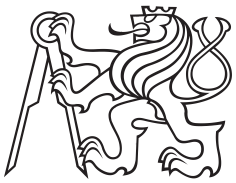


České
vysoké
učení technické
v Praze

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 s využitím design patternů. 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]

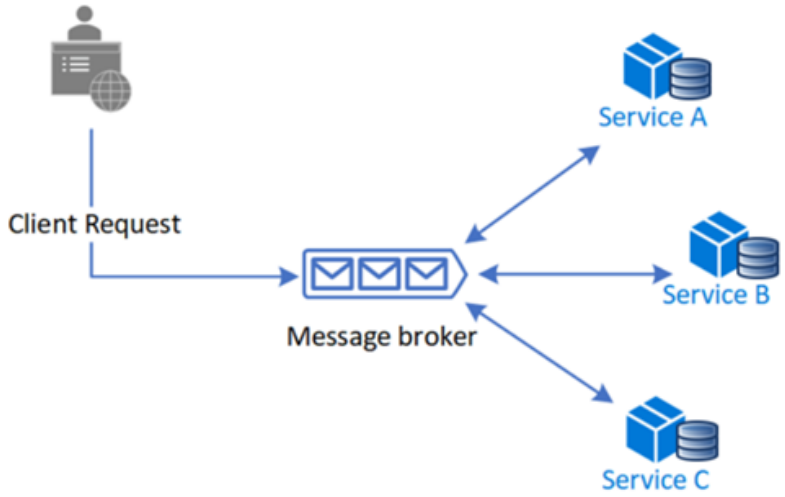
Transakce musí splňovat pravidla ACID: A - Atomicity (Nedělitelnost) znamená, že operace musí být nedělitelné a neredukovatelné, musí se provést buď všechny nebo žádná. C - Consistency (Konvergence) znamená, že transakce převádí data pouze z platného stavu do dalšího platného stavu. I - Isolation (Izolace) 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]

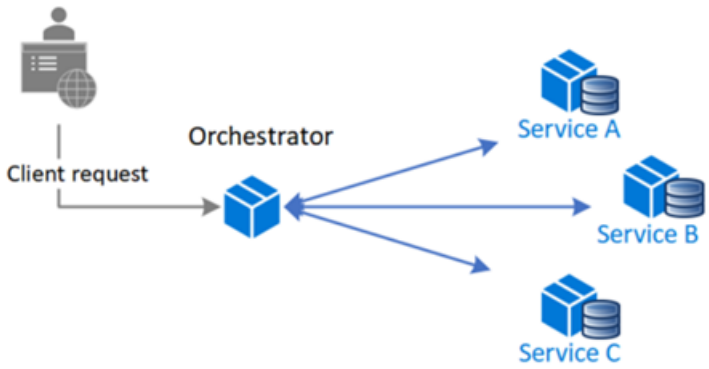


Obrázek 2.1: Choreography pattern [26]

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 [26]

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]

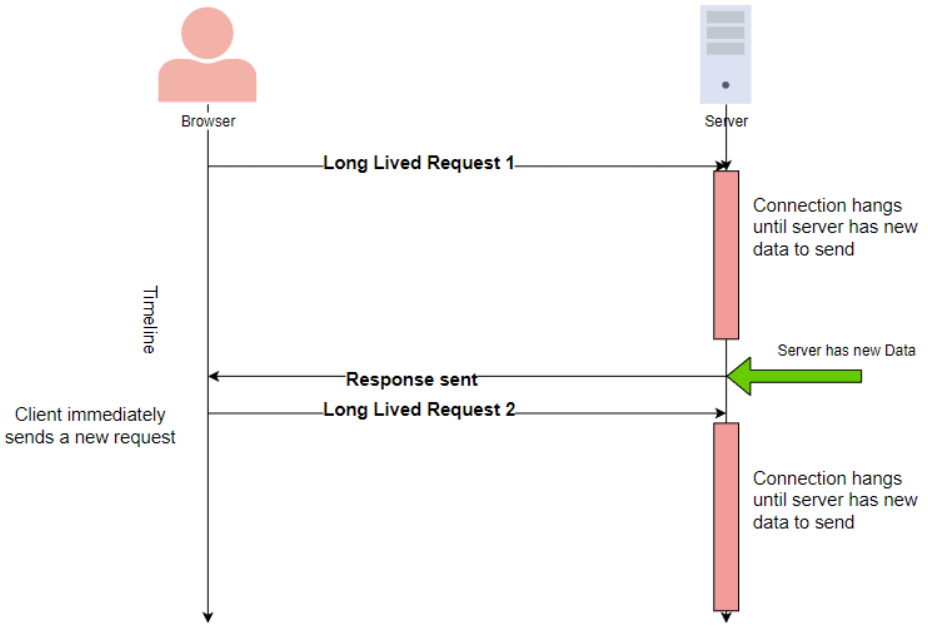
2.2 Technologie k asynchronnímu volání backendu.

Asynchronost je jedním z hlavních patternů mikroservisních architektur. Jednotlivé mikroslužby využívají faktu, že některé časově náročné operace lze zpracovávat na pozadí. Toho se docílí tím, že jedna mikroslužba pošle druhé zprávu a nečeká na její odpověď, tím se komunikace značně urychlí.

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]



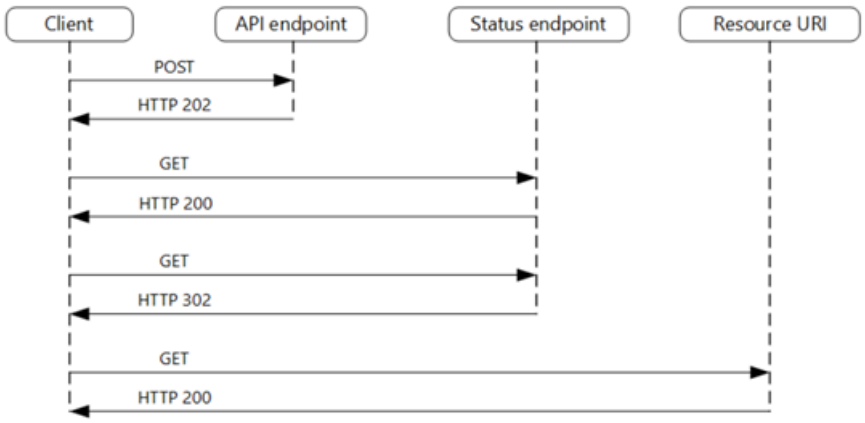
Obrázek 2.3: Long polling [9]

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).

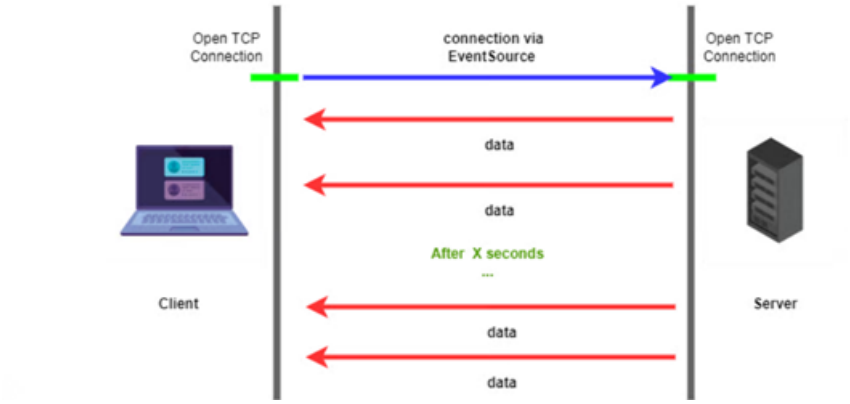
Pokud klient volá stavový endpoint, API odpovídá s kódem HTTP 200. Pokud se požadavek stále zpracovává, endpoint vrátí zprávu, že požadavek ještě není dokončen. Po dokončení procesu endpoint informuje o dokončení požadavku, nebo přesměruje klienta na jinou URL. Například pokud asynchronní operace vytvoří nový zdroj, endpoint přesměruje uživatele na URL tohoto zdroje [10] (viz Obrázek 2.4).



Obrázek 2.4: httpPolling [10]

2.2.3 Server-sent events

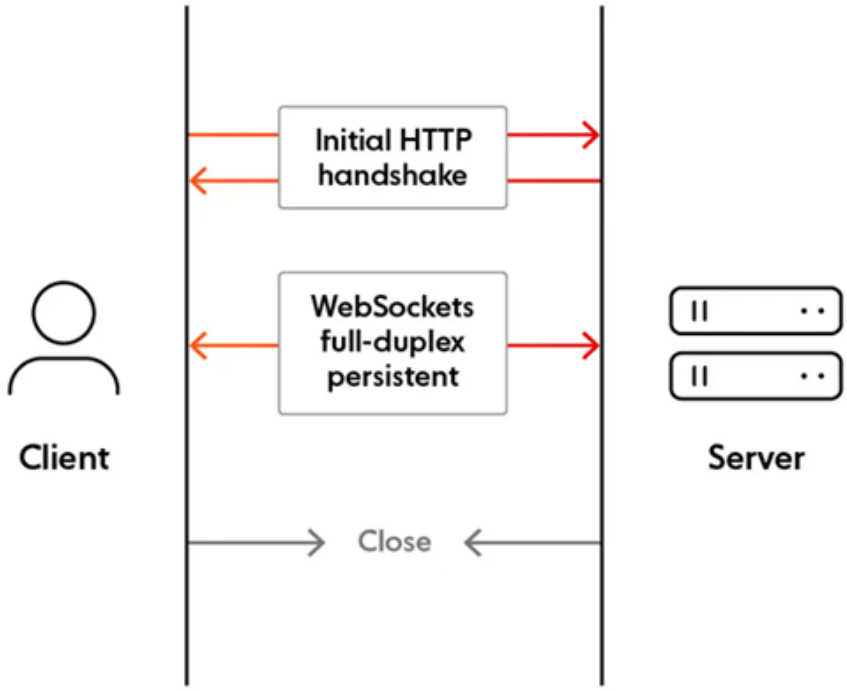
Technologie, která umožňuje klientovi automaticky přijímat data ze serveru přes HTTP spojení, se nazývá Server-sent events (SSE). Po navázání spojení mezi klientem a serverem může server aktivně odesílat aktualizace klientovi (viz Obrázek 2.5). Toto spojení je realizováno pomocí JavaScript API EventSource. 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 [7]

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: Weboscketes [6]

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]

2.2.6 Apache Kafka

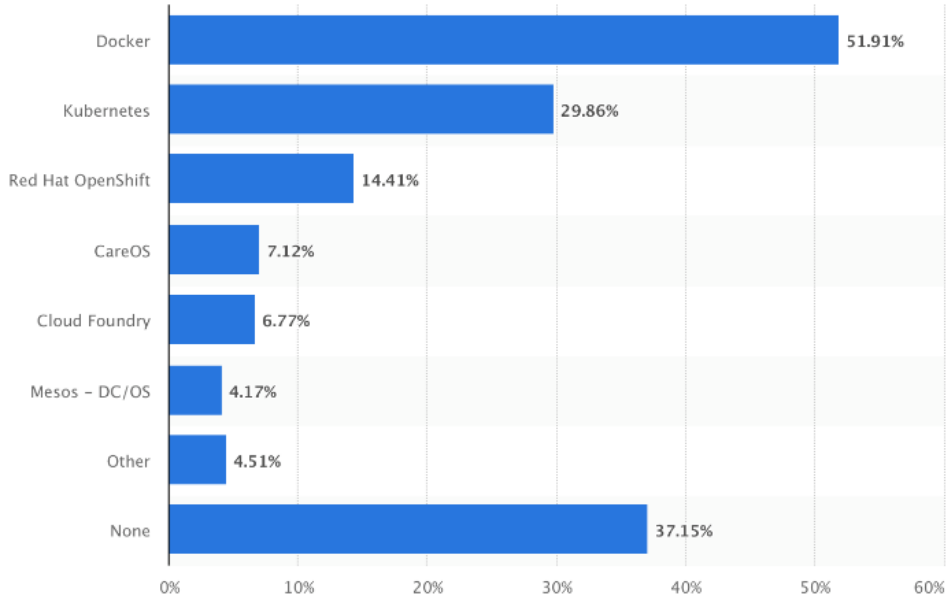
Je distribuovaná platforma pro zpracování a streamování dat, navržena pro efektivní, spolehlivou a škálovatelnou komunikaci mezi a systémy v reálném čase. Je často používaná pro zpracování velkého objemu dat, distribuovaného zpracování a logování událostí. Je postavena na základě publish-subscribe modelu [23]. Centrálním prvkem je Apache ZooKeeper, ten slouží jako distribuovaný kooridnátor pro celý Kafka cluster. Uschovává metadata, sleduje stav brokerů a zajišťuje, aby celý cluster byl správně koordinován. Broker je server, který ukládá a spravuje data. Brokeři tvoří kafka cluster, který se skládá z několik severů, mezi nimiž jsou replikována data a navzájem spolu komunikují, aby zajistily vysokou dostupnost a spolehlivost. Producer

a Consumer jsou další dva základní koncepty. Producer je komponenta, která generuje a posílá zprávy do Kafka topiců. Consumer zase naslouchá a čte zprávy z topiců. Eventy vytvořené producentem jsou ukládané jako topicy. Topic může být mít nula či několik producentů, co do topicu vytvářejí eventy. Zároveň může mít nula až několik consumerů, co naslouchají na tyto eventy. Poslední důležitou částí je partition, ten nám říká, mezi kolika brokery je topic rozdělen, čím vyšší jeho hodnota, tím více brokerů bude mít daný topic.

Mikroslužby využívájí Kafku zejména k asynchronní komunikaci, logování, sdílení dat a distribuovanému zpracování dat. Jednotlivé služby mohou mezi sebou komunikovat asynchronně, což zvyšuje škálovatelnost a robustnost systému. Kafka také poskytuje uložiště pro logy, což usnadňuje monitorování a diagnostiku systému. [27]

2.3 Virtualizační technologie pro vývoj mikroservisních architektur

Virtualizační technologie hrají při vývoji mikroservisních architektur důležitou roli, jelikož umožňují izolovat jednotlivé mikroslužby a vytvářet prostředí pro jejich provoz. Díky těmto technologiím lze jednotlivé mikroslužby snadno škálovat dle požadavků. Dále umožňuje běh několika operačních systémů na jednom počítači jako virtuální stroje. Od roku 2018 více než polovina velkých IT projektů využívá Docker. Na druhém místě je kubernetes, společně tyto technologie jsou revoluční právě pro vývoj mikroservisních architektur (viz Obrázek 2.7). [29]

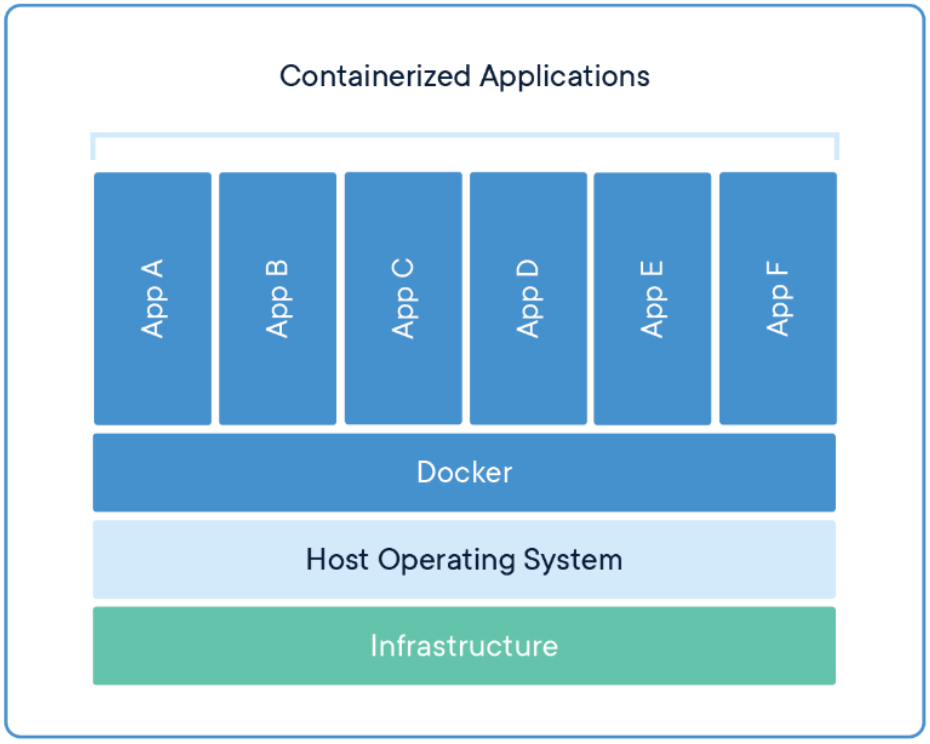


Obrázek 2.7: Poměr zastoupení technologií ke kontejnerizaci [29]

2.3.1 Docker

Je nástroj, který se stal klíčovým prvkem v oblasti kontejnerizace aplikací. To je technologie, která umožňuje zabalit a spouštět aplikace a všechny jejich závislosti v izolovaných kontejnerech. Každý kontejner obsahuje, vše co potřebuje k běhu aplikace, včetně kódu, knihoven, runtime prostředí a konfigurace. Tím se zajišťuje, že aplikace poběží na jakémkoliv systému, kde je Docker dostupný, bez ohledu na rozdíly mezi jednotlivými systémy. Díky Dockeru lze tyto kontejnery vytvářet, spouštět a spravovat. Jeho klíčovou funkcí je izolace kontejnerů, což znamená, že aplikace běžící v jednom kontejneru neovlivní ostatní kontejnery. Díky izolaci kontejnerů se minimalizují konflikty napříč mikroslužbami, což zvyšuje bezpečnost celé aplikace. Jelikož uvnitř kontejneru je vše, co je potřeba pro běh aplikace, tak je nasazení aplikací značně rychlejší a jednodušší. Další výhodou je škálovatelnost, duplikací kontejnerů můžeme libovolně rozmístit výpočetní sílu systému, na kterém aplikace běží. Obecnou strukturu Dockeru lze vidět na Obrázku 2.8.

Hlavním stavebním blokem Dockeru je Image. V něm je uložena celá aplikace, včetně knihoven, konfigurací a souborů. Image je tvořena podle předpisu, který je definován v souboru nazývaném Dockerfile. To je textový soubor obsahující seznam instrukcí pro sestavení Image. Image je poté nahrána do kontejneru, ve kterém lze spustit jako celá aplikace. [24]

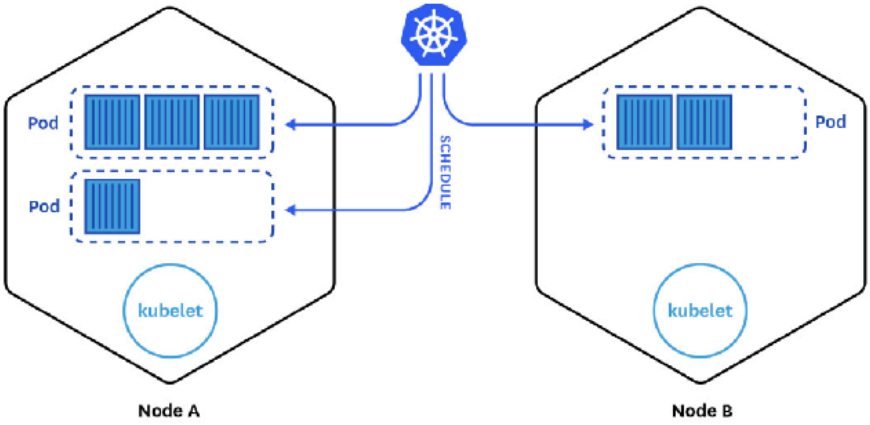


Obrázek 2.8: Docker struktura [28]

2.3.2 Kubernetes

Kubernetes (někdy zkracováno jako K8s) je open-source platforma pro orchestraci kontejnerů. Cílem kubernetes je usnadňovat správu, škálování a nasazení kontejnerizovaných aplikací, zejména v mikroservisním prostředí.

