контенту та пов'язаних даних. У таблиці зберігається інформація, необхідна для доставки. DynamoDB обрана через свою здатність забезпечувати надзвичайно низькі затримки доступу до даних при будь-якому масштабі навантаження: вона спроектована для часу відгуку порядку одиниць мілісекунд навіть під високим паралельним навантаженням [16, с. 163 - 164]. Цей сервіс автоматично розподіляє дані на декількох вузлах та динамічно масштабує продуктивність, тому таблиця може одночасно обслуговувати величезну кількість запитів читання/запису без деградації швидкодії. Як і інші керовані сервіси AWS, DynamoDB розподіляє дані як мінімум між трьома фізично відокремленими вузлами (Availability Zones), що гарантує збереження даних навіть у разі відмови цілої зони. Завдяки гнучкій схемі даних (модель ключ-значення з підтримкою вкладених структур) DynamoDB підходить для зберігання різнорідної інформації про контент, а її тісна інтеграція з Lambda забезпечує швидкий обмін даними безпосередньо всередині хмарної інфраструктури AWS.

Terraform обрано як засіб опису та автоматизації інфраструктури (IaC) через його гнучкість і незалежність від конкретного хмарного провайдера. [10, с. 3 - 4] Він використовує декларативну мову HCL, яка є простою для читання та написання, і дозволяє описувати складні ресурси однією мовою і єдиним інструментом. Основні переваги Terraform: можливість працювати з різними хмарами (AWS, Azure, Google Cloud тощо), підтримка великої кількості провайдерів і модулей, а також відстеження стану інфраструктури (state-файл). [10, с. 5 - 7] Це означає, що можна масштабувати застосунок в майбутньому під різні платформи, уникнувши «прив'язки» тільки до AWS (на відміну від CloudFormation, який орієнтований виключно на AWS. Також Terraform дозволяє

зберігати конфігурації в системі контролю версій, що забезпечує повторюваність та відкат змін. У порівнянні з AWS CloudFormation, Terraform надає більшу свободу та підтримку спільноти.

Всі ці заходи разом створюють довірену систему, в якій користувач отримує потрібний цифровий контент оперативно, без збоїв та із впевненістю в його цілісності.

2.3 Проєктування інфраструктури завантаження та публікації контенту

Проєктування інфраструктури завантаження і публікації цифрового контенту ґрунтується на CI/CD-процесах для забезпечення автоматизації, надійності та швидкості оновлень. У межах такого підходу кожне оновлення контенту (наприклад, ігрових ресурсів) проходить повний цикл – від моменту коміту в репозиторій до розміщення у хмарному сховищі – без втручання людини. Це досягається шляхом інтеграції репозиторію з хмарними сервісами через GitHub Actions – вбудовану систему автоматизації від GitHub, що дозволяє виконувати збірку, тестування та деплой безпосередньо при оновленні коду в репозиторії.

2.3.1 Роль GitHub Actions у CI/CD-процесі

GitHub Actions відіграє ключову роль у реалізації CI/CD-процесу для публікації контенту. В репозиторії проєкту створюється workflow-сценарій, який запускається автоматично при фіксації змін (commit/push) у визначеній гілці. Цей сценарій виконує кілька завдань:

– Збірка та підготовка контенту. Якщо контент потребує попередньої обробки (наприклад, компіляції або пакетування файлів), GitHub Actions запускає відповідні скрипти. Під час цього етапу може виконуватися автоматичне тестування або перевірка цілісності даних.

– Цифровий підпис файлів. Підготовлені файли контенту автоматично підписуються криптографічним підписом для гарантування їх автентичності та

цілісності. Процес підпису відбувається в CI/CD-пайплайні.

– Передача контенту до хмарного сховища. Після успішної збірки та підпису GitHub Actions здійснює автоматичне завантаження нових або оновлених файлів до хмарного сховища Amazon S3. Завантаження через API AWS виконується GitHub Actions із використанням облікових даних IAM, що надають право запису в відповідний бакет S3.

– Оновлення метаданих контенту. Після розміщення файлів у S3 необхідно оновити метадані про новий контент у базі даних Amazon DynamoDB. Для цього GitHub Actions викликає серверless-функцію AWS Lambda з відповідними параметрами. Lambda-функція отримує інформацію про новий файл (наприклад, ім’я бакету та ключ об’єкта в S3, розмір, а також криптографічний хеш чи публічний ключ для перевірки підпису) і здійснює запис в таблиці DynamoDB. У даному випадку виклик Lambda здійснюється безпосередньо з CI/CD-пайплайна, що дає більш явний контроль послідовності операцій.

2.3.2 Автоматизація процесу та стабільність системи

Повністю автоматизований конвеєр завантаження та публікації контенту забезпечує високу стабільність та повторюваність процесів. По-перше, усунення ручних дій зводить до мінімуму фактор людської помилки і прискорює вихід оновлень. Кожен коміт запускає стандартизований набір перевірених скриптів, що гарантує однакову послідовність операцій для кожної ітерації оновлення. По-друге, CI/CD-підхід підвищує впевненість у якості релізів: автоматизовані тести та перевірки, інтегровані в GitHub Actions, дозволяють виявити проблеми на ранніх етапах, що сприяє стабільності фінального контенту [20, с. 336]. По-третє, синхронізація розміщення файлів і оновлення метаданих відбувається атомарно в рамках єдиного workflow. Це означає, що система завжди перебуває в узгодженому стані: якщо контент успішно завантажено до S3, то і записи в DynamoDB гарантовано оновлені (в разі будь-якої помилки на проміжному кроці деплоймент переривається, і неконсистентні зміни не застосовуються). Подібна автоматизація публікації контенту є сучасним

підходом, який рекомендується як індустрією ігор, так і хмарними провайдерами [1, с. 7]. Таким чином, спроєктована інфраструктура використовує потенціал хмарних сервісів AWS для ефективного керування життєвим циклом цифрового контенту: від моменту внесення змін у репозиторій до доставки файлів кінцевим користувачам. На рисунку 2.3 схематично зображено цей процес: потік даних від розробника через CI/CD-пайплайн (GitHub Actions) до хмарного сховища Amazon S3 та бази метаданих Amazon DynamoDB, а також взаємодію допоміжних компонентів.

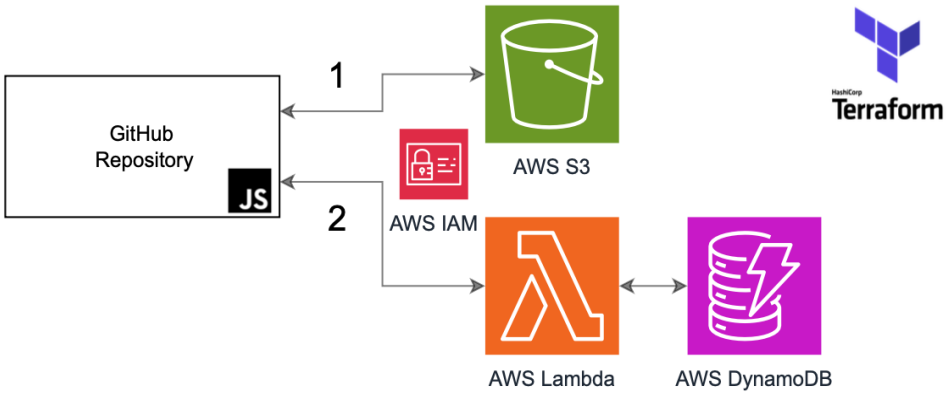


Рис. 2.3 Архітектура завантаження цифрового контенту

Це рішення демонструє, що поєднання автоматизованого CI/CD та керованих хмарних служб дозволяє досягти стабільного, безпечного і масштабованого механізму доставки контенту.

2.4 Проєктування клієнтського застосунку та користувацьких сценаріїв

Клієнтський застосунок розроблено з урахуванням постійної актуалізації цифрового контенту та зручності взаємодії користувача. Архітектурно він не є простим лаунчером, а виконує роль менеджера контенту: при запуску завжди перевіряє через API Gateway, чи актуальні ресурси контенту, і за потреби ініціює їхнє оновлення. Верифікація і завантаження нових ресурсів здійснюються окремим допоміжним модулем, який запускається клієнтською

частиною. Такий підхід розподіляє відповідальності між інтерфейсом та механізмом доставки контенту, підвищуючи надійність: основний клієнт фокусується на взаємодії з користувачем, тоді як окремий процес виконує перевірку версій файлів і завантаження оновлень. З точки зору користувача це відбувається прозоро – клієнтський застосунок передає запит на API Gateway для отримання метаданих останньої версії контенту і приймає рішення про необхідність оновлення. Якщо оновлення потрібне, запускається потік завантаження ресурсів (через CDN) із відображенням відповідного індикатора прогресу.

За замовчуванням клієнтська частина автоматично перевіряє наявність оновлень при кожному запуску програми. Такий механізм відповідає найкращим практикам: зокрема, у типовій реалізації Electron модуль автооновлення виконує запит до сервера одразу при старті застосунку. Додатково передбачено кнопку ручної перевірки оновлень в інтерфейсі, яка дозволяє користувачу ініціювати перевірку актуальності контенту у будь-який момент. Наявність такої опції підвищує рівень контролю користувача над застосунком та відповідає принципу user control and freedom (керованості інтерфейсу користувачем) з класичних евристик юзабіліті.

Якщо виявлено, що контент застарілий, клієнтська частина переходить до режиму оновлення, відображаючи спеціальний екран. Під час завантаження оновлень інтерфейс повинен надавати зворотний зв’язок користувачу про перебіг процесу. Відповідно до принципу visibility of system status (видимості стану системи), інтерфейс постійно інформує, що система виконує оновлення. Після успішного оновлення контенту система може повідомити користувача про це (наприклад, коротким повідомленням "Контент оновлено до останньої версії"). У випадку помилки система може повідомити про це користувача.

Для розробки клієнтського застосунку обрано фреймворк Electron.js, що дозволяє створювати крос-платформні десктопні програми засобами веб-технологій. Electron надає інтегроване оточення, яке поєднує рушій Chromium (для відображення інтерфейсу на базі HTML/CSS) та бекенд на Node.js для

доступу до системних ресурсів [21, с. 2]. Таким чином, реалізовано найкраще з двох світів: сучасний Web UI та можливості рідного десктопного застосунку. Переваги Electron для нашого проєкту включають насамперед крос-платформеність: один кодовий база працює на Windows, macOS та Linux без необхідності окремої розробки під кожну ОС. Це було важливо, адже цільовий контент (гра Minecraft і пов'язані модулі) використовується на різних платформах. Electron забезпечує упаковку застосунку у виконувані файли під кожну платформу і автоматично враховує специфіку ОС при побудові інсталятора [21, с. 2]. Другим ключовим фактором є швидкість розробки та знайомий стек технологій: команда розробників володіє HTML/CSS/JavaScript, тому створення інтерфейсу в Electron значно спрощує процес у порівнянні з нативною розробкою для кожної ОС. Electron дозволяє використовувати тисячі готових веб-бібліотек UI, що прискорює імплементацію бажаних компонентів інтерфейсу. Крім того, Electron має розширений API для доступу до функцій настільної ОС – від показу діалогових вікон і нативних меню до роботи з файловою системою та мережевими запитами [21, с. 2]. Це було критично для нашого застосунку, оскільки потрібно зберігати кеш завантаженого контенту на диску, перевіряти наявність файлів, керувати процесом завантаження – все це можливо безпосередньо через Node-модулі в середовищі Electron. Окрім технічних переваг, вибір Electron підтверджено практикою: багато сучасних успішних застосунків побудовані на цьому фреймворку (наприклад, Visual Studio Code, Discord тощо), що доводить його надійність та продуктивність у реальних умовах [21, с. 6].

Electron-застосунок складається з двох основних типів процесів:

– Main process це головний процес, що керує створенням вікон, взаємодією з файловою системою та виконується на базі Node.js. Він має доступ до низькорівневих API операційної системи.

– Render process це процеси відображення, які рендерять UI за допомогою Chromium. Кожне вікно застосунку створює окремий render process.

