Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Reaktivní webové služby na platformě Java

(Porovnání přínosů reaktivního programování pro tvorbu mikroservisních webových služeb)

Diplomová práce

Autor: Bc. Vít Jouda
Studijní obor: Aplikovaná informatika

Vedoucí práce: Ing. Pavel Kříž, Ph.D.

Hradec Králové duben 2019

Prohlášení:

Prohlašuji, že jsem diplomovou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 30.4.2019 Vít Jouda

Poděkování:

Chtěl bych velmi poděkovat vedoucímu diplomové práce, panu Ing. Pavlovi Křížovi, Ph. D. za jeho čas a cenné rady při vypracování této práce.

Anotace

Diplomová práce se věnuje problematice vývoje mikroservisních webových aplikací založených platformě Java za pomoci reaktivního programování. Hlavním cílem práce je popsat výhody a nevýhody reaktivního přístupu oproti klasickému imperativnímu stylu. Práce popisuje modelové situace, které odpovídají reálným potřebám moderních distribuovaných systémů. Na těchto jsou provedena měření základních metrik, které jsou porovnány s imperativním blokačním modelem.

Annotation

Title: Reactive Web Java Applications

The diploma thesis deals with the development of reactive based microservices on Java platform. The main purpose of the thesis is to describe main advantages and disadvantages of reactive approach in contrast to classical imperative style. The thesis describes model situations which correspond to the real work requirements of modern distributed systems. These are used to measure basic metrics, and compare them to imperative, blocking model.

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

S velkým rozšířením využití výpočetní techniky v rámci každodenních činností člověka se zvyšují nároky na vlastnosti systémů, které zajišťují provoz celé řady služeb. Aby byly tyto nároky splněny, vznikají nové technologie a metodiky vývoje, jejichž cílem je maximalizace využití dostupných prostředků.

Reaktivní programování jako varianta neblokačního, funkcionálního stylu je známá již řadu let a přímá podpora či pomocné knihovny existují pro velké množství programovacích jazyků. Ve světě webových aplikací na platformě Java se však tento styl dosud ve větší míře nerozšířil. S příchodem frameworku Spring Boot 2 s plnou podporou neblokačních, reaktivních serverů WebFlux se situace změnila a mnoho vývojářů i firem zvažuje, zda je pro ně tato technologie přínosem a měla by být adaptována.

Cílem této práce je popsat principy reaktivního programování a porovnat jeho reálné kvantitativní vlastnosti s blokačním, imperativním stylem na modelovém systému, který splňuje typické požadavky na moderní, distribuovanou, mikroservisně orientovanou architekturu, provozovanou v cloudovém prostředí. Tyto vlastnosti jsou důležitým vodítkem pro určení reálného přínosu této technologie.

První teoretická část práce představuje platformu Java a mikroservisně orientovanou architekturu. Popisuje dílčí technologie, nástroje a principy, které jsou využívány pro tvorbu a provoz webových systémů založených na této architektuře.

V další části práce jsou představeny základní principy reaktivního programování, spolu s hlavními knihovnami, které tento styl podporují. Jsou zde rozebrány jednotlivé frameworky a neblokační webové servery, na jejichž základě je možné vyvíjet plně reaktivní webové aplikace v jazyce Java.

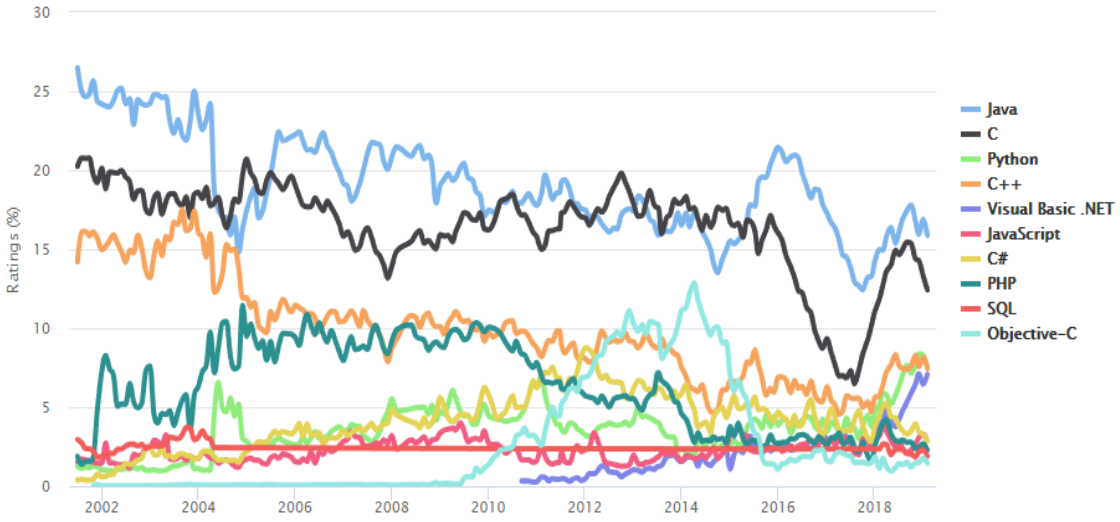
Třetí část práce se zabývá praktickým srovnáním popsaných technologií aplikovaných na modelový systém, který byl navržen na základě poznatků z přechozích kapitol. Jsou zde podrobně popsány jednotlivé testovací implementace, infrastruktura, použité technologie a metodika měření.

V závěru práce jsou rozebrány výsledky měření, na jejichž základě je uvedeno doporučení pro aplikaci této technologie.

2 Platforma Java

Platforma Java byla představena firmou Sun Microsystems v roce 1995. Jedná se o sadu, která obsahuje stejnojmenný programovací jazyk, nástroje a specifikace, jejichž hlavním cílem bylo sjednocení vývoje pro různé systémy. To je symbolizováno mottem „Napiš jednou, spusť kdekoliv“ (z anglického „Write once, run anywhere“). V této době byl pro vývoj v komerční sféře nejvíce využíván programovací jazyk C, případně jeho objektová varianta C++, které však trpěly problémy s přenositelností kódu a méně obsáhlou knihovnou často používaných funkcionalit. Java si tak rychle získala velkou uživatelskou základnu [Binstock, 2015].

I přes relativní stáří této platformy a rozmachu programovacích jazyků v posledních několika letech patří Java či její součásti dlouhodobě mezi nejvíce využívané platformy pro vývoj komerčních či mobilních aplikací, jak lze vidět na obrázku 1.



Obrázek 1 - TIOBE languages index

Zdroj: [TIOBE, 2019]

Java se dále dělí na několik platforem, které si liší podle využití. Mezi základní kategorie patří:

- **Java Standard Edition (SE)** – Poskytuje základní funkcionalitu jazyka Java. Definuje základní datové struktury i pokročilé API pro práci se sítí, zabezpečením či grafickým uživatelským rozhraním [Oracle, 2019b].

- **Java Enterprise Edition (EE)** – Rozšiřuje platformu Java SE o podporu pro vývoj a provoz síťových klient – server aplikací [Oracle, 2019a].

2.1 Java Virtual Machine

Základem platformy Java je její virtuální stroj, takzvaná Java Virtual Machine, dále jen JVM. Jedná se o klíčovou část umožňující abstrakci kódu napsaného ve vyšším programovacím jazyce na jednotlivé HW či SW platformy. Kód napsaný v jazyce Java je nejprve přeložen do takzvaného bytecode. Jedná se o instrukční sadu JVM, která je pak následně interpretována na cílové platformě.

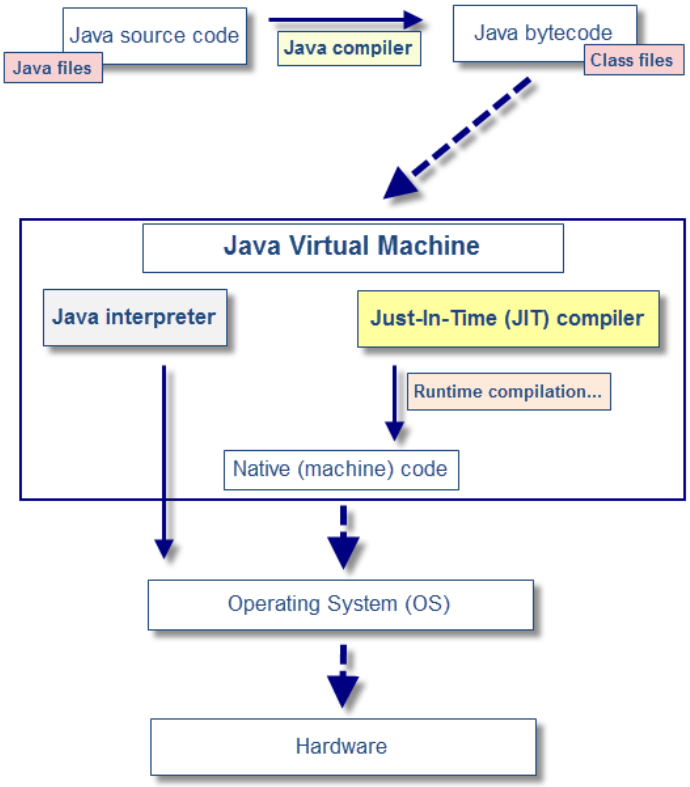
Způsob interpretace bytecode je různý v závislosti na konkrétní JVM. Existuje několik možných přístupů, které se odlišují zejména v náročnosti při spuštění a za běhu programu.

2.1.1 Přímá interpretace

JVM napřímo vykonává instrukce zapsané v daném bytecode. Program je tak spuštěn velmi rychle, avšak rychlost zpracování instrukcí je výrazně pomalejší oproti nativnímu kódu. Tento přístup byl využíván v prvních implementacích JVM pro osobní a serverové počítače, avšak již ve verzi 1.2 byl představen první Just-in-time kompilátor jako volitelná součást JVM [Schildt, 2000].

2.1.2 Just-in-time

Česky „právě včas“. JVM na základě nastavené úrovně optimalizace kompiluje kritické části kódu do nativních instrukcí dané platformy. Program se typicky skládá z různých částí kódu, některé jsou zodpovědné za jeho inicializaci a jsou vykonány pouze jednou. Jejich kompilace by tak byla zbytečným plýtváním zdrojů. Naopak jiné části jsou zodpovědné za klíčovou funckionalitu programu a jsou volány velmi často. Jejich kompilací lze dosáhnout výrazného zvýšení rychlosti programu. Analýzou toku kódu a určením těchto bloků lze výrazně zvýšit rychlost zpracování programu za cenu delší inicializace [Oaks, 2014]. Blokové schéma JVM s podporou JIT je znázorněno na obrázku 2.



Obrázek 2 - Blokové schéma JVM s JIT kompilací

Zdroj: [Charbonneau, 2013]

2.1.3 Ahead-of-time

Česky „předem“. Jedná se o přístup, kdy je bytecode zkompilován přímo do nativního kódu dané platformy, ještě před jeho spuštěním. Na rozdíl od klasické kompilace kódu před jeho distribucí se v tomto případě kompilace provádí na cílové platformě v rámci jeho instalace, případně lze využít výstupu JIT kompilace po prvním spuštění programu a zrychlit tak následné spuštění. Tento přístup je vhodný zejména pro platformy s omezenými systémovými prostředky [Hong et al., 2007].

2.2 Programovací jazyky

Přestože primárním jazykem této platformy je Java, díky využití JVM a jeho existujících implementací pro většinu populárních systémů vznikla celá řada dalších, vyšších programovacích jazyků, které se kompilují do Java bytecode. Tato kapitola popisuje nejrozšířenější jazyky, které jsou založené či kompilovatelné pro JVM.

2.2.1 Java

Java je statický typovaný, objektově orientovaný, paralelní jazyk představený spolu s platformou Java v roce 1995. Od té doby prošel tento jazyk přirozenou evolucí, a byly do něj zapracovány prvky funkcionálního i reaktivního přístupu. Jedná se zejména o podporu proudového zpracování dat přidanou ve verzi Java SE 8. Spolu s ní začal jazyk podporovat také lambda výrazy, které výrazným způsobem zpřehlednily zápis metod vyššího řádu a umožnili tak širší adaptaci funkcionálního přístupu. Ve verzi Java SE 9 byla přidána podpora reaktivního zpracování dat dle specifikace [Reactive Streams Special Interest Group, 2019]. Došlo tak k oficiálnímu sjednocení API napříč knihovnami podporujícími reaktivní programování.

2.2.2 Groovy

Groovy je dynamický jazyk s volitelným typováním. Díky tomu, že je založený na JVM, nabízí plnou kompatibilitu s knihovnami napsanými v jazyce Java. Umožňuje psaní samostatně vykonatelných skriptů a také podporuje tvorbu doménově specifických jazyků (z anglického „Domain Specific Language, dále jen DSL). To využívá řada nástrojů, mezi nejznámější patří například sestavovací systém Gradle. Groovy obsahuje takzvané closures, tedy bloky kódu, které lze přiřadit do proměnné a pracovat s nimi jako s objekty [König et al., 2015]. Díky tomu byl Groovy jeden z prvních jazyků pro JVM podporující základy funkcionální programování.

2.2.3 Kotlin

Kotlin je staticky typovaný, objektově orientovaný jazyk s podporou pro funkce vyššího řádu a lambda výrazy. První oficiální vydaní verze 1.0 bylo v roce 2016 společností JetBrains. Kotlin je prezentován jako pokročilejší, stručnější alternativa k jazyku Java, se kterým je interoperabilní. Existující Java kód lze zavolat z Kotlinu a naopak. Umožňuje tak postupný přechod mezi těmito jazyky. V roce 2017 byl Kotlin zvolen hlavním programovacím jazykem pro OS Android a nahradil tak doposud používaný jazyk Java [Shafirov, 2017]. Kotlin je také možné zkompilovat do jazyka JavaScript či do nativního kódu bez nutnosti využití virtuálního stroje v rámci projektu Kotlin/Native [JetBrains, 2018a].

2.2.4 Scala

Scala je silně typovaný programovací jazyk s plnou podporou pro funkcionální a objektově orientovaný model. Stejně jako ostatní jazyky představené v této kapitole je plně interoperabilní s knihovnami napsanými v jazyce Java. V současné době podporuje kompilaci a běh v rámci JVM nebo JavaScript VM. Scala je hojně využívána pro zpracování analytických operací nad velkou množinou dat. Spolu s Kotlinem patří mezi nejrychleji adaptované jazyky na platformě Java (JVM) [JetBrains, 2018b].

3 Webové služby

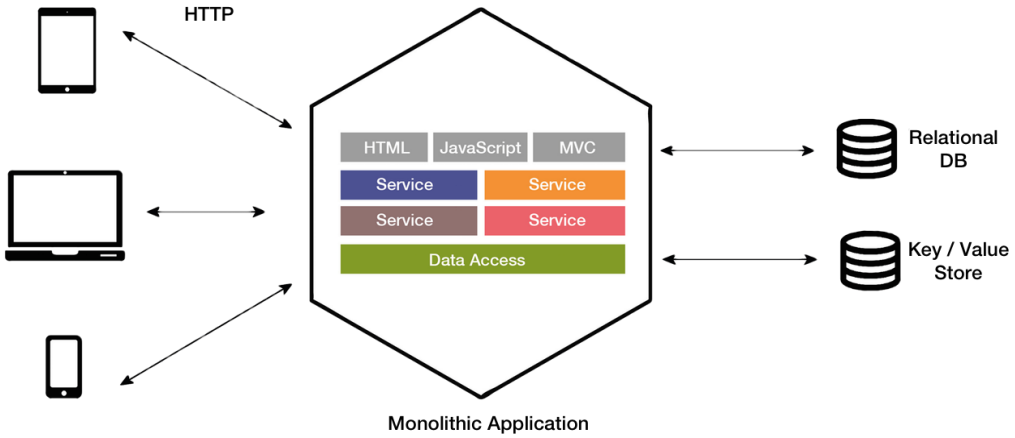
Termín webové služby se používá pro popis architektury, kde komunikace mezi jednotlivými aplikacemi probíhá za pomoci protokolů a technologií původně určených pro web, jako je například HTTP. Díky tomu lze využít již existující infrastruktury pro přenos dat.

Historicky byl tento termín nejprve využíván zejména pro služby založené na komunikaci pomocí protokolu SOAP [W3C, 2007a], ale díky popularizaci dalších metodik byl postupně zobecněn a dnes zahrnuje široké spektrum komunikačních protokolů a datových formátů. S rozvojem tzv. Single Page Application, dále jen SPA, se webové služby využívají také jako primární zdroj a úložiště dat pro webové aplikace [Mikowski et al., 2013].

Tato kapitola popisuje základní architektury webových služeb včetně jejich správy, komunikačních protokolů, systémů ukládání dat a provozu v rámci lokální či cloudové infrastruktury.

3.1 Monolitická architektura

Pokud jedna aplikace obsahuje veškerou logiku potřebnou pro splnění všech požadavků dané business domény jako celku, lze jí považovat za monolitickou. Výhodou těchto aplikací je snadné nasazení do produkčního prostředí a její správa. Struktura monolitické aplikace je znázorněna na obrázku 3.



Obrázek 3 - Schéma monolitické aplikace

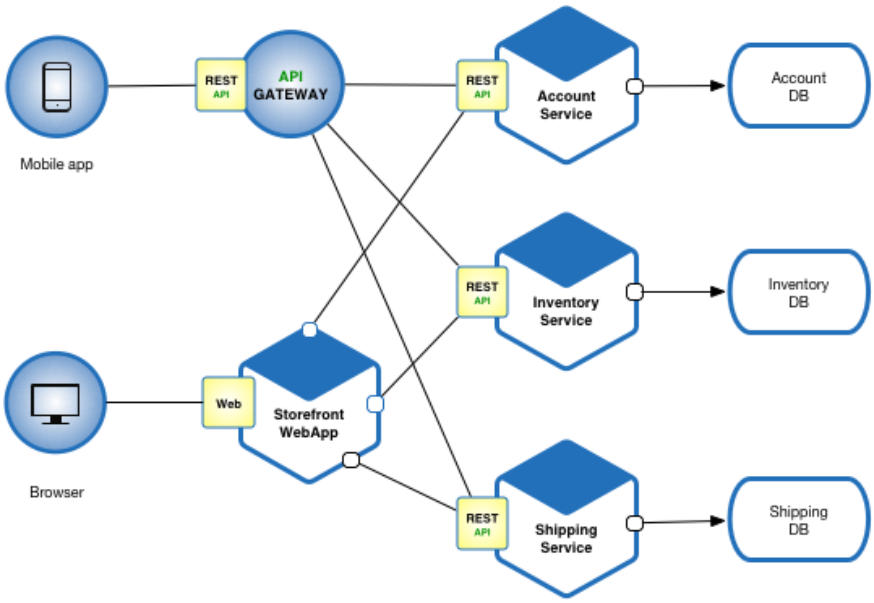
Zdroj: [Khan, 2018]

Problémy při vývoji a údržbě monolitických aplikací často vznikají, když se původní návrh začne rozšiřovat o další business požadavky, na kterých může pracovat i několik vývojových týmů najednou. Vznikají tak situace, kdy není možné přesně určit, který tým pracuje na konkrétní části aplikace. To přináší kolize v kódu aplikace, překrývání jednotlivých domén a tím degraduje kvalita kódu aplikace.

Dále v případě monolitických aplikací není možné škálovat jednotlivé domény nezávisle na ostatních. Pokud je jedna část aplikace přetěžována, je nutné škálovat aplikaci jako celek. Obdobně, pokud jedna část aplikace selže, ohrozí tím dostupnost ostatních částí, přestože mohou být doménově soběstačné [Atchison, 2016].

3.2 Mikroservisně orientovaná architektura

Mikroservisy lze definovat jako malé, nezávislé služby, které spolu navzájem spolupracují. Každá služba má vymezenu svoji subdoménu a operace, které v jejím rámci vykonává. Jednotlivé služby jsou odděleny svým rozhraním, a lze k nim přistupovat jako k tzv. černým skříňkám, tedy nezávislým blokům, jejichž vnitřní funkcionalita ani technologie nemusí být programátorovi známy. Důležité je pouze jejich veřejné rozhraní, takzvané API. Obrázek 4 znázorňuje tento typ architektury na příkladu platformy pro fiktivní internetový obchod.



Obrázek 4 - Příklad mikroservisní architektury

Zdroj: [Richardson, 2018]

Mikroservisní architektura vznikla na základě nedostatků monolitického modelu popsaného výše, které se snaží řešit. Avšak nejedná se o ultimátní řešení a přináší i některé nové problémy, se kterými je třeba se zabývat [Newman, 2015].

3.2.1 Hlavní výhody

3.2.1.1 Technologická variabilita

Jelikož se jedná o nezávislé služby existující jako samostatné aplikace, můžeme dle potřeby zvolit rozdílné technologie, vhodné pro danou subdoménu. Každá mikroservisa tak může být vystavena na jiném frameworku nebo dokonce v jiném programovacím jazyce. V případě monolitu by toto nebylo možné nebo velmi obtížné.

3.2.1.2 Odolnost

Zatímco v případě monolitu je v případě selhání jedné části ohrožena funkcionalita celého systému, selhání jedné mikroservisy zapříčiní, pokud je systém správně navržen, výpadek pouze jí spravované subdomény.

3.2.1.3 Škálovatelnost

Díky tomu, že je systém jako celek rozdělen na jednotlivé subdomény, je možné v případě zatížení jedné či více částí provést její horizontální či vertikální škálování. To v případě monolitické aplikace není možné. Pokud navíc provozujeme aplikace na některé z cloudových platforem, lze nastavit pravidla pro škálování dle aktuální zátěže a zefektivnit tak provoz aplikace a vypořádání se s výkonnostními špičkami.

3.2.1.4 Oddělení logiky

Pro správné vymezení jednotlivých subdomén v mikroservisní architektuře je nutné správně zanalyzovat a rozdělit celý systém do jednotlivých bloků. Díky tomu lze implementovat nové části systému bez nutnosti znalosti chování celku. Každou službu či skupinu služeb řešících konkrétní oblast může vyvíjet samostatný tým programátorů, aniž by vznikaly kolize jako v případě monolitu.

3.2.2 Časté problémy

Tato podkapitola popisuje časté problémy a výzvy při návrhu a provozu mikroservisní architektury.

3.2.2.1 Komplexita systému

Jelikož se systém skládá z mnoha jednotlivých částí, je složitější udržovat přehled o jeho chování jako celku, zejména ve vztahu k jednotlivým subdoménám. Tyto jsou typicky spravovány osobami, které se označují jako „vlastníci“. Při implementaci nové funkcionality je třeba koordinace mezi jednotlivými vlastníky a jejich týmy.