V kubernetes jsou 4 základní koncepty. Pod, ReplicaSet, Deployment a Service. Pod je nejmenší jednotkou v Kubernetes. Ten obsahuje jeden nebo více kontejnerů. ReplicaSet je číslo, které určuje počet kopií daného podu, díky čemuž lze aplikaci snadno škálovat. Deployment říká kubernetes, jak vytvořit a modifikovat instace podu v kontejnerizované aplikaci. Service slouží jako abstrakce nad souborem podů, která umožňuje komunikaci mezi nimi. Způsob vytváření podů, servic a deployment je popsán v Yaml souboru [25]. Ukázka architektury s využitím kubernetes a různým rozložením podů je na Obrázku 2.9.



Obrázek 2.9: Architektura s kubernetes [29]

2.4 Shrnutí rešerše

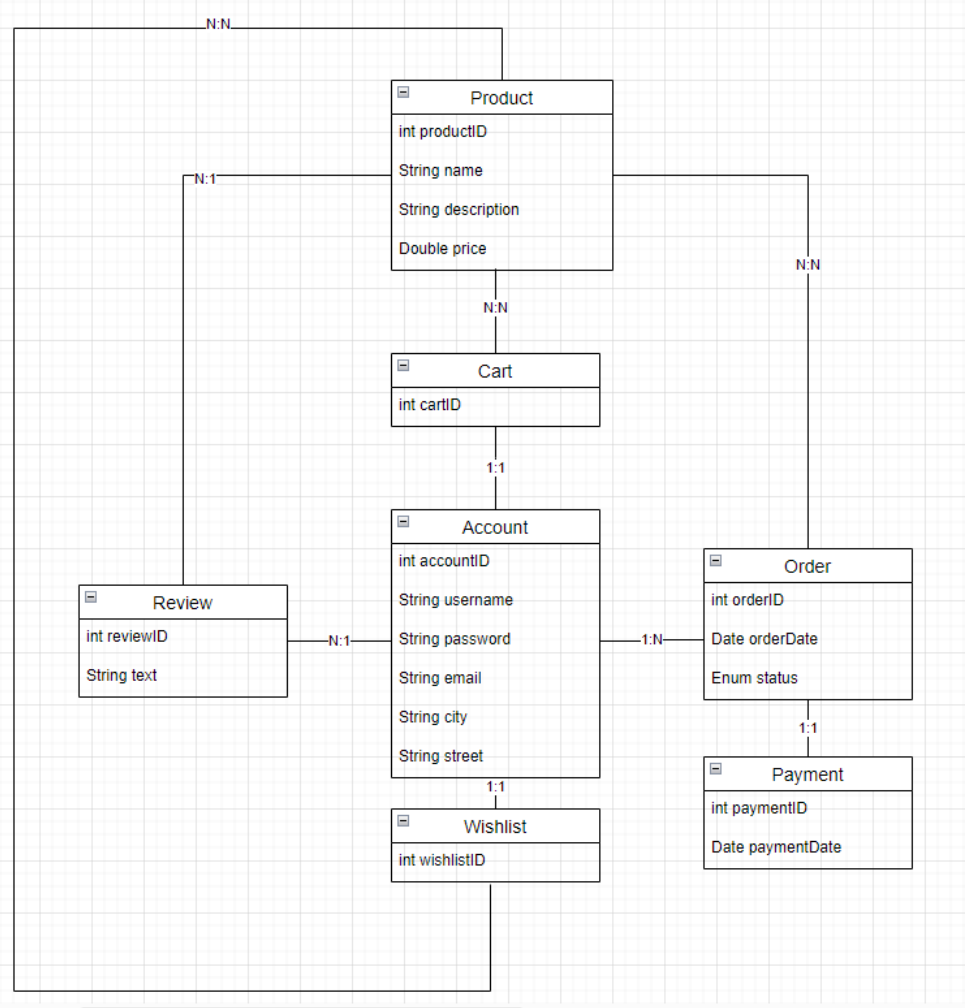
V první části byly představeny design patterny, které jsou využívané při vývoji mikroservisních architektur. Tyto patterny budou v tomto projektu dále využity při implementaci demo aplikace. Z technologií, které jsou využívané k asynchronní komunikaci, byla v dalších částech této práci využita pouze Apache Kafka. Virtualizační technologie, popsané ve třetí části rešerše, byly využity obě dvě, jak Docker, tak Kubernetes.

Kapitola 3
Návrh systému

Naším cílem je porovnání dvou architektur, a to monolitní a mikroservisní a zároveň v mikroservisní architektuře co nejlépe využít desing patterny, které byly zmíněny v teoretické části práce. Nejprve byl vytvořen datový model, který popisuje základní entity v systému, jejich atributy a vzájemné relace. Na jeho základě byla vytvořena analýza endpointů, které každé entitě přiřazuje různé endpointy, které budou fungovat jako API našeho systému. Poté byly vytvořeny diagramy komponent, jeden pro monolitní a druhý pro mikroservisní architekturu, které detailněji popisují jednotlivé součásti systému. U mikroservisní architektury je vždy nudné brát v potaz, že jednotlivé služby spolu musí navzájem komunikovat, tudíž je důležité si rozmyslet, jakým způsobem bude tato komunikace implemetována. V našem případě to je RestTemplate a Kafka. Nakonec byly vytvořeny sekvenční diagramy, které reprezentují jednotlivé use-casy systému.

3.1 Datový model

Model systému vychází z obecné struktury e-shopu (Obrázek 3.1), druh aplikace tohoto projekt byl zvolen na základě konzultace s vedoucím práce. V diagramu se nachází 7 entity. Základem je entita account, která reprezentuje uživatelský účet. Každý účet má vlastní wishlist (seznam přání) a cart (nákupní košík). Wishlist si ukládá předměty, které by si uživatel rád pořídil. Cart je nákupní košík, kam si uživatel přidává produkty. Každý produkt má několik reviews (recenzí), která má právě jednoho autora. Další entitou je objednávka. Ta je vázaná právě k jednomu účtu a obsahuje různé množství produktů. Při implementaci byly vytvořeny další dvě pomocné entity. CartItem a OrderItem, tyto entity slouží k udržení si počtu produktů (např. když si uživatel dá do košíku jeden produkt dvakrát). Tyto dvě entity nejsou zahrnuty v diagramu, jelikož nejsou součástí tzv. business entit.

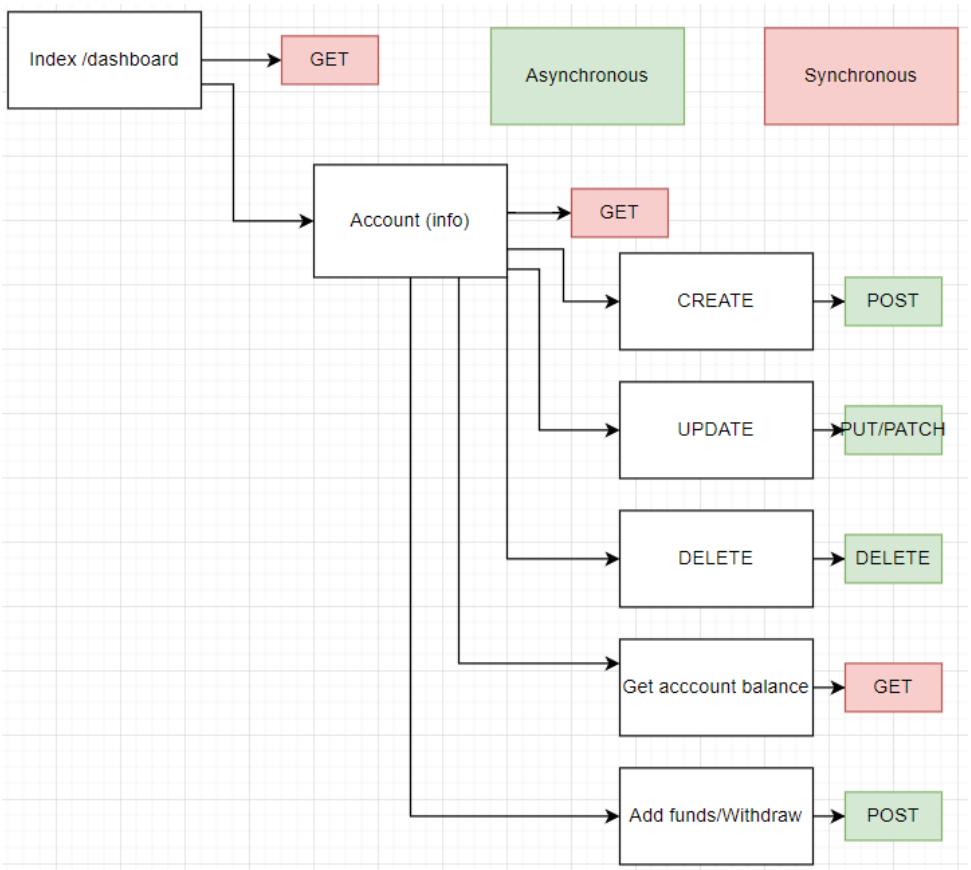


Obrázek 3.1: Datový model

3.2 Analýza endpointů

Pro každou entitu z datového modelu bylo vytvořeno několik endpointů. Pro každý druh požadavku byl přiřazen druh HTTP metody. Create jsou požadavky, které vytvářejí novou entitu a tu ukládají do systému, update bere již existující entitu, tu pozmění a pozměněnou ji uloží do databáze. Delete odstraňuje existující entitu z databáze a get ji v databázi najde a vrátí ji. Requesty, které musí být vykonány plně synchronně, jsou znázorněny červenou barvou. Naopak požadavky, co lze vykonat buď částečně nebo plně asynchronně, poskytují mikroservisní architektuře jistou výhodu, jelikož odpoví téměř okamžitě, a časově náročnou část vyřídí na pozadí. Na Orbázku. 3.2 lze vidět ukázku této analýzy. Veškeré metody get jsou plně synchronní, jelikož abychom vrátili entitu z databáze, musíme počkat na vyřešení celého požadavku. Například operace create v našem případě lze vykonat asynchronně. Jak probíhá toto asynchronní volání je blíže vysvětleno

v sekvenčních diagramech.



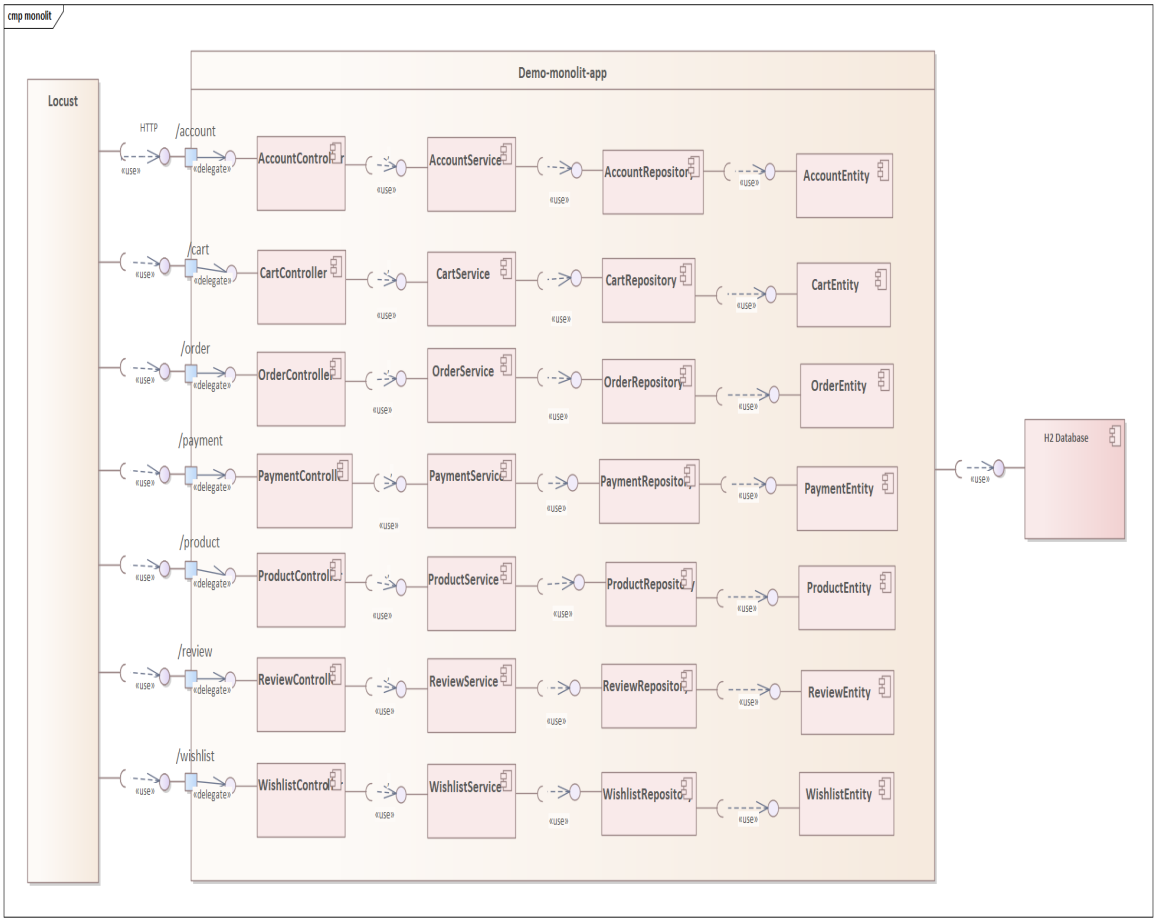
Obrázek 3.2: Analýza endpintů

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 repository vrstvou, která posílá požadavky do databáze. Poslední vrstva obsahuje jednotlivé entity systému, které jsou ukládány do databáze pomocí repository vrstvy a jejich chování je spravováno pomocí servisní vrstvy. Monolitní aplikace má k dispozici právě jednu databázi.

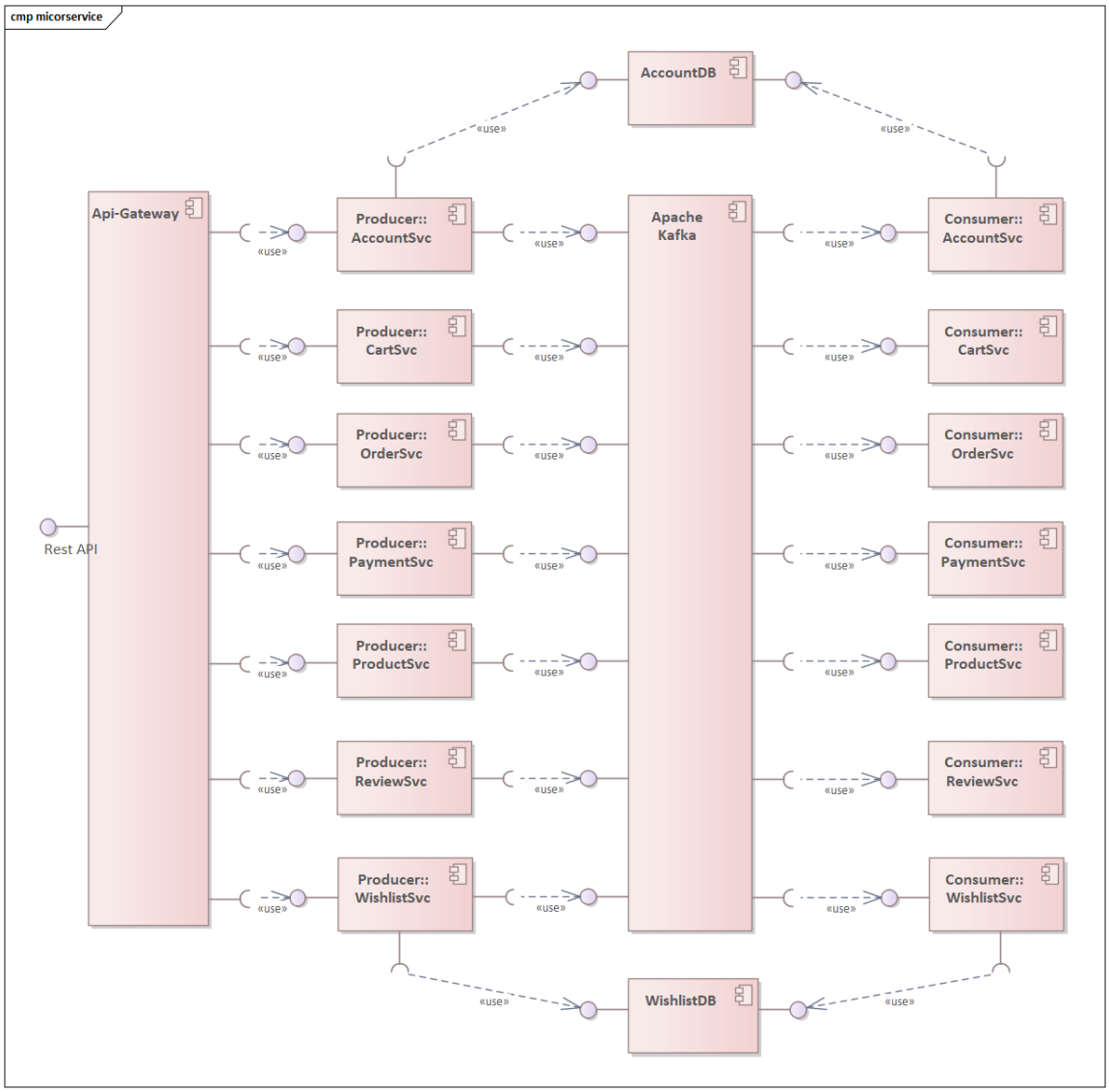


Obrázek 3.3: Diagram komponent pro monolitní aplikaci

3.3.2 Mikroservisní architektura

Aplikace se skládá z 5 hlavních podčástí (Obrázek 3.4). Api-gateway, producenti, kafka, konzumenti a databáze. Api-gateway slouží jako vstupní bod do našeho systému. Jednotlivé mikroslužby, které jsou napojeny přímo na Api-gateway, zpracovávají požadavek a vrací na něj odpověď zpět. Při zpracování mohou tyto služby (producenti) vytvořit event a poslat jej do kafky. Kafka je třetí částí, ta je využita k asynchronní komunikaci napříč jednotlivými službami. Poslední částí jsou konzumenti, ti naslouchají na eventy vytvořené producenty a vykonávají časově náročné funkce v databázi. Každá mikroslužba sdílí databázi pro producenta i konzumenta (na Obrázku 3.4 znázorněno pouze u AccountSvc a WishlistSvc z důvodu přehlednosti diagramu). Producent z databáze čte, zatímco konzument do ní zapisuje. Důvod rozkladu jednotlivých mikroslužeb na producent a konzument je škálovatelnost. Tato architektura nám dovoluje volně škálovat ty komponenty, které zrovna potřebujeme. Api-gateway se skládá z controlleru, ten přijímá HTTP požadavky a přeposílá jej na servisní vrstvu (Obrázek 7.3). Ze servisní vrstvy jde požadavek na agregator, který má v sobě uložené adresy všech

mikroslužeb a požadavek přepošle pomocí RestTemplate na zodpovědnou službu. Struktura jednotlivých mikroslužeb je obdobná monolitní architektuře Obsahuje kontrolní vrstvu, servisní vrstvu a repository vrstvu. Zásadní rozdíl je zde v tom, že každá mikroslužba disponuje vlastní databází.



