F3

Fakulta elektrotechnická
Katedra počítačů

Diplomová práce

Řešení pro migraci souborů do cloudu

Migrace souborů z on-premises NAS na platformu Microsoft SharePoint

Bc. Jakub Adamík

Květen 2024
Vedoucí práce: Ing. Karel Frajták, Ph.D.

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Adamík** Jméno: **Jakub** Osobní číslo: **483423**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Řešení pro migraci souborů do cloudu

Název diplomové práce anglicky:

Solution for migration of files into the cloud

Pokyny pro vypracování:

Cílem práce je navrhnout a implementovat řešení pro migraci souborů do cloudu podle požadavků zákazníka. Navrhněte a implementujte řešení pro migraci souborů. Řešení nasaďte a provozujte u zákazníka. Popište business zadání a organizaci projektu, návrh, architekturu a technické řešení. Dále popište služby, návrhové vzory použité v řešení. Diskutujte vhodnost použitých služeb a návrhových vzorů. Diskutujte zda dodané řešení splňuje požadavky zákazníka. Navrhněte vhodný testovací plán pro řešení a řešení otestujte.

Seznam doporučené literatury:

AMIN, Ruhul; VADLAMUDI, Siddhartha; RAHAMAN, Md Mahbubur. Opportunities and challenges of data migration in cloud. Engineering International, 2021, 9.1: 41-50.
SOEWITO, Benfano, et al. A systematic literature Review: Risk analysis in cloud migration. Journal of King Saud University-Computer and Information Sciences, 2022, 34.6: 3111-3120.
ZHAO, Jun-Feng; ZHOU, Jian-Tao. Strategies and methods for cloud migration. international Journal of Automation and Computing, 2014, 11.2: 143-152.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Karel Frajták, Ph.D. laboratoř inteligentního testování systémů FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **05.02.2024** Termín odevzdání diplomové práce: _____

Platnost zadání diplomové práce: **21.09.2025**

Ing. Karel Frajták, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.	
Datum převzetí zadání	Podpis studenta

Poděkování / Prohlášení

Děkuji rodině, partnerce a přátelům, kteří mě podporovali během studia. Především děkuji svému vedoucímu diplomové práce Ing. Karlu Frajtákovi, Ph.D. za cenné rady a podnětné připomínky při vypracování mojí práce. Dále děkuji všem vyučujícím, kteří mě inspirovali a vedli na cestě za věděním.

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 23. 05. 2024

.....

Abstrakt / Abstract

Organizace migrují do cloudu za účelem snížení nákladů na infrastrukturu a zvýšení spolehlivosti svých služeb. V rámci migračního projektu do Microsoft 365 byl zadán požadavek na vývoj řešení pro migraci souborů z NAS na platformu SharePoint. Byla navržena, implementována a nasazena dvě řešení – Migrator a SharePoint protector. Konzolová aplikace Migrator migruje soubory z NAS na SharePoint pomocí Microsoft Graph API. Před nahráním na SharePoint jsou soubory klasifikovány pomocí AIP štítků. Řešení SharePoint protector klasifikuje soubory, které jsou na SharePoint nahrány uživatelem po aktu migrace. Je implementováno pomocí Azure Functions a provozované na platformě Microsoft Azure. Obě řešení byla akceptována, úspěšně nasazena a jsou provozována v prostředí zákazníka. V rámci práce vznikly tři *open source* GitHub repozitáře – adamijak/cosmos, adamijak/azure-auth, adamijak/azure-http. Repozitář adamijak/cosmos obsahuje implementaci pro zjednodušení práce s Azure Cosmos DB. Repozitáře adamijak/azure-auth a adamijak/azure-http obsahují kód a nástroje, které umožňují testovat Microsoft Graph API a služby Microsoft 365.

Klíčová slova: migrace, cloud, Microsoft, Azure, SharePoint, AIP

Organizations are migrating to the cloud to reduce infrastructure costs and increase the reliability of their services. As part of a migration project to Microsoft 365, a requirement was made to develop a solution to migrate files from NAS to the SharePoint platform. Two solutions were designed, implemented and deployed – Migrator and SharePoint protector. The Migrator console application migrates files from NAS to SharePoint using the Microsoft Graph API. Before uploading to SharePoint, the files are classified with AIP labels. The SharePoint protector solution classifies files that are uploaded to SharePoint by the user after the act of migration. It is implemented using Azure Functions and runs on the Microsoft Azure platform. Both solutions have been accepted, successfully, deployed and are running in the customer's environment. Three *open source* GitHub repositories were created as part of the work – adamijak/cosmos, adamijak/azure-auth, adamijak/azure-http. The adamijak/cosmos repository contains an implementation to simplify working with Azure Cosmos DB. The adamijak/azure-auth and adamijak/azure-http repositories contain code and tools that allow you to test the Microsoft Graph API and Microsoft 365 services.

Keywords: migration, cloud, Microsoft, Azure, SharePoint, AIP

Title translation: Solution for migration of files into the cloud (File migration from on-premises NAS to Microsoft SharePoint platform)

Obsah /

1 Úvod **1**

2 Microsoft Azure **2**
2.1 Hierarchie Microsoft Azure . . . 3
2.2 Azure Functions 4
2.2.1 Vertikální a horizontální škálování 5
2.2.2 Škálovatelnost 6
2.3 Azure Storage 8
2.4 Azure App Service 8
2.5 Azure Key Vault 9
2.6 Azure Cosmos DB 9
2.7 Azure App Configuration . . . 10
2.8 Azure Application Insights . . . 11
2.9 Azure Event Grid 11
2.10 Microsoft Entra ID 12

3 Microsoft Graph **13**
3.1 Autentizace 14
3.2 Autorizace 14
3.3 JWT 15
3.4 Microsoft Graph SDK 15
3.4.1 OpenAPI 15
3.5 azure-http 16
3.6 azure-auth 17

4 adamijak/cosmos **19**
4.1 Paralelní a asynchronní programování 19
4.1.1 Asynchronní programování v C# 21
4.1.2 Řízení souběžně běžících úloh 21
4.2 CosmosBenchmark 23
4.3 Adamijak.Cosmos.Extensions . 24
4.3.1 FeedIteratorExtensions . . 25
4.3.2 AsyncEnumerableExtensions 26
4.4 Adamijak.Cosmos.Queue 27
4.4.1 Metoda ToAsyncEnumerable 27
4.4.2 Metoda DeleteAsync 27
4.4.3 Fronty zpráv vs CosmosQueue 28
4.5 CosmosTest 28
4.5.1 GitHub Actions 29
4.6 NuGet Adamijak.Cosmos . . . 29

5 Řešení Migrator **30**
5.1 Azure Information Protection . 31
5.2 Získání složky k migraci 31
5.2.1 Optimistic concurrency control 31
5.3 Detekce souborů 33
5.4 Migrace souborů 34
5.5 Dokončení migrace 35
5.6 Nasazení 35
5.7 Konfigurace 35
5.7.1 INI formát 35
5.8 Logování 36
5.9 Testování 36
5.10 Problémy s migrovanými soubory 36

6 Řešení SharePoint protector **37**
6.1 Funkcionalita SharePoint protector 37
6.2 Funkcionalita Externí sdílení . 38
6.3 Funkcionalita SharePoint unprotector 39
6.4 Konfigurace 40
6.4.1 DefaultAzureCredential . . 40
6.5 Nasazení 41
6.5.1 Dev vs Prod prostředí . . . 41
6.6 Testování 41

7 Závěr **42**

Literatura **43**

A Slovník zkratek **47**

B Funkce SharePointProtector **48**

Tabulky / Obrázky

2.1 IaaS, PaaS a SaaS 2
2.2 Cloud platformy 3
2.3 Hierarchie Microsoft Azure 4
2.4 Vertikální škálování 5
2.5 Horizontální škálování 6
2.6 Horizontální škálování Azure Functions 6
2.7 Škálovací krychle 7
2.8 Rozdělení podle jména 7
2.9 *Round robin* rozdělení 8
2.10 Azure Application Insights 11
2.11 Návrhový vzor publisher-subscriber 12
3.1 Microsoft Graph API 13
3.2 Autentizační flow 14
4.1 Sériový výpočet na jednom vlákně 19
4.2 Sériový výpočet paralelně na dvou vláknech 20
4.3 Asynchronní výpočet na jednom vlákně 20
4.4 Asynchronní výpočet na dvou vláknech 20
4.5 *Chunk based* zpracování úloh . . 22
4.6 *Set based* zpracování úloh 23
4.7 Volání funkcí při použití `IAsyncEnumerable<T>` 26
5.1 Architektura řešení Migrator . . 30
5.2 Optimistic concurrency control 32
5.3 Stromová struktura složek 34
5.4 Zploštěná struktura složek 34
6.1 Funkcionalita SharePoint protector 38
6.2 Zpracovaná žádost na externí sdílení 38
6.3 Funkcionalita Externí sdílení . . 39
6.4 Funkcionalita SharePoint unprotector 40

4.1 Azure Cosmos DB benchmark . 24
4.2 Systém použitý pro benchmark 24

Kapitola 1
Úvod

Migrace do cloudu je v dnešní době velmi aktuální téma, protože firmy stále více chtějí využívat výhod cloudových služeb. Díky těmto výhodám mohou zvýšit svoji efektivitu, flexibilitu a snížit svoje provozní náklady. Migraci do cloudu podléhají nejen aplikační řešení, ale i pracovní prostředí. Příkladem pracovního prostředí může být Microsoft 365, jehož součástí jsou například služby SharePoint, Exchange a Teams. Migrovaná data mohou být různého formátu, typu a zdroje. V rámci migrace pracovního prostředí lze migrovat například kalendáře, e-maily, soubory nebo skupiny uživatelů. Zdrojem a cílem migrace může být on-premises[1] či cloudová platforma. Například lze migrovat data z on-premises do Microsoft Azure nebo mezi cloudy různých poskytovatelů – z Microsoft Azure do Amazon Web Service nebo mezi organizacemi u stejného poskytovatele – z Microsoft Azure do Microsoft Azure. Zdroje a cíle se mohou lišit nejen mezi poskytovateli, ale i mezi typem služby. Například může být požadováno zmigrovat zprávy z Microsoft Teams do e-mailu.

Každý zákazník má specifické požadavky, proto je pro migraci dat do cloudu velmi obtížné navrhnout univerzální, systémové řešení. Za tímto účelem vznikají řešení, která jsou zákazníkovi šitá na míru. Největší výzvou migrace do cloudu je modelování zákazníkových požadavků na migrovanou platformu, což vyžaduje dobrou znalost obou prostředí – zákazníkova i cloudového.

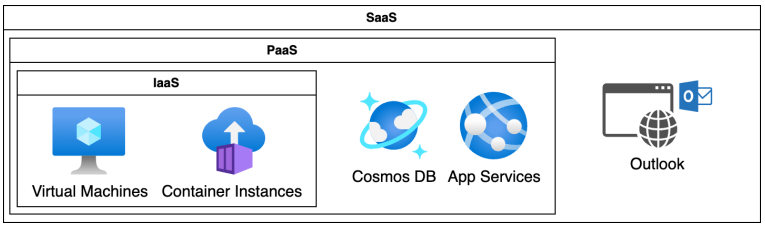
V rámci migračního projektu do Microsoft 365 zadal zákazník firmě GreyCorbel[2] požadavek na migraci souborů z NAS serveru na platformu Microsoft SharePoint. Microsoft SharePoint umožňuje ukládat, spravovat a sdílet dokumenty se skupinami uživatelů nebo anonymně pomocí odkazu pro sdílení souborů. Tato funkcionalita umožňuje snadno kolaborovat na dokumentech, ale zároveň je bezpečnostní hrozbou. Proto jedním z požadavků na migraci bylo, že soubory uložené na cloudu musí být zabezpečeny proti neoprávněnému přístupu použitím AIP. AIP je popsáno v Kapitole 5.1.

Tato práce popisuje návrh a vývoj migračního řešení, které je rozdělené na dvě části – Migrator a SharePoint protector. Tyto části jsou popsány v Kapitole 5 a v Kapitole 6. Řešení SharePoint protector bylo postaveno nad cloudovou platformou Microsoft Azure, která je popsána v Kapitole 2. Pro komunikaci s Microsoft 365 službami je použit Microsoft Graph, který je popsán v Kapitole 3. Řešení využívá implementace z repozitáře adamijak/cosmos, který je popsán v Kapitole 4.

[1] infrastruktura fyzicky spravována a provozována v sídle organizace
[2] `https://www.greycorbel.com/`

Kapitola 2
Microsoft Azure

Microsoft Azure[1] je cloudová platforma, která nabízí široké spektrum nástrojů a služeb pro vývoj a provoz řešení v cloudu v globálním měřítku. Mezi typy služeb které Microsoft Azure nabízí, patří například *Infrastructure as a Service* a *Platform as a Service*. Při použití služeb typu *Infrastructure as a Service* poskytovatel spravuje hardware, virtualizaci a síťovou infrastrukturu. U služeb typu *Platform as a Service* poskytovatel navíc spravuje operační systém, middleware a runtime. Obrázek 2.1 ilustruje, jak se tyto modely protínají. Příkladem služeb poskytovaných Microsoft Azure jsou virtuální stroje, relační a NoSQL databáze, web hosting, serverless computing, distribuovaná souborová úložiště, platformy pro analýzu dat a AI služby. Tyto služby často staví nad *open source* nástroji, jako je například Redis nebo PostgreSQL. [1–3]

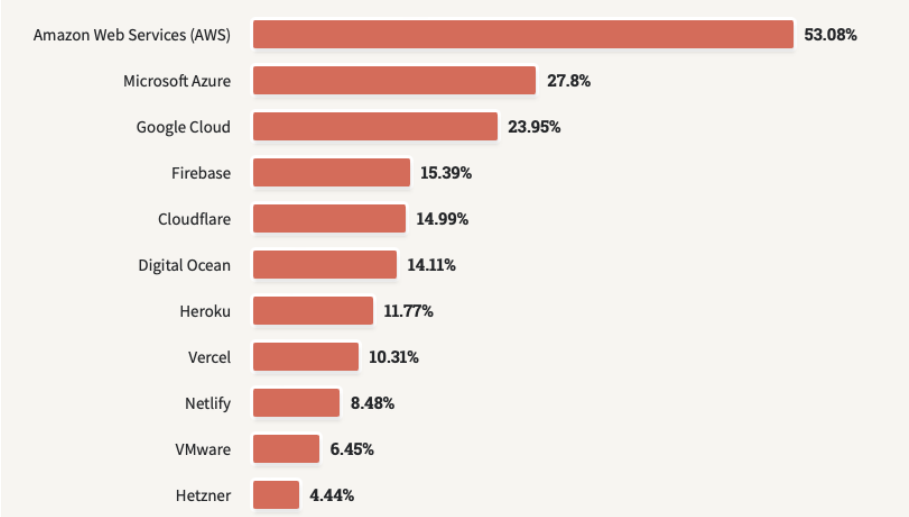


Obrázek 2.1. *Infrastructure as a Service*, *Platform as a Service* a *Software as a Service*

Mezi hlavní konkurenty Microsoft Azure patří například platforma Amazon Web Services[2] nebo Google Cloud Platform[3]. Existují i další platformy například Firebase[4], Digital Ocean[5] nebo Oracle Cloud Infrastructure[6]. Každá z těchto platforem nabízí jiné množství a typy služeb. Obrázek 2.2 zobrazuje poměr počtu uživatelů cloudové platformy vůči celkovému počtu respondentů z průzkumu Stack Overflow. [4–5]

Přední výhodou využití cloudové platformy je snížení nákladů na infrastrukturu a zkrácení doby dodání funkčního software. Například pro nasazení databáze si uživatel při použití cloudové platformy nemusí stavět datacentrum. Nasazení databáze lze v Microsoft Azure vyřídit založením účtu, registrací kreditní karty a vyplněním jednoduchého formuláře. Díky tomu lze zkrátit dobu od zadání úlohy k dokončení nasazení z řádu měsíců do řádu minut. Další výhodou je usnadnění škálování služeb. Zejména lze snadno škálovat služby, které nabízejí cenový model pay-as-you-go. Tyto služby často nabízejí *free tier* a umožňují uživateli provozovat aplikaci v nízkém vytížení za velmi nízké až žádné náklady, což je vhodné zejména pro vývoj prototypů. Pokud se vytížení aplikace zvýší, uživatel má možnost služby škálovat podle potřeby. Microsoft Azure nabízí výhodu oproti konkurenci v integraci s existujícími službami od společnosti Microsoft. Příkladem těchto služeb je Microsoft Windows nebo nástroje

[1] `https://azure.microsoft.com`
[2] `https://aws.amazon.com`
[3] `https://cloud.google.com`
[4] `https://firebase.google.com`
[5] `https://www.digitalocean.com`
[6] `https://www.oracle.com/cloud`



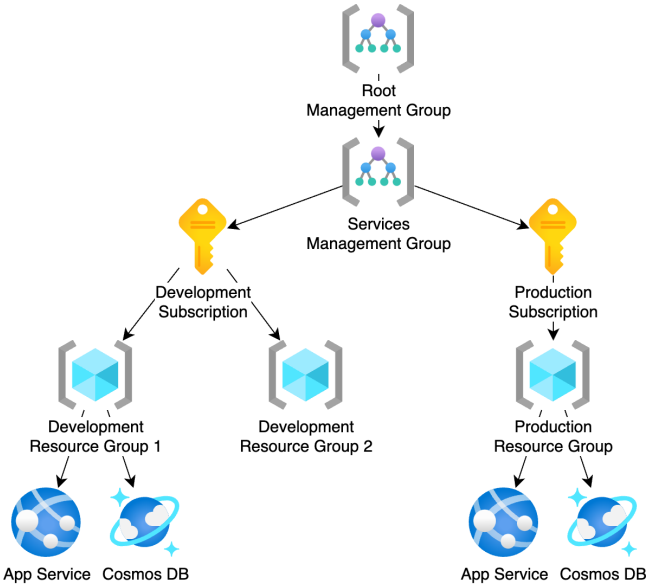
Obrázek 2.2. *Na kterých cloudových platformách jste v uplynulém roce intenzivně pracovali a hodláte pracovat v roce příštím?* – odpovědi 55540 profesionálních vývojářů [5]

Microsoft 365 – Outlook, OneDrive, SharePoint, Teams, Word, Excel PowerPoint. To je atraktivní zejména pro enterprise zákazníky, kteří jsou největšími spotřebiteli Microsoft Azure. Díky těmto výhodám je Microsoft Azure volen start-upy a společnostmi jako prostředí, ve kterém vyvíjí prototypy, nebo do kterého migrují celá řešení. Tento trend má za důsledek to, že cloudové platformy stále více dominují tradičním on-premises datacenterům [1, 6].

