Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

МЕТОДИ І ЗАСОБИ РОЗРОБКИ ПОДІЄ-КЕРОВАНИХ ЗАСТОСУНКІВ НА SERVERLESS АРХІТЕКТУРІ

Текстова частина до курсової роботи
за спеціальністю “Комп’ютерні науки” 122

Керівник курсової роботи
асистент Шабінський А. С.
_____
(підпис)
“____” _____ 2020 р.

Виконав студент Моренець І. Е.
“____” _____ 2020 р.

Київ 2020

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Зав. кафедри інформатики,
доцент, к.ф.-м.н.
_____ С. С. Гороховський
(підпис)
“____”_____ 202_ р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студенту Моренцю Ігорю Едуардовичу

1-го курсу магістерської програми факультету інформатики

ТЕМА: Методи і засоби розробки подіє-керованих застосунків на serverless архітектурі

Вихідні дані:

Аналіз та приклади застосування основних засобів, методів та паттернів розробки подіє-керованих застосунків на serverless архітектурі.

Зміст ТЧ до курсової роботи:

Вступ
1. Аналіз предметної області
2. Опис типових сценаріїв використання
Висновки
Список джерел
Додатки (за необхідністю)

Дата видачі “___” _____ 202_ р.

Керівник _____ Завдання отримано _____

Календарний план виконання курсової роботи

№	Назва етапу курсового проекту	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу	листопад 2019 р.	
2.	Огляд літератури за темою роботи	листопад - грудень 2019 р.	
3.	Оволодіння навичками розробки подіє-керованих застосунків на serverless архітектурі	грудень - лютий 2020 р.	
4.	Експерименти та дослідження типових засобів, методів та паттернів	лютий - березень 2020 р.	
5.	Написання пояснювальної роботи	квітень 2020 р.	
6.	Створення слайдів для доповіді та написання доповіді	травень 2020 р.	
7.	Надання роботи керівнику для перевірки	травень 2020 р.	
8.	Корегування роботи за результатами перевірки керівником	травень 2020 р.	
9.	Захист курсової роботи	травень 2020 р.	

Студент Моренець І. Е. _____

Керівник Шабінський А. С. _____

“_____” _____ _____ р.

Зміст

Анотація

В цій роботі були досліджені та проаналізовані сучасні методи та інструменти для розробки подіє-керованих застосунків використовуючи serverless архітектуру, а також виокремленні та описані основні типові сценарії використання. Всі досліджені паттерни супроводжуються кодом для демонстрації.

Не пояснені далі важливі терміни:

A. Хмарні обчислення – модель швидкого та зручного доступу до обчислювальних ресурсів, якими керує інша компанія-постачальник.
B. Platform-as-a-Service – вид хмарних сервісів що дозволяють розробляти застосування майже не піклуючись про інфраструктуру та розміщення.
C. Гнучке масштабування – автоматичне масштабування застосування в залежності від поточного навантаження.
D. "Rich client" застосування – застосування де основна робота відбувається на клієнтській частині.
E. Логгер (Logger) – об'єкт що дозволяє логувати (записувати) повідомлення під час роботи програми.
F. Proof of Concept – реалізація якоїсь ідеї для доказу того що вона має потенціал. Зазвичай є не оптимальним, але швидко досягаємим.
G. MapReduce – програмна модель для паралельної розподіленої обробки даних на кластері.

Вступ

З часом веб-застосунки ростуть до величезних розмірів, все більше і більше людей приходять на проект, створюються окремі команди для частин проекту і навіть з гарною масштабованою архітектурою рано чи пізно стає дуже складно працювати з застосунком як одне ціле - монолітом. Через це був розроблений мікросервісний підхід, який "розбиває" моноліт на декілька сервісів, що працюють всі разом як одне розподілене застосування. Такий підхід вже вважається стандартом в багатьох компаніях.

Та з розвитком хмарних технологій прийшов новий погляд, у якого є декілька назв - serverless, Function-as-a-Service та наносервіси. Цей підхід підіймає гнучкість на новий рівень, значно зменшуючи типову одиницю розгортання і надає нові можливості розробникам. Serverless архітектура активно розвивається і є предметом цікавості як для великих корпорацій, так і поодиноких розробників та дослідників.

Однак, цей напрям ще недостатньо стабільний і, хоча його популярність почала стрімко зростати ще в 2014-му році [1], досі не було точно визначено як саме варто будувати застосування використовуючи такий підхід, особливо великі.

Через це за мету цієї роботи було поставлено дослідити галузь, популярні інструменти та підходи, та описати типові сценарії застосування FaaS.

Роботу можна використовувати як посібник по основним поняттям serverless, для ознайомлення зі станом галузі та типовими способами використання на момент написання.

Робота складається з двох розділів.

В першому описано особливості, переваги та недоліки досліджуваного підходу, проаналізовано сучасні інструменти та аргументовано вибір одного для дослідження.

Другий розділ присвячено дослідженню обраного інструмента та демонстрації та розбору типових сценаріїв використання.

1. Аналіз serverless моделі та стан галузі

1.1. Вступ до Serverless

1.1.1. Історія та зв'язки зі схожими підходами

Термін "Serverless" може легко збентежити, адже він не означає ні те що вам не потрібні сервери, ні те що не потрібно писати backend код. Одне з перших вживань терміну було ще в 2012 році [2] і з часом він еволюціонував.

Насправді, вживаючи це слово можуть мати на увазі дві досить різні речі - вже згаданий Function-as-a-Service та також популярний Backend-as-a-Service - але обидва покладаються на хмарні технології. BaaS це такі сервіси які можна використовувати замість написання свого бекенду, наприклад інтерфейси для хмарних баз даних, сервіси авторизації та автентифікації, аналітики, тощо. Такі сервіси можуть бути дуже корисні так званим "rich client" застосуванням, в тому числі веб та мобільним [5]. Зазвичай у типового постачальника хмарних обчислень багато таких сервісів з найрізноманітнішими можливостями.

Однак, під терміном Serverless майже завжди мають на увазі саме FaaS, де розробники пишуть бекенд код, але не підтримують серверну інфраструктуру. Це може дуже нагадувати Platform-as-a-Service, але ці підходи хоча і мають спільні риси, та це пов'язано з тим що обидва спираються на "хмари", а у FaaS є декілька суттєвих відмінностей.

1.1.2. Особливості FaaS

Однією з основних відмінностей FaaS від PaaS є значно менші одиниці розгортання, тобто те що вважається окремим застосунком в хмарі. Для Platform-as-a-Service це окремий сервіс чи мікро-сервіс, який може бути достатньо великим і виконувати декілька пов'язаних задач. В Function-as-a-Service розробники оперують окремими функціями, що приймають певні аргументи та, або щось повертають, або якось впливають на середовище. Ці функції також можуть бути досить великими, але зазвичай постачальники обмежують максимальний час виконання. З такими малими одиницями гнучке масштабування виходить на новий рівень, дозволяючи окремим функціям бути масштабованими. Це може бути дуже корисно якщо, наприклад, мікросервіс має операції на читання та на запис, але перша в рази частіше використовується за другу. Якщо ці дві задачі розділені на окремі функції, то немає жодних перешкод масштабувати їх по-різному [5].