Obrázek 3.4: Diagram komponent pro mikroservisní architekturu

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. Pro přiblížení funkcionality systému byly vytvořeny tři

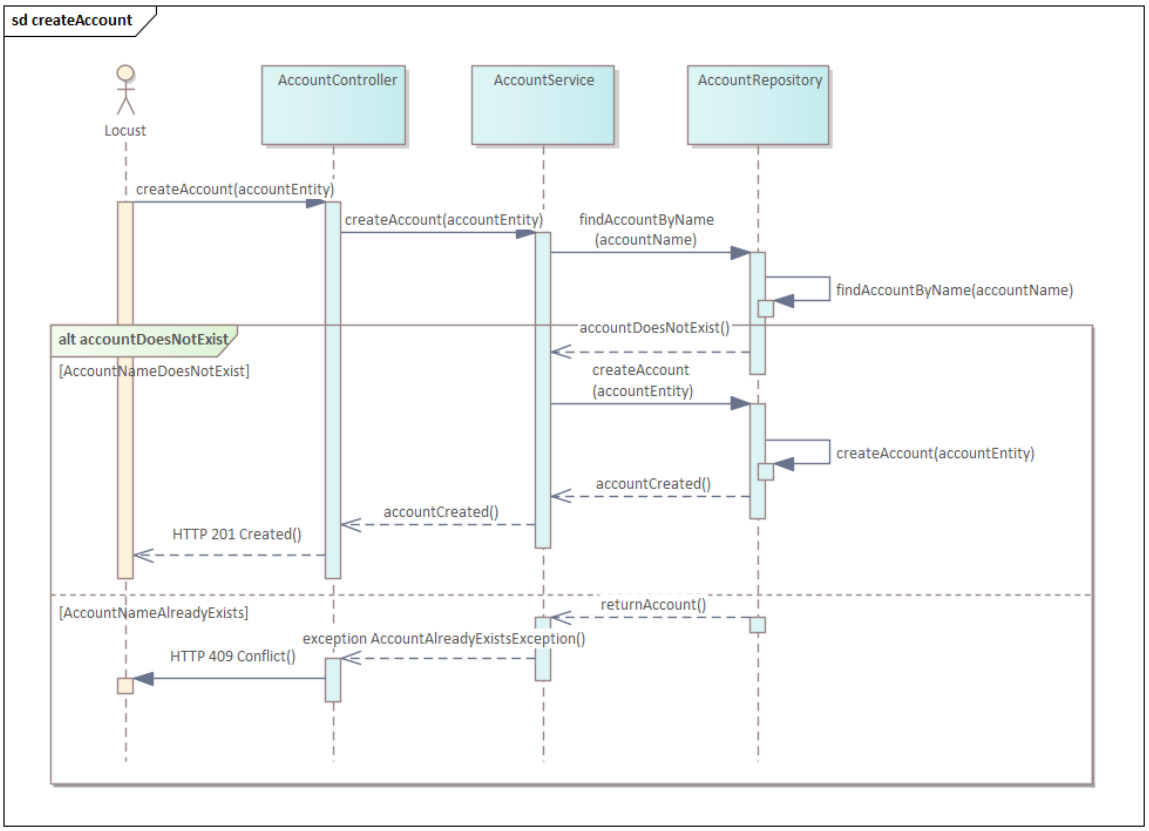
scénáře pro aplikaci. Vytvoření účtu, vytvoření objednávky (checkout košíku) a zaplacení objednávky.

3.4.1 Vytvoření účtu

Vytvoření účtu zahrnuje vytvoření dalších 2 entit a to cartEntity a wishlistEntity. V monolitní aplikaci je toto výrazně jednodušší, jelikož proces ukládání do databáze vyřeší Hibernate. V mikroservisní architektuře je konzistence dat řešena pomocí SAGA Choreography patternu.

Monolitní architektura

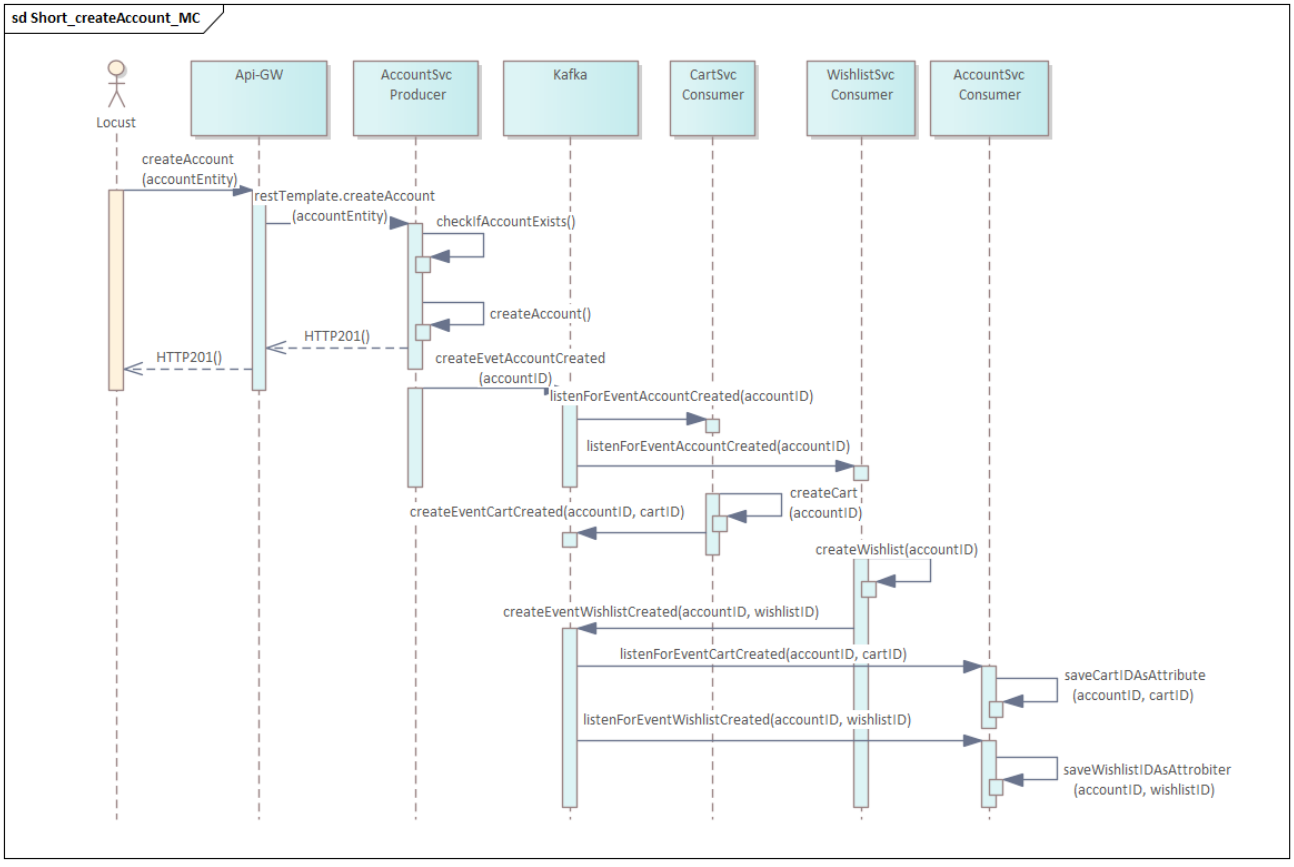
Požadavek vytvořit účet je předán na servisní vrstvu, ta zkontroluje, zda-li již neexistuje účet se stejným uživatelským jménem. Pokud účet neexistuje, vytvoří účet nový (zároveň s ním i entity cart a wishlist, to se děje na úrovni Entity, takže se o to servisní vrstva už nemusí starat). V tomto případě je na klienta vrácen HTTP status 201. V opačném případě servisní vrstva vyhodí exception, že takový účet již existuje a na klienta se vrací HTTP 409 (viz Obrázek 3.5).



Obrázek 3.5: Sekvenční diagram createAccount pro monolitn

Mikroservisní architektura

Požadavek na vytvoření účtu přijde na API-gateway. Odtud pomocí agregátoru je poslán RestTemplate požadavek na API Account service. Zde proběhne kontrola, jestli takový účet již neexistuje. Pokud ano, je vrácen HTTP 409, což již na diagramu není znázorněno kvůli přehlednosti. Pokud takový účet neexistuje, repository vrstva vytvoří nový účet a systém vrací HTTP 201. Současně v tuto chvíli servisní vrstva vytváří v kafce topic "accountCreated"a zasílá k němu ID vytvořeného účtu. Na tento event naslouchají dva konzumenti a to CartService a WishlistService. Každý si vytvoří danou entitu, uloží si ID účtu, ke kterému patří, a poté opět vytvoří topic v kafkce, že byl vytvořen košík resp. wishlist. Do topicu vloží ID účtu a ID vytvořené entity. Na tyto dva eventy už opět naslouchá accountServiceConsumer. Ten obdrží ID účtu a dané vytvořené entity a uloží si jej jako atribut (viz Obrázek 3.6). Celá tato část vytváření košíku a wishlistu probíhá čistě asynchronně a díky tomu je doba trvání požadavku drasticky zkrácena. Detailnější diagram s rozepsanými komponenty lze najít mezi přílohami jako Obrázek 7.6.



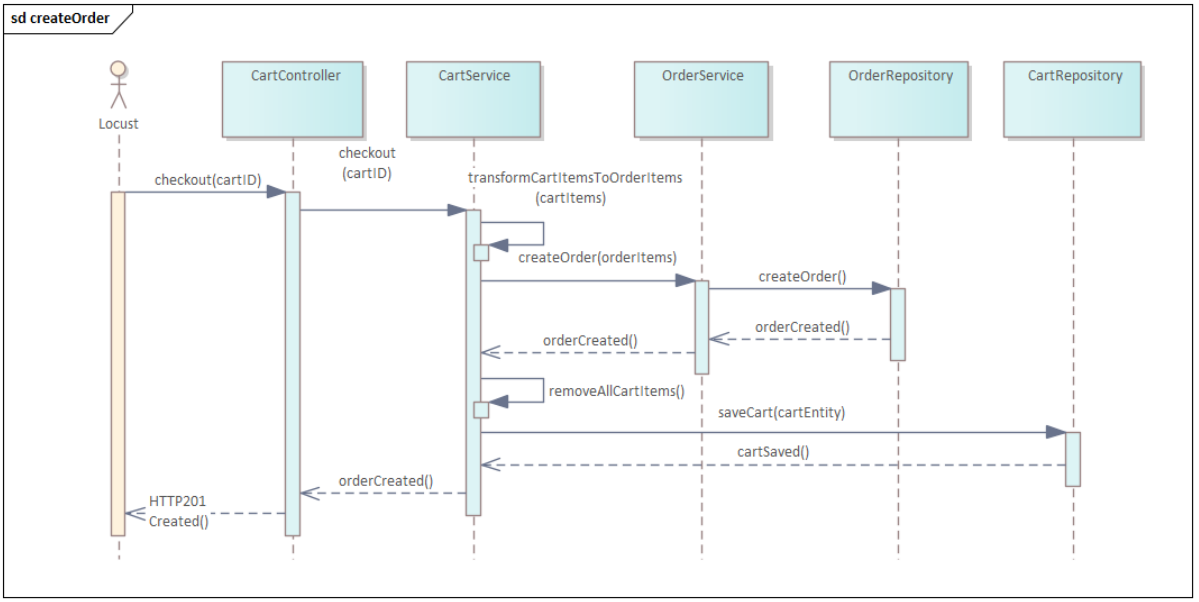
Obrázek 3.6: Sekvenční diagram createAccount pro mikroslužby zkrácený

3.4.2 Vytvoření objednávky

Vytvoření objednávky je proces, při kterém jsou všechny položky z košíku transformovány na položky objednávky. Konzistence databáze u mikroservisní architektury je opět zajištěna pomocí saga petternu.

Monolitní architektura

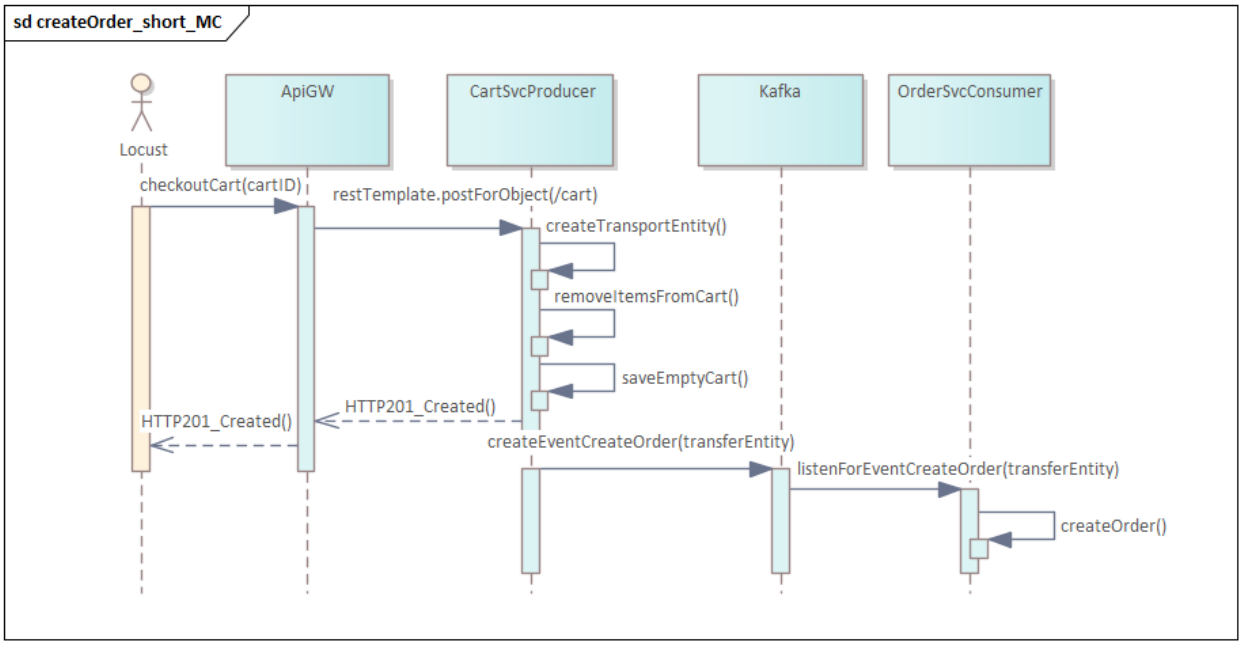
Požadavek je zachycen na CartControlleru, který jej pošle na servisní vrstvu. Zde proběhne transformace všech položek košíku na položky objednávky. Poté je poslán požadavek na OrderService, který objednávku vytvoří. Následně CartService odstraní veškeré položky z košíku a uloží prázdný košík. Na konci transakce je vrácen HTTP201. (viz Obrázek 3.7)



Obrázek 3.7: Sekvenční diagram createOrder pro monolitní architekturu

Mikroservisní architektura

Požadavek je přeposlán na zodpovědnou servisní vrstvu dané mikroslužby, v tomto případě CartService. Zde servisní vrstva transformuje všechny produkty v košíku na transportníEntitu, kterou pošle do kafky. Mezitím odstraní všechny položky z košíku, prázdný košík uloží a vrací HTTP201. OrderServiceConsumer zachytí topic v kafce a na základě itemů, co zaslal cartService, vytvoří objednávku (viz Obrázek 3.8). Detailnější přehled procesu je mezi přílohami jako Obrázek 7.5.



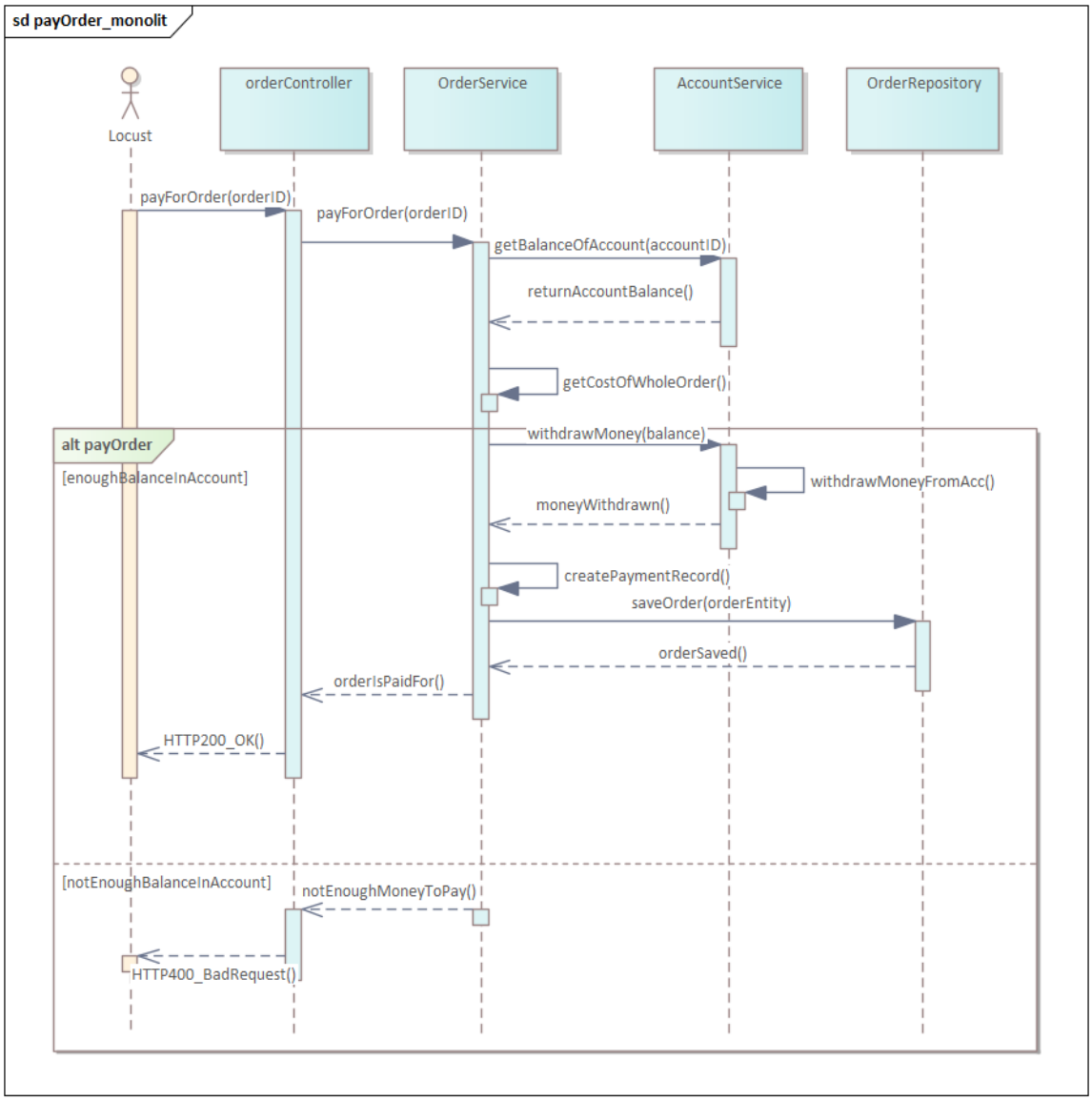
Obrázek 3.8: Zkrácený sekvenční diagram createOrder pro mikroservisní architekturu

3.4.3 Zaplacení objednávky

Zaplacení objednávky probíhá tak, že se zkontroluje stav účtu, pokud je větší nebo roven celkové ceně objednávky, je tato suma stržena z účtu a vytvoří se záznam o zaplacení.