Migrace do cloudu však přináší i řadu nevýhod. Velkou nevýhodou je právě jednoduchost vytváření instancí služeb. Při nedostatečné znalosti služby a jejího cenového modelu lze v cloudu snadno utratit vysoké částky. V nejhorším případě se zákazník útratu dozví až z výpisu kreditní karty. Další nevýhodou je *vendor lock-in*. Pro maximální využití platformy je nejlepší využít nativních služeb a funkcionalit. To znamená, že zákazník je motivován modelovat svá řešení závislá na dané platformě a dochází tak k *vendor lock-in*. Poskytovatel cloudových služeb určuje podmínky pro využívání služeb, které může v čase měnit. Pro zákazníka může být v případě nesouhlasu s novými podmínkami velmi obtížné změnit poskytovatele cloudových služeb či migrovat zpět do on-premises. [6–7]

2.1 Hierarchie Microsoft Azure

V Microsoft Azure se rozlišují čtyři úrovně hierarchie správy zdrojů – resources, resource groups, subscriptions a management groups. Jejich hierarchie je vyobrazena na Obrázku 2.3. Za resource je považována jednotlivá instance cloudové služby, jako je například Azure Cosmos DB account nebo Azure App Service. Aby bylo možné vytvořit resource, je nutné mít založené resource group a subscription, pod které bude daný resource patřit. Resource groups slouží k seskupení jednotlivých zdrojů pod jednou subscription a umožňují spravovat a nastavovat hromadně několik resources. Standardně jedna resource group spravuje dohromady zdroje jednoho řešení. Subscription je logická jednotka, která slouží k účtování nákladů za využité podřazené zdroje a licence. Subscriptions nemohou být do sebe vnořené. Management groups spravují politiky, přístupy a compliance pro subscriptions a ostatní management groups. Management groups do sebe vnořené být mohou. [1, 8]



Obrázek 2.3. Hierarchie Microsoft Azure

2.2 Azure Functions

Azure Functions je *serverless computing* služba typu *Function as a Service*, která je založená na konceptu na sobě nezávislých funkcí. Hlavní myšlenka rozdělení aplikační logiky do nezávislých funkcí umožňuje jejich oddělení na logické a infrastrukturní úrovni. Díky tomuto rozdělení lze Azure Functions velmi dobře horizontálně škálovat. Každá funkce musí mít definovaný trigger. Ten spouští funkci na základě události – například časované události, HTTP požadavku nebo změny dokumentu v Azure Cosmos DB. Díky tomu, že jsou funkce závislé na události, která nese potřebná vstupní data, mohou být funkce stateless. [9]

Azure Functions nabízí tři cenové kategorie – Consumption plan, Premium plan a Dedicated plan. Consumption plan je typu pay-as-you-go, který je placen za počet spuštění funkcí a za počet využité paměti počítáno v GB/s. Consumption plan také nabízí *free tier*, ve kterém má uživatel zdarma 1 000 000 spuštění funkcí a 400 000GB/s paměti za měsíc. Premium plan je také typu pay-as-you-go, v tomto případě uživatel platí za dobu využití vCPU a paměti. Dedicated plan na rozdíl od předchozích kategorií vyžaduje dedikovanou Azure App Service s Azure App Service plan, která slouží jako hosting. Kvůli tomu, že se jedná o dedikovanou infrastrukturu, platí uživatel statickou cenu za dobu běhu hostingu podle cenových kategorií Azure App Service plan. [9]

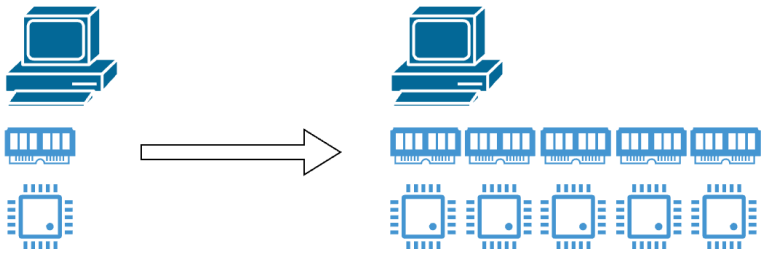
Každá instance funkce má omezenou maximální dobu běhu. Pokud je tato maximální doba přesažena, pošle host požadavek na ukončení běhu funkce. Například při použití C# SDK se běh jednotlivých funkcí ukončuje pomocí notifikace přes `CancellationToken`. Pokud není běh funkce korektně ukončen včas, riskuje tím uživatel ztrátu dat. Maximální doba běhu jedné instance funkce je závislá na cenové kategorii, pod kterou funkce běží. Pro Consumption plan je tato doba 10 minut, pro Premium plan a Dedicated plan tato doba není omezena. Přestože uživatel použije neomezenou dobu maximálního běhu, musí zajistit korektní přerušení běhu na vyžádání. Azure Functions host si může vyžádat přerušení běhu funkce například kvůli updatu operačního systému. Příklad korektně ukončené Azure Function je vyobrazen v následujícím kódu. [9]

```
public class MyFunction
{
    [Function(nameof(MyFunction))]
    public async Task Run([TimerTrigger("0 */5 * * * *")]
        TimerInfo timer, ILogger logger, CancellationToken ct)
    {
        try
        {
            var batch = GetBatch();
            foreach (var item in batch)
            {
                ct.ThrowIfCancellationRequested();
                var result = await CriticalCalculation(item);
                logger.LogInformation("{Result}", result);
            }
        }
        catch (OperationCanceledException)
        {
            logger.LogWarning("Cancelled");
        }
    }

    object[] GetBatch() {...}
    Task<string> CriticalCalculation(object item) {...}
}
```

Pro použití Azure Functions v prostředí cloudu je nutné mít vytvořený Azure Storage account. V Azure Storage si Azure Functions host ukládá metadata pro běh funkcí. Například *singleton lock* zajišťuje běh pouze jedné instance funkce, která je spouštěna pomocí timer trigger, a to i v případě, kdy se rozvrh spuštění funkce překrývá. Azure Storage je popsán v Kapitole 2.3. Azure Functions byly použity v řešení SharePoint protector, které je popsané v Kapitole 6. [9]

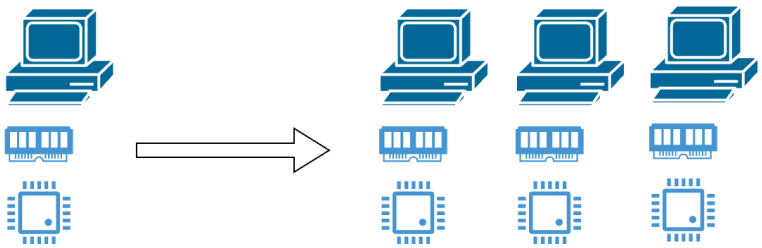
2.2.1 Vertikální a horizontální škálování

Metoda vertikálního škálování zvyšuje výkon přidáním výpočetních zdrojů například CPU nebo paměti. Je vyobrazena na Obrázku 2.4. Například aplikace náročné na výpočetní výkon lze škálovat přidáním CPU s více jádry. Nevýhodou vertikálního škálování je, že je omezeno fyzickými limity hardwaru, protože nelze donekonečna zvyšovat výkon jednoho stroje. [10]



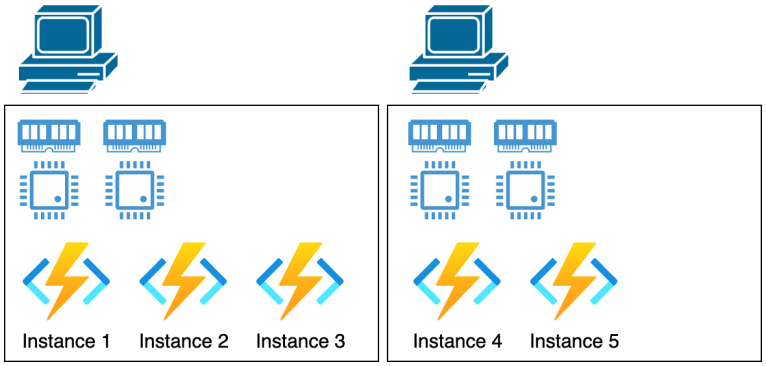
Obrázek 2.4. Vertikální škálování

Horizontální škálování je škálovací metoda, který spoléhá na distribuci zátěže mezi více strojů a instancí aplikace, viz Obrázek 2.5. Aby bylo možně zátěž dobře distribuovat, je nutné řešení navrhnout tak, aby byly jednotlivé funkcionality na sobě nezávislé a sdílení zdrojů minimální. Tento model škálování je velmi vhodný pro *serverless computing* služby typu *Function as a Service*, jako je například Azure Functions. [10]



Obrázek 2.5. Horizontální škálování

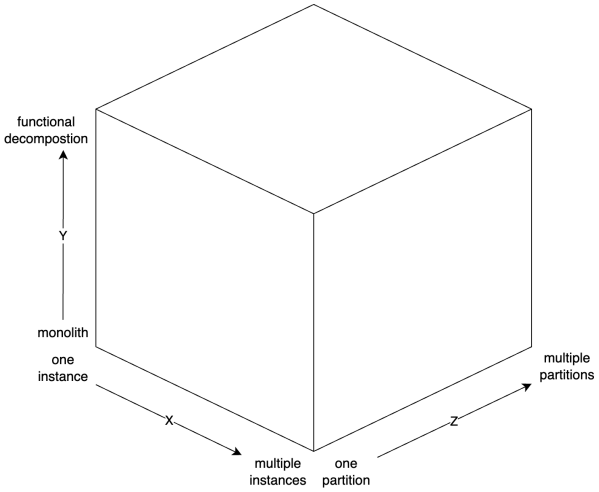
V případě Azure Functions Consumption a Premium plánu je hosting a škálování spravováno Microsoft Azure. Díky tomu může hosting nevyužité zdroje alokovat funkcím jiného zákazníka. Další výhodou spravovaného hostingu je možnost automatického škálování. Například při použití *event driven* architektury, kdy jednotlivé funkce reagují na události, je možné v případě výskytu nárazového vytížení funkce škálovat automaticky téměř s nulovou latencí. Na Obrázku 2.6 je vyobrazeno, jak na jednom hostingu běží několik instanci Azure Functions. Díky omezení na čas běhu jedné instance funkce a počtu výpočetních zdrojů může Microsoft Azure předpovídat vytížení a efektivně rozvrhovat běh funkcí tak, aby služba zůstala vysoce dostupná s nízkou latencí. [9–10]



Obrázek 2.6. Horizontální škálování Azure Functions

2.2.2 Škálovatelnost

Škálovatelnost aplikace je závislá nejen na správné implementaci, ale i na jejím logickém návrhu. Škálovat lze nezávsile na sobě v několika dimenzích, které jsou vyobrazeny na škálovací krychli na Obrázku 2.7



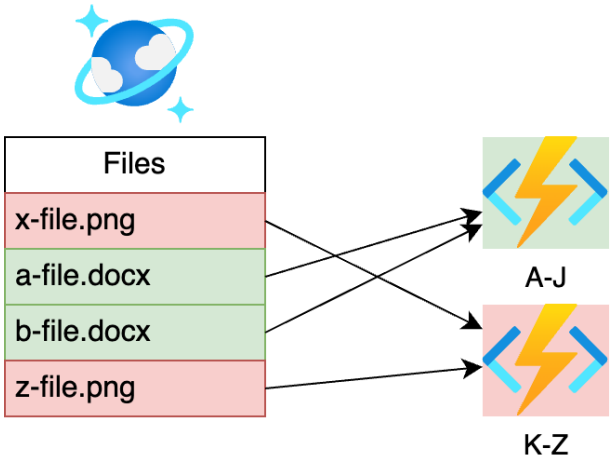
Obrázek 2.7. Škálovací krychle

První možností je škálovat použitím několika instancí stejné aplikace. Tato metoda škálování je nejvhodnější pro stateless aplikace, kdy instance mezi sebou sdílí minimální zdroje. Při sdílení zdrojů může docházet ke konfliktům souběžného přístupu. Řešení konfliktů může být časově náročné a může způsobovat bottleneck výkonu aplikace.

Druhou možností je použít microservice architekturu a rozdělit aplikaci podle funkcionality na sobě nezávislé služby. Díky tomu lze škálovat jednotlivé služby nezávisle na sobě. Nevýhodou této architektury je přidaná komplexita při orchestraci vývoje, nasazení a správy jednotlivých služeb.

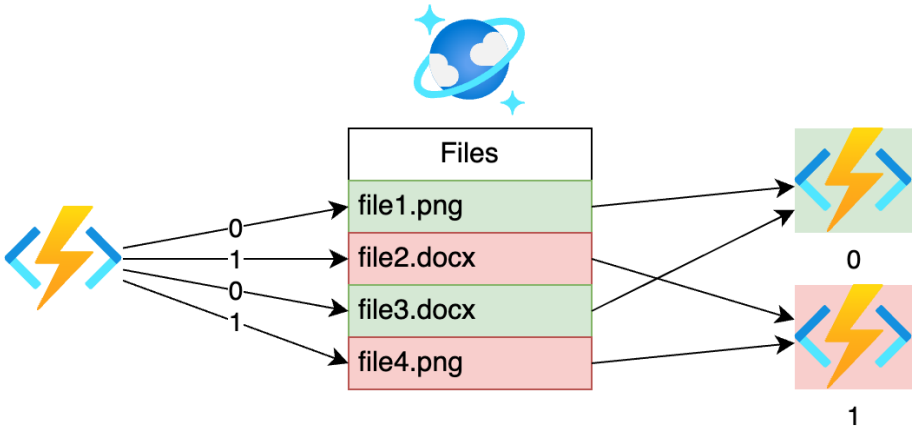
Třetí možností je škálovat rozdělením dat na části a každou část přiřadit jedné instanci aplikace na zpracování. Například při použití fronty lze vkládat data do fronty s více partitions a použít více konzumentů pro zpracování záznamů z fronty. Pro rozdělení do partitions lze použít například strategii *round robin* nebo strategii rozdělení podle jména souboru.

Na Obrázku 2.8 je vyobrazeno rozdělení do partitions podle začátku jména souboru. Všechny soubory začínající písmenem A-J jsou odebírány prvním konzumentem fronty a všechny soubory začínající písmenem K-Z druhým. Toto rozdělení nemusí být zcela férové, protože nelze zaručit rovnoměrné rozdělení souborů podle prvního písmena jejich názvu.



Obrázek 2.8. Rozdělení do partitions podle jména

Obrázek 2.9 zobrazuje rozdělení do partitions podle strategie *round robin*. Vkladatel do fronty si udržuje informaci o tom, do které partition vložil naposledy, a inkrementálně zvyšuje číslo partition s každým vloženým záznamem. Díky této strategii lze zaručit férové rozdělení záznamů do partitions. Rozdělení je férové pouze v počtu záznamů. Pro ještě větší férovost by bylo možné implementovat strategii, která by například respektovala velikost souborů a vkládala by záznamy do nejméně vytížené partition.



Obrázek 2.9. Rozdělení do partitions strategií *round robin*

Metody škálování lze navzájem kombinovat a vytvořit tak velmi dobře škálovatelné řešení. Příkladem může být řešení implementující microservice architekturu, které rozděluje data podle regionu uživatele, v němž má každá služba několik instancí.

2.3 Azure Storage

Azure Storage je služba pro ukládání velkého množství strukturovaných nebo nestrukturovaných dat. Pod Azure Storage spadají služby Azure Blob Storage, Azure Queue Storage, Azure Table Storage a Azure Files. Každá z těchto služeb je určena pro jiné typy dat a scénáře použití. Do Azure Blob Storage lze ukládat nestrukturovaná data velkého objemu – blobů. Maximální velikost jednoho uloženého blobu je 190.73TiB. Azure Blob Storage umožňuje ukládat bloby v různých úrovních přístupu – Hot, Cool, Cold a Archive. Úrovně se liší podle ceny a doby odezvy při čtení a zápisu dat. Například úroveň Hot je určena pro často čtená a zapisovaná data. Zatímco úroveň Archive je určena pro data, která jsou čtena a zapisována zřídka. Pro snížení nákladů na uložená data lze rezervovat kapacitu na jeden nebo tři roky dopředu. Azure Blob Storage podporuje tři protokoly pro práci s daty – NFSv3, HTTP REST a Data Lake Storage Gen2. Azure Storage byl použit pro ukládání metadat Azure Functions v řešení SharePoint protector popsaném v Kapitole 6. [11]

2.4 Azure App Service

Azure App Service je spravovaná služba pro hostování webových aplikací a aplikačních rozhraní – API. Služba podporuje několik programovacích jazyků, jako je například .NET C#, Java, Node.js a Python. Webové služby lze provozovat v App Service na spravovaných virtuálních strojích Windows a Linux. Azure App Service nabízí tři možnosti škálování – Manual, Automatic a Rule Based. U škálování typu Manual Azure App Service vytvoří konstantní počet instancí služby. Automatic typ škálování automaticky

přidává nebo odebírá instance služby podle aktuálního vytížení. Škálování typu Rule Based umožňuje definovat pravidla pro škálování služby podle potřeb uživatele. Lze například nastavit počet instancí pro daný časový interval. Počet instancí lze nastavit i podle určité metriky, jako je například vytížení CPU. Azure App Service pro svůj běh vyžaduje Azure App Service plan, který určuje výpočetní a paměťové zdroje přidělené službě. App Service plan se dělí do několika cenových kategorií – Free, Shared, Basic, Standard, Premium a Isolated. [12]

Výhodou použití App Service je snadné nasazení a správa webových služeb. Snadné nasazení zajišťuje integrace s CI/CD nástroji jako je například GitHub Actions, Azure DevOps Pipelines nebo Bitbucket Pipelines. Další výhodou App Service je například funkce snadné autentizace, která je zajištěna pomocí autentizačního middleware, který je součástí služby. Autentizační middleware podporuje v základu běžné identity providers, jako je například Microsoft Identity platform, Google, Apple, Facebook, Twitter, GitHub nebo jiný libovolný OpenID Connect provider. Azure App Service byl použit pro hostování aplikačních rozhraní řešení SharePoint protector, které je popsané v Kapitole 6. [12]

2.5 Azure Key Vault

Ukládat plaintext secrets[7] v kódu či v konfiguračním souboru aplikace je velkou bezpečnostní hrozbou. Uniklý secret lze snadno zneužít pro přístup k citlivým informacím. Velkým problémem jsou například konfigurační soubory, které vývojáři sdílí mezi sebou nebo ukládají do verzovacího systému. Čím větší je distribuce konfiguračních souborů, tím větší je i riziko úniku secrets. Proto je vhodné přístup k secrets centralizovat a secrets šifrovat.

