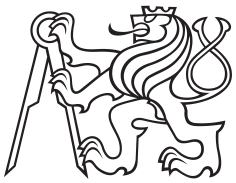


České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

Microservices - design patterns

David Sonntag

Školitel: Ing. Jiří Šebek
Květen 2023

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Sonntag** Jméno: **David** Osobní číslo: **503213**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Microservices - design patterns

Název bakalářské práce anglicky:

Microservices - design patterns

Pokyny pro vypracování:

Mikroslužby jsou efektivní způsob implementace komplexních aplikací a momentálně je to velmi moderní a žádané téma. Implementace takové služby je však poměrně náročná a její nesprávné provedení může být kontraproduktivní.
Cíle práce:
1) Seznamte se s moderními design patterny týkajících se microserviců a jejich následné využití v praxy.
2) Proveďte analýzu jednotlivých design patternů a popište jejich implementaci v rámci různých procesů
3) Proveďte jak datovou analýzu tak analýzu služeb endpointů (rozdělení backendových operací na GET, CREATE, atd.)
4) Vytvořte demo aplikaci, která bude implementovat jak design patterny tak technologie pro microservice architekturu
5) Vytvořte testovací scénáře, které porovnájí microservice architekturu s monolitickou architekturou
6) Zhodnoťte výsledky provedených testů

Seznam doporučené literatury:

De Lauretis, Lorenzo. "From monolithic architecture to microservices architecture." 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). IEEE, 2019.
ALSHUQAYRAN, Nuha; ALI, Nour; EVANS, Roger. A systematic mapping study in microservice architecture. In: 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA). IEEE, 2016. p. 44-51.
DMITRY, Namiot; MANFRED, Sneps-Sneppe. On micro-services architecture. International Journal of Open Information Technologies, 2014, 2.9: 24-27.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Jiří Šebek kabinet výuky informatiky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **17.02.2023** Termín odevzdání bakalářské práce: **26.05.2023**

Platnost zadání bakalářské práce: **22.09.2024**

Ing. Jiří Šebek podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. Datum převzetí zadání

Podpis studenta

Poděkování

Tímto bych rád poděkoval vedoucímu mé bakalářské práce panu Ing. Jiřímu Šebkovi za jeho trpělivost, cenné rady a pravidelné konzultace během zpracovávání této bakalářské práce.

Prohlášení

Tímto prohlašuji, že jsem tuto práci vypracoval samostatně. Veškeré zdroje informací, které jsem využil k této práci, jsou uvedeny v seznamu použité literatury.

V Praze, 16. května, 2023

Abstrakt

Cílem této práce je vytvořit a porovnat dvě backendové aplikace na základě jejich architektury. První aplikace je navržena jako monolitická , zatímco druhá jako mikroservisní.

Nejprve jsou analyzovány různé design patterny využívané při vývoji mikroservisních aplikací. Následuje návrh obou aplikací a jejich implementace na základě tohoto návrhu. Pro porovnání výkonu obou aplikací jsou provedeny zátěžové testy pomocí nástroje Locust.

Klíčová slova: backendová aplikace, design patterny, monolitická architektura, mikroservisní architektura, zátěžové testy, Locust.

Školitel: Ing. Jiří Šebek

Abstract

The goal of this thesis is to create and compare two backend applications based on their architecture. The first application is designed as monolithic and the second one follows a microservice approach.

Initially, various design patterns commonly used in the development of microservice architecture are analyzed. Subsequently, the design of both applications is proposed, followed by implementation according to the respective design. To compare the performance of both applications, load testing is conducted using the Locust tool.

Keywords: backend application, design patterny, monolithic architecture, microservice architecture, load tests, Locust

Obsah

Obrázky

Tabulky

Kapitola 1
Úvod

Tématem této práce je vývoj mikroservisní architektury a různé přístupy k tomuto úkolu. V posledních letech se mikroservisní architektura stává stále populárnější díky své flexibilitě, škálovatelnosti a odolnosti vůči chybám. Její kvalitní vytvoření vyžaduje kombinaci různých design patternů a technologií.

Práce začíná popisem jednotlivých design patternů, které se používají při vývoji mikroservisních architektur. Tyto patterny slouží k zajištění větší odolnosti systému, konzistenci dat a usnadnění práce vývojářům.

Poté následuje implementační část, ve které je nejprve navržen model systému pro testování výkonu a výhod mikroservisní architektury ve srovnání s monolitickou architekturou. Dále je provedena analýza služeb endpointů, která rozděluje operace na get, create a další. Požadavky jsou dále rozděleny na synchronní a plně asynchronní. Na základě modelu systému jsou vytvořeny dvě demo aplikace, které mají stejnou funkčnost, ale jedna je navržena jako monolitní a druhá jako mikrosevisní architektura.

Nakonec je provedeno testování implementace pomocí zátěžových testů implementovaných s využitím knihovny Locust. Výsledkem této práce je porovnání výhod a nevýhod mikroservisní architektury v porovnání s monolitickou architekturou a získání užitečných poznatků pro navrhování a implementaci mikroservisních aplikací v budoucnosti.

Kapitola 2
Rešerše

První část této kapitoly se věnuje různým design patternům vhodným k tvorbě mikroservní architektury. Druhá část je zaměřena technologie používané k asynchronnímu volání backendu.

2.1 Design patterny

V této kapitole se zaměříme na různé design patterny, které jsou využívány při tvorbě mikroservisních architektur. Každý pattern bude podrobně popsán včetně jeho významu, výhod, nevýhod a vhodných použití.

2.1.1 Database per service pattern

Tento pattern je dnes spíše takovou zásadou vytváření mikroservisních architektur. Tento pattern říká, že každá mikroslužba má mít svou vlastní databázi, což znamená, že změny v jedné databázi neovlivní ostatní mikroslužby. Toto výrazně snižuje riziko přenesení chyby napříč celým systémem. Další výhodou je, že každá mikroslužba si může zvolit vlastní druh databáze, např. relační, NoSQL atd. Tento přístup ale přináší problém koordinace transakcí, aby se udržela konzistence napříč databázemi mezi jednotlivými službami. Tyto problémy ovšem mají na starosti další design patterny. [1]

2.1.2 SAGA

Saga pattern je způsob, jak udržet konzistenci dat mezi mikroslužbami. Jedná se o posloupnost transakcí, které aktualizují službu a pomocí zpráv nebo událostí, které je spouštějí, se vyvolává další krok v sekvenci. Pokud jeden z kroků selže, saga provede kompenzační transakce, aby vrátila všechny předchozí kroky do původního stavu. [13]

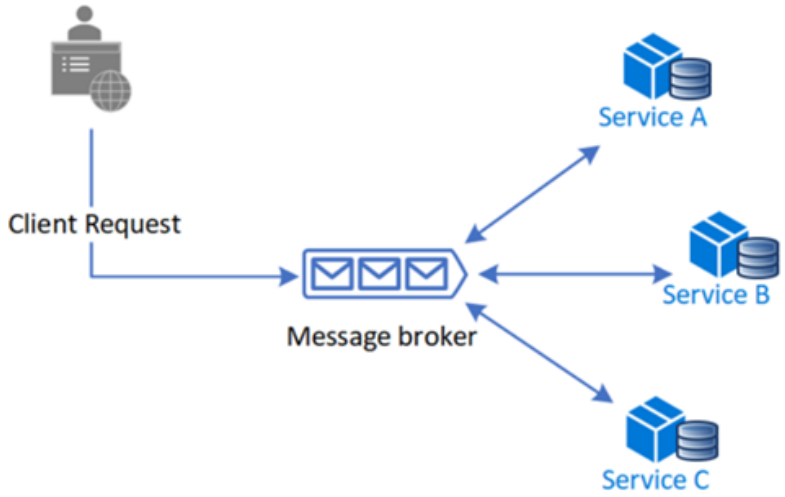
Transakce musí splňovat pravidla ACID: A - Atomicity (Nedělitelnost) znamená, že operace musí být nedělitelné a neredukovatelné, musí se provést buď všechny nebo žádná. C - Consistency (Konvergence) znamená, že transakce převádí data pouze z platného stavu do dalšího platného stavu. I - Isolation (Izolace) zaručuje, že souběžné transakce budou mít stejný výsledek, jako by byly provedeny postupně. D - Durability (Trvanlivost) zajišťuje, že

potvrzené transakce zůstanou potvrzené i v případě selhání systému nebo výpadku napájení. Dále platí, že každou transakci lze zvrátit pomocí transakce s opačným efektem. Dvě nejčastěji používané implementace jsou choreografie a orchestrace. [2]

Choreography

Tento způsob implementace, který zajišťuje konzistenci napříč databázemi, spočívá v koordinaci správného sledu událostí v sadě bez centrální kontroly (viz Obrázek 2.1). Každá transakce odesílá zprávu, která spouští transakci v ostatních mikroslužbách. Tato implementace je vhodná zejména pro jednoduché procesy s malým počtem účastníků a nenáročnou koordinační logikou. Jednou z výhod tohoto přístupu je absence potřeby další mikroslužby pro koordinaci transakcí, čímž se eliminuje riziko tzv. "single point of failure", protože odpovědnost za transakce je rozložena mezi jednotlivé mikroslužby. [13]

Nevýhody této implementace zahrnují obtížnější integrační testování, protože všechny mikroslužby musí být spuštěny pro simulaci transakce. Při vyšším počtu procesů může být obtížné sledovat, která transakce volá kterou. Existuje také riziko zacyklení, kdy jedna nebo více služeb vzájemně naslouchají na své zprávy. Mezi frameworky implementující choreografii patří například Zeebe nebo Axon Framework. [2]

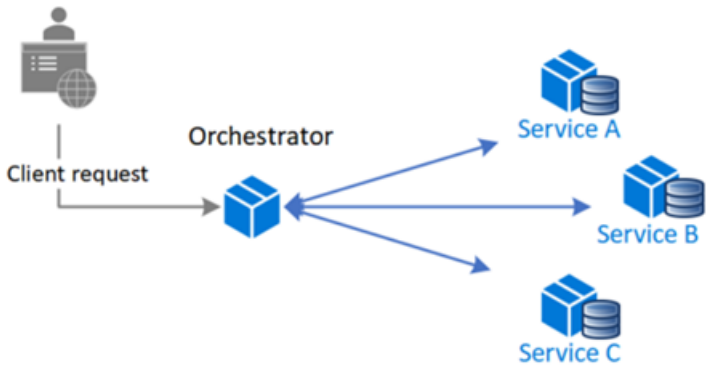


Obrázek 2.1: Choreography pattern

Orchestration

Tato implementace je založena na tzv. orchestrátoru, který funguje jako hlavní řídící jednotka procesu (viz Obrázek 2.2) a rozhoduje, jaké transakce mají být provedeny. Orchestrátor je zodpovědný za spuštění všech lokálních transakcí

v mikroslužbách na základě událostí, které nastanou. Dále se stará o ukládání stavu jednotlivých úkolů a zvládá pracovat s chybami v jednotlivých mikroslužbách (například spuštění kompenzačních transakcí). Tato implementace je vhodná pro komplexní systémy s velkým počtem mikroslužeb nebo pro systémy, které budou pravděpodobně v budoucnu rozšiřovány o další služby. Velkou výhodou je, že se zde nevyskytuje riziko zacyklení, protože veškerá koordinace probíhá na jednom místě. Navíc jednotliví účastníci nemusí mít žádné informace o ostatních, což podporuje designový princip "separation of concerns". Nicméně implementace takového systému je složitější a přináší riziko tzv. "single point of failure", kdy selhání orchestrátoru znamená selhání celého systému. [2] Mezi frameworky pro orchestraci patří například Camunda BPM nebo Apache ServiceComb. [13]



Obrázek 2.2: Orchestration pattern

2.1.3 Circuit breaker

Tento vzor slouží k ochraně systému před neustálým voláním nefunkčních služeb. Jeho častým využitím je ochrana mikroslužební architektury před přetížením jednotlivých služeb.

Circuit Breaker (přerušovač obvodu) sleduje stav volání služby a po dosažení určitého počtu neúspěšných volání v krátkém časovém intervalu přepne do stavu "open"(otevřený). V tomto stavu je volání služby blokováno a všechna další volání jsou okamžitě přerušena s chybovou zprávou. Po určité době se Circuit Breaker přepne do stavu "half-open"(polootevřený), kdy se opět zkouší volat službu. Pokud volání opět selže, Circuit Breaker se vrátí do stavu "open", jinak přejde do stavu "closed"(uzavřený) a volání služby je opět povoleno.