3.2.2.2 Latence

Jednotlivé mikroservisy spolu komunikují pomocí některého z protokolů popsaných níže, vždy však přes síťové rozhraní. To s sebou přináší problémy v podobě latence, která je násobně vyšší nežli komunikace v rámci jedné monolitické aplikace. To je nutné zvážit při rozdělení na jednotlivé subdomény, tak aby při zpracování požadavku nedocházelo k nadbytečným voláním napříč službami.

3.2.2.3 Oddělená data

Každá mikroservisa pracuje s vlastním zdrojem dat, typicky se jedná o databázový systém, lokální úložiště či jiný systém persistence. Data celé domény jsou tak distribuována mezi jednotlivými službami. To s sebou přináší některé problémy:

- **Integrita** – Pokud dojde k výpadku některé služby v průběhu zpracování požadavku, systém může být v nekonzistentním stavu.
- **Synchronizace** – Pokud jsou některé služby často dotazovány za účelem získání jejich dat (například pro pohled v klientské aplikaci), přináší to s sebou latenci. To lze řešit duplikací těchto dat tak, aby byly ihned k dispozici. V tomto případě je však nutné udržovat oba zdroje dat v synchronním stavu, což může být obtížné.

3.2.2.4 Verzování

Pokud dojde ke změně rozhraní jedné nebo více služeb, je nutné upravit všechny, které na ní závisí. Pří vývoji je proto důležité počítat se zpětnou i dopřednou kompatibilitou.

3.3 Komunikace

Při návrhu mikroservisní architektury je nutné zvolit vhodný způsob, jakým mezi sebou budou jednotlivé služby komunikovat. To je možné docílit pomocí celé řady synchronních či asynchronních protokolů a návrhových vzorů. Tato podkapitola popisuje nejčastěji používané mechanismy komunikace mezi mikroservisami.

3.3.1 RPC

Vzdálené volání procedur (z anglického „Remote Procedure Call“) je způsob komunikace mezi klientem a serverem. Server definuje svoje rozhraní pomocí metod, které mají definovány vstupní parametry a návratovou hodnotu. Každá taková metoda vyjadřuje konkrétní akci, kterou lze na serveru vykonat a jejich názvy typicky obsahují sloveso. Klient tyto metody volá pomocí některého transportního protokolu, v případě webových služeb se jedná o HTTP. Existují různé varianty RPC, které se liší na základě formátu přenášených dat či protokolu jejich výměny. Jedná

se například o XML-RPC [Merrick et al., 2006], JSON-RPC (2.0) [Morley, Matt, 2013] či dále podrobněji rozebraný SOAP.

3.3.2 SOAP

Původně označován jako Simple Object Access Protocol je protokol pro výměnu zpráv pomocí datového formátu XML. Lze ho využít s různými transportními protokoly, nejčastěji se však využívá ve spojení s HTTP. Navazuje na starší a jednodušší protokol XML-RPC. Zpráva v SOAP je standartní XML dokument, který se skládá z následujících elementů [W3C, 2007a]:

- **Envelope** – tvoří kořen XML dokumentu, identifikuje že se jedná o zprávu protokolu SOAP
- **Header** – nepovinný element, obsahuje informace specifické pro danou aplikaci, například ověření identity
- **Body** – obsahuje tělo zprávy pro zpracování cílovou metodou služby
- **Faul** – nepovinný element, používá se pro přenos chybových stavů

Příklad dotazu pomocí protokolu SOAP přes HTTP je znázorněn na obrázku 5.

```
POST http://www.webservicex.com/globalweather.asmx HTTP/1.1
Host: www.webservicex.com
Content-Length: 341
Content-Type: application/soap+xml

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Header></soap:Header>
  <soap:Body>
    <ns1:GetWeather xmlns:ns1='http://www.webserviceX.NET'>
       <ns1:CityName>Montreal</ns1:CityName>
       <ns1:CountryName>Canada</ns1:CountryName>
    </ns1:GetWeather>
  </soap:Body>
</soap:Envelope>
```

Obrázek 5 - Příklad zprávy protokolu SOAP

Zdroj: [Darrel, 2015]

SOAP je často používán ve spojení s jazykem pro definování webové služby WSDL (Web Services Description Language). Tento jazyk také vychází z formátu XML. Lze pomocí něj popsat funkce a datové struktury, které jsou poskytovány

danou službou. Z WSDL dokumentu lze pak pomocí automatizovaných nástrojů vygenerovat strukturu zpráv v protokolu SOAP [W3C, 2007b]. Webové služby tak mohou vystavit WSDL jako součást jejich rozhraní pro jednodušší napojení externích aplikací.

3.3.3 REST

Representational State Transfer [Fielding et al., 2000] není protokol, ale spíše metodika návrhu rozhraní webových služeb vystavených na protokolu HTTP. Rozhraní REST služby nereprezentuje funkce, ale datovou strukturu entit, označovanou jako zdroje (z anglického resources). Každá entita je identifikována svým unikátním ID. Jednotlivé zdroje je možné a vhodné dále logicky zanořovat, například pokud existuje zdroj pro získání dat o článku (blogovací platforma) „*/articles/1*“, jednotlivé komentáře k němu mohou být dostupné na URL „*/articles/1/comments*“.

Operace nad těmito entitami, jako je čtení, přidání, mazání či editování jsou definovány pomocí metod HTTP protokolu. Důležité je také zachování významu těchto metod z hlediska idempotence. Typicky používané metody jsou:

- **GET** – čtení dat, neovlivňuje obsah dat služby
- **POST** – vytvoření dat, každé volání vytvoří novou entitu
- **PUT** – vložení celé entity, opakované volání má stejný výsledek
- **PATCH** – úprava dat entity, v praxi se používají různé přístupy a formáty, například [Nottingham et al., 2013]
- **DELETE** – smazání celé entity, opakované volání neovlivňuje data služby

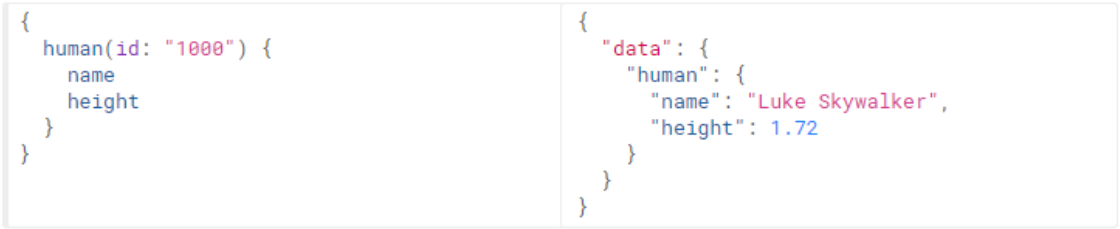
Formát pro přenos dat není pevně určen, nejčastěji se však využívá XML či JSON.

Ve spojení s REST lze také použít metodiku „Hypermedia As The Engine Of Application State“. Ta udává, že zdroje a operace relevantní k dané entitě by měly být součástí odpovědi serveru v podobě HTTP odkazů. Klient tak může procházet API služby pomocí těchto odkazů. Konkrétní formát udává například specifikace HAL [Kelly, 2013].

3.3.4 GraphQL

GraphQL je dotazovací jazyk pro webové služby a serverové běhové prostředí pro vykonávání těchto dotazů. Byl vytvořen společností Facebook za cílem zefektivnit přenos dat, jak po stránce obsahu, tak počtu dotazů nutných pro sestavení konkrétního pohledu, mezi klientskou aplikací a webovou službou. Na rozdíl od RESTu popsaného výše, aplikace vystavuje pouze jednu metodu pro každou doménu. Tato metoda přijímá GraphQL dotazy, na jejichž základě vrací požadovaná data napříč entitami. Také umožňuje jejich modifikaci pomocí tzv. mutací.

Pokud klientská aplikace potřebuje získat data, které by v REST či SOAP byly rozděleny do více entit či zdrojů, stačí je pouze specifikovat v rámci jednoho dotazu, včetně požadované podmnožiny jejich atributů. Tento přístup nejen snižuje nároky na počet dotazů, ale také snižuje množství potenciálně nepotřebných dat. Dále umožňuje aplikovat další optimalizace nad těmito daty, například efektivní cache [Buna, 2016]. Ukázka GraphQL dotazu spolu s odpovědí je znázorněn na obrázku 6.

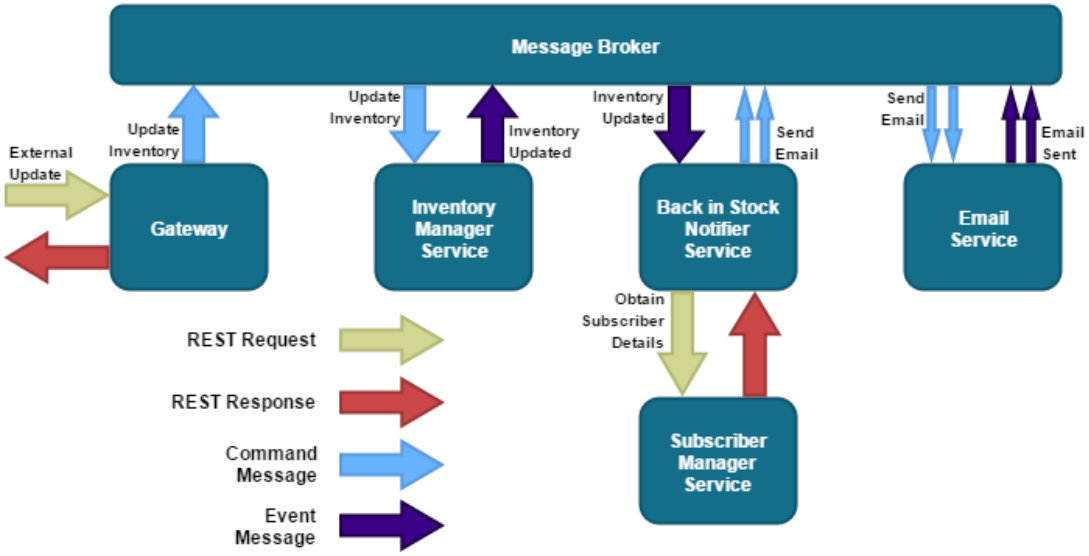


Obrázek 6 – Ukázka GraphQL dotazu

Zdroj: [Facebook Inc., 2019]

3.3.5 Message Queue

Jedná se kategorii protokolů a architektury, kdy spolu jednotlivé služby komunikují za pomoci prostředníka, tzv. Message Brokera, dále jen MB. Na rozdíl od výše zmíněných protokolů a metodik se nejedná o způsob komunikace pomocí HTTP, avšak je možné ho s ním kombinovat v rámci jedné služby, či použít pro interní komunikaci v rámci mikroservisní architektury, zatímco vnější komunikace je řešená pomocí vstupní brány, která pracuje na HTTP. Tato architektura je znázorněna na obrázku 7.



Obrázek 7 - Příklad komunikace pomocí Message Queue

Zdroj: [Williams, 2015]

Komunikace probíhá tak, že služba zašle zprávu do fronty, která je spravovaná některým MB. Na této frontě naslouchá další služba či služby, které danou zprávu obdrží a mohou na ni reagovat. Tato komunikace je z principu asynchronní, odesílatel tedy ihned po odeslání nedostává potvrzení o zpracování této zprávy.

Výhodou tohoto modelu je, že v případě selhání konkrétní služby nedochází k přerušení a ztracení informace. Ta nadále existuje v rámci fronty a bude zpracována, jakmile se služba obnoví a opět začne na této frontě naslouchat. Nevýhodou je pak jistá centralizace, tedy pokud by došlo k selhání MB, jednotlivé služby by spolu nemohly dále komunikovat.

Existují různé protokoly a implementace MB, které se liší v přístupu ke zpracování zpráv a cílí na různé použití. Níže jsou představeny některá populární řešení.

3.3.5.1 RabbitMQ

Odlehčený, open-source MB pro lokální či cloudové použití. Podporuje různé operační systémy a platformy a poskytuje širokou škálu nástrojů pro většinu populárních programovacích jazyků. V současnosti se jedná o nejpoužívanější open-source MB na trhu [Pivotal, 2019].

3.3.5.2 Apache Kafka

Kafka je open-source distribuovaná streamovací platforma, někdy též popisovaná jako distribuovaný logovací systém. Je založena na systému ZooKeeper, který je využíván pro sledování stavu jednotlivých uzlů. Mimo funkcionality jako MB také nabízí stream API, které slouží pro zpracování dat (mapování, agregace atd.) v jednotlivých frontách, v terminologii Kafky označovaných jako Topic. Kafka je navržena s orientací na rychlost zpracování dat [Apache Software Foundation, 2019].

3.4 Rozkládání zátěže

Pro zpracování velkého množství paralelně příchozích požadavků může dojít k využití všech zdrojů služby, jako jsou výkon procesoru, paměť RAM či propustnost síťového rozhraní. Tyto parametry lze postupně navyšovat (horizontální škálování), avšak tento přístup má svoje horní limity, kdy to již buď není technicky možné nebo velmi obtížné. Také je z hlediska dostupnosti nevýhodné mít pouze jednu instanci služby pro případ výpadku.

Alternativou je pak škálování horizontální, kdy je nasazeno několik instancí (uzlů) dané služby, které se dělí o zátěž produkovanou klienty. V tomto případě lze snadno pokrýt výkonnostní špičky dočasným přidáním dalších uzlů, a zároveň jsou dostupné alternativy v případě výpadku některé služby. Pokud zvolíme tuto možnost, je nutné požadavky distribuovat mezi dostupné instance. To lze provést na straně serveru, klienta či kombinací těchto přístupů [Atchison, 2016].

3.4.1 Serverové

V případě serverového rozkládání zátěže, dále jen LB z anglického Load Balancing či Load Balancer, je před skupinu aplikačních serverů (označovaných jako cluster) předřazen buď HW LB v podobě dedikovaného zařízení, nebo některé SW řešení nasazené na samostatném serveru. Jeho úkolem je analyzovat zátěž, která směřuje na jednotlivé uzly a přerozdělovat příchozí požadavky pro jejich rovnoměrné zatížení, pomocí přednastavených kritérií. Výhodou tohoto řešení je, že klientské aplikace nemusejí monitorovat stav dostupných instancí a jejich adresy.

3.4.2 Klientské

Klientský LB je na rozdíl od serverového součástí klientské aplikace či modulu pro volání konkrétní služby. Obdobně jako serverový LB si udržuje seznam dostupných instancí a na základě zvolené strategie vybere, na kterou provede volání. Výhodou tohoto řešení je jednodušší síťová infrastruktura a vyšší odolnost systému. Požadavky nechodí přes centrální bod (LB) a je tedy více tolerantní vůči výpadkům.

3.4.3 Detekce služeb

Pro získání adres konkrétních uzlů je možné použít statický seznam, který je součástí klienta. Lepší je však využít Service Discovery, dále SD. Každá služba se v rámci inicializace zaregistruje do SD serveru. Ten pak udržuje informaci o její dostupnosti. To je realizováno tak, že SD server periodicky volá jednotlivé uzly, případně naopak jednotlivé uzly periodicky provolávají SD. Pokud je některý uzel nedostupný, respektive se nehlásí, je vyřazen z tabulky instancí. I v tomto případě je důležité mít SD servery replikovány, aby jejich kolapsem nedošlo k vyřazení veškeré komunikace.

3.5 Datová úložiště

Pro ukládání persistentních dat lze využít různé přístupy a systémy. Nejčastěji je však použit některý systém řízení báze dat (DBMS), běžněji označovaný jako databázový systém. Těch existuje velké množství, které se dělí do několika základních kategorií, na základě toho, s jakými daty pracují, jaké operace nad nimi lze provádět a přístupem k jejich organizaci. Níže jsou blíže představeny základní kategorie těchto systémů.

3.5.1 Relační databáze

Relační databáze mají svůj název odvozený od přístupu k ukládaným datům. Ty jsou rozděleny do tabulek uspořádaných n-tic na základě jejich logického seskupení. Jednotlivé sloupce reprezentují atributy dat, která mají přiřazený konkrétní datový typ. Řádky pak jednotlivé záznamy. Tyto tabulky jsou označovány jako relace.

Vztahy mezi jednotlivými tabulkami (záznamy) jsou reprezentovány pomocí identifikátoru (ID). Pro modelování vztahu 1:N lze použít tzv. cizího klíče v rámci

odkazujícího se záznamu. Pro modelování vztahu M:N se pak využívá tzv. spojové tabulky, která obsahuje páry ID propojených záznamů. Existují různé úrovně normalizace dat, které jsou označovány jako Normální formy, dále NF. V praxi se často data dělí do úrovně 3. NF, kdy musí být všechny neklíčové atributy vzájemně nezávislé.

Relační databáze mají pevně danou strukturu dat, tzv. schéma. Nabízejí pokročilé možnosti pro jejich správu, zajištění integrity, transakce a velmi rychlé vyhledávání v datech za pomoci indexovaných atributů. Dotazování na konkrétní data je provedeno pomocí vlastního jazyka SQL (Structured Query Language), který má další dialekty v rámci odchylek od standardu konkrétních RDBMS[Halpin et al., 2008].

Jedná se o jeden z nejvyspělejších systémů pro persistenci a čtení strukturovaných dat. Mezi nejznámější systémy patří Oracle DBMS, MySQL, PostgreSQL, MicrosoftSQL a další.

3.5.2 Dokumentové databáze

Dokumentové databáze spadají do kategorie tzv. NoSQL databází. Od RDBMS se odlišují zejména v přístupu ke struktuře dat. Zatímco RDBMS mají přesně definované schéma dat, dokumentové DB umožňují ukládání polostrukturovaných dat, které lze reprezentovat v podobě klíč – hodnota a lze je libovolně zanořovat. Tento typ dat je označován jako dokument a obvykle je reprezentován pomocí datového formátu JSON.

Dokumentové databáze vznikly na základě požadavku na zrychlení vývoje aplikací a možnosti jejich evoluce v podobě měnícího se schématu dat. Jako hlavní výhody těchto databází se uvádí schopnost pojmout velké množství polo či nestrukturovaných dat, agilní vývoj a rychlá iterace schématu, podpora OOP a geograficky distribuovaná, škálovatelná architektura [MongoDB Inc., 2019].

Oproti RDBMS mají tyto DB typicky horší podporu pro udržení integrity dat, zajištění provázání jednotlivých záznamů či transakce. Je tak na aplikační logice, aby tyto vlastnosti zajistila nebo byla navržena tak, aby minimalizovala jejich potřebu.

Pří návrhu datové struktury se typicky volí mezi dvěma přístupy. Data lze buď seskupit a zanořovat do dokumentů na základě konkrétních potřeb daného

pohledu aplikace či jejího API, případně lze volit více normalizovanou formu v podobě malých, navzájem provázaných dokumentů. Často se také používá kombinace těchto přístupů. Nejznámější zástupce této kategorie je MongoDB.

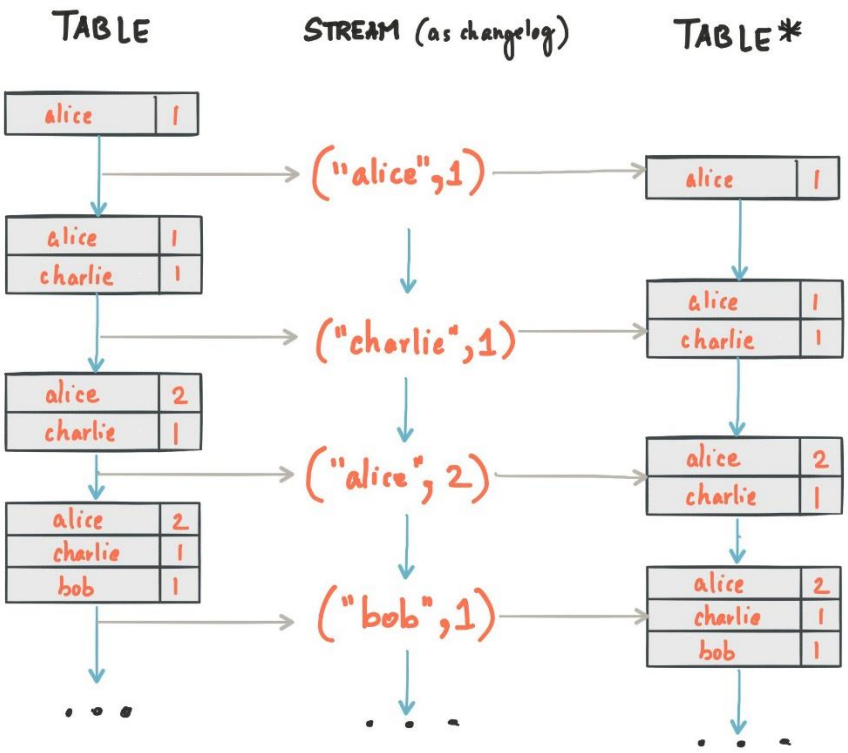
3.5.3 Databáze časových řad

Databáze časových řad (TSDB) ukládají data ve formátu klíč-hodnota vzhledem k jejich časové souslednosti. Umožňují tak zachytit jejich časový průběh. Jako klíč je využit právě čas vzniku události, jako hodnota pak číslo reprezentující hodnotu této události, ta je také označována jako metrika. Poskytují analytické dotazovací funkce, pomocí kterých lze získat podrobné údaje o ukládaném jevu.

Tento typ databází je často využíván pro monitoring aplikací. Vybrané události jako je počet příchozích požadavků, čas jejich zpracování atd. jsou uloženy do TSDB. Z těchto dat lze pak sestavit podrobné přehledy a grafy, například pomocí systému Grafana [Grafana Labs, 2019]. Zástupci této kategorie jsou například Prometheus a InfluxDB.

3.5.4 Streamovací platformy

Ačkoliv se nejedná přímo o databázový systém, k uložení a následnému dotázání na data lze využít také některé streamovací platformy, například výše zmíněnou platformu Apache Kafka. Využívá se zde tzv. stream-table duality, kdy lze pomocí agregace toku dat v čase sestavit jejich tabulární reprezentaci. To je znázorněno na obrázku 8.



Obrázek 8 - Dualita toku dat a tabulky

Zdroj: [Confluent Inc., 2019]

Výhodou tohoto principu je možnost dotazovat a transformovat data vzniklá na základě souslednosti událostí. V některých případech tak není nutné pro jejich persistenci používat další separátní systém.

3.6 Provoz webových služeb

Webové služby lze provozovat v rámci vlastní infrastruktury lokálně, nebo lze využít provozu v rámci virtuální, sdílené infrastruktury, tzv. cloudu. Následující podkapitola popisuje hlavní výhody a nevýhody obou řešení.

