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

Text abstraktu česky

Klíčová slova: Přehled klíčových slov

ABSTRACT

Text of the abstract

Keywords: Some keywords

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 a běh 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 kompilace do nativního kódu, který je spouštěn přímo na architektuře HW. Tato funkce přichází do období masivní migrace řešení do cloudu a implementace FaaS, kdy zpoplatněn pouze skutečná doba běhu systému + režie. A právě v prostředí cloudu mají nastávat situace, kdy bude využití nativního kódu výhodnější. V kterých případech však opravdu takto napsaný program exceluje či selhává a lze kvantifikacovat rozdíly mezi JIT a AOT kompilací?

Tato práce se zabývá porovnáním výkonu a chování JIT a AOT kompilace 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* (dále jen CLR), runtime prostředí zodpovídající za běh aplikací. Druhým nástrojem je *dotnet CLI*, konzolový nástroj-rozhraní, zodpovědné za interakci s dílčími nástroji v platformě. [2]

1.1 Historie

Využití runtime prostředí, respektive v originální podobě virtuálního stroje, má historický původ. V dřívějších dobách byly 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 finálnímu kroku a to interpretaci (JIT kompilaci) Bytecode do cílové architektury 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. KV roce 2014 byla vydána první multiplatformní verze .NETu. Byl vydán .NET Core, který umožňoval spouštět kód v jazyce C# na operačních systémech Windows, Linux a macOS. [2]

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** - běhové prostředí pro programy kompilované do IL

- **.NET CLI** (konzolové rozhraní) - konzolové rozhraní pro interakci s nástroji .NET
- **MSBuild** - buildovací engine pro kompilaci, testování a balíčkování aplikací
- **.NET SDK nástroje** - soubor nástrojů pro testování, debuggování a migraci .NET aplikací
- **Roslyn** - kompilační nástroj
- **NuGet** - balíčkovací manager

Tvorbu aplikace zprostředkovává sada nástrojů a knihoven .NET SDK. Tyto nástroje umožňují vývojářům vytvářet, sestavovat a spouštět aplikace v podporovaných programovacích jazycích. Nástroje .NET SDK poskytují sadu nástrojů pro vývoj aplikací, včetně nástrojů pro sestavení, testování a publikaci. Nástroje .NET SDK jsou dostupné pro operační systémy Windows, Linux a macOS a umožňují vývoj aplikací pro různé platformy. Vývojáři mohou vytvářet aplikace pro různé platformy pomocí jednoho kódu a sdílet kód mezi různými platformami. [2]

1.3 Frameworky a technologie

Platforma .NET poskytuje mnoho frameworků a technologií pro vývoj aplikací.

1.3.1 Foundtation frameworky

.NET .NET je hlavní framework platformy .NET. Poskytuje sadu nástrojů a knihoven pro vývoj aplikací v jazyce C# a .NET. .NET obsahuje sadu tříd a metod, které mohou být použity při vývoji aplikací. .NET poskytuje sadu nástrojů pro vývoj aplikací, včetně nástrojů pro sestavení, testování a publikaci. .NET umožňuje vývoj aplikací pro platformy Windows, Linux a macOS pomocí jednoho kódu. Vývojáři mohou vytvářet aplikace pro různé platformy pomocí jednoho kódu a sdílet kód mezi různými platformami.

Mono Mono je open-source implementace platformy .NET. Poskytuje sadu nástrojů a knihoven pro vývoj aplikací v jazyce C# a .NET. Mono umožňuje vývoj aplikací pro platformy Windows, Linux a macOS pomocí jednoho kódu. Vývojáři mohou vytvářet aplikace pro různé platformy pomocí jednoho kódu a sdílet kód mezi různými platformami. Mono je dostupný jako open-source projekt a je vyvíjen komunitou vývojářů. Mono poskytuje sadu nástrojů pro vývoj aplikací, včetně nástrojů pro sestavení, testování a publikaci. Mono umožňuje vývoj aplikací pro platformy Windows, Linux a

macOS pomocí jednoho kódu. Vývojáři mohou vytvářet aplikace pro různé platformy pomocí jednoho kódu a sdílet kód mezi různými platformami.

1.3.2 Specializované frameworky

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 pro rychlý vývojový model prostřednictvím rozhraní přetahování a serverových ovládacích prvků, tak architekturu MVC (Model-View-Controller), která podporuje čisté oddělení problémů, testovatelnost a výkonné směrování URL pro SEO. -přátelské webové aplikace. S uvedením ASP.NET Core bylo přepracováno pro cloudovou škálovatelnost, vývoj napříč platformami a vysoký výkon, takže je ideální pro vývoj moderních webových aplikací, které lze spustit na Linuxu, Windows a 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 všeho od malých webů po složité webové platformy.

MAUI je moderní specializovaný framework pro vývoj aplikací napříč platformami v rámci ekosystému .NET. Umožňuje vývojářům vytvářet aplikace pro Android, iOS, macOS a Windows z jediné kódové základny, která zahrnuje to nejlepší 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. Vývojářům, kteří chtějí využít sílu .NET napříč mobilními a desktopovými platformami, nabízí .NET MAUI komplexní a moderní sadu nástrojů navrženou tak, aby vyhovovala dnešním vývojovým výzvám a zároveň zajistila přenositelnost a udržovatelnost kódu.

Předchůdcem MAUI je framework Xamarin, který sloužil pro vytváření mobilních aplikací na platformě .NET.

Blazor je specializovaný rámec v rámci ekosystému .NET, který přináší revoluci ve vývoji webu tím, že umožňuje vývojářům vytvářet interaktivní webová uživatelská rozhraní pomocí C# namísto JavaScriptu. Blazor může běžet na serveru (Blazor Server), kde zpracovává interakce s uživatelským rozhraním přes připojení SignalR, nebo v prohlížeči pomocí WebAssembly (Blazor WebAssembly) ke spouštění kódu C# přímo ve

webovém prohlížeči vedle tradičních webových technologií, jako jsou HTML a CSS. Tento inovativní přístup umožňuje vývojářům využít jejich dovednosti .NET pro komplexní vývoj webových aplikací a vytvářet bohaté webové aplikace na straně klienta s .NET spuštěným v prohlížeči. Architektura Blazor založená na komponentách usnadňuje opětovné použití komponent uživatelského rozhraní a podporuje modulární vývojový přístup. Vývojářům .NET, kteří chtějí vytvářet moderní webové aplikace a přitom zůstat v ekosystému .NET, nabízí Blazor přesvědčivou cestu vpřed, která kombinuje produktivitu a sílu C# s flexibilitou a dosahem webových technologií.

