ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Vodvářka** Jméno: **Otto** Osobní číslo: **474502**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Architektura systému pro analýzu online trhu

Název diplomové práce anglicky:

Architecture of system for online market analysis

Pokyny pro vypracování:

Cílem práce je návrh nové softwarové architektury systému pro zpracování a analýzu velkého množství dat z online trhu, která plní požadavky klíčových vlastníků systému. Součástí návrhu architektury je:
1. analýza stavu současného řešení, identifikace problematických částí a návrh možného zlepšení,
2. analýza požadavků na systém,
3. identifikace/výběr klíčových požadavků, které formují architekturu,
4. rešerše možných přístupů (architektonické styly, vzory, normy/standardy, technologie),
5. návrh architektury, včetně návrhu klíčových komponent a jejich zodpovědností, technologií a návrhu procesu vývoje a provozu (postupů, rolí, metod, nástrojů),
6. hodnocení architektury z pohledu požadavků, včetně výčtu nutných kompromisů v návrhu a jejich odůvodnění,
7. ověření architektury pomocí prototypu, který demonstruje užití klíčových technologií a ověří naplnění vybraných kvalitativních atributů (např. odolnost proti výpadku, horizontální škálovatelnost), funkčně se zaměří na příjem a uložení velkého množství dat skrz API a distribuované výpočty (transformace) nad obsáhlými kolekcemi.

Seznam doporučené literatury:

Mark Richards - Software Architecture Patterns
Martin Kleppmann - Designing Data-Intensive Applications
Michael T. Nygard - Release It!: Design and Deploy Production-Ready Software 2nd Edition
Tadeusz Oszubski - Implementing Domain-Driven Design
Len Bass, Paul Clements, Rick Kazman - Software Architecture in Practice (4th edition)

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Josef Smolka katedra softwarového inženýrství FJFI

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

Datum zadání diplomové práce: **11.02.2022** Termín odevzdání diplomové práce: **20.05.2022**

Platnost zadání diplomové práce: **30.09.2023**

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

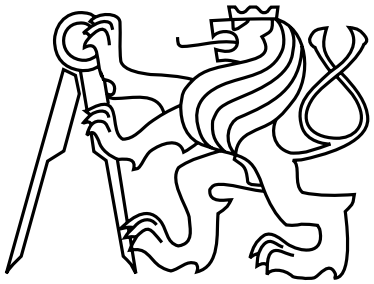
III. PŘEVZETÍ ZADÁNÍ

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

. _____
Datum převzetí zadání

_____
Podpis studenta

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Diplomová práce

Architektura systému pro analýzu online trhu

Bc. Otto Vodvářka

Vedoucí práce: Ing. Josef Smolka

Studijní program: Otevřená informatika
Specializace: Softwarové inženýrství

Květen 2022

Poděkování

Chtěl bych zde poděkovat vedoucímu diplomové práce Ing. Josefovi Smolkovi za cenné rady, věcné připomínky a vstřícnost při konzultacích.

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.

V Praze dne 20. 5. 2022 .....

Abstract

The aim of the thesis is to design a new software architecture of a system for processing and analyzing large amounts of online market dynamics data. The data is continuously extracted from online channels using agents and then processed in a central system for distribution to end users using a web application. The application is used to support managerial decisions in marketing and business.
The thesis analyses the requirements of key owners for the system, the architecture of the current solution, identifies shortcomings and suggestions for improvement and proposes a new architecture, which is then verified on a prototype. The research part of the thesis discusses possible approaches using known architectures with an assessment of strengths and weaknesses. The key requirements for the architecture are identified as scalability, availability, big data processing and flexibility in further development of the system. The work concludes with the selection and evaluation of a suitable architecture and its validation in a prototype.

Keywords: Software architecture, Distributed system, Online marketing, Big Data, DevOps, Software development, Microservices, Proof of concept

Abstrakt

Cílem práce je návrh nové softwarové architektury systému pro zpracování a analýzu velkého množství dat dynamiky online trhu. Data jsou průběžně extrahována z jednotlivých online kanálů pomocí agentů a následně zpracována v centrálním systému pro distribuci koncovým uživatelům pomocí webové aplikace. Aplikace slouží pro podporu manažerského rozhodování v oblasti marketingu a obchodu.
Práce analyzuje požadavky klíčových vlastníků na systém, architekturu současného řešení, identifikuje nedostatky a podněty ke zlepšení a navrhuje novou architekturu, kterou následně ověřuje na prototypu. V rešeršní části práce rozebírá možné přístupy pomocí známých architektur s hodnocením silných a slabých stránek. Jako klíčové požadavky na architekturu jsou identifikovány škálování, dostupnost, zpracování velkého objemu dat a flexibilita v dalším rozvoji systému. Práce je zakončena výběrem a zhodnocením vhodné architektury a jejím ověřením v rámci prototypu.

Klíčová slova: Softwarová architektura, Distribuovaný systém, Online marketing, Big Data, DevOps, Vývoj software, Microservices, Prototyp

Obsah

1 Úvod **1**
1.1 Popis . . . 1
1.2 Motivace . . . 1

2 Zadání **3**

3 Analýza business case **5**
3.1 Popis aplikace . . . 5
3.2 Cílová skupina . . . 5
3.3 Funkce uživatelské aplikace . . . 6
3.3.1 Report webu . . . 6
3.3.1.1 Klíčová slova . . . 7
3.3.1.2 Návštěvnost . . . 7
3.3.1.3 Odkazy . . . 7
3.3.2 Report klíčového slova . . . 7
3.3.3 Report vlastníka . . . 7
3.3.4 Sledování . . . 8
3.3.5 Globální přehledy . . . 8

4 Analýza současného řešení **9**
4.1 Komponenty a architektura . . . 9
4.1.1 Diagram komponent . . . 9
4.1.2 Master service . . . 10
4.1.3 Registry service . . . 10
4.1.4 Messaging service . . . 10
4.1.5 RabbitMQ . . . 10
4.1.6 ScyllaDB cluster . . . 11
4.1.7 Roboti . . . 11
4.1.8 Webová aplikace . . . 11
4.2 Hardware . . . 11
4.3 Vývoj . . . 11
4.3.1 Vývojové prostředí . . . 11
4.3.2 Version Control System . . . 12
4.3.3 Metodika . . . 12
4.4 DevOps . . . 12

4.4.1 Vytváření artefaktů . . . 12
4.4.2 Testování . . . 12
4.4.3 Release management . . . 12
4.4.4 Nasazení . . . 12
4.4.5 Monitoring . . . 13
4.5 Zátěž systému . . . 13
4.5.1 Externí zátěž . . . 13
4.5.2 Databázový cluster . . . 13
4.6 Identifikované problémy . . . 13
4.6.1 Vysoká závislost služeb . . . 13
4.6.2 Distribuce závislá na platformě . . . 13
4.6.3 Manuální nasazování . . . 14
4.6.4 Škálovatelnost a robustnost . . . 14
4.6.5 Monitoring a logování . . . 14