3.6.1 Lokální

Lokální provoz s sebou nese nutnost pořízení a údržby vlastního HW a SW, včetně nezbytné režie pro jejich správu. Podle požadavků na jejich parametry se může jednat o rozsáhlé investice, zejména pro začínající společnost. Pokud má provozovaná aplikace či služba výkonnostní špičky, které je třeba vykrýt, je nutné poskytnou dostatečně výkonný hardware, přestože nemusí být většinu času využit. Výhodou tohoto řešení je kompletní správa životního cyklu aplikace i jejich dat. To

může být požadováno či dokonce vyžadováno v případě zpracování citlivých dat. Také lze architekturu lépe přizpůsobit konkrétním potřebám aplikace.

3.6.2 Cloud

Cloudová řešení mohou být privátní či veřejná. Privátní cloud je většinou provozován v rámci velkých firem, kdy se interní aplikace mohou dělit o dostupné HW prostředky. Jedná se o určitý kompromis mezi dedikovanou infrastrukturou a využitím externího poskytovatele, kde firma musí stále udržovat vlastní HW, avšak lze ho efektivněji využít, zejména pokud se potřeby provozovaných aplikací mění a nepřekrývají se jejich výkonnostní špičky.

V případě veřejného cloudu je infrastruktura poskytována externím zákazníkům, za účelem zisku či sdílení nákladů. Zákazník tak nemusí budovat vlastní infrastrukturu a zejména v začátcích tak může ušetřit výrazné náklady na provoz aplikací. V praxi se používají různé modely účtování a nabízených služeb. Od samostatných virtuálních instancí lišících se dostupnými prostředky (počet vláken, velikost RAM…) po dedikované služby jako je load balancing, DB servery atd., které bývají účtovány za přenesená data, počet požadavků či velikost a rychlost persistentního úložiště. Níže jsou představeny dvě populární cloudová řešení.

3.6.2.1 Amazon Web Services

Amazon Web Services, dále jen AWS, je název pro širokou škálu cloudových služeb a řešení poskytovaných společností Amazon.com. Jednou z hlavních služeb je Elastic Compute Cloud, dále jen EC2. Jedná se o poskytování virtuálních serverových instancí [Amazon.com Inc., 2019]. Ty se dělí do různých kategorií na základě jejich zaměření, architektury a prostředků. Mezi základní, aktuálně nabízené kategorie patří:

- **A1** – instance založené na architektuře Arm, poskytují značnou úsporu nákladů, ideální pro škálovatelnou zátěž
- **T2** – nejlevnější víceúčelové instance, vhodné pro provoz webových aplikací, mikroservis a dalších komerčních aplikací
- **M5** – aktuální generace víceúčelových instancí, vhodná pro malé až střední databáze, zpracování dat či aplikace vyžadující dodatečnou paměť

Každá z těchto kategorií se dále dělí podle velikosti instancí na základě počtu vláken CPU, operační paměti, persistentního úložiště a síťové propustnosti. Za instance je možné platit podle způsobu jejich využití [Amazon.com Inc., 2019].

On-Demand

Pří využití tohoto modelu se platí za každou využitou hodinu či sekundu, v závislosti na typu instancí. Nevyžaduje dlouhodobé závazky a výpočetní kapacitu lze měnit dynamicky dle aktuálních požadavků. To je vhodné zejména pro aplikace, které dosahují krátkodobých výkonnostních špiček či nepředvídatelné zátěže, které nemohou být přerušeny.

Spot instances

Jedná se o „nadbytečné“ instance, které mohou být přerušeny 2 minuty po obdržení notifikace v případě, že je jejich výkon zapotřebí pro jiné využití. Použitím těchto instancí lze ušetřit až 90 % nákladů na provoz. Z jejich podstaty jsou vhodné pouze pro zátěže, které jsou tolerantní vůči výpadkům, jako jsou zpracování velkých dat či nestavové webové servery.

Reserved instances

Instance lze dopředu rezervovat a tím ušetřit až 75 % nákladů oproti instancím typu On-Demand. Jsou vhodné pro aplikace, které mají předvídatelné a dlouhodobé zatížení.

Dedicated Hosts

Fyzické EC2 servery, dostupné pro využití klientem. Vhodné zejména pro provoz aplikací, které jsou licenčně spjaté s konkrétním serverem, mezi které patří Windows Server, SQL Server SUSE Linux Enterprise Server a další.

3.6.2.2 Google Cloud Platform

Google Cloud Platform, dále jen GCP, je obdobně jako AWS označení pro soubor služeb a řešení pro provoz aplikací na infrastruktuře společnosti Google. Poskytování virtuálních instancí je označováno jako Google Compute Engine. Ten nabízí jak předpřipravené typy instancí, tak možnost vytvořit specifický typ dle

potřeb zákazníka. V tomto případě je možné zvolit požadovanou kombinaci virtuálního CPU a množství dostupné paměti. Mezi základní nabízené typy patří:

- **n1-standard** – základní typ vhodný pro aplikace které mají vyvážené potřeby na využití CPU a paměti
- **n1-highmem** – typ se zvýšeným množstvím operační paměti, která dosahuje až 624 GB, vhodný pro aplikace náročné na operační paměť
- **n1-highcpu** – typ se zvýšeným počtem vCPU oproti operační paměti, vhodné pro aplikace zaměřené na výkon
- **f1-micro** – sdílené využití procesorového vlákna, nejvýhodnější typ pro provoz aplikací, které nejsou náročné na systémové prostředky

Také tyto typy se dále dělí do podtypů, které se liší konkrétním množstvím dostupných vCPU a operační paměti.

Cena je v případě všech typů instancí odvozena od skutečného počtu vCPU, GPU či množství operační paměti, nikoliv podle typu instance. Lze tak zohlednit vlastní typy instancí. Účtuje se první celá minuta a poté každá započatá sekunda. Pokud zákazník využije pouze 30 s platí za celou jednu minutu času. Na základě využití těchto zdrojů je možné získat jednu z následujících slev [Google Inc., 2019].

Dlouhodobé využití

Jedná se o automaticky aplikovanou slevu, která bude započítána, pokud bude specifický zdroj využit výraznou část zúčtovacího měsíce. Tuto slevu lze uplatnit na vCPU, GPU či dedikované servery. Platí pouze na instance, které jsou vytvořeny v rámci Google Kubernetes Engine nebo Google Compute Engine.

Závazné využití

Compute Engine nabízí možnost zamluvit si konkrétní množství vCPU a operační paměti na 1 nebo 3 roky, výměnou za výraznou slevu na tyto zdroje. V tomto případě jsou služby účtovány měsíčně, bez ohledu na to, zda jsou využity či nikoliv.

Preemptivní instance

Preemptivní instance lze provozovat za výrazně nižší cenu oproti klasickým instancím. Mohou však být kdykoliv ukončeny, v případě že jsou jejich zdroje

zapotřebí jinde. Také mohou nepřetržitě běžet pouze po dobu 24 hodin. Tento typ instancí je doporučován zejména pro dávkové zpracování dat.

4 Reaktivní programování

Reaktivní programování, dále označováno také jako Rx, je typ asynchronního zpracování dat a událostí, které je založeno na konceptu tzv. streamů a jejich modifikaci. Na rozdíl od klasického imperativního modelu reaguje na změny v aplikaci (může se jednat o interní či externí událost), které zpracovává asynchronně. Zaměřuje zejména na tyto body:

- čitelnost kódu a jeho kompozici
- datový tok, který lze modifikovat obsáhlým slovníkem operátorů
- žádná akce se neprovede, pokud není konzumována
- možnost signalizovat vydavateli, že emituje akce příliš rychle
- vysokou abstrakci, která je agnostická vůči konkurenci

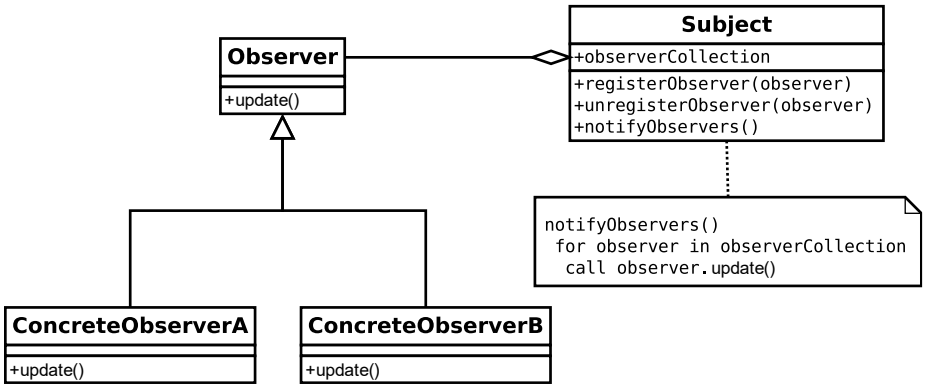
Tato kapitola popisuje hlavní principy Rx v teoretické rovině a jeho praktické využití v programovacím jazyce Java pro tvorbu webových služeb schopných zpracovat velké množství příchozích požadavků.

4.1 Základní principy

Rx v sobě kombinuje několik přístupů a metodik, které dále rozvíjí. Naslouchání událostí vychází z návrhového vzoru Observer, který je využíván v rámci objektově orientovaného programování. Dále jsou zde využity prvky funkcionálního a asynchronního programování, zejména pro zpracování a modifikaci dat v rámci jednotlivých operátorů.

4.1.1 Observer

Návrhový vzor Observer byl vytvořen k zachycení a zpracování toku událostí. Objekt emitující události, které jsou typicky navázané na změnu vnitřního stavu, je nazýván jako observable, někdy též jako subject. Ten umožňuje přijímat instance tříd naslouchajících na změny, které jsou označovány jako observer. Jakmile nastane změna stavu, observable zavolá příslušné metody všech registrovaných observerů. Těch může být 0-N dle konkrétní implementace. Obrázek 9 znázorňuje tento návrhový vzor v diagramu UML.



Obrázek 9 - UML diagram návrhového vzoru pozorovatel

Zdroj: [Gregorybleiker, 2018]

V Javě je implementace tohoto vzoru využita mimo jiné například v rámci knihovny pro tvorbu grafického uživatelské rozhraní Swing. Pro zpracování události o tom, že uživatel provedl nějakou akci, například stisknutí tlačítka, je nutné vytvořit instanci třídy implementující rozhraní *ActionListener* a následně ji zaregistrovat u instance příslušného tlačítka.

4.1.2 Funkcionální programováni

Funkcionální programování lze definovat jako styl tvorby aplikací sestávajících se z jednotlivých funkcí, které nemohou způsobovat vedlejší efekty ani modifikovat vstupní parametry. Tento název je odvozen z matematického pojetí funkce jako závislosti výstupní hodnoty pouze na vstupních parametrech. Funkce splňující tyto podmínky je označována jako „čistá“. Vedlejší efekt může být změna proměnné v kontextu dané funkce, či závislost na jiné funkci, která samo o sobě není čistá. Pokud takovou funkci voláme opakovaně se stejnými vstupními parametry, vždy musíme získat stejný výsledek.

Abychom mohli označit programovací jazyk za funkcionální, nebo alespoň podporující tento přístup, musí umožňovat pracovat s funkcemi obdobně jako s jinými typy proměnných. Lze je tedy vytvářet, přiřazovat a volat dle potřeby. Také musí obsahovat podporu pro funkce vyššího řádu, tedy funkce, které mohou jako parametr přijímat další funkce a lze je tak libovolně skládat a zanořovat.

Ačkoliv Java umožňovala tento typ programování za použití anonymních vnitřních tříd od verze 1.1, větší rozšíření nastalo až s příchodem lambda výrazů

v Java 1.8 spolu se sadou tříd a rozhraní pro usnadnění tohoto přístupu. Jedná se zejména o Stream API pro práci s kolekcemi a sadu funkcionálních rozhraní.

Funkcionální programování je ze svého principu vhodné pro paralelní zpracování dat, jelikož omezuje možnost kolizí jednotlivých vláken a zmenšuje potřebu pro použití některého z paměťových synchronizačních mechanismů. Proto jsou jeho prvky využity v rámci Rx.

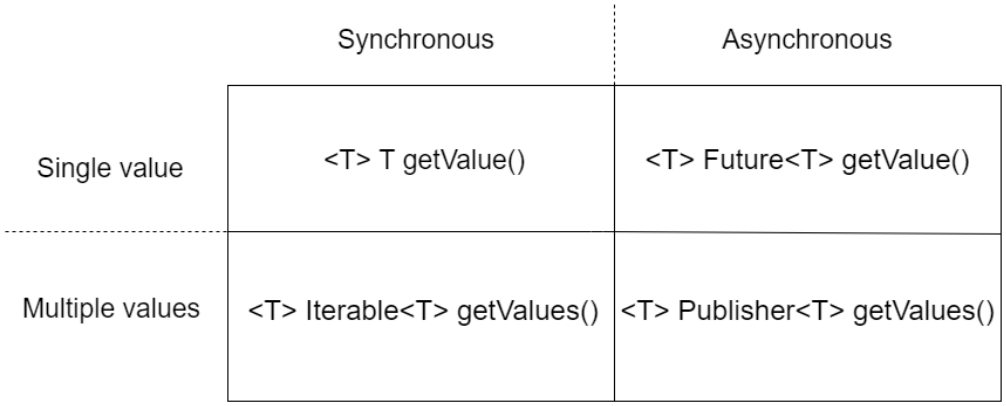
4.1.3 Asynchronnost

V případě synchronního zpracování programu jsou jednotlivé akce vykonávány postupně, v pořadí jejich definice. Pro započetí následující akce musí být dokončena akce předchozí. Pokud tedy potřebujeme vykonat výpočetně náročnou operaci, dojde k blokaci celého vlákna do doby jejího dokončení.

Asynchronní programování umožňuje definovat jednotlivé akce i jejich souslednost, avšak jejich vykonání může být odloženo do doby, kdy bude k dispozici dostatek času či prostředků, nedochází tak k blokaci celého vlákna. Tyto akce mohou být dále vykonávány konkurentně či paralelně:

- **konkurence** – Jedno procesorové vlákno může rozpracovat více akcí najednou, avšak pracuje vždy pouze na jedné akci v jeden čas.
- **paralelismus** – Několik akcí se vykonává souběžně v jeden čas, typicky ve vlastních procesorových vláknech.

Synchronní či asynchronní zpracování lze dále dělit na základě kardinality výstupu. To je v jazyce Java znázorněno na obrázku 10.



Obrázek 10 - Zpracování dat na základě kardinality

Zdroj: [Kassis, 2019], upraveno autorem

Pokud označíme výsledek zpracování akce jako T, lze synchronní proces vyjádřit prostým volání funkce, jejíž návratovou hodnotou je buď přímo T či kolekce hodnot T. V případě asynchronního zpracování je návratovou hodnotou obalový typ, který vyjadřuje budoucí výsledek. Pokud se jedná pouze o jednu hodnotu, může to být instance rozhraní *Future<T>*.

Toto rozhraní bylo přidáno v rámci Java 1.7 za účelem zapouzdření výsledku asynchronní akce. Poskytuje metody pro získání stavu zpracování akce a jejího výsledku. Ten lze získat voláním metody *get()*, která zablokuje vykonávání volajícího vlákna do doby, než je výsledek k dispozici, předčasně pokud jeho zpracování skončí výjimkou. Dále lze akci přerušit před jejím dokončením voláním metody *cancel()*. V Javě 8 byla přidána třída *CompletableFuture<T>*, která toto rozhraní implementuje. Umožňuje řetězení asynchronních akcí, které budou vykonány v závislosti na dokončení akce předchozí.

Pokud se jedná o předem neurčený počtu hodnot v intervalu 0-N, pak lze použít instanci rozhraní *Publisher<T>*, které je podrobněji rozebráno v další kapitole.

4.2 Reactive streams

Cílem iniciativy [Reactive Streams Special Interest Group, 2019] je vytvoření minimálního API, které bude plně podporovat asynchronní zpracování dat za použití neblokačního backpressure, který je rozebrán níže. Toto API by mělo být standardizováno a implementováno knihovnami, které podporují reaktivní model za účelem jejich vzájemné kompatibility. V současné podobě obsahuje rozhraní pro programovací jazyky Java a JavaScript, spolu s definicí síťových protokolů pro přenášení reaktivních streamů za pomoci existujících síťových protokolů, včetně jejich serializace a deserializace.

V případě jazyka Java se jedná o modul „*org.reactivestreams:reactive-streams*“, v době psaní této práce ve verzi 1.0.2. Využívá podobný model registrace jako v případě návrhového vzoru Observer. Jako zdroj událostí zde slouží rozhraní *Publisher<T>* (vydavatel), které emituje 0-N prvků. K němu se registruje *Subscriber<T>* (odběratel), jehož cílem je tato data konzumovat. V případě, kdy je potřeba data modifikovat a dále propagovat se jedná o tzv. *Procesor<T>*, který se

chová jako odběratel i vydavatel zároveň. Dále je zde definováno rozhraní *Subscription*, které slouží jako propojení mezi těmito prvky. Komunikace mezi vydavatelem a odběratelem probíhá na základě zasílání signálů, které jsou definovány v rozhraní *Subscriber<T>*. Jedná se o následující metody:

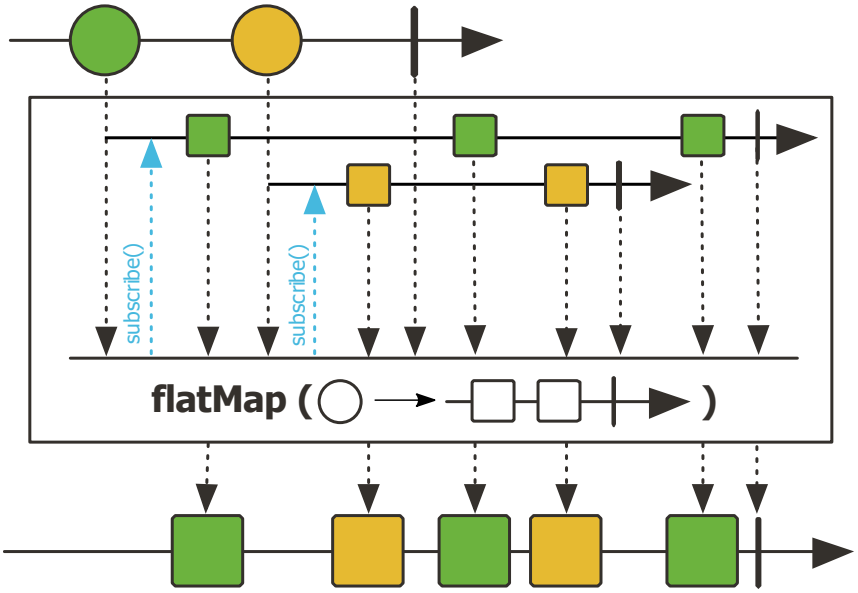
- ***onSubscribe(Subscription)*** – Metoda volaná pro zahájení komunikace. Odběratel zde obdrží instanci konkrétního subscription, pomocí kterého si může vyžádat N prvků (metoda request) či přerušit jejich zasílání (metoda cancel).
- ***onNext(T)*** – Metoda volaná v případě, kdy je k dispozici další prvek ve streamu.
- ***onError(Throwable)*** – Metoda volaná, pokud se v průběhu zpracování vyskytne chyba, ze které se nelze zotavit.
- ***onComplete()*** – Metoda volaná v momentě, kdy byla vyčerpána zdrojová data vydavatele a nadále nebude zasílat další signály.

S příchodem Java verze 9 byla představena třída *java.util.concurrent.Flow*, která obsahuje rozhraní pro reaktivní API, která jsou sémanticky plně kompatibilní s reactive-streams specifikací. Díky tomu se očekává, že knihovny, které dosud využívaly specifikaci reactive-streams přejdou na tuto nativní variantu. Do té doby je k dispozici adaptér pro převod mezi těmito typy v rámci modulu „*reactive-streams-flow-adapters*“.

4.2.1 Operátory

Pro usnadnění modifikace dat emitovaných vydavatelem představuje Rx koncept tzv. operátorů. Jedná se o sadu metod, které zjednodušují a standardizují operace s událostmi nebo jednotlivými daty. Tyto operátory se využívají jako základní stavební bloky, ze kterých jsou sestaveny asynchronní řetězce akcí, které se označují jako streamy. Z hlediska metodologie představené výše se nejčastěji jedná o procesory, tedy prvky, které se chovají jako odběratel i vydavatel zároveň.

Pro vizualizaci toku dat v rámci streamů se používá tzv. kuličkový diagram. Pro operátor *flatMap* je znázorněný na obrázku 11.



Obrázek 11 - Kuličkový diagram operátoru flatMap

Zdroj: [Spring Reactor, 2019]

Horizontální čáry představují směr toku dat daného vydavatele, jednotlivé geometrické obrazce pak emitované prvky stejného typu. Plná vertikální čára reprezentuje ukončení emise, tedy signál *onComplete*. Po emisi prvku z prvního vydavatele je na něj aplikována mapovací funkce, která vrací nového vydavatele. Ke všem takto mapovaným vydavatelům je přihlášen stejný odběratel (operátor), který jejich emise spojuje a dále propaguje do výsledného streamu. Také je zde vidět, že jednotlivé prvky výsledného streamu jsou emitovány v pořadí jejich emise z vnitřních vydavatelů, nikoliv v pořadí vstupních dat. To je značeno různými barvami daných obrazců. Případná výjimka je značena křížkem, který reprezentuje emisi signálu *onError*.

4.2.2 Backpressure

Existují 2 základní modely, kterými může být realizován vztah mezi vydavatelem a odběratelem událostí:

- **push** – Jednotlivé události jsou vydavatelem propagovány odběrateli vždy po jejich výskytu.
- **pull** – Odběratel se aktivně dotazuje vydavatele, zda má k dispozici nějaké události. Pokud ano, odebere jejich určité množství.

Termín backpressure vyjadřuje možnost odběratele událostí signalizovat jejich vydavateli, že jsou emitovány příliš rychle a nejsou včas zpracovány. Tato zpětná propagace se označuje jako hybridní push-pull model, jelikož si odběratel může zažádat o N prvků, pokud jsou k dispozici. Pokud však nejsou, budou automaticky zpropagovány ihned po jejich vytvoření.

4.2.3 Typy vydavatelů

Každý vydavatel zapouzdřuje zdroj dat, které emituje. Na základě toho, kdy je tento zdroj vytvořen a povaze dat dělíme vydavatele na tzv. studené a horké zdroje.