Через те що відбувся перехід до такої значно більш "легкої" абстрації як функція, змінилося і те як такі застосування запускаються та відпрацьовують. На відміну від PaaS, де зазвичай сервіс це веб-сервер що постійно чекає на запити та оброблює їх, функції реагують на певні події (такі як спрацювання таймеру, нове повідомлення в черзі, або той ж HTTP запит), запускаються, відпрацьовують та вимикаються. Наступний виклик тієї ж функції є абсолютно незалежним від попереднього та наступних.

Така модель пов'язана з ще однією важливою особливістю Serverless - на відміну від PaaS, де розробники платять за певні виділені одиниці обчислювальної потужності (такі як Dynos в Heroku [3]), тут оплата

відбувається за час виконання функцій. Тобто, якщо функція виконувалась 10 хвилин, то зменшивши час роботи вдвічі і плата за її виконання зменшиться пропорційно. Це та можливість масштабувати окремі функції дозволяють значно оптимізувати витрати, порівняно з більш класичними рішеннями [5].

Як вже було зазначено вище, такі функції викликаються у відповідь на певні події і спілкуються асинхронно. На відміну від мікросервісів, які є звичайними застосуваннями, наносервіси не можуть просто викликати іншу функцію. Також може бути зрозуміло що не раціонально чекати поки одна функція виконається та тримати запущеною іншу. Перехід до виключно асинхронної комунікації є не простим і може стати недоліком, в залежності від застосунку.

1.1.3. Недоліки FaaS

Справді, хоча і в мікросервісному підході асинхронна комунікація це звичайний паттерн, все ж робота всередині самих сервісів відбувається синхронно. Serverless функціям доводиться користуватись чергами повідомлень, що може негативно вплинути не тільки на простоту та прозорість комунікацій, але і на швидкодію такої звичної операції як виклик функції. Через це є сенс шукати баланс між декомпозицією, простотою підтримки та ефективністю.

Та коли таких функцій стає дуже багато - як і трапляється в звичайному великому застосуванні - навіть якщо їх розмір добре збалансований, може стати складно розуміти як вони між собою пов'язані. Теоретично, будь яка функція може зреагувати на подію створену будь-якою іншою. Це надає великий потенціал до змін та розширення системи через слабку зв'язність, але

навіть звичайний послідовний виклик функцій може бути не просто відслідкувати.

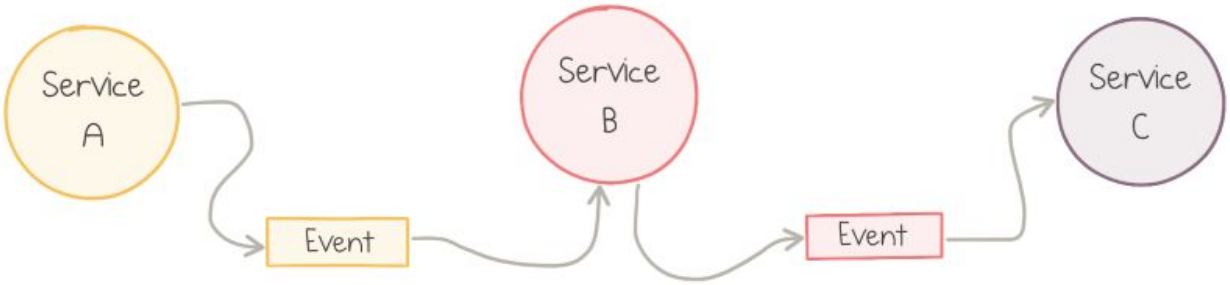

Рисунок 1.1 - Послідовне виконання serverless функцій [4]

Але якщо цей простий приклад ще теоретично можна звести до використання всього однієї функції, більш складні ситуації та паттерни дуже складно реалізувати можливостями лише звичайного FaaS.

Наприклад, обробка помилок та повторні спроби це операції що вимагають певного стану - наступні спроби мають знати про результати та кількість попередніх. Зазвичай FaaS рішення мають вбудовані системи для вирішення подібних задач, але не завжди можна їх достатньо гнучко налаштувати, наприклад встановити експоненціально зростаючий проміжок між повторними спробами, або задати команди на випадок якщо всі спроби були провальні [4].

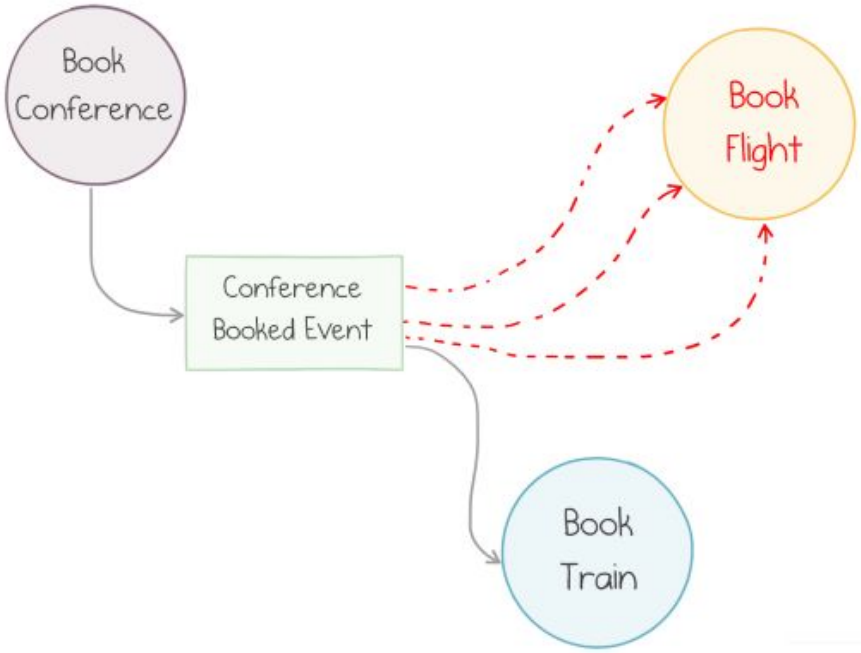

Рисунок 1.2 - Приклад події, після якої йдуть три провальні спроби виконати функцію, після чого виконується інша [4]

Іншою нетривіальною задачею для звичайного FaaS є агрегація результату параллельного виконання функцій. "Викликати" декілька serverless функцій зазвичай не є проблемою, але зібрати результати і обробити вже досить складно [4].

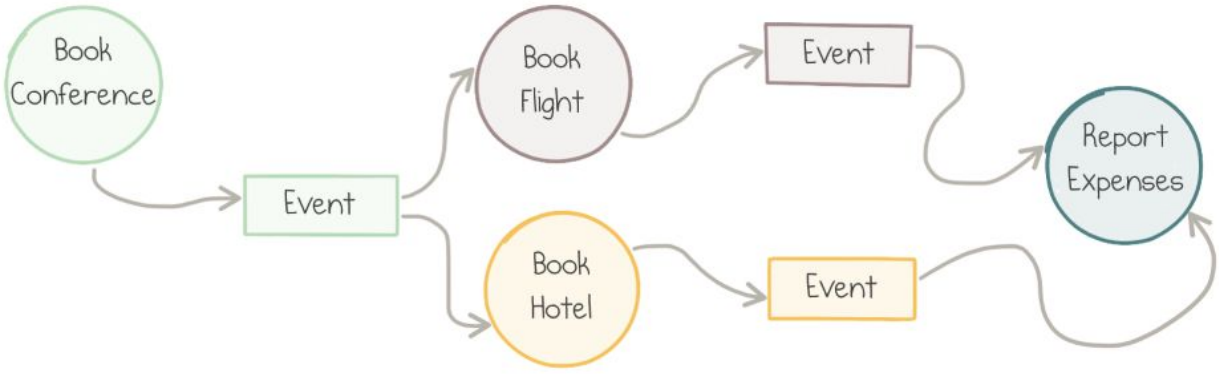

Рисунок 1.3 - Приклад параллельних функцій [4]

Ці проблеми зазвичай вирішуються оркестрацією, яку, в тому чи іншому вигляді, підтримують всі основні постачальники Serverless. Оркестрація - це згрупування викликів декількох функцій в одному місці, що дозволяє не тільки відносно легко відслідковувати порядок викликів функцій та обробки результатів, але і реалізовувати такі не тривіальні для звичайних функцій операції як обробка помилок та повторні спроби, розпаралелювання задачі та інші [4].

