МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
КАФЕДРА КОМП'ЮТЕРИЗОВАНИХ СИСТЕМ ЗАХИСТУ ІНФОРМАЦІЇ

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри
____________ С.В. Казмірчук

«_____»____________ 2020 р.

ДИПЛОМНА РОБОТА

ВИПУСКНИКА ОСВІТНЬО СТУПЕНЯ
«МАГІСТР»

Тема: Система автентифікації складових мікросервісної архітектури під час їх взаємодії

Автор: **А.О. Поліщук**

Науковий керівник: к.т.н., доц. **О.В. Дубчак**

Нормоконтролер: к.т.н., доц. **О.В. Дубчак**

Київ 2020

ВСТУП

Актуальність. За даними експертного дослідження, оскільки компанії приносять більше бізнес-функцій через Інтернет-ініціативи, використання мікросервісної архітектури стає популярним, згідно з опитуванням 354 підприємств. А саме, з опитаних вже 63% перейшли і активно використовують її у своїх програмних продуктах.

"Покращення ефективності працівників", "Покращена робота з клієнтами / кінцевими користувачами", та "Економія витрат на інфраструктуру та інші інструменти розвитку" - три основні причини, що приносять користь бізнесу. Глобальне опитування було проведено в липні 2020 року і представляє погляди архітекторів програмного забезпечення, інженерних менеджерів та інших експертів з розробки додатків, у 51 країні і в 12 галузях.

За статистикою використання або планування переходу на мікросервісну архітектуру:

- 60% респондентів повідомляють, що їхня інженерна організація прийняла або планує прийняти архітектуру мікросервісів для досягнення більш швидкого часу для виходу на ринок нових продуктів і послуг, а 54% роблять це для підтримки цифрових перетворень і живлення додатків наступного покоління;
- 58% виконують або планують працювати між 10-49 мікро-обслуговуванням у виробництві, а 15% виконують або планують працювати більше 100.

Звортною стороною швидкого розповсюдження новітніх технологій є щоденно зростаюча кількість кіберзагроз та складність кібератак, що вимагає прискіпливої уваги до вдосконалення існуючих методів протидії ним та створення нових розробок.

Метою дипломної роботи є створення прототипної системи аутентифікації, а саме реалізація серверної частини для обробки внутрішніх запитів системи та створення відповідних сервісів для прикладу обробки та валідації.

Для досягнення даної мети, потрібно розв'язати ряд задач:

- дослідити методи автентифікації компонентів в мікросервісній архітектурі;
- розробити систему аутентифікації компонентів в мікросервісній архітектурі на базі Amazon Services;
- імітувати реальне середовище та провести тестування даного компоненту.

Об'єкт дослідження: процес аутентифікації компонентів системи в мікросервісній архітектурі

Предмет дослідження: архітектурні методи, патерни та протоколи для реалізації програмних програмних засобів та інфраструктурних компонентів в мікросервісній архітектурі.

Методи дослідження:

- метод порівняльного аналізу монолітної та мікросервісної архітектури, існуючих методів аутентифікації компонентів в мікросервісній архітектурі;
- метод об'єктно-орієнтованого програмування.

Слід зазначити, що серед науковців та практиків, які проводять дослідження в напрямку мікросервісної архітектури та удосконалення захисних функцій, переважать зарубіжні, серед яких Мартін Фаулер [1], Лен Басс [2], Роджер С. Прессман [3], Карл І. Віггерс, Джой Бітті [4].

Галузь застосування. Дана система міжсервісної аутентифікації під час їх комунікації може використовуватися у великих корпоративних програмних продуктах та високонавантажених системах, в яких для підтримання стабільної роботи системи потребується горизонтальне

розширення систем, тобто запуск додаткових серверів для обробки усіх запитів користувачів. Дана система в буде використовуватися для зменшення імовірності несанкціонованого доступу до інформаційної системи і централізації процесу автентифікації в одному закритому для доступу ззовні системи місці.

Практична цінність роботи полягає у проведенні аналізу монолітної та мікросервісної архітектур, дослідженні методів автентифікації в мікросервісній архітектурі під час взаємодії їх компонентів, що дозволило зробити висновок про наявність в існуючих варіантах, попри переваги, досить вагомих недоліків. Запропоновано створення додаткового рівня безпеки на програмному рівні та на рівні інфраструктури інформаційної системи. Для реалізації використано мову програмування Java, популярний фреймворк Spring та відповідні підмодулі даного фреймворку, а також можливості хмарних рішень на базі сервісу Amazon Web Services. Створена система автентифікації може використовуватися в масштабних інформаційних системах різноманітного спрямування.

Апробація роботи. Результати досліджень розглянуто на Міжнародній науково-практичній конференції «Наука і Інновації - 2020» (Жовтень 2020 р., м. Пшемисль, Польща).

Розділ 1. СУЧАСНІ МЕТОДИ АВТЕНТИФІКАЦІЇ В МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ

У зв'язку з постійним розширення функцій та постійним збільшенням систем складності та навантаження на систему, з'явилася необхідність у нових підходах до проектування та розробки інформаційних систем для задоволення потреб користувачів. Для цього почали використовувати розподілені системи з горизонтальним розширенням, що призвело до створення нового підходу та патерну мікросевісної архітектури.

1.1. Мікросервісна архітектура

Мікросервісна архітектура (МСА) або мікросервіси - це патерн розробки архітектури застосунку, при якому застосунок створюють як сукупність сервісів, кожен з яких функціонує у своєму власному процесі та середовищі й комунікує з рештою сервісів за допомогою таких механізмів, як HTTP, черг команд тощо. Подібні сервіси створюються на основі потреб користувача і розгортаються незалежно один від одного, з використанням автоматизованого середовища. Сервіси можуть бути написані різними мовами програмування з використанням різних типів сховищ для даних.

Зі зростанням складності систем, навантаження на окремі частини та необхідності відповідати вимогам бізнесу, виникає потреба в нових та гнучких методах проектування програмного забезпечення та інфраструктури для його розгортання.

В основі МСА лежить створення окремих застосунків, на відміну від сервісно-орієнтованої архітектури, в основі якої лежить модульний підхід

під час розробки програмного забезпечення, який заснований на використанні розподілених компонентів забезпечених стандартними інтерфейсами для взаємодії застосунків визначених протоколом.

На сьогоднішній день більшість масштабних проектів активно використовують МСА під час розробки. Їх порівнюють з потужними застосунком, який працює, як єдине ціле. Наприклад, застосунок на рівні підприємства базовими рівнями є: дружній інтерфейс (HTML сторінки, тощо), база даних (надалі БД) та сервер. В даному випадку серверна частина відповідає за обробку HTTP запитів та оновлення даних в базі, а потім відправляє новостворені або оновленні дані в браузер або інший тип клієнтського програмного забезпечення. При монолітній архітектурі все більше користувачів мають проблеми через зростаючу складність розгортання та підтримку застосунків, розроблених, як єдине ціле, особливо з урахуванням того, що застосунки розгортають із використанням хмарних рішень. Із розгортанням за часом, стає важче зберігати модульну структуру, змінювати логіку одного модуля, що, у свою чергу, впливає на програмний код інших модулів, тому масштабувати доводиться застосунок у цілому. Але при МСА все спрощується, оскільки обробка даних зосереджена в одному місці, то при зміні бізнес-логіки змінюється і розгортається тільки серверна частина, окрім того можливості багатьох мов програмування дозволяють робити поділ прикладного застосунку на модулі, класи, функції тощо. А для масштабування можна горизонтально запустити декілька серверів, навантаження на які будуть регулюватися за допомогою балансувальника навантаження. [5]

Попри всі переваги, залишилися проблеми, які ще досі потребують вирішення, такі як: коректне виділення сервісів та розподіл бізнес-логіки, стандартизація транспорту і протоколу взаємодії між сервісами.[6]

Порівняльна схема архітектур наведена на рис 1.1.

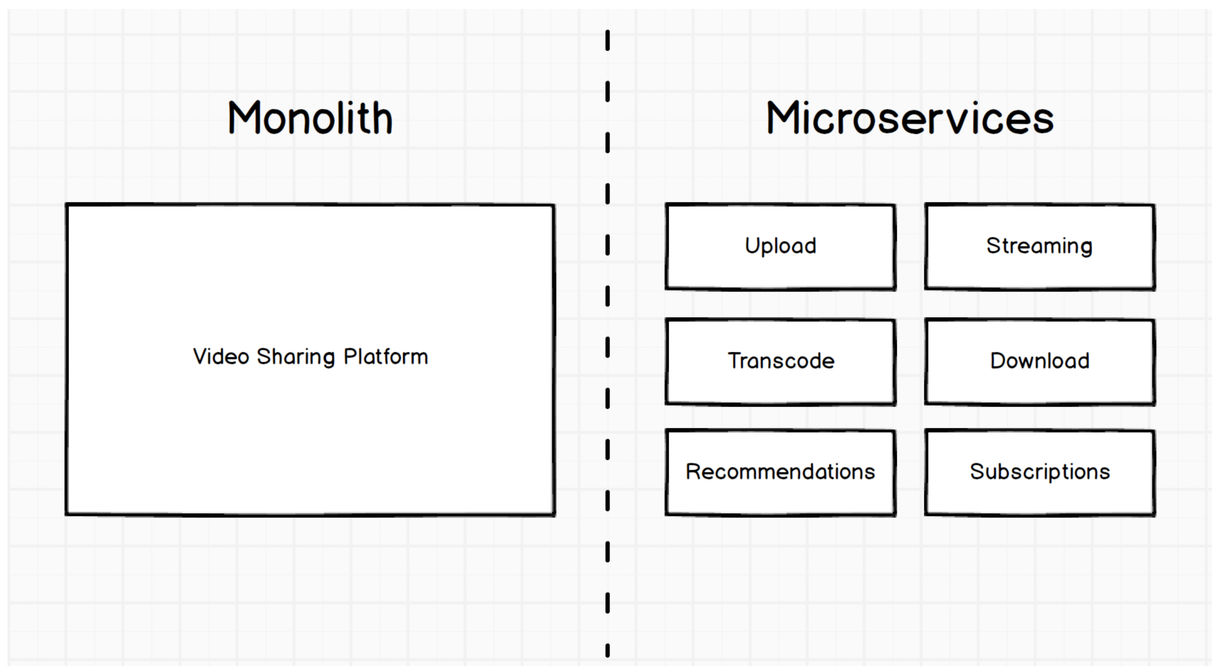

Рис 1.1 Порівняльна схема монолітної і мікросервісної архітектук [6]

Стандартна структура монолітного застосунку, зазвичай, має наступні складові: інтерфейс – бізнес-логіка – дані.

Мікросервіси є архітектурним шаблоном, в якому всі сервіси мають відповідати наступним вимогам: не потребують значної кількості розробників (ресурсність); один сервіс призначений для вирішення одного завдання (сфокусованість); зміни в одному сервісі не впливають на інші (слабопов’язаність); компонент/клас створюється з урахуванням усіх методів вирішення завдання (погодженість).

Для з’ясування переваг та недоліків існуючих архітектур проведено порівняльний аналіз їхніх характеристик за критеріями, що наведено в табл. 1.1. Дане порівняння базується на шести основних критеріях, а саме: складність розгортання додатку; подальша підтримка і розробка; надійність і стійкість до збоїв; можливість збільшення потужності системи та здатність до масштабування; кількість коштів, потрібних для розробки; складність початкової розробки на етапі старту.

Таблиця 1.1

Порівняльна характеристика монолітної та мікросервісної архітектур

Критерії/ архітектура	Монолітна	Мікросервісна
Розгортання	Монолітні програми дозволяють встановлювати розгортання один раз, а потім просто налаштовувати його на основі поточних змін. У той же час, однак, є лише одна точка відмови під час розгортання, і якщо все піде не так, може зламатися весь застосунок.	При розгортанні мікросервісів, основна складність полягає в тому, що для кожного сервіса це потрібно робити окремо, що потребує часу, окрім того потрібно вирішувати проблему з оркестратором для управління. Тим не менш, при невдачі з одним сервісом, не зламаються інші.
Підтримка	З часом важко підтримується через складність системи	Через розділеність дозволяє легко підтримувати окремі компоненти
Надійність	Будь-яка несправність системи призведе до зупинення роботи всього додатку.	При відмові одного сервіса, вся інша система буде працювати і надалі.
Масштабування	Складно масштабується, оскільки зміна або додавання компонента може мати ефект на весь додаток.	Легко масштабується, оскільки кожен компонент незалежний.
Ціна	При початковій розробці цей варіант є дешевший, але при подальшому розширенні та масштабуванні, підтримка стає дорожчою.	При початковій розробці цей варіант є дорожчий, але при подальшому розширенні та масштабуванні, підтримка стає дешевшою.
Складність розробки	Для малих застосунків цей варіант є доступний, оскільки не потребує великих зусиль при початку	Для великих корпоративних застосунків цей варінт є кращий, але тим не менш виникає проблема з

	розробки.	розділенням сервісів та відповідальності.

1.2 Методи міжкомпонентної комунікації в мікросервісній архітектурі

Одною з ключових складових МСА являються методи комунікації в мікросервісній архітектурі, як зовні системи так і між її компонентами. Метод комунікації може залежати від потреб, які виникають в системі задля забезпечення стабільної та надійної роботи, збереження консистентності даних, потужностей системи при обслуговування тої чи іншої кількості користувачів. Розглянемо 3 ключові методи:

- REST API (HTTP запити),
- з допомогою повідомлень та команд,
- з допомогою подій.

1.2.1 HTTP запити

Одним за найпопулярніших варіантів комунікації в мікросервісній архітектурі являють звичайне REST API з допомогою HTTP запитів. Даний метод є популярним так як він є один з найпростіших методів під час розробки.

На рис. 1.2 можна побачити приклад логіки програми, що побудована на основі мікросервісної архітектури і використовує REST API для комунікації, для прикладу використана система організації польотів та подорожей для пасажирів. Розглянувши рисунок детально можна побачити, що пасажир використовуючи смартфон, як клієнтський додаток для взаємодії з системої робить запит (POST /trips) на сервіс для бронювання поїздки (Trip Managment). В цей час система для перевірки існування даного користувача в системі робить запит (GET /passengers/<<passengerId>>) до

внутрішнього сервісу, що відповідає за користувачів та пасажирів (Passenger Management). Далі у випадку, якщо пасажира не існує система видасть помилку, на запит сервісу управління подорожами, то і клієнтська програма також отримує помилку. В успішному випадку відповідно створюється бронювання на подорож.

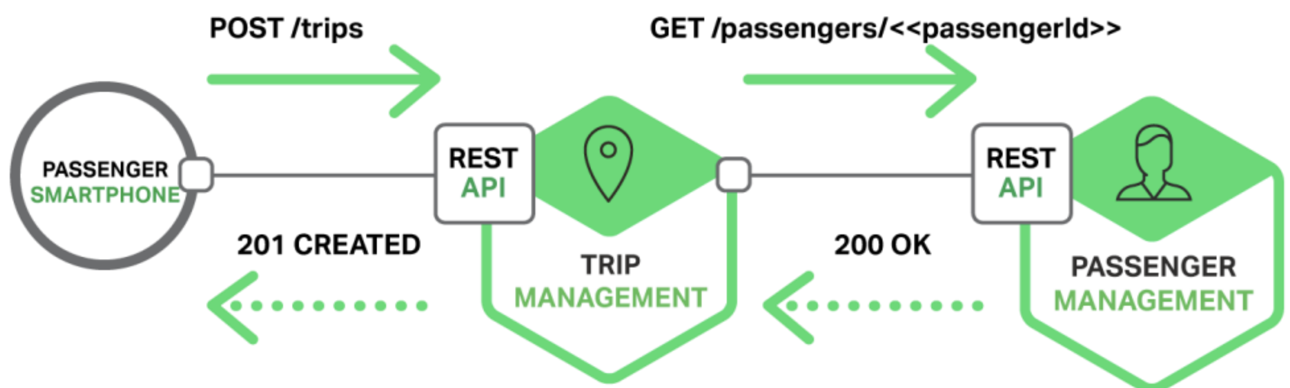

Рис 1.2 Приклад побудови логіки на основі REST API та HTTP запитів[1]

Даний метод комунікації базується на схемі запит-відповідь, тобто на кожен запит сервіс очікує отримати відповідь. В даному підході є як і плюси так і мінуси.

З одного боку система буде отримувати результат виконання запитів відразу по завершенню. Це дає змогу продовжити роботу вже з виконаним результатом, і зменшує певну складність системи і допомагає побудувати просту і прямолінійну логіку.