Monolitní architektura

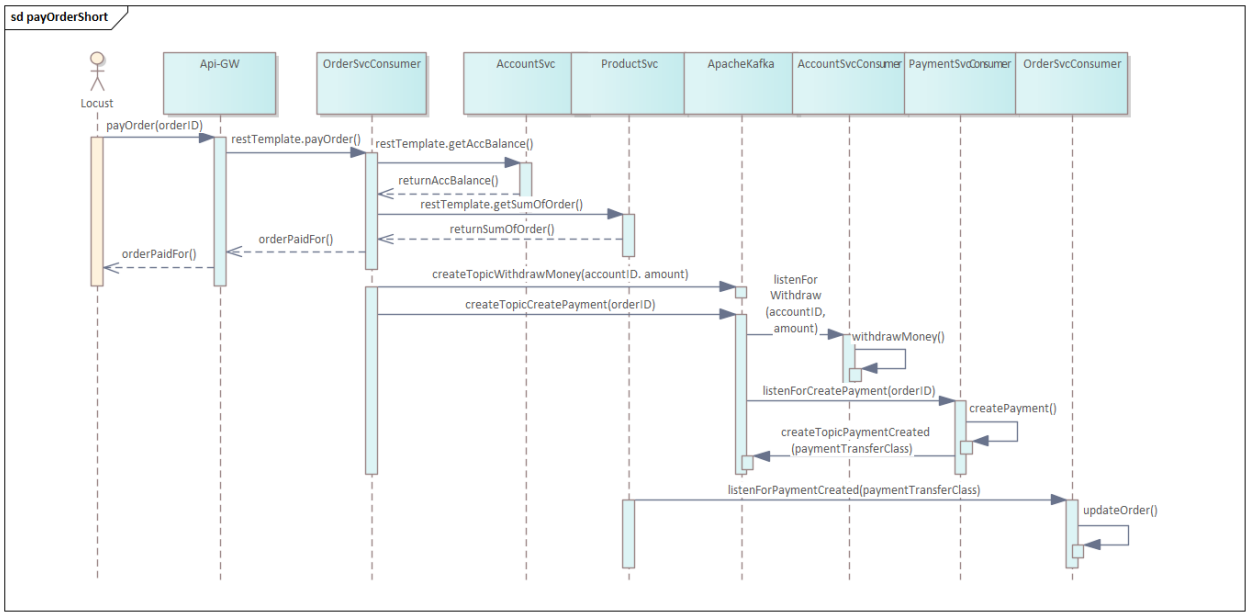
Požadavek přijde na orderController, ten zavolá orderService. OrderService zjistí stav účtu, sečte sumu za objednávku a pokud je na účtu dostatek peněz, peníze strhne a vytvoří PaymentEntitu. Poté uloží objednávku i se záznamem o zaplacení. Pokud na účtu není dostatek peněz, systém vrací HTTP400 (viz Obrázek 3.9).



Obrázek 3.9: Sekvenční diagram payOrder pro monolitní architekturu

Mikroservisní architektura

Obrázek 3.10 popisuje proces placení objednávky. Požadavek je zachycen API-gateway a odeslán na order-service. Order service pomocí RestTemplate synchronně zavolá AccountMicroservice, aby zjistil stav účtu klienta, ke kterému je objednávka vázána. Poté opět pomocí pomocí RestTemplate zavolá ProductMicroservice, aby zjistil celkovou cenu za objednávku. Pokud nemá dostatek financí, systém vrací chybu HTTP400 (na diagramu není znázorněno z důvodu přehlednosti). Pokud má dostatek financí, odpoví klientovi a zároveň pošle dva topicy do kafky, withdrawMoney a createPayment. AccountServiceConsumer zachytí zprávu a z účtu strhne stanovenou částku. PaymetServiceConsumer zachytí zprávu, že má být vytvořena entita Payment, služba tuto entitu vytvoří, uloží si ID objednávky, a opět pošle do kafky zprávu o vytvoření platby. OrderServiceConsumer tuto zprávu zachytí a k danému objektu objednávky přiřadí ID platby.



Obrázek 3.10: Sekvenční diagram payOrder pro mikroservisní architekturu

Kapitola 4

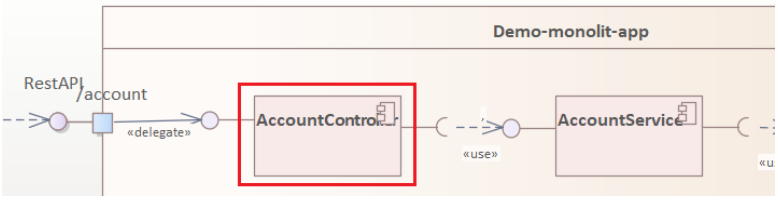
Implementace

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. Na Rest controlleru byly definovány jednotlivé endpointy, popsané v analýze endpointů, které dále odkazují na servisní vrstvu. Pokud servisní vrstva vyhodí exception, controller vrátí chybový stav. Pokud vše proběhne v pořádku, vrací stav na základě operace, která byla volána. Například pro vytvoření účtu je HTTP201-Created. Pokud účet již existuje, servis vyhodí HTTP 409-Conflict. Ukázka takového kontroleru je na Obrázku 4.2. V mikroservisní architektuře je zároveň vhodné využívat DTO objekty, které jasně definují input a output z kontrolní vrstvy.

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



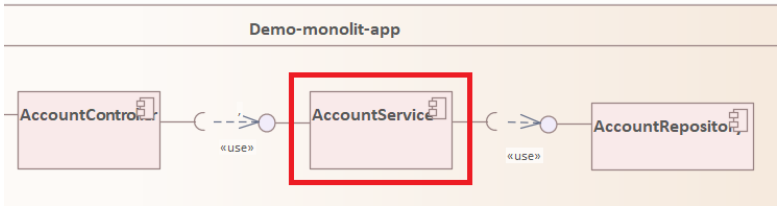
Obrázek 4.1: Account controller z diagramu komponent

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

    @Autowired
    public AccountController(AccountService accountService) { this.accountService = accountService; }

    @PostMapping
    public ResponseEntity<?> createAccount(@RequestBody AccountEntity account) {
        try {
            accountService.createAccount(account);
            return ResponseEntity.status(HttpStatus.CREATED).build();
        } catch (AccountAlreadyExistsException e) {
            return ResponseEntity.status(HttpStatus.CONFLICT).body(e.getMessage());
        }
    }

    @GetMapping
    public ResponseEntity<List<AccountEntity>> getAllAccounts(){
        List<AccountEntity> allAccounts = accountService.getAllAccounts();
        return ResponseEntity.ok(allAccounts);
    }

    @GetMapping("/byName/{accountName}")
    public ResponseEntity<AccountEntity> getAccountByName(@PathVariable String accountName){
        try {
            AccountEntity account = accountService.getAccount(accountName);
            return ResponseEntity.ok(account);
        } catch (EntityNotFoundException e) {
```

Obrázek 4.2: Ukázka kódu z account controlleru

4.2 Servisní vrstva

Servisní vrstva zodpovídá za logiku systému. Zde jsou největší rozdíly mezi monolitní a mikroservisní architekturou. Monolitní architektura má výhodu, že konzistenci dat napříč tabulkami v databázi řeší Hibernate (implementace JPA). V mikroservisní architektuře konzistenci dat zajišťuje Saga pattern, takže když servisní změna provede v databázi takovou změnu, která ovlivní jiné mikroslužby, je vytvořen topic v kafce a ostatní služby na tento topic reagují. Rozdíl v kódu můžeme vidět na Obrázku 4.4 a Obrázku 4.5. Obě aplikace zkontrolují, jestli může být účet vytvořen, pokud ano, tak ho uloží do databáze. Mikroservisní aplikace ještě vytvoří topic v kafce, aby na jeho základě mohli reagovat ostatní mikroslužby.



Obrázek 4.3: Account service z diagramu komponent

```java
/**
 * Implementation of the AccountService interface.
 */
@Service
@AllArgsConstructor
@Slf4j
public class AccountServiceImpl implements AccountService {
    private AccountRepository accountRepository;

    /**
     * Creates a new account.
     *
     * @param account The account entity to create.
     * @throws AccountAlreadyExistsException If an account with the same ID already exists.
     */
    @Override
    public void createAccount(AccountEntity account) {
        if (account.getId() == null || accountRepository.findById(account.getId()).isEmpty()) {
            accountRepository.save(account);
        } else {
            throw new AccountAlreadyExistsException("Account with this ID already exists.");
        }
    }
```

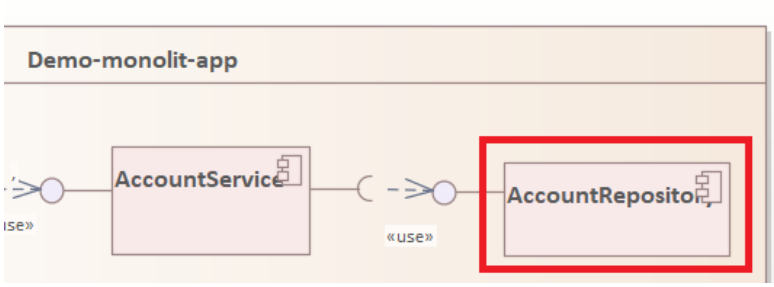
Obrázek 4.4: Ukázka kódu z account service monolitní aplikace

```java
/**
 * Implementation of the AccountService interface.
 */
@Service
@AllArgsConstructor
@Slf4j
public class AccountServiceImpl implements AccountService {
    private AccountRepository accountRepository;
    private MyKafkaProducer kafkaProducer;

    /**
     * Creates a new account.
     *
     * @param account The account entity to create.
     * @throws AccountAlreadyExistsException If an account with the same ID already exists.
     */
    @Override
    public void createAccount(AccountEntity account) {
        if (account.getId() == null || accountRepository.findById(account.getId()).isEmpty()) {
            if (account.getCartEntity() == null){
                accountRepository.save(account);
                kafkaProducer.accountCreated(account.getId());
            }else {
                accountRepository.save(account);
            }
        } else {
            throw new AccountAlreadyExistsException("Account with this ID already exists.");
        }
    }
```

Obrázek 4.5: Ukázka kódu z account service mikroservisní aplikace

4.3 Repository vrstva

Zodpovědnost repository vrstvy je v komunikaci s databází. Obě aplikace využívají interface JpaRepository, který se stará o základní queries databáze. Další možností je využitím DAO objektů, nebo nařetězením DAO a repository, v našem případě bylo ovšem využito pouze repository. Pro přidání dalších queries lze využít buď možnosti zadání parametrů funkce a nechat třídu JpaRepository přeložit náš požadavek jako databázové query, nebo vytvořit si query vlastní (viz Obrázek 4.7). Při vytváření repository zadáme, o jakou entitu se jedná a co je její ID. Funkce findByUsername je definovaná tak, že ji vložíme String a očekávaný výsledek je AccountEntity. Query je automaticky vytvořena knihovnou. Pokud je potřeba vytvořit vlastní query, ukázka je ve funkci incrementId na Obrázku 4.7.



Obrázek 4.6: Account repository z diagramu komponent

```java
public interface AccountRepository extends JpaRepository<AccountEntity, Integer> {

    AccountEntity findByUsername(String username);

    @Query(value = "SELECT MAX(id) + 1 FROM AccountEntity")
    Integer getNextAccountId();

    @Modifying
    @Transactional
    @Query(value = "UPDATE AccountEntity SET id = ?1 WHERE id = ?1 - 1")
    void incrementId(Integer id);

    default Integer getAndIncrementNextAccountId() {
        Integer nextId = getNextAccountId();
        incrementId(nextId);
        return nextId;
    }


}
```

Obrázek 4.7: Ukázka kódu z account repository

4.4 Modelová vrstva

Obrázek 4.8 popisuje modelovou vrstvu, která reprezentuje byznys entity systému. Každý atribut třídy reprezentuje atribut z datového modelu (viz Obrázek 3.1). @Id a @GeneratedValue nad atributem Id značí, že při vytvoření nové entity je automaticky vytvořeno nové ID pro tento objekt. Takže pokud pomocí repository je do databáze uložen objekt třídy Account, kde ID je null, po uložení se ID automaticky zvolí na hodnotu, kterou mu Hibernate přidělí. Kvůli tomu např. při vytváření účtu u mikroservisní architektury je nejdříve nutné uložit nový účet a až potom vytvořit topic kafkou, abychom získali ID nově vytvořeného objektu. @Column(unique = true) značí, že každý účet musí mít unikátní uživatelské jméno, kdyby to tak nebylo, funkci findByUsername na Obrázku 4.7 by nešlo splnit a tudíž by vracela exception. Relace mezi entitami se řeší pomocí anotací @OneToOne, @OneToMany, atd. Při pokusu získat Json reprezentaci entity, která má oboustranný vztah s další entitou, tak by se náš požadavek zacyklil a systém by vrátil StackOverflow, jelikož jedna entita má jako atribut druhou a druhá zase první. Tento problém se dá řešit buď využitím Dto objektů, nebo pomocí @JsonManagedReference a @JsonBackReference, jak jde vidět na Obrázku 4.8. Mikroservisní architektura si drží referenci na objekty pomocí jejich ID, takže wishlistEntity a cartEntity jsou reprezentovány jako Integer (viz Obrázek 4.9). @ElementCollection umožňuje vytvořit list základních typů.

```java
@Entity
@Data
@AllArgsConstructor
public class AccountEntity {
    public AccountEntity(){
        this.cartEntity = new CartEntity();
        this.wishlist = new WishlistEntity();
        this.orders = new ArrayList<>();
        this.balance = 0.0;
    }

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer id;
    @Column(unique = true)
    private String username;
    private String password;
    private String email;
    private String cityName;
    private String street;
    private Double balance;

    @JsonManagedReference
    @OneToOne(cascade = CascadeType.ALL)
    private CartEntity cartEntity;

    @JsonManagedReference
    @OneToMany(cascade = CascadeType.ALL,mappedBy = "account", fetch = FetchType.EAGER)
    private List<OrderEntity> orders;

    @JsonManagedReference
    @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    private WishlistEntity wishlist;
```

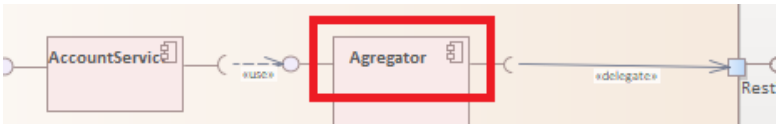
Obrázek 4.8: Ukázka kódu z account entity pro monolit

```java
@Entity
@Data
@AllArgsConstructor
public class AccountEntity {
    public AccountEntity() {
        this.orders = new ArrayList<>();
        this.balance = 0.0;
    }

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer id;

    @Column(unique = true)
    private String username;
    private String password;
    private String email;
    private String cityName;
    private String street;
    private Double balance;

    private Integer wishlist;
    private Integer cartEntity;

    @ElementCollection(fetch = FetchType.EAGER) // Changed the fetch type to EAGER
    @CollectionTable(name = "account_orders")
    private List<Integer> orders;
```

Obrázek 4.9: Ukázka kódu z account entity pro mikroservisní architekturu

4.5 Agregator

Tato komponenta se nachází pouze v mikroservisní architektuře, a to jako nejnižší vrstva API gateway (viz Obrázek 4.10). Zodpovídá za vybrání správného endpointu a komunikaci s ním. Má v sobě uložený způsob, jak najít všechny ostatní mikroslužby. Na základě volání ze servisní vrstvy komunikuje synchronně s ostatními mikroslužbami pomocí RestTemplate (viz Obrázek 4.9). Pokud by aplikace běžela pouze na lokálním počítači, adresy jednotlivých mikroslužeb by byly definovány jako na Obrázku 4.11. V kubernetes je ovšem situace složitější, takže místo localhost je v adrese název service v kubernetes (viz Obrázek 4.12). Díky tomu je i uvnitř kubernetes clusteru schopna api-gateway komunikovat s ostatními mikroslužbami.



Obrázek 4.10: 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 4.11: Ukázka kódu z agregatoru v api gateway

```java
@Component
@AllArgsConstructor
@Slf4j
public class Aggregator {
    private RestTemplate restTemplate;
    private final String baseAccountUrl = "http://account-svc-service:8081/account";
    private final String baseCartUrl = "http://cart-svc-service:8083/cart";
    private final String baseOrderUrl = "http://order-service:8084/order";
    private final String basePaymentUrl = "http://payment-service:8085/payment";
    private final String baseProductUrl = "http://product-service-service:8082/product";
    private final String baseReviewUrl = "http://review-service-service:8086/review";
    private final String baseWishlistUrl = "http://wishlist-service-service:8087/wishlist";

    // ------------------------------------ Account service ---------------------------------

    public ResponseEntity<AccountDto> getAccountByName(String accountName) {
        String url = baseAccountUrl + "/byName/" + accountName;
        return restTemplate.getForEntity(url, AccountDto.class);
    }

    public ResponseEntity<AccountDto> getAccount(Integer accountId) {
        String url = baseAccountUrl + "/" + accountId;
        return restTemplate.getForEntity(url, AccountDto.class);
```

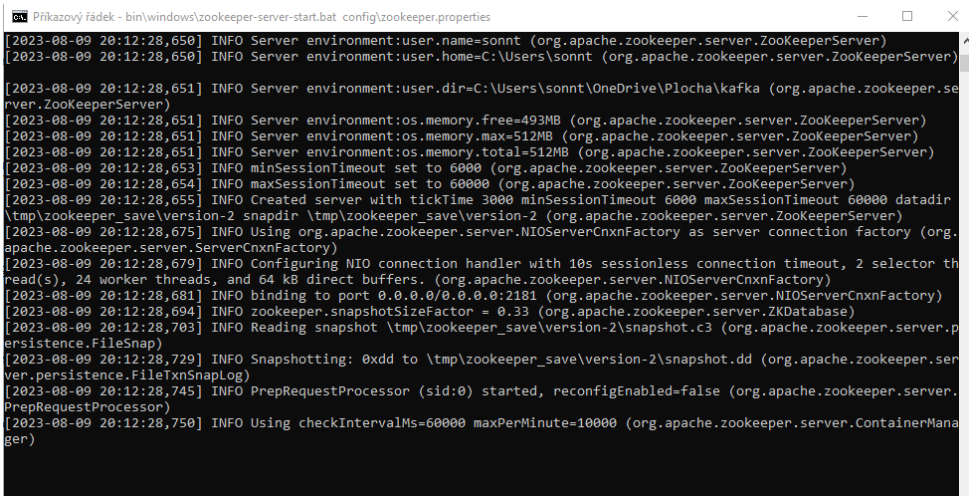
Obrázek 4.12: Agregatoru v api gateway pro nasazení do kubernetes

4.6 Kafka

Pro implementaci systému byla využita Kafka verze 2.0. Spuštění proběhlo v cmd na lokálním počítači. Prvně je nutno zapnout zookeeper, ten se spouští pomocí příkazu na Obrázku 4.13. Správné spuštění vypadá jako na Obrázku 4.14. Dalším krokem je spuštění brokerů. Příkaz na spuštění brokera je uveden na Obrázku 4.15 a jak vypadá jeho správné spuštění lze vidět na Obrázku 4.16.

