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

*** Nascanované zadání, strana 1 ***

*** Nascanované zadání, strana 2 ***

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

Zde je místo pro případné poděkování, motto, úryvky knih, básní atp.

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 jazyce C# a jeho derivátech. 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 Nástroje .NET

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

- **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

1.3 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).

1.3.1 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). 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).

1.3.2 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.

1.3.3 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

1.3.4 Cíle kompilace

TODO

1.4 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.

1.4.1 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.

1.4.2 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.

1.5 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.

1.5.1 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 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.

1.5.2 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.

1.5.3 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.

Deklarace unmanaged rozhraní

Trimming

1.5.4 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

1.6 Srovnání

1.6.1 JIT

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.

1.6.2 AoT

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.

1.7 Závěr

xx

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 Popis

Architektura mikroslužeb rozděluje složité softwarové aplikace na menší, spravovatelné části, které lze vyvíjet, nasazovat a škálovat nezávisle.

2.2.1 Virtualizace a kontejnerizace

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

Kontejnerizace je obzvláště důležitá pro mikroslužby, protože zapouzdřuje každou mikroslužbu do vlastního kontejneru, což usnadňuje její nasazení, škálování a správu nezávisle na ostatních. Synonymem kontejnerizace se staly nástroje jako Docker, které nabízejí ekosystém pro vývoj, odesílání a provoz kontejnerových aplikací.

2.2.2 Orchestrace

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

2.2.3 Základní principy

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.

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

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.

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.

2.2.4 Serverless a 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.

2.3 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.

2.4 Výhody a nevýhody

2.4.1 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.

2.4.2 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.

2.5 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

Na této stránce je k vidění způsob tvorby různých úrovní nadpisů.

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

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 simuluje zátěž na mikroslužby a zjišťuje, jak se chovají za různých podmínek.

Konfigurace aplikace Aplikace musí být schopna konfigurovat testovací scénáře, které se mají provést, a způsob, jakým se mají provést.

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 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.

4.3.2 Persistenční vrstva

Pro persistenci relačních dat byla zvolena technologie PostgreSQL. PostgreSQL je open-source relační databázový systém, který je schopný zvládnout velké množství dat a zároveň poskytovat vysoký výkon.

4.3.3 Komunikační protokoly

Pro komunikaci mezi službami byl zvolen protokol HTTP. Verze HTTP/2 byla zvolena pro její schopnost zvládnout vysoký počet požadavků a zároveň poskytovat vysoký výkon.

4.3.4 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 je open-source platforma pro vizualizaci a analýzu dat. Grafana umožňuje vizualizovat data z různých zdrojů, včetně časových řad, protokolů a tras.

Prometheus je open-source systém pro sběr a vizualizaci metrik. Prometheus umožňuje sbírat metriky z různých zdrojů, včetně aplikací, systémů a služeb.

Loki je open-source systém pro sběr a vizualizaci protokolů. Loki umožňuje sbírat protokoly z různých zdrojů, včetně aplikací, systémů a služeb.

Tempo je open-source systém pro sběr a vizualizaci tras. Tempo umožňuje sbírat trasy z různých zdrojů, včetně aplikací, systémů a služeb.

OpenTelemetry je open-source systém pro sběr a vizualizaci metrik, protokolů a tras. OpenTelemetry umožňuje sbírat metriky, protokoly a trasy z různých zdrojů, včetně aplikací, systémů a služeb.

K6 K6 je 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.

4.3.5 Testovací služby

Pro implementaci testovacích služeb byl zvolena technologie dotnet, konkrétně jazyk C#. Dotnet je open-source platforma pro vývoj a provozování aplikací, která umožňuje vytvářet výkonné a škálovatelné aplikace. Služby budou implementovány jako mikroslužby, které budou spouštěny v kontejnerech. Služby jsou vytvořeny ve dvou verzích, které se liší v použitém způsobu kompilace, a to JIT a AoT.

4.4 Návrh a implementace testovacích služeb

4.4.1 Předpoklady služeb

Služby musí být implementovány tak, aby v obou kompilačních verzích poskytovaly totožnou funkcionalitu. Jejich chování musí být konfigurovatelné na úrovni kontejneru, který je spouští. Zároveň musí sbírat data o svém chování a poskytovat je monitorovacím nástrojům.

AoT kompilované služby budou otestovány s ohledem na možné kompilační optimalizace, které ovlivňují výsledný program. Toto chování je ovlivňeno atributem OptimizationPreference, který je součástí konfigurace služby.

4.4.2 Implementace služeb

SRS - Signal Readings Service Služba v systému hraje roli čtecího zařízení, které čte data ze zdroje a poskytuje je ostatním službám. Tato služba simulu základní kámen celého systému, značně ovlivňuje výkon a škálovatelnost celého systému. Očekává se velké množství požadavků na tuto službu.

Za účelem 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. Tato služba je implementována jako REST API (TODO: pokud konečná implementace bude gRPC, změň tuto sekci), které

poskytuje data ve formátu JSON. (TODO: Obrázek návrhu architektury a rozhraní služby).

FUS - File Upload Service Služba v systému hraje roli zapisovacího zařízení, které zapisuje data do zdroje. Tato služba hraje roli méně vytíženého služby, která nemá značný vliv na fungování systému jako celku. Požadavky, jenž musí vyřídit nejsou kritické a nutné řešit s minimální odezvou.

Služba je implementována s REST API rozhraním. (TODO: Obrázek návrhu architektury a rozhraní služby).

4.5 Konfigurace aplikace

4.5.1 Konfigurace služeb

Grafana

Prometheus

Loki

Tempo

OpenTelemetry

Postgres

K6

SRS - Signal Readings Service

FUS - File Upload Service

TWS - Test Workers Service

4.5.2 Konfigurace monitorovacích nástrojů

4.5.3 Nastavení uživatelského rozhraní

5 TESTOVÁNÍ SCÉNÁŘŮ

Na této stránce je k vidění způsob tvorby různých úrovní nadpisů.

5.1 Požadavky na scénáře

5.2 Popis scénářů

5.2.1 Scénář 1 - TBS

5.2.2 Scénář 2 - TBS

5.3 Zpracování a vizualizace dat

5.3.1 Monitorování v reálném čase

5.3.2 Sběr historických dat

III. ANALYTICKÁ ČÁST

6 ANALÝZA APLIKACE

6.1 Architektura

Popis výsledné struktury aplikace, včetně popisu jednotlivých služeb a jejich vzájemné interakce.

TODO: NDepend graf

7 ANALÝZA TESTOVÁNÍ

7.1 Charakteristika testovacího prostředí

7.2 Výsledky testování

8 DOPORUČENÍ PRO POUŽITÍ AOT KOMPILACE V DOTNET

TODO: AoT kompilovaný kód přináší xxx za cenu yyy. Na základě výsledků zzz usuzuji...

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