1.3.3 Knihovny

Knihovny představují soubor funkcí a nástrojů, které mohou být použity při vývoji aplikací.

Microsoft knihovny 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.

- **System.IO** - knihovna pro práci se souborovým systémem
- **System.Net** - knihovna pro práci se sítí
- **System.Data** - knihovna pro práci s databázemi
- **System.Windows.Forms** - knihovna pro tvorbu grafického rozhraní
- **System.Drawing** - knihovna pro práci s grafikou

Knihovny třetích stran 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 funkcionality.

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 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. IDE poskytuje také nástroje pro správu projektů, jako jsou sestavení, testování a publikace. Umožňuje provádět různorodé operace na 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, JetBrains Rider a další.

1.4.2 Balíčky

Pro jednoduchou distribuci knihoven, jak systémových tak třetích stran, je využíván balíčkovací manager 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 přidatelné 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/`.

1.5 Jazyky a struktura aplikace

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

1.5.1 Jazyky

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# je 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 grafických uživatelských rozhraní (GUI). Tento přístup zpřístupnil programování širšímu okruhu uživatelů a kladl důraz na rychlý vývoj aplikací (RAD).

1.5.2 Aplikační struktura

Základním stavebním prvkem aplikace v .NET 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 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** - konfigurační soubor, který obsahuje nastavení aplikace
- **launchsettings.json** - konfigurační soubor, který obsahuje nastavení pro spuštění aplikace
- **Directory.Build.props** - konfigurační soubor, který obsahuje globální nastavení atributů pro všechny projekty v solution

- **Directory.Build.targets** - konfigurační soubor, který obsahuje globální nastavení cílů kompilace pro všechny projekty v solution
- **NuGet.config** - konfigurační soubor, který obsahuje nastavení pro balíčkovací manager 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 2 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:

PC Zacílení na platformy PC nabízí 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 poskytuje .NET možnost kompilace Ahead-Of-Time (AOT) prostřednictvím nástrojů jako .NET Native.

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 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*.

Výstup 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).

1.6.3 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.

Výstup 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.

1.6.4 Popis procesu

1. Analýza zdrojového kódu, provádí kontrolu syntaxe
2. Kontrola syntaxe
3. Transformace vybrané syntaxe - například pro C# je to transformace "High level C#"na "Low level C#"
4. Generování pre-build kódu
5. Překlad kódu

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.

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

- Správa paměti - spravuje alokaci paměti, obsluhuje GC
- Bezpečnost

- Zpracování vyjímek - obsluhuje zpracování chyb/vyjímek v programu
- Generování typů
- Reflexe

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.

JIT kompilace Při JIT kompilaci v rámci CLR dochází ke kompilaci kódu do nativní podoby těsně před spuštěním kódu. Kompilace veškerého kódu aplikace JIT umožňuje optimalizovat běh programu současnému stavu systému. Za běhu je prováděna inspekce kódu, dochází ke kontrole validity, typování a adresace IL kódu. Kompilovány jsou pouze ty části kódu, jenž jsou relevantní pro aktuální stav programu.

R2R kompilace 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.

1.7.2 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 dotnet

Následující část popisuje obecnou koncepci a strukturu projektu aplikace v dotnet. 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 Jedná se

Deklarace unmanaged rozhraní

Trimming

1.8.4 Přehled podpory

Funkcionalita Následující přehled představuje rozsah funkcionality implementované v rámci .NET frameworku 8.0, konkrétně APS.NET k datu zvěřejnění práce.

- **REST minimal API**
- **gRPC API**
- **JWT Authentication**
- **CORS**
- **HealthChecks**
- **HttpLogging**
- **Localization**
- **OutputCaching**
- **RateLimiting**
- **RequestDecompression**

- **ResponseCaching**
- **ResponseCompression**
- **Rewrite**
- **StaticFiles**
- **WebSockets**

Cíle kompilace .NET poskytuje podporu pro kompilaci zdrojového kódu v režimu AoT pouze pro určité operační systémy:

- **Windows** - plná podpora
- **Linux** - plná podpora
- **macOS** - plná podpora
- **Android** - částečná podpora
- **iOS** - částečná podpora
- **WebAssembly** - částečná podpora

2 MICROSERVICE ARCHITEKTURA

Při vývoji softwaru je možné využít z několika architektur. Jednou z těchto architektur je monolitická architektura. V monolitické architektuře je celá aplikace rozdělena do několika vrstev, které jsou využívány k oddělení logiky aplikace.

Oproti tomu microservice architektura 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. [3]

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

2.2.1 Mikroslužby

Serverless je model vývoje aplikací, který umožňuje vývojářům psát a nasazovat kód bez starostí o infrastrukturu. Tento model je založen na konceptu funkcí jako služby (FaaS), které jsou jednotlivé kusy kódu, které jsou spouštěny na základě událostí. Serverless a mikroslužby se často používají společně, protože oba modely podporují škálovatelnost, agilitu a odolnost. Serverless může být výhodný pro mikroslužby, které jsou založeny na událostech, jako jsou zpracování obrázků, zpracování zpráv nebo plánování úloh.

2.2.2 Virtualizace a kontejnerizace

Virtualizace a kontejnerizace jsou klíčové technologie, které umožňují architekturu mikroslužeb. Virtualizace umožňuje provozovat více operačních systémů na jednom fy-

zickém 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 se staly nástroje jako Docker, které nabízejí ekosystém pro vývoj, odesílání 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.