З іншого боку дана прямолінійність системи може спровокувати проблеми з тривалістю виконання та загальним навантаженням під час виконання логіки, оскільки на кожен запит може блокуватися потік в системі створюючи надлишкове навантаження, і в разі якщо запит виконується довгий час то і відповідно затримки в системі, що може вивести її з ладу. Окрім того виникає складність під час розгортання нових версій REST API, оскільки в разі якщо не має зворотної сумістності сервісів,

то це може призвести до, знову ж таки, несправності та виведення системи з ладу.

1.2.2 З допомогою повідомлень.

Інший популярний тип комунікації являється комунікація з допомогою повідомлень.

На рис. 1.3 та 1.4 можна побачити приклад систем, що використовують повідомлення в якості механізму комунікацій між сервісами. Наприклад, на сервіс подорожей прийшов запит від певного клієнта на створення нової поїздки. Відповідно для цього необхідно також додати дану поїздку в профіль, зробити оплату тощо. Сервіс що відповідає за поїздки відправляє повідомлення в спеціальний компонент системи (Dispatcher, Message broker), що відповідає за зберігання та видачу даних повідомлень іншим сервісам на запит або ж з допомогою різних протоколів та технологій, сам інформую інші сервіси про певне повідомлення. Відповідно певний сервіс в якому закладена логіка обробки даного повідомлення, виконує інструкції пов’язані з цим повідомленням. І в залежності від реалізації системи сервіс-ініціатор може заблокувати ж свою роботу допоки інший сервіс не надішле повідомлення про успішне закінчення виконання інструкцій. Або відправивши повідомлення не буде чекати на закінчення виконання іншого сервісу.

З одного боку, даний метод дозволяє побудувати досить таки потужну і стійку систему, яка буде продовжувати роботу в разі виникнення проблем під час роботи одного з компонентів системи. Окрім того зменшується залежність компонентів та складність під час підтримки сумісності систем, так як єдине, що потрібно буде підтримувати це формат повідомлення. Завдяки сервісу обробки повідомлення, вони зберігаються в черзі таким

чином не втрачаються під час роботи системи і відповідно в разі помилки під час обробки, дане повідомлення може не видалятися і потім знову обробитися, даючи змогу системі мати консистентний стан.

З іншого боку даний метод потребує додатковий інфраструктурний компонент, який необхідно підтримувати і під час збільшення навантаження потрібно слідкувати, щоб дана черга не вийшла з ладу. Наступна складність полягає у налаштуванні запит-відповідь комунікації, тобто отримання результату виконання того чи іншого сервісу. Складність полягає в тому, що необхідно налаштувати канал зворотного зв'язку та і відповідно передачу результату, так як тут може бути декілька варіантів даного процесу.

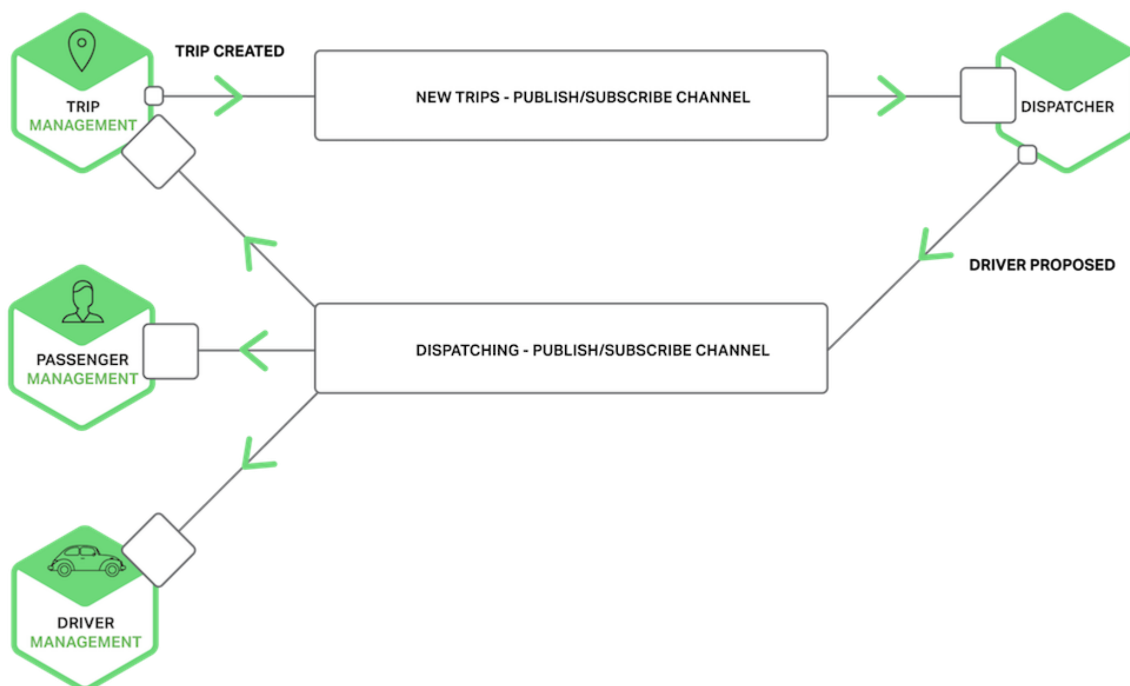

Рис 1.3 Приклад побудови логіки на основі повідомлень[2]

1.2.3 З допомогою подій

Ще один популярний метод комунікації в мікросервісній архітектурі являються події.

Розглянемо рис. 1.4. Даний метод з першого погляду нічим не відрізняється від попереднього з технічної точки зору, оскільки використовується майже така сама схема та компоненти. Тим не менш даний метод має дещо іншу концепцію. Вона полягає в тому, що якщо повідомлення може використовуватися для виконання запиту на дані і відповідно очікування результату, то події лише інформують про виконання певної операції або отримання запиту.

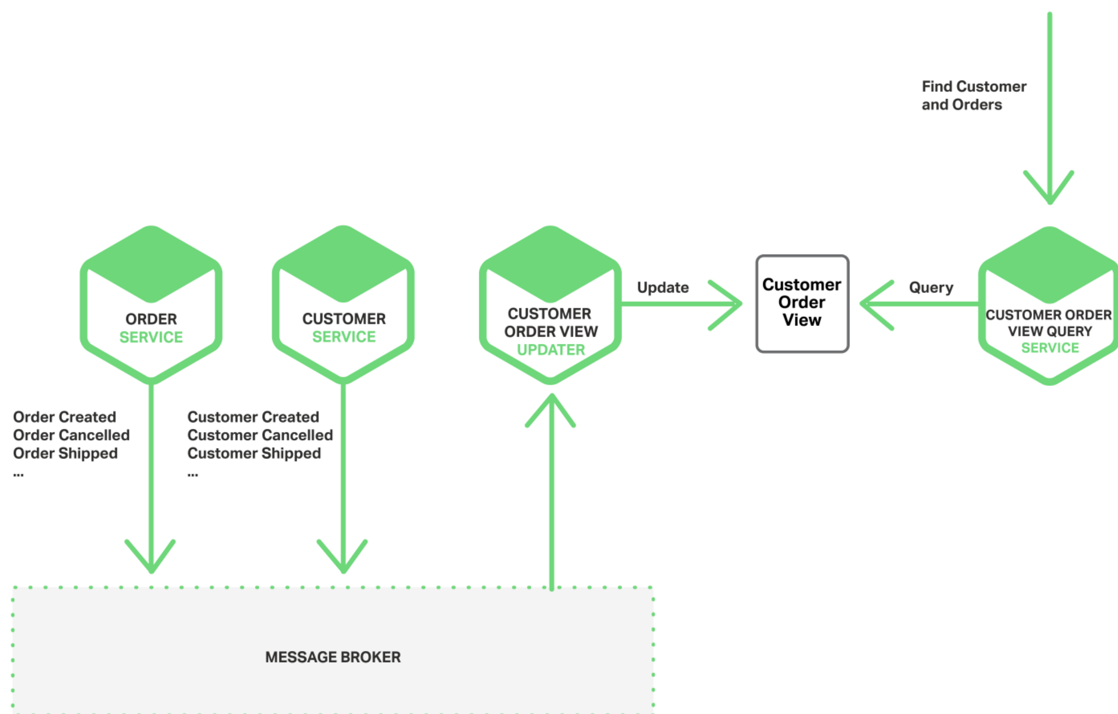

Рис 1.4 Приклад побудови логіки на основі подій[3]

З одного боку, в порівнянні з попереднім методом, події дозволяють максимально знизити залежність компонентів системи один від одного та залежність результату виконання.

З іншого гостріше постає проблема у отриманні результату виконання тої чи іншої операції, так як знову ж таки потрібно налаштувати двосторонню комунікацію.

Розглянувши найпопулярніші методи комунікації, можна зробити висновок, що кожен з цих підходів має свої недоліки так і переваги. Відповідно визначити найкращий з цих підходів не можна.

У великих корпоративних системах, часто використовуються комбінація даних методів. Приклад такої комбінації можна побачити на рис 1.5. Можна побачити що використовуються як і звичайні REST API, так і повідомлення та події - кожен з цих підходів вирішує певне завдання для бізнес-логіки системи використовуючи свої позитивні якості.[3]

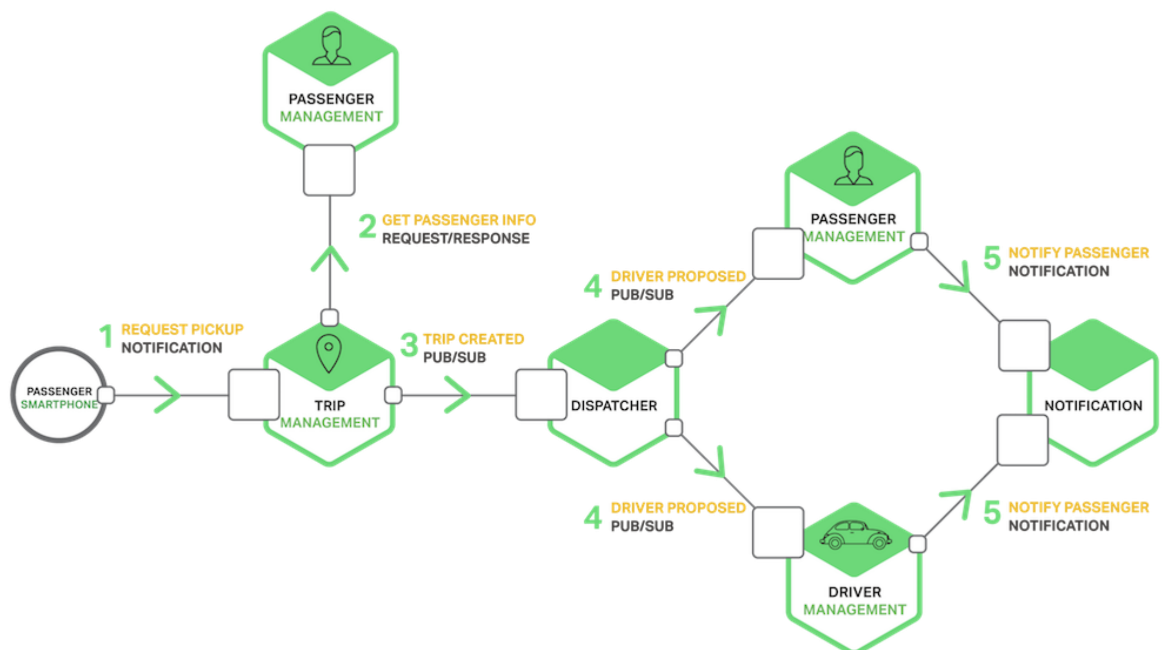

Рис 1.5 Приклад побудови логіки на основі подій[3]

1.3 Існуючі методи автентифікації в мікросервісній архітектурі

Розглянемо існуючі методи автентифікації мікросервісів під час їх взаємодії.

1.3.1 Використання JWT(Json Web Token)

JWT (Json Web Token) - це відкритий стандарт RFC 7519, за допомогою якого визначається компактний і самовизначний спосіб для

безпечного передавання інформації між учасниками передачі даних у вигляді JSON- об’єкту. Дана інформація може бути підтверджена і довірена через підтвердження її за допомогою цифрового підпису. Даний токен може бути підписаний за допомогою секретного (алгоритм HMAC) або ж за допомогою приватного та публічного ключів, використовуючи RSA чи ECDSA. [9]

Розглянемо структуру даного токену. Він складається з 3 частин: заголовків, корисної частини та цифрового підпису, які розділені між собою крапкою.

Рис. 1.6. Схема JWT, xxxx - заголовки токену , yyyy - корисна частина в якій містяться дані, zzzzz - цифровий підпис

Заголовок містить в собі дані про даний токен та має певні поля (рис.1.7.). Поле *alg* - це поле, що містить в собі алгоритм шифрування або цифрового підпису даного токену. Дане поле є обов’язковим, оскільки надалі буде використовуватися при дешифруванні або підтвердженні підпису. А також поля *typ* та *cty,* де в першому вказується тип токену, який використовується при змішуванні токенів з іншими об’єктами, що мають JOSE заголовки. Друге поле, відповідно, містить в собі дані про тип контенту, який зберігається в корисній частині. [4]

```
{
 "typ": "JWT",
 "alg": "HS256"
}
```

Рис 1.7. Приклад заголовку в токені

Корисна частина - це частина токена, яка зберігає в собі дані про користувача, сесію, права користувача тощо (рия.1.8). Окрім того, є поля,

такі як *iss, sub, aud, exp* та декілька інших, що містять в собі додаткову інформації в тілі корисної частини.


```
{
    "username": "andrew130499@gmail.com"
}
```


Рис 1.8. Приклад корисної частини в токені

А також, остання частина, яка є необов’язковою, - це частина з цифровим підписом для підтвердження достовірності та цілісності корисної частини (рис.1.9.).

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VybmFtZSI6ImFuZHJldzEzMDQ5OUBnbWFpbC5jb20iLCJqdGkiOiIzZTU0ZmRiOS1hYTBkLTQzYmUtYWQ3My1kNDI5NTA2MjgyZmMiLCJpYXQiOjE1NTgyNzY2MDAsImV4cCI6MTU1ODI4MDIwMH0.B3IkfnFVqMI_vZkglxftggNl5blnT56FfqixEF4BZA4

Рис 1.9. Приклад JWT

Розглянемо детальніше приклад JWT. Червоним кольором до крапки показано зашифрований заголовок, значення якого можна побачити на рис 1.4. Помаранчевим кольором визначено корисну частину даного токену, в якій зберігається дані про користувача значення яких можна побачити на рис. 1.5.

Попри, здається, переваги даного підходу, є багато мінусів і вразливостей, якими зловмисники можуть в подальшому скористатися для отримання несанкціонованого доступу. Одною з таких є видалення значення цифрового підпису, що дає змогу змінювати значення заголовків та корисної частини.

Окрім того, розповсєдженою загрозою є зміна алгоритму шифрування до значення *none,* що дає змогу також отримати доступ до ресурсів. А

також, при використанні алгоритму RS256, є загроза зміни алгоритму на HS256, тобто зміни асиметричного шифрування на симетричне. Це, відповідно, призведе до ситуації, коли сервер буде намагатися розшифрувати дані тим самим публічним ключем, яким їх зашифровували. А якщо зловмисник дізнався значення публічного ключа, то він може підписувати власні дані на сервер, що призведе до несанкціонованого доступу.[10]

1.3.2 Взаємна аутентифікації по TLS

Сам TLS також називається одностороннім TLS, здебільшого тому, що він допомагає клієнту визначити сервер, з яким він спілкується, але не навпаки. Двосторонній TLS, або взаємний TLS, заповнює цю прогалину, допомагаючи клієнту та серверу ідентифікувати себе один з одним. Подібно до того, як клієнт знає, з яким сервером він розмовляє в односторонньому TLS, при взаємному TLS (mTLS), сервер знає клієнта, з яким розмовляє.Приклад можна розглянути на рис. 2 Для участі в каналі зв'язку, захищеному mTLS, і сервер, і клієнт повинні мати дійсні сертифікати, і кожна сторона повинна довіряти видавцю відповідного сертифіката. Іншими словами, коли mTLS використовується для захисту зв'язку між двома мікросервісами, кожна мікрослужба може законно ідентифікувати, з ким вона спілкується, крім досягнення конфіденційності та цілісності даних, що передаються між двома мікросервісами.