На рисунку 2.4 зображено, що з одного main process можуть ініціюватися два

render process:

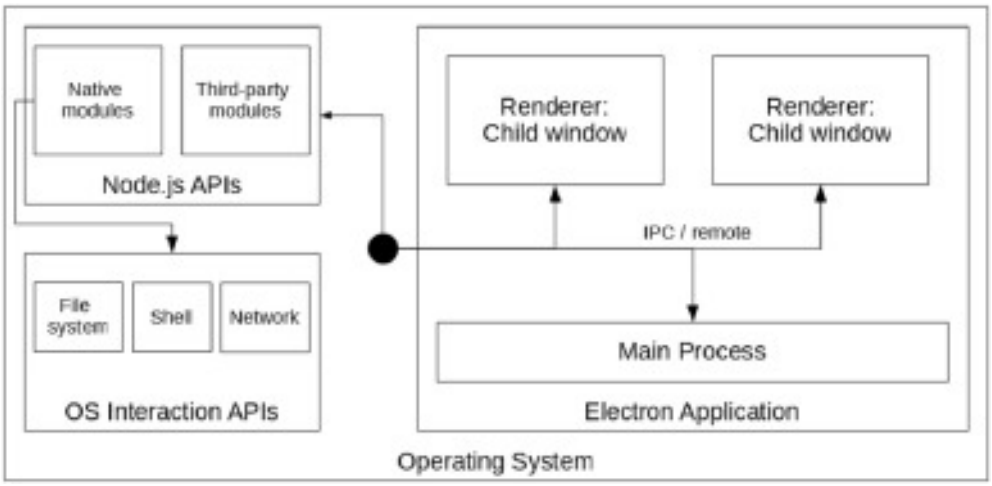


Рис. 2.4 Архітектура процесів у Electron.js [21, с. 3]

App Window – основне вікно застосунку, яке відображає інтерфейс користувача.

Hidden Window – приховане вікно, яке може виконувати фонові завдання (наприклад, перевірку оновлень, запуск оновлювача або перед завантаження контенту).

Нижче головного процесу зображено рівень Electron, що виступає як платформа, яка об’єднує рушій Chromium (для UI) та Node.js (для логіки).

Ця модель є типовою для Electron-застосунків і дозволяє:

– розділяти відповідальність між інтерфейсом і логікою.

– забезпечити реактивність UI навіть під час виконання фонових задач.

– використовувати приховані вікна для технічних операцій (наприклад, автооновлення контенту без впливу на основний UI).

У межах системи доставки контенту така архітектура дозволяє окремо реалізовувати інтерфейс користувача і процес завантаження ресурсів, не блокуючи основне вікно застосунку. Це підвищує стабільність і дозволяє легко масштабувати функціональність.

2.5 Визначення політик безпеки та контролю доступу

Архітектура системи доставки цифрового контенту реалізує набір політик безпеки та контролю доступу, що утворюють багаторівневу стратегію захисту. Такий підхід забезпечує захист даних і стабільність доступу до контенту навіть за наявності зловмисної активності. Інфраструктура розроблена з урахуванням принципів «zero trust» та сегментації прав доступу, що дозволяє локалізувати потенційні загрози й мінімізувати ризики. Основні компоненти цієї стратегії включають:

– Криптографічний підпис RSA-SHA256 для цілісності файлів: Кожен файл контенту підписується цифровим підписом RSA-SHA256, що дозволяє перевірити його цілісність та автентичність. Будь-яке несанкціоноване змінення файлу буде виявлено при перевірці підпису. Такий механізм працює на боці клієнта, тож перевірка здійснюється незалежно від середовища доставки, що значно підвищує надійність системи

– Політики доступу AWS IAM (Least Privilege): Використовується принцип найменших привілеїв – кожен компонент системи (AWS Lambda, Amazon S3, Amazon DynamoDB тощо) має суворо обмежені права доступу. За допомогою політик AWS Identity and Access Management визначаються дозволені дії тільки в межах необхідного для роботи сервісу, наприклад, функції Lambda можуть читати з певної таблиці DynamoDB або бакету S3, але не більше [19, с. 293]. Це запобігає виконанню зайвих або шкідливих операцій компонентами системи. Також це обмежує шкоду у разі компрометації одного з компонентів: доступ до інших ресурсів залишиться заблокованим.

– CloudFront OAI для захисту S3: Запроваджується CloudFront Origin Access Identity (OAI) для обмеження прямого доступу до вмісту в Amazon S3. Статичний контент зберігається в приватному S3-бакеті, а CloudFront (CDN) виступає єдиною точкою доступу. OAI гарантує, що користувачі можуть завантажувати файли лише через CloudFront, і не можуть звертатися до них напряму на S3 [1, с. 14]. Такий підхід підвищує безпеку, ізолюючи базове

сховище від прямого інтернет-доступу. Крім того, використання CloudFront дозволяє реалізувати додаткові механізми кешування, обмеження за географією, а також TLS-шифрування.

– AWS WAF для фільтрації HTTP-запитів: На рівні веб-додатка використовується AWS Web Application Firewall. AWS WAF аналізує та відфільтровує вхідні HTTP(S)-запити, блокуючи поширені атаки на рівні додатків. Зокрема, налаштовуються правила, що запобігають типовим веб-вразливостям (наприклад, SQL-ін'єкції, XSS) і обмежують трафік ботів [12, с. 122 - 124]. Це дозволяє допустити лише легітимний трафік до серверів контенту. Крім стандартних правил, можливо застосовувати користувацькі списки IP-адрес або шаблони запитів, що підвищує адаптивність системи захисту до конкретних загроз.

– AWS Shield для захисту від DDoS: На мережевому рівні система захищена сервісом AWS Shield. Стандартна версія AWS Shield (Standard) автоматично активна і безкоштовно забезпечує базовий захист від найбільш розповсюджених DDoS-атак на мережевому/транспортному рівнях (L3/L4) [12, с. 136 - 137]. Для розширеного захисту від високоінтенсивних або складних атак можливе використання AWS Shield Advanced. Shield Advanced також дозволяє отримувати аналітику щодо спроб атак та використовувати механізми автоматичного виявлення аномальної активності. Це рішення мінімізує ризик простою або деградації продуктивності системи внаслідок DDoS.

Багаторівневий підхід до безпеки, що включає перелічені вище механізми, гарантує як надійний захист даних (від підроблення, несанкціонованого доступу чи атак), так і стабільність доступу до цифрового контенту для легітимних користувачів. Усі шари архітектури працюють спільно, утворюючи комплексну систему стримувань і протидії загрозам, яка адаптується до змін у навантаженні та типах атак.

3 РЕАЛІЗАЦІЯ СИСТЕМИ ДОСТАВКИ ЦИФРОВОГО КОНТЕНТУ ТА КЛІЄНТСЬКОГО ЗАСТОСУНКУ

3.1 Загальна структура та компоненти системи

Репозиторій системи структуровано у дві основні директорії: content та delivery. Директорія content містить код, призначений для завантаження цифрового контенту на хмарне сховище та ініціації процесу публікації – зокрема, файл index.ts, який виконує основну логіку завантаження. Директорія delivery, своєю чергою, включає компоненти інфраструктури системи (під директорії terraform і backend), необхідні для розгортання та функціонування хмарних сервісів та директорія application, де описується робота клієнтського застосунку.

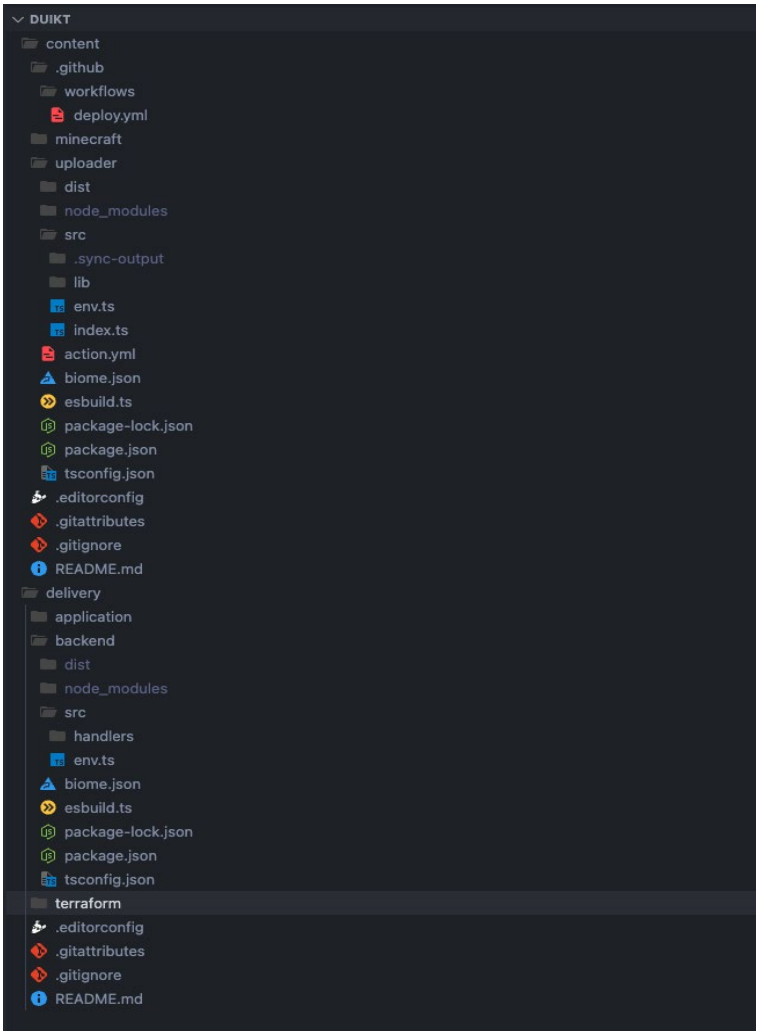


Рис. 3.1 Структура проєкту

У піддиректорії terraform зберігаються конфігурації інфраструктури, визначені засобами Terraform. Цей декларативний код охоплює налаштування ключових хмарних ресурсів AWS, таких як Route 53 (служба DNS для керування доменними іменами), Amazon API Gateway (шлюз для доступу до API Lambda-функцій), AWS Lambda (безсерверні обчислювальні функції) тощо – тобто всіх компонентів, що формують бекенд інфраструктури системи.

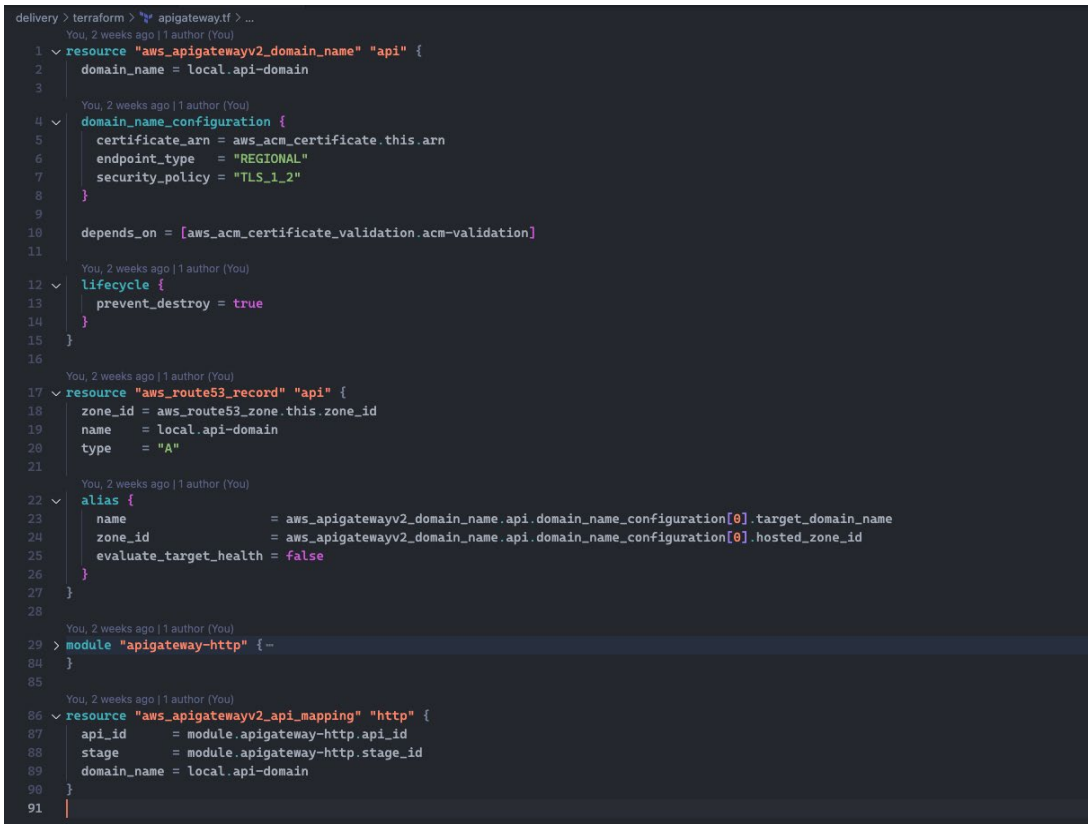


Рис. 3.2 Приклад налаштування Amazon API Gateway через Terraform

У межах реалізації було зареєстровано унікальне доменне ім’я duikt-content-delivery.online, яке використовується як основна адреса для доступу до API системи. Конфігурація DNS-записів здійснюється через Route 53, що дозволяє інтегрувати домен з іншими сервісами AWS, забезпечити коректну маршрутизацію запитів та подальшу масштабованість рішення. Використання Terraform дозволяє застосувати підхід «інфраструктура як код», що спрощує автоматизацію розгортання, відтворюваність середовища та масштабування системи.