5 Nefunkční požadavky systému 15
5.1 Dostupnost . . . 15
5.1.1 Definice . . . 15
5.1.2 Požadavky vlastníků systému . . . 16
5.1.3 Testovací scénáře . . . 16
5.1.3.1 Selhání komponenty . . . 16
5.1.3.2 Load balancing požadavků . . . 17
5.2 Složitost nasazení . . . 17
5.2.1 Definice . . . 17
5.2.2 Požadavky vlastníků systému . . . 18
5.2.3 Testovací scénáře . . . 18
5.2.3.1 Nasazení nové verze komponenty . . . 18
5.2.3.2 Oprava kritické chyby v produkčním prostředí . . . 19
5.3 Výkonnost . . . 20
5.3.1 Definice . . . 20
5.3.2 Požadavky vlastníků systému . . . 20
5.3.2.1 Testovací scénáře . . . 20
5.3.2.2 Prioritizace analytických požadavků . . . 20
5.3.2.3 Velké množství současně připojených uživatelů . . . 21
5.4 Škálovatelnost . . . 21
5.4.1 Definice . . . 21
5.4.2 Požadavky vlastníků systému . . . 21
5.4.3 Testovací scénáře . . . 22
5.4.4 Automatická orchestrace počtu spuštěných instancí na základě metrik 22
5.5 Udržovatelnost . . . 22
5.5.1 Definice . . . 22
5.5.2 Požadavky vlastníků systému . . . 23
5.5.3 Testovací scénáře . . . 23
5.5.3.1 Celkový stav systému . . . 23
5.5.3.2 Podrobnosti chybového stavu . . . 23
5.5.3.3 Změna konfigurace komponenty . . . 24

5.5.3.4 Dekompozice komponenty . . . 24
5.6 Bezpečnost . . . 24
5.6.1 Definice . . . 24
5.6.2 Požadavky vlastníků systému . . . 25
5.6.3 Testovací scénáře . . . 25
5.6.3.1 Neoprávněný přístup . . . 25
5.6.3.2 Detekce vytěžování dat . . . 26

6 Analýza možností distribuovaných architektur 27
6.1 Architektonické styly . . . 27
6.1.1 Vrstevnatá architektura . . . 27
6.1.2 Microservices . . . 28
6.1.3 SOA . . . 29
6.1.4 Event-driven architektura . . . 30
6.1.4.1 Topologie Mediator . . . 31
6.1.4.2 Topologie Broker . . . 31
6.1.4.3 Hodnocení . . . 32
6.2 Architektonické návrhové vzory . . . 32
6.2.1 Databáze . . . 32
6.2.1.1 Database per service . . . 32
6.2.1.2 Shared database . . . 32
6.2.1.3 Replikace . . . 33
6.2.1.4 Sharding . . . 33
6.2.2 Datová konzistence . . . 33
6.2.2.1 Saga . . . 34
6.2.3 Event sourcing . . . 34
6.2.4 Komunikace . . . 35
6.2.4.1 REST API . . . 35
6.2.4.2 Events . . . 36
6.2.4.3 Messaging . . . 36
6.2.5 Service discovery . . . 36
6.2.5.1 Client-side service discovery . . . 36
6.2.5.2 Server-side service discovery . . . 36
6.2.6 Infrastruktura . . . 37
6.2.6.1 Service per host . . . 37
6.2.6.2 Multiple Services per host . . . 37
6.2.6.3 Service per container . . . 37
6.2.6.4 Serverless . . . 37
6.2.6.5 Sidecar . . . 37
6.2.7 Observability . . . 38
6.2.7.1 Agregace logů . . . 38
6.2.7.2 Agregace aplikačních metrik . . . 38
6.2.7.3 Distributed tracing . . . 38
6.2.8 Nasazení . . . 38
6.2.8.1 Blue/green deployment . . . 39
6.2.8.2 Rolling upgrade . . . 39

6.2.9 Bezpečnost - Autentizace . . . 39
6.2.9.1 Access token . . . 39
6.2.9.2 Klientské certifikáty . . . 39
6.3 Technologie . . . 39
6.3.1 Framework . . . 39
6.3.1.1 Spring . . . 40
6.3.1.2 Vert.x . . . 40
6.3.1.3 Quarkus . . . 40
6.3.1.4 MicroNaut . . . 40
6.3.2 Kontejnerizace . . . 41
6.3.2.1 Docker . . . 41
6.3.2.2 Podman . . . 41
6.3.3 Orchestrace . . . 41
6.3.3.1 Kubernetes . . . 41
6.3.3.2 Docker Swarm . . . 42
6.3.3.3 Apache Mesos a Marathon . . . 42
6.4 The Twelve-Factor App . . . 43

7 Návrh finální architektury 47
7.1 Použitá metodika . . . 47
7.2 Architektura . . . 47
7.2.1 Hlavní komponenty . . . 48
7.2.1.1 User Service . . . 48
7.2.1.2 Web Domain Service . . . 48
7.2.1.3 Keyword Service . . . 50
7.2.1.4 Link Service . . . 50
7.2.1.5 Analytics Service . . . 51
7.2.1.6 Apache Spark . . . 51
7.2.1.7 RabbitMQ messaging . . . 51
7.2.1.8 API Gateway . . . 51
7.2.1.9 Identity and access management . . . 52
7.2.2 Podpůrné komponenty . . . 52
7.2.2.1 Logování . . . 52
7.2.2.2 Monitoring . . . 52
7.3 Vývoj . . . 54
7.3.1 Mikroslužby . . . 54
7.3.2 Podpůrné služby . . . 55
7.3.3 Apache Spark . . . 55
7.4 DevOps . . . 55
7.4.1 Continuous integration . . . 56
7.4.2 Continuous delivery . . . 56
7.4.3 Continuous deployment . . . 57
7.4.4 Orchestrace . . . 57
7.4.5 Infrastruktura . . . 58
7.5 Kontrola testovacích scénářů . . . 59
7.6 Kontrola principů Twelve-factor app . . . 61