Взаємна автентифікація TLS (mTLS) гарантує безпеку та надійність трафіку в обох напрямках між клієнтом та сервером. Це дозволяє запитам, які не входять в систему із постачальником ідентифікаційних даних (наприклад, пристроями IoT), продемонструвати, що вони можуть отримати доступ до певного ресурсу. Аутентифікація клієнтських сертифікатів також є другим рівнем безпеки для членів команди, які одночасно входять в систему із постачальником ідентифікаційних даних і представляють дійсний сертифікат клієнта.[5]

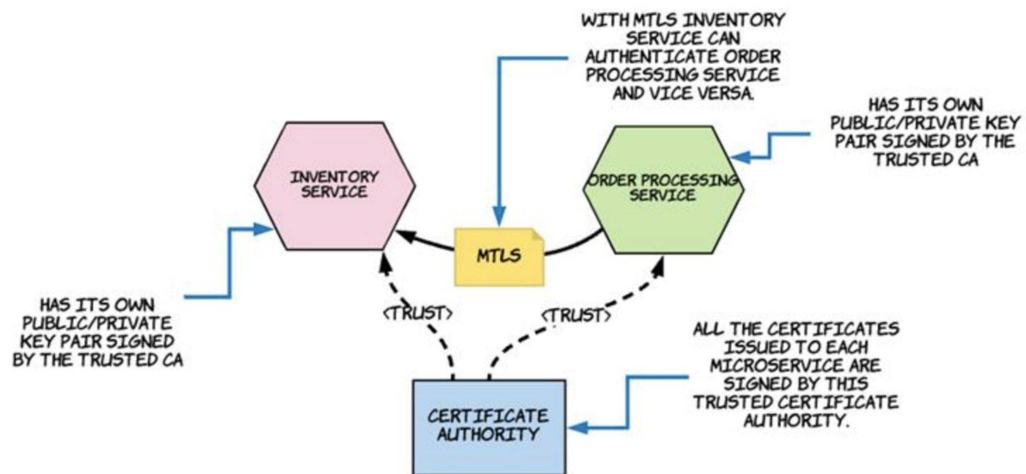

Рис 1.10. Взаємна аутентифікація по TLS з єдиним власником сертифікатів[5]

1.4 Модель загроз

Розглянемо модель загроз, які можуть потенційно виникнути в МСА (рис.1.11).

Загрози в будь - якій автоматизованій системі можуть виникнути як у внутрішньому, так і у зовнішньому середовищі системи.

Загрози в МСА можуть бути націлені на порушення наступних властивостей інформації:

- доступності;
- цілісності даних;
- спостережності та керованості системи.

Джерелом порушення можуть бути як внутрішні, так і зовнішні загрози. Загрози можуть бути здійснені шляхом:

- прослуховуванням каналів зв’язку в системі;
- перенавантаженням системи, що призведе до подальшої відмови. [1]

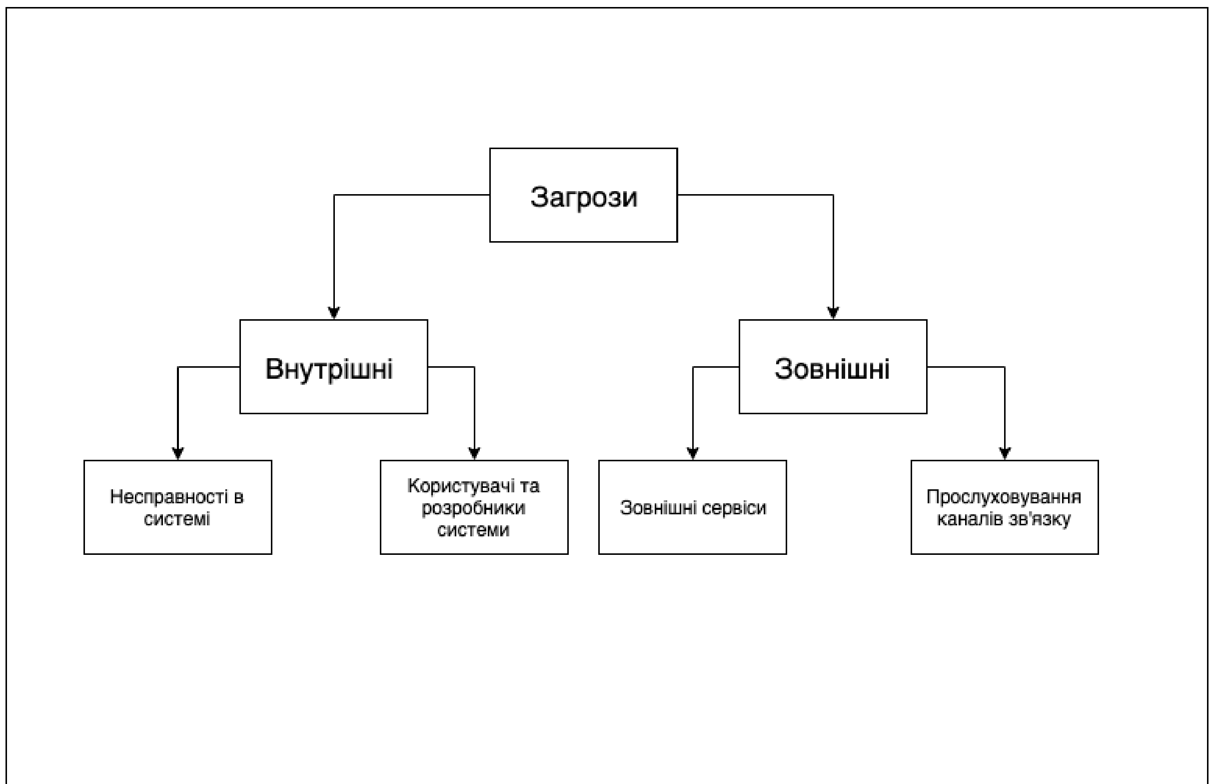

Рис. 1.11. Модель загроз

1.5 Модель порушника

Як відомо [15], порушник - це юридична або фізична особа, яка здійснює неправомірні дії відносно до системи, задля нашкодження, отримання доступу тощо. Модель порушника наведено на рис.1.12.

До внутрішніх порушників можна віднести наступні категорії співробітників:

- розробники системи;
- інженерний склад.

До зовнішніх порушників можна віднести:

- зовнішні сервіси;
- користувачі.

Мета порушника:

- виведення з ладу системи та її компонентів для нанесення матеріальної шкоди підприємству;
- отримання доступу до стратегічно важливої та конфіденційної інформації;
- зміна даних в системі та налаштування.

Усі дії порушника можуть здійснюватися шляхом наступних засобів:

- програмних;
- апаратних;

Дії порушника:

- отримання віддаленого доступу до одного з компонентів системи;
- виведення компонентів з ладу для погіршення доступності даних.

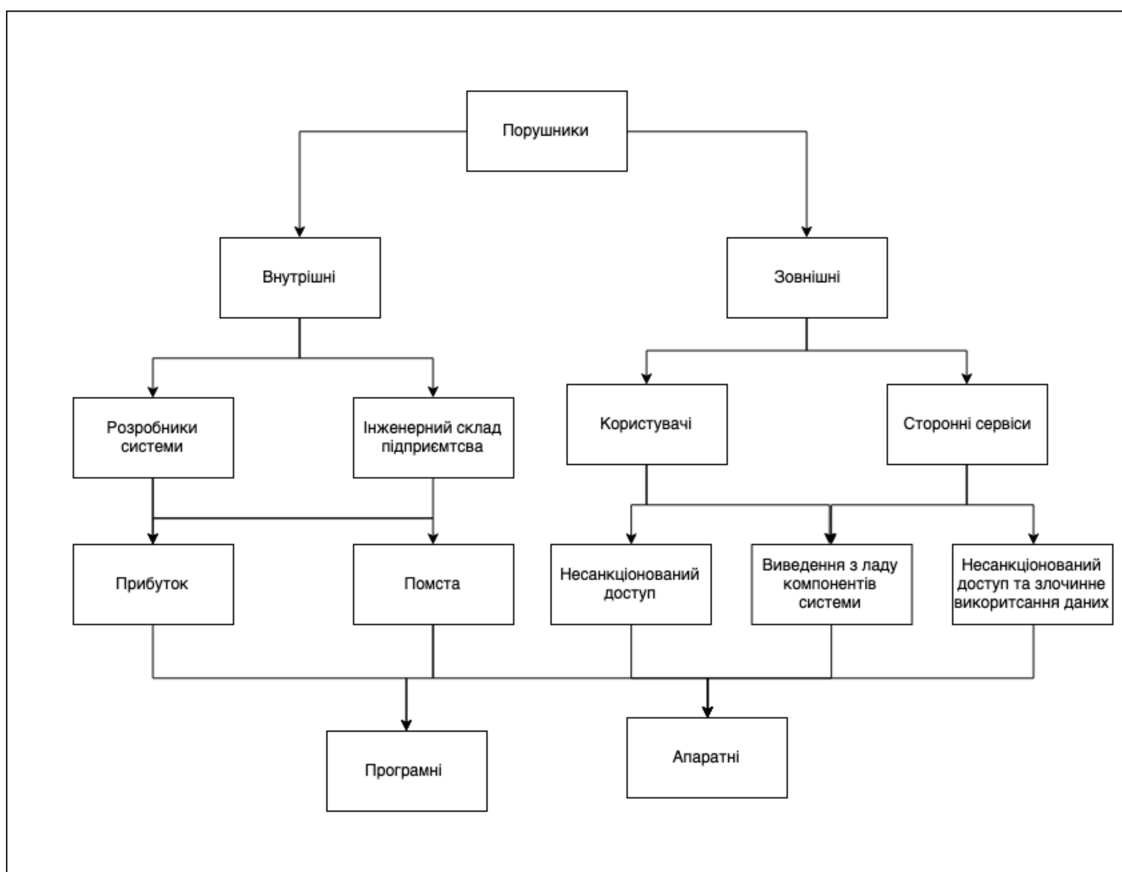

Рис. 1.12 Модель порушника

1.6 Правове забезпечення захисту інформації та процесу автентифікації

Основним напрямком в системі захисту інформації є правовий захист інформації, актуальність якого зростає в умовах побудови інформаційного суспільства.

Як відомо [15], правова форма захисту інформації (ЗІ) – це ЗІ, який базується на використанні статей Конституції та законів будь-якої держави, положень цивільного і кримінального кодексів та інших нормативно-правових документів в галузі інформатики, інформаційних відносин та захисту інформації. Вона регламентує права і обов'язки суб'єктів інформаційних відносин, правовий статус органів, технічних засобів і способів ЗІ і є підгрунтям для створення морально-етичних норм в області ЗІ.

Правовий ЗІ визнаний як на міжнародному (міжнародні договори, угоди, конвенції, декларації тощо), так і на державному рівні. На державному рівні правовий ЗІ регулюється державними та відомчими нормативно-правовими актами.

Система законодавчих актів та розроблених на їх підґрунті нормативних та організаційно-розпорядчих документів повинна забезпечувати організацію ефективного нагляду за їх виконанням з боку правоохоронних органів та реалізацію засобів судового захисту та відповідальності суб'єктів інформаційних відносин. До цієї системи можна віднести і морально-етичні норми поведінки, які склалися традиційно або складаються в міру поширення обчислювальних засобів в суспільстві. [15]

Основу державного регулювання суспільних відносин в сфері технічного ЗІ становить Конституція України. До спеціального

законодавства, що врегульовує дану галузь, належать закони України: “Про інформацію”, “Про державну таємницю”, “Про захист інформації в автоматизованих системах”, “Про Національну систему конфіденційного зв’язку”, “Про Концепцію Національної програми інформатизації”, “Про науково - технічну інформацію” та інші. У тому числі у сфері правоохоронної діяльності — такі, що визначають компетенцію та функції окремих державних органів влади: Служби безпеки України, поліції, прокуратури тощо (закони України “Про Службу безпеки України”, “Про поліцію” та інші).[15] Деякі вимоги щодо технічного захисту окремих видів інформації містяться у законодавстві України про оперативно-розшукову діяльність, про організаційно-правові основи боротьби з організованою злочинністю.

Свідченням про позитивні зрушення в області нормативно-правового регулювання є введення в дію Закону України “Про основні засади забезпечення кібербезпеки” від 05.10.2017 р. [16] Даним законом встановлено наступні важливі терміни, які є новими в правовому полі України, такі як: кібербезпека, кіберзагроза, кіберпростір, кіберінцидент, кібертероризм та багато інших. Дія закону поширюється тільки на об’єкти критичної структури, а також комунікаційні системи усіх форм власності, які обробляють національні інформаційні ресурси та використовують у сферах місцевого самоврядування, правоохоронних органів, оборони та інших сферах державної влади.

В інформаційних системах (ІС) є можливість приховано отримати доступ до інформаційних архівів, що концентруються в одному місці у великих об’ємах. Крім того, з’явилася можливість дистанційного отримання інформації. Тому для ЗІ використовуються принципово нові методи та засоби, розроблені із врахуванням цінності інформації, умов роботи, технічних та програмних можливостей ЕОМ та інших засобів збирання, передачі та обробки даних. Особливі захисні заходи необхідні у

випадках, коли ресурси ЕОМ використовуються кількома користувачами в багатопрограмному режимі та режимі розділення часу. Тут виникає ряд правових проблем, пов'язаних з масивами інформації, які представляють собою суспільну та національну цінність, а їх зміст – національну таємницю. Використання такої інформації не за призначенням може привести до значних збитків як суспільству в цілому, так і окремій особі. [16]

Окремо слід звернути увагу на правові аспекти ЗІ, які можуть виникнути при недостатньо продуманому чи зловмисному використанні електронно-обчислювальної техніки. Грунтуючись на [16], до них слід віднести:

- правові питання захисту масивів інформації від спотворень та встановлення юридичної відповідальності за порушення збереженості інформації;
- юридичні та технічні питання захисту інформації від НСД до неї, які виключають можливість неправомірного її використання;
- встановлення юридично закріплених норм та методів захисту авторських прав та пріоритетів розробників програмного продукту;
- розробка заходів з надання юридичної сили електронним документам та засобів, які перешкоджають фальсифікації таких документів;
- правовий захист інтересів експертів, які передають свої знання до фондів банків даних;
- встановлення правових норм та юридичної відповідальності за використання електронно-обчислювальних засобів в особистих цілях, що суперечать інтересам інших осіб та суспільства і можуть завдати їм шкоду.

Відсутність належної реєстрації та контролю робіт, низька трудова і виробнича дисципліна персоналу, доступ сторонніх осіб до обчислювальних ресурсів створюють умови для зловживань і ускладнюють

їх виявлення. У кожному обчислювальному центрі прийнято встановлювати та строго дотримуватися регламенту доступу в різні службові приміщення для різних категорій співробітників. Ступінь ЗІ від неправомірного доступу та протизаконних дій залежить від якості розробки організаційних заходів, направлених на унеможливлення:

- доступу сторонніх до апаратури обробки інформації;
- безконтрольного виносу персоналом різноманітних носіїв інформації;
- несанкціонованого введення даних в пам'ять, зміни чи витирання інформації, що зберігається в пам'яті;
- незаконного користування системами обробки інформації та отриманими даними;
- доступу до систем обробки інформації з використанням саморобних пристроїв;
- несанкціонована передача даних по каналах зв'язку із інформаційно-обчислювального центру;
- безконтрольне введення даних в систему;
- обробка даних без відповідної вимоги замовника;
- несанкціоноване зчитування, зміна чи знищення даних в процесі їх передачі чи транспортування носіїв інформації. [15]

Як визначено у [15], метою захисту інформації є:

- запобігання витоку, викраденню, втраті, спотворенню, підробці інформації;
- запобігання загрозам безпеки особи, суспільства, держави;
- запобігання несанкціонованим діям по знищенню, модифікації, спотворенню, копіюванню, блокуванню інформації;
- запобігання іншим формам незаконного втручання в інформаційні ресурси та інформаційні системи;

- забезпечення правового режиму документованої інформації, як об’єкта власності;
- захист конституційних прав громадян на зберігання особистої таємниці та конфіденційності персональних даних, доступних в ІС;
- зберігання державної таємниці, конфіденційності документованої інформації відповідно до законодавства;
- гарантування прав суб’єктів в інформаційних процесах та при розробці, виробництві і використанні ІС, технологій та засобів їх забезпечення.

Основоположними нормативно-правовими актами, якими регламентуються засади розвитку схем та механізмів автентифікації в Україні є низка законів України, серед яких закони “Про інформацію”, “Про захист персональних даних”, “Про доступ до публічної інформації”, “Про захист інформації в інформаційно-телекомунікаційних системах”. [15]

Окремо необхідно звернути увагу на термін “персональні дані”, який наведено у Законі України “Про захист персональних даних” [17], за яким визначено, що це сукупність відомостей про фізичну особу, яка ідентифікована або може бути ідентифікована. У цьому понятті тлумачиться визначення “ідентифікація особи”. Це поняття та його зміст є ключовим при розгляді питань електронної ідентифікації та автентифікації в ІС. Ще заплутанішою є ситуація з описом процесу автентифікації осіб як користувачів ІС у тих випадках, коли дія відбувається в автоматичному режимі.

