Zadání bakalářské práce

Název:	Optimalizace výkonu backendu služby pro sdílení vozidel Uniqway
Student:	Hoang Nam Tran
Vedoucí:	Ing. Filip Ravas
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Bakalářská práce se zaměřuje na optimalizaci výkonu a celkové zrychlení backendové části carsharingového systému Uniqway.
* Automatizovaným způsobem zatěžte aktuální systém, analyzujte využití zdrojů a identifikujte jeho úzké hrdla.
* Navrhněte a implementujte změny, které povedou k odstranění úzkých hrdel a celkovému zrychlení systému. Ověřte, že zásahy do systému nezmění jeho funkcionalitu.
* Empiricky změřte dosáhnuté zrychlení v porovnání s původním systémem.
* Zhodnoťte význam dosáhnutého zlepšení výkonu systému v kontextu aktuálního stavu projektu. Navrhněte další kroky vedoucí k zlepšení systému.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 7. února 2022 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Optimalizace backendu služby sdílení vozidel Uniqway

Hoang Nam Tran

Katedra softwarového inženýrství
Vedoucí práce: Ing. Filip Ravas

9. května 2022

Poděkování

V první řadě bych chtěl poděkovat svému vedoucímu Ing. Filipu Ravasovi za jeho rady a ochotu při tvorbě této práce. Dále bych chtěl poděkovat lidem z týmu Uniqway za jejich pomoc a laskavost při řešení problémů, se kterými jsem se během tvorby této práce setkal.

Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona a to na dobu určitou do skončení trvání ochrany dle Smlouvy.

Nakládání s předloženou prací se řídí Smlouvou o spolupráci uzavřenou v návaznosti na spolupráci mezi Českým vysokým učením technickým v Praze a společností ŠKODA AUTO a.s. a Smart City Lab s.r.o na výzkumném projektu „CarSharing pro vysokoškolské studenty”, uveřejněné v registru smluv na adrese `https://smlouvy.gov.cz/smlouva/5973503`.

Jsem vázán Smlouvou o zachování mlčenlivosti, že nezpřístupním třetí osobě důvěrné informace, které jsem při své práci na Projektu získal.

V Praze dne 9. května 2022 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2022 Hoang Nam Tran. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Tran, Hoang Nam. *Optimalizace backendu služby sdílení vozidel Uniqway*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2022.

Abstrakt

Práce se zabývá optimalizací výkonu serverové aplikace služby pro sdílení vozidel. Serverová aplikace propojuje všechny klientské aplikace této služby a obsluhuje stovky požadavků v reálném čase. Je proto zapotřebí, aby tyto požadavky byly zpracovávány co nejrychleji. Optimalizaci předchází zátěžové testování aktuálního systému pro analýzu koncových bodů API, měření celkové uživatelské spokojenosti s výkonem systému a identifikaci jeho úzkých hrdel. Následně jsou implementovány změny, které tyto úzká hrdla odstraní a vedou tak k výraznému zrychlení celé služby.

Klíčová slova optimalizace výkonu, zátěžové testování, carsharing, Java, Python, SQL, ORM, analýza algoritmů

Abstract

The thesis deals with the performance optimization of the server application of the vehicle sharing service. The server application connects all client applications of this service and serves hundreds of requests in real time. It is therefore necessary to process these requests as fast as possible. The optimization is preceded by load testing of the current system to analyze API endpoints, measure overall user satisfaction with the system's performance, and identify bottlenecks. Changes are then implemented to remove these bottlenecks, resulting in a significant speedup of the overall service.

Keywords performance optimization, performance testing, carsharing, Java, Python, SQL, ORM, algorithm analysis

Obsah

Úvod **1**

1 Cíl práce **3**

2 Analýza **5**
2.1 Služba Uniqway 5
2.2 Základní pojmy 6
2.3 Aktuální systém 7
2.3.1 Client API 8
2.3.2 Module API 8
2.3.3 Admin API 9
2.3.4 Citymove API 9
2.4 Použité technologie 9
2.4.1 Play Framework 10
2.4.1.1 Actor Model 10
2.4.2 PostgreSQL 11
2.4.3 Ebean ORM 12
2.4.3.1 Častý problém 14

3 Zátěžové testování **17**
3.1 Výkonnostní testování 17
3.2 Metriky 17
3.2.1 Apdex 18
3.3 Nástroj Locust 19
3.3.1 Události 20
3.4 Původní nástroj pro zátěžové testování 20
3.4.1 Zprovoznění a úpravy nástroje 21
3.5 Návrh testů 23
3.5.1 Scénář Admin API 23
3.5.2 Scénář Client API 24

3.6 Spouštění testů 25
3.7 Lokální server 26
3.8 Sledování využití zdrojů 27
3.9 Zátěž systému 28
3.9.1 Výsledky testu Admin API 28
3.9.2 Výsledky prvního testu Client API 29
3.9.3 Výsledky druhého testu Client API 30
3.9.4 Souhrn 31

4 Identifikace úzkých hrdel 33
4.1 Parkovací místa 33
4.1.1 Analýza SQL dotazů 35
4.2 Aktuální lokace vozidel 36
4.2.1 Analýza SQL dotazů 37
4.3 Detailní informace o vozidlech 38

5 Návrh a implementace změn 41
5.1 Optimalizace parkovacích míst 41
5.1.1 Ověření funkčnosti 42
5.2 Optimalizace aktuálních lokací vozidel 43
5.2.1 Ověření funkčnosti 45
5.3 Optimalizace detailu vozidel 45
5.3.1 Ověření funkčnosti 46
5.4 Měření výsledného zrychlení 47
5.4.1 V kontextu škálovatelnosti 48

6 Zhodnocení výsledků 49
6.1 Návrhy na další zlepšení 49
6.1.1 Optimalizace dalších koncových bodů 49
6.1.2 Rozdělení výkonu na více serverů 50

Závěr 51

Literatura 53

A Seznam použitých zkratek 57

B Obsah přiloženého CD 59

Seznam obrázků

2.1 Sekvenční diagram zpracování požadavku pro výpis rezervací . . . 7
2.2 Diagram posílání zpráv mezi aktory [19] 11
2.3 Diagram entity vozidla a jeho stavu 13

3.1 Využití procesoru a paměti při testu Admin API s 50 uživateli . . 29
3.2 Využití procesoru a paměti při testu Client API s 200 uživateli . . 30
3.3 Využití procesoru a paměti při testu Client API se 400 uživateli . . 31

5.1 Využití procesoru a paměti testu Client API při 400 uživatelích . . 48

Seznam tabulek

3.1 Standardní kategorizace požadavků dle Apdex 19
3.2 Statistika zátěžových testů na Stage 26
3.3 Statistika zátěžových testů na Local 26
3.4 Porovnání apdexů Stage vs Local 27
3.5 Souhrn navržených testů 28
3.6 Statistika zátěžových testů na Admin API 28
3.7 Statistika zátěžových testů na Client API při 200 uživatelích . . . 29
3.8 Statistika zátěžových testů na Client API při 400 uživatelích . . . 30
3.9 Výsledky indexů spokojenosti ze zátěžových testů 31

5.1 Porovnání zátěžových testů na Client API při 200 uživatelích . . . 47
5.2 Porovnání zátěžových testů na Client API při 400 uživatelích . . . 47
5.3 Srovnání Apdex všech výsledných testů před a po optimalizaci . . 48

Úvod

Služby pro sdílení vozidel jsou moderním řešením pohodlné přepravy s ekonomicky i ekologicky pozitivním dopadem pro společnost. Právě Uniqway tuto službu nabízí studentům a zaměstnancům českých univerzit. Všechny klientské aplikace na straně zákazníků této služby, ať už se jedná o aplikace webové, mobilní nebo aplikace zabudované ve vozidlech Uniqway, komunikují s centrální serverovou aplikací (tzv. backend), která musí v reálném čase obsluhovat všechny požadavky v co nejkratším čase.

Výstup práce umožní rychlejší běh všech těchto aplikací a pomůže společnosti Uniqway se snáze rozšířit a zároveň ekonomicky ušetřit. Důvodem výběru tohoto tématu je praktický přínos nejen pro společnost Uniqway a zvýšení komfortu jejích zákazníků zrychlením komunikace všech jejích aplikací, ale zároveň pozitivní ekonomický dopad při rozšiřování systémů podobného charakteru.

Práce nejprve analyzuje stav a architekturu aktuálního systému, zkoumá použité technologie a způsob jejich použití. Dále navrhuje scénáře zátěžových testů a za jejich pomoci měří výkonnost systému v kontextu spokojenosti koncových uživatelů. Na základě výsledků z tohoto měření a analýzy kódu jsou detekována úzká hrdla systému, která způsobují jeho zpomalení. Návrh se posléze zaměřuje na implementaci změn, které vedou k odstranění těchto nalezených úzkých hrdel pro dosažení zrychlení celého systému.

Kapitola 1

Cíl práce

Hlavním cílem této práce je optimalizace výkonu serverové části systému služby pro sdílení vozidel Uniqway. Optimalizace klade důraz na dosáhnutí významného zrychlení celého systému při srovnání s empiricky naměřenými daty výkonu aktuálního systému.

Analytická část práce si klade za cíl zmapování architektury aktuálního systému, provedení analýzy využití zdrojů a identifikování úzkých hrdel při automatizovaném zatížení tohoto systému. Součástí zmapování architektury je prozkoumání zdrojového kódu a knihoven, které systém využívá a analyzování způsobu jejich použití.

Cílem praktické části práce je implementace změn v systému na základě poznatků zjištěných v předchozí, analytické, části práce, které povedou k celkovému zrychlení celého systému.

Přínosem práce je snížení nároků systému na infrastrukturu služby Uniqway, kterou používá v jeden okamžik až několik desítek zákazníků, a tím umožnit službě rozšířit počet svých zákazníků bez nutnosti investování do výkonnějších serverů.

Kapitola 2

Analýza

Tato kapitola nejprve představí službu Uniqway, popíše aktuální stav jejího systému a prozkoumá základní technologie, na kterých je postavena serverová část tohoto systému. V rámci průzkumu použitých technologií analyzuje různé možnosti jejich použití, jež by mohly mít vliv na výkon. V neposlední řadě je provedena analýza důležitých koncových bodů, na které je optimalizace v rámci této práce zaměřena nejvíce.

2.1 Služba Uniqway

Služby sdílení vozidel, tzv. carsharing, kombinují komfort při řízení vlastního automobilu s pozitivními ekonomickými dopady při využívání sdílené dopravy. Uživatelé využívající těchto služeb jednak ušetří na nákladech spojené s vlastněním automobilu a navíc si mohou automobil zapůjčit téměř odkudkoliv. Sdílená ekonomika má samozřejmě i své nevýhody a není vždy určena pro každého. Podle propočtů ŠKODA AUTO DigiLab se carsharing vyplatí těm, kteří za rok ujedou maximálně 15 000 kilometrů. Významnou skupinou uživatelů jsou lidé, kteří například dojíždějí vlakem za prací do velkých měst, nemá pro ně význam mít v dané lokalitě vlastní auto a nechtějí platit za parkování a na cesty za město si vystačí s půjčením právě sdíleného vozu. [1]

Uniqway je unikátní carsharingový studentský projekt, který vznikl ve spolupráci studentů ze tří českých univerzit – Českého vysokého učení technického, Vysoké školy ekonomické a České zemědělské univerzity – a dále ve spolupráci s automobilkou Škoda Auto, která projektu poskytuje svá vozidla. Projekt vyvíjí právě studenti a proto se neustále rozvíjí. Služeb tohoto projektu mohou pak využívat studenti a zaměstnanci všech vysokých škol v ČR.

Od svého vzniku na jaře roku 2017, kdy bylo poprvé úspěšně odemčeno auto studentskou kartou, poskytuje Uniqway k dnešnímu dni až 36 aut ke sdílení a má již více než 6000 registrovaných uživatelů. [2]

2.2 Základní pojmy

V následujících kapitolách bude často zmíněno několik pojmů, které je potřeba si nejdříve uvést a krátce vysvětlit.

API Application Programming Interface je rozhraní, které umožňuje jedné softwarové službě získat data od služby jiné bez nutnosti znalosti implementačních detailů této služby. Může se jednat o sadu protokolů, funkcí nebo příkazů, které tuto komunikaci zprostředkovávají. [9]

REST Representational State Transfer je architektonický styl definující sadu principů, jak by mezi sebou měly webové služby ideálně komunikovat. API, které vyhovuje tomuto stylu, se nazývá REST API (nebo též RESTful API). [10]

HTTP Hypertext Transfer Protocol je fundamentální protokol pro přenos dat přes internet. Postupem času vzešel z tohoto protokolu protokol bezpečnější, který obsah dat před přenosem šifruje, zvaný HTTPS (Hypertext Transfer Protocol Secure). [11]

XML Extensible Markup Language je značkovací jazyk, který umožňuje efektivně strukturovat data do dokumentů ve stejnojmenném formátu pro jejich snadné prohledávání a zpracování. Tento formát se často používá pro data přenášená přes internet. [12]

JSON JavaScript Object Notation je šetrný textový formát pro reprezentaci strukturovaných dat založených na syntaxi objektů v programovacím jazyce JavaScript. [13] Tento formát se také díky své šetrné syntaxi často používá pro přenos dat právě prostřednictvím REST API.

SQL Structured Query Language je standardní dotazovací jazyk, který se běžně používá pro práci s daty v relačních databázích.