8 Proof of concept **63**
8.1 Prostředí . . . 63
8.2 Technologie . . . 63
8.3 Implementované procesy . . . 64
8.3.1 Získávání odkazů z domén . . . 64
8.3.2 Analýza zpětných odkazů . . . 64
8.3.3 API pro získání nejpopulárnějších webových domén . . . 65
8.4 Ověření testovacích scénářů . . . 66
8.4.1 TC 1 - Selhání komponenty . . . 66
8.4.1.1 Popis testu . . . 66
8.4.1.2 Naměřené výsledky . . . 66
8.4.1.3 Hodnocení . . . 67
8.4.2 TC 2 - Load balancing . . . 67
8.4.2.1 Popis testu . . . 67
8.4.2.2 Naměřené výsledky . . . 67
8.4.2.3 Hodnocení . . . 68
8.4.3 TC 3, TC 4 - Nasazení nové verze komponenty . . . 68
8.4.3.1 Popis testu . . . 68
8.4.3.2 Naměřené výsledky . . . 68
8.4.3.3 Hodnocení . . . 69
8.4.4 TC 5 - Prioritizace analytických požadavků . . . 69
8.4.5 TC 6 - Velké množství připojených uživatelů . . . 69
8.4.5.1 Popis testu . . . 69
8.4.5.2 Naměřené výsledky . . . 69
8.4.5.3 Hodnocení . . . 70
8.4.6 TC 7 - Automatické škálování . . . 70
8.4.6.1 Popis testu . . . 70
8.4.6.2 Naměřené výsledky . . . 70
8.4.6.3 Hodnocení . . . 71
8.4.7 TC 8 - Celkový stav systému . . . 71
8.4.8 TC 9 - Podrobnosti chybového stavu . . . 72
8.4.8.1 Popis testu . . . 72
8.4.8.2 Naměřené výsledky . . . 72
8.4.8.3 Hodnocení . . . 72
8.4.9 TC 10 - Změna konfigurace . . . 72
8.4.9.1 Popis testu . . . 72
8.4.9.2 Naměřené výsledky . . . 73
8.4.9.3 Hodnocení . . . 73
8.4.10 TC 11 - Dekompozice komponenty . . . 73
8.4.10.1 Popis testu . . . 73
8.4.10.2 Naměřené výsledky . . . 73
8.4.10.3 Hodnocení . . . 74
8.4.11 TC 12 - Neoprávněný přístup . . . 74
8.4.11.1 Popis testu . . . 74
8.4.11.2 Naměřené výsledky . . . 74
8.4.11.3 Hodnocení . . . 75

8.4.12 TC 13 - Detekce těžení dat 75
8.4.12.1 Popis testu 75
8.4.12.2 Naměřené výsledky 75
8.4.12.3 Hodnocení 76
8.5 Nalezené nedostatky 76

9 Závěr **77**
9.1 Možnosti dalšího postupu 77

Literatura **79**

A Seznam použitých zkratek **83**

B Python kód zátěžových testů pro nástroj Locust **85**
B.1 Testovací scénář 1 85
B.2 Testovací scénář 3,4 85
B.3 Testovací scénář 6 85
B.4 Testovací scénář 11 86
B.5 Testovací scénář 13 86

C Seznam digitálních příloh **87**

Seznam obrázků

4.1 Diagram komponent současné architektury 9

6.1 Průběh požadavku ve vrstevnaté architektuře [25] 28
6.2 Microservices - Jednoduchý eshop 29
6.3 SOA vs Microservices [29] 30
6.4 Event-driven architektura - Mediator Topology [25] 31
6.5 Event-driven architektura - Broker Topology [25] 32
6.6 Saga pattern 34
6.7 Event sourcing pattern [18] 35
6.8 Sidecar pattern [3] 38
6.9 Komponenty Kubernetes [19] 42
6.10 Architetura Apache Mesos 43
6.11 Twelve factor app - One codebase, many deploys [32] 44
6.12 Twelve factor app - Škálování [32] 45

7.1 Diagram komponent navrhované architektury 49
7.2 Zpracování logů v systému 53
7.3 Sběr aplikačních metrik v systému 53
7.4 Model GitFlow 54
7.5 Devops - Grafické znázornění [8] 56
7.6 Architektura automatického škálování 58

8.1 Diagram procesu získávání odkazů konkrétní domény 64
8.2 Diagram procesu analýzy zpětných odkazů 65
8.3 Diagram procesu REST API požadavku uživatele 66
8.4 Výsledek měření TC 1 v programu Locust 67
8.5 Úspěšný běh CI/CD pipeliny 68
8.6 Výsledek měření TC 3,4 v programu Locust 69
8.7 Výsledek měření TC 6 v programu Locust 70
8.8 Snímek obrazovky nástroje Grafana 71
8.9 Log chyby v nástroji Kibana 72
8.10 Výsledek měření TC 11 v programu Locust 73
8.11 Neautorizovaný požadavek v aplikaci Postman 74
8.12 Podrobnosti nepovedeného pokusu o přihlášení 74
8.13 Výsledek měření TC 13 v programu Locust 75
8.14 Odpověď na požadavek po překročení limitu v aplikaci Postman 75

Seznam tabulek

3.1 Klasifikace potřeb uživatele 6

5.1 Dostupnost dle počtu devítek [9] 16
5.2 Testovací scénář 1 - Selhání komponenty 17
5.3 Testovací scénář 2 - Load balancing 17
5.4 Testovací scénář 3 - Nasazení nové verze komponenty 19
5.5 Testovací scénář 4 - Oprava kritické chyby 19
5.6 Testovací scénář 5 - Prioritizace analytických požadavků 20
5.7 Testovací scénář 6 - Velké množství uživatelů 21
5.8 Testovací scénář 7 - Automatická orchestrace instancí 22
5.9 Testovací scénář 8 - Celkový stav systému 23
5.10 Testovací scénář 9 - Podrobnosti chybového stavu 23
5.11 Testovací scénář 10 - Změna konfigurace 24
5.12 Testovací scénář 11 - Dekompozice komponenty 24
5.13 Testovací scénář 12 - Neoprávněný přístup 25
5.14 Testovací scénář 13 - Detekce vytěžování dat 26

7.1 Přehled splnění testovacích scénářů 60
7.2 Přehled splnění principů Twelve-Factor app 62

Kapitola 1

Úvod

1.1 Popis

Tato práce se zabývá problematikou návrhu softwarové architektury systému, který slouží k analýze dat online trhu. Práce je rozdělena do čtyř hlavních částí: analýzy současného systému, sběru nefunkčních požadavků, teoretickému návrhu nové architektury a vytvoření prototypu. Na základě identifikovaných problémů současné architektury a požadavků vlastníků systému je vytvořena kompletně nová architektura bez ohledu na migraci z původního systému. Kvalita architektury je formálně verifikována pomocí testovacích scénářů, které jsou použity proti vytvořenému prototypu. Zajímavostí práce je použití technologií specializovaných na práci s velkým objemem dat.

Výsledná architektura splňuje požadavky nejmodernějších distribuovaných systémů a podporuje pomocí DevOps procesů v maximální míře automatizaci a možnosti dalšího rozvoje. Součástí návrhu jsou i všechny podpůrné komponenty (logování, monitoring atd.), které jsou potřeba při produkčním provozu.

1.2 Motivace

V průběhu celého studia na vysoké škole jsem si měl možnost zkusit velkou škálu IT odvětví a nejvíce mne zaujaly právě softwarové architektury. Tato práce je příležitostí zkusit si nejen navrhnout distribuovanou architekturu komplexního systému, ale také se podrobněji naučit s pokročilými nástroji orchestrace, moderními systémy monitoringu nebo NoSQL databázemi.

Businessová doména systému má dle mého názoru velký potenciál, ale současná architektura zaostává za moderními systémy. Navržená architektura by měla pomoci v realizaci businessových cílů a urychlit tzv. time-to-market, který je často velmi kritický pro úspěch celé aplikace.