```
>bin\windows\zookeeper-server-start.bat config\zookeeper.properties
```

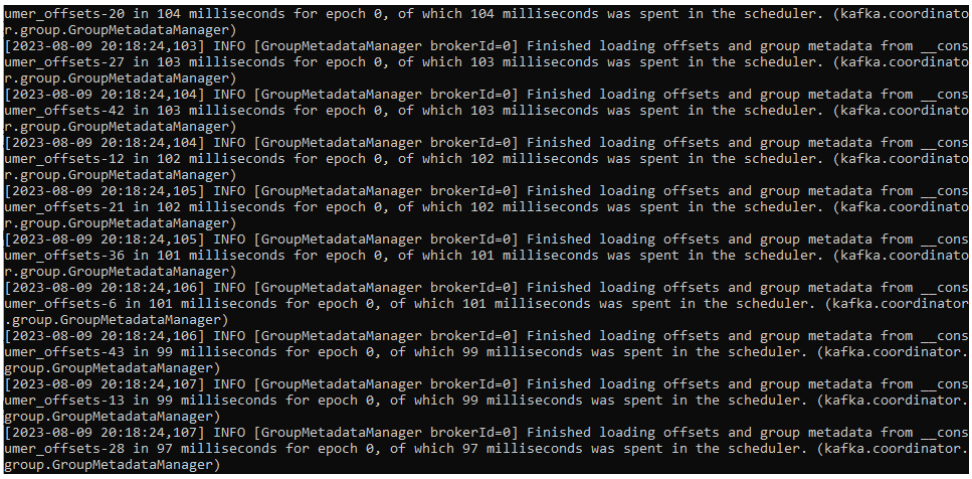
Obrázek 4.13: Příkaz pro spuštění zookeper



Obrázek 4.14: Správně spuštění zookeper

```
>bin\windows\kafka-server-start.bat config\server.properties
```

Obrázek 4.15: Příkaz pro spuštění brokera



Obrázek 4.16: Správně spuštěný broker

4.6.1 Kafka Producer

Kafka producer je třída zodpovědná za vytváření topiců. V konfigurace KafkaTemplate lze nastavit možnost posílání různých druhů tříd, zde bylo využito posílání pouze Stringů a Integrů. Integer byl poslán v případě, kdy do zprávy stačilo poslat pouze ID nějaké entity. String byl využit v složitějším případě, kdy bylo potřeba přenést více atributů, a pro tyto případy byly vytvořeny Transportní třídy, které jsou prvně převedeny do Json formátu a poté poslané jako jeden String. Přiklad, jak vypadá vytvoření topicu pro

vytvoření účtu, lze vidět na Obrázku 4.17 u funkce accountCreated.

```
@Component
public class MyKafkaProducer{
    private final KafkaTemplate<String, String> kafkaTemplate;

    @Autowired
    public MyKafkaProducer(KafkaTemplate<String, String> kafkaTemplate) { this.kafkaTemplate = kafkaTemplate; }

    public void accountCreated(Integer accountID){
        kafkaTemplate.send( topic: "accountCreated",accountID.toString());
    }

}
```

Obrázek 4.17: Kafka producer

4.6.2 Kafka Consumer

Kafka konzument naslouchá na vytvoření topicu, na Obrázku 4.18 je to topic s názvem "accountCreated". Ve chvíli, kdy je topic vytvořen nějakým producentem, je spuštěna funkce accountWasCreated, která vytvoří nový košík a pošle zpět do Kafky novou zprávu o tom, že byl košík vytvořen.

```
@Component
public class MyKafkaListener {
    private final CartService cartService;
    private final MyKafkaProducer kafkaProducer;

    @Autowired
    public MyKafkaListener(CartService cartService, MyKafkaProducer kafkaProducer) {
        this.cartService = cartService;
        this.kafkaProducer = kafkaProducer;
    }

    @KafkaListener(topics = "accountCreated", groupId = "cart_group_id")
    public void accountWasCreated(Integer accountID){
        CartEntity cart = cartService.createCart(accountID);
        CartCreatedResponse cartCreatedResponse = new CartCreatedResponse(accountID, cart.getCartId());
        kafkaProducer.cartWasCreated(cartCreatedResponse);
    }
}
```

Obrázek 4.18: Kafka consumer

4.6.3 Transfer class

V případě, že je pomocí Kafky nutno poslat do topicu složitější entity, je využito transfer tříd. Tyto třídy si drží důležité informace, které potřebujeme dostat do jiné mikroslužby. V situaci, kdy je vytvořen nákupní košík, je potřeba do Account mikroslužby zaslat jak ID nového košíku, tak ID účtu (viz atributy na Obrázku 4.19), pro který byl vytvořen, ale služba přiřadila košík správnému účtu. Tato třída musí mít prázdný konstruktor a nastavené gettery a settery na všechny atributy, aby šla převést na Json pomocí knihovny jackson. Implementace transofrmace třídy na Json a zpět lze vidět na Obrázku 4.20.

```java
@NoArgsConstructor
public class CartCreatedResponse {
    private Integer accountId;
    private Integer cartId;

    public CartCreatedResponse(Integer accountId, Integer cartId) {
        this.accountId = accountId;
        this.cartId = cartId;
    }

    public Integer getAccountId() { return accountId; }

    public void setAccountId(Integer accountId) { this.accountId = accountId; }

    public Integer getCartId() { return cartId; }

    public void setCartId(Integer cartId) { this.cartId = cartId; }
}
```

Obrázek 4.19: CartCreated transfer class

```java
public class JsonSerializer {
    private final ObjectMapper obj = new ObjectMapper();

    public String transferClassToJson(Object o) throws JsonProcessingException {
        return obj.writeValueAsString(o);
    }

    public CartCreatedResponse departmentFromJson(String json) throws JsonProcessingException {
        return obj.readValue(json,CartCreatedResponse.class);
    }

    public WithdrawMoneyResponse withdrawFromJson(String json) throws  JsonProcessingException {
        return obj.readValue(json,WithdrawMoneyResponse.class);
    }

    public OrderCreatedResponse orderFromJson(String json) throws JsonProcessingException {
        return obj.readValue(json,OrderCreatedResponse.class);
    }

    public WishlistCreatedResponse wishlistFromJson(String json) throws  JsonProcessingException {
        return obj.readValue(json,WishlistCreatedResponse.class);
    }
}
```

Obrázek 4.20: Json Serializer třída

4.6.4 Důležité poznámky

Při využití kafky je důležité dát si pozor na možnost, že dva různé asynchronní procesy mohou zasahovat do atributů jedné entity současně. Tento příklad lze vidět na procesu vytváření uživatelského účtu (viz Obrázek 3.6). Po vytvoření košíku a wishlistu jsou vytvořeny topicy, které ukládají ID vytvořeného objektu do objektu daného účtu. Zde hrozí nebezpečí, že tyto dvě zprávy dorazí do systému současně, oba procesy si načtou entitu z databáze, entitu pozmění a následně ji uloží. Pokud jeden proces načte entitu z databáze, dřív, než ji druhý proces do databáze uloží, jsou změny prvního procesu ztraceny, jelikož je druhý proces přeuloží. V našem případě se dělo to, že při vytvoření účtu k němu byl přiřazen pouze buď wishlist, nebo cart (podle toho, která message proběhla v kafce rychleji). Tento problém se řešil pomocí synchronizace vláken aplikace těchto dvou funkcí (viz Obrázek 4.21).

```
@KafkaListener(topics = "wishlistCreated", groupId = "account_group_id")
public void wishlistWasCreated(String  jsonCartCreatedResponse){
    try {
        synchronized (threadLocker){
            WishlistCreatedResponse wishlistCreatedResponse = jsonSerializer.wishlistFromJson(jsonCartCreatedResponse);

            Integer accountID = wishlistCreatedResponse.getAccountID();
            Integer wishlistID = wishlistCreatedResponse.getWishlistID();

            AccountEntity account = accountService.getAccount(accountID);
            account.setWishlist(wishlistID);
            accountService.updateAccount(accountID,account);
        }
    } catch (JsonProcessingException e) {
        e.printStackTrace();
    }
}
```

Obrázek 4.21: Thread safe funkce

4.7 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.

4.7.1 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 (viz Obrázek 4.21) začneme využívat již vytvořené

HTTP spojení. To znamená, že spojení nemusí být neustále otevíráno, čímž ušetříme čas. [21]

```
@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 4.22: Konfigurace RestTemplate

4.7.2 Konfigurace Tomcat

Výchozí nastavení tomcatu je takové, že maximální počet vláken je nastaven na 200. [22] Díky jeho navýšení na 600 (viz Obrázek 4.22) jsme zvedli maximální počet požadavků za vteřinu z 2800 na 2900 (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 4.23: Konfigurace RestTemplate

4.8 Docker Image

Pro vytvoření docker image byl použit Dockerfile, který lze vidět na Obrázku 4.23. K vytvoření image je ještě nutné vytvořit .jar soubor. Ten lze vytvořit pomocí příkazu v terminálu "mvn clean package". Poté co je vytvořena jar file, lze vytvořit image pomocí příkazu: "docker build -t <nazev.tagu>:verze .". Tímto přikazem se v dockeru vytvoří Image aplikace.

Image postavené v dockeru není volně dostupná pro kubernetes. Pro využití image v kubernetes je potřeba buď nahrát na DockerHub, ze kterého si danou

image stáhne, nebo před vytvořením image přepnout terminál do prostředí minikube pomocí příkazu "minikube docker-env | Invoke-Expression". Až poté zadat příkaz docker build, to způsobí, že se image nepostaví v Docker Desktop aplikaci jako obvykle, ale místo toho bude vložena přímo do Minikube. Pro tento přístup je nutné v yaml souboru nastavit imagePullPolicy: Never. Pro tento projekt bylo využit Docker Hubu. Image se prvně vloží do Docker Desktopu pomocí příkazu docker build. Poté, co je image uložena lokálně, lze použít příkaz docker push, který uloží Image v Docker Hubu a kubernetes si ho bude moci stáhnout. Při vytváření kontejneru je důležité dát pozor na komunikaci s ostatními komponenty aplikace.

Při využití odkazu na adresu localhost uvnitř kontejneru, izolace jednotlivých kontejnerů způsobí to, že http://localhost neodkazuje na localhost počítače, na kterém kontejner běží, ale na localhost uvnitř kontejneru. Proto je potřeba, pokud má systém využívat komponenty mimo kontejner, změnit adresu z localhost na host.docker.internal. V případě tohoto systému to bylo nutné provést u referency na kafku, která je spuštěna lokálně mimo Docker (viz Obrázek 4.25).

```
FROM openjdk:17-jdk-slim as 🔨 builder
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} application.jar
RUN java -Djarmode=layertools -jar application.jar extract

FROM openjdk:17-jdk-slim
COPY --from=builder dependencies/ ./
COPY --from=builder spring-boot-loader/ ./
COPY --from=builder snapshot-dependencies/ ./
COPY --from=builder application/ ./
ENTRYPOINT ["java", "org.springframework.boot.loader.JarLauncher"]
```

Obrázek 4.24: Dockerfile

```
spring.kafka.bootstrap-servers=host.docker.internal:9092
```

Obrázek 4.25: Reference na kafku v dependecies

4.9 Kubernetes

Minikube je nástroj pro vytváření jednoduchých, lokálních Kubernetes cluseterů. Jako první krok je nutné zapnout minikube cluster, díky kterému je možné ovládat kubernetes. Minikube lze spustit, pokud je zapnutý docker, příkazem "minikube start". Následně pomocí definovaného souboru .yaml lze vytvořit deployment a service. Na Obrázku 4.25 lze vidět příklad deployment.yaml AccountMicroservice. Je zde definován jak deployment, tak service.

Počet replik je nastaven na 1, v případě, že by bylo potřeba tuto službu vyškálovat, toto číslo by se zvětšilo. Dále lze vidět, že image do containeru se stahuje z DockerHubu sonntdav00. Název služby "account-svc-service"je velmi důležitý, díky němu tento servis lze nalézt v Kubernetes clusteru (viz Obrázek 4.12). Dále existuje možnost přiřadit jednotlivým kontejnerům výpočetní sílu v podobě paměti a CPU. Pro vytvoření deployment a service na základě tohoto souboru, stačí do terminálu napsat: "kubectl apply -f deployment.yaml". Pro zpřístupnění jednotlivých služeb se musí do terminálu napsat "minikube service –<nazev sluzby>". Popřípadě minikube serice –all, pokud je potřeba mít přístup z lokálního počítače k jednotlivým mikroslužbám. Po provedení tohoto příkazu kubernetes přiřadí jednotlivým službám na localhostu port definovaný v .yaml složce jako "nodePort".

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: account-svc-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: account-svc
  template:
    metadata:
      labels:
        app: account-svc
    spec:
      containers:
        - name: account-svc
          image: sonntdav00/account_svc:1.0
          ports:
            - containerPort: 8081
          imagePullPolicy: Always
          resources:
            requests:
              memory: "256Mi"
              cpu: "100m"
            limits:
              memory: "512Mi"
              cpu: "500m"
---
apiVersion: v1
kind: Service
metadata:
  name: account-svc-service
spec:
  selector:
    app: account-svc
  ports:
    - protocol: TCP
      port: 8081
      targetPort: 8081
      nodePort : 30001
  type: NodePort
```

Obrázek 4.26: deployment.yaml soubor

4.10 Implementace jednotlivých design patternů

4.10.1 Database per service

Tento pattern byl implementován tak, že každá mikroslužba má svou vlastní databázi. Všechny mikroslužby disponují H2 databází. Její vlastnosti jsou popsány v application.properties, které lze vidět na Obrázku 4.26.

```
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.url=jdbc:h2:mem:TEST;DB_CLOSE_DELAY=-1;MODE=Oracle;DEFAULT_LOCK_TIMEOUT=10000;LOCK_MODE=0;DB_CLOSE_ON_EXIT=FALSE
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

spring.datasource.hikari.maximum-pool-size=600
spring.datasource.hikari.idle-timeout=180
spring.datasource.hikari.connection-timeout=10000
```

Obrázek 4.27: Resilience4j v Pom.xml

4.10.2 Saga

Pro tento systém byla zvolena implementace saga petternu v podobě Choreography. Důvodem je, že v tomto systému neprobíhá příliš velké množství transakcí, tudíž není příliš náročné kontrolovat konzistenci dat a není nutné vytvářet další službu, co by spravovala všechny transakce. Funkci message brokeru z Obrázku 2.1 plní kafka, ta zajišťuje, že se jednotlivé zprávy dostanou z jedné služby do druhé. Pokaždé, když jedna mikroslužba provede transakce, pro kterou je nutné upravit stav ostatních služeb, tak daná mikroslužba vytvoří topic, díky kterému dá vědět ostatním službám, jakou operaci je potřeba udělat, aby byla udržena konzistence dat.

4.10.3 Circuit breaker

Circuit breaker byl implenetován pomocí knihovny resilience4J (viz Obrázek 4.27). Nyní lze k metodám na controlleru přidat anotaci @CircuitBreaker a k němu přidat název metody. V případě, že tato metoda nestíhá a circuit breaker se otevře, namísto původní funcke bude zavolaná tato callback metoda. Na Obrázku 4.28 lze vidět implementace CircuitBreaker na metodě getAllAcounts. Ve chvíli, kdy systém přestane stíhat, bude zavolaná metoda fallbackGetAllAccounts. Obrázek 4.29 ukazuje nastavení circuit breakeru v souboru application.yaml.

```xml
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-spring-boot2</artifactId>
    <version>2.0.2</version>
</dependency>
```

Obrázek 4.28: Resilience4j v Pom.xml

```java
@Autowired
public AccountController(AccountService accountService) { this.accountService = accountService; }

@CircuitBreaker(name = "getAllAccount",fallbackMethod = "fallbackGetAllAccounts")
@GetMapping
public ResponseEntity<List<AccountEntity>> getAllAccounts(){
    try {
        List<AccountEntity> allAccounts = accountService.getAllAccounts();
        return ResponseEntity.ok(allAccounts);
    } catch (EntityNotFoundException e) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(null);
    }
}
public String fallbackGetAllAccounts(Throwable throwable){
    return "Circuit breaker stopped this request with" + throwable.getMessage();
}
```

Obrázek 4.29: Ukázka circuit breakeru na controllerul

```
resilience4j.circuitbreaker:
  configs:
    default:
      registerHealthIndicator: true
      slidingWindowSize: 10
      minimumNumberOfCalls: 5
      permittedNumberOfCallsInHalfOpenState: 3
      automaticTransitionFromOpenToHalfOpenEnabled: true
      waitDurationInOpenState: 5s
      failureRateThreshold: 50
      eventConsumerBufferSize: 10
```

Obrázek 4.30: Nastavení circuit breakeru v application.yaml

4.10.4 Bulkhead

Implementace bulkhead patternu v tomto systému probíhá díky kubernetes. To dovoluje přiřadit jednotlivým částem aplikace paměť a cpu (viz Obrázek 4.31), díky čemuž je zaručeno, že jedna služba nevypotřebuje veškerou výpočetní sílu celému systému.