Framework je obecně podpůrná struktura, kolem které se něco buduje. Může se jednat o nějaký systém pravidel, myšlenek nebo představ, použitý pro plánování nebo rozhodování. [14]

V kontextu softwaru se hovoří o platformě určené pro vývojáře, na které se vyvíjí nějaká aplikace. Většinou se jedná o sadu knihoven, nástrojů, předpřipravených tříd a funkcí. [15]

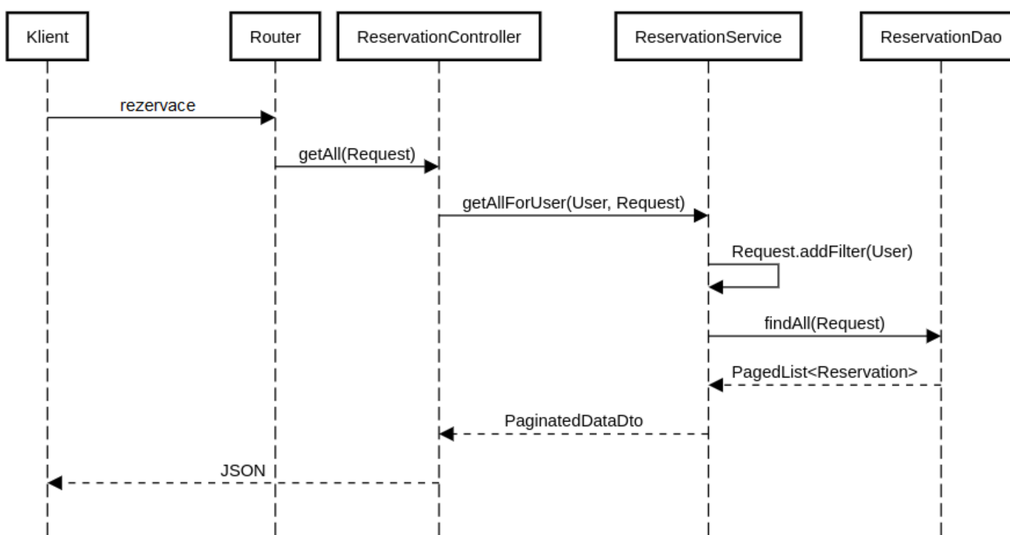
ORM Object-Relational Mapping představuje objektově-orientovanou vrstvu mezi relačními databázemi a objektově-orientovaným programovacím paradigmatem, která usnadňuje práci s těmito daty bez nutnosti psaní složitých SQL dotazů. [16]

V praxi tato vrstva usnadňuje vývojářům práci s persistencí dat, zlepšuje čitelnost kódu a zvyšuje tak celkovou produktivitu.

2.3 Aktuální systém

Architektura systému by se dala rozdělit na 2 hlavní části. První částí je tzv. backend, tedy serverová aplikace napojená na databázi, která obsluhuje veškeré požadavky, poskytuje a zpracovává veškerá potřebná data o uživatelích, o dostupnosti automobilů a jejich lokalitách apod. Druhou částí jsou klientské aplikace, které se serverovou aplikací komunikují prostřednictvím REST API. Mezi tyto klientské aplikace patří mobilní aplikace uživatelů, které slouží k vyhledávání lokality dostupných automobilů, jejich rezervování apod. Dále mezi klientské aplikace patří také moduly zabudované v samotných automobilech, které dávají serverové aplikaci vědět o své lokalitě, o svém aktuálním stavu apod. V neposlední řadě mezi klientské aplikace patří správcovská aplikace, která slouží uživatelům s příslušným administrátorským oprávněním ke správě dat.

Tato práce se zabývá analýzou a optimalizací pouze první části systému, tedy serverové aplikace za účelem dosažení rychlejšího odbavování požadavků, jenž na server přichází ze všech výše zmíněných klientských aplikací. API této serverové aplikace pracuje výhradně s formátem JSON. Každý vystavený koncový bod API této aplikace volá metodu příslušného kontroléru, které jako parametr předá svůj požadavek. Tento požadavek ve formátu JSON je příslušným kontrolérem nejprve validován, následně převeden na objekt DTO, který reprezentuje strukturu tohoto JSON objektu a pokud je vše v pořádku, předá se ke zpracování příslušné službě (service). Pro lepší představu tohoto procesu lze nahlédnout do sekvenčního diagramu pro zpracování požadavku výpisu rezervací pro daného uživatele na obrázku 2.1.



Obrázek 2.1: Sekvenční diagram zpracování požadavku pro výpis rezervací

Serverová aplikace je z velké části postavena na architektuře mikroslužeb a pro manipulaci s daty využívá návrhového vzoru DAO. Většina služeb je potomkem abstraktní třídy `BaseService`, od které dědí základní metody pro zpracování dat jako například `findAll` pro výběr všech záznamů z příslušného objektu DAO. Objekty DAO jsou dále potomky abstraktní třídy `BaseDao` obsahující základní metody pro získávání všech záznamů, vyhledávání konkrétního záznamu dle identifikátoru, persistenci záznamu, mazání, úpravu apod.

Veškeré vystavené koncové body API této aplikace, které slouží klientským aplikacím ke komunikaci se serverem, lze rozdělit na několik základních skupin.

2.3.1 Client API

Skupina Client API zahrnuje zhruba 45 koncových bodů, se kterými komunikují především mobilní aplikace koncových uživatelů. Mezi tyto koncové body patří například zpracování požadavku na přihlášení uživatele, poskytování informací o aktivních rezervacích, vytváření rezervací, poskytování informací o dostupných vozidlech a jejich lokalitách, poskytování informací o dostupných parkovacích místech, zpracování požadavku na odemykání a zamykání automobilu, ale také poskytování informací o samotném přihlášeném uživateli, poskytování možnosti obnovy hesla apod.

Mezi ty nejzajímavější koncové body z pohledu typického uživatele mobilní aplikace lze zahrnout tyto:

- Parkovací místa – údaje o parkovacích místech, které se načítají do mapy
- Aktuální lokace vozidel – údaje o aktuálních lokacích dostupných automobilů
- Detaily vozidel – detailní informace o dostupných vozidlech
- Rezervace – historie rezervací
- Uživatel – údaje o přihlášeném uživateli
- Registrace – informace potřebné pro registraci, jedná se například o výpis univerzit apod.

2.3.2 Module API

Modul zabudovaný v automobilu využívá některé koncové body například pro periodické posílání telemetrických dat na server. [5] Konkrétně periodicky každých 5 vteřin odesílá data na koncový bod:

- Data - informace o aktuální lokaci vozidla a dalších datech

2.3.3 Admin API

Tato skupina zahrnuje zhruba 186 koncových bodů pro správcovskou aplikaci. Kromě poskytování přehledu a kompletní správy uživatelů, vozidel a rezervací, sem spadá například také přehled o platbách, o jízdách, správa novinek, ceníku, odměn a spoustu dalšího.

Mezi ty nejčastěji používané koncové body lze zahrnout mimo jiné tyto:

- Přehled – základní informace na hlavní stránce, zahrnuje přehledné údaje o vozidlech, rezervacích a jízdách
- Správa vozidel
- Správa jízd
- Správa rezervací

2.3.4 Citymove API

Server poskytuje také několik koncových bodů pro mobilní aplikaci Citymove, což je aplikace od firmy ŠKODA Auto DigiLab, která poskytuje plánování tras pro efektivní přepravu s informacemi o možnosti využití různých dopravních prostředků, včetně automobilů Uniqway. Koncové body pro tuto aplikaci poskytují jen informace o dostupných automobilech a parkovacích místech.

2.4 Použité technologie

Serverová aplikace využívá celou řadu technologií. Základem je programovací jazyk Java ve verzi 11, který byl původně zvolen díky tomu, že se jej tým, který stál za vznikem této aplikace, učil na škole a byl s ním tedy dobře obeznámen. [5]

Java byla oficiálně představena v roce 1995 firmou Sun Microsystems, původně s cílem vyvinout multiplatformní programovací jazyk vhodný pro psaní programů určených pro zařízení spotřební elektroniky, jako jsou televizory a počítače používané v automobilech. Již o čtyři roky později se stala Java typickým programovacím jazykem pro velké podnikové webové aplikace. [3]

V porovnání s dnes nejpopulárnějším programovacím jazykem pro tvorbu webových aplikací – PHP, který je interpretovaný, je Java jazyk kompilovaný. Zatímco interpretovaný jazyk spouští celý skript vždy, když se načte stránka, tak díky kompilaci do tzv. bajtkódu jsou Java aplikace bezpečnější a efektivnější při práci s pamětí a z pohledu obsluhy požadavků především výkonnější. [6]

Kromě standardních jednotkových a několika integračních testů, které jsou opět napsané v Javě, využívajících frameworku JUnit, obsahuje projekt také adresář quality-assurance, ve kterém jsou obsaženy různé Python a Bash skripty pro přídavné testování jednotlivých koncových bodů API.

2.4.1 Play Framework

Důraz při výběru technologií, které budou využity při vývoji celého projektu, byl kladen především na fakt, že systém budou vyvíjet studenti. Ačkoliv Java disponuje edicí určené pro velké podnikové aplikace zvanou Java Enterprise Edition, tak byl kvůli menší komplexitě a strmější křivce učení vybrán jednoduchý framework Play. [5]

Play je Open Source framework umožňující jednoduchý a rychlý vývoj webových aplikací v jazycích Java a Scala. Tento framework je navíc specifický svou malou velikostí a je tedy nenáročný z hlediska využití zdrojů. Zároveň se jedná o relativně populární framework s aktivní komunitou a kvalitně zpracovanou dokumentací. Své využití nachází i ve větších podnicích, jmenovitě jej využívají například firmy Electronic Arts, SAMSUNG, LinkedIn nebo UniCredit Group. [7]

Tento framework je postaven na architektuře po vzoru MVC, který odděluje logickou část aplikace od té prezentační. Zároveň disponuje všemi různými užitečnými komponentami pro budování jak klasických webových aplikací, tak služeb REST. Tyto komponenty zahrnují například mocný směrovací mechanismus, tzv. routing, využitý právě pro přehledné vyčlenění jednotlivých, již výše zmíněných, koncových bodů API serverové aplikace.

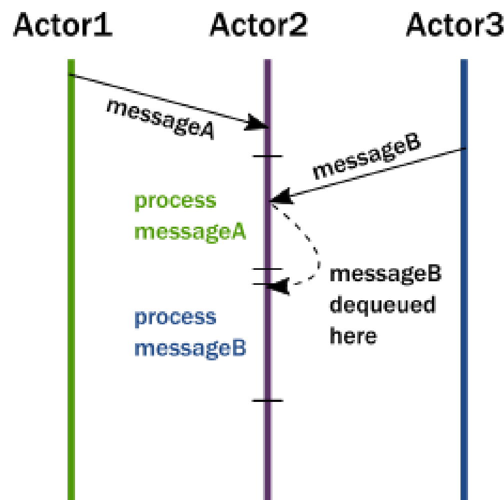
Svou nenáročnost z hlediska využití zdrojů a vysokou výkonnost připisuje balíčku Akka vyvinutý společností Lightbend, který pro jazyky Java a Scala implementuje tzv. Actor Model (viz 2.4.1.1 Actor Model) a slouží k vyvíjení škálovatelných systémů, které efektivně využívají dostupné zdroje a zároveň mohou běžet na více serverech naráz. Konkrétně uvádí zpracování až 50 milionů zpráv za vteřinu na jednom stroji, přičemž využití paměti se odhaduje až na 2,5 milionů aktorů na 1 GB haldy. [17]

2.4.1.1 Actor Model

Na tomto místě je vhodné popsat koncept a fungování výše zmíněného Actor Modelu, na kterém je balíček Akka postaven. Tento model byl poprvé představen Carl Hewittem v roce 1973 jako modulární ACTOR architektura, konceptuálně založena na jediném typu objektu – Aktoru (actor). Tento objekt může reprezentovat virtuální procesor, nějaký stream nebo zásobník. [20]

Tehdy byl tento model navržen jako způsob pro manipulaci paralelního zpracovávání ve vysokorychlostní síti, ačkoliv vysokorychlostní sítě nebyly v té době ještě dostupné. Dnes je tomu však jinak a dnešní hardware a infrastruktura konečně dostihla Hewittovu vizi. Budování distribuovaných systémů často představuje jisté výzvy, které nelze řešit klasickým objektově-orientovaným přístupem, ale dají se vyřešit díky tomuto modelu. [18]

Místo klasického volání metod nějaké třídy, zpracování jejího výsledku a zachytávání případných výjimek tohoto volání, aktoři v tomto modelu mezi sebou komunikují prostřednictvím posílání a příjímání zpráv, které si ukládají



Obrázek 2.2: Diagram posílání zpráv mezi aktory [19]

do lokální fronty, ze které zprávy postupně zpracovávají (viz obrázek 2.2). Takový model jednak lépe odpovídá klasické představě objektového přístupu, kde spolu objekty skutečně komunikují namísto toho, aby si jen navzájem volaly veřejné metody, a zároveň, v případě paralelního zpracovávání, eliminuje tento model několik nevýhod klasického přístupu. Jednou z nich je eliminace nutnosti zamykání vláken, které představují spoustu problémů – jednak může snadno vést k deadlock situaci a jednak jsou zámky poměrně výkonnostně náročné, neboť režie systému pro správu zámků je vysoká. To platí především v případě distribuovaných systémů, kdy se vlákna zamykají mezi jednotlivými stroji napříč sítí.

Hlavním rozdílem mezi posíláním zpráv mezi aktory a voláním metod mezi objekty je, že zprávy nemají návratovou hodnotu. Odesláním zprávy aktoru mu je delegována nějaká práce. V případě volání metod by musel volající objekt čekat na odpověď volaného objektu. V tomto případě, dostane-li aktor nějakou zprávu a zpracuje jí, tak její výsledek může prvnímu aktoru poslat opět prostřednictvím zprávy a první aktor mezitím může řešit práci jinou.

2.4.2 PostgreSQL

Pro databázové uložiště používá serverová aplikace Open Source objektově-relační databázový systém PostgreSQL. Tento systém je aktivně vyvíjen již více než 30 let a pyšní se svou spolehlivostí a výkonností. [8]

Kromě primitivních datových typů jako je INTEGER, STRING nebo BOOLEAN, podporuje také moderní dokumentové datové typy jako jsou například JSON nebo XML. Plně též podporuje cizí klíče a indexování pro zachování integrity dat a zlepšení výkonu při jejich prohledávání. Nelze nezmínit

také možnost spouštění vlastních procedur napsaných v jazycích Perl, Python, C nebo PL/PGSQL.

Klíčovou vlastností PostgreSQL je však jeho objektově-relační orientace, tzv. ORDBMS, využívající přednosti jak z relačních databází, při kterých se využívají dotazovací jazyky pro přístup k datům jak je SQL, tak z objektových databází, díky čemuž podporuje databázový model například dědičnost tabulek apod. Tato vlastnost následně umožňuje jednoduchý, přehledný a efektivní přístup k datům pomocí ORM.

Rychlost databázového systému je pro rychlost celé aplikace stěžejní, neboť drtivá většina operací, ať už se jedná o načítání informací o uživatelích, informací o dostupnosti vozidel, změnu polohy, změnu stavu nebo ukládání historických dat, tak všechny tyto operace zahrnují zásah do databázového systému. Existuje tak mnoho způsobů, jak výkon PostgreSQL ještě více optimalizovat. V konfiguraci lze například nastavit velikost sdíleného bufferu, maximální počet připojení, využití paměti apod. Při ladění výkonu databáze je však mnohem efektivnější optimalizaci provádět na úrovni samotných SQL dotazů a databázové struktury.