4.2.3.1 Studený zdroj

Pokud data vznikají v rámci jejich vydavatele, jedná se o studený zdroj. Ten je specifický tím, že generuje nová data vždy po připojení nového odběratele. Pokud se jich připojí více, každému bude vygenerován samostatný zdroj dat, který bude přehrán od začátku. Každý odběratel tak může kdykoliv získat nezávislá data.

4.2.3.2 Horký zdroj

V případě, že jsou data vytvářena mimo jejich vydavatele se jedná o zdroj horký. Jejich emise není určena připojením odběratele. Všichni připojení odběratelé získávají v jeden čas identická data. V tomto případě se jedná o tzv. multicast. Příkladem může být reakce na vstup uživatele. Události vznikají bez ohledu na to, zda existuje odběratel a jeho připojením je nelze zpětně vygenerovat. Je však možné použít různé strategie pro jejich uchování ve vyrovnávací paměti.

4.2.4 Sestavení vs. vykonání streamu

Při práci s reaktivními streamy je důležité odlišovat dvě fáze jejich životního cyklu. První fází je tzv. sestavení. Při ní vzniká konkrétní podoba streamu, tedy objekty poskytovatelů dat a řetězce operátorů. Tato fáze by měla být vykonána co možná nejrychleji, protože například v případě webové služby může být vykonávána pro každý příchozí požadavek a blokuje vlákno aplikace.

Jak bylo uvedeno výše, jedním z principů Rx je, že žádná akce není provedena, dokud není stream odebírán. Z toho vyplývá druhá fáze, tzv. vykonání streamu. Ta

započne v momentě, kdy se ke streamu přihlásí jeho odběratel. Dochází zde ke zpracování dat a jejich mutaci, typicky v rámci uživatelem definovaných metod v rámci jednotlivých operátorů. Veškeré blokující akce by měly být přesunuty do této fáze a dále vykonány asynchronně, případně v rámci jiného vlákna.

4.3 Spring Reactor

Pro jazyk Java existují 2 hlavní knihovny, které vycházejí z reaktivního specifikace [Reactive Streams Special Interest Group, 2019]. Jedná se o Spring Reactor a RxJava. Z jejich vzájemného porovnání z hlediska funkcionality [Nurkiewicz, 2019] či výkonu [Akarnokd, 2018] vyplývá, že tyto knihovny nabízejí obdobné funkce i rychlost zpracování jednotlivých operátorů. Hlavním rozdílem mezi nimi je, že Spring Reactor podporuje jazyk Java od verze 8 a pro mobilní vývoj na platformě Android lze tedy použít až od verze Oreo, zatímco RxJava podporuje i starší verze této platformy. Vzhledem k přímé podpoře Spring Reactor v rámci frameworku Spring WebFlux je dále představena a použita tato knihovna. Ukázky kódu v této kapitole byly vytvořeny autorem na základě oficiální dokumentace [Spring, 2019a].

4.3.1 Základní datové typy

Spring Reactor obsahuje 2 základní datové typy: *Mono<T>* a *Flux<T>*, které slouží jako vydavatelé. Oba implementují rozhraní *org.reactivestreams.Publisher<T>*. Hlavním rozdílem mezi těmito typy je jejich význam z hlediska kardinality dat a emise signálů.

4.3.1.1 Flux<T>

Datový typ *Flux* je standardní *Publisher*, reprezentující asynchronní sekvenci dat nabývající 0-N hodnot. Ta je ukončena voláním signálu *onComplete* případně *onError* v důsledku chyby. Může se také jednat o nekonečnou sekvenci, kde tyto signály nemusí být nikdy emitovány.

4.3.1.2 Mono<T>

Datový typ *Mono* reprezentuje speciální asynchronní zdroj dat, který může nabývat maximálně jedné hodnoty. Po připojení odběratele tak může zavolat pouze jednou

metodu *onNext()* a následně skončit zasláním signálu *onComlete*, případně *onError*, pokud dojde k chybě. Nabízí menší množství operátorů oproti typu *Flux*. *Mono* lze být také využito k reprezentaci probíhajícího asynchronního procesu, který pouze informuje o svém dokončení. V takovém případě lze použít *Mono<Void>.*

4.3.1.3 Vytvoření streamu

Pro tvorbu streamu hodnot z existujících dat jsou k dispozici statické factory metody. V případě jedné hodnoty se jedná o *Mono.just(T)* či *Flux.just(T)*, pro propagaci pouze signálu *onComplete* pak metody *Mono.emtpy()*, *Flux.empty()*.

Pro vytvoření sekvence více hodnot lze použít *Flux.just(T...)*. Ta lze také vytvořit na základě iterovatelné kolekce dat (například *ArrayList<T>*) pomocí metody *Flux.fromIterable(Iterable<T>)*. Chybový signál je možné vytvořit a propagovat pomocí metod *Mono.error(Throwable)*, *Flux.error(Throwable)*. Data lze také tvořit a vkládat do streamu programaticky. K tomu lze použit například následující metody:

generate

Metodu *generate* lze použít pro synchronní emisi dat na vyžádání, založené na předchozím stavu. Poskytuje rozhraní *SynchronousSink*, na kterém lze volat jednotlivé signály, vždy však maximálně jeden *onNext* v rámci volání callbacku. Níže je ukázka použití této metody pro tvorbu streamu, který emituje lichá čísla od 1 do hodnoty 5 včetně, po které se stream ukončí.

```
Flux.generate(
        // vychozi stav pro noveho odberatele
        () -> -1,
        (state, sink) -> {
            var next = state + 2;
            sink.next(next);
            if (next == 5) {
                sink.complete();
            }
            return next;
        });
```

create

Metoda *create* se využívá pro asynchronní, vícenásobnou emisi dat a umožňuje využít více vláken. Na rozdíl od metody *generate* poskytuje rozhraní *FluxSink* a data lze emitovat libovolně, jak je znázorněno v příkladu níže.

```
Flux.create(sink -> {
            new Thread(() -> {
                sink.next(1);
                sink.next(3);
                sink.next(5);
                sink.complete();
            }
            ).run();
        }
);
```

4.3.2 Operátory

Spring Reactor nabízí velké množství operátorů sloužících pro modifikaci streamů. Tato podkapitola představuje vybrané základní operátory, které jsou společné typům *Mono* i *Flux*, spolu s ukázkou jejich použití.

filter

Umožňuje eliminovat prvky ve streamu na základě dodaného predikátu. Příklad níže dále emituje pouze lichá čísla.

```
Flux.just(1, 2, 3, 4, 5)
        .filter(nr -> nr % 2 == 1);
```

map

Synchronní mapování prvků ve streamu. Jelikož se jedné o synchronní operaci, nemění pořadí dat v sekvenci. Příklad níže převádí čísla na řetězce.

```
Flux.just(1, 2)
        .map(Object::toString)
```

flatMap

Asynchronní mapování prvků ve streamu. Mapovací funkce vrací novou instanci vydavatele. Poté, co některý z takto vytvořených vydavatelů emituje další prvek, je předán do výsledného streamu. Jelikož se jedná o asynchronní operace, mohou takto být provedeny i časově náročné úlohy bez blokace volajícího vlákna, například

vzdálené volání či výpočetně komplexní algoritmy. Příklad níže znázorňuje uložení prvků do reaktivní databáze.

```
interface ReactiveDb<T> {
    Mono<T> save(T value);
}

Flux.just(1, 2)
        .flatMap(reactiveDb::save);
```

then

Operátor *then* ignoruje hodnoty ve streamu a dále propaguje pouze *onComplete* signál. Používá se, pokud není potřeba propagovat emitovanou hodnotu, ale pouze informaci o dokončení operace, například uložení do DB.

```
public <T> Mono<Void> save(Flux<T> values) {
    return values
            .flatMap(reactiveDb::save)
            .then();
}
```

onError

Sada operátorů, které reagují na výjimky ve streamu, tedy na signál *onError*. Existují 3 základní varianty:

- ***onErrorMap*** – umožňuje mapovat výjimku, která je dále propagována v rámci *onError* signálu
- ***onErrorResume*** – v reakci na výjimku je možné přepnout na jiného vydavatele
- ***onErrorContinue*** – eliminuje prvek, který způsobil výjimku a pokračuje prvkem následujícím

Následující příklad ukazuje použití operátoru *onErrorMap* k překladu generické výjimky databáze na doménovou. Výjimky jiného typu zůstávají bez překladu.

```
userService.createUser(new User("Mr. Mock"))
    .onErrorMap(DuplicateKeyException.class, UserServiceException::new);
```

doOn

Obdobně jako *onError* se jedná o sadu operátorů, které umožnují vykonat synchronní akci v závislosti na emisi některého z příslušných signálů – *doOnNext*,

doOnSuccess či *doOnError*. Tyto operátory se často využívají pro logování operací či mutaci vnějšího stavu aplikace, například v reakci na uložení hodnoty do DB.

```
public <T> Flux<T> save(Flux<T> values) {
    return values
            .flatMap(reactiveDb::save)
            .doOnNext((v) -> log.info("Value '{}' saved to DB", v));
}
```

4.3.3 Plánovače

Jelikož JVM plně podporuje paralelní zpracování dat v rámci jednotlivých vláken, je důležité určit, které vlákno má být použito pro zpracování dané akce. K tomu slouží tzv. plánovače. Spring Reactor k jejich zapouzdření využívá třídu *Schedulers*. Ta poskytuje funkcionalitu pro tvorbu vlastních plánovačů založených na poskytnuté instanci rozhraní *java.util.concurrent.Executor*. Dále obsahuje základní fond předpřipravených plánovačů, které se rozdělují podle jejich vlastností (zejména počtu pracovních vláken) a použití na následující:

- **immediate** – zastupuje aktuální vlákno
- **single** – využívá dedikované vlákno pro každou akci
- **elastic** – vytváří či znovu využívá vlákna ze svého fondu pro vykonání akcí, je vhodná pro zapouzdření blokujících I/O akcí
- **parallel** – speciální plánovač pro paralelní akce, používá tolik vláken, kolik je k dispozici na procesoru

Dále jsou k dispozici metody pro tvorbu nového plánovače na základě výše uvedených strategií pomocí metod *newSingle()*, *newElastic()* atd. Pro určení, který plánovač bude použit pro vykonání daného streamu či konkrétních operátorů lze použít jeden z operátorů *subscribeOn* či *publishOn*.

publishOn

Tento operátor slouží pro změnu plánovače následujících operátorů. Odebírá signály ze streamu a přehrává je dále, avšak již za použití dodaného plánovače. Tato změna trvá do doby, než je plánovač opět pozměněn použitím tohoto operátoru.

subscribeOn

Na rozdíl od předešlého operátoru ovlivňuje celý proces odběru událostí. Kdykoliv je přihlášen nový odběratel, vždy jsou všechny akce vykonány v plánovači definovaném tímto operátorem. Použití *publishOn* však stále mění plánovač pro všechny následné operátory ve streamu.

4.3.4 Testování

Pro testování reaktivních streamů založených na Spring Reactor existuje doprovodná knihovna „*io.projectreactor:reactor-test*" obsahující potřebné testovací utility. Základem je třída *StepVerifier*, která poskytuje rozhraní pro tvorbu testovací pipeline. Lze tak definovat, jaké hodnoty, případně v jakém pořadí očekáváme na výstupu testovaného streamu, včetně konkrétních signálů. Základem je metoda *expectNext(T)*, která ověřuje emisi jednoho konkrétního prvku. Tuto i další podobné metody lze řetězit až do zavolání posledního kroku testu, například některé z metod *verify()*, *verifyComplete()* či *verifyError()*. Tyto metody zajistí odebírání dat z testovaného vydavatele a případné ověření, že jsou zakončeny daným signálem. Příklad níže znázorňuje test streamu, který emituje jednotlivá písmena vstupního řetězce a poté je ukončen. Je-li vstupní řetězec prázdný, je emitována chyba.

```
public Flux<String> split(String word){
    var source = word.split("");
    return Flux.generate(() -> 0, (it, sink) -> {
        if(word.isEmpty()) sink.error(new IllegalArgumentException());
        sink.next(source[it]);
        if(it == source.length - 1) sink.complete();
        return ++it;
    });
}

@Test
public void testSplit(){
    StepVerifier.create(split("ab"))
            .expectNext("a")
            .expectNext("b")
            .verifyComplete();

    StepVerifier.create(split(""))
            .expectError(IllegalArgumentException.class);
}
```

Pokud je výstupem streamu emise konkrétní hodnoty v závislosti na jejím vstupu, lze jí testovat způsobem popsaným výše. Avšak pokud je výstupem pouze

Mono<Void>, nebo je třeba testovat zda byla provolána externí závislost, nelze spoléhat pouze na volání metody, jejíž návratovou hodnotou je vydavatel, protože může být zavolána již během sestavení a nikoliv během vykonání streamu. To lze demonstrovat na následujícím příkladu vytvoření či aktualizace entity na základě její existence.

```
public Mono<Void> createOrUpdateUser(User user) {
    return userService.getUser(user.getName())
            .flatMap((oldUser) -> userService.updateUser(oldUser, user))
            .switchIfEmpty(userService.createUser(user))
            .then();
}
```

V rámci unit testu této metody je vhodné otestovat, že pokud uživatel existuje, je vykonána akce *updateUser*, v opačném případě akce *createUser*. Jelikož *userService* není předmětem testu, lze použít její mock či stub, například za použití knihovny Mockito. Následující příklad zachycuje naivní test na druhý jmenovaný případ.

```
@Test
public void createIfDoesNotExist() {
    var testUser = new User("MockUser");
    when(userService.getUser(anyString())).thenReturn(Mono.empty());
    when(userService.createUser(any())).thenReturn(Mono.empty());
    StepVerifier.create(createOrUpdateUser(testUser))
            .verifyComplete();
    verify(userService).createUser(testUser);
}
```

Tento test je však falešně pozitivní, jelikož metoda *createUser(User)* je volána vždy v rámci sestavení výsledného streamu. V případě že uživatel neexistuje však nebude odebírána. Pro tyto případy existuje pomocná třída *PublisherProbe<T>*, která umožňuje vytvořit testovacího vydavatele, na kterém lze ověřit že byl skutečně odebírán. Níže je ukázka upraveného testu, který již není falešně pozitivní.

```
@Test
public void createIfDoesNotExist() {
    var testUser = new User("MockUser");
    var createProbe = PublisherProbe.<User>empty();
    when(userService.getUser(anyString())).thenReturn(Mono.empty());
    when(userService.createUser(any())).thenReturn(createProbe.mono());
    StepVerifier.create(createOrUpdateUser(testUser))
            .verifyComplete();
    createProbe.assertWasSubscribed();
}
```

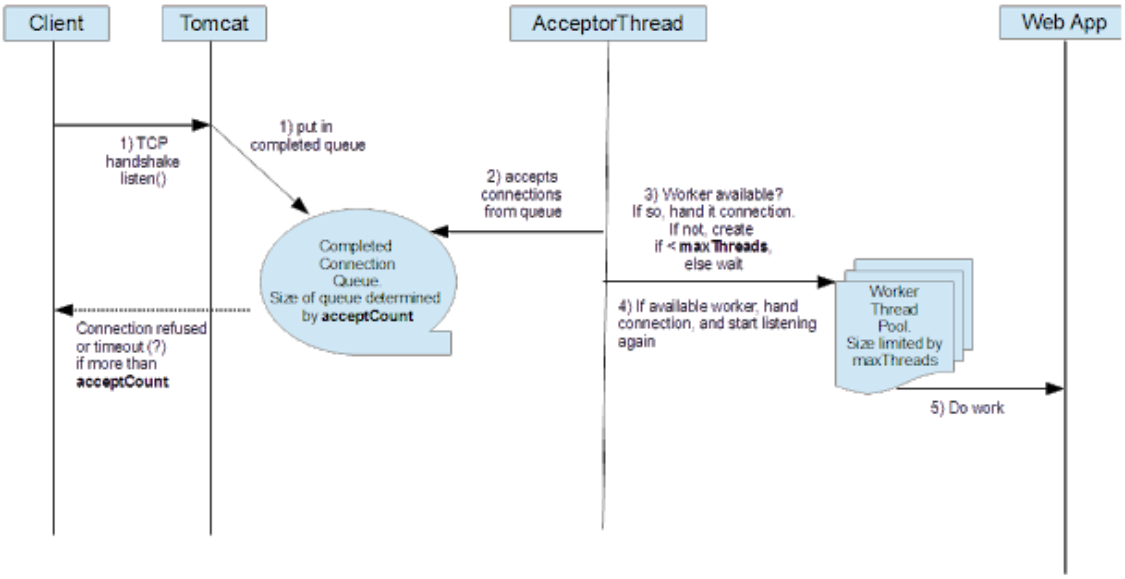
4.4 Zpracování HTTP požadavků

Webové mikroservisní aplikace v Javě jsou často založeny na některé z existujících technologií, ať už se jedná o aplikační servery definované v Java EE, servlet kontejnery či přímo na HTTP serverech za použití dalších frameworků. Tato podkapitola se zabývá modely zpracování příchozích požadavků z hlediska konkurence a představuje vybrané asynchronní HTTP servery, které mohou sloužit jako základ plně reaktivních mikroservis.

4.4.1 Blokační model

Klasický vláknový model webové Javy založené na specifikaci *javax.Servlet* definuje tzv servlety. Jedná se typicky o vlastní implementace, které dědí z abstraktní třídy *HttpServlet*. Ta obsahuje jednotlivé metody pro zpracování konkrétních HTTP požadavků. Tyto servlety jsou poté registrovány do některého z tzv. servlet kontejnerů, například populárního open-source kontejneru Apache Tomcat. Ten mimo jiné obsahuje HTTP konektor, zabezpečuje životní cyklus servletů a mapování příchozích požadavků.

Pro zpracování každého požadavku je vytvořeno či znovu použito vlákno, které je mu dedikováno až do jeho zpracování. Pokud jsou součástí jeho obsluhy časově náročné operace, které zavádějí latenci (například práce s DB), je celé vlákno zablokováno do jejich dokončení. V případě, že existuje více požadavků, než je maximální povolený počet vláken, je jejich zpracování pozdrženo do doby uvolnění některého vlákna. Pokud počet takto pozdržených požadavků překročí nastavené maximum, další požadavky jsou odmítnuty. Tento proces je znázorněn na obrázku 12.



Obrázek 12 - Zpracování požadavků v Apache Tomcat

Zdroj: [Techie, 2016]

Tento model zpracování má své výhody i nevýhody. Z hlediska uživatelského kódu v rámci servletu je jednodušší, jelikož je z principu synchronní a veškeré operace probíhají v jednom vlákně, pokud není explicitně specifikováno jinak. Nevýhodu je pak nutnost vytvářet velké množství vláken. S jejich počtem se typicky snižuje responzivita celé aplikace, zvyšuje se využití systémových prostředků kvůli změnám kontextu a také lze narazit na limity dané JVM či operačním systémem.

4.4.2 Asynchronní model

Asynchronní model na rozdíl od vláknového nevytváří pro každý příchozí požadavek nové vlákno. Místo toho definuje fond vláken, které jsou dedikovány obsluze těchto požadavků. Jejich úkolem je přijmou a předat požadavek ke zpracování dále, ať už v rámci jejich plánovače či jiného, pokud se jedná o blokující operace. Tato vlákna nesmějí být nikdy dlouhodobě blokována, jelikož by aplikace nebyla schopna přijímat nové požadavky.

Výhodou tohoto modelu je, že je možné využít přirozené latence v rámci celého systému. Ty jsou nejčastěji způsobeny I/O operacemi, jako jsou síťová komunikace a práce s databází. Vlákna v tomto případě nemusejí čekat na dokončení operace, ale mohou konkurentně zpracovávat další požadavky.

Nevýhodou tohoto modelu je pak vyšší zodpovědnost na straně programátora. Také je tento průběh zpracování obtížněji uchopitelný, zejména pro programátory, kteří mají zažitý blokační model. Níže jsou představeny některé technologie, které jsou využívány jako základ pro tvorbu reaktivních mikroservis v jazyce Java.

4.4.3 Servlet 3.1

Již specifikace Servlet 3.0 podporuje asynchronní zpracování požadavků v podobě rozhraní *AsyncContext*. Obsluha požadavků tak může být delegována do samostatného fondu vláken, který je nezávislý na kontejneru. Avšak teprve specifikace Servlet 3.1 přidala podporu pro asynchronní I/O. Z uživatelského pohledu se jedná zejména o rozhraní *ReadListener* a *WriteListener*, které umožňují registrovat callbacky, které jsou volány v okamžiku, kdy jsou k dispozici nová data (respektive v okamžiku možnosti jejich zápisu). Díky tomu nejsou vlákna zpracovávající I/O nadále blokována a mohou obsluhovat další požadavky. Ve spojení s *AsyncContext* tak lze Servlet 3.1 a vyšší využít k tvorbě plně asynchronních webových mikroservis.

4.4.4 Netty

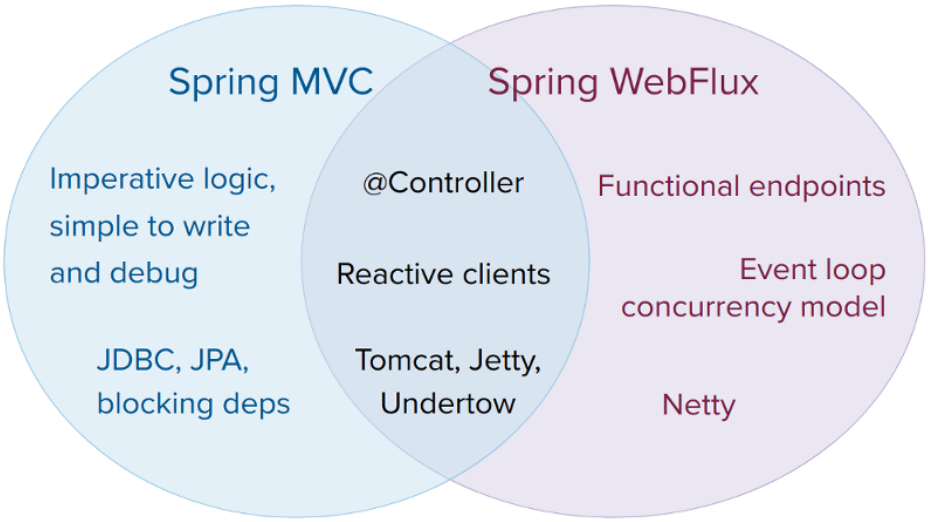
Netty je asynchronní, událostmi řízený síťový aplikační framework sloužící k rychlému vývoji snadno spravovatelných, výkonných severů a klientů [Netty project, 2019]. Nejedená se tedy přímo o HTTP server, ačkoliv nabízí podporu tohoto protokolu pomocí tříd *HttpRequestDecoder* a *HttpResponseEncoder*.