Однак оркестрація не може змінити того що функції ефемерні - неможливо надійно зберегти стан між викликами не використовуючи стійкі контейнери на кшталт баз даних. Цей момент може бентежити, однак розробники популярної методології побудови веб-застосувань The Twelve-Factor App, радять "виконувати застосування як один або більше процесів без стану", а також зазначають, що "процеси не розділяють стан" [6].

Останнім з основних болючих питань serverless є те що використання FaaS часто досить сильно прив'язує застосування до постачальника через спеціальний синтаксис, тісні зв'язки з іншими сервісами платформи та іншими специфічними для кожного вендора нюансами. Мігрувати з однієї такої платформи на іншу може бути дуже складно без значних змін, що лякає багатьох розробників, але деякі експерти радять просто прийняти цей факт, обрати надійного постачальника на кшталт Amazon чи Microsoft та використовувати їх потужності на максимум [7].

1.2. Огляд основних постачальників serverless

Лідерами в сфері FaaS вважаються ті ж Amazon та Microsoft, але сильними гравцями також є Tencent, Google, Cloudflare та Alibaba [8]. Amazon є не тільки піонерами FaaS [1] але і лідерами в хмарних обчисленнях з величезною кількістю веб сервісів на власній платформі AWS, але Microsoft зі своєю хмарною платформою Azure, випустивши свою альтернативу AWS Lambda через два роки [9], швидко вийшли на близький рівень. Обидва технологічних гіганти надають неймовірно потужні хмарні рішення з практично рівними можливостями [10].

Але Microsoft має особливість - так звані Durable Functions. Цей метод оркестрації значно потужніший за те що надають інші вендори, так як зазвичай оркестрація serverless функцій (наприклад в AWS Step Functions та Google Cloud Composer) відбувається на досить високому й абстрактному рівні, якого, однак, вистачає для більшості класичних застосувань FaaS. Так в AWS Step Functions оркестрація відбувається за допомогою скінченного автомату станів.

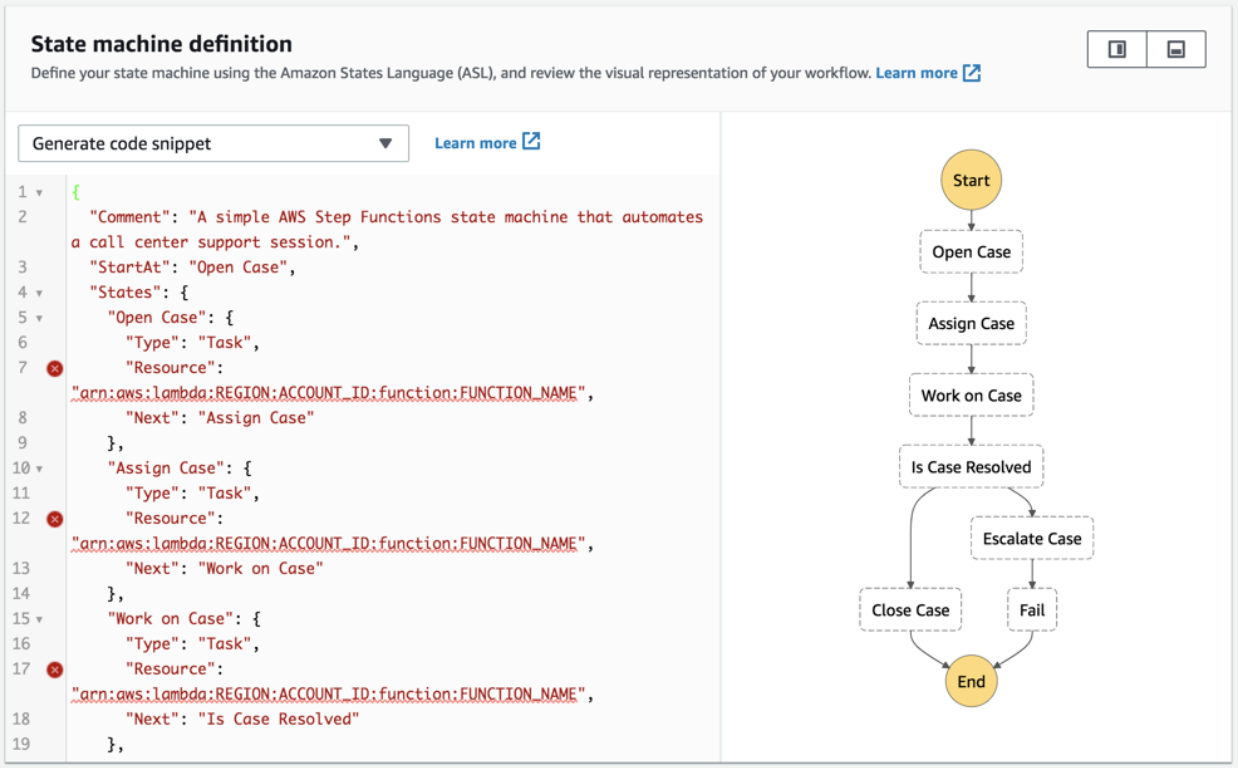

Рисунок 1.4 - оркестрація за допомогою AWS Step Functions [11]

В той же час Microsoft мають не тільки альтернативу такому опису у вигляді Azure Logic Apps, але і більш низькорівневі Azure Durable Functions, що дозволяють описувати складну поведінку прямо в коді, найбільше наближаючи FaaS до наносервісів, на відміну від звичної для них простої "зклейки" інших сервісів хмари.

На думку автора роботи, Microsoft Azure Functions становили найбільший інтерес для дослідження та експериментів саме через цю особливість, тому далі більш детально було розглянуто саме цей інструмент. Однак, більшість з розглянутих паттернів та сценаріїв використання в тому чи іншому вигляді реалізується і в альтернативних середовищах.

1.3. Вступ до Azure Functions

1.3.1. Загальні відомості

Azure Functions є частиною великої хмарної платформи Azure і добре інтегруються з іншими сервісами. Станом на перший квартал 2020 року, вони підтримують такі мови програмування: C#, F#, JavaScript, TypeScript, Java, Python та Powershell. Однак помітно, що більший акцент робиться на перші три, так як в тих же Durable Functions на час написання роботи підтримуються лише вони, хоча і планується підтримувати всі перелічені мови.

1.3.2. Тригери та зв'язки

Функції працюють повністю асинхронно, тому для того щоб їх запустити використовуються так звані тригери - події у відповідь на які виконується функція. Прикладами тригерів є таймер, HTTP запит, нове повідомлення в черзі та запис в базі даних. У кожної функції є один і тільки один тригер, який несе з собою певні дані [13].