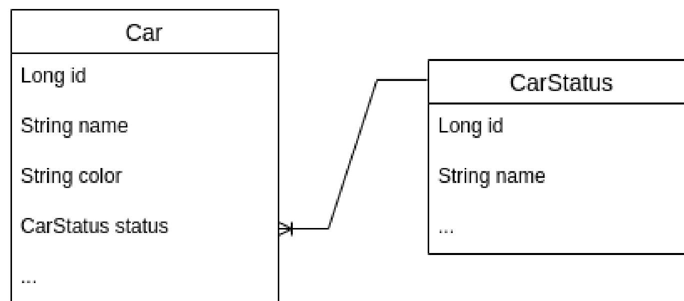
Pro potřeby ladění SQL dotazů a možnosti jejich hlubší analýzy je užitečný modul *pg_stat_statements*, který umožňuje přehledné sledování statistik vykonaných SQL dotazů, jako je počet jejich vykonání, délka jejich trvání nebo velikost jejich výseldků.

2.4.3 Ebean ORM

Pro persistenci samotných dat používá aplikace framework Ebean ORM, který poskytuje mnoho funkcí včetně sestavování složitých ORM dotazů, ukládání dat, mazání dat nebo podporu transakcí.

Základem při persistenci dat jsou modely, které reprezentují jednotlivé entity systému, jako jsou například vozidla, uživatelé, rezervace, značky vozidel apod. Každá z těchto entit dále obsahuje různé vlastnosti, například pro entitu vozidla (`Car`) hovoříme o vlastnostech jako je název nebo barva, avšak patří sem i vlastnosti, které tuto entitu provazují s entitou jinou, jako je například vlastnost jeho aktuálního stavu (`status`), která je provázána s entitou `CarStatus` (pro diagram provázání těchto dvou entit viz obrázek 2.3). Tyto vlastnosti se v kódu definují tzv. dekorátorem, který Ebean framework poskytuje. Každá entita je následně v databázi uložena v nějaké tabulce.

Místo sestavování složitých SQL dotazů do databáze se pro práci s těmito entitami používají metody z poskytovaného rozhraní Ebean frameworku, díky čemuž je práce s daty jednoduchá a především čitelná, což obecně vede k vyšší produktivitě, ale především k nižší chybovosti. Ebean ORM je navíc navržen tak, aby z výkonnostních důvodů nikdy nevygeneroval tzv. kartézský součin záznamů. [24]



Obrázek 2.3: Diagram entity vozidla a jeho stavu

Pro výše uvedený smyšlený příklad na obrázku 2.3 by sestavení ORM dotazu pro výběr všech vozidel, seřazené dle názvu, vypadal například takto:

```
public List<Car> findAllOrderByName() {
   return Ebean.find(Car.class)
         .orderBy("name")
         .findList();
}
```

Zdrojový kód 2.1: Příklad EBean ORM pro výběr vozidel

Pomocí řetězení metod lze jednoduše sestavit i mnohem složitější dotazy. Skutečný příklad složitějšího ORM v Uniqway projektu byl nalezen například u metody pro získání seznamu kategorií (entita `Category`), filtrovaný dle data viditelnosti (vlastnosti `visibleFrom` a `visibleUntil`) a seřazená vzestupně podle vlastnosti `ordering`:

```
public List<Category> findAllVisibleOrdered(OffsetDateTime now) {
   return finder.query()
         .where()
         .le("visibleFrom", now)
         .or(
            Expr.isNull("visibleUntil"),
            Expr.gt("visibleUntil", now)
         )
         .orderBy()
         .asc("ordering")
         .findList();
}
```

Zdrojový kód 2.2: Příklad použití EBean ORM v Uniqway projektu

Zároveň je nutné poukázat na fakt, že se v případě ORM jedná o poměrně vysokou úroveň abstrakce, a že sestavování ORM dotazů má oproti běžným SQL dotazům svá omezení. Nejedná se tudíž o náhražku SQL a v případě potřeby je vhodné využít kombinace ORM s čistým SQL, ačkoliv je z důvodu čitelnosti a konzistence kódu dobré využití SQL minimalizovat. V případě potřeby poskytuje Ebean k tomuto účelu například rozhraní SqlQuery.

Příklad využití tohoto rozhraní, které kombinuje čisté SQL s ORM pro výběr záznamu vozidla na základě jeho `id` je uveden v kódu 2.3.

```
public List<Car> getCarById(Long id) {
   String sql = "SELECT id, name, color FROM car WHERE id = :id";

   return Ebean.createSqlQuery(sql)
         .setParameter("id", id)
         .findOne();
}
```

Zdrojový kód 2.3: Příklad použití klasického SQL v Ebean ORM

2.4.3.1 Častý problém

Problém s chováním Ebean s negativním dopadem na výkon při použití ORM, jež je podrobně odhalen v kapitole 4 (Identifikace úzkých hrdel), dochází při práci se seznamem entit, kdy se přistupuje k jejich položkám, které se vážou na jinou entitu, aniž by byla tato vazba brána v potaz při prvotním vykonání dotazu, který z databáze vybere jen záznamy primární entity. Při přístupu k těmto položkám pak Ebean vykoná přídavné dotazy na pozadí. Konkrétní příklad takového použití na schématu databáze z obrázku 2.3 je následující:

```
List<Car> cars = Ebean.find(Car.class).findList();
...
Car firstCar = cars.get(0);

FirstCarWithStatusDto dto = new FirstCarWithStatusDto();
dto.setCarName(firstCar.getName());
dto.setCarStatus(firstCar.getStatus().getName());
```

Zdrojový kód 2.4: Příklad použití neefektivního ORM

Kód 2.4 nejprve pomocí ORM vybere z databáze seznam všech vozidel. Následně kód z tohoto seznamu vybere první záznam a vytvoří z něj objekt DTO, který v sobě uchovává jednak informaci o názvu vozidla a jednak informaci o názvu jeho stavu, který je reprezentován entitou `CarStatus`. Problém při takovém přístupu nastává při dotazu na stav vozidla metodou `getStatus()`, pro kterou vytvoří Ebean nový dotaz do databáze, který vybere všechny záznamy z tabulky `CarStatus` bez ohledu na to, že chtěný byl jen jediný záznam a to ten, jež má vazbu na `firstCar`. Přiřazení příslušného stavu danému vozidlu však proběhne správně a výsledkem je správný objekt DTO. Nicméně pokud by v databázi bylo N vozidel a M různých stavů a podobný kód byl proveden pro každý z N vozů, pak by chování Ebean mohlo vést k vykonání až $N + 1$ SQL dotazů, přičemž první dotaz by vrátil N výsledků a zbylých N dotazů by byly totožné a vracely by pokaždé M výsledků.

Řešení takového problému je několik v závislosti na potřebě. V případě, že by hrozila potřeba provádět přístup ke stavu u všech vozidel, je možné entitu

`CarStatus` zohlednit již při prvním dotazu. Ukázka takového příkladu je uvedena v kódu 2.5.

```
List<Car> cars = Ebean.find(Car.class).fetch("status").findList();
...
Car firstCar = cars.get(0);
FirstCarWithStatusDto dto = new FirstCarWithStatusDto();
dto.setCarName(firstCar.getName());
dto.setCarStatus(firstCar.getStatus().getName());
```

Zdrojový kód 2.5: Příklad použití neefektivního použití ORM

Takový příklad umožní Ebean vytvořit v SQL dotazu příslušné `JOIN` klauzule, které ke každému vozidlu z výsledku rovnou přiřadí i jeho příslušný stav. Ve výsledku je proveden pouze 1 dotaz do databáze a v kódu lze k položkám typu `getStatus` přistupovat libovolně.

Kapitola 3

Zátěžové testování

Za účelem efektivní a podrobné analýzy výkonu aplikace je potřeba nechat danou aplikaci podstoupit zátěžovým testům. V této kapitole je nejprve stručně představeno výkonnostní testování aplikací obecně a následně popsán seznam metrik, které jsou při měření zátěžových testů nejpodstatnější. Následně v návaznosti na diplomovou práci Ing. Filipa Ravase [5] je zde proveden návrh a implementace několika scénářů zátěžových testů s využitím Open Source nástroje Locust, které simulují definované scénáře chování uživatelů v reálném čase. Nakonec je popsán průběh samotného zátěžového testování v rámci kterého probíhá měření a sbírání relevantních dat včetně dat o využití zdrojů.

3.1 Výkonnostní testování

Výkonnostní testování aplikací se dělí na několik druhů, z hlediska zátěžového testování jsou hlavní 2 druhy – load testing a stress testing. Zatímco stress testing má za cíl zatížit systém co nejvíce, nad míru toho, co je schopen snést a umožnit tak analýzu jeho chování například při DDoS útoku, load testing má za cíl zatížit systém do takové míry, která by umožnila analýzu jeho chování při očekávané zátěži. Vzhledem k tomu, že si tato práce klade za cíl analyzovat a odstranit úzká hrdla pro zrychlení systému a nikoliv zkoumat jakou maximální zátěž je systém schopný snést, bude se kapitola nadále zabývá převážně load testingem.

3.2 Metriky

[4] Na půdě měření výkonu každého počítačového systému existují čtyři základní parametry popisující jeho výkonnost:

Odezva je čas mezi vznikem požadavku a jeho dokončením, měří se v jednotkách času, nejčastěji v milisekundách. Cílem optimalizace výkonu je snížení tohoto parametru na minimum.

Propustnost je počet položek zpracovaných za jednotku času, například počet HTTP požadavků za 1 sekundu nebo počet přenesených bitů za den. Cílem optimalizace výkonu je zvýšení tohoto parametru na maximum. Obvykle platí, že systémy s vysokou propustností mají nízkou odezvu, avšak není to pravidlem, neboť mezi těmito parametry neexistuje příčinná souvislost. Špatnou odezvu při vysoké propustnosti mají například pevné disky.

Využití udává poměrné množství kapacity nějaké komponenty, která se právě využívá. Nemusí být vždy žádoucí, aby se využití všech komponent blížilo ke 100 %, například u diskových jednotek připojené k Ethernetu klesá odezva zvyšováním jejich využití.

Efektivita je obvykle definována jako podíl propustnosti a využití. Tedy má-li například jedna komponenta vyšší propustnost při stejné úrovni využití než druhá, pak je považována za efektivnější. Cílem optimalizace výkonu je maximalizace tzv. nákladové efektivity, což je míra udávající výkonnost na jednotku nákladu.

Prakticky se tato práce bude věnovat především měření a zlepšení odezvy a propustnosti aplikace. Zbylé parametry je však nutné též zohledňovat, neboť je nežádoucí provádět například nákladné přepsání celé aplikace do jiného jazyka, které by vedlo jen k nepatrnému zrychlení, které by pro koncového uživatele bylo doslova nepoznatelné.

3.2.1 Apdex

Pro efektivní optimalizaci systému je nutné si zavést další metriku, na základě které se lze rozhodnout, zda a do jaké míry je pro jakou část systému optimalizaci vhodné provést. Nemá smysl se například snažit optimalizovat každý koncový bod, neboť by taková optimalizace nebyla nákladově efektivní. Cílem je provést rozumnou optimalizaci, tedy zohlednit kritérium, při kterém je uživatel schopen rozdíl zrychlení ocenit nebo ne.

Index spokojenosti zvaný Apdex je průmyslový standard, který měří spokojenost koncového uživatele s výkonem webové aplikace, kterou používá. Konkrétně měří spokojenost s rychlostí odezvy dané aplikace s ohledem na chybovost zpracování požadavků a tuto spokojenost udává v procentech (tedy číslo od 0 do 1 kde 0 znamená naprostou nespokojenost a 1 znamená naprostou spokojenost s výkonem). [21]

Vzorec pro výpočet tohoto indexu spokojenosti je definován jako:

$$Apdex = \frac{S + (T \cdot \frac{1}{2}) + (F \cdot 0)}{S + T + F},$$

kde S (satisfied) značí počet vyhovujících požadavků, T (tolerated) značí počet tolerovaných požadavků a F (frustrated) značí počet nevyhovujících požadavků.

Rozdělení požadavků do výše uvedených kategorií se řídí v závislosti na zvoleném parametru t (threshold), který udává čas odbavení požadavku brán jako vyhovující, podle tabulky 3.1:

Tabulka 3.1: Standardní kategorizace požadavků dle Apdex

Kategorie	Podmínka splnění
Vyhovující	odezva je $\leq t$
Tolerovaný	odezva je $> t$ a zároveň $\leq 4t$
Nevyhovující	odezva je $> 4t$ nebo vrátí chybu (failed)

Daný parametr t se volí na základě cílové skupiny uživatelů a druhu aplikace. Například pro uživatele, který stojí u automobilu a snaží si jej odemknout na dálku bude tento parametr řádově nižší než požadavek na výpis všech vozidel pro správce aplikace.

3.3 Nástroj Locust

Locust je Open Source nástroj pro zátěžové testování typu load testing napsaný v jazyce Python. Je jednoduše škálovatelný a veškeré scénáře a simulované chování jednotlivých uživatelů umožňuje definovat Python skriptem. Tento nástroj umožňuje testování distribuovat i mezi více stroji a podporuje tzv. události (events), které umožňují jednomu procesu simulovat až tisícovku uživatelů naráz. Zároveň umožňuje sbírání přehledných statistik z testování ve formátu CSV. [22] Tato statistika mimo jiné zahrnuje počet vykonaných požadavků, medián a průměr délky odezvy.

Definice scénáře chování uživatele probíhá tak, že se vytvoří klasická třída, která dědí z některé třídy typu `User` od Locustu. Pro testování webové aplikace se může jednat o třídu `HttpUser`. Vytvořená třída pak může obsahovat libovolné podpůrné metody. Klíčové pro Locust jsou metody, které budou označené dekorátorem `@task`. Takové metody jsou ty, které si budou jednotliví simulovaní uživatelé v náhodném pořadí Locustem postupně spouštět. Dekorátoru `@task` lze přiřadit v parametru i prioritu spuštění. Například u metody s dekorátorem `@task(3)`, značící metodu s prioritou 3, bude pravděpodobnost výběru této metody 3 krát vyšší.