4.4.5 Undertow

Undertow je lehký, flexibilní, výkonnostní webový server napsaný v jazyce Java, podporující blokační i neblokační zpracování požadavků. Podporuje HTTP/2, Servlet 4.0 a je možné ho snadno zabudovat do samostatné aplikace. Jeho vývoj je sponzorován JBoss a slouží jako výchozí webový server v aplikačním serveru Wildfly [Red Hat Inc., 2019].

4.5 Spring WebFlux

Původní webový framework, který byl součástí ekosystému Spring, Spring MVC byl vytvořen specificky tak, aby podporoval Servlet API a Servlet kontejnery. Využívá tak blokační, vláknový model. Ve verzi Spring Framework 5.0 byla přidána jeho reaktivní alternativa, Spring WebFlux. Ten je založen na Rx a umožňuje ke svému běhu využít všechny výše uvedené asynchronní servery. Nejedná se o náhradu Spring MVC ale jeho doplněk. Vývojář si tak může vybrat, která varianta je vhodnější pro danou úlohu a typ aplikace [Spring, 2019e]. Jejich společné a rozdílné prvky jsou znázorněny na obrázku 13.



Obrázek 13 - Porovnání Spring MVC a Spring WebFlux

Zdroj: [Spring, 2019e]

Oba tyto FW jsou také plně podporovány v rámci Spring Boot od verze 2.0. Ukázky kódu v této kapitole byly vytvořeny autorem na základě oficiální dokumentace popisovaných technologií.

4.5.1 Definice koncových bodů

WebFlux podporuje 2 hlavní přístupy k definování koncových bodů pro zpracování příchozích požadavků, funkcionální a anotační. Funkcionální přistup je založen na rozhraní *RouterFunction<T extends ServerResponse>*. Implementaci tohoto funkcionálního rozhraní lze zaregistrovat do aplikačního kontextu jako standartní bean. K vytvoření instance se nejčastěji používá pomocná metoda

RouterFunctions.route(), která přijímá mapování v podobě *RequestPredicate* a dále implementaci funkcionálního rozhraní *HandlerFunction<T>*, která je zodpovědná za zpracování požadavku. Jednotlivé definice mapování lze dále řetězit například pomocí metody *and()*. Níže je ukázka REST API pro vytvoření a získání uživatele za pomoci tohoto přístupu.

```
@Configuration
public class FunctionalEndpoint {

  @Autowired
  private ReactiveRepository repository;

  @Bean
  public RouterFunction<ServerResponse> routers() {
   return
         route(GET("/user/{id}"), request -> {
           var id = request.pathVariable("id");
           return ok().body(repository.getUser(id), User.class);
         }).and(
         route(POST("/users"), request ->
               request.bodyToMono(User.class)
                     .flatMap(repository::saveUser)
                     .then(ServerResponse.ok().build())));
  }
}
```

Anotační API vychází z anotací používaných v rámci Spring MVC. Reaktivní koncové body je tak možné definovat obdobně, hlavním rozdílem je zde využití Rx datových typů.

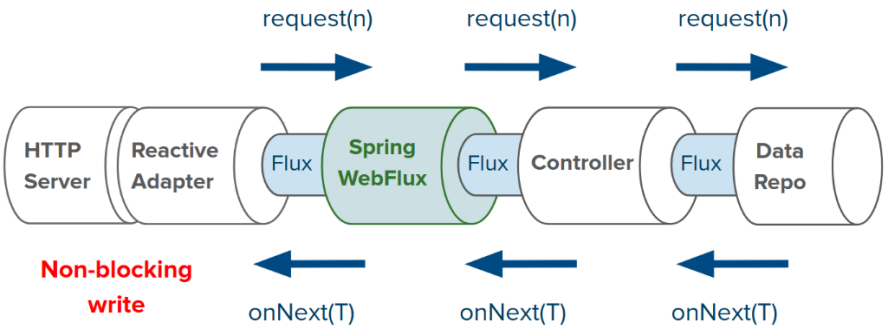
```
@RestController
@RequestMapping("/users")
public class AnnotationEndpoint {

  @Autowired
  private ReactiveRepository repository;

  @GetMapping("/{id}")
  public Mono<ResponseEntity<User>> getUser(@PathVariable String id) {
   return repository.getUser(id)
         .map(ResponseEntity::ok);
  }

  @PostMapping
  public Mono<ResponseEntity<Void>> createUser(
        @RequestBody Mono<User> user) {
   return user
         .flatMap(repository::saveUser)
         .thenReturn(ResponseEntity.ok().build());
  }
}
```

Nezávisle na použitém stylu je proces obsluhy stejný. Příchozí požadavek je přijat některým z dedikovaných vláken. Vybraná obslužná metoda je poté provolána v rámci sestavovací fáze, a získaná instance vydavatele je následně odebírána za pomoci push-pull modelu popsaného výše. Tento tok dat je znázorněn na obrázku 14. Počet obslužných vláken je závislý mimo jiné na použitém serveru, kdy nasazení v rámci servlet kontejneru může způsobit vytvoření více vláken než nasazení za použití reaktivního adaptéru, například Netty.



Obrázek 14 - Tok dat ve Spring WebFlux

Zdroj: [Rossen, 2018]

4.5.2 Zápis odpovědi

Způsob, kterým jsou data ze streamu odebírána a zapisována klientovi je odlišný na základě použitého *Content-Type*. V případě mikroservis využívajících rozhraní JSON se jedná nejčastěji o *application/json* a *application/stream+json*. Rozdíl mezi těmito typy lze znázornit na situaci, kdy odpověď obsahuje pole dat, například data uživatelů. Takový controller by měl signaturu

```
@GetMapping
public Flux<User> getAllUsers()
```

- ***application/json*** – Při použití toho typu budou emitovaná data postupně zapisována do výstupního bufferu. Po ukončení streamu bude tento buffer zapsán klientovi jako celek.
- ***application/stream+json*** – Použitím toho typu lze jednotlivé prvky zapisovat a odesílat klientovi samostatně. Lze tak snížit množství dat, která jsou v jeden čas uchována v paměti serveru i klienta.

Který z těchto přístupů je vhodnější záleží především na charakteru a množství dat, stejně tak jejich zpracování klientem. Pokud je schopný data postupně odebírat a dále zpracovávat po prvcích, může být vhodnější využít *stream+json*. Pokud však data pouze kompletuje pro další operace, může tento způsob představovat zbytečnou režii v podobě postupné serializace / deserializace.

4.5.3 Zpracování chyb

Aplikační výjimky lze rozdělit do 2 základních kategorií. Obecné chyby zpracování, které jsou způsobeny na straně serveru, například výpadkem závislé služby, databáze či nedostatečně otestovaným kódem. Tato kategorie lze reprezentovat pomocí HTTP kódů řady 5##. Druhým typem jsou pak výjimky business (doménového) charakteru, které mohou být vyvolány porušením validace dat, neexistující entitou, duplikací dat atd. Ty lze reprezentovat jako HTTP kódy řady 4##.

První zmiňovaná kategorie chyb může být zpracována a propagována klientovi přímo v rámci Rx streamu, pomocí metod pro práci s chybovými signály. Avšak pouze pokud jsou všechny chybové situace propagovány v rámci vykonávání streamu. Výjimky, které vzniknou v rámci sestavovací fáze tímto způsobem nemohou být zachyceny. Je proto vhodnější použít mapování chyb na vyšší úrovni, například pomocí anotace *ExceptionMapping* v rámci samostatné třídy anotované jako *ControllerAdvice*. Ta zachytí jak výjimky způsobené v rámci sestavovací fáze, tak výjimky, které jsou propagovány z obslužných metod pomocí chybových signálů. Tento způsob je znázorněn v následujícím příkladu.

```
@ControllerAdvice
public class RestControllerAdvice {

  @ExceptionHandler(RuntimeException.class)
  public Mono<ResponseEntity<ErrorResponse>> process500(
          RuntimeException e) {
   return Mono.just(ResponseEntity
           .status(HttpStatus.INTERNAL_SERVER_ERROR)
           .body(new ErrorResponse("UNKNOWN_ERR",
               e.getMessage())));
  }

  @ExceptionHandler(BusinessException.class)
  public Mono<ResponseEntity<ErrorResponse>> process400(
          BusinessException e) {
   return Mono.just(ResponseEntity
           .status(e.getState())
           .body(new ErrorResponse(e.getCode(),
               e.getMessage())));
  }
}
```

4.5.4 Testování

Existují různé úrovně testování webových aplikací, které se dělí podle přístupu a typu izolace, od základních jednotkových testů po globální testy chování celého systému v podobě end-to-end testů. Pro podporu testování Rx mikroservis v průběhu jejich vývoje je v rámci starteru *„spring-boot-starter-webflux“* dostupný reaktivní *WebTestClient*. Ten nabízí základní funkcionalitu pro sestavení, vykonání a ověření správnosti HTTP volání. Celý průběh testu lze vyjádřit pomocí funkcionálního API, včetně ověření správnosti JSON odpovědi pomocí JSONPath [Goessner, 2007]. Následující příklad znázorňuje test na uložení a následné získání správného uživatele.

```
@Test
public void createsNewUser() {
  webTestClient.post()
      .uri("/users")
      .syncBody(new User("Test User"))
      .exchange()
      .expectStatus().isOk();
  webTestClient.get()
      .uri("/users")
      .exchange()
      .expectBody()
      .jsonPath("$[?(@.name=='Test User')]").exists();
}
```

5 Praktické testy

Reaktivní přístup k tvorbě mikroservisních webových služeb na platformě Java existuje již několik let. Jelikož však velká část hobby i komerčních projektů využívá některou z komponent Spring Framework [JetBrains, 2018b], větší zájem o tuto technologii přišel s vydáním Spring Boot 2 s podporou Spring WebFlux. Teoretické výhody tohoto přístupu vyplývají z vlastností popsaných v kapitole 4. Jedná se zejména o lepší využití CPU v rámci síťových latencí a menší paměťovou náročnost díky streamování dat. Praktický přínos se však může lišit na základě požadavků každého projektu či jeho části. Tato kapitola popisuje testování kvantitativních vlastností této technologie v porovnání s blokačním modelem používaným v rámci Spring MVC.

5.1 Model

Aby bylo možné testovat reálné přínosy reaktivního zpracování, je třeba definovat obecné modelové scénáře, které reprezentují typické požadavky společné pro velké množství komerčních i nekomerčních projektů založených na mikroservisní architektuře. Mezi takové požadavky patří například obsluha základních CRUD operací, práce s persistentním úložištěm v podobě databázových systémů a kooperace mezi několika mikroservisami v rámci naplnění jednoho požadavku. Tato kapitola popisuje zvolený modelový systém a odvození jednotlivých testovacích scénářů.

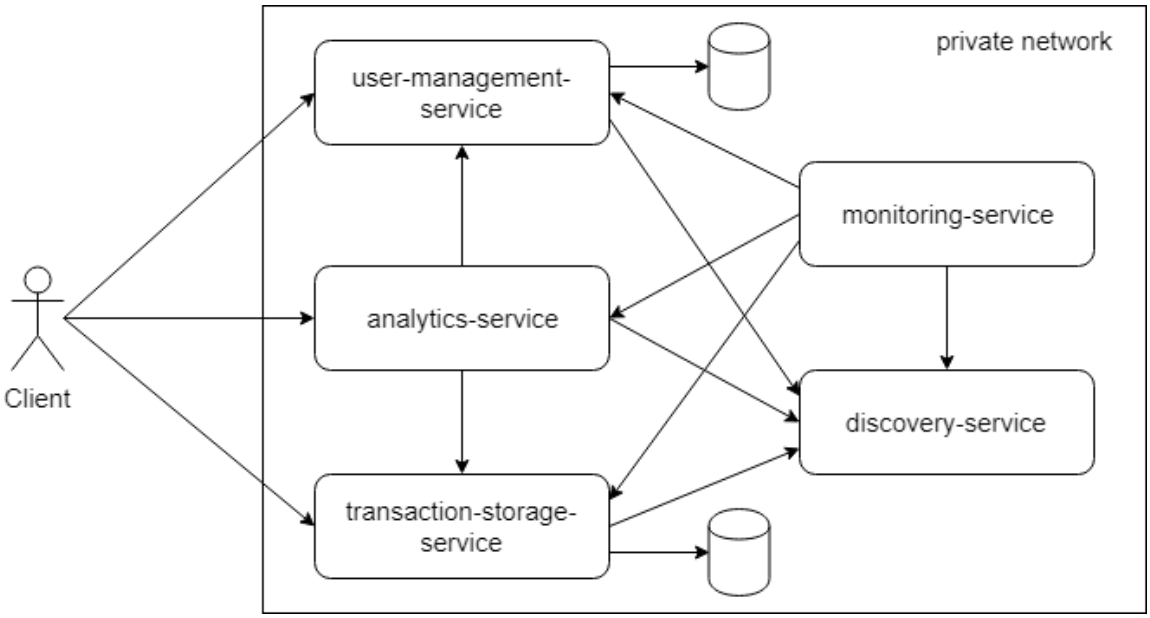
5.1.1 Modelový systém

Pro potřeby praktických testů je zaveden fiktivní systém pro správu osobních financí, konkrétně jeho subdoména pro správu uživatelů a dat finančních transakcí uživatelů. Na základě metodiky návrhu mikroservisních aplikací popsané v kapitole 3.2 byly definovány následující komponenty, včetně podpůrných služeb pro jejich provoz a monitoring:

- služba zajišťující service discovery, *discovery-service*
- služba pro monitoring aplikací *monitoring-service*
- služba pro správu dat uživatelů *user-management-service*

- služba pro persistenci dat transakcí *transaction-storage-service*
- služba pro zpracování analytických dat *analytics-service*

Služby *user-management-service* a *transaction-storage-service* jsou dále napojeny na persistentní úložiště dat. Služba *analytics-service* pak tyto využívá jako zdroj dat pro jednotlivé operace (metody). Blokový diagram jednotlivých služeb a jejich vzájemná komunikace je znázorněna na obrázku 15. Klientem v tomto modelu muže být například webová či mobilní aplikace nebo externí systém.



Obrázek 15 - Blokový diagram modelového systému

Zdroj: Autor

Klient může s jednotlivými službami komunikovat napřímo. Pro potřeby testování není aplikováno žádné zabezpečení, v reálném nasazení by však mohla být službám například předsazena vstupní brána, která by provedla autentizaci klienta na základě dat z *user-management-service*.

5.1.2 Testovací případy

Interakci mezi klientem a systémem lze z hlediska náročnosti na zpracování dat, množství přenášených dat a množství systémů zapojených do vykonání požadavku rozdělit do několika kategorií, které jsou blíže popsány níže. Pro každou z nich je vytvořen základní testovací případ v rámci daného modelového systému, který lze podrobit testování.

Nepersistentní

Požadavky z této kategorie nepotřebují ke svému zpracování žádné persistentní úložiště ani napojení na další služby. Mohou tak být vyřízeny pouze na základě dat, které jsou zaslány v rámci požadavku.

V modelovém systému je tato kategorie reprezentována požadavkem na ověření identity a platnosti zaslaného JWT token v rámci *user-management-service*.

Persistentní

Základní požadavky na manipulaci menších datových entit (CRUD), které lze plně obsloužit v rámci jedné služby napojené na persistentní úložiště. Jedná se o relativně nenáročné operace, které však mohou tvořit podstatnou část rozhraní systému a jsou volány opakovaně.

V modelovém systému se jedná o požadavky na správu uživatele v rámci *user-management-service*. Klient může vytvářet nové uživatele a získávat jejich data na základě unikátního identifikátoru, uživatelského jména.

Objemové

Jedná se o požadavky na získání velkého množství datových entit z jednoho či více zdrojů v řádech stovek až tisíců, které jsou reprezentovány jako pole. Požadavky z této kategorie představují vyšší zátěž na úložiště a v závislosti na způsobu jejich zpracování také vyšší paměťovou náročnost.

V modelovém systému se jedná o získání dat finančních transakcí pomocí volání *transaction-storage-service*. Klient může získat seznam všech transakcí pro daného uživatele.

Kooperativní

Pro obsloužení požadavků z této kategorie je nutné využít komunikaci mezi více servisními aplikacemi v rámci jedné domény. Může se jednat o libovolnou kombinaci dílčích požadavků z předešlých kategorií v závislosti na konkrétní operaci.

V modelovém systému se jedná o získání analytických dat pomoci volání *analytics-service*. Klient může získat anonymní statistiku o průměrné útratě všech

uživatelů ve zvoleném věkovém rozmezí. Je tedy nutné získat data z *user-management-service* a *transaction-storage-sevice*, ze kterých se následně provede výpočet.

5.1.3 Optimalizace

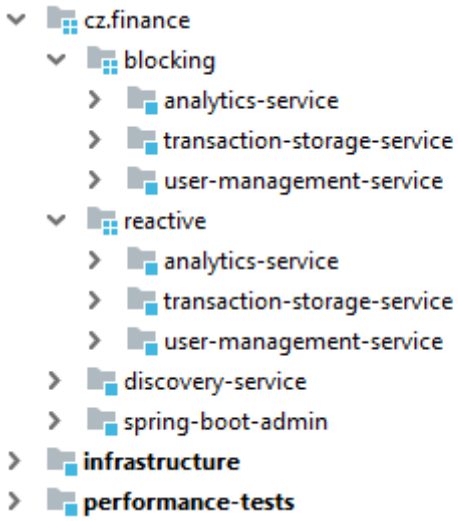
V mikroservisní architektuře lze použít různé metody optimalizace rychlosti zpracování dat. Může se jednat například o využití cache pro data často volaných služeb či jejich duplikace pro zpracování náročnějších požadavků. Každá taková metoda však zavádí nové problémy, které je třeba řešit. Například evikce a konzistence dat napříč službami. Jelikož se testovací model zaměřuje především na měření přínosu reaktivního zpracování dat, nejsou tyto optimalizační metody aplikovány.

5.2 Implementace

Pro potřeby testování byl implementován model systému pro správu financí, včetně všech metod potřebných pro naplnění testovacích případů ve dvou variantách. První variantou je kompletně blokační systém založený na Spring MVC a blokačním databázovém driveru, druhou variantou je pak kompletně reaktivní systém založený na Spring WebFlux a neblokačním databázovém driveru. V této kapitole jsou popsány jednotlivé technologie použité v rámci implementace modelového systému. Zdrojové kódy všech modulů jsou součástí digitální přílohy práce.

5.2.1 Struktura projektu

Každý modul je samostatná mikroservisní aplikace založená na jazyku Java 11, frameworku Spring Boot 2 a sestavovacím nástroji Gradle. Service discovery a monitorovací moduly (*spring-boot-admin*) jsou společné blokačním i reaktivním mikroservisám, jelikož se jedná pouze o podpůrné nástroje a mají minimální vliv na zátěžové testy. Jednotlivé verze testovaných komponent jsou hierarchicky členěny do adresářů na základě jejich typu. Obdobně mají také odlišná groupId - cz.*finance.blocking* a *cz.finance.reactive*. Zdrojové kódy dále obsahují modul pro správu infrastruktury a modul obsahující zátěžové testy. Hierarchie všech modulů je znázorněna na obrázku 16.



Obrázek 16 - Struktura modelového projektu

Zdroj: Autor

5.2.2 Spring Boot 2

Jako základ obou variant byl použit framework Spring Boot verze 2.1.4-RELEASE, dále také SB. Jedná se o nadstavbu nad projekty z platformy Spring ve spojení s dalšími knihovnami třetích stran. Byl vytvořen za účelem zjednodušení tvorby profesionálních aplikací na platformě Java, které lze provozovat bez nutnosti využití dalších technologií, jako jsou například servlet kontejnery. Aplikace takto vytvořené lze tedy distribuovat v podobě JAR či WAR balíčků, které jsou samostatně spustitelné [Spring, 2019b].

Základem SB je automatická konfigurace v závislosti na skenování knihoven, které jsou dostupné na *classpath* aplikace. To je realizováno pomocí tzv. autokonfigurací. V základu se jedná o standardní konfigurační beany anotované jako *@Configuration*, které musí být registrovány pomocí souboru *META-INF/spring.factories*. Ty mohou využít některé z podmíněných anotací, například *@ConditionalOnClass* nebo *@ConditionalOnBean*. SB je dodáván se souborem předpřipravených konfigurací pro velké množství nejčastějších případů užití v rámci knihovny *spring-boot-autoconfigure*.

Dále jsou poskytovány tzv. startery. Jedná se o soubory vzájemně kompatibilních závislostí, které jsou potřebné k implementaci konkrétní funkcionality. V modelovém systému jsou využity tyto startery:

- *spring-boot-starter-actuator*
- *spring-boot-starter-web*
- *spring-boot-starter-webflux*
- *spring-boot-starter-data-mongodb*
- *spring-boot-starter-data-mongodb-reactive*
- *spring-cloud-starter-netflix-eureka-server*
- *spring-cloud-starter-netflix-eureka-client*

Všechny moduly lze spustit jako standardní Spring Boot aplikace buď pomocí nástroje Gradle s využitím *spring-boot-plugin* příkazem *„gradlew bootRun“* po buildu aplikace pomocí příkazu *„gradlew build*“, jako standardní JAR příkazem *„java -jar build/libs/${service-name:version}.jar*“ nebo v libovolném IDE pomocí metody *main* ve třídě *„*Application.java*“.*

5.2.2.1 Actuator

Spring Boot Actuator je sada předpřipravených koncových bodů, které slouží pro monitoring a interakci s instancemi jednotlivých služeb. Ty jsou vystaveny na URL, kterou lze zvolit pomocí property *management.web.enpoits.base-path*. Výchozí hodnota je *„/actuator“*. Pomocí konfigurace lze také zvolit, které koncové body budou aktivní. Pro potřeby testování byly zapnuty všechny dostupné, zejména:

- */health* – zobrazuje informace o zdraví aplikace
- */beans* – zobrazuje všechny načtené beans v rámci jednotlivých kontextů
- */loggers* – zobrazuje dostupné loggery a umožňuje měnit úroveň logování
- */metrics* – zobrazuje základní metriky instance a serveru
- */env* – zobrazuje a umožňuje modifikovat proměnné prostředí

Uživatel může také vytvořit a zaregistrovat vlastní koncové body pro monitoring či správu dodatečných funkcionalit aplikace.

5.2.3 Spring Cloud

Spring Cloud je sada knihoven a nástrojů pro usnadnění implementace základních návrhových vzorů, používaných v rámci vývoje a provozu distribuovaných systémů.

