Analýza služeb kompilovaných v režimu Ahead-of-Time a Just-In-Time na platformě .NET

Bc. Noe Švanda

Diplomová práce
2024

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2023/2024

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Noe Švanda**
Osobní číslo:	**A22497**
Studijní program:	**N0613A140022 Informační technologie**
Specializace:	**Softwarové inženýrství**
Forma studia:	**Kombinovaná**
Téma práce:	**Analýza služeb kompilovaných v režimu Ahead-of-Time a Just-In-Time na platformě .NET**
Téma práce anglicky:	**Analysis of Services Compiled in Ahead-of-Time and Just-in-Time Modes on the .NET Platform**

Zásady pro vypracování

1. Proveďte rozbor režimů kompilace v dotNET, zvolte dvě kompilační metody.
2. Vytvořte mikroslužby ve frameworku dotnet s využitím těchto kompilačních metod.
3. Popište konfigurace a nasazení ve vybrané platformě.
4. Připravte monitorovací a vizualizační nástroje pro srovnání výkonu kompilací.
5. Navrhněte testovací scénáře nad aplikačním stackem pro obě kompilační metody.
6. Srovnejte výsledky testování a tyto prezentujte v interaktivní podobě.

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. KOKOSA, Konrad. Pro .NET memory management: for better code, performance and scalability. For professionals by professionals. New York: Apress, [2018]. ISBN 978-1484240267.
2. RICHTER, Jeffrey. CLR via C#: the common language runtime for .NET programmers. [4th ed.]. Redmond, Wash.: Microsoft Press, [2012]. ISBN 978-0735667457.
3. RICHARDSON, Chris. Microservices patterns: with examples in Java. Sebastopol, Calif.: O'Reilly Media, [2018]. ISBN 978-1617294549.
4. NICKOLOFF, James, KUENZIL, Steffen. Docker in action. 2nd ed. Greenwich, CT: Manning Publications, [2019]. ISBN 978-1617294761.
5. GARRISON, Josh, NOVA, Kelsey. Cloud native infrastructure: designing, building, and running scalable microservices applications. 1st ed. Sebastopol, Calif.: O'Reilly Media, [2017]. ISBN 978-1491984307.
6. GAMMELGAARD, Christian Horsdal. Microservices for .NET developers: a hands-on guide to building and deploying microservices-based applications using .NET Core. 2nd ed. Apress, [2021]. ISBN 978-1617297922.
7. LOCK, Andrew. ASP.NET Core in action. Greenwich. 2nd ed. CT: Manning Publications, [2021]. ISBN 978-1617298301.

Vedoucí diplomové práce: **doc. Ing. Petr Šilhavý, Ph.D.**
Ústav počítačových a komunikačních systémů

Datum zadání diplomové práce: **5. listopadu 2023**
Termín odevzdání diplomové práce: **13. května 2024**

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
DĚKAN
L.S.

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 5. ledna 2024

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomové práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové práce bude uložen v příruční knihovně Fakulty aplikované informatiky. Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má Univerzita Tomáše Bati ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne .....

podpis studenta

ABSTRAKT

Tato diplomová práce analyzuje možnosti kompilace služeb v režimu Ahead-of-Time a Just-In-Time na platformě .NET. Práce se zaměřuje na srovnání vývojového procesu, výstupu a výkonu služeb kompilovaných v obou režimech. Za tímto účelem je pro účely práce vytvořena platforma pro testování scénářů a monitorování. Výsledkem práce je srovnání výhod a nevýhod obou režimů kompilace a doporučení pro vývojáře.

Klíčová slova: .NET, kompilace, služby, telemetrie, trimming, obraz, stack, metriky

ABSTRACT

This thesis analyses the possibilities of compiling services in Ahead-of-Time and Just-In-Time mode on the .NET platform. The thesis focuses on comparing the development process, output and performance of services compiled in both modes. For this purpose, a scenario testing and monitoring platform is created for the purpose of the thesis. The result of the work is a comparison of the advantages and disadvantages of both compilation modes and recommendations for developers.

Keywords: .NET, compilation, services keywords, telemetry, trimming, image, stack, metrics

Děkuji svému vedoucímu práce, doc. Ing. Petru Šilhavému, Ph.D., za jeho cenné rady, trpělivost a ochotu věnovat mi svůj čas. Dále bych chtěl poděkovat své rodině a přátelům za podporu a pochopení během mého studia.

OBSAH

ÚVOD

Programovací jazyky jsou základním kamenem softwarovévo vývoje respektive celého moderního světa v období informací. Představují způsob, kterým vývojář komunikuje s virtuálním prostředím OS a následně HW rozhraním. Vývoj výkonu HW, znalostí a zkušeností vývojářů a požadavků na vyvíjené systémy byl hnacím strojem technologického rozvoje. Postupným vývojem přicházeli další a další variace programovacích jazyků, některé rozdílné inkeremntálně, jiné zcela diametrálně. Významným mezníkem v přístupu k tvorbě a běhu strojového kódu je vznik virtuálních strojů, které umožňují běh kódu nezávisle na HW. Tento přístup umožňuje vývojářům psát kód v jazyce, který je jim přirozený a následně jej spouštět na různých platformách.

Dotnet je platforma od společnosti Microsoft, která umožňuje vytvářet kód určený pro následnou kompilaci za běhu (Just-in-Time, dále JIT) a spuštění pomocí tzv. běhového prostředí (Common Language Runtime), jenž operuje jako virtuální stroj. Jedná se o relativně vyvinutou a zkušenou platformu s využitím v mnoha projektech a firmách. Přesto právě na této platformě byla dodatečně vyvinuta možnost pro PC platformy kompilace do nativního kódu (Ahead-of-Time, dále AOT), který je spouštěn přímo na OS a konkrétní architektuře HW. Tato funkce přichází do období rozmachu vývoje a migrace nativních cloudových řešení. Ty charakterizuje snaha dodávat pouze nezbytnou část infrastruktury a zploplatnit reálnou dobu běhu systému s režií. Právě v prostředí cloudu mají nastávat situace, kdy bude využití služeb zkompilovaných do nativního kódu výhodnější. V kterých případech však opravdu takto napsaný program exceluje či selhává a lze kvantifikacovat rozdíly mezi kopilací pro běhové prostředí JIT a AOT kompilací?

Tato práce se zabývá porovnáním vývojového procesu, charakteristik a výkonu JIT a AOT kompilovaných služeb na platformě Dotnet. Cílem je zjistit, zda a v jakých případech je možné využít AOT kompilace pro zvýšení výkonu a zlepšení chování aplikací. Výsledkem práce je kvantifikace, respektive srovnání výkonu a chování JIT a AOT kompilace na platformě Dotnet. Na základě těchto výsledků je možné posoudit a doporučit vhodné případy pro využití AOT kompilace.

I. TEORETICKÁ ČÁST

1 PLATFORMA .NET

Platforma .NET od společnosti Microsoft představuje sadu nástrojů k vývoji aplikací v podporovaných jazycích. Tato platforma je multiplatformní a umožňuje vývoj aplikací pro operační systémy jako Windows, Linux, MacOS ale i pro mobilní platformy. Vývojáři mohou využívat nástroje pro vývoj webových aplikací, desktopových aplikací, mobilních aplikací a dalších. Platforma .NET je postavena na dvou hlavních nástrojích. Prvním z nich je *Common Language Runtime* (CLR), běhové prostředí zodpovídající za běh aplikací. Druhým nástrojem je *.NET CLI*, konzolový nástroj-rozhraní, zodpovědné za interakci s dílčími nástroji v platformě. [4]

1.1 Historie

Využití runtime prostředí, respektive v originální podobě virtuálního stroje, má historický původ. V dřívějších dobách byli programátoři limitování nutností kompilace kódu do nativní reprezentace přímo pro architekturu systému. Kód vytvořen pro jednu konkrétní architekturu se zpravidla neobešel bez modifikací, pokud měl fungovat i na odlišné architektuře.

V průběhu 90. let 20. století představila společnost Sun-Microsystems virtuální stroj Java Virtual Machine (JVM). Jedná se o komponentu runtime prostředí Javy, která zprostředkovává spuštění specifického kódu, správu paměti, vytváření tříd a typů a další. Kompilací Javy do tzv. bytecode (Intermediate Language, zkráceně IL), tedy provedením mezikroku v procesu transformace zdrojového kódu do strojového kódu, je získána reprezentace programu, jenž běží na každém zařízení s implementovaným JVM. V rámci JVM dochází k finálním krokům mezi které patří interpretace (JIT kompilace) bytecode do nativního kódu pro cílovou architekturu systému.

Microsoft v reakci na JVM vydal v roce 2000 první .NET Framework, který umožňoval spouštět kód v jazyce C# na operačním systému Windows. Cílem prvních verzí .NET Framework nebylo primárně umožnit vývoj pro různé zařízení a operační systémy, ale zprostředkovat lepší nástroje pro vývoj aplikací. V roce 2014 byla vydána první multiplatformní verze .NETu, .NET Core, který umožňoval spouštět kód v jazyce C# na operačních systémech Windows, Linux a macOS. [4]

1.2 Architektura

Platforma .NET je postavena na několika klíčových komponentách, které zajišťují běh aplikací a poskytují nástroje pro vývoj aplikací. Mezi nejdůležitější komponenty patří:

- **CLR** - CLR je základním kamenem frameworku .NET a poskytuje běhové prostředí pro spouštění aplikací .NET. Překládá IL do nativního kódu, spravuje alo-

kaci paměti a garbage collection, zajišťuje zpracování výjimek (exceptions). CLR také kontroluje datové typy, interoperabilitu a zprostředkovává služby nezbytné pro spouštění nejrůznějších aplikací .NET.

- **.NET CLI** (konzolové rozhraní) - Všestranný nástroj pro vývoj, kompilaci a nasazení aplikací .NET prostřednictvím rozhraní příkazové řádky. Podporuje širokou škálu operací, od vytváření projektů a správy závislostí až po testování a publikování aplikací. Prostředí .NET CLI je multiplatformní a umožňuje sjednocení rozhraní uživatelských nástrojů pro vývoj aplikací .NET.

- **MSBuild** - Engine používaný v platformě .NET, který umožňuje sestavovat aplikace a vytvářet balíčky pro nasazení. Tento nástroj používá k organizaci a řízení procesu sestavení projektový soubor *csproj* na bázi XML. Tím je zajištěna kontrola nad kompilací a průběhem sestavení. V rámci MSBuild lze doplnit vlastní úlohy a cíle kompilace, což poskytuje flexibilitu sestavení pro komplexní procesy ve velkých projektech.

- **.NET SDK nástroje** - Soubor nástrojů a knihoven podporujících vývoj, debugging a testování aplikací .NET. Zahrnují různé CLI a GUI nástroje, které pomáhají vývojářům spravovat práci s kódem, optimalizovat výkon a zajistit kvalitní výstup programu v platformě .NET.

- **Roslyn** - Roslyn je sada kompilátorů platformy .NET, která poskytuje bohaté API pro analýzu kódu. Umožňuje vývojářům používat implementace kompilátorů jazyka C# a Visual Basic jako služby. Roslyn zlepšuje výkonnost vývojářů poskytnutím funkcí jako je refaktoring, generování kódu a sémantická analýza.

- **NuGet** - Správce balíčků pro platformu .NET. dodává standardizovanou metodu správy externích knihoven, na nichž závisí aplikace v .NET. Zjednodušuje proces inkorporace knihoven, systémových i třetích stran, do projektu. Rovněž spravuje závislosti, čímž zajišťuje, že projekty zůstávají aktuální a kompatibilní. Tento nástroj je téměř nezbytný pro vývoj na platformě .NET, neboť umožňuje modulární vývoj softwaru.

TODO: Přidat obrázek architektury nástrojů platformy .NET

1.3 Frameworky a technologie

Platforma .NET poskytuje mnoho frameworků a technologií pro vývoj aplikací. Jednotlivé frameworky plní různé role a poskytují různé úrovně funkcionality pro vývoj aplikací. Z hlediska struktury a účelu je lze kategorizovat následujícím způsobem.

1.3.1 Foundtation frameworky

Jedná se o komplexní frameworky poskytující základní funkcionalitu pro vývoj aplikací. Představují stavební kameny a fundamentální sadu nástrojů platformy .NET.

- **.NET** - Hlavní framework platformy. .NET je robustní framework pro vývoj softwaru. Podporuje tvorbu a provoz moderních aplikací a služeb. Původně známý jako .NET Framework a primárně zaměřen na prostředí Windows, s příchodem .NET Core a novějších verzí se vyvinul v modulární platformu s open-source zdrojovým kódem známou jednoduše jako .NET. Umožňuje vývojářům vytvářet aplikace, které jsou škálovatelné, výkonné a multiplatformní.
- **Mono** - Open-source implementace .NET Framework. Mono obsahuje vlastní verzi CLR, která je kompatibilní s .NET verzí CLR a podporuje různé jazyky a knihovny .NET. Primárním cílem Mono je nabídnout .NET kompatibilitu pro systémy jako Linux a MacOS, kde .NET nebyl před vydáním .NET Core tradičně podporován. Zvláštní význam nachází v mobilním vývoji, kde slouží jako běhové prostředí, které pohání aplikace Xamarin.iOS a Xamarin.Android. Kromě toho hraje Mono klíčovou roli v uživatelském rozhraní .NET Multiplatform App UI (MAUI), což je evoluce Xamarinu do modernějšího a komplexnějšího frameworku pro multiplatformní vývoj.

1.3.2 Specializované frameworky

Představují sady nástrojů a knihoven zaměřených na konkrétní oblasti vývoje aplikací a zjednodušují tvorbu specifických typů aplikací. Zaměřují se na konkrétní potřeby vývoje, jako jsou webové, mobilní nebo desktopové aplikace, a umožňují vývojářům efektivně implementovat funkcionalitu optimalizovanou pro konkrétní platformu.

- **ASP.NET** - Robustní framework pro vytváření webových aplikací a služeb. Je součástí ekosystému .NET navržený tak, aby umožňoval vývoj vysoce výkonných, dynamických webových stránek, RESTful API a webových aplikací v reálném čase. ASP.NET podporuje jak webové formuláře, tak architekturu MVC (Model-View-Controller). S uvedením ASP.NET Core byl framework přepracován pro cloudovou nasazením škálevatolnost, vývoj napříč platformami a vysoký výkon. Poskytuje komplexní základ pro vývoj moderních webových aplikací, které lze spustit jak na Linuxu, Windows tak MacOS. ASP.NET Core také představuje Blazor, který umožňuje vývojářům používat C# při vývoji webu, což dále zvyšuje všestrannost ekosystému. Vývojářům, kteří chtějí využít .NET pro vývoj webu, poskytuje ASP.NET komplexní a flexibilní sadu nástrojů pro vytváření všech řešení, od malých webů až po složité webové platformy.

- **MAUI** - Moderní specializovaný framework pro vývoj aplikací napříč platformami v rámci ekosystému .NET. Umožňuje vývojářům vytvářet aplikace pro Android, iOS, MacOS a Windows z jedné kódové základny (codebase). Zakládá na populárních konceptech z Xamarin.Forms a zároveň rozšiřuje jeho možnosti na desktopové aplikace. .NET MAUI zjednodušuje vývojový proces tím, že poskytuje jednotnou sadu rozhraní API pro vývoj uživatelského rozhraní na všech platformách s možností přístupu k funkcím specifickým pro platformu v případě potřeby. Podporuje moderní vývojové vzory a nástroje, včetně MVVM, datové vazby a asynchronního programování, což usnadňuje vytváření sofistikovaných a citlivých aplikací. Předchůdcem MAUI je framework Xamarin, který sloužil pro vytváření mobilních aplikací na platformě .NET.

- **Blazor** - Specializovaný framework v rámci ekosystému .NET, který zprostředkovává vývojářům tvorbu interaktivních webových uživatelských rozhraní pomocí C# namísto JavaScriptu. Blazor může běžet na serveru (Blazor Server), kde zpracovává požadavky a komunikuje s uživatelským rozhraním pomocí knihovny SignalR. Nebo také v prohlížeči pomocí WebAssembly (Blazor WebAssembly), kdy dochází k přeložení C# kódu do nativního kódu WASM a je spouštěn přímo ve webovém prohlížeči vedle tradičních webových technologií, jako jsou HTML a CSS. Umožňuje vývojářům využít znalosti .NET pro komplexní vývoj webových aplikací a vytvářet bohaté webové aplikace běžící na straně klienta v prohlížeči. Architektura Blazor je založená na komponentách a usnadňuje jejich opětovné použití pro uživatelského rozhraní. Podporuje modulární vývojový přístup. Poskytuje možnost vyvíjet webové aplikace a zůstat v ekosystému .NET.

1.3.3 Knihovny

Knihovny představují soubor funkcí a tříd, které mohou být použity při vývoji ve více aplikacích. Typicky představují logicky oddělenou a obecnou část funkcionality aplikace. Umožňují distribuovat běžnou funkcionalitu napříč různými projekty. Knihovny v .NET mohou být tvořeny binárnimi DLL soubory (Dynamic Link Libraries) nebo organizované jako samostatný projekt v rámci kontejneru projektů solution. Distribuce knihoven je obecně prováděna jejich zabalením pomocí nástroje NuGet a sdíleny přes internetová úložiště.

Běžnou praxí tvůrce platformy, programovacího jazyka nebo frameworku je poskytnutí sad knihoven, které usnadňují vývoj aplikací. Zároveň tyto knihovny zpravidla implementují nejběžnější funkcionality, které mohou programátoři vyžadovat. Typicky se jedná o přístup k souborovému systému, síti, databázím, grafickému rozhraní a další.

Následující seznam obsahuje některé z nejběžněji používaných knihoven v .NET:

- **System** - Tato základní knihovna obsahuje třídy, typy a rozhraní, které umožňují a podporují širokou škálu operací na úrovni systému, jako jsou vstupy a výstupy (IO), vlákna, kolekce, diagnostika a další. Je nezbytná prakticky pro každou aplikaci .NET.
- **System.IO** - Dodáva funkcionalitu čtení z datových proudů, souborů a zápis do nich a práci se souborovým systémem.
- **System.Net** - Obsahuje třídy a abstrakce pro síťovou komunikaci, včetně HTTP, socketů TCP/IP a SMTP pro elektronickou poštu.
- **System.Data** - Poskytuje přístup k datovým zdrojům, jako jsou databáze nebo XML soubory, a obsahuje ADO.NET pro přístup k datům ze serveru SQL Server a dalších databází.
- **System.Collections** - Rozhraní a třídy, které definují různé kolekce objektů, jako jsou seznamy, fronty, bitová pole, hašovací tabulky a slovníky.
- **System.Linq** - Zaštiťuje dotazování nad kolekcemi objektů pomocí LINQ (Language Integrated Query).
- **System.Threading** - Zprostředkovává správu vláken, synchronizační primitiva nebo například thread pool. Umožňuje vyvíjet paralelizované aplikace.
- **System.Security** - Spravuje ověřování, autorizaci a šifrování, a je základem pro vývoj bezpečných aplikací.
- **Entity Framework** - ORM (Object-Relational Mapping) framework, který umožňuje vývojářům pracovat s databázemi pomocí objektově orientovaného přístupu. Poskytuje abstrakci nad databázovými systémy a umožňuje vývojářům pracovat s daty pomocí objektů a tříd.

Kromě knihoven poskytovaných společností Microsoft existuje mnoho knihoven třetích stran. Za vývojem těchto knihoven mohou stát vývojařské komunity nebo být vydány velkými společnostmi. Běžně tyto knihovny navazují na sadu funkcí poskytovaných Microsoftem a rozšiřují je o další novou funkcionalitu. Mezi příklady nejznámějších knihovny třetích stran v .NET patří Dapper, AutoMapper, Newtonsoft.Json a další.

1.4 Nástroje .NET

Platforma .NET zprostředkovává širokou sadu nástrojů za účelem tvorby, sestavení a spuštění aplikace. Mezi nejdůležitější lze zařadit následující:

1.4.1 IDE

Neméně důležitým prvkem vývoje aplikací je integrované vývojové prostředí (IDE). I když není povinné, pro spoustu vývojářů je jeho použití neodmyslitené. IDE je nástroj, který zprostředkovává vývoj aplikací, správu projektů, debuggování a další. IDE poskytuje uživatelské rozhraní, které umožňuje vývojářům vytvářet, upravovat a testovat kód. Zprostředkovává nástroje pro správu projektů, jako jsou sestavení, testování a publikace. Umožňuje provádět různorodé operace nad aplikací, jako je refaktorování kódu, hledání chyb a ladění.

Jedním z nejpoužívanějších IDE je Visual Studio, vydávané společností Microsoft. Visual Studio poskytuje prvotřídní podporu pro vývoj na platformě .NET. Mezi další populární IDE patří Visual Studio Code a JetBrains Rider.

