FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Návrh řešení Service Discovery Server pro projekt “Manažerský Informační Systém"
Student:	Kostiantyn Kuznietsov
Vedoucí:	Ing. Jiří Hunka
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Cílem práce je navrhnout nejlepší řešeni Service Discovery Serveru pro použití load balancingu v projektu “Manažerský Informační Systém”.

Postupujte dle uvedených kroků:
1. Analyzujte stávající obdobná řešeni problému. Na jednotlivých SDS (Service Discovery Server) analyzujte různé typy load balancingu.
2. Nastudujte základní znalosti použití microserivis architektury a load balancingu v enterprise projektech.
3. Pro účely analýzy load balancingu navrhněte vhodný koncept a implementaci mikroservis, které se následně budou používat v MIS: ARES (Administrativní registr ekonomických subjektů), RÚIAN (Registr územní identifikace, adres a nemovitostí), elektronických bankovnictvíc ČSOB banky, FIO banky a KB banky. Využijte pro implementaci odpovídající API.
4. Pomocí vhodných testů funkčnost microservis řádně otestujte.
5. Na jednotlivých SDS implementujte konfiguraci pro load balancingu.
6. Analyzujte realizované konfigurace a v závěru navrhněte nejlepší řešení pro budoucí použití.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 19. února 2021 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Návrh řešení Service Discovery Server pro projekt „Manažerský Informační Systém“

Kostiantyn Kuznietsov

Katedra softwarového inženýrství
Vedoucí práce: Ing. Jiří Hunka

13. května 2021

Poděkování

Chtěl bych poděkovat svému vedoucímu Ing. Jiřímu Hunkovi za vedení mé bakalářské práce, dále celému programátorskému týmu společnosti e-invent s.r.o. za rady a připomínky při implementaci a také své rodině za podporu během studia.

Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 13. května 2021 . . . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2021 Kostiantyn Kuznietsov. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Kuznietsov, Kostiantyn. *Návrh řešení Service Discovery Server pro projekt „Manažerský Informační Systém“*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021. Dostupný také z WWW: ⟨`https://gitlab.fit.cvut.cz/kuznikos/mis-bp`⟩.

Abstrakt

Tato práce se zabývá návrhem řešení Service Discovery Serveru pro projekt společnosti e-invent s.r.o. „Manažerský Informační Systém“. V teoretické části se čtenář obohatí znalostmi komunikaci služeb v mikroslužební architektuře, dozví se o service discovery a procesu vyvažování zátěže. V praktické části budou realizovány jednotlivé mikroslužby a na ně aplikována řešení existujících Service Discovery Serveru. Na základě testovacích výsledků bude zvoleno nejlepší řešení tohoto problému.

Klíčová slova mikroslužby, objevování síťových služeb, vyvažování zátěže, škálovatelnost, klient, server, Eureka, Consul

Abstract

This thesis offers Service Discovery Service solution for project „Manažerský Informační Systém“ developing by company e-invent s.r.o. . In the teoretical part reader will be able to know service's communication in microservice architecture, service discovery and load balancing process. In the practice part particular microservices will be implemented and Service Discovery Server's solution will be applied on them. Based on testing results the best solution will be offered.

Keywords microservices, service discovery, load balancing, scalability, client, server, Eureka, Consul

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Dnešní doba je už těžce představitelná bez hledání informací na internetu a návštěvy webových stránek. Používáme je nejen pro získání jakýchkoli informací, ale i pro administrativu, nákup či poskytování služeb. Vzhledem k tomu, že web plní hodně našich každodenních požadavků, je nutné zajistit jeho pohodlný provoz a použití.

V koronavirové krizi se vnímání kvality jednotlivých webů více zesílilo než kdy před tím. Skoro každý z nás se v této době setkal s nějakou chybou na webové stránce. To je způsobeno tím, že firmy podceňují kvalitu a kladou důraz na rozšíření funkcionality webové aplikace.

Softwarová společnost e-invent s.r.o., mající na starosti projekt pro sledování všech firemních procesů „Manažerský Informační Systém“, k otázce kvality se chová velmi zodpovědně. Během údržby se zjistilo, že existující realizace už neodpovídá dnešním požadavkům a je těžce rozšiřitelná. Vzhledem k tomu bylo rozhodnuto vytvořit novou generaci projektu s využitím nových technologií a funkcionalit. Pro implementaci byl zvolen programovací jazyk Kotlin a framework Spring.

Jelikož se autor této práce zúčastnil backendového vývoje výše zmíněné starší verze projektu a je seznámený s předešlou implementaci aplikace, byla mu nabídnuta účast ve vývoji nové verze.

Cílem práce je provést rešerši a navrhnout nejvhodnější řešení Service Discovery Serveru pro projekt „Manažerský Informační Systém“. Dílčím cílem je analyzovat metody komunikace s webovými API[1] a následně implementovat mikroslužby pro integraci externích zdrojů dat, které budou v rámci MIS[2] použité a bude možné na nich jednotlivé Service Discovery Servery testovat. Pro testování byly použité veřejně známé konfigurace load balancingu.

Výstupy této práce by měly zjednodušit práci vývojářům a usnadnit další vývoj systému.

[1] API – Application programming interface
[2] MIS – Manažerský Informační Systém

Tato bakalářská práce se skládá z teoretické a praktické části. Teoretická část je věnována vysvětlení základních pojmů, analýze technologií, které budou použité při vývoji. Na základě analýzy bude v praktické části navrhnuta vhodná architektura pro implementaci mikroslužeb a realizovaná jejich implementace. Zároveň na základě implementovaných mikroslužeb budou realizovány konfigurace pro propojení s registrem služeb a nastavení pro vyvažování zátěže. Nakonec bude uděláno testování jednotlivých konfigurací a analýza výsledků.

Kapitola 1

Analýza

Tato kapitola popisuje teoretické základy, přístupy k implementaci různých komponent potřebných pro práci na bakalářském projektu.

1.1 Architektonické vzory návrhu aplikace

Volba architektonického vzoru nové aplikace záleží na rozsahu programu, době aktualizace softwaru a také na tom, jaké jsou vývojové požadavky. Tato kapitola popíše základní architektonické vzory využívané ve webových aplikacích.

1.1.1 Monolitická architektura

Historický program se skládal z jednoho souboru. Časem se rozsáhlost aplikace zvětšovala, měla už několik souborů a bylo potřeba zvažovat, jaké soubory vytvářet a jak je spojovat. Důsledkem toho začaly být aplikace ještě složitějšími, soubory byly seskupovány do abstraktních celků, které nazýváme moduly. Hlavní myšlenkou monolitické architektury je to, že veškerý kód ze všech modulů je nasazen a spuštěn jako jediný spustitelný celek běžící na jednem aplikačním serveru [1]. Nevýhodami takové architektury je komplikované rozšíření programu, složité debugování a refaktorování.

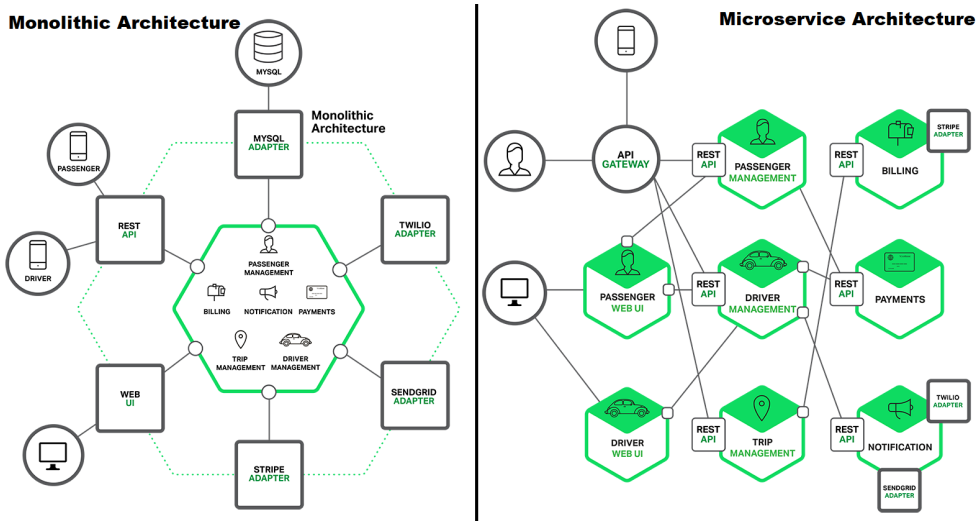
1.1.1.1 Vícevrstvá architektura

Vícevrstvá architektura je podtypem monolitické architektury, kde je aplikace rozdělena na několik (typicky 3) logických vrstev, z nichž je každá zodpovědná za přesně definovaný funkcionální celek. Každou vrstvu lze nezávisle upravit. Výhodou takového principu tvorby systému je rozdělení aplikace na celky, standardizace funkcionality vrstev a efektivnější testování. Mezi nevýhody patří možné kaskádové změny při úpravách kódu a snížení výkonu systému při přidávání další vrstvy [2].

1.1.2 Mikroservisni architektura

Architektura mikroslužeb(mikroservis) je architektonický koncept, který se staví proti monolitickému přístupu v aplikační architektuře. Aplikace se rozděluje na jednotlivé, nezávisle spustitelné celky – mikroslužby. Typicky se rozdělení provádí podle business požadavků [3]. To znamená, že budeme mít několik menších aplikací, které odpovídají každému z požadavků, běžících na různých aplikačních serverech, komunikujících mezi sebou přes síť pomocí protokolu HTTP[3].

Přínosem použití architektury mikroslužeb je rychlé dodání kvalitního softwaru díky tomu, že se může současně vyvíjet několik mikroslužeb. Díky tomu, že jsou služby nasazené nezávislé, není potřeba po provedení změn nasazovat celou aplikaci a stačí jen nasadit změněnou mikroslužbu. Použití mikrosluzeb přináší také lepší škálovatelnost a flexibilitu volby programovacího jazyka.



Obrázek 1.1: Ukázka rozdělení projektu navrženého pomocí monolitické architektury do jednotlivých mikrosluzeb (zdroj: [4])

1.2 Webové služby

V dnešni době je velmí těžké si představit webovou aplikaci, která by nevyužívala rozhraní jiné webové aplikace. Častými případy užití jsou, například internetové platby, získaní transakcí z internetového bankovnictví. Tuto komunikaci je ale potřeba stardizovat pomocí webových služeb. Webová služba je softwarový systém umožňující interakci dvou strojů v síti prostrednictim zpráv

[3]HTTP – Hyper Text Transfer Protocol

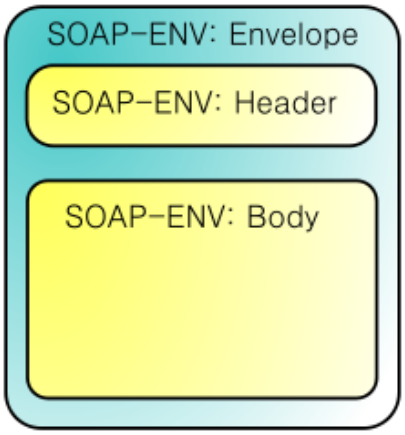
založených na určitých protokolech [5]. V této kapitole popíšeme základní architektury webových služeb.

1.2.1 SOAP

SOAP (angl. *Simple Object Access Protocol*) je protokole pro výměnu zpráv založených na XML přes síť, hlavně pomocí HTTP [6]. Je to architektura, která je orientovaná na služby (angl. *service-oriented*) to znamená že je zaměřena na konání jednotlivých služeb, které poskytuje aplikace. Pro popis webové služby a přístup k ní se používá jazyk WSDL, který je založený na XML. Navíc, protokol SOAP pro přenos dat podporuje nejenom HTTP ale také jiné protokoly, například SMTP[4].

Struktura SOAP zprávy se skládá ze 4 základních tagů [7]:

- `<Envelope>` je kořenový a povinný v každé SOAP zprávě;
- `<Header>` není povinný, popisuje atributy zprávy;
- `<Body>` povinný, popisuje samotnou zprávu;
- `<Fault>` není povinný, je podtagem `<Body>` a obsahuje popis chyb vznikajících při zpracování zprávy.



Obrázek 1.2: Ukázka formátu SOAP zprávy (zdroj: [8])

1.2.1.1 WSDL

WSDL (angl. *Web Services Description Language*) je definiční jazyk založený na XML. Definice WSDL popisují rozhraní webové služby. Klientský program

[4]SMTP – Simple Mail Transfer Protocol

připojující se k webové službě může číst WSDL a určit, jaké funkce jsou k dispozici na serveru. Je nedílnou součástí UDDI[5].

1.2.1.2 XML

XML (angl. *eXtensible Markup Language*) je značkovací jazyk, který slouží pro ukládání a přenos strukturovaných dat. Je podobný HTML[6], ale na rozdíl od něho nemá přesně definovanou sadu tagů. XML soubor je v textovém formátu, ve kterém jsou datové prvky vytvářeny pomocí speciálních značek, jejichž pořadí a vnoření určuje strukturu souboru a jeho obsah.

```
<note>
    <to>Maros</to>
    <from>Honza</from>
    <heading>Congratulations</heading>
    <body>Happy Birthday!</body>
</note>
```

Obrázek 1.3: Ukázka XML formátu

1.2.2 REST

REST (angl. *Representational State Transfer*) je architektonický návrh rozhraní distribuovaných systém a způsob komunikace mezi nimi [9]. Poprvé byl navržen v roce 2000 Royem Fieldingem. Architektura je orientována na zdroje (angl. *resource-oriented*) a popisuje jakým způsobem můžeme od zdrojů získat data. Základními charakteristiky REST jsou [10]:

- komunikace se provádí prostřednictvím protokolu HTTP
- všechny zdroje musí mít unikátní URI[7]
- na rozdíl od protokolu SOAP, přenos dat se neomezuje formátem XML, často využívaný je například JSON (viz 1.2.2.1)
- vícevrstvá architektura (viz 1.1.1.1)

Přístup ke zdrojům je řízený pomocí čtyř operací zvaných CRUD[8] [10]:

[5]UDDI – Universál Description, Discovery and Integration je repositář, umožňující registraci a vyhledávání webových služeb
[6]HTML – HyperText Markup Language
[7]URI – Universal Resource Identifier
[8]CRUD – Create, Read, Update, Delete

- GET je nejpoužívanější metoda, slouží pro získání zdroje. Tato operace je idempotentní. To znamená, že při opakovaném volání vrátí vždy stejný výsledek. V případě úspěšného volání vrátí kód `200(OK)`. V případě nalezení chyby bude vrácen kód `404(NOT FOUND)` nebo `400(BAD REQUEST)`.
- POST se používá pro tvorbu nového zdroje na serveru. Tato operace není idempotentní. Po úspěšném vytvoření je navrácen kód `201(CREATED)`. V odpovědi také posílá URL nově vytvořeného zdroje.
- PUT je metoda pro modifikaci už existujícího zdroje. V případě úspěšného provedení operace získáme kód `200(OK)` nebo `204(NO CONTENT)`.
- DELETE slouží pro smazání zdroje identifikovaného konkrétním URI. Operace je idempotentní. V případě úspěchu server vrací kód `200(OK)`, nebo `204(NO CONTENT)`. Pokud už byl zdroj smazán, získáme kód `404(NOT FOUND)`.

