Explore Azure Functions

General info

Підходить для:

- Інтеграції систем
- IoT
- Побудова простих API
- Побудова дуже малих мікросервісів

Functions vs Logic Apps

Logic Apps - це класичний приклад workflow сервісів, коли декілька степів об’єднуються в один сценарій. Щоб налаштувати сценарії, потрібно використовувати UI або файли конфігурацій.

Function - це сервіс, який дозволяє імплементувати самодостатній обчислювальний елемент. Декілька функцій можна між собою об’єднувати в сценарії, але це потрібно самому розробляти в коді.

Також Functions та Logic Apps можна використовувати разом. Вони вміють викликати один одного.

Ще деталі:

	Azure Function	Logic Apps
Development	Code-first	Designer-first
Connectivity	Built-in and custom binding type	Large collection of built-in and custom connectors
Actions	Each activity is an Azure Function; write code for activity functions	Large collections of ready-made actions
Monitoring	Azure Application Insights	Azure Portal, Azure Monitor logs
Management	REST API, Visual Studio	Azure Portal, REST API, PowerShell, Visual Studio
Execution context	Can run locally or in the cloud	Supports run-anywhere scenarios

Functions vs WebJobs

Azure Functions побудовані на базі WebJobs SDK. Вони між собою духе схожі. Якщо так подивитись, то Functions як сервіс просто більш розвинуті, пропонують більше можливостей.

Деталі:

	Functions	WebJobs
Serverless app model with automatic scaling	Yes	No
Develop and test in browser	Yes	No
Pay-per-use pricing	Yes	No
Integration with Logic Apps	Yes	No
Trigger events	Timer Storage Queue and Blobs Service Bus CosmoDB Event Hubs HTTP/WebHook Event Grid	Timer Storage Queue and Blobs Service Bus CosmoDB Event Hubs File System

Hosting Plans

Azure пропонує три плани:

- Consumption
- Premium
- App service (Dedicated)

Ці плани контролюють доступні ресурси, скейлінг та різні додаткові можливості.

Consumption

Стандартний план. Деплоймент та розширення відбуваються автоматично. Оплата відбувається лише за витрачені ресурси. Якщо навантаження менше, то інстанси функції видаляються.

Premium

Azure створює пул воркерів, які постійно доступні. Функції працюють на цих заготовлених воркерах. Тому немає паузи, коли функція запускається після простою/паузи. Також ці воркери більш потужні та мають доступ до віртуальних мереж.

Dedicated

Функції запускаються в рамках плану App Service. Ціна також залежить від налаштувань App Service. Підходить для довготривалих сценаріїв (не плутати з Durable Functions - це інше)

Hosting options

- ASE - можливість App Service, коли функція запускається в ізольованому середовищі. Це потрібно, щоб безпечно запускати функції під час високого масштабування
- Kubernetes - інтеграція з AKS

Always On

Якщо використовуєте Dedicated план, то потрібно увімкнути це налаштування. Оскільки App Service не контролює Function, то вона може “заснути” назавжди після простою.

Storage Account requirements

Azure Functions підтримує не всі типи Storage Accounts, оскільки для своєї роботи цей механізм потребує зовнішні механізми.

Краще виділяти Functions в окремий Storage Account, щоб не змішувати роботу з даними (Blobs, Queue). По суті принцип isolation.

Scale Azure Functions

Для Consumption та Premium планів Azure масштабує CPU та Memory додаючи нові хости (машини). Це залежить від кількості запитів.

Один хост для Consumption план має 1.5 GB пам’яті та один CPU. Всі Functions можна об’єднати в Function App. Тоді вони будуть працювати на одному хості та разом масштабуватись.