Але якщо функції потрібно надіслати кудись результат роботи, або отримати дані більше ніж з одного джерела, використовуються bindings - зв'язки. Є два види - вхідні та вихідні - що дозволяють отримувати на вхід додаткові дані, або виводити їх - наприклад, писати в чергу, в базу даних або повернути відповідь на HTTP запит [13].

1.3.3. Durable Functions

З Durable Functions ситуація дещо складніша - вводиться чотири нові види функцій:

- Оркестратори - функції що визначають порядок та умови виконання дій. Вони абстрагують виклик activity функцій, під-оркестраторів, надсилання сигналів сутностям та інші не тривіальні для звичайних функцій речі. Оркестратори не працюють напряму з тригерами та зв'язками, а мають спеціальний тригер.
- Дії (Activity) - більше схожі на звичайні функції, але можуть бути викливані лише оркестратором.
- Сутності (Durable entity) - функція що має стан. Оперує над певними даними, може змінювати їх у відповідь на сигнали, повертати, або надсилати сигнали іншим сутностям.
- Клієнтські функції - звичайні функції, які реагують на тригери і можуть запускати оркестратори і надсилати сигнали сутностям [14].

Насправді ні оркестратори, ні клієнтські функції не викликають інші на пряму, а лише абстрагують запис в та читання з черг повідомлень.

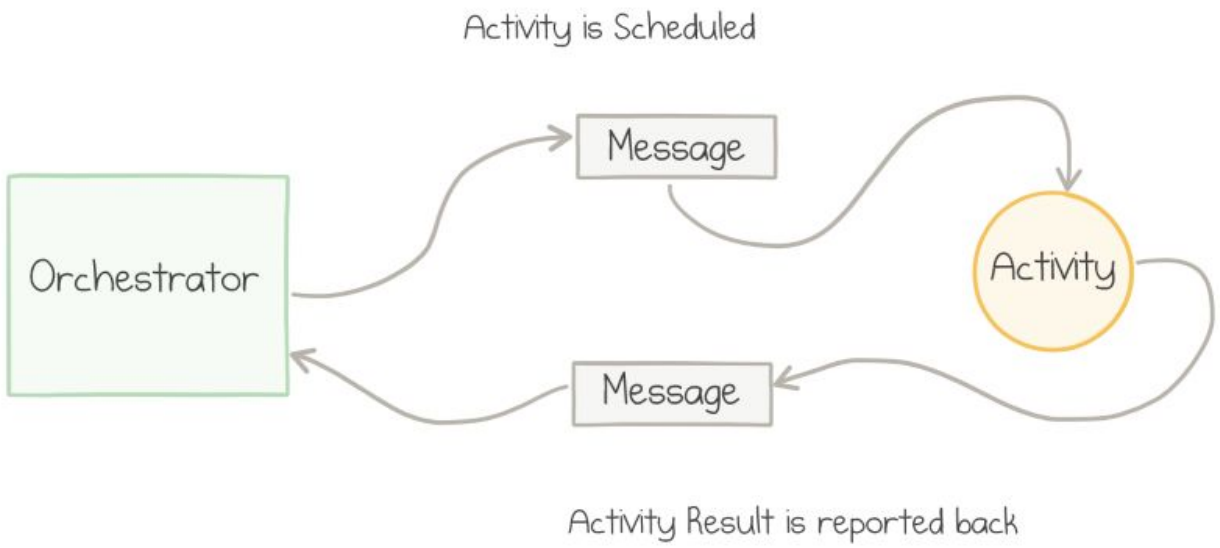

Рисунок 1.5 - Як оркестратор "викликає" activity функцію [4]

Крім переліченого, оркестратори також вміють чекати на зовнішню подію. Але що робити, якщо ця подія станеться через годину, день або тиждень після запуску функції? Навіть якщо б їй дозволялося так довго бути запущеною, це коштувало б дуже багато. Але це не проблема через те як влаштовані ці функції. Коли оркестратор запускає асинхронну дію (виклик іншої функції, очікування на подію, тощо), вона не чекає її завершення, а вимикається. Потім, коли асинхронна дія завершилася, оркестратор запускається з самого початку, але коли доходить до дії, результат якої вже отримував раніше, то не викликає її знов, а йде далі з тим результатом. Це досягається завдяки паттерну even sourcing - всі асинхронні дії та їх результати зберігаються та використовуються для відбудови стану при повторних запусках того ж екземпляра оркестратора [15].

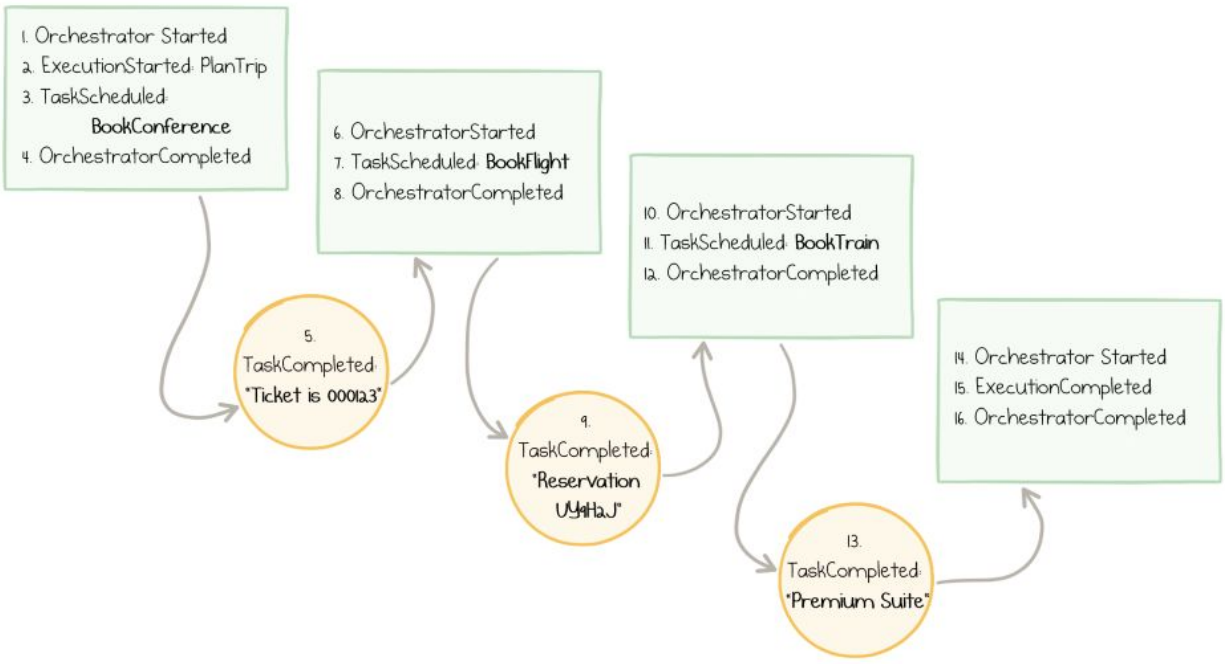

Рисунок 1.6 - Приклад оркестратору з трьома асинхронними операціями [4]

Через таку систему, на оркестатори накладаються певні обмеження - вони мають бути детерміновані. Це означає, що при перезапуску функції і тими ж результатами від асинхронних подій, вона має видавати однаковий результат. Отже, не можна використовувати генерацію випадкових значень, брати поточну дату та час, генерувати унікальні ідентифікатори, робити HTTP запити та виконувати інші не детерміновані операції [16].