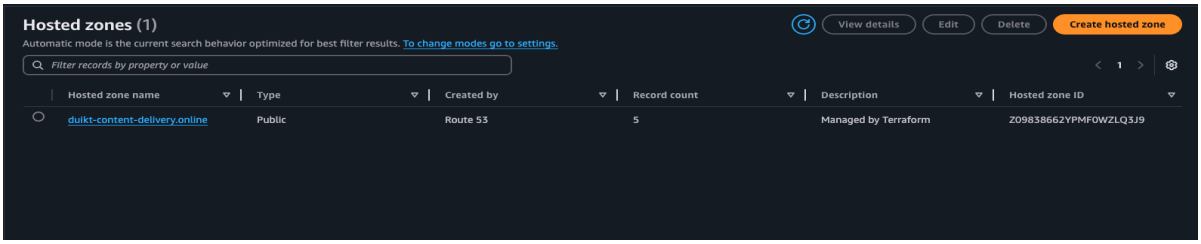


Рис. 3.3 Реалізація DNS-конфігурації

Під директорія backend містить вихідний код AWS Lambda-функцій (в директорії handlers), які реалізують бізнес-логіку керування версіями (ревізіями) контенту. Одна з цих функцій (http-get-revision.ts) опрацьовує HTTP-запит для отримання інформації про найновішу ревізію контенту з бази даних DynamoDB. Інша Lambda-функція (invk-create-revision.ts) призначена для створення нової ревізії: вона генерує унікальну пару криптографічних ключів (публічний та приватний ключі) за алгоритмом RSA-SHA256, формує цифровий підпис (signature) для опублікованого контенту за допомогою стандартної бібліотеки «node:crypto», а також записує усі відповідні метадані – включно з підписом, хешем контенту, версією, ID ревізії тощо – до таблиці DynamoDB.

```
function generateKeypair(options: { encoding: BufferEncoding }): {
  publicKey: string;
  privateKey: string;
} {
  const keys = generateKeyPairSync('rsa', { modulusLength: 2048 });

  return {
    publicKey: Buffer.from(keys.publicKey.export({ type: 'pkcs1', format: 'pem' })).toString(options.encoding),
    privateKey: Buffer.from(keys.privateKey.export({ type: 'pkcs1', format: 'pem' })).toString(options.encoding),
  };
}

function generateSignature(options: {
  data: string;
  privateKey: string;
  privateKeyEncoding: BufferEncoding;
  signatureEncoding: BinaryToTextEncoding;
}): string {
  return createSign('RSA-SHA256')
    .update(options.data)
    .sign(Buffer.from(options.privateKey, options.privateKeyEncoding), options.signatureEncoding);
}
```

Рис. 3.4 Приклад генерації криптографічних ключів та формування цифрового підпису

Зазначені функції інтегровані з налаштованим API Gateway, що дозволяє зовнішнім клієнтам викликати їх через стандартизовані HTTP-запити – таким

чином реалізується API для отримання актуальної версії контенту або додавання нової ревізії в систему.

Важливою складовою є процес CI/CD, налаштований за допомогою YAML-файлів GitHub Actions (наприклад, action.yml та deploy.yml). Ці файли описують автоматизований сценарій публікації контенту при оновленні репозиторію. Зокрема, конфігурація GitHub Actions визначає, що після кожного пушу змін до головної гілки репозиторію нова версія контенту автоматично завантажується до Amazon S3, а Lambda-функція реєстрації ревізії викликається для запису відповідних метаданих (ідентифікатору ревізії, автора, опису, криптографічних ключів тощо) в базу DynamoDB.

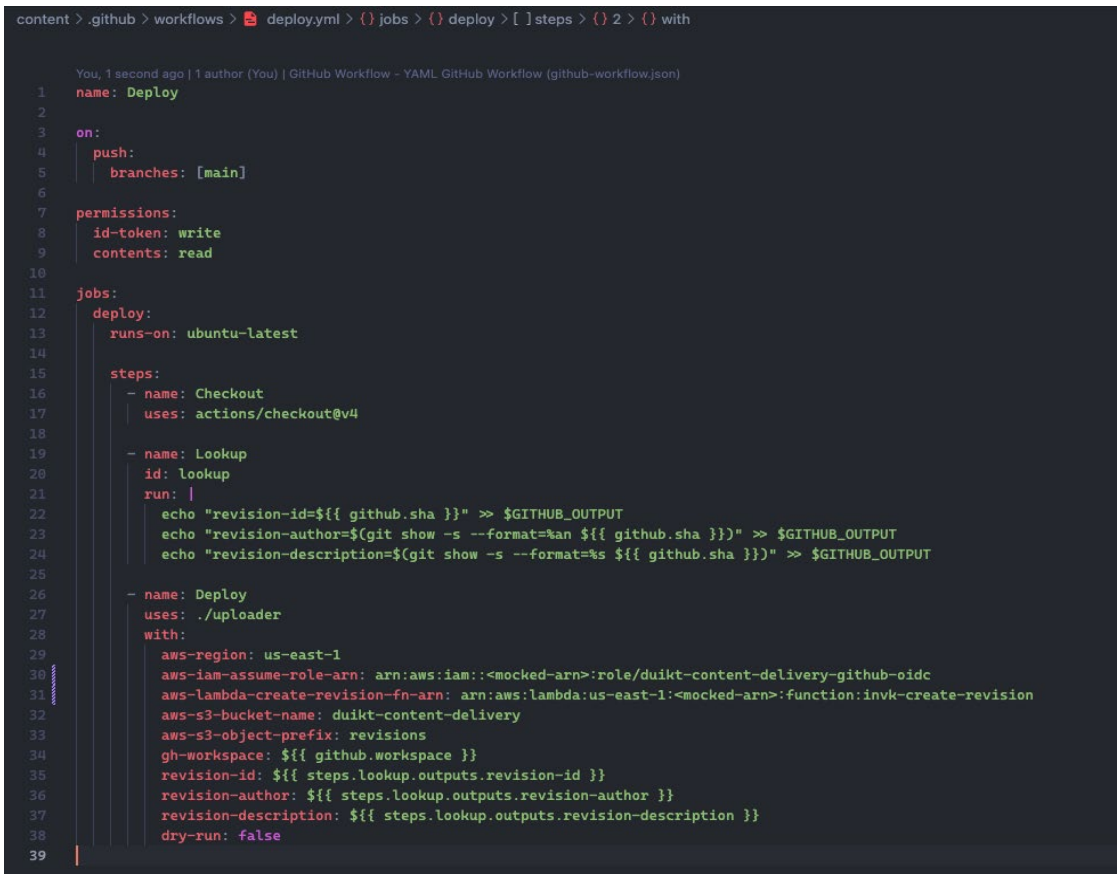


Рис. 3.5 Приклад коду з файла deploy.yml

Файл deploy.yml містить опис цього workflow (послідовності кроків розгортання), включно з використанням спеціальної GitHub Action, визначеної у action.yml, що безпосередньо виконує завантаження файлів і виклик Lambda-функції.

Таким чином, завдяки інтеграції з GitHub Actions, система забезпечує безперервну інтеграцію та доставку контенту без ручного втручання, автоматично публікуючи новий контент і оновлюючи необхідні компоненти інфраструктури.

3.2 Реалізація клієнтського застосунку

Для реалізації клієнтського застосунку було використано платформу Electron.js та мову TypeScript. Це рішення дозволило створити багатоплатформний настільний додаток із використанням веб-технологій (HTML, CSS, JavaScript). Робота клієнтського застосунку описана в директорії application, яка знаходиться в директорії delivery.

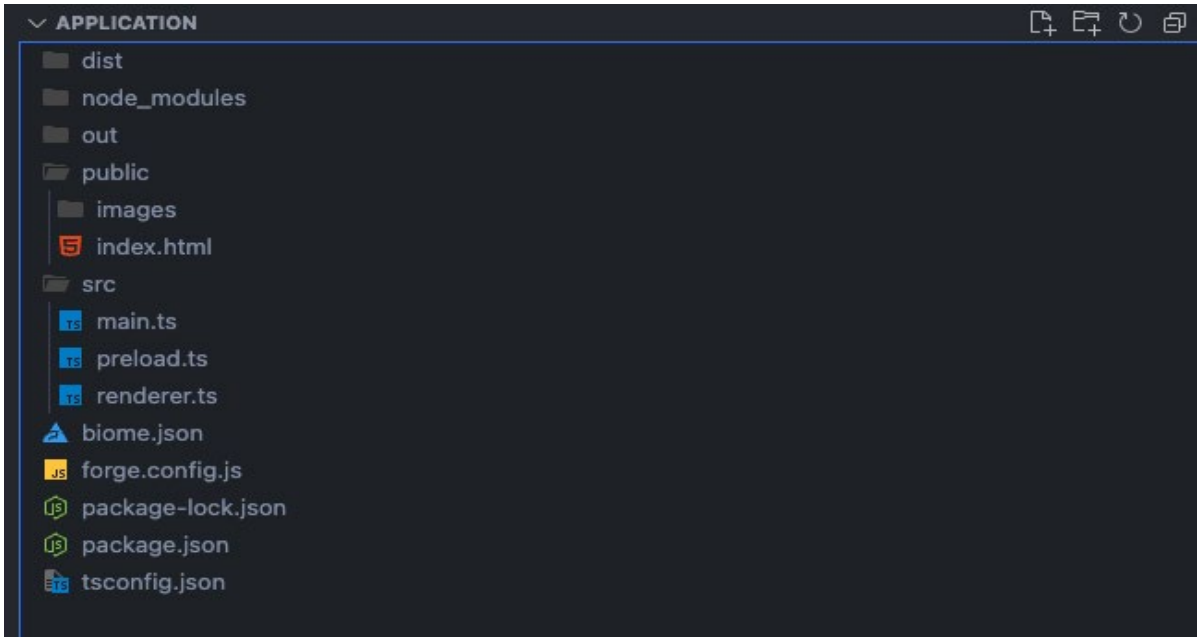


Рис. 3.6 Структура клієнтського застосунку

Код клієнтського застосунку структуровано у кількох файлах відповідно до архітектури Electron. Основними компонентами є головний процес (main.ts), процес інтерфейсу (renderer.ts) та скрипт попереднього завантаження (preload.ts), а також інші допоміжні модулі. Файл main.ts відповідає за запуск програми, створення вікна, роботу з файловою системою та виконання криптографічних перевірок (зокрема, перевірки підпису та хешів файлів). Логіка інтерфейсу

реалізована у renderer.ts: цей компонент ініціює запити до API, отримує та обробляє дані оновлення, відображає журнал прогресу та реагує на дії користувача. Файл preload.ts використовується для безпечної взаємодії між головним та відображуваним процесами – через contextBridge він експортує необхідні функції (window.electronAPI.checkSignature, window.electronAPI.sync) для виклику методів головного процесу з рівня інтерфейсу. В директорії public знаходяться HTML та CSS код разом з логотипами.

Клієнтський застосунок ініціює запит до шлюзу API (API Gateway) для отримання переліку файлів з відповідними метаданими останньої ревізії контенту. Отримавши цю інформацію, застосунок виконує перевірку цілісності даних за допомогою цифрового підпису (алгоритм RSA-SHA256). Для цього використовується відкритий ключ, отриманий від сервера, що дозволяє підтвердити автентичність і цілісність отриманого списку файлів (маніфесту).

Застосунок у разі успішної верифікації підпису автоматично завантажує всі файли з отриманого списку. На рисунку 3.7 видно, що в журналі прогресу відображаються повідомлення про перебіг завантаження кожного файлу, а по завершенні процесу – підсумкове повідомлення про успішне виконання (позначене зеленою іконкою).

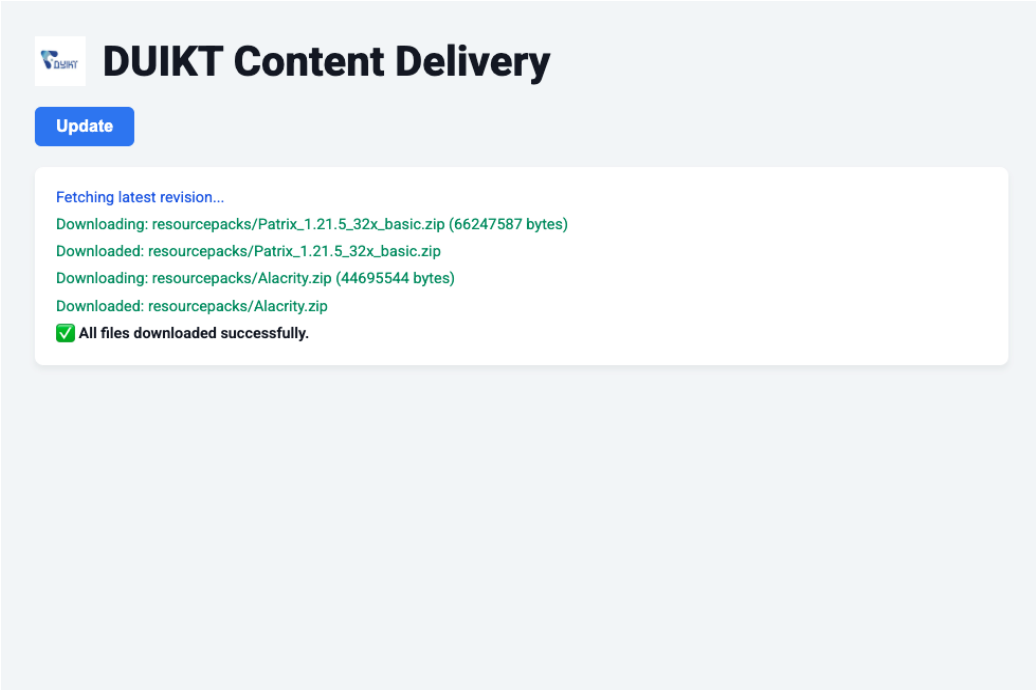


Рис. 3.7 Форма завантаження актуального контенту (Успіх)

Таким чином, користувач чітко інформується про успішне завершення оновлення контенту.