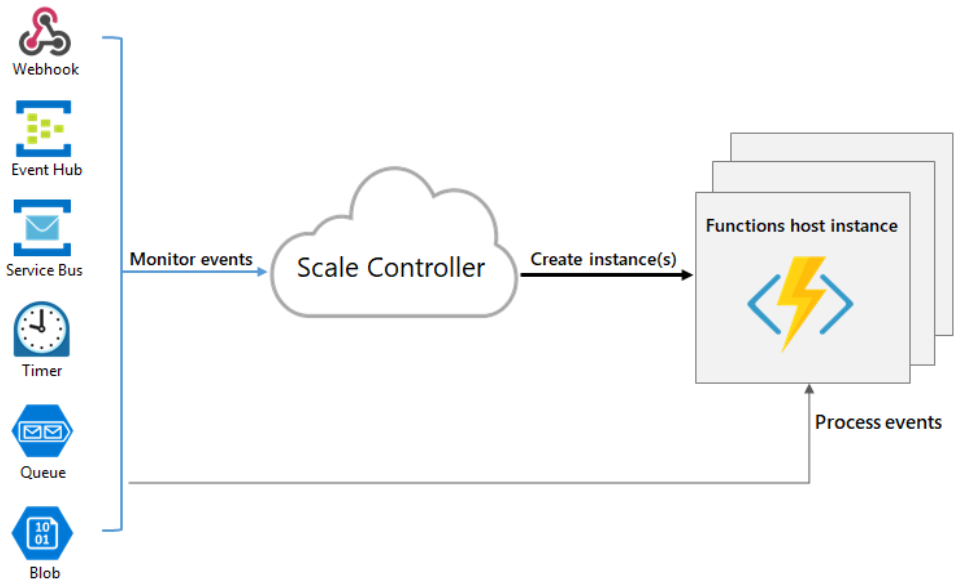
Різні Function Apps масштабуються окремо.

В Premium плані можна вказати розмір ресурсів. Ці налаштування будуть працювати для всіх Function App в рамках одного плана.

Runtime Scaling

Є окремий компонент scale controller, який слідкує за навантаженням та вирішує, коли масштабувати Functions. Він слідкує за різними евристиками в залежності від типу тригеру. Це може бути довжина черги та вік найстарішого повідомлення.

Масштабування відбувається на рівні Function App. Якщо взагалі немає навантаження, то scale controller може вимкнути всі машини.



Scaling behaviors

Є певні обмеження:

- Maximum instances - для одного Function App максимум - 200 хостів. Один хост може процесити декілька запитів одночасно
- New instance rate - для HTTP тригерів система може деплоїти новий хост кожну секунду; для non-HTTP ця величина складає 30 секунд

Limit scale out

Можна встановити обмеження для масштабування. Є налаштування functionAppScaleLimit

Scaling in an App Service plan

В рамках App Service ми самі контролюємо кількість віртуальних машин.

Develop Azure Functions

Кожна Function складається з коду та налаштування в function.json файлі. Цей файл може бути сгенерованим автоматично на основі анотацій в коді.

Цей файл описує тригери, прив'язки та інші налаштування. Кожна функція може мати лише один тригер.

```
{
    "disabled":false,
    "bindings":[
        // ... bindings here
        {
            "type": "bindingType",
            "direction": "in",
            "name": "myParamName",
            // ... more depending on binding
        }
    ]
}
```

Кожен binding потребує 3 параметри:

- type - ім'я прив'язки (наприклад, queueTrigger)
- direction - вказує чи функція отримує ці дані, чи надсилає (in або out)
- name - ім'я аргументу для прив'язки в коді

Folder structure

Простіше створити Functions через студію, щоб точно була правильна структура папок.
https://learn.microsoft.com/en-us/azure/azure-functions/functions-dotnet-class-library#functions-class-library-project

https://learn.microsoft.com/en-us/azure/azure-functions/functions-reference-csharp#folder-structure

Також потрібно створити host.json файл та описати налаштування для запуску функції.

Local development environments

Не потрібно міксувати розробку функцій локально та на Azure Portal в рамках одного Function App. Це може призвести до некоректної роботи системи.

Function локально через connection string може мати доступ до всіх ресурсів в Azure. Тож їх зручно розробляти.

Create triggers and bindings

Більшість тригерів мають свій контекст, який доступний в самій Function.

Binding дозволяє додати до функції інші ресурси. Вони можуть бути input та output. Також вони необов'язкові.

Trigger and binding definitions

- C# - атрибути
- Java - анотації
- JS/TS/Python/PowerShell - лише через function.json файл

Також є UI механізм на Azure Portal, щоб додавати bindings.
В мовах програмування з статичною типізацією тип аргументу вказує на тип прив'язки.

Binding direction

Всі тригери та прив'язки мають direction параметр.