2.3 Vlastnosti

2.3.1 Komunikace

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

gRPC

Message brokers

2.3.2 Škálování

Architektura mikroslužeb zvyšuje škálovatelnost. Služby lze škálovat nezávisle, což umožňuje efektivnější využití zdrojů a zlepšuje schopnost systému zvládat velké ob-

jemy požadavků. Běžně se používá horizontální škálování (přidávání dalších instancí služby), které usnadňují nástroje pro kontejnerizaci a orchestraci.

2.3.3 Odolnost

Robustnosti mikroslužeb je dosaženo pomocí strategií, jako jsou přerušovače, záložní řešení a opakované pokusy, 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.

2.3.4 Vývoj

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.

2.4 Testování

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.

2.5 Výhody a nevýhody

2.5.1 Výhody

Zvýšená agilita 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.

2.5.2 Nevýhody

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 hranicemi vyžaduje pečlivou koordinaci a vzory jako Saga.

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 Závěr

Architektura mikroslužeb je metoda vývoje softwarových systémů, které jsou rozděleny do malých, nezávislých služeb komunikujících prostřednictvím přesně definovaných rozhraní API. Tyto služby jsou vysoce udržovatelné a testovatelné, volně provázané, nezávisle nasaditelné a organizované podle obchodních schopností. Tento přístup k architektuře umožňuje organizacím dosáhnout větší agility a škálování jejich aplikací.

3 MONITOROVÁNÍ APLIKACE

Monitorování aplikací je klíčovým aspektem moderního vývoje a provozu softwaru, který týmům 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 Druhy dat

Pro efektivní monitorování aplikace je nezbytné porozumět různým typům dat a informací, které lze shromažďovat:

3.1.1 Logy

Protokoly jsou záznamy o událostech, ke kterým dochází v rámci aplikace nebo jejího provozního prostředí. Poskytují podrobné, časově označené záznamy o činnostech, chybách a transakcích, které mohou vývojáři a provozní týmy použít k řešení problémů, pochopení chování aplikace a zlepšení spolehlivosti systému.

3.1.2 Traces

Trasy se používají ke sledování toku požadavků v aplikaci, zejména v distribuovaných systémech, kde jedna transakce může zahrnovat více služeb nebo komponent. Sledování pomáhá identifikovat úzká místa, pochopit problémy s latencí a zlepšit celkový výkon aplikací.

3.1.3 Metriky

Metriky jsou kvantitativní údaje, které poskytují přehled o výkonu a stavu aplikace. Mezi běžné metriky patří doba odezvy, využití systémových prostředků (CPU, paměť, diskové I/O), chybovost a propustnost. Sledování těchto metrik pomáhá při proaktivním ladění výkonu a plánování kapacity.

3.2 Sběr dat

Efektivita monitorování aplikací do značné míry závisí na schopnosti efektivně shromažďovat relevantní data.

3.2.1 Collectory

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.

3.3 Vizualizace dat

Vizualizace dat je klíčovým aspektem monitorování aplikací, který umožňuje rychle porozumět stavu a chování aplikací. Vizualizace může zahrnovat různé typy grafů, tabulek, dashboardů a dalších nástrojů, které umožňují zobrazit data v uživatelsky přívětivé podobě. Vizualizace dat umožňuje týmům identifikovat vzory, problémy a příležitosti, které by jinak mohly zůstat skryty v datech.

3.4 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.

3.4.1 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.

3.4.2 Nasazení služeb pro správu a kolekci dat

Nasazení služeb pro správu a kolekci dat je zajištěno pomocí nástrojů, které jsou schopny zprostředkovat sběr dat z různých zdrojů a zároveň zajišťují jejich zpracování a zobrazení. Tímto je zajištěno, že data jsou zpracována a zobrazena v reálném čase.

3.4.3 Vizualizace dat

Vizualizace dat je zajištěna pomocí nástrojů, které jsou schopny zobrazit data v uživatelsky přívětivé podobě. Tímto je zajištěno, že data jsou zobrazena v reálném čase a jsou přehledná a srozumitelná.

3.5 Konfigurace

Konfigurace monitorování je zajištěna pomocí konfiguračních souborů, které definují chování monitorovacích nástrojů a sběr dat. Ovlivnit chování monitorovacího systému může být provedeno jak na straně monitorovacích nástrojů, respektive služeb, tak i na straně aplikací a služeb, které jsou monitorovány.

3.6 Závěr

Monitorování aplikací je nezbytným nástrojem pro vývoj a provoz moderních softwarových systémů. Zahrnuje shromažďování, analýzu a interpretaci různých typů dat a informací, které umožňují týmům sledovat výkon, stav a chování aplikací v reálném čase. Tímto je zajištěno, že aplikace jsou spolehlivé, výkonné a efektivní.

II. PRAKTICKÁ ČÁST

4 TVORBA TECH 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

Aplikace pro svůj účel nezávislého testování výkonu a škálovatelnosti mikroslužeb vyžaduje několik požadavků, které jsou rozděleny na funkční a nefunkční.

4.1.1 Funkční požadavky

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, protokolů a tras.

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.

4.1.2 Nefunkční požadavky

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 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.

4.3.1 Organizace a správa zdrojů

Git 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.

Struktura 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.

GitHub 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.

4.3.2 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.

4.3.3 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.

4.3.4 Persistenční vrstva

Persistenční vrsta zprostředovává dlouhodobé uchování dat. Z důvodů požadavků na persistenci byly vybrány následující.

Postgres Open-source relační databáze, která umožňuje ukládat a spravovat data. Postgres je schopná pracovat s relačními daty, které jsou uloženy v tabulkách. Poskytuje základní klientský balíček pro .NET, který umožňuje komunikaci s databází. Tento balíček je kompatibilní s Native AoT kompilací.

InfluxDB Open-source timeseries databáze, která umožňuje ukládat a spravovat časové řady. Využití InfluxDb je pragramtické z důvodu nativní podbory Influxdb napojení z nástroje K6 pro export testovacích dat.

4.3.5 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.

4.3.6 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.

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 collector 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.