1.4.2 Balíčky

Pro jednoduchou distribuci knihoven, jak systémových tak třetích stran, je využíván nástroj pro správu balíčků NuGet. Projekt, jenž má být distribuován je buďto opatřen atributem *<PackageOnBuild>* a sestaven nebo je využito příkazu *dotnet pack*.

Takto vytvořené balíčky lze distribuovat např. přes NuGet.org, což je veřejný repozitář knihoven, který je dostupný pro všechny vývojáře. Možná je také implementace vlastních řešení. Distribuované knihovny jsou jednoduše importovatelné do projektu a umožňují snadnou správu závislostí.

1.4.3 Dokumentace

Dokumentace je důležitou součástí vývoje aplikací. Poskytuje informace o tom, jak používat nástroje a technologie, které jsou součástí platformy .NET. Dokumentace obsahuje informace o API, knihovnách, nástrojích a dalších součástech platformy .NET. Dokumentace je dostupná online na oficiálních webových stránkách platformy .NET a obsahuje podrobné informace o mnoha aspektech vývoje aplikací v .NET. Dotnet dokumentace je k nalezení na webu `https://docs.microsoft.com/en-us/dotnet/`.

1.4.4 Jazyky

Základem aplikace je zdrojový kód, který je v případě platformy .NET reprezentován nejčastěji jedním ze podporovaných jazyků, Mezi nejčastěji využívané patří VisualBasic (VB), C# a F#.

- **C#** - Představuje všestranný, objektově orientovaný jazyk navržený tak, aby umožnil vývojářům vytvářet širokou škálu bezpečných a robustních aplikací, které

běží na .NET Framework. Kombinuje sílu a flexibilitu C++ s jednoduchostí jazyka Visual Basic.

- **F#** - Funkční jazyk, který také podporuje imperativní a objektově orientované programování. Primárně je vhodný pro vědecké aplikace a aplikace náročné na data. Zakládá na silném typování, umožňuje stručný, robustní a výkonný kód.
- **Visual Basic** - Programovací jazyk vyvinutý společností Microsoft. VB, představený v roce 1991, byl navržen jako uživatelsky přívětivé programovací prostředí založené na jazyce BASIC; jeho drag-and-drop rozhraní umožňovalo snadné vytváření GUI. Tento přístup zpřístupnil programování širšímu okruhu uživatelů a kladl důraz na rychlý vývoj aplikací (RAD).

1.5 Aplikační struktura

Základním stavebním prvkem aplikace v .NET je projektový soubor. Jedná se o soubor na bázi XML disponující příponou *.csproj*. V rámci něj dochází ke konfiguraci a deklaraci, jak bude .NET CLI, respektive nástroje sestavení, s projektem pracovat. Zároveň jsou zde definovány závislosti na další projekty a knihovny. Mezi základní charakteristiky běžně určené v projektovém souboru patří verze .NET, verze projektu/assembly, seznam závislostí, konfigurace pro buildování, testování a publikaci.

Pro tvorbu složitějších aplikací je možné využít více projektových souborů, které jsou následně propojeny. Tento způsob je využíván především v případě větších aplikací, které jsou rozděleny do více částí. Propojení a vazby mezi více projekty v aplikaci je definované pomocí tzv. solution souboru. Jedná se o kontejnerový soubor s příponou *.sln*, jenž popisuje závislosti mezi projektovými soubory, konfigurace sestavení a nasazení a správu pomocných souborů.

Mezi další běžně používané známé konfigurační soubory patří následující:

- **appsettings.json** - obsahuje nastavení aplikace
- **launchsettings.json** - deklaruje konfiguraci pro spuštění aplikace
- **Directory.Build.props** - zprostředkovává globální nastavení atributů pro všechny projekty v solution
- **Directory.Build.targets** - obsahuje globální nastavení cílů kompilace pro všechny projekty v solution
- **NuGet.config** - nastavení pro balíčkovací správce NuGet

Dalším příkladem projektového souboru je *.fsproj* pro F# projekty, *.vbproj* pro Visual Basic projekty a *.nuspec* pro balíčkovací soubory NuGet. Speciálně pro IDE Visual Studio je využíván soubor *.dcsproj*, který obsahuje nastavení pro spustění a debuggování aplikace spuštěné v Docker kontejneru.

1.6 Kompilace zdrojového kódu

Kompilace je proces transformace zdrojového kódu do jiné podoby. Kód je zpravidla kompilován do podoby bližší cílové architektuře, ať je touto architekturou OS, případně konkrétní HW, nebo runtime prostředí (virtuální stroj). V rámci platformy .NET jsou k dispozici dva hlavní fundamentálně odlišné režimy kompilace zdrojového kódu: kompilace pro běhové prostředí (CLR) do tzv. *assembly* a kompilace do nativního kódu přímo pro cílovou architekturu (Native AOT).

1.6.1 Cíle kompilace

Cílem kompilace je převést zdrojový kód do podoby, kterou je možné spustit na cílovém zařízení. Platforma .NET podporuje zacílit na několik cílových zařízení, jako jsou desktopové počítače, mobilní zařízení nebo cloudové služby na základě použitých nástrojů a frameworků. Mezi podporované cíle patří:

- **Desktopové počítače** - Zacílení na platformy PC probíhá několika způsoby. Aplikace obvykle běží na CLR, kde je kód kompilován do jazyka IL a poté spouštěn prostřednictvím .NET runtime, přičemž je za běhu převeden na nativní kód. Pro situace, kdy .NET runtime není nebo nemůže být přítomen lze využít kompilace AOT pro vygenerování nativního kódu.
- **Mobilní zařízení** - Pro mobilní vývoj poskytuje .NET MAUI (nebo také Xamarin), sadu nástrojů, které umožňují vývojářům psát nativní aplikace pro Android, iOS a Windows s jedinou sdílenou kódovou základnou .NET. Xamarin se integruje do sady Visual Studio a umožňuje vývojářům používat knihovny C# a .NET k vytváření a spouštění mobilních aplikací. Tyto aplikace jsou kompilovány specificky pro každou platformu a mohou využívat nativní funkce zařízení.

1.6.2 Obecné postupy kompilace

Proces kompilace zahrnuje několik charakteristických postupů, které jsou přizpůsobeny optimalizaci výkonu aplikací při vývoji i za běhu. Jejich cílem je zvýšení výkonu a zabezpečení aplikací, ale také zajištění větší kompatibility a efektivity programu napříč platformami. Mezi tyto postupy kompilace patří:

- **Linkování** - Kompilátor během procesu sestavení aplikace propojuje množství zdrojových souborů a dll, aby vytvořily jeden spustitelný soubor nebo knihovnu. To zahrnuje řešení odkazů mezi různými dll a integraci všech požadovaných zdrojů.
- **Optimalizace** - Různé optimalizace IL a nativního kódu pro zlepšení výkonu. Tyto optimalizace probíhají jak během kompilace IL, tak během kompilace JIT nebo AOT do nativního kódu.
- **Tree shaking a trimming** - V moderních aplikacích .NET odstraňují nástroje, jako je IL Linker, během procesu sestavování nepoužívaný kód z konečné sestavy. Toto "protřepávání stromu"snižuje velikost aplikace tím, že vylučuje nepotřebné knihovny a cesty ke kódu.
- **Vytváření metadat a manifestů** - Překladače .NET také vytvářejí metadata a soubory manifestů, které popisují obsah a závislosti sestav, což má zásadní význam pro verzování, zabezpečení a rozlišení sestav za běhu.

1.6.3 Kompilace pro CLR

Standartním výstupem sestavení aplikace v .NETu je transformace zdrojového kódu z vybraného podporovaného jazyka do assembly v jazyce IL. Tento výstupní IL se v .NET konkrétně navývaná Common Intermediate language (CIL) nebo také Microsoft Intermediate Language (MSIL). IL je jazyk nezávislý na platformě, který je následně kompilován do nativního kódu za běhu aplikace.

CLR je zodpovědný za interpretaci IL kódu a jeho kompilaci do nativního kódu. Kompilace do nativního kódu je prováděna v rámci běhu aplikace, což zajišťuje, že kód je optimalizován pro konkrétní architekturu systému. V případě jazyka C# na platformě Windows slouží ke kompilaci spustitelný soubor *csc.exe*.

Výstupem kompilace pro CLR je assembly s popisnými metadaty a IL (v případě režimu R2R i částečně nativním) kódem. Assembly typicky disponují příponou *.dll*, případně jsou zabaleny do spustilného souboru dle cílové platformy a výstupu. Takovýto výstup je následně připraven buďto ke spuštění za pomocí CLR, případně pro využití a referenci při tvorbě dalšího .NET kódu. Kód IL je sada instrukcí nezávislá na procesoru, kterou může spustit běhové prostředí .NET (CLR).

Speciálním případem JIT kompilace je aplikace R2R. Zdrojový kód je při sestavení zkompilován do podoby nativního kódu pomocí nástroje crossgen, čímž vzniknou sestavy R2R. Za běhu se sestavy R2R načtou a spustí s minimální kompilací JIT, protože většina kódu je již v nativní podobě. CLR může přesto JIT kompilovat některé části kódu, které nelze staticky zkompilovat předem. Využití je v aplikacích, které potřebují

zkrátit dobu spouštění, ale zachovat určitou funkcionalitu nebo úroveň optimalizace poskytovanou JIT kompilací.

1.6.4 Kompilace do nativního kódu

Přímá nativní AOT kompilace je proces, při kterém je kód kompilován do podoby sytémově nativního kódu při sestavení programu ze zdrojového kódu. V případě .NETu je tato funkcionalita dostupná při použití jazyka C# a speciálních projektových atributů.

Jedná se o funkcionalitu, jenž prošla několika iteracemi. První možnosti sestavení aplikace v nativním kódu na .NET platformě byly aplikace Universal Windows Platform. Jednalo se o aplikace využívající specifické rozhraní, nativní pro produkty Microsoftu. S verzí .NET framework 7 byly rozšířeny možnosti sestavení aplikace jako do podoby nativního kódu i pro další architektury a typy aplikací. Tato nová funkcionalita získala vyráznější podporu v roce 2023 s vydáním dotent framework 8. Filozofie Microsoftu ohledně AOT kompilace je, že vývojáři by měli mít možnost využít AOT kompilace v .NETu, pokud je to pro daný scénář vhodné. Scénáře kladoucí takovéto požadavky se vyskytují především v cloudovém nasazení, na které v současné filosofii apelují.

Výstupem nativní AOT kompilace .NET aplikace je spustitelný soubor ve formátu podporovaném operačním systémem konfigurovaným v procesu kompilace. Takto vytvořený soubor je možné spustit přímo bez potřeby CLR.

1.7 Běh kódu

Spuštění, respektive běh kódu na HW počítačového zařízení vyžaduje instrukční sadu, které daná architektura rozumí, tedy nativní kód. V případě nativní AOT kompilace v .NET tento kód získáme již při sestavení aplikace. Při využití kompilace do IL je nutné kód získat pomocí jednoho z kompilačních způsobů podporovaného CLR. Výsledná nativní reprezentace se v obou případech spouští zavoláním vstupní metody v binárním souboru dle specifikace architektury.

1.7.1 CLR

Common Language Runtime (CLR) je běhové prostředí frameworku .NET. Poskytuje spravované prostředí pro spouštění aplikací .NET. Podporuje více programovacích jazyků, včetně jazyků C#, VB.NET a F#, a umožňuje jejich bezproblémovou spolupráci. Spravuje paměť prostřednictvím automatického garbage collection, který pomáhá předcházet únikům paměti a optimalizuje využití prostředků. CLR také zajišťuje typovou bezpečnost a ověřuje, zda jsou všechny operace typově bezpečné, aby se minimalizovaly chyby při programování.

CLR je zodpovědný za několik důležitých funkcí, které zvyšují produktivitu vývojářů a výkon aplikací.

- **Správa paměti** - CLR spravuje alokaci a dealokalizaci paměti, čímž zajišťuje efektivní využití systémových prostředků a zabraňuje memory leaks (únikům paměti). Obsahuje garbage collector (GC), který automaticky přiděluje a sbírá paměť obsazenou objekty.
- **Bezpečnost** - CLR poskytuje komplexní model zabezpečení, který pomáhá chránit aplikace před neoprávněným přístupem, poškozením dat a dalšími bezpečnostními hrozbami. Vynucuje zásady zabezpečení, jako je zabezpečení přístupu ke kódu (CAS) a zabezpečení založené na rolích. Zajišťuje, aby kód byl spouštěn s příslušnými oprávněními na základě svého původu a úrovně důvěryhodnosti, čímž chrání citlivá data a systémové prostředky.
- **Zpracování vyjímek** - Zpracování výjimek v CLR zahrnuje detekci, propagaci a zpracování chyb a stavů, které mohou nastat během provádění programu. Mechanismus vyjímek umožňuje elegantně řešit neočekávané situace a zachovat stabilitu aplikace.
- **Generování typů** - CLR podporuje dynamické generování typů za běhu, což aplikacím umožňuje za běhu dynamicky vytvářet a manipulovat typy dle potřeby. To dává možnost scénářům, jako je dynamické generování kódu, kompilace kódu za běhu a dynamické vytváření objektů.
- **Reflexe** - Reflexe umožňuje validaci a manipulaci typů, attributů a metadat načtených z dll za běhu. Umožňuje vývojářům dotazovat se a upravovat typy a jejich attributy za běhu, dynamicky volat metody a přistupovat k informacím o metadatech. Díky tomu mají aplikace .NET využívající běhové prostředí výrazné možnosti introspekce a přizpůsobení.

Klíčovými vlasnostmi jsou CLR jsou multiplatformnost kódu, reflexe, optimalizace kódu pro konkrétní architekturu a bezpečnost. CLR nabízí mechanismy, jako je zabezpečení přístupu ke kódu (CAS), které zabraňují neoprávněným operacím. Kompilace JIT (just-in-time) znamená, že kód zprostředkujícího jazyka je zkompilován do nativního kódu těsně před spuštěním, což zajišťuje optimální výkon na cílovém hardwaru. CLR usnadňuje zpracování chyb v různých jazycích a poskytuje konzistentní přístup k řešení výjimek. Navíc obsahuje nástroje pro ladění a profilování, které vývojářům pomáhají efektivně identifikovat a odstraňovat problémy s výkonem.

Aby mohl být kód z IL reprezentace spuštěn na systému, respektive HW stroje, musí být dodatečně kompilován. Za tímto účelem existuje v CLR několik technik, které s sebou přináší různé benefity a negativa a mají využití v specifických scénářích.

1.7.2 Nativní kód

Běh nativního kódu je závislý na konkrétní architektuře systému, pro které jsou nativní programové soubory vytvořeny. Nepodléhá další úpravě ze strany .NET nástrojů. Spuštění probíhá nativním příkazem operačního systému, který zprostředkuje spuštění programu.

1.8 Tvorba programu v .NET

Následující část popisuje obecnou koncepci a strukturu projektu aplikace v .NET. Součástí je postup pro tvorbu a vydání projektu. Blížší pozornost bude věnována tvorbě nativního AOT projektu.

1.8.1 Struktura aplikačních zdrojů

Základním strukturovaným prvkem v .NET aplikaci je projektový soubor. Jedná se o XML soubor disponující příponou *.csproj*. V rámci něj dochází ke konfiguraci a deklaraci, jak bude .NET CLI s projektem pracovat. Zároveň jsou zde definováný závislosti na další projekty a knihovny. Mezi základní charakteristiky běžně určené v projektovém souboru patří verze .NET, verze projektu/assembly, seznam závislostí, konfigurace pro buildování, testování a publikaci.

Pro tvorbu složitějších aplikací je možné využít více projektových souborů, které jsou následně propojeny. Tento způsob je využíván především v případě větších aplikací, které jsou rozděleny do více částí. Propojení a vazby mezi více projekty v aplikaci je definované pomocí tzv. solution souboru. Jedná se o kontejnerový soubor s příponou *.sln*, jenž popisuje závislosti mezi projektovými soubory, konfigurace sestavení a nasazení a správu pomocných souborů.

1.8.2 Obecný postup

1. **Nastavení vývojového prostředí**: Sestává z instalace sady nástrojů .NET SDK.

2. **Vytvoření projektu**: Pomocí příkazu `dotnet new` nebo skrze GUI IDE je vytvořen nový projekt a solution soubor. Součástí je výběr typu projektu, jazyka, frameworku a dalších konfiguračních parametrů.

3. **Programování**: Sestává z tvorby kódu aplikace, testování a ladění.

4. **Správa závislostí**: Pomocí nástrojů .NET CLI je možno referencovat balíčky a knihovny v rámci projektu.

5. **Kompilace**: Kompilace aplikace probíhá pomocí příkazu `dotnet build`, který převede vysokoúrovňový kód do IL. V případě AOT dochází k dodatečné kompilace do nativního kódu dle cílové architektury.

6. **Publikování**: Použitím příkazu příkazu `dotnet publish` dochází k vydání aplikace, tedy specifickému sestavení v konfigurovaném nastavení.

1.8.3 Tvorba nativního programu

Pro tvorbu nativního programu v .NET je nutné využít speciálního atributu *PublishAoT* v projektovém souboru. Tento atribut je zodpovědný za konfiguraci projektu pro nativní AOT kompilaci. Při jeho použití je nutné specifikovat cílovou architekturu, pro kterou je nativní kód vytvářen. Po kompilaci kódu do IL dochází k dodatečné kompilaci do nativního kódu, která dodává další konzolový výstup s informacemi o průběhu kompilace.

Vzhledem k tomu, že nativní AOT kompilace je v .NETu stále vývojově nezralá, samotný proces kompilace, tak jako analýza kompilovaného kódu není dostatečně informativní. Za účelem přenesení vysokoúrovňových konceptů a formálních zápisů v C# je při kompilace prováděno široké spektrum transformací a gerování kódu.

- **EmitCompilerGeneratedFiles** - Pokud je v projektu nastaven atribut EmitCompilerGeneratedFiles na pravdivou hodnotu, kompilátor generuje soubory, které obsahují podrobné informace o stavu zkompilované aplikace. Ty zahrnují i meziprodukty nebo výpisy strojového kódu, které jsou cenné pro procest ladění a analýzy výstupního produktu. Pomáhají při zacílení na kompilaci do nativního AOT lépe pochopit, jak je vysokoúrovňový kód překládán do strojového kódu.

- **Deklarace unmanaged rozhraní** - Deklarace unmanaged nebo také nespravovaného rozhraní zahrnuje definování způsobů, jakými spravovaný kód spolupracuje s nespravovanými nativními knihovnami nebo systémy. To je zásadní v nativním AOT, protože celá aplikace je kompilována do nativního kódu před spuštěním, takže nezbývá žádný prostor pro úpravy za běhu, které se obvykle provádějí při kompilaci Just-In-Time (JIT). Tyto deklarace specifikují způsob volání unmanaged funkcí, což jsou funkce operující mimo .NET aplikaci, například v C/C++ knihovně. U nativních AOT aplikací je důležité, aby tato rozhraní byla přesně definována a dodržována, protože jakýkoli nesoulad nebo chyba v deklaraci může vést k chybám za běhu, které se hůře diagnostikují a opravují kvůli absenci runtime prostředí a dynamickým funkcím.

- **Trimming** - Proces odstraňování nepotřebného kódu a zdrojů z aplikace během sestavení, za účelem snížení velikosti a zvýšení výkonu. V rámci platformy .NET představuje trimming jednu z technik postupu "tree shaking", kdy překladač analyzuje, které části zdrojového kódu jsou skutečně používány, a zbytek vyloučí z výstupního produktu. To je zvláště důležité pro aplikace, kde je úložiště nebo paměť omezená. Ořezávání pomáhá při optimalizaci aplikací a zajišťuje, aby běhové prostředí obsahovalo pouze nezbytné části nutné pro fungování aplikace.

1.8.4 Přehled podpory

Následující přehled představuje rozsah funkcionality implementované v rámci .NET frameworku k datu vydání verze 8.0.0.

- **REST minimal API**
- **gRPC API**
- **JWT Authentication**
- **CORS**
- **HealthChecks**
- **HttpLogging**
- **Localization**
- **OutputCaching**
- **RateLimiting**
- **RequestDecompression**
- **ResponseCaching**
- **ResponseCompression**
- **Rewrite**
- **StaticFiles**
- **WebSockets**

2 MICROSERVICE ARCHITEKTURA

Při vývoji softwaru je možné aplikovat různé architektury a návrhové vzory. Mezi obecně populární a rozšířenou architekturu patří monolitická. V monolitické architektuře je celá aplikace rozdělena do několika vrstev, které jsou využívány k oddělení logiky. Tato architektura je jednoduchá na vývoj a nasazení, ale může být obtížné škálovat a udržovat s rostoucí složitostí aplikace. Monolitická architektura je založena na principu, že celá aplikace je spuštěna jako jediný proces a sdílí stejný kód a zdroje. To může způsobit problémy s výkonem, škálovatelností a odolností aplikace.