Tento vzor je užitečným nástrojem pro ochranu systému před nekonečným voláním nefunkčních služeb a snižuje zatížení systému způsobené neúspěšnými voláními. Je však důležité si uvědomit, že Circuit Breaker vzor sám o sobě neopravuje nefunkční službu, ale pouze ji detekuje a omezuje její vliv na ostatní části systému. Implementace Circuit Breaker vzoru lze nalézt v různých knihovnách, například v Hystrix nebo Resilience4j. [3]

2.1.4 Bulkhead pattern

Samotný Bulkhead pattern je software design pattern, který pomáhá zlepšit odolnost vůči chybám a stabilitu systému tím, že izoluje jednotlivé části systému. Jeho hlavním cílem je zabránit šíření chyb z jedné části systému na celý systém.

Tento pattern rozděluje celý systém do logických komor (bulkheads), které fungují nezávisle na sobě. Každá komora má své vlastní limity zdrojů, jako je paměť nebo CPU. Pokud jedna komora selže nebo dosáhne svých limitů, ostatní komory zůstávají nedotčené a systém může pokračovat v práci.

Bulkhead pattern má několik výhod. Jednou z hlavních je izolace jednotlivých částí systému. To znamená, že selhání jedné komory nemá dopad na ostatní části systému, což minimalizuje riziko pádu celého systému. Tímto způsobem lze dosáhnout vyšší stability a spolehlivosti systému. Další výhodou je lepší plánování a řízení zdrojů, protože každá komora má svůj vlastní limit zdrojů, což umožňuje efektivnější využití systémových prostředků.

Existuje několik knihoven a frameworků, které implementují Bulkhead pattern, jako například Hystrix nebo Istio. Tyto nástroje poskytují funkcionalitu pro izolaci a správu jednotlivých komor v systému.

Celkově lze Bulkhead pattern považovat za užitečný nástroj pro zlepšení odolnosti a stability systému, který minimalizuje šíření chyb a umožňuje efektivní využití zdrojů.[4]

2.1.5 Agregator

Agregační pattern kombinuje výsledky z různých služeb do jednoho, čímž zjednodušuje práci s velkým množstvím mikroslužeb.

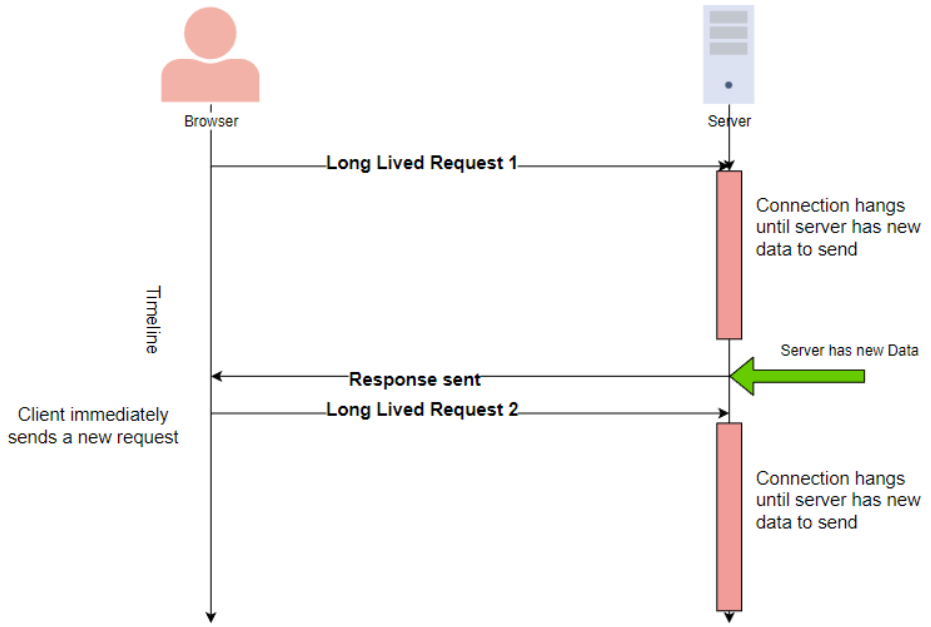
Agregátor posílá požadavky na data do několika různých služeb, následně tato data zkombinuje do jednoho výsledku a ten pak odesílá klientovi. Tento pattern výrazně zjednodušuje architekturu systému, protože všechna data jsou dostupná přímo z jedné služby. Dále umožňuje snadno přidávat nebo odebírat další služby. Implementaci tohoto patternu lze nalézt například v knihovnách Spring Cloud nebo Netflix Zuul. [5]

2.2 Technologie k asynchronnímu volání backendu.

Nejčastější metodou asynchronní komunikace mezi frontendem a backendem je využití výchozí odpovědi frontendu a následné asynchronní odeslání požadavku na backend. Například při úspěšné registraci uživatele mu pouze zobrazíme zprávu, že mu zasíláme potvrzovací e-mail, zatímco backend stále zpracovává jeho požadavek. Další možností je použití tzv. Promise (asynchronní volání na backend s přidruženým event listenerem pro odpověď ze serveru). Tímto způsobem můžeme požadavek rozdělit na jednotlivé části a postupně je doručovat uživateli v závislosti na jejich načítání, místo aby musel čekat na kompletní odpověď najednou. [15]

2.2.1 Long Polling

Long polling je technika jak pushnout informace ze serveru na klienta tak rychle, jak je to jenom možné. Funguje na principu neustále otevřeného spojení mezi klientem a serverem. Klient pošle HTTP request na server, který si request drží do doby, než je zpracován. Poté zašle klientovi aktualizovaná data. Klient nato pošle hned další request a tím se spojení drží naživu a smiuluje se real-time aplikace(viz Obrázek 2.3).[9]



Obrázek 2.3: Long polling

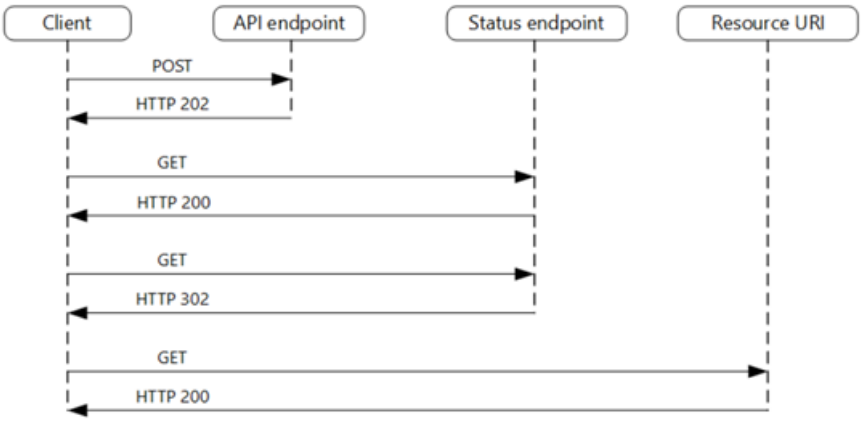
2.2.2 HTTP Polling

Klient využívá synchronní metodu a zasílá požadavek na API. API okamžitě odpovídá s co nejrychlejší odpovědí, validuje požadavek a vrátí odpověď

klientovi. Odpověď může být ve formě kódu HTTP 202 (Accepted), což značí, že požadavek byl přijat ke zpracování, nebo kódu HTTP 400 (Bad request), pokud validace selhala.

Pokud byl požadavek úspěšně zvalidován, API odpovídá s odkazem na stavový endpoint, kde si klient může ověřit stav svého požadavku. Zpracování požadavku na backendu probíhá asynchronně. API rychle poskytuje odpověď a deleguje zpracování požadavku na jiné části systému, například na frontu se zprávami (message queue).

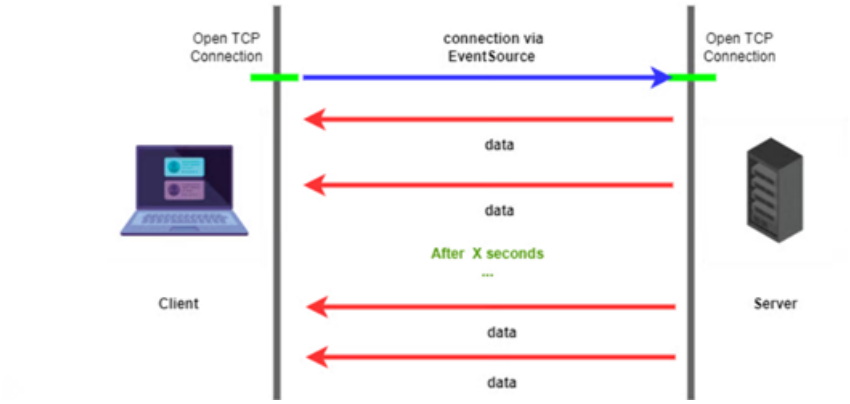
Pokud klient volá stavový endpoint, API odpovídá s kódem HTTP 200. Pokud se požadavek stále zpracovává, endpoint vrátí zprávu, že požadavek ještě není dokončen. Po dokončení procesu endpoint informuje o dokončení požadavku, nebo přesměruje klienta na jinou URL. Například pokud asynchronní operace vytvoří nový zdroj, endpoint přesměruje uživatele na URL tohoto zdroje (viz Obrázek 2.4).



Obrázek 2.4: httpPolling

2.2.3 Server-sent events

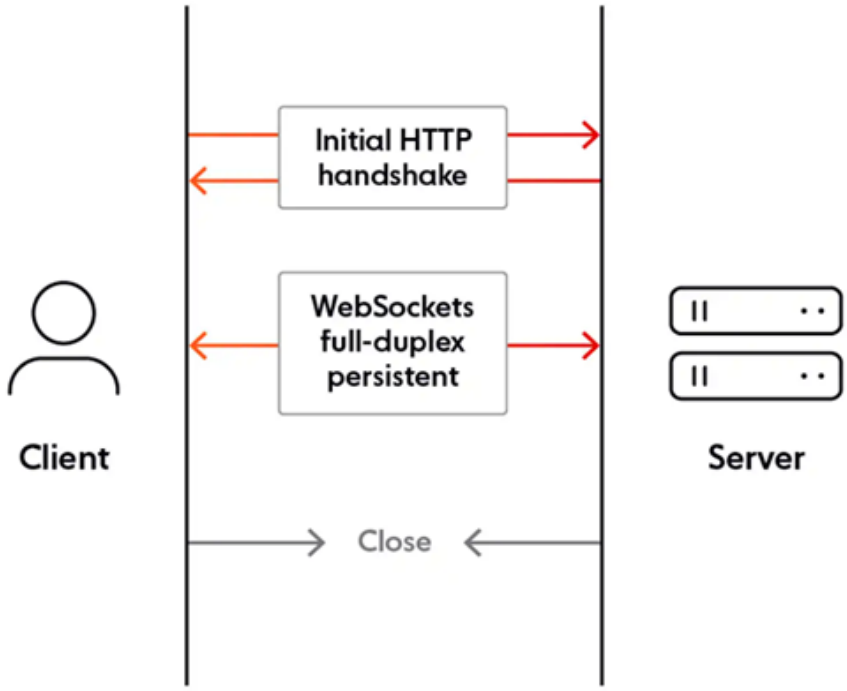
Technologie, která umožňuje klientovi automaticky přijímat data ze serveru přes HTTP spojení, se nazývá Server-sent events (SSE). Po navázání spojení mezi klientem a serverem může server aktivně odesílat aktualizace klientovi (viz Obrázek 2.5). Toto spojení je realizováno pomocí JavaScript API EventSource. SSE bylo navrženo jako efektivnější alternativa k long pollingu a zároveň poskytuje automatické obnovení spojení v případě, že klient ztratí spojení se serverem. Každý event může být identifikován pomocí přiřazeného ID a umožňuje posílat libovolné typy událostí. [7]



Obrázek 2.5: Server-sent events

2.2.4 WebSockets

WebSockets jsou vytvářeny pomocí handshake požadavku, který je iniciován uživatelem a směřuje na server. Poté se navazuje oboustranné WebSocket spojení (viz obrázek 2.6), které umožňuje přenos zpráv mezi serverem a klientem po celou dobu, kdy je spojení aktivní. Jak server, tak klient mohou přímo posílat zprávy druhé straně, dokud jedna z nich neukončí spojení. Komunikace probíhá pomocí WebSocket protokolu, na rozdíl například od Server-sent events (SSE), který je založen na HTTP protokolu. Tato technologie je široce využívána při vývoji real-time webových aplikací, jako je například burza s Bitcoinem, chatovací aplikace nebo online hry. [6]