1.2.2.1 JSON

JSON (angl. *JavaScript Object Notation*) je formát pro strukturování dat. Používá se především k přenosu dat mezi serverem a webovou aplikací jako alternativa k XML. Formát JSON se skládá z párů klíč a hodnota.

```
{
    "name":"John",
    "age":30,
    "car":"Audi"
}
```

Obrázek 1.4: Ukázka JSON formátu

1.2.2.2 RESTful

Aby distribuovaný systém byl považován za RESTful, musí splňovat následující požadavky [10]:

- rozdělení na klientskou a serverovou část
- jednotné rozhraní
- bezestavová komunikace[9]

[9]Server nemusí ukládat žádné informace o klientech. Požadavek má obsahovat všechny potřebné informace ke zpracování a v případě potřeby k identifikaci klienta.

- HATEOAS (angl. *Hypermedia as the Engine of Application State*)[10]
- idempotence[11]

1.2.3 Shrnutí

V dnešní době se většinou používá REST, protože je jednodušší na použití. Nemůžeme ale napřímo porovnávat REST a SOAP, protože první je architektonický styl a druhý protokol. Z hlediska budoucího rozšíření aplikace není vhodné použít SOAP, protože v případě změn ve službách vznikne potřeba měnit klientský a serverový WSDL soubor. SOAP je naopak vhodnější, když potřebujeme mít jasně určenou specifikaci rozhraní služby, nebo jsou kladené požadavky na bezpečnost. Díky tomu, že data přenášené pomocí REST nemají složitou strukturu a přesto kladou menší nároky na parsování, tak tento architektonický styl je vhodné použít v případě, že je omezený objem přenášených dat.

1.3 Service discovery

V dnešní době je mikroservisní atchitektura běžnou architektonickou metodou pro vývoj cloudových softwarových aplikací. Cloudové aplikace jsou software, ke kterému uživatelé přistupují primárně prostřednictvím internetu, což znamená, že alespoň část z nich je spravována serverem a nikoli uživatelskými počítači [11]. Projekt bakalářské práce je založen na mikroslužbách a jelikož jednotlivé mikroslužby mohou používat data od jiných mikroslužeb, je potřeba zajistit jejich komunikaci a „viditelnost“ pro ostatní. Pro tyto účely se používá service discovery.

1.3.1 Hyper Text Transfer Protocol

Jelikož se pro komunikaci mezi mikroslužbami bude využívat protokol HTTP, je potřeba se o něm zmínit. Hyper Text Transfer Protocol (HTTP) je jeden z nejdůležitějších protokolu pro přenos dat mezi klientem a serverem. Je založen na principu požadavek/odpověď [12]. Klientská aplikace vytváří požadavek a posílá ho na server, který následně vytváří odpověď’ a posílá ji zpátky klientovi. Primárně se používá pro přenos dat mezi uživatelskou aplikací a web serverem.

1.3.2 Service discovery na straně klienta

V tomto návrhovém vzoru klient je zodpovědný za určení umístění dostupných instancí služeb a volbu služby pomocí algoritmu pro vyvažování zátěže. Klient

[10] Stav zdroje se předává prostřednictvím obsahu těla, záhlaví požadavku, nebo požadovaného identifikátoru URI

[11] Odesílání stejného požadavku způsobuje stejné operace na serveru.

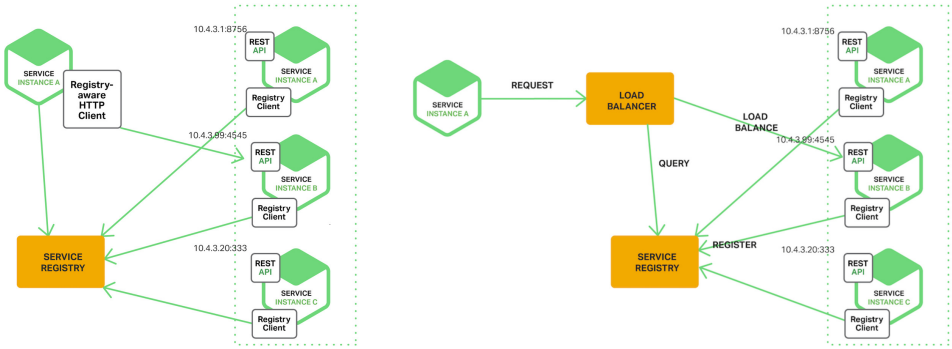
pomocí protokolu HTTP posílá požadavek na službu. Díky tomu následně od registru služeb získává seznam IP adres dostupných instancí požadované služby. Následně pomocí algoritmu pro vyvažování zátěže zvolí adresu služby, kam přesměruje požadavek. Přínosem takového přístupu je snadná realizace. Nevýhodou je vyvažování zátěže na straně klienta, což ho zbytečně zatěžuje.

1.3.3 Service discovery na straně serveru

V aplikaci, kde service discovery je implementován na straně serveru, je za vyhledávání dostupných instancí služeb zodpovědný server. V tomto případě je realizován proxy server, který získává HTTP požadavky od klienta. Získá seznam IP adres dostupných instancí od registru služeb a následně s využitím algoritmu pro vyvažování zátěže zašle požadavek do odpovídající služby. Hlavní výhodou takové implementace je uvolnění klienta od vyvažování zátěže a delegace tohoto problému proxy serverem. Problém může nastat v takovém případě, že proxy server spadne, což znamená že nebudeme mít přístup na mikroslužby. Proto je potřeba se postarat o jeho vysokou spolehlivost.

1.3.3.1 Proxy server

Proxy server „funguje jako prostředník mezi klientem a cílovým počítačem (serverem). Překládá klientské požadavky a vůči cílovému počítači vystupuje sám jako klient“ [13]. Moderní proxy servery realizují mnohem víc než předávání webových požadavků. Starají se o anonimizaci klienta, vstup do privátních sítí, chránění web-serveru proti útokům a vyvažují zátěž.



Obrázek 1.5: Ukázka service discovery na straně klienta a serveru (zdroj: [14])

1.3.4 Service registry

Registr služeb je reprezentován databází, kde je umístěna informace o IP adresách jednotlivých instancí aplikačních služeb. V procesu vývoje se mohou

vytvářet nové služby, zanikat staré, nebo se mohou jednotlivé služby přesouvat na jinou IP adresu. Administraci těchto jevů a následné aktualizaci databáze služeb se zabývá registr služeb.

1.3.4.1 Samostatná registrace

V tomto případě za registraci a odhlášení z registru služeb je zodpovědná samotná služba. Navíc, v případě potřeby, posílá požadavek na registr služeb potvrzující její fungování (angl. *heartbeat*).

1.3.4.2 Registrace pomocí třetí strany

Při použití této metody se používá nová komponenta – správce služeb, která řeší registraci. V případě, že se vytváří nová instance služby, správce je zodpovědný za její registraci v registru služeb a případně, když se zastavuje, za její deregistraci. Tento přístup má výhodu v tom, že není potřeba konfigurovat každý klient pro účely registraci zvlášť, což způsobuje lepší škálovatelnost.

1.4 Vyvažování zátěže

K selhání serveru vždy dochází neočekávaně a s velmi vážnými důsledky. Na začátku lze problémy nízkého výkonu serverů kvůli zvýšení zátěže vyřešit zvýšením kapacity serverů nebo optimalizací algoritmů. Dříve nebo později však přijde čas, kdy tato opatření budou nedostatečná.

Pokud je počet požadavků tak veliký, že je server už nedokáže zpracovat, je potřeba vytvořit několik kopií serverů a pro distribuci provozu použít vyvažování zátěže. Čím rovnoměrnější je distribuce, tím více uživatelů bude moci web používat. Cílem vyvažování zátěže je maximalizovat propustnost, minimalizovat dobu odezvy a zabránit přetížení jednotlivých zdrojů.

1.4.1 Definice základních pojmů

Pro posouzení o vyvažování zátěže je potřeba se obohatit o teoretický základ používaných pojmů.

1.4.1.1 Model ISO/OSI

Jelikož se vyvažování zátěže realizovat na různých síťových vrstvách, má smysl si vzpomenout na každou z nich pomocí referenčního modelu ISO/OSI (angl. *International Organization fór Standardization*). Model ISO/OSI „vypracovala organizace ISO jako hlavní část snahy o standardizaci počítačových sítí nazvané OSI a v roce 1984 ho přijala jako mezinárodní normu ISO 7498“ [15]. Referenční model je tvořen sedmi vrstvami (od nejnižší):

1. Fyzická – definuje, jak jsou počítače fyzicky připojené k síti. Funkce prostředků souvisejících s touto vrstvou je přenos jednotlivých bitů mezi příjemcem a odesílatelem.
2. Spojová – zajišťuje integritu rámců (blok bitů). Příjemce na spojové vrstvě je reprezentován MAC adresou.
3. Síťová – určuje směrování dat. Na této vrstvě jsou pakety směrovány pomocí překladu MAC adres na síťové adresy, například IP.
4. Transportní – tato úroveň zajišťuje spolehlivost přenosu dat od odesílatele k příjemci.
5. Relační – slouží pro administraci spojení mezi koncovými účastníky. Řídí navázáni, rušení a údržbu spojení.
6. Prezentační – úkolem této úrovně je formátování dat pro veřejné použití, například kódování nebo komprese.
7. Aplikační – je nejvyšší vrstva OSI. Má za cíl zajistit přístup aplikace k síťovým službám.

1.4.1.2 MAC adresa

MAC adresa (angl. *Media Access Control*) – posloupnost bitů, která je unikátní pro každé síťové zařízení. Používá se na spojové vrstvě modelu ISO/OSI a skládá se z 6 oktetů. Příklad: `00:16:17:e1:28:5f`.

1.4.1.3 IP adresa

IP adresa je identifikátor zařízení v počítačové síti. Existuje 2 typů:

- IPv4 – 32-bitová posloupnost 4 oktetů oddělených tečkou. Zapisuje se v desítkové soustavě. Příklad `127.0.0.1` .
- IPv6 – 128-bitová posloupnost hexadecimálních čísel. Zapisuje se jako osm skupin po čtyřech hexadecimálních číslicích.

  Příklad: `2001:0db8:85a3:0000:0000:8a2e:0370:7334`.

1.4.1.4 ARP protokol

Pro přenos paketů zařízení síťové vrstvy potřebuje vědět MAC adresu příjemce. ARP protokol (angl. *Address Resolution Protocol*) – používá se pro získání MAC adresy na základě IP adresy. Na začátku zařízení se podívá do své ARP tabulky, která se skládá z dvojic: IP adresa, MAC adresa. V případě, že nebude mít vyhovující záznam v tabulce, pošle *broadcast* požadavek na MAC adresu na všechna zařízení v podsíti. Zařízení, které bude mít žádoucí IP adresu, odpoví na tento požadavek a do odpovědi vloží MAC adresu.

1.4.1.5 DNS

Z angličtiny se překládá jako *Domain Name System*. Je to technologie umožňující získat IP adresu aplikačního serveru na základě doménového jména. DNS server je síťové zařízení které pro podsíť, ve které se nachází, uchovává dvojici: IP adresa, doménové jméno. Dále odpovídá na DNS požadavky klientů.

1.4.1.6 NAT

NAT (angl. *Network Address Translation*) překládá IP adresu počítače v privátní síti na adresu ve veřejné síti. To umožňuje komunikaci mezi počítači nacházejících se v různých podsítích.

1.4.1.7 TCP protokol

TCP protokol (angl. *Transmission Control Protocol*) – protokol transportní vrstvy, zaručuje doručení zprávy adresátovi a také zachovává správnou posloupnost přenosu dat.

Obrázek 1.6: Ukázka hlavičky TCP paketu (zdroj: [16])

1.4.1.8 Reverse proxy

Proxy server, přijímající požadavky z veřejné sítě a přesměrovávající je na sadu serverů, které nejsou dostupné z venku a jsou umístěné ve vnitřní síti.

1.4.2 Vyvažování zátěže na síťové vrstvě

Proces vyvažování zátěže na této vrstvě se provádí pomocí směrovacího protokolu BGP a protokolu pro vysokou dostupnost služeb VRRP [17].

1.4.2.1 VRRP protokol

Protokol VRRP(angl. *Virtual Router Redundancy Protocol*) je založen na tvorbě jednoho virtuálního směrovače, který spojuje všechny směrovače do

skupiny [18]. Navíc všechny směrovače mají společnou virtuální IP adresu (VIP) a identifikátor skupiny (VRID). Směrovače se rozdělují do 2 skupin:

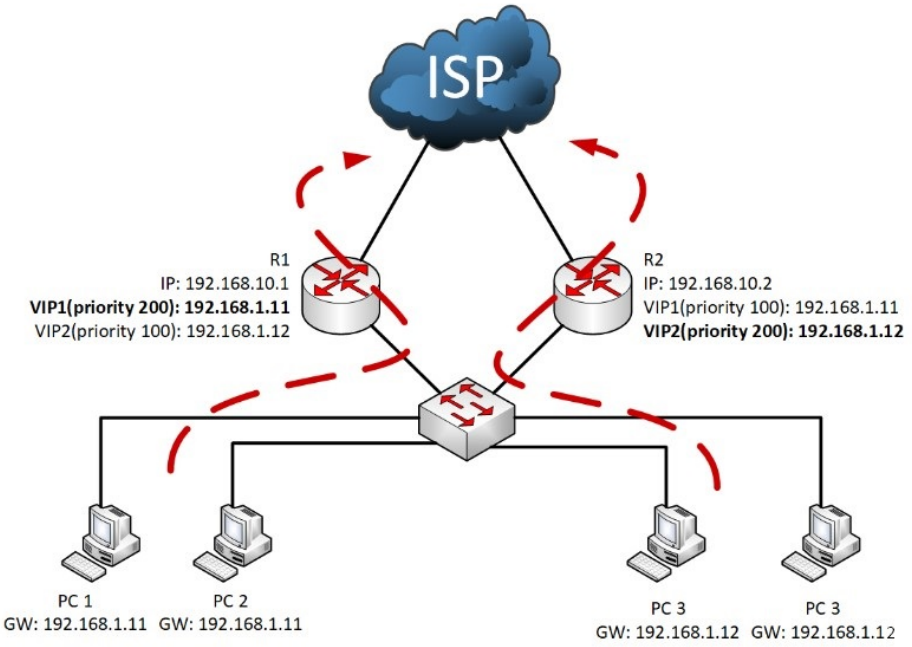
- VRRP Master – směrovač který se zabývá bezprostředně přenosem paketů a odpovídá na požadavky jdoucí na virtuální směrovač.
- VRRP Backup – směrovač očekávající potvrzení aktivního stavu VRRP Master. V případě že 3 krát neobdrží potvrzení od Master směrovače, tak se změní svůj stav na Master.