Krátký příklad jednoduchého skriptu definující scénář uživatele, který by se neustále dotazoval na stejný koncový bod, je uveden v ukázce kódu 3.1.

```
from locust import HttpUser, task

class TestUser(HttpUser):
   @task
   def isLoggedIn(self):
      self.client.get("/kontrola")
```

Zdrojový kód 3.1: Jednoduchý Python skript pro Locust test

3.3.1 Události

Další důležitou součástí nástroje Locust jsou události, které umožňují spouštět libovolný kód v reakci na nějakou událost, a tak umožňují jednotlivý skript snadno rozšířit o vlastní funkce. Seznam všech událostí lze nalézt v dokumentaci, viz [23]. Mezi ty nejpodstatnější události patří:

test_start událost při spuštění testu

test_stop událost při ukončení testu

init událost při spuštění jednotlivých Locust procesů, užitečná například při distribuovaném testování

request událost při dokončení požadavku

3.4 Původní nástroj pro zátěžové testování

V rámci diplomové práce Ing. Filipa Ravase [5] z roku 2019, která se zabývá škálovatelností serverové aplikace Uniqway, již v projektu vznikly základní skripty pro zátěžové testování, součástí kterých byla i třída pro výpočet indexu Apdex. Tyto testovací scénáře se člení do 3 skupin:

client jehož scénářem bylo zatížit koncový bod pro získávání údajů o aktuálních lokacích dostupných vozidel

- Aktuální lokace vozidel

admin jehož scénářem bylo zatížit následující koncové body, které využívají správci:

- Přehled
- Správa vozidel
- Správa rezervací
- Správa jízd

module jehož scénářem bylo simulovat modul v automobilech, který periodicky každých 5 vteřin odesílá na server telemetrické údaje

- Data

Filip ve své práci využívá knihovnu Locust ve verzi 0.13.0, která již však v současnosti neexistuje, není podporována a vytvořený nástroj je tak dnes již nepoužitelný. Locust má ke dnešnímu dni nejaktuálnější verzi 2.8, která prošla spousty změnami. Bylo rozhodnuto využití těchto skriptů včetně použití nástroje Locust zachovat tak, že je v rámci této práce každý z těchto skriptů upraven pro podporu nejnovější verze Locust.

3.4.1 Zprovoznění a úpravy nástroje

V první řadě byla upravena třída pro výpočet indexu Apdex tak, aby přijímala parametr `tolerated` jako nepovinný a v případě jeho absence si jej sama vypočítala z parametru `satisfied` dle tabulky 3.1.

```
def set_data(self, satisfied, tolerated = None, apdex_file_path = '
apdex.csv'):
   self.__satisfied = float(satisfied)
   if (None == tolerated):
      self.__tolerated = 4 * self.__satisfied
   else:
      self.__tolerated = float(tolerated)
   ...
```

Zdrojový kód 3.2: Parametr tolerated se sám vypočítává z parametru satisfied

Třída pro výpočet indexu Apdex byla následně upravena tak, aby tento index bylo možné vypočítat nejen pro výsledek celého testu, ale i pro jednotlivého uživatele zvlášť. Takový přehled může přinést relevantnější informace o spokojenosti jednotlivých uživatelů. Každý testovací uživatel si tak v konstruktoru inicializuje vlastní Apdex.

Tento Apdex by měl monitorovat výsledky odezvy jednotlivých požadavků daného uživatele. Pro tento účel byla implementována metoda, reprezentující volání na libovolný koncový bod metodou GET, která požadavku současně přiřadí autorizační token JWT, aby byl uživatel neustále přihlášený, a která zároveň informuje Apdex o výsledku požadavku:

```
def request_get(self, url):
   with self.client.get(
      url = url,
      headers = self._auth_header,
      catch_response = True
   ) as resp:
      if resp.status_code == 200:
         self._apdex.process_request(resp.elapsed.total_seconds() *
1000)
      else:
         self._apdex.process_failed()
```

Zdrojový kód 3.3: Metoda pro odeslání GET požadavku

Po ukončení testování daného uživatele je jeho výsledný Apdex vypsán do konzole a uložen do souboru specifikovaném v konstruktoru. Při ukončení testu jednotlivých uživatelů se zavolá metoda `on_stop()`:

```
def on_stop(self):
   self._apdex.print()
   self._apdex.append_apdex_to_file()
```

Zdrojový kód 3.4: Vypsání uživatelského Apdex do konzole a do souboru

Pro integraci třídy pro výpočet indexu Apdex celého testu s již existujícími skripty byly využity právě Locust události. Inicializace této třídy proběhne na začátku každého testu:

```
@events.test_start.add_listener
def init_apdex(environment, **kw):
   apdex.set_data(
      environment.parsed_options.apdex_satisfied,
      environment.parsed_options.apdex_tolerated,
      environment.parsed_options.apdex_file_total
   )
```

Zdrojový kód 3.5: Inicializace celkového indexu Apdex před zahájením testu

Uvedené parametry `satisfied` a `tolerated` jsou pro účel jednoduché konfigurovatelnosti a možnosti jednoduchého spuštění předány skriptu jako argumenty při spuštění nástroje Locust:

```
@events.init_command_line_parser.add_listener
def _(parser):
   parser.add_argument("--apdex-satisfied", type=int, env_var="
LOCUST_APDEX_SATISFIED", default=1000)
   parser.add_argument("--apdex-tolerated", type=int, env_var="
LOCUST_APDEX_TOLERATED", default=4000)
   parser.add_argument("--apdex-file-total", type=str, env_var="
LOCUST_APDEX_FILE_TOTAL", default="apdex_module_total")
   parser.add_argument("--apdex-file-per-user", type=str, env_var="
LOCUST_APDEX_FILE_PER_USER", default="apdex_module_per_user")
```

Zdrojový kód 3.6: Registrace vlastních argumentů nástroje Locust

Pro účel monitorování každého požadavku z testu a zahrnutí jeho výsledku do výsledného indexu Apdex, byla použita událost `events.request`:

```
@events.request.add_listener
def apdex_request_handler(
   request_type, name, response_time, response_length, response,
context, exception, start_time, url, **kwargs
):
   if response.status_code == 200:
      apdex.process_request(response.elapsed.total_seconds() * 1000)
   else:
      apdex.process_failed()
```

Zdrojový kód 3.7: Zahrnutí požadavku do výsledného Apdex celého testu

V poslední řadě, stejně jako u jednotlivých uživatelů, bude výsledný Apdex vypsán a uložen do specifikovaného souboru po dokončení celého testu:

```
@events.test_stop.add_listener
def finish_handler(environment, **kwargs):
   apdex.print()
   apdex.write_apdex_to_file()
```

Zdrojový kód 3.8: Vypsání a uložení výsledného Apdex celého testu

Výsledný nástroj po výše uvedených úpravách nyní podporuje původní skripty navržené Filipem, zároveň vypočítává Apdex i pro jednotlivé testovací uživatele zvlášť a umožňuje příslušné parametry indexu Apdex jednoduše předat v argumentu při spuštění testu. Skripty testů jsou tak navíc snadno rozšířitelné o další scénáře.

3.5 Návrh testů

Pro realizaci testovacích scénářů byl využit výše zprovozněný nástroj. Vzhledem k již existujícím návrhům testovacích scénářů pro skupinu Admin a Module, byl návrh nových testů soustředěn převážně na skupinu Client. Každý navržený zátěžový test zahrnuje následující parametry:

- users – maximální počet souběžně aktivních uživatelů, jež budou vykonávat navržený scénář
- spawn-rate – rychlost vytváření nových uživatelů za sekundu
- run-time – doba běhu celého testu
- apdex-satisfied – maximální čas odezvy v ms pro zahrnutí požadavku mezi vyhovující
- apdex-tolerated – maximální čas odezvy v ms pro zahrnutí požadavku mezi tolerované
- wait-time – doba čekání mezi jednotlivými vykonanými úlohami

Použitá testovací databáze obsahuje v základu celkem 161 aktivních parkovacích míst a celkem 18 dostupných vozidel.

3.5.1 Scénář Admin API

Jak již bylo v předchozích podkapitolách řečeno, původně navržené scénáře testů již relativně dobře pokrývají koncové body, se kterými správci aplikace nejčastěji pracují.

Aktuálně má projekt kolem 50 uživatelů s oprávněním pro komunikaci s touto skupinou koncových bodů a toto číslo se mění jen zřídka. Do budoucna se pravděpodobně nepředpokládá, že by toto číslo mohlo extrémně narůst, proto nastavení počtu uživatelů pro tento test roven 50 se jeví jako rozumné.

Doba čekání mezi vykonáváním jednotlivých úloh u správců byla v původním skriptu nastavena na interval mezi 3 a 9 vteřinami, což se též jeví jako rozumná volba. Drobná změna nastane u koncového bodu:

- Přehled – jedná se o pravděpodobně nejčastěji používaný koncový bod, a proto mu byla oproti ostatním koncovým bodům přiřazena priorita 3

Další změna proběhne v nastavení parametrů Apdex. Zatímco původní návrh měl zvolené hodnoty:

```
apdex_satisfied = 3s
apdex_tolerated = 6s
```

Nově bude test o něco striktnější a pro testovací scénář správce aplikace jsou Apdex parametry stanoveny na:

```
apdex_satisfied = 1s
apdex_tolerated = 4s
```

3.5.2 Scénář Client API

Komfort uživatelů při používání nějaké aplikace je přímo úměrný jejímu výkonu. Pomalé načítání dat při každém kliknutí může být pro uživatele často velice frustrující a může snadno vést k odinstalování dané aplikace.

Původní návrh testů klientských uživatelů počítá, stejně jako u správců, s dobou čekání mezi vykonáváním jednotlivých úloh na náhodnou dobu mezi 3 a 9 vteřinami. V této práci budou bráni v potaz opravdu aktivní uživatelé, kteří v klientské aplikaci překlikávají mezi funkcemi v intervalu od 1 do 5 vteřin. Zároveň se scénáře testu rozšíří o další koncové body, které jsou uživateli frekventovaně používány, a to:

- Parkovací místa – s prioritou 5
- Aktuální lokace vozidel – s prioritou 4
- Detaily vozidel – s prioritou 3
- Rezervace – s prioritou 1
- Registrace – s prioritou 1

Parametry Apdex byly pro klienty stanoveny tak, aby při překlikávání mezi funkcemi v aplikaci naráželi uživatelé na co nejmenší možný odpor, a to z původních:

```
apdex_satisfied = 1500ms
apdex_tolerated = 3s
```

Na nově striktnějších:

```
apdex_satisfied = 500ms
apdex_tolerated = 2s
```

Ačkoliv aplikace eviduje více než 6000 registrovaných uživatelů, nepředpokládá se, že by byli všichni v tak hojném počtu tak aktivní, jak bylo definováno v testovacím scénáři. Pro skupinu Client API jsou navrženy celkem 2 testy. První otestuje zátěž při souběžně aktivních 200 uživatelích. Druhý pak otestuje zátěž při souběžně aktivních 400 uživatelích.

3.6 Spouštění testů

Scénáře testů jsou definovány. Jejich konfigurace byly odděleny do samostatných konfiguračních souborů, pro možnost snadné automatizace a rozšiřitelnosti těchto testů. Ukázka konfiguračního souboru pro test Client API s výše zavedeným nastavením parametrů Apdex, simulující běh 200 souběžně aktivních uživatelů po dobu 10 minut je uvedena v kódu 3.9.

```
host = http://localhost:9000
headless = true
only-summary = true

users = 200
spawn-rate = 10
run-time = 10m
apdex-satisfied = 500
apdex-tolerated = 2000
apdex-file-total = "stats/apdex_client_200_total.txt"
apdex-file-per-user = "stats/apdex_client_200_per_user.txt"
csv = "stats/locust_client_200_stats.csv"
```

Zdrojový kód 3.9: Ukázka konfigurace client.conf pro 200 uživatelů

Pro samotné spouštění zátěžových testů byl modifikován původní skript od Filipa tak, aby načítal výše uvedenou konfiguraci (viz kód 3.10).

```
#!/bin/bash

client_type=$1

if [[ ${client_type} == "module" ]]; then
   echo "Performance testing of Module"
elif [[ ${client_type} == "client" ]]; then
   echo "Performance testing of Client"
elif [[ ${client_type} == "admin" ]]; then
   echo "Performance testing of Admin"
else
   echo "Invalid client type. Use module|client|admin"
   exit 1
fi

mkdir -p stats

pipenv install
pipenv run locust -f performance/${client_type}.py --config
performance/config/${client_type}.conf || true
```

Zdrojový kód 3.10: Ukázka skriptu pro automatizované testování

Díky tomuto skriptu pak lze testy spouštět jediným příkazem, například:

```
$ ./run-performance-script.sh client
```

3.7 Lokální server

Veškeré zátěžové testování probíhá na lokálním serveru zprovozněném na domácím laptopu s operačním systémem Xubuntu, procesorem Intel Core i7 2,4 GHz se 4 jádry a pamětí RAM 12 GB. Z důvodu toho, aby samotné testování nezatěžovalo výkon lokálního serveru a nezkreslovalo tak výsledky měření, byl test spuštěn na lokální síti z jiného stroje.

Důvodů zvolení lokálního serveru pro zátěžový test oproti dostupnému prostředí Stage, jež se infrastrukturou nejvíce podobá té, na které běží produkční aplikace, je hned několik. Hlavním z nich je flexibilita při přístupu ke stroji a nakládání s jeho logy a databázi. Dalším je flexibilita testování, neboť Stage bývá v rámci firmy často využívaná i pro jiné účely a neočekávaná zátěž by mohla ostatním členům týmu způsobit nemilé problémy. V neposlední řadě je důvodem možnost měření využití zdrojů. Stage je aktuálně nasazená na infrastruktuře AWS, která v rámci své služby CloudWatch sice nabízí sofistikovaný monitoring využití zdrojů, avšak s jistým zpožděním, a navíc v definovaných intervalech.

Pro rychlé srovnání lokálního serveru s infrastrukturou na Stage byl proveden krátký 20 vteřinový test na Client API o 50 uživatelích s následujícími parametry Apdex:

```
apdex-satisfied = 1s
apdex-tolerated = 4s
```

Statistika požadavků zátěžového testu na Stage je uvedena v tabulce 3.2 a statistika stejného testu na lokální server je uvedena v tabulce 3.3.

Tabulka 3.2: Statistika zátěžových testů na Stage

Koncový bod	Požadavků	Medián (ms)	Průměr (ms)
Detaily vozidel	354	180	221
Aktuální lokace vozidel	528	97	128
Parkovací místa	633	250	309
Rezervace	125	53	63
Registrace	122	56	80

Tabulka 3.3: Statistika zátěžových testů na Local

Koncový bod	Požadavků	Medián (ms)	Průměr (ms)
Detaily vozidel	347	73	96
Aktuální lokace vozidel	486	42	60
Parkovací místa	653	140	167
Rezervace	143	12	33
Registrace	127	13	37

Při porovnání obou tabulek je zřejmé, že byly průměrné časy požadavků na lokální server odbavovány zhruba 2x rychleji. V absolutních číslech se však jedná jen o několik desítek milisekund. Jedná se navíc o velmi malý a krátký test, který neměl šanci nijak významně vytížit zdroje a tento rozdíl může být částečně přisuzován rozdílu v testovacích datech i odezvě internetu. Stále se však jedná o rozdíl mezi prostředím využitém pro zkoumání v této práci a prostředím, na kterém běží produkční aplikace. Na to je potřeba myslet při zohledňování výsledků měření v následujících kapitolách. Tabulka 3.4 ukazuje porovnání indexů Apdex z obou těchto testů. Zde se výsledky pohybují v obou případech nad 99 %.