```
resources:
  requests:
    memory: "256Mi"
    cpu: "100m"
  limits:
    memory: "512Mi"
    cpu: "500m"
```

Obrázek 4.31: Ukázka rozložení resaurces

4.10.5 Agregator

Agregator byl implementován v podobě API-gateway, která slouží jako vstupní bod aplikace a agreguje jednotlivé požadavky uživatele. Díky tomu není nutné volat API jednotlivých mikroslužeb, ale stačí zavolat pouze jeden na API-gateway (viz Obrázek 3.4).

4.11 Locust

Jednotlivé endpointy, které chceme testovat pomocí locustu, se musí definovat v python kódu (viz Obrázek 4.26). 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 4.27). 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 4.32: Ukázka kódu z Locustu

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

Obrázek 4.33: Spuštění Locustu přes konzoli

Kapitola 5
Testování

Obě aplikace, jak monolitní, tak asynchronní, byly podrobeny jednotkovým, integračním a end-to-end testům. Unit testy byly provedeny na servisní vrstvě. Integrační testy byly implementovány mezi všemi vrstvami. End to end testy proběhly pomocí RestTemplate a voláním na REST API. Celkem bylo pro tento projekt vytvořena 142 testů pro obě aplikace a bylo využito knihoven jako Mockito, Junit a Jupiter.

Pro účely zátěžového 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. Obě aplikace byly testovány s 92% line coverage na servisní vrstvě a 70% na controlleru.

5.1 Unit testy

Pro unit testy bylo využito frameworku Mockito, který umožňuje simulovat chování ostatních komponent systému, aniž bychom byli závislí na jejich správné funkcionalitě. Na Obrázku 5.1 lze vidět příklad takového testu.

```
@Test
void updateProduct_existingProduct_savesUpdatedProduct() {
    // Arrange
    ProductEntity originalProduct = new ProductEntity();
    originalProduct.setProductID(1);
    ProductEntity updatedProduct = new ProductEntity();
    updatedProduct.setProductID(1);
    when(productRepository.findById(1)).thenReturn(Optional.of(originalProduct));

    // Act
    productService.updateProduct(updatedProduct);

    // Assert
    verify(productRepository, times( wantedNumberOfInvocations: 1)).save(updatedProduct);
}
```

Obrázek 5.1: Unit test

5.2 Integrační testy

Integrační testy kontrolují správné chování napříč komponentami. Na Obrázku 5.2 je vidět příklad Integračního testu, který kontroluje správný chování servisní třídy a její integraci s databází. Zároveň je zde vidět příklad využití knihovny Jupiter k rekurzivní kontrole identity dvou objektů.

```
@Test
public void creatingAccountSavesCorrectlyInDatabase(){
    AccountEntity account = getAccount();
    accountService.createAccount(account);
    AccountEntity result = accountService.getAccount(account.getUsername());
    assertThat(account)
            .usingRecursiveComparison()
            .isEqualTo(result);
}
```

Obrázek 5.2: Integrační test

5.3 End to end testy

End to end testy kontrolují aplikaci jako celek. Pomocí RestTemplate je prováděno volání na controller. Pomocí POST metod jsou objekty vytvářeny a upravovány. Následně pomocí GET metody se lze ujistit, že objekt byl modifikován úspěšně. Pro přiblížení testů je zde příklad několika scénářů, které byly vytvořeny pro testování aplikace. Vytvoření účtu automaticky přiřadí účtu jak košík, tak wishlist. Vytvoření produktu, přidání recenze k němu. Přidání tohoto produktu do wishlistu uživatele. Přidání tohoto produktu do košíku. Následný checkout košíku. Zaplacení objednávky, co byla vytvořena checkoutem košíku. Na Obrázku 5.3 lze vidět ukázka testu, kde jsou vytvořeny dva produkty a následná kontrola, že oba v systému jsou.

```
ResponseEntity<Void> createProductResponse = restTemplate.postForEntity(productUrl,product,Void.class);
ResponseEntity<Void> createProductResponse2 = restTemplate.postForEntity(productUrl,product2,Void.class);

assertEquals(HttpStatus.CREATED, createProductResponse.getStatusCode());
assertEquals(HttpStatus.CREATED, createProductResponse2.getStatusCode());

product = restTemplate.getForEntity( url: productUrl + "/byName/" + product.getName(),ProductEntity.class).getBody();
product2 = restTemplate.getForEntity( url: productUrl + "/byName/" + product2.getName(),ProductEntity.class).getBody();