Obrázek 2.6: Webosckets

2.2.5 HTTP2

Druhá verze HTTP, nazývaná HTTP/2, představuje evoluci protokolu HTTP a má za cíl zrychlit a zjednodušit proces komunikace mezi klientem a serverem. Jedním z hlavních důvodů pro vývoj HTTP/2 bylo odstranění neefektivit a omezení původní verze HTTP. Jednou z mnoha výhod HTTP/2 je podpora server-side push, která umožňuje serveru aktivně posílat zprávy klientovi bez jeho předchozího požadavku (je však třeba upozornit, že dnes již existuje protokol HTTP/3). Tím se snižuje latence a zrychluje načítání webových stránek.[19]

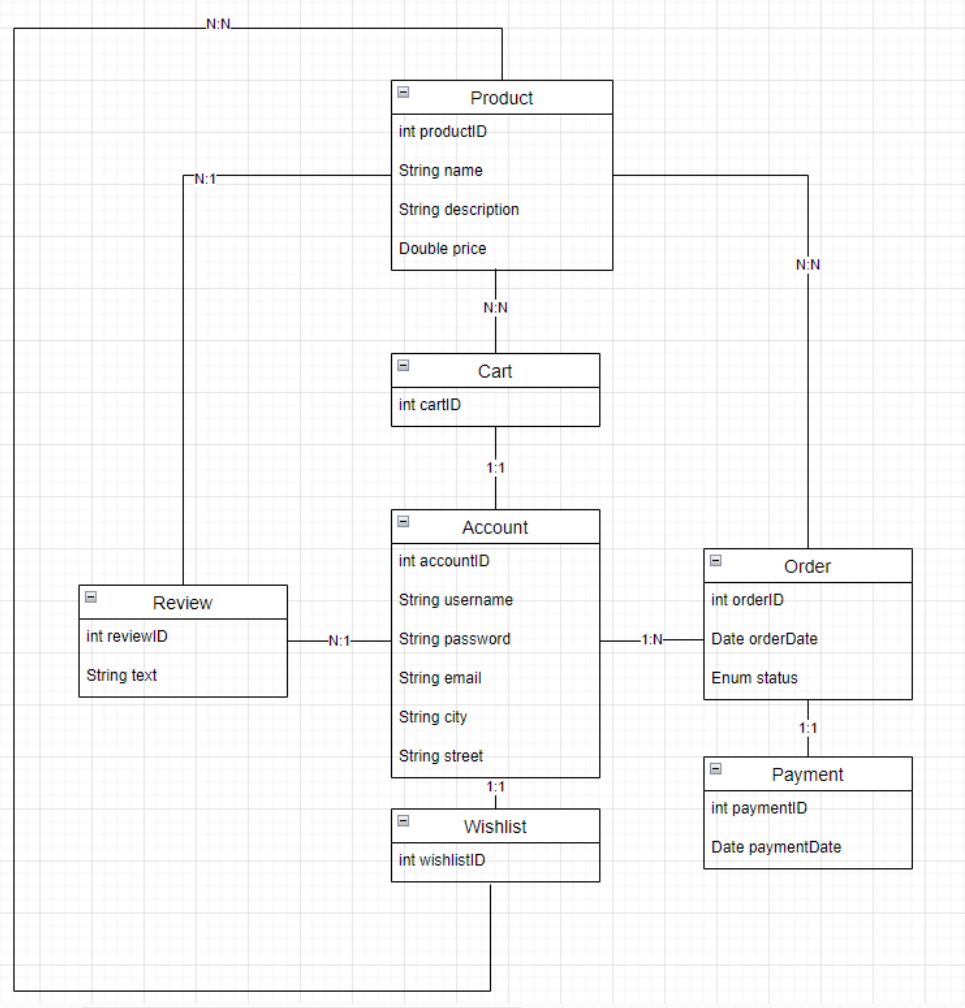
Kapitola 3

Návrh systému

Cílem tohoto projektu je vytvořit demo aplikaci, která co nejlépe simuluje reálný provoz. Projekt začal analýzou systému a následnou implementací a testováním pomocí nástroje Locust.

3.1 Datový model

Tento diagram vychází z obecné architektury e-shopu, jelikož je to jedna z nejčastějších architektur webových aplikací, jak je znázorněno v Obrázku 3.1. Hlavním přínosem tohoto návrhu v našem případě není vytvoření konkrétního modelu nebo databázového schématu, ale spíše predikce složitosti různých databázových požadavků na jednotlivé entity. Tento diagram slouží jako základ pro analýzu endpointů v našem systému.

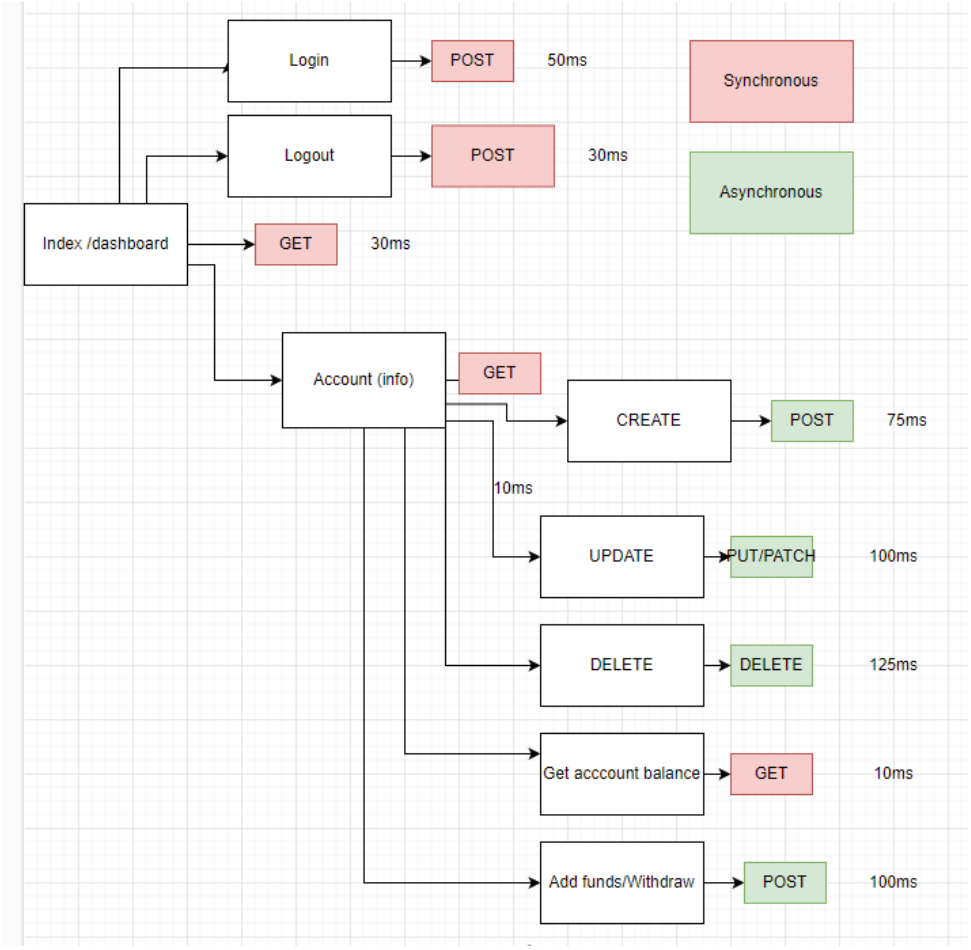


Obrázek 3.1: Datový model

3.2 Analýza endpointů

Na základě počtu atributů a relací mezi entitami byla odhadnuta předpokládaná doba trvání transakcí v databázi. Pro každý endpoint byla brána v potaz složitost operace, složitost dané entity a počet entit, se kterými transakce pracuje. V případě seznamu entit bylo předpokládáno, že se jedná o deset entit v seznamu. Operace CRUD (vytvoření, čtení, aktualizace, smazání) byly hodnoceny podle jejich složitosti, kde čtení < vytvoření < aktualizace < smazání. Dále byly operace rozděleny na synchronní a asynchronní. Synchronní operace byly označeny červeně, zatímco asynchronní operace byly označeny zeleně. U každého endpointu byla také uvedena předpokládaná doba trvání transakce. Na základě této analýzy byl nastaven timeout na repository vrstvě aplikace. Je však třeba poznamenat, že asynchronní operace v mikroslužbové architektuře nejsou omezeny touto dobou timeoutu, čímž simulujeme dobrý design mikroservisní aplikace. Na obrázku 3.2 je zobrazena analýza jednoho

ze 7 endpointů. Celá anlýza je k dispozici mezi přílohami (viz Obrázek 7.1 a Obrázek 7.2)



Obrázek 3.2: Analýza endpintů

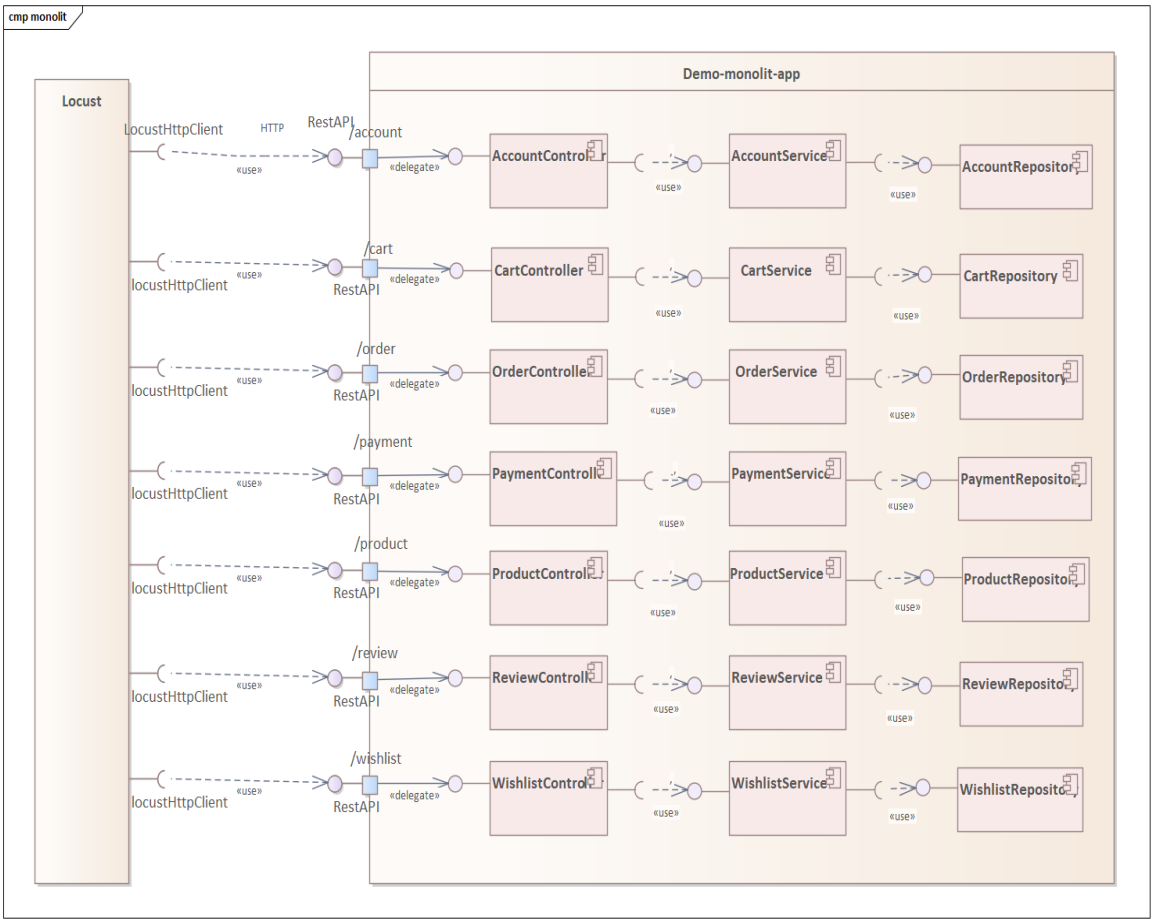
3.3 Diagram komponent

Diagram komponent je vizuální reprezentace struktury a interakcí mezi jednotlivými komponentami v systému. Hlavním cílem tohoto diagramu je identifikovat a vizualizovat jednotlivé komponenty systému a jejich vzájemné vztahy.