Principiálně opačná je architektura microservice. Ta je založena na principu oddělení aplikace do několika samostatných služeb. Každá z těchto služeb je zodpovědná za určitou část funkcionality aplikace. Služby jsou navzájem nezávislé a komunikují mezi sebou pomocí definovaných rozhraní. Tím je zajištěno, že každá služba může být vyvíjena, nasazována a škálována nezávisle na ostatních. Tato architektura umožňuje vývojářům pracovat na menších a jednodušších částech aplikace, což zvyšuje produktivitu a umožňuje rychlejší iterace. Díky nezávislosti služeb je také možné dosáhnout vyšší odolnosti a škálovatelnosti aplikace. [6]

2.1 Historie

Původ microservice architektury nelze přesně definovat, důležitý moment však nastal v roce 2011, kdy Martin Fowler publikoval článek *Microservices* na svém blogu. V tomto článku popsal výhody a nevýhody této architektury a zároveň popsal způsob, jakým je možné tuto architekturu využít. Dalším popularizačním momentem pro popularizaci bylo vydání knihy *Building Microservices* od Sama Newmana v roce 2015. Tato kniha popisuje způsob, jakým je možné využít microservice architekturu v praxi.

Opravdový přelom přišel postupně, nástupem a popularizací virtualizace a kontejnerizace v průběhu let 2013 až 2015. Tímto bylo umožněno vytvářet a spouštět mikroslužby v izolovaných prostředích. Tímto bylo umožněno vytvářet mikroslužby, které jsou nezávislé na operačním systému a hardwaru, na kterém jsou spouštěny. Nejdůležitější v tomto ohledu je nepochybně projekt Docker, který byl vydán v roce 2013. Díky Dockeru bylo možno jednoduše definovat, vytvářet a spouštět kontejnerizované aplikace.

2.2 Základní principy

V oblasti architektury mikroslužeb existuje několik základních principů, které tento přístup odlišují od tradičnějších softwarových architektur. Tyto principy nejsou jen teoretické, mají přímý dopad na to, jak jsou služby vyvíjeny, nasazovány a udržovány.

Jejich využití přispívá k tvorbě vysoce flexibilní, škálovatelné a odolné architektuře.

2.2.1 Decentralizace

Jedním z hlavních principů microservice architektury je decentralizace. To znamená, že každá mikroslužba je zodpovědná za určitou část funkcionality aplikace. Mikroslužby jsou navzájem nezávislé a komunikují mezi sebou pomocí definovaných rozhraní. Tím je zajištěno, že každá mikroslužba může být vyvíjena, nasazována a škálována nezávisle na ostatních.

2.2.2 Virtualizace a kontejnerizace

Virtualizace a kontejnerizace jsou klíčové technologie, které umožňují architekturu mikroslužeb. Virtualizace umožňuje provozovat více operačních systémů na jednom fyzickém hardwarovém hostiteli, čímž se snižuje počet potřebných fyzických strojů a zvyšuje efektivita využití zdrojů. Kontejnerizace jde ještě o krok dále tím, že zabalí aplikaci a její závislosti do kontejneru, který může běžet na libovolném serveru Linux nebo Windows. Tím je zajištěno, že aplikace funguje jednotně i přes rozdíly v prostředí nasazení.

Kontejnerizace je obzvláště důležitá pro mikroslužby, protože zapouzdřuje každou mikroslužbu do vlastního kontejneru, což usnadňuje její nasazení, škálování a správu nezávisle na ostatních. Synonymem kontejnerizace je nástroj Docker, který nabízí ekosystém pro vývoj, správu a provoz kontejnerových aplikací.

2.2.3 Orchestrace

S rozšiřováním mikroslužeb a kontejnerů se jejich správa stává složitou. Nástroje pro orchestraci pomáhají automatizovat nasazení, škálování a správu kontejnerů. Mezi oblíbené orchestrační nástroje patří Kubernetes, Docker Swarm a Mesos. Zejména Kubernetes se stal de facto standardem, který poskytuje robustní rámec pro nasazení, škálování a provoz kontejnerových aplikací v clusteru strojů. Řeší vyhledávání služeb, vyvažování zátěže, sledování přidělování prostředků a škálování na základě výkonu pracovní zátěže.

2.2.4 Odolnost

Robustnosti mikroslužeb je dosaženo pomocí strategií, jako jsou přerušení, záložní řešení a opakování, které pomáhají zabránit tomu, aby se selhání jedné služby kaskádově přeneslo na ostatní. Izolace služeb také znamená, že problémy lze omezit a vyřešit s minimálním dopadem na celý systém. Kromě toho jsou kontroly stavu a monitorování nezbytné pro včasné odhalení a řešení problémů.

2.2.5 Škálování

Architektura mikroslužeb zvyšuje škálovatelnost. Služby lze škálovat nezávisle, což umožňuje efektivnější využití zdrojů a zlepšuje schopnost systému zvládat velké objemy požadavků. Běžně se používá horizontální škálování (přidávání dalších instancí služby), které usnadňují nástroje pro kontejnerizaci a orchestraci.

2.3 Komponenty

Architektura mikroslužeb rozkládá aplikace do menších, oddělených služeb, z nichž každá plní samostatnou funkci. Pro efektivní správu těchto služeb, zejména v distribuovaném prostředí, se používá několik základních komponent. Tato část se zabývá klíčovými architektonickými komponentami, které usnadňují robustní provoz, komunikaci a škálovatelnost mikroslužeb.

2.3.1 API Gateway

API Gateway je služba, která slouží jako vstupní bod pro komunikaci s mikroslužbami. Zajišťuje směrování požadavků, autentizaci, autorizaci, zabezpečení a další funkce, které jsou společné pro všechny služby. API Gateway může také poskytovat další funkce, jako jsou cachování, transformace zpráv a řízení toku dat. Tím zjednodušuje a centralizuje správu komunikace mezi klienty a mikroslužbami.

2.3.2 Service Discovery

Service Discovery je mechanismus, který umožňuje mikroslužbám dynamicky najít a komunikovat s ostatními službami v systému. To je důležité pro dynamické škálování, nasazování a správu služeb. Service Discovery může být implementován pomocí centrálního registru služeb nebo distribuovaného protokolu, jako je DNS nebo Consul.

2.3.3 Load Balancer

Load Balancer je služba, která rozděluje provoz mezi několik instancí služby, aby se zajistila rovnoměrná zátěž a zvýšila odolnost proti chybám. Load Balancer může být implementován jako hardwareové zařízení nebo softwarová služba, která poskytuje rozhraní pro konfiguraci a správu zátěže.

2.3.4 Komunikační systémy

Mikroslužby spolu komunikují prostřednictvím rozhraní API, obvykle prostřednictvím protokolů HTTP/HTTPS, i když pro aplikace citlivější na výkon lze použít i jiné protokoly, například gRPC. Komunikační vzory zahrnují synchronní požadavky (např.

RESTful API) a asynchronní zasílání zpráv (např. pomocí brokerů zpráv jako RabbitMQ nebo Kafka). Tím je zajištěno volné propojení mezi službami, což umožňuje jejich nezávislý vývoj a nasazení.

- **REST API** - Představuje vysoce rozšířenou možnost komunikace mezi mikroslužbami. Využívají se při ní standardní metody HTTP, jako jsou GET, POST, PUT a DELETE, k provádění operací na rozhraní identifikovaným prostřednictvím adresy URL. Díky bezstavové povaze je rozhraní REST vysoce škálovatelné a vhodné pro veřejně přístupné služby. Má širokou podporu na různých platformách a v různých jazycích, což pomáhá zajistit interoperabilitu v rozmanitém ekosystému mikroslužeb.

- **RPC** - Komunikační metoda používaná v distribuovaných systémech, včetně mikroslužeb, kdy program způsobí, že se procedura spustí v jiném adresním prostoru (obvykle na jiné virtualizované ve sdílené síti). Tato technika abstrahuje složitost síťové komunikace do jednoduchosti volání lokální funkce nebo metody. Mezi běžné implementace RPC patří gRPC, Thrift a Apache Avro.

- **Message Broker** - Jedná se o komunikační vzor kdy broker - prostředník, spravuje asynchronní komunikaci mezi mikroslužbami pomocí front zpráv. Tato metoda odděluje mikroslužby tím, že jim umožňuje publikovat zprávy do fronty, aniž by znaly podrobnosti o tom, které služby je budou spotřebovávat. Mezi běžné zprostředkovatele zpráv patří RabbitMQ, Apache Kafka a AWS SQS. Tato komunikační architektura zvyšuje odolnost proti chybám, protože zprostředkovatel zpráv může zajistit, že zprávy nebudou ztraceny při přenosu, i když je spotřebitelská služba dočasně nedostupná.

2.3.5 Databáze

V architekturách mikroslužeb si každá služba obvykle spravuje vlastní databázi, což je přístup, který podtrhuje princip decentralizované správy dat. Tato izolace pomáhá službám být volně provázané a nezávisle nasaditelné, přičemž každé databázové schéma je přizpůsobeno konkrétním potřebám služby. V závislosti na případu použití mohou služby používat různé typy databází - SQL pro transakční data vyžadující silnou konzistenci a vlastnosti ACID nebo NoSQL pro flexibilnější možnosti ukládání dat, které umožňují horizontální škálování a podporu velkých objemů strukturovaných, částečně strukturovaných nebo nestrukturovaných dat. Tato různorodost databázových technologií přináší výzvy, jako je udržování konzistence dat napříč službami, což se často řeší pomocí strategií, jako je vzor Saga nebo případná konzistence pro zajištění integrity dat napříč distribuovanými transakcemi.

2.3.6 Bezpečnost

Bezpečnost v architektuře mikroslužeb je velmi důležitá, protože distribuovaná povaha těchto systémů přináší mnoho zranitelných míst. Bezpečnostní prvky se zaměřují na ochranu dat při přenosu a v klidovém stavu a zajišťují, že ke službám a datům mají přístup pouze oprávněné subjekty. Mezi klíčové strategie patří implementace bran API s vestavěnými bezpečnostními prvky, jako je ověřování, autorizace a ukončení SSL. Zásadní význam mají systémy správy identit a přístupu (IAM), často integrované s tokeny OAuth a JWT pro správu identit uživatelů a řízení přístupu na základě definovaných zásad. Zajištění šifrované komunikace mezi službami pomocí protokolů, jako je TLS, může navíc chránit před odposlechem a manipulací. Zásadní jsou také účinné strategie protokolování a monitorování, které poskytují možnost odhalovat bezpečnostní hrozby, reagovat na ně a zmírňovat je v reálném čase. Každá z těchto složek hraje klíčovou roli při vytváření bezpečného ekosystému mikroslužeb a umožňuje robustní obranné mechanismy proti interním i externím bezpečnostním rizikům.

2.4 Vývoj, testování a nasazení

Mikroslužby umožňují agilní vývojové postupy. Týmy mohou vyvíjet, testovat a nasazovat služby nezávisle, což umožňuje rychlejší iteraci a zpětnou vazbu. Nedílnou součástí jsou pipelines pro kontinuální integraci a doručování (CI/CD), které umožňují automatizované testování a nasazení. Tento přístup podporuje kulturu DevOps a podporuje užší spolupráci mezi vývojovými a provozními týmy.

Testování mikroslužeb je klíčové pro zajištění kvality a spolehlivosti systému. Mikroslužby lze testovat na několika úrovních, včetně jednotkových testů, integračních testů a testů end-to-end. Jednotkové testy se zaměřují na testování jednotlivých komponent služby, zatímco integrační testy testují komunikaci mezi službami. Testy end-to-end testují celý systém z pohledu uživatele. Automatizované testování je klíčové pro rychlé a spolehlivé nasazení.

Integrace kontinuální integrace a kontinuálního nasazování (CI/CD) je zásadní pro správu mikroslužeb, jelikož podporuje rychlé iterace a minimalizuje rizika spojená s nasazováním změn. CI/CD automatizuje procesy sestavení, testování a nasazení softwaru, což zajišťuje, že nový kód prochází důkladným testováním a je pravidelně a bezpečně nasazován do produkce. V prostředí mikroslužeb umožňuje CI/CD týmům aktualizovat služby nezávisle na ostatních částech systému, což vede k vyšší agilitě a rychlejší reakci na požadavky trhu nebo na potřeby zákazníků.

2.5 Výhody a nevýhody

Mezi hlavní výhody microservice architekture lze zařadit:

- **Přizpůsobitelnost** - Mikroslužby umožňují rychlé, časté a spolehlivé poskytování rozsáhlých a komplexních aplikací. Týmy mohou aktualizovat určité oblasti aplikace, aniž by to mělo dopad na celý systém, což umožňuje rychlejší iterace.
- **Škálovatelnost** - Služby lze škálovat nezávisle, což umožňuje přesnější přidělování zdrojů na základě poptávky. To usnadňuje zvládání proměnlivého zatížení a může zlepšit celkovou efektivitu aplikace.
- **Odolnost** - Decentralizovaná povaha mikroslužeb pomáhá izolovat selhání na jedinou službu nebo malou skupinu služeb, čímž zabraňuje selhání celé aplikace. Techniky, jako jsou jističe, zvyšují odolnost systému.
- **Technologická rozmanitost** - Týmy si mohou vybrat nejlepší nástroj pro danou práci a podle potřeby používat různé programovací jazyky, databáze nebo jiné nástroje pro různé služby, což vede k potenciálně optimalizovanějším řešením.
- **Flexibilita nasazení** - Mikroslužby lze nasazovat nezávisle, což je ideální pro kontinuální nasazení a integrační pracovní postupy. To také umožňuje průběžné aktualizace, modrozelené nasazení a kanárkové verze, což snižuje prostoje a rizika.
- **Modularita** - Tato architektura zvyšuje modularitu, což usnadňuje pochopení, vývoj, testování a údržbu aplikací. Týmy se mohou zaměřit na konkrétní obchodní funkce, což zvyšuje produktivitu a kvalitu.

Zatímco mezi nevýhody patří:

- **Komplexnost** - Správa více služeb na rozdíl od monolitické aplikace přináší složitost při nasazování, monitorování a řízení komunikace mezi službami.
- **Správa dat** - Konzistence dat mezi službami může být náročná, zejména pokud si každá mikroslužba spravuje vlastní databázi. Implementace transakcí napříč rozhraními vyžaduje pečlivou koordinaci.
- **Zpoždění sítě** - Komunikace mezi službami po síti přináší zpoždění, které může ovlivnit výkonnost aplikace. Ke zmírnění tohoto jevu jsou nutné efektivní komunikační protokoly a vzory.
- **Provozní režie** - S počtem služeb roste potřeba orchestrace, monitorování, protokolování a dalších provozních záležitostí. To vyžaduje další nástroje a odborné znalosti.
- **Složitost vývoje a testování** - Mikroslužby sice zvyšují flexibilitu vývoje, ale také komplikují testování, zejména pokud jde o testování end-to-end, které zahrnuje více služeb.

- **Integrace služeb** - Zajištění bezproblémové spolupráce služeb vyžaduje robustní správu API, řízení verzí a strategie zpětné kompatibility.

2.6 Nasazení založené na mikroslužbách

Efektivní nasazení mikroslužeb je klíčové pro využití jejich potenciálních výhod, jako je škálovatelnost, flexibilita a odolnost. Tato část se zabývá různými strategiemi nasazení, které jsou pro mikroslužby obzvláště vhodné, zejména v prostředí cloud-native. Tyto strategie zajišťují, že mikroslužby lze efektivně spravovat a škálovat, dynamicky reagovat na změny zatížení a minimalizovat prostoje.

2.6.1 Strategie

Existuje několik strategií nasazení, které jsou pro mikroslužby obzvláště vhodné:

- **Jedna služba na hostitele** - Strategie zahrnuje nasazení každé mikroslužby na vlastní server, ať už virtuální, nebo fyzický. Tento přístup zjednodušuje ladění a izolaci služeb, ale může vést k nedostatečnému využití zdrojů a vyšším nákladům.
- **Více služeb na jednoho hostitele** - Nasazení více služeb na jednom hostiteli maximalizuje využití zdrojů a snižuje náklady. Vyžaduje však pečlivou správu, aby nedocházelo ke konfliktům a aby se služby vzájemně nerušily.
- **Instance služby na kontejner** - Moderní nasazení mikroslužeb často používají kontejnery (například Docker) pro umístění jednotlivých služeb. Kontejnery poskytují odlehčené, konzistentní prostředí pro každou službu, zjednodušují nasazení a škálování v různých prostředích a zajišťují, že každá služba má splněny své závislosti bez konfliktů.

2.6.2 Cloud native nasazení

Mikroslužby jsou obzvláště vhodné pro cloudová nativní prostředí, která podporují jejich dynamickou povahu:

- **Kontejnery a orchestrace** - Nástroje jako Kubernetes orchestrují kontejnerové služby a řídí jejich životní cyklus od nasazení až po ukončení. Kubernetes se stará o škálování, vyrovnávání zátěže a obnovu, což usnadňuje vysokou dostupnost a efektivní využívání zdrojů.
- **Mikroslužby na platformě jako služba (PaaS)** - Nabídky PaaS, jako jsou AWS Elastic Beanstalk, Microsoft Azure App Service a Google App Engine, poskytují prostředí, kde lze mikroslužby snadno nasadit, škálovat a spravovat bez nutnosti starat se o základní infrastrukturu.

- **Serverless** - Bezserverové výpočetní modely umožňují nasazení mikroslužeb jako funkcí (FaaS), které se spouštějí v reakci na události, přičemž poskytovatel cloudu spravuje prostředí pro jejich provádění. Tento model je vysoce škálovatelný a nákladově efektivní, protože zdroje jsou spotřebovávány pouze během provádění funkcí.

3 MONITOROVÁNÍ APLIKACE

Monitorování aplikací je klíčovým aspektem moderního vývoje a provozu softwaru, který umožňuje sledovat výkon, stav a celkové chování aplikací v reálném čase. Zahrnuje shromažďování, analýzu a interpretaci různých typů dat a informací, které zajišťují hladký a efektivní chod aplikací a umožňují rychle identifikovat a řešit případné problémy.

3.1 Cíle monitorování

Cílem monitorování v kontextu mikroslužeb je poskytnout využitelné informace v několika klíčových oblastech:

- **Výkonnost systému** - Monitorování se snaží zachytit kritické výkonnostní metriky, jako je latence, propustnost a chybovost. Tyto metriky pomáhají pochopit, jak dobře služby fungují za normálních podmínek a při zátěži.
- **Využití zdrojů** - Je důležité sledovat využití systémových prostředků včetně procesoru, paměti a diskových I/O. Poznatky o využití zdrojů pomáhají při optimalizaci výkonu aplikací a při plánování škálovacích operací.
- **Stav a dostupnost služeb** - Sledování stavu a dostupnosti jednotlivých mikroslužeb zajišťuje, že lze rychle identifikovat a odstranit případné problémy, aby byla zachována integrita a spolehlivost systému.
- **Dopad na vývoj a nasazení** - Ačkoli je monitorování spíše kvalitativní, může také poskytnout zpětnou vazbu o dopadu různých strategií nasazení a sestavení na výkon systému. To zahrnuje míru úspěšnosti nasazení, problémy vyplývající z nových nasazení a chování nových funkcí v ostrém prostředí.

3.2 Druhy dat

Monitorovací data hrají zásadní roli při údržbě a optimalizaci moderních softwarových systémů. Sběrem, analýzou a interpretací různých typů dat lze získat cenné informace o výkonu, stavu a celkovém chování aplikací. Následující část kategorizuje tři základní typy monitorovacích dat: metriky, logy a traces. Jednolitvé druhy slouží různým účelům a poskytují pohled na výkon a stav systému z určitěného úhlu.

3.2.1 Logy

Logy jsou chronologické záznamy o událostech, ke kterým dochází v rámci aplikace nebo využíváného runtime prostředí. Pomáhají určit hlavní příčiny konkrétních pro-

blémů nebo odhalit vzory svědčící o větších problémech. Generuje je jak operační systém tak i aplikace na něm běžící. Logy mohou nabývat různých struktur, nejjednodušší způsob je obyčejný textový řetězec, ale některé nástroje podporují strukturované logy, což jsou datové záznamy nabývající podoby typicky json nebo xml formátu. Obvykle obsahují mimo jiné data o systémových aktivitách, chybách, systémových zprávách, změnách konfigurace a síťových požadavcích. Analýzou logů mohou vývojáři a správci systému odstraňovat problémy, porozumět kontextu aplikace a zajistit soulad s očekávaným chováním. Nástroje pro správu logů poskytují funkce pro vyhledávání, filtrování a analýzu záznamů. Mezi populární nástroje správy patří Elasticsearch, Logstash nebo Loki.