Рисунок 3.8 ілюструє ситуацію, коли перевірка цифрового підпису не вдалася. У такому разі користувачу виводиться повідомлення про помилку «Signature verification failed» (червоним кольором), і процес завантаження контенту припиняється.

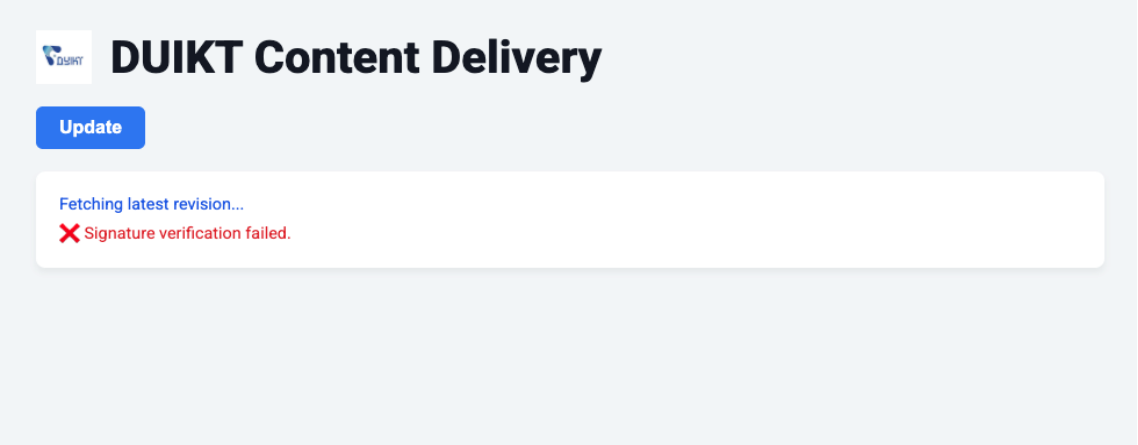


Рис. 3.8 Форма завантаження актуального контенту (Збій перевірки цілісності)

Таким чином, користувач негайно дізнається про помилку цілісності, і оновлення не виконується.

Інтерфейс застосунку реалізовано з урахуванням зручності та наочності для користувача. Всі етапи процесу оновлення супроводжуються зрозумілими повідомленнями про прогрес, а після завершення або у разі помилки надається чітке відповідне сповіщення.

3.3 Сценарій роботи системи

Робота системи доставки цифрового контенту реалізована у вигляді набору автоматизованих етапів, які починаються з моменту запуску клієнтського застосунку та охоплюють як взаємодію з інфраструктурою AWS, так і перевірку цілісності та завантаження контенту користувачеві. Для ілюстрації логіки роботи системи використано дві діаграми: діаграму послідовностей та схему взаємодії компонентів.

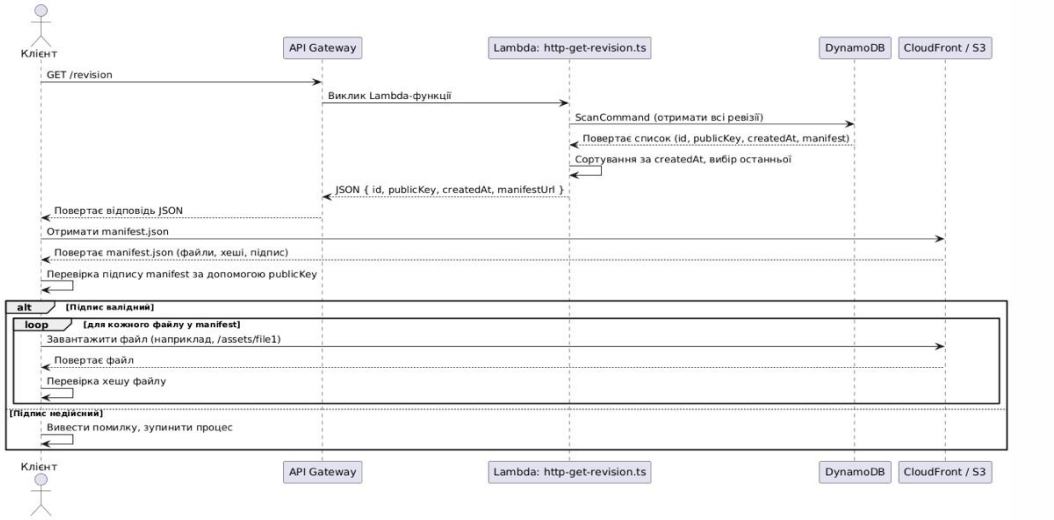


Рис. 3.8 Діаграма послідовностей

На рисунку 3.8 показано типовий сценарій отримання останньої ревізії контенту клієнтським застосунком. Після запуску, клієнт надсилає запит GET /revision до API Gateway, який у свою чергу викликає відповідну Lambda-функцію (http-get-revision.ts). Ця функція читає записи з DynamoDB, сортує їх за датою створення (createdAt) і повертає останню ревізію разом з URL до manifest.json, відкритим ключем та підписом.

Клієнт після цього завантажує файл manifest.json з CDN (через CloudFront) та перевіряє підпис за допомогою відкритого ключа. Якщо підпис виявляється дійсним, застосунок переходить до циклічного завантаження кожного з файлів, вказаних у manifest. Кожен файл перевіряється на відповідність хешу, після чого вважається завантаженим. У разі недійсного підпису система повідомляє про помилку та припиняє оновлення.

На рисунку 3.9 подано загальну архітектурну схему взаємодії між усіма компонентами системи, включно з користувачем (User), розробником (Developer) та середовищем AWS. Сценарій роботи системи включає:

– Користувач запускає клієнтський застосунок, який звертається до Route 53 для резолвінгу домену duikt-content-delivery.online, після чого здійснює запит до API Gateway.

– API Gateway взаємодіє з Lambda (http-get-revision.ts), яка витягує метадані з DynamoDB.

– Клієнт отримує посилання на manifest.json та завантажує його разом з контентом через CloudFront.

– У паралельному сценарії, розробник публікує нову ревізію контенту, здійснюючи push до репозиторію. Після цього GitHub Actions виконує CI/CD-процес: завантажує файли до Amazon S3 та викликає Lambda-функцію invk-create-revision.ts, яка записує нову ревізію в DynamoDB.

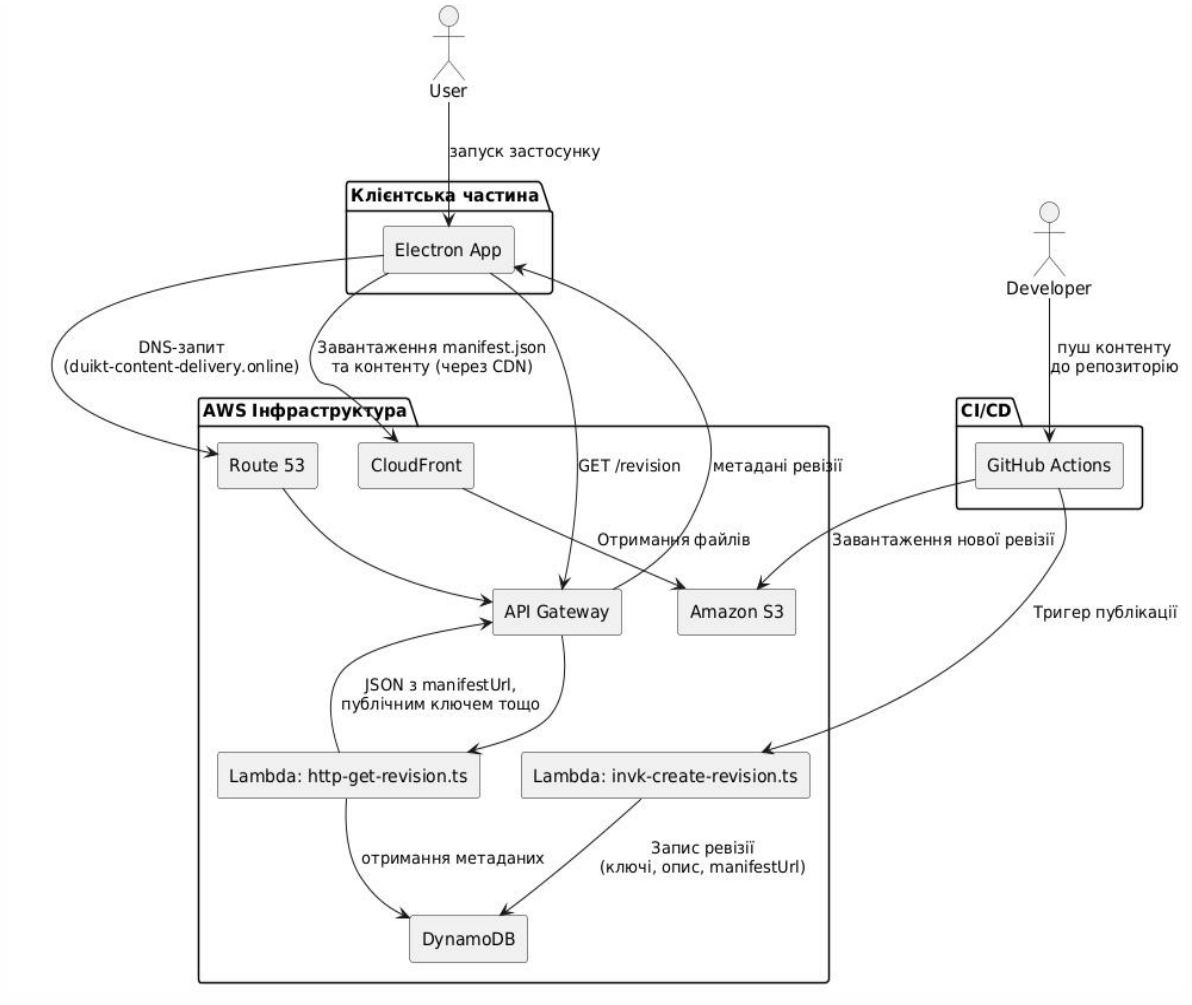


Рис. 3.9 Схема взаємодії компонентів

Ця схема демонструє повний життєвий цикл роботи системи — від публікації контенту розробником до отримання актуальної версії користувачем.

4 ТЕСТУВАННЯ ТА ОЦІНЮВАННЯ ЕФЕКТИВНОСТІ ФУНКЦІОНУВАННЯ СИСТЕМИ ДОСТАВКИ КОНТЕНТУ

4.1 Оцінювання ефективності функціонування системи доставки контенту

Оцінювання ефективності системи доставки цифрового контенту є важливою складовою при впровадженні подібних рішень, особливо у контексті ігрових застосунків, де критичним є час завантаження, стабільність з'єднання, а також цілісність та актуальність даних. Відповідно до рекомендацій, наведених у whitepaper Content Delivery for Games [1], основними критеріями ефективності системи є:

– Пропускна здатність та частота помилок: основним показником продуктивності виступає доступна ширина каналу до клієнта. В реалізованій системі використовується Amazon CloudFront, який забезпечує кешування контенту ближче до користувача, мінімізуючи затримку та зменшуючи ймовірність помилок при завантаженні. Завдяки цьому, навіть у разі пікових навантажень, система зберігає стабільну швидкість обслуговування.

– Географічне охоплення: оскільки CDN має сервери у багатьох регіонах світу, контент завжди доставляється з найближчого до користувача вузла. Це особливо важливо для гравців, що перебувають у країнах з обмеженим доступом до швидкісного інтернету. Реалізована система не потребує налаштування кількох CDN — Amazon CloudFront оптимізує маршрут доставки автоматично.

– Функціональність інтеграції та автоматизації: система інтегрована з CI/CD-процесом через GitHub Actions. Після кожного оновлення вміст автоматично публікується у сховищі (Amazon S3), а оновлення метаданих виконується через Lambda. Це мінімізує ручну участь розробника і знижує ризик людських помилок.