3.3.1 Monolitní architektura

V monolitní aplikaci je nejvyšší vrstvou kontroler, který přijímá HTTP požadavky pomocí REST API. Kontroler předává požadavky do servisní vrstvy, která se zabývá byznysovou logikou systému. Tato vrstva interaguje s repozitářovou vrstvou, která obvykle komunikuje přímo s databází (viz Obrázek 3.3). V našem případě však žádná databáze není přítomna a požadavek pouze čeká

na vypršení timeoutu, jehož délka je závislá na konkrétním typu požadavku a je popsána na obrázku 3.2.



Obrázek 3.3: Diagram komponent pro monolitní aplikaci

3.3.2 Mikroservisní architektura

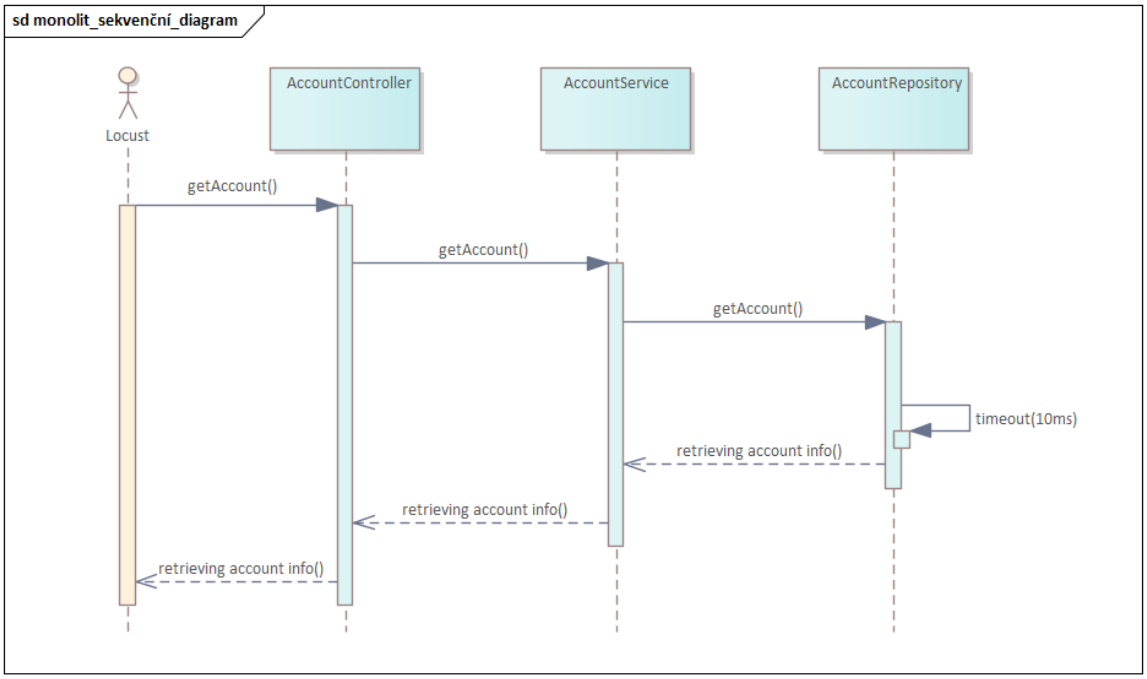
Nejvyšší vrstvou aplikace je kontroler API Gateway. Sem přichází požadavek, který je následně předán do servisní vrstvy. Poté je požadavek přeposlán na agregátor, který zajišťuje komunikaci mezi API Gateway a jednotlivými mikroslužbami. Komunikace mezi těmito komponentami probíhá pomocí RestTemplate, který odesílá synchronní HTTP požadavek na jednotlivé mikroslužby. Architektura jednotlivých mikroslužeb a jejich ekvivalentních komponent v monolitické architektuře je identická s výjimkou repositoru vrstvy. Ta se liší u požadavků, které jsou v analýze endpointů označeny jako asynchronní, jelikož u těchto požadavků je v mikroservisní architektuře timeout 0. Diagram se nachází mezi přílohami jako Obrázek 7.3.

3.4 Sekvenční diagramy

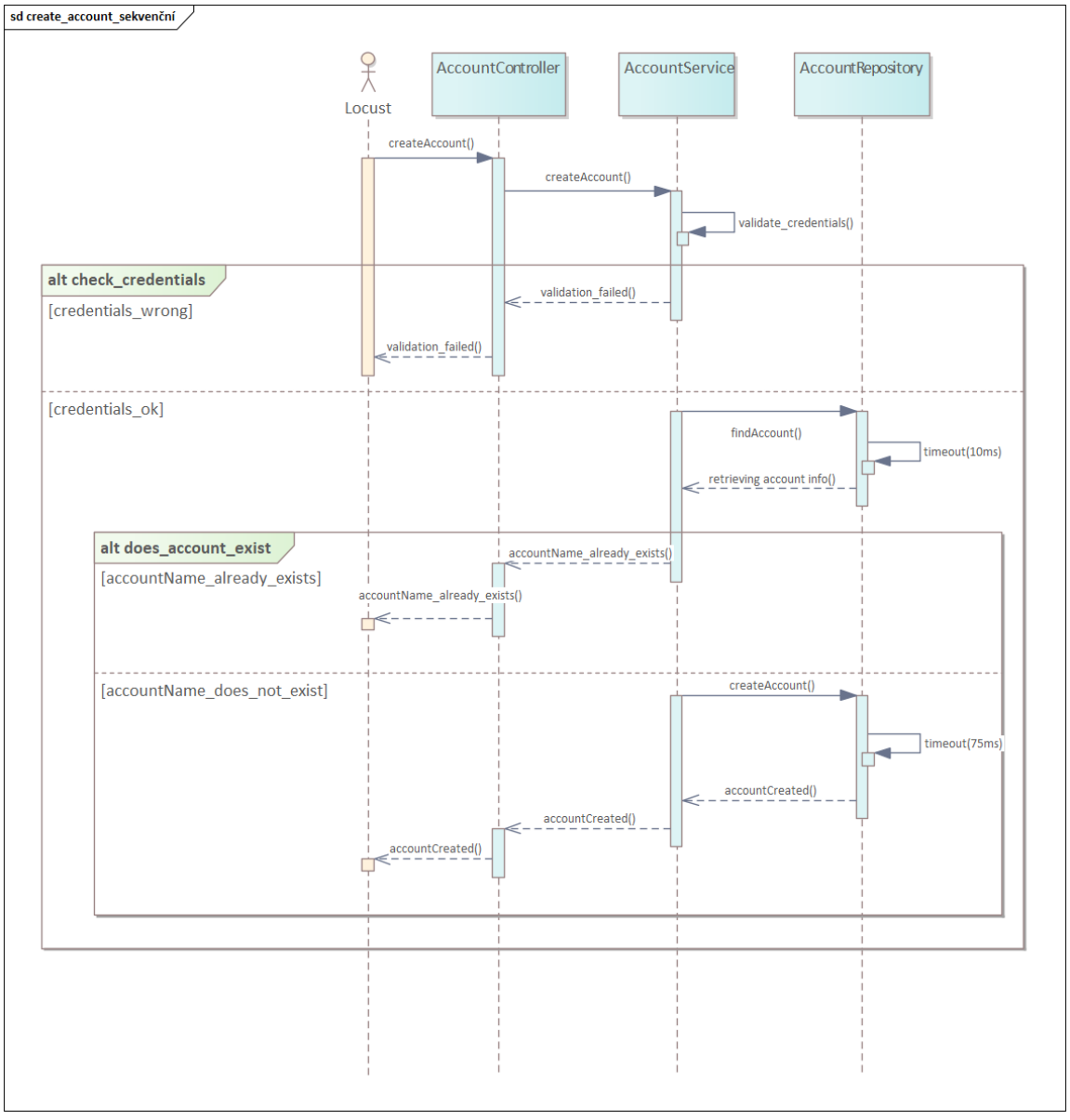
Sekvenční diagram je grafická reprezentace sledu interakcí mezi objekty v určitém pořadí. Hlavním cílem sekvenčního diagramu je znázornit, jak jednotlivé objekty spolupracují a komunikují mezi sebou při provádění konkrétních akcí nebo scénářů.

3.4.1 Monolitní architektura

Komunikace u všech endpointů probíhá obdobně jako na Obrázku 3.4, kde je zobrazen příklad volání endpointu na zjištění informací o uživatelském účtu. Požadavek prochází jednotlivými částmi systému, až se dostane do repository vrstvy, kde vyčká na timeout, a poté vrátí výsledek. Na obrázku 3.5 můžeme vidět složitější scénář vytvoření účtu. Prvně je validován uživatelský vstup, pokud je vše v pořádku, proběhne kontrola v databázi, jestli takový účet už neexistuje. Poté co v databázi není nalezen účet se stejným jménem, tak je vytvořen nový účet.



Obrázek 3.4: Sekvenční diagram getAccount pro monolit



Obrázek 3.5: Sekvenční diagram createAccount pro monolit

3.4.2 Mikroservisní architektura

Na obou scénářích (viz Obrázek 7.4 a Obrázek 7.5 v příloze) je patrné, že mikroslužby přinášejí zvýšenou složitost ve srovnání s monolitickou architekturou. V mikroslužbách je požadavek směrován přes API Gateway do příslušné mikroslužby a zpět. Tím se celý proces značně zpomaluje, zejména v případě, kdy dochází ke komunikaci mezi mikroslužbami pomocí RestTemplate.

V prvním případě, který se týká AccountMicroservice, je proces stále podobný tomu, který probíhá v monolitické architektuře s tím rozdílem, že přidáváme další vrstvu komunikace prostřednictvím API Gateway.

Ve druhém případě je situace obdobná ale s tím rozdílem, že při vytváření účtu je timeout v repository vrstvě nastaven na hodnotu 0, což umožňuje asynchronní provádění části procesu vytváření účtu (viz Obrázek 3.2).

Celkově lze říci, že mikroslužby přinášejí složitější architekturu a zvýšenou komplexitu komunikace mezi jednotlivými komponentami. To může mít vliv na celkový čas zpracování požadavku.

Kapitola 4

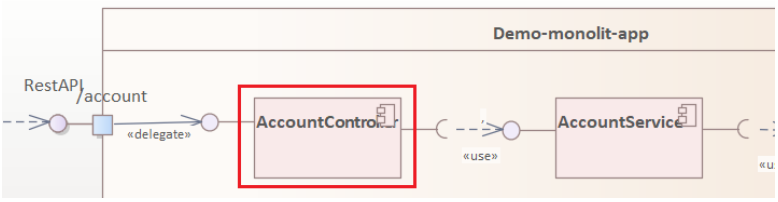
Implementace

V této kapitole se zaměříme na implementaci navrženého systému. Cílem této části je představit výslednou implementaci a poskytnout ucelený pohled na fungování našeho systému.

4.1 Kontrolní vrstva

Tato aplikační vrstva se nachází na vrcholu aplikace (viz Obrázek 4.1). Jejím hlavním úkolem je zajištění komunikace s okolím aplikace a poskytování funkcionalit prostřednictvím definovaných endpointů. V našem konkrétním případě jsme tuto vrstvu implementovali jako RestController, což nám umožňuje definovat a spravovat jednotlivé endpointy našeho systému. Tímto způsobem jsme schopni poskytovat a zpracovávat požadavky, které přicházejí z vnějších zdrojů a interagovat s dalšími komponentami naší aplikace. Jednotlivé požadavky jsou poté odesílány do servisní vrstvy, příklad, jak vypadá endpoint vytvoření účtu, je vidět na Obrázku 4.2.

Tato vrstva je identická jak v mikroservisní, tak monolitické aplikaci. V mikroservisní je tato vrstva zároveň jak na API-gateway, tak u každé mikroslužby.