Pro každý směrovač v tomto protokolu je možné nastavit prioritu v rozsahu 1 až 254 a v případě, že Master-směrovač spadne, nový Master bude ten, který má nejvyšší prioritu. Jestliže navíc Backup směrovač má nastavený režim *preempt*, může převzat roli Master v případě, že má vyšší prioritu ve srovnání s Master bez ohledu na to, jestli Master spadl nebo ne. Komunikace Master směrovače s počítačem se provádí tak, že počítač pošle ARP požadavek na virtuální směrovač, všechny Backup směrovače zamítnou tento požadavek a Master odpoví. To znamená, že počítač bude mít ve své ARP tabulce MAC adresu Master směrovače a bude komunikovat právě s ním.

Vyvažování zátěže se provádí následujícím způsobem:

1. Vytvoří se 2 VRRP skupiny. V podstatě to znamená, že se vytvoří 2 virtuálních směrovače.
2. V nastaveních pro první skupinu je priorita první skupiny vyšší než druhé.
3. V nastaveních pro druhou skupinu je priorita druhé skupniny vyšší než první.
4. Pro půlku počítačů se nastaví jako defaultní brána první virtuální směrovač a pro ostatní druhý směrovač.

Ve výsledku to znamená že první virtuální směrovač bude zpracovávat první polovinu počítačů a druhý bude zpracovávat druhou půlku.



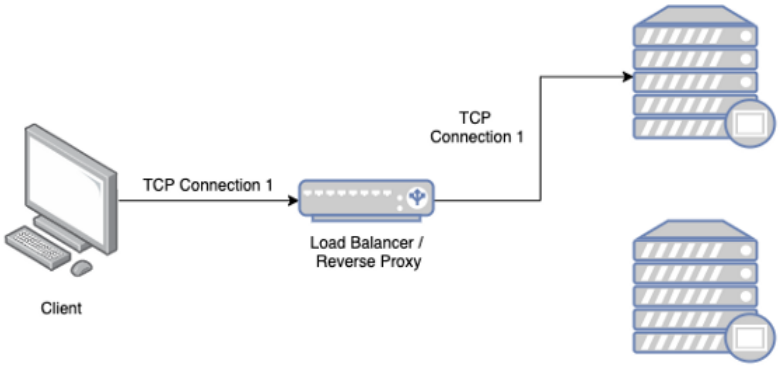
Obrázek 1.7: Ukázka vyvažování zátěže pomoci protokolu VRRP (zdroj: [19])

1.4.3 Vyvažování zátěže na transportní vrstvě

Na této vrstvě se vyvažování zátěže provádí pomocí TCP/UDP protokolu. Server pro vyvažování zátěže v tomto případě vystupuje jako *reverse-proxy*. Nachází se v jedné podsíti s aplikačními servery. Klient neví o existenci jiných serverů kromě serverů pro vyvažování zátěže. V případě navázání TCP spojení od klienta, server provádí NAT(angl. *Network Address Translation*) transformaci získaného paketu a změní cílovou destinací v TCP paketu na jeden z aplikačních serverů. Stejně tak změní cílovou destinací na klienta v případě zpáteční cesty. Volba aplikačního serveru je určena prvními několika pakety získanými od klienta [20].

Tento typ vyvažování zátěže je jednoduchý na realizaci a přestože nekontroluje obsah paketů má velkou propustnost dat. Na této úrovni může problém nastat v případě zaslání velkého počtu paketů od jednoho klienta, protože celý počet bude vždycky zpracovávat stejný server.

Všimněme si, že celá tato akce se děje nejenom na transportní vrstvě ale také i na síťové, přestože pracujeme s IP adresami.



Obrázek 1.8: Ukázka vyvažování zátěže na transportní vrstvě (zdroj: [21])

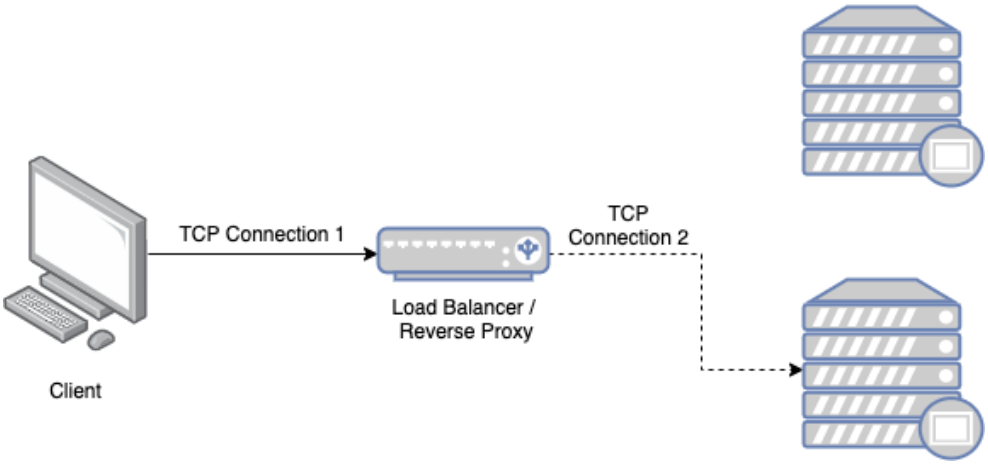
1.4.4 Vyvažování zátěže na aplikační vrstvě

Podobně jako na transportní vrstvě je vyvažovač zátěže reprezentován jako *reverse proxy*. V případě získání HTTP požadavků od klienta, zpracuje vyvažovač zátěže jeho obsah, vytvoří nové TCP spojení a na základě získaných údajů určí odpovídající aplikační server.

Díky tomu, že se objevila možnost zpracovávat HTTP požadavky vznikl prostor pro zlepšení algoritmů pro vyvažování zátěže. Pro zpracování HTTP požadavků na transportní vrstvě se používá protokol TCP. To znamená, že oproti předchozí kapitole v průběhu řízení se vytvoří 2 TCP spojení.

Existují 2 typy odesílání odpovědi klientovi:

- Prostřednictvím serveru pro vyvažování zátěže.
- DSR (angl. *Direct Server Return*) – odesílání odpovědí od aplikačního serveru klientovi napřímo. Tento typ je lepší v případě, že velikost odpovědí je důrazné větší než velikost požadavku. Přestože není odpověď zpracována vyvažovačem zátěže, zvětšuje se rychlost přenosu paketů. Jako příklad můžeme uvést přenos médií.



Obrázek 1.9: Ukázka vyvažování zátěže na aplikační vrstvě (zdroj: [21])

1.4.5 DNS vyvažování zátěže

V DNS tabulce pro jedno doménové jméno může být více záznamů. Navíc, DNS server má takovou vlastnost, že každému klientovi vrací jinou IP adresu pro požadované doménové jméno. [22].

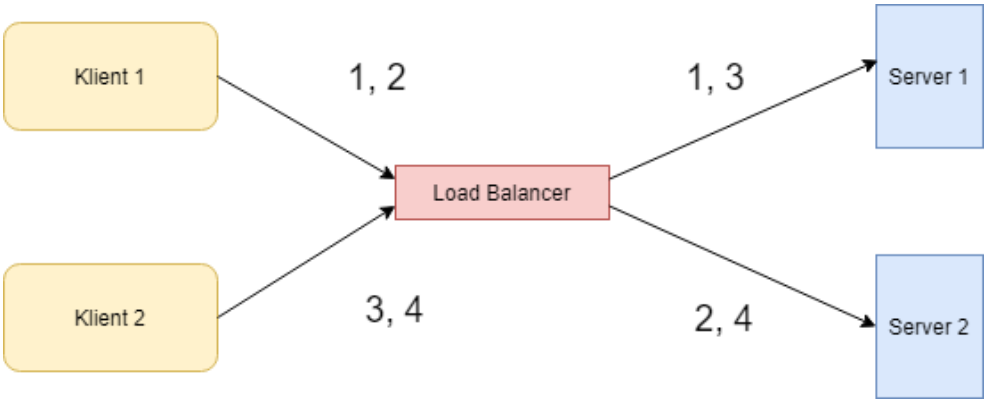
Problém nastává v případě, když jeden z aplikačních serverů spadne a DNS server o tom nebude vědět. V této situaci nastává potřeba čekat vypršení doby validity IP adresy nedostupného serveru a následné aktualizace DNS záznamů.

1.4.6 Algoritmy pro vyvažování zátěže

V teto sekci budou popsané algoritmy, které se využívají pro vyvažování zátěže.

1.4.6.1 Round-Robin

Nejpoužívanější algoritmus pro vyvažování zátěže. Je založen na iterativním principu. Od prvního serveru postupně posílá požadavky na následující server v pořadí. Předpokládáme, že máme 2 aplikační servery a získáváme 4 požadavky od klientů. První požadavek je přesměrován do prvního aplikačního serveru, druhý do druhého. Přestože nám končí počet serverů, začínáme zase od prvního. Třetí požadavek se přesměruje do prvního aplikačního serveru, čtvrtý do druhého (obr. 1.10). Tento algoritmus se používá na všech vrstvách, kde se provádí vyvažování zátěže, je lehký na pochopení a implementaci. Není vhodný v případě různé výkonnosti aplikačních serverů.



Obrázek 1.10: Ukázka Round-Robin vyvažování zátěže

1.4.6.2 Weighted Round-Robin

Je podobný standardnímu Round-Robin s výjimkou toho, že každému aplikačnímu serveru je přiřazena váha v závislosti na výkonu. V podstatě to znamená, kolikrát více požadavků může zpracovávat jeden server ve srovnání s druhým. To znamená, že výkonnější server bude získávat více požadavků. Avšak i tady se nekontroluje jejich výpočetní náročnost. Může nastat situace kdy se všechny obtížné požadavky přesměrují na jeden server, což způsobí přetížení.

1.4.6.3 Least Connections

Vyřešit problém, který je popsaný výše, je možné pomocí algoritmu Least Connections. Tento algoritmus bere do úvahy počet aktuálně zpracovaných požadavků každým serverem a na základě získané informace pošle požadavek na ten, který má aktivních zpracování nejméně.

1.4.6.4 Weighted Least Connections

Stejně jako u algoritmu Weighted Round Robin (viz. 1.4.6.2) se jednotlivým serverům přidává váha v závislosti na výkonu. Na rozdíl od všech dříve uvedených algoritmů kontroluje tento algoritmus 2 hodnoty: počet aktuálních zpracování požadavků a výkonnost serveru.

1.4.6.5 Sticky Session

Na IP adresy klienta aplikuje vyvažovač zátěže hešovací funkcí v době zpracování požadavků. Podle získané haše se klientovi přiřadí trvalý aplikační server, se kterým bude vždycky komunikovat. Hešovací funkce taky uvažuje výkonnost a počet aplikačních serverů. To se může hodit pro kešování nebo autorizaci.

Kapitola 2

Návrh

V této kapitole popíšeme projekt, který byl realizován a také popíšeme návrh k implementaci dílčích částí.

2.1 Projekt „Manažerský Informační Systém“

Manažerský Informační Systém (MIS) – web aplikace pro sledování firemních procesů, je novou verzí už existujícího informačního systému. Projekt je založen na principu mikroservisní architektury, kde implementace každé komponenty zaměřená na splnění jednotlivého business požadavků. Komponenty mezi sebou komunikují prostřednictvím REST API. MIS bude možné nakonfigurovat podle potřeb zákazníka, přidat nebo odebrat jednotlivé služby. V případě použití aplikace velkou společností je potřeba zajistit vysokou dostupnost pomocí volby vhodného Service Discovery Serveru.

2.2 Požadavky na aplikaci

Na aplikaci jsou kladené následující funkční požadavky:

- aplikace má získávat údaje o ekonomických subjektech prostřednictvím komunikace s ARES API;
- aplikace má získávat bankovní transakce prostřednictvím komunikace s ČSOB API;
- aplikace má získávat bankovní transakce prostřednictvím komunikace s FIO API;
- aplikace má získávat bankovní transakce prostřednictvím komunikace s KB API;
- aplikace má získávat údaje z registru územní identifikace prostřednictvím komunikace s RUIAN API.

Na aplikaci jsou kladené následující nefunkční požadavky:

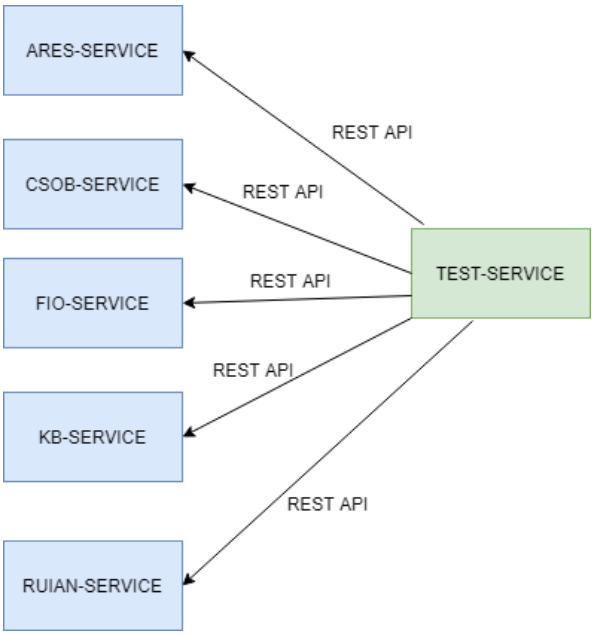
- škálovatelnost – schopnost zvětšovat počet aplikačních serverů;
- dostupnost – klienti mohou komunikovat pomocí REST API a HTTP protokolu;
- v případě velké zátěže, minimalizovat dobu čekání na zpracování požadavku.

2.3 Návrh implementace mikroslužeb

Pro implementaci mikroservis použijeme architektonický vzor – MVCS (Model View Controller Service). Popíšeme jednotlivé části vzoru pro naší aplikaci:

- Model je reprezentovaný datovými třídy.
- Jelikož frontend není požadovaný, View reprezentovaný nebude.
- Controller je reprezentován REST rozhraním.
- Service je reprezentována service třídou spravující business logiku.

Přestože cílem implementace mikroslužeb je testování Service Discovery Serveru a vyvažování zátěže, databázová vrstva implementovaná nebude.