– Стійкість до пікових навантажень: система здатна ефективно обробляти підвищений трафік, наприклад під час масового оновлення клієнтів. CDN-провайдер (CloudFront) автоматично масштабується та не допускає деградації продуктивності навіть за різкого збільшення кількості запитів.

– Безпека: контент доставляється через HTTPS із використанням CloudFront Origin Access Identity (OAI), що обмежує прямий доступ до Amazon S3. Усі файли мають цифровий підпис, який перевіряється на клієнтській стороні. У разі фальсифікації або зміни підпису система блокує завантаження, забезпечуючи надійний контроль цілісності.

Згідно з вищезазначеними критеріями, реалізована система демонструє високу ефективність: вона забезпечує швидку, безпечну та стабільну доставку контенту користувачу незалежно від географічного розташування, підтримує CI/CD-автоматизацію та успішно витримує пікові навантаження. Такий підхід відповідає сучасним вимогам до ігрових платформ і забезпечує позитивний користувацький досвід.

4.2 Логування процесів у CI/CD та моніторинг виконання

Однією з ключових переваг реалізованої системи є повна прозорість процесу публікації контенту завдяки інтеграції CI/CD та механізмів централізованого логування. Весь процес публікації автоматизовано за допомогою GitHub Actions, що забезпечує як повторюваність, так і відстежуваність кожного окремого кроку. На прикладі рисунку 4.1 продемонстровано, як GitHub Actions детально фіксує кожен етап — від обчислення хешів і генерації manifest.json, до завантаження файлів до Amazon S3 та реєстрації нової ревізії в базі даних. Це значно полегшує аудит, відкат змін і відстеження помилок у разі збоїв або непередбачених ситуацій.

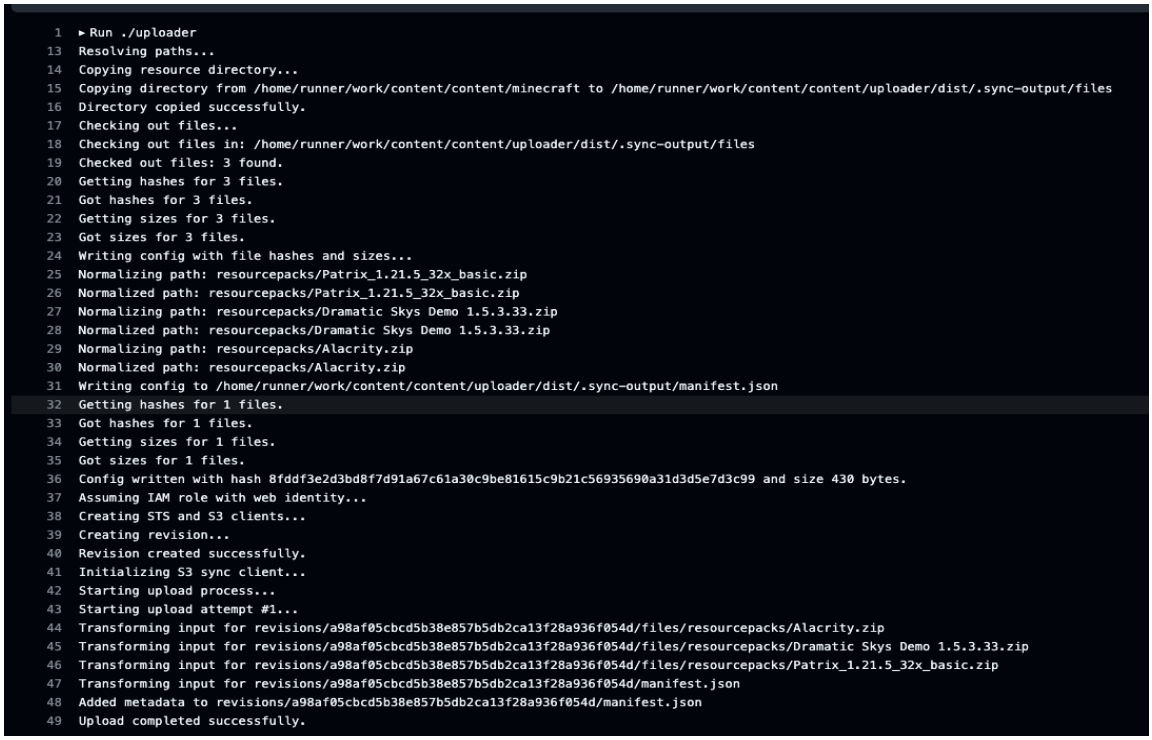


Рис. 4.1 Приклад логів CI/CD процесу публікації контенту

Кожна дія в CI/CD-процесі виводиться у лог, що дозволяє швидко виявити потенційні помилки або аномалії у виконанні. У разі виникнення проблем або збоїв на будь-якому з етапів (наприклад, недоступність сховища або некоректна IAM-роль), GitHub автоматично позначає помилку та зупиняє виконання сценарію.

Крім того, всі серверні компоненти (AWS Lambda-функції) надсилають свої логи до AWS CloudWatch — централізованого сервісу моніторингу та спостереження. Це дозволяє відстежувати виконання запитів у реальному часі, переглядати історію помилок, а також здійснювати фільтрацію логів за ідентифікатором запиту або ревізії. Завдяки цьому можлива як оперативна діагностика, так і довгостроковий аудит роботи системи.

4.3 Функціональне тестування клієнтського застосунку

Функціональне тестування було проведено з метою перевірки коректної роботи клієнтської частини системи доставки цифрового контенту, а саме: отримання актуальної ревізії, завантаження вмісту та використання його за призначенням. Для перевірки було обрано реальний приклад – завантаження ресурс-паків (resource packs) до гри Minecraft, які змінюють візуальні елементи гри. На початку тестування каталог resourcepacks був порожнім.

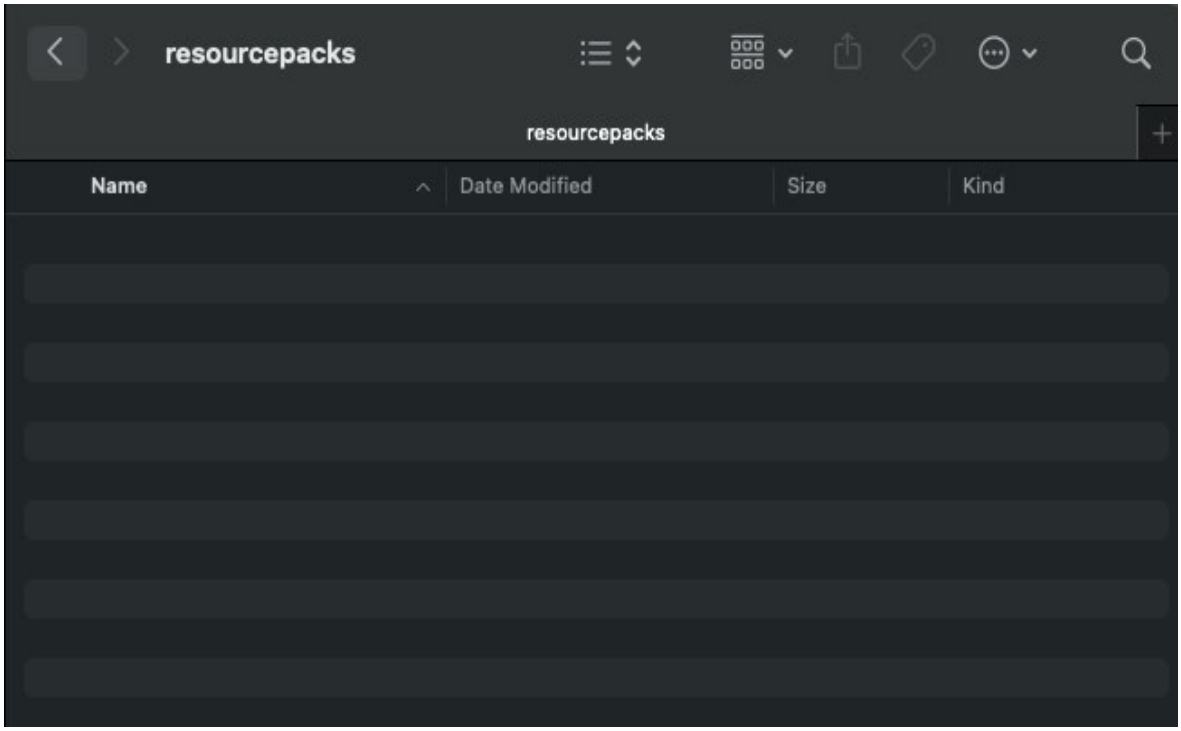


Рис. 4.2 Порожній каталог resourcepacks до початку завантаження

Після запуску клієнтського застосунку було виконано запит до API, отримано manifest.json, верифіковано підпис і завантажено відповідні архіви через CloudFront. Відображення прогресу та статусу завантаження можна побачити у вікні застосунку (рисунок 4.3), а також у мережевій активності (DevTools).

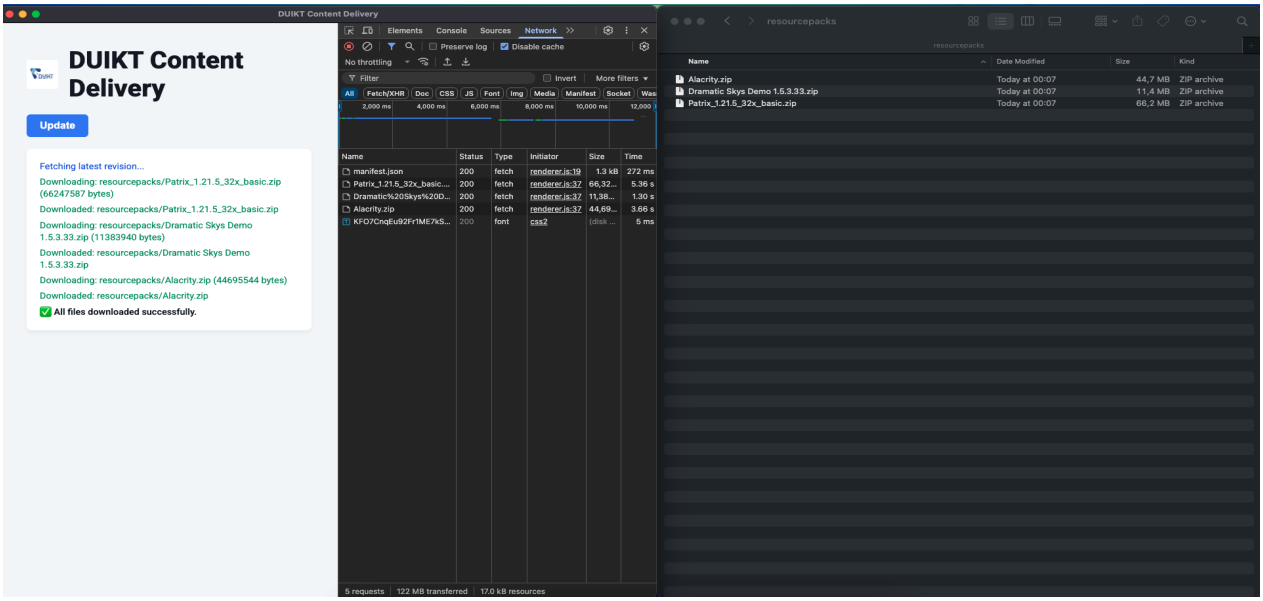


Рис. 4.3 Вікно клієнтського застосунку з успішним статусом завантаження та завантажені архіви у файловій системі

Після завершення процесу, у папці resourcepacks з’явилися всі необхідні ZIP-архіви, і при наступному запуску Minecraft вони були автоматично розпізнані і відображені в меню вибору ресурс-паків (рисунок 4.4). Це підтверджує, що:

– файли було доставлено коректно.

– архіви не були пошкоджені або змінені.

– застосунок успішно інтегрується з ігровим середовищем.

Рис. 4.4 Інтерфейс вибору ресурс-паків у Minecraft

Таким чином, тестування показало повну відповідність очікуваному функціоналу: автоматичне оновлення, локальне збереження, відповідність формату та подальше застосування ресурсів у цільовому середовищі.

4.4 Тестування механізмів безпеки

У межах тестування безпекових механізмів системи було здійснено перевірку одного з ключових компонентів захисту — перевірки цифрового підпису manifest-файлу. Цей підпис формується на серверній стороні під час публікації контенту та додається до об’єкта в Amazon S3 у вигляді метаданих (рисунок 4.5).