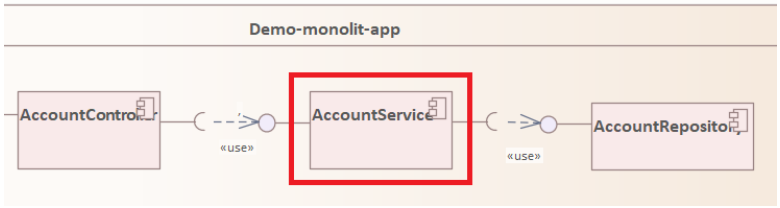
Obrázek 4.1: Account controller z diagramu komponent

```java
@RestController
@RequestMapping("/account")
@AllArgsConstructor
public class AccountController {
    private AccountService accountService;

    @PostMapping()
    public String createAccount() { return accountService.createAccount(); }

    @GetMapping( "{id}")
```

Obrázek 4.2: Úryvek kódu z account controlleru

4.2 Servisní vrstva

Servisní vrstva zodpovídá za byznys logiku systému. V našem případě posílá požadavky na repository vrstvu (viz Obrázek 4.3) a rozhoduje o tom, jestli bude požadavek proveden asynchronně nebo synchronně. Na obrázku 4.4 můžeme vidět příklad obou scénářů. Funkce updateAccount probíhá asynchroně, proto uživatel na nic nečeká a rovnou dostane odpověď na frontendu. V případě funkce getAccount můžeme vidět, že musíme čekat až systém splní požadavek na databázi (v našem případě vyčká timeout). Servisní vrstva v případě monolitické architektury vždy volá repository a nezpracovává úkoly assynchroně.



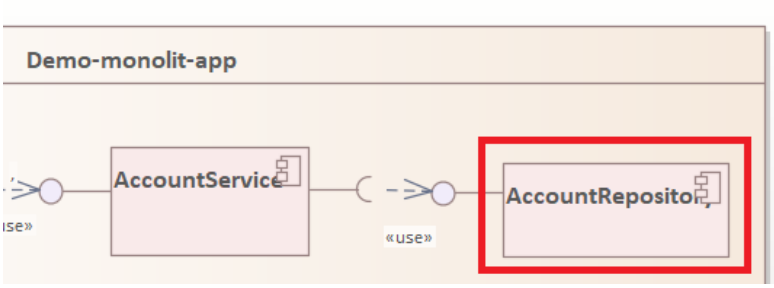
Obrázek 4.3: Account service z diagramu komponent

```java
@Service
@AllArgsConstructor
@Slf4j
public class AccountServiceImpl implements AccountService {
    private AccountRepository accountRepository;

    @Override
    public String updateAccount() {
        return "Updating account asynchronously";
//        return accountRepository.updateAccount();
    }

    @Override
    public String getAccount() { return accountRepository.getAccount(); }

    @Override
```

Obrázek 4.4: Úryvek kódu z account service

4.3 Repository vrstva

Repository vrstva je nejnižší vrstvou aplikace (viz Obrázek 4.5) a zodpovídá za komunikaci s databází. Jelikož v našem případě databáze není implementována, jsou zde nasimulované timeouty u jednotlivých funkcí, jak lze vidět na Obrázku 4.6. Tato vrstva je u obou architektur identická.



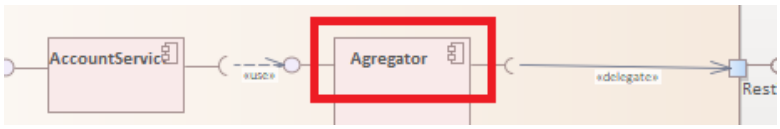
Obrázek 4.5: Account repository z diagramu komponent

```
public class AccountRepository {
    public String login(){
        try {
            Thread.sleep( millis: 50);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return "logging in!";
    }

    public String logout(){
        try {
            Thread.sleep( millis: 30);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return "logging out!";
    }
```

Obrázek 4.6: Úryvek kódu z account repository

4.4 Agregator

Tato komponenta se nachází pouze v mikroservisní architektuře, a to jako nejnižší vrstva API gateway (viz Obrázek 4.7). Zodpovídá za vybrání správného endpointu a komunikaci s ním. Má v sobě uložené všechny endpointy všech mikroslužeb. Na základě volání ze servisní vrstvy komunikuje synchronně s ostatními mikroslužbami pomocí RestTemplate (viz Obrázek 4.8)



Obrázek 4.7: Agregator v diagramu komponent

```
@Component
@AllArgsConstructor
@Slf4j
public class Aggregator {
    private RestTemplate restTemplate;
    private final String baseAccountUrl = "http://localhost:8081/account";
    private final String baseCartUrl = "http://localhost:8083/cart";
    private final String baseOrderUrl = "http://localhost:8084/order";
    private final String basePaymentUrl = "http://localhost:8085/payment";
    private final String baseProductUrl = "http://localhost:8082/product";
    private final String baseReviewUrl = "http://localhost:8086/review";
    private final String baseWishlistUrl = "http://localhost:8087/wishlist";
    // ------------------------------------ Account service ----------------------------------------------------------
    public String createAccount() { return restTemplate.postForObject(baseAccountUrl, request: null, String.class); }

    public String getAccount() {
        String url = "http://localhost:8081/account/1";
        return restTemplate.getForObject(url, String.class);
    }

    public String updateAccount() {
        return restTemplate.exchange(baseAccountUrl, HttpMethod.PUT, requestEntity: null, String.class).getBody();
    }
```

Obrázek 4.8: Úryvek kódu z agregatoru v api gateway

4.5 Konfigurace aplikace

Během vývoje aplikace byly identifikovány dvě části, které významně zpomalovaly běh aplikace a fungovaly jako tzv. bottlenecky. Prvním z nich byl RestTemplate, který se používal pro komunikaci mezi jednotlivými částmi systému. Druhým identifikovaným bottleneckem byl server Tomcat, který byl používán pro nasazení aplikace.

4.5.1 Konfigurace Rest Template

Vytvoření HTTP spojení je časově náročná operace. Výchozí nastavení RestTemplate funguje tak, že každé volání vytvoří nové HTTP spojení a po skončení spojení jej zase zavře. To znamená, že každý požadavek otevře

svůj vlastní port a vytvoří nové spojení. Při rychlosti naší aplikace se velmi rychle zaplní všechny dostupné porty, čímž se aplikace značně zpomalí a část požadavků se začne vracet chybově.

S využitím tzv. poolingu začneme využívat již vytvořené HTTP spojení. To znamená, že spojení nemusí být neustále otevíráno, čímž ušetříme čas. [21]

V našem případě jsme se díky této změně (viz Obrázek 4.9) zvedli maximální počet požadavků za vteřinu z 1600 na 2500 (testováno pomocí FastUser).

```java
@Configuration
public class Config {

    @Bean
    public RestTemplate pooledRestTemplate() {
         PoolingHttpClientConnectionManager connectionManager = new PoolingHttpClientConnectionManager();
        connectionManager.setMaxTotal(2000);
        connectionManager.setDefaultMaxPerRoute(2000);

        HttpClient httpClient = HttpClientBuilder.create()
                .setConnectionManager(connectionManager)
                .build();

        return new RestTemplateBuilder().rootUri("http://service-b-base-url:8080/")
                .setConnectTimeout(Duration.ofMillis(1000))
//                  .setReadTimeout(Duration.ofMillis(1000))
                .messageConverters(new StringHttpMessageConverter(), new MappingJackson2HttpMessageConverter())
                .requestFactory(() -> new HttpComponentsClientHttpRequestFactory(httpClient))
                .build();
    }
```

Obrázek 4.9: Konfigurace RestTemplate

4.5.2 Konfigurace Tomcat

Výchozí nastavení tomcatu je takové, že maximální počet vláken je nastaven na 200. [22] Díky jeho navýšení na 600 (viz Obrátzek 4.10) jsme zvedli maximální počet požadavků za vteřinu z 2200 na 2500 (testováno pomocí FastUser)

```
application.properties | ApiGatewayController.java | Config.java
1  spring.datasource.url=jdbc:h2:mem:testdb
2  spring.datasource.driverClassName=org.h2.Driver
3  spring.datasource.username=sa
4  spring.datasource.password=password
5  spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
6  server.port=8080
7  spring.kafka.bootstrap-servers=localhost:9092
8  server.tomcat.threads.max=600
```

Obrázek 4.10: Konfigurace RestTemplate

4.6 Locust

Jednotlivé endpointy, které chceme testovat pomocí locustu, se musí definovat v python kódu (viz Obrázek 4.11). V našem případě je definováno 33 endpointů, které testujeme napříč 7 mikroslužbami. Jednotlivé testy se jsou definovány pomocí anotace @task a k ní přiřazenému endpointu. Na začátku kódu upřesňujeme, jakou třídu HTTP clienta chceme importovat. Zde si můžeme vybrat, jestli budeme používat třídu HTTPUser nebo FastHTTPUser. Pod definicí třídy můžeme také vidět definovanou proměnou waitTime. Ta nám říká, jak dlouho jeden uživatel čeká, než zavolá další endpoint. Locust se spouští přímo z konzole (viz Obrázek 4.12). Zde můžeme nadefinovat vstupní parametry, nebo později v UI, které nalezneme na adrese http://localhost:8089. Pokud chceme spustit locust distribuovaně, tak za příkaz přidáme –master a ke každé další instanci –worker.

```python
from locust import FastHttpUser, task, between

class HelloWorldUser(FastHttpUser):
    wait_time = between(1, 5)
#--------------- account --------------------
    @task
    def get_account(self):
        self.client.get("/account/1")

    @task
    def get_account_balance(self):
        self.client.get("/account/balance/1")

    @task
    def create_account(self):
        self.client.post("/account")

    @task
    def update_account(self):
```

Obrázek 4.11: Úryvek kódu z Locustu

```
locust -f locustfile.py --host=http://localhost:8080
```

Obrázek 4.12: Spuštění Locustu přes konzoli

Kapitola 5
Testování

Pro účely testování aplikace byl použit load-testing nástroj Locust, který nabízí možnost konfigurace a spouštění testů přímo v kódu. Tímto způsobem má programátor plnou kontrolu nad definováním endpointů, které chce otestovat, a nad nastavením vstupních parametrů pro testování.

5.1 Locust

Při spuštění Locustu si uživatel definuje maximální počet uživatelů, kteří budou generováni, a také rychlost, jakou se bude počet uživatelů navyšovat za vteřinu. To umožňuje simulovat postupné zatěžování aplikace a monitorovat její odezvu při různých zátěžových scénářích.

Každý vygenerovaný uživatel v Locustu náhodně vybere jednu z nadefinovaných akcí nebo endpointů a odešle korespondující požadavek na server. To umožňuje testovat různé části aplikace a zjistit, jak se chovají při zvýšeném počtu uživatelů a zátěži. Je také možné nakonfigurovat pravděpodobnost, s jakou se jednotlivé akce vybírají, čímž se simuluje různé uživatelské scénáře.

Díky Locustu lze tedy provádět detailní zátěžové testování aplikace, monitorovat její výkon a získat cenné informace o chování aplikace při různých zátěžových podmínkách. [20]

Náš systém byl testován podle 3 různých parametrů. Zdali byla využita locustem vytvořena třída HttpUser nebo FastHttpUser. Jestli byl test distribuovaný nebo pouze jednotlivý proces a také pokud jsme testovali všech 7 mikroslužeb, nebo pouze jednu (Account microservice).

5.1.1 Nastavení testů

Pro testování systému byly použity dva typy tříd v Locustu: HttpUser a FastHttpUser. Třída HttpUser poskytuje základní funkcionalitu pro vytváření HTTP požadavků a simulaci uživatelů, zatímco třída FastHttpUser je optimalizovaná pro vysoce výkonné testování s nižší režií a lepší škálovatelností.

Co se týče distribuce testu, byly provedeny testy jak s distribuovaným scénářem, tak s použitím jednoho procesu. Distribuovaný test umožňuje spouštět Locust skript na více vláknech procesoru současně, což umožňuje simulovat vyšší zátěž a získat lepší představu o chování systému za reálných

podmínek. Testy s použitím jednoho procesu jsou vhodné pro menší zátěže, nebo pro testování na jednom lokálním stroji.

Poslední kategorií testovacích scénářů je počet testovaných mikroslužeb. V jednom případě testujeme všech 7 mikroslužeb, v druhém pouze 1. Díky tomu bude možné pozorovat změny při vyšší náročnosti systému a zátěže síťové komunikace.

Celkově byly provedeny testy s různými parametry a scénáři, aby se získalo co nejvíce informací o výkonu systému a jeho jednotlivých komponentech. To umožňuje identifikovat a řešit případné problémy a zajistit, že systém splňuje požadované výkonnostní a odolnostní požadavky.