Kapitola 2

Zadání

Cílem práce je návrh nové softwarové architektury systému pro zpracování a analýzu velkého množství dat z online trhu, která plní požadavky klíčových vlastníků systému. Součástí návrhu architektury je analýza současného stavu, požadavků na systém, identifikace/výběr klíčových požadavků, které formují architekturu, výčet nutných kompromisů a jejich odůvodnění, návrh klíčových komponent a jejich zodpovědností, návrh technologií a návrh procesu vývoje a provozu (postupů, rolí, metod, nástrojů).

Požadavky na novou architekturu:

- Umožňuje se při vývoji soustředit na konkrétní věcnou oblast nebo technický aspekt, tj. mít možnost odděleně vyvíjet, testovat a nasazovat pouze část aplikace bez vlivu na jiné části systému.
- V maximální možné míře automatizuje prvky testování, sestavování a nasazování dílčích částí aplikace.
- Je provozována takovým stylem, který:
  - Dílčí aplikace/služby abstrahuje od podkladového serverového prostředí (fyzický server, VPS). Aplikace/služby jsou lokačně transparentní, mohou být libovolně přesouvány.
  - Poskytuje jasný a ucelený pohled na stav celého systému.
  - Je horizontálně škálovatelný.
  - Je vysoce dostupný.
  - Je robustní - počítá se s výpadky částí systému, je tzv. fault-tolerant.
- Minimalizuje nároky/náklady na výpočetní výkon.
- Minimalizuje nároky/náklady na lidskou obsluhu.
- Je bezpečná - dostatečně chrání data systému a funkce před neoprávněným přístupem.
- Dokáže flexibilně reagovat na změny v čase.
- Dokáže pracovat s velkým množstvím záznamů (v řádu jednotek a desítek miliard).

- Dokáže data a práci distribuovat mezi heterogenní prostředí výpočetních prostředků.

Současně s návrhem architektury bude vytvořen prototyp (Proof of Concept), který bude demonstrovat reakci architektury na klíčové požadavky.

Kapitola 3

Analýza business case

3.1 Popis aplikace

Z pohledu koncového uživatele se jedná o analytickou aplikaci, pomocí které může zákazník prozkoumat stav a změny na online trhu. Online trhem se rozumí prostředí webu, vyhledávačů, sociálních sítí a dalších online komunikačních kanálů, na kterých probíhá střet poptávky a nabídky. Zákazník je na základě informací získaných z aplikace schopen zhodnotit dopad online marketingu vlastní firmy/projektu nebo analyzovat online marketingovou strategii své konkurence. Obecně lze říci, že aplikace sbírá/těží informace z internetových vyhledávačů, sociálních sítí a samotných webů. Ty následně zpracovává a prezentuje uživateli ve formě ucelených reportů složených ze statistik, přehledů a grafů.

Systém je orientován čistě na český trh, ale s budoucím plánem expanze do zahraničí. Získává tedy informace čistě z českých stránek a využívá dva nejpopulárnější zdejší vyhledávače: Google[1] a Seznam[2].

3.2 Cílová skupina

Cílovou skupinou aplikace jsou především firmy. Nezáleží na jejich velikosti, může se jednat o malý startup nebo korporaci. Zde je kompletní výčet cílových skupin s jejich ambicemi:

- **Startup** - Potřebuje získat informace jak moc velký zájem je o jeho nový produkt/službu a jak si vede v online marketingu. Chce zjistit, jak by mohl produkt více propagovat.
- **Eshop** - Zkoumá konkurenční internetové obchody a zjišťuje, v jakých oblastech zaostává nebo vyniká. Na základě těchto informací je schopen adekvátně upravit nabídku, aby maximalizoval svůj marketingový potenciál.
- **Marketing** - Marketing lze rozdělit do 2 kategorií. První kategorií je B2C[3] marketing, kde jsou ambice v zásadě stejné jako u eshopů. Druhou kategorií je B2B[4] martketing,

[1] `https://www.google.com/`
[2] `https://www.seznam.cz/`
[3] B2C - Business to Customers
[4] B2B - Business to Business

který analyzuje trh a získává tím informace o potenciálních klientech. Informace o svých konkurentech mohou pomoci v získání konkurenční výhody.

- **Obchodník** - Hledá potencionální klienty, podobně jako B2B marketing. Zároveň potřebuje podklady pro návrh obchodní strategie.
- **Podnikatel** - Chce kontrolovat vlastní výkon a porovnávat ho s konkurencí. Potřebuje jednoduchý a intuitivní nástroj pro analýzu svého potenciálu v online marketingu. Zjišťuje v jakých aspektech je horší/lepší v porovnání s konkurencí.

Obecně lze shrnout potřeby uživatelů do následující tabulky:

Klasifikace	**Popis**	**Příklad**
Stav	Chci zjistit stav nebo získat data	Kdo je nejsilnější v tématu X? Jaká je hledanost klíčového slova Y? Jak je na tom web Z?
Rizika	Chci definovat rizika	Kdo je můj největší konkurent? Jak se prezentuje? Kdo používá nejvíce internetové reklamy?
Příležitosti	Chci objevit příležitosti	Co dělají úspěšné weby? Čím mohu zlepšit svůj marketing?
Důvody	Chci se dozvědět důvody	Proč má web X vysokou návštěvnost? Odkud vedou odkazy na web Y? Jaký je důvod úspěšnosti webu firmy Z?

Tabulka 3.1: Klasifikace potřeb uživatele

3.3 Funkce uživatelské aplikace

3.3.1 Report webu

Report shrnuje klíčové marketingové statistiky charakterizující výkon webu, např. impakt v přirozeném hledání, PPC[5] (včetně odhadu nákladů) nebo složení odkazového portfolia. Weby jsou kategorizovány dle účelu (např. Magazín) a tématu (např. Finance). Weby lze mezi sebou porovnávat. Report lze rozdělit do tří důležitých kategorií: Klíčová slova, Návštěvnost a Odkazy

[5]PPC - Pay per click reklama

3.3.1.1 Klíčová slova

Tato kategorie je přehledem, který říká z jakých klíčových slov se daná webová stránka objeví ve výsledcích vyhledávače. Příkladem může být, zda se při vyhledávání klíčovým slovem „auto“ zobrazí ve výsledcích webová stránka „www.volkswagen.cz“. Je tu tedy přehled všech slov, ze kterých je možné danou stránku najít ve vyhledávačích, včetně jejich vývoje v čase a na jakém místě v seznamu se stránka objeví. Součástí je přehled pozic ostatních firem ve stejné business doméně. Tato kategorie je rozdělena do dvou částí: přehled klíčových slov pro organického vyhledávání a přehled klíčových slov pro zobrazení webové stránky v reklamách.

3.3.1.2 Návštěvnost

Sekce návštěvnost říká o dané webové stránce měsíční odhad, kolik ji navštíví lidí. Je to opět rozděleno do dvou kategorií: návštěvy z organického vyhledávání a návštěvy z placených kanálů (reklam). Nabízí pohled pro oba podporované vyhledávače Google a Seznam, včetně historických dat.