Obrázek 2.1: Ukázka komunikace mezi mikroservisy

2.4 Volba Service Discovery Serveru

Jedním z nejdůležitějších bodu bakalářské práce je návrh Service Discovery Serveru pro aplikaci MIS. Na SDS byly kladené následující požadavky:

1. automatická registrace a deregistrace služeb
2. vysoká dostupnost
3. klienty mají komunikovat mezi sebou pomocí DNS požadavků
4. snadná integrace do frameworku Spring
5. integrovaný UI[12] pro lepší odhalení případných chyb
6. key-value datastore pro uchování konfiguračních proměnných je výhodou

Pro diskuzi byly zvolené 4 nejpoužívanější řešení service discovery:

- Zookeeper;
- etcd;
- Eureka;
- HashiCorp Consul.

2.4.1 Zookeeper

Produkt společnosti Apache. Model datových zdrojů implementovaný v podobě systému souborů, kde každý soubor je identifikovaný cestou k němu. Kořenem modelu je `/`, jednotlivé datové zdroje jsou potomky kořene jmenované *znode*. Prozkoumáme Zookeeper na splnění požadavků discovery serveru, které jsme uvedli na začátku podkapitoly [23]:

1. automatická registrace je dostupná pomocí nástroje Registrátor;
2. znody jsou rozdělené do 2 kategorií:
   - leader – spravuje všechny požadavky na zápis;
   - follower – spravuje požadavky na čtení.

   Jednotlivé znody jsou provázané mezi sebou a mají díky propagaci změn stejnou kopii databáze jako leader. V případě výpadku leaderu serveru se novým leaderem stane jeden z followerů. Což zaručí dostupnost.
3. klienty mezi sebou nekomunikují pomocí DNS požadavků

[12] UI – User Interface

4. snadno se integruje do Spring pomocí přidání závislosti do nástroje pro automatizaci buildu (Maven, Gradle)
5. neexistuje integrovaný UI
6. nemá key-value datastore

2.4.2 etcd

Primárně se používá jako key/value datastore pro Kubernetes ale je možné použít i pro service discovery. Prozkoumáme jednotlivé body:

1. automatická registrace je dostupná pomocí nástroje Registrátor
2. dostupnost je zaručena stejným způsobem, jako u Zookeeper (viz. 2.4.1)
3. klienty mezi sebou nekomunikují pomocí DNS požadavků
4. snadno se integruje do Spring pomocí přidání závislosti do nástrojů pro automatizaci buildu (Maven, Gradle)
5. neexistuje integrovaný UI, je možné stáhnout jako doplněk, napr. *etcd-console* [24]
6. má key-value datastore

2.4.3 Eureka

Service Discovery Server společnosti Netflix. Podíváme se detailně, jestli vyhovuje požadavkům pro service discovery:

1. Aplikační servery se při spuštění automaticky registrují do Eureka Serveru pomocí REST API. V případě 3 krát neodeslaného potvrzení o aktivním stavu server deregistruje klienta.
2. Eureka zaručuje vysokou dostupnost v případě spuštění klastru, který se skládá z minimálně 3 Eureka serverů. Eureka klienti vědí o všech spuštěných serverech a v případě výpadku některého z nich, se připojují na jiný fungující server. V extrémním případě, kdy všechny Eureka servery spadnou, klienti budou moct komunikovat mezi sebou díky kešování dat z Eureka serveru.
3. Klienty mezi sebou komunikují pomocí DNS požadavků.
4. Snadno se integruje do Spring pomocí přidání závislosti do nástrojů pro automatizaci buildu (Maven, Gradle).
5. Má integrovaný UI.
6. Je možné integrovat key-value datastore Vault.

2.4.4 Consul

Nejpoužívanější Service Discovery Server. Produkt společnosti Hashicorp. Prozkoumáme jestli je vhodný pro účely naše aplikace:

1. Aplikační servery se při spuštění automaticky registrují do Consul Agent pomocí REST API a HTTP. V případě 3 krát neodeslaného potvrzení o aktivním stavu server deregistruje klienta.
2. Pro dosazení vysoké spolehlivosti je potřeba spustit minimálně 3 Consul servery v režimu `server` ve stejném klášterů pro dosažení kvorumu. Kvorum je nutný pro volbu nového leader serveru v případě výpadku leadera.
3. Klienti mezi sebou komunikují pomocí DNS požadavků.
4. Snadno se integruje do Spring pomocí přidání závislosti do nástrojů pro automatizaci buildu (Maven, Gradle).
5. Má integrovaný UI.
6. Má integrovaný key-value datastore.

2.4.5 Shrnutí

Na základě analýzy jednotlivých Service Discovery Serveru bylo zjištěno, že ZooKeeper a etcd potřebují pro pohodlné použití instalaci doplňkových komponent, což není pro nás vhodné. Navíc nemají integrovaný DNS server pro komunikaci mezi jednotlivými službami. Eureka a Consul splňují požadavky kladené pro service discovery, jsou snadno konfigurovatelné, mají UI a jsou vhodné pro první seznámení s service discovery. Proto bylo rozhodnuto o implementaci a následném testování Eureka a Consul.

Kapitola 3

Implementace

V této kapitole se popíše implementace jednotlivých komponent aplikace.

3.1 Zvolené technologie

Aplikace je implementována v programovacím jazyce Kotlin. Jednotlivé mikroslužby používají framework Spring a jsou nasazeny na aplikační server Tomcat. Správa závislostí je vyřešena pomocí nástroje Gradle. Pro service discovery jsou použité Netflix Eureka a Hashicorp Consul. Řešením pro vyvažování zátěže je knihovna Ribbon.

3.1.1 Programovací jazyk

Kotlin – staticky typovaný, objektově orientovaný jazyk, kompatibilní s Java. Je relativně nový, začátek vývoje je datovaný rokem 2011, o vývoj se stará společnost JetBrains. Primárně je oficiálním jazykem pro Android aplikace. Mimo jiné podporuje knihovnu Spring a používá se i pro webový vývoj. Cílem vývojářů Kotlinu bylo vytvořit lehce pochopitelný jazyk, který je zaměřený na efektivitu vývoje a řeší časté problémy, se kterými programátoři se setkávají v Javě. Díky kompatibilitě s Javou je také možné vytvářet smíšené projekty, které požívají oba dva jazyky.

3.1.2 Nástroj pro automatizaci sestavení programu

Jako nástroj pro automatizaci sestavení programu se používá Gradle. Na rozdíl od konkurentů je flexibilnější díky možnosti napsání skriptu pro jednotlivé úlohy. Pro popis úloh se používá jazyk Groovy nebo Kotlin.

3.1.3 Web framework

Spring je nejpoužívanější framework pro vývoj webových aplikací, podporuje jazyky Java, Kotlin, Groovy. Skládá se z 21 modulů. Každý z modulů

je zodpovědný za integrování určité funkcionality. Všechny moduly jsou implementovány takovým způsobem, aby se programátor mohl zabývat jenom implementací business logiky. Jádro Spring je založeno na principu obráceného řízení (angl. *Inversion of Control*) a realizuje tento princip pomocí vkládání zavislosti(angl. *Dependency Injection*). Tento přístup přináší nižší provázanost mezi jednotlivými komponenty a usnadňuje konfiguraci aplikace. Pro implementaci této bakalářské práce jsou využité moduly Spring Boot, Spring Web a Spring Cloud.

3.1.4 Service discovery

Na základě analýzy provedené v části 2.4 pro service discovery jsou použité Netflix Eureka a Hashicorp Consul.

3.1.5 Vyvažovač zátěže

Pro vyvažování zátěže je použitý Ribbon. Je to klient-side vyvažovač zátěže, kompatibilní s Eureka a Consul. Má dostupné následující pravidla pro vyvažování zátěže:

- `AvailabilityFilteringRule`
- `BestAvailableRule`
- `ClientConfigEnabledRoundRobinRule`
- `RoundRobinRule`
- `WeightedResponseTimeRule`
- `ZoneAvoidanceRule`

Detalnejsi přehled jednotlivých metod je možné získat v dokumentaci Ribbon [25].

3.1.6 Dokumentace

Pro tvorbu dokumentace je použitý framework Swagger. Jednou z jeho výhod je to, že umožňuje nejen interaktivně zobrazit specifikací, ale také vyzkoušet jednotlivé požadavky pomocí Swagger UI. Z hlediska vývojáře API je vhodný díky tomu, že dokumentace je snadno generovatelná pomocí anotací.

3.2 Správa závislostí

Pro napsání Gradle tasku a správu závislostí je používán Kotlin DSL[13]. Organizaci závislosti je vhodné zařídit takovým způsobem, aby je bylo možné vyzvednout z jednoho místa a použít ve všech modulech projektu. Navíc, tento přístup dovoluje vyhnout se problémům s nekompatibilními verzemi závislostí v různých modulech. Pro splnění těchto cílů byl vytvořen modul `buildSrc` a třída `Dependencies.kt`, ve které jsou popsané jednotlivé závislosti. Modul `buildSrc` je dostupný všem souborům `build.gradle.kts` jako knihovna.

```
1  object Kotlin {
2      const val jvmId = "jvm"
3      const val springId = "plugin.spring"
4      const val jpaId = "org.jetbrains.kotlin.plugin.jpa"
5      const val allOpenId = "org.jetbrains.kotlin.plugin.allopen"
6      const val noArgsId = "org.jetbrains.kotlin.plugin.noarg"
7      const val version = "1.4.10"
8  }
9
10 object SpringBoot {
11     const val id = "org.springframework.boot"
12     const val version = "2.3.5.RELEASE"
13 }
```

Listing 1: Fragment souboru Dependecies.kt

3.3 Implementace jednotlivých mikroslužeb

Tato sekce popíše realizaci jednotlivých mikroslužeb pro účely service discovery a vyvažování zátěže. Hlavním cílem implementace je konfigurace propojení mezi mikroslužby a veřejnými API pro následné upřesnění požadavků.

3.3.1 ARES

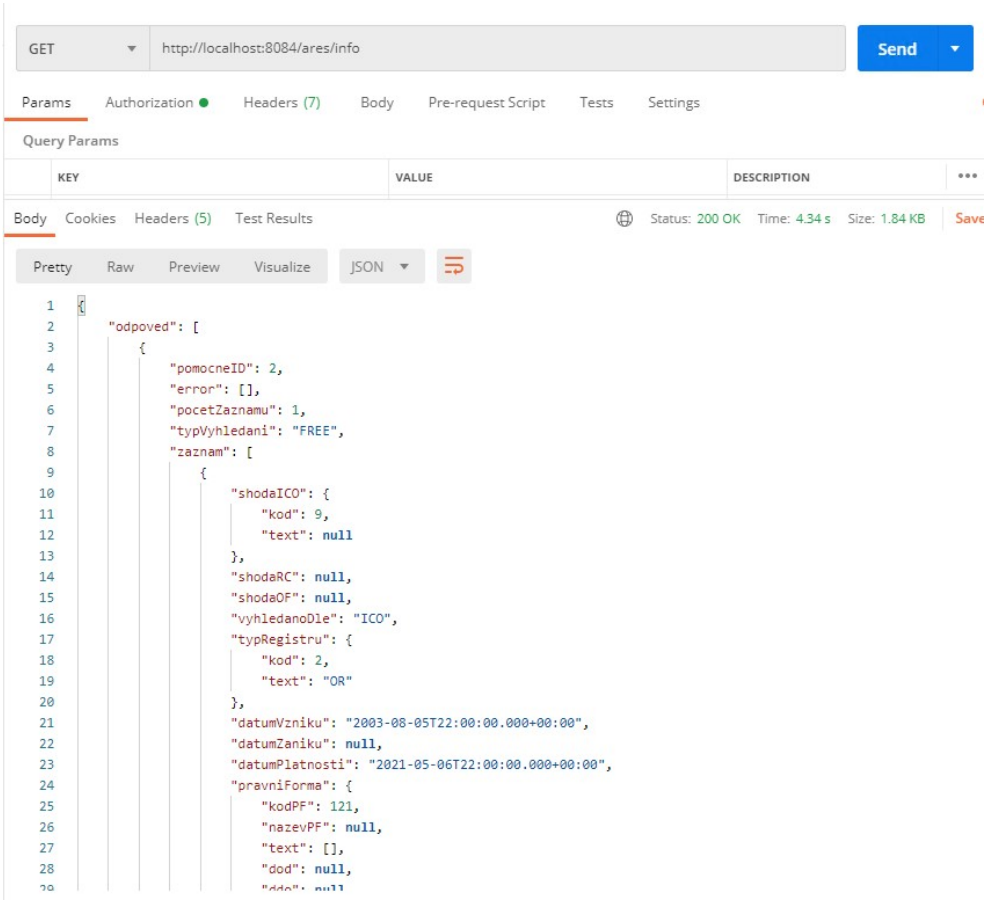
Administrativní registr ekonomických subjektů je systém umožňující vyhledávání ekonomických subjektů v České Republice. Pro popis webového API využívá webovou službu SOAP a jednotlivá rozhraní jsou popsána pomocí WSDL. Pro přístup k jednotlivým datům z ARES je možné použít metodu GET která nevyžaduje digitální podpis nebo metodu POST. Pro generaci tříd popsaných ve WSDL souboru `standard.wsdl` je používán příkaz:

Jelikož účelem implementace tohoto modulu je zabezpečení komunikace s webovým API, tak bylo pro generaci požadavků vytvořeno REST rozhraní s

[13]DSL – Domain Specific Language

```
wsimport -keep -verbose standard.wsdl
```

endpointem /ares/info, kde pomocí HTTP metody GET se získá informace o konkrétním IČO uvedeném v dokumentaci ARES. Výsledek je mapovaný na třídu `AresOdpovedi.java`