Tabulka 3.4: Porovnání apdexů Stage vs Local

Server	Uživatelů	Celkový Apdex	Nejhorší Apdex	Počet Apdex=1
Stage	50	0.9946	0.9720	33/50
Lokální server	50	0.9974	0.9843	41/50

3.8 Sledování využití zdrojů

Ačkoliv pro potřeby sledování využití zdrojů existuje mnoho sofistikovaných nástrojů od základního Správce úloh po složité nástroje podporující interaktivní vizualizaci, jen málokterý by vyhovoval tomu požadavku, aby sbíral data v čase, ukládal je do souboru pro budoucí analýzu a zároveň byl co nejméně výkonnostně náročný. Samotný Správce úloh, který umožňuje jednoduchou vizualizaci využití zdrojů dokáže sám vytížit procesor i na 5 %. Z tohoto důvodu byl pro účel sběru dat o využití procesoru a paměti vytvořen jednoduchý skript, který k tomuto účelu využívá zabudované nástroje v systému jako `vmstat` a `free`. Tento skript data ukládá do CSV souboru, který je pro účel vizualizace zpracován online nástrojem DataWrapper[1].

```
for (( ; ; ))
do
   TIMESTAMP=$(date +%H:%M:%S)
   CPU=$[100-$(vmstat 1 2|tail -1|awk '{print $15}')]
   MEM=$(free -t | awk 'NR == 2 {print $3/$2*100}')
   echo "$TIMESTAMP; $CPU; $MEM"
   echo "$TIMESTAMP; $CPU; $MEM" >> $1
   sleep 1
done
```

Zdrojový kód 3.11: Skript pro sběr dat o využití paměti a procesoru v čase

[1] https://www.datawrapper.de/

3.9 Zátěž systému

Před zahájením samotného zátěžového testování je vhodné shrnout navržené testy spolu s jejich konfiguracemi v jedné tabulce (viz tabulku 3.5). Po dokončení každého z uvedených testů byly uloženy logy z *pg_stat_statements* pro účely analýzy SQL dotazů z databázového systému PostgreSQL.

Tabulka 3.5: Souhrn navržených testů

Test	Uživatelů	Apdex Satisfied	Apdex Tolerated	Doba běhu
Admin API	50	1 s	4 s	10 min
Client API	200	500 ms	2 s	10 min
Client API	400	500 ms	2 s	10 min

3.9.1 Výsledky testu Admin API

Výsledky odezev na jednotlivý koncový bod z prvního zátěžového testu na Admin API při 50 souběžně aktivních uživatelích jsou uvedeny v tabulce 3.6.

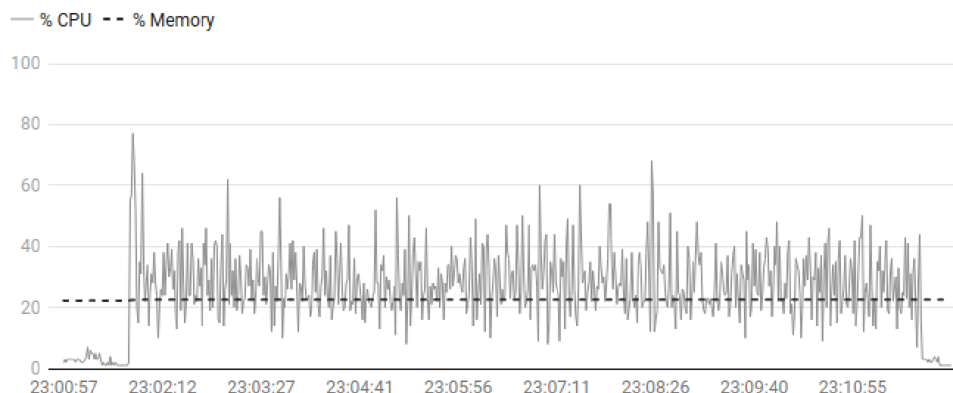
Tabulka 3.6: Statistika zátěžových testů na Admin API

Koncový bod	Požadavků	Medián (ms)	Průměr (ms)
Správa vozidel	811	520	528
Přehled	2393	67	69
Správa rezervací	789	460	464
Správa jízd	772	420	431

Odezvy navržených testovacích scénářů na Admin API se pohybují v rozumných časech. Zajímavé je podotknout, že právě nejčastěji volaný koncový bod má zároveň nejlepší časy. Z toho lze usoudit, že na tento fakt bylo při vývoji tohoto koncového bodu již myšleno a dá se předpokládat, že je již dostatečně optimalizován. Vizualizace statistiky využití zdrojů při tomto zátěžovém testu je uvedena na obrázku 3.1.

Nejvyšší zatížení procesoru nastalo při spuštění testu, které na začátku inicializace každého ze simulovaných uživatelů vytvoří požadavek na přihlášení do aplikace, a až pak následně teprve spouští testování. Během testu pak zátěž kolísá v závislosti na tom, jestli jsou uživatelé zrovna aktivní nebo ne. Pozoruhodný je vývoj využití paměti, které se spuštěním zátěžových testů nijak výrazně nezměnilo.

Výsledný Apdex z tohoto testu je roven 1, tedy každý ze simulovaných uživatelů zaznamenal v průběhu celého testu 100% spokojenost s výkonem.



Obrázek 3.1: Využití procesoru a paměti při testu Admin API s 50 uživateli

3.9.2 Výsledky prvního testu Client API

Výsledky 10 minutového zátěžového testu na Client API při 200 souběžně aktivních uživatelích, při kterém každý z uživatelů překlikává mezi koncovými body v intervalu mezi 1 a 5 vteřinami, jsou uvedeny v tabulce 3.7.

Tabulka 3.7: Statistika zátěžových testů na Client API při 200 uživatelích

Koncový bod	Požadavků	Medián (ms)	Průměr (ms)
Detaily vozidel	8105	130	166
Aktuální lokace vozidel	10641	74	110
Parkovací místa	13098	240	266
Rezervace	2723	29	70
Registrace	2677	38	78

Z této tabulky je patrné, že ten nejčastěji používaný koncový bod, tedy bod pro výpis parkovacích míst, je zároveň tím nejpomalejším, vykazující průměrnou odezvu 266 ms. Dalším relativně pomalým koncovým bodem je výpis detailních informací o dostupných vozidlech. Přesto, že jsou průměrné hodnoty v akceptovatelných číslech, výsledný Apdex zas tak pozitivní není. Jeho celková hodnota činí 98,97 %, přičemž uživatel s nejhorším indexem dosáhl 97,58 %. Celkový počet uživatelů, kteří si během celého testu udrželi 100% index spokojenosti, byl z 200 testovaných pouze jeden jediný.

Vizualizace využití zdrojů v průběhu tohoto testu je vidět na obrázku 3.2. Nejvyšší hodnoty zatížení procesoru se zde pohybují kolem 80 %, což je pravděpodobně ten okamžik, při kterém se doba odezvy jednotlivých požadavků začne zpomalovat. Využití paměti je i zde v průběhu celého testu neměnné.

Obrázek 3.2: Využití procesoru a paměti při testu Client API s 200 uživateli

3.9.3 Výsledky druhého testu Client API

Poslední test probíhal na Client API při 400 souběžně aktivních uživatelích. Výsledky měření z tohoto testu jsou uvedeny v tabulce 3.8.

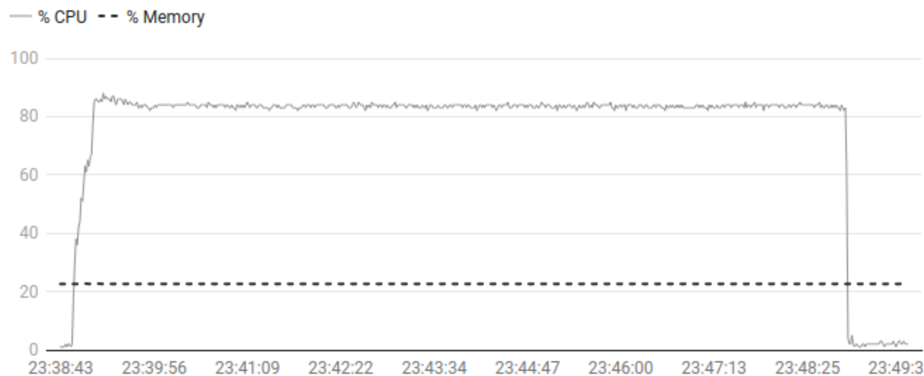
Tabulka 3.8: Statistika zátěžových testů na Client API při 400 uživatelích

Koncový bod	Požadavků	Medián (ms)	Průměr (ms)
Detaily vozidel	8544	2200	2812
Aktuální lokace vozidel	11398	2000	2659
Parkovací místa	14269	2200	2868
Rezervace	2902	2000	2654
Registrace	2877	2100	2714

Průměrné časy odezvy jsou při této zátěži takřka katastrofální, průměrná čekací doba na vykonání požadavku činí skoro 3 vteřiny, a to se odráží i do výsledků indexů Apdex. Celkový Apdex tohoto testu činí 29,43 %, přičemž uživatel s nejhorším indexem se dostal až na 19,56 %. Není potřeba říci, že při těchto číslech neměl žádný ze 400 uživatelů 100% Apdex.

Pozoruhodné na naměřených hodnotách je však nepatrný rozdíl mezi časy jednotlivých koncových bodů. Zatímco v předchozím testu byly požadavky na výpis parkovacích míst mnohonásobně pomalejší než požadavky na Registraci, tak v tomto případě byly u všech koncových bodů časy skoro stejné. Z toho lze jednoznačně vyvodit, že při vysokém zatížení některého z koncových bodů, se extrémně zpomalí i všechny ostatní.

Vizualizace využití zdrojů při tomto testu je vidět na obrázku 3.3. Na této vizualizaci je vidět vytížení procesoru, které se takřka beze změn drží na více než 80 %, což je hodnota, která byla v předchozím testu vyhodnocena jako linie, při které mohou požadavky začít vykazovat dlouhou odezvu.



Obrázek 3.3: Využití procesoru a paměti při testu Client API se 400 uživateli

3.9.4 Souhrn

Souhrn indexů spokojenosti ze všech vykonaných zátěžových testů je uveden v tabulce 3.9.

Tabulka 3.9: Výsledky indexů spokojenosti ze zátěžových testů

Test	Uživatelů	Celkový Apdex	Nejhorší Apdex	Počet Apdex=1
Admin API	50	1,0	1,0	50/50
Client API	200	0,9897	0,9758	1/200
Client API	400	0,2943	0,1956	0/400

Je vhodné zdůraznit také fakt, že některé koncové body Client API mohou využívat i neregistrovaní uživatelé. Například každý návštěvník, který navštíví webovou stránku Uniqway, se stává aktivním uživatelem dotazující se na koncový bod pro výpis parkovacích míst, neboť mapa, která tato místa zobrazuje, se nachází na hlavní stránce. Je proto důležité, aby právě tato skupina koncových bodů zvládala nápor co nejvíce uživatelů a zároveň udržovala co nejvyšší index spokojenosti pro každého z nich.

Výsledky druhého testu ukazují, že již při 200 souběžně aktivních uživatelích není index spokojenosti zcela ideální. Při počtu 400, což činí méně než desetinu z aktuálního registrovaného počtu uživatelů, se již dá očekávat velká nespokojenost uživatelů při používání kterékoliv z klientských aplikací.

Zároveň bylo při posledním testu při simulaci 400 uživatelů odhaleno, že vytížení některých pomalých koncových bodů způsobí výrazné zpomalení i všech ostatních. Optimalizace nejpomalejšího z koncových bodů by tudíž měla při vysoké zátěži vést ke zrychlení i všech ostatních.

Kapitola 4

Identifikace úzkých hrdel

Na základě naměřených dat a výsledků ze zátěžového testování aktuálního stavu systému z předchozí kapitoly se tato kapitola zabývá analýzou těchto dat, analýzou zdrojového kódu příslušných koncových bodů a zkoumáním použití jednotlivých nástrojů s cílem identifikovat úzká hrdla systému, která jeho zpomalení způsobují a která by se dala odstranit.

Analýza zahrnuje popis odpovědi požadavku daného koncového bodu, popis algoritmu při zpracovávání požadavku a dále zkoumá vykonávané SQL dotazy, které jsou analyzovány ze dvou zdrojů – z logů aplikace, které zaznamenávají pořadí vykonaných dotazů a z *pg_stats_statements*, které přehledně vypisuje a řadí vykonané dotazy dle počtu jejich volání a délky jejich trvání. V neposlední řadě je součástí této kapitoly na základě analyzovaných SQL dotazů také zkoumání zkonstruovaných ORM.

Pro účel efektivní optimalizace probíhá analýza pouze u těch nejpomalejších koncových bodů ze skupiny Client API, které jsou patrné z výsledků zátěžových testů v tabulce 3.7 z přechozí kapitoly. Konkrétně se jedná o tyto koncové body:

- Parkovací místa
- Aktuální lokace vozidel
- Detaily vozidel

Neboť optimalizace těchto 3 koncových bodů by měla dostatečně vést k celkovému zlepšení indexu spokojenosti všech uživatelů.

4.1 Parkovací místa

Průměrná odezva při zpracování jednoho požadavku na tento koncový bod při běžném zatížení (viz tabulku 3.7) činí 266 ms, přičemž medián je 240 ms.