Mimo jiné obsahuje podporu pro správu konfigurací, service-discovery, routování, micro-proxy a mnoho dalších [Spring, 2019c]. V projektu je využita závislost *spring-cloud-dependencies* ve verzi *Greenwich.SR1*.

5.2.3.1 Netflix Eureka

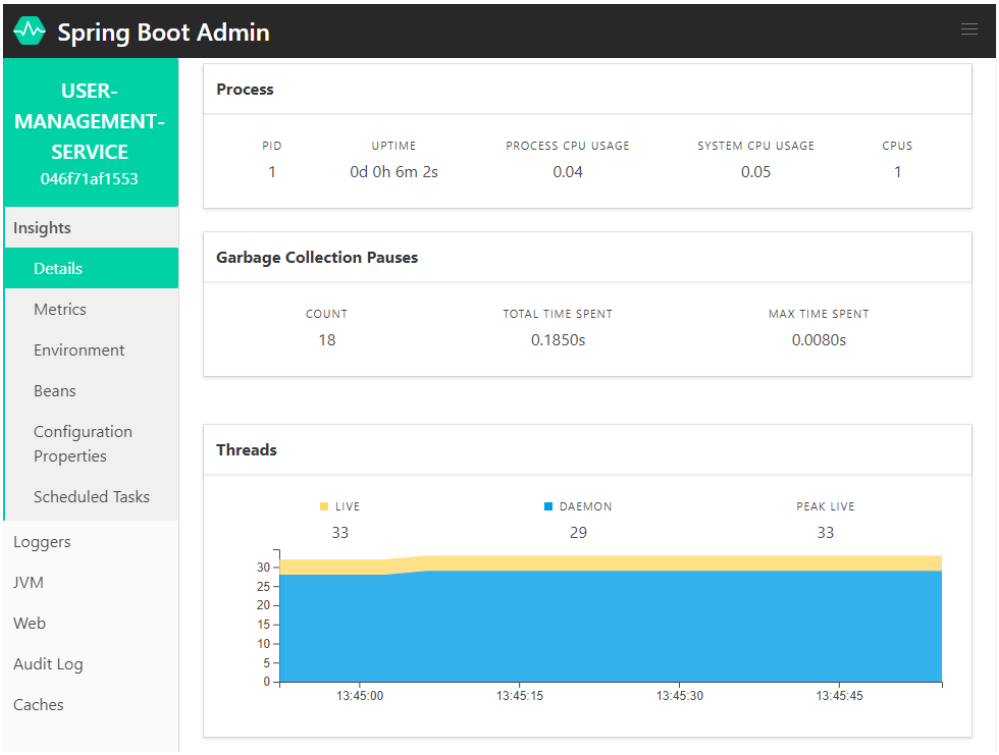
V projektu použitá service discovery služba Netflix Eureka je součástí platformy Netflix Open Source Software [Netflix, 2016]. Integrace do aplikací založených na platformě Spring je dostupná v podobě projektu *spring-cloud-netflix*.

Základem je webová služba, kterou lze vytvořit jako standardní Spring Boot aplikaci, anotovanou pomocí *@EnableEurekaServer* s příslušnou konfigurací. Přestože tato služba podporuje mnoho pokročilých funkcionalit jako je například replikace, v projektu je využita pouze základní funkcionalita pro usnadnění nasazení bez potřeby konfigurace adres jednotlivých služeb v rámci systému.

Na straně klienta, kterým je daná mikroservisní aplikace, je nutné použít anotaci *@EnableDiscoveryClient* a nastavit adresu, na které je dostupný Eureka server. Klientská aplikace se pak v rámci svého startu zaregistruje do repositáře pod svým jménem (property *spring.application.name*). Dále zasílá pravidelné notifikace, tzv. heartbeat, kterými značí, že je daná instance stále aktivní. Pokud tyto notifikace přestane zasílat, bude Eurekou považována za neaktivní a bude vyhoštěna.

5.2.4 Spring Boot Admin

Spring Boot Admin, dále také SBA, je komunitní projekt poskytující administrátorské rozhraní pro aplikace založené na Spring Boot. Obdobně jako v případě Netflix Eureka lze server vytvořit jako standardní SB aplikaci anotovanou pomocí *@EnableAdminServer*. Tento server pak poskytuje přehledné GUI, ve kterém jsou zobrazeny všechny registrované instance služeb spolu s informacemi, které jsou čerpány z jednotlivých koncových bodů v rámci Spring Boot Actuator. Také umožňuje přímo editovat podporované hodnoty, například proměnné prostředí či úroveň jednotlivých loggerů. Na straně klienta pak stačí přidat závislost na *spring-boot-admin-starter-client.* Ukázka administračního rozhraní je znázorněna na obrázku 17.



Obrázek 17- Ukázka rozhraní Spring Boot Admin

Zdroj: Autor

SBA lze také integrovat s libovolnou implementací Spring Cloud Discovery. Pokud je do Admin serveru přidána anotace *@EnableDiscoveryClient* a nastavena URL na repositář, bude automaticky použit pro identifikaci aktivních mikroservis a na klientech nemusí být použita závislost *admin-starter-client*. Tato konfigurace je použita v modelovém systému.

5.2.5 MongoDB

Jako persistentní úložiště je využita dokumentová databáze MongoDB, představená v kapitole 3.5.2. Jedná se o jednu z nejvyužívanějších DB tohoto typu a poskytuje klienta pro jazyk Java v blokační i reaktivní verzi. Lze tak přímo porovnat přínosy jednotlivých programovacích modelů bez nutnosti změny úložiště, která by mohla zanést nepřesnosti do měření.

Pro mapování doménových entit na dokumenty je použit *spring-data-mongo*, opět v blokační (*spring-boot-starter-data-mongodb*) i reaktivní (*spring-boot-starter-data-mongodb-reactive*) podobě. Veškeré operace s persistentní vrstvou jsou

realizovány pomocí jednotlivých repositářů v příslušných balíčcích „**.repository*“. Entitní modely jsou pak umístěny standardně v balíčcích „**.entity*“.

5.2.6 Feign

K implementaci klientů mikroservisních aplikací byl využit projekt [OpenFeign, 2019]. Ten umožňuje deklarativně definovat jednotlivé klienty a jejich metody pomocí Java rozhraní a sady základních anotací. Na jejich základě je vygenerována implementace, kterou lze registrovat do Spring kontextu a injektovat do dalších vrstev aplikace. Níže je ukázka definice klienta s metodou pro získání seznamu všech transakcí.

```
@FeignClient(FeignConfig.TRANSACTION_STORAGE_NAME)
@Headers({ "Accept: application/json" })
public interface TransactionStorageClient {

    @GetMapping("/transactions")
    List<Transaction> getAllTransactions();
}
```

Spring Boot nabízí integraci pro tento projekt v rámci autokonfigurací, včetně dodatečné podpory anotací ze Spring MVC. Klienta tak lze deklarovat obdobně jako koncové body dané služby, což zlepšuje přehlednost kódu.

Jelikož tento projekt v době tvorby této práce nepodporuje reaktivní webové klienty, jako alternativa byla v reaktivních službách využita knihovna *feign-reactive* [Playtika, 2019].

5.3 Infrastruktura

Pro simulaci reálného nasazení celého systému byly všechny testy provedeny na cloudové infrastruktuře v rámci Amazon Web Services, blíže představené v kapitole 3.6.2.1. Pro potřeby zátěžových testů byla vytvořena izolované sekce označovaná jako Virtual Private Cloud, *finance-services-vpc*, která je hostována v regionu eu-west-1 (Irské datacentrum). Ta obsahuje jednu podsíť s názvem *finance-subnet.* Instance v této podsíti jsou dostupné z internetu pomocí přiřazené internetové brány a příslušných cest. Dále byly vytvořeny 2 bezpečností skupiny:

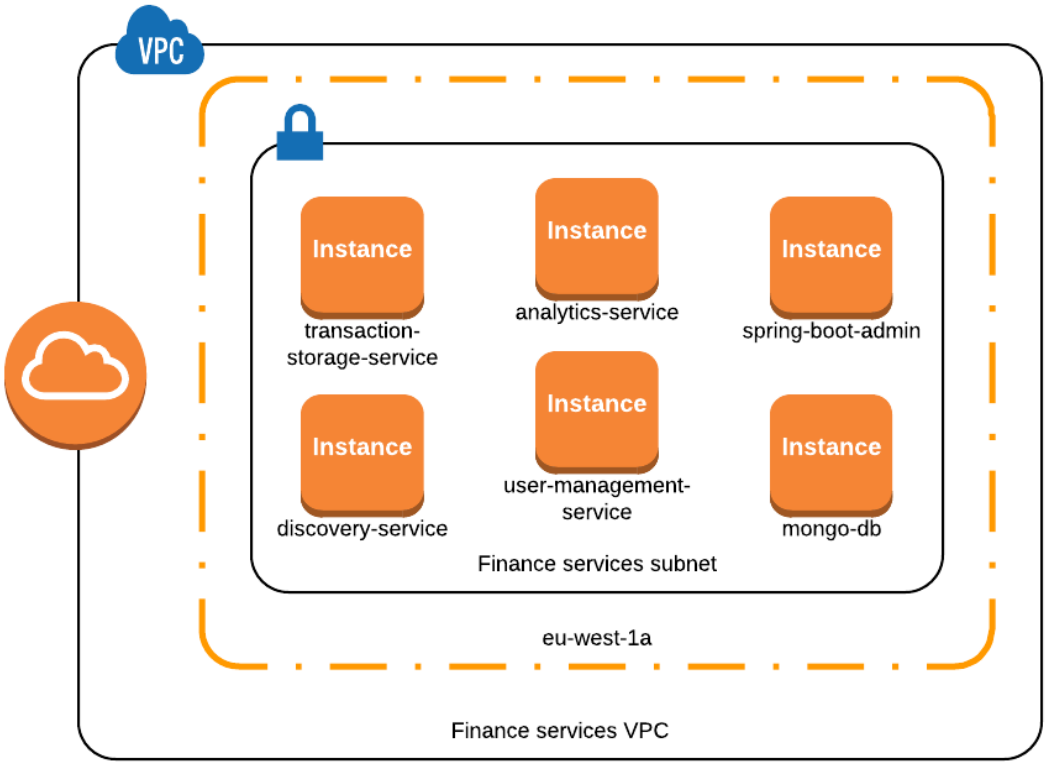
- ***finance-public-sg***, která povoluje veškerou interní komunikaci uvnitř skupiny a dále externí komunikaci pro standartní porty a protokoly (HTTP,

SSH, ICMP, MongoDB). Tato skupina je přiřazena všem instancím založeným na Spring Boot.

- ***finance-mongo-sg***, která slouží pro instanci databáze MongoDB a povoluje pouze vnější připojení na portech 27017 (MongoDB) a 22 (SSH).

Pro hostování jednotlivých služeb byly vytvořeny 2 obrazy AMI založené na Linuxové distribuci Ubuntu 18.4. První obsahuje mimo standardní utility také kontejnerizační systém Docker, druhý obsahuje připravené MongoDB s předdefinovaným zabezpečením. Aby byla minimalizována možnost soupeření více služeb o zdroje serveru, je pro každou mikroservisní aplikaci včetně DB vytvořena samostatná EC2 instance.

Aby nebylo nutné po každém znovuvytvoření instancí měnit IP adresy pro discovery-service a MongoDB, byly vytvořeny příslušné DNS záznamy v rámci služby Route 53, které jsou následně směrovány na IP adresy konkrétních EC2 instancí. Diagram této infrastruktury je znázorněn na obrázku 18.



Obrázek 18 - Infrastruktura testovacího modelu

Zdroj: Autor

5.3.1 Terraform

Pro správu cloudové infrastruktury, zejména jejího snadného vytvoření, editace typů instancí a následného zničení všech prostředků, byl využit nástroj Terraform [HashiCorp, 2019]. Ten je založen na metodice „infrastructure as code“, kdy lze celou infrastrukturu systému definovat pomocí kódu, který lze následně uložit spolu se zdrojovými kódy aplikací do repositáře verzovacího systému.

Základem nástroje Terraform je jazyk HCL, který je využit napříč nástroji od společnosti HashiCorp. V něm lze definovat konfiguraci za pomocí jednotlivých poskytovatelů, kteří slouží jako abstrakce pro konkrétní podporované služby (AWS, MS Azure, GCP…). Konfiguraci je také možné vyjádřit pomocí JSON, avšak tato metoda je doporučená spíše pro potřeby strojového zpracování / generování kódu. Následující příklad ukazuje definici instance EC2 pro službu *analytics-service*.

```
resource "aws_instance" "analytics-service" {
  ami = "${lookup(var.amis, "docker")}"
  instance_type = "t2.micro"
  subnet_id = "${aws_subnet.finance-subnet.id}"
  vpc_security_group_ids = ["${aws_security_group.finance-public-sg.id}"]
  associate_public_ip_address = true
  tags {
    Name = "analytics-service"
  }
}
```

Po definování či změně infrastruktury ji lze vytvořit pomocí příkazu „*terraform apply*“, který nejprve provede analýzu grafu závislostí, které je třeba vytvořit, editovat či odstranit vzhledem k aktuální konfiguraci daného poskytovatele, zjištěné pomocí jeho API. Odstranění všech využitých prostředků lze pak provést pomocí příkazu „*terrafrom destroy*“. Citlivé údaje, kterými jsou typicky kryptografické klíče či hesla lze externalizovat do příslušných souborů proměnných *tfvars*, které nejsou uloženy do repositáře a je možné je přiložit k danému příkazu pomocí modifikátoru „*--var-file*“. Tento přístup je využit v modelovém projektu. Zdrojové kódy obsahují veškerou deklaraci infrastruktury vyjma klíče pro SSH autorizaci k jednotlivým instancím a přístupu k AWS účtu.

5.3.2 Docker

Nástroj Docker slouží pro vytváření a správu Linux kontejnerů (LXC). Jedná se o odlehčenou verzi virtualizace, kdy je možné jednotlivé aplikace zabalit do

samostatných, izolovaných jednotek, které lze spustit v rámci hostitelského OS. Hlavní výhodou tohoto přístupu je jednotné prostředí pro každou aplikaci včetně potřebných závislostí ve všech stupních vývoje, od počítače vývojáře po nasazení do produkčního prostředí. Odpadají tak problémy s konfigurací prostředí pro potřeby dané aplikace a nekonzistence mezi jednotlivými servery [Docker Inc., 2019].

Základem jsou tzv. images, neboli obrazy systému včetně dané aplikace (či aplikací), ze kterých lze vytvořit živé kontejnery. Využívá se zde princip vrstvení, kdy lze nové obrazy vytvářet na základě již existujících. Například aplikace napsaná v jazyce Java vyžaduje ke svému spuštění JRE, obraz pro ni lze tedy vytvořit jako další vrstvu nad již existujícím obrazem, který obsahuje JRE požadované verze. Součástí ekosystému Docker je také platforma Docker Hub, která slouží jako repositář obrazů. Umožňuje nahrávat jak veřejné, tak privátní obrazy.

Pro potřeby modelového systému byl vytvořen veřejný repositář *vitjouda/thesis-finance*, který obsahuje obrazy všech nasazovaných služeb. Ty jsou od sebe odlišeny pomocí tagu se jménem dané služby. Tyto obrazy jsou vytvořeny jako součást build procesu pomocí Google Jib Gradle plugin [Google Container Tools, 2019]. Ten umožňuje vytvořit obraz bez nutnosti instalace Dockeru na hostitelském PC a následně jej nahrát na Docker Hub. Pro nasazení na daný server pak stačí lokálně čí vzdáleně pomocí SSH vytvořit kontejner z tohoto obrazu pomocí příkazu „*docker run*“ spolu s parametry pro mapování potřebných portů. Pro všechny služby modelového systému byl jako základní obraz využit *openjdk:11-jre*.

Konfigurace jednotlivých služeb je předávána jako proměnné prostředí daným kontejnerům a načtena do aplikačního kontextu díky podpoře Spring Boot Relaxed Binding. Ta umožňuje nastavit či přetížit jednotlivé aplikační properties pomocí jmenné konvence pro proměnné prostředí. Například pro nastavení portu aplikace se jedná o parametr „*-e SERVER_PORT=80*“. Tento postup postačuje pro testovací účely, v reálném nasazení by však bylo lepší využít například dynamické načtení konfigurace pomocí Spring Cloud Config [Spring, 2019d].

5.3.3 Gitlab

Pro hostování Git repositářů všech mikroservisních aplikací a nástrojů v rámci testovacího modelu je využita služba Gitlab. Ta poskytuje mimo hostingu také

nástroje pro continuous integration. Pomocí souboru „*.gitlab-ci.yml*“ lze definovat jednotlivé kroky (stages), které se vykonají v daném pořadí buď automaticky po commitu, nebo manuálním spuštěním. Pro zajištění aktuálních verzí obrazů jednotlivých služeb byla vytvořena pipeline, která provede build celé aplikace, následně vytvoří příslušný obraz a provede jeho nahrání na Docker Hub. Spuštění této pipeline je provedeno na veřejných Docker runnerech. Níže je uveden příklad deklarace vytvoření a nahrání obrazu v master větvi. Kompletní konfigurace je součástí zdrojových kódů, přístupové údaje jsou nahrány jako proměnné prostředí v Gitlab.

```
dockerHub:
  stage: publish
  script:
    - ./gradlew jib
        -Djib.to.auth.username=$DOCKER_USERNAME
        -Djib.to.auth.password=$DOCKER_PASSWORD
  only:
    - master
```

5.4 Metodika testování

Tato kapitola popisuje nástroje a metodiku použitou při provedení jednotlivých testů v rámci praktické části práce.

5.4.1 Testovací simulace

Pro každý testovací případ popsaný v kapitole 485.1.2 byla vytvořena odpovídající simulace pomocí nástroje Gatling. Ten umožňuje spouštění testů vytvořených pomocí vlastního DSL, který je založen na jazyce Scala. Dále sleduje průběh jednotlivých testů, sbírá základní metriky a umožňuje generovat přehledné reporty [Gatling Corp, 2019]. Každá simulace byla spuštěna pro 250, 1 000 a 5 000 simultánních uživatelů (průběhů celým scénářem), představující rozdílné typy zátěže.

Ověření tokenu

Pro simulaci ověření tokenu byl vygenerován jeden platný a jeden neplatný Json Web Token, dále JWT. Ty jsou zasílány v poměru 9:1, což představuje 10% chybovost.

```
val TOTAL_USERS = 250;
val ERR_RATE = 0.1;
val invalidUsers = (TOTAL_USERS * ERR_RATE).toInt;
val validUsers = TOTAL_USERS - invalidUsers;

val USER_MANAGEMENT_URL = "http://localhost"
val VERIFY_TOKEN_URL = "/users/verification"

val VALID_JWT_TOKEN = ""
val INVALID_JWT_TOKEN = ""
val httpProtocol = http
  .baseUrl(USER_MANAGEMENT_URL)
  .acceptHeader("application/json")

val validToken = scenario("ValidTokenScenario")
  .exec(http("verify")
    .get(VERIFY_TOKEN_URL)
    .queryParam("token", VALID_JWT_TOKEN))

val invalidToken = scenario("InvalidTokenScenario")
  .exec(http("verify")
    .get(VERIFY_TOKEN_URL)
    .queryParam("token", INVALID_JWT_TOKEN))

setUp(
  validToken.inject(atOnceUsers(validUsers)),
  invalidToken.inject(atOnceUsers(invalidUsers))
).protocols(httpProtocol)
```

Vytvoření a získání uživatele

Simulace nejprve vytvoří uživatele s náhodným uživatelským jménem o délce 10 znaků, kterého uloží pomocí metody POST. Následně se na tohoto uživatele dotáže pomocí metody GET.

```
val USER_MANAGEMENT_URL = "http://localhost"

val httpProtocol = http
  .baseUrl(USER_MANAGEMENT_URL)
  .acceptHeader("application/json")

val usernameFeeder = Iterator.continually(
  Map("randUsername" ->
    (Random.alphanumeric.take(10).mkString)
  )
)
```

```
val scn = scenario("CreateGetUserScenario")
  .feed(usernameFeeder)
  .exec(http("create")
    .post("/users")
    .body(
      StringBody(
        """{
          "username": "${randUsername}",
          "name": "Mr",
          "surname": "Mock"
        }
        """))
  )
  .exec(http("get")
    .get("/users/${randUsername}"))

setUp(
  scn.inject(atOnceUsers(250))
).protocols(httpProtocol)
```

Získání všech transakcí

Simulace náhodně vybere jeden z limitů počtu transakcí: 1, 500 nebo 1 000. Poté zašle dotaz na získání všech transakcí s tímto limitem.

```
val TRANSACTION_STORAGE_URL = "http://localhost"

val httpProtocol = http
  .baseUrl(TRANSACTION_STORAGE_URL)
  .acceptHeader("application/json")

val limitFeeder = Array(
  Map("limit" -> 1),
  Map("limit" -> 500),
  Map("limit" -> 1000)
).random

val getAllScenario = scenario("GetAllTransactionsScenario")
  .feed(limitFeeder)
  .exec(http("getAll")
    .get("/transactions")
    .queryParam("limit", "${limit}"))


setUp(
  getAllScenario.inject(atOnceUsers(250)),
).protocols(httpProtocol)
```

Průměrná útrata

Simulace náhodně vybere jedno z následujících věkových rozmezí: 20-29, 30-39, 40-49. Poté zašle dotaz na výpočet průměrné útraty všech uživatelů v tomto věkovém rozmezí.

```
val ANALYTICS_URL = "http://localhost"

val httpProtocol = http
  .baseUrl(ANALYTICS_URL)
  .acceptHeader("application/json")

val ageFeeder = Array(
  Map("fromAge" -> 20, "toAge" -> 29),
  Map("fromAge" -> 30, "toAge" -> 39),
  Map("fromAge" -> 40, "toAge" -> 49)
).random

val averageSpending = scenario("AverageSpendingScenario")
  .exec(http("calculate")
    .get("analytics/spendings")
    .queryParam("fromAge", "${fromAge}")
    .queryParam("toAge", "${toAge}"))

setUp(
  averageSpending.inject(atOnceUsers(250)),
).protocols(httpProtocol)
```

5.4.2 Testovací data

Pro potřeby simulací byla vygenerována sada testovacích dat, která byla před jejich spuštěním nahrána do MongoDB. Data byla vždy shodná pro ekvivalentní testy mezi reaktivním a blokačním modelem tak, aby jejich případná změna nemohla mít vliv na naměřené hodnoty.

Testovací data byla vytvořena tak, aby mohla být použita pro všechny simulace. Obsahují tedy celkem 1 000 transakcí, které jsou rozděleny mezi uživatele věkových kategorií, jak je uvedeno v tabulce 1. Simulují tak nízké, střední a velké zatížení systému z pohledu objemu dat.