3.3.1.3 Odkazy

Kategorie „Odkazy“ nabízí uživateli přehled z jakých stránek je na danou stránku odkazováno a zároveň kam je odkazováno z vyhledávané stránky. Tento ukazatel je velmi důležitý pro pozici stránky v organickém vyhledávání, jelikož počet odkazů na stránku z jiných webů je jedním z hlavních aspektů řazení.

3.3.2 Report klíčového slova

Report shrnuje základní statistiky významu klíčového slova v rámci vyhledávače Google a Seznam včetně přehledu konkurence, která se na dané klíčové slovo/množinu slov soustředí v organickém hledání a PPC.

Hledanost klíčového slova je měsíční odhad, kolikrát se dané slovo objevilo v organickém vyhledávání nebo placených reklamách. Obsahuje grafický přehled vývoje hledanosti daného slova pro oba podporované vyhledávače.

Velmi důležitým aspektem pro každé klíčové slovo je list a pořadí stránek, které pro dané slovo vyhledávač vrátí. Aplikace pro klíčové slovo tyto stránky zaznamenává a ukazuje uživateli přehled nejvýše umístěných stránek a reklam. Při sloučení s historickými daty vzniká zajímavá vizualizace vývoje umístění jednotlivých webových stránek.

3.3.3 Report vlastníka

Report agreguje statistiky jednotlivých webů přes stejného vlastníka (z pohledu doménového registru).

3.3.4 Sledování

Umožňuje podrobné sledování vlastních i konkurenčních webových projektů v čase včetně změn přirozených a placených pozic na jednotlivá klíčová slova a celkový impakt v daném segmentu.

3.3.5 Globální přehledy

Report agreguje klíčové statistiky napříč celými segmenty (např. Poskytovatel internetu).

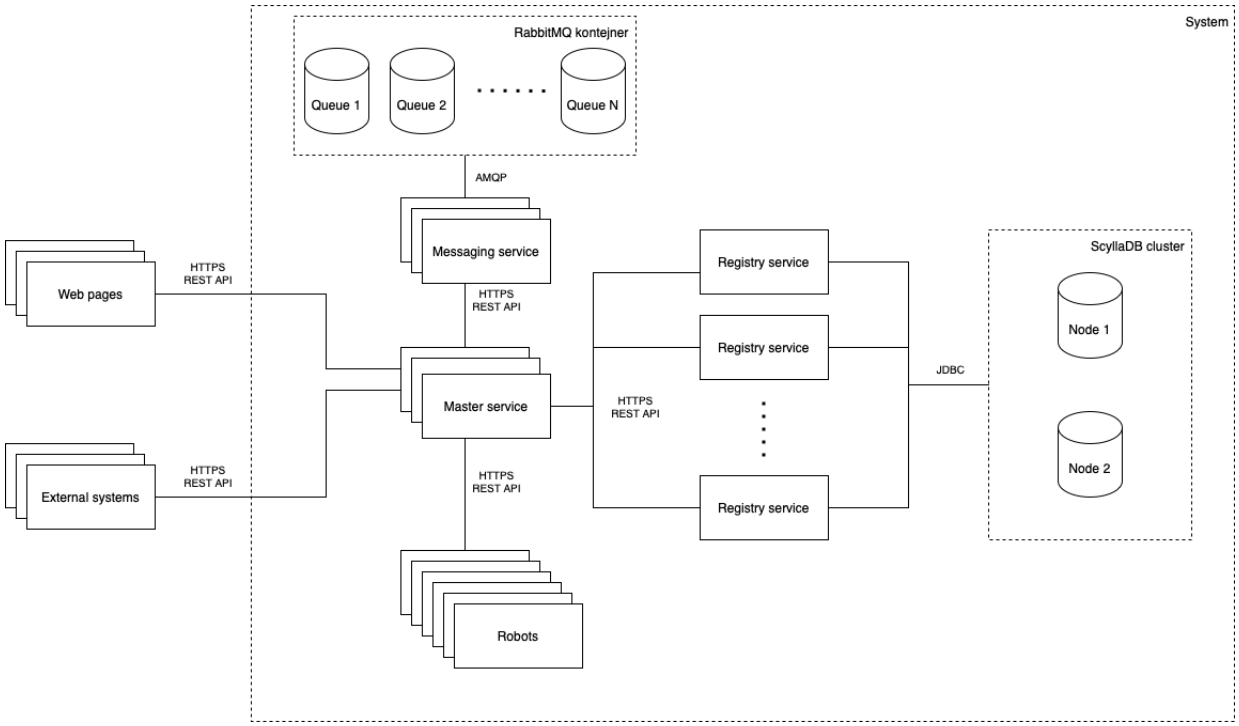
Kapitola 4

Analýza současného řešení

4.1 Komponenty a architektura

4.1.1 Diagram komponent

Následující diagram znázorňuje komponenty současného systému. Systém můžeme rozdělit do šesti logický celků: Master service, Registry service, Messaging service, ScyllaDB databázový cluster, RabbitMQ kontejner a roboti. Roboti se starají o těžení dat z internetu a nejsou předmětem této práce, takže je jejich struktura značně zjednodušena. Současná architektura by se dala označit za distribuovanou vrstevnatou architekturu.



Obrázek 4.1: Diagram komponent současné architektury

4.1.2 Master service

Master service je řídící jednotkou celého systému. V systému může běžet několik instancí této služby, ale na základě její konfigurace se každá služba specializuje na jiný typ úlohy. Je napsána v jazyce Java ve frameworku Spring Boot[1]. Jsou zde spuštěné úlohy, které řídí interní práci nad daty spolu s jejich analýzou a plánují úlohy pro roboty. Master service zároveň poskytuje konfiguraci robotům při jejich spuštění a funguje jako zdroj informací pro všechny části systému. Poskytuje dvě důležité REST API. První je pro komunikaci s vnějším světem tzn. s webovou aplikaci/externím systémem. Druhým je REST API pro komunikaci s roboty, kde je jim zadávána práce. Master service je zároveň konzumentem REST API Registry service pro práci s daty a REST API Messaging service pro práci s frontami úloh. Můžeme si ji tedy představit jako jednu komplexní komponentu se všemi funkcemi systému.

4.1.3 Registry service

Registry service tvoří abstrakci nad databází. Je napsána v jazyce Java ve frameworku Spring Boot. Touto komponentou prochází naprosto všechny požadavky pro práci s daty a řídí jejich poskytování a ukládání. Výjimečně zde mohou být implementován části výpočtů nad daty v rámci jejich optimalizace. Drtivá většina analýzy dat je ale implementována v rámci Master služby. Poskytuje REST API pro master uzel.