4.3.7 Testovací nástroje

K6 Nástroj pro výkonové testování, který umožňuje vývojářům testovat výkon svých aplikací. K6 umožňuje vývojářům vytvářet a spouštět testy, které simuluji reálné uživatelské scénáře. Tímto je zajištěno, že aplikace je schopna zvládnout požadavky uživatelů. K6 je nástroj, který je možné využít pro testování mikroslužeb, protože umožňuje vývojářům vytvářet testy, které simuluji reálné uživatelské scénáře.

4.3.8 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.

4.4 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.

4.4.1 Architektura

Pro implementaci požadované funkcionality bylo zvoleno následující rozdělení zodpovědnosti služeb:

- **SRS - Signal reading service** - služba, která simuluje čtecí zařízení, které čte data ze zdroje a poskytuje je ostatním službám. Poskytuje REST API rozhraní.
- **FUS - File Upload Service** - služba, která simuluje zapisovací zařízení, zapisuje nebo čte data do persistentního úložiště. 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.

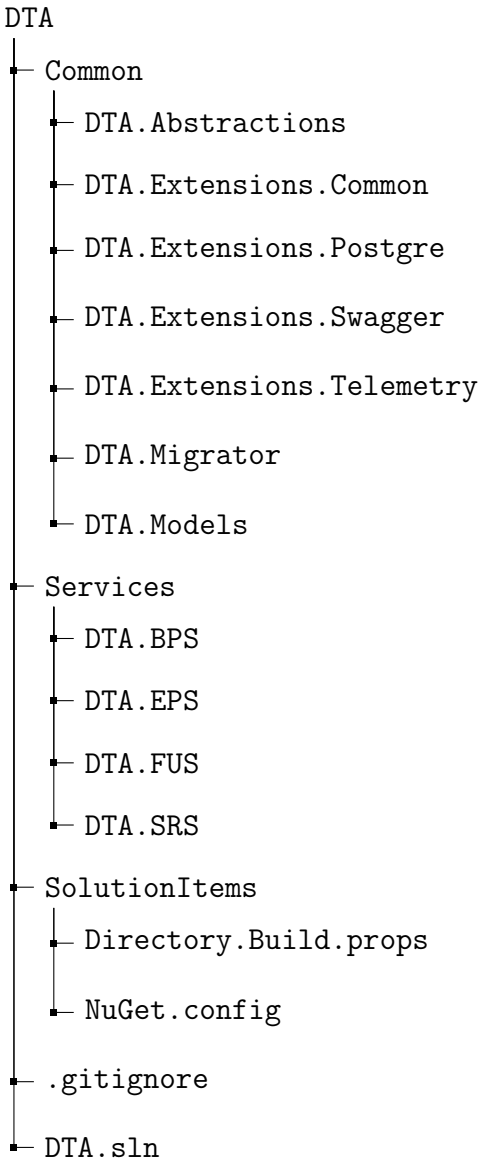
obrázek architektury

Řešení kompilačních cílů Pro kompilaci do nativního AOT kódu byl využit atribut 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.

foto využití konstant v kódu

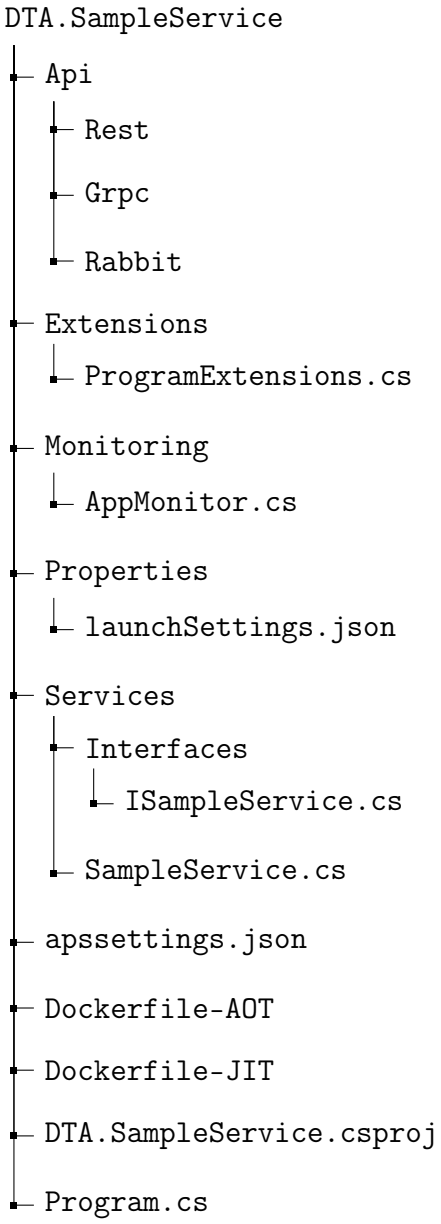
4.4.2 Organizace zdrojových souborů služeb

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.



4.4.3 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.

4.4.4 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 Dapper je open-source 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 umožňeno 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*.

4.4.5 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.Common*. Zde je poskytnuta funkcionalita pro sestavení názvů pro službu.

4.4.6 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.

4.4.7 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 simulovaným zdržením, časově založenému na měření skutečného zdržení systému při čtení dat ze vzdáleného zdroje u obdobného systému.

TODO: API docs

4.4.8 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.

TODO: API docs

4.4.9 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.