Výsledkem požadavku je JSON obsahující seznam objektů, které reprezentující parkovací místa včetně údajů o jejich lokacích. Tyto lokace jsou reprezentovány seznamem objektů, jež nesou informaci o zeměpisné šířce a zeměpisné délce dané lokace. Kromě základních údajů o dané lokaci obsahuje výsledný objekt i údaj o tom, zda má uživatel, jenž požadavek odeslal, dané parkovací místo v odběrech (položka `subscribed`). Ukázka příkladu odpovědi je uvedena v kódu 4.1.

```
[
   {
      "id": 123,
      "name": "Parkovaci Misto 1",
      "nearestAddress": "Parkovaci Misto 1",
      "capacity": -1,
      "occupied": 0,
      "place": [
         {
            "lat": 50.0976929,
            "lng": 14.3448584
         },
         ...
      ],
      "subscribed": false
   },
   ...
]
```

Zdrojový kód 4.1: JSON odpověď výpisu parkovacích míst

Při zpracování požadavku na tento koncový bod je volána metoda `getAll` třídy `ParkingPlaceController`. Tato metoda se stará o autentizaci požadavku a vrácení JSON odpovědi, kterou vyrobí z příslušného objektu DTO, který získá od služby `ParkingPlaceService`. Tato služba pro sestavení výsledného objektu DTO vykoná několik kroků:

1. získej z `ParkingPlaceDao` všechny záznamy o parkovacích místech
2. z těchto záznamů odstraň ty, které nejsou aktivní, tedy mají vlastnost `enabled` nastavenou na `false`
3. na zbylé záznamy aplikuj metodu `get`, která vrátí výsledný DTO, konkrétně vrátí objekt `GetParkingPlaceDto`

Aplikování metody `get` na jednotlivý záznam ve třetím kroku dále zahrnuje tyto kroky:

1. vytvoř prázdný objekt `GetParkingPlaceDTO`
2. přiřaď mu příslušné položky dostupné z entity `ParkingPlace`
3. informaci o odběru parkovacího místa přiřad zvlášť

Pro přiřazení informace o odběru je pro každé z parkovacích míst využita služba `ParkingPlaceSubscriptionService`, konkrétně je zavolána její metoda, která zjistí, zda je dané parkovací místo uložené v tabulce odběrů přihlášeného uživatele. Konkrétní ORM, které tuto informaci získá je uvedeno v kódu 4.2.

```
public class ParkingPlaceSubscriptionDao ... {
...
   public boolean isActiveForParkingPlaceAndUser(
      Long parkingPlaceId,
      ...
   ) {
      ...
      return finder
            .query()
            .where()
            .eq("parkingPlaceId", parkingPlaceId)
            ...
            .or(
               Expr.isNull("deleted_at"),
               Expr.gt("deleted_at", now)
            )
            ...
            .exists();
   }
}
```

Zdrojový kód 4.2: ORM pro ověření existence odběru parkovacího místa

Toto ORM vytvoří dotaz do tabulky `tbl_parkingSubscription`, při kterém zjistí, zda tato tabulka obsahuje záznam s příslušným identifikátorem parkovacího místa a identifikátorem uživatele a zároveň byl tento odběr vytvořen v minulosti a zároveň není označen jako již odstraněn.

4.1.1 Analýza SQL dotazů

Analýza seznamu vykonaných SQL dotazů z logů při odeslání požadavku na tento koncový bod ukázala nadměrné množství samostatných `SELECT` dotazů. Nejprve je vykonán dotaz 4.3, který z databáze vybere všechny záznamy o parkovacích místech z tabulky `tbl_parkingPlace`.

```
select
t0.id,
...
from
tbl_parkingPlace t0;
```

Zdrojový kód 4.3: Vykonaný SQL dotaz pro výběr záznamů parkovacích míst

Tento dotaz vrátí desítky až stovky záznamů s různými identifikátory `id`. Následně je pro každý z těchto identifikátorů vykonán SQL dotaz, který odpovídá ORM z kódu 4.2.

```
select
t0.id
from
tbl_parkingSubscription t0
where
tbl_parkingPlaceId = :parkingPlaceId
and
userId = :userId
and
added_at <= to_timestamp(:addedAt, ...)
and
(
   deleted_at is null
   or
   deleted_at > to_timestamp(:removedAt, ...)
)
limit 1;
```

Zdrojový kód 4.4: SQL dotaz pro ověření existence odběru parkovacího místa

Tedy pokud první dotaz vrátil 161 parkovacích míst, pak následně bude SQL dotaz z kódu 4.4 vykonán samostatně 161 krát, vždy pro jiný parametr `:parkingPlaceId`.

Přesto, že se jedná o dotaz, který vrátí pouze jeden záznam, je nutné počítat s režií při provádění jednotlivých dotazů. Každý samostatně vykonaný SQL dotaz vyžaduje například navázání nového spojení s databázovým systémem. V takovém případě je lepší preferovat málo větších dotazů, které vrátí více záznamů, než více malých dotazů, které vrátí výsledky po jednom. V tomto případě provede tento koncový bod pro každý požadavek, při N parkovacích místech uložených v databázi, až $N + 1$ dotazů do databáze. Z toho 1 dotaz vrátí N záznamů a zbylých N dotazů vrátí nejvýše 1 záznam. Ve výsledku to znamená, že bude odezva na tento koncový bod růst lineárně s počtem parkovacích míst.

4.2 Aktuální lokace vozidel

Průměrná odezva při zpracování jednoho požadavku na tento koncový bod při běžném zatížení (viz tabulku 3.7) činí 110 ms, přičemž medián je 74 ms.

Výsledkem požadavku na tento koncový bod je JSON, který reprezentuje seznam dostupných vozidel (s identifikátorem a názvem) spolu s informacemi o jejich posledních lokacích. Ukázka příkladu odpovědi je uvedena v kódu 4.5.

Služba `CarLocationService`, která příslušnému kontroléru vrátí výsledný objekt DTO, funguje tak, že nejprve využije službu `CarService` k získání záznamů o dostupných vozidlech a na tuto kolekci záznamů poté aplikuje funkci pro přiřazení údajů o lokacích. `CarService` v mezikroku na kolekci záznamů aplikuje příslušné filtry dle uživatelské skupiny (provázanost

```
[
   {
      "car": {
         "id": 123,
         "name": "Vozidlo 1"
      },
      "lat": 50.067364,
      "lng": 14.50811
   },
   ...
]
```

Zdrojový kód 4.5: JSON odpovědi výpisu aktuálních lokací vozidel

s entitou `UserGroup`). `CarLocationService` poté každému z těchto záznamů přiřadí údaje o lokacích provedením následujících kroků pro každé vozidlo zvlášť:

1. nastav zeměpisnou šířku na -1
2. získej od služby `LatestCarDataService` všechny poslední záznamy o zeměpisných šířkách
3. na tuto kolekci záznamů aplikuj filtr, který v kolekci nechá jen ty zeměpisné šířky, které souvisí s daným vozidlem
4. pokud byl nalezen nějaký záznam, pak tento záznam přiřaď výsledné zeměpisné šířce
5. krok 1–4 zopakuj pro zeměpisnou délku
6. vytvoř a vrať výsledný objekt DTO s informacemi o vozidle a příslušnými údaji o lokaci

Problém zde nastává především ve 2. kroku, neboť kromě toho, že se jedná o dotaz do databáze provedený pro každé dostupné vozidlo, tak vrátí všechny záznamy, kterých může být více než samotný počet vozidel. Tyto záznamy se následně filtrují až v kódu. Navíc je ten samý krok proveden ještě jednou pro zeměpisnou délku.

4.2.1 Analýza SQL dotazů

Analýza vykonaných SQL dotazů pak ukázala, že se dotaz v kódu 4.6 v identické formě opakuje v průběhu zpracování jednoho požadavku na tento koncový bod několikrát, a vždy vrací stejný počet výsledků. Jedná se o dotaz, který vrátí seznam všech posledních zeměpisných šířek. Na testovací databázi to činilo vždy 56 záznamů. To samé platí ještě jednou pro zeměpisnou délku.

```
select
t0.id, t0.type_id, ...
from
tbl_latestCarData t0
where
t0.type_id = :latitude;
```

Zdrojový kód 4.6: SQL dotaz pro výběr posledních zeměpisných šířek

Ve výsledku vykoná aplikace při zpracování jednoho požadavku na tento koncový bod, pro N vozidel až $2N + 1$ dotazů do databáze, přičemž první dotaz vrátí N záznamů o vozidlech a u zbylých $2N$ dotazů vrátí každý z nich i více než N záznamů o zeměpisných šířkách, resp. délkách.

4.3 Detailní informace o vozidlech

Průměrná odezva při zpracování jednoho požadavku na tento koncový bod při běžném zatížení (viz tabulku 3.7) činí 166 ms, přičemž medián je 130 ms.

Výsledkem požadavku je JSON obsahující seznam dostupných vozidel a k těmto vozidlům detailní informace o stavu nádrže (položka `tank`), informace o aktuální lokaci (položka `location`), informace o platné palivové kartě (položka `fuelCard`) a spoustu dalšího. Ukázka takové odpovědi je uvedena v kódu 4.7.

```
[
   {
      "id": 123,
      "name": "Vozidlo 1",
      ...
      "tank": { ... },
      "position": {
         "car": {
            "id": 123,
            "name": "Vozidlo 1"
         },
         "lat": 50.067364,
         "lng": 14.50811
      },
      "locked": true,
      "model": { ... },
      "engine": { ... },
      "status": { ... },
      "fuelCard": { ... }
   },
   ...
]
```

Zdrojový kód 4.7: JSON odpovědi výpisu detailních informací o vozidlech

Postup zpracování požadavku je zde podobný jako u předchozích koncových bodů. Získávání seznamu všech dostupných vozidel zde obstarává opět

služba `CarService` a to stejným způsobem, jako v předchozí podkapitole. Na výsledný seznam vozidel je poté aplikována metoda, která pro každé vozidlo vytvoří příslušný objekt DTO v následujících krocích:

1. vytvoř prázdný objekt `GetCarDto`
2. tomuto objektu inicializuj položky z entity `Car` jako je identifikátor, název vozidla, výrobce, barvu atd.
3. za využití služby `CarLocationService` přiřaď výslednému DTO informace o aktuální lokaci daného vozidla
4. za využití služby `CarTankService` přiřaď výslednému DTO informace o aktuálním stavu nádrže daného vozidla
5. za využití služby `CarModelService` přiřaď výslednému DTO informace o příslušném modelu daného vozidla
6. za využití služby `CarEngineService` přiřaď výslednému DTO informace o příslušném motoru daného vozidla
7. za využití služby `CarCarFeatureService` přiřaď výslednému DTO informace o příslušných vlastnostech daného vozidla
8. za využití služby `FuelCardService` přiřaď výslednému DTO informace o příslušné platné palivové kartě daného vozidla
9. vrať výsledný objekt DTO

Průzkum zdrojového kódu a analýza vykonaných SQL dotazů ukázaly, že z výše uvedených kroků je nejproblematičtější 3. krok (informace o lokaci), 4. krok (informace o stavu nádrži) a 8. krok (informace o platné palivové kartě).

Získání informace o lokacích reprezentuje totožný problém jako v předchozí podkapitole 4.2, tedy pro N vozidel představuje tato položka samotná až $2N + 1$ dotazů do databáze.

Služba `CarTankService` používá pro získání informace o stavu nádrže u vozidel se spalovacími motory velice podobný algoritmus jako předchozí služba pro získání informace o lokaci vozidla. Tedy že údaje o aktuálním stavu nádrže získává přes službu `LatestCarData` tak, že vybere všechny poslední údaje o nádržích. Rozdíl oproti algoritmu v `CarLocationService` je ten, že v tomto případě jsou záznamy filtrovány dle identifikátoru vozidla již na úrovni ORM, nikoliv až v kódu. To vede k minimalizaci počtu vrácených záznamů z databáze a jedná se tedy o optimalizovanější verzi algoritmu. Tento algoritmus je aplikován zvlášť pro úroveň paliva (`fuelLevel`) a zvlášť pro procentuální stav nádrže (`tankPercentage`). To ve výsledku vede, pro N vozidel, k dalším $2N$ dotazům do databáze.

Služba `FuelCardService`, která získává informace o platné palivové kartě daného vozu, volá jediné ORM (viz kód 4.8), který filtruje výsledky podle identifikátoru příslušného vozidla a datu expirace, které musí být v budoucnu. Z těchto záznamů dále vybere jen jeden jediný.

```
return finder.query().where()
      .eq("carId", carId)
      .gt("expiration", new Date())
      .findOne();
```

Zdrojový kód 4.8: ORM pro získání platné palivové karty příslušného vozidla

Ačkoliv je získávání informací o nádrži ve srovnání se získáváním informací o lokacích poměrně optimalizované a získávání informací o palivových kartách provádí jen jediný dotaz do databáze, stále jsou všechny tyto kroky vykonávané opakovaně pro každý vůz zvlášť, což ve výsledku vede, pro N vozidel, až k $5N + 1$ dotazům do databáze. Přestože většina z těchto dotazů vrací nejvýše pouze jediný záznam, je stále nutné počítat s režií při vykonávání jednotlivých dotazů bez ohledu na velikost jejich výsledku.

Kapitola 5

Návrh a implementace změn

Tato kapitola se věnuje návrhu a implementaci změn v systému, které vedou k odstranění úzkých hrdel popsaných v předchozí kapitole, a vedou tak ve výsledku k celkovému zrychlení aplikace, které je na konci kapitoly empiricky změřeno a porovnáno s daty naměřenými ze zátěžových testů aktuálního stavu systému. Součástí implementace je i návrh a implementace testů, které ověří, že navržené změny nezmění chování aplikace.

5.1 Optimalizace parkovacích míst

Konkrétně se jedná o optimalizaci koncového bodu pro výpis parkovacích míst pro přihlášeného uživatele, jak bylo prozkoumáno v předchozí kapitole 4.1, která odhalila problém, při kterém se pro N parkovacích míst vykoná až $N+1$ dotazů do databáze. Na základě analýzy chování aplikace při výpisu parkovacích míst je potřeba dbát na zachování těchto bodů:

- výsledný seznam musí obsahovat jen dostupná parkovací místa (položka `enabled` musí být `true`)
- výsledné parkovací místo musí obsahovat informaci o odběru, která je zaznamenána v entitě `ParkingPlaceSubscription`

Ačkoliv se zde na pohled nabízí jednoduché řešení spočívající v sestavení SQL dotazu s příslušnou `JOIN` klauzulí pro získání informace o odběru z příslušné tabulky, tak je nutné brát v potaz fakt, že se tato informace určuje až v kódu na základě existence příslušného záznamu v databázi. Tedy je nutné vybrat i ta parkovací místa, která nemají o odběru žádnou informaci a takovým nastavit položku `subscribed` na `false`.