4.1.4 Messaging service

Messaging service je komponenta tvořící abstrakci nad frontou zpráv implementovaných pomocí RabbitMQ. Je napsána v jazyce Java ve frameworku Spring Boot. Kromě základních operací nad frontami úloh poskytuje i business logiku. Např. nedovoluje vložit do fronty duplicitní úlohu pro roboty. Pro komunikaci s Master service vystavuje komponenta REST API. Služba může v systému běžet ve více instancích, kde každá instance se specializuje na různé fronty dle konfigurace z databáze. Můžeme si ji tedy představit, podobně jako u Master služby, jako jednu komplexní komponentu starající se o všechny fronty.

4.1.5 RabbitMQ

RabbitMQ[2] je populární message-broker software, který mimo jiné implementuje AMQP[3]. Právě tento protokol je využíván v našem systému. V RabbitMQ kontejneru je vytvořeno přibližně 50 front obsahujících úkoly rozdělených dle business kritérií. Fronty jsou perzistentní a prioritizované. Tato komponenta je obsluhována pouze Messaging komponentou, ovšem samotné zprávy/úlohy ukládané do front jsou generovány v Master komponentě.

[1] `https://spring.io/projects/spring-boot`
[2] `https://www.rabbitmq.com`
[3] AQMP - Advanced Message Queuing protocol

4.1.6 ScyllaDB cluster

ScyllaDB[4] je distribuovaná big data NoSQL databáze. Konkrétně se jedná o úložiště typu „wide-column store“[5]. V současné době je databázový cluster provozován jako vývojové prostředí a z ekonomických důvodů jsou použity pouze dva uzly bez nastavené replikace. Instance ScyllaDB je rozdělena do několika keyspaců dle věcných oblastí a služeb. Celková velikost databáze se pohybuje kolem 1 TB. Největší tabulky mají až 2 miliardy záznamů a ostatní tabulky obsahují jednotky miliónů záznamů.

4.1.7 Roboti

Roboti se starají o těžení dat z internetu. Jedná se o vyšší desítky serverů, kde každý má svoji specializaci. Architektura a fungování robotů není předmětem této práce, proto je pohled na ně zjednodušen. Z pohledu této práce je důležité, že jsou to konzumenti REST API na master komponentě.

4.1.8 Webová aplikace

Webová aplikace slouží pro vizualizaci a prezentaci výsledků uživatelům. Je napsána v jazyce PHP ve frameworku Nette[6]. Architektura webové aplikace není předmětem této práce, proto je pohled na ni značně zjednodušen a nebude ji věnována velká pozornost. Z pohledu této práce je důležité, že je konzumentem REST API master komponenty.

4.2 Hardware

Systém je provozován na vlastnoručně spravovaných fyzických serverech. Celkem jich je pět a služby jsou do nich logicky rozděleny podle typu: Databáze, Služby (Master, Registry, Messaging), Webová aplikace (Relační DB, Backend), RabbitMQ, Vybraní roboti. Na každém stroji běží zpravidla více typů služeb. Všechny servery mají operační systém Debian Linux. Samotné servery jsou spravovány nástrojem Ansible[7], který se stará o jejich konfiguraci a instalaci požadovaných programů.

4.3 Vývoj

4.3.1 Vývojové prostředí

Pro samotný vývoj je využíváno IDE IntelliJ Idea community edition[8]. Jedná se o jeden z nejpopulárnějších IDE současnosti a slouží především k vývoji v jazyce Java. V našem systému tedy pro všechny služby. Z pohledu architektury není používané IDE relevantní.

[4] `https://www.scylladb.com`
[5] `https://blog.logrocket.com/nosql-wide-column-stores-demystified/`
[6] `https://nette.org/cs/`
[7] `https://www.ansible.com`
[8] `https://www.jetbrains.com/idea/`

4.3.2 Version Control System

Pro verzování zdrojových kódů je využit především Git, konkrétněji Gitlab[9]. V začátcích projektu se jako VCS používalo SVN a v dnešní době je ještě malá část systému verzována pravě v něm. Za primární VCS se v současné době bere Git a je snaha do něho zmigrovat i projekty, které jsou v SVN. Každá služba (Master, Registry, Messaging) má vlastní repozitář.

4.3.3 Metodika

Pro management vývoje je použita metodika SCRUM. Jedná se o iterativní a agilní způsob vývoje, který klade důraz na postupné vytváření software v menším týmu a bere v potaz to, že se zadání může během projektu měnit. Jako podpůrný nástroj pro tento styl vývoje je použita webová aplikace Trello[10]. Architektura systému by měla být připravená tak, aby vyhovovala způsobu agilního vývoje.

4.4 DevOps

4.4.1 Vytváření artefaktů

Pro automatické vytváření spustitelných artefaktů se používá Gitlab CI/CD pipeline. Jako buildovací nástroj se používá Maven[11]. Výsledkem jsou soubory JAR.

4.4.2 Testování

Automatické testování probíhá také v Gitlab CI/CD pipeline. Používá se opět buildovací nástroj Maven. Reporty z testování se negenerují a výsledky včetně logů jsou dostupné pouze v logu pipeliny.

4.4.3 Release management

V rámci Gitlab CI/CD pipeline se vytvořený JAR artefakt zabalí do DEB balíčku a je následně nahrán do interního repozitáře, odkud jsou následně distribuovány.

4.4.4 Nasazení

Nasazení probíhá stáhnutím DEB balíčků z interního repozitáře a pomocí startup skriptů se nainstaluje a spustí JAR soubor s aplikací. Tato část není automatizovaná. Je dělána ručně administrátorem.

[9] `https://gitlab.com`
[10] `https://trello.com`
[11] `https://maven.apache.org`

4.4.5 Monitoring

Monitoring sítě a jednotlivých serverů je realizován pomocí nástroje Zabbix[12]. Jedná se o opensource nástroj, který nabízí široké možnosti monitoringu. V tomto systému je to použito čistě pro notifikace v případě výpadků. Samotné aplikace nejsou nijak monitorované a je tedy velmi náročné zjistit stav systému.

4.5 Zátěž systému

4.5.1 Externí zátěž

Současný systém je provozován v rámci několika desítek uživatelů. Toto číslo není pro aktuální řešení nijak kritické. Hlavní zatížení systému pochází z komunikace mezi jednotlivými službami a roboty. Jelikož dochází k neustálému extrahování dat z internetu a analýzou nad nimi, tak se zátěž na REST API jednotlivých služeb pohybuje v rámci stovek požadavků za sekundu. Zde již současný systém může mít problémy.

4.5.2 Databázový cluster

Celková velikost databáze se pohybuje kolem 1TB dat a je rozdělena do dvou částí(partitions), které mají oba podobnou velikost. Samotná data nemají žádné TTL[13], takže zůstávají uložena navždy. Neustálé těžení informací roboty vytváří konstantní tok nových dat do databáze a tím lineárně roste její velikost V současné době je zátěž databáze tvořena cca ze 75% zápisovými požadavky a 25% čtením dat.