Azure Key Vault je služba pro ukládání secrets. Služba centralizuje správu a přístup k secrets, šifruje je v úložišti i při přenosu. Díky tomu snižuje riziko jejich úniku a případného zneužití. Přední výhodou Azure Key Vault je integrace s existujícími službami Microsoft Azure. Lze tak například pomocí RBAC a Managed Identity nastavit přístup k secrets webové aplikaci běžící pod Azure App Service, aniž by bylo nutné ukládat přístupový klíč do App settings nebo do konfiguračního souboru aplikace. RBAC a Managed Identity jsou popsány v Kapitole 2.10. K secrets uložených v Azure Key Vault lze v Azure App Service App settings odkazovat pomocí Azure Key Vault reference. Dalšími funkcionalitami Azure Key Vault jsou rotace šifrovacích klíčů, automatická obnova certifikátů a upozornění na expiraci secrets. Azure Key Vault byl použit pro ukládání secrets řešení Migrator a SharePoint protector popsaných v Kapitole 5 a v Kapitole 6. [13]

2.6 Azure Cosmos DB

Azure Cosmos DB je globálně distribuovaná databázová služba, která garantuje nízkou latenci a vysokou dostupnost. Cosmos DB kombinuje relační, grafové a NoSQL databáze podle zvoleného API. Služba poskytuje výběr z šesti API – NoSQL, PostgreSQL, MongoDB, Apache Cassandra, Apache Gremlin a Table. *Azure Cosmos DB for PostgreSQL* kombinuje PostgreSQL a Citus rozšíření, které udělá z PostgreSQL distribuovanou databázi a umožní horizontální škálování. Pomocí *Azure Cosmos DB for Appache Gremlin*

[7] například heslo, šifrovací klíč, certifikát nebo *connection string*

a za použití Gremlin dotazovacího jazyku lze vytvářet grafy a efektivně se na ně dotazovat. [14]

Azure Cosmos DB for NoSQL je výchozí typ API, který ukládá data ve formátu JSON dokumentů, nad kterými se lze dotazovat pomocí jazyku SQL. Při použití výchozího API je služba rozdělena na dvě logické jednotky – databáze a kontejnery. Databáze se skládá z jednoho nebo více kontejnerů. Kontejner obsahuje jednotlivé dokumenty a definuje cestu k položce *partition key*. Každý dokument musí definovat položku *id*. Dvojice *id* a *partition key* jednoznačně identifikuje dokument v rámci kontejneru. Správný výběr *partition key* je klíčový pro výkon databáze. *Partition key* by měl být zvolen tak, aby zátěž byla rovnoměrně distribuována mezi logické partitions. Logická partition je kolekce dokumentů se stejným *partition key*, která definuje transakční scope a může obsahovat až 20GB dat. Logické partitions jsou automaticky distribuovány do fyzických partitions. Každá fyzická partition má omezenou propustnost 10 000RU/s a maximální velikost 50GB. [14]

Azure Cosmos DB používá RU jako měřítko vytížení databáze, na základě kterého určuje cenu za provoz. 1RU odpovídá *point read*[8] operaci o velikosti 1KB dat. Uživatel má na výběr ze dvou cenových modelů – *provisioned throughput* a serverless. *Provisioned throughput* model umožňuje uživateli zarezervovat počet RU, které chce mít k dispozici, a funguje ve dvou režimech – manual a autoscale. V manual režimu uživatel ručně nastavuje počet RU, které chce mít k dispozici. V autoscale režimu Azure Cosmos DB automaticky škáluje počet RU podle aktuálního vytížení databáze v rozmezí 10% až 100% z rezervovaného množství RU. Serverless model nevynucuje rezervaci RU a uživatel platí pouze za počet RU, které byly spotřebovány. Je vhodný zejména pro scénáře, kdy je těžké odhadnout vytížení databáze, nebo když dochází k nárazovému vytížení databáze. Služba Azure Cosmos DB byla použita v řešení SharePoint protector popsaném v Kapitole 6 za použití implementace z repozitáře adamijak/cosmos popsané v Kapitole 4. [14]

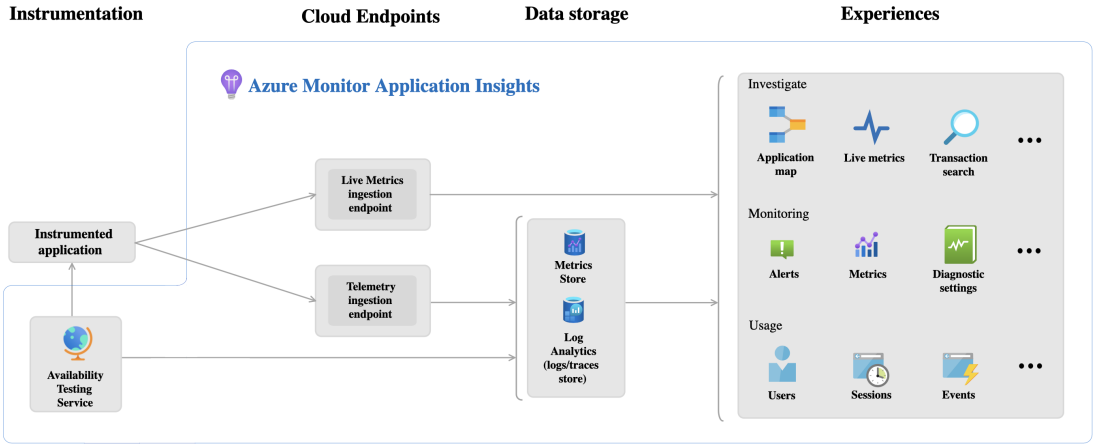
2.7 Azure App Configuration

Azure App Configuration je služba pro centralizovanou správu konfigurace aplikace. Konfiguraci lze importovat nebo exportovat různými zdroji, jako jsou například JSON a YAML soubory, App Service settings nebo další instance App Configuration. Při použití Azure App Configuration SDK a třídy `AzureAppConfigurationProvider` lze celou konfiguraci aplikace načíst pouhým definováním URL App Configuration instance. Konfiguraci tak lze sdílet mezi více uživateli a aplikacemi v lokálním nebo cloudovém prostředí. To je umožněno zejména kvůli RBAC, které umožňuje definovat přístup ke konfiguraci pro aplikace nebo jednotlivé uživatele. Další výhodou je, že App Configuration SDK podporuje dynamickou změnu konfigurace za běhu aplikace. Ta se buď periodicky dotazuje na změny konfigurace, nebo je o změně informovaná zvenčí. *Push based* mechanismus lze implementovat použitím Azure Event Grid, který aplikaci notifikuje o změně konfigurace. *Pull based* metoda je nativně podporována App Configuration SDK, které definuje interval pro dotazování na změnu konfigurace. Azure App Configuration byla použita pro konfiguraci řešení SharePoint protector popsaném v Kapitole 6. [15]

[8] načtení jednoho dokumentu podle jeho *id* a *partition key*

2.8 Azure Application Insights

Monitorování korektního chodu aplikace je důležité pro zajištění dostupnosti služby. Klíčové metriky jsou využití CPU a paměti, latence, počet dotazů, počet chyb a výjimek. Azure Application Insights je funkcionalita služby Azure Monitor, která centralizuje monitorování výkonu a chování aplikace. Její logický model je vyobrazený na Obrázku 2.10. [16]



Obrázek 2.10. Logický model Azure Application Insights [17]

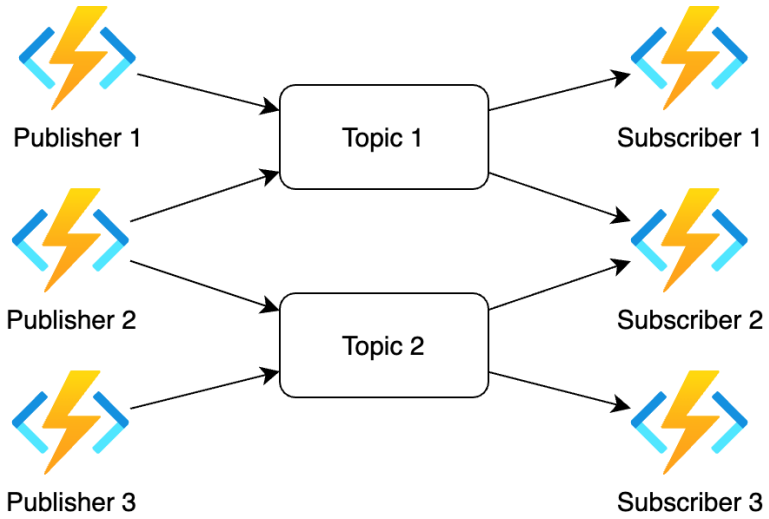
Pomocí služby Application Insights lze například monitorovat výjimky v reálném čase či se dotazovat nad logy aplikace pomocí KQL. U každé výjimky lze pomocí *stack trace* dohledat místo, kde došlo k chybě. Funkcionalita *availability alerts* umožňuje definovat pravidla, která upozorní správce na nedostupnost služby. Každému pravidlu lze definovat například URL služby, HTTP metodu, očekávaný *HTTP status code* a časový interval spouštění testu. Pokud dojde k selhání testu, je správce upozorněn například pomocí e-mailu. Alerting lze nastavit i na vlastní metriky, lze tak například dostávat upozornění na výskyt řetězce v logu. Díky funkci *autoinstrumentation* není nutné pro monitoring měnit kód aplikace. Pro zapnutí monitorace stačí vytvořit instanci Application Insights a nastavit *connection string* v proměnné prostředí aplikace. Azure Application Insights je použito pro monitorování řešení SharePoint protector popsaném v Kapitole 6. [16]

2.9 Azure Event Grid

Azure Event Grid je služba pro flexibilní distribuci zpráv, která podporuje MQTT a HTTP protokol. Služba poskytuje *push based* a *pull based* doručení zpráv. Výhodou Event Grid je, že se nativně integruje se službami Microsoft Azure, což umožňuje konzumentům reagovat na systémové události platformy. Systémová událost platformy je například vytvoření nového blobu v Azure Blob Storage nebo změna konfigurace v Azure App Configuration. [18]

Služba implementuje návrhový vzor publisher-subscriber. Publisher publikuje zprávy do sběrnice, ze které jsou distribuovány na jednoho nebo více subscribers. Subscriber definuje topic, jehož zprávy chce odebírat. Příklad publisher-subscriber modelu s filtrováním zpráv podle topicu je vyobrazený na Obrázku 2.11. Příklad znázorňuje tři publishers, tři subscribers a dva topics. *Subscriber 1* je přihlášený k odběru zpráv z *Topic 1*, které může publikovat *Publisher 1* nebo *Publisher 2*. Azure Event Grid byl použit

pro komunikaci s aplikačním rozhraním v řešení SharePoint protector popsaném v Kapitole 6. [18]



Obrázek 2.11. Návrhový vzor publisher-subscriber

2.10 Microsoft Entra ID

Microsoft Entra ID je služba pro správu identit a přístupu ke zdrojům. Spravovanou identitou může být uživatel, služba nebo zařízení. Identity lze v Entra ID seskupovat do skupin a administrativních jednotek. Služby vystupují v Entra ID jako objekt *serivce principal*, který vzniká při vytvoření App registration nebo Managed Identity. [19]

App registration definuje identitu aplikace, scopes, ke kterým má aplikace či uživatel přístup, a typ autentizace. Ověřit se vůči App registration lze v uživatelském nebo aplikačním kontextu. Autentizace v kontextu aplikace lze provést pouze pomocí *application secret* nebo certifikátu. Ověření v kontextu uživatele je proveditelné pomocí standardních autentizačních metod. Příkladem může být uživatelské jméno a heslo, *device code* nebo *passwordless* ověření přes mobilní aplikaci. Pokud se uživatel v aplikaci ověří vůči Entra ID, dává jí tak možnost vystupovat jeho jménem. [19]

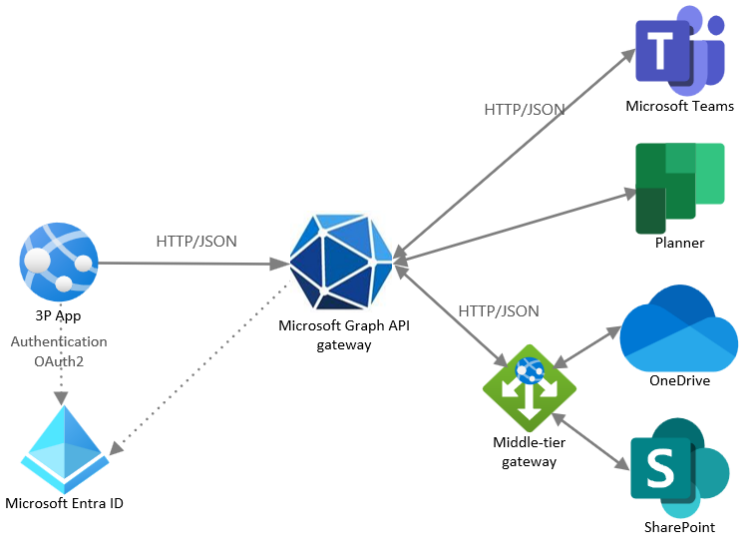
Managed Identity je identita spravovaná Microsoft Azure, která je přiřazena službě a umožňuje jí autentizovat se vůči Entra ID. Díky Managed Identity lze snadno řešit problém přístupu mezi jednotlivými službami v Azure. Například lze zapnout Managed Identity pro Azure App Service a přidělit jí přístup k Azure Key Vault. Díky tomu lze v Azure App Service používat reference pro přístup k secrets uložených v Azure Key Vault. [19]

Pro řízení přístupu používá Entra ID RBAC. Každá služba implementující RBAC má definované role, které určují, jaké operace může identita či skupina identit uskutečnit. Lze tak například přiřadit uživateli roli *Key Vault Secrets User*, která umožní uživateli číst secrets z Azure Key Vault. [19]

Microsoft Entra ID, App registration, Managed Identity a RBAC byly použity k řízení přístupu v řešeních Migrator a SharePoint protector, která jsou popsána v Kapitole 5 a v Kapitole 6.

Kapitola 3
Microsoft Graph

Microsoft 365 se skládá z mnoha služeb, které poskytují nástroje pro pracovní prostředí, komunikaci a produktivitu. Příkladem těchto služeb může být Entra ID, Outlook, OneDrive, SharePoint nebo Teams. Při vývoji řešení postavených nad Microsoft 365 je často potřeba interagovat s více službami najednou. Microsoft Graph sjednocuje přístup k datům služeb Microsoft 365 a operacím nad nimi. Příklad použití sjednoceného rozhraní je znázorněn na Obrázku 3.1.



Obrázek 3.1. Použití Microsoft Graph API pro přístup ke službám Microsoft 365 [20]

Přes Microsoft Graph lze uskutečnit rozmanité množství akcí na službách Microsoft 365. Příkladem je následující seznam:

- Získat jméno a UPN uživatele z Entra ID
- Poslat email v kontextu uživatele z Outlook
- Nahrát soubor na OneDrive uživatele
- Získat všechny upravené soubory od určité doby z SharePoint site
- Založit nový tým a přidat do něj uživatele v Teams

S Microsoft Graph lze interagovat několika způsoby. Mezi tyto způsoby patří například Microsoft Graph API, Microsoft Graph connectors a Microsoft Graph Data Connect. Microsoft Graph REST API je dostupný ve dvou verzích v1.0 a beta poskytovaných na adresách `https://graph.microsoft.com/v1.0` a `https://graph.microsoft.com/beta`. Pro použití Microsoft Graph API je nutné se autentizovat a autorizovat. Tento proces je popsán v Kapitole 3.1 a v Kapitole 3.2. [21]

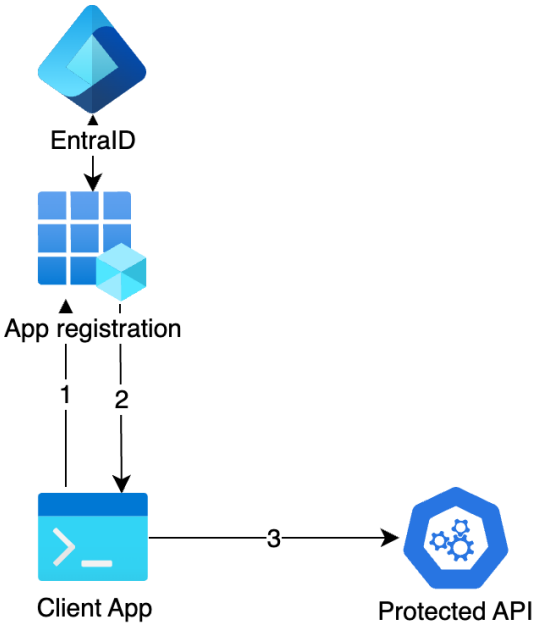
3.1 Autentizace

Autentizace je proces dokazování, že jste skutečně tím, za koho se vydáváte. Toho lze dosáhnout ověřením identity, osoby nebo zařízení. Microsoft identity platform používá pro zpracování ověřování protokol OpenID Connect. [22]

Autentizační flow je vyobrazený na Obrázku 3.2, a je rozdělený do následujících kroků:

- 1. Ověření vůči Entra ID zvolenou autentizační metodou
- 2. Vydání přístupového tokenu ve formátu JWT, který je popsán v Kapitole 3.3
- 3. Použití přístupového tokenu pro volání chráněného API

Aby bylo možné používat Microsoft Graph API, je nutné nejprve vytvořit *App Registration* v Entra ID a autentizovat se za použití *Microsoft identity platform*. Po ověření je vydán přístupový token, se kterým lze volat Microsoft Graph API.



Obrázek 3.2. Autentizační flow

3.2 Autorizace

Autorizace je akt udělení oprávnění autentizované straně něco udělat. Určuje, k jakým datům je povolen přístup a jak lze s těmito daty pracovat. Microsoft identity platform používá pro zpracování autorizace protokol OAuth 2.0. [22]

Microsoft Graph API rozlišuje dva druhy oprávnění – aplikační a delegované. Aplikační oprávnění lze získat po ověření vůči Entra ID v kontextu aplikace a jsou omezená v rozsahu tenantu[1]. Například oprávnění *Files.Read.All* umožňuje aplikaci číst všechny soubory v tenantu. Delegovaná oprávnění slouží pro přístup k Microsoft Graph API v kontextu uživatele. Oprávnění jsou aplikaci vydána na vyžádání uživatele, po jeho ověření. Delegovaná oprávnění jsou pak omezená na scope uživatele. Například oprávnění *Files.Read.All* umožňuje aplikaci číst všechny soubory, ke kterým má přístup uživatel, za kterého se aplikace vydává. [21]

[1] konkrétní instance Microsoft Entra ID pro danou organizaci

V současné době Microsoft Graph API nepodporuje větší granularitu oprávnění, což vede k vytváření servisních účtů, aby se zamezilo vytváření aplikací s příliš velkým rozsahem oprávnění. Servisní účet se tváří jako uživatel a platí pro něj stejná pravidla – například v přidělování licencí. Vytváření a správa servisních účtů proto není považována za *best practice*.