- Для тригерів це завжди in
- Прив'язки можуть як in так і out
- Є спеціальні прив'язки, які підтримують inout direction тип. Це доступно лише через Advanced Editor в Azure Portal

Example

Розглянемо сценарій, коли потрібно додати новий запис в БД. Триггером повинна виступати Azure Queue.

Function.json файл для цього сценарію:

```
{
  "bindings": [
    {
      "type": "queueTrigger",
      "direction": "in",
      "name": "order",
      "queueName": "myqueue-items",
      "connection": "MY_STORAGE_ACCT_APP_SETTING"
    },
    {
      "type": "table",
      "direction": "out",
      "name": "$return",
      "tableName": "outTable",
      "connection": "MY_TABLE_STORAGE_ACCT_APP_SETTING"
    }
  ]
}
```

$return означає, що функція поверне цей результат як return object;

C# script example

Цей скрипт для сценарію вище.

```
using Microsoft.Extensions.Logging;
using Newtonsoft.Json.Linq;

// From an incoming queue message that is a JSON object, add fields and write to Table storage
// The method return value creates a new row in Table Storage
public static Person Run(JObject order, ILogger log)
{
    return new Person() {
            PartitionKey = "Orders",
            RowKey = Guid.NewGuid().ToString(),
            Name = order["Name"].ToString(),
            MobileNumber = order["MobileNumber"].ToString() };
}

public class Person
{
    public string PartitionKey { get; set; }
    public string RowKey { get; set; }
    public string Name { get; set; }
    public string MobileNumber { get; set; }
}
```

Class library example

В цьому прикладі ми використовуємо атрибути замість function.json файлу.

```
public static class QueueTriggerTableOutput
{
    [FunctionName("QueueTriggerTableOutput")]
    [return: Table("outTable", Connection = "MY_TABLE_STORAGE_ACCT_APP_SETTING")]
    public static Person Run(
        [QueueTrigger("myqueue-items", Connection = "MY_STORAGE_ACCT_APP_SETTING")]JObject order,
        ILogger log)
    {
        return new Person() {
                PartitionKey = "Orders",
                RowKey = Guid.NewGuid().ToString(),
                Name = order["Name"].ToString(),
                MobileNumber = order["MobileNumber"].ToString() };
    }
}

public class Person
{
    public string PartitionKey { get; set; }
    public string RowKey { get; set; }
    public string Name { get; set; }
    public string MobileNumber { get; set; }
}
```

Additional resource

https://learn.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-blob

https://learn.microsoft.com/en-us/azure/azure-functions/functions-bindings-cosmosdb-v2

https://learn.microsoft.com/en-us/azure/azure-functions/functions-bindings-timer

https://learn.microsoft.com/en-us/azure/azure-functions/functions-bindings-http-webhook

Connect functions to Azure services

З’єднання до ресурсів відбувається з допомогою connection string. Але їх не можна напряму вказати в function.json файлі. В самому файлі ми вказуємо ім’я змінної, а реальне значення додаємо до environment variables.

Є декілька провайдерів налаштувань. За замовчуванням використовується environment variables, але це можна змінювати в App Settings чи в локальному файлі з налаштуваннями.