2. Типові сценарії використання

2.1. REST наносервіси

Перший паттерн це REST API у вигляді наносервісів. Azure Functions мають тригер по HTTP запитам, та вихідний зв'язок, що дозволяє повертати відповідь на запит. Також досить просто інтегруватись з потужною мульти-модельною базою даних Cosmos DB. Однак, хоча база підтримує багато моделей, найпростіше працювати з SQL варіантом, адже, на момент написання роботи, лише для неї були реалізовані додаткові вхідні та вихідні зв'язки, які будуть продемонстровані далі. Ймовірно, в майбутньому більше моделей будуть підтримувати більш лаконічний синтаксис. Цей та інші приклади будуть реалізовані мовою програмування C#, але не є проблемою імплементувати те саме з допомогою інших мов, які підтримує платформа.

Для демонстрації була взята предметна область блогу, де декілька авторів пишуть пости. Для простоти було вирішено обмежитись трьома операціями: запит по пост за ідентифікатором, створення нового поста та видалення поста. Сутність пост наведена в Додатку Б.

Хоча фреймворк і не обмежує розробників, є сенс створювати окрему функцію на кожну операцію за кращими практиками. Запит на отримання поста за ідентифікатором може виглядати так:

```
public static class GetPost
{
    [FunctionName("GetPost")]
    public static IActionResult RunAsync(
        [HttpTrigger(
            authLevel: AuthorizationLevel.Function,
            "get",
            Route = "posts/{author}/{id}"
        )]
        HttpRequest req,
        [CosmosDB(
            databaseName: ConnectionParams.DatabaseName,
            collectionName: ConnectionParams.CollectionName,
            ConnectionStringSetting = ConnectionParams.DbConnectionStringSetting,
            PartitionKey = "{author}",
            Id = "{id}"
        )]
        Post? post,
        string id,
        ILogger log
    )
    {
        log.LogInformation($"GetPost {post}");

        if (post == null)
        {
            return new NotFoundObjectResult(
                new {message = $"Couldn't find a post with id {id}"}
            );
        }

        return new OkObjectResult(post);
    }
}
```

Рисунок 2.1 - GET запит на отримання поста

Метадані HTTP запиту та з'єднання з базою задаються в C# як атрибути (в Java як анотації, в інших мовах в окремому файлі). Функція на вхід отримує HTTP запит, шуканий пост або null, ідентифікатор із шляху запиту,

та логгер. Згадані раніше додаткові зв'язки з Cosmos DB дозволяють відразу зробити запит до бази за даними переданими в атрибуті. Таким чином можна значно скоротити кількість коду на такі прості операції. ConnectionParams це просто константи, що повторюються в різних функціях.

Функція для створення посту дещо складніша і повний лістинг можна знайти в додатку В, та найважливіші частини наведені тут:

```
[FunctionName("AddPost")]
public static async Task<IActionResult> RunAsync(
    [HttpTrigger(
        authLevel: AuthorizationLevel.Function,
        "post",
        Route = "posts"
    )]
    HttpRequest req,
    [CosmosDB(
        databaseName: ConnectionParams.DatabaseName,
        collectionName: ConnectionParams.CollectionName,
        ConnectionStringSetting = ConnectionParams.DbConnectionStringSetting
    )]
    IAsyncCollector<Post> collector
)
{
    var data = JsonConvert.DeserializeObject<RequestBody?>(
        await new StreamReader(req.Body).ReadToEndAsync()
    );

    // ... валідація ...

    await collector.AddAsync(
        new Post {Author = data.Author!, Body = data.Body!, Title = data.Title!}
    );

    return new OkResult();
}
```

Рисунок 2.2 - POST запит на створення поста

Отже, тепер на вхід функція отримує вже коллектор, що дозволяє записувати дані до бази. Спочатку треба зчитати тіло запиту, провалідувати (упущена частина). Після чого створюється пост та додається в колекцію. Тут же можна зазначити, що якщо користувач надішле запит за неправильною адресою, або з неправильним методом, то йому повернуться відповідь зі статусом 404, як і хотілося б. Це та інші вбудовані додаткові можливості надають функціонал на рівні популярних веб-фреймворків.

Але якщо фреймворк не надає зв'язків для полегшення розробки, завжди можна використати клієнт бази, як продемонстровано на прикладі запиту на видалення:

```
[FunctionName("DeletePost")]
public static async Task<IActionResult> RunAsync(
    [HttpTrigger(
        authLevel: AuthorizationLevel.Function,
        "delete",
        Route = "posts/{author}/{id}"
    )]
    HttpRequest req,
    [CosmosDB(
        databaseName: ConnectionParams.DatabaseName,
        collectionName: ConnectionParams.CollectionName,
        ConnectionStringSetting = ConnectionParams.DbConnectionStringSetting
    )]
    DocumentClient client,
    string author,
    string id
)
{
    var collectionUri = UriFactory.CreateDocumentCollectionUri(
        databaseId: ConnectionParams.DatabaseName,
        collectionId: ConnectionParams.CollectionName
    );
```

```
    var posts = client
                .CreateDocumentQuery<Document>(
                    documentCollectionUri: collectionUri,
                    feedOptions: new FeedOptions{EnableCrossPartitionQuery = true}
                )
                .Where(d => d.Id == id);

    foreach (var doc in posts)
    {
        await client.DeleteDocumentAsync(
            documentLink: doc.SelfLink,
            options: new RequestOptions {PartitionKey = new PartitionKey(author)}
        );
    }

    return new OkResult();
}
```

Рисунок 2.3 - DELETE запит на видалення поста

Цей код вже більше схожий на звичайне застосування, але з деякими особливостями Cosmos DB, такими як шлях до колекції та ключ розподілу. Клієнт можна використовувати замість будь-яких раніше продемонстрованих зв'язків для побудови складніших запитів.

Для такої реалізації вистачило простих функцій, але для більш складних систем можуть знадобитись Durable Functions.

2.2. "Зклейка" сервісів

Найбільш класичний приклад використання serverless - невеликий прошарок логіки між декількома сервісами хмарної платформи. Через свою подіє-керовану природу, FaaS гарно підходить для таких задач - при якійсь події з одного сервісу виконати якусь дію в іншому.

Наприклад, на нові пости в Cosmos DB можна надсилати повідомлення на пошту всім хто підписаний на автора поста, за допомогою сервісу SendGrid:

```
[FunctionName("Emailer")]
public static async Task RunAsync(
    [CosmosDBTrigger(
        databaseName: ConnectionParams.Database,
        collectionName: ConnectionParams.PostsCollection,
        ConnectionStringSetting = ConnectionParams.DbConnectionStringSetting,
        CreateLeaseCollectionIfNotExists = true,
        LeaseCollectionName = "leases"
    )]
    IReadOnlyList<Document> posts,
    [CosmosDB(
        databaseName: ConnectionParams.Database,
        collectionName: ConnectionParams.PostsCollection,
        ConnectionStringSetting = ConnectionParams.DbConnectionStringSetting
    )]
    DocumentClient client,
    [SendGrid(ApiKey = "SendGridApiKey")]
    IAsyncCollector<SendGridMessage> collector
)
{
    var authorsUri = UriFactory.CreateDocumentCollectionUri(
        databaseId: ConnectionParams.Database,
        collectionId: ConnectionParams.AuthorsCollection
    );

    foreach (var doc in posts)
    {
        var author = doc.GetPropertyValue<string>("Author");
```

```
        var messages = client
                        .CreateDocumentQuery<Author>(
                            documentCollectionUri: authorsUri,
                            feedOptions: new FeedOptions {
                                EnableCrossPartitionQuery = true
                            }
                        )
                        .Where(a => a.Name == author)
                        .Select(a => a.Subscribers)
                        .ToList()
                        .FirstOrDefault()
                        .Select(
                            sub => ComposeMessage(
                                from: "i.morenets@ukma.edu.ua",
                                to: sub,
                                author: author,
                                title: doc.GetPropertyValue<string>("Title")
                            )
                        );

        foreach (var message in messages)
            await collector.AddAsync(message);
    }
}
```

Рисунок 2.4 - Функція що на кожний новий пост робить розсилку підписникам автора

Цей приклад використовує тригер Cosmos DB та два зв'язки - вихідний SendGrid та універсальний Cosmos DB. Реалізацію допоміжних функцій та класів упущено, так як вони не так важливі і реалізацію можна зрозуміти з контексту.