3.3 JWT

Přístupové tokeny JWT umožňují klientům bezpečně volat chráněná webová rozhraní API. Webová API používají přístupové tokeny při autentizaci a autorizaci. *Identity provider* po autentizaci vydá token ve formátu JWT. Ten se skládá ze tří částí oddělených tečkou – hlavičky, obsahu a podpisu. [22]

Hlavička JWT je base64 URL kódovaný JSON a specifikuje, jakým algoritmem je token podepsán. Payload JWT standardně obsahuje množinu *Registered Claim Names* a vlastní claims definované službou vydávající token. Množina *Registered Claim Names* obsahuje hodnoty Issuer, Subject, Audience, Expiration Time, Not Before, Issued At a JWT ID. Payload je stejně jako hlavička base64 URL kódovaný JSON. Řetězec obsahu pro podpis vznikne spojením hlavičky a payloadu znakem tečky. Podpis vznikne zašifrováním řetězce obsahu pomocí šifrovacího algoritmu definovaného v hlavičce. Výsledný token vznikne spojením řetězce obsahu a podpisu znakem tečky. [23]

3.4 Microsoft Graph SDK

Microsoft Graph SDK je dostupný pro několik programovacích jazyků, mezi které patří například C#, Java, JavaScript, Python a Go. Microsoft generuje klientské SDK pomocí OpenAPI dokumentu a nástroje Kiota dostupného online[2] v GitHub repozitáři. OpenAPI specifikace je popsaná v Kapitole 3.4.1. OpenAPI dokument a všechny dostupné SDK jsou publikovány online v repozitářích Microsoft Graph GitHub organizace[3]. [21]

3.4.1 OpenAPI

OpenAPI specifikace poskytuje formální standard pro popis HTTP API. Pomocí této specifikace lze například generovat kód pro klientský SDK nebo vytvářet testy [24]. OpenAPI dokument, který odpovídá OpenAPI specifikaci, je reprezentován v JSON nebo YAML formátu. Každý OpenAPI dokument musí obsahovat položku `openapi` a položku `info`. Položka `openapi` specifikuje verzi OpenAPI specifikace, kterou OpenAPI dokument používá. Položka `info` je objekt, který musí obsahovat alespoň dvě položky `title` a `version`. Položka `title` je název API a `version` je verze API. Kompletní OpenAPI specifikace je dostupná online[4]. Příklad OpenAPI dokumentu v YAML formátu je vyobrazen níže. [25]

[2] `https://github.com/microsoft/kiota`
[3] `https://github.com/microsoftgraph`
[4] `https://spec.openapis.org/oas/latest.html`

```
openapi: 3.1.0
info:
  title: Store
  version: 0.1.0
paths:
  /items/{itemId}:
    get:
      summary: Get specific item
      parameters:
        - name: itemId
          in: path
          required: true
      responses:
        200:
          content:
            application/json:
              schema:
              $ref: "#/components/schemas/Item"
components:
  schemas:
    Item:
      required:
        - id
        - name
      properties:
        id:
          type: integer
          format: int64
        name:
          type: string
```

Položka `paths` definuje cesty a metody nad nimi. V příkladu je vyobrazena cesta `/items/itemId`, nad kterou je definována jediná metoda `GET`. Operace má definovaný jediný povinný parametr `itemId` a jedinou odpověď, která vrací *HTTP status code 200 OK* a JSON reprezentaci objektu Item. Položka `components` definuje znovupoužitelné komponenty – například parametry, schéma objektů nebo odpovědí. V příkladu je definované jediné schema objektu Item, který má definované dvě povinné položky `id` a `name`.

3.5 azure-http

Kolekce HTTP dotazů *azure-http* vznikla za účelem testování Microsoft Azure, Microsoft Graph API a Microsoft 365 služeb. Kolekce je dostupná online[5] v GitHub repozitáři pod licencí MIT. Důvodem vzniku kolekce byly problémy, které se vyskytovaly při vývoji řešení postavených nad Microsoft Graph API.

Hlavním problémem byla neúplnost Graph API SDK, který se generuje na základě OpenAPI dokumentu. Chybějící data v OpenAPI dokumentu způsobují špatně vygenerovaný SDK. Takovému SDK pak chybějí funkce a položky, které způsobují nepoužitelnost SDK. Důsledkem toho je uživatel SDK nucen psát vlastní implementaci, která je

[5] `https://github.com/adamijak/azure-http`

založená na prostých HTTP dotazech. Příčinou chybějících dat v OpenAPI dokumentu nejsou pouze chyby při vytváření OpenAPI dokumentu, ale i úmyslné vynechání některých cest, aby nedocházelo k explozivnímu nárůstu velikosti SDK. Příkladem problémů, které způsobily chybějící data v OpenAPI dokumentu, mohou být následující GitHub issues.

- Drive Root does not contain Delta property #2231[6]
- [Client bug]: AdministrativeUnits[].Members.PostAsync() missing #1828[7]

Dalším problémem byly rozdíly mezi dokumentací a skutečností – ne vždy se Graph API chovalo tak, jak bylo popsáno v dokumentaci. Proto nešlo spoléhat pouze na dokumentaci a bylo nutné Graph API otestovat ručně, zda poskytuje nebo zda mu chybí popsané funkcionality. Kolekce *azure-http* se skládá z http souborů rozdělených do složek podle typu služby a operace. Příkladem dokumentu v http formátu je následující kód.

```
@drive_id=drive1
@id=item1

GET https://graph.microsoft.com/v1.0/drives/{{drive_id}}/items/{{id}}
Authorization: Bearer {{token}}
```

Příklad zobrazuje dotaz na Microsoft Graph API, který vyčte informace o souboru uloženém v SharePoint úložišti. Dále definuje dvě proměnné `drive_id` a `id`, které při poslání dotazu vyplní konkrétními hodnotami. K autentizaci použije přístupový token, který je vkládán do hlavičky `Authorization` pomocí proměnné `token`.

Výhodou http formátu je jednoduchost a přehlednost. Integrují ho běžná vývojová prostředí, jako jsou například JetBrain's IDEs, Visual Studio a Visual Studio Code s rozšířením REST Client. Díky tomu, že se formát snaží co nejvíce imitovat HTTP protokol, je jednoznačné, co se na službu odesílá. Nevýhodou http formátu je, že pro něj v současné době neexistuje standard, a každý z nástrojů používá trochu jiný formát. Kolekce se snaží dodržovat takový formát, aby maximalizovala přenositelnost mezi těmito nástroji.

Pro testování Graph API nebyl použit populární nástroj Postman[8], protože například neumožňuje snadno vkládat proměnné, které jsou omezené na daný dotaz. Použití Postman vyžaduje instalaci aplikace nebo vytvoření účtu a použití webové aplikace. Dále se Postman snaží prosazovat funkcionality, které jsou v neplacené licenci omezené. Díky *azure-http* lze spoléhat na Git a vývojové prostředí, které vývojář používá na denní bázi.

3.6 azure-auth

Aplikace *azure-auth* vznikla za účelem ověření vůči Microsoft Entra ID v kontextu aplikace za použití certifikátu. Je dostupná online[9] v GitHub repozitáři pod licencí MIT. Aplikace získá pomocí vybrané autentizační metody přístupový token, který vypíše na standardní výstup.

[6] https://github.com/microsoftgraph/msgraph-sdk-dotnet/issues/2231
[7] https://github.com/microsoftgraph/msgraph-sdk-dotnet/issues/1828
[8] https://www.postman.com
[9] https://github.com/adamijak/azure-auth

Pokud se aplikace ověří vůči Entra ID a získá potřebná aplikační práva, dostane přístup k daným scopes[10] v celé organizaci. Je tedy nutné dodržovat bezpečnostní opatření, aby nedošlo k zneužití. Ověření pomocí *application secret* nelze považovat za bezpečné, protože navádí k ukládání secrets v kódu nebo v konfiguračních souborech. Zároveň má *application secret* omezenou platnost maximálně 2 roky. Ověření pomocí certifikátu je považované za bezpečnější, protože certifikát je možné uložit do uživatelského úložiště certifikátů chráněného heslem. Certifikáty mohou mít delší platnost než *application secret* v řádu desítek let v závislosti na vydavateli certifikátu.

Ověření vůči Entra ID lze dosáhnout pomocí již existující aplikace azure-cli[11]. Tato aplikace však nepodporuje ověření pomocí certifikátu uloženého v uživatelském úložišti certifikátů[12]. Příklad použití *azure-auth* a získání scopes pro Graph API je vyobrazen níže.

```
azure-auth cert --raw -t TENANT_ID -c CLIENT_ID
--cert CERT_THUMBPRINT --scopes "https://graph.microsoft.com/.default"
```

[10] množina oprávnění udělená držiteli tokenu
[11] `https://github.com/Azure/azure-cli`
[12] `https://github.com/Azure/azure-cli/issues/23448`

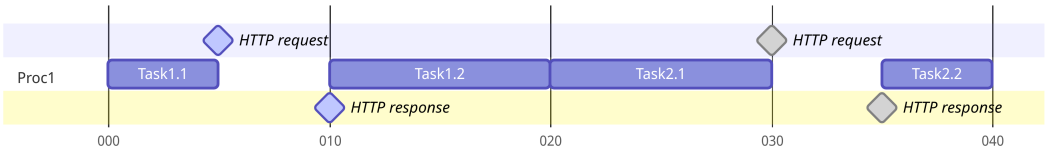
Kapitola 4
adamijak/cosmos

Při práci s Azure Cosmos DB docházelo k opakované implementaci stejných funkcionalit mezi projekty. GitHub repozitář adamijak/cosmos vznikl za účelem sjednocení implementace pro práci s Azure Cosmos DB. Repozitář je dostupný online[1] v GitHub repozitáři pod licencí MIT a obsahuje tři projekty – Cosmos, CosmosTest a CosmosBenchmark.

Projekt Cosmos se skládá ze tří namespaces – `Adamijak.Cosmos.Queue` popsaný v Kapitole 4.4, `Adamijak.Cosmos.Extensions` popsaný v Kapitole 4.3 a `Adamijak.Cosmos.Configuration`. Projekty CosmosTest a CosmosBenchmark jsou popsané v Kapitole 4.5 a 4.2. Pro maximalizaci výkonu a efektivity využití vláken využívá celé řešení asynchronního programování, které je popsané v Kapitole 4.1.

4.1 Paralelní a asynchronní programování

Paralelní a asynchronní programování slouží k implementaci programů, které vykonávají více úloh souběžně. V nejjednodušším případě program nevyužívá žádných technik souběžného programování a všechny úlohy vykonává sériově jednu po druhé. Obrázek 4.1 zobrazuje dvě nezávislé úlohy *Task 1* a *Task 2*, které používají blokující operaci HTTP dotazu. Blokující volání HTTP endpointu může trvat v řádu desítek milisekund až sekund. Během této doby je vlákno, které čeká na odpověď, blokované a nemůže vykonávat jiné úlohy.



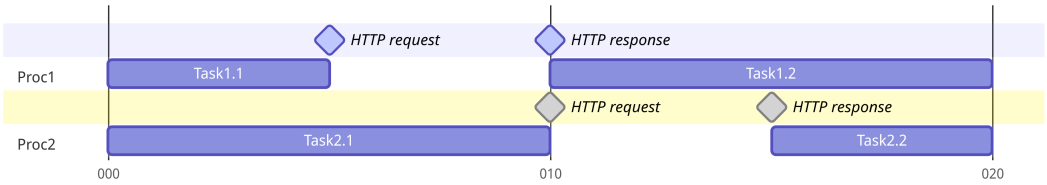
Obrázek 4.1. Příklad sériového výpočtu vykonávaného na jednom vlákně

První možností jak zvýšit výkon výpočtu je použít paralelní programování. V příkladu znázorněném na Obrázku 4.2 jsou výpočtu přiřazena dvě vlákna a každé z nich vykonává jednu úlohu. Při zanedbání orchestrace se výpočet zrychlí maximálně úměrně k počtu vláken. V příkladu znázorněném na Obrázku 4.2 se celková doba zrychlí maximálně dvakrát oproti sériovému výpočtu znázorněném na Obrázku 4.1.

Ačkoliv se absolutní doba výpočtu zkrátila, problémem stále zůstává škálovatelnost aplikace kvůli blokujícím operacím. Navzdory tomu, že v dnešní době lze pořídit procesory se stovkami vláken, jsou stále omezeny jejich maximálním počtem [26]. Vytížíme-li všechna vlákna procesoru blokujícími operacemi, musíme počkat na dokončení jedné z operací, a to i přes to, že vlákna během čekání nevykonávají žádnou práci. Řešením

[1] `https://github.com/adamijak/cosmos`

tohoto problému by bylo využít dobu, kdy vlákno čeká na odpověď z blokující operace, k vykonávání jiných úloh. K tomuto účelu slouží asynchronní programování.

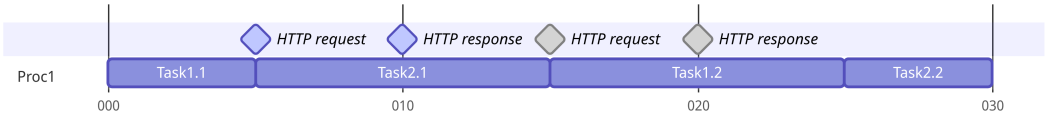


Obrázek 4.2. Příklad sériového výpočtu vykonávaného paralelně na dvou vláknech

Každý program lze rozdělit na úlohy, které lze vykonávat nezávisle. Tímto rozdělením vznikne orientovaný acyklický graf závislostí úloh. Pokud by vzniklý graf nebyl DAG, znamenalo by to, že některé úlohy jsou závislé, což by způsobilo deadlock. [27]

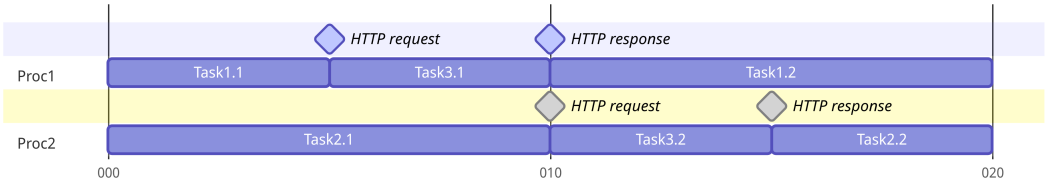
Při použití asynchronního programování je výpočet rozdělen do úloh, které se navzájem střídají ve vykonávání, podle strategie plánovače. V případě, že je právě vykonávaná úloha blokována, může plánovač přepnout na jinou úlohu a využít čas, který by byl ztracen v případě čekání na dokončení blokující operace.

Díky tomu, že není asynchronní výpočet závislý na počtu jader, může probíhat i na jednom vlákně. Dobrým příkladem jsou embedded systémy, u kterých je běžná absence operačního systému a vláken. Obrázek 4.3 znázorňuje asynchronní výpočet na jednom vlákně. Oproti příkladu z Obrázku 4.1 je zde využit čas čekání na odpověď z blokující operace k vykonávání jiné nezávislé úlohy. Konkrétně v tomto případě se výpočet zrychlil o 25%.



Obrázek 4.3. Příklad asynchronního výpočtu vykonávaného na jednom vlákně

K dalšímu zrychlení asynchronního výpočtu můžeme využít více vláken. Obrázek 4.4 zobrazuje asynchronní výpočet na dvou vláknech, kde je čas čekání na dokončení blokující operace využit k vykonávání úlohy *Task 3*. Runtime .NET si udržuje *thread pool*, ze kterého každému vláknu přiřazuje jednotlivé úlohy na vykonání. [28]



Obrázek 4.4. Příklad asynchronního výpočtu vykonávaného na dvou vláknech

Implementace asynchronního programování v C# je popsána v Kapitole 4.1.1.

4.1.1 Asynchronní programování v C#

V C# je asynchronní programování založeno na třídě `Task` a klíčových slovech `async` a `await`. Klíčové slovo `async` se používá pro označení asynchronní metody. Asynchronní metoda by měla vždy vracet `Task` nebo `Task<T>`. Třída `Task` reprezentuje úlohu, která běží na pozadí a která má očekávaný konec. Instance třídy `Task` může být v jednom z následujících stavů:

- Created - `Task` byl vytvořen, ale ještě nebylo naplánováno jeho spuštění
- WaitingForActivation - `Task` čeká na aktivaci a plánování spuštění
- WaitingToRun - `Task` byl naplánován, ale ještě nebyl spuštěn
- Running - `Task` běží, ale ještě nebyl dokončen
- WaitingForChildrenToComplete - `Task` dokončil svůj běh a čeká na dokončení běhu všech svých potomků
- RanToCompletion - `Task` úspěšně dokončil svůj běh
- Canceled - `Task` byl přerušen vyhozením výjimky `OperationCanceledException`
- Faulted - `Task` skončil svůj běh s neošetřenou výjimkou

Klíčové slovo `await` se používá pro označení `Task` nebo `Task<T>`, u kterého se čeká na jeho dokončení. Definují se tak místa, kde vzniká závislost mezi jednotlivými úlohami. Následující příklad vyobrazuje rozdíl mezi sériovým a souběžným voláním HTTP enpoints. [28]

```
// Serial
var res1 = await httpClient.GetAsync("https://fel.cvut.cz"); // 1
var res2 = await httpClient.GetAsync("https://fit.cvut.cz"); // 2
var res3 = await httpClient.GetAsync("https://fjfi.cvut.cz"); // 3
var res4 = await httpClient.GetAsync("https://fs.cvut.cz"); // 4

// Concurrent
Task[] tasks =
[
    httpClient.GetAsync("https://fel.cvut.cz"), // 1
    httpClient.GetAsync("https://fit.cvut.cz"), // 2
    httpClient.GetAsync("https://fjfi.cvut.cz"), // 3
    httpClient.GetAsync("https://fs.cvut.cz"), // 4
];
await Task.WhenAll(tasks);
```

V první části kódu jsou volány HTTP endpointy sériově a před započetím dalšího volání se čeká na dokončení předchozího. Klíčové slovo `await` zde definuje závislosti úloh. Například třetí úloha je závislá na dokončení první a druhé úlohy. První úloha není závislá na dokončení druhé, třetí ani čtvrté úlohy.

V druhé části kódu jsou HTTP endpointy volány asynchronně. Pokud není použito klíčové slovo `await` u jednotlivých úloh, na jejich dokončení se nečeká, nevytváří se mezi nimi závislost a mohou být vykonávány souběžně. Závislost vzniká až při volání metody `Task.WhenAll(tasks)`. To vytvoří novou úlohu, která je závislá na dokončení všech úloh v poli `tasks`.