3.2.2 Traces

Traces trasují cestu požadavku, při průchodu různými součástmi distribuovaného systému. Každá trace se skládá z jednoho nebo více segmentů, které zaznamenávají cestu a latenci požadavku napříč různými službami a zdroji. Sledování je zvláště důležité v architekturách mikroslužeb, kde jedna transakce může zahrnovat více, volně propojených služeb. Poskytuje přehled o výkonu a chování jednotlivých služeb a systému jako celku a pomáhá identifikovat bottleneck a problémy s latencí v komplexní funkcionalitě. Traces pomáhají pochopit vztahy a závislosti mezi službami, což umožňuje efektivnější ladění a optimalizaci. Mezi populární nástroje pro správu traces patří Jaeger, Zipkin anebo Tempo.

3.2.3 Metriky

Metriky jsou kvantitativní údaje, které poskytují přehled o výkonu a stavu aplikace v reálném čase. Tyto datové body jsou obvykle numerické a jsou shromažďovány v pravidelných intervalech. Běžnými programovými strukturami, které umožňují zaznamenání metrik jsou čítače a historgramy. Typické data představují doba odezvy, využití systémových prostředků (CPU, paměť, I/O, ...), chybovost a propustnost. Sledování těchto metrik pomáhá při proaktivním ladění výkonu a plánování kapacity. Mezi populárí nástroje pro sběr a vizualizaci metrik jsou Prometheus, Datadog nebo Splunk.

3.3 Sběr dat

Efektivita monitorování aplikací do značné míry závisí na schopnosti efektivně shromažďovat relevantní data z různých zdrojů a schopnosti zprostředkování těchto dat do monitorovacích nástrojů. Kolektory jsou nástroje nebo agenti, kteří shromažďují data z různých zdrojů v rámci aplikace a jejího prostředí. Mohou být nasazeny jako součást infrastruktury aplikace nebo mohou být provozovány jako externí služby. Kolektory

jsou zodpovědné za shromažďování protokolů, stop a metrik a za předávání těchto dat do monitorovacích řešení, kde je lze analyzovat a vizualizovat. Efektivní sběr dat je nezbytný pro monitorování v reálném čase a pro zajištění toho, aby shromážděná data přesně odrážela stav a výkon aplikace. Nejpopulárnější univerzální kolektor představuje nástroj OpenTelemetry.

3.4 Analýza a interpretace

V oblasti monitorování systému je sběr dat pouze prvním krokem. Skutečná hodnota spočívá v tom, jak jsou tato data analyzována a interpretována. Analýza a interpretace transformují nezpracovaná data na praktické poznatky, které organizacím umožňují porozumět nejen tomu, co se děje v jejich systémech, ale také tomu, proč k těmto událostem dochází. Tyto procesy jsou úzce propojeny. Vizualizace zpřístupňuje komplexní data a pomáhá zúčastněným stranám rozpoznat trendy a anomálie na první pohled. Mezitím pokročilé analytické techniky poskytují hlubší porozumění dat, odhalují základní vzorce a předpovídají budoucí trendy, které informují o strategickém rozhodování. Společně umožňují reagovat na aktuální stavy systému a proaktivně spravovat a optimalizovat budoucí výkon a robustnost.

3.4.1 Vizualizace dat

Vizualizace dat je klíčovým aspektem monitorování aplikací, který umožňuje rychle porozumět stavu a chování aplikací. Grafickým znázorněním složitých datových lze snadněji odhalit trendy a vzorce, které nemusí být patrné ze samotných nezpracovaných dat. Vizualizace mohou mít různé formáty:

- **Grafy** - Spojnicové grafy, sloupcové grafy a bodové grafy, které mohou zobrazovat změny v čase, distribuce a korelace.
- **Tabulky** - Prezentují nezpracovaná data zarovnaná do sloupců pro přímé srovnání.
- **Řídicí panely** - Integrují více vizualizací do jediného rozhraní a nabízejí holistický pohled na výkon a stav systému.
- **Heatmapy a Sankeyho diagramy** - Ilustrují složité vztahy a toky mezi komponentami systému.

Použitím a kombinací vizualizačních komponent vzniká unikátní pohled na dostupná telemetrická data. Tím je umožněno rychle identifikovat klíčová a kritická místa systému, jako jsou bottlenecks (úzká místa výkonu) a řešit potenciální problémy dříve, než ovlivní stabilitu systému nebo uživatelský dojem z aplikace.

3.4.2 Techniky analýzy dat

Analýza dat v kontextu monitorování systému zahrnuje více než jen vizuální interpretaci:

- **Statistická analýza** - Použití statistických technik k pochopení chování systému za různých podmínek. To může zahrnovat analýzu rozptylu, regresní modely pro předpovídání budoucích trendů a algoritmy detekce anomálií k odhalení neočekávaného chování.
- **Korelační analýza** - Určení vztahů mezi různými metrikami k identifikaci hlavních příčin problémů. Například korelování špiček využití CPU s konkrétními událostmi nebo operacemi aplikace.
- **Analýza logů** - Použití pokročilé textové analýzy a strojového učení k extrahování užitečných informací z nestrukturovaných protokolů, což pomáhá určit přesnou sekvenci událostí vedoucích k selhání nebo snížení výkonu.
- **Prediktivní analýza** - Používání prediktivních modelů k předpovídání budoucího chování systému na základě historických dat. To pomáhá při plánování kapacity a proaktivním řešení problémů.

3.4.3 Využití dat pro informované rozhodování

Konečným cílem analýzy a vizualizace dat je podpora informovaného rozhodování. Interpretovaná data poskytují užitečné poznatky, které mohou vést strategická rozhodnutí:

- **Přidělování zdrojů** - Úprava přidělování zdrojů na základě údajů o výkonu za účelem optimalizace nákladové efektivity a výkonu, jako je škálování zdrojů nahoru nebo dolů v reakci na očekávanou poptávku.
- **Optimalizace výkonu** - Identifikace a řešení překážek výkonu s cílem zlepšit odezvu aplikací a spokojenost uživatelů.
- **Vylepšení zabezpečení** - Rozpoznání vzorců indikujících bezpečnostní hrozby za účelem posílení obrany a zmírnění zranitelnosti.
- **Vylepšení služeb** - Používání dat o interakci uživatelů k vylepšení a vylepšení funkčnosti a rozhraní aplikací, což vede k lepším uživatelským zkušenostem.

3.5 Implementace monitorování

Implementace monitorování aplikací zahrnuje několik klíčových kroků, včetně definice klíčových metrik, výběru monitorovacích nástrojů, nasazení kolektorů a vizualizaci dat. Týmy by měly také vytvořit procesy pro řešení problémů, které byly identifikovány prostřednictvím monitorování, a pro využití dat k plánování kapacity a optimalizaci výkonu. Obecně implementace monitorování zahrnuje následující kroky:

1. **Sběr dat v monitorovaných službách** - Implementace sběru dat zahrnuje inkorporaci funkcionality monitorování a zprostředkování dat v rámci předdefinovaného rozhraní. Sběr je realizován zpravidla sérií čítačů a zapisovačů, které jsou využívány k získávání dat z různých zdrojů. Takto sbíraná datá jsou kategorizována a značkována pro identifikaci. Realizace monitorování je zajištěna buďto použitím existujících implementací v rámci sw knihoven nebo vytvořením vlastní implementace dle potřeb aplikace a monitorovacích protokolů.

2. **Nasazení služeb pro správu a kolekci dat** - Je zajištěno pomocí nasazení nástrojů, které jsou schopny zprostředkovat sběr a distribuci telemetrických dat z různých zdrojů. Zároveň mohou i zajišťovat jejich zpracování a zobrazení. Klíčový je výběr nástrojů s ohledem na zprostředkování adekvátního rozhraní pro kolekci dat a distribuci k následné vizualizaci. Splněním je dosaženo, že data jsou zpracována, uložena a dále zprostředkována v reálném čase.

3. **Vizualizace dat** - Vizualizace dat je implementována nasazením nástrojů, které jsou schopny zobrazit data z dostupných zdrojů v uživatelsky přívětivé podobě. Formát připojení na datové zdroje s monitorovacími daty je definován protokoly relevantním služeb. Vizualiace konkrétním dat je předmětem vytvoření vizualizačních prvků. Data jsou zobrazena v reálném čase a jsou přehledná a srozumitelná.

Konfigurace monitorování v rámci aplikace obecně zahrnuje zmapování interakcí mezi monitorovanými komponentami a monitorovacími nástroji. To zahrnuje určení, které metriky, logy a traces jsou relevantní na základě architektury aplikace a doménových požadavků. Konfigurace musí zajistit, že shromážděná data budou smysluplná a spravovatelná, a vyvarovat se nadměrné granularity, která může vést k přetížení systému. Obvykle tento proces zahrnuje nastavení agentů nebo integrací v rámci aplikace nebo služeb, které efektivně sbírají telemetrická data a přenáší je do centralizovaného monitorovacího systému, aniž by došlo k narušení výkonu aplikace. Komunikace mezi aplikačními komponentami a monitorovacími nástroji často využívá stávající síťové protokoly a metody bezpečného přenosu dat. Proces konfigurace může dále zahrnovat

nastavení hraničních hodnot pro výstrahy, definování pravidel retence dat a nastavení parametrů pro automatické reakce na určité typy událostí, což pomáhá udržovat celkový stav a výkon aplikace. Tento přístup zajišťuje, že monitorovací systém poskytuje užitečné informace, je v souladu s doménovými požadavky a poskytuje komplexní bázi telemetrických dat pro analýzu a optimalizaci systému.

II. PRAKTICKÁ ČÁST

4 TVORBA APLIKAČNÍHO STACKU

Za účelem důkladného testování výkonu a škálovatelnosti mikroslužeb byl vytvořen tech stack, který zahrnuje technologie pro kontejnerizaci, orchestraci, persistenci, komunikaci, monitorování a testování.

4.1 Požadavky na SW

Na aplikaci jsou pro splnění účelu analýzy vývoje, výstupu a výkonu služeb kladeny přímé i nepřímé požadavky. Je klíčové navrhnout řešení, vybrat technologie a provést implementaci a konfiguraci s ohledem na tyto požadavky, jenž rozděleny na funkční a nefunkční.

4.1.1 Funkční požadavky

Funkční požadavky definují chování, funkce a vlastnosti, které musí systém poskytovat. Přímo souvisejí s doménovými požadavky a zahrnují specifikace, jako je zpracování dat, provádění výpočtů nebo podpora konkrétních procesů. Funkční požadavky v podstatě popisují očekávané operace systému, včetně vstupů, chování a výstupů, a jsou tak klíčové pro vývoj a testování.

- **Mikroslužby** - Každá aplikace musí poskytovat rozhraní REST API s healtcheck enpointem pro informování celého systému o svém stavu. Dalším požadavkem je obecná komunikace mezi službami pomocí vybraných protokolů. Aplikace musí být schopna sbírat a vizualizovat data o výkonu a škálovatelnosti mikroslužeb. To zahrnuje sběr a vizualizaci metrik, logů a traces.
- **Stack** - Aplikační stack jakožto celek musí zahrnovat komunikaci pomocí protokolů HTTP/2 a gRPC. Je nutné aby implementoval publish - subscribe pattern pro komunikaci mezi vybranými službami. Stack musí zprostředkovat přístup a ukládání dat do relační a timeseries databáze. Musí poskytovat nutné rozhraní pro sběr, uchování a vizualizaci metrik a testovacích dat. Stack musí být schopen konfigurovat testovací scénáře, které se mají provést a také je spouštět v manuálním a automatizovaném režimu.
- **Sběr a vizualizace dat** - Aplikace musí být schopna sbírat a vizualizovat data o výkonu a škálovatelnosti mikroslužeb. To zahrnuje sběr a vizualizaci metrik, logů a traces. Data musí být dostupná v reálném čase a musí být možné je analyzovat a porovnávat. Zároveň je klíčové aby vizualizace byla konfigurovatelná a bylo umožněno vytvářet vlastní dashboardy pomocí komponent.

- **Testování scénářů** - Aplikace musí být schopna provádět testování scénářů, které simulují fungování systému a zátěž na mikroslužby. Testovací scénáře musí být konfigurovatelné a spustitelné v manuálním a automatizovaném režimu.
- **Konfigurace aplikace** - V rámci aplikace musí být možnost konfigurovat chování nasazených služeb. To se týká jak konfigurace komunikace mezi službami, tak i konfigurace monitorovacích nástrojů. Konfigurace musí být uložena v konfiguračních souborech ve standardním formátu dle konvencí dané služby.

4.1.2 Nefunkční požadavky

Nefunkční požadavky specifikují celkové vlastnosti systému. Definují atributy kvality, které musí systém splňovat. Nefunkční požadavky mohou zahrnovat omezení týkající se návrhu a implementace systému, jako jsou bezpečnostní standardy, soulad s právními a regulačními směrnicemi, doba odezvy při zpracování dat, kapacita pro souběžné uživatele, integrita dat a mechanismy převzetí služeb při selhání. Mají zásadní význam pro zajištění životaschopnosti a efektivity systému v jeho provozním prostředí a často ovlivňují celkovou uživatelskou zkušenost, výkonnost systému a splnění regulačních podmínek.

- **Použitelnost** - Aplikace musí být snadno použitelná a přístupná pro uživatele. To zahrnuje snadnou konfiguraci a spuštění testovacích scénářů.
- **Udržitelnost** - Aplikace musí být udržitelná a snadno rozšiřitelná. To zahrnuje schopnost přidávat nové služby a rozšiřovat stávající služby.
- **Výkon** - Implementace aplikace, respektive jejich služeb, musí být schopna zvládnout zátěž, která je na ně kladena. To zahrnuje schopnost zvládnout požadavky na výkon a škálovatelnost.

4.2 Požadavky na HW

Hardware, na kterém bude aplikace provozována, musí výkonnostně dostačovat pro provozování testovacích scénářů a sběr a vizualizaci dat. Týká se to primárně počtu jader, velikosti paměti a rychlosti diskového I/O. Provozované služby mají určitou základní režii, která se musí brát v potaz.

4.3 Cíle monitorování

Efektivní monitorování hraje klíčovou roli při vyhodnocení a porovnání výkonnosti a technických vlastností kompilace JIT a nativní AOT. Hlavní cíle tohoto monitorování jsou následující:

- **Zkušenosti s vývojem** - Jedním z klíčových cílů je zachytit a analyzovat dopad různých kompilačních strategií na proces vývoje. To zahrnuje sledování doby sestavení, cyklů nasazení a celkové snadnosti integrace a nasazení v rámci architektury mikroslužeb. Posouzením těchto faktorů můžeme poskytnout subjektivní i objektivní náhled na to, jak jednotlivé metody kompilace ovlivňují každodenní zkušenosti vývojářů, včetně potenciálních problémů nebo efektivity, které přinášejí přístupy JIT nebo AOT.

- **Srovnání výstupů** - Tento cíl se zaměřuje na přímé porovnání hmatatelných výstupů metod kompilace JIT a AOT. Konkrétně se bude sledovat velikost vytvořených spustitelných souborů, inicializační časy (jak rychle jsou služby po nasazení funkční) a využití zdrojů během běhu (využití procesoru a paměti). Pochopení těchto aspektů pomůže vymezit provozní efektivitu nebo režijní náklady spojené s každou kompilační strategií.

- **Výkonnostní metriky** - Pro tuto studii je rozhodující porovnání výkonnostních ukazatelů za podobných provozních podmínek. Mezi sledované metriky patří doba odezvy, propustnost (počet požadavků, které je služba schopna zpracovat za jednotku času), chybovost a stabilita systému při zatížení. Tyto ukazatele poskytnou kvantitativní základ pro porovnání účinnosti kompilací JIT a AOT při zvládání reálné provozní zátěže.

K dosažení těchto cílů bude zavedena komplexní monitorovací sestava zahrnující nástroje a postupy, které poskytují údaje a poznatky v reálném čase. Tento přístup zajistí, že shromážděná data budou robustní, spolehlivá a vhodná pro provedení důkladné srovnávací analýzy, a podpoří tak informovaná rozhodnutí týkající se optimálního využití kompilace JIT a AOT při nasazení mikroslužeb.

4.4 Výběr technlogií

Součástí tvorby tech stacku je výběr technologií, které budou použity pro implementaci aplikace. Výběr technologií je závislý na požadavcích na aplikaci a HW, na kterém bude aplikace provozována.

4.4.1 Organizace a správa zdrojů

Pro správu souborů práce byl zvolen SCM Git. Git je open-source verzovací, který umožňuje vytvářet, spravovat a sdílet soubory. Git je schopný pracovat s větvemi, které umožňují vytvářet paralelní vývojové větve.

Za účelem jednoduché organizace souborů bylo zvoleno řešení monorepozitáře. Monorepozitář je repozitář, který obsahuje veškeré soubory projektu, ale také relevantní

dokumentaci, obrázky, podpůrné nástroj a zdrojové soubory diplomové práce. Následující struktura adresářů byla zvolena pro organizaci souborů.

- **Documentation** - adresář obsahující dokumentaci aplikace.
- **Source** - adresář obsahující zdrojové soubory aplikace.
- **Thesis** - adresář obsahující zdrojové soubory textu diplomové práce a práci samotnou ve formátu pdf.

Pro zaručení dostupnosti a sdílení veškerých prostředků souvisejících s prací byl vybrán GitHub, jakožto server pro hostování repozitáře. GitHub je open-source platforma pro verzování souborů a projektů. Navíc poskytuje rozšiřující možnosti jako je CI/CD, správa dokumentace a další. Repozitář projektu je veden jako veřejný s licencí MIT.

4.4.2 Kontejnerizace a orchestrace

Základním prvkem nasazení aplikace je kontejnerizace a orchestrace. Kontejnerizace zajišťuje, že aplikace bude spouštěna v izolovaném prostředí, které je nezávislé na hostitelském systému. Orchestrace zajišťuje, že aplikace bude spouštěna na dostupných zdrojích a bude schopna zvládnout zátěž, která je na ni kladena.

Pro kontejnerizaci byla zvolena technologie Docker. Docker je open-source platforma pro kontejnerizaci aplikací, která umožňuje vytvářet, spouštět a spravovat kontejnery.

Pro orchestraci byla vybrána technologie Kubernetes. Kubernetes je open-source platforma pro orchestraci kontejnerů, která umožňuje automatizovat nasazování, škálování a správu aplikací. Kubernetes je schopný pracovat s kontejnery, které jsou vytvořeny pomocí Dockeru.

4.4.3 Konfigurace nasazení

Pro konfiguraci nasazení byla zvolena technologie Helm. Helm je open-source platforma pro správu balíčků, která umožňuje vytvářet, spravovat a nasazovat balíčky. Helm je schopný pracovat s balíčky, které jsou vytvořeny pomocí Kubernetes.

Definice balíčků je řešena pomocí konfiguračních souborů, které jsou použity již při tvorbě obecného obrazu. V rámci Helm je základním prvkem chart, který obsahuje definici balíčku a šablonu, která je použita pro generování konfigurace.

4.4.4 Persistenční vrstva

Jako relační databáze byla vybrána Postgres. Je schopná pracovat s relačními daty, které jsou uloženy v tabulkách. Poskytuje základní klientský balíček pro .NET, jenž

umožňuje komunikaci s databází. Tento balíček je kompatibilní s nativní AOT kompilací.

Za účelem persistence a zprostředkování dat z testování je použita InfluxDB. Tato timeseries databáze umožňuje ukládat a spravovat časové řady. Využití InfluxDb je pragramtické z důvodu nativní podpory napojení InfluxDB v1 z nástroje K6 pro export testovacích dat.

4.4.5 Komunikační metody

Za účelem analýzy možností komunikace klienta se službami, ale i interní komunikace, bylo vybráno k implementaci hned několik protokolů.

- **REST API** - V rámci Kestrel serveru každé služby je využit protokol HTTP/1 a komunikace pomocí REST API. Toto rozhraní slouží pro komunikaci klienta se službou a poskytuje data ve formátu JSON.
- **gRPC** - Vybrané služby implementují komunikaci pomocí protokolu HTTP/2 a gRPC. Za tímto účelem mají zmíněné služby otevřené rozhraní na dodatečném portu. gRPC protokol je využit přístupem model first, tedy rozhraní je definováno pomocí protobuf souboru a následně je vygenerován kód pro komunikaci.
- **RabbitMQ** - Pro implementaci komunikace prodle vzoru Publish - Subscribe byl vybrán message broker RabbitMQ. Umožňuje službám odebírat data z jiných služeb a zároveň poskytovat data jiným službám. Tím je zajištěna asynchronní messaging mezi službami.