4.6 Identifikované problémy

4.6.1 Vysoká závislost služeb

Systém je v zásadě implementován jako monolitická aplikace. Drtivá většina funkcionalit je implementována ve službách Master a Registry, které jsou na sobě závislé, tzv. změna v jedné ovlivní i druhou službu. Tato provázanost je v distribuovaných systémech nevhodná. Je velmi obtížné nasazovat změny jen pro jednu službu bez nutnosti měnit celý systém.

4.6.2 Distribuce závislá na platformě

V současném řešení se aplikace (JAR soubory) zabalí do DEB balíčku a jsou nahrány do interního repozitáře. Tímto řešením distribuce jsme limitováni pouze na platformy podporující DEB balíčky a nejsme schopni nasadit aplikaci mimo Debian Linux.

[12] `https://www.zabbix.com`
[13] TTL - Time to live

4.6.3 Manuální nasazování

Systém je nasazován pomocí DEB balíčků a startup skriptů. Tento proces je prováděn manuálně a je tedy náchylnější na potencionální chyby. Zároveň zde není možně automaticky nasazovat/vypínat služby při různých zátěžích.

4.6.4 Škálovatelnost a robustnost

Škálovatelnost systému říká, jak je systém schopen reagovat na razantní změnu zátěže. Architektura současného systému škálovatelná moc není. Jak naznačuje diagram komponent (viz Obrázek 4.1), všechny požadavky a úkoly procházejí přes Master Service. Při vzrůstajícím počtu nejen uživatelů, ale i počtu úkolů pro analýzu dat nebo úkoly pro roboty rapidně narůstají nároky na Master Service. Master Service není bezstavová služba, jelikož může provádět dlouhotrvající analýzy, kde je důležité si pamatovat mezivýsledky. Lze tedy považovat Master Service za úzké hrdlo celého systému, které není jednoduše škálovatelné.

Robustností systému se myslí jeho fault-tolerance, tzv. jak si systém poradí v případě selhání některé z jeho částí. V této oblasti současná architektura příliš nevyniká. Neexistuje žádná orchestrace procesů, která by se starala o opětovné spuštění/restart v případě selhání. Můžeme tedy Master service označit za single-point-of-failure. V případě, že služba selže, stane se část systému za kterou služba odpovídá nepoužitelnou.

Databázový cluster ScyllaDB v současné konfiguraci lze považovat za dostatečně škálovatelný pouze za předpokladu, že není problém v případě velkého nárůstu dat přidat nový node. Stejná věc nelze říci o robustnosti. V databázových clusterech je robustnost řešena replikací. Tím pádem v případě selhání jednoho z nodů clusteru uživatel nic nepozná, jelikož stejná data jsou replikována mezi různými nody. Současný databázový cluster replikaci nastavenou nemá, takže není odolný proti výpadku některých serverů.

Kontejner s RabbitMQ frontami sám o sobě škálovatelný a robustní není. Pro zajištění těchto vlastností by systém neměl mít pouze jeden kontejner, ale RabbitMQ cluster se zapnutou replikací a několika nody.

4.6.5 Monitoring a logování

V aktuálním stavu není provoz systému dostatečně monitorován a je těžké se dozvědět jeho aktuální stav. Jak již bylo zmíněno v sekci 4.4.5, je pro monitoring samotných serverů použit nástroj Zabbix. Ten sice monitoruje fyzický stav jednotlivých serverů a je schopný ho hezky vizualizovat, ale vůbec neřeší monitoring samotného systému. Může tedy docházet k výpadkům jednotlivých služeb bez toho, abychom měli způsob, jak to jednoduše zjistit. Zároveň nejsme schopni zjistit, jaké verze jednotlivých služeb běží a na jakých serverech. To může způsobit nekompatibilitu jednotlivých služeb.

V současném systému kompletně chybí komplexnější řešení pro sběr logů a jejich vizualizaci. Logy jsou dostupné pouze na samotném serveru, kde daná služba běží. Je tedy velmi obtížní zjistit příčinu chyb, které jsou distribuované napříč celým systémem.

Kapitola 5

Nefunkční požadavky systému

Schopnost systému splnit nefunkční požadavky je dána především zvolenou softwarovou architekturou. Z tohoto důvodu je důležité si jednotlivé požadavky formálně zadefinovat, určit si konkrétní požadavky na náš systém a sestavit testovací scénáře.

Testovací scénář je specifikace událostí, která umožní zvolenou architekturu formálně otestovat, zda splňuje nároky vlastníků systému a ověřit, jak reaguje na různé interní/externí podněty. Definici scénáře můžeme dle knihy [2] rozdělit do 6 hlavních částí:

- ID - Jednoznačný identifikátor scénáře.
- Akce - Událost v našem systému.
- Zdroj akce - Zdroj události (externí systém, člověk...).
- Prostředí - Okolnosti za jakých scénář probíhá (ve vývoji, velká zátěž...).
- Artefakt - Konkrétní část systému, na který je akce mířena.
- Reakce - Popis reakce systému/artefaktu na příchozí akci.
- Měřitelný výsledek - měřitelný výsledek reakce (latence systému, délka výpadku...).

5.1 Dostupnost

5.1.1 Definice

Dostupnost lze definovat jako vlastnost softwaru říkající v jakém časovém rozmezí je systém připraven vykonávat práci. Pokud definici obrátíme, lze také dostupnost definovat jako doplněk času v jakém software není schopen svoji práci vykonávat. Systém pod plánovanou odstávkou se bere jako nedostupný.

Velmi často je pojem dostupnost nahrazován spolehlivostí, ale existují mezi nimi drobné rozdíly. Spolehlivost označuje dobu, kdy je software schopen pracovat bez selhání komponent, což nutně nemusí znamenat, že systém není dostupný.

Matematicky lze dostupnost A vyjádřit následujícím vzorcem:

$$A = (T - \sum_{i} F_{\mathrm{i}})/T \tag{5.1}$$

kde T je celkový čas, ve kterém nás dostupnost zajímá a F_i je čas nedostupnosti při výpadku *i*.

Jednotlivé úrovně dostupnosti jsou často označovány počtem devítek ve výsledku uvedeného vzorce. Rozdíly v dostupnosti typu 0,1% nebo 0,01% se můžou zdát jako zanedbatelné, ale při převodu procentuálního vyjádření na čas jsou rozdíly razantní.

Dostupnost	**Nedostupnost v roce**	**Nedostupnost za 24h**
90,0%	36,5 dní	2,4 h
99,0%	3,65 dne	14 min
99,9%	8,76 h	86 s
99,99%	52,6 min	8,6 s
99,999%	5,25 min	0,86 s
99,9999%	31,5 s	8,6 ms

Tabulka 5.1: Dostupnost dle počtu devítek [9]

Obecně lze říci, že systém je vysoce dostupný právě tehdy, když má dostupnost větší nebo roven 99,999%. [2]

5.1.2 Požadavky vlastníků systému

Na základě business cílů je po systému požadována vysoká dostupnost 99,999%. Specifickou vlastností systému je, že na rozdíl od většiny systémů není klíčová dostupnost pouze z hlediska přístupu uživatelů, ale zároveň z hlediska výpočtů a poskytování práce pro roboty. Jelikož jde o systém zpracovávající „big data“ z dat sesbíraných z internetu, tak jakýkoli delší výpadek může způsobit ztrátu důležitých dat. Zároveň jakýkoli výpadek uživatelské části systému může mít velký dopad na reputaci systému u koncových uživatelů.