Obrázek 3.1: Ukázka HTTP požadavku na službu ARES

GET:	/ares/info

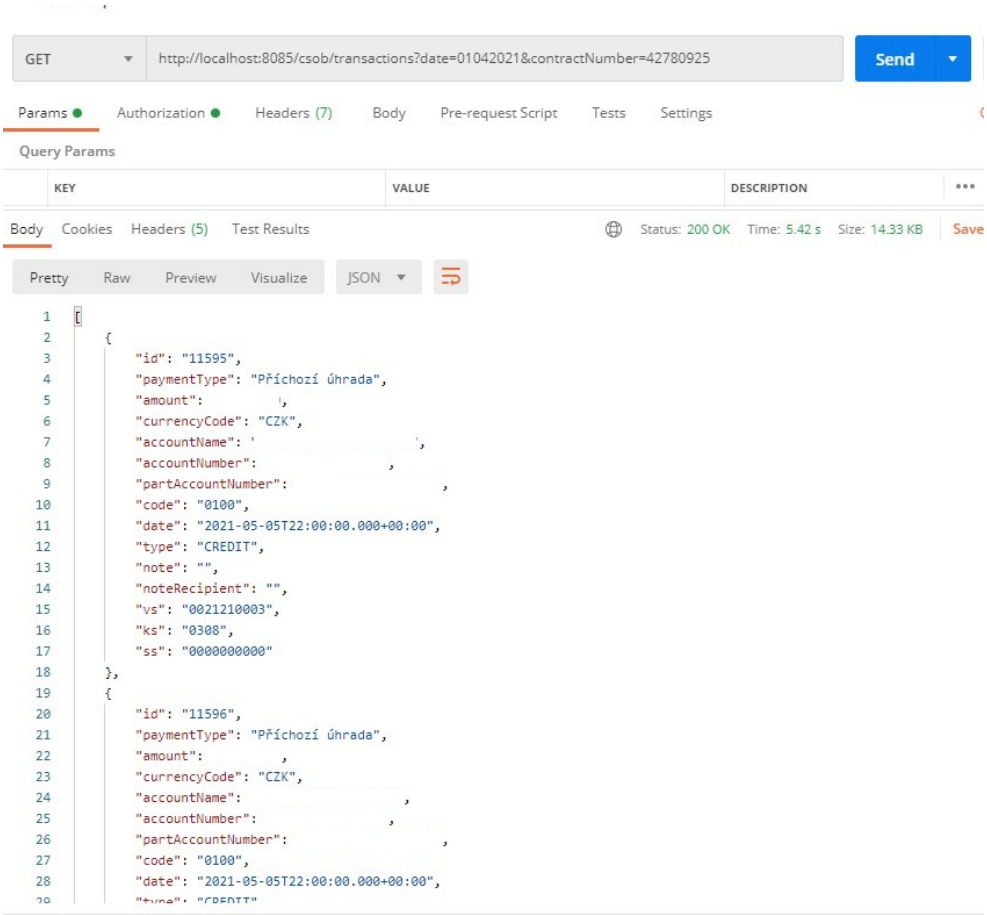
Tabulka 3.1: ARES tabulka endpointů

3.3.2 ČSOB

Československá obchodní banka poskytuje API pro získání transakcí z bankovního účtu. Pro přístup k datům je možné použít testovací prostředí, nebo produkční prostředí pomocí certifikátu a hesla. Díky tomu, že společnost e-invent s.r.o. má účet v této bance se podařilo použít produkční prostředí a bylo vytvořeno REST rozhraní s endpointem /csob/transactions, kde `date` je datum od kterého chceme získávat transakce, `contractNumber` je číslo smlouvy. Výsledek je mapován na datovou třídu `BankTransaction.kt` .

GET:	/csob/transactions?date&contractNumber

Tabulka 3.2: ČSOB tabulka endpointů



Obrázek 3.2: Ukázka HTTP požadavku na ČSOB službu

3.3.3 FIO

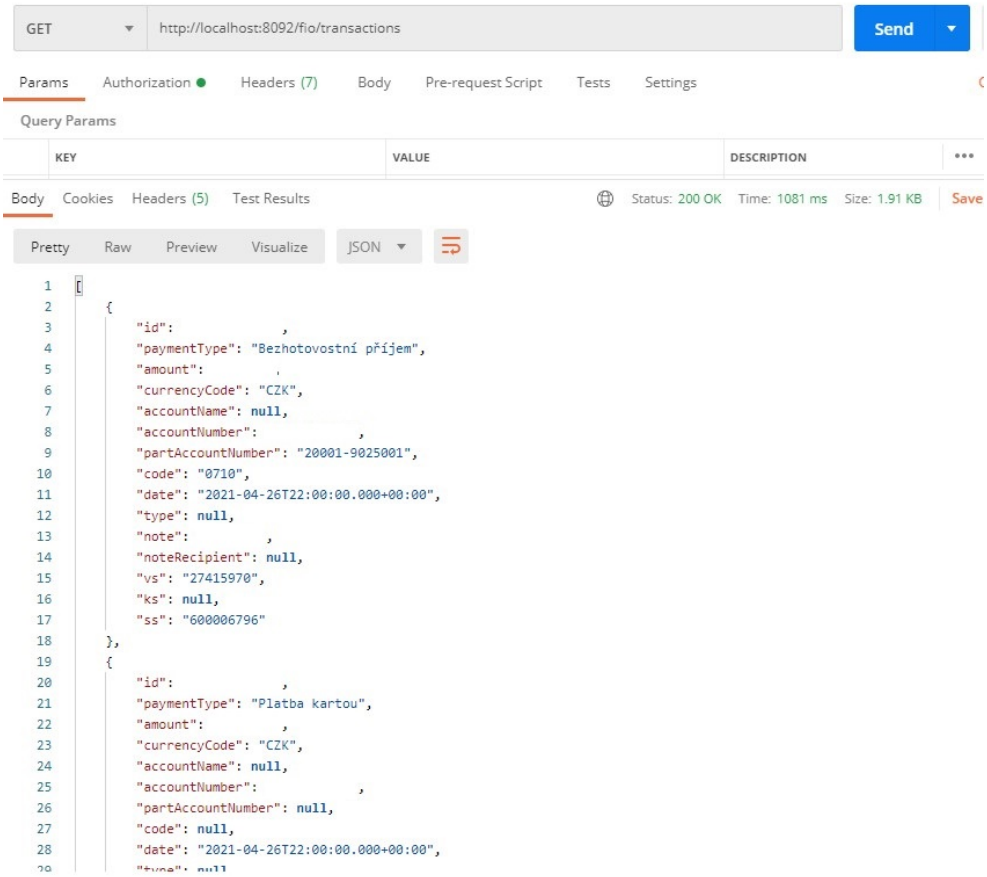
FIO banka poskytuje API pro získávání transakcí z bankovního účtu. Jako webová služba se používá SOAP a datové třídy jsou generované pomocí XSD schématu. Pro generování datových tříd je použitý příkaz:

```
1 xjc -d tmp_xsd schema.xsd
```

Pro získání výpisu z účtu je potřeba mít access token nebo použít testovací verzi. Jelikož e-innvent s.r.o. má účet v FIO bance, získání výpisů bylo provedeno pro produkční prostředí. Následně je pro získání výpisu z účtu vytvořené REST rozhraní s endpointem /fio/transactions/. Výsledek se mapuje do třídy `BankTransaction.kt`.

GET:	/fio/transactions

Tabulka 3.3: FIO tabulka endpointů



Obrázek 3.3: Ukázka HTTP požadavku na FIO službu

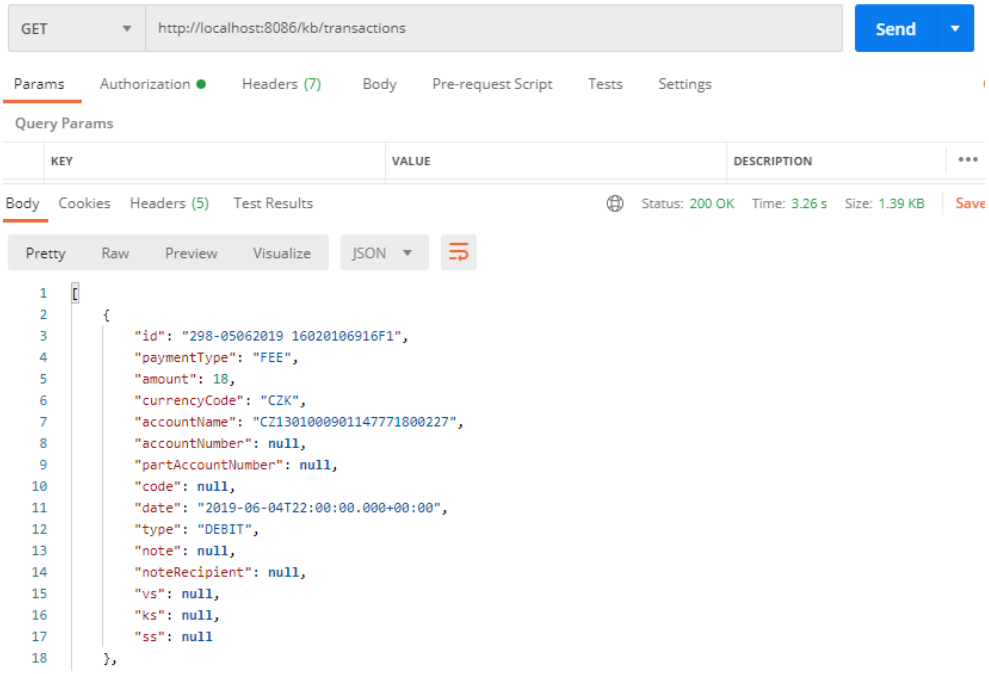
3.3.4 KB

Komerční banka nabízí API pro získání transakce z bankovního účtu. Toto rozhraní se zatím nachází ve stavu beta testování. Jako webová služba se používá architektura REST. Jelikož e-invent s.r.o. nemá účet v KB bance, bylo rozhodnuto použít testovací verzi. KB banka nabízí `.jar` knihovny, kde jsou implementované metody, potřebné pro komunikaci s API a jednotlivé datové typy.

Během integrace knihoven do projektu avšak vznikl problém s chybějícími v nich závislostmi, což v důsledku znemožnilo build modulů v projektu bakalářské práce. Naštěstí se tento problém podařilo vyřešit díky tomu, že zdrojové kódy knihoven jsou umístěné na vzdáleném repositáři, což umožnilo jejich upravování. Následně byl vytvořen issue v GitHub repositáři s navrženými úpravami [26]. Pro získání transakce bylo vytvořeno REST rozhraní mající endpoint /kb/transactions/ a s výsledkem mapujícím do třídy `BankTransaction.kt`.

GET:	/kb/transactions

Tabulka 3.4: KB tabulka endpointů



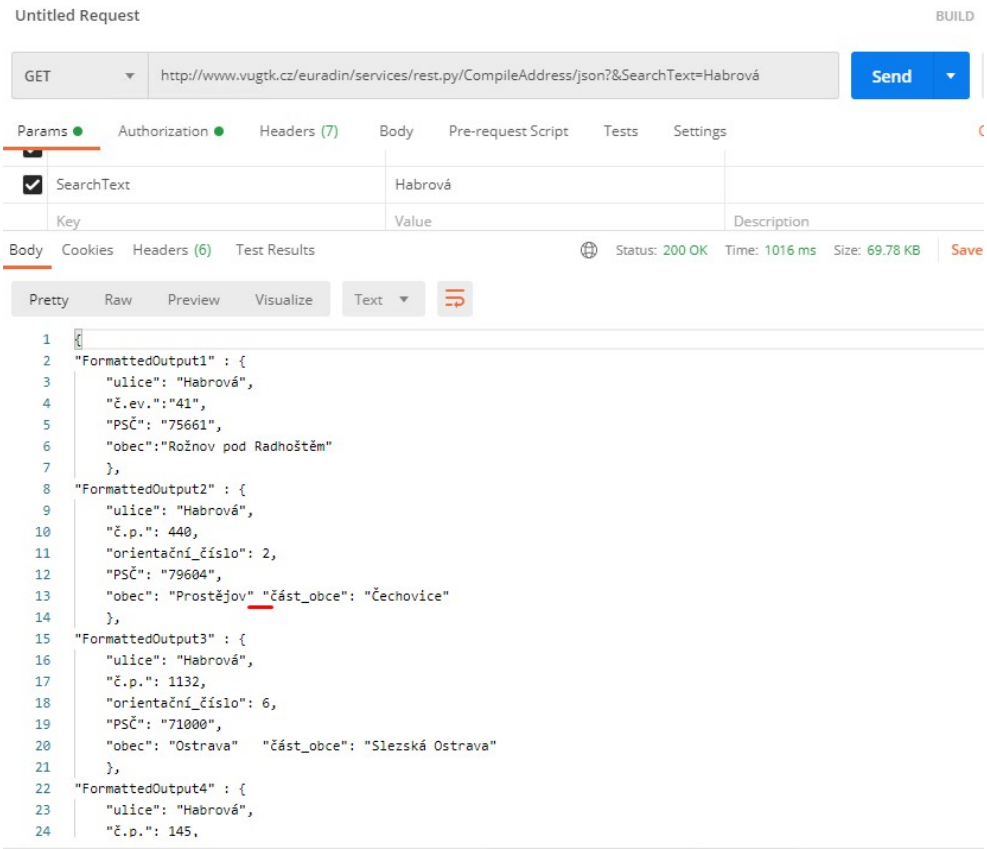
Obrázek 3.4: Ukázka HTTP požadavku na KB službu

3.3.5 RÚIAN

RÚIAN – Registr územní identifikace, adres a nemovitostí poskytuje API, které je možné volat i pomocí REST i pomocí SOAP. V implementaci bakalářské práce bylo rozhodnuto o použití REST. Pro získání údajů od RUIAN API bylo vytvořeno REST rozhraní a endpoint /ruian/info/. Výsledek se mapuje do třídy `RuianData`. Během implementace této služby bylo zjištěno, že odpověď získávána od API ve formátu JSON je špatně formátovaná, což znemožňuje správné parsování. Vzhledem k tomu, že data posílaná pomocí REST je možné posílat v různých formátech byl zvolen formát XML.

GET:	/ruian/info

Tabulka 3.5: RUIAN tabulka endpointů



Obrázek 3.5: Ukázka vadného formátování JSON odpovědi od RÚIAN API

3.3.6 TEST

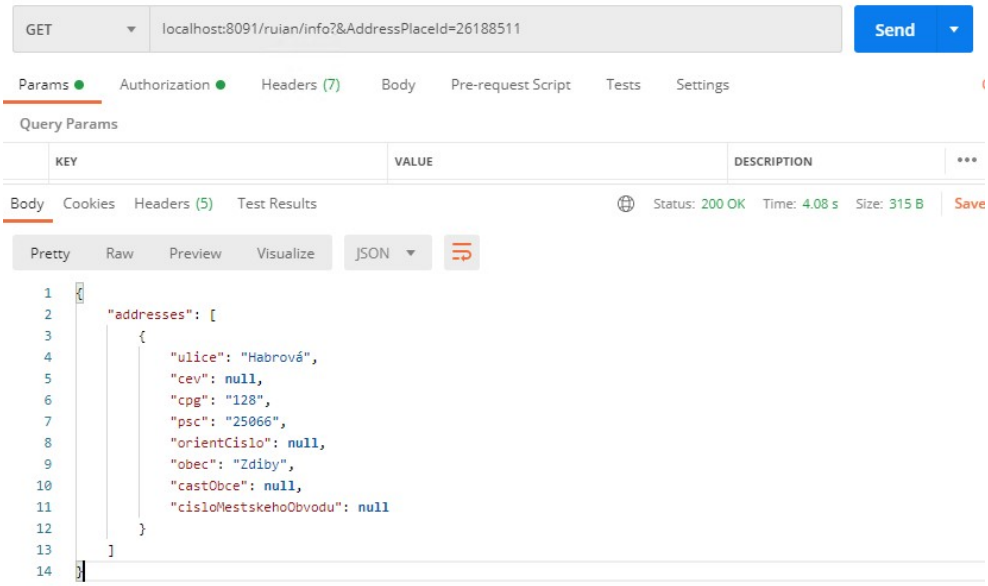
Tato mikroslužba byla vytvořena jako testovací pro testování vyvažování zátěže. Má implementováno vyvažování zátěže (viz 3.5) a právě z ní budou posílané požadavky na jednotlivé mikroslužby.