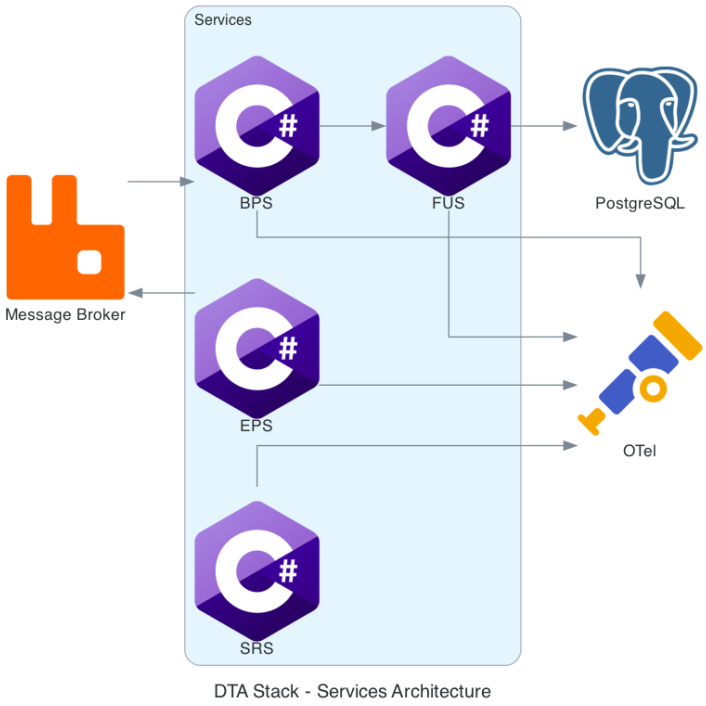
TODO: API docs

4.4.10 EPS - Event Publishing Service

Jednoduchá službami umožňující vyvolat událost v systému a docílit spuštění dodatečných operací v systému. V systému simuluje roli vydavatele událostí.

Služba poskytuje REST API rozhraní pro vyvolání události. Po vyvolání události je zpráva publikována na kanál *simulated* na službě *RabbitMQ*.

4.4.11 Přehled řešení



Obrázek 4.1 Diagram .NET služeb a závislých služeb

Následující diagram znázorňuje vztahy mezi jednotlivými službami.

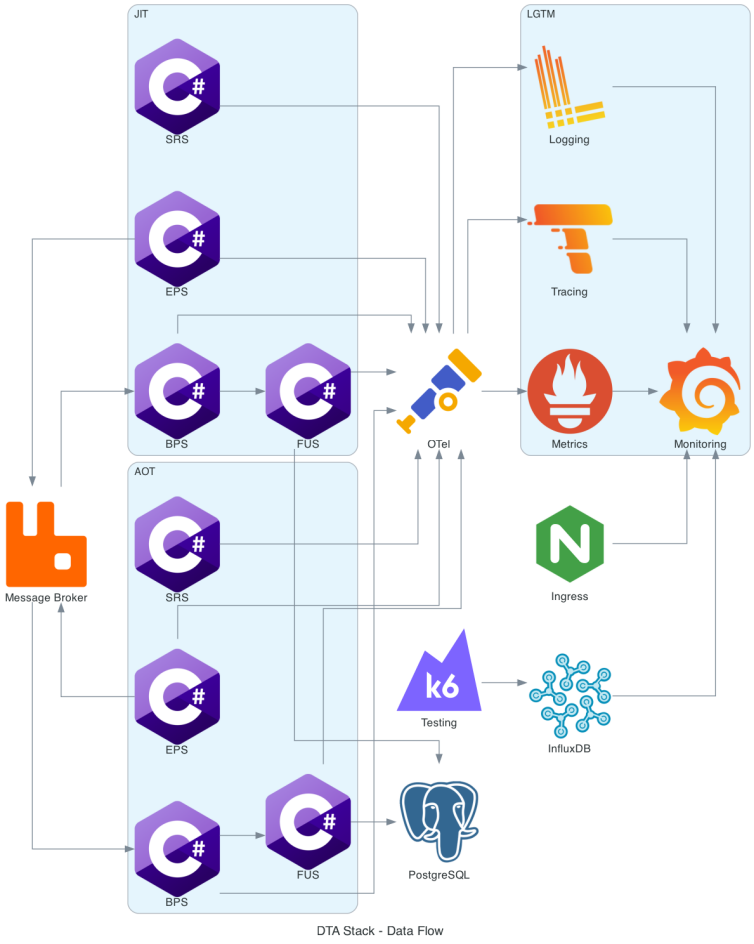
4.5 Konfigurace aplikace

4.5.1 Konfigurace služeb

Nginx Pro nginx je dodatečná konfigurace dodána pomocí souboru nginx.conf jenž je namountován do kontejneru. Tento soubor obsahuje konfiguraci pro nginx, která je použita při spuštění kontejneru.

Základní pravidla směrování

- / - cesta na statickou hlavní stránku-rozcestník aplikace
- /grafana - směrování na Grafanu



Obrázek 4.2 Telemetrie ve stacku

Statická stránka html.index je obdobným způsobem napojena do virualizovaného repozitáře kontejneru.

LGTM - Monitorovací stack LGTM jakožto monitorovací stack zároveň konfiguruje veškeré monitorovací nástroje. Značnou část konfigurace představuje propojení nástrojů a tato konfigurace je řešena pomocí konfiguračních souborů, které jsou použity již při tvorbě obecného obrazu.

Dodatečná konfigurace je řešena podle proměnných prostředí a týká se pouze malé množiny nastavení specifických pro správný běh monitorovacích nástrojů v celém stacku.

- **GF_SERVER_ROOT_URL** - nastavení URL, na které bude Grafana do-

stupná. Toto nastavení je důležité pro správné směrování požadavků na Grafanu.

- **GF_SERVER_SERVE_FROM_SUB_PATH** - nastavení, které určuje, zda bude Grafana dostupná z podadresáře v URL. Toto nastavení je důležité pro správné směrování požadavků na Grafanu.
- **GF_AUTH_ANONYMOUS_ENABLED** - nastavení, které určuje, zda bude povoleno anonymní přihlášení do Grafany.

SRS - Signal Reading Service Nasazení obsahuje konfiguraci definifující úroveň logování a cíl exportu telemetrických dat.

FUS - File Upload Service Nasazení obsahuje konfiguraci definifující úroveň logování a cíl exportu telemetrických dat.

BPS - Batch Processing Service Nasazení obsahuje konfiguraci definifující úroveň logování a cíl exportu telemetrických dat.

EPS - Fast Response Service Nasazení obsahuje konfiguraci definifující úroveň logování a cíl exportu telemetrických dat.

4.5.2 Konfigurace persistence

PostgreSQL PostgreSQL je konfigurována pomocí proměnných prostředí, kdy rozdíl od základní konfiguraci činí pouze definice přihlašovacích údajů pro připojení k databázi.