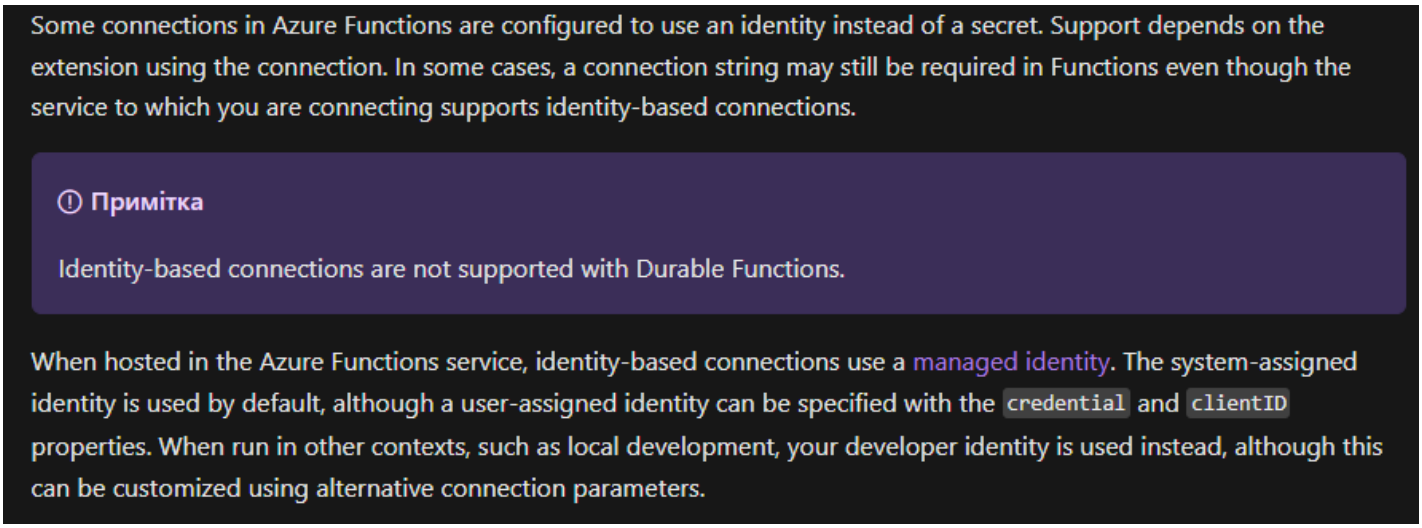
Connection values

Інколи connection string це атомарне значення. Тому достатньо в environment variables вказати MyCustomConnectionString=”value”

Але інколи це може бути цілий об’єкт, тому потрібно використовувати формат MyConnString__QueueName=”***”; MyConnString__StorageAccount=”***”

Configure an identity-based solution

Azure використовує Active Directory для авторизації юзерів. Деякі сервіси також можуть використовувати AD, тому є можливість інтегрувати AD в Functions, щоб отримати доступ до цих сервісів.



Practice

Для локальної розробки потрібно встановити
https://github.com/Azure/azure-functions-core-tools#installing

https://github.com/telamar/AzureLearning/tree/main/src/Functions

Durable functions

Можна реалізовувати різні stateful сценарії з оркестрацією та shared даними.

Application pattern

Коли є складний процес, то його можно розділити на окремі Functions. Самі по собі Functions не мають стану, але в варіанті durable оркестратор зберігає стан та передає його дочірнім функціям. По суті оркестратор - це функція, яка в своєму коді викликає інші функції та передає їм вхідні параметри.

Основні сценарії:

- Function chaining
- Fan-out/fan-in
- Async HTTP APIs
- Monitor
- Human interaction

Function chaining

Найпростіший варіант, коли функції відпрацьовують по черзі. Результат роботи однієї функції стає вхідним параметром для іншої.

Приклад:

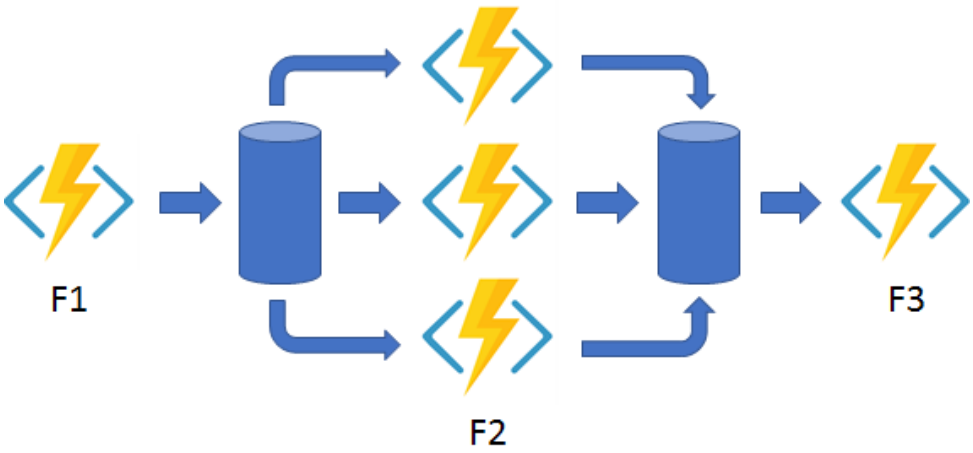
```
[FunctionName("Chaining")]
public static async Task<object> Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    try
    {
        var x = await context.CallActivityAsync<object>("F1", null);
        var y = await context.CallActivityAsync<object>("F2", x);
        var z = await context.CallActivityAsync<object>("F3", y);
        return  await context.CallActivityAsync<object>("F4", z);
    }
    catch (Exception)
    {
        // Error handling or compensation goes here.
    }
}
```

F1, F2 і так далі - імена функцій.