4.4.6 Monitorovací nástroje

Pro monitorování aplikace byl zvolen Grafana observability stack pro jeho pokrytí komplexní škály monitorovacích dat. Grafana observability stack zahrnuje nástroje pro sběr, vizualizaci a analýzu dat. Následující nástroje byly vybrány pro implementaci monitorovacího stacku:

- **Grafana** - Grafana je open source webová aplikace pro analýzu a interaktivní vizualizaci dat. Poskytuje možnost sestavit dashboard z komponent jako jsou grafy, tabulky a další. Jedná se o velmi populární technologii v doménách serverové infrastruktury a monitorování. Grafana umožňuje sjednotit monitorovací služby a zobrazit data v reálném čase. Podporuje širokou škálu datových zdrojů, jako jsou Prometheus, InfluxDB, Tempo, Loki nebo Elasticsearch, což umožňuje jednoduchou konfiguraci a připojení cílových dat. Kombinací dat z různých zdrojů umožňuje vytvářet komplexní pohled na celý systém. To je obzlvášť cenné pro analýzu systému pomocí kombinací metrických dat.

- **Prometheus** - Open-source monitorovací systém. Shromažďuje a ukládá metriky jako time-series data a umožňuje se na ně dotazovat pomocí vlastního výkonného jazyka PromQL. Prometheus je zvláště vhodný pro monitorování microservice architektur díky své schopnosti automaticky objevovat cíle. Jeho architektura podporuje více modelů získávání dat, stahování metrik z cílových služeb nebo collectorů, odesílání metrik přes gateway a zprostředkování notifikací.

- **Loki** - Škálovatelný agregátor logů. Na rozdíl od obdobných systémů pro agregaci logů, jenž indexují všechna data, Loki indexuje pouze metadata, přičemž ukládá celá data logu efektivním způsobem. Loki je navržen tak, aby jednoduše spolupracoval s Grafanou a umožňuje rychle vyhledávat a vizualizovat logy.

- **Tempo** - Je snadno ovladatelný open-source backend pro distribuované sledování požadavků. Tempo podporuje ukládání a načítání traces, které jsou přijímány ze zdrojů jako Jaeger, Zipkin a OpenTelemetry. Na rozdíl od mnoha jiných systémů pro traces nevyžaduje Tempo žadné předem definované schéma. Je navržen tak, aby se bezproblémově integroval s Prometheus a Loki.

- **OpenTelemetry** - Open source kolektor telemetrických dat. Poskytuje jednotný, vendor-agnostic způsob sběru, zpracování a exportu telemetrických dat Je konfigurovatelný a podporuje více pipeline, které mohou upravovat telemetrická data při jejich průchodu. Výrazně zjednodušuje instrumentaci služeb, protože umožňuje agregovat a exportovat metriky, taces a logy do různých analytických a monitorovacích nástrojů. Poskytuje podporu pro export dat do Prometheus, Tempo i Loki.

- **Cadvisor** - Open-source monitorovací nástroj pro kontejnery. Poskytuje informace o využití zdrojů a výkonu. Cadvisor je schopen monitorovat kontejnery běžící na Dockeru, Kubernetes nebo jiných kontejnerových platformách. Poskytuje informace o využití procesoru, paměti, sítě a diskového I/O.

- **NodeExporter** - Nástroj pro sběr metrik z hostitelského systému. Poskytuje informace o využití procesoru, paměti, sítě a diskového I/O.

4.4.7 Testovací nástroje

Za účelem testování monitorovacího stacku byl vybrán nástroj K6. Jedná se o moderní open-source nástroj pro testování zátěže. Slouží k vytváření, provádění a analýze výkonnostních testů softwarových aplikací. Nabízí čisté skriptovací rozhraní s jazykem JavaScript, které umožňuje psát, jak komplexní testovací scénáře napodobující reálný provoz systému, tak i vytvářet nereálné nebo hraniční situace. K6 podporuje různé

systémové metriky, jako je doba odezvy, propustnost a chybovost. Nabízí široké možnosti rozšíření skrze API, což umožňuje přizpůsobení a integraci s dalšími nástroji pro komplexní sledování výkonu.

4.4.8 Testovací služby

Pro implementaci testovacích služeb z podstaty práce zvolena technologie .NET, konkrétně jazyk C#. Služby budou implementovány jako mikroslužby a budou podporovat kontejnerizované nasazení v microservice architektuře. Služby budou vytvořeny tak, že každou dílčí službu reprezentuje projektový soubor s doménových kódem. Celé řešení spolu s dílčími knihovnami bude součástí jednoho solution souboru.

Pro řešení byla vybrána nejnovější verze .NET SDK 8.0, která poskytuje nejrozsáhlejší impelmentaci a podporu pro nativní AOT kompilaci. Jakožto nástroj pro vývoj a správu projektů byl zvolen JetBrains Rider. Rider je IDE, které poskytuje širokou škálu funkcí pro vývoj aplikací v .NET.

Konkrétní knihovny použité v rámci implementace budou záviset na konkrétních požadavcích na služby a popsány v následující sekci.

4.5 Návrh a implementace testovacích služeb

Následující pasáž se zabírá návrhem a implementací testovacích služeb, které budou využity pro analýzu vývoje a výkonu jednotlivých kompilací AOT a JIT v rámci .NET.

4.5.1 Architektura

Pro implementaci požadované funkcionality bylo zvoleno následující rozdělení zodpovědnosti služeb:

- **SRS - Signal reading service** - Služba simuluje roli čtecího zařízení. Generuje data a poskytuje je ostatním službám. Poskytuje REST API rozhraní.
- **FUS - File Upload Service** - Zprostředkovává datové persistentní zapisovací zařízení. Čte nebo zapisuje data do PostgreSQL databáze. Poskytuje REST API a gRPC rozhraní.
- **BPS - Batch Processing Service** - služba, která zpracovává data z jiných služeb. Reaguje na požadavek o hromadném zpracování při předem definovaném splnění podmínek. Poskytuje REST API a gRPC rozhraní. Je přihlášena do RabbitMQ jako subscriber.

- **EPS - Event Publishing Service** - slouží k vyvolání události, která je následně zpracována jinými službami. Poskytuje REST API rozhraní. Je přihlášena do RabbitMQ jako publisher.

obrázek architektury

Kompilaci do nativního AOT kódu je deklarována použitím atributu PublishAoT v projektovém souboru. Za účelem zajištění co největší podobnosti služeb zacílených na AOT a JIT kompilaci, bude využito zadefinování konstantních hodnot v rámci projektu. Konstanty *JIT* a *AOT* budou využity pro rozlišení chování služeb v rámci obou kompilačních verzí. S použitím direktiv kompilátoru a zmíněných konstant bude v nutných případech docíleno rozdílného volání API při snaze zachovat totožnou funkcionalitu.

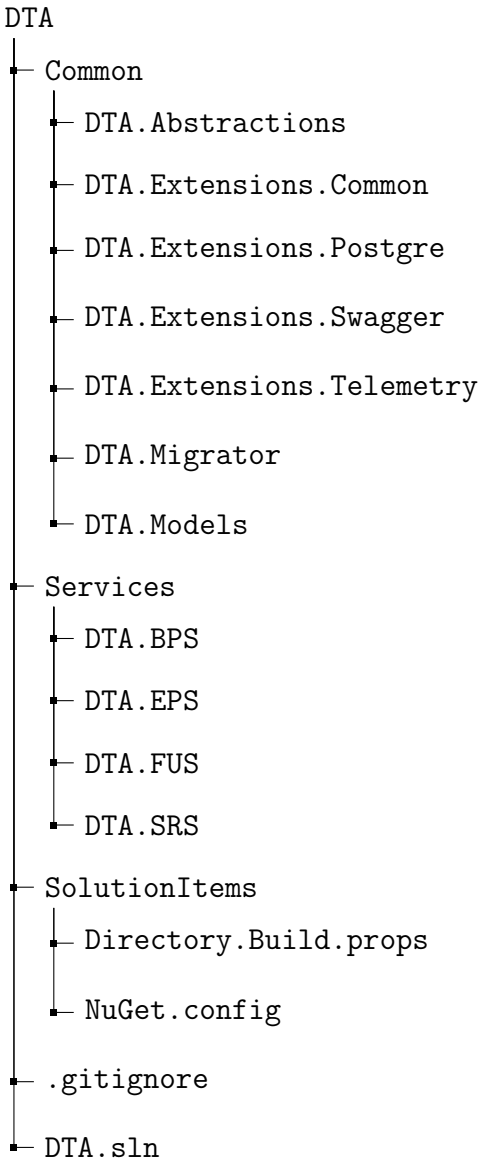
4.5.2 Očekávání vývojového procesu

Na základě podporované funkcionality, tak jak je definována týmem .NET a popsána v rámci rešerše, je očekáváno, že vývojový proces bude probíhat bez výrazných problémů a bude možné vytvořit služby, které budou schopny zvládnout definované funkční a nefunkční požadavky. Podpora 3. stran byla předem prozkoumána v rámci dostupných dokumentací vybraných knihoven .NET. Konkrétní podoba a rozsah této podpory bude plně ověřitelná až v rámci implementace a testování služeb.

foto využití konstant v kódu

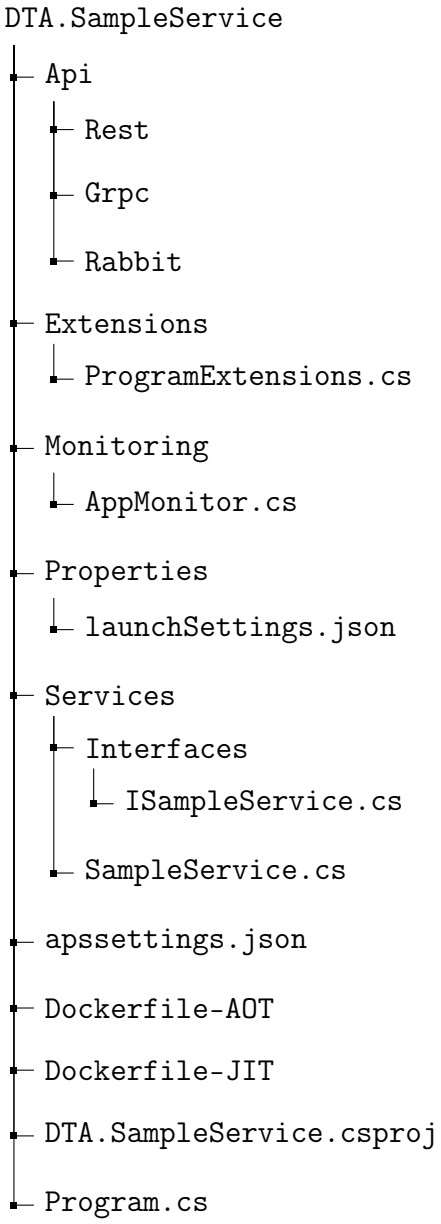
4.5.3 Organizace zdrojových souborů služeb

Organizace zdrojových souborů služeb, knihoven a pomocných souborů je řešena v rámci hlavního adresáře obsahujícího .NET solution soubor, pomocné soubory a solution složky s konkrétními projekty služeb a knihoven. Následující stromový graf představuje adresářovou strukturu projektu.



4.5.4 Společná struktura služeb

Každá z vyvinutých služeb využívá konkrétní .NET SDK *Microsoft.NET.Sdk.Web*, které umožňuje využít WebApplication pro registraci a konfiguraci funkcionality služby a zároveň poskytuje konfigurovatelný Kestrel server. Pro zajištění jednotného přístupu k logování, metrikám a konfiguraci byly vytvořeny společné knihovny, které jsou využity ve všech službách.



- **Api** - obsahuje implementaci rozhraní služby
- **Extensions** - implementuje extension metody specifické pro doménu služby
- **Monitoring** - obsahuje statickou třídu, která drží reference na počítadla metrik
- **Service** - ve složce jsou implementovány služby, které provádějí doménovou logiku služby
- **Properties** - drží konfiguraci pro spuštění služby
- **Program.cs** - vstupní bod služby
- **appsettings.json** - konfigurace služby

- **Dockerfile-AOT** - soubor pro tvorbu Docker obrazu pro AOT kompilaci
- **Dockerfile-JIT** - soubor pro tvorbu Docker obrazu pro JIT kompilaci

Specifické služby obsahují dodatečné adresáře a soubory nutné pro implementaci jejich doménové funkce.

4.5.5 Knihovny 3. stran

Pro implementaci funkcionality aplikace byly využity následující knihovny třetích stran:

- **Npgsql** - Npgsql je open-source ADO.NET provider pro PostgreSQL, který umožňuje komunikaci s PostgreSQL databází. Npgsql poskytuje základní balíček funkcí pro vytvoření připojení na základě standardizovaného řetězce pro připojení. Tento balíček sice není plně kompatibilní s AOT kompilací, funkce které jsou využity v rámci aplikace jsou avšak kompatibilní.
- **Dapper** - ORM knihovna pro .NET, která umožňuje mapovat databázové struktury na C# objekty a vytvářet a provádět dotazy na databázi. *Dapper.AOT* je dílčí knihovna, která umožňuje vytvářet a provádět dotazy na databázi v rámci AOT kompilace. Toho je zajištěno tím, že Dapper.AOT generuje kód pro dotazy na databázi v době kompilace. Využívá k tomu interceptorů a generátorů. Samotný balíček Dapper.AOT obsahuje další knihovnu - *Dapper.Advisor*, která pomáhá s analýzou zdrojového kódu a generováním kódu pro dotazy na databázi.
- **OpenTelemetry** - OpenTelemetry zprostředkovává množinu knihoven pro sběr, zpracování a export telemetrických dat. V rámci knihovny je umožněno registrace vlastních metrik, logů a traces, ale také nastavení exportu vybraných systémových dat sbíraných v rámci knihoven .NET.
- **Grpc** - Knihovny pro implementaci komunikace pomocí protokolu HTTP/2 a gRPC. Konkrétně jsou využity *Grpc.AspNetCore* v případě serveru, *Grpc.Net.Client* pro klienta a *Google.Protobuf* s *Grpc.Tools* pro generování modelů v přístupu model first.
- **RabbitMQ** - Komunikace a implementace publish subscribe vzoru je umožněna knihovnou *RabbitMQ.Client*. S její pomocí jsou vytvářeny fronty, dochází k přihlášení k odběru zpráv a jejich publikování.
- **Swagger** - Grafické rozhraní pro vizualizaci a testování REST API služeb. Swagger je využit pouze v kombinaci konfigurací *JIT Debug*. K tomuto účelou jsou využity knihovny *Swashbuckle.AspNetCore* a *Microsoft.AspNetCore.OpenApi*.

4.5.6 Společné knihovny

V rámci zjednodušení tvorby služeb, jednotné implementaci a konfiguraci, ale také z důvodu zajištění některé základní ale klíčové funkcionality, byly vytvořeny společné knihovny. Tyto knihovny obsahují společné třídy, rozhraní a konfigurace, které jsou použity ve všech službách.

- **Persistence** - Pro implementaci persistence byla vytvořena pomocná knihovna *DTA.Extensions.Postgres*, která poskytuje pomocnou funkcionalitu pro zajištění existence databáze pro službu, dle konfigurace v řetězci pro připojení.
- **Migrace** - Zajištění migrace databáze bylo implementováno po vlastní ose minimalistickým migrátorem v knihovně *DTA.Migrator*. Tato knihovna poskytuje základní funkcionalitu pro vytvoření databáze, vytvoření tabulek a indexů, ale také zajištění migrace dat a verzování změn.
- **Telemetrie** - Knihovna *DTA.Extensions.Telemetry* zprostředkovává extensions metody pro jednotnou a jednoduchou registraci sběru a export telemetrických dat napříč službami.
- **Modely** - Knihovna *DTA.Models* obsahuje společné modely, které jsou využity ve službách. Je tím docílena viditelnost na datové struktury rozhraní aplikace napříč všemi službami, jež knihovnu referencují.
- **Obecná funkcionalita** - Za účelem sjednocení funkcionality využité napříč všemi službami jsou implementovány extension metody v knihovně *DTA.Extensions.Commor* Zde je poskytnuta funkcionalita pro sestavení názvů pro službu.

4.5.7 Společná konfigurace

Součástí řešení je společná konfigurace, která je využita ve všech službách. Ta je řešena jedna na úrovni solution souboru, tak i Directory.Build.props souboru. Týká se jednotné distribuce projektových atributů pro verzi, kompatibilitu s AOT, vynucení konkrétních pravidel pro kód a analyzéry.

4.5.8 SRS - Signal reading service

Za účelem simulace funkce čtecího zařízení byla vytvořena služba SRS. Tato služba poskytuje základní rozhraní pro získání dat signálu včetně jednotek a značek formou REST API. Pro zjednodušení implementace není využito čtení dat ze skutečného zdroje, ale jsou generována náhodná data. Načež data jsou následně poskytována se zdržením simulujícím čtení dat ze vzdáleného zdroje.

Služba poskytuje následující rozhraní

- **GET /api/signals/{int:amount}** - Vygeneruje zadané množství náhodných signálů

4.5.9 FUS - File Upload Service

Služba v systému hraje roli zapisovacího zařízení, které zapisuje a čte data z perzistentního úložiště. Jakožto úložiště je využito PostgreSQL databáze. Služba využívá vlastní databázovou instanci, spravuje vlastní tabulky pomocí migrací.

Poskytuje rozhraní formou REST API pro zápis a čtení dat. Daty je myšlen libovolný soubor v libovolném formátu. Samotná podstata nahraných dat není pro službu důležitá, ale je zpracována a uložena do databáze. Za účelem sehrání testovacích scénářů poskytuje SRS také gRPC rozhraní, které je zajištěno na dedikovaném portu. V rámci gRPC komunikace slouží služba jako server, který splňuje volání vzdálené procudery.

Služba poskytuje následující rozhraní

- **GET /api/file/download/{int:id}** - Stáhne soubor podle zadaného ID.
- **POST /api/file/upload** - Nahraje soubor do systému.
- **gRPC Operation FileServer.GetFile** - Stáhne soubor podle zadaného objektu s ID.

4.5.10 BPS - Business Processing Service

Pro splnění role a požadavků na zpracování dat z jiných služeb byla vytvořena služba BPS. Tato služba získává data, provádí náročné výpočetní operace, sloužící k simulaci obtížných doménových operací. Konkrétně implementaváno je neefiktivní rekurzivní výpočet Fibonacciho posloupnosti a faktoriálu.

Služba se po spuštění přihlašuje k odběru zpráv na předem definovaný kanál *simulated* na službe *RabbitMQ*. Po získání zprávy získává data ze služby FUS pomocí volání vzdálené procedury. Po získání dat provádí náročné výpočetní operace, které jsou simulovány náhodným čekáním.

Služba poskytuje následující rozhraní

- **GET /api/processFibonacci/{int:degree}** - Vypočítá číslo z Fibonacciho posloupnosti na zadané pozici náročným rekurzivním způsobem.
- **Event subscribed: {queue-name}_simulated** - Přihlášení k odběru zpráv v rámci kanálu na službě RabbitMQ.

4.5.11 EPS - Event Publishing Service

Jednoduchá službami umožňující vyvolat událost v systému a docílit spuštění dodatečných operací v systému. V systému simuluje roli vydavatele událostí Poskytuje REST API rozhraní pro vyvolání události. Po vyvolání události je zpráva publikována do RabbitMQ kanálu, kde je zpracována jinými službami.

Služba poskytuje následující rozhraní

- **GET api/simulateEvent/{int:id}** - Vyvolá simulovanou událost s daným ID.
- **Event published: {queue-name}_simulated** - Vyvolá údalost se zprávou obsahující identifikator na konfigurovaném kanálu do služby RabbitMQ.

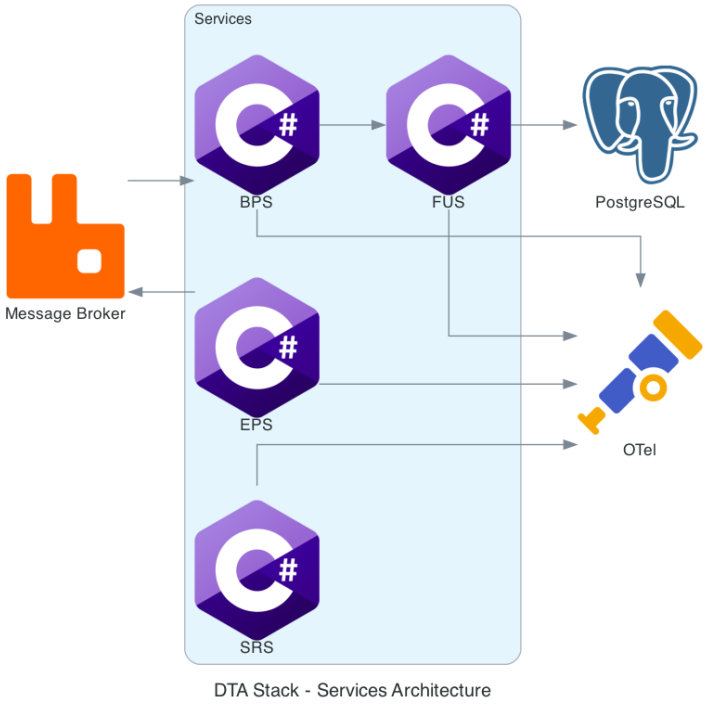
4.5.12 Přehled řešení

Řešení aplikace sestává z následujících sekcí a jednotlivých obrazů služeb a verzí, definovaných v rámci Docker Compose souboru. Obrazy jsou zapsány ve formátu *repository:tag*.