InfluxDB InfluxDB má upraven název výchozí databáze a nastavení autentifikace a přihlašovacích údajů. Tyto změny jsou provedeny za pomocí proměnných prostředí.

4.5.3 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.

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 Požadavky na scénáře

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.

5.2 Definice scénářů

Scénáře jsou vytvořeny jako množina javascriptových souborů splňujících požadavku API nástroje K6. Každý scénář je definován přes jeden a 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.

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.

5.3 Popis scénářů

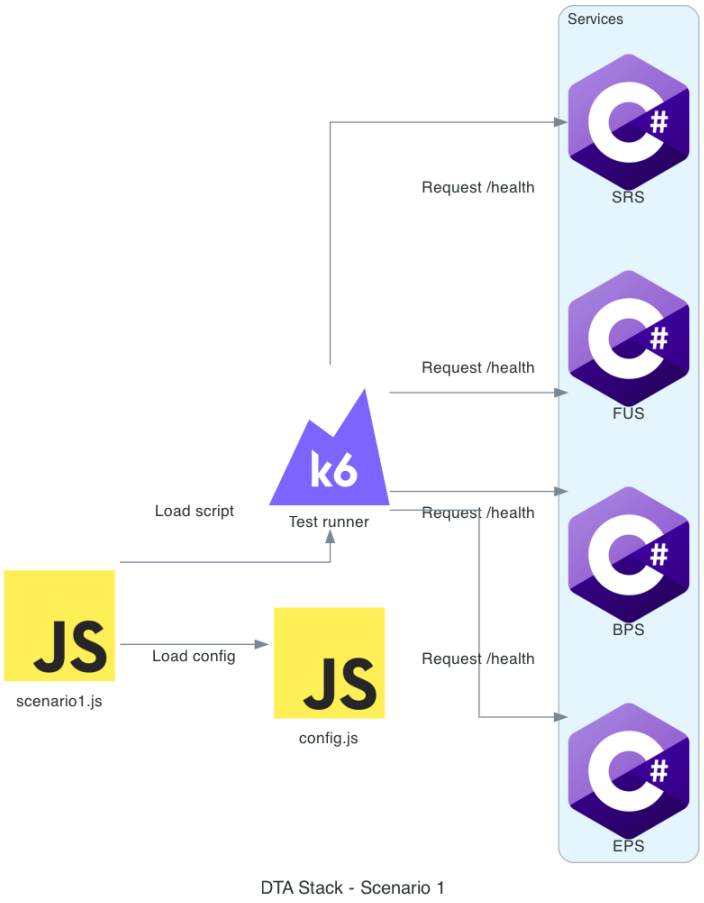
Následující sekce obsahuje popis scénářů, které byly vytvořeny pro testování výkonu a škálovatelnosti mikroslužeb kompilovaných JIT a AoT. Ke každému scénáři patři odpovídající sada souborů scriptů a konfigurací. Rovněž každý scénář disponuje vlastním interaktivním dashboardem v Grafaně, který umožňuje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

5.3.1 Scénář 1 - schopnost odpovídat služeb

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

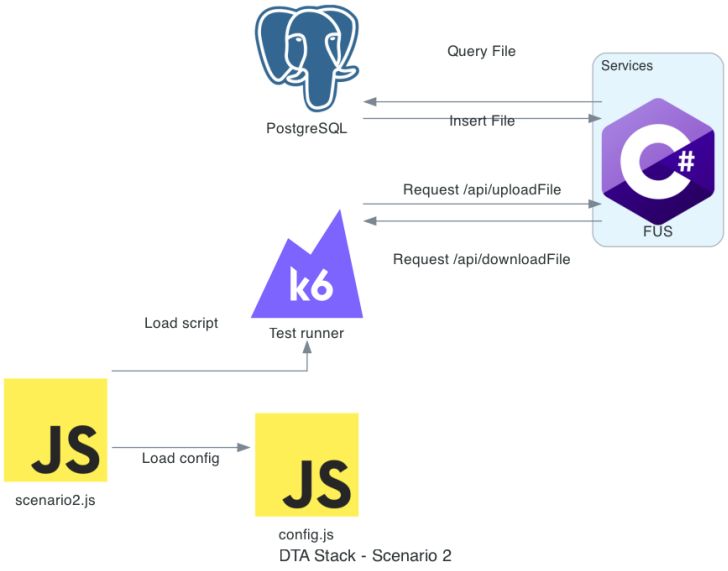
Průběh scénáře

- **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

5.3.2 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.



Obrázek 5.2 Diagram scénáře 2

Relevantní služby

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

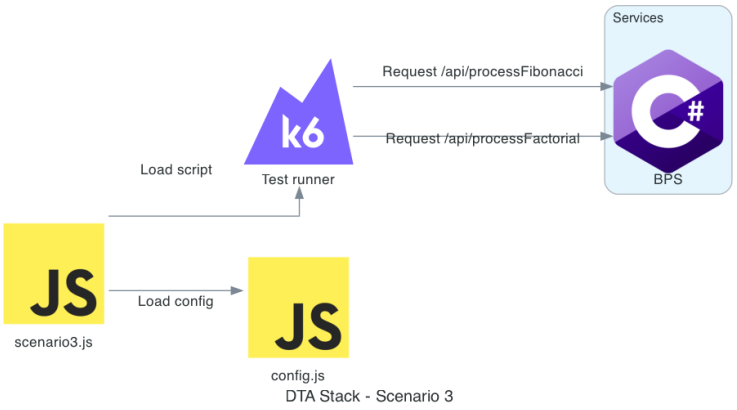
Průběh scénáře

- **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

5.3.3 Scénář 3 - zátěž zpracování dat

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. 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.