5.1.2 HttpUser nebo FastHttpUser

Třída HttpUser v rámci Locustu využívá klasickou knihovnu Python Requests pro odesílání HTTP požadavků. Tato třída poskytuje základní funkcionality pro simulaci uživatelů a odesílání požadavků na testovaný systém.

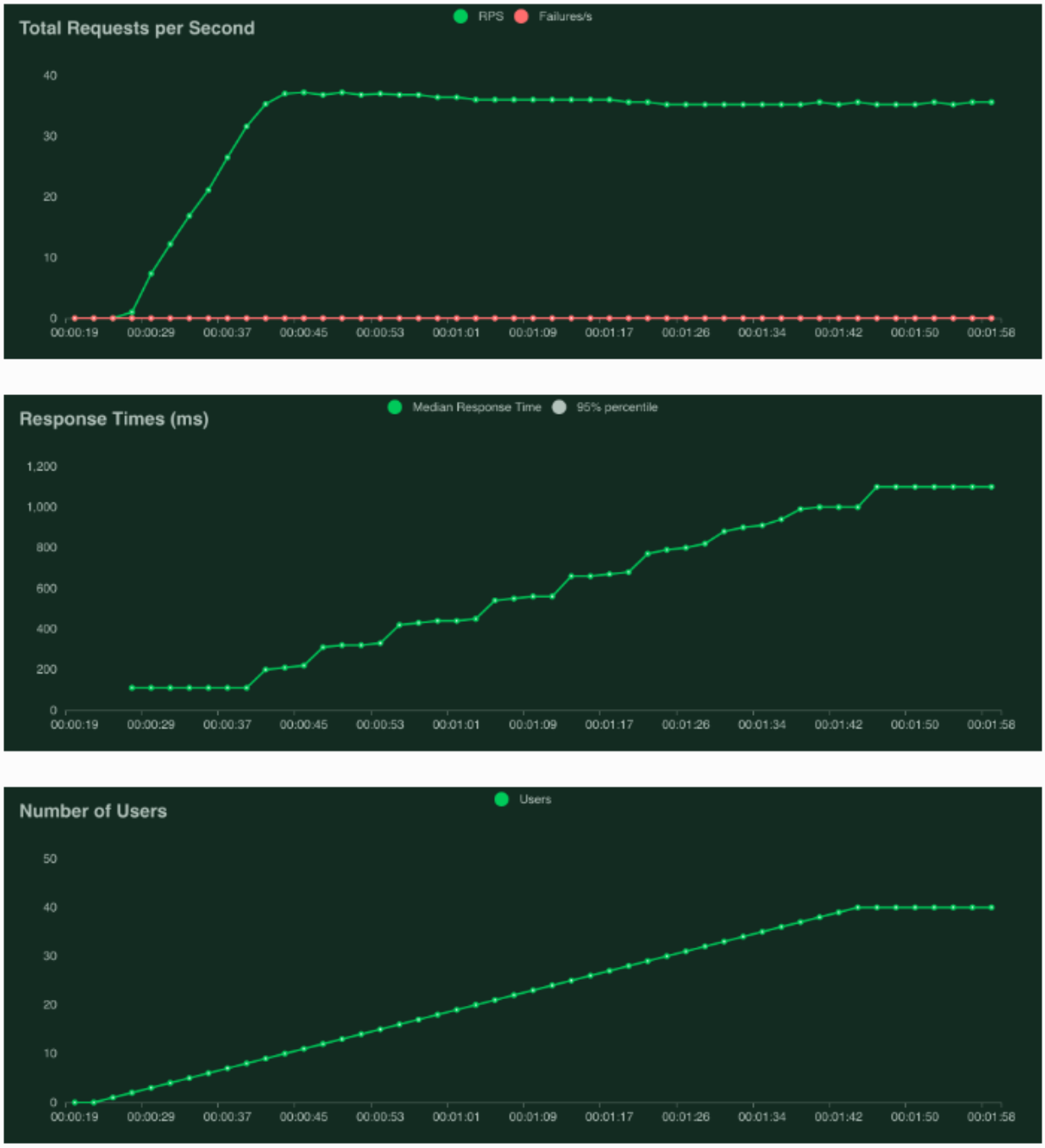
Na druhou stranu, třída FastHttpUser využívá knihovnu GeventHttpclient, která je napsaná v jazyce C. Díky tomu je tato třída výkonnější a má nižší režii než třída HttpUser. Díky optimalizaci v jazyce C je FastHttpUser schopný dosáhnout vyššího počtu odeslaných požadavků za jednotku času a efektivněji využívat CPU zdroje. To umožňuje dosáhnout vyššího maximálního počtu uživatelů na stejném hardwaru v porovnání s běžným HttpUserem, a to až na 5-6 násobek. [16]

5.1.3 Distribuované testy

Distribuované testování pomocí Locustu umožňuje dosáhnout ještě vyššího výkonu a zátěže na testovaném systému. Při použití více instancí Locustu současně je možné simulovat vysoký počet požadavků za sekundu. Master instance koordinuje a řídí testování, zatímco worker instance vykonávají požadavky na testovaný systém. Distribuované testování využívá více jader procesoru a umožňuje dosáhnout extrémní zátěže. Je vhodné pro testování velkých a komplexních systémů.[17]

5.1.4 Jak číst výsledek testu

Na základě pozorování grafu lze identifikovat zlomový bod počtu uživatelů, při kterém se dosahuje maximálního výkonu systému. Zlomový bod je takový, kdy počet uživatelů narůstá, ale počet požadavků za sekundu (RPS) již nezvyšuje rovnoměrně s počtem uživatelů. V příkladu testu na obrázku 5.1 je tento bod lokalizován kolem 9 uživatelů. Okolo tohoto bodu můžeme pozorovat že počet uživatelů stále konstatně stoupá, ale RPS stagnuje okolo 38 uživatelů. Pokud náš server opravdu nestíhá, může se stát, že odpověď serveru je chybová hláška, poměr volání, které se vrátili chybně, jsou označeny na grafu RPS jako červená křivka.



Obrázek 5.1: Vzorový locust test

5.1.5 Chybové hlášky

Ve chvíli, kdy jsme dosáhli zlomového bodu našeho systému, část požadavků se vracela s chybovými hláškami. Při identifikaci příčin těchto chyb se nám podařilo eliminovat několik z nich díky určité úpravě systému. Je důležité upozornit, že základní třída HTTPUser s jednovláknovým testem nedokázala v žádném případě nasimulovat dostatečný počet požadavků, který by náš systém dostal do kritického stavu a začal vracet chybové hlášky.

[Errno 10061] [WinError 10061] Nemohlo být vytvořeno žádné připojení, protože cílový počítač je aktivně odmítl.

Tento chybový stav říká, že server aktivně odmítl spojení. V našem konkrétním případě to znamená, že server byl přetížen a nedokázal přijmout další spojení.

URL http://localhost:8080/review/1: 1, original=timed out

Timeout požadavku překročil maximální čekací dobu a server nestihl odpovědět. K tomuto dochází vzhledem k tomu, že s narůstajícím počtem požadavků na server se zvyšuje i doba, kterou server potřebuje k odpovědi. Pokud tato doba překročí stanovený limit pro čekání, požadavek je zahozen.

URL http://localhost:8080/account: code=500

Tato chybová zpráva označuje "neočekávanou chybu na straně serveru". V našem případě se tato chyba vyskytla pouze v mikroslužbové architektuře a byla způsobena tím, že RestTemplate nedokázal dostatečně rychle zpracovávat požadavky. Tento problém vznikal kvůli tomu, že RestTemplate výchozím nastavením pro každý požadavek otevírá nový dočasný port, vyřídí požadavek a port poté uzavře. Při vysoké rychlosti našich požadavků se všechny dostupné porty rychle zaplní, a proto většina požadavků končí chybou.

Nicméně pokud využijeme Apache HTTP Connection pooling, budeme využívat již otevřená spojení. To znamená, že se pro každý požadavek nemusí vytvářet nové spojení, což výrazně šetří čas. Zejména HTTP handshake zabíral významné množství času. [18]

RetriesExceeded('http://localhost:8080/wishlist/product/remove/1', 1, original=timed out))

Tato chyba nastane, pokud necháme defaultně nastavený thread pool tomcatu. Toto defaultní nastavení není dostačující a některé požadavky jsou zahozeny kvůli tomu, že bylo dosaženo maximálního počtu vláken. V našem případě jsme chybu odstranili nastavení počtu vláken na 600.

5.1.6 Výsledky testování

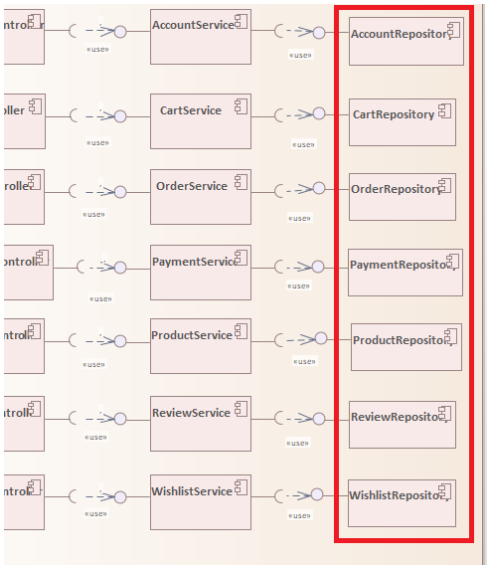
Výsledky testů jednoznačně ukazují, že testovat náš systém pomocí základní knihovny využívající třídy HTTPClient na testování je nedostačující, jelikož

simulace požadavků uživatelů byla pro procesor náročnější, než samotný běh aplikace. Pokud jsme ovšem využili třídy fastHTTPClient, limit obou aplikací byl někde okolo 2700 RPS pro monolitní i mikroservisní architekturu. Distribuované testy byly nastavené na 3 instance workerů a 1 mastera. Ani toto ovšem nedosáhlo takového výkonu, jako fastHTTPClient. Při kombinaci distribuovaného testu a fast HTTP Clienta se nám výsledek již výrazně neměnil, to je indikátor toho, že jsme skutečně našli limit našeho systému. Z Obrázku 5.2 je patrné, že maximální počet požadavků za vteřinu u obou aplikací se pohyboval kolem 2700. Tento výsledek je zajímavý vzhledem k výhodě mikroservisní architektury, která má všechny asynchronní požadavky nastaveny s timeoutem 0.

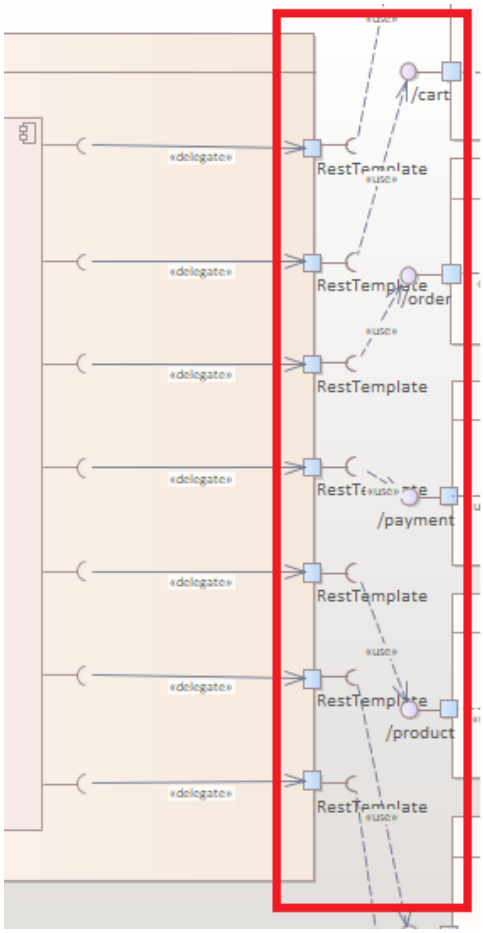
Existuje několik faktorů, které k tomuto výsledku přispívají. Jedním z nich je zvýšená síťová komunikace mezi jednotlivými mikroslužbami, což vede ke zpomalení systému jako celku. Dalším nedostatkem je, že při testování jednotlivých služeb byla aplikace spuštěna pouze v InteliJ Idea (IDE) a ne např. v Dockeru. Největším bottleneckem mikroservisní aplikace byla komunikace napříč komponenty (viz Obrázek 5.3), zatímco u monolitní aplikace to byla repository vrstva (viz Obrázek 5.4).