4.1.2 Řízení souběžně běžících úloh

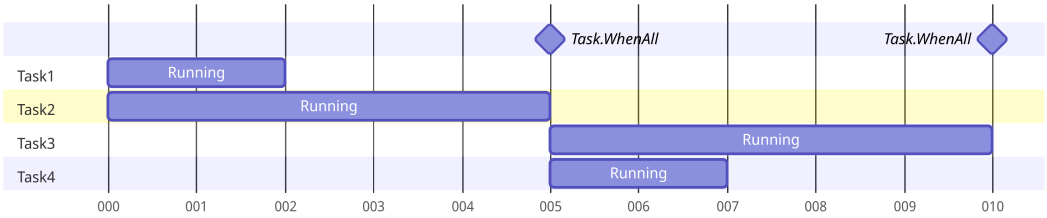
Je běžná praxe, že většina HTTP služeb implementuje throttling. Například Azure Cosmos DB při překročení maximálního počtu RU/s vrátí *HTTP status code 429 Too*

Many Requests. Pokud bychom chtěli například vložit jeden milion dokumentů do Azure Cosmos DB a pro každý dokument bychom vytvořili novou asynchronní úlohu, vytvořili bychom milion asynchronních úloh a velmi rychle bychom překročili maximální počet RU/s. Řešením tohoto problému je omezení počtu souběžně běžících úloh. Toho lze docílit tak, že počet nahrávaných dokumentů rozdělíme na menší části – chunks. [14]

Následující kód rozdělí pole URL do chunks po dvou prvcích a asynchronně odešle HTTP GET požadavek na každý prvek v chunku.

```
string[] urls = ["https://fel.cvut.cz", "https://fit.cvut.cz",
"https://fjfi.cvut.cz", "https://fs.cvut.cz"];

foreach (var chunk in urls.Chunk(2))
{
    var tasks = new List<Task>();
    foreach (var url in chunk)
    {
        tasks.Add(httpClient.GetAsync(url));
    }
    await Task.WhenAll(tasks);
}
```

Tento kód sice omezuje počet souběžně běžících úloh na počet prvků v chunku, ale zároveň čeká na dokončení všech úloh. Tento přístup není vhodný zejména v případě, kdy se může výrazně lišit doba vykonání jednotlivých úloh v chunku. Program musí počkat na dokončení nejdelší z nich, jako je znázorněno na Obrázku 4.5.

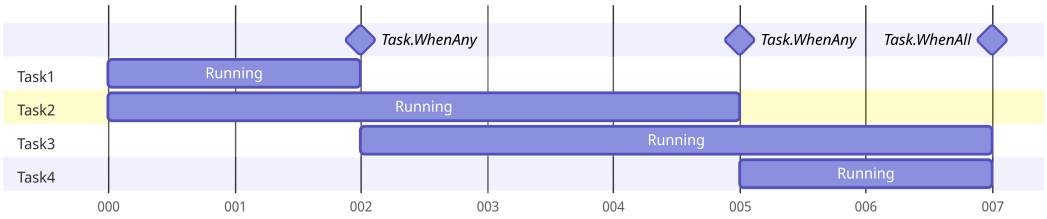


Obrázek 4.5. *Chunk based* zpracování úloh

Tento přístup lze zefektivnit tak, že namísto čekání na dokončení všech úloh se bude čekat na dokončení nejkratší z nich. Pokud jedna úloha skončí, je nahrazena novou, a zachovává se tak počet souběžně běžících úloh. Tento přístup lze implementovat například použitím množiny úloh `HashSet<Task>`.

```
var tasks = new HashSet<Task>();
foreach (var url in urls)
{
    if (2 <= tasks.Count)
    {
        var task = await Task.WhenAny(tasks);
        tasks.Remove(task);
    }
    tasks.Add(httpClient.GetAsync(url));
}
await Task.WhenAll(tasks);
```

Příklad rozvržení úloh při použití množiny úloh je vyobrazen na Obrázku 4.6. Rozvrhování pomocí této strategie nezajišťuje optimální rozvržení úloh. To je primárně způsobeno tím, že doba vykonání jednotlivých úloh není známá. Pro tento přístup je dostačující, že bude vždy efektivnější nebo stejně efektivní jako *chunk based* přístup.



Obrázek 4.6. *Set based* zpracování úloh

Chunk based a *set based* řízení souběžně běžících úloh nereflektují aktuální vytížení volané služby a nezaručují dodržení maximálního povoleného vytížení. Empiricky se ukázalo, že pro většinu služeb jsou tyto přístupy dostačující, zejména když lze reflektovat aktuální vytížení služby na jiné úrovni. Například Azure Cosmos DB SDK implementuje *retry policy*, která respektuje hlavičku `x-ms-retry-after-ms` a zopakuje požadavek po definované době. Tato hlavička je vyplněna, pokud Azure Cosmos DB vrátí odpověď na dotaz s *HTTP status code 429 Too Many Requests*. [14]

Benchmark *chunk based* a *set based* řízení souběžně běžících úloh pro Azure Cosmos DB je popsán v Kapitole 4.2.

4.2 CosmosBenchmark

Projekt CosmosBenchmark byl použit pro testování výkonu *chunk based* a *set based* přístupů pro řízení souběžně běžících úloh při vkládání dokumentů do Azure Cosmos DB. Při výkonnostním testu bylo do kontejneru vloženo 10 000 dokumentů o velikosti přibližně 1KB. Každý dokument se skládal z GUID identifikátoru a konstantního řetězce o délce 1000 znaků. Pro vytváření dokumentů byla použita metoda `CreateItemAsync` třídy `Microsoft.Azure.Cosmos`. Kontejner byl vytvořen s *partition key* rovnému identifikátoru dokumentu a s propustností 1000RU/s. 1000RU/s odpovídá čtení 1000 dokumentů o velikosti 1KB za sekundu. Zápis do Azure Cosmos DB využívá více RU než *point read* operace. Díky tomuto faktu nebude překročena rychlost 1MB/s. Benchmark byl spuštěn na síti s rychlostí připojení 200Mb/s. Tím bylo zajištěno, že rychlost připojení není bottleneck výkonnostního testu. [14]

Výsledky benchmarku v Tabulce 4.1 zobrazují, že neexistuje významný rozdíl mezi použitím *chunk based* a *set based* přístupů pro řízení souběžně běžících úloh při vkládání do Azure Cosmos DB. To je způsobeno tím, že Azure Cosmos DB SDK implementuje seskupovací mechanismus a nevytváří dotaz pro každý dokument zvlášť. Výsledky dále ukazují, že není rozdíl, jestli je počet souběžně běžících úloh omezen na 1000 nebo na 2^{31}. To je způsobeno nastaveným limitem propustnosti 1000RU/s. Výrazně pomalejší je vkládání dokumentů při omezení na 100 souběžně běžících úloh z důvodu neúplného využití propustnosti testovacího kontejneru v Azure Cosmos DB. Benchmark využívá knihovnu BenchmarkDotNet a byl spuštěn na systému popsaném v Tabulce 4.2. [14]

Method	Items	DataSize	Tasks	Mean	Error	StdDev
ChunkInsert	10000	1000	100	23.97s	6.241s	1.621s
SetInsert	10000	1000	100	23.96s	5.878s	1.527s
ChunkInsert	10000	1000	1000	12.59s	1.459s	0.379s
SetInsert	10000	1000	1000	12.81s	1.850s	0.480s
ChunkInsert	10000	1000	2^{31}	12.56s	0.996s	0.259s
SetInsert	10000	1000	2^{31}	12.73s	1.738s	0.451s

Tabulka 4.1. Porovnání *chunk based* a *set based* přístupů pro vkládání dokumentů do Azure Cosmos DB

BenchmarkDotNet	v0.13.12
Operating system	macOS Sonoma 14.4.1 (23E224) [Darwin 23.4.0]
Processor	Apple M1 Pro, 1 CPU, 8 logical and 8 physical cores
.NET	8.0.100
Run strategy	ColdStart
Iteration count	5

Tabulka 4.2. Specifikace systému použitého pro benchmark

4.3 Adamijak.Cosmos.Extensions

V jazyce C# existuje koncept *extension methods*, který umožňuje rozšiřovat implementace tříd o nové metody. Velkou výhodou *extension methods* je fakt, že programátor nemusí znát implementaci třídy, kterou obohacuje či k ní mít přístup. Lze tak rozšiřovat třídy třetích stran nebo třídy, které jsou součástí .NET runtime, například třídu `System.String`. *Extension methods* jsou statické metody statické třídy, jejichž první parametr je objekt označený klíčovým slovem `this`. Příklad použití *extension methods* pro třídu `System.String` je vyobrazen níže. [28]

```
public static class StringExtensions
{
    private const string End = "...";
    public static string Shorten(this string str, int maxLength)
    {
        if (str.Length <= maxLength)
        {
            return str;
        }

        return string.Concat(str.AsSpan(0, maxLength-End.Length), End);
    }
}
```

Díky takto definované *extension method* lze volat metodu `Shorten` pomocí tečkové notace nad každou instancí třídy `string`. Příkladem je následující kód.

```
var short = "HelloWorld!!!".Shorten(8);
Console.WriteLine(short); // "Hello..."
```

Namespace `Adamijak.Cosmos.Extensions` definuje *extension methods* pro třídy Azure Cosmos SDK a .NET runtime, jako je například třída `Container` nebo `FeedIterator<T>` z namespace `Microsoft.Azure.Cosmos` nebo `IAsyncEnumerable<T>` z namespace `System.Collections.Generic`. Třídy `FeedIteratorExtensions` a `AsyncEnumerableExtensions` jsou popsány v Kapitole 4.3.1 a v Kapitole 4.3.2.

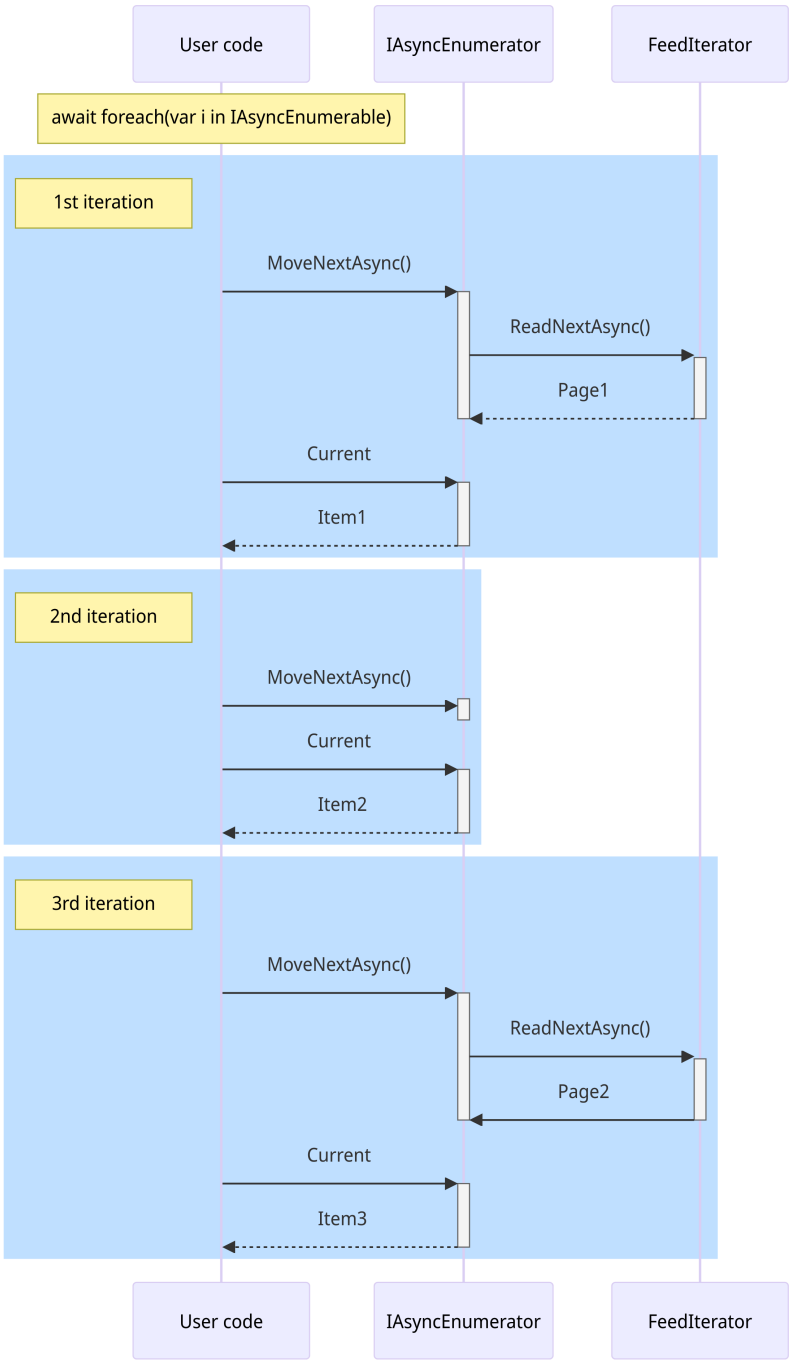
4.3.1 FeedIteratorExtensions

Typickou operací při práci s dokumenty uloženými v Azure Cosmos DB kontejneru je iterace po stránkách. Azure Cosmos SDK poskytuje třídu `Microsoft.Azure.Cosmos.FeedIterator<T>`, která umožňuje iterovat dokumenty pomocí položky `HasMoreResults` a metody `ReadNextAsync`. Metoda `ReadNextAsync` vrací stránku dokumentů. Třída `Adamijak.Cosmos.Extensions.FeedIteratorExtensions` implementuje *extension method* s názvem `ToAsyncEnumerable<T>` pro třídu `FeedIterator<T>`. Použitím této metody lze převést instanci třídy `FeedIterator<T>` na `System.Collections.Generic.IAsyncEnumerable<T>`. `IAsyncEnumerable<T>` lze použít pro iteraci nad kolekcí prvků, které jsou načítány asynchronně, například stránkováním. Výhodou je, že pro iteraci přes `IAsyncEnumerable<T>` může programátor použít klasický `foreach` cyklus s klíčovým slovem `await` a nemusí se starat o správu stránkování. Příklad použití `IAsyncEnumerable<T>` s `foreach` cyklem je vyobrazen níže. [14, 28]

```
await foreach (var item in feedIterator.ToAsyncEnumerable())
{
    Console.WriteLine(item);
}
```

Díky abstrakci z příkladu nemusí být patrné, že kolekce není načtena celá najednou v paměti. Pokud `IAsyncEnumerator<T>` nemá v paměti načtenou další stránku, musí si ji vyžádat. Výhodou je, že za použití `ToAsyncEnumerable` *extension method* může programátor k `FeedIterator<T>` přistupovat jako ke klasické kolekci. Rozdíl mezi klasickou kolekcí načtenou v paměti a `IAsyncEnumerable<T>` pocítí pouze v odezvě, pokud se zrovna načítá další stránka.

Na Obrázku 4.7 je zobrazeno, jak probíhá iterace nad kolekcí implementující `IAsyncEnumerable<T>`. Při prvním volání metody `MoveNextAsync` je načtena první stránka do paměti o velikosti dvou prvků. Následné volání položky `Current` vrátí první prvek. Další volání metody `MoveNextAsync` nemusí načítat další stránku, protože prvky jsou načtené v paměti, a pouze nastaví položku `Current` na druhý prvek. Jelikož už nejsou v paměti načteny žádné další prvky, musí třetí volání metody `MoveNextAsync` načíst další stránku a vrátit první prvek z této stránky, který je celkově třetí v pořadí. [28]



Obrázek 4.7. Příklad volání funkcí při iterování za použití kolekce implementující `IAsyncEnumerable<T>`

4.3.2 AsyncEnumerableExtensions

Třída `Adamijak.Cosmos.Extensions.AsyncEnumerableExtensions` obsahuje *extension method* s názvem `ForEachAsync<T>` pro třídu `System.Collections.Generic.IAsyncEnumerable<T>`. Metoda rozšiřuje `IAsyncEnumerable<T>` o možnost asynchronní iterace a zpracování prvků kolekce. Metoda `ForEachAsync<T>` implementuje *set based* řízení souběžně běžících úloh. Jejími parametry jsou asynchronní funkce s typem `Func<T, Task>`, maximální počet souběžně běžících úloh a `Cancellation-`

`Token`. Metoda propaguje výjimky z asynchronních funkcí a zastaví zpracování prvků kolekce, pokud je to vyžádané pomocí `CancellationToken`.

4.4 Adamijak.Cosmos.Queue

`Adamijak.Cosmos.Queue.CosmosQueue` je třída, která vznikla za účelem zjednodušení přístupu k dokumentům uložených v Azure Cosmos DB. Třída obsahuje metody pro vkládání, mazaní a iteraci dokumentů. Metoda `ToAsyncEnumerable` je popsána v Kapitole 4.4.1 a metoda `DeleteAsync` v Kapitole 4.4.2.

4.4.1 Metoda ToAsyncEnumerable

Metoda `ToAsyncEnumerable` využívá třídu `FeedIterator<T>` a `FeedIteratorExtensions` k implementování kolekce, přes kterou lze asynchronně iterovat. Jedním z argumentů je funkce, která definuje dotaz, díku kterému lze filtrovat a řadit dokumenty. Příklad použití metody `ToAsyncEnumerable` je vyobrazen v kódu níže. Kód vypíše všechny položky z fronty seřazené podle času poslední modifikace, které mají nastavenou položku `Enabled` na hodnotu `true`.

```
var items = queue.ToAsyncEnumerable(q =>
    q.Where(i => i.Enabled)
        .OrderBy(i => i.Timestamp));
await foreach (var item in items)
{
    Console.WriteLine(item);
}
```

4.4.2 Metoda DeleteAsync

Asynchronní metoda `DeleteAsync` smaže dokument z Azure Cosmos DB kontejneru podle identifikátoru a *partition key*. Pomocí této metody lze implementovat *at least once* a *at most once* zpracování dokumentů. *At least once* zpracování zaručuje, že dokument bude zpracován alespoň jednou. Příkladem implementace je následující kód. Z příkladu je patrné, že dokument bude smazán až po úspěšném dokončení funkce `ProcessAsync`.

```
var items = queue.ToAsyncEnumerable();
await foreach (var item in items)
{
    await ProcessAsync(item);
    await queue.DeleteAsync(item);
}
```

At most once zpracování zaručuje, že dokument bude zpracován nejvýše jednou. Příklad *at most once* zpracování dokumentů je zobrazen níže. Dokument je z Azure Cosmos DB smazán v nezávislosti na tom, jestli funkce `ProcessAsync` vyhodí výjimku nebo ne.

```
var items = queue.ToAsyncEnumerable();
await foreach (var item in items)
{
    try { await ProcessAsync(item); }
    finally { await queue.DeleteAsync(item); }
}
```

4.4.3 Fronty zpráv vs CosmosQueue

Přední výhodou CosmosQueue oproti tradičním frontám zpráv, jako je například Azure Event Hub nebo Apache Kafka, je možnost flexibilního dotazování nad zprávami pomocí SQL. Díky tomu lze implementovat například filtrování nebo řazení dokumentů, což lze použít například pro implementaci prioritní fronty. Další výhodou je, že pokud již uživatel používá Azure Cosmos DB, nemusí do projektu přidávat závislost na další službě. Azure Event Hub ve standardním plánu umožňuje ukládat zprávy o velikosti maximálně 1MB, zatímco pro Azure Cosmos DB je maximální velikost dokumentu 2MB při použití NoSQL API a 16MB při použití MongoDB API. [14]

4.5 CosmosTest

CosmosTest je testovací projekt sloužící k testování implementace z namespace `Adamijak.Cosmos`. Obsahuje unit testy a integrační testy pro třídy `CosmosQueue` a `CosmosQueueExtensions`. Pro testování byl použit testovací framework MSTest, který na rozdíl od xUnit používá jasně pojmenované atributy pro označení testů.