Tabulka 1 - Rozdělení testovacích dat

Věková kategorie	Počet uživatelů	Počet transakcí celkem
20-29	10	100
30-39	50	300
40-49	200	600

Zdroj: Autor

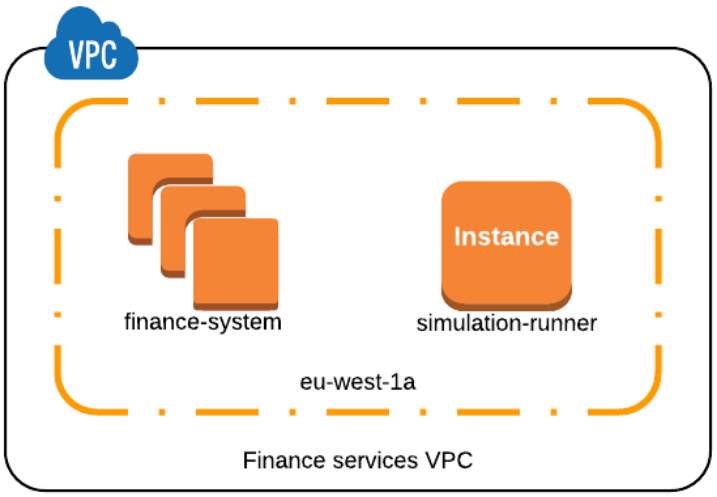
5.4.3 Warmup

Před všemi níže popsanými testy byl po každé změně prostředí či nasazení služeb nejprve proveden warmup celého systému tak, aby proběhly optimalizační procesy

v rámci JVM, které by mohly zkreslovat naměřené hodnoty. Toho bylo docíleno spuštěním všech relevantních testovacích simulací, aniž by byly ukládány naměřené hodnoty.

5.4.4 Testy bez vlivu síťové latence klienta

Cílem těchto testů je ověřit výkonnost modelového systému, který je kompletně reaktivní či kompletně blokační v případě, kdy mezi systémem a klientem existuje pouze zanedbatelná síťová latence. Všechny služby jsou založeny na stejné technologii včetně databázového driveru. Pro vykonání všech simulací byla vytvořena dodatečná instance typu t2.xlarge ve stejném regionu a zóně, jak je znázorněno na obrázku 19. Průměrná naměřená latence mezi touto instancí a testovacími službami byla menší než 0,5ms.



Obrázek 19 - Diagram testování bez síťové latence klienta

Zdroj: Autor

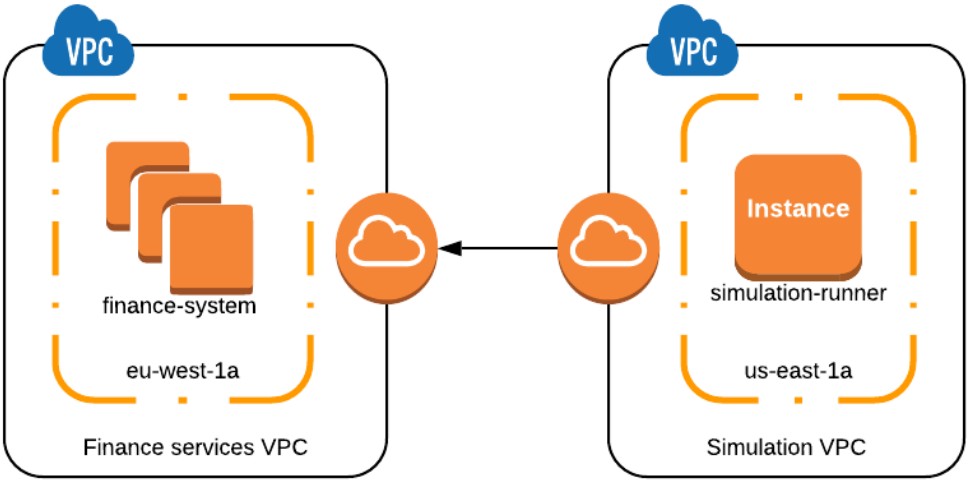
Nejprve byly spuštěny simulace s výchozí konfigurací všech služeb na instancích typu t2.micro, které jsou dostupné v rámci bezplatné úrovně AWS a obecně se jedná o jeden z nejdostupnějších typů. Následně byly všechny instance povýšeny na typ t2.large, který nabízí výrazně větší množství operační paměti a vyšší počet vCPU. Opět byly spuštěny všechny simulace, tak aby byl otestován vliv horizontálního škálování na jednotlivé typy systémů.

Za účelem určení vlivu maximálního počtu vláken pro zpracování požadavků byly testy blokačního modelu provedeny ve dvou variantách, se základním

a dvojnásobným limitem. Při testech reaktivního modelu docházelo k zahlcení fronty aktivních spojení s MongoDB, která má nastaven výchozí limit 500 spojení. Tyto testy byly zopakovány po jeho navýšení na 10 000 spojení.

5.4.5 Testy vlivu síťové latence klienta

Pro určení případného vlivu síťové latence mezi klientem a testovaným systémem byly následně provedeny vybrané simulace pomocí nově vytvořené instance typu t2.xlarge v regionu US East (N. Virginia), jak je znázorněno na obrázku 20. Spojení mezi touto instancí a všemi testovanými službami vykazovalo v průběhu simulací stabilní latenci v intervalu 66 - 71ms. Před spuštěním všech simulací byly nově nasazeny všechny služby v rámci testovaného modelu.



Obrázek 20 - Diagram testování s vlivem síťové latence klienta

Zdroj: Autor

5.4.6 Sledované parametry

Sledované parametry v rámci výkonnostních testů lze rozdělit do následujících kategorií, na základě jejich zaměření.

5.4.6.1 Odezva systému

Z hlediska klientů je důležitá především propustnost systému a rychlost zpracování požadavků. Pro posouzení těchto vlastností je sledována doba odezvy každého požadavku v závislosti na počtu simultánních uživatelů systému. Na základě těchto dat lze odvodit další statistiky – minimální, maximální a průměrná doba odezvy, maximální množství zpracovaných požadavků za sekundu a počet požadavků při

jejichž zpracování došlo k chybě či vypršení časového limitu. Tyto parametry byly sledovány na straně klienta (zátěžového testu) pomocí nástroje Gatling.

5.4.6.2 Využití systémových prostředků

Z hlediska využití systémových prostředků serveru je u všech služeb sledováno především vytížení CPU, využití operační paměti, počet aktivních vláken a údaje o četnosti spouštění garbage collectoru. Tyto parametry byly sledovány na straně serveru pomocí monitorovacího nástroje *spring-boot-admin* popsaného v kapitole 5.2.4 a nástroje VisualVM pomocí Java Management Extensions. Připojení na sledované instance bylo realizováno pomocí SSH tunelu.

5.5 Naměřené hodnoty

Naměřené hodnoty z jednotlivých dílčích simulací byly sesbírány a zpracovány v rámci souboru „*performance-tests/results.xlsx*“, který je součástí digitální přílohy práce. Tento dokument je rozdělen na 4 záložky, na základě AWS zóny testovací instance a druhu systému (reaktivní a blokační). Typ simulace označuje kategorii požadavků na základě kapitoly 5.1.2. Jednotlivé požadavky v rámci každé simulace byly na základě doby zpracování t rozděleny do následujících kategorií:

- nízká – $t < 800ms$
- střední – $800ms < t < 1200ms$
- vysoká – $t > 1200ms$

Data dále obsahují maximální hodnoty vytížení CPU a paměti typu heap v průběhu každé simulace. Podrobný rozbor těchto hodnot je obsažen v kapitole 6, spolu s poznatky získanými v průběhu měření.

6 Shrnutí výsledků

Tato kapitola se zabývá rozborem dat, které byly získány na základě provedených simulací. Výsledky zde prezentované se vztahují k verzím knihoven využitých v rámci implementace a jejich výchozímu nastavení, s výjimkou parametrizace maximálního počtu pracovních vláken pro blokační model, dále také BLM, a maximálního množství aktivních spojení s MongoDB pro reaktivní model, dále také RXM, a mohou být odlišné v jiných verzích či typech aplikace.

Typ simulace označuje jednotlivé kategorie požadavků, jak byly definovány v kapitole 5.1.2:

- **NP** – nepersistentní, bez volání DB či dalšího systému
- **P** – persistentní, volající DB
- **O** – objemové, navracejí velké množství entit
- **K** – kooperativní, pro své zpracování vyžadují interakci s dalšími službami

6.1 Testy bez vlivu síťové latence klienta

Naměřené hodnoty byly rozděleny do kategorií, které reprezentují základní požadavky na systém z hlediska spolehlivosti, rychlosti zpracování požadavků a využití systémových prostředků.

6.1.1 Rychlost zpracování

Pro porovnání rychlosti zpracování požadavků je sledována průměrná doba zpracování $\bar{t}$ a směrodatná odchylka σ.

t2.micro

Tabulka 2 obsahuje data o průměrné době zpracování požadavku na základě typu simulace a počtu uživatelů modelu nasazeného na instancích typu t2.micro ve výchozím nastavení. Při zpracování požadavků typu O a K pro 1000 a více uživatelů docházelo v RXM k zahlcení fronty aktivních spojení s MongoDB. Obdobně u některých požadavků na BLM docházelo k vypršení časového limitu při komunikaci uvnitř modelového systému. Odpovídající buňky jsou v tabulce

označeny stínováním. Tyto případy tak nelze objektivně porovnat z hlediska rychlosti zpracování a jsou blíže rozebrány v rámci kapitoly 6.1.2.

Tabulka 2 - Průměrná doba zpracování, t2.micro

Simulace		RXM		BLM	
Typ	Uživatelé	$\bar{t}$ [ms]	σ [ms]	$\bar{t}$ [ms]	σ [ms]
NP	250	59	34	14	18
	1000	16	16	13	12
	5000	261	449	625	713
P	250	285	157	245	91
	1000	589	487	628	370
	5000	2161	2991	1955	1494
O	250	1219	477	1112	493
	1000	2993	1642	2880	1720
	5000	8171	3559	15082	9613
K	250	1778	632	1057	380
	1000	5534	2464	3123	1330
	5000	21734	12490	9467	5529

Zdroj: Autor

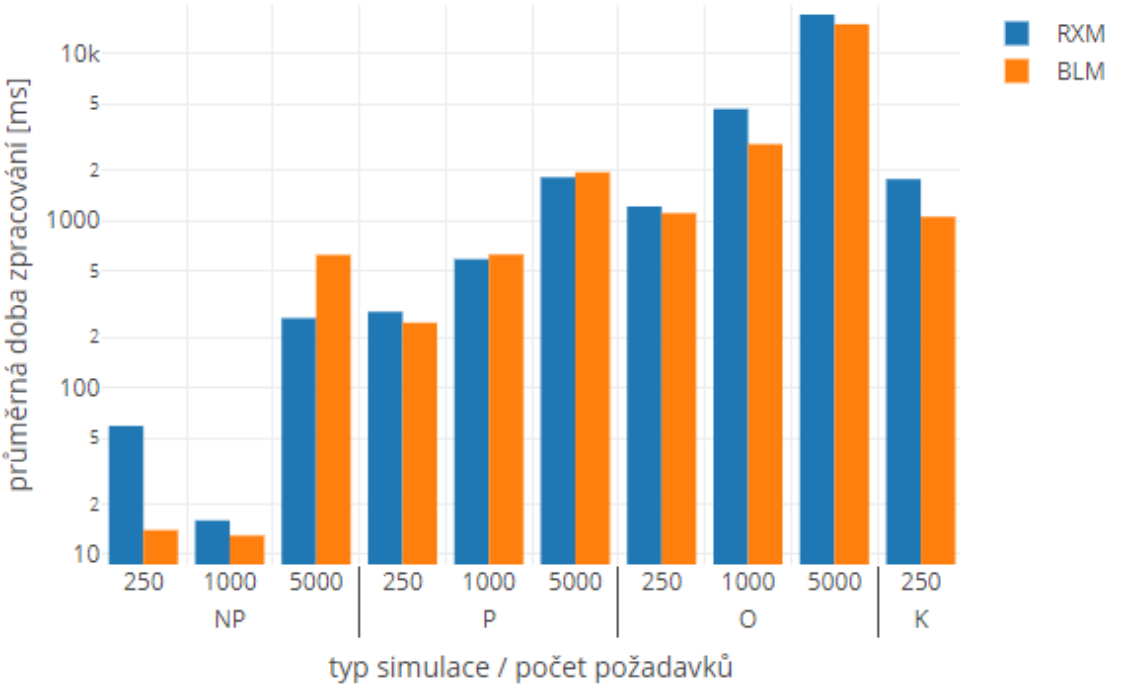
Pro další simulace byl odstraněn hlavní limitující faktor RXM. Maximální počet aktivních spojení s MongoDB byl navýšen na 10 000. Výsledky těchto simulací jsou zobrazeny v tabulce 3.

Tabulka 3 - Navýšený limit aktivních spojení s MongoDB, t2.micro

Simulace		RXM		BLM	
Typ	Uživatelé	$\bar{t}$ [ms]	σ [ms]	$\bar{t}$ [ms]	σ [ms]
P	5000	1823	2859	1955	1494
O	1000	4685	1976	2880	1720
	5000	17230	8090	15082	9613
K	1000	5614	2286	3123	1330
	5000	20365	11849	9467	5529

Zdroj: Autor

Po této úpravě je možné porovnat naměřené hodnoty obou systémů pro více typů požadavků. Pro přehlednost byla průměrná doba zpracování vynesena do grafu znázorněného na obrázku 21. Pro osu y bylo zvoleno logaritmické měřítko.



Obrázek 21 - Graf průměrné doby zpracování požadavků bez latence klienta

Zdroj: Autor

Z grafu vyplývá, že pro malý počet požadavků typu NP, blízký maximálnímu počtu aktivních vláken BLM (200), je tento model výrazně rychlejší. To odpovídá teoretickým předpokladům, jelikož jsou téměř všechny požadavky zpracovány paralelně a v systému nejsou žádné latence v podobě komunikace s DB. Při maximálním počtu 1 000 aktivních uživatelů je již rychlost zpracování obou modelů obdobná, jelikož vlákna BLM již nedostačují a vzniká tak latence v podobě pozastavení dalších požadavků. Pro 5 000 uživatelů je již výrazně rychlejší RXM.

U požadavků typu P je však i pro malý počet uživatelů (250) pouze zanedbatelný rozdíl mezi průměrnou dobou zpracování obou modelů, jelikož zde již vzniká síťová latence při požadavcích na DB, kterou RXM využívá pro zpracování dalších požadavků, zatímco BLM musí vyčkat na jejich dokončení. S rostoucím počtem uživatelů pak RXM dosahuje lepších hodnot.

Při zpracování požadavků typu O lze pozorovat, že BLM poskytuje ve všech případech lepších hodnot. To může být způsobeno tím, že pro přenos objemových dat ke klientovi byl využit typ obsahu „*application/json*“, který nepodporuje postupný stream dat a vlákno, které data zapisuje, tak nemůže zpracovávat další

požadavky. Požadavky typu K nelze kvůli velké chybovosti obou modelů objektivně porovnat.

V případech, kdy oba modely dosahují obdobné průměrné doby zpracování je směrodatná odchylka výrazně nižší u BLM, poskytuje tak stabilnější dobu zpracování. Z dat lze také pozorovat, že u obou modelů se snížila rychlost zpracování NP požadavků při zvýšení počtu uživatelů z 250 na 1 000. To je způsobeno tím, že pro nižší počet uživatelů nebyl plně vytížen vCPU dané instance, a tudíž nedosahoval maximálního výkonu. Jedná se o standardní chování instancí typu t2.

Pro další simulace byl navýšen maximální limit vláken pro BLM, na instancích typu t2.micro však docházelo k velkému vytížení či překročení maximální velikosti paměti typu heap. Následkem toho docházelo k častému spouštění garbage collectoru či došlo k úplnému výpadku služby a výsledky tudíž nejsou porovnatelné.

t2.large

Při testování na instancích typu t2.large docházelo v případě RXM ve výchozím nastavení opět k zahlcení maximálního počtu spojení s MongoDB. V tabulce 4 jsou zobrazeny hodnoty pro oba modely s navýšeným limitem spojení s DB a počtem aktivních vláken. I v tomto případě však docházelo při maximálním zatížení k chybám při komunikaci mezi jednotlivými službami v obou modelech, a proto je nelze porovnat. Již nejsou zahrnuty požadavky typu NP a simulace s 250 uživateli, jelikož nepředstavují dostatečnou zátěž pro tento typ instance a nastavené limity.

Tabulka 4 - Navýšené limity obou modelů, t2.large

Simulace		RXM		BLM	
Typ	Uživatelé	$\bar{t}$ [ms]	σ [ms]	$\bar{t}$ [ms]	σ [ms]
P	1000	689	328	505	275
	5000	921	548	1084	757
O	1000	2903	1415	3071	1882
	5000	6893	3679	7846	4941
K	1000	2468	939	2133	792
	5000	11774	7402	4954	3303

Zdroj: Autor

V případě zvýšeného limitu aktivních vláken, množství operační paměti a dostupných vláken vCPU lze pozorovat podobné výsledky jako v předchozích případech. Pro nižší počet uživatelů blízký zvýšenému limitu aktivních vláken

dosahuje BLM podobných či lepších hodnot než RXM. Se zvyšující se zátěží a počtem uživatelů pak lepších hodnot dosahuje RXM.

6.1.2 Spolehlivost

Pro určení spolehlivosti modelu jsou sledovány požadavky, které byly naplněny bez ohledu na dobu zpracování a požadavky, při jejichž zpracování došlo k jakékoliv chybě. Jelikož oba modely zpracovaly požadavky typu NP se 100% úspěšností, nejsou zahrnuty v zobrazených hodnotách.

t2.micro

Z hodnot zobrazených v tabulce 5 je zřejmé, že BLM ve výchozím nastavení poskytuje výrazně vyšší spolehlivost než RXM. To je způsobeno tím, že reaktivní služby jsou schopny zahltit databázový driver i přes jeho výchozí limit 500 aktivních spojení. Oproti tomu blokační služby, díky svému omezení na 200 aktivních vláken, slouží jako přirozený omezovač propustnosti.

Tabulka 5 - Spolehlivost s výchozími limity, t2.micro

Simulace		RXM		BLM	
Typ	Uživatelé	Naplněné [%]	Chybové [%]	Naplněné [%]	Chybové [%]
P	1000	100	0	100	0
	5000	96,41	3,59	100	0
O	1000	82,4	17,6	100	0
	5000	30,26	69,74	100	0
K	1000	79,1	20,9	65,8	34,2
	5000	48,96	51,04	83,68	16,32

Zdroj: Autor

Po navýšení limitu maximálních aktivních spojení s MongoDB se spolehlivost RXM výrazně zlepšila, jak lze vidět z tabulky 6. Stále však selhává při zpracování velkého množství kooperativních požadavků.

Tabulka 6 - Spolehlivost s navýšeným limitem spojení s MongoDB, t2.micro

Simulace		RXM		BLM	
Typ	Uživatelé	Naplněné [%]	Chybové [%]	Naplněné [%]	Chybové [%]
P	1000	100	0	100	0
	5000	100	0	100	0
O	1000	100	0	100	0
	5000	100	0	100	0
K	1000	100	0	65,8	34,2
	5000	53,24	46,76	83,68	16,32

Zdroj: Autor

t2.large

V případě nasazení modelu na instance typu t2.large a navýšení obou limitů dochází nadále k chybám při zpracování 5 000 aktivních uživatelů pro požadavky typu K. RXM v tomto případě dosahuje chybovosti 2,18%, BLM pak chybuje v 29,32% požadavků. Zajímavé také je, že v případě, kdy není navýšen limit vláken, BLM dosahuje při stejném počtu uživatelů a typu požadavku 100% naplnění. Ve všech případech se jedná o chyby na straně komunikace mezi službami na straně Feign klienta.

6.1.3 Využití systémových prostředků

Porovnávány jsou maximální hodnoty využití vCPU a paměti typu heap v průběhu simulace tak, jak byly reportovány aplikací VisualVM. Uvedeny jsou pouze hodnoty naměřené na službách, které sloužily jako koncový bod pro danou simulaci.

Tabulka 7 zobrazuje data ze simulací na instancích typu t2.micro se zvýšeným maximálním počtem spojení s MongoDB. Díky tomu lze porovnat paměťovou náročnost obou modelů pro více simulací, jelikož došlo ke zpracování všech požadavků, a tedy přenosu dat o odpovídající velikosti.

Tabulka 7 - Využití systémových prostředků, t2.micro

Simulace		RXM		BLM	
Typ	Uživatelé	CPU [%]	Heap [MB]	CPU [%]	Heap [MB]
NP	250	58	56	62,2	77
	1000	100	67	100	90
	5000	100	82	100	125
P	250	75	58	59	90
	1000	100	80	95,8	101
	5000	100	114	100	180
O	250	100	81	100	190
	1000	100	120	100	200
	5000	100	160	100	232
K	250	92	87	100	134

Zdroj: Autor

Oba modely jsou schopny při větším množství požadavků naplno vytížit vCPU dané služby. Drobné rozdíly jsou pozorovatelné pouze při nižší zátěži. Naopak je zřejmé, že BLM využíval více paměti téměř v každé simulaci, někdy i téměř o 200% více v porovnání s RXM. To je pravděpodobně způsobeno vyšší paralelizací zpracování dat. Obdobné výsledky lze pozorovat i u modelů nasazených na instancích t2.large.

6.2 Testy vlivu síťové latence klienta

Naměřené hodnoty byly stejně jako v předchozí kapitole rozděleny na základě rychlosti zpracování, spolehlivosti a využití systémových prostředků. Jelikož jde především o vliv latence, jsou porovnány na instancích typu t2.micro se zvýšeným limitem pro aktivní počet spojení s MongoDB.