Fan out/fan in

Щось схоже на Map/Reduce.

Запускаємо декілька функцій паралельно, чекаємо поки всі відпрацюють, потім ще можемо запустити функцію агрегатор результату.



Приклад:

```
[FunctionName("FanOutFanIn")]
public static async Task Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    var parallelTasks = new List<Task<int>>();

    // Get a list of N work items to process in parallel.
    object[] workBatch = await context.CallActivityAsync<object[]>("F1", null);
    for (int i = 0; i < workBatch.Length; i++)
    {
        Task<int> task = context.CallActivityAsync<int>("F2", workBatch[i]);
        parallelTasks.Add(task);
    }

    await Task.WhenAll(parallelTasks);

    // Aggregate all N outputs and send the result to F3.
    int sum = parallelTasks.Sum(t => t.Result);
    await context.CallActivityAsync("F3", sum);
}
```

Async HTTP API

Типовий шаблон, коли ми запускаємо background задачу одним HTTP запитом, а потім трекаєм статус операції вже іншим HTTP запитом (наприклад, з taskId).

Для цього є вбудовані типи та підтримка з коробки.

Приклад:

```
public static class HttpStart
{
    [FunctionName("HttpStart")]
    public static async Task<HttpResponseMessage> Run(
        [HttpTrigger(AuthorizationLevel.Function, methods: "post", Route = "orchestrators/{functionName}")] HttpRequestMessage req,
        [DurableClient] IDurableClient starter,
        string functionName,
        ILogger log)
    {
        // Function input comes from the request content.
        object eventData = await req.Content.ReadAsAsync<object>();
        string instanceId = await starter.StartNewAsync(functionName, eventData);

        log.LogInformation($"Started orchestration with ID = '{instanceId}'.");

        return starter.CreateCheckStatusResponse(req, instanceId);
    }
}
```

Monitor

Це шаблон, коли якусь операцію потрібно виконувати постійно з якоюсь періодичністю. Наприклад, кожні 5 хвилин очищати якийсь кастомний кеш. Є багато альтернатив, як це реалізувати, наприклад,cron або timer тригер для звичайної функції.

Приклад, як це реалізувати через durable functions:

```
[FunctionName("MonitorJobStatus")]
public static async Task Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    int jobId = context.GetInput<int>();
    int pollingInterval = GetPollingInterval();
    DateTime expiryTime = GetExpiryTime();

    while (context.CurrentUtcDateTime < expiryTime)
    {
        var jobStatus = await context.CallActivityAsync<string>("GetJobStatus", jobId);
        if (jobStatus == "Completed")
        {
            // Perform an action when a condition is met.
            await context.CallActivityAsync("SendAlert", machineId);
            break;
        }

        // Orchestration sleeps until this time.
        var nextCheck = context.CurrentUtcDateTime.AddSeconds(pollingInterval);
        await context.CreateTimer(nextCheck, CancellationToken.None);
    }

    // Perform more work here, or let the orchestration end.
}
```

Human interaction

Це можуть бути сценарії, коли людині щось потрібно затвердити. В цьому випадку оркестратор може відправити запит на підтвердження, а потім контролювати timeout. Якщо людина не встигла виконати дії, то оркестратор може відправити новий запит.

```
[FunctionName("ApprovalWorkflow")]
public static async Task Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    await context.CallActivityAsync("RequestApproval", null);
    using (var timeoutCts = new CancellationTokenSource())
    {
        DateTime dueTime = context.CurrentUtcDateTime.AddHours(72);
        Task durableTimeout = context.CreateTimer(dueTime, timeoutCts.Token);

        Task<bool> approvalEvent = context.WaitForExternalEvent<bool>("ApprovalEvent");
        if (approvalEvent == await Task.WhenAny(approvalEvent, durableTimeout))
        {
            timeoutCts.Cancel();
            await context.CallActivityAsync("ProcessApproval", approvalEvent.Result);
        }
        else
        {
            await context.CallActivityAsync("Escalate", null);
        }
    }
}
```

Durable function types

- Orchestrator
- Activity
- Entity
- Client

Orchestrator functions

Оркестратори включають взаємодію з activity functions, sub-orchestration, waiting for external events, HTTP, timers. Вони можуть використовувати entity functions.