3.4 Implementace Service Discovery Serveru

Abychom mohli komunikovat mezi jednotlivými mikroservisy a připojovat nové instance služeb do klasteru, je potřeba implemetovat Service Discovery Servery. V této sekci popíšeme jejich implementace a integrace do Spring Framework.

3.4.1 Eureka

Eureka je klient-side service discovery a se skládá ze 2 základních komponent:



Obrázek 3.6: Ukázka HTTP požadavku na RÚIAN službu

- Eureka Server - server, používaný pro registraci a deregistraci klientů a taky pro service discovery.
- Eureka Client - libovolná aplikace, která se dotazuje na registr služeb, rozmístěny na Eureka Server pro identifikaci spuštěných instancí.

3.4.1.1 Implementace serveru

Pro spuštění Eureka Server je potřeba vytvořit standardní Spring Boot aplikaci anotovanou `@EnableEurekaServer`. Výsledná implementace potom bude vypadat takto:

```
1 @EnableEurekaServer
2 @SpringBootApplication
3 class EurekaApplication
4
5 fun main(args: Array<String>) {
6         runApplication<EurekaApplication>(*args)
7 }
```

Listing 2: Serverová třída Eureka EurekaApplication.kt

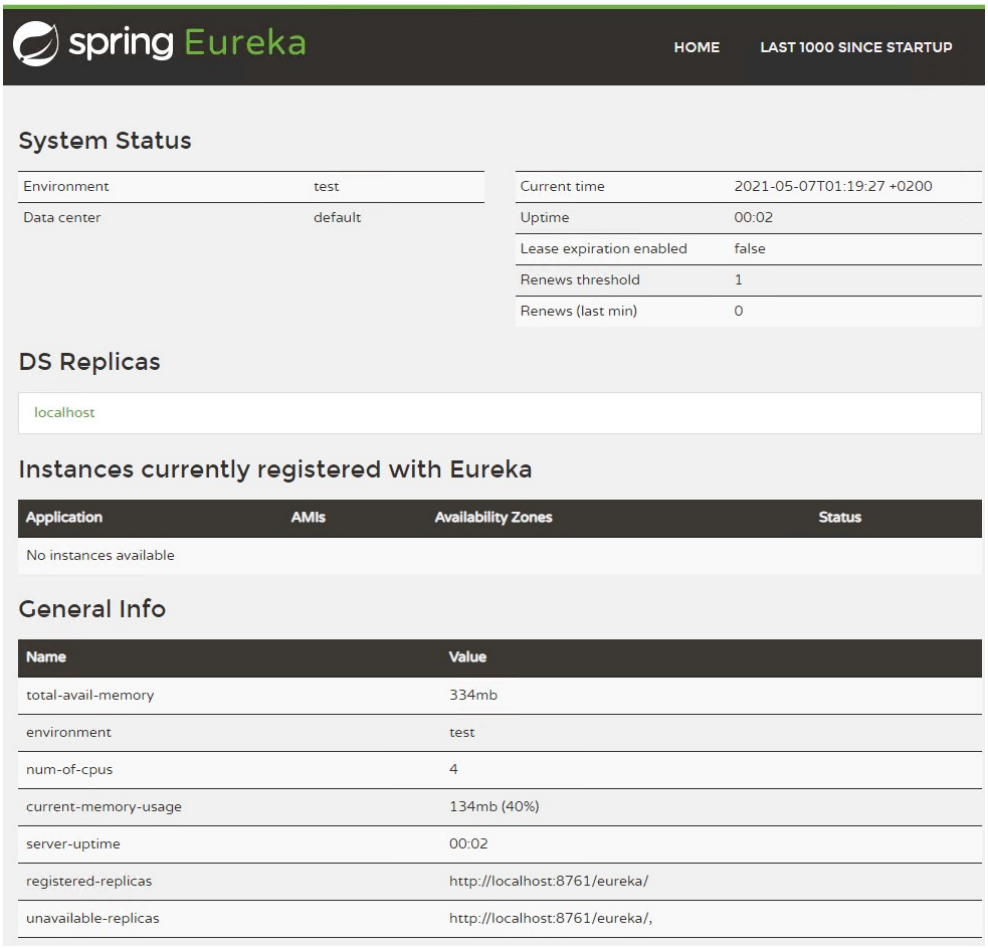
Pro konfiguraci serverů se využívá soubor `application.properties`. Standardní nastavení jsou následující:

```
server.port=8761
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

Listing 3: Serverový soubor application.properties

Popíšeme význam jednotlivých vlastností:

- `server.port` – port na kterém bude umístěný server
- `eureka.client.register-with-eureka` – jelikož Eureka Server může vystupovat jako klient [27], je potřeba vypnout registraci klienta
- `eureka.client.fetch-registry` – dotazování na serverový registr služeb



Obrázek 3.7: Eureka server uživatelské rozhraní

3.4.1.2 Implementace klienta

Jako klienty v našem případě figurují všechny implementované mikroslužby. Při použití Eureka na straně klienta je potřeba přidat závislost do Gradle build souborů: `spring-cloud-starter-netflix-eureka-client`. Abychom mohli použít naše služby v procesu service discovery a udělat je „viditelnými“ pro ostatní, je potřeba použít anotaci `@EnableEurekaClient`. Výsledná implementace vypadá takto:

```
@EnableEurekaClient
@SpringBootApplication
class AresApplication

fun main(args: Array<String>) {
    runApplication<AresApplication>(*args)
}
```

Listing 4: Klientská třída Eureka AresApplication.kt

Pro konfiguraci vlastností klienta se používá soubor `application.properties`.

```
spring.application.name=ares-service
server.port=8081
```

Listing 5: Klientský soubor application.properties

Vlastnost `spring.application.name` se používá pro nastavení jména služby sloužící k identifikaci a zobrazení v service discovery user interface.

3.4.2 Consul

Consul taky provádí service discovery na straně klienta a skládá se ze 2 komponent:

- Consul Server – server, provádějící registraci a deregistraci klientů a taky pro service discovery. Navíc má v sobě funkcionalitu Agentu
- Consul Agent – slouží pro komunikaci se službami, ověření jejich „zdraví“ následně odesílá informaci o stavu služeb na Consul Server

3.4.2.1 Spuštění serveru

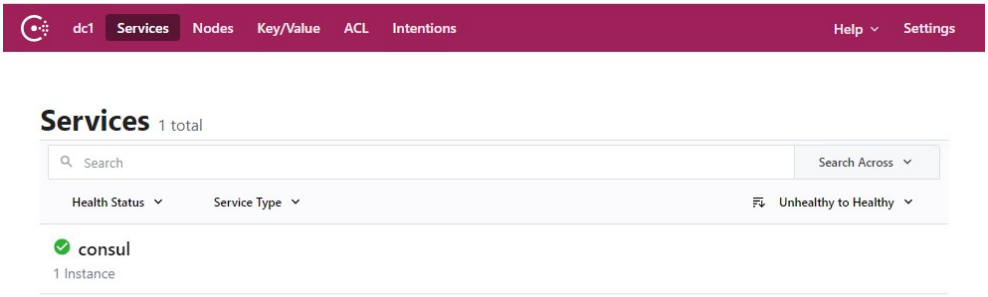
Pro spuštění serveru je potřeba stáhnout knihovnu Consul z webu aplikace [28]. Následně provést příkaz v příkazové řádce:

`consul agent -server -bootstrap-expect=1 -data-dir=consul-data -ui -bind=172.16.32.76`.

Vysvětlení jednotlivých přepínačů:

- `-bootstrap-expect` očekávaný počet Consul serverů
- `-data-dir` adresář pro uložení stavu Consul
- `-ui` zapnutí user interface

- `-bind` IP adresa pro poslech jiných členů Consul. Má být přístupná všem prvkům v mezích podsítě.



Obrázek 3.8: Consul server uživatelské rozhraní

Je potřeba poznamenat, že v tomto případě Consul Server i Consul Agent hrají svoji roli. Po spuštění serveru se objeví jedna služba pojmenována „consul“ (viz. 3.4.2.1). Je to na obrazku vidíme jednu službu pojmenovanou „consul“

3.4.2.2 Implementace klienta

Pro implementaci klienta je nutná závislost: `org.springframework.cloud:spring-cloud-starter-consul-all` . Následně je nutné přidat anotaci pro získání údajů o jiných službách od Consul Agentů `@EnableDiscoveryClient`. Výsledná klientská třída bude vypadat takto:

```
1 @EnableDiscoveryClient
2 @SpringBootApplication
3 class TestApplication
4
5
6 fun main(args: Array<String>) {
7         runApplication<TestApplication>(*args)
8 }
```

Listing 6: Klientská třída Consul TestApplication.kt

3.5 Implementace vyvažování zátěže

Pro vyvažování zátěže používáme Netflix Ribbon. Tento nástroj je kompatibilní s oběma použitými SDS a proto má pro oba dva stejnou konfiguraci.

```
@Configuration
class RibbonConfiguration {

    @Bean
    fun ribbonRule(): IRule {
        return WeightedResponseTimeRule()
    }

}
```

Listing 7: Konfigurační třída RibbonConfiguration.kt

Pro použití je potřeba navíc do klientské třídy přidat anotaci `@RibbonClients(defaultConfiguration = [RibbonConfiguration::class])`
Pro odesílání požadavků je použita třída `RestTemplate`. Jelikož používáme vyvažování zátěže, je potřeba beanu této třídy anotovat `@LoadBalanced`. Tímto způsobem je zajištěna automatická konfigurace směrování požadavků.

```
@RestController
class RequestController @Autowired constructor (
    var restTemplate: RestTemplate,
        ) {

    @RequestMapping("/test")
    fun testRequest(){
        restTemplate.getForObject(URI("http://ares-service/ares/info"),
        AresOdpovedi::class.java)
    }

}
```

Listing 8: Ukázka použití RestTemplate

Kapitola 4

Testování

Pro testování aplikace byly zvoleny 4 typy testů: integrační a unit testy pro jednotlivé mikroslužby, integrační testy komunikace mezi nimi a zátěžový test vyvažování zátěže. První typ testu je potřebný pro ověření správného fungování jednotlivých metod. Integrační testy pro mikroslužby zkontrolují komunikaci mikroslužeb s veřejnými API. Integrační test komunikace mezi mikroslužbami a ověří, jestli mezi sebou mohou jednotlivé služby komunikovat a následný zátěžový test ověří fungování vyvažování zátěže. Popsané typy testu pokryjí celou funkcionalitu programu. V této sekci bude detailně rozepsaný postup a výsledky jednotlivých testů.

4.1 Unit testovaní

Unit testovaní je důležitým prvkem testovaní celé aplikace. Pomoci tohoto typu testu je možné ověřit korektnost jednotlivých metod programu a odhalit chyby ještě v procesu vývoje. V kontextu projektu bakalářské práce jsou unit testy aplikované na metody, generující HTTP požadavek a následné mapování odpovědí do datové třídy.

```
1      fun testGetInfo(){
2          assertNotNull(aresService.getInfo("27074358", null, null))
3          var aresResponse = aresService.getInfo("27074358", null, null)
4          assertEquals(aresResponse.odpovedPocet, 1)
5          aresResponse = aresService.getInfo(null, "Asseco", null)
6          assertEquals(aresResponse.odpovedPocet, 1)
7          aresResponse = aresService.getInfo(null, null, null)
8          assertNotNull(aresResponse.odpoved[0].error)
9      }
```

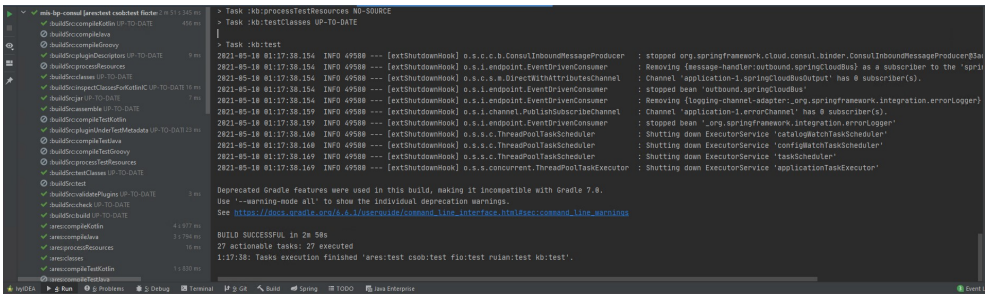
Listing 9: Ukázka unit testu mikroslužby ARES

4.2 Integrační testování jednotlivých mikroslužeb

Pro každou z mikroslužeb jsou velmi důležité implementované testy kontorolleru a servis zaručující její správnou komunikaci s vnějšími API. Bylo ověřeno jejich správné sestavování požadavků, testování endpointů pro kontroller a v případě statických údajů, testování mapování do datové třídy. Pro ilustraci výsledku všech testů byl vytvořen Gradle příkaz: `gradle ares:test csob:test fio:test ruian:test kb:test .`