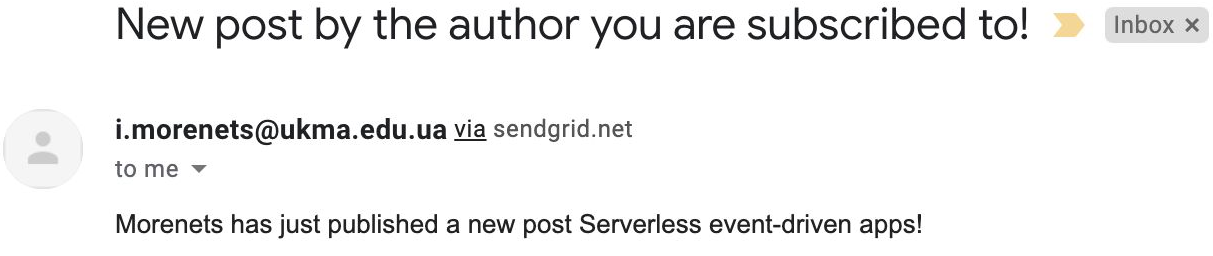

Рисунок 2.5 - Результат виконання попередньої функції

2.3. Розподілені обчислення

Як вже було згадано раніше, немає проблем запустити декілька (або дуже багато) функцій параллельно, але є проблема потім зібрати результати - доведеться використовувати сховище даних, в яке будуть одночасно намагатись записати всі функції, що може бути не ефективно. В вирішенні цієї проблеми допомагає оркестрація, в цьому випадку - Durable Functions. Багато задач можна паралізувати - від виконання декількох незалежних бізнес-операцій до обробки та трансформації великої кількості даних.

Як Proof of Concept було реалізовано програму для підрахунку кількості кожного зі слів у файлі, на основі власної реалізації MapReduce. В результаті вийшло чимало коду, тому інші частини наведені в Додатку В.

Для того щоб запустити алгоритм треба завантажити файл для зчитування в Azure сховище, куди і буде після виконання записаний результат.

```
[FunctionName(nameof(WordCountTrigger))]
public static async Task WordCount_Trigger(
    [BlobTrigger("word-count/input/{name}")]
    string file,
    string name,
    [DurableClient] IDurableOrchestrationClient starter
) =>
    await starter.StartNewAsync(
        orchestratorFunctionName: nameof(WordCount),
        input: new WordCountInput {Name = name, Content = file}
    );
```

Рисунок 2.6 - Тригер на завантаження файлу у сховище

```
public static class WordCount
{
    [FunctionName(nameof(WordCount))]
    public static async Task WordCountOrchestrator(
        [OrchestrationTrigger] IDurableOrchestrationContext ctx
    )
    {
        var input = ctx.GetInput<WordCountInput>();
        var batches = Batching.ToBatches(ToLines(input.Content));

        var mapResults = await Task.WhenAll(
            batches.Select(
                batch => ctx.CallActivityAsync<IList<Result<string, int>>>(
                    functionName: nameof(WordCountMap),
                    input: batch
                )
            )
        );

        var groups = await ctx.CallActivityAsync<IList<Group<string, int>>>(
            functionName: nameof(WordCountGroup),
            input: mapResults
        );

        var reduceResults = await Task.WhenAll(
            groups.Select(
                group => ctx.CallActivityAsync<Result<string, int>>(
                    functionName: nameof(WordCountReduce),
                    input: group
                )
            )
        );

        await ctx.CallActivityAsync<string>(
            functionName: nameof(WordCountOutput),
            input: reduceResults
        );
    }
}
```

Рисунок 2.7 - Реалізація алгоритму Word count на власній MapReduce архітектурі

```
[FunctionName(nameof(WordCountOutput))]
public static async Task WordCount_Output(
   [ActivityTrigger] Result<string, int>[] results,
   [Blob("word-count/output/word-count.txt", FileAccess.Write)]
   TextWriter output
) =>
   await output.WriteAsync(
       string.Join(
           separator: '\n',
           values: results.Select(res => $"{res.Key}:{res.Value}")
       )
   );
```

Рисунок 2.8 - Вивід результату виконання у сховище

Реалізація намагається симулювати класичний MapReduce - вхідні дані рівномірно розбиваються на пакети, що надсилаються map-функціям. Вони повертають списки пар (ключ, значення), які зливаються та розбиваються на групи по ключам. Ці групи розсилаються редьюсерам, які повертають вже одну по одній парі (ключ, значення). Результуючий список пар записується у сховище.

Звичайно, реалізовувати таку архітектуру на Azure Functions не раціонально - платформа не оптимізована під такі задачі і є чіткі жорсткі обмеження на час виконання функцій та розмір вхідних даних. Однак як PoC цей експеримент показує що якщо можна і такий алгоритм реалізувати на serverless платформі, то і більш прості задачі, які зазвичай вимагає бізнес-логіка, скоріш за все, вдастся.

2.4. Модель акторів

Досі паттерни не вимагали стану, не враховуючи оркестрацію. Але деяким застосуванням все ж потрібно працювати зі станом. Більше того, для ефективнішої роботи може використовуватись кілька потоків. Однак, в багатопоточному середовищі складно працювати зі змінними даними, адже треба використовувати механізми синхронізації щоб підтримувати їх консистентність. Особливо популярним та відомим таким засобом є блокування потоків - коли виконання коду в потоці доходить до секції що працює зі спільними даними, поток або блокує секцію та виконує операцію, або призупиняється допоки інший потік, що вже заблокував секцію, не розблокує її [17].

Однак, використання таких механізмів породило ряд нових проблем та їх же вирішень. Типовими проблемами є: додаткові витрати на підтримку блокувань, взаємне блокування, складність розробки та налагодження (debugging). Для вирішення цих проблем було розроблено багато рішень, одним з яких є модель акторів. Ця модель вводить абстрацію актор - сутність, що реагує на повідомлення від інших акторів і може у відповідь:

- Змінити свій стан.
- Надіслати повідомлення іншому актору.
- Створити нового актора.

Для передачі повідомлень використовуються черги, через що відпадає потреба у використанні типових механізмів синхронізації. Більше того, зазвичай інструменти що реалізують таку модель дозволяють відносно просто масштабувати застосування [18].

Однією з таких реалізацій є Azure Durable Entities, які гарантують те що повідомлення буде доставлено, що повідомлення надходять по черзі і що блокування окремих сутностей не блокує функції що надсилають їм повідомлення [19].