Příkladem integračního testu může být metoda `BulkCreate`, jejíž kód je vyobrazen níže. Test vloží definovaný počet dokumentů do Azure Cosmos DB pomocí metod `ForEachAsync` a `CreateAsync` a zkontroluje jejich počet.

```
[TestMethod]
public async Task BulkCreate()
{
    var items = Enumerable.Range(0, Count)
        .Select(i => new QueueItemPk
        {
            Id = Guid.NewGuid().ToString(),
            Pk = nameof(BulkCreate),
        })
        .ToList();
    await items.ForEachAsync(queue.CreateAsync, 100);
    Assert.AreEqual(Count, await queue.CountAsync());
}
```

Pro automatické testování byla použita služba GitHub Actions, která je popsána v Kapitole 4.5.1. Testy se spouští při pushnutí do větve main, a trigger spuštění testů reaguje pouze na změny v adresáři `./Cosmos` a `./CosmosTest`.

Pro testování a CI/CD Microsoft poskytuje Azure Cosmos DB emulator. Azure Cosmos DB emulator je dostupný jako Docker image na Docker Hub pod jménem `microsoft/azure-cosmos-emulator-linux` nebo na MCR jako `mcr.microsoft.com/cosmosdb/linux/azure-cosmos-emulator`. Emulátor k testování nebyl použit z důvodu, že nepodporuje ARM architekturu. Namísto toho byla použita instance Azure Cosmos DB, ke které se připojuje pomocí *connection string*. *Connection string* je nastavován pomocí proměnné prostředí, což jej umožňuje uložit do GitHub Secrets a použít v GitHub Actions. V lokálním prostředí se *connection string* nastavuje do proměnné prostředí pomocí `.runsettings` souboru. Díky tomu není nutné v kódu rozlišovat mezi lokálním a CI/CD prostředím. [14]

4.5.1 GitHub Actions

GitHub Actions je CI/CD služba poskytovaná společností GitHub. Tato služba umožňuje definovat workflows pomocí YAML souborů, díky kterým lze například spouštět testy, sestavení nebo publikovat balíky. Každé GitHub workflow se skládá z triggeru a z jednoho nebo více jobs. Trigger definuje událost, která workflow spustí. Touto událostí může být například pushnutí do větve, vytvoření release nebo manuální spuštění. Job se skládá ze steps, které se vykonávají sekvenčně. Step může být například bash skript nebo separátní akce vytvořená třetí stranou a publikovaná na GitHub Marketplace. [29]

4.6 NuGet Adamijak.Cosmos

NuGet je správce balíků pro .NET. Pomocí něj lze instalovat a aktualizovat balíky a závislosti. NuGet Gallery je centrální repozitář NuGet, kam lze publikovat vlastní balíky. NuGet Gallery je veřejně přístupná online[2]. [30]

Adamijak.Cosmos je NuGet balík obsahující namespace `Adamijak.Cosmos`, který je publikovaný do NuGet Gallery[3]. Do NuGet Gallery je balík publikován automaticky pomocí GitHub Actions při vytvoření nového release. Workflow spustí nejdříve sestavení a testy. Po úspěšném sestavení a otestování se balík publikuje. Build cílí na poslední LTS verzi .NET runtime – `net8.0`. Bylo tak rozhodnuto z důvodu, že Microsoft označuje předchozí verze .NET jako *end of life*, aby donutil vývojáře updatovat a používat nejnovější verze .NET. NuGet Adamijak.Cosmos bude publikován pro .NET runtime `net8.0` do té doby, dokud nebude označen jako *end of life*. Build necílí na předchozí LTS .NET runtime `net6.0`, jehož podpora bude ukončena 12. listopadu 2024. [31]

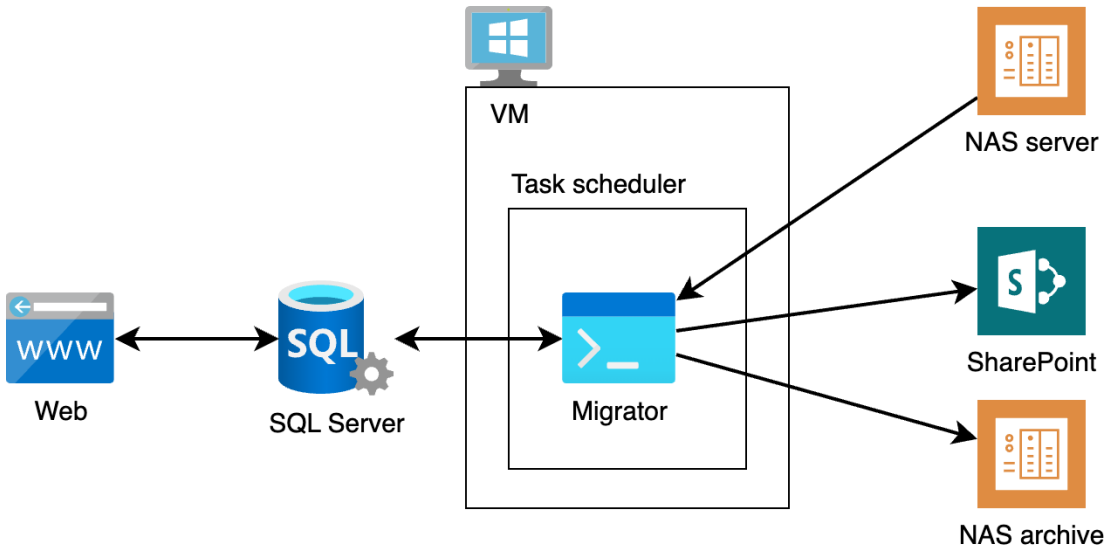
[2] `https://www.nuget.org`
[3] `https://www.nuget.org/packages/Adamijak.Cosmos`

Kapitola 5
Řešení Migrator

Zadáním pro řešení Migrator bylo vytvořit aplikaci, která zmigruje soubory z NAS do cílových úložišť. Funkční požadavky zákazníka na řešení by se daly shrnout do následujících bodů:

- 1. Uživatel si vybírá složky, jejichž obsah chce zmigrovat
- 2. Administrátor potvrzuje migraci
- 3. Možnými cílovými úložišti jsou nová NAS, SharePoint, Teams a uživatelský OneDrive
- 4. Aplikace detekuje změny na souborech, migruje upravené soubory, a je navržena tak, aby běžela za normálního provozu zákazníka
- 5. Aplikace respektuje výjimky v podadresářích
- 6. Pro klasifikaci a šifrování souborů je použit AIP
- 7. V cloudu by se neměly vyskytovat neklasifikované soubory

Z těchto požadavků vyplynula architektura vyobrazená na Obrázku 5.1. První dva body požadavků pokrývá plánovací web, ve kterém má uživatel možnost naplánovat migraci složky a administrátor ji potvrdit. Samotný Migrator byl navržen jako konzolová aplikace, která je hostována na virtuálním stroji, jež je připojený k NAS. Migrator buď soubory kopíruje mezi NAS, nebo je nahrává do cloudu pomocí Microsoft Graph API. Při použití plánovače úloh je možné aplikaci spouštět v pravidelných intervalech. Pro získání složky k migraci, detekci změn na souborech, detekci výjimek a udržení stavu migrace byl použit SQL Server. Pro implementaci byl zvolen jazyk C#, pro který Microsoft poskytuje knihovny pro práci s Microsoft Graph a AIP. AIP je popsáno v Kapitole 5.1. Pro detekci nezašifrovaných souborů na cloudu bylo vyvinuto samostatné řešení, které je popsáno v Kapitole 6.



Obrázek 5.1. Architektura řešení Migrator

Migrator byl navržen tak, aby běh jedné instance aplikace zmigroval právě jednu složku. Každý takový běh lze rozdělit čtyř kroků:

- Získání složky k migraci
- Detekce souborů
- Migrace souborů
- Dokončení migrace

Jednotlivé kroky jsou popsány v Kapitole 5.2, Kapitole 5.3, Kapitole 5.4 a Kapitole 5.5.

5.1 Azure Information Protection

Azure Information Protection umožňuje nastavovat citlivost a šifrování souboru pomocí klasifikačních štítků. V závislosti na typu nastaveného štítku se soubor zašifruje. Formáty Microsoft Office, jako je například DOCX nebo XLSX, nativně podporují nastavení štítků z klientských aplikací, jako je například Microsoft Word nebo Microsoft Excel. Pro nastavování štítku lze použít i AIP klienta, pomocí kterého lze nastavit štítek i na nenativních formátech souborů. Z nenativních formátů jako PNG vznikne po zašifrování speciální typ souboru – PFILE, který nese informace o štítku a o původním názvu a formátu souboru. Díky použití AIP lze omezit na Entra ID skupiny nebo jednotlivé uživatele. [1]

5.2 Získání složky k migraci

Pro započetí migrace je nutné získat složku k migraci. Migrator vyčte z databáze složku, jejíž datum poslední migrace je nejmenší ze všech naplánovaných složek. Zároveň musí být pro započetí migrace naplánovaná složka v jednom ze stavů `Ready`, `Succeeded` nebo `Failed`. Schválně se vyčítají i složky, které jsou ve stavech `Succeeded` a `Failed`. Složky se statusem `Succeeded` se znovu vyčítají, aby na nich mohlo dojít k detekci změněných souborů. Složky se statusem `Failed` se vyčítají, protože v předchozím běhu došlo k chybě a k vyhození výjimky, kvůli které nešlo pokračovat ve vykonávání migrace. Díky tomu je vyřešeno automatické opakování migrace v případě chyby.

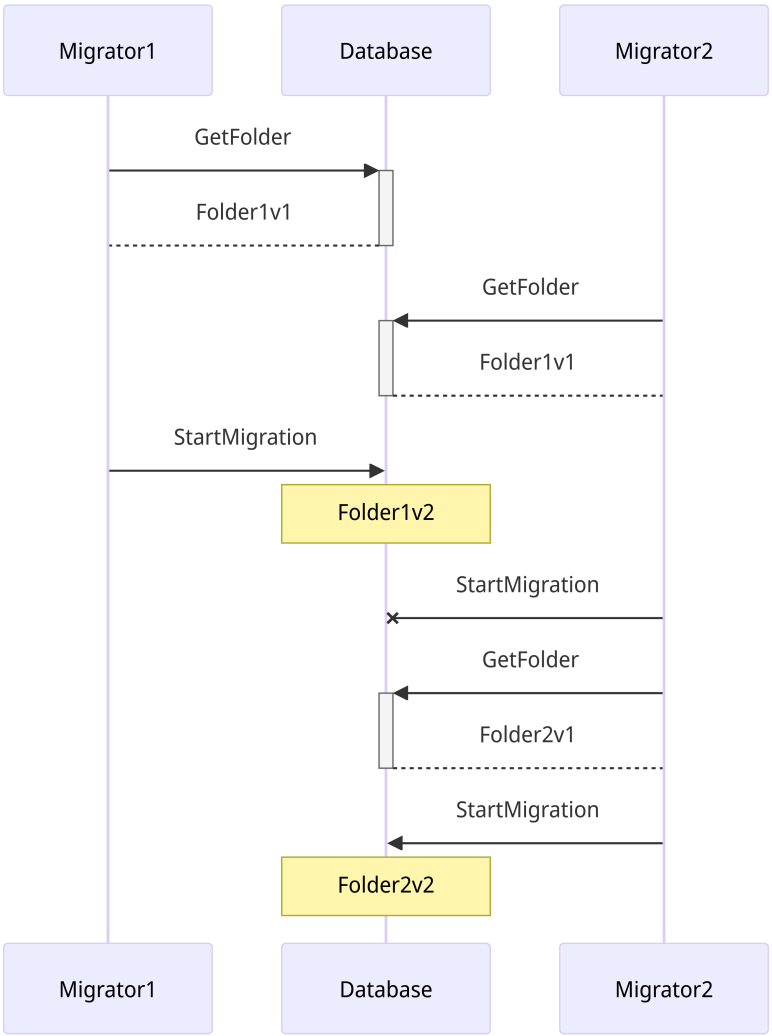
Aplikace migrator je navržena tak, že lze paralelizovat na úrovni složek a jednotlivých souborů. Souběžné zpracování souborů je implementované pomocí asynchronního programování, *set based* přístupu a implementace z namespace `Adamijak.Cosmos`. Orchestrace běží v jedné úloze na jednom vlákně, není proto nutné řešit konflikty souběžného přístupu. Aplikaci lze paralelizovat na úrovni složek spuštěním více instancí programu. Protože dvě instance programu by mohly v databázi přistoupit k jedné stejné složce, bylo nutné vyřešit konflikty souběžného přístupu. K řešení konfliktů byla použita metoda *optimistic concurrency control*, která je popsaná v Kapitole 5.2.1.

5.2.1 Optimistic concurrency control

Optimistic concurrency control je metoda pro řešení konfliktů souběžného přístupu bez použití zámku nad sdíleným zdrojem. V řešení Migrator byla metoda OCC použita, protože je to jednoduchý způsob pro řešení konfliktů souběžného přístupu na úrovni procesů. Při použití této metody není potřeba komunikace mezi procesy, sdílených zámků či implementace transakce na úrovni databáze. Použití této metody je vhodé zejména, když se očekává, že konflikty nastávají zřídka, a že řešení konfliktů je časově zanedbatelné vůči času běhu samotné migrace. [32]

OCC bylo implementováno pomocí EntityFramework a atributu `TimestampAttribute` z namespace `System.ComponentModel.DataAnnotations`. EntityFramework při zápisu do databáze porovná hodnotu položky označené atributem `TimestampAttribute` uložené v paměti s hodnotou v databázi. Pokud se hodnoty liší, EntityFramework vyhodí výjimku `DbUpdateConcurrencyException`.

Příklad na Obrázku 5.2 znázorňuje řešení konfliktu souběžného přístupu pomocí metody OCC. Obrázek znázorňuje dvě instance aplikace Migrator, které vyčetly stejnou složku k migraci. První instanci se podařilo nastavit status složky na `InProgress` a začít s migrací díky tomu, že verze složky uložené v databázi byla stejná jako verze složky vyčtené aplikací. Druhá instance aplikace Migrator se také pokusila nastavit status složky na `InProgress`, ale v databázi již byla uložena novější verze složky. V reakci na to EntityFramework vyhodí výjimku `DbUpdateConcurrencyException`. Vyhozenou výjimku lze ošetřit a poté vyčíst novou složku z databáze. Nově vyčtenou složkou již nemůže být původní složka, kterou vyčetla první instance aplikace, protože není ve stavu `Ready`, `Succeeded` nebo `Failed`.



Obrázek 5.2. Příklad řešení konfliktů souběžného přístupu, pomocí metody OCC

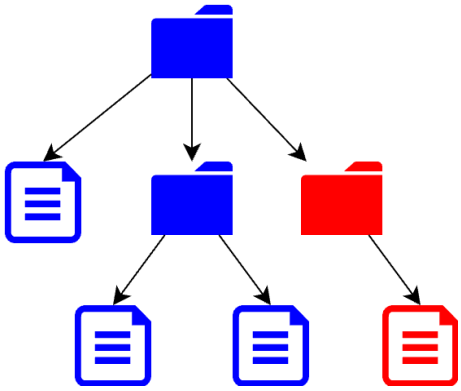
Příkladem implementace OCC je následující funkce, která nastaví status na `InProgress` a vrátí složku migraci.

```
public async Task<Folder?> StartAsync()
{
    while (true)
    {
        var folder = await db.Folders
            .OrderBy(m => m.StartTime)
            .FirstOrDefaultAsync(m => m.Status == Status.Ready
                || m.Status == Status.Succeeded
                || m.Status == Status.Failed);

        if (folder is null)
        {
            return null;
        }

        folder.Status = Status.InProgress;
        folder.StartTime = DateTime.UtcNow;

        try
        {
            await db.SaveChangesAsync();
            return folder;
        }
        catch (DbUpdateConcurrencyException e)
        {
            foreach (var entry in e.Entries)
            {
                await entry.ReloadAsync();
            }
        }
    }
}
```

5.3 Detekce souborů

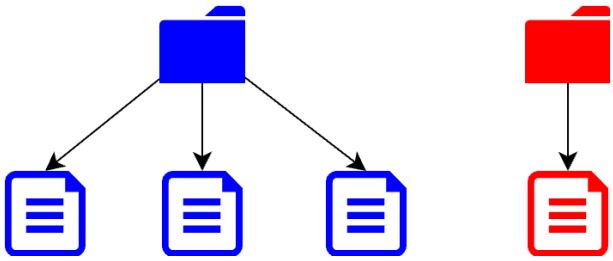
Pro migrovanou složku si Migrator vyčte všechny její podsložky. Pro každou podsložku ke zpracování si Migrator vyčte všechny její soubory, které pak uloží do databáze. Pokud soubor již v databázi existuje, aplikace upraví hodnoty záznamu, aby odpovídaly aktuálnímu stavu souboru. Během detekce souborů se soubory filtrují podle jména pomocí regulárního výrazu definovaného v konfiguračním souboru. Filtrované soubory nejsou migrovány. Příkladem mohou být soubory, které začínají znaky ~$, což jsou dočasné soubory vytvořené Microsoft Office aplikacemi. Lze také přeskakovat podle přípon souborů – například zástupce s příponou *.lnk*.

Při procházení může Migrator narazit na podsložku, která byla také zadána pro migraci. Například jako je vyobrazeno na Obrázku 5.3 pomocí červené složky. V takovém případě ji ignoruje, protože se o migraci této složky postará další instance aplikace.



Obrázek 5.3. Stromová struktura složek

Po dokončení obcházení složek a detekci souborů se stromová struktura logicky zploští. Každá složka k migraci má v databázi uloženou kolekci souborů ke zpracování, jako je znázorněno na Obrázku 5.4. Díky tomu lze jednoduše iterovat jednotlivé soubory a migrovat je. Další výhodou je, žel lze migraci jednoduše paralelizovat na úrovni migrovaných složek a jednotlivých souborů.



Obrázek 5.4. Stromová struktura složek po zploštění

5.4 Migrace souborů

Při samotné migraci aplikace iteruje přes kolekci souborů migrované složky uložených v databázi. Soubory se k migraci vyčítají podle jejich stavu. Soubory ve stavu `Ready` nebo `Failed` se migrují vždy. Pokud je soubor ve stavu `Succeeded` a byl od poslední migrace upraven, tak se také zmigruje.