- **Testovací služby - JIT**
  - **dta-srs:jit-latest**
  - **dta-fus:jit-latest**
  - **dta-bps:jit-latest**
  - **dta-eps:jit-latest**
- **Testovací služby - AOT**
  - **dta-srs:aot-latest**
  - **dta-fus:aot-latest**
  - **dta-bps:aot-latest**
  - **dta-eps:aot-latest**
- **Komunikace**
  - **rabbitmq:3-management-alpine**
- **Monitorovací nástroje**
  - **dta-lgtm:latest**
  - **node-exporter:latest**

  - **cadvisor-arm64:v0.49.1**

- **Persistence**
  - **postgres:latest**
  - **influxdb:1.8.10**
- **Směrování**
  - **nginx:alpine**

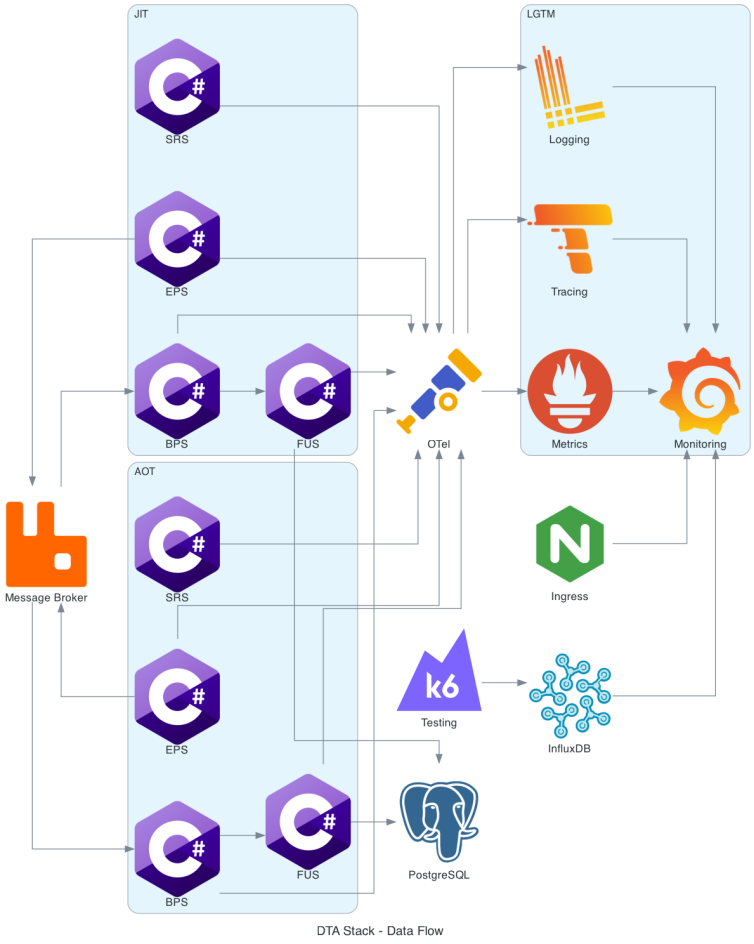


Obrázek 4.1 Diagram .NET služeb a závislých služeb

Následující diagram znázorňuje vztahy mezi jednotlivými službami.

4.6 Konfigurace aplikace

Za účelem běhu aplikace je klíčové správné nastavení konfigurace. Konfigurace je řešena na různých úrovních. Základní úroveň představuje soubor *compose.yaml*, který



Obrázek 4.2 Telemetrie ve stacku

inforumje Docker orchestrátor jaká infrastrukura má být vytvořena a jak má být nastavena, které služby mají být vytvořeny a jaké zdroje jim mají být přiděleny. Dále jsou v tomto souboru nastaveny základní parametry jako je jméno sítě, názvy volume (persistentní úložiště orchestrátoru), závislosti mezi službami a další. V neposlední řadě je dílčí konfigurace jednotlivých služeb řešena v rámci konfiguračních souborů a proměnných prostředí.

4.6.1 Konfigurace infrastruktury

Jak již bylo zmíněno, základní konfigurace je řešena v souboru *compose.yaml* a je rozdělena do následujících částí:

- **volumes** - V této sekci jsou definovány všechny volume, které jsou využity v rámci stacku. Volume jsou definovány názvem a využity pro ukládání dat služeb. Konkrétně jsou využity úložiště pro data Grafany, OpenTelemetry, Prometheus a InfluxDB.

- **networks** - Konfigurace pro vnitřní síť aplikace, která je využita pro komunikaci mezi službami. Síť je definována názvem a typem. Pro potřeby aplikace se využívá síť s názvem *stack-network* a typ *bridge*, jenž funguje jako síťový most.

- **services** - Definuji sekci pro jednotlivé služby, které jsou součástí stacku. Každá služba je definována názvem služby - *container_name*, názvem obrazu - *image*, v případě lokálně sestavených služeb také definicí sestavení - *build*. Dále je definováno, jaké porty jsou mapovány z kontejneru do hostitelského systému - *ports*, jaké volume jsou připojeny k kontejneru - *volumes*, použité síťové rozhraní - *networks*, závislosti služby - *depends_on* a proměnné prostředí - *environment*. V případech testovaných služeb je uvedena dodatečná konfigurační sekce nasazení - *deploy*, jenž limituje dostupné zdroje paměti a procesoru.

4.6.2 Konfigurace směrování

Nastavení směrování v rámci stacku je řešeno konfigurací proxy služby Nginx. To je řešeno odděleně dodatečným konfiguračním souborem nginx.conf jenž je připojen do kontejneru služby. Soubor obsahuje následující směrovací pravidla:

- / - cesta na statickou hlavní stránku-rozcestník aplikace

- /grafana - směrování na Grafanu

Rozcestník představuje soubor html.index a je připojen do virualizovaného repozitáře kontejneru obdobným způsobem jako konfigurační soubor.

4.6.3 Konfigurace telemetrie

Nastavení telemetrie spočívá v definici rozhraní, nastavení chování služeb a systémů z niž se telemetrická data sbírají, jejich cíl pro zpracování, správu a vizualizaci. Značnou část konfigurace představuje propojení nástrojů stacku LGTM, k čemuž slouží konfigurační soubory. Ty obsahují výchozí minimalistickou konfiguraci pro jednotlivé nástroje. Výstupem LGTM je individuální kontejner a zmíněná konfigurace je náturou interní a není potřeba s ní manipulovat s ohledem na požadavky práce uživateli. Mezi dodatečné konfigurace telemetrie napříč službami patří:

- **Testované služby** - Veškeré testovací služby mají nastavený endpoint pro export telemetrických dat. Toto nastavení je zprostředkováno proměnnou prostředí *OpenTelemetrySettings_ExporterEndpoint*.
- **LGTM** - Konfigurace pro LGTM je řešena pomocí proměnných prostředí. Je definována adresa a cesta, z které je k dispozici Grafana. Dále je umožněno anonymní přihlášení do Grafany.
- **Cadvisor** - Nastavení služby představuje dodatečné nastavení volumes, které jsou připojeny k kontejneru. Připojením systémových souborů je zajištěno sběr dat o využití systémových zdrojů. Toto nastavení je závislé na operačním systému.

4.6.4 Konfigurace testovacích služeb

Jednotlivé služby mají vlastní dodatečné konfigurace, které jsou řešeny pomocí kombinace dle standardizovaného postupu pro konfiguraci .NET aplikací, a to konfiguračního souboru *appsettings.json* a proměnných prostředí. V první řadě řadě je použita konfigurace ze souboru, načež je přepsána odpovídajícími hodnotami proměnných prostředí. Každá služba má definovaný specifický prefix pro identifikaci proměnných prostředí. Následující seznam popisuje dodatečné konfigurace jednotlivých služeb.

- **FUS - File Upload Service** - Obsahuje connection string (přístupový řetězec) pro připojení do databáze PostgreSQL.
- **BPS - Batch Processing Service** - Disponuje konfigurací pro připojení k RabbitMQ a konkrétní frontě.
- **EPS - Event Publishing Service** - Drží informaci o rozhraní RabbitMQ, konkrétní frontě a gRPC rozhraní služby FUS.

4.6.5 Konfigurace persistence

Služby využité pro perzistentní ukládání dat jsou konfigurovány pomocí proměnných prostředí. Jedná se o následující služby:

- **PostgreSQL** - Jsou definovány údaje pro uživatele databáze a název databáze.
- **InfluxDB** - Proměnné prostředí definují název databáze, uživatelské údaje a povolení přihlášení pomocí http.

4.6.6 Nastavení uživatelského rozhraní

Definice uživatelského rozhraní, respektive dostupných dashboardů, je dána při sestavení obrazu LGTM. V rámci něj jsou předdefinovány hodnoty pro připojení zdrojů dat, tj. Prometheus, Loki, Tempo a InfluxDb. Patřičné dashboardy zobrazující relevantní data pro různé scénáře systému byly předem připraveny a jsou k dispozici po otevření Grafany anonymním uživatelem. Následující seznam popisuje klíčové soubory konfigurace uživatelského rozhraní.

- **grafana-dashboards.json** - Definuje dostupné dashboardy v Grafaně. Dashboardy jsou definovány v JSON formátu a obsahují definici panelů, zdrojů dat a dalších parametrů.
- **grafana-datasources.json** - Obsahuje zdroje dat z kterých Grafana, respektive dashboardy, čerpají data.

5 TESTOVÁNÍ SCÉNÁŘŮ

Testování scénářů je klíčovou součástí testování výkonu mikroslužeb. Scénáře jsou definovány jako soubor kroků, které mají být provedeny, a jsou použity k simulaci zátěže na mikroslužby. Scénáře jsou vytvořeny pomocí testovacích nástrojů, které umožňují vytvářet a spouštět testy, které simuluji reálné uživatelské scénáře.

5.1 Předpoklad scénářů

Scénáře musí být vytvořeny tak, aby simulovali reálné uživatelské scénáře. To znamená, že musí být vytvořeny tak, aby obsahovaly kroky, které mají být provedeny, a musí být vytvořeny tak, aby obsahovaly data, která mají být použita.

5.1.1 Očekávání výkonnosti služeb

Pro výsledné obrazy služeb kompilovaných AOT do nativního kódu je očekáváno, že budou zabírat výrazně menší paměť, než je tomu u ekvivalentních obrazů služeb kompilovaných pro JIT. Předpoklad zakládá na požadavku běhového prostředí, kdy nativní služby vyžadují pouze malou množinu knihoven nad OS Alpine. Oproti tomu služby kompilované do JIT vyžadují nad OS Alpine běžící .NET runtime, i když zredukovaný o nepoužívané knihovny. Samotná velikost výstupních souborů služeb je předpokládaná větší u nativního výstupu, než u ekvivalentního JIT výstupu. To je dáno tím, že nativní výstup obsahuje dodatečné třídy, konstrukce a funkce, které substituují běhové prostředí .NET.

Další očekávání se týká rychlosti odezvy služeb po spuštění. Předpokládá se, že služby kompilované do nativního kódu budou mít rychlejší odezvu než služby kompilované do JIT. To je dáno tím, že nativní kód je přímo spustitelný na cílovém systému, zatímco JIT kód vyžaduje dodatečný čas na kompilaci a optimalizaci.

5.2 Metodika testování

Cílem práce je prouzkomat dopady kompilace AOT a JIT na výkon. Pokouší se určit, která strategie kompilace nabízí lepší výkon při různémé zatížení a podmínkách. Metriky výkonu jsou shromážděny z dat o výkonu systému, doby odezvy služeb a využití zdrojů během standardizovaných testů.

Experimentální nastavení zahruje dvě hlavní součásti řízení testovací prostředí a testovací služby. Tyto služby jsou sestaveny ze stejné (až na vybranné nekompatibilní API) kódové báze v kompilačních režimech AOT i JIT. Testovací prostředí bylo standardizováno ve všech experimentech, aby bylo zajištěno, že jakékoli pozorované rozdíly ve výkonu lze připsat výhradně metodám kompilace a nikoli variantách hardwaru nebo

softwaru.

Testy budou prováděny na identických hardwarových nastaveních s následujícími specifikacemi:

- **Operační systém**: MacOS Sonoma 14.4.1
- **Procesor**: Apple M1 8-core CPU
- **Paměť**: 8 GB LPDDR4X

Operační systém a všechny služby na pozadí budou udržovány konzistentní, aby se minimalizovaly vnější vlivy na výsledky výkonu. Testovaným službám budou omezeny zdroje v orchestraci, aby se zabránilo nespravedlivé výhodě služby v jedné kompilaci před druhou. Služby budou připraveny do obrazů a použity v rámci orchestrace pomocí kontejnerů. Jednotlivé obrazy jsou založeny na architektuře Arm64 a Linuxových distribucích Alpine.

Sběr dat je automatizován pomocí kombinace nástrojů pro monitorování systému a skriptů. K sběru dat hostitelského systému byl použit Cadvisor a Node Explorer. Telemetrie jednotlivých služeb je sbírána implementovanými metry a exportována do kolektoru OpenTelemetry. Testovací data jsou získán Statické informace o průběhu a výstupech kopilace jsou získány kombinací nástroj .NET CLI, Docker a souborového systému.

K porovnání strategií kompilace při různých podmínkách využívají testovací služby vybrané scénáře testující propustnost a vytížení zdrojů. Úlohy prováděné ve scénářích vystavují služby datovým transakcím, přístupů k externím zdrojům, komunikací pomocí různých metodik anebo samotné spuštění v rámci orchestrace. Údaje o výkonnosti budou analyzovány pomocí statistických metod, aby se určily významné rozdíly mezi službami kompilovanými v režimu AOT a JIT.

5.2.1 Hypotézy

Na základě předběžného přehledu literatury a teoretických výhod každého kompilačního režimu byly formulovány následující hypotézy:

- **Hypotéza 1**: Vývoj služeb bude možný za pomocí obou kompilačních režimů, s minimálními rozdíly v použitém API a systémových knihovnách. Při použití knihoven 3. stran bude dostatečně jasná kompatibilita mezi oběma režimy.
- **Hypotéza 2**: Kompilace AOT má za následek rychlejší spouštění, ale může vést k větším binárním velikostem ve srovnání s kompilací JIT. Výsledný virtualizovaný obraz bude ovšem mít výrazně nižší velikost kvůli absenci .NET runtime.

- **Hypotéza 3**: Kompilace JIT poskytuje lepší optimalizaci výkonu a paměti za běhu díky své schopnosti přizpůsobit se skutečnému běhovému prostředí.

5.3 Definice scénářů

Scénáře jsou vytvořeny jako množina javascriptových souborů splňujících požadavky API nástroje K6. Každý scénář je definován přes jeden nebo více scriptových souborů. Tyto soubory obsahují kroky, které mají být provedeny, a data, která mají být použita. Pro sjednocení obecných nastavení jsou vytvořený konfigurační soubory, které jsou využity ve více scénářích. Pro zjednodušené a automatizované spuštění testovacích scénářů jsou definovány runner skripty, které zajišťují spuštění testů spolu se správou orchestrace.

Pro dodatečnou identifikaci dat jednotlivých scénářů je užito InfluxDB tagů, které jsou přidány k jednotlivým voláním v testech. Tím je zajištěno, že data z jednotlivých scénářů jsou jednoznačně identifikována a lze je následně zpracovat.

- **dta_service** - Značka pro identifikaci služby, která je testována. Má standardní formát hodnot *Služba-Kompilační režim*, kdy služba může nabývat hodnot *SRS, FUS, BPS, EPS* a kompilační režim nabývá hodnot *JIT, AOT*.
- **test_scenario** - Značka pro identifikaci scénáře, který je testován. Má standardní formát hodnot *scenario + číslo*.
- **test_id** - Identifikátor konkrétního testovacího scénáře. Nabývá libovolné hodnoty a slouží pro identifikaci konkrétních instancí, tedy spuštění testovacího scénáře.

Každý scénař má definován vlastní dashboard v Grafaně, který je využit pro sledování výsledků testů v reálném čase. Zároveň je součástní každého scénáře readme soubor, jenž podrobněji popisuje jednotlivé kroky a data, která jsou využita.

5.4 Popis scénářů

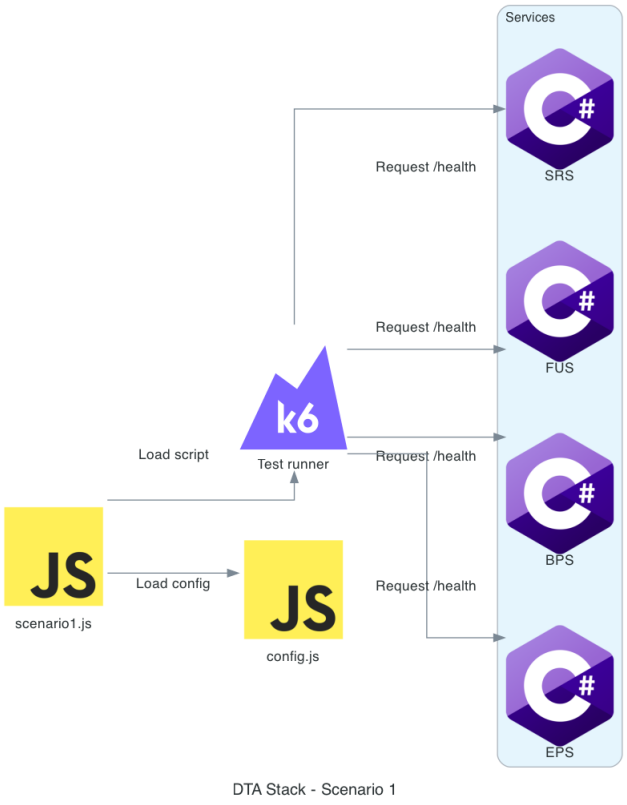
Následující sekce obsahuje popis scénářů, které byly vytvořeny pro testování výkonu a škálovatelnosti mikroslužeb kompilovaných JIT a AOT. Ke každému scénáři patří odpovídající sada souborů scriptů a konfigurací. Rovněž každý scénář disponuje vlastním interaktivním dashboardem v Grafaně, který umožňuje sledovat výsledky testů v reálném čase.

5.4.1 Scénář 1 - Výkonnost komunikace

Scénář 1 je zaměřen na schopnost mikroslužeb odpovídat na požadavky. K tomuto účelu je využit základní endpoint */health*, který informuje o stavu služby. Scénář je vytvořen tak, aby simuloval zátěž na mikroslužby a zjišťoval, zda jsou schopny odpovídat na požadavky.

Jelikož healthcheck endpoint je triviální ve své implementaci, nehraje roli další režie spojená se zpracováním logiky požadavku. Tímto je zajištěno, že se otestuje maximální vliv jednotlivých nasazení na výkon a škálovatelnost mikroslužeb.

Scénář se dělí na více kroků, aby při každém byl zjištěn dostatek zdrojů pro v systému pro testovanou službu. Krok je proveden vždy po určitém časovém intervalu, který je definován v konfiguračním souboru testu.



Obrázek 5.1 Diagram scénáře 1

Relevantní služby

- **SRS, FUS, BPS, EPS** - všechny služby s definovaným healthcheck endpointem

Průběh scénáře

- **Krok 1** - Spuštění služeb v rámci stacku
- **Krok 2** - Na služby jsou zasílány požadavky na healthcheck endpoint. Charakter požadavků je stupňující se k konfigurovanému maximu, načež zase klesá.
- **Krok 3** - Služby ukončují svoji činnost a zasílají data o provedeném testu

5.4.2 Scénář 2 - Přístup k perzistenci

Cílem tohoto scénáře je otestovat schopnost poradit si s vysokým množství asynchroních operací přístupu k datům. Scénář se pokouší identifikovat dodatečné režie spojené s přístupem k perzistenci a zjišťuje, zda jsou služby schopny zpracovat vysoký počet požadavků na databázi. Zejména je cílem pozorovat potencionál rozdíl v přístupu AOT a JIT zkompilované služby k systémovému API.