Z toho důvodu byla implementována nová metoda (viz kód 5.1), která pomocí jednoduchého ORM zahrnuje jak informace o dostupných parkovacích místech, tak v odděleném dotazu vybere i příslušné informace o odběrech. Díky tomu není nutné výsledky nijak dále filtrovat ani zpracovávat. Metoda,

která přiřazuje informaci o odběrech příslušnému DTO byla přejmenována z `get` na `getWithSubscriptions` a pro přiřazení využívá nově navrženou metodu.

```
return Ebean
      .find(ParkingPlace.class)
      .where()
      .eq("enabled", true)
      .filterMany("parkingPlaceSubscriptions")
      ...
      .or(
         Expr.isNull("deletedAt"),
         Expr.gt("deletedAt", now)
      )
      ...
      .findList();
```

Zdrojový kód 5.1: Optimalizované ORM pro výběr parkovacích míst

Následně byl upraven algoritmus pro přiřazování informace o odběru tak, aby počítal s tím, že odběr existuje jen u těch parkovacích míst, které mají jako objekty neprázdnou položku `ParkingPlaceSubscription`. Výsledná implementace by tak měla zredukovat počet dotazů do databáze, nezávisle na počtu parkovacích míst, na pouhé 2 SQL dotazy.

5.1.1 Ověření funkčnosti

Pro ověření, že aplikované změny v systému neovlivní jeho chování se používají jednotkové nebo integrační testy. V tomto případě zásah ovlivnil metodu, jež se dotazuje do databáze. Jelikož databázovou strukturu nelze pro účel jednotkového testu jednoduše simulovat (mocking), nabízí se test celého koncového bodu s využitím Python nástroje Pytest.

Prozkoumáním adresáře těchto testů bylo odhaleno, že v systému již test tohoto koncového bodu, který testuje správnost přiřazení informace o odběru parkovacího místa, existuje. Konkrétně se jedná o test v souboru *test_parking_places.py*. Tento existující test má následující scénář:

1. odstraň všem parkovacím místům informaci o odběru
2. otestuj, že mají všechna parkovací místa odběr nastaven na `false`
3. vyber 4 náhodná parkovací místa ze seznamu parkovacích míst
4. postupně přiřaď těmto 4 parkovacím místům odběr
5. během přiřazování otestuj, že odběr mají jen ta parkovací místa, jimž byl odběr už přiřazen
6. zároveň otestuj, že odběr parkovacího místa, který je již v odběru, nezmění stav tohoto odběru

Definovaný scénář testu byl vyhodnocen jako dostatečně pokrývající potřebné funkcionality, které byly ovlivněny aplikovanými změnami. Při testování byla však odhalena chyba v tomto testu, konkrétně v prvním kroku, který namísto odstranění odběru všem parkovacím místům odstraní odběr jen těm místům, která měla již odběr nastavený na `false`. Tento problém způsoboval neúspěch testu při testování aplikace v původním stavu. Odstraněním této podmínky byl test proveden úspěšně na aplikaci před aplikováním změn a se stejným pozitivním výsledkem byl proveden i po aplikování změn.

5.2 Optimalizace aktuálních lokací vozidel

Problém s výpisem aktuálních lokacích dostupných vozidel byl analyzován v předchozí kapitole 4.2. Jedná se o problém, kdy každému z vozidel je zvlášť přiřazena informace o lokacích, přičemž každý z těchto údajů vyžaduje nový dotaz do databáze, což ve výsledku vede pro N vozidel k až $2N + 1$ dotazům.

Vhodným řešením tohoto problému je sestavení optimalizovaného ORM, které při výběru všech vozidel zohlední také vazbu na entitu `LatestCarData`, která obsahuje potřebná data o lokacích. Návrh prvního takového ORM je uveden v kódu 5.2.

```
return finder.query()
      .orderBy("name")
      .fetch("latestCarData")
      .where()
      .or(
            Expr.eq("latestCarData.type.id", CarDataType.Type.
LATITUDE.getType()),
            Expr.eq("latestCarData.type.id", CarDataType.Type.
LONGITUDE.getType())
      )
      .findList();
```

Zdrojový kód 5.2: ORM pro výběr vozidel s vazbou na položku latestCarData

Bylo však zjištěno, že ačkoliv toto ORM eliminuje veškeré dodatečné dotazy do databáze do tabulky `tbl_latestCarData`, tak výsledný SQL dotaz (viz kód 5.3) rozhodně není optimální.

```
select distinct on (t0.name, t0.id, t1.id) t0.id, t0.name, ...
from tbl_cars t0
left join tbl_latestCarData t1
on t1.car_id = t0.id
left join tbl_latestCarData u1
on u1.car_id = t0.id
where
(u1.type_id = :longitude or u1.type_id = :latitude)
order by t0.name, t0.id;
```

Zdrojový kód 5.3: Vygenerované SQL z navrženého ORM

Jak je vidět, tento dotaz obsahuje 2 vazby na tabulku `tbl_latestCarData`, z nichž první nemá žádnou podmínku. Ve výsledku tak dotaz vrátí kromě údajů o lokacích (zeměpisná šířka a délka) i všechny ostatní údaje z této tabulky, kterých je zhruba 18, což činí pro N vozidel až $18N$ záznamů. Důvod je zřejmě způsoben tím, že uvedené podmínky v ORM pro `type.id` jsou aplikovány v EBean zvlášť a příkaz `fetch` je zohledňován také zvlášť.

Řešením by tedy mohlo být odstranění příkazu `fetch` jak je uvedeno v kódu 5.4.

```
return finder.query()
      .orderBy("name")
      .where()
      .or(
            Expr.eq("latestCarData.type.id", CarDataType.Type.
LATITUDE.getType()),
            Expr.eq("latestCarData.type.id", CarDataType.Type.
LONGITUDE.getType())
      )
      .findList();
```

Zdrojový kód 5.4: ORM pro výběr vozidel bez příkazu fetch

Toto ORM vedlo ke chtěnému odstranění první vazby z SQL 5.3 a ponechání jen druhé vazby s podmínkami na údaje o lokacích, což by se mohlo zdát jako správné. Avšak podrobná analýza SQL logů následně ukázala, že Ebean tuto vazbu nijak nebere v potaz a při každém přístupu k lokaci na entitě `Car` opět vykoná nový dotaz do databáze. Toto ORM tedy jednoznačně nevede k řešení, neboť problém $2N + 1$ dotazů do databáze stále přetrvává.

Výsledným návrhem je ORM uvedené v kódu 5.5, využívající příkazu `filterMany`, které provede dotaz na entitu `LatestCarData` sice zvlášť, ale jednak se správnými podmínkami a jednak tím Ebean registruje fakt, že se pro informace o lokacích daného vozidla nemusí dotazovat do databáze.

```
return finder.query()
      .orderBy("name")
      .where()
      .filterMany("latestCarData")
      .or(
            Expr.eq("type.id", CarDataType.Type.LATITUDE.getType()),
            Expr.eq("type.id", CarDataType.Type.LONGITUDE.getType())
      )
      .findList();
```

Zdrojový kód 5.5: ORM pro výběr vozidel a jejich posledních lokacích

Ve výsledku toto ORM tak vykoná bez ohledu na počet vozidel pouhé 2 dotazy, z nichž první vrátí N záznamů o vozidlech a druhý vrátí $2N$ záznamů o lokacích.

Pro zavedení tohoto ORM byl zároveň upraven algoritmus pro přiřazení údajů o lokacích příslušným vozidlům. Zatímco původní algoritmus získal nejprve všechny zeměpisné šířky a ty dále filtroval dle identifikátoru příslušného

vozidla, upravený algoritmus již počítá s tím, že má údaje o lokacích k dispozici, a tak výslednému DTO zeměpisný údaj přiřadí rovnou.

5.2.1 Ověření funkčnosti

Pro ověření, že provedené změny neovlivní chování aplikace, byl navržen test koncového bodu pro výpis aktuálních pozic vozidel. Jelikož se jedná o test koncového bodu, a nikoliv klasický jednotkový test, byl využit Python skript s využitím nástroje Pytest a příslušnými třídami reprezentující klienta a administrátora, které v systému již existují. Tento test nejprve získá všechny záznamy o aktuálních lokacích vozidel z optimalizovaného koncového bodu a každý z těchto záznamů následně porovnává s hodnotami lokací záznamů získané z příslušného koncového bodu o detailech vozu z Admin API, jež pro výpis využívá jiných služeb, kterých se optimalizace nijak netýkala. Výsledky tohoto testu byly pozitivní jak v systému před aplikováním změn, tak po provedení optimalizace.

Ačkoliv byl proveden pokus o rozšíření testu o další scénáře, jež zahrnovaly chování modulu na změnu lokace, tak se simulace modulu pro účel testování na lokálním serveru ukázala být značně obtížná, a proto byla změna lokací provedena v databázi manuálně načež test, který porovnal příslušná data po této změně lokace, byl opět úspěšný.

5.3 Optimalizace detailu vozidel

Jak bylo popsáno v kapitole 4.3, výpis detailů všech dostupných vozidel obsahuje úzkých hrdel hned několik. Prvním z nich je přiřazení každému vozu údajů o jeho aktuálních lokacích. Druhý problém se týká získání údajů o stavu nádrže každého vozidla a poslední problém se týká získávání údajů o platných palivových kartách.

Zatímco první dva problémy využívají velice podobného algoritmu (neboť jsou v obou případech údaje uložené v entitě `LatestCarData`), z nichž první byl navíc již vyřešen v předchozí podkapitole 5.2, tak získávání údajů o platných palivových kartách řeší služba `FuelCardService` pomocí speciálního ORM (viz kód 5.6).

```
return finder.query().where()
      .eq("carId", carId)
      .gt("expiration", new Date())
      .findOne();
```

Zdrojový kód 5.6: ORM pro získání platné palivové karty příslušného vozidla

Za využití znalostí z předchozí podkapitoly bylo navrženo ORM, které řeší všechny 3 problémy naráz, viz kód 5.7. Nejprve se vyberou všechna vozidla seřazená dle názvu (jako tomu bylo doposud), následně se za využití příkazu

filterMany přidá dotaz pro výběr všech platných palivových karet, a nakonec se za využití dalšího příkazu filterMany přidá další dotaz pro výběr všech údajů o stavu nádrží a lokacích.

```
return finder.query()
      .orderBy("name")
      .where()
      .filterMany("fuelCards")
      .gt("expiration", new Date())
      .filterMany("latestCarData")
      .in("type.id",
            CarDataType.Type.LATITUDE.getType(),
            CarDataType.Type.LONGITUDE.getType(),
            CarDataType.Type.FUEL_LEVEL.getType(),
            CarDataType.Type.TANK_PERCENTAGE.getType()
      )
      .findList();
```

Zdrojový kód 5.7: Optimalizované ORM pro výběr detailu vozidel

Výsledné algoritmy pro filtraci a přiřazení lokací a údajů o stavu nádrže byly upraveny v podobném duchu jako v předchozí podkapitole.

Výrazně se tak zredukuje počet vykonaných dotazů závislých na počtu vozidel. Ačkoliv tento koncový bod zahrnuje ještě spoustu dalších neprodiskutovaných údajů, které provádějí samotné dotazy do databáze jako je zjištění informací o modelu, motoru apod., tak je jejich počet zanedbatelný. Počet vykonaných SQL dotazů zahrnující údaje o vozidlech, nádrži, lokaci a palivových kartách byl tak zredukován pro N vozidel z původních $5N + 1$ na pouhé 3 bez ohledu na počet vozidel.

5.3.1 Ověření funkčnosti

Ověření funkčnosti tohoto koncového bodu opět probíhá v rámci Python skriptu. V tomto případě byl test navržen tak, že nejprve získá data z příslušného koncového bodu, kterého se optimalizace týkala. Následně u každého ze záznamů porovná výsledky položek stavu nádrže, lokace a palivové karty s daty získané z příslušného koncového bodu ze skupiny Admin API, který u daných položek vrací stejné výsledky a zároveň se ho optimalizace nijak netýkala. Výsledky tohoto testu byly pozitivní jak v systému před aplikováním změn, tak po provedení optimalizace.

Podobně jako při ověření funkčností předchozího koncového bodu byl i zde následně proveden manuální zásah do testovacích dat pro simulaci aktualizace příslušných údajů některých vozidel. Po provedených změnách byl test zopakován a opět proběhl úspěšně.

5.4 Měření výsledného zrychlení

Po aplikaci všech výše uvedených změn byly pro srovnání zopakovány zátěžové testy na Client API s 200 a se 400 uživateli z kapitoly 3.9 o zátěžovém testování. Srovnání statistik testu provedeného před a po aplikaci změn je uveden v tabulce 5.1 pro 200 uživatelů, resp. v tabulce 5.2 pro 400 uživatelů.

V tabulce 5.1 lze při běžné zátěži vypozorovat výrazné, tedy i více než dvojnásobné zrychlení u každého z testovaných koncových bodů, včetně těch, kterých se optimalizace netýkala. Výsledný celkový Apdex byl u tohoto testu zlepšen z původních 98,97 % na celých 100 %.

Tabulka 5.1: Porovnání zátěžových testů na Client API při 200 uživatelích

Koncový bod	Požadavků		Medián (ms)		Průměr (ms)	
	Před	Po	Před	Po	Před	Po
Detaily vozidel	8105	8324	130	55	166	57
Aktuální lokace vozidel	10641	11042	74	33	110	35
Parkovací místa	13098	13809	240	49	266	53
Rezervace	2723	2808	29	24	70	26
Registrace	2677	2708	38	31	78	34

Tabulka 5.2: Porovnání zátěžových testů na Client API při 400 uživatelích

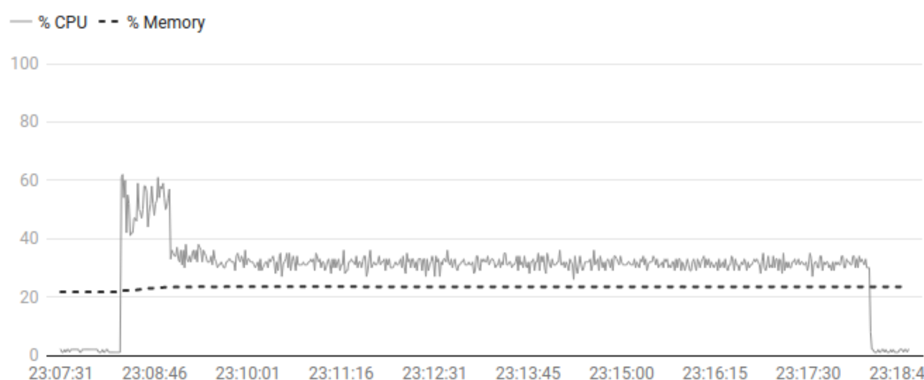
Koncový bod	Požadavků		Medián (ms)		Průměr (ms)	
	Před	Po	Před	Po	Před	Po
Detaily vozidel	8544	16337	2200	63	2812	71
Aktuální lokace vozidel	11398	21694	2000	37	2659	42
Parkovací místa	14269	27058	2200	65	2868	76
Rezervace	2902	5501	2000	29	2654	35
Registrace	2877	5319	2100	41	2714	50

V případě výsledku testu při 400 uživatelích v tabulce 5.2 jsou čísla po optimalizaci oproti původním číslům mnohem optimističtější. Nejen, že měl při stejném testu optimalizovaný systém téměř dvojnásobnou propustnost požadavků, ale navíc nepřekročila průměrná odezva žádného z koncových bodů 80 ms. Výsledný celkový Apdex tohoto testu se zlepšil z původních 29,43 % na 99,96 %.