Відмінними від термінів “фізична особа” та “юридична особа”, що застосовуються у “нецифровому полі”, є визначення “користувача інформації в системі”. Згідно Закону України “Про захист інформації в інформаційно-телекомунікаційних системах” [15] – це фізична або юридична особа, яка в установленому законодавством порядку отримала право доступу до інформації в системі. Таким чином, зазначеним

документом описаний механізм ідентифікації окремої категорії осіб - користувачів автоматизованих (інформаційних) систем (підписувачів).

У чинному законодавстві Україні сфера електронної ідентифікації та автентифікації не є системно урегульованою. Наразі ведеться робота з реформування законодавства, зокрема, до Верховної Ради України подано до повторного розгляду у другому читанні Проект Закону "Про електронні довірчі послуги" [18]. Закон покликаний внести зміни у сфері використання інфраструктури відкритих ключів та надання електронних довірчих послуг з урахуванням досвіду Європейського Союзу, розбудови єдиного простору довіри на основі системи електронних довірчих послуг, визнання в Україні електронних довірчих послуг, які надаються іноземними постачальниками електронних довірчих послуг, що забезпечить активний розвиток транскордонного співробітництва та інтеграцію України у світовий електронний інформаційний простір.

1.7 Висновки до розділу

Отже, МСА на сьогоднішній день знаходиться на піку популярності. Багато розробників та інженерів використовують даний підхід при проектуванні та розробленні своїх систем. Одною з переваг даного підходу є чітке розділення сфери відповідальності в системі.

Окрім того, мікросервіси дозволяють використовувати різні технології, а саме мови програмування, бібліотеки, фреймворки, операційні системи, тощо при побудові даної системи. Це дає волю розробникам в плані вибору технологій і не обмежує їх при використанні різних бібліотек.

Слід зазначити, що даний підхід дозволяє витримувати великий потік даних та запитів у системі, що дає змогу будувати ефективні

високонавантажені системи. А також забезпечити надійність функціонування системи при порушення роботи одного із сервісів.

Попри переваги використання МСА в розробленні застосунків, щодня розробники зустрічаються з проблемами різного роду при безпосередній побудові системи. Одною з таких проблем є НСД, для протидії якому необхідна розробка надійного процесу автентифікації, а саме побудування ефективної системи із забезпеченням відповідного рівня захищеності даних.

На сьогоднішній день проводяться дослідження та публікуються наукові праці, переважно зарубіжних авторів, з теми МСА та процесу автентифікації в ній, як і для користувачів так і для сервісів в середині системи. Попри значну кількість матеріалів, семінарів та дискусій з приводу цієї теми, єдиного рішення не існує, оскільки кожен експерт, підприємство, компанія тощо пропонує власний варіант вирішення даної проблеми, що часто не задовольняє іншим сервісам.

Проаналізувавши сучасні варіанти, можна зробити висновок, що усі вони мають право на існування, але не є ідеальними чи універсальними, а тільки покривають зону власної відповідальності. Тому в даній дипломній роботі запропоновано більш універсальний варіант автентифікації компонентів в МСА під час їх взаємодії.

Проаналізувавши законодавство України в питанні автентифікації користувача ІС, можна зробити висновок, що нормативно-правова база держави не надає повної картини вимог до процесу автентифікації. Таким чином, нормативно-правова база має бути допрацьована і удосконалена з урахуванням вимог міжнародних стандартів щодо подальшого забезпечення надійного процесу автентифікації.

Розділ 2. ОПИС ТА РЕАЛІЗАЦІЯ АВТОРСЬКОГО ПРОГРАМНОГО МОДУЛЯ АВТЕНТИФІКАЦІЇ В МСА

Для досягнення поставленої мети пропонується авторський варіант реалізації системи автентифікації компонентів МСА. Суть полягає в пропозиції використання центрального комбінації уже існуючих методів і використання найкращих характеристик кожного з них, а саме створення реєстру компонентів системи з публічними ключами шифрування та використання JWT токенів для аутентифікації та відповідного створення фреймворку, для синхронізації з реєстром і відповідно налаштування безпечної комунікації.

Для реалізації використано наступні засоби:

- мова програмування – Java;
- набір бібліотек та фреймворків Spring;
- Liquibase для міграції бази;
- Docker та Amazon ECS для запуску даної системи.
- Amazon SQS для організації комунікації через події та повідомлення.

2.1 Засоби технічної реалізації програмного модуля

Для розуміння процесу створення програмного модля слід докладніше розглянути технічні засоби, використання яких надало змогу досягти поставленої мети.

2.1.1 Мова програмування

Для реалізації програмного модуля автентифікації в МСА було використано мову програмування Java.

Java є мовою програмування загального призначення, яка є класовою, хоча і не чистою об'єктно-орієнтованою мовою [19]. Розроблена з метою мати якомога менше залежностей реалізації. Вона призначена для того, щоб розробники програм “писали один раз, запускали де-небудь” (WORA - write once - run anywhere) [20], а це означає, що скомпільований код Java може працювати на всіх платформах, які підтримують Java без необхідності перекомпіляції. Програми Java зазвичай компілюються до “байт-коду”, який може працювати на будь-якій віртуальній машині Java (JVM), незалежно від базової архітектури комп'ютера. Синтаксис Java схожий на синтаксис мов C і C++, але він має менше можливостей низького рівня, ніж будь-який з них. Станом на 2018 рік, Java була однією з найпопулярніших мов програмування, що використовуються відповідно до GitHub [21], особливо для клієнт-серверних веб-додатків, з 9 мільйонами розробників.

Java була спочатку розроблена Джеймсом Госліном у Sun Microsystems (який з тих пір був придбаний Oracle) і випущена в 1995 році, як основний компонент платформи Java Sun Microsystems. Оригінальні та довідкові реалізації Java-компіляторів, JVM і бібліотек класів спочатку були випущені Sun під власними ліцензіями. Станом на травень 2007 року, відповідно до специфікацій Java Community Process, Sun повторно ліцензувала більшість своїх технологій Java за ліцензією GNU General Public License. [22] Тим часом інші компанії розробили альтернативні реалізації даних технологій Sun, такі як компілятор GNU для Java (компілятор bytecode), GNU Classpath (стандартні бібліотеки) і IcedTea-Web (браузерний додаток для аплетів).

Як відомо [23], остання версія - Java SE 12, випущена в березні 2019 року. Оскільки Java 9 більше не підтримується, Oracle радить своїм

користувачам “негайно перейти” до Java 12. Oracle випустила останнє публічне оновлення для застарілого Java 8 LTS, яке є безкоштовним, для комерційного використання, у січні 2019 року. Java 8 буде підтримуватися публічними оновленнями для особистого користування принаймні до грудня 2020 року. Oracle та інші “настійно рекомендують видалити старі версії Java” через серйозні ризики щодо невирішеності питань безпеки. Розширена підтримка Oracle для Java 6 завершилася в грудні 2018 року.

Однією з цілей дизайну Java є портативність, що означає, що програми, написані для платформи Java, повинні працювати аналогічно на будь-якій комбінації апаратних засобів і операційної системи з відповідною підтримкою часу виконання. Це досягається шляхом компіляції коду мови Java до проміжного представлення, яке називається байт-кодом Java, а не безпосередньо до машинного коду, специфічного для архітектури. Інструкції байт-коду Java аналогічні машинному коду, але вони призначені для виконання JVM, написаною спеціально для хост-обладнання. Кінцеві користувачі зазвичай використовують середовище виконання Java (JRE), встановлене на власній машині для окремих програм Java або веб-браузера для аплетів Java.

Стандартні бібліотеки надають загальний спосіб доступу до функцій, характерних для хоста, таких як графіка, багатопоточність та мережа.

Використання універсального байт-коду робить перенесення простим. Однак накладні витрати на інтерпретацію байт-коду в машинні інструкції, зроблені інтерпретованими програмами, майже завжди виконуються повільніше, ніж рідні виконувальні файли. Компілятори Just-in-Time (JiT), які компілюють коди байтів до машинного коду під час виконання, були введені на ранній стадії. Сама Java є незалежною від платформи і адаптується до певної платформи, на якій вона повинна працювати, віртуальною машиною Java, яка переводить байт-код Java у машинну мову платформи. [24]

Java використовує автоматичний збірник сміття (ЗС) для керування пам'яттю в життєвому циклі об'єкта. Програміст визначає, коли створюються об'єкти, а виконання Java відповідає за відновлення пам'яті, коли об'єкти більше не використовуються. Після того, як не залишиться жодних посилань на об'єкт, недоступна пам'ять може бути звільнена автоматично за допомогою ЗС. Щось подібне до витоку пам'яті все ще може відбуватися, якщо код програміста містить посилання на об'єкт, який більше не потрібний. Якщо викликаються методи для неіснуючого об'єкта, викидається "виняток з нульовим покажчиком". [25]

Однією з ідей, які стоять за Java-моделлю автоматичного керування пам'яттю, є те, що програмістам можна не турбуватися з приводу необхідності ручного керування пам'яттю. У деяких мовах пам'ять для створення об'єктів неявно виділяється на стеку або явно виділяється і звільняється з купи. В останньому випадку відповідальність за управління пам'яттю покладається на програміста. Якщо програма не звільняє об'єкт, відбувається витік пам'яті. Якщо програма намагається отримати доступ або звільнити пам'ять, яка вже була звільнена, результат є невизначеним і важко передбачуваним, і програма, ймовірно, стане нестабільною або аварійною. Це може бути частково усунуто за допомогою розумних покажчиків, але вони додають накладні витрати та складність. Зауважимо, що ЗС своїми діями не запобігає "логічним" витокам пам'яті, тобто тим, де пам'ять все ще посилається, але ніколи не використовується.

ЗС може спрацювати в будь-який час. В ідеалі, це відбуватиметься, коли програма не працює. Гарантується, що він буде спрацьовувати, якщо на купі недостатньо вільної пам'яті, щоб виділити новий об'єкт; це може призвести до тимчасового припинення програми. Явне управління пам'яттю в Java неможливе. [26]

Java не підтримує арифметику покажчиків стилів C / C++, коли адреси об'єктів можуть бути арифметично маніпульовані (наприклад, шляхом

додавання або віднімання зміщення). Це дозволяє ЗС переміщувати об'єкти, на які посилаються, і забезпечує безпеку та безпечність типу.

Як і в C++, і деяких інших об'єктно-орієнтованих мовах, змінні примітивних типів даних Java або зберігаються безпосередньо в полях (для об'єктів) або в стеку (для методів), а не в купі, як це звичайно справедливо для непримітивних даних типів. Це було свідоме рішення дизайнерів Java з причин продуктивності.

Java містить кілька типів ЗС. За замовчуванням HotSpot використовує ЗС паралельного скидання [27]. Однак, є також кілька інших ЗС, які можна використовувати для керування купою. Для 90% додатків на Java достатньо ЗС Concurrent Mark-Sweep (CMS). Oracle має на меті замінити CMS на ЗС "Сміття-перший" (G1). [28]

Вирішивши проблему управління пам'яттю, програміст не звільняється від тягару обробки належним чином інших видів ресурсів, таких як мережеві або підключення до бази даних, ручки файлів і т.д., особливо за наявності виключень. Парадоксально, але наявність ЗС зменшила необхідність використання явного методу деструкції в класах, що ускладнює управління цими ресурсами. [26]

2.1.2 Бібліотеки та інші допоміжні інструменти

Spring Framework. Як відомо [29], Spring Framework (або коротко Spring) - універсальний фреймворк з відкритим вихідним кодом для Java-платформ. Також існує форк для платформ .NET Framework, названий Spring.NET.

Перша версія була написана Родом Джонсоном, який вперше опублікував свою книгу «Expert One-on-One Java EE Design and Development» (Wrox Press, жовтень 2002 року).

Фреймворк був вперше випущений під ліцензією Apache 2.0 в липні 2003 року. Spring 2.0 був опублікований в жовтні 2006 року, Spring 2.5 -

вересень 2007, Spring 3.0 - в грудні 2009, і Spring 3.1 - в грудні 2011. Теперішня версія - Spring 5.1.2.

Незважаючи на те, що Spring не забезпечив яку-небудь конкретну модель програмування, він став широко розповсюдженим в Java-спільноті, головним чином, як альтернатива і заміна моделей Enterprise JavaBeans. Spring надає велику свободу Java-розробникам у проектах. Окрім того, Spring надає добре задокументовані і легкі у використанні рішення проблем, які створюються в додатках корпоративного масштабу.

Між тим, особливість ядра Spring можна використати в будь-якій Java - програмі, і існує безліч доповнень і удосконалень для побудови веб-додатків на платформі Java Enterprise. Тому Spring отримав велику популярність і визнається розробниками як стратегічно важливий фреймворк.

Як зазначено у [29], Spring забезпечує вирішення багатьох завдань, з якими зустрічаються Java-розробники і організації, які хочуть створити ІС, що базується на платформі Java. Через широку функціональність досить важко визначити найбільш значні структурні елементи, з яких він складається. В Spring не все це пов'язано з платформою Java Enterprise, незважаючи на його масштабну інтеграцію, яка є важливим підґрунтям його популярності.

Spring, напевно, є найвідомішим, як джерело розширень (функцій), необхідних для ефективної розробки бізнес-програмного забезпечення, що є домінуючими в промисловості. Ще одна його перевага полягає в тому, що він увів функціональні можливості, які були раніше недоступні, а сьогодні - найбільш популярні методи розробки, навіть поза межами платформи Java.

Цей фреймворк пропонує послідовну модель і робить її приємною до більшості типових програм, які вже створені на основі платформ Java. Вважається, що Spring реалізує модель розробки, засновану на кращих

стандартах промисловості, і робить її доступною в багатьох областях Java. [29]

Spring Boot. Spring boot - один з основних модулів стеку фреймворка Spring. З його допомогою значно спрощується створення *.jar файлу в проекті, а також сам старт додатку. [30]

Слід зазначити, що всередині Spring Boot має в собі усі базові залежності для Spring. Окрім того, задля старту веб-додатків на Java використовується веб-сервери, такі як Apache Tomcat, Jetty та інші. При використання звичайного контексту Spring потрібно власноруч налаштовувати та запускати даний веб-сервер на сервер з програмою. Spring Boot вирішує цю проблему просто, помістивши в собі Apache Tomcat, що значно полегшує запуск даного додатку для розробників. Так, для запуску досить написати код, приклад якого наведено на рис. 2.1.

```
@SpringBootApplication
public class Main {
    public static void main(String[] args) {
        SpringApplication.run(Main.class, args);

    }
}
```

Рис. 2.1 Приклад коду для запуску веб додатку Spring Boot

Spring Data. Як зазначено у [31], Spring Data - додатковий зручний механізм для взаємодії із сутностями бази даних, організації їх в репозиторії, вилучення даних, зміна, в яких випадках для цього буде достатньо оголосити інтерфейс і метод в ньому, без імплементації.

Основне поняття в Spring Data - це репозиторій. Це кілька інтерфейсів, які використовують JPA Entity для взаємодії з нею. Так, наприклад, інтерфейс, зображений фрагментом коду на рис. 2.2, забезпечує основні операції із пошуку, збереження, видалення даних, тобто звичайними CRUD-операціями.

```
@Repository
public interface ClientRepository extends JpaRepository<Client, Long>
```

Рис. 2.2 Фрагмент коду для створення репозиторію

Якщо того переліку, що надає інтерфейс, не досить для взаємодії з сутністю, то можна прямо розширити базовий інтерфейс для своєї сутності, доповнити його своїми методами запитів і виконувати операції, як наведено на рис. 2.3.

```
@Repository
public interface ClientRepository extends JpaRepository<Client, Long> {
    Optional<Client> findByClientIdAndClientSecret(String clientId, String clientSecret);
}
```

Рис. 2.3 Фрагмент коду для створення розширення репозиторію

Запити до суті можна будувати прямо з імені методу. Для цього використовується механізм префіксів “find...By”, “read ... By”, “query ... By”, “count ... By”, і “get ... By”, далі від префікса методу починає розбирання решти. Вступна пропозиція може містити додаткові вирази, наприклад, Distinct. Далі перший «By» діє як роздільник, щоб вказати початок фактичних критеріїв. Можна визначити умови для властивостей сутностей і об'єднати їх за допомогою “And” і “Or”. [31]

2.1.3 Середовище запуску та подальшої роботи системи

Для запуску даної системи будуть використовуватися такі інструменти як Docker, для створення власних образів системи та запуску контейнерів, а також оркестратор контейнерів який використовується для управління контейнерами.