Relevantní služby

- **FUS** - služba pro přístup k perzistenci na databázi Postgres

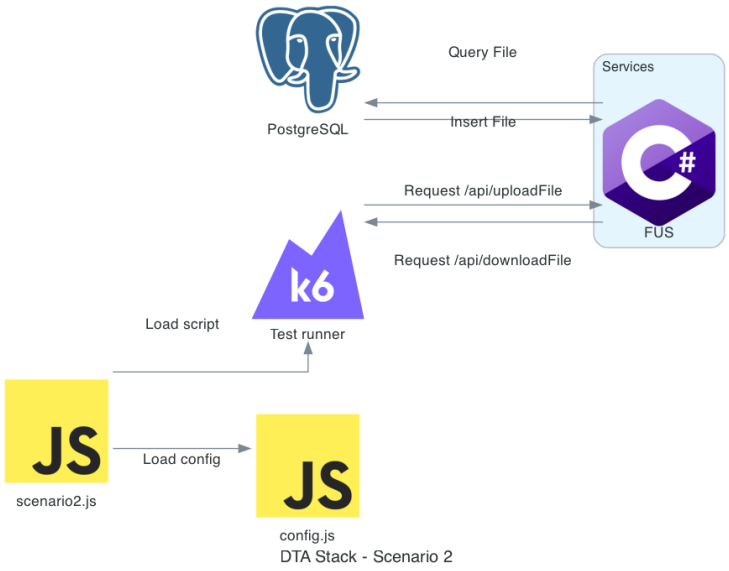
Průběh scénáře

- **Krok 1** - Služba je spuštěna v rámci stacku
- **Krok 2** - Na službu jsou zasílány požadavky na zápis i čtení dat z perzistentního úložiště. Charakter požadavků je stupňující se k konfigurovanému maximu, načež zase klesá.
- **Krok 3** - Služba ukončuje svoji činnost a zasílá data o prevedeném testu

5.4.3 Scénář 3 - Výpočetní zátěž

Cílem tohoto scénáře je otestovat schopnost mikroslužeb v jednotlivých kompilacích zpracovat náročnější operace. Scénář se zaměřuje na samotnou podstatu přístupu k vnitřnímu systémového API, efektivitě jeho využití a další režii, která by mohla být odlišná mezi JIT a AOT kompilací.

Předmětem scénář jsou dva výpočetně náročné algoritmy - faktoriál a Fibonacciho posloupnost. Tyto algoritmy jsou implementovány v rámci služby a jsou volány zvenčí.



Obrázek 5.2 Diagram scénáře 2

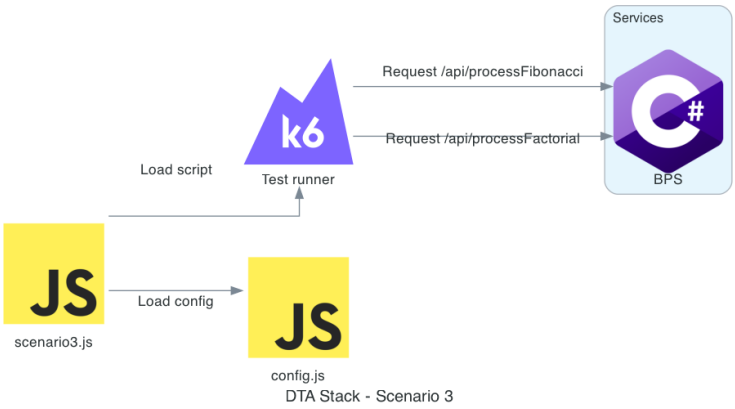
Scénář je vytvořen tak, aby simuloval zátěž na výpočetní jednotku a prozkoumal tak potencionální výkonnostní rozdíly v rámci přístupu k systémovému API a organizaci instrukcí.

Relevantní služby

- **BPS** - služba, která poskytuje rozhraní a logiku pro výpočet faktoriálu a Fibonacciho posloupnosti

Průběh scénáře

- **Krok 1** - Služba je spuštěna v rámci stacku
- **Krok 2** - Na službu jsou zasílány požadavky na výpočet faktoriálu a Fibonacciho posloupnosti. Charakter požadavků je stupňující se k konfigurovanému maximu, načež zase klesá.
- **Krok 3** - Služba ukončuje svoji činnost a zasílá data o prevedeném testu



Obrázek 5.3 Diagram scénáře 3

5.4.4 Scénář 4 - Vzájemná komunikace služeb

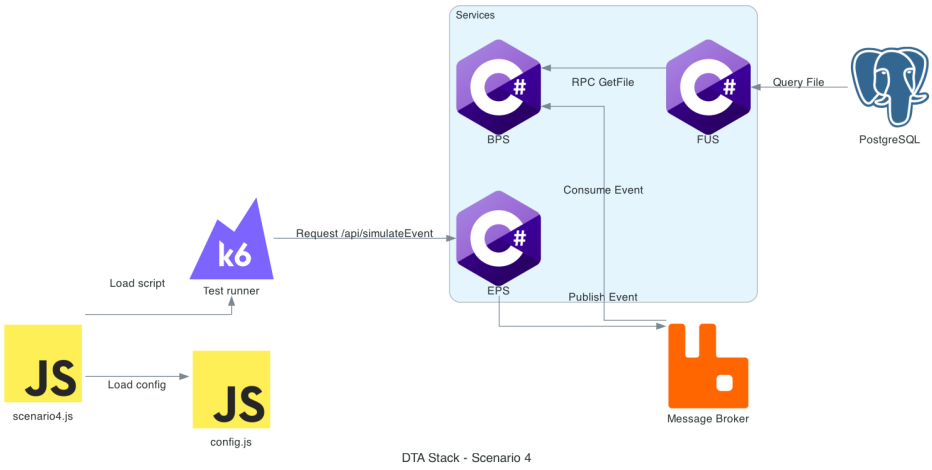
Tento scénář je zaměřen na rychlost a zátěž celkového systému při splnění požadavků vyžadující komunikaci mezi službami. Scénář je vytvořen tak, aby vyvolal událost z jedné služby, která je zpracována jinou službou. Pro splnění události je potřeba dat z perzistentního úložiště, která jsou získána ze třetí služby.

Relevantní služby

- **FUS** - služba hraje roli serveru, na něž se dotáže klient gRPC voláním. Následně přistupuje k perzistenci pro získání dat k splnění volání.
- **BPS** - poslouchá nad předem definovanou frontou a vyčkává na zprávu pro zpracování. V momentu přijetí zprávy, zpracovává vyvolanou událost a získává data ze vzdáleného volání z FUS.
- **EPS** - na základě přijatého volání přes REST API, zasílá služba EPS zprávu do předem definované fronty, na niž naslouchá BPS.

Průběh scénáře

- **Krok 1** - Služby jsou spuštěny v rámci stacku



Obrázek 5.4 Diagram scénáře 4

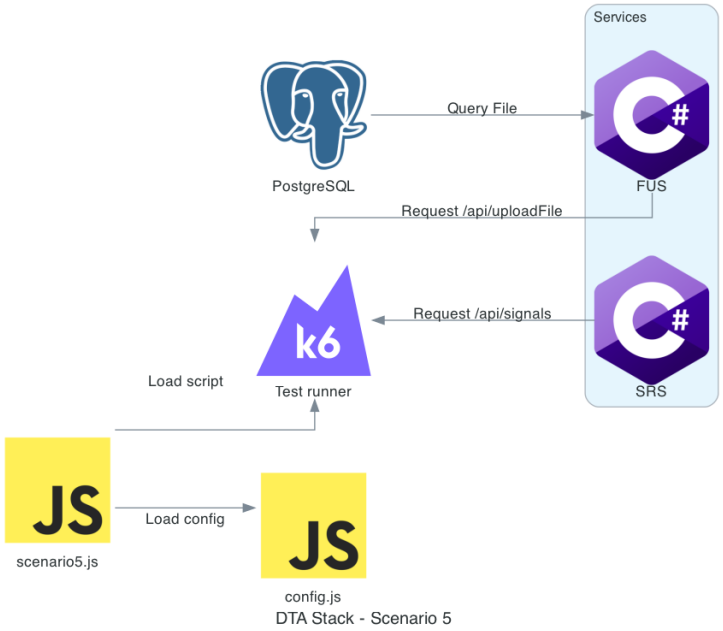
- **Krok 2** - Do služby EPS je zaslán požadavek na zpracování dat.
- **Krok 3** - Služba EPS zprávu zasílá do fronty, na kterou naslouchá služba BPS.
- **Krok 4** - Služba BPS zprávu zpracovává a získává data ze vzdáleného volání na službu FUS.
- **Krok 5** - Služba FUS získává data z perzistence a zasílá je zpět službě BPS.
- **Krok 6** - Služba BPS zpracovává data.
- **Krok 7** - Služby ukončují svoji činnost a zasílají data o prevedeném testu

5.4.5 Scénář 5 - Rychlost odpovědi po startu služby

Cílem tohoto scénáře je otestovat rychlost spuštění služby. Scénář testuje, jak rychle je služba schopna odpovědět na požadavek po spuštění. V rámci testu jsou testovány různé endpointy, které jsou volány po spuštění služby.

Základem scénáře je pomocí CLI příkazů vyvolat spuštění služby a ihned po jejím spuštění zaslat požadavek na získání dat.

Relevantní služby



Obrázek 5.5 Diagram scénáře 5

- **SRS** - služba je testována pro svoji nutnost serializace vygenerované datové odpovědi. Vyžaduje určitou množinu operací, jenž se podepíše dále nad rychlostí odpovědi služby a více přiblíží reálnému scénáři.
- **FUS** - za účelem otestování rychlosti odpovědi služby s ohledem na vazbu do dalšího systému je využita i služba FUS. S jejím přístupem k persistentnímu úložišti přiblíží scénář, kdy je nutné pro zpracování odpovědi nejen nastartovat službu, ale i získat data ze vzdáleného zdroje.

Průběh scénáře

- **Krok 1** - Služba je spuštěn v rámci stacku
- **Krok 2** - V návaznosti na spuštění služby je zaslán požadavek na získání dat.
- **Krok 3** - Služba SRS/FUS zpracovává požadavek a zprostředkovává data.
- **Krok 4** - Data jsou zaslána zpět klientovi.

- **Krok 5** - Služba ukončuje svoji činnost.

5.5 Spouštění scénářů

Jednotlivé scénáře jsou spouštěny formou pomocných runner skriptů v jazyce bash. Tyto scripty představují zjednodušenný způsob spuštění testů a zajišťují správu orchestrace testů, včetně spouštění a ukončování služeb, označení a exportování výsledků testů. Runner skripty mají jeden obecný parametr, kterým je identifikátor testu. Tento identifikátor je následně použit pro identifikaci výsledků testu a pro zobrazení výsledků v Grafaně.

5.6 Zpracování a vizualizace dat

Po provedení testování scénářů je nutné zpracovat a vizualizovat data, která byla získána. To zahrnuje jejich získání z patřičných zdrojů, zpracování, uložení a následné zprostředkování.

5.6.1 Zpracování dat

Data z průběhu testů jednotlivých scénářů jsou zpracována pomocí InfluxDB a zobrazena prostřednictvím Grafany. Týkající se výstupu nástroje K6 a v případě scénáře 5 pouze konkrétní metriky napřímo vložené po vzoru exportu K6 do jeho databáze v InfluxDB. Data o výkonu kontejnerů jsou získávána pomocí OpenTelemetry metrů z testovaných služeb a pomocí NodeExporteru a Cadvisor z hostitelského systému. Tato data jsou zaslána a spravována ve službě Prometheus a zprostředkována Grafaně formou data source.

5.6.2 Monitorování v reálném čase

Monitorování v reálném čase je klíčovou součástí testování výkonu a škálovatelnosti mikroslužeb. Umožňuje sledovat výkon a škálovatelnost mikroslužeb při běhu testů. Toho je docíleno využitím dashboardů v Grafaně, důkladnou konfigurací a zobrazením metrik, kterých sběr je implementován v rámci mikroslužeb.

Dalším aspektem monitorování v reálném čase je zobrazení výsledků testů v reálném čase. Toho je rovněž docíleno pomocí specifických dashboardů v Grafaně, které integrují data z K6 testovacího nástroje a zaslané do InfluxDb. Díky propojení Grafany s InfluxDb je možné sledovat výsledky testů v reálném čase.

5.6.3 Sběr historických dat

Historická data jsou automaticky ukládána do jednotlivých databází při sběru. Po propagaci telemetrických dat do jednotného collectoru OpenTelemetry jsou data dále

poskytována službám Loki, Tempo a Prometheus. Ty jedna jednotlivá telemetrická data zpracují, zároveň ale slouží jako jejich persistence. Data z výsledků testů K6 jsou ukládána do InfluxDb. Výkonnostní data z hostitelského systému jsou sbírana ze služby NodeExporter přimo do Prometheus v pravidelných intervalech.

III. ANALYTICKÁ ČÁST

6 ANALÝZA APLIKACE

Tato kapitola se zabývá analýzou aplikace z hlediska vývoje, výstupu a výkonu. Využívá k tomu definovanou metodiku a scénáře testování. Výsledky jsou důkladně analyzovány a závěry shrnuty v jednotlivých sekcích.

6.1 Architektura

Výsledná architektura aplikace je založena na mikroslužbách. Splňuje předem definované funkční a nefunkční požadavky. V případě testovaných služeb, zapojuje základní množinu systémových knihoven a knihoven 3. stran. Po straně telemetrie, implementuje sběr a zpracování dat z různých zdrojů. Výsledná data jsou následně zpracována a uložena do databáze, dle druhu dat. Veškeré dostupné zdroje jsou uživatelsky přívětivě vizualizovány v rámci webového aplikace Grafana. Aplikační stack je testovatelný a nasaditelný na všech hlavních platformách (po sestavení se zacílením na vybranou architekturu a využitím variant služeb třetích stran s cílovou architekturou).

6.2 Vývojový proces

Následující sekce popisuje vývojový proces, tak jak se týkal testovaných služeb. Vývojový proces byl založen na experimentaci a snaze využít co nejvíc dostupných knihoven a nástrojů, za cenu nutnosti řešení problémů, případně změny implementace.

6.2.1 JIT

Vývojový proces pro kompilaci služeb JIT se zacílením na .NET runtime probíhal standarntím způsobem. Veškeré dostupné knihovny a nástroje byly plně kompatibilní s JIT kompilací. Nedošlo k žádným nepředpokládaným problémům.

Znatelný rozdíl oproti běžnému vývoji byl výběr technologií, který přihlížel k potencionální kompatibilitě s AOT a tedy řešení, které inherntně vyžadovala funkce rezervované pro využití .NET runtime, byly ihned zavrženy.

Výhody Mezi hlavní výhody se řadí zprostředkování následujícího:

- **Reflexe** - CLR umožňuje využívat reflexi, která umožňuje získat informace o kódu za běhu aplikace. Tímto je umožněno vytvářet aplikace, které jsou schopny měnit své chování za běhu.
- **Dynamické načítání** - CLR umožňuje dynamicky načítat knihovny za běhu aplikace. Tímto je umožněno vytvářet aplikace, které jsou schopny měnit své chování za běhu.

- **Větší bezpečnost** - CLR zajišťuje, že aplikace nemůže přistupovat k paměti, která jí nebyla přidělena. Tímto je zajištěna bezpečnost aplikace a zabráněno chybám, které by mohly vést k pádu aplikace.
- **Správa paměti** - CLR zajišťuje správu paměti pomocí GC. Tímto je zajištěno, že paměť je uvolněna vždy, když ji aplikace již nepotřebuje. Tímto je zabráněno tzv. memory leakům, které by mohly vést k pádu aplikace.
- **Větší přenositelnost** - CLR zajišťuje, že aplikace je spustitelná na všech operačních systémech, na kterých je dostupné běhové prostředí CLR.

Nevýhody Zatímco za nevýhody CLR se dá považovat:

- **Výkonnost** - I když určité optimalizace jsou prováděny pro konkrétní systém a architekturu, výkon CLR je nižší než výkon nativního kódu. Dalším výkonnostním měřítkem je rychlost startu aplikace, která je pro CLR vyšší než v případě nativního kódu.
- **Operační paměť** - CLR využívá více operační paměti, jak pro aplikaci, tak i pro běhové prostředí.
- **Velikost aplikace** - Přítomnost CLR nehraje zásádní roli v případě monolitických aplikací, ale v případě mikroslužeb je nutné CLR přidat ke každé službě. Tímto se zvyšuje velikost jedné aplikační instance.

6.2.2 AOT

Kompilace do nativního kódu probíhala s průběžnými problémy. Podpora ze strany knihoven 3. stran ve spoustě případů neodopvídala deklarovaným možnostem. Vývojový proces byl značně zpomalován nutností řešení problémů, které byly způsobeny nedostatečnou podporou. Experimentace s řešeními často vyústila v nutnost změny implementace, případě v implementaci zcela vlastní.

Výhody Mezi hlavní výhody se řadí zprostředkování následujícího:

- **Výkonnost** - CLR umožňuje využívat reflexi, která umožňuje získat informace o kódu za běhu aplikace. Tímto je umožněno vytvářet aplikace, které jsou schopny měnit své chování za běhu.
- **Paměťová zátěž** - CLR umožňuje dynamicky načítat knihovny za běhu aplikace. Tímto je umožněno vytvářet aplikace, které jsou schopny měnit své chování za běhu.

Nevýhody Zatímco za nevýhody CLR se dá považovat:

- **Absence nástrojů z CLR** - Mnoho nástrojů, které jsou dostupné v CLR, nejsou dostupné v AOT kompilaci. Mezi tyto nástroje patří například reflexe, dynamické načítání knihoven a další.
- **Absence dynamického načítání** - například Assembly.LoadFile.
- **Bez generování kódu za běhu** - například System.Reflection.Emit.
- **Žádné C++/CLI** - např. System.Runtime.InteropServices.WindowsRuntime
- **Windows: absence COM** - např. System.Runtime.InteropServices.ComTypes
- **Vyžaduje trimming (ořezávání)** - má určitá omezení, je však klíčový pro rozumnou velikost výsledného programu
- **Kompilace do jediného souboru**
- **Připojení běhových knihoven** - požadované běhové knihovny jsou součástí výsledného aplikačního souboru. To zvyšuje velikost samoteného programu ve srovnání s aplikacemi závislými na frameworku.
- **System.Linq.Expressions** - výsledný kód používá svou interpretovanou podobu, která je pomalejší než run-time generovaný kompilovaný kód.
- **Kompatibilita knihoven s AOT** - né všechny knihovny runtime jsou plně anotovány tak, aby byly kompatibilní s Native AOT. To znamená, že některá varování v knihovnách runtime nejsou pro koncové vývojáře použitelná.

6.3 Výstup služeb

Samotný proces nativní AOT a JIT kompilace je různě výkonnostně náročný. Při tvorbě obrazu službeb, ale i kompilace je hlavní náročná operace *restore*, která stahuje potřebné závislosti a balíčky pro projekt. Proces kompilace je vysoce závislý na specifickém HW, SW a přítomnosti závislostí. Pro účely testování byly potřebné NuGet balíčky nacachovány v systému. Následující tabulka zobrazuje přehled časové náročnosti kompilace služeb pro oba kompilační cíle. K získání času výstupu bylo využity diagnostického režimu příkazu *dotnet*. Pro AOT byl použit příkaz *dotnet publish -v d -c Release-AOT -r osx-x64*, pro získání výstupu JIT byl použit příkaz *dotnet publish -v d -c Release-JIT -r osx-x64 –self-contained false*.

Velikost samotného výstupního programu je dle očekávání výrazně menší v případě JIT kompilace. To je dáno tím, že výstupní program je závislý na .NET runtime,

Tabulka 6.1 Čas kompilace služeb

	JIT (s)	AOT (s)	AOT % nárůst
SRS	01.99	19.49	979.3
FUS	03.85	30.36	788.5
BPS	02.02	20.74	1026.7
EPS	01.85	20.05	1083.7

který poskytuje dodatečnou obecnou funkcionalitu a vytváří nativní kód včetně generování typů až za běhu aplikace. Následující tabulka zobrazuje velikost služeb pro oba kompilační cíle. Pro vytvoření výstupů na základě JIT byl použit příkaz *dotnet publish -c Release-JIT -r osx-x64 /p:PublishSingleFile=true –self-contained false*, pro vytvoření výstupů AOT byl použit příkaz *dotnet publish -c Release-JIT -r osx-x64 /p:PublishSingleFile=true –self-contained false.*

Tabulka 6.2 Velikost programu služeb

	JIT (MB)	AOT (MB)	AOT % nárůst
SRS	05.70	21.40	375.4
FUS	12.40	28.40	229.0
BPS	06.00	21.80	363.3
EPS	06.00	21.70	361.6

Sestavení obrazu je závislé na přípravu prostředí, vyhodnocení a stažení závislostí, kompilaci a publikování aplikace. Výstupné obrazy jsou založené na linuxovém systému, Alpine s .NET runtime v případě JIT výstupu služby, zredukované Ubuntu v případě nativního AOT výstupu. Z pohledu použitelnosti výsledného obrazu služeb má smysl měřit velikost výstupního obrazu. Následující tabulka zobrazuje velikost obrazu služeb pro oba kompilační cíle. Použitý příkaz je *docker build -t <service>:<tag> -f Dockerfile-<target>* ., kdy *<target>* představuje vybranou kompilační metodu AOT nebo JIT. Před každým sestavením byl obraz a cache smazány. I přes toto opatření není zaručena konzistentní časová náročnost sestavení obrazu.