Типовим прикладом є симуляція банківських акаунтів, які можна поповнювати, отримувати поточний стан, з яких можна виводити кошти та між якими можна проводити перекази.

```
public class Account : IAccount
{
   public decimal Balance { get; set; } = decimal.Zero;

   public Task Replenish(decimal amount)
   {
       Balance += amount;
       return Task.CompletedTask;
   }

   public Task<bool> Withdraw(decimal amount)
   {
       var canWithdraw = amount <= Balance;

       if (canWithdraw)
           Balance -= amount;

       return Task.FromResult(canWithdraw);
   }

   public Task<decimal> GetBalance() => Task.FromResult(Balance);

   [FunctionName(nameof(Account))]
   public static Task Run([EntityTrigger] IDurableEntityContext ctx) =>
       ctx.DispatchAsync<Account>();
}
```

Рисунок 2.9 - Сутність Банківський аккаунт

Durable Entities підкоряються тим же правилам що і звичайні актори з декількома спрощенями:

- Сутність автоматично створюється при першому повідомленні до неї.

- Надсилати сутностям повідомлення та виклики можуть також клієнтські функції та оркестратори.
- Повідомлення від виклику відлічається тим, що виклик повертає якийсь результат.

```
[FunctionName(nameof(Replenish))]
public static async Task<IActionResult> ReplenishFun(
    [HttpTrigger(AuthorizationLevel.Function, "post", Route = "replenish")]
    HttpRequest req,
    [DurableClient] IDurableEntityClient client
)
{
    var data = JsonConvert.DeserializeObject<RequestBody?>(
        await new StreamReader(req.Body).ReadToEndAsync()
    );

    // validation...

    await client.SignalEntityAsync<IAccount>(
        entityKey: data.Account,
        operation: account => account.Replenish(data.Amount.Value)
    );

    return new OkObjectResult(new {message = "Successfully replenished"});
}
```

Рисунок 2.10 - Клієнтська функція поповнення рахунку надсилає повідомлення сутності

Для прикладу використано звичайний HTTP тригер, який по ідентифікатору сутності (entityKey) надсилає їй повідомлення (сигналізує). Однак, якщо потрібно зробити виклик і отримати його результат, доведеться використовувати оркестратори, адже лише вони вміють очікувати відповідь:

```
[FunctionName(nameof(Withdraw))]
public static async Task<bool> WithdrawFun(
   [OrchestrationTrigger] IDurableOrchestrationContext ctx
)
{
   var input = ctx.GetInput<WithdrawArgs>();
   var account = ctx.CreateEntityProxy<IAccount>(input.Account);

   var wasSuccessful = await account.Withdraw(input.Amount);

   return wasSuccessful;
}
```

Рисунок 2.11 - Оркестратор для зняття коштів

В цьому випадку функція очікує на відповідь від сутності, та повертає результат - чи вдалося вивести кошти. Також тут продемонстрований інший синтаксис що використовує проксі для абстрагування передачі повідомлень.

Але клієнтські функції все ж можуть отримати копію сутності, однак вона може бути не ідеально актуальна, адже це лише знімок стану, що робиться не після кожної операції:

```
public static async Task<IActionResult> GetBalanceFun(
   [HttpTrigger(AuthorizationLevel.Function, "get", Route = "{account}/balance")]
   HttpRequest req,
   string account,
   [DurableClient] IDurableEntityClient client
)
{
   var entity = await client.ReadEntityStateAsync<Account>(
       new EntityId(entityName: nameof(Account), entityKey: account)
   );

   if (!entity.EntityExists)
   {
       return new NotFoundObjectResult(
           new {message = $"Couldn't find the account {account}"}
       );
   }
```

```
    return new OkObjectResult(new {balance = await entity.EntityState.GetBalance()});
}
```

Рисунок 2.12 - Клієнтська функція отримує знімок стану сутності

Ці функції вже гарантують нам консистентність даних при роботі з окремими аккаунтами, але для одночасної узгодженої зміни кількох сутностей все ж потрібні деякі засоби синхронізації, однак зі своїми особливостями. Оркестратори дозволяють створювати так звані критичні секції, що блокують окремі сутності (а не тих хто з ними працює) [19]:

```
[FunctionName(nameof(Transfer))]
public static async Task<bool> TransferFun(
    [OrchestrationTrigger] IDurableOrchestrationContext ctx
)
{
    var input = ctx.GetInput<TransferArgs>();

    var fromEntity = new EntityId(nameof(Account), input.FromAccount);
    var toEntity = new EntityId(nameof(Account), input.ToAccount);

    using (await ctx.LockAsync(fromEntity, toEntity))
    {
        var fromAccount = ctx.CreateEntityProxy<IAccount>(fromEntity);
        var toAccount = ctx.CreateEntityProxy<IAccount>(toEntity);

        var hasEnoughFunds = await fromAccount.Withdraw(input.Amount);

        if (!hasEnoughFunds)
        {
            return false;
        }

        await toAccount.Replenish(input.Amount);
    }

    return true;
}
```

Рисунок 2.13 - Оркестратор що має критичну секцію

Ідея в тому, що всі надіслані повідомлення (та блокування) потрапляють в чергу і будуть оброблені відразу як сутності будуть розблоковані. Також критичні секції гарантують відсутність взаємних блокувань, бо накладають такі обмеження:

- Критичні секції не можна вкладати одна в одну.
- Вони не можуть викликати під-оркестратори та сутності що не заблоковані цією секцію.
- Можуть надсилати повідомлення тільки сутностям які секція НЕ заблокувала.
- Критичні секції не можуть викликати одну сутність використовуючи паралельні виклики [19].

Таким чином, досягається надійність комунікацій в розподіленому параллельному середовищі з максимальною прозорістю для розробника.

Висновки

В результаті виконаної роботи було успішно досліджено serverless як метод та засіб розробки подіє-керованих розподілених застосунків. Також окремо було проаналізовано Azure Functions як одну з найпопулярніших реалізацій Function-as-a-Service.

Крім цього, було розроблено, продемонстровано та описано чотири основні сценарії використання serverless на момент написання роботи, такі як REST наносервіси, зв'язування хмарних сервісів між собою, розподілені обчислення та модель акторів.

Результати роботи можуть бути використані як прикладний практичний посібник по основним поняттям та паттернам serverless.

Як розвиток роботи, може бути розроблене справжнє застосування на serverless платформі з використанням наведених та не тільки паттернів, яким би користувались справжні користувачі, та досліджено які переваги та обмеження FaaS має на практиці.

Список джерел

1. Amazon Launches Lambda, An Event-Driven Compute Service [Електронний ресурс]: Ron Miller. *TechCrunch* – 2014. Перевірено 3 травня 2020. Режим доступу: https://techcrunch.com/2014/11/13/amazon-launches-lambda-an-event-driven-compute-service.
2. Why The Future Of Software And Apps Is Serverless [Електронний ресурс]: Ken Fromm. *Readwrite* – 2012. Перевірено 3 травня 2020. Режим доступу: https://readwrite.com/2012/10/15/why-the-future-of-software-and-apps-is-serverless.
3. Heroku Dynos [Електронний ресурс]: Матеріал з офіційної документації Heroku. Перевірено 3 травня 2020. Режим доступу: https://www.heroku.com/dynos.
4. Making Sense of Azure Durable Functions [Електронний ресурс]: Mikhail Shilkov – 2018. Перевірено 3 травня 2020. Режим доступу: https://mikhail.io/2018/12/making-sense-of-azure-durable-functions.
5. Serverless Architectures [Електронний ресурс]: Mike Roberts. *MartinFowler.com* – 2018. Перевірено 3 травня 2020. Режим доступу: https://martinfowler.com/articles/serverless.html.
6. The Twelve-Factor App. Processes [Електронний ресурс]: Adam Wiggins. *The Twelve-Factor App* - 2017. Перевірено 3 травня 2020. Режим доступу: https://12factor.net/processes.
7. On Serverless, Multi-Cloud, and Vendor Lock In [Електронний ресурс]: Mike Roberts. *Medium* - 2017. Перевірено 3 травня 2020. Режим доступу: https://medium.com/the-symphonium/on-serverless-multi-cloud-and-vendor-lock-in-da930b3993f.