Образ вдає із себе файлову систему з параметрами використовуваними при запуску. Він не має стану і ніколи не змінюється. А контейнер є запущеним екземпляром образу. Коли ви виконуєте команду запуску, Docker Engine:

- перевіряє чи існує даний образ;
- викачує образ в хмарного сховища або ж бере його з локальногож
- завантажує образ в контейнер і запускає його.

Залежно від того, як образ був зібраний, контейнер може виконати просту команду і потім завершитися

Завдяки малому розміру і орієнтованості на додаток контейнери ідеально підходять для середовищ з гнучкою настроюванням доставки і архітектур, заснованих на мікрослужбах. Але в середовищі з контейнерами і мікрослужбамі можуть існувати сотні і тисячі компонентів, за якими потрібно стежити. Ви можете вручну керувати кількома десятками віртуальних машин і фізичних серверів. Але контейнерної середовищем промислового масштабу не можна нормально управляти до без засобів автоматизації. Це завдання потрібно передати оркестратор, тобто спеціальний процес, який дозволяє автоматизувати велику кількість контейнерів, а також керувати ними і взаємодією між ними.

Служба Amazon Elastic Container Service (Amazon ECS) - це надзвичайно масштабована, швидка служба управління контейнерами, яка дозволяє легко запускати, зупиняти та керувати контейнерами в кластері. Ваші контейнери визначені у визначенні завдання, яке ви використовуєте для запуску окремих завдань або завдань у службі. У цьому контексті служба - це конфігурація, яка дозволяє одночасно запускати та підтримувати певну кількість завдань у кластері. Ви можете запускати свої завдання та служби на безсерверній інфраструктурі, якою керує AWS Fargate. Крім того, для більшого контролю над своєю інфраструктурою ви можете запускати свої завдання та служби на кластері екземплярів Amazon EC2, якими ви керуєте.[20]

Amazon ECS дозволяє запускати та зупиняти додатки на основі контейнерів за допомогою простих викликів API. Ви також можете отримати стан свого кластера з централізованої служби та мати доступ до багатьох знайомих функцій Amazon EC2.[20]

Ви можете запланувати розміщення контейнерів у вашому кластері на основі ваших потреб у ресурсах, політики ізоляції та вимог щодо

доступності. За допомогою Amazon ECS вам не потрібно керувати власними системами управління кластерами та управління конфігурацією або турбуватися про масштабування інфраструктури управління.[20]

Amazon ECS можна використовувати для створення послідовного досвіду збірки та розгортання, управління та масштабування робочих навантажень пакетного та вилучення-перетворення-навантаження та побудови складних архітектур додатків на моделі мікросервісів.

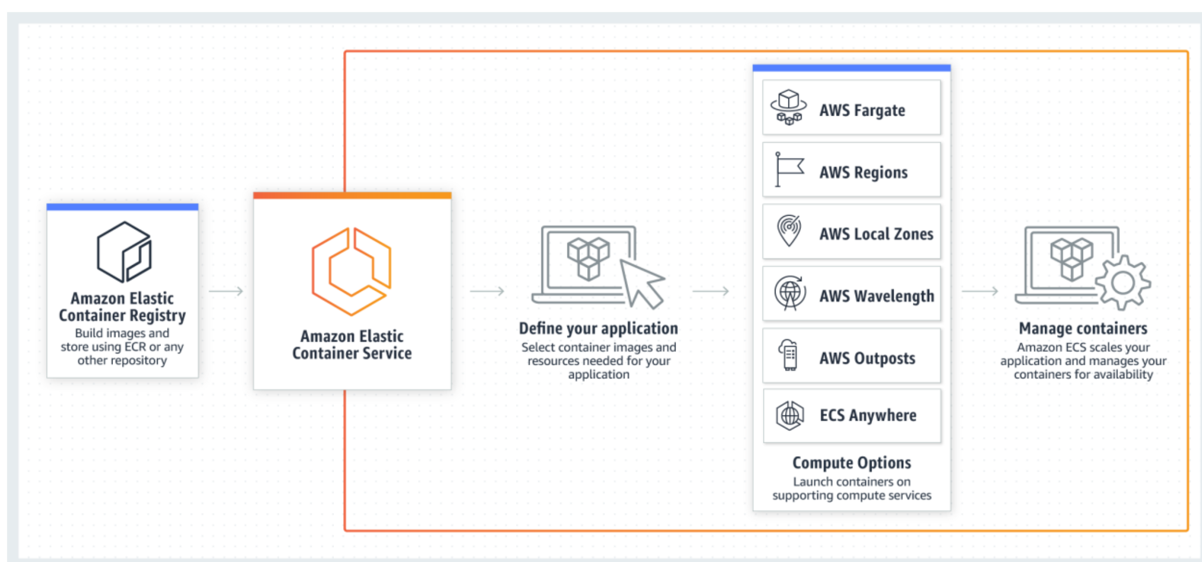

Рис. 2.4 Приклад роботи Amazon ECS[20]

2.2 Опис авторської системи аутентифікації компонентів

За основу розробки системи аутентифікації компонентів МСА було взято відразу декілька підходів та відповідна їх комбінації, а саме JWT токен з JWKS для генерації токенів доступу, центральний сервіс аутентифікації, який буде використовуватися в якості регістра сервісів.

Загальний алгоритм дій можна побачити на рис. 2.5 та 2.6.

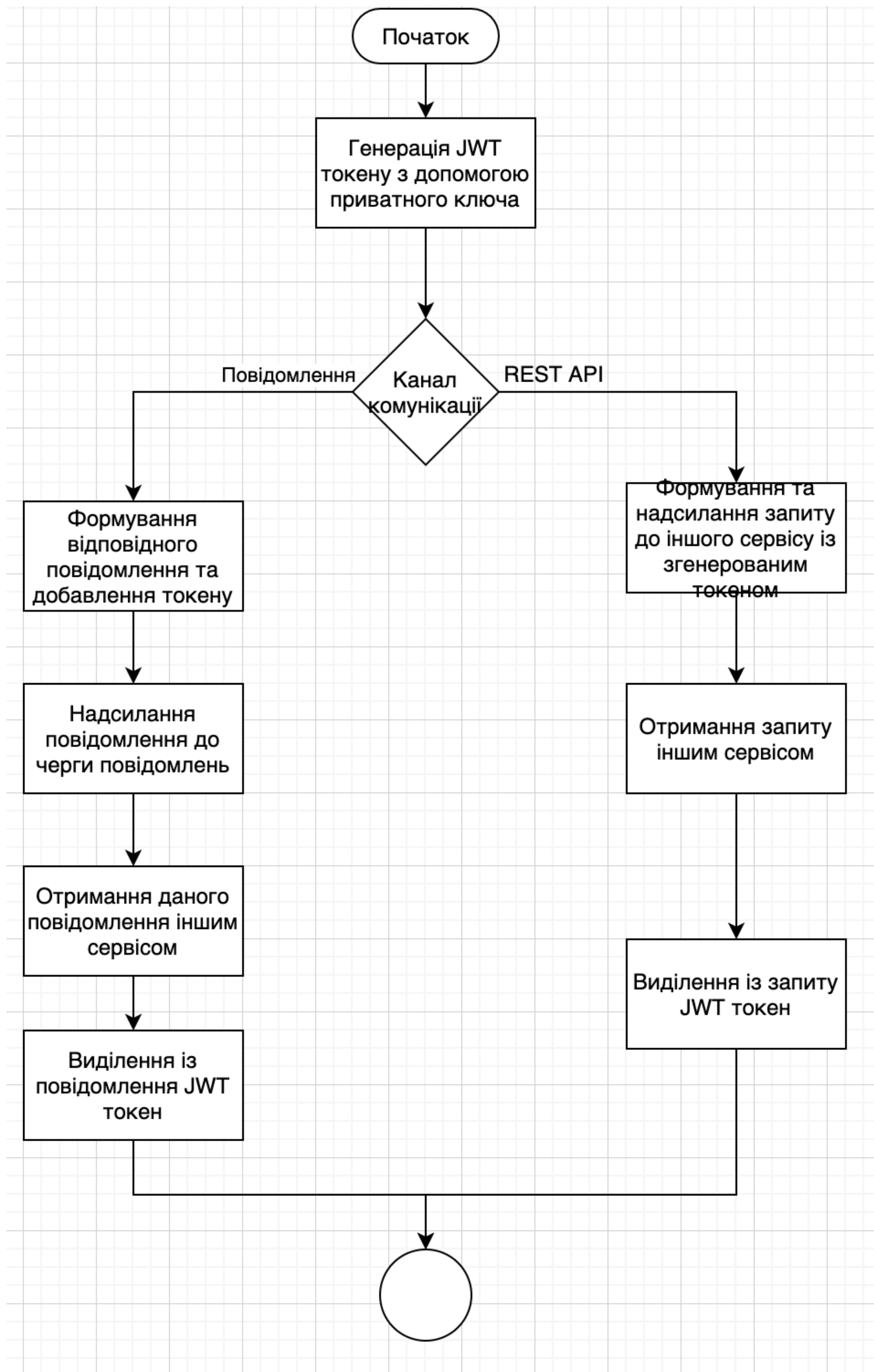

Рис. 2.5 Загальний алгоритм автентифікації частина 1

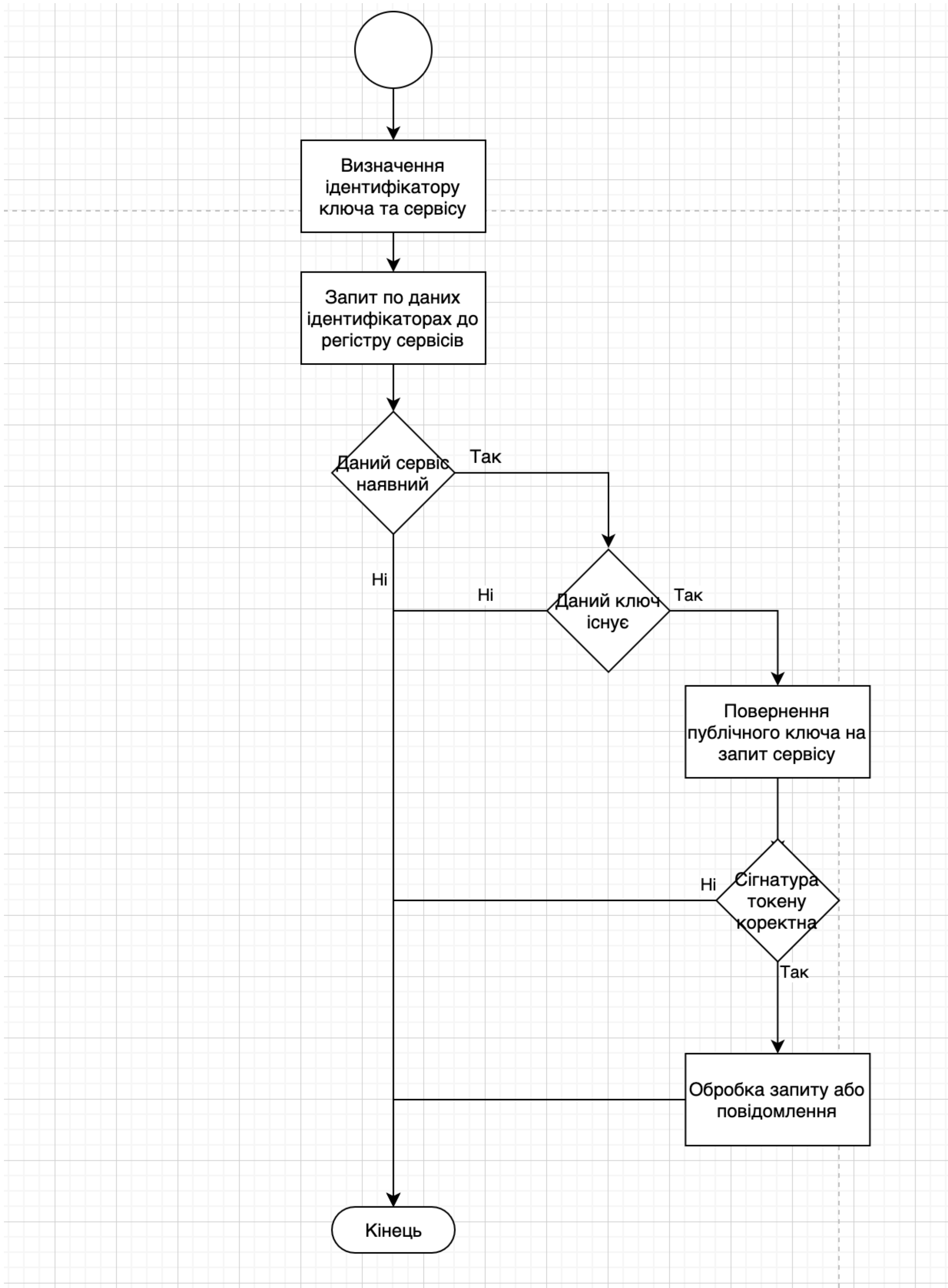

Рис. 2.6 Загальний алгоритм автентифікації частина 2

2.2.1 Опис компонента, що існує в МСА

Найчастіше компонентом в мікросервісній архітектурі, тобто сервісом, являється контейнер створений на основі образу, який створили розробники системи. На рис. 2.7 схематично зображено один із таких контейнерів в який використовує дану систему аутентифікації. Тобто є у контейнері (Application container) знаходиться бізнес логіка даного сервісу (Application bussiness logic), а також додатковий модуль в якому підключений фреймворк для організації безпечної комунікації.