Tabulka 6.3 Velikost obrazu služeb

	JIT (MB)	AOT (MB)	AOT % zmenšení
SRS	125.63	31.41	75.0
FUS	143.19	38.32	73.2
BPS	126.50	31.40	75.2
EPS	126.45	31.74	74.9

6.3.1 Vývojové prostředí

K vývoji byl použit IDE Rider od společnosti JetBrains. Vyzkoušena byla rovněž i práce ve Visual Studio 2022 Community Edition a Visual Studio Code s doporučenými

rozšířeními od Microsoft. Všechna vývojová prostředí jsou kompatibilní, co se týče procesu kompilace respektive sestavení, jelikož to se odehrává pomocí CLI .NET.

Samotný vývoj s ohledem na práci s direktivami pro různé kompilace byl značně zjednodušen vizualicemi, jenž poskytovala vývojová prostředí Rider a Visual Studio. Obdobně byla v těchto IDE zjednodušena i analýza a hledání chyb díky integraci referencí na kód generovaný na pozadí pro kompatibilitu s AOT. V tomto ohledu Visual Studio Code zaostávalo. S ohledem na aktivní vývoj a podporu, jenž je ze strany Microsoft poskytována podpoře vývoje .NET ve Visual Studio Code (po diskontuaci produktu Visual Studio pro Mac), lze očekávat, že se tato situace v budoucnu změní.

6.3.2 Knihovny třetích stran

Pro zjednodušení procesu vývoje a využití existující funkcionality byly využity knihovny třetích stran. Následující seznam obsahuje knihovny, které byly využity použity v rámci vývoje a zda byly kompatibilní s AOT kompilací.

- **Entity Framework** - Entity framework se pyšní vysokou kompatibilitou s AOT kompilací. V rámci vývoje nebyly zaznamenány problémy, avšak následné testování se ukázalo problematické. EF jakožto plnohodnotný ORM framework stopuje stav objektu a jeho změny. Toto chování bohužel vyžaduje dynamické generování kódu, což je v rozporu s možnostmi AOT kompilovaného kódu. Vypnutí této funkcionality je pouze částečné, neb EF stále vyžaduje reflexi při vkládání nových entit do databáze.

- **Fluent Migrator** - Fluent Migrator je knihovna, která umožňuje verzování databáze pomocí kódu. V rámci testování bylo zjištěno, že knihovna využívá reflexi pro načítání migrací. Toto chování je v rozporu s AOT kompilací a výsledkem je chyba při spuštění migrace. Problém byl vyřešen vytvořením vlastního minimalistického migrátoru, který nepoužívá reflexi.

- **Grpc** - Vytváření rozhrání a modelů pro gRPC komunikaci vyžadovalo využití přístupu model first. Tento přístup využívá generátorů pro tvorbu kódu, definijucího kódového rozhraní pro .NET. Tímto je dosaženo vygenerování veškerého potřebného kódu v době kompilace a je zajištěna kompatibila s AOT. Pro definici modelu code first ovšem kombatibila s AOT není zajištěna.

- **Párování konfigurace** - V rámci systémové .NET knihovny je umožněno volání API, jenž načte data ze sjednocení stavu proměnných prostředi a konfiguračního souboru. Součástí API je volání metody mapující tuto konfiguraci na předem definovaný objekt. Toto chování dle dostupných informací není v rozporu s AOT

kompilací a volání relevantního kódu neprodukuje AOT warning. Z testování však vyplynulo, že mapování konfigurace ne objekt bylo problematické a neprobíhalo správně. Z toho důvodu je v případě AOT kompilace za pomocí deriktivy použité přímé načtení jednotlivých hodnot z konfigurace, dle stromomvého klíče.

6.4 Analýza testování

Následující sekce se zabývá analýzou testovacích scénářů a výsledků testování. Testování bylo provedeno na základě předem definované metodiky. Podkladem testů byly definované scénáře, které byly vytvořeny s ohledem na funkční a nefunkční požadavky. Při testování byl nezávisle na spuštěný test zaznaménáván stav hostitelského systému s ohledem na spuštěné kontejnery a využití systémových prostředků. Samotné služby využívaly předem definované metry ve frameworku ASP.NET pro dodatečnou diagnostiku a monitorování. Výsledky testování byly zaznamenány a analyzovány.

6.4.1 Scénář 1 - Výkonnost komunikace

První scénář se zabíral jednoduchou funkcionalitou dotazu na healthcheck endpoint a meřením výkonu kestrel serveru u odpovědí na požadavky skrze REST API. Testování přineslo rozdílné výkonostní výsledky mezi JIT a AOT kompilací. Druhá zmíněna si vyžádaval větší systémové zdroje pro obsloužení požadavků a obecně byl pomalejší ve zpracování požadvaků. Čistá rychlost zpracování požadavku a odpovědi není v mnoha případech kritickým faktorem. Avšak v případě velkého množství požadavků, může být rozdíl v řádech milisekund zásadní.

TODO: Foto dashboard + tabulka výsledků + stručné vysvětlení

6.4.2 Scénář 2 - Přístup k perzistenci

Scénář se zabýval výkonností přístupu k persistenci, respektive zachystením reálného scénáře, kdy jsou data získávána a ukládána do databáze. Faktorem byla jak samotná rychlost služby v ohledu komunikace a serializace dat, tak rychlost zpracování požadavku databází. Ve výsledku je znatelný rozdíl rychlosti odpovědi v porovnání s prvním scénářem. Databáze vnesla další režii a zmenšila procentuální rozdíl mezi AOT a JIT.

TODO: Foto dashboard + tabulka výsledků + stručné vysvětlení

6.4.3 Scénář 3 - Výpočetní zátěž

Za účelem zjištění výkonnosti služeb, jejich potencionálně odlišné využití systémového API byl otestován scénář výpočetní zátěže. Na jednotlivé služby byly vysílány požadavky na výpočet 40-tého Fibonacciho čísla rekurzivní metodou. Výsledky testování ukázaly konsistnetně vyšší počet zpracovaných požadavků v případě AOT kompilace.

TODO: Foto dashboard + tabulka výsledků + stručné vysvětlení

6.4.4 Scénář 4 - Vzájemná komunikace služeb

Komplexnější situace pro aplikaci byla simulována ve čtvrtém scénáři. Zde na základě požadavku na EPS byla vyvolána událost do RabbitMQ, načež byla zpracována službou BPS. Ta na jejím základě stáhla patřičný záznam pomocí RPC z FUS a provedla simulaci zpracování dat výpočtem Fibonacciho čísla. Situace simulovala kombinaci synchronní a asynchronní komunikace mezi službami doplněnou o výpočetní zátěž. Výsledky přiblížily služby v obou kompilačních režimech a ukázaly pohled bližší reálnému nasazení, kdy čistě výkkonostní rozdíly kompilací nehrají tak zásadní roli.

TODO: Foto dashboard + tabulka výsledků

6.4.5 Scénář 5 - Rychlost odpovědi služby po startu

Simulaci serveless nasazení byla vyvolána v tomto scénáři. Jednotlivé varianty služby SRS byly v rámci testu zpuštěny, kontrolovány než se dostaly do stavu *healthy* a následně nad nimi zavolán dotaz na generovaná data. Výsledky ukázaly, že služba kompilované nativním AOT způsobem byly mnohem rychleji dostupné a odpovídaly na požadavky dříve, než služba kompilované do .NET runtime.

TODO: Foto dashboard + tabulka výsledků

6.5 Závěr analýzy

Na základě výsledků vývoje, výstupu a testování služeb lze odpovědět na definované hypotézy následujícím způsobem:

- **Hypotéza 1** - Hypotéza, že vývoj služeb s jak AOT, tak JIT kompilací je v rámci podporované funkcionality systémových knihoven a ASP.NET možný s podobným API se ukázal jako ne zcela pravdivý. Při vývoji nastaly komplikace se serializací konfigurace, na které bylo nutné reagovat využitím odlišného API. Zároveň tento způsob serializace nebyl kompilátorem označen jako potencionálně problematický. Další problémy nastaly s využitím Entity Framework. Tento ORM využívá pro provádění operací nad databází tzv. tracking, který zaznamená změny nad aplikačními objekty a podle nich tvoří výsledné databázové operace. Vypnutím trackingu bylo umožněno se na datové entity dotázat a aktualizovat je. Operace vložení nové entity však bez trackingu nebyla možná. Pro knihovny 3. stran lze obecně říci, že podpora AOT kompilace není vždy úplně zřejmá a i v situacích kdy AOT varování jsou implementovány, lze očekávat chybné chování.

- **Hypotéza 2** - Výsledky ukazují, že služby napsané v nativním kódu se výrazněji rychleji spouští jak na hostitelských systémech, tak ve virtualizovaném prostředí. Zároveň binární velikosti samotných aplikací jsou mnohonásobně větší, než je tomu u služeb vyžadující .NET runtime. To je ovšem kompenzováno při virtualizovaném spuštění, kdy obraz služby pro vytvoření plnohodnotného kontejneru vyžaduje mnohem méně závislotí z hlediska paměti. Výsledné obrazy jsou tedy menší a rychleji spustitelné. Hypotéza byla potvrzena.

- **Hypotéza 3** - Na základě dostupných metrik bylo potvrzeno, že obecně služby kompilované do nativního kódu poskytují o něco horší výkon a jsou paměťově náročnější než služby kompilované do .NET runtime. Tento fakt je způsoben optimalizacemi runtime za běhu, které jsou v případě nativního kódu ztraceny. Dynamická kompilace a generování typů za běhu je náročnější než statická kompilace a generování typů při kompilaci, ale pouze při prvním použití. Rozdíl však není natolik patrný a je rychle zastíněn režií na doménovou logiku, výpočet či přístup k vzdáleným datům. Výsledky testování potvrzují hypotézu.

7 DOPORUČENÍ PRO POUŽITÍ AOT KOMPILACE V DOTNET

Služby kompilované do nativního AOT kódu přináší specifické výkonnostní výhody za cenu kompatibility API. Vývoj kódu je s ohledem na zažité postupy a praktiky v .NET nestandartní. Využití interceptorů a generátorů je odebrána část inciativy z rukou vývojáře a vytváří se na pozadí kompilace v .NET další úroveň abstrakce. Podpora knihoven a frameworků třetích stran je omezena a nelze se spolehnout na jejich plnou funkčnost. Tím připadá na vývojáře zodpovědnost za implementaci vlastních řešení, která by jinak byla dostupná.

Většina výhod, jenž z .NET plyne souvisí s možnostmi jeho runtime prostředí. Nativní AOT kompilace má smysl ve specifických případech, jenž plynou z nutnosti rychlosti spuštění a velikosti výstupu aplikace (s přihlednutím k velikosti .NET runtime). Případy konkurenční výhody pro AOT kompilaci staví na předpokladu a tím je zájem či potřeba mít zdrojové kódy v .NET, respektive jazyce C#. Při rozmanitém technologickém přístupu, kdy je vývojář, respektive zapojený tým schopen přijmout jiný jazyk a framework, jsou výhody AOT kompilace ztraceny, zatímco nedostatky jsou zvýrazněny. Tento předpoklad je relativně v rozporu s požadavky na poskytování cloudových služeb, kdy je očekáváno silné technické a vědomostní zázemí a flexibilní přístup k technologiím.

Nativní AOT kompilace má oproti JIT kompilaci nesporné výhody za splnění uričtých podmínek na požadavky vůči nasazení, kódu a vývojového týmu. Zaplňuje specifickou díru v portfoliu technologií platformy .NET, která je klíčová pro kompletní řešení cloudové platformy pouze s použitím této platformy. Vývojáři, kteří se rozhodnou pro AOT kompilaci, by měli být obeznámeni s těmito specifiky a měli by být schopni je zohlednit v návrhu a implementaci řešení.

8 ROZŠÍŘENÍ A BUDOUCÍ PRÁCE

V návaznosti na platformu, která v práci vznikla v rámci výkonnostního testování služeb, je možné doplnit implementaci dalších služeb, případně rozšířit stávající. Podle vzoru současného řešení lze dodat další funkcionalitu, nastavit další zdroje telemetrie, případně rozšířit možnosti vizualizace dat.

Z pohledu uživatelské přívětivosti se nabízí rozšíření o webovou aplikaci využívající princip Docker outside of Docker (DooD). Tímto by bylo možné zjednodušit spouštění konkrétních testovacích scénářů v GUI. V rámci webové aplikace by bylo možné nastavit parametry testování, spustit konkrétní test a prokliknout se odkazem na relevantní dashboard v Grafaně.

S ohledem na citlivé data a přístupy, které aplikace zprostředkovává, se nabízí rozšíření o autentizaci a autorizaci v případě vystavení stacku v síti. Jelikož grafické rozhraní aplikace je založeno na aplikaci Grafana, jež je schopna připojit se k externím zprostředkovatelům autentizace, bylo by vhodné zapojit službu jako Keycloak pro sjednocení autentifikace napříč stackem.

ZÁVĚR

V rámci diplomové práce byly analyzovány kompilační režimy JIT a nativní AOT na platformě .NET. První částí byla rešerše, ve které byly popsány základní principy fungování platformy .NET, jejich kompilačních režimů a cílů kompilace. Následně byla popsána architektura microservice, která slouží jako primární zacílení nativních AOT aplikací a která poskytuje vzor pro testovací nasazení. V neposlední řadě byla popsána problematika testování, telemetrie a monitorovacích řešení.

Praktická část se zabývala vývojem testovacích služeb a testovací platformy.Dále byly popsány nástroje, které umožňují vytváření nativních AOT aplikací na platformě .NET. V rámci rešerše byly také popsány nástroje, které umožňují měření výkonu aplikací.

V analytické části byly výsledky praktické části popsány a vyhodnoceny. Zhodnocení vývoje probíhalo ve třech režimech: analýza vývoje, výstupu a výkonu.

Výsledkem práce je komplexní analýza použití kompilačních režimů JIT a nativní AOT. Vývojový proces při kompilaci do nativního AOT kódu se ukázal nepřívětivý. Primárně podpora knihoven 3. stran a princip interceptorů a generátorů má za vinu subjektivně neintuitivní proces debugování kódu. Samotný programový výstup vyšel dle očekávání. V porovnání byly obrazy nativních služeb výrazně paměťově efektivnější. Výsledky testování ukázaly, že na platformě .NET nativní AOT aplikace mají obecně srovnatelný výkon jako aplikace v režimu JIT. Rozdíl je znatelný v situacích, kdy je nutno využít velké množství instancí stejné služby (plyne z velikosti obrazu) a v situacích, kdy je pro systém rozhodující rychlost zpracování služby včetně spuštění (Serverless platformy). Výsledky výkonnostního testování byly zaznamenány a zpracovány do dashboardů a grafů, které jsou připraveny v uživatelské rozhraní platformy Grafana.

Dále byla vytvořena sada testovacích služeb, které slouží jako ukázka možností platformy. V neposlední řadě pro účely analýzy byla vytvořena testovací platforma, která umožňuje vytváření a nasazování testovacích služeb v kompilačním režimu JIT a nativní AOT.

SEZNAM POUŽITÉ LITERATURY

[1] PRICE, Mark J. *Apps and Services with .NET 8. Second Edition.* Packt Publishing, 2023. ISBN 978-1837637133.

[2] DANYLKO, Jonathan R. *ASP.NET 8 Best Practices.* Packt Publishing, 2023. ISBN 978-1-83763-713-3.

[3] KOKOSA, K. *Pro .NET Memory Management: For Better Code, Performance, and Scalability.* For Professionals By Professionals. Apress, New York, 2018. ISBN 978-1484240267.

[4] RICHTER, J. *CLR via C#: The Common Language Runtime for .NET Programmers.* 4th ed. Microsoft Press, Redmond, Wash., 2012. ISBN 978-0735667457.

[5] PFLUG, Kenny. *Native AOT with ASP.NET Core - Overview* [online]. 2023 [cit. 2024-02-23]. Available from: https://www.thinktecture.com/en/net/native-aot-with-asp-net-core-overview/

[6] MARTIN, Robert C. *Clean architecture: a craftsman's guide to software structure and design.* Robert C. Martin series. London, England: Prentice Hall, [2018]. ISBN 978-0134494166.

[7] WILLIAMS, Trevoir. *Microservices Design Patterns in .NET.* Packt Publishing, 2023. ISBN 978-1-80461-030-5.

[8] RICHARDSON, C. *Microservices Patterns: With Examples in Java.* O'Reilly Media, Sebastopol, Calif., 2018. ISBN 978-1617294549.

[9] ALLS, Jason. *Clean Code with C#.* Second edition. Packt Publishing, 2023. ISBN 978-1-83763-519-1.

[10] ESPOSIO, Dino. *Microservices Design Patterns in .NET.* Pearson Education, 2024. ISBN 978-0-13-820336-8.

[11] MARCOTTE, Carl-Hugo. *Architecting ASP.NET Core Applications.* Third Edition. Packt Publishing, 2024. ISBN 9781805123385.

[12] NICKOLOFF, J.; KUENZIL, S. *Docker in Action.* 2nd ed. Manning Publications, Greenwich, CT, 2019. ISBN 978-1617294761.

[13] SALITURO, Eric. *Learn Grafana 10.x.* Second Edition. Packt Publishing, 2023. ISBN 978-1-80323-108-2.

[14] CHAPMAN, Rob a Peter HOLMES. *Observability with Grafana*. Packt Publishing, 2023. ISBN 978-1-80324-800-4.

[15] MOLKOVA, Liudmila a Sergey KANZHELEV. *Modern Distributed Tracing in .NET*. Packt Publishing, 2023. ISBN 978-1-83763-613-6.

[16] AKINSHIN, Andrey. *Pro .NET Benchmarking*. Apress Berkeley, CA, 2019. ISBN 978-1-4842-4941-3.

[17] GARRISON, J.; NOVA, K. *Cloud Native Infrastructure: Designing, Building, and Running Scalable Microservices Applications*. 1st ed. O'Reilly Media, Sebastopol, Calif., 2017. ISBN 978-1491984307.

[18] GARVERICK, Joshua a Omar Dean MCIVER. Implementing Event-Driven Microservices Architecture in .NET 7. Packt Publishing, 2023. ISBN 978-1-80323-278-2.

[19] LIBERY, Jesse a Rodrigo JUAREZ. *.NET MAUI for C# Developers*. Packt Publishing, 2023. ISBN 978-1-83763-169-8.

[20] .NET 7 Preview 3 Is All About Native AOT. RAMEL, David. Visual Studio Magazine [online]. 2022 [cit. 2024-03-19]. Dostupné z: https://visualstudiomagazine.com/articles/2022/04/15/net-7-preview-3.aspx

[21] GAMMELGAARD, C. H. *Microservices for .NET Developers: A Hands-On Guide to Building and Deploying Microservices-Based Applications Using .NET Core*. 2nd ed. Apress, 2021, ISBN 978-1617297922.

[22] SAZANAVETS, Fiodar. *Microservice Communication in .NET Using gRPC*. Packt Publishing, 2022. ISBN 978-1-80323-643-8.

[23] LOCK, A. *ASP.NET Core in Action*. 2nd ed. Manning Publications, Greenwich, CT, 2021. ISBN 978-1617298301.

[24] PFLB, INC. *User Manual for k6, an Open Source Tool for Load Testing*. PFLB [online]. 2021 [cit. 2024-04-01]. Dostupné z: https://pflb.us/blog/k6-user-manual/

[25] MICROSOFT CORPORATION. *.NET Documentation* [online]. [cit. 2024-03-18]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/

[26] MICROSOFT CORPORATION. *ASP.NET Documentation* [online]. [cit. 2024-03-18]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/?view=aspnetcore-8.0

[27] DOCKER INC. *Docker Docs* [online]. 2013 [cit. 2024-03-13]. Dostupné z: https://docs.docker.com

[28] RAINTANK, INC. *Grafana Labs - Technical Documentation* [online]. [cit. 2024-03-22]. Dostupné z: https://grafana.com/docs/

[29] F5, INC. *NGINX Product Documentation* [online]. [cit. 2024-04-12]. Dostupné z: https://docs.nginx.com

[30] THE POSTGRESQL GLOBAL DEVELOPMENT GROUP. *PostgreSQL Documentation* [online]. [cit. 2024-04-14]. Dostupné z: https://www.postgresql.org/docs/

[31] INFLUXDATA, INC. *InfluxDB v1 Documentation* [online]. [cit. 2024-04-14]. Dostupné z: https://docs.influxdata.com/influxdb/v1/

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

CPU	Central Processing Unit
RAM	Random Access Memory
JIT	Just in Time
AOT	Ahead of Time
CLI	Command Line Interface
CLR	Common Language Runtime
IL	Intermediate Language
API	Application Programming Interface
RPC	Remote Procedure Call
SDK	Software Development Kit
IDE	Integrated Development Environment
GUI	Graphical User Interface

SEZNAM OBRÁZKŮ

SEZNAM TABULEK