Приклади коду таких оркестраторів є в розділах вище.

Оркестратори повинні лише викликати інші функції, не потрібно їх перевантажувати чимось іншим.

Є вимоги до коду оркестраторів, щоб вони не падали під час роботи.
https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-code-constraints

Activity functions

По суті основні workers, які можуть виконувати будь-які CPU/Network операції.

Потребують DurableActivityContext як вхідний параметр. Цей вхідний параметр повинен бути один. Якщо даних багато, то можна використовувати Tuple або колекції.

Entity functions

Потрібні для зчитування чи зміни маленького об'єму стану. Для них є окремий entity trigger. Вони не мають обмежень до коду. Якщо оркестратори впливають на стан через потік керування іншими функціями, то entity функції змінюють його напряму.

Доступ до сутностей відбувається через унікальний ідентифікатор. Це пара рядків (strings)…. Шта?)

Операції по зміні стану сутностей вимагають вказати їх ідентифікатор, а також ім'я операції (це також string).

Client functions

Azure має власний task hub для запуску функцій. Але, щоб запустити функцію оркестратор, потрібно в цей task hub помістити новий запит (event). Це не можна зробити руками в Azure Portal. Єдиний варіант, це написати функцію client. Яка по суті буде виступати тригером для старту всього сценарію.

Це може бути сама звичайна НЕ durable функція.

Task hubs

Task hub - це логічний контейнер для durable функцій, який зберігає їх ресурси. Durable функцію можуть взаємодіяти одна з одною лише, якщо вони належать до одного task hub.

Якщо в Storage Container багато Function Apps, то для кожного потрібно налаштувати свій унікальний task hub.

Azure Storage resources

Task hub включає в себе наступні ресурси:

- Control queues
- One work-item queue
- One history table
- One instances table
- One storage container with one or more lease blobs
- Storage container with large message payloads, if applicable

Всі ресурси Azure виділяє та менеджить автоматично.

Task hub names

Правила:

- Лише літери та цифри
- Починається з літери
- Довжина від 3 до 45 символів

Ім’я task hub потрібно вказати в host.json файлі

```
{
  "version": "2.0",
  "extensions": {
    "durableTask": {
      "hubName": "MyTaskHub"
    }
  }
}
```

Durable orchestrations

Оркестратори можуть викликати інші durable функції. Це можуть бути складні сценарії з ієрархією.

Оркестратор має наступні характеристики:

- Описує сценарій звичайним процедурним кодом. Ніяких схем чи дизайнерів не потрібно
- Оркестратор може викликати інші durable функції синхронно та асинхронно. Output можна зберігати в локальні змінні
- Оркестратор durable та reliable. Статус та прогрес автоматично зберігається в ресурсах task hub, коли ми викликаємо await чи yield. Також task hub зберігає локальний стейт функції, тож навіть перезапуск віртуальної машини нічого не зламає
- Оркестратори можуть буди довготривалими. Це можуть бути секунди, дні, місяці, ніколи

Orchestration identity

Кожен інстант оркестратора має власний ID. Це автосгенерований GUID. Цю логіку можна змінити.

Цей ID потім можна використовувати для діагностики, аналітики та пошуку проблем. Тому краще зберігати ID десь в іншому місці, щоб не втратити.

Reliability

Під капотом імплементовано event sourcing та використано append-only сховище. Тому кожен виклик оркестратором іншої функції чи таймеру додає в це сховище новий event. Команда Azure кастомізувала .NET Task Scheduler. Коли ми викликаємо await, то управління потоком повертається до Durable Task Framework dispatcher. Цей dispatcher зберігає стейт оркестратора в окремій таблиці, а також додає до schedule queue нове повідомлення, щоб запустити дочірню функцію. Після цього основний оркестратор можна вигрузити з пам'яті.