			Mikroservisy [RPS]	Monolit [RPS]
1 endpoint	jednovláknový test	HTTP Client	1250	1223
		Fast HTTP Client	2720	2650
	distribuovaný test	HTTP Client	1818	2501
		Fast HTTP Client	2700	2713
7 endpointů	jednovláknový test	HTTP Client	1175	1209
		Fast HTTP Client	2500	2688
	distribuovaný test	HTTP Client	2330	2617
		Fast HTTP Client	2670	2700

Obrázek 5.2: Výsledky testování pro mikroservisní architekturu



Obrázek 5.3: bottleneck monolitní architektury



Obrázek 5.4: bottleneck mikroservisní architektury

Kapitola 6

Závěr

Tato práce se zaměřovala na výzkum design patternů a technologií vhodných k vývoji mikroservisní architektury. Hlavním cílem bylo vytvořit a porovnat dvě aplikace: jednu postavenou na mikroservisní architektuře a druhou na monolitní architektuře. Práce se skládala ze čtyř hlavních částí: rešerše, návrhu, implementace systému a testování.

V rešeršní části byly podrobněji popsány různé design patterny používané při vytváření komponent mikroservisních architektur. Dále byly zkoumány různé technologie pro asynchronní volání backendu. Tyto technologie byly původně zvažovány pro implementaci projektu, avšak nakonec bylo rozhodnuto o jejich vynechání z důvodu jejich složitosti implementace a testování.

Při návrhu systému byl nejprve vytvořen datový model, který sloužil jako základ pro další analýzu. Na základě tohoto diagramu byla provedena analýza endpointů, což umožnilo vznik obou architektur, které jsou obě popsány v diagramu komponent. V případě mikroservisní architektury byla využita implementace proxy patternu pomocí api-gateway. Komunikace mezi jednotlivými mikroslužbami byla realizována pomocí RestTemplate. Pro dva vybrané scénáře, tj. vytvoření účtu a získání informací o účtu, byly vytvořeny sekvenční diagramy pro obě architektury.

Nakonec byly obě aplikace podrobeny testování pomocí Locustu. Během testování byly využity dva přístupy, konkrétně HTTP User a Fast HTTP User, pro provádění testování výkonu. Tyto dvě metody testování, včetně distribuovaných testů, byly vzájemně porovnány s cílem co nejlépe odhadnout výkonový limit naší aplikace. Překvapivě výsledky testování neprokázaly významný rozdíl mezi mikroservisní a monolitní architekturou, přestože mikroservisní architektura disponovala výhodou asynchronních endpointů. Hlavním faktorem bylo zvýšení komunikace mezi jednotlivými částmi systému u mikroslužeb.

Je důležité zdůraznit, že při jiné konfiguraci systému, kde by si plně využilo potenciálu škálování mikroservisní architektury, by mikroservisní architektura dosáhla mnohem lepších výsledků než monolitní. Jelikož bychom mohli libovolně škálovat jednotlivé instance systému dle potřeby pomocí virtualizace systému.

Další kroky

Jednou z možností, jak navázat na tuto práci, je implementovat modelovou vrstvu na základě diagramu tříd(viz Obrázek 3.1). Tím by bylo možné vyzkoušet a otestovat různé design patterny při reálném provozu aplikace. Dále by bylo vhodné navázat na analýzu endpointů a odhadnout frekvenci volání jednotlivých endpointů. Tato analýza by umožnila škálování jednotlivých komponent nové aplikace podle potřeby.

Kapitola 7

Literatura

1. Chris Richardson. Pattern: Database per service.
URL: <https://microservices.io/patterns/data/database-per-service.html>[cit. 8.12.2022]

2. Chris Richardson: Pattern: SAGA.
URL: <https://microservices.io/patterns/data/saga.html>[cit. 8.12.2022]

3. Kasun Dissanayake: Circuit Breaker Pattern — Microservice Architecture.
URL: <https://medium.com/nerd-for-tech/circuit-breaker-pattern-microservice-architecture-4c6b1a06f3f3>[cit. 15.12.2022]

4. Microsoft: Bulkhead pattern.
URL: <https://learn.microsoft.com/en-us/azure/architecture/patterns/bulkhead>[cit. 16.12.2022]

5. Mehmet Ozkaya: Service Aggregator Pattern.
URL: <https://medium.com/design-microservices-architecture-with-patterns/service-aggregator-pattern-e87561a47ac6>[cit. 16.12.2022]

6. Irushinie Muthunayake: Proxy Microservice Design Pattern.
URL: <https://medium.com/nerd-for-tech/proxy-microservice-design-pattern-91d455b0d05a>[cit. 17.12.2022]

7. Kieran Kilbride-Singh: WebSockets vs Long Polling: Key differences and which to use.
URL: <https://ably.com/blog/websockets-vs-long-polling>[cit. 20.12.2022]

8. Samuel Olusola: Server-sent events vs. WebSockets.
URL: <https://blog.logrocket.com/server-sent-events-vs-websockets>[cit. 17.12.2022]

9. Ilya Grigorik: Introduction to HTTP/2.
URL: <https://web.dev/performance-http2/>[cit. 28.12.2022]

10. Siddharth Singh: What is HTTP Long Polling.
URL: <https://www.educative.io/answers/what-is-http-long-polling>[cit. 28.12.2022]

11. Microsoft: Asynchronous Request-Reply pattern.
URL: <https://learn.microsoft.com/en-us/azure/architecture/patterns/async-request-reply>[cit. 2.1.2022]

12. De Lauretis, Lorenzo. "From monolithic architecture to microservices architecture."2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). IEEE, 2019.

13. Alshuqayran, Nuha, Nour Aali, and Roger Evans. "A systematic mapping study in microservice architecture."2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA). IEEE, 2016.

14. Rudrabhatla, Chaitanya K. "Comparison of event choreography and orchestration techniques in microservice architecture."International Journal of Advanced Computer Science and Applications 9.8 (2018).

15. Madsen, Magnus, Ondřej Lhoták, and Frank Tip. "A model for reasoning about JavaScript promises."Proceedings of the ACM on Programming Languages 1.OOPSLA (2017): 1-24.

16. Locust: FastHttpClient.
URL: <https://docs.locust.io/en/stable/increase-performance.html>[cit. 13.05.2023]

17. Locust: Distributed load generation.
URL: <https://docs.locust.io/en/stable/running-distributed.html>[cit. 14.05.2023]

18. Nitin Vohra: How to improve performance of Spring RestTemplate.
URL: <https://medium.com/@nitinvohra/how-to-improve-performance-of-spring-resttemplate-6af37e0a0f33>[cit. 14.05.2023]

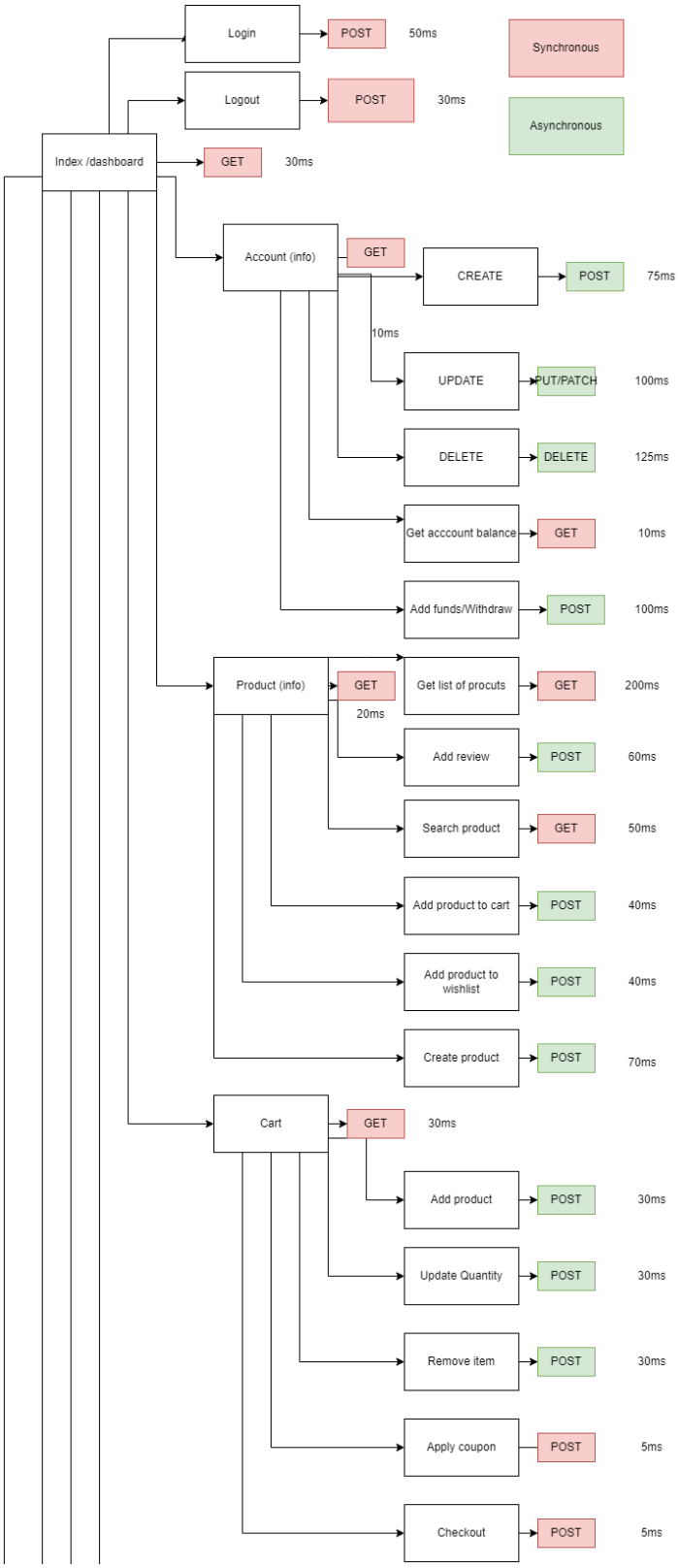
19. Zimmermann, Torsten, et al. "How HTTP/2 pushes the web: An empirical study of HTTP/2 server push."2017 IFIP Networking Conference (IFIP Networking) and Workshops. IEEE, 2017.

20. Locust: What is Locust?
URL: <https://docs.locust.io/en/stable/what-is-locust.html>[cit. 21.05.2023]

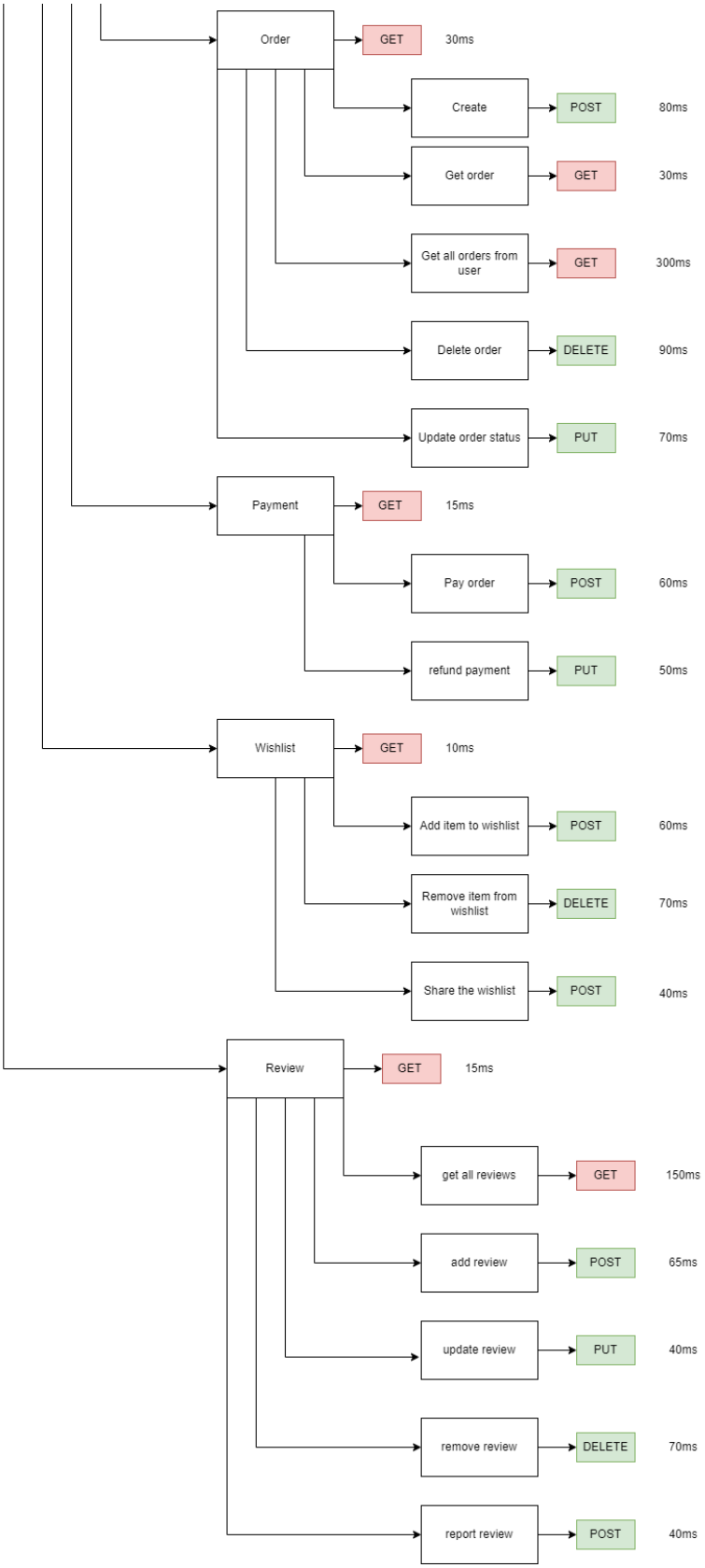
21. Alshuqayran, Nuha, Nour Aali, and Roger Evans. "A systematic mapping study in microservice architecture."2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA). IEEE, 2016.