Obrázek 5.3 Diagram scénáře 3

Relevantní služby

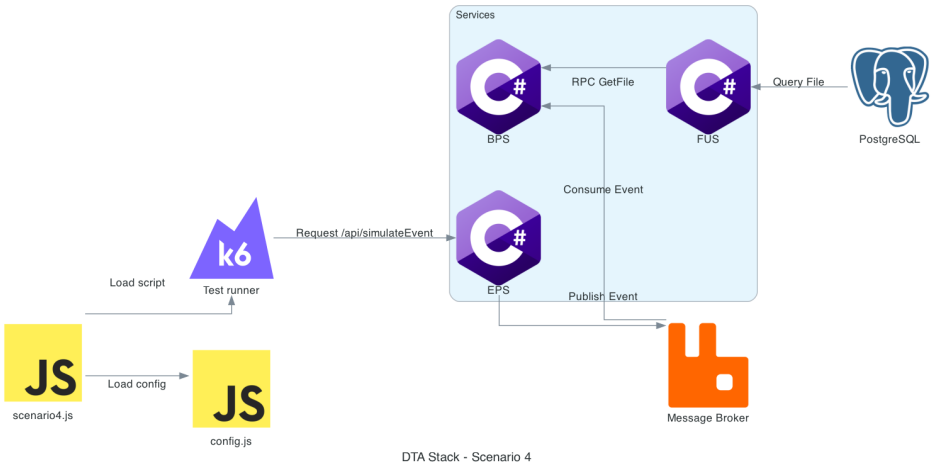
- **BPS** - služba, která poskytuje rozhraní a logiku pro výpočet faktoriálu a Fibonacciho posloupnosti

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

5.3.4 Scénář 4 - komunikace mezi službami

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.



Obrázek 5.4 Diagram scénáře 4

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
- **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

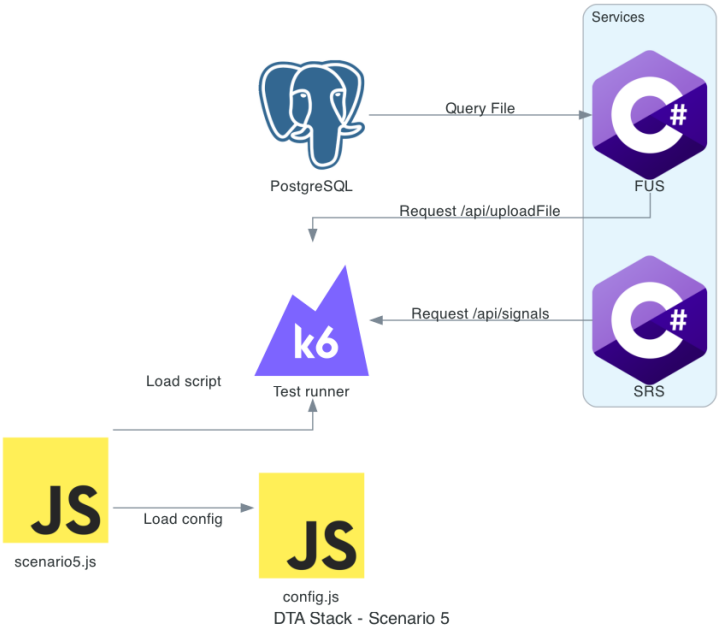
5.3.5 Scénář 5 - rychlost odpovědi po startu služby

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

- **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.



Obrázek 5.5 Diagram scénáře 5

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.

5.4 Spouštění scénářů

Jednotlivé scénáře jsou spouštěny dle definice v příslušném readme souboru. Jedná se o sekvenci instrkcí/příkazů pro přípravu požadovaného stavu systému a spuštění K6 testu v rámci kontejneru.

5.5 Zpracování a vizualizace dat

Po provedení testování scénářů je nutné zpracovat a vizualizovat data, která byla získána. To zahrnuje zpracování dat, která byla získána z testování scénářů, a zpracování dat, která byla získána z monitorovacích nástrojů.

5.5.1 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.

5.5.2 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 jsou ukládána do InfluxDb.

III. ANALYTICKÁ ČÁST

6 ANALÝZA APLIKACE

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 přívětive vizualizovány v rámci webového rozhraní.

Stack je testovatelný a nasaditelný na všech hlavních platformách.

6.2 Výstup služeb

Samotný proces nativní AOT a JIT kompilace je různě výkonnostně náročný. Při tvorbě samotného obrazu služby, ale i kompilace je hlavní náročná operace *restore*, která stahuje potřebné závislosti a balíčky pro projekt. Následující tabulka zobrazuje přehled časové náročnosti kompilace služeb pro oba kompilační cíle. Kompilace probíhá v rámci systému MacOS Sonoma 14.4.1, na čipu M1 v dispozici s 8GB RAM. Použitý příkaz je *dotnet build /p:Rebuild=True -c Release-<target>*, kdy *<target>* představuje vybranou kompilační metodu AOT nebo JIT.

Proces kompilace je vysoce závislý na specifickém HW a SW, následující tabulka poukazuje na výsledky testování na konkrétním HW a SW a má pouze informativní charakter.

Tabulka 6.1 Čas kompilace služeb

	JIT (s)	AOT (s)	AOT % nárůst
SRS	01.01	01.92	90.0
FUS	01.98	02.24	13.1
BPS	01.44	01.59	10.4
EPS	01.41	01.55	9.9

Výstupem služby jsou obrazy založené na linuxovém systému, Alpine s dotnet runtime v případě JIT výstupu služby, zredukované Ubuntu v případě nativního AOT výstupu. Z pohledu použitelnosti výsledných služeb má smysl měřit samotný výstupní obraz služby se všemi závislostmi. Následující tabulka zobrazuje velikost obrazu služeb pro oba kompilační cíle.

6.3 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

Tabulka 6.2 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

a nástrojů, za cenu nutnosti řešení problémů, případně změny implementace.

6.3.1 JIT

Vývojový proces pro kompilaci služeb JIT se zacílením na dotnet 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 dotnet 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.

6.3.2 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.

6.3.3 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 dotnet.

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.

6.3.4 Knihovny třetích stran

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.

7 ANALÝZA TESTOVÁNÍ

7.1 Charakteristika testovacího prostředí

Testovací prostředí, na němž došlo k testování, bylo založeno na operačním systému OSX Ventura. Pro testování byl využit docker engine. Testování bylo provedeno na stroji s čipem Mac M1.