V případě, že se soubor migruje do nové NAS, je k migraci použita funkce `Copy` třídy `System.IO.File`. Pokud je za cíl zvolen SharePoint, tak se nejdříve nastaví AIP štítek, a poté se soubor nahraje pomocí Microsoft Graph API. Při nastavování štítku se nejdříve vytvoří kopie souboru, na kterou se štítek aplikuje. Pokud soubor má štítek přiřazený, vytvoří se kopie souboru beze změny. Pokud soubor štítek nemá nastavený, aplikuje se štítek, který je nastavený jako výchozí pro danou SharePoint knihovnu. Pokud daná SharePoint knihovna nemá nastavený výchozí štítek, tak se použije globální výchozí štítek nastavený v konfiguračním souboru. Při migraci do cloudu se nerozlišuje mezi SharePoint, Teams a OneDrive, protože všechna tři cílová úložiště mají za sebou vytvořenou SharePoint site. Díky tomu lze používat stejný endpoint a API pro nahrávání souborů. Sériové nahrávání souborů bylo pomalé, proto bylo implementováno souběžné nahrávání souborů pomocí asynchronního programování a implementace popsané v Kapitole 4. Když nebylo nahrávání souborů omezeno na počet úloh, velmi rychle došlo k přetížení Graph API, vrácení *HTTP status code 429 Too Many Requests* a vyhození výjimky. Kvůli tomu byl maximální počet souběžně běžících úloh omezen *set based* metodou pro řízení souběžně běžících úloh.

Pokud je při migraci souboru vyhozená výjimka, tak se soubor přesune do stavu `Failed` a hodnota položky `RetryCount`, která určuje počet pokusů o migraci, se zvýší o jeden. Pokud se při migraci souboru vyskytla dočasná chyba, jako je například nedostupnost služby nebo chyba na síti, bude vyřešena při dalším běhu aplikace. V případě výskytu trvalé chyby se dá nastavit konfigurační hodnota `max_retry_count`, která určuje maximální počet pokusů o migraci předtím, než soubor přejde do stavu `HardFailed`.

5.5 Dokončení migrace

Po migraci všech souborů se nastaví stav složky na `Succeeded` a vypočítají se statistiky migrace. Výjimky by se neměly vyskytovat, protože aplikace je navržena tak, aby je odchytávala a řešila na úrovni migrace souborů. Pokud je i přes to výjimka vyhozena, bude odchycena, a složka si nastaví stav na `Failed`, aby nezůstala ve nedefinovaném či rozpracovaném stavu.

5.6 Nasazení

Nasazení se provádí ručně přes RDP clienta, protože zákazník používá RDP jako jediný způsob přístupu k virtuálním strojům. Pro kompilaci je použit program *task* – moderní alternativa utility *make*. Celý projekt se jedním příkazem zkompiluje a zabalí do zip archivu, který je poté ručně nahrán a rozbalen na serveru. Plánovač úloh spouští PowerShell skript, který má na starost spuštění aplikace Migrator. Ve skriptu je definovaná statická cesta k binárnímu souboru aplikace a konfiguračnímu souboru. Díky tomu lze pro nasazení nové verze pouze nahradit starý binární soubor novým, a není tak nutné vypínat staré úlohy nebo rušit jejich běh.

5.7 Konfigurace

Běh aplikace Migrator je řízen konfiguračním souborem, který je programu předán v podobě cesty jako argument při spuštění. Konfigurace je načtena jednou při startu aplikace a nemění se dynamicky za běhu. Díky tomu lze Migrator konfigurovat na úrovni instancí aplikace. Například lze nastavením konfigurační hodnoty `is_enabled` vypnout celý chod aplikace pro každou další instanci aplikace bez nutnosti upravení rozvrhu v plánovači úloh. Pro autentizaci aplikace vůči Microsoft Entra ID je použit certifikát, který byl uložen v úložišti certifikátů Windows. Byl použit z bezpečnostních důvodů, aby nedošlo k úniku secretu s aplikačními právy do Microsoft Graph API pro tenant zákazníka. V plánovači úloh je aplikace spouštěna v kontextu servisního účtu, který má práva pro čtení certifikátu z úložiště certifikátů. Díky tomu mohou heslo k servisnímu účtu znát pouze správci serveru a nemusí být sdíleno se správci aplikace. Certifikát aplikace vyhledává na základě thumbprint, který je uložen v konfiguračním souboru. Ke konfiguraci je použit konfigurační soubor v INI formátu, který je popsán v Kapitole 5.7.1.

5.7.1 INI formát

INI je historický konfigurační formát, který však stále nabízí výhody oproti jiným moderním formátům. Výhodou oproti formátům YAML a TOML je, že součástí .NET runtime je pro INI formát implementován `IniConfigurationProvider`. Díky tomu je

možné konfiguraci načíst bez nutnosti použití externí knihovny či implementace vlastního parseru. Příklad konfiguračního souboru v INI formátu je znázorněn v kódu níže.

```
; Service is enabled
is_enabled = true

[auth]
# Client certificate thumbprint
client_cert = "12345678"
```

Pro označení sekce používá INI formát hranaté závorky s názvem, které definují začátek sekce. Díky tomu je tento formát oproti JSON formátu, kde sekce vznikají vnořením objektů, značně přehlednější a čitelnější. Dalšími výhodami oproti JSON formátu jsou například možnost vkládání komentářů a fakt, že názvy klíčů a řetězců nemusí být v uvozovkách.

5.8 Logování

Pro jednoduchost aplikace loguje do souboru a při každém běhu je vytvořen nový soubor s názvem `log-{timestamp}.txt` ve složce logs. Aby šlo v takto vytvořených log souborech snadno vyhledávat, obsahují standardně řádky logu závažnost logované zprávy a timestamp ve formátu ISO 8601. Pro samotné vyhledávání je vhodné použít skriptovací jazyk nebo specializovaný nástroj jako je například *grep*.

5.9 Testování

Pro aplikaci nebylo implementováno komplexní automatické testování. Jelikož se jedná o jednorázovou aplikaci, byla by implementace testů neefektivní. Aplikace se testuje manuálně ve vývojovém a produkčním prostředí. V produkčním prostředí se aplikace po každém nasazení otestuje na pilotních uživatelích a pilotních souborech.

5.10 Problémy s migrovanými soubory

Při testování aplikace bylo zjištěno, že aplikace AIP štítku na digitálně podepsané PDF soubory rozbije digitální podpis. Z toho důvodu bylo rozhodnuto, že PDF soubory budou vyjmuty z automatické migrace. V návaznosti na to bylo zjištěno, že AIP nedokáže štítkovat soubory, které jsou digitálně podepsáné či jinak šifrované. Například emaily uložené ve formátu MSG, které byly šifrovány pomocí S/MIME. Tyto soubory byly také vyjmuty z automatické migrace.

Kapitola 6
Řešení SharePoint protector

Po migraci souborů na platformu SharePoint zákazník potřeboval vynutit klasifikaci pomocí AIP štítků i na souborech, které byly na cloud nahrány uživateli po samotném aktu migrace. To vedlo k vytvoření řešení SharePoint protector, které periodicky obchází SharePoint úložiště a detekuje soubory modifikované uživatelem. Pokud modifikované soubory nejsou klasifikovány, SharePoint protector aplikuje příslušný štítek. Za SharePoint úložiště je považováno vše, co je vystavené pod Microsoft Graph API jako entita `drive`. Příkladem drive je OneDrive uživatele nebo soubory přístupné přes tým v Microsoft Teams. Všechny drives jako backend využívají SharePoint dokumentové knihovny.

Návrh řešení vyplývá z požadavku zákazníka na funkcionality. Funkcionality řešení SharePoint protector lze shrnout do tří bodů:

- SharePoint protector – detekce nezabezpečených souborů na drive a jejich klasifikace popsané v Kapitole 6.1
- Externí sdílení – odšifrování souborů na žádost uživatele popsané v Kapitole 6.2
- SharePoint unprotector – odšifrování souborů na žádost administrátora pro celý drive nebo podle přípony souboru popsané v Kapitole 6.3

Řešení SharePoint protector je rozděleno do tří projektů – Core, CoreApi a BotApi. Core tvoří celý backend řešení. Je implementován pomocí Azure Functions, Azure Cosmos DB, Cosmos Queue a další implementace popsané v Kapitole 4. Core tvoří architektura založená na frontách, která byla zvolena kvůli tomu, aby šel jednoduše limitovat objem dat protékající systémem. Zároveň díky rozdělení operací do více funkcí a front, kdy každá funkce vykonává právě jednu konkrétní operaci, bylo docíleno toho, že je systém snadno rozšiřitelný a odolnější vůči chybám. Například o funkcionalitu SharePoint unprotector zákanzík požádal až po implementaci CosmosQueue. Rozšíření řešení spočívalo pouze v přidání dvou nových funkcí a dvou Azure Cosmos DB kontejnerů, díky čemuž rozšíření neovlivnilo zbytek projektu. Implementace nové funkcionality se zjednodušila použitím již implementovaných tříd `CosmosQueue` a `AsyncEnumerableExtensions`. Aby nedošlo k přesycení front, byl implementován konfigurovatelný treshold, který určuje maximální počet záznamů uložených ve frontě.

Projekt CoreApi tvoří rozhraní pro komunikaci s Core, pomocí kterého lze například vytvářet nové požadavky na odšifrování nebo povolit či zakázat šifrování pro jednotlivé drives. BotApi tvoří backend pro bota, se kterým uživatelé interagují pomocí Microsoft Teams. Pro vývoj řešení byla použita platforma .NET a jazyk C#. Tato platforma byla zvolena kvůli tomu, že se dobře integruje s prostředím zákazníka a se službami Microsoft Azure, Microsoft Graph a Microsoft 365. Pro logování všech projektů řešení byla použita služba Azure App Insights.

6.1 Funkcionalita SharePoint protector

Architektura funkcionality SharePoint protector je vyobrazena na Obrázku 6.1. Skládá se ze tří funkcí – `DriveGetter`, `ChangeTracker` a `SharePointProtector`. Funkce `Dri-`

veGetter slouží k získání seznamu všech drives, které se v vyskytují na SharePoint zákazníka. Pro každý takový drive, který splňuje filtrační podmínky, je vytvořen nový dokument v kontejneru Drives v Azure Cosmos DB. Funkce `ChangeTracker` slouží k detekci změn na souborech v drives. Pokud `ChangeTracker` detekuje změnu a soubor projde skrz filtrační podmínky, vloží nový dokument do kontejneru `SharePointProtectorQueue`. Funkce `SharePointProtector` iterativně prochází `SharePointProtectorQueue` a pro každý záznam musí udělat tři operace:

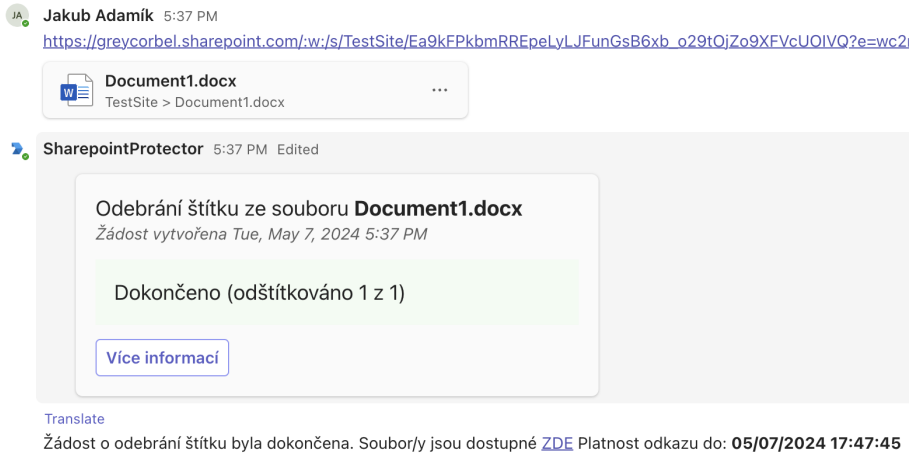
- 1. Stáhnout soubor z SharePoint
- 2. Aplikovat štítek na kopii souboru
- 3. Nahradit soubor na SharePoint

Všechny tři Azure Functions jsou časovaně spouštěny pomocí *timer trigger*. Funkce `DriveGetter` je navržena tak, aby se spouštěla v řádu hodin až dnů, zatímco `ChangeTracker` a `SharePointProtector` v řádu desítek sekund až hodin. Ukázka kódu funkce `SharePointProtector` je uvedena v Příloze B.

Obrázek 6.1. Architektura funkcionality SharePoint protector

6.2 Funkcionalita Externí sdílení

Uživatelé si mohou zažádat o odšifrování souboru. Žádost se vydává pomocí bota a adaptivních karet v Microsoft Teams. Bot vytvoří žádost na odštítkování v reakci na zprávu obsahující odkaz na soubor nebo složku. Žádost podléhá schvalovacímu workflow a musí jí schválit manažer uživatele. Zpracovaná žádost na odštítkování je vyobrazena na Obrázku 6.2.



Obrázek 6.2. Zpracovaná žádost na externí sdílení

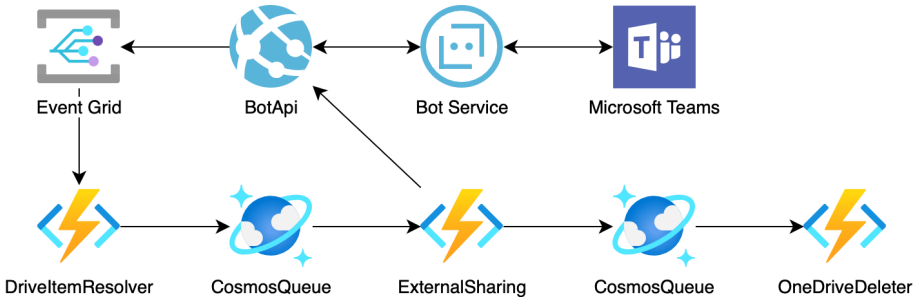
Po schválení žádosti dostane BotApi notifikaci přes Bot Service a pošle událost do Azure Event Grid. Azure Event Grid byl použit z důvodu, že nedoručený event se automaticky pokusí doručit znovu například v případě nedostupnosti služby.

Architektura funkcionality Externí sdílení je znázorněna na Obrázku 6.3. Funkce `DriveItemResolver` je spouštěna pomocí Azure Functions Event Grid trigger. `DriveItemResolver` získá informace o souboru či složce z Microsoft Graph API. Pokud je

položkou soubor, vytvoří pro něj záznam v `ExternalSharingQueue`. Pokud je položkou složka, vytvoří pro každý soubor uložený v této složce nový záznam v `ExternalSharingQueue`. Funkce `ExternalSharing` je spouštěna časovaně a iteruje přes záznamy v `ExternalSharingQueue`. Pro každý záznam z `ExternalSharingQueue` funkce udělá pět následujících operací:

- Stáhne soubor z SharePoint
- Aplikuje nebo odebere štítek pro externí sdílení na kopii souboru
- Nahraje soubor na OneDrive žadatele
- Pošle notifikaci o odšifrování na BotApi, aby zobrazil uživateli odkaz na odštítkovaný soubor
- Vytvoří záznam v `OneDriveDeleterQueue` pro odstranění souboru z OneDrive

Funkce `OneDriveDeleter` je spouštěna časovaně a pro každý záznam z `OneDriveDeleterQueue` odstraní soubor z OneDrive uživatele. Soubor je odstraněn pouze tehdy, pokud překročil určitý časový limit od jeho poslední modifikace.

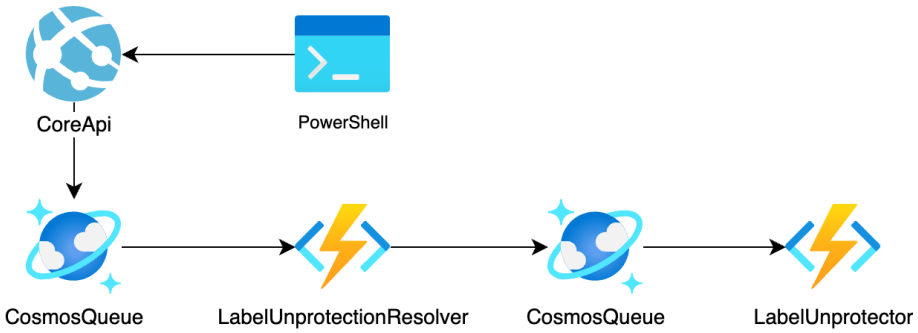


Obrázek 6.3. Architektura funkcionality Externí sdílení

6.3 Funkcionalita SharePoint unprotector

Díky položce `IsEnabled` na dokumentech v kontejneru Drives je možné povolit nebo zakázat aplikování štítků pro daný drive. Pokud se administrátor rozhode, že nechce aplikovat štítky na souborech na daném drivu, může tuto položku nastavit na `false`. Na takovém drivu se nebudou aplikovat štítky na nově modifikované soubory, ale stávající soubory zůstanou bez změny. Díky funkcionalitě SharePoint unprotector je možné odebrat štítky na souborech uložených na drivu.

Architektura funkcionality SharePoint unprotector je znázorněna na Obrázku 6.4. Díky CoreApi lze například vylistovat všechny drives, které mají položku `IsEnabled` nastavenou na `false` a vložit je do kontejneru `LabelUnprotectionResolverQueue`. Funkce `LabelUnprotectionResolver` je časovaně spouštěna a iteruje přes záznamy v `LabelUnprotectionResolverQueue`. Pro každý takový záznam funkce získá informace o všech souborech na daném drivu. Pokud projdou přes filtr, tak je vloží jako záznam do `LabelUnprotectionQueue`. Funkce `LabelUnprotection` je spouštěna časovaně, iteruje přes záznamy v `LabelUnprotectionQueue` a odstraňuje štítky ze souborů. Pokud je zapnutý filtr na odštítkování pouze souborů s určitou příponou, tak funkce odšifruje pouze soubory s touto příponou. Nevýhodou je, že se musí stáhnout každý soubor, protože není dopředu známa jeho originální přípona. Například soubor s názvem *file1.pfile* může mít po odšifrování název *file1.png*, což není patrné z názvu šifrovaného souboru. Kvůli tomu byl na SharePoint založen skrytý sloupec `OriginalFileName`. V tomto sloupci se ukládá originální název souboru před aplikováním štítku, díky čemuž může funkce `LabelUnprotectionResolver` ignorovat nechtěné soubory.



Obrázek 6.4. Architektura funkcionality SharePoint unprotector

6.4 Konfigurace

Pro konfiguraci projektu byly použity Azure App Configuration, Azure Key Vault a Azure Functions App settings. Všechny konfigurační hodnoty jsou uloženy v Azure App Configuration, ke které se aplikace ověřuje pomocí Managed Identity, respektive pomocí třídy `DefaultAzureCredential` popsané v Kapitole 6.4.1 Aby se konfigurační hodnoty nemusely vkládat ručně, byl vytvořen YAML soubor, který lze pomocí Azure CLI nahrát do Azure App Configuration. Všechna hesla a *connection strings* jsou uložena v Azure Key Vault a do Azure App Configuration se k nim přistupuje pomocí reference. Některé hodnoty musely být uloženy v Azure Functions App settings, protože je musí hosting znát ještě před startem aplikace. Příkladem těchto hodnot je `AzureWebJobsStorage` a `DriveGetterSchedule`. Nicméně i tyto hodnoty jsou uloženy v Azure App Configuration a do Azure Functions App settings jsou vloženy pouze v podobě reference.