8. The Forrester New Wave™: Function-As-A-Service Platforms, Q1 2020 [Електронний ресурс]: Jeffrey Hammond, Christopher Mines, Abigail Livingston, Kara Hartig. *Forrester* - 2020. Перевірено 3 травня 2020. Режим доступу: https://reprints.forrester.com/#/assets/2/108/RES155938/reports.
9. Microsoft answers AWS Lambda's event-triggered serverless apps with Azure Functions [Електронний ресурс]: Ron Miller. *TechCrunch* – 2016. Перевірено 3 травня 2020. Режим доступу: https://techcrunch.com/2016/03/31/microsoft-answers-aws-lambdas-event-triggered-serverless-apps-with-azure-functions.
10. AWS Lambda vs. Azure Functions: 10 Major Differences [Електронний ресурс]: Mikhail Shilkov. *IAMONDEMAND* – 2019. Перевірено 3 травня 2020. Режим доступу: https://iamondemand.com/blog/aws-lambda-vs-azure-functions-ten-major-differences.
11. Create a Serverless Workflow with AWS Step Functions and AWS Lambda. [Електронний ресурс]: Матеріал з офіційної документації AWS. Перевірено 9 травня 2020. Режим доступу: https://aws.amazon.com/getting-started/hands-on/create-a-serverless-workflow-step-functions-lambda.
12. Supported languages in Azure Functions [Електронний ресурс]: Eamon O'Reilly, Tom Dykstra, Glenn Gailey, Jeff Hollan. *Microsoft Azure Docs* - 2019. Перевірено 9 травня 2020. Режим доступу: https://docs.microsoft.com/en-us/azure/azure-functions/supported-languages.
13. Azure Functions triggers and bindings concepts [Електронний ресурс]: Tom Dykstra, Glenn Gailey, Wesley McSwain, Craig Shoemaker та інші. *Microsoft Azure Docs* - 2019. Перевірено 9 травня 2020. Режим доступу: https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings.

14. Durable Functions types and features [Електронний ресурс]: Chris Gillum, Glenn Gailey, Kristine Toliver та інші. *Microsoft Azure Docs* - 2019. Перевірено 9 травня 2020. Режим доступу: https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-types-features-overview.
15. Durable Orchestrations [Електронний ресурс]: Chris Gillum, Glenn Gailey, Anthony Chu. *Microsoft Azure Docs* - 2019. Перевірено 9 травня 2020. Режим доступу: https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-orchestrations.
16. Orchestrator function code constraints [Електронний ресурс]: Chris Gillum, Glenn Gailey, Anthony Chu та інші. *Microsoft Azure Docs* - 2019. Перевірено 9 травня 2020. Режим доступу: https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-code-constraints.
17. Further Threads Programming:Synchronization [Електронний ресурс]: Dave Marshall. *Офіційний сайт Cardiff School of Computer Science & Informatics* - 1999. Перевірено 9 травня 2020. Режим доступу: http://users.cs.cf.ac.uk/Dave.Marshall/C/node31.html.
18. Hewitt, Meijer and Szyperski: The Actor Model (everything you wanted to know, but were afraid to ask) [Електронний ресурс]: Erik Meijer, Carl Hewitt, Clemens Szyperski. *Channel 9* - 2012. Перевірено 9 травня 2020. Режим доступу: https://channel9.msdn.com/Shows/Going+Deep/Hewitt-Meijer-and-Szyperski-The-Actor-Model-everything-you-wanted-to-know-but-were-afraid-to-ask.
19. Entity functions [Електронний ресурс]: Chris Gillum, Glenn Gailey, Anthony Chu та інші. *Microsoft Azure Docs* - 2019. Перевірено 9 травня 2020. Режим доступу: https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-entities.

Додаток А
(довідниковий)
Перелік прийнятих скорочень

FaaS – Function-as-a-Service.

BaaS – Backend-as-a-Service.

PaaS – Platform-as-a-Service.

HTTP – Hypertext Transfer Protocol.

AWS – Amazon Web Services.

REST – Representational State Transfer.

API – Application Programming Interface.

DB – Database.

SQL – Structured Query Language.

PoC – Proof of Concept.

Додаток Б
(обов’язковий)
Сутність Пост в блозі

```
public class Post
{
   public string Id { get; set; }
   public string Title { get; set; }
   public string Author { get; set; }
   public string Body { get; set; }
   public int? Rating { get; set; }
}
```

Додаток В
(обов'язковий)
Функція для створення посту

```
internal class RequestBody
{
   public string? Title { get; set; }
   public string? Author { get; set; }
   public string? Body { get; set; }
}

public static class AddPost
{
   [FunctionName("AddPost")]
   public static async Task<IActionResult> RunAsync(
       [HttpTrigger(authLevel: AuthorizationLevel.Function, "post",
Route = "posts")]
       HttpRequest req,
       [CosmosDB(
           databaseName: ConnectionParams.DatabaseName,
           collectionName: ConnectionParams.CollectionName,
           ConnectionStringSetting =
ConnectionParams.DbConnectionStringSetting
       )]
       IAsyncCollector<Post> collector,
       ILogger log
   )
   {
       var data = JsonConvert.DeserializeObject<RequestBody?>(
           await new StreamReader(req.Body).ReadToEndAsync()
       );

       log.LogInformation($"AddPost {data}");

       if (data == null)
       {
           return new BadRequestObjectResult(
               new {message = "The body of the request is missing"}
           );
       }
```

```
        if (new List<string?> {data.Author, data.Body, data.Title}.Any(
            string.IsNullOrWhiteSpace
        ))
        {
            return new BadRequestObjectResult(
                new {message = "One of the required fields is missing"}
            );
        }

        await collector.AddAsync(
            new Post {
                 Author = data.Author!,
                 Body = data.Body!,
                 Title = data.Title!
            }
        );

        return new OkResult();
    }
}
```

Додаток Г
(обов'язковий)
Важливі елементи реалізації MapReduce WordCount

```
public class WordCountInput
{
   public string Name { get; set; }
   public string Content { get; set; }
}


public class Result<KeyT, ValueT>
{
   public KeyT Key { get; set; }
   public ValueT Value { get; set; }
}


public class Group<KeyT, ValueT>
{
   public KeyT Key { get; set; }
   public IList<ValueT> Values { get; set; }
}


[FunctionName(nameof(WordCountMap))]
public static IList<Result<string, int>>
   WordCount_Map([ActivityTrigger] IList<string> lines) =>
   lines.SelectMany(ToTerms).Select(ToMapReduceResult).ToList();


[FunctionName(nameof(WordCountReduce))]
public static Result<string, int> WordCount_Reduce(
   [ActivityTrigger] Group<string, int> group
) =>
   new Result<string, int>(key: group.Key, value: group.Values.Sum());
```