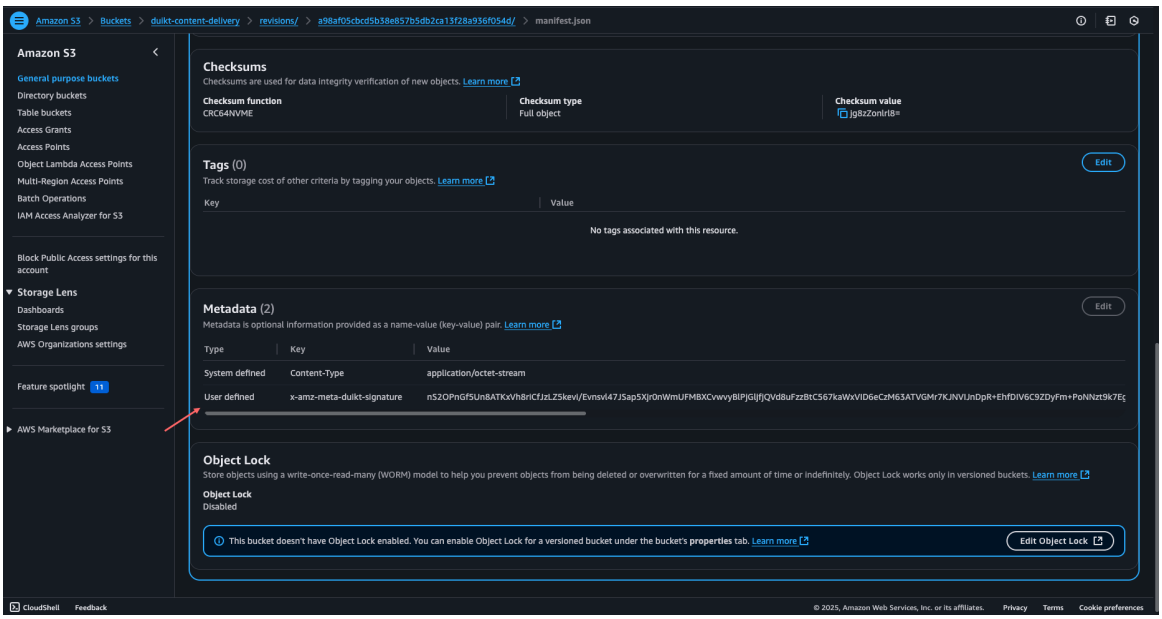


Рис. 4.4 Валідний цифровий підпис у метаданих S3-об’єкта

Для перевірки надійності реалізації механізму перевірки цілісності, вручну було змінено значення підпису (x-amz-meta-duikt-signature) на значення «test», без внесення жодних змін до самих даних у файлі manifest.json. Після цього було вручну інвалідовано кеш у CloudFront для відповідного об’єкта, щоб гарантувати, що клієнтський застосунок отримає саме оновлену версію з підробленим підписом, а не закешовану попередню. У результаті, під час наступного запуску клієнтського застосунку механізм перевірки підпису

очікувано не пройшов валідацію, після чого система автоматично припинила завантаження з відповідним повідомленням про помилку. Це підтверджує, що перевірка справно спрацьовує у випадках виявлення змін або пошкодження підпису.

Окремо було протестовано IAM-політики, які регулюють доступ до об’єктів Amazon S3. Було здійснено спробу прямого доступу до manifest.json через URL-адресу, згенеровану для ревізії. У відповідь сервер повернув помилку Access Denied (рисунок 4.5), що свідчить про правильно налаштовану політику обмеження доступу — зокрема, активне використання CloudFront Origin Access Identity (OAI).

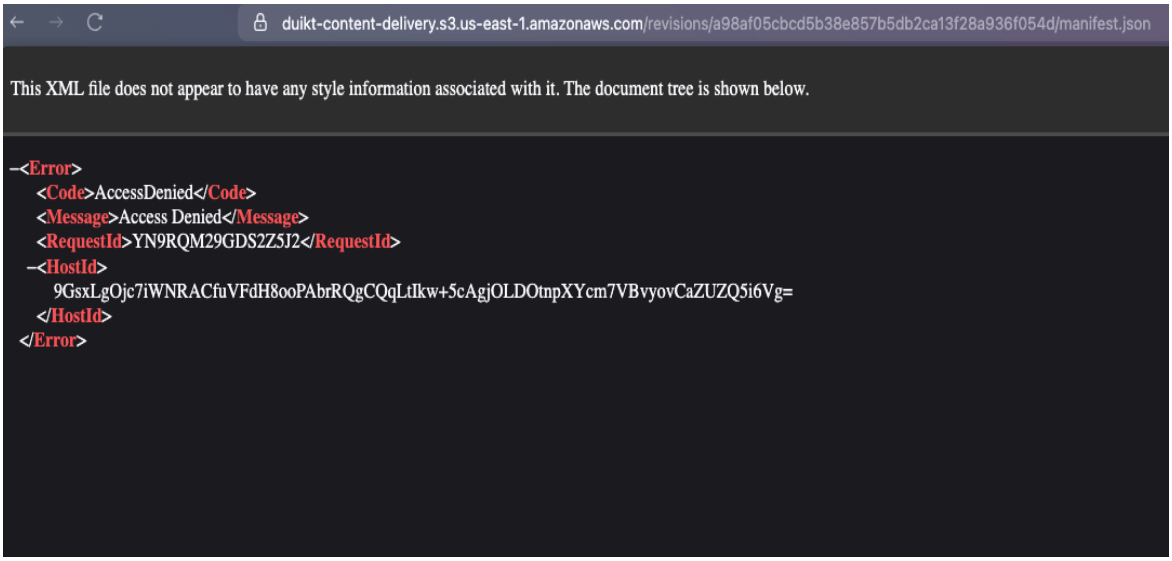


Рис. 4.5 Відмова в доступі до об’єкта manifest.json при порушенні IAM policy

Таким чином, доступ до контенту можливий лише через авторизовані канали (через CloudFront або клієнтський застосунок), що забезпечує захист від несанкціонованого отримання даних.

ВИСНОВКИ

У процесі виконання кваліфікаційної роботи було спроєктовано та реалізовано систему доставки цифрового контенту, орієнтовану на використання в ігрових застосунках. Результатом роботи стало створення ефективного рішення, яке забезпечує автоматизовану публікацію, безпечну доставку та верифікацію цілісності контенту з урахуванням географічного розташування користувача.

У межах дослідження виконано наступні завдання:

– Проведено аналіз сучасних підходів до доставки цифрового контенту в ігрових системах. Було виявлено низку обмежень традиційних рішень, зокрема недостатню підтримку модифікованих ресурсів, слабку гнучкість та автоматизації. Враховуючи ці недоліки, було обґрунтовано доцільність побудови власної хмарної системи доставки.

– Обґрунтовано вибір стеку технологій для реалізації системи: сервіси Amazon Web Services (S3, CloudFront, Lambda, API Gateway, Route 53, IAM, WAF, Shield), інструмент інфраструктури як код (Terraform), систему CI/CD (GitHub Actions), фреймворк для клієнтських застосунків (Electron.js) та мову програмування TypeScript. Такий вибір забезпечив кросплатформеність, масштабованість і високу безпеку.

– Сформульовано функціональні та нефункціональні вимоги до системи, які включали: автоматизацію доставки контенту, географічне кешування, перевірку цифрових підписів, захищений доступ, відмовостійкість і підтримку пікових навантажень.

– Розроблено архітектуру хмарної інфраструктури з використанням сервісів AWS. Особливу увагу приділено реалізації безсерверної логіки за допомогою AWS Lambda, зберіганню ревізій у DynamoDB та захисту контенту за допомогою CloudFront з OAI. Власне доменне ім’я duikt-content-delivery.online було налаштовано через Route 53.

– Реалізовано механізм CI/CD: під час публікації контенту GitHub Actions автоматично генерує хеші, створює manifest.json, підписує його, завантажує до S3 та викликає Lambda-функцію для створення ревізії.

– Розроблено клієнтський застосунок на Electron.js, який під час запуску ініціює отримання актуальної ревізії, перевіряє цифровий підпис manifest-файлу (RSA-SHA256), завантажує ресурси через CloudFront і виводить повідомлення про успіх або помилку. Додаток протестовано на прикладі завантаження ресурс-паків для гри Minecraft — результати підтвердили працездатність усіх компонентів.

– Налаштовано політики доступу до об’єктів у S3. Проведено тестування безпеки: змінений цифровий підпис призводить до очікуваного блокування завантаження, а спроба прямого доступу до об’єкта повертає помилку AccessDenied, що свідчить про коректну роботу IAM-політик.

– Система була протестована у повному циклі: від генерації нової ревізії до її отримання клієнтом. Завдяки логуванню у GitHub Actions та AWS CloudWatch забезпечено просте моніторингування й діагностику помилок.

Робота пройшла апробацію. За її результатами було опубліковано наступні тези доповідей:

1. Стратович М. Я., Аброскін Ю. Ю. Забезпечення безпеки даних у системі розподіленої доставки цифрового контенту на прикладі гри Minecraft. Матеріали XII Всеукраїнської науково-практичної конференції молодих учених «Інформаційні технології – 2025», 15.05.25, КСУ імені Бориса Грінченка, Київ, Україна, С. 331-332.
2. Стратович М. Я., Аброскін Ю. Ю. Архітектурне рішення для системи розподіленої доставки цифрового контенту на прикладі гри Minecraft. // Матеріали VI Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». 24.04.2025, ДУІКТ, Київ, Україна, С. 490-494.

ПЕРЕЛІК ПОСИЛАНЬ

1. Somers K., Krasikov E., Gopala N. Content Delivery for Games: Optimizing the Delivery of Game Assets to Players. AWS Whitepaper. 2020. URL: https://d1.awsstatic.com/whitepapers/content-delivery-for-games.pdf (date of access: 26.02.2025).
2. Pereira L. S., Bernardes M. Modding as game development: Investigating the influences over how mods are made. Journal of Gaming & Virtual Worlds. 2021. Vol. 13, no. 2. P. 151–172. URL: https://www.researchgate.net/publication/356143106_Modding_as_game_development_Investigating_the_influences_over_how_mods_are_made (date of access: 27.02.2025).
3. Edge cloud strategies for gaming. *Powering the best of the internet | Fastly*. URL: https://www.fastly.com/resources/industry-report/gaming#introduction (date of access: 28.02.2025).
4. Unlock UGC for your game | Getting Started CurseForge for Studios API. *Getting Started CurseForge for Studios API*. URL: https://docs.curseforge.com/docs/curseforge-for-studios/intro/ (date of access: 28.02.2025).
5. Scacchi W. Modding as a Basis for Developing Game Systems. Position paper. 2011. URL: https://ics.uci.edu/~wscacchi/GameLab/Scacchi-GAS-Position-Paper-Feb11.pdf (date of access: 01.03.2025).
6. Steam Workshop (Steamworks Documentation). *Steamworks*. URL: https://partner.steamgames.com/doc/features/workshop (date of access: 01.03.2025).
7. Epic Developer Resources (Documentation). *Epic Games*. URL: https://dev.epicgames.com/docs/resources-overview (date of access: 01.03.2025).
8. Murillo J., Hanafy W. A., Irwin D., Sitaraman R., Shenoy P. CDN-Shifter: Leveraging Spatial Workload Shifting to Decarbonize Content Delivery Networks. In: Proceedings of the ACM Symposium on Cloud Computing (SoCC

'24), November 20–22, 2024, Redmond, WA, USA. ACM, New York, NY, USA, 17 pages. URL: https://lass.cs.umass.edu/papers/pdf/socc24_cdn_sust.pdf (date of access: 16.03.2025).

9. Connected Devices Team. Beyond Speed Tests: Steam Performance. ThousandEyes Blog. URL: https://www.thousandeyes.com/blog/beyond-speed-tests-steam-performance (date of access: 18.03.2025).
10. Winkler S. Terraform in Action. Manning Publications Co. LLC, 2021. 408 p.
11. What is a content delivery network (CDN)? Cloudflare Learning Center. URL: https://www.cloudflare.com/learning/cdn/what-is-a-cdn/ (date of access: 20.03.2025).
12. Shields D. AWS Security. Manning Publications Co. LLC, 2022. 312 p.
13. Helping Players Truly Own In-Game Assets Using AWS with Mystic Moose and Sequence. AWS Case Study. URL: https://aws.amazon.com/solutions/case-studies/mystic-moose-case-study/ (date of access: 22.03.2025).
14. Baumgartner S. TypeScript Cookbook: Real World Type-Level Programming. O'Reilly Media, Incorporated, 2023. 419 p.
15. Torres A. Leveraging Golang for Game Development and Operations. Riot Games Technology Blog. URL: https://technology.riotgames.com/news/leveraging-golang-game-development-and-operations (date of access: 25.03.2025).
16. Culkin J., Zazon M. AWS Cookbook: Recipes for Success on AWS. O'Reilly Media, Incorporated, 2022. 356 p.
17. The Node.js Book for Enterprise. Platformatic HQ. URL: https://www.platformatichq.com/reports/the-node-book (date of access: 05.04.2025).
18. Johnson R. The AWS Networking Handbook: A Practical Guide to Cloud Connectivity, Security, and Optimization. HiTeX Press, 2025. 375 p.
19. Brisals S., Hedger L. Serverless Development on AWS: Building Enterprise-Scale Serverless Solutions. O'Reilly Media, Incorporated, 2024. 498 p.
20. Saleh S. M., Madhavji N., Steinbacher J. A Systematic Literature Review of

Cloud-based CI/CD Security. In: Proceedings of the 20th International Conference on Web Information Systems and Technologies – WEBIST. 2024. P. 331–341. URL: https://www.scitepress.org/Papers/2024/130185/130185.pdf (date of access: 10.04.2025).