6.4.1 DefaultAzureCredential

Třída `DefaultAzureCredential` z namespace `Azure.Identity` implementuje třídu `Azure.Core.TokenCredential` a umožňuje získat autorizační token v nezávislosti na implementaci autentizační metody. Interně třída instancuje třídy specializované pro určitý typ autentizace s různou prioritou. Priorita tříd je určena následujícím pořadím.

- EnvironmentCredential
- WorkloadIdentityCredential
- ManagedIdentityCredential
- SharedTokenCacheCredential
- VisualStudioCredential
- VisualStudioCodeCredential
- AzureCliCredential
- AzurePowerShellCredential
- AzureDeveloperCliCredential
- InteractiveBrowserCredential

Největší prioritu má třída `EnvironmentCredential`, která získává autorizační token na základě proměnných prostředí. Třída `DefaultAzureCredential` zprvu zkouší získat token pomocí třídy `EnvironmentCredential`, pokud se jí to nepodaří, tak pokračuje s další třídou v pořadí. Díky třídě `DefaultAzureCredential` lze aplikaci spustit lokálně i v produkčním prostředí Azure bez změny kódu. Například lze v případě lokálního běhu použít pro autentizaci proměnné prostředí a pro produkční prostředí Managed Identity.

6.5 Nasazení

Pro nasazení řešení SharePoint protector byla použita služba GitHub Actions. Nasazuje se do dvou prostředí s názvy Dev a Prod. Dev je prostředí testovacího tenantu, ve kterém se testují nové verze řešení. Prod je produkční prostředí zákazníka. Pro každý projekt řešení – Core, CoreApi a BotApi vznikl jeden workflow, který je rozdělený do dvou částí – kompilaci a nasazení. Pro kompilaci je použita *composite action*, která zkompiluje projekt a nahraje zkompilovaný balík mezi artefakty. Díky použití *composite action* je možné použít stejnou definici workflow pro všechny projekty.

Pro nasazení je použita matrix strategie, která umožňuje nasazovat do více prostředí. Pro každé prostředí vznikl v nastavení repozitáře nový environment. Každé takové prostředí obsahuje proměnné a secrets jemu příslušící. Nasazení do obou prostředí podléhá schvalovacímu procesu. Do Azure se GitHub Actions autentizuje pomocí *publish profile*, což je soubor ve formátu XML, který obsahuje všechna potřebná data a secrets pro nasazení Azure Functions nebo Azure App Service.

6.5.1 Dev vs Prod prostředí

Během nasazování aplikace se vyskytly potíže kvůli rozdílům mezi Dev a Prod prostředí. Ve vývojovém prostředí většina Azure služeb používala k přístupu RBAC a Managed Identity. Tento způsob se nedařilo zprovoznit v Prod prostředí, pravděpodobně vinou konfigurace prostředí. Například místo proměnné prostředí `AzureWebJobsStorage__accountName`, která pro autentizaci používá jméno instance Azure Storage a RBAC, bylo nutné použít proměnnou `AzureWebJobsStorage` a *connection string*. V důsledku toho nelze služby konfigurovat stejně v obou prostředích, což může vést k chybě v konfiguraci a následné nefunkčnosti celého systému.

6.6 Testování

Pro aplikaci byly napsány jednoduché unit testy, které testují zejména funkce, jež pracují se soubory na disku. Příkladem mohou být unit testy funkcí pro práci s příponami souborů, které testují správné přejmenování souboru. Komplexní testy nebyly implementovány, protože zadání bylo zákazníkem často měněno. Úprava testů by v tomto případě byla časově nezanedbatelná a neefektivní. Aplikace se testuje ručně, nejdřív ve vývojovém prostředí a poté v prostředí produkčním. V produkčním prostředí se aplikace po každém nasazení manuálně otestuje. Po každém nasazení se otestuje pouze základní funkčnost SharePoint protector a Externího sdílení. Každá další nová funkcionalita se testuje zvlášť na testovacích uživatelích a souborech.

Kapitola 7
Závěr

Řešení Migrator a SharePoint protector byla navržena, implementována a předána zákazníkovi. Řešení Migrator bylo implementováno jako konzolová aplikace, která umožňuje migrovat soubory z NAS do SharePoint. Aplikace při migraci nastavuje AIP štítky a je paralelizovatelná na dvou úrovních. Souběžné zpracování jednotlivých souborů bylo implementováno pomocí asynchronního programování. Zpracování složek pro migraci bylo paralelizováno pomocí několika instancí aplikace a konflikty sdíleného přístupu byly vyřešeny metodou Optimistic Concurrency Control.

Řešení SharePoint protector udržuje správně klasifikované soubory pomocí AIP štítků na souborech uložených na platformě SharePoint. Bylo implementováno pomocí Azure Functions a CosmosQueue, díky čemuž je škálovatelné a flexibilní. Řešení bylo nasazeno a je provozováno na cloudové platformě Microsoft Azure. Obě řešení byla zákazníkem úspěšně akceptována a otestována na pilotních uživatelích a souborech.

Dalším krokem v řešení SharePoint protector je implementace komplexních automatických *end to end* testů a dynamické konfigurace díky použití Azure App Configuration. Automatické testování umožní ověřit správnost implementace a dodá jistotu při dalším vývoji. Dynamická konfigurace umožní změnu chování aplikace bez nutnosti jejího restartování. Díky nabytým znalostem byla obě řešení navržena tak, aby byla snadno škálovatelná, vysoce dostupná a snadno rozšiřitelná.

V rámci práce vznikly tři *open source* GitHub repozitáře – adamijak/cosmos, adamijak/azure-auth a adamijak/azure-http. Řešení z repozitáře adamijak/cosmos, publikované jako NuGet balík, obsahuje implementaci několika tříd pro práci s Azure Cosmos DB, která umožňuje zejména snadnou iteraci nad dokumenty uložených v Azure Cosmos DB. Implementace byla založena na dlouhodobé zkušenosti se službou, díky které šlo identifikovat a publikovat funkcionality, jež lze nyní použít i v dalších projektech. Řešení z repozitáře adamijak/azure-auth obsahuje projekt, který umožňuje bezpečnou autentizaci vůči Entra ID v aplikačním kontextu za použití certifikátu. Token získaný touto aplikací je možné použít pro testování Microsoft 365 služeb v aplikačním kontextu. Pomocí kolekce HTTP dotazů z repozitáře adamijak/azure-http lze jednoduše vytvářet a sdílet HTTP dotazy na Microsoft 365 služby za využití již používaných vývojových prostředí. To bylo využito pro testování schopností Microsoft Graph API a detekci chyb v implementaci Microsoft Graph API SDK.

Literatura

[1] Microsoft. *Azure documentation* [online]. Redmond: Microsoft, © 2024 [vid. 2024-05-08]. Dostupné na https://learn.microsoft.com/en-us/azure.

[2] Yang, Alvin, Siva Kantamneni, Ying Li, Awel Dico, Xiangang Chen, Rajesh Subramanyan a Liang-Jie Zhang. *Services - SERVICES 2018*. Cham: Springer International Publishing, 2018 [vid. 2024-04-13]. ISBN 978-3-319-94471-5. Dostupné na DOI 10.1007/978-3-319-94472-2. Dostupné na https://link.springer.com/book/10.1007/978-3-319-94472-2.

[3] Zhao, Jun-Feng a Jian-Tao Zhou. Strategies and Methods for Cloud Migration. *International Journal of Automation and Computing*. 2014, ročník 11, č. 2, s. 143-152 [vid. 2024-04-16]. ISSN 1476-8186. Dostupné na DOI 10.1007/s11633-014-0776-7. Dostupné na http://link.springer.com/10.1007/s11633-014-0776-7.

[4] Stack Exchange Inc. *Stack Overflow Developer Survey 2022* [online]. New York: Stack Exchange Inc, © 2024 [vid. 2024-04-02]. Dostupné na https://survey.stackoverflow.co/2022/#section-most-popular-technologies-cloud-platforms.

[5] Stack Exchange Inc. *Stack Overflow Developer Survey 2023* [online]. New York: Stack Exchange Inc, © 2024 [vid. 2024-04-02]. Dostupné na https://survey.stackoverflow.co/2023/#most-popular-technologies-platform-prof.

[6] Soewito, Benfano, Ford Lumban Gaol, Edi Abdurachman a others. A systematic literature Review: Risk analysis in cloud migration. *Journal of King Saud University - Computer and Information Sciences*. 2022, ročník 34, č. 6, s. 3111-3120 [vid. 2024-04-12]. ISSN 1319-1578. Dostupné na DOI 10.1016/j.jksuci.2021.01.008. Dostupné na https://www.sciencedirect.com/science/article/pii/S1319157821000082.

[7] Amin, Ruhul a Siddhartha Vadlamudi. Opportunities and Challenges of Data Migration in Cloud. *Engineering International*. 2021-04-02, ročník 9, č. 1, s. 41-50 [vid. 2024-04-12]. ISSN 2409-3629. Dostupné na DOI 10.18034/ei.v9i1.529. Dostupné na https://abc.us.org/ojs/index.php/ei/article/view/529.

[8] Mishra, Abhishek. *Microsoft Azure for Java Developers*. Berkeley: Apress, 2022 [vid. 2024-04-17]. ISBN 978-1-4842-8250-2. Dostupné na DOI 10.1007/978-1-4842--8251-9. Dostupné na https://link.springer.com/book/10.1007/978-1-4842-8251-9#bibliographic-information.

[9] Microsoft. *Azure Functions documentation* [online]. Redmond: Microsoft, © 2024 [vid. 2024-05-05]. Dostupné na https://learn.microsoft.com/en-us/azure/azure-functions.

[10] Microsoft. *Azure resources* [online]. Redmond: Microsoft, © 2024 [vid. 2024-05--06]. Dostupné na https://azure.microsoft.com/en-us/resources.

[11] Microsoft. *Azure Storage documentation* [online]. Redmond: Microsoft, © 2024 [vid. 2024-05-02]. Dostupné na `https://learn.microsoft.com/en-us/azure/storage`.

[12] Microsoft. *App Service documentation* [online]. Redmond: Microsoft, © 2024 [vid. 2024-05-01]. Dostupné na `https://learn.microsoft.com/en-us/azure/app-service`.

[13] Microsoft. *Azure Key Vault documentation* [online]. Redmond: Microsoft, © 2024 [vid. 2024-05-01]. Dostupné na `https://learn.microsoft.com/en-us/azure/key-vault/`.

[14] Microsoft. *Azure Cosmos DB documentation* [online]. Redmond: Microsoft, ©2024 [vid. 2024-05-06]. Dostupné na `https://learn.microsoft.com/en-us/azure/cosmos-db`.

[15] Microsoft. *Azure App Configuration documentation* [online]. Redmond: Microsoft, © 2024 [vid. 2024-05-05]. Dostupné na `https://learn.microsoft.com/en-us/azure/azure-app-configuration`.

[16] Microsoft. *Azure Monitor documentation* [online]. Redmond: Microsoft, © 2024 [vid. 2024-05-05]. Dostupné na `https://learn.microsoft.com/en-us/azure/azure-monitor`.

[17] Microsoft. *Application Insights overview* [online]. Redmond: Microsoft, © 2024 [vid. 2024-05-02]. Dostupné na `https://learn.microsoft.com/en-us/azure/azure-monitor/app/app-insights-overview`.

[18] Microsoft. *Azure Event Grid documentation* [online]. Redmond: Microsoft, © 2024 [vid. 2024-05-06]. Dostupné na `https://learn.microsoft.com/en-us/azure/event-grid/`.

[19] Microsoft. *Microsoft Entra documentation* [online]. Redmond: Microsoft, © 2024 [vid. 2024-05-04]. Dostupné na `https://learn.microsoft.com/en-us/entra/`.

[20] Microsoft. *Build interactive apps by using Microsoft Graph APIs* [online]. Redmond: Microsoft, © 2024 [vid. 2024-05-06]. Dostupné na `https://learn.microsoft.com/en-us/graph/patterns/interactive-applications`.

[21] Microsoft. *Microsoft Graph documentation* [online]. Redmond: Microsoft, © 2024 [vid. 2024-05-06]. Dostupné na `https://learn.microsoft.com/en-us/graph/`.

[22] Microsoft. *Microsoft identity platform documentation* [online]. Redmond: Microsoft, © 2024 [vid. 2024-05-04]. Dostupné na `https://learn.microsoft.com/en-us/entra/identity-platform/`.

[23] Jones, Michael B., John Bradley a Nat Sakimura. *JSON Web Token (JWT)* [RFC 7519]. [vid. 2024-05-06]. Dostupné na DOI 10.17487/RFC7519. Dostupné na `https://www.rfc-editor.org/info/rfc7519`.

[24] The Linux Foundation. *OpenAPI* [online]. San Francisco: The Linux Foundation, © 2024 [vid. 2024-05-05]. Dostupné na `https://www.openapis.org`.

[25] The Linux Foundation. *OpenAPI Specification v3.1.0* [online]. San Francisco: The Linux Foundation, © 2024 [vid. 2024-05-06]. Dostupné na `https://spec.openapis.org/oas/latest.html`.

[26] Advanced Micro Devices Inc. *AMD Ryzen Threadripper Processors Specifications* [online]. Santa Clara: Advanced Micro Devices Inc, © 2024 [vid. 2024-04-19]. Dostupné na `https://www.amd.com/en/products/processors/workstations/ryzen-threadripper.html`.

[27] Jiang, Xu, Nan Guan, Maolin Yang, Yang Wang, Yue Tang a Wang Yi. Real-Time Scheduling of Parallel Task Graphs With Critical Sections Across Different Vertices. *IEEE Transactions on Parallel and Distributed Systems*. 2022, ročník 33, č. 12, s. 4117-4133 [vid. 2024-04-27]. ISSN 1045-9219. Dostupné na DOI 10.1109/TPDS.2022.3179328. Dostupné na `https://ieeexplore.ieee.org/document/9785844/`.

[28] Microsoft. *.NET documentation* [online]. Redmond: Microsoft, © 2024 [vid. 2024-05-15]. Dostupné na `https://learn.microsoft.com/en-us/dotnet/`.

[29] GitHub Inc. *GitHub Docs* [online]. San Francisco: GitHub Inc, © 2024 [vid. 2024-05-14]. Dostupné na `https://docs.github.com/en`.

[30] Microsoft. *NuGet documentation* [online]. Redmond: Microsoft, © 2024 [vid. 2024-05-06]. Dostupné na `https://learn.microsoft.com/en-us/nuget/`.

[31] Microsoft. *.NET* [online]. Redmond: Microsoft, © 2024 [vid. 2024-05-04]. Dostupné na `https://dotnet.microsoft.com/en-us/`.

[32] Härder, Theo. Observations on optimistic concurrency control schemes. *Information Systems*. 1984, ročník 9, č. 2, s. 111-120 [vid. 2024-04-27]. ISSN 0306-4379. Dostupné na DOI 10.1016/0306-4379(84)90020-6. Dostupné na `https://linkinghub.elsevier.com/retrieve/pii/0306437984900206`.

Příloha A
Slovník zkratek

AIP	Azure Information Protection
API	Application Programming Interface
CI/CD	Continuous Integration/Continuous Deployment
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
GB	Gigabyte
GB/s	Gigabyte per second
GUID	Globally Unique Identifier
HTTP	Hypertext Transfer Protocol
ID	Identificator
INI	Initialization
JSON	JavaScript Object Notation
JWT	JSON Web Token
KB	Kilobyte
KQL	Kusto Query Language
MB	Megabyte
Mb/s	Megabit per second
MB/s	Megabyte per second
MCR	Microsoft Container Registry
NAS	Network Attached Storage
NFSv3	Network File System version 3
NoSQL	Not Only SQL
OCC	Optimistic Concurrency Control
RBAC	Role-Based Access Control
REST	Representational State Transfer
RU	Request Units
RU/s	Request Units per second
SDK	Software Development Kit
SQL	Structured Query Language
TiB	Tebibyte
TOML	Tom's Obvious Minimal Language
UPN	User Principal Name
URL	Uniform Resource Locator
vCPU	virtual CPU
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

Příloha B
Funkce SharePointProtector

```
namespace SharePointProtector.Core.Functions;
public class SharePointProtector(
    ILoggerFactory loggerFactory,
    IConfiguration config,
    CosmosQueue<SharePointProtectorQueueItem> cosmosQueue,
    GraphServiceClient graphClient,
    MipService mipService)
{
    ILogger logger = loggerFactory.CreateLogger<SharePointProtector>();
    int maxItemsConcurrent = config.MaxItemsConcurrent();

    [Function(nameof(SharePointProtector))]
    public Task Run(
        [TimerTrigger("%SHAREPOINT_PROTECTOR_SCHEDULE%",
        RunOnStartup = false,UseMonitor = false)] TimerInfo timer,
        CancellationToken ct) =>
        cosmosQueue.ToAsyncEnumerable(q => q.OrderBy(i => i.Timestamp))
            .ForEachAsync(async i =>
            {
                var inputFile = new FileInfo(Path.GetTempFileName());
                var outputFile = new FileInfo(Path.GetTempFileName());
                try
                {
                    await ProcessItem(i, inputFile, outputFile);
                }
                finally
                {
                    if (inputFile.Exists)
                    {
                        inputFile.Delete();
                    }

                    if (outputFile.Exists)
                    {
                        outputFile.Delete();
                    }
                }
            }, maxItemsConcurrent, ct);
```

```
    private async Task ProcessItem(
        SharePointProtectorQueueItem queueItem,
        FileInfo inputFile,
        FileInfo outputFile)
    {
        var path = Path.ChangeExtension(inputFile.FullName,
            Path.GetExtension(queueItem.ItemName));
        inputFile.MoveTo(path);

        try
        {
            await graphClient.DownloadById(inputFile,
            queueItem.DriveId, queueItem.ItemId);
            logger.LogInformation("Downloaded file with size: {Size}",
                inputFile.Length);

            var itemName = await mipService.SetLabel(inputFile,
                outputFile, queueItem);
            logger.LogInformation("Label set");

            if (queueItem.ItemName != itemName)
            {
                var actualName = await graphClient.RenameToNextAvailable(
                    queueItem.DriveId, queueItem.Id, itemName);
                logger.LogInformation("Renamed item to {Name}",
                    actualName);
            }

            var item = await graphClient.UploadFileByIdAsync(outputFile,
                queueItem.DriveId, queueItem.ItemId);
            logger.LogInformation("File uploaded. {ItemWebUrl}",
                item.WebUrl);
        }
        catch (Exception e)
        {
            logger.LogError(e, "Could not process item");
        }
        finally
        {
            await cosmosQueue.DeleteAsync(queueItem);
        }
    }
}
```