```
1     @Test
2     fun testRequest() {
3         mockMvc.perform(MockMvcRequestBuilders
4         .get("/ruian/info?AddressPlaceId=26188511&SuppressID=off",1))
5             .andExpect(MockMvcResultMatchers.status().isOk())
6             .andExpect(MockMvcResultMatchers.content()
7             .contentType(MediaType.APPLICATION_JSON))
8             .andExpect(jsonPath("$.addresses.[0].ulice", equalTo("Habrová")))
9             .andExpect(jsonPath("$.addresses.[0].psc", equalTo("25066")))
10            .andExpect(jsonPath("$.addresses.[0].obec", equalTo("Zdiby")))
11    }
```

Listing 10: Ukázka integracniho testu mikroslužby ARES



Obrázek 4.1: Výsledky integračního testování mikroslužeb

4.3 Integrační testování komunikace mezi mikroslužbami

Pro komunikaci mezi mikroslužbami se používá SDS, a proto je potřeba tento typ testování rozdělit na 2 části:

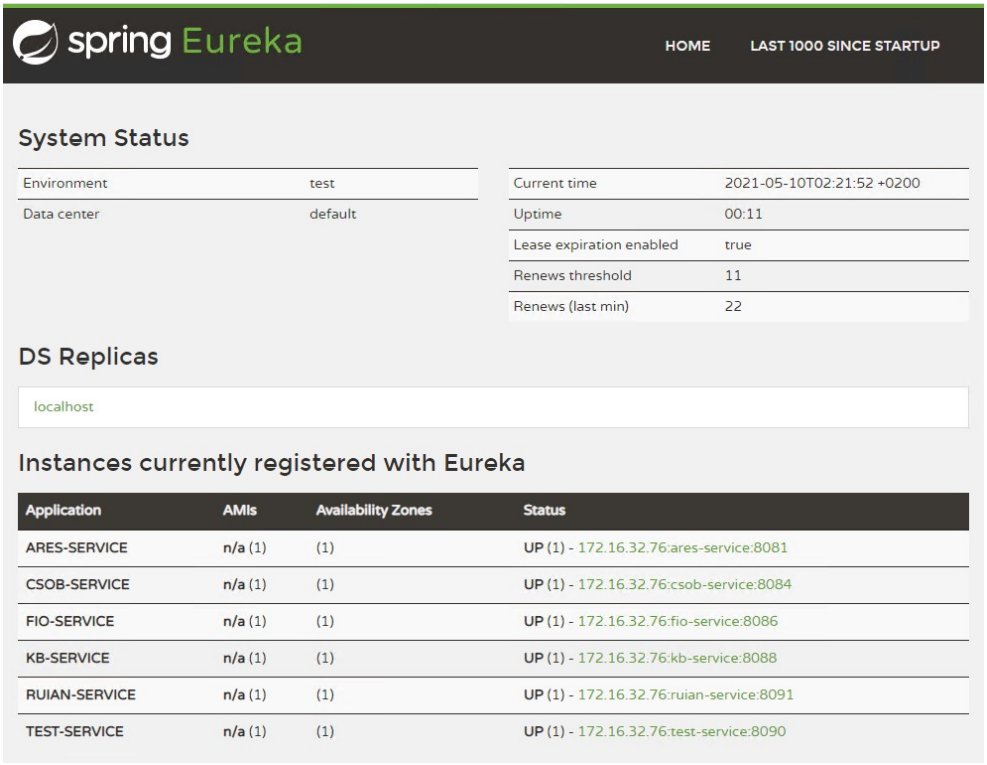
- správná registrace a deregistrace jednotlivých mikroslužeb v SDS;

- bezchybná komunikace mezi mikroslužbami pomocí doménové adresy.

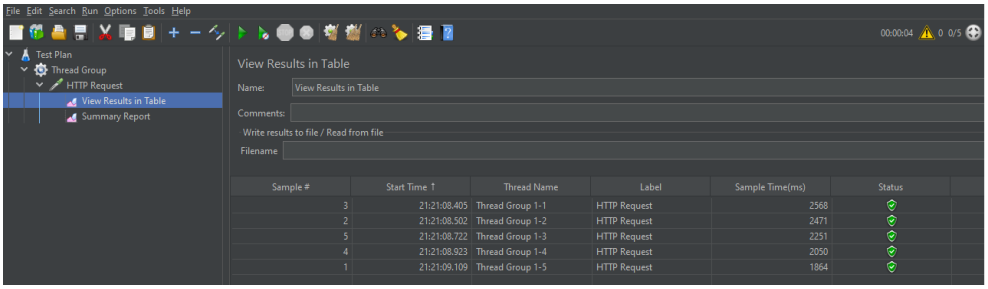
Pro tyto účely byla v mikroslužbě `test` vytvořená metoda `testRequest()`, kde vytvořené HTTP požadavky s cílovou doménovou adresou každé z mikroslužeb. Pomocí této metody můžeme ověřit komunikaci se všemi mikroslužbami. Pro testování komunikace byl použit nástroj Apache JMeter, který dovoluje testovat HTTP požadavky.

4.3.1 Eureka

Po spuštění byly všechny mikroslužby úspěšně zaregistrované v SDS a následně po zastavení byly deregistrované. Testování komunikace s mikroslužbami proběhlo úspěšně.



Obrázek 4.2: Eureka tabulka registrace mikroslužby

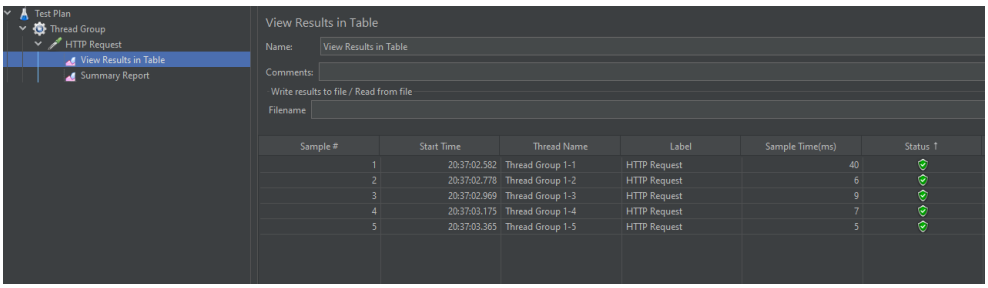


Obrázek 4.3: Eureka test komunikace s mikroslužbami

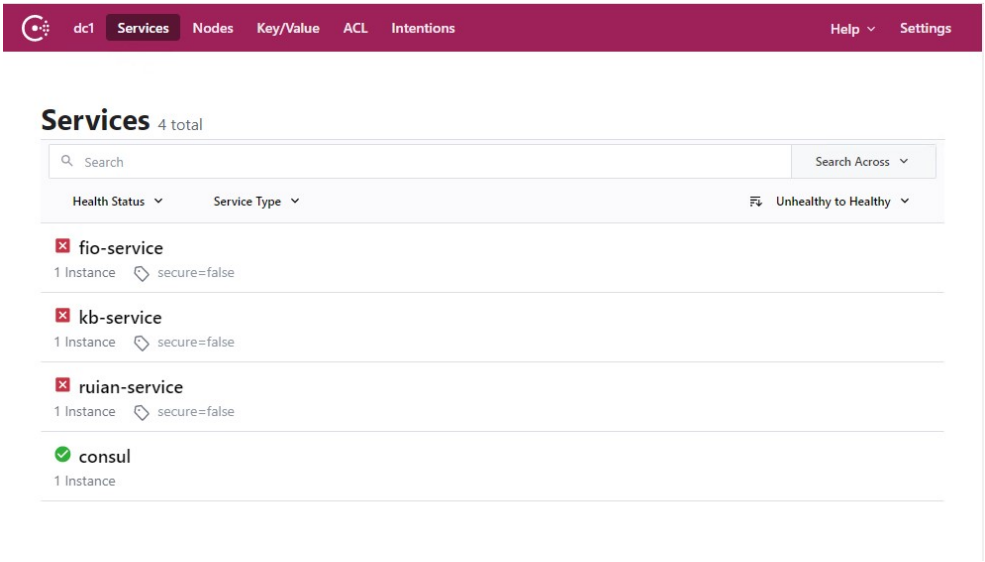
4.3.2 Consul

Všechny mikroslužby byly po spuštění úspěšně zaregistrované, avšak po zastavení nebyly ihned deregistrované. Podle dokumentace je termín odhlášení mikroslužby 72 hodin [29]. Tato hodnota se ovšem dá zmenšit a nastavit pomocí parametru:
`spring.cloud.consul.discovery.health-check-critical-timeout`.

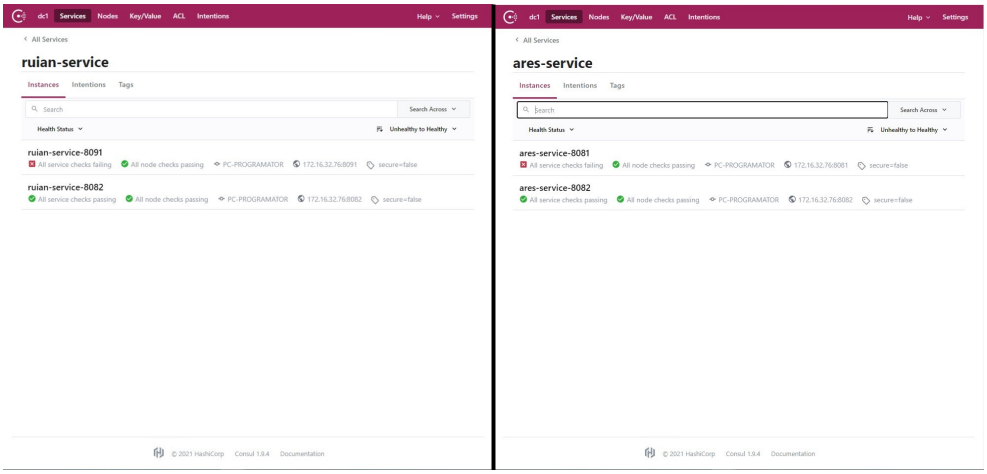
Během testování funkčnosti tohoto parametru jsem následně narazil na chybu v reprezentaci aktivních služeb. V momentě když je služba vypnutá a čas nastavený parametrem ještě nevypršel se spouští jiná službu na stejném portu. Consul ukáže, že jsou aktivní obě dvě služby. Následně se zjistilo, že `health-check` odeslaný serverem, se odesílá na IP adresu a port, které této služby mají stejné. Tohle se vyřešilo nastavením `heartbeat` a `time to live` u klienta. Přidání těchto nastavení znamená, že klienti budou odesílat na server potvrzení toho, že jsou v aktivním stavu. Neaktivní klient toto potvrzení odeslat nemůže a tedy bude deregistrován. Obrázek, popisující tuto situaci



Obrázek 4.4: Consul test komunikace s mikroslužbami



Obrázek 4.5: Consul tabulka registrace mikroslužeb



Obrázek 4.6: Consul 2 aktivní služby na stejném portu

4.4 Testování vyvažování zátěže

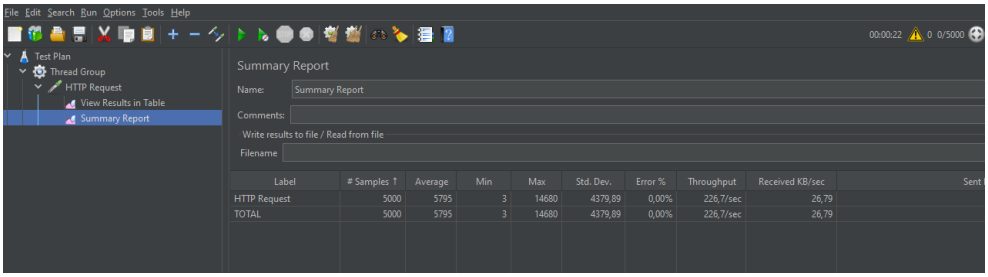
Pro testování vyvažování zátěže použijeme nástroj Apache JMeter. Má pro tyto účely dobrou funkcionalitu. Je určený pro simulaci zátěže na serveru pomocí odesílání velkého počtu HTTP požadavků. Účelem tohoto testu je prozkoumat chování SDS v průběhu velké zátěže serveru. Během testování vyvažování zátěže se zjistila nemožnost testování pro metody komunikující s

veřejnými API kvůli blokování velkého počtu požadavků.

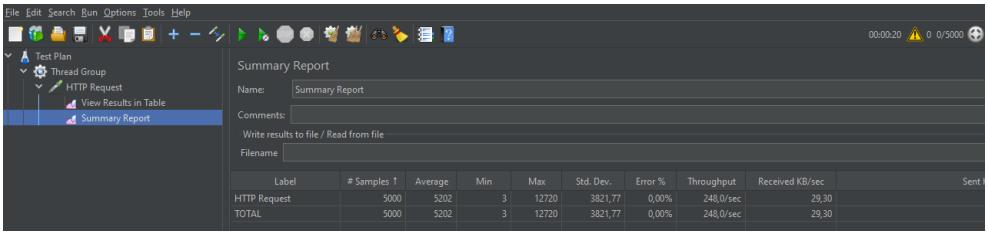
S ohledem na tuto záležitost `test` byla v modulu vytvořena metoda s endpointem /test/load-balancing jmenovaná `testLoadBalancing()` a v mikroslužbě `ares` byla vytvořená metoda se stejným názvem a endpointem /ares/load-balancing.

4.4.1 Eureka

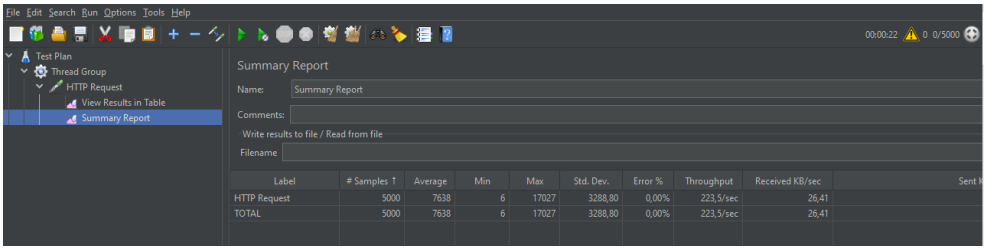
Pro testování vyvažování zátěže byly spuštěné 2 instance mikroslužby `ares` a byly použité následující metody: RoundRobinRule, WeightedResponseTimeRule, AvailabilityFilteringRule. Počet požadavků byl nastavený na 5000 a čas, během kterého tento počet bude odeslaný(*Ramp-up*) byl nastaven na 1 vteřinu.



Obrázek 4.7: Eureka RoundRobinRule



Obrázek 4.8: Eureka WeightedResponseTimeRule

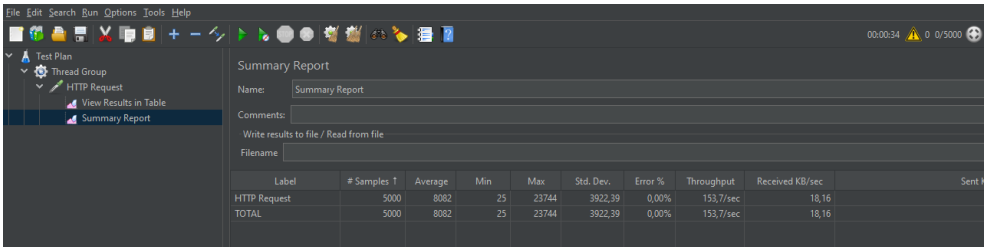


Obrázek 4.9: Eureka AvailabilityFilteringRule

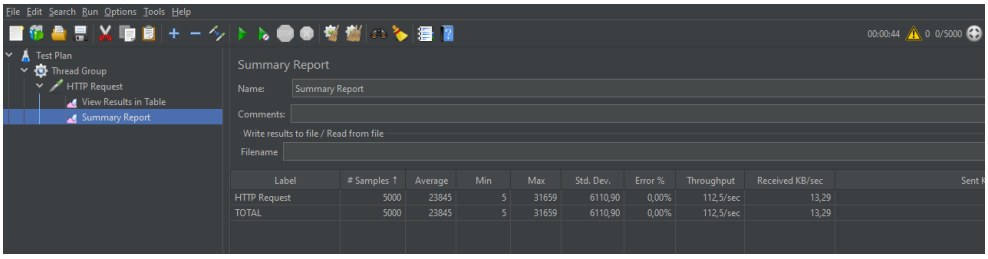
4.4.2 Consul

Pro testování vyvažování zátěže byly spuštěné 2 instance mikroslužby `ares` a byly použity následující metody: RoundRobinRule, WeightedResponseTimeRule, AvailabilityFilteringRule. Počet požadavků byl nastavený na 5000 a čas, během kterého tento počet bude odeslaný(*Ramp-up*) byl nastaven na 1 vteřinu.

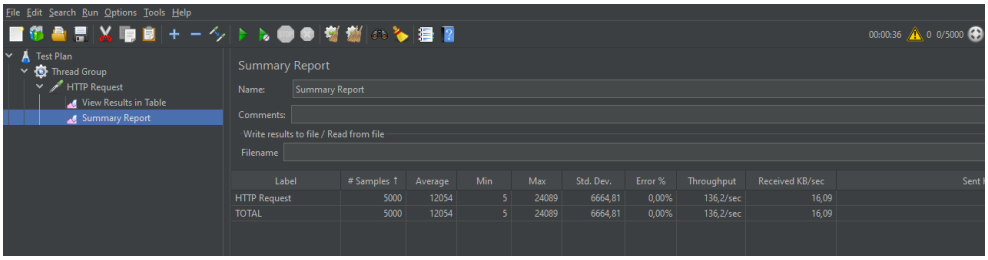
V průběhu testování vyvažování zátěže se na tomto SDS objevila chyba, související s předešlými vadami, popsanými v sekci 4.3.2. Server občas vracel chybu `404: Not Found`. Během debugování se zjistilo, že vyvažovač zátěže do pole dostupných IP adres pro danou doménovou adresu přiřazoval chybnou IP. Následně se zjistilo, že to se stává kvůli tomu, že zastaveným službám se nepodařilo deregistrovat a nově spouštěné měly stejný port jako ty zastavené.



Obrázek 4.10: Consul RoundRobinRule



Obrázek 4.11: Consul WeightedResponseTimeRule



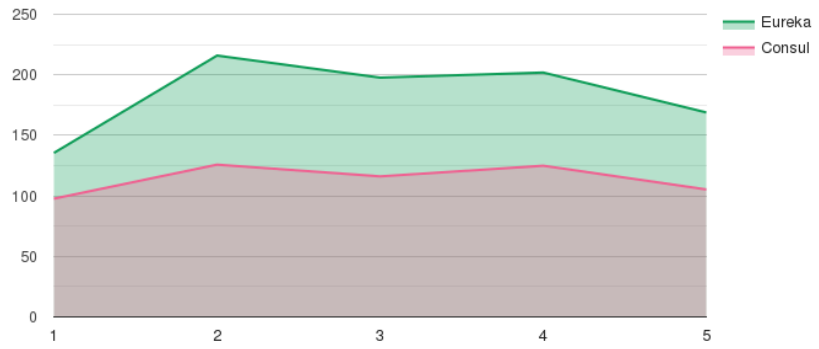
Obrázek 4.12: Consul AvailabilityFilteringRule

4.4.3 Rychlost zpracování požadavků

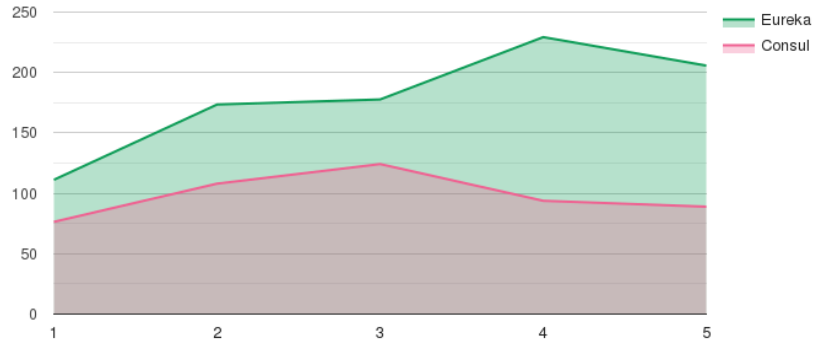
Větší rychlost zpracování požadavků umožní zpracovat větší počet požadavků za jednotku času. Pro testování této hodnoty byl použitý nástroj JMeter díky kterému můžeme sledovat střední hodnotu počtu zpracovaných HTTP požadavků za vteřinu času. Pro každé pravidlo vyvažování zátěže zvoleno 5 experimentů s počtem HTTP požadavků – 5000. Výsledky ve formě grafu jsou uvedené na následující stránce.

4.4.4 Shrnutí

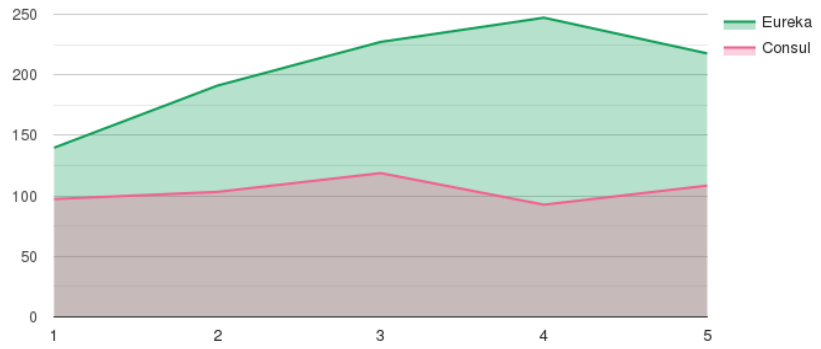
Po testování Service Discovery Serveru je vidět, že ačkoli Consul a Eureka mají podobnou funkcionalitu, Eureka je vhodnější pro použití, a to díky své jednoduché konfigurace a spolehlivosti během service discovery. Během testování vyvažování zátěže se navíc zjistilo, že má vyšší rychlost zpracování požadavků.



Obrázek 4.13: Porovnání výkonnosti Service Discovery Serveru během Round-Robin vyvažování zátěže (číslo experimentu/počet požadavků za vteřinu)



Obrázek 4.14: Porovnání výkonnosti Service Discovery Serveru během Weighted Response Time vyvažování zátěže (číslo experimentu/počet požadavků za vteřinu)



Obrázek 4.15: Porovnání výkonnosti Service Discovery Serveru během Availability Filtering vyvažování zátěže (číslo experimentu/počet požadavků za vteřinu)

Závěr