7.2 Výsledky testování

7.2.1 Scénář 1

První scénář se zabíral jednoduchou funkcionalitou dotazu na healthcheck endpoint a meřením výkonu kestrel serveru u odpovědí na požadavky skrze REST API.

Výsledky Následující výsledky představují průměrnou dobu odezvy serveru na požadavek v milisekundách.

Na snímcích jsou zachyceny vybrané dashboardy z nástroje Grafana určené pro monitorování scénaře 1.

Závěr Testování přineslo překvapení ve výkonnostním rozdílu, jenž byl zaznamenán mezi JIT a AOT kompilací. Výsledky byly výrazně v neprospěch AOT kompilace. Samotná rychlost serveru není v mnoha případech kritickým faktorem, avšak v případě velkého množství požadavků, může být rozdíl v řádech milisekund zásadní.

7.2.2 Scénář 2

První scénář se zabíral jednoduchou funkcionalitou a meřením výkonu kestrel serveru a odpovědí na požadavky v REST API.

Výsledky Následující výsledky představují průměrnou dobu odezvy serveru na požadavek v milisekundách.

Na snímcích jsou zachyceny vybrané dashboardy z nástroje Grafana určené pro monitorování scénaře 1.

Závěr Testování přineslo překvapení ve výkonnostním rozdílu, jenž byl zaznamenán mezi JIT a AOT kompilací. Výsledky byly výrazně v neprospěch AOT kompilace. Samotná rychlost serveru není v mnoha případech kritickým faktorem, avšak v případě velkého množství požadavků, může být rozdíl v řádech milisekund zásadní.

7.2.3 Scénář 3

První scénář se zabíral jednoduchou funkcionalitou a meřením výkonu kestrel serveru a odpovědí na požadavky v REST API.

Výsledky Následující výsledky představují průměrnou dobu odezvy serveru na požadavek v milisekundách.

Na snímcích jsou zachyceny vybrané dashboardy z nástroje Grafana určené pro monitorování scénaře 1.

Závěr Testování přineslo překvapení ve výkonnostním rozdílu, jenž byl zaznamenán mezi JIT a AOT kompilací. Výsledky byly výrazně v neprospěch AOT kompilace. Samotná rychlost serveru není v mnoha případech kritickým faktorem, avšak v případě velkého množství požadavků, může být rozdíl v řádech milisekund zásadní.

7.2.4 Scénář 4

První scénář se zabíral jednoduchou funkcionalitou a meřením výkonu kestrel serveru a odpovědí na požadavky v REST API.

Výsledky Následující výsledky představují průměrnou dobu odezvy serveru na požadavek v milisekundách.

Na snímcích jsou zachyceny vybrané dashboardy z nástroje Grafana určené pro monitorování scénaře 1.

Závěr Testování přineslo překvapení ve výkonnostním rozdílu, jenž byl zaznamenán mezi JIT a AOT kompilací. Výsledky byly výrazně v neprospěch AOT kompilace. Samotná rychlost serveru není v mnoha případech kritickým faktorem, avšak v případě velkého množství požadavků, může být rozdíl v řádech milisekund zásadní.

7.2.5 Scénář 5

První scénář se zabíral jednoduchou funkcionalitou a meřením výkonu kestrel serveru a odpovědí na požadavky v REST API.

Výsledky Následující výsledky představují průměrnou dobu odezvy serveru na požadavek v milisekundách.

Na snímcích jsou zachyceny vybrané dashboardy z nástroje Grafana určené pro monitorování scénaře 1.

Závěr Testování přineslo překvapení ve výkonnostním rozdílu, jenž byl zaznamenán mezi JIT a AOT kompilací. Výsledky byly výrazně v neprospěch AOT kompilace. Samotná rychlost serveru není v mnoha případech kritickým faktorem, avšak v případě velkého množství požadavků, může být rozdíl v řádech milisekund zásadní.

8 DOPORUČENÍ PRO POUŽITÍ AOT KOMPILACE V DOTNET

AOT kód přináší jasné výhody výkonnostní výhody za cenu kompatibility. Řešení tvorby toho kódu, je však s ohledem na běžný postup velmi nešťastné. Využití interceptorů a generátorů bere inciativu z rukou vývojáře a vytváří naprosto nový program. Toto chování není natolik odlišné od průběhu kompilace do nativního systémového kódu v jiných jazycích, v případě .NET avšak bylo dodáno znatelně "post mortem".

Valná většina konkurenčních výhod, jenž z .NET plyne souvisí s možnostmi jeho runtime prostředí. Nativní AOT kompilace má smysl jen ve velice specifický situacích, jenž lze blíže identifikovat jako poskytování cludové infrastruktury a s tím spojenou potřebu běhu velkého množství instancí. Dalším příkladem je využití Serverless nebo také jako lambda funkce, kdy je poskytována funkcionalita a běh spuštění aplikace pro její vykonání je dílčí režie.

Případy konkurenční výhody pro AOT kompilaci staví na jednom předpokladu a to je zájem či potřeba mít zdrojové kódy v .NET, respektive jazyce C#. Při unimodálním přístupu, kdy je vývojář, respektive zapojený tým schopen přijmout jiný jazyk, jsou výhody AOT kompilace značně zmenšeny, zatímco nedostatky jsou zvýrazněny.

Nativní AOT kompilace má nesporné výhody za splnění uričtých podmínek na požadavky vůči nasazení, codebase a vývojového týmu. Zaplňuje určitou díru na trhu, která je však vzhledem k výše uvedeným podmínkám velice specifická. Pro běžné vývojáře usilující o výkonostní výhody, je však AOT kompilace v současné podobě nevhodná.

ZÁVĚR

Text závěru.

SEZNAM POUŽITÉ LITERATURY

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

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

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

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

[5] 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.

[6] 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.

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

[8] 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/

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

CPU	Central Processing Unit
PTFE	Polytetrafluoroethylene
VNA	Vector Network Analyser

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM PŘÍLOH

PŘÍLOHA P I. NÁZEV PŘÍLOHY

Obsah přílohy