22. Brittain, Jason, and Ian F. Darwin. Tomcat: The Definitive Guide: The Definitive Guide. "O'Reilly Media, Inc.", 2007.

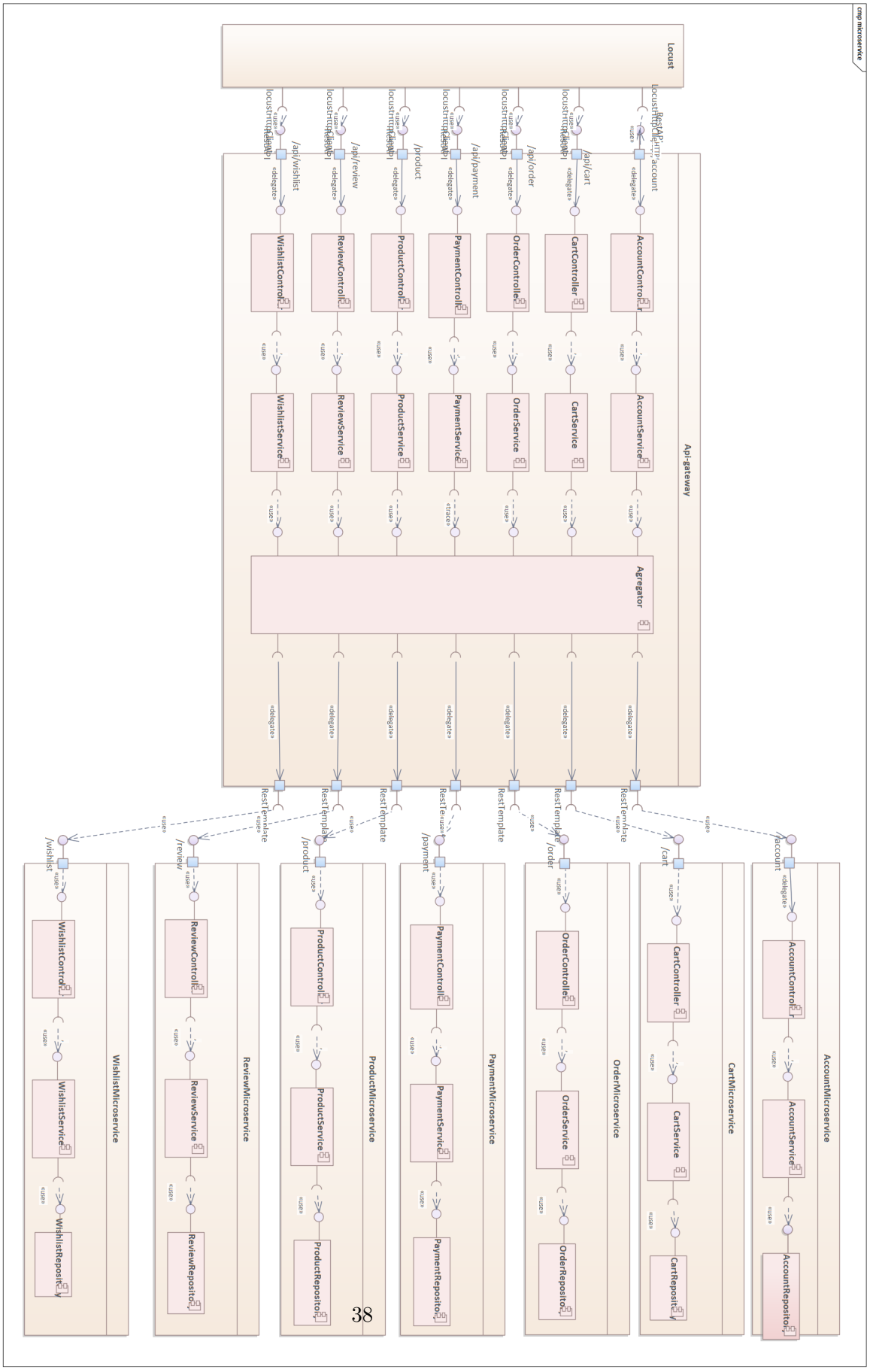
Příloha



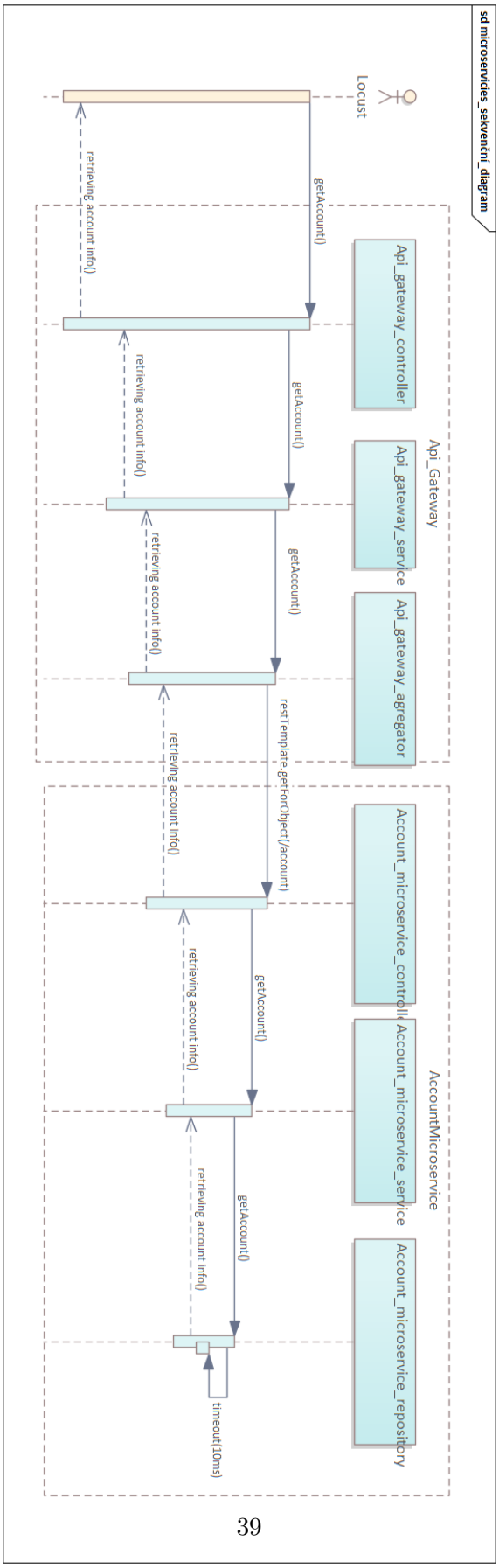
Obrázek 7.1: Analýza endpointů 1



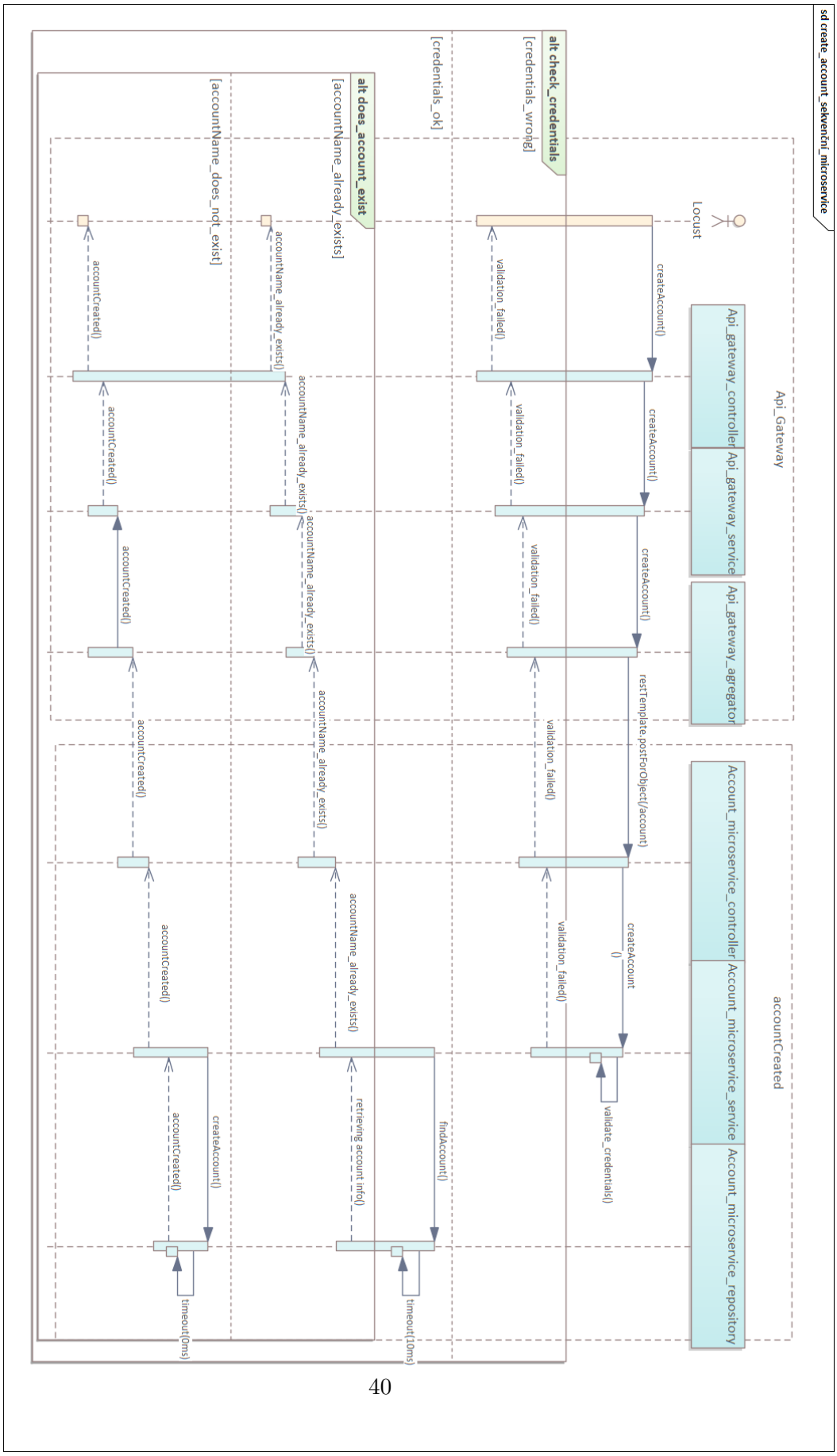
Obrázek 7.2: Analýza endpointů 2



Obrázek 7.3: Diagram komponent pro mikroservisní architekturu



Obrázek 7.4: Sekvenční diagram getAccount pro mikroslužby



Obrázek 7.5: Sekvenční diagram createAccount pro mikroslužby