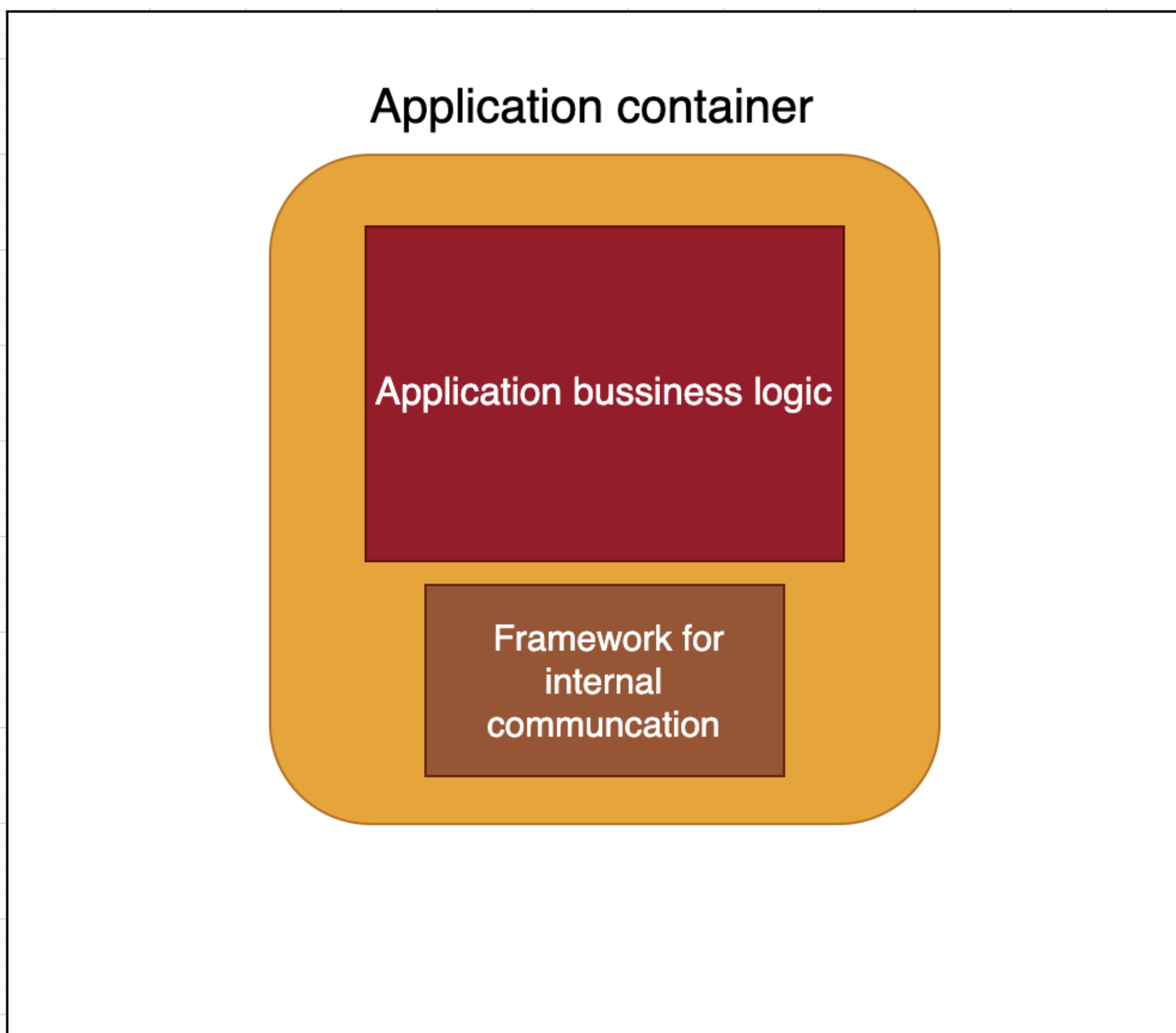

Рис. 2.7 Схема сервісу як окремого компоненту МСА

2.2.2 Опис регістру сервісів

Регістр сервісів - це додатковий сервіс, що знаходиться в середині системи і доступ до якого відкритий тільки в середині приватної мережі задля забезпечення його недоступностів від зловмисників. Даний сервіс створений для організацію списку доступних сервісів в системі. Окрім того він буде виконувати функцію сховища спеціальних публічних ключів для, будуть використовуватися для підтвердження JWT токенів в системі, а саме - JWKS (Json Web Key Set). JSON Web Key (JWK) - це структура даних, що представляє криптографічний ключ у форматі JSON. Приклад JWKS можна побачити на рис. 2.8.


```
{
"keys": [
  {
    "alg": "RS256",
    "kty": "RSA",
    "use": "sig",
    "x5c": [
      "MIIC+DCCAeCgAwIBAgIJBIGjYW6hFpn2MA0GCSqGSIb3DQEBBQUAMCMxITAfBgNVBAMTGGN1c3RvbWVyLWRlbW9zLmF1dGg
      wLmNvbTAeFw0xNjExMjIyMjIyMDVaFw0zMDA4MDEyMjIyMDVaMCMxITAfBgNVBAMTGGN1c3RvbWVyLWRlbW9zLmF1dGgwLmN
      vbTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAMnjZc5bm/
      eGIHq09N9HKHahM7Y31P0ul+A2wwP4lSpIwFrWHzxw88/7Dwk9QMc+orGXX95R6av4GF+Es/nG3uK45ooMVMa/hYCh0Mtx3g
      nSuoTavQEkLzCvSwTqVwzZ+5noukWVqJuMKNwjL77GNcPLY7Xy2/skMCT5bR8UoWaufooQvYq6SyPcRAU4BtdquZRiBT4U5f
      +4pwNTxSvey7ki50yc1tG49Per/0zA4O6Tlpv8x7Red6m1bCNHt7+Z5nSl3RX/
      QYyAEUX1a28VcYmR41Osy+o2OUCXYdUAphDaHo4/8rbKTJhlu8jEcc1KoMXAKjgaVZtG/
      v5ltx6AXY0CAwEAAaMvMC0wDAYDVR0TBAUwAwEB/zAdBgNVHQ4EFgQUQxFG602h1cG+pnyvJoy9pGJJoCswDQYJKoZIhvcNA
      QEFBQADggEBAGvtCbzGNBUJPLICth3mLsX0Z4z8T8iu4tyoiuAshP/Ry/
      ZBnFnXmhD8vwgMZ2lTgUWwlrvlgN+fAtYKnwFO2G3BOCFw96Nm8So9sjTda9CCZ3dhoH57F/
      hVMBB0K6xhklAc0b5ZxUpCIN92v/w+xZoz1XQBHe8ZbRHaP1HpRM4M7DJk2G5cgUCyu3UBvYS41sHvzrxQ3z7vIePRA4WF4b
      EkfX12gvny0RsPkrbVMXX1Rj9t6V7QXrbPYBAO+43JvDGYawxYVvLhz+BJ45x50GFQmHszfY3BR9TPK8xmMmQwtIvLu1PMtt
      NCs7niCYkSiUv2sc2mlq1i3IashGkkgmo="
    ],
    "n": "yeNlzlub94YgerT030codqEztjfU_S6X4DbDA_iVKkjAWtYfPHDzz_sPCT1Axz6isZdf3lHpq_gYX4Sz-cbe4rjmigxU
    xr-FgKHQy3HeCdK6hNq9ASQvMK9LBOpXDNn7mei6RZWom4wo3CMvvsY1w8tjtfLb-yQwJPltHxShZq5-ihC9irpLI9xEBTgG12
    q5lGIFPhTl_7inA1PFK97LuSLnTJzW0bj096v_TMDg7pOWm_zHtF53qbVsI0e3v5
    nmdKXdFf9BjIARRfVrbxVxiZHjU6zL6jY5QJdh1QCmENoejj_ytspMmGW7yMRxzUqgxcAqOBpVm0b-_mW3HoBdjQ",
    "e": "AQAB",
    "kid": "NjVBRjY5MDlCMUIwNzU4RTA2QzZFMDQ4QzQ2MDAyQjVDNjk1RTM2Qg",
    "x5t": "NjVBRjY5MDlCMUIwNzU4RTA2QzZFMDQ4QzQ2MDAyQjVDNjk1RTM2Qg"
  }
]}
```


Рис. 2.6 Приклад публічних JWKS

На рис. 2.8 можна побачити наступні поля даної структури:

- **alg** - криптографічний алгоритм, що використовувався для даного ключа;
- **kty** - сімейство криптографічних алгоритмів;
- **use** - призначення даного ключа;

- **x5c** - ланцюг X.509 сертифікату
- **n** - модуль RSA публічного ключа;
- **e** - експонента для RSA публічного ключа;
- **kid** - унікальний ідентифікатор ключа;
- **x5t** - цифровий підпис X.509 сертифікату.

Так як даний сервіс являється регістром усіх сервісів в системі відповідно існує процес реєстрації цих сервісів в систему та подальшої їх взаємодії. На рис. 2.9 розглянута поетапна схема відповідного процесу.

1. Перший крок полягає в тому, що створений фреймворк для компоненту, посилає запит на реєстрацію даного сервісу в системі. В запиті передається також і JWKS з декількома публічними ключами.
2. Регістр зберігає даний JWKS у власну базу, асоціюючи його з відповідним сервісом в системі, і віддає унікальний ідентифікатор сервісу назад у відповідь на запит. Даний ідентифікатор буде використовуватися для подальшої автентифікації сервісу в системі.
3. Через деякі проміжки часу сервіс буде відправляти запит, що будуть сигналізувати про стабільну роботу даного сервісу, таким чином підтримуючи актуальний статус системи.
4. Останній етап взаємодії полягатиме у постійній ротації ключів, що дасть змогу зменшити імовірність несанкціонованого доступу в систему.

Даний підхід дозволяє системі підтримувати актуальний стан, і в разі, якщо сервіс буде недоступний, то він видалиться з регістру. Таким чином в разі, якщо зловмисник отримає доступ до приватного ключа сервісу, який вийшов з ладу, або отрмає застарілий приватний ключ, він не зможе здійснити дії котрі зможуть нашкодити інформаційній системі, так як система автентифікації йому не дозволить.

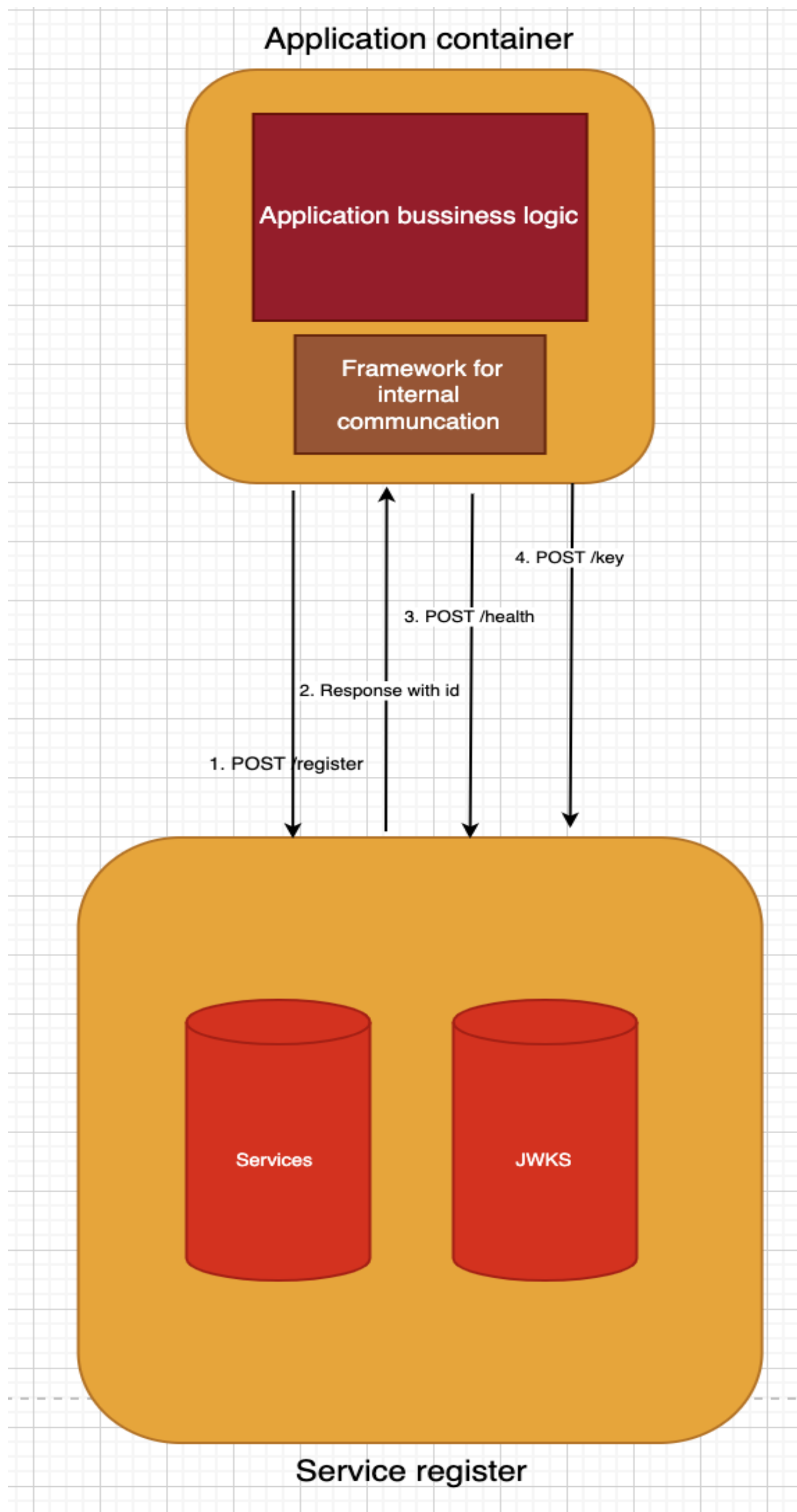

Рис. 2.9 Приклад взаємодії регістру сервісів та сервісу МСА

2.2.3 Опис комунікації з допомогою HTTP запитів

Як зазначалося раніше метод комунікації через HTTP запити є досить таки популярний, тож необхідно розглянути метод аутентифікації під час використання даного методу комунікації.

На рис. 2.10 зображено приклад даної комунікації.

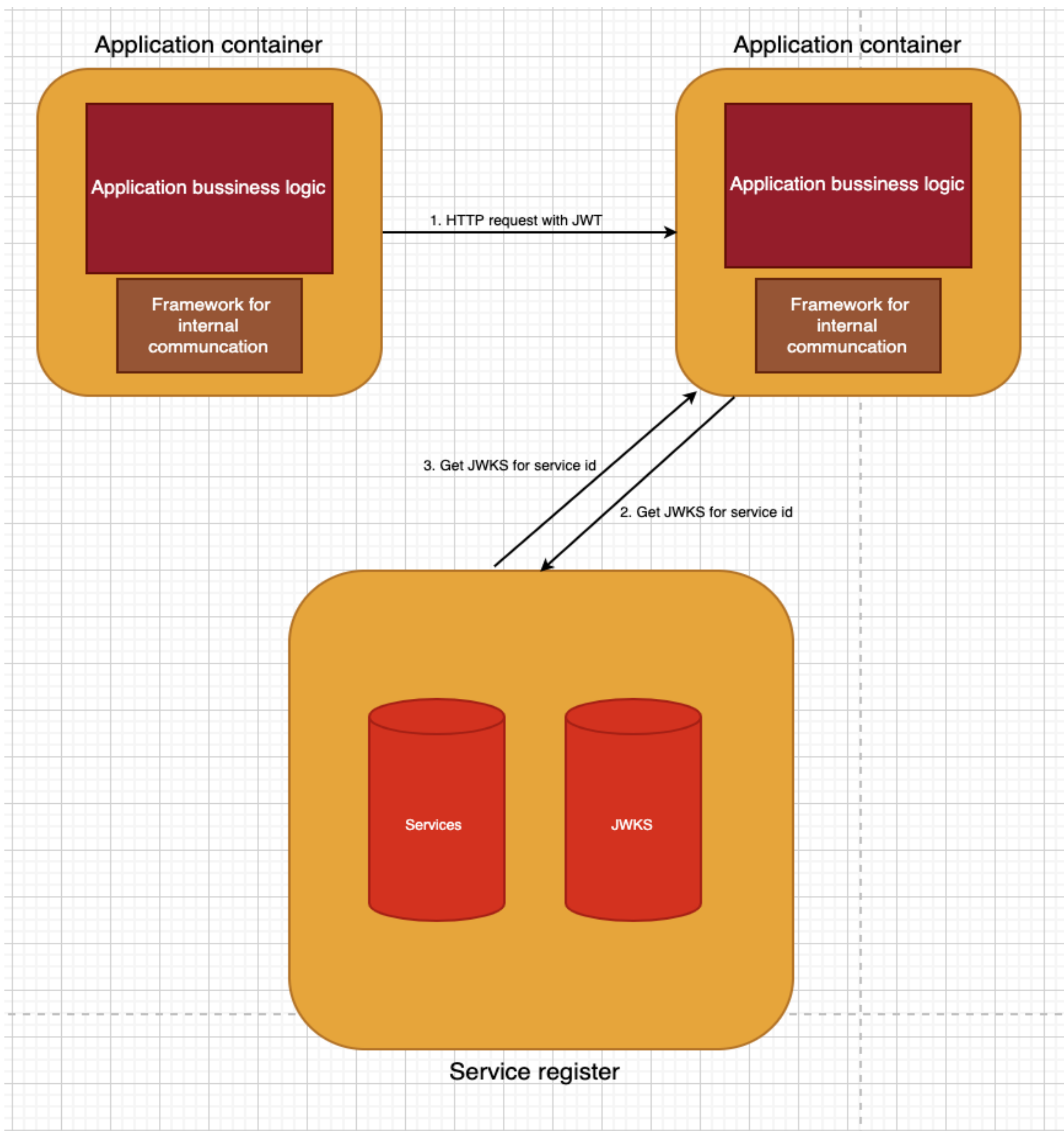

Рис. 2.7 Приклад взаємодії регістру сервісів та сервісу MCA

Взявши за приклад рис. 2.10 розглянемо поетапний процес автентифікації під час комунікації через HTTP запити. Під цифрою 1 позначено запит до сервісу А до сервісу В, задля отримання певного ресурсу. Для автентифікації використовується JWT токен, який являється не настільки простим і звичайним який використовується під час автентифікації користувача. Даний токен для створення сігнатури для підтвердження, використовує приватний актуальний JWK.

Під цифрою 2 позначено запит до регістру сервісу, задля отримання JWK з відповідним унікальним ідентифікатором ключа та ідентифікатором сервісу.

Під цифрою позначено відповідь регістру, яка може бути як і помилка в разі якщо сервісу не має в базі або ключ недійсний. Або вертає публічний ключ відповідно з допомогою якого, підтверджується сігнатура токену і відповідно надається доступ до ресурсів та функцій даного сервісу.

2.2.4 Опис комунікації з допомогою повідомлень та подій

Розглянемо як дана система буде працювати під час комунікації з допомогою повідомлень та подій на рис. 2.11.

Сервіс C під цифрою 1 відправляє повідомлення в обробник повідомлень, задля того, щоб інший сервіс D дістав це повідомлення і виконав певні дії пов’язані з цим повідомленням. Особливістю даного повідомлення являється те, що він містить у собі JWT токен згенерований сервісом C під час створення повідомлення, та сігнатура якого зашифрована з допомогою актуального JWK ключа.

Сервіс D отримавши дане повідомлення перевіряє чи містить воно JWT токен чи ні. В разі якщо відсутній то відповідно дано повідомлення не буде оброблятися. В іншому випадку з JWT токену дістається інформація про сервіс, а саме його ідентифікатор та ідентифікатор ключа з допомогою якого шифрувала сігнатура даного токену і робить запит на регістр в 3 кроці. В разі якщо сервісу або ключа не існує, регістр видасть відповідну помилку і повідомлення не буде надалі оброблюватися і видалиться із черги. Або якщо даний ключ існує і з його допомогою підтверджується джерело даного повідомлення.