21. Peguero K., Cheng X. Electrolint and security of electron applications. High-Confidence Computing. 2021. Vol. 1, no. 2. Article 100032. URL: https://www.sciencedirect.com/science/article/pii/S2667295221000222 (date of access: 11.04.2025).
22. Wittig A., Wittig M. Amazon Web Services in Action, Third Edition: An in-Depth Guide to AWS. 3rd ed. Manning Publications Co. LLC, 2023. 552 p.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка системи розподіленої доставки цифрового контенту на прикладі гри Minecraft

Виконав студент 4 курсу
групи ПД-41
Стратович Михайло Ярославович
Керівник роботи
асистент кафедри ІПЗ Аброскін Юрій Юрійович

Київ – 2025

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

- **Мета роботи** - спрощення, масштабованість і підвищення гнучкості системи доставки цифрового контенту шляхом використання принципів хмарних технологій і мереж CDN.
- **Об’єкт дослідження -** процес доставки цифрового контенту користувачам у розподілених системах на прикладі гри Minecraft.
- **Предмет дослідження -** система автоматизованої доставки цифрового контенту.

ЗАДАЧІ ДИПЛОМНОЇ РОБОТИ

1. Проаналізувати існуючі підходи до розподіленої доставки цифрового контенту в ігрових системах, і виявити їх ключові особливості та недоліки.
2. Обгрунтувати вибір інструментів і технологій для реалізації системи розподіленої доставки цифрового контенту.
3. Сформувати функціональні та нефункціональні вимоги до системи доставки цифрового контенту.
4. Розробити архітектуру хмарної інфраструктури доставки цифрового контенту з використанням AWS-сервісів та Terraform.
5. Реалізувати механізм автоматичного завантаження і публікації цифрового контенту до Amazon S3 за допомогою GitHub Actions у межах CI/CD-процесу.
6. Реалізувати серверну логіку формування посилань на контент і передачі публічних ключів за допомогою AWS Lambda, API Gateway, Route 53, CloudFront і DynamoDB.
7. Розробити клієнтський застосунок для отримання контенту та перевірки його цілісності з використанням CloudFront та Electron.js.
8. Налаштувати контроль доступу до ресурсів через AWS IAM та впровадити захист від DDoS-атак за допомогою AWS WAF та AWS Shield.
9. Забезпечити доступ до системи через власне доменне ім'я, налаштувавши Route 53, alias-записи та HTTPS-сертифікати за допомогою AWS ACM.
10. Перевірити працездатність повного циклу доставки контенту: генерації посилань, отримання даних клієнтом, перевірки підписів і завантаження файлів через CDN.

3

АНАЛІЗ АНАЛОГІВ

Система	Основне призначення	Архітектура / Технології	Гнучкість і кастомізація	Метод доставки та кешування	Підтримка модів	Міжплатформність
Steam Client	Доставка ігор, оновлень та DLC через інтегровану CDN-платформу.	Моноліт з додатковими модулями, сервери Valve.	Мінімальна, закрита екосистема.	Власна CDN-мережа Steam з глобальними кеш-серверами (Steam Content Servers).	Часткова.	Так.
Epic Games Launcher	Доставка офіційного ігрового контенту (ігри, оновлення, івенти) через власну інфраструктуру.	Монолітна структура лаунчера з вбудованими службами Epic.	Низька, кастомізація обмежена.	Власна CDN-інфраструктура Epic (Epic CDN) із розподіленим кешуванням.	Відсутня.	Так.
CurseForge	Доставка модів, аддонів та користувацького контенту для відкритих ігор (Minecraft тощо).	Модульна інфраструктура з API, white-label UI, cloud cooking, CDN-платформа від Overwolf.	Висока: публічний API, кастомізація, інтеграція в інші ігри.	Хмарна інфраструктура CurseForge із кешуванням на CDN, підтримкою модпаки і версій.	Повна.	Так.
DUIKT Content Delivery	Доставка модифікацій, ресурсів і оновлень для Minecraft через автоматизовану хмарну систему.	Клієнт реалізований на Electron + TypeScript; взаємодія з AWS через API Gateway, Lambda, S3, DynamoDB, CloudFront.	Висока: open-source, інтеграція в інші ігри.	Доставка через Amazon CloudFront (CDN), зберігання на Amazon S3, кешування на edge-вузлах; валідація через підпис файлів.	Повна.	Так.

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

- Клієнтська частина завжди ініціює завантаження найактуальнішої версії ігрових ресурсів.
- Система повинна використовувати CDN (Amazon CloudFront) для швидкої доставки файлів з урахуванням географічного розташування користувача.
- Кожен файл має перевірятися за допомогою публічних ключів, щоб гарантувати, що він не був змінений або пошкоджений під час передачі.
- Після оновлення репозиторію контент автоматично завантажується за допомогою CI/CD до Amazon S3 і реєструється в базі метаданих.

Нефункціональні вимоги:

- Швидкість завантаження контенту має бути однаковою для всіх регіонів.
- Завантаження контенту повинно відбуватись з мінімальними затримками при великому навантаженні на систему.
- Весь трафік має йти через HTTPS, доступ до S3 обмежується через CloudFront OAI. Дані підписуються, а доступ до API захищено IAM політиками.
- Контент реплікується між зонами доступності, забезпечуючи стійкість до збоїв інфраструктури та втрати даних.

5

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ


AWS Lambda


Electron.js


Amazon DynamoDB

TypeScript

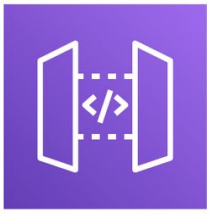

Amazon CloudFront


Amazon S3


Terraform

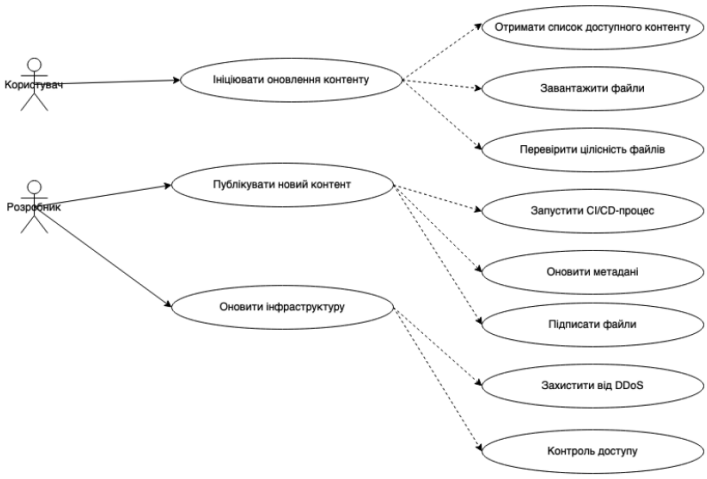

AWS IAM

Amazon Route 53


Amazon API Gateway

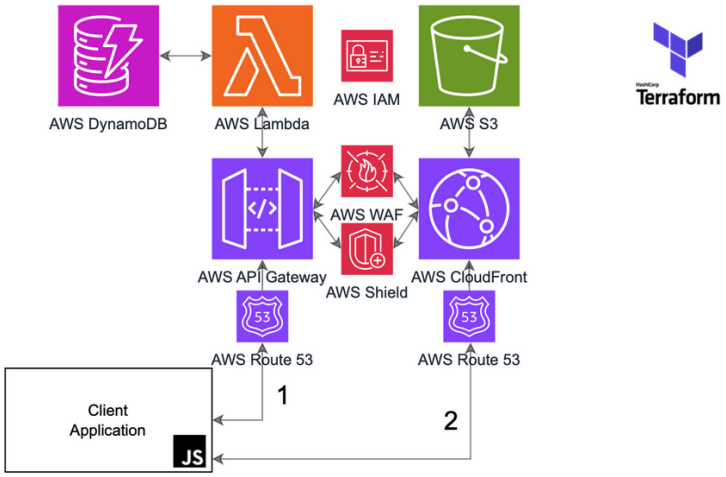
6

Діаграма варіантів використання



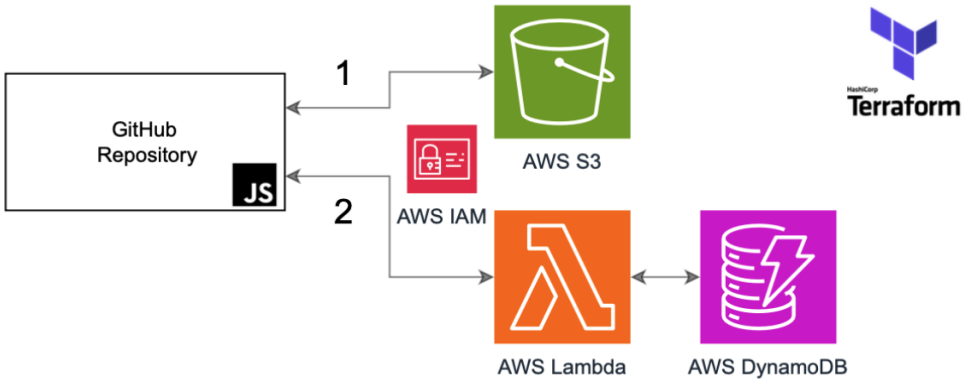
7

Схема доставки цифрового контенту користувачу



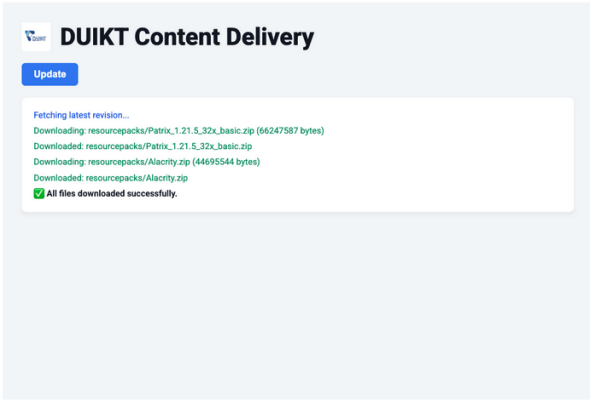
8

Схема архітектури завантаження цифрового контенту

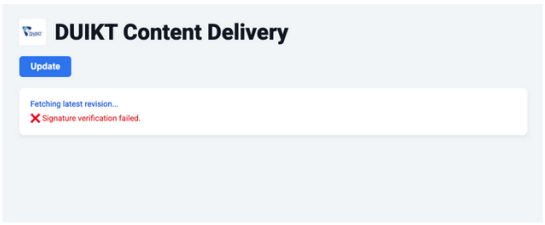


9

ЕКРАННІ ФОРМИ



Форма завантаження актуального контенту (Успіх)



Форма завантаження актуального контенту (Збій перевірки цілісності)

10

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Стратович М. Я., Аброскін Ю. Ю. Забезпечення безпеки даних у системі розподіленої доставки цифрового контенту на прикладі гри Minecraft. Матеріали XII Всеукраїнської науково-практичної конференції молодих учених «Інформаційні технології – 2025», 15.05.25, КСУ імені Бориса Грінченка, Київ, Україна с. 331 – 332.
2. Стратович М. Я., Аброскін Ю. Ю. Архітектурне рішення для системи розподіленої доставки цифрового контенту на прикладі гри Minecraft. // Матеріали VI Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». 24.04.2025, ДУІКТ, Київ, Україна, с. 450 – 454.

ВИСНОВКИ

1. Проведено аналіз сучасних підходів до доставки цифрового контенту в ігрових системах, виявлено їх обмеження щодо гнучкості, автоматизації та підтримки модифікацій.
2. Обґрунтовано вибір AWS-сервісів, Terraform, Electron.js та CI/CD як найдоцільніших інструментів для побудови незалежної системи доставки контенту.
3. Сформульовано функціональні та нефункціональні вимоги, орієнтовані на швидкість, безпеку, надійність і автоматизованість процесів.
4. Розроблено архітектуру хмарної інфраструктури з використанням AWS S3, CloudFront, Lambda, API Gateway та Terraform.
5. Реалізовано механізм автоматичного завантаження контенту до S3 за допомогою GitHub Actions як частину CI/CD-процесу.
6. Створено серверну логіку, що формує посилання на контент і підписи файлів через Lambda, API Gateway та DynamoDB.
7. Розроблено клієнтський застосунок на Electron.js, що отримує посилання, перевіряє підписи та завантажує файли через CloudFront.
8. Налаштовано контроль доступу до контенту за допомогою AWS IAM, а також впроваджено захист від атак через AWS WAF і AWS Shield.
9. Сконфігуровано власне доменне ім'я з використанням Route 53, ACM та alias-записів для забезпечення HTTPS-доступу.
10. Проведено тестування повного циклу доставки: генерація посилань, отримання контенту, перевірка підписів і завантаження через CDN пройшли успішно.