Коли функція оркестратор може продовжити свою роботу, то фреймворк викличе її код з нуля. Якщо оркестратор хоче викликати дочірню функцію, то thread dispatcher дивиться в таблицю з історією викликів. Якщо дочірня функція вже була викликана, то він просто поверне результат роботи.

Features and patterns

- Sub-orchestration - оркестратор може викликати інші оркестратори
- Durable timers - оркестратори мають власний таймер, який дозволяє робити затримки та таймаути. Їх обов'язково потрібно використовувати замість Thread.Sleep чи Task.Delay
- External events - оркестратор може чекати поки не виповниться зовнішня операція (наприклад, взаємодія з людиною)
- Error handling - оркестратор підтримує стандартні try/catch
- Critical sections - оркестратор однопоточний, не потрібно думати про race condition. Але ця проблема може вилізти при взаємодії з зовнішніми системами. Для таких випадків потрібно використовувати LockAsync метод
- Calling HTTP endpoints - оркестратор не повинен робити I/O. В цьому випадку потрібно зробити окрему activity function та через неї відправляти запит
- Passing multiple parameters - потрібно використовувати Tuple чи колекції

Timing in Durable Functions

В арсеналі є durable timer. Він має всі потрібні методи для затримок та таймаутів. Для його створення є метод CreateTimer. Цей метод повертає таску, яка виповниться в конкретний момент часу.

Timer limitations

Коли ми створюємо таймер на конкретний час, то фреймворк додає в чергу запит (повідомлення), яке стане видим лише в цей час. Якщо використовується Consumption план для функцій, то можлива ситуація, коли всі віртуальні машини вимкнені. Це не проблема, як тільки повідомлення стане видимим, фреймворк одразу запустить віртуальну машину.

Usage for delay

```
[FunctionName("BillingIssuer")]
public static async Task Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    for (int i = 0; i < 10; i++)
    {
        DateTime deadline = context.CurrentUtcDateTime.Add(TimeSpan.FromDays(i + 1));
        await context.CreateTimer(deadline, CancellationToken.None);
        await context.CallActivityAsync("SendBillingEvent");
    }
}
```

Usage for timeout

```
[FunctionName("TryGetQuote")]
public static async Task<bool> Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    TimeSpan timeout = TimeSpan.FromSeconds(30);
    DateTime deadline = context.CurrentUtcDateTime.Add(timeout);

    using (var cts = new CancellationTokenSource())
    {
        Task activityTask = context.CallActivityAsync("GetQuote");
        Task timeoutTask = context.CreateTimer(deadline, cts.Token);

        Task winner = await Task.WhenAny(activityTask, timeoutTask);
        if (winner == activityTask)
        {
            // success case
            cts.Cancel();
            return true;
        }
        else
        {
            // timeout case
            return false;
        }
    }
}
```

В цьому прикладі використовується cancellation token. Тут є важливий момент, що якщо якась дочірня функція вже запущена, то фреймворк НЕ буде її зупиняти. Оркестратор просто проігнорує результат роботи функції та піде далі.

Wait for events

```
[FunctionName("BudgetApproval")]
public static async Task Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    bool approved = await context.WaitForExternalEvent<bool>("Approval");
    if (approved)
    {
        // approval granted - do the approved action
    }
    else
    {
        // approval denied - send a notification
    }
}
```

Send events

Метод RaiseEventAsync потребує назви події та дані, які можна серіалізувати в JSON.

В прикладі нище імплементовано client функцію, яка відпрацьовує від queue та додає результат події. В свою чергу оркестратор вже чекає на її результат.

Під капотом метод RaiseEventAsync вичитує повідомлення створене оркестратором. Якщо в цьому повідомленні механізм бачить, що оркестратор чекає на результат саме нашої події, то йому одразу передаються ці дані. Якщо ж оркестратор ще не дійшов до очікування нашої події, то ці дані додаються до in memory queue, де вони чекають свого часу.

```
[FunctionName("ApprovalQueueProcessor")]
public static async Task Run(
    [QueueTrigger("approval-queue")] string instanceId,
    [DurableClient] IDurableOrchestrationClient client)
{
    await client.RaiseEventAsync(instanceId, "Approval", true);
}
```