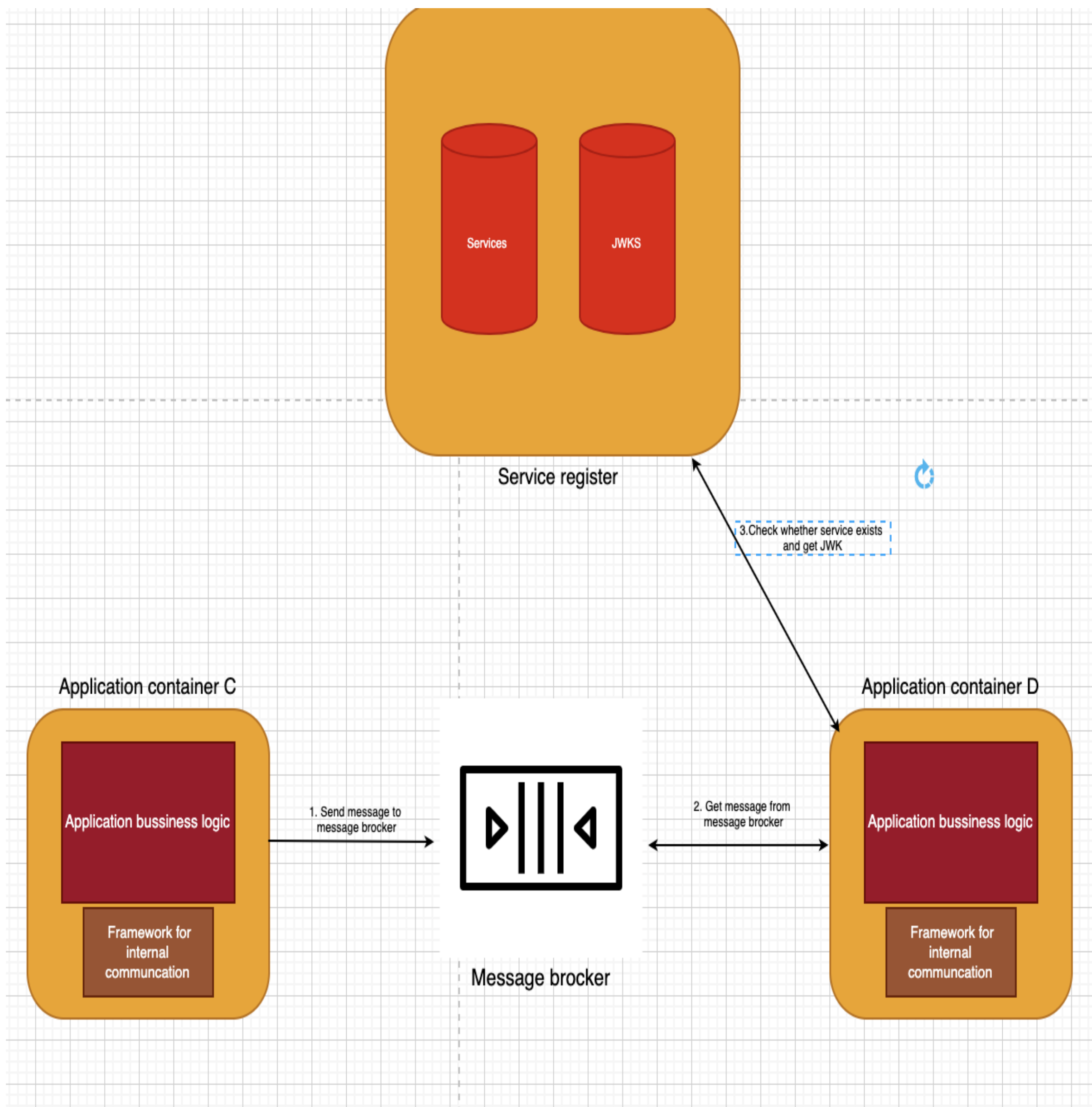

Рис. 2.11 Приклад взаємодії регістру сервісів та сервісу під час комунікації через повідомлення та події

2.3 Програмна та інфраструктурна реалізація

Для демонстрації запропонованої системи автентифікації було створено тестову інформаційну систему на базі мікросервісної архітектурти, схему якої можна побачити на рис. 2.11. За даною схемою було створено два сервіси User Service та Trip Service, а також запущений

контейнер Message Brocker, який використовує Apache Kafka для забезпечення функцій брокера повідомленнь.

Дані сервіси та брокер повідомлень у вигляді докер контейнерів з допомогою хмарного сервісу Amazon Elastic Cluster Service, який являється оркестратором контейнерів, тобто він запускає та перевіряє доступність та статус контейнера, чи він не вийшов з ладу тощо. В разі якщо один із контейнерів через помилку вийшов з ладу, то оркестратор перезапускає даний сервіс для відновлення дієздатності системи. Також даний оркестратор, може збільшувати кількість сервісів в разі високої навантаженості системи, що дає змогу витримати наплив великої кількості користувачів в разі необхідності. А також зменшує кількість запущених сервісів, в разі якщо в них немає необхідності, задля зменшення використаних ресурсів.

Даний кластер запущений в приватній підмережі, що дає змогу зменшити імовірність проникнення зловмисника в систему. Окім того доступ до сервісів організований через спеціальний шлюз, який в залежності від параметрів запиту, перенаправляє його на відповідний сервіс.

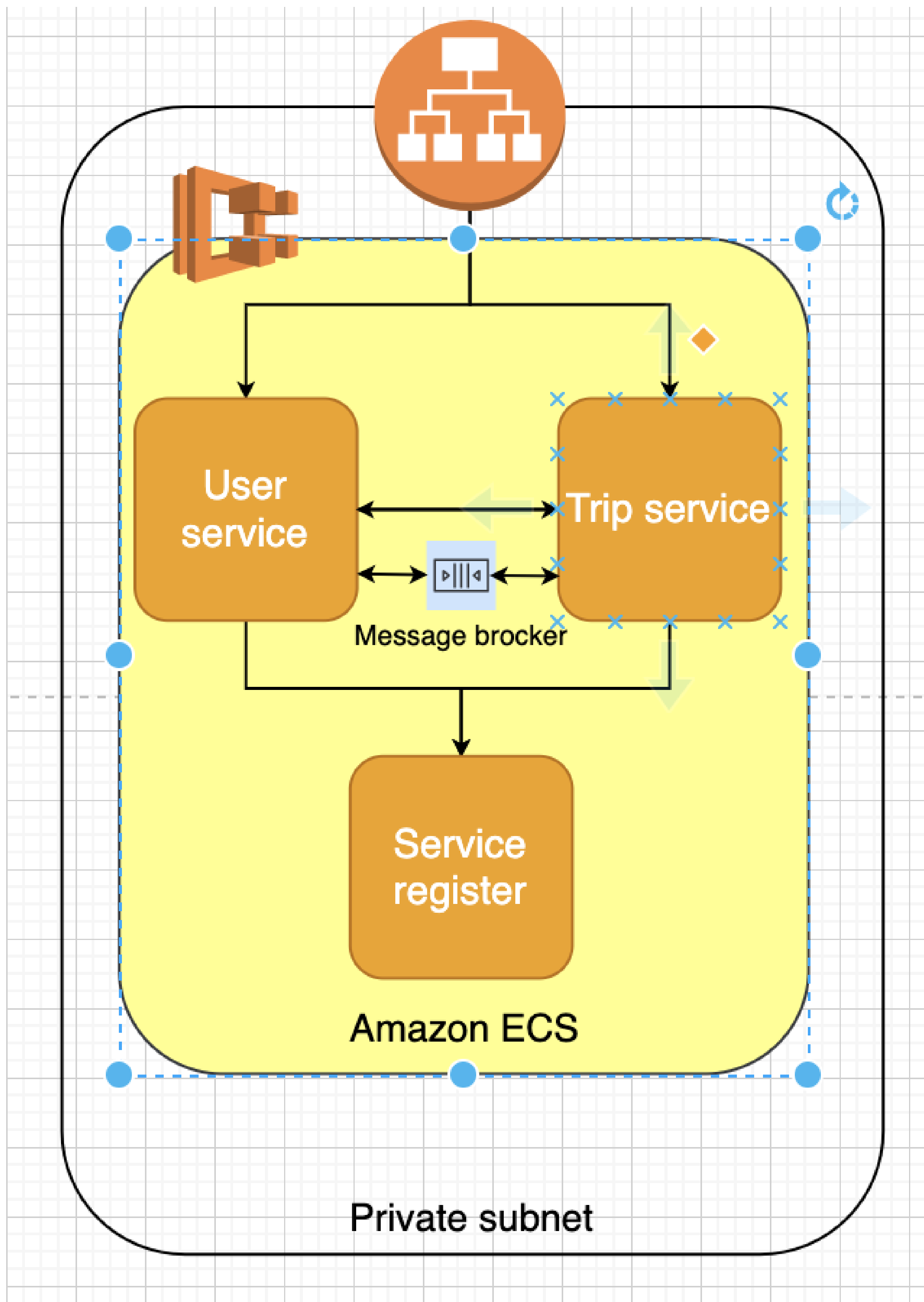

Рис. 2.11 Схема інфраструктури інформаційної системи на базі МСА за використанням

На базі даної інфраструктури розглянемо програмну реалізації запропонованого алгоритму дій для автентифікації сервісів в МСА. Для прикладу розглянути процес старту сервісу користувачів та його реєстрації в реєстрі сервісів та генерації ключів і відповідно ротації даних ключів.

Під час старту сервісу користувачів, авторський підключений фреймворк надсилає запит на з приводу його реєстрації в системі. Приклад даного запиту можна побачити на рис. 2.12.

```
public OAuth2AccessToken registerService(OAuth2Authentication authentication) throws AuthenticationException {

    OAuth2AccessToken existingAccessToken = tokenStore.getAccessToken(authentication);
    OAuth2RefreshToken refreshToken = null;
    if (existingAccessToken != null) {
        if (existingAccessToken.getRefreshToken() != null) {
            refreshToken = existingAccessToken.getRefreshToken();
            tokenStore.removeRefreshToken(refreshToken);
        }
        tokenStore.removeAccessToken(existingAccessToken);
    }


    refreshToken = createRefreshToken(authentication);
    OAuth2AccessToken accessToken = createAccessToken(authentication, refreshToken);
    tokenStore.storeAccessToken(accessToken, authentication);
    refreshToken = accessToken.getRefreshToken();
    if (refreshToken != null) {
        tokenStore.storeRefreshToken(refreshToken, authentication);
    }
    eventPublisher.publishEvent(new UserInSystem(((YazaUserDetails) authentication.getPrincipal()).getId()));

    return accessToken;
```

Рис. 2.12 Приклад коду надсилання запиту на реєстрації сервісу в системі

Під час даного запиту реєстр сервісів генерує ідентифікатор, для даного сервісу зберігаючи інформацію, яку даний сервіс про себе надав. Приклад коду на даний процес можна побачити на рис. 2.13.

```
public void storeService(OAuth2AccessToken token, OAuth2Authentication authentication) {
    AccessTokenDTO accessToken = new AccessTokenDTO();
    accessToken.setTokenId(extractTokenKey(token.getValue()));
    accessToken.setToken(accessTokenMapper.toDto(token));
    accessToken.setAuthenticationId(authenticationKeyGenerator.extractKey(authentication));
    accessToken.setUserName(authentication.isClientOnly() ? null : authentication.getName());
    accessToken.setClientId(authentication.getOAuth2Request().getClientId());
    accessToken.setAuthentication(authenticationMapper.toDto(authentication));
    tokenRepository.save(accessToken);

}
```

Рис. 2.13 Приклад коду обробки запиту сервісу на реєстрацію в реєстрі

Після цього, сервіс отримавши ідентифікатор, зберігає його даний ідентифікатор в локальне сховище для подальшого використання під комунікації з сервісом поїздок, що можна побачити на рис. 2.12. Далі відбувається генерація пари RSA ключів і їх декодування в формат JSON, а саме JWK, задля подальшого зручного опрацювання. Під час генерації та декодування даного ключа, до моделі генерується і додається унікальний ідентифікатор, який записується як властивість *kid* для JSON формату. Даний ідентифікатор використовується для того, щоб в разі для одного сервісу існує відразу декілька ключів шифрування, то з допомогою унікального ідентифікатора визначити необхідний для подальшої обробки токену. Приклад даного коду можна побачити на рис 2.14.

```
    val generator = RSAKeyGenerator( size: 2048)
    val jwkExpirationHours: Long = 48
    val jwkCreationgHours: Long = 24

    fun get(): IO<Unit> = IO.fx {  this: ConcurrentSyntax<ForIO>
        val savedJWK = jwksIORepository.findAll().bind()
        val filterCreatedDuringLastDay =
            { it: JWKEntity -> LocalDateTime.now().minusHours(jwkCreationgHours).isBefore(it.createdAt) }
        val filterCreatedMoreThan2Days =
            { it: JWKEntity ->
                LocalDateTime.now().minusHours(jwkExpirationHours).isAfter(it.createdAt)
            }

        if (savedJWK.none(filterCreatedDuringLastDay)) {
            jwksIORepository.saveJWK(createJWKEntity()).bind()
        }
        jwksIORepository.deleteAll(savedJWK.filter(filterCreatedMoreThan2Days)).bind()
    }



    private fun createJWKEntity(): JWKEntity {
        val id = UUID.randomUUID().toString()
        return JWKEntity(
            id,
            createNewJWK(id).toJSONString(),
            LocalDateTime.now()
        )
    }

    private fun createNewJWK(id: String): JWK {
        return generator
            .keyUse(KeyUse.SIGNATURE)
            .algorithm(JWSAlgorithm.RS256)
            .keyID(id)
            .generate()

    }
}
```

Рис. 2.14 Приклад коду створення та ротації ключів сервісу в системі

Далі розглянемо приклад запиту до сервісу подорожей з використанням визначеного методу автентифікації. На рис 2.15 зображено код генерації JWT токену на базі актуально JWK ключа. Під час створення даного токену, в заголовок додається параметер *kid*, що являється унікальним ідентифікатором ключа з допомогою, якого буде підписаний даний токен. Окрім того для визначення сервісу з якого надійшов запит, до безпосередньо тіла токену додається параметер *iss*, який являється ідентифікатором сервісу відповідно з якого був зроблений запит. Приклад згенерованого токену можна побачити на рис. 2.16, що поділений крапками на 3 частини: заголовки, тіло, сігнатура .

```
val clientAssertion =
    Jwts.builder()
        .setAudience("audience")
        .setId(UUID.randomUUID().toString())
        .setExpiration(Date.from(Instant.now().plusSeconds( secondsToAdd: 3600)))
        .setIssuer(properties.clientId)
        .setSubject(properties.clientId)
        .setHeaderParam( name: "kid", key.keyID)
        .signWith(key.toRSAKey().toRSAPrivateKey())
        .compact()
```

Рис. 2.15 Приклад коду


```
eyJraWQiOiI3NDJiOGFkNC1lZDNjLTQyZjQtODRmYi1hNzBjNzI5OGYzODYiLCJhbGciOiJSUzI1NiJ9.
eyJhdWQiOiJodHRwczovL2lkZW50aXR5Lm1vbmV5aHViLmNvLnVrL29pZGMiLCJqdGkiOiJjYjFmOTU4OC1kYzZkLTRhOTktOTU2Ni
1lNTg4Mjg3OWMzNmQiLCJleHAiOjE1OTQxMTg0ODQsImlzcyI6ImM2Y2E4YWFkLTBhZjgtNGUxNi1iMDM5LTQzZTI0Yzc0OTBkNCIs
InN1YiI6ImM2Y2E4YWFkLTBhZjgtNGUxNi1iMDM5LTQzZTI0Yzc0OTBkNCJ9.
gYQuBc4wEFFy-0SUuj4ZjN2slM7nrWh3jf6ZGhNaH1CResO-MGNnDLlOMO4T_ZQ-1gpohzEN9BehgZE6Uf1pi2VsfCWLUtne5P6MLo
1KR6yh8IiyVHqGB6QMNxj4Qizy58ZEczcUceZYl4DqTmuP1XaI0vODixy57J2fx4mQJE7-orR1PwW2NcZ167NLKcG2ixUWRoiBNl36
oAC3KtIWmlib5Lkxuwnl_Gq202M-rDkfSeR0Swt8oBX_oPpHkTi
-8pVUdsvlsqgIEWhL-dD3scTZ5XrIkq0xpgjKRjKDplhmmSe0pjD03xkpNeKe0uExm5Urku5gP3bibKV3b-SnYw
```