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

Код з invk-create-revision.ts файлу:

```
import {
  BinaryToTextEncoding,
  createSign,
  generateKeyPairSync,
} from 'node:crypto';
import {
  ConditionalCheckFailedException,
  DynamoDBClient,
  PutItemCommand,
} from '@aws-sdk/client-dynamodb';
import { Handler } from 'aws-lambda';
import { Env } from '../env.js';

const dynamodb = new DynamoDBClient({ region:
Env.AWS_REGION });

export const handler: Handler<{
  id: string;
  author: string;
  description: string;
  manifest: {
              hash: string;
              size: number;
  };
}> = async (event) => {
  console.debug(event);

  const keypair = generateKeypair({ encoding: 'base64'
});

  let revision: Revision = {
              id: event.id,
              author: event.author,
              description: event.description,
              publicKey: keypair.publicKey,
              privateKey: keypair.privateKey,
              createdAt: new Date().toISOString(),
  };

  await dynamodb
              .send(
              new PutItemCommand({
              TableName:
Env.AWS_DDB_TABLE_NAME,
              ConditionExpression:
'attribute_not_exists(id)',
              ReturnValuesOnConditionCheckFailure:
'ALL_OLD',
              Item: {
              id: { S: revision.id },
              author: { S: revision.author },
              description: { S: revision.description },
              publicKey: { S: revision.publicKey },
              privateKey: { S: revision.privateKey },
              createdAt: { S: revision.createdAt },
              },
              })
              )
              .catch((error) => {
              if (error instanceof
ConditionalCheckFailedException) {
              revision = {
              id: error.Item!['id']!.S!,
              author: error.Item!['author']!.S!,
              description: error.Item!['description']!.S!,
              publicKey: error.Item!['publicKey']!.S!,
              privateKey: error.Item!['privateKey']!.S!,
              createdAt: error.Item!['createdAt']!.S!,
              };
              return;
              }
              throw error;
              });

  const signature = generateSignature({
              data: `duikt-${revision.id}-
${event.manifest.hash}-${event.manifest.size}`,
              privateKey: revision.privateKey,
              privateKeyEncoding: 'base64',
              signatureEncoding: 'base64',
  });

  return { id: revision.id, signature };
};

type Revision = {
  id: string;
  author: string;
  description: string;
  publicKey: string;
  privateKey: string;
  createdAt: string;
};

function generateKeypair(options: { encoding:
BufferEncoding }): {
  publicKey: string;
  privateKey: string;
} {
  const keys = generateKeyPairSync('rsa', {
modulusLength: 2048 });

  return {
              publicKey: Buffer.from(
              keys.publicKey.export({ type: 'pkcs1',
format: 'pem' })
              ).toString(options.encoding),
              privateKey: Buffer.from(
```

```
        keys.privateKey.export({ type: 'pkcs1',
format: 'pem' })
        ).toString(options.encoding),
  };
}

function generateSignature(options: {
  data: string;
  privateKey: string;
  privateKeyEncoding: BufferEncoding;
  signatureEncoding: BinaryToTextEncoding;
}): string {
  return createSign('RSA-SHA256')
        .update(options.data)
        .sign(
        Buffer.from(options.privateKey,
options.privateKeyEncoding),
        options.signatureEncoding
        );
}
```

Код з http-get-revision.ts файлу:

```
import {
  DynamoDBClient,
  ScanCommand,
} from '@aws-sdk/client-dynamodb';
import { APIGatewayProxyHandlerV2 } from 'aws-
lambda';
import { Env } from '../env.js';

const dynamodb = new DynamoDBClient({ region:
Env.AWS_REGION });

export const handler: APIGatewayProxyHandlerV2 =
async (event) => {
  console.debug(event);

  const output = await dynamodb.send(
        new ScanCommand({
        TableName:
Env.AWS_DDB_TABLE_NAME,
        })
  );

  const revisions = output.Items!.map((item) => ({
        id: item['id']!.S,
        publicKey: item['publicKey']!.S,
        createdAt: item['createdAt']!.S,
  }));

  revisions.sort((a, b) => {
        const aDate = new Date(a['createdAt']!);
        const bDate = new Date(b['createdAt']!);
        return bDate.getTime() - aDate.getTime();
  });

  return {
        statusCode: 200,
        body: JSON.stringify({
        revision: revisions[0],
        }),
  };
};
```

Код з uploader/src/index.ts файлу:

```
import path from 'node:path';
import * as actions from '@actions/core';
import { InvokeCommand, LambdaClient } from
'@aws-sdk/client-lambda';
import { CreateMultipartUploadCommandInput,
PutObjectCommandInput, S3Client } from '@aws-
sdk/client-s3';
import { STSClient } from '@aws-sdk/client-sts';
import { NodeHttpHandler } from '@smithy/node-http-
handler';
import { S3SyncClient, TransferMonitor } from 's3-
sync-client';
import { Env } from './env.js';
import { assumeIamRoleWithWebIdentity } from
'./lib/aws.js';
import { FS } from './lib/fs.js';
actions.info('Resolving paths...');
const resourcesPath =
path.resolve(import.meta.dirname, '../../minecraft/');
const syncOutputPath =
path.resolve(import.meta.dirname, '.sync-output');
const assetsDirOutputFilesPath =
path.resolve(syncOutputPath, 'files');
actions.info('Copying resource directory...');
await FS.copyDir(resourcesPath,
assetsDirOutputFilesPath);
actions.info('Checking out files...');
const manifest = await
FS.checkoutFiles(assetsDirOutputFilesPath).then((files
) =>
Promise.all([FS.getFilesHashes(files),
FS.getFilesSizes(files)]).then(async ([hashes, sizes])
=> {
actions.info('Writing config with file hashes and
sizes...');
return await FS.writeConfig<Config>(syncOutputPath,
{
files: files.map((file, fileIndex) => ({
path:
FS.normalizePath(path.relative(assetsDirOutputFilesPa
th, file)),
hash: hashes[fileIndex]!,
size: sizes[fileIndex]!,
})),
}).then((config) => ({
path: path.relative(syncOutputPath, config.path),
hash: config.hash,
size: config.size,
}));
}),
```

```
);
const region = Env.AWS_REGION;
actions.info('Assuming IAM role with web identity...');
const credentials = await
assumeIamRoleWithWebIdentity(new STSClient({
region }), {
roleArn: Env.AWS_IAM_ASSUME_ROLE_ARN,
roleSessionName: 'github-to-aws-via-federated-oidc',
webIdentityToken: await
actions.getIDToken('sts.amazonaws.com'),
});
actions.info('Creating STS and S3 clients...');
const sts = new STSClient({ region, credentials });
const s3 = new S3Client({
region,
credentials: {
accessKeyId: credentials.accessKeyId,
secretAccessKey: credentials.secretAccessKey,
sessionToken: credentials.sessionToken!,
},
requestHandler: new NodeHttpHandler({
connectionTimeout: 30 * 1000,
socketTimeout: 30 * 1000,
}),
maxAttempts: 5,
});
const lambda = new LambdaClient({
region,
credentials: {
accessKeyId: credentials.accessKeyId,
secretAccessKey: credentials.secretAccessKey,
sessionToken: credentials.sessionToken!,
},
});
actions.info('Creating revision...');
const revision: Revision = await lambda
.send(
new InvokeCommand({
FunctionName:
Env.AWS_LAMBDA_CREATE_REVISION_FN_AR
N,
Payload: Buffer.from(
JSON.stringify({
id: Env.REVISION_ID,
author: Env.REVISION_AUTHOR,
description: Env.REVISION_DESCRIPTION,
manifest: {
hash: manifest.hash,
size: manifest.size,
},
}),
),
}),
)
.then(({ StatusCode, FunctionError, Payload }) => {
if (StatusCode !== 200 || FunctionError || !Payload) {
throw new Error(`Function invocation failed with
status code ${StatusCode}`, { cause: FunctionError });
}
return JSON.parse(Payload.transformToString());
});
actions.info('Revision created successfully.');
actions.info('Initializing S3 sync client...');
const s3Sync = new S3SyncClient({ client: s3 });
const options: {
s3BucketName: string;
s3ObjectPrefix: string;
inputPath: string;
syncPartSizeInMb: number;
syncConcurrency: number;
inputTransformer: <T extends
PutObjectCommandInput |
CreateMultipartUploadCommandInput>(
properties: Partial<T>,
) => Partial<T>;
} = {
s3BucketName: Env.AWS_S3_BUCKET_NAME,
s3ObjectPrefix:
`${Env.AWS_S3_OBJECT_PREFIX}/${Env.REVISI
ON_ID}`,
inputPath: syncOutputPath,
syncPartSizeInMb: 10,
syncConcurrency: 10,
inputTransformer: (properties) => {
actions.info(`Transforming input for
${properties.Key}`);
if (properties.Key === options.s3ObjectPrefix + '/' +
manifest.path) {
properties.Metadata ??= {};
properties.Metadata['duikt-signature'] =
revision.signature;
actions.info(`Added metadata to ${properties.Key}`);
}
return properties;
},
};
const upload = async (retries = 0): Promise<void> => {
if (retries > 5) {
throw new Error('Upload revision: MAX RETRIES
EXCEEDED');
}
actions.info(`Starting upload attempt #${retries +
1}...`);
const monitor = new TransferMonitor();
const interval = setInterval(() => {
const { count, size } = monitor.getStatus();
actions.info(`Uploaded ${count.current} parts of
${count.total} files (${size.current} of ${size.total}
bytes)`);
}, 2000);
try {
await s3Sync.sync(options.inputPath,
`s3://${options.s3BucketName}/${options.s3ObjectPre
fix}`, {
partSize: options.syncPartSizeInMb * 1024 * 1024,
maxConcurrentTransfers: options.syncConcurrency,
monitor,
commandInput: options.inputTransformer,
});
actions.info('Upload completed successfully.');
} catch (e) {
if (
e instanceof Error &&
(e.message.includes('timed out') ||
e.message.includes('socket hang up') ||
e.message.includes('ECONNRESET') ||
```

```
e.message.includes('EAI_AGAIN'))
) {
// biome-ignore lint/style/noParameterAssign:
retries++;
actions.info(`Upload revision: RETRYING
(#${retries}) due to error: ${e.message}`);
return upload(retries);
} else {
if (e instanceof Error) {
e.message = `Upload revision: FAILED due to error:
${e.message}`;
}
throw e;
}
} finally {
clearInterval(interval);
}
};
actions.info('Starting upload process...');
await upload();
type Config = {
files: ConfigFile[];
};
type ConfigFile = {
path: string;
hash: string;
size: number;
};
type Revision = {
id: string;
signature: string;
};
```

Код з iam-lambda.tf файлу:

```
resource "aws_iam_role" "lambda" {
 name               = "duikt-content-delivery-lambda"
 assume_role_policy =
data.aws_iam_policy_document.lambda-assume-
role.json
}

data "aws_iam_policy_document" "lambda-assume-
role" {
 statement {
              effect  = "Allow"
              actions = ["sts:AssumeRole"]

              principals {
              type        = "Service"
              identifiers = ["lambda.amazonaws.com"]
              }
 }
}

resource "aws_iam_policy" "lambda" {
 name   = aws_iam_role.lambda.name
 policy = data.aws_iam_policy_document.lambda.json
}

data "aws_iam_policy_document" "lambda" {
 statement {
              effect    = "Allow"
              actions   = [
              "logs:CreateLogStream",
              "logs:PutLogEvents"
              ]
              resources = [

              "arn:aws:logs:${data.aws_region.current.n
ame}:${data.aws_caller_identity.current.account_id}:l
og-group:/aws/lambda/*"
              ]
 }

 statement {
              effect    = "Allow"
              actions   = ["dynamodb:*"]
              resources = [module.dynamodb-
table.dynamodb_table_arn]
 }
}

resource "aws_iam_role_policy_attachment" "lambda"
{
 role       = aws_iam_role.lambda.name
 policy_arn = aws_iam_policy.lambda.arn
}
```

Код з s3.tf файлу:

```
resource "aws_s3_bucket" "this" {
bucket = "duikt-content-delivery"
lifecycle {
prevent_destroy = true
}
}
resource "aws_s3_bucket_policy" "this" {
bucket = aws_s3_bucket.this.id
policy = data.aws_iam_policy_document.s3.json
}
data "aws_iam_policy_document" "s3" {
statement {
effect = "Allow"
actions = ["s3:GetObject"]
resources = ["${aws_s3_bucket.this.arn}/*"]
principals {
type = "Service"
identifiers = ["cloudfront.amazonaws.com"]
} condition {
test = "ArnEquals"
variable = "aws:SourceArn"
values = [aws_cloudfront_distribution.this.arn]}}}
```