assert product != null;
assert product2 != null;
```

Obrázek 5.3: EndToEnd test

5.4 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]

5.4.1 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]

5.4.2 Distribuované testy

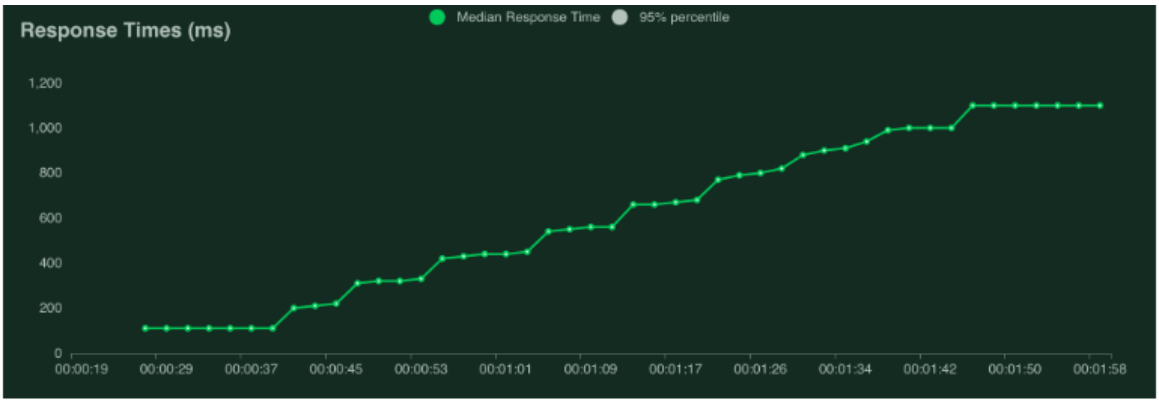
Distribuované testování pomocí Locustu umožňuje dosáhnout ještě vyššího výkonu a zátěže na testovaném systému. Při použití více instancí Locustu současně je možné simulovat vysoký počet požadavků za sekundu. Master instance koordinuje a řídí testování, zatímco worker instance vykonávají požadavky na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

5.4.3 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) se již nezvyšuje rovnoměrně s počtem uživatelů. Vzorový příklad je na Obrázku 5.3, 5.4 a 5.5. Všechny tři grafy mají sdílenou osu x, na které je čas. Obrázek 5.3 popisuje růst počtu požadavků za vteřinu, který náš server dokáže odbavit. Ve chvíli, kdy toto číslo přestane přestane růst rovnoměrně s počtem uživatelů, co vytváří locust (Obrázek 5.6), tak byl nalezen zlomový bod systému. Na Obrázku 5.5 lze vidět, že i když se počet požadavků již nemění, doba, za kterou server odpovídá, stále roste. Třída fastHttpUser má timeout nastaven na 60 s, ve chvíli, kdy požadavek na server trvá déle, je požadavek vyhodnocen jako

error. Na vzorovém testu lze vidět, že když se počet uživatelů dostane okolo čísla 9 (cca 40 s na ose x), response time serveru roste tak rychle, že server přestane stíhat a limit je cca 38RPS. Tímto způsobem byly vyhodnoceny testy pro všechny scénáře zátěžového testování. [20]

Obrázek 5.4: Vzorový locust test - RPS [20]



Obrázek 5.5: Vzorový locust test - Response time [20]

Obrázek 5.6: Vzorový locust test - Number of users [20]

5.4.4 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 nastavením počtu vláken na 600.

5.4.5 Nastavení testů

Pro testování systému bylo využito jak třídy FastHttpUser, tak využití vícevláknového testování. Každý test proběhl s 1 masterem a 5 workerky, aby bylo zaručeno, že výsledek testu není omezen locustem, ale testovaným systémem. Následně byly testy provedeny v několika kategoriích. První kategorie je rovnoměrné rozložení testované zátěže mezi všechny testované endpointy systému. Pro zátěžové testy bylo využito pouze metod GET, jelikož H2 databáze nedokázala při složitějších příkazech vydržet nápor testů a stávala se bottleneckem systému. Tento způsob testování tedy ukazuje výkon systému, nikoliv pouze jeho databáze. Druhá kategorie je ztrojnásobení zátěže na jedné mikroslužbě a třetí kategorie je pětinásobek této zátěže. Důvod tohoto druhu testování je takový, že v reálném nasazení není zátěž na všechny komponenty stejná, v čemž je výhody mikroservisní architektury, jelikož může díky virtualizaci na základě zátěže rozmístit výpočetní sílu dle potřeby, zatímco monolitní aplikace má všechnu výpočetní sílu na jednom místě.

5.4.6 Podmínky testování

Systém byl testován na v kubernetes Windows, což běží díky WSL (technologie ke spuštění linuxových spustitelných souborů na windows). To omezilo maximální zdroje systému na 7500 MB a 10 CPU. Na základě testování bylo zjištěno, že aby pod běžel bez problému, potřebuje cca 0.5 CPU a 0.5 GB RAM. Při nižších hodnotách se stávalo, že daný padal a celý systém byl nestabilní. To znamená, že maximum podů, co lze vytvořit pro systém, je $7500/500 = 15$. Monolit měl vždy přiřazen 1 pod, který měl k dispozici veškerý výkon systému. U mikroservisní architektury bylo minimum využitých podů 8 (7 mikroslužeb + api gateway). Tím zbývá maximum 7 podů, které lze rozložit napříč systém.

5.4.7 Výsledky testování

První kategorie testování je rovnoměrné rozložení zátěže mezi endpointy. Druhá kategorie rozděluje zátěž tak, že na account microservice chodí třikrát tolik požadavků než na ostatní služby. Třetí kategorie má na account service zátěž pětinásobnou. Výsledek RPS nelze přesně změřit, jelikož tato křivka

není konstantní a obvykle osciluje okolo nějakého čísla. Pro zaznamenání výsledků se počítal výsledek jako zaokrouhlené číslo, okolo kterého křivka RPS oscilovala, aniž by toto číslo dále rostlo.

Rovnoměrné rozložení zátěže

Monolitní aplikace při tomto testu dosáhla výsledku okolo 3800 RPS, průběh celého testu lze vidět na Obrázku 7.7, který je mezi přílohami. Výsledek mikroservisní architektury je v tomto případě 3400 RPS. Důvod výrazně horšího výsledku mikroservisní architektury v tomto případě je zejména v tom, že průmerně požadavek zde trvá mnohem déle a je složitější, jelikož zde probíhá komunikace mezi jednotlivými službami, která je mnohem pomalejší, než komunikace uvnitř jednoho systému. Průběh testu mikroservisní architektury lze vidět na Obrázku 7.8.

Trojnásobná zátěž

V tomto testu byla zátěž rozložena tak, že na account mikroslužbu bylo posíláno v průměru 3x tolik požadavků. Monolitní aplikace měla výsledné RPS okolo 3600 (viz Obrázek 7.9). U mikroservisní architektury je v tomto případě důležité rozložení podů. Normálně by se tato situace řešila pomocí technologie, jako je například Grafana, co dokáže monitorovat výkon a zátěž jednotlivých mikroslužeb. V našem případě na základě pokusů bylo zjištěno, že nejvyšší výkon lze získat. když na API-Gateway připadnou 4 pody a Account-Microservice 3 pody. Jak jde vidět na Obrázku 7.10, tak výkon se od původního výsledku téměř nezměnil a finální výsledek se pohyboval okolo 3350 RPS.

Pětinásobná zátěž

Průběh testu monolitní architektury z Obrázku 7.11 ukazuje, že RPS se pohybuje okolo hodnoty 3000, což je o 600 méně než při minulém testu. Rozložení podů mikroservisní architektury zde je o něco složitější než v minulém případě. Pokud bychom chtěli vyškálovat api-gateway i account microservice pětinásobný výkon, tak bychom každému museli přidělit 5 podů. Což v našem případě nelze z důvodu hardwarových omezení, které byly zmíněny v kapitole podmínky testování. Z tohoto důvodu bylo rozložení podů pro tento test takové, že API-GW dostala 5 podů a account service 4 pody. Výsledek tohoto testu byl 3200 RPS.