Рис. 2.16 Приклад згенерованого JWT токену

На рис. 2.17 можна побачити безпосередньо те, що містить даний токен і яку інформацію можна з нього дістати внаслідок його декодування. Як бачимо в разі декодування перших 2 частин токену, відобразиться інформація, яка була закладена в токен на етапі його створення. В заголовку (header) маємо два поля: *kid* - унікальний ідентифікатор ключа з допомогою якого підписали даний токен, *alg* - алгоритм шифрування, що був використаний для підписання даного токену. В тілі (payload) даного токену

можна побачити декілька полів на які варто звернути увагу: *exp* - це час через який даний токен буде недійсний, та *iss* - це ідентифікатор сервісу з якого був зроблений даний запит.

```
{ 2 items
  "header" : { 2 items
    "kid" : "742b8ad4-ed3c-42f4-84fb-a70c7298f386"
    "alg" : "RS256"
  }
  "payload" : { 5 items
    "aud" : "https://identity.moneyhub.co.uk/oidc"
    "jti" : "cb1f9588-dc6d-4a99-9566-e5882879c36d"
    "exp" : 1594118484
    "iss" : "c6ca8aad-0af8-4e16-b039-43e24c7490d4"
    "sub" : "c6ca8aad-0af8-4e16-b039-43e24c7490d4"
  }
}
```

Рис.2.17 Результат декодування токену

Під час використання каналу зв'язку через REST API, даний токен передається з допомогою HTTP заголовку *Authorization* з приставкою перед даним токеном *bearer*. Приклад даного запиту з топомогою інструменту curl можна побачити на рис. 2.18

```
curl --request GET \
  --url https://userservice/v2.0/users/49e350c8-a368-4978-96a8-66e09c7cacfb \
  --header 'Authorization: Bearer eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCIsImtpZCI6IlRxVk1laV9XdUtqZW5HWlJU
  bnJpeUxXRnZuS2tzTjNvLWFuWXBqS0JEbVUifQ.eyJqdGkiOiJGN0lqZmwwZWJicV9hdng0RE1ueEkiLCJzdWIiOiI1ZWQ4ZDdiN
  jg1NjUzMDFjZDkzOWIxM2QiLCJpc3MiOiJodHRwczovL2lkZW50aXR5Lm1vbmV5aHViLmNvLnVrL29pZGMiLCJpYXQiOjE1OTc2N
  jA5NDUsImV4cCI6MTU5NzY2ODE0NSwic2NvcGUiOiJhY2NvdW50czpyZWFkIiwiYXVkIjoiZWQ1MWUwNDEtZjVmYS00NTkyLTg2M
  DgtNzM3MzIyNjhlMmUxIn0.TsN26UYahTmpzvmNrqG0Lxj1mTKNK4WVxVuZjQLigb-HHsw2JdO1c5g69Arpra-nZMAwjwaPiTrnv
  81qUv5nYYF7R6oaYwZhYiczQLGg5G5fbm4kW13Aq1qJIzZ8VqT5rS6i5WkAJbvs23Mvz_jlCCYXC3zu9WYWFCgvxArR1qXKo92zj
  Nth4Qqm26LuDug7bw1uaGOgVOUIfVdKRgUJhLdP9dzEL00in7ru2BvZS0txI8ctgQViHlQbvWW-oMWKlaRF84rrboae_g8PtrIP4
  rv9XwCrbv8nsPsOYW4X2oOZmG21cy4u6aC6lyMurK4de_ZfMtshxEfLiRXE8phDpw '
```

Рис.2.18 Приклад запиту з JWT токеном

Отже запит з даним токеном надсилається до сервісу користувача для того, щоб перевірити чи даний користувач існує в системі і відповідно повернути необхідну інформації з приводу цього користувача. Сервіс подорожей отримавши даний запит, спробує дістати JWT токен з *Authorization* заголовку. В разі якщо даного заголовку запиту немає або ж він в некоректному форматі, то такий запит не виконається так як він являється не автифікаваним. В разі ж, якщо даний токен існує в запиті і він відповідає формату, то надалі він декодується і дістаються необхідні дані. Приклад вище описаного в програмному коді можна побачити на рис. 2.19.

```
public OAuth2Authentication checkToken(String accessTokenValue) throws AuthenticationException,
        InvalidTokenException {
    OAuth2AccessToken accessToken = tokenStore.readAccessToken(accessTokenValue);
    if (accessToken == null) {
        throw new InvalidTokenException("Invalid access token: " + accessTokenValue);
    } else if (accessToken.isExpired()) {
        tokenStore.removeAccessToken(accessToken);
        throw new InvalidTokenException("Access token expired: " + accessTokenValue);
    }

    OAuth2Authentication result = tokenStore.readAuthentication(accessToken);
    if (clientDetailsService != null) {
        String clientId = result.getOAuth2Request().getClientId();
        try {
            clientDetailsService.loadClientByClientId(clientId);
        } catch (ClientRegistrationException e) {
            throw new InvalidTokenException("Client not valid: " + clientId, e);
        }
    }
    return result;
}
```

Рис.2.19 Приклад запиту з JWT токеном

Отримавши необхідну інформацію з токену, а саме ідентифікатор сервісу та ідентифікатор ключа, робиться запит на реєстр сервісів на основі цих даних, і відповідно перевіряється наявність даного сервісу та наявність

даного ключа в реєстрі. Приклад програмної реалізації можна побачити на рис. 2.20.

```java
@Service
@Transactional
public class ClientService {
    private final ClientRepository clientRepository;

    public ClientService(ClientRepository clientRepository) { this.clientRepository = clientRepository; }

    public Client validateClient(String id, String secret, String action) {
        Client client = clientRepository.findByClientIdAndClientSecret(id, secret)
                .orElseThrow(() -> new InvalidClientException("Client is invalid"));
        Set<String> authorities = client.getAuthorities().stream().map(Authority::getName).collect(Collectors.toSet());
        if (!authorities.contains(action)) {
            throw new ClientActionForbiddenException();
        }

        return client;

    }
}
```

Рис.2.20 Приклад запиту перевірки сервісу та ключа

Відповідно, в разі якщо даний ключ та сервіс існує в реєстрі. То вертається публічний ключ в JSON форматі приклад якого можна побачити на рис. 2.21.


```json
  "keys": [
    {
      "kty": "RSA",
      "e": "AQAB",
      "use": "sig",
      "kid": "2f381926-a069-4b28-aeb3-97dcfa2a8b5e",
      "alg": "RS256",
      "n": "j8qGMAhJZCwdrYlrUyV2JZJDSZWGy33EPWBmakWxezZbsABcYIqYmjbR0VdEtbar0E_VlDlFn0-Pn7AmS-
0WO8u7hzPs122CudGhtQ95cTAl_6C3lsCYHIzWIOgBqzOP7sGczZONLPQnKTj2dMftKxXaea2hDSS1oqIF8QmC7AFq6GSrxku0GeO6yH18H8tHCxAebeoa6Hg
iiLNNyCTWiFZpBllpZr_Lj_xa4bHqJVjqShx7wYKD-k9x-gjJFHLNv4HSA6smArdn_OOLMEfW-j5bTnMslF111Dsb37tuIFO5AqUZCo95aC-
DAZLeAKKif44fA3RKvF0PS_52-mceYQ"
    },
```


Рис.2.21 Приклад формату публічного JWK ключа

Наступний і фінальний крок це підтвердження сігнатури токену, а саме порівняння тіла токену та дешифрованої сігнатури з допомогою отриманого публічного ключа. І в разі співпадіння процес автентифікації завершується, так як пройшло все успішно.

2.4 Тестування системи

Проведемо тестування даної системи на правильність функціонування за допомого програми Insomnia, а також на основі побудованої системи логування. Дана програма дозволяє формувати та надсилати HTTP - запити з використанням зручного графічного інтерфейсу.

Проведемо тестування реєстру сервісів. Для початку слід здійснити запит на збереження токену доступу в системі. Успішний результат виконання можна побачити на рис. 2.22.

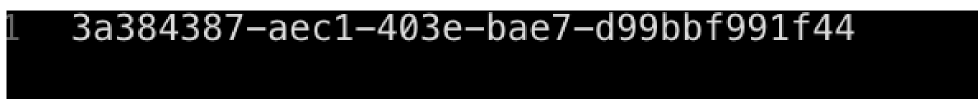

Рис. 2.22 Приклад відповіді на успішне виконання запиту на реєстрацію

У разі, якщо сервіс не має дозволу на створення токену в системі, буде надано відповідь на створення токену в системі, приклад якої наведено на рис. 2.24 та рис. 2.25.

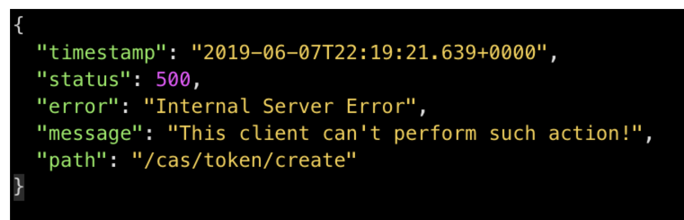

Рис. 2.24 Результат запиту в разі відсутності прав на збереження токену

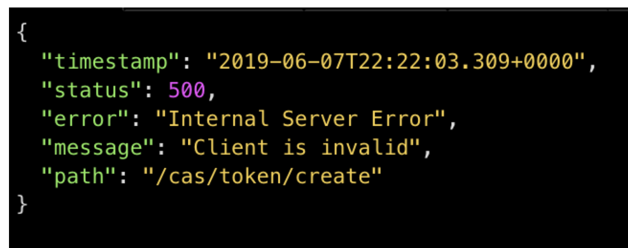

Рис. 2.25 Результат запиту в разі відсутності даного клієнта в зареєстрованих

Далі протестуємо взаємодію ЦСА-сервісу та сервісу авторизації користувача. На даний URL "http://18.184.48.7:8082/auth/login?username =andrew&password=password" надішлемо запит, у відповіді якого отримаємо токен для подальшого доступу до ресурсів системи, приклад відповіді наведено на рис. 2.26.

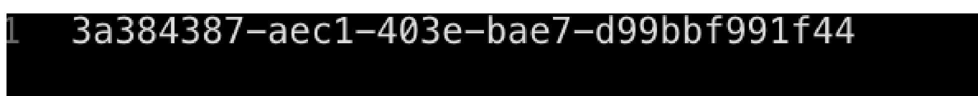

Рис. 2.26 Фрагмент відповіді на запит авторизації

Для демонстрації того, що дійсно використовується МСА, на локальному хості запустимо сервер з ресурсами, який знаходиться на віддаленому сервері. На рис. 2.27 наведено приклад запиту та результат, який повернув даний сервіс.

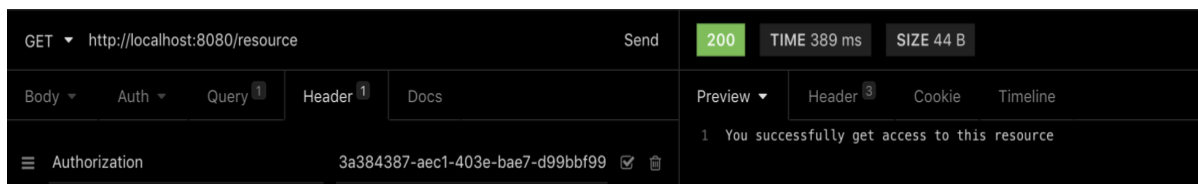

Рис. 2.27 Приклад успішно виконаного запиту до серверу ресурсів.

Отже, можна констатувати, що розроблений програмний модуль успішно функціонує, виконуючи авторизацію користувача.

2.5. Висновки до розділу

Отже, для досягнення поставленої мети було запропоновано розробити систему автентифікації компоентів МСА. У процесі створення систему було використано наступні технології:

- мова програмування – Java, оскільки Java є об'єкто - орієнтованою мовою програмування і має систему автоматичного очищення пам'яті, створення програмного модуля набагато спрощується за рахунок того, що непотрібно перейматися низькорівневим програмуванням;
- фреймворк - Spring Framework. Даний фреймворк є найпопулярнішим у середовищі Java-розробників, оскільки автоматизує і пропонує готові схеми для вирішення певних типів

задач. А також використовувався один із підпроектів Spring Boot для спрощення старту модуля;

- бібліотека OpenFeign. Дана бібліотека надає зручний спосіб програмного налаштування та використання REST-клієнта в серверній частині програмного модуля;
- збірник проектів Gradle, який використовується для підключення відповідних залежностей, бібліотек тощо;
- Docker - використовувався для запуску програмного модуля в окремому системному середовищі на основі технології контейниризації;

Концепція даного системи є універсальною для багатьох видів і підходів до організації комунікації між компонентами в інформаційних системах побудованих на базі МСА. Тим не менш, це - не є фінальною версією. При подальшому розвитку і розробленні даного системи, її функціональність можна розширити до авторизації сервісу в системі: створення ролей в системі, надання ролям дозволів і заборон до різних компонентів системи тощо.

ВИСНОВКИ

Відповідно до досліджень та висновкам експертів мікросервісна архітектура на сьогоднішній день знаходиться на піку популярності. Величезна кількість розробників та інженерів вже впровадили або знаходяться в процесі впровадження даного підходу до своїх інформаційних систем.

Як встановлено за результатами проведеного аналізу, характерною особливістю мікросервісної архітектура є можливість побудувати комплексну високонавантежену систему, яка в подальшому використанні зможе дуже легко масштабуватися, дозволяючи витримати великий наплив користувачів в системі.

Ще однією з важливих переваг мікросевісів є незалежність від використаних технологій під час розроблення будь - якого сервісу, будь - то мова програмування, бібліотека, фреймворки тощо. Єдине, що необхідно урахувати, - це стандартизувати канал спілкування між сервісами для їхньої взаємодії.

Тим не менш, оскільки даний підхід знаходиться тільки на етапі масштабного впровадження, існує багато невирішених проблем, розв’язанню яких присвячено значну кількість досліджень та наукових праць, переважно, зарубіжних авторів. Однією з таких проблем являється побудова системи автентифікації та авторизації користувачів в системі. Попри надзвичайно велику кількість матеріалів, семінарів та дискусій з приводу цієї теми, єдиного рішення не існує, оскільки кожен експерт, компанія, підприємство тощо пропонують свій варіант вирішення даної проблеми, що часто не влаштовує інших.

Під час виконання дипломної роботи було проаналізовано існуючі варіанти та методи щодо забезпечення безпечної комунікації компонентів

системи під час їх взаємодії. З них виділено найбільш популярні: JWT та взаємна автентифікація по TLS. Попри всі переваги даних рішень, існують недоліки та вразливості, через які дані варіанти не задовольняють вимогам щодо забезпечення безпечної комунікації в мікросервісах.

Нормативно-правова база України в сфері автентифікації є досить недосконалою в порівнянні із західними аналогами, що, зрозуміло, негативно впливає на реалізацію процесу автентифікації під час розроблення інформаційних систем. Тим не менш, існує чимало міжнародних стандартів та законодавчих актів, на основі яких можна побудувати власну систему нормативно-правового забезпечення для регулювання даного питання в усіх системах, де використивуються персональні дані для автентифікації.

Результати проведеного аналізу та досліджень дали змогу дійти висновку щодо необхідності створення власної системи автентифікації компонентів МСА.

Для реалізації даного модуля було використано наступні технології:

- мова програмування - Java;
- фреймворк - Spring Framework;
- бібліотека OpenFeign;
- Збірник проектів Gradle;
- Docker;
- Amazon ECS;
- Apache Kafka;

Така організація процесу автентифікації під час комунікації для перешкоджання діям зловмисників дала змогу впровадити додатковий рівень захисту, а саме:

- на рівні інфраструктури - обмеження доступу до реєстру сервісів глобальної мережі, що ускладнює процес проникнення порушника та отримання доступу до ресурсів;

- на рівні комунікації - шифрування даних відомими ключами.
- на програмному рівні - аутентифікації та авторизації компонентів системи, задля попередження несанкціоного доступу та дій в системі.

Практичне використання базової концепції ЦСА - серверу, можна побачити у багатьох великих корпоративних системах, в яких використовуються різні методи надання автентифікації, такі, як звичайний логін та пароль, магнітні ключі та перепустки, розпізнавання лиця, відбиток пальцю тощо.

Відповідно до завдання, було проведено успішне тестування розробленого програмного модуля, що продемонструвало його дієвість та доведено досягнення поставленої перед автором мети.

Отже, розроблено система, що розрахована для використання розробниками інформаційних систем на основі мікросервісної архітектури у масштабних корпоративних рішеннях. Для успішної експлуатації модуля розроблено рекомендації, які передбачають наступне:

- запускати даний програмний модуль у приватній мережі та використати протокол HTTPS.