Důvod 99,96% indexu spokojenosti při tak nízké průměrné odezvě ukázala statistika Apdex jednotlivých uživatelů, která zaznamenala, že u 50 ze 400 testovaných uživatelů spadl 1 požadavek do kategorie tolerovaných. Vzhledem k tomu, že každý ze simulovaných uživatelů vykonal během testu zhruba 200 požadavků, se jedná o zanedbatelné číslo.

O tom, že se zároveň významně snížilo vytížení serveru z hlediska využití jeho zdrojů, vypovídá graf na obrázku 5.1, který ukazuje, že vytížení procesoru

během celého testu nepřekročilo 40 %. Výjimkou je začátek testu, při kterém se každý ze 400 simulovaných uživatelů před zahájením testování musí na server přihlásit, což zároveň mohlo být tím důvodem, proč některým uživatelům spadl 1 požadavek do kategorie tolerovaných. Mírnou změnu oproti testům původního stavu systému lze vypozorovat u využití paměti, která se v průběhu nového testu mírně zvýšila o několik jednotek procent.



Obrázek 5.1: Využití procesoru a paměti testu Client API při 400 uživatelích

5.4.1 V kontextu škálovatelnosti

Vzhledem k tomu, že se všechny identifikované výkonnostní problémy ukázaly být lineárně závislé na počtu parkovacích míst nebo počtu vozidel v databázi, bylo navrženo několik dodatečných zátěžových testů, které srovnávají výsledky optimalizace v kontextu rozšiřování systému o další vozidla. Konkrétně se jedná o stejný test na Client API při 200 uživatelích, avšak tentokrát pro různý počet dostupných vozidel v databázi. Výsledná tabulka 5.3 srovnává celkový Apdex všech těchto testů včetně standardního testu při 400 uživatelích.

Tabulka 5.3: Srovnání Apdex všech výsledných testů před a po optimalizaci

Test	Uživatelů	Vozidel	Celkový Apdex před	Celkový Apdex po
Client API	200	18	0,9897	1,0
Client API	200	30	0,8113	1,0
Client API	200	50	0,4938	1,0
Client API	200	100	0,2113	0,9999
Client API	400	18	0,2943	0,9996

Lineární závislost výkonu celé aplikace na počtu dostupných vozidel před optimalizací je z tabulky evidentní. Zatímco pro 50 dostupných vozidel klesl v původním systému celkový Apdex pod 50 %, po aplikaci změn je systém i při 100 dostupných vozidlech na uspokojivých 99,99 %.

Kapitola 6

Zhodnocení výsledků

Optimalizace vytipovaných tří nejpomalejších koncových bodů se ukázala být správná a předpoklad, že zrychlení právě těchto tří bodů povede k celkovému zrychlení zpracování požadavků i dalších koncových bodů, se též ukázal být na místě. Výsledná optimalizace odstranila u těchto koncových bodů lineární závislost počtu vykonaných dotazů do databáze na počtu parkovacích míst a počtu dostupných vozidel, což vedlo k výraznému zrychlení nejen při neobvykle vysoké zátěži, ale také při případném rozšiřování projektu o další vozidla a parkovací místa. Striktně nastavený index spokojenosti byl pro 18 dostupných vozidel a 161 parkovacích míst při 200 souběžně aktivních uživatelích zlepšen z 98,97 % na 100 %, přičemž odezva každého z vykonaných dotazů na server nepřesáhla 500 ms. V případě vysoké zátěže při 400 souběžně aktivních uživatelích byl tento index spokojenosti zlepšen z původních 29,43 %, při kterém byla průměrná doba odezvy téměř 3 vteřiny, na nových 99,96 %. Systém je navíc po provedené optimalizaci nyní schopný si takto vysoký index spokojenosti udržet i při rozšíření systému o velké množství nových vozidel a nových parkovacích míst.

6.1 Návrhy na další zlepšení

Ačkoliv výsledky optimalizace ukazují výrazné zrychlení aplikace, je potřeba podotknout, že se práce zabývá jen malou podmnožinou koncových bodů, a to pouze těch, které nevyžadují na pozadí žádné složité výpočty. Možnosti další optimalizace systému tedy rozhodně nejsou u konce a v této podkapitole bude uvedeno několik návrhů na další změny, jež by vedly k dalšímu zrychlení celého systému.

6.1.1 Optimalizace dalších koncových bodů

Podobnou pozornost pro optimalizaci si zaslouží i všechny ostatní koncové body. Problémy s neoptimálním dotazováním do databáze se vyskytují u větši-

ny z nich. Ačkoliv by optimalizace každého koncového bodu vedla k výraznému zrychlení, jednalo by se o poměrně pracnou záležitost. Hlavním problémem, jež sdílely všechny 3 koncové body optimalizované v této práci, je neefektivní nakládání s daty při sestavování objektů DTO, jež vyžadují data z více entit. Jelikož se o získávání záznamů jednotlivých entit starají příslušné služby (services), přičemž každá z těchto služeb sestavuje samostatné ORM a tudíž zároveň vykonává samostatné SQL dotazy, tak snadno dochází k lineární závislosti mezi počtem vykonaných SQL dotazů a počtem záznamů v databázi. Ačkoliv výhodou takového návrhu je zlepšení čitelnosti kódu, a především zvýšení znovupoužitelnosti těchto služeb, výrazně se tím zvyšuje zatížení serveru.

Jedním z možných návrhů pro odstranění tohoto problému je zavedení samostatné služby pro sestavování objektů DTO, přičemž by každá z těchto služeb sestavovala a vykonávala optimální ORM právě pro svůj účel a nevyužívala žádnou z přídavných služeb. Problémem tohoto návrhu by však byla snížená znovupoužitelnost některých již existujících služeb.

6.1.2 Rozdělení výkonu na více serverů

V extrémním případě, tedy při nutnosti zpracovávání obrovského množství požadavků, je možné například využít zabudovaného balíčku Akka a Actor Modelu (viz kapitola 2.4.1.1 Actor Model) pro rozdělení zátěže systému na více serverů. Způsobů takového rozdělení je mnoho. Jedním z možných návrhů tohoto rozdělení je například rozdělit aplikaci zvlášť na aplikaci obsluhující požadavky na Client API, zvlášť aplikaci obsluhující požadavky na Admin API apod.

Závěr

Cílem této práce bylo automatizované zatížení aktuálního systému, analyzování využití jeho zdrojů během této zátěže a identifikování a odstranění úzkých hrdel, které způsobují zpomalení systému. Cílem odstranění těchto úzkých hrdel mělo být dosažení výrazného zrychlení celého systému.

Praktické části této práce předcházela analýza architektury systému a použitých technologií. Před započetím zátěžového testování byly popsány základní metriky pro měření výkonu a byl představen index spokojenosti, který měří z příslušných naměřených dat ze zátěžových testů celkovou spokojenost uživatelů s výkonem dané aplikace. Pro účel automatizovaného zátěžového testování byl zprovozněn a upraven původní, již existující, avšak zastaralý nástroj. V rámci těchto úprav byl nástroj uzpůsoben tak, aby mohl jednoduchým způsobem spouštět snadno konfigurovatelné zátěžové testy. Byl taktéž napsán vlastní nástroj pro sběr dat o využití zdrojů pro jejich hlubší analýzu.

Na základě naměřených dat ze zátěžových testů byly následně vybrány tři nejpomalejší koncové body, na kterých proběhla analýza jejich algoritmů, která zahrnovala analýzu kódu a SQL dotazů. Z analýzy vyplynulo několik zásadních problémů, které u daných koncových bodů vedly k lineární závislosti mezi počtem vykonaných dotazů do databáze a počtu uložených záznamů. Tyto problémy byly v praktické části odstraněny implementací optimalizovaných ORM a úpravou algoritmů vybraných koncových bodů pro použití těchto ORM. Výsledné měření po provedení optimalizace ukázalo výrazné zlepšení jak v kontextu spokojenosti uživatelů s používáním aktuálního systému, tak v kontextu případného rozšiřování projektu o další vozidla a parkovací místa.

Velké nedostatky s výkonem v celé aplikaci však nadále přetrvávají. Díky zprovozněnému nástroji pro jednoduché a konfigurovatelné zátěžové testování lze tyto testy však snadno rozšířit o další scénáře a za využití znalostí získaných v této práci nebude obtížné tyto problémy do budoucna identifikovat a odstranit a aplikaci tím optimalizovat mnohem více.

Literatura

[1] DÖRNER, Petr. CARSHARING: JEZDIT AUTEM, ALE NEVLASTNIT HO. *Škoda Storyboard* [online]. 2019 [cit. 2022-03-12]. Dostupné z: https://www.skoda-storyboard.com/cs/e-mobilita-cs/carsharing-jezdit-autem-ale-nevlastnit-ho

[2] Uniqway První český carsharing pro studenty a zaměstnance univerzit. *Škoda Auto Digilab* [online]. [cit. 2022-03-12]. Dostupné z: https://skodaautodigilab.com/cs/projects/uniqway

[3] KEOGH, James Edward. *Java bez předchozích znalostí: průvodce pro samouky.* Brno: CP Books, 2005. ISBN 978-80-251-0839-0.

[4] KILLELEA, Patrick. *Vylaďování webového výkonu.* Praha: Computer Press, 1999. ISBN 80-7226-181-6.

[5] RAVAS, Filip. *Scalability of Car Sharing System.* Praha, 2019. Diplomová práce. FEL ČVUT.

[6] PEABODY, Brad. *Server-side I/O Performance: Node vs. PHP vs. Java vs. Go* [online]. [cit. 2022-03-15]. Dostupné z: https://www.toptal.com/back-end/server-side-io-performance-node-php-java-go

[7] *Play Framework* [online]. [cit. 2022-03-15]. Dostupné z: https://www.playframework.com/

[8] PostgreSQL - About. *PostgreSQL* [online]. [cit. 2022-03-16]. Dostupné z: https://www.postgresql.org/about/

[9] Application Programming Interface (API). *Techopedia* [online]. [cit. 2022-02-25]. Dostupné z: https://www.techopedia.com/definition/24407/application-programming-interface-api

[10] What is a REST API?. *Redhat* [online]. [cit. 2022-02-27]. Dostupné z: https://www.redhat.com/en/topics/api/what-is-a-rest-api

[11] Hypertext Transfer Protocol (HTTP). *Techopedia* [online]. [cit. 2022-02-27]. Dostupné z: https://www.techopedia.com/definition/2336/hypertext-transfer-protocol-http

[12] XML introduction. *MDN Web Docs* [online]. MOZILLA CONTRIBUTORS [cit. 2022-02-27]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/XML/XML_introduction

[13] Working with JSON. *MDN Web Docs* [online]. MOZILLA CONTRIBUTORS [cit. 2022-02-27]. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON

[14] FRAMEWORK. *Cambridge Dictionary* [online]. [cit. 2022-03-20]. Dostupné z: https://dictionary.cambridge.org/dictionary/english/framework

[15] Framework. *Techterms* [online]. [cit. 2022-03-20]. Dostupné z: https://techterms.com/definition/framework

[16] Understanding Object-Relational Mapping: Pros, Cons, and Types. *Altexsoft Blog* [online]. [cit. 2022-03-20]. Dostupné z: https://www.altexsoft.com/blog/object-relational-mapping/

[17] Akka: Build powerful reactive, concurrent, and distributed applications more easily. *Akka.io* [online]. [cit. 2022-04-02]. Dostupné z: https://akka.io/

[18] Why modern systems need a new programming model. *Akka Docs* [online]. [cit. 2022-04-02]. Dostupné z: https://doc.akka.io/docs/akka/current/typed/guide/actors-motivation.html

[19] How the Actor Model Meets the Needs of Modern, Distributed Systems. *Akka Docs* [online]. [cit. 2022-04-02]. Dostupné z: https://doc.akka.io/docs/akka/current/typed/guide/actors-intro.html

[20] HEWITT, Carl, Peter BISHOP a Richard STEIGER. *A Universal Modular ACTOR Formalism for Artificial Intelligence.* [online]. IJCAI, 1973, 235-245 [cit. 2022-05-03]. Dostupné z: https://www.researchgate.net/publication/220812785_A_Universal_Modular_ACTOR_Formalism_for_Artificial_Intelligence

[21] Apdex: Measure user satisfactio. *New Relic docs* [online]. [cit. 2022-04-04]. Dostupné z: https://docs.newrelic.com/docs/apm/new-relic-apm/apdex/apdex-measure-user-satisfaction

[22] Locust Documentation - Locust 2.8.6. *Locust docs* [online]. [cit. 2022-04-04]. Dostupné z: http://docs.locust.io/en/stable/index.html

[23] Event Hooks - API - Locust 2.8.6 Documentation. *Locust docs* [online]. [cit. 2022-04-04]. Dostupné z: http://docs.locust.io/en/stable/api.html#events

[24] N + 1 queries - Ebean ORM. *EBean ORM docs* [online]. [cit. 2022-04-04]. Dostupné z: https://ebean.io/docs/query/background/nplus1

Příloha **A**

Seznam použitých zkratek

API Application Programming Interface

AWS Amazon Web Services

CSV Comma-Separated Values

DAO Data Access Object

DDoS Distributed Denial of Service

DTO Data Transfer Object

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

JSON JavaScript Object Notation

JWT JSON Web Token

MVC Model-View-Controller

ORDBMS Object-Relational Database Management System

ORM Object-Relational Mapping

PHP PHP: Hypertext Preprocessor

REST Representational State Transfer

SQL Structured Query Language

XML Extensible Markup Language

Příloha B

Obsah přiloženého CD

```
readme.txt ................................ stručný popis obsahu CD
src
├── impl ................................. zdrojové kódy implementace
│   ├── app .............. optimalizované zdrojové kódy implementace
│   ├── quality-assurance
│   │   ├── utils ............ přídavné nástroje pro zátěžové testování
│   │   ├── performance .... scénáře zátěžových testů a jejich konfigurace
│   │   └── tests .......................... testy koncových bodů API
│   └── tests ............................................ jednotkové testy
└── thesis ................. zdrojová forma práce ve formátu LaTeX
text ...................................................... text práce
└── thesis.pdf ........................ text práce ve formátu PDF
```