shrnutí

Při vyrovnané zátěži na systém byl lepší výsledek jednoznačně na straně monolitní architektury. Takový výsledek lze očekávat, jelikož veškerá činnost probíhá uvnitř jedné aplikace a komunikace mezi komponentami u mikroslužeb proces vyřizování požadavku značně brzdí. Ve chvíli, kdy se začala zvedat zátěž na pouze na jedné části aplikace, monolitní architektura začala ztrácet výkon.

To je z důvodu, že monolit nelze škálovat libovolně a spousta výpočetní síly byla využita tam, kde nebyla potřeba. Na druhou stranu výsledek u mikroslužeb zůstal ve všech třech případech téměř stejný navzdory tomu, že poslední testovací scénář nebylo možné naškálovat pody. jak bylo potřeba. Tento test jasně ukazuje výhodu ve škálovatelnosti mikroservisní architektury nad monolitní. Ve chvíli, kdy zátěž na jednu mikroslužbu byla pětinásobná, tak mikroservisní architektura měla lepší výsledek, než monolitní. Jednotlivé výsledky jsou zaznamenány v tabulce na Obrázku 5.7.

Výsledky zátěžových testů	**Monolit [RPS]**	**Mikroslužby [RPS]**
rovnoměrné rozložení	3800	3400
trojnásobná zátěž na AccSvc	3600	3350
pětinásobná zátěž na AccSvc	3000	3200

Obrázek 5.7: Výsledky zátěžových testů

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 komunikaci. V praktické části byly využity všechny design patterny z teoretické části a byl popsán způsob jejich implementace. Z technologií k asynchronní komunikaci byla v praktické části blíže představena a prakticky využita Apache Kafka.

Při návrhu systému byl nejprve vytvořen datový model, který sloužil jako základ pro další analýzu. Na základě tohoto diagramu byla provedena analýza endpointů, na jejíž základě byly definovány endpointy systému. Dalším krokem byl diagram komponent, kde byla detailně představena architektura obou systémů. V případě mikroservisní architektury byla využita implementace proxy patternu a agregatoru pomocí api-gateway. Komunikace mezi jednotlivými mikroslužbami byla realizována pomocí RestTemplate. Pro tři vybrané scénáře, tj. vytvoření účtu, vytvoření objednávky a její následné zaplacení byly vytvořeny sekvenční diagramy, pro bližší představení procesů uvnitř aplikace.

V implementační části byla nejprve na základě návrhu systému postavena monolitní a mikroservisní architektura. Do mikroservisní architektury byly implementovány všechny design patterny zmíněné v teoretické části práce. Pro mikroservisní architekruru byla implementována do systému Kafka, která sloužila pro asynchronní zasílání zpráv. Obě aplikace byly podrobně otestovány pomocí unit, integračních a end to end testů. Následně byla vytvořena Docker Image obou aplikací, která poté byla nahrána do Kubernetes a v ní následně testována pomocí Locustu.

Při zátěžovém testování měla při rovnoměrném rozložení testů navrch monolitická architektura, ale ve chvíli, kdy se rozložení zátěže na systém začalo měnit, mikroservisní architektura si díky škálovatelnosti udržela téměř stejné výsledky, zatímco monolitní rychle odpadávala.

Budoucí práce

Možnost, jak navázat na tuto aplikaci, je nasazení na reálný systém, kde by byla možnost lépe škálovat obě architektury. Dále by bylo možné popsat a využít více technologií, které se užívají při vývoji mikroservisních architektur, jako je Grafana, díky které by bylo jednodušší škálování aplikace. Nebo by šlo také využít Helm Chart, což je nástroj pro orchestraci kontejnerů. Díky této technologii by šlo lépe konfigurovat aplikace jak z pohledu proměnných, tak pro kubernetes objekty.

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.

23. Microsoft: Publisher subscribe pattern
URL: <https://learn.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber >[cit. 12.08.2023]

24. Docker: Overview
URL: <https://docs.docker.com/get-started/overview/>[cit. 9.08.2023]

25. Kubernetes: Components overview
URL: <https://kubernetes.io/docs/concepts/overview/components//intro>[cit. 9.08.2023]

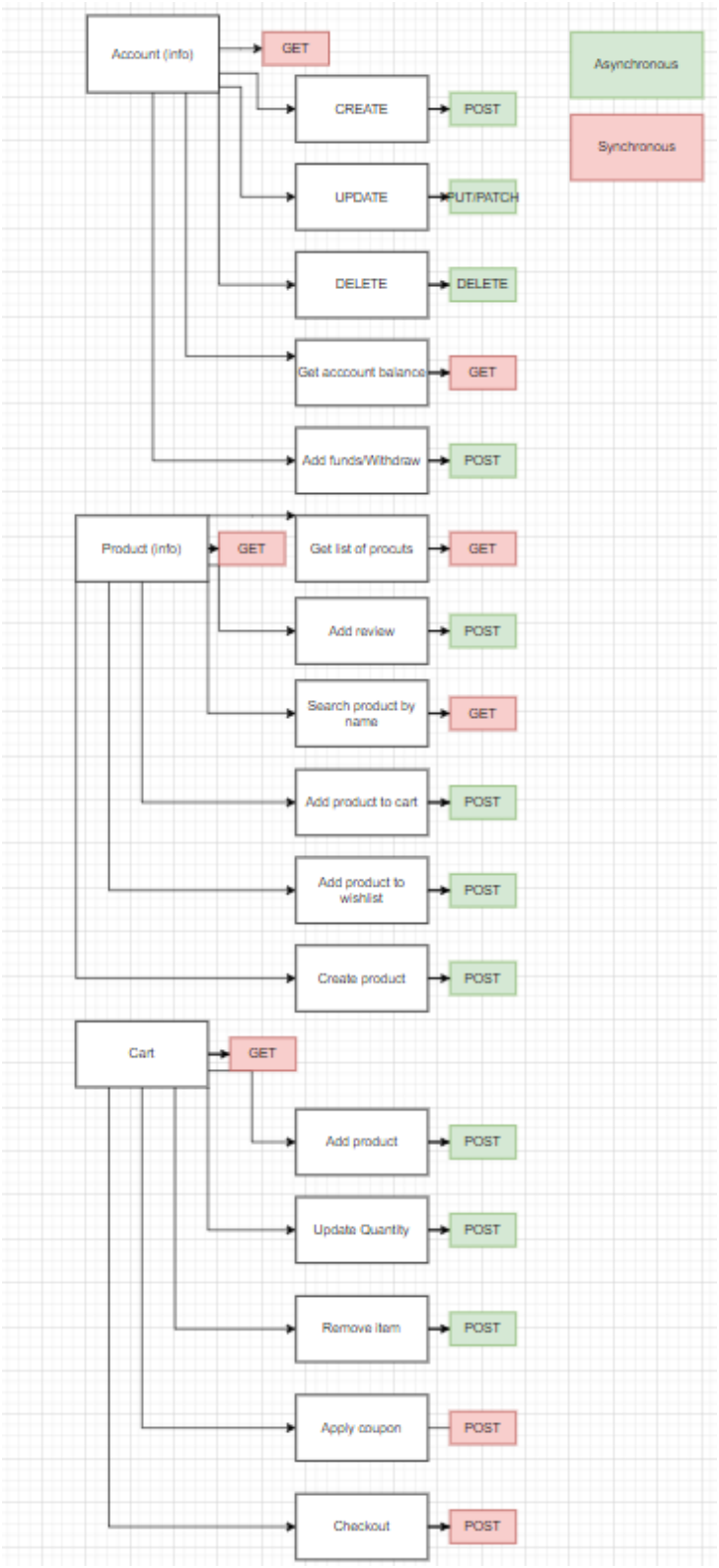
26. Microsoft: Saga pattern
URL: <https://learn.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga>[cit. 9.08.2023]

27. ApacheKafka: Inrto
URL: <https://kafka.apache.org/intro>[cit. 7.08.2023]

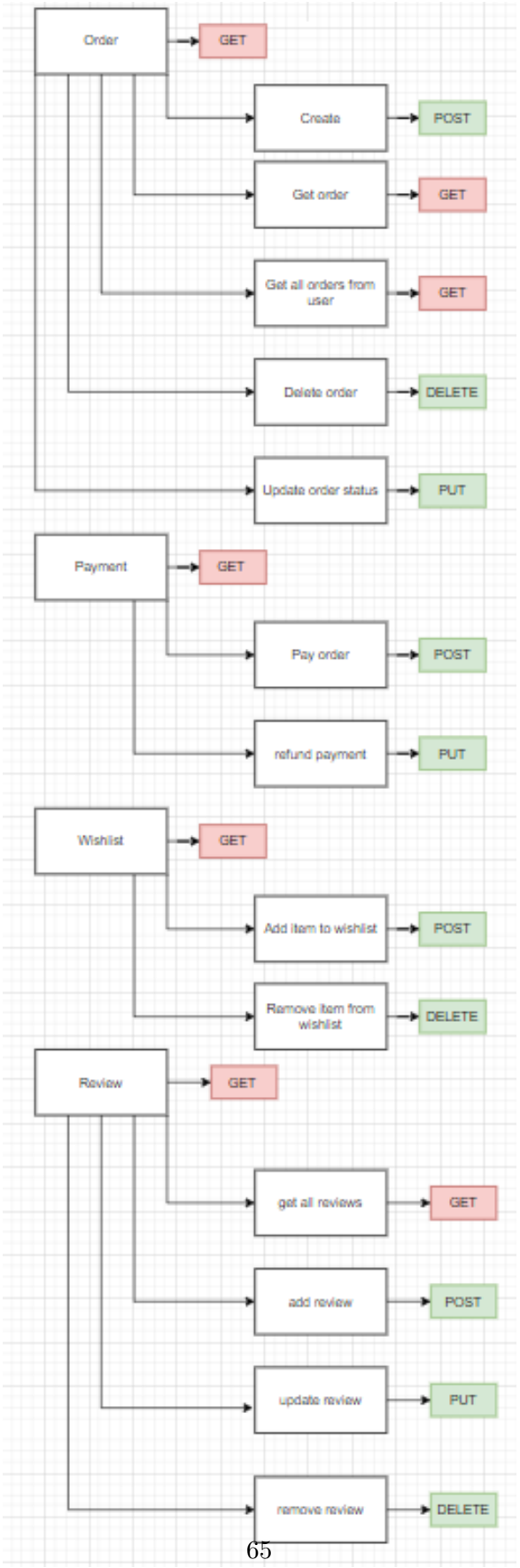
28. Docker: Seznámení s dockerem
URL: <https://b2a.cz/2021/02/seznameni-s-docker/ >[cit. 14.08.2023]

29. Kubernetes workshop
URL: <https://cms-opendata-workshop.github.io/workshop2022-lesson-introcloud/aio/index.html/ >[cit. 14.08.2023]

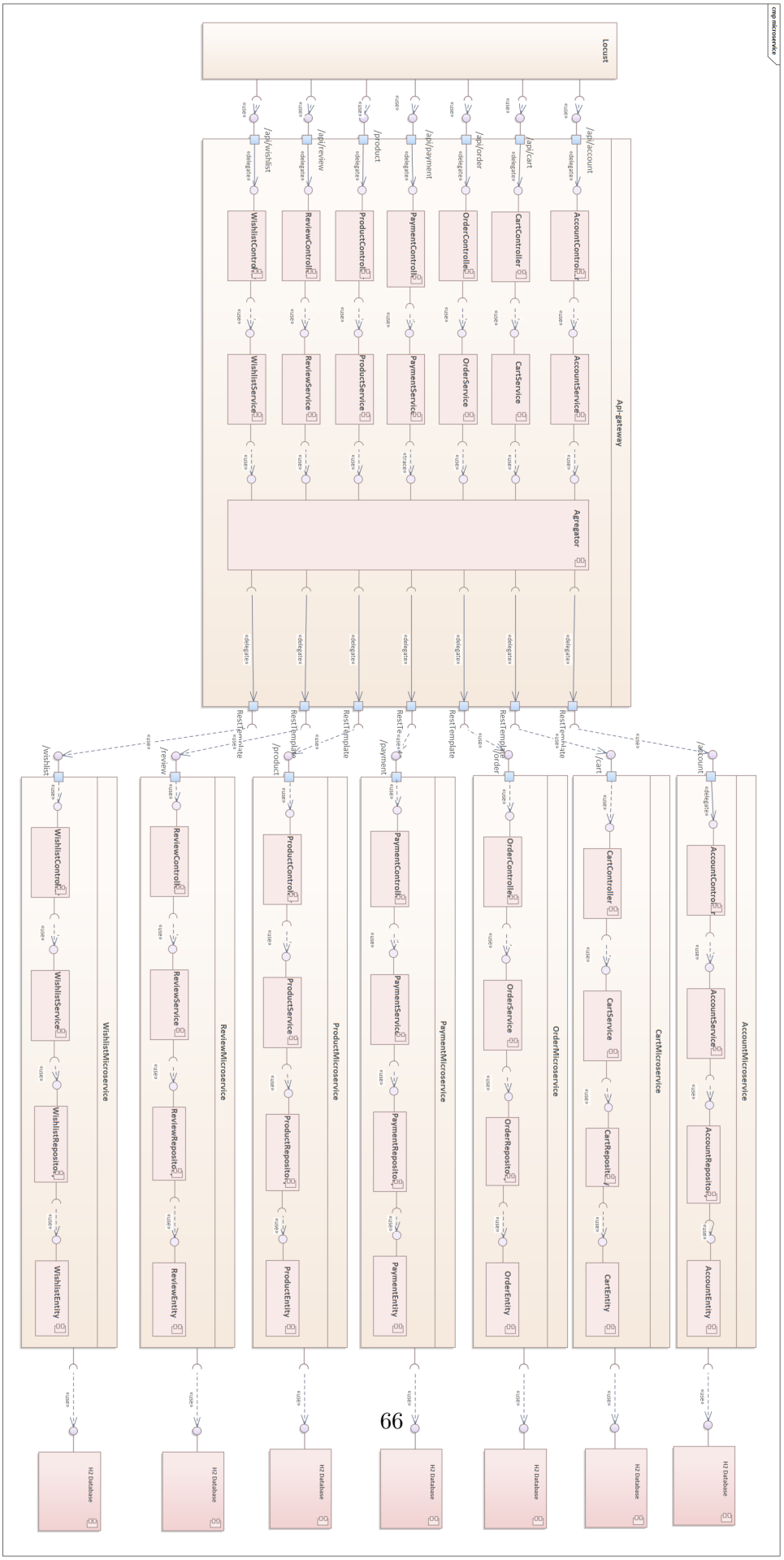
Příloha



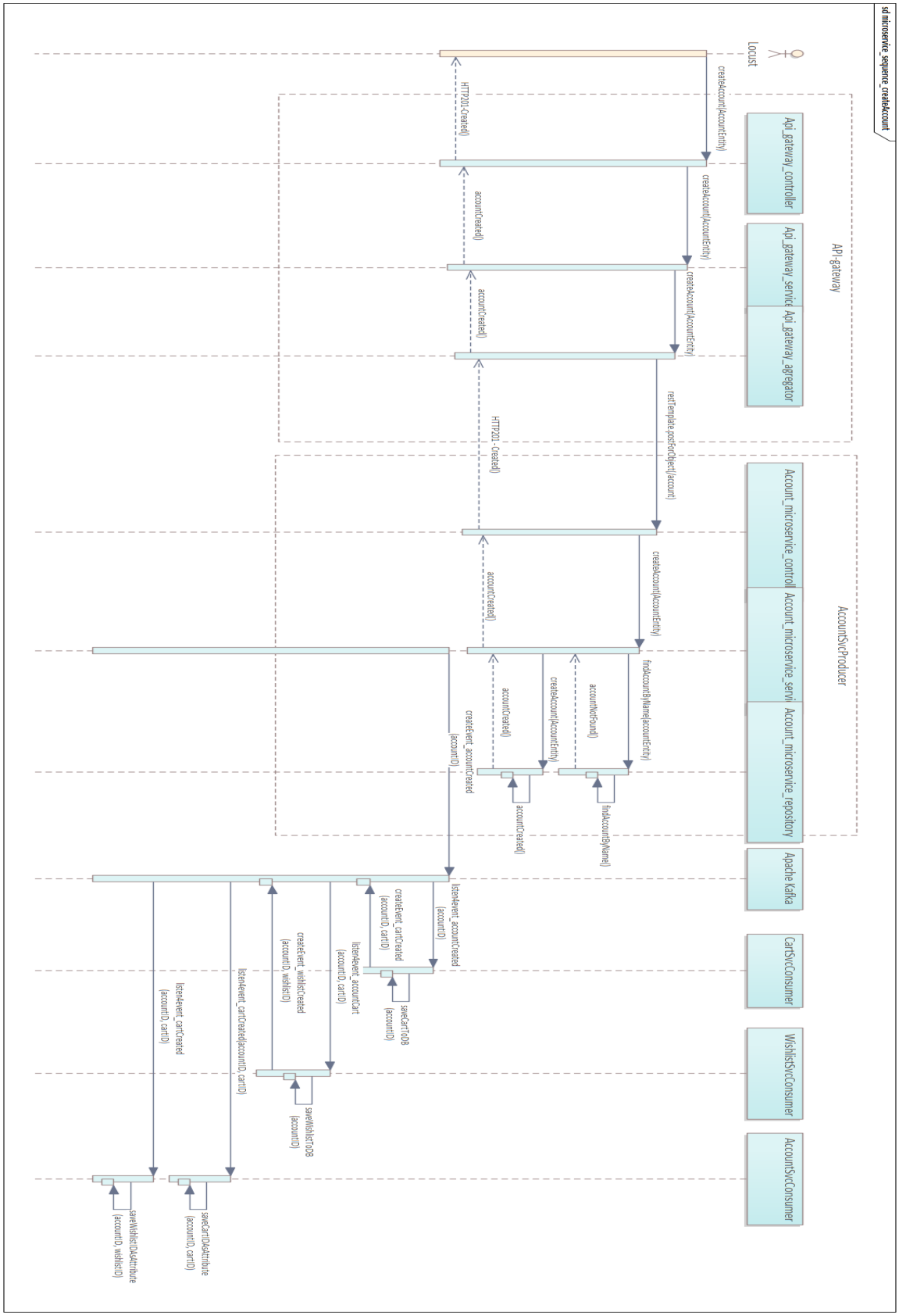
Obrázek 7.1: Analýza endpointů 1



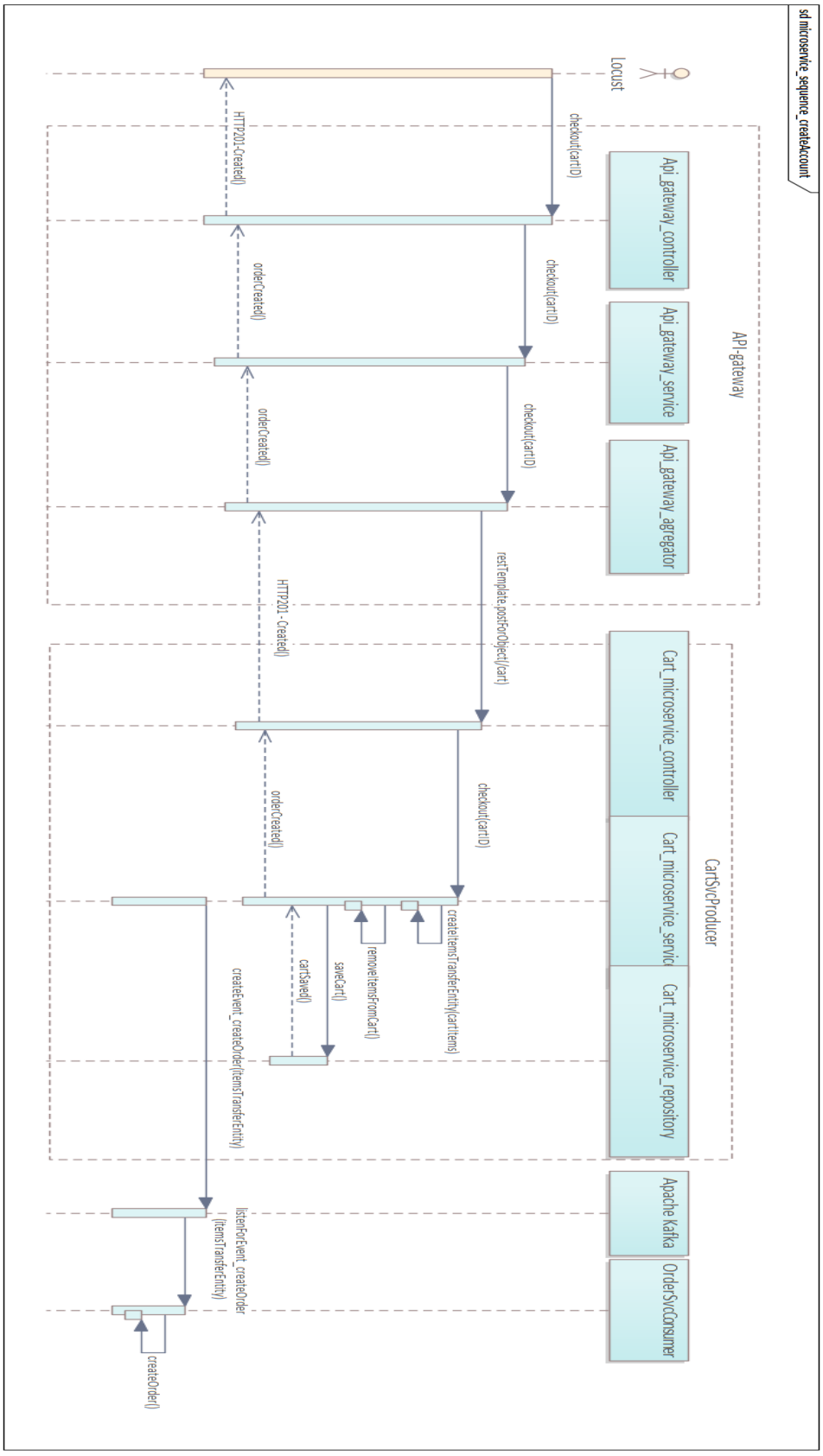
Obrázek 7.2: Analýza endpointů 2



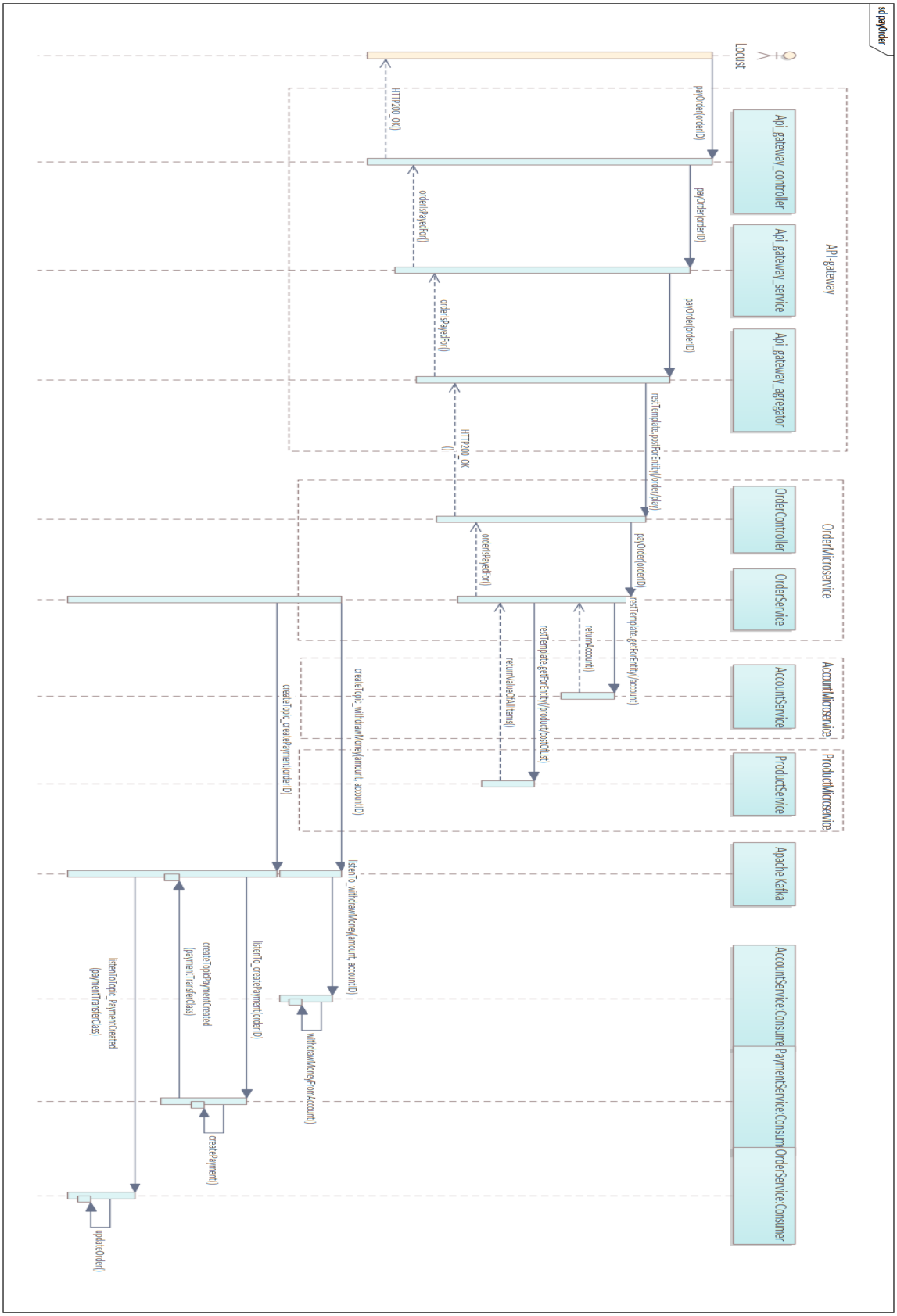
Obrázek 7.3: Rozpad komponent mikroservisní architektury



Obrázek 7.4: Sekvenční diagram createAccount pro mikroslužby

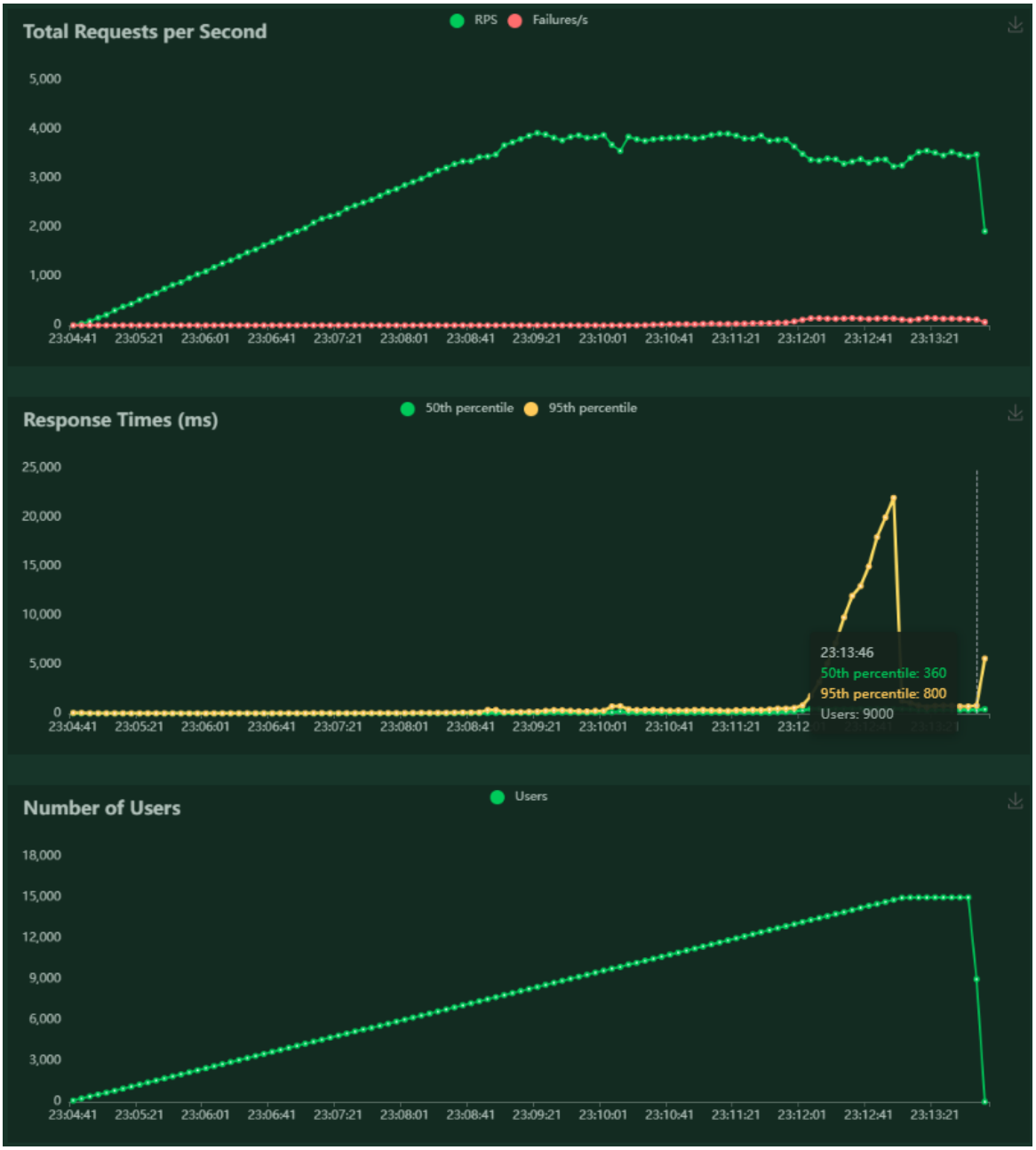


Obrázek 7.5: Sekvenční diagram createOrder pro mikroservisní architekturu

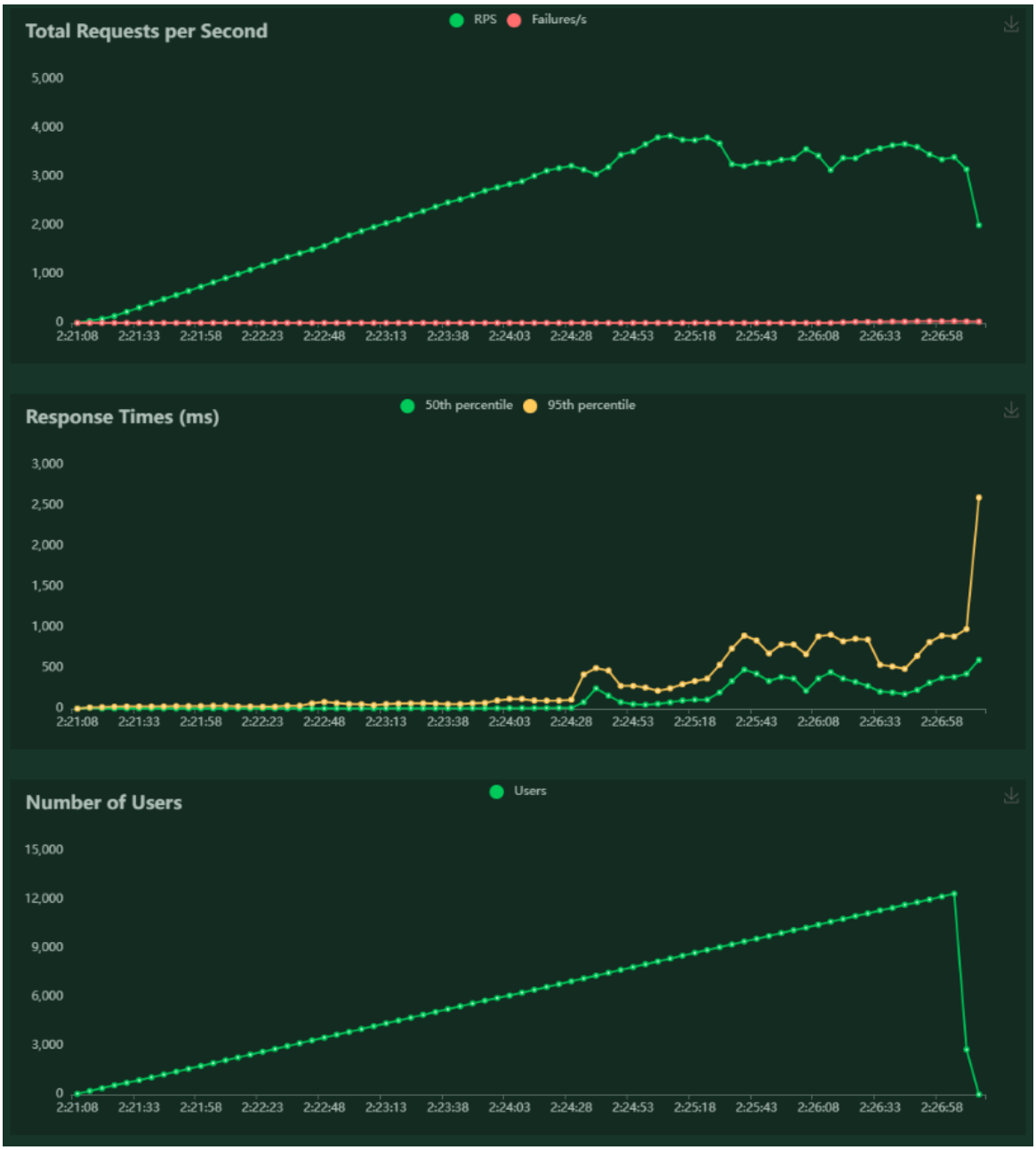


Obrázek 7.6: Sekvenční diagram payOrder pro mikroservisní architekturu

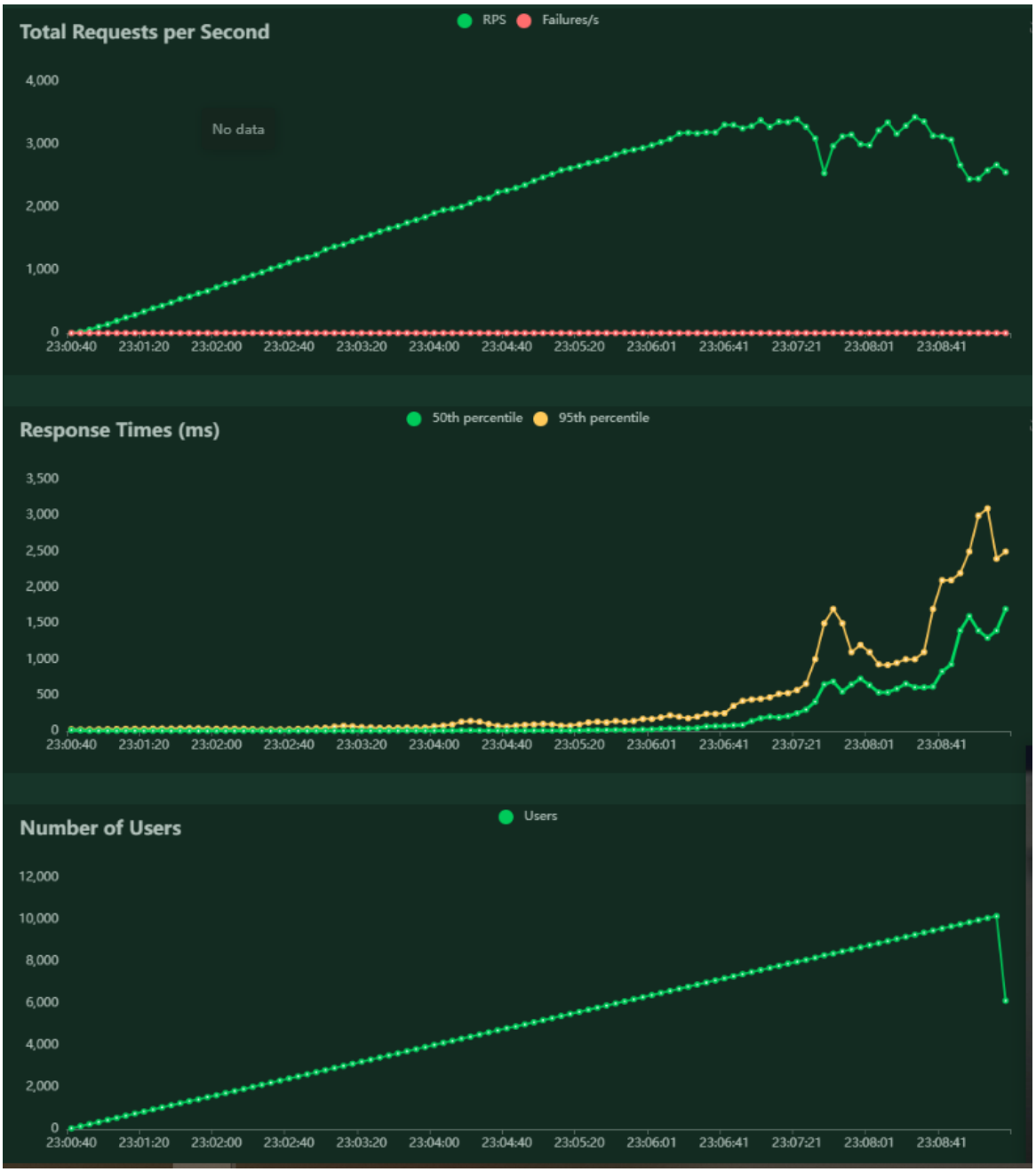
Obrázek 7.7: Locust test pro monolit s rovnoměrným rozložením



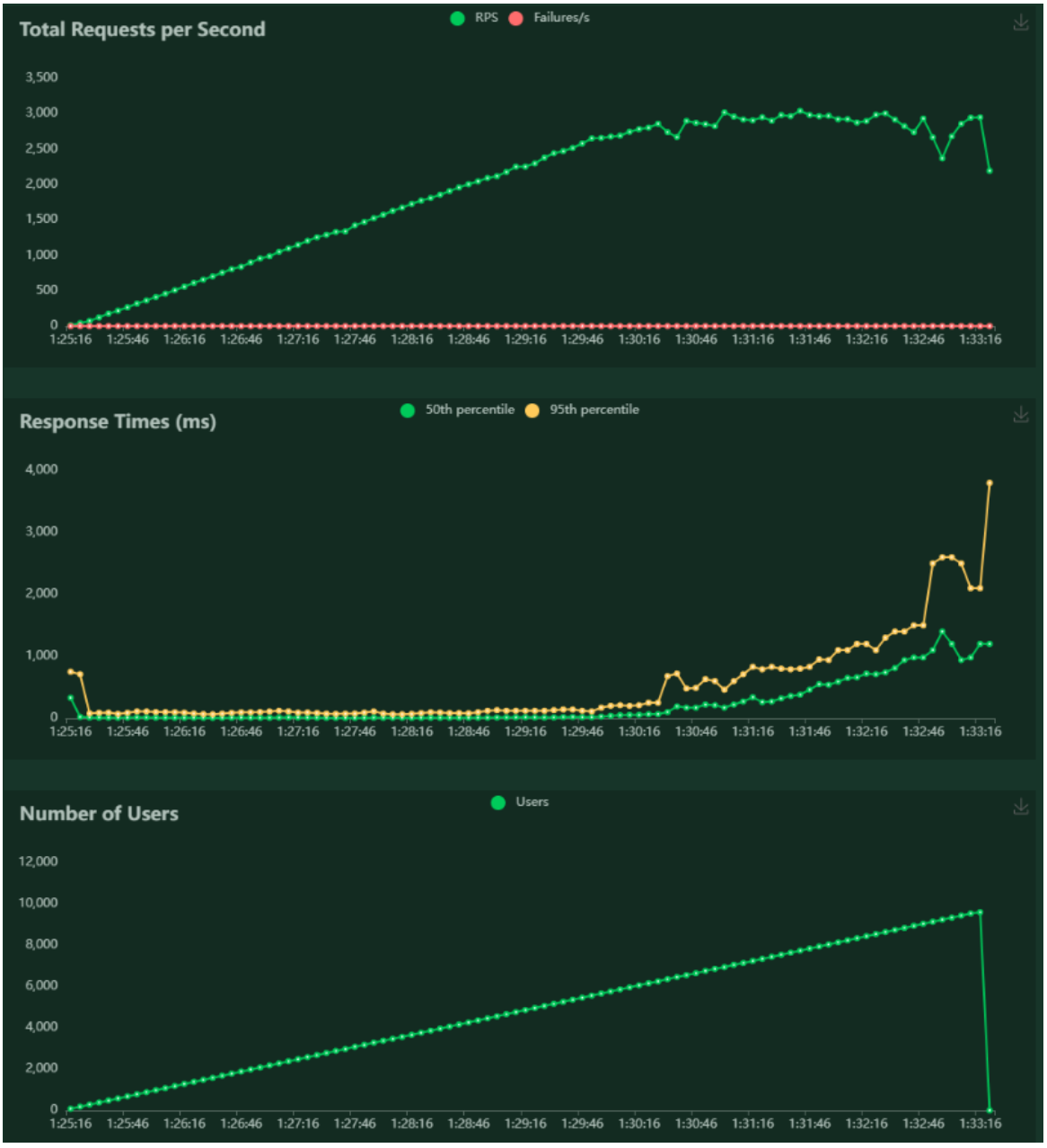
Obrázek 7.8: Locust test pro mikroslužbu s rovnoměrným rozložením



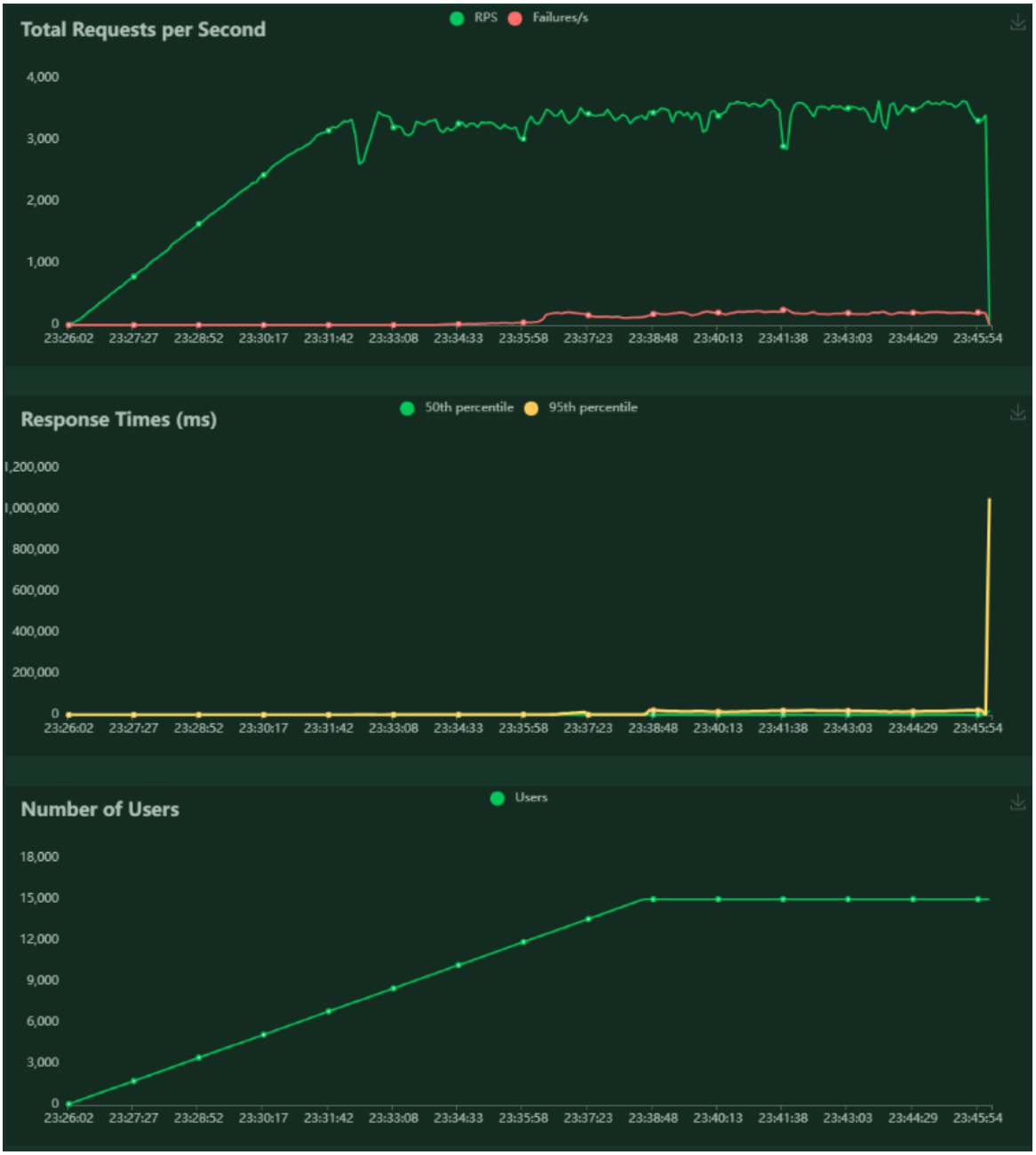
Obrázek 7.9: Locust test pro monolit s koeficientem 3



Obrázek 7.10: Locust test pro mikroslužbu s koeficientem 3



Obrázek 7.11: Locust test pro monolit s koeficientem 5



Obrázek 7.12: Locust test pro mikroslužbu s koeficientem 5