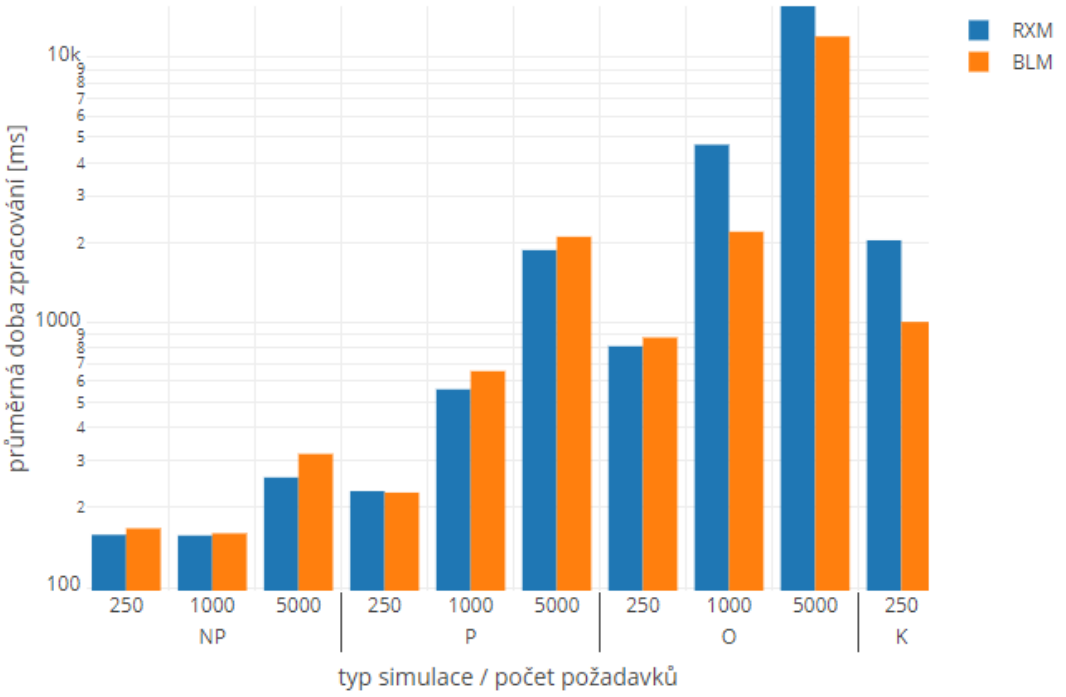
6.2.1 Rychlost zpracování

V simulacích, kdy existuje vyšší síťová latence mezi klientem a cílovým modelem lze pozorovat výrazné snížení rozdílu mezi modely při nízkém počtu uživatelů. Naopak v simulacích typu O, kdy jsou přenášena objemová data ke klientovi dosahuje RXM výrazně horších hodnot i pro velký počet uživatelů. To je konzistentní s testy bez latencí klienta. Tato data jsou zobrazena v tabulce 8 a pro přehlednost vynesena do grafu na obrázku 22. Simulace, při kterých došlo k chybovosti vyšší než 1% nejsou uvedeny.

Tabulka 8 - Rychlost zpracování se síťovou latencí klienta

Simulace		RXM		BLM	
Typ	Uživatelé	$\bar{t}$ [ms]	σ [ms]	$\bar{t}$ [ms]	σ [ms]
NP	250	158	18	167	20
	1000	157	20	160	35
	5000	260	248	319	220
P	250	231	105	228	86
	1000	559	283	655	384
	5000	1875	2261	2105	1684
O	250	813	297	877	374
	1000	4412	1726	2199	1189
	5000	16723	7856	11991	7574
K	250	2042	623	1004	281

Zdroj: Autor



Obrázek 22 - Graf průměrné doby zpracování požadavku s latencí klienta

Zdroj: Autor

6.2.2 Spolehlivost

Tabulka 9 zobrazuje spolehlivost obou modelů s vlivem síťové latence klienta. Při simulacích typu NP a P měly oba modely 100% úspěšnost, nejsou proto v tabulce zaneseny.

Tabulka 9 - Spolehlivost se síťovou latencí klienta

Simulace		RXM		BLM	
Typ	Uživatelé	Naplněné [%]	Chybové [%]	Naplněné [%]	Chybové [%]
O	1000	100	0	100	0
	5000	100	0	99,98	0,02
K	1000	100	0	99,5	0,5
	5000	50,28	49,72	96,66	3,34

Zdroj: Autor

BLM v tomto případě vykazuje zanedbatelné množství chyb pro většinu požadavků. Pro 5 000 uživatelů simulace typu K má tento model chybovost pouze 3,34%. Oproti tomu RXM při stejné simulaci vykazuje 50,28% chybovost. To je konzistentní s testy bez síťové latence klienta. RXM v tomto případě zahltí spojení mezi službami na straně klienta Feign a dochází k vypršení časových limitů.

6.2.3 Využití systémových prostředků

Jak lze vidět z tabulky 10, z hlediska využití času vCPU jsou výrazné rozdíly pouze v požadavcích typu NP a P s nízkým počtem uživatelů. Zde RXM dosahuje mnohem nižšího vytížení. Spotřeba paměti typu heap je v těchto simulacích v porovnání se simulacemi bez latence klienta obdobná pro BLM, avšak výrazně vyšší v případě RXM. To může být způsobeno delší dobou, po kterou jsou v paměti uchována data, něž jsou zapsána klientovi.

Tabulka 10 - Využití systémových prostředků s latencí klienta

Simulace		RXM		BLM	
Typ	Uživatelé	CPU [%]	Heap [MB]	CPU [%]	Heap [MB]
NP	250	20	88	59,4	73
	1000	60	100	100	92
	5000	100	118	100	119
P	250	39	96	61	92
	1000	100	133	97	111
	5000	100	144	100	192
O	250	86	200	100	185
	1000	100	115	100	197
	5000	100	145	100	232
K	250	100	240	94	127
	1000	100	203	100	194

Zdroj: Autor

7 Závěry a doporučení

Cílem práce bylo představení problematiky vývoje mikroservisních webových aplikací za použití reaktivního programování a porovnání jeho vlastností s blokačním modelem. Nejprve byly představeny základní principy, technologie a nástroje, které jsou v praxi využívány pro tvorbu a provoz mikroservisně orientované architektury. Dále byla rozebrána problematika reaktivního zpracování dat z teoretického hlediska i jeho aplikace za využití moderního frameworku Spring Boot 2 s podporou Spring WebFlux.

Na základě těchto poznatků byl sestaven modelový systém, na kterém lze demonstrovat vlastnosti obou přístupů. Dále byly identifikovány základní kategorie požadavků, které odpovídají potřebám reálných systémů. Pro každou kategorii byl v rámci modelového systému sestaven testovací případ a odpovídající zátěžová simulace v nástroji Gatling. Tento model byl následně implementován v reaktivní i blokační podobě za využití aktuálních verzí ekvivalentních knihoven frameworku Spring. Pro nasazení byla zvolena cloudová platforma Amazon Web Services, na které byly vytvořeny odpovídající konstrukty a jednotlivé instance pro každý prvek systému.

Z provedených zátěžových simulací vyplývá, že nelze jednoznačně určit, který model je globálně vhodnější. Blokační model poskytuje rychlejší a stabilnější dobu zpracování požadavků v případě, kdy je počet aktivních uživatelů menší či blízký nastavenému limitu vláken. Při jeho výrazném překročení však rychle ztrácí tuto vlastnost a reaktivní model poskytuje rychlejší dobu zpracování. Záleží také na typu jednotlivých požadavků. Pokud je klientovi zapisováno velké množství dat, reaktivní model dosahuje i při větším množství požadavků horších hodnot.

Z hlediska spolehlivosti bylo zjištěno několik problémů s výchozí konfigurací jednotlivých knihoven. Reaktivní driver pro MongoDB obsahuje relativně nízký limit maximálních aktivních spojení, který lze rychle zahltit. Tento problém se v blokačním modelu nevyskytoval, jelikož jsou vlákna, a tedy i databázové spojení, limitována již na straně HTTP konektoru. Limit lze však navýšit a tím výrazně zlepšit spolehlivost reaktivního modelu. Pokud je navýšen maximální počet vláken pro zpracování požadavků v blokačním modelu, na instancích typu t2.micro dochází při

větším množství požadavků k zahlcení operační paměti a zaseknutí či pádu celé aplikace. V několika případech bylo nutné restartovat konkrétní EC2 instanci.

Vývoj a provoz mikroservisní architektury je komplexní problematika, která se neustále rozvíjí. Je zde velké množství parametrizace, kterou lze dále upravit a nastavit tak požadované chování odpovídající potřebám dané aplikace. Může se jednat například o navýšení limitů pro vypršení spojení klientů jednotlivých služeb, replikaci nodů, mechanismy řešící výpadky služeb a další. Tato práce se zaměřila především na výchozí nastavení knihoven tak, jak jsou k dispozici vývojáři. Důležitým faktorem je také zvolené prostředí, na kterém bude systém nasazen. Výsledky zde zjištěné lze tak brát pouze jako doporučení a před rozhodnutím o použití některé technologie je vhodné provést vlastní analýzu pro konkrétní doménu.

8 Seznam použité literatury

AKARNOKD. *RxJava 1, 2 & Reactor Comparison Benchmarks 14-03-2018 · Issue #7 · Akarnokd/Akarnokd-Misc* [online]. 2018, GitHub. [Cit. 3.3.2019]. Dostupné z WWW: <https://github.com/akarnokd/akarnokd-misc/issues/7>.

AMAZON.COM INC. *Amazon Web Services (AWS) - Cloud Computing Services* [online]. 2019, Amazon Web Services, Inc. [Cit. 23.2.2019]. Dostupné z WWW: <https://aws.amazon.com/>.

APACHE SOFTWARE FOUNDATION. *Apache Kafka* [online]. 2019, Apache Kafka. [Cit. 15.2.2019]. Dostupné z WWW: <https://kafka.apache.org/>.

ATCHISON, L. *Architecting for Scale*. 2016. ISBN 978-1-4919-4339-7.

BINSTOCK, A. *Java's 20 Years Of Innovation* [online]. 2015, Forbes. [Cit. 2.2.2019]. Dostupné z WWW: <https://www.forbes.com/sites/oracle/2015/05/20/javas-20-years-of-innovation/>.

BUNA, S. *Learning GraphQL and Relay*. Packt Publishing Ltd, 2016. ISBN 978-1-78646-197-1.

CONFLUENT INC. *Streams Concepts — Confluent Platform* [online]. 2019. [Cit. 22.2.2019]. Dostupné z WWW: <https://docs.confluent.io/current/streams/concepts.html>.

DARREL, M. *Bizcoder - A Fresh Coat Of REST Paint On A SOAP Stack* [online]. 2015. [Cit. 14.2.2019]. Dostupné z WWW: <http://www.bizcoder.com/a-fresh-coat-of-rest-paint-on-a-soap-stack>.

DOCKER INC. *Enterprise Application Container Platform* [online]. 2019, Docker. [Cit. 12.4.2019]. Dostupné z WWW: <https://www.docker.com/>.

FACEBOOK INC. *GraphQL: A query language for APIs.* [online]. 2019. [Cit. 15.2.2019]. Dostupné z WWW: <http://graphql.org/>.

FIELDING, R.T. & TAYLOR, R.N. *Architectural styles and the design of network-based software architectures*. University of California, Irvine Irvine, USA, 2000.

GATLING CORP. *Gatling Open-Source Load Testing - For DevOps and CI/CD* [online]. 2019, Gatling Open-Source Load Testing. [Cit. 12.4.2019]. Dostupné z WWW: <https://gatling.io/>.

GOESSNER, S. *JSONPath - XPath for JSON* [online]. 2007. [Cit. 16.3.2019]. Dostupné z WWW: <https://goessner.net/articles/JsonPath/>.

GOOGLE INC. *Compute Engine - IaaS | Compute Engine* [online]. 2019, Google Cloud. [Cit. 23.2.2019]. Dostupné z WWW: <https://cloud.google.com/compute/>.

GOOGLE CONTAINER TOOLS. *Google Jib* [online]. 2019, GitHub. [Cit. 12.4.2019]. Dostupné z WWW: <https://github.com/GoogleContainerTools/jib>.

GRAFANA LABS. *Grafana - The Open Platform for Analytics and Monitoring* [online]. 2019, Grafana Labs. [Cit. 22.2.2019]. Dostupné z WWW: <https://grafana.com/>.

GREGORYBLEIKER. *English: Observer Pattern*. 2018.

HALPIN, T. & MORGAN, T. *Information Modeling and Relational Databases* 2 edition. Burlington, MA : Morgan Kaufmann, 2008. ISBN 978-0-12-373568-3.

HASHICORP. *Terraform by HashiCorp* [online]. 2019, Terraform by HashiCorp. [Cit. 12.4.2019]. Dostupné z WWW: <https://www.terraform.io/index.html>.

HONG, S., KIM, J.-C., SHIN, J.W., MOON, S.-M., OH, H.-S., LEE, J. & CHOI, H.-K. Java Client Ahead-of-time Compiler for Embedded Systems. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES '07. New York, NY, USA : ACM, 2007, s. 63–72. ISBN 978-1-59593-632-5.

CHARBONNEAU. *Java Just-In-Time Compilation: More Than Just a Buzzword - DZone Java* [online]. 2013, Dzone.Com. [Cit. 10.2.2019]. Dostupné z WWW: <https://dzone.com/articles/java-just-time-compilation>.

JETBRAINS. *Kotlin/Native - Kotlin Programming Language* [online]. 2018a, Kotlin. [Cit. 9.2.2019]. Dostupné z WWW: <https://kotlinlang.org/docs/reference/native-overview.html>.

JETBRAINS. *The State of Developer Ecosystem 2018 - Infographic* [online]. 2018b, JetBrains. [Cit. 9.2.2019]. Dostupné z WWW: <https://www.jetbrains.com/research/devecosystem-2018/>.

KASSIS, M. *Advanced Topics in Concurrency and Reactive Programming: The Observable Contract Majeed Kassis. - ppt download* [online]. 2019. [Cit. 15.4.2019]. Dostupné z WWW: <https://slideplayer.com/slide/13323652/>.

KELLY, M. *The Hypertext Application Language* [online]. 2013. [Cit. 15.2.2019]. Dostupné z WWW: <http://stateless.co/hal_specification.html>.

KHAN, S. *Microservices and it's Architecture* [online]. 2018, sayantan khan. [Cit. 10.2.2019]. Dostupné z WWW: <https://medium.com/@iamstk14/microservices-and-its-architecture-ec7de96f8f73>.

KÖNIG, D., KING, P., LAFORGE, G., D'ARCY, H., CHAMPEAU, C., PRAGT, E. & SKEET, J. *Groovy in Action* Second edition. Shelter Island, NY : Manning Publications, 2015. ISBN 978-1-935182-44-3.

MERRICK, P., ALLEN, S. & LAPP, J. *XML remote procedure call (XML-RPC)* [online]. 2006. [Cit. 12.2.2019]. Dostupné z WWW: <https://patents.google.com/patent/US7028312B1/en>.

MIKOWSKI, M. & POWELL, J. *Single Page Web Applications: JavaScript End-to-end* 1st vyd. Greenwich, CT, USA : Manning Publications Co., 2013. ISBN 978-1-61729-075-6.

MONGODB INC. *NoSQL Databases Explained* [online]. 2019, MongoDB. [Cit. 22.2.2019]. Dostupné z WWW: <https://www.mongodb.com/nosql-explained>.

MORLEY, MATT. *JSON-RPC 2.0 Specification* [online]. 2013. [Cit. 12.2.2019]. Dostupné z WWW: <https://www.jsonrpc.org/specification>.

NETFLIX. *Netflix Open Source Software Center* [online]. 2016. [Cit. 15.4.2019]. Dostupné z WWW: <https://netflix.github.io/>.

NETTY PROJECT. *Netty: Home* [online]. 2019. [Cit. 10.3.2019]. Dostupné z WWW: <https://netty.io/index.html>.

NEWMAN, S. *Building Microservices*. Beijing Sebastopol, CA, 2015. ISBN 978-1-4919-5035-7.

NOTTINGHAM, M. & BRYAN, P. *JavaScript Object Notation (JSON) Patch* [online]. 2013. [Cit. 15.2.2019]. Dostupné z WWW: <https://tools.ietf.org/html/rfc6902>.

NURKIEWICZ, T. *RxJava vs Reactor* [online]. 2019. [Cit. 3.3.2019]. Dostupné z WWW: <https://www.nurkiewicz.com/2019/02/rxjava-vs-reactor.html>.

OAKS, S. *Java Performance: The Definitive Guide: Getting the Most Out of Your Code* 1 edition. Sebastopol, CA : O'Reilly Media, 2014. ISBN 978-1-4493-5845-7.

OPENFEIGN. *OpenFeign*. OpenFeign, 2019.

ORACLE. *Java Platform, Enterprise Edition (Java EE) | Oracle Technology Network | Oracle* [online]. 2019a. [Cit. 9.2.2019]. Dostupné z WWW: <https://www.oracle.com/technetwork/java/javaee/overview/index.html>.

ORACLE. *Java SE | Oracle Technology Network | Oracle* [online]. 2019b. [Cit. 9.2.2019]. Dostupné z WWW: <https://www.oracle.com/technetwork/java/javase/overview/index.html>.

PIVOTAL. *Messaging that just works — RabbitMQ* [online]. 2019. [Cit. 15.2.2019]. Dostupné z WWW: <https://www.rabbitmq.com/>.

PLAYTIKA. *Reactive Feign client inspired by https://github.com/OpenFeign project: Playtika/feign-reactive*. Playtika, 2019.

REACTIVE STREAMS SPECIAL INTEREST GROUP. *Reactive Streams* [online]. 2019. [Cit. 9.2.2019]. Dostupné z WWW: <http://www.reactive-streams.org/>.

RED HAT INC. *Undertow · JBoss Community* [online]. 2019. [Cit. 10.3.2019]. Dostupné z WWW: <http://undertow.io/>.

RICHARDSON, C. *Microservices Pattern: Microservice Architecture pattern* [online]. 2018, microservices.io. [Cit. 10.2.2019]. Dostupné z WWW: <http://microservices.io/patterns/microservices.html>.

ROSSEN, S. *Servlet and Reactive Stacks in Spring Framework 5* [online]. 2018, InfoQ. [Cit. 16.3.2019]. Dostupné z WWW: <https://www.infoq.com/articles/Servlet-and-Reactive-Stacks-Spring-Framework-5>.

SHAFIROV, M. *Kotlin on Android. Now Official* [online]. 2017, Kotlin Blog. [Cit. 9.2.2019]. Dostupné z WWW: <https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official/>.

SCHILDT, H. *Java 2: The Complete Reference* 4th vyd. McGraw-Hill Professional, 2000. ISBN 978-0-07-213084-3.

SPRING. *Reactor 3 Reference Guide* [online]. 2019a. [Cit. 21.3.2019]. Dostupné z WWW: <https://projectreactor.io/docs/core/release/reference/>.

SPRING. *Spring Boot* [online]. 2019b. [Cit. 6.4.2019]. Dostupné z WWW: <https://spring.io/projects/spring-boot>.

SPRING. *Spring Cloud* [online]. 2019c. [Cit. 6.4.2019]. Dostupné z WWW: <https://spring.io/projects/spring-cloud>.

SPRING. *Spring Cloud Config* [online]. 2019d. [Cit. 12.4.2019]. Dostupné z WWW: <https://spring.io/projects/spring-cloud-config>.

SPRING. *Web on Reactive Stack* [online]. 2019e. [Cit. 10.3.2019]. Dostupné z WWW: <https://docs.spring.io/spring/docs/current/spring-framework-reference/web-reactive.html#webflux>.

SPRING REACTOR. *Flux (Reactor Core 3.2.6.RELEASE)* [online]. 2019. [Cit. 8.3.2019]. Dostupné z WWW: <https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html#flatMap-java.util.function.Function->.

TECHIE. *Tomcat 7 request processing threading architecture and performance tuning* [online]. 2016. [Cit. 10.3.2019]. Dostupné z WWW:

<http://ttlnews.blogspot.com/2016/02/tomcat-7-request-processing-threading.html>.

TIOBE. *TIOBE Index | TIOBE - The Software Quality Company* [online]. 2019. [Cit. 9.2.2019]. Dostupné z WWW: <https://www.tiobe.com/tiobe-index/>.

W3C. *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)* [online]. 2007a. [Cit. 10.2.2019]. Dostupné z WWW: <https://www.w3.org/TR/soap12/>.

W3C. *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language* [online]. 2007b. [Cit. 14.2.2019]. Dostupné z WWW: <https://www.w3.org/TR/wsdl/#intro>.

WILLIAMS, C. *Is REST Best in a Microservices Architecture?* [online]. 2015, Capgemini Engineering. [Cit. 15.2.2019]. Dostupné z WWW: <https://capgemini.github.io/architecture/is-rest-best-microservices/>.

DVD přiložené k práci obsahuje zdrojové kódy všech služeb a doprovodných nástrojů, které byly použity v rámci testování. Dále obsahuje soubor všech naměřených hodnot a tento text ve formátu PDF. Jeho struktura je následující:

- /api – Postman kolekce pro testování API služeb
- /discovery-service – zdrojové kódy SD služby
- /spring-boot-admin – zdrojové kódy monitorovací služby
- /finance-system-blocking – zdrojové kódy služeb blokačního modelu
- /finance-system-reactive – zdrojové kódy služeb reaktivního modelu
- /infrastructure – bash scripty a zdrojové kódy související s infrastrukturou
- /performance-tests – zdrojové kódy simulací, zdrojové kódy generátoru dat, soubor s naměřenými hodnotami

Univerzita Hradec Králové
Fakulta informatiky a managementu
Akademický rok: 2018/2019

Studijní program: Aplikovaná informatika
Forma: Prezenční
Obor/komb.: Aplikovaná informatika (ai2-p)

Podklad pro zadání DIPLOMOVÉ práce studenta

PŘEDKLÁDÁ:	ADRESA	OSOBNÍ ČÍSLO
Jouda Vít	Libčany 6, Libčany	I14776

TÉMA ČESKY:

Reaktivní webové služby na platformě Java

TÉMA ANGLICKY:

Reactive Web Java Applications

VEDOUCÍ PRÁCE:

Ing. Pavel Kříž, Ph.D. - KIKM

ZÁSADY PRO VYPRACOVÁNÍ:

Cíl: Popsat výhody a nevýhody reaktivního přístupu oproti klasickému imperativnímu stylu. Ukázat na modelových příkladech, které odpovídají reálným potřebám moderních distribuovaných aplikací. Změřit základních metrik a porovnat výsledky s imperativním blokačním modelem.

Osnova:
1. Úvod
2. Platforma Java
3. Webové služby
4. Reaktivní programování
5. Praktické testy
6. Diskuse
7. Závěry a doporučení

SEZNAM DOPORUČENÉ LITERATURY:

1) Spring, "Reactor 3 Reference Guide", 2019. [Online]. Dostupné z: https://projectreactor.io/docs/core/release/reference/
2) S. Newman, Building Microservices. Beijing Sebastopol, CA, 2015.
3) L. Atchison, Architecting for Scale. 2016

Podpis studenta: ..... **Datum:** 21.3.19

Podpis vedoucího práce: ..... **Datum:** 21.3.19