Cílem této bakalářské práce bylo navrhnout vhodné řešení Service Discovery Serveru pro použití vyvažování zátěže v projektu „Manažerský Informační Systém“. Dílčím cílem také bylo implementovat mikroslužby, které se následně budou používat v MIS pro vyvažování zátěže.

V průběhu analýzy byly prozkoumané základní architektury vývoje aplikace, obzvlášť byl kladen důraz na architektuře založené na mikroslužbach. Následně byly vysvětleny webové služby, pomocí kterých mohou mezi sebou komunikovat jednotlivé mikroslužby. Dále byly prozkoumaná jednotlivá řešení service discovery a service registry potřebné pro vhodné škálování aplikace a konec analýzy byl věnován vyvažování zátěže.

Další část práce byla věnována návrhu budoucí aplikace, popisu jednotlivých technologií, které se následně budou používat a volbě SDS pro implementaci a testování.

Během implementace mikroslužeb se objevilo několik závad ze strany veřejných API, které následně byly úspěšně vyřešené.

Následujícím krokem při vypracování bakalářské práce byla implementace a testování Service Discovery Serveru. V práci jsou detailně rozepsané vlastnosti každého z nich, provedeno integrační testování a testování vyvažování zátěže.

Na základě získaných znalostí pro projekt MIS bylo zvoleno nejvhodnější řešení Service Discovery Serveru, což bylo cílem této bakalářské práce.

V budoucnu je plánováno rozšířit implementaci jednotlivých mikroslužeb a spustit je v Docker kontejnerech pro následné testování Service Discovery.

Příloha A

Seznam použitých zkratek

MIS Manažerský Informační Systém

SDS Service Discovery Server

API Application Programming Interface

HTTP Hyper Text Transfer Protocol

WSDL Web Services Description Language

XML eXtensible Markup Language

JSON JavaScript Object Notation

REST Representational State Transfer

MAC Media Access Control

IP Internet Protocol

ARP Address Resolution Protocol

DNS Domain Name System

NAT Network Address Translation

TCP Transmission Control Protocol

VRRP Virtual Router Redundancy Protocol

ARES Administrativní registr ekonomických subjektů

ČSOB Československá obchodní banka

KB Komerční banka

RÚIAN Registr územní identifikace, adres a nemovitostí

SDS Service Discovery Server

UI User Interface

Příloha B

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `BP_Kuznietsov_Kostiantyn_2021.pdf` .... text práce ve formátu PDF

Bibliografie

1. HGRACA. Monolithic Architecture. *hgraca* [online]. 2017 [cit. 2021-04-17]. Dostupné z: https://herbertograca.com/2017/07/31/monolithic-architecture/.
2. DRESLER, Robert. Vícevrstvé architektury aplikací. *Robert Dresler* [online]. 2011 [cit. 2021-04-17]. Dostupné z: http://www.robertdresler.cz/2011/04/vicevrstve-architektury-aplikaci.html.
3. RICHARDSON, Chris. Pattern: Decompose by business capability context. *Microservice Architecture* [online]. 2020 [cit. 2021-04-17]. Dostupné z: https://microservices.io/patterns/decomposition/decompose-by-business-capability.html.
4. RICHARDSON, Chris. Introduction to Microservices [online]. 2015 [cit. 2021-04-18]. Dostupné z: https://www.nginx.com/blog/introduction-to-microservices/.
5. Webová služba. *Wikipedia* [online]. 2020 [cit. 2021-04-20]. Dostupné z: https://cs.wikipedia.org/wiki/Webov%C3%A1_slu%C5%BEba.
6. Penetrační testy API / web services. *DCIT* [online]. 2020 [cit. 2021-04-25]. Dostupné z: https://www.dcit.cz/cs/bezpecnost/penetracni-testy-API-webservices.
7. ADEDEJI, Clement. WEBSERVICES; SOAP AND REST - A SIMPLE INTRODUCTION. *Reply* [online]. 2020 [cit. 2021-04-22]. Dostupné z: https://www.reply.com/solidsoft-reply/en/content/webservices-soap-and-rest-a-simple-introduction.
8. SOAP. *Wikipedia* [online]. 2020 [cit. 2021-04-22]. Dostupné z: https://cs.wikipedia.org/wiki/SOAP.
9. Representational State Transfer. *Wikipedia* [online]. 2020 [cit. 2021-04-23]. Dostupné z: https://cs.wikipedia.org/wiki/Representational_State_Transfer.

10. HORDĚJČUK, Vojtěch. REST. *VOHO* [online]. 2020 [cit. 2021-04-23]. Dostupné z: http://voho.eu/wiki/rest/.

11. What are cloud applications? *Red Hat* [online]. 2015 [cit. 2021-04-19]. Dostupné z: https://www.redhat.com/en/topics/cloud-native-apps/what-are-cloud-applications.

12. Protokol HTTP. *Zdroják.cz* [online]. 2011 [cit. 2021-04-20]. Dostupné z: https://zdrojak.cz/clanky/protokol-http/.

13. Proxy server. *Wikipedia* [online]. 2020 [cit. 2021-04-20]. Dostupné z: https://cs.wikipedia.org/wiki/Proxy_server.

14. RICHARDSON, Chris. Introduction to Microservices. *NGINX* [online]. 2015 [cit. 2021-04-19]. Dostupné z: https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/.

15. Referenční model ISO/OSI. *Wikipedia* [online]. 2020 [cit. 2021-04-25]. Dostupné z: https://cs.wikipedia.org/wiki/Referen%C4%8Dn%C3%AD_model_ISO/OSI.

16. FESL, Jan. Transportní vrstva, protokoly TCP a UDP. *České vysoké učení technické v Praze, Fakulta informačních technologií* [online]. 2021 [cit. 2021-04-30]. Dostupné z: https://courses.fit.cvut.cz/BI-PSI/media/lectures.

17. VONDRA, Tomáš. IaaS - pokročilé služby, load balancing[přednáška]. *České vysoké učení technické v Praze, Fakulta informačních technologií* [online]. 2021 [cit. 2021-04-26]. Dostupné z: https://courses.fit.cvut.cz/NI-VCC/lectures/index.html.

18. MORIC LUBOMÍR, Vilč Jaroslav. VRRP. 2006. Dostupné také z: http://www.cs.vsb.cz/grygarek/SPS/projekty0506/vrrp. Accessed: 27-4-2021.

19. How VRRP Works. *Habr* [online]. 2021 [cit. 2021-05-03]. Dostupné z: https://habr.com/ru/post/452490/.

20. What Is Layer 4 Load Balancing? *NGINX* [online]. 2020 [cit. 2021-04-26]. Dostupné z: https://www.nginx.com/resources/glossary/layer-4-load-balancing/.

21. PURI, Mohak. L4 vs L7 Load Balancing. *gitconnected* [online]. 2020 [cit. 2021-04-26]. Dostupné z: https://levelup.gitconnected.com/l4-vs-l7-load-balancing-d2012e271f56.

22. What Is DNS Load Balancing? *NGINX* [online]. 2020 [cit. 2021-04-28]. Dostupné z: https://www.nginx.com/resources/glossary/dns-load-balancing/.

23. Zookeeper 3.7 Documentation. *ZooKeeper* [online]. 2021 [cit. 2021-05-01]. Dostupné z: https://zookeeper.apache.org/doc/r3.7.0/zookeeperOver.html.

24. *etcd-viewer* [online]. 2021 [cit. 2021-05-01]. Dostupné z: https://github.com/matishsiao/etcd-console.

25. Ribbon. *GitHub repository*. 2021. Dostupné také z: https://github.com/Netflix/ribbon/blob/master/README.md.

26. GitHub Issue. *GitHub repository*. 2021. Dostupné také z: https://github.com/komercka/adaa-example-spring-boot/issues/2.

27. *Service Discovery: Eureka Server*. 2021. Dostupné také z: https://cloud.spring.io/spring-cloud-netflix/multi/multi_spring-cloud-eureka-server.html.

28. *Consul*. 2021. Dostupné také z: https://www.consul.io/.

29. *Consul Force Leave*. 2021. Dostupné také z: https://www.consul.io/commands/force-leave.