5.1.3 Testovací scénáře

5.1.3.1 Selhání komponenty

V rozsáhlých distribuovaných systémech musíme brát selhání komponenty jako běžnou věc a systém na ni musí být připraven. Tento scénář lze použít pro všechny komponenty systému bez ohledu na to, jestli jde o hardwarovou nebo softwarovou komponentu.

ID	TC 1
Akce	Selhání
Zdroj akce	Softwarová komponenta
Prostředí	Standardní provoz
Artefakt	Businessová komponenta
Reakce	• Detekce selhání komponenty • Log chyby • Spustit novou instanci stejného typu komponenty
Měřitelný výsledek	• Detekce selhání komponenty do 60 s • Nová instance úspěšně spuštěna do 120 s

Tabulka 5.2: Testovací scénář 1 - Selhání komponenty

5.1.3.2 Load balancing požadavků

V distribuovaných systémech typicky běží více instancí stejné komponenty. Je důležité, aby systém uměl rovnoměrně požadavky distribuovat mezi jednotlivé instance.

ID	TC 2
Akce	10 požadavků na systém
Zdroj akce	Přihlášený uživatel
Prostředí	Standardní provoz
Artefakt	Businessová komponenta
Reakce	• Požadavky jsou distribuovány rovnoměrně mezi instance • Je použit mechanismus Round Robin[1]
Měřitelný výsledek	Každá služba zpracuje přesně 10/N požadavků, kde N je počet spuštěných instancí

Tabulka 5.3: Testovací scénář 2 - Load balancing

5.2 Složitost nasazení

5.2.1 Definice

Pod termínem „Složitost nasazení“ si lze představit více věcí. V této práci se pracuje s následující definicí:

Složitost nasazení je čas, jak dlouho trvá úspěšně nasadit novou verzi komponenty do produkčního prostředí od dokončení vývoje.

Tento atribut kvality architektury nesouvisí přímo s funkčností systému, ale v dnešní době se stává čím dál tím více důležitější. Schopnost častého vydávání nových funkcionalit a rychlost oprav chyb má velký dopad na konkurence schopnost systému. Zároveň by vydávání nových verzí komponent nemělo mít žádný dopad na jejich dostupnost. Automatická orchestrace nasazení oproti manuální také dramaticky snižuje možnost zanesení chyb do produkčního prostředí.

5.2.2 Požadavky vlastníků systému

Složitost nasazení je jedním z hlavních požadavků na navrhovaný systém. Celé nasazení by mělo fungovat stylem "kliknutí na jedno tlačítko". Celý proces by měl být plně automatizován, na což musí být architektura připravena. Nasazovaný artefakt by měl postupně projít následujícími prostředími:

1. vývojové - pouze samotný artefakt,
2. integrační - artefakt a mockované ostatní služby,
3. staging - kompletní systém pro testování,
4. produkční - kompletní systém pro produkci.

Na každém prostředí je artefakt testován jinými typy testů, kterým se práce věnuje v sekci Testovatelnost.

5.2.3 Testovací scénáře

5.2.3.1 Nasazení nové verze komponenty

Nasazování nových plánovaných verzí by mělo být běžnou praxí v jakémkoli komplexnějším systému. V tomto scénáři jde o situaci, kdy chceme dostat novou verzi do staging prostředí pro otestování. Následné testování se do výsledného času nezapočítává.

ID	TC 3
Akce	Spuštění akce "Nasadit do testovacího prostředí"
Zdroj akce	Product owner
Prostředí	Staging prostředí
Artefakt	Businessová komponenta
Reakce	• Spuštění N instancí nových komponent • Monitoring všech nových komponent • Přenastavení routování požadavků pouze do nových komponent • Zastavení N instancí starých komponent
Měřitelný výsledek	• Úspěšně provedené jednotkové a integrační testy • Nasazení proběhlo do 60 min • Úspěšné se nasadilo alespoň 90% komponent

Tabulka 5.4: Testovací scénář 3 - Nasazení nové verze komponenty

5.2.3.2 Oprava kritické chyby v produkčním prostředí

Velmi často je potřeba opravit v produkčním prostředí bug kategorie „kritický“ velmi rychle. To vše bez vlivu na dostupnost systému.

ID	TC 4
Akce	Spuštění akce "Nasadit do produkčního prostředí"
Zdroj akce	Developer
Prostředí	Produkční prostředí, Standardní provoz
Artefakt	Businessová komponenta
Reakce	• Spuštění N instancí nových komponent • Monitoring všech nových komponent • Automatický test funkčnosti nových komponent • Přenastavení routování požadavků pouze do nových komponent • Zastavení N instancí starých komponent
Měřitelný výsledek	• Úspěšně provedené jednotkové a integrační testy • Nasazení proběhlo do 60 min • Úspěšné se nasadilo alespoň 90% komponent • Neproběhl žádný výpadek funkcionality

Tabulka 5.5: Testovací scénář 4 - Oprava kritické chyby

5.3 Výkonnost

5.3.1 Definice

Výkonnost systému je jednou ze základních metrik pro určení kvality softwarové architektury. V této práci se výkonnost systému bere jako latence vykonání/zpracování požadavku. Tedy je to čas uplynulý mezi přijetím požadavku a jeho dokončením.

5.3.2 Požadavky vlastníků systému

Systém musí obsloužit 2 typy požadavků. Prvním je požadavek samotného uživatele z webového prohlížeče. Tento požadavek by neměl přesáhnout průměrnou latenci 1,7 sekundy. Druhým typem je analytický požadavek vygenerovaný interním systémem. Jelikož analýza může trvat delší dobu a požadavky jsou heterogenní, tak jsou rozděleny do tří kategorií dle priority:

- HIGH - Požadavek musí být vyzvednut ke zpracování do 30 sekund
- MEDIUM - Požadavek musí být vyzvednut ke zpracování do 10 minut
- LOW - Požadavek musí být vyzvednut ke zpracování do 5 dní

Není zde důležitá latence dokončení, ale latence vyzvednutí úlohy ke zpracování na základě priority požadavku.

5.3.2.1 Testovací scénáře

5.3.2.2 Prioritizace analytických požadavků

Hlavní funkcí celého systému jsou analýzy velkého množství dat. Všechny tyto požadavky by měly být zpracovány v daném časovém intervalu.

ID	TC 5
Akce	30 požadavků priority „HIGH“ v minutovém intervalu
Zdroj akce	Generátor analytických požadavků
Prostředí	Standardní provoz
Artefakt	Systém
Reakce	Zpracuje všechny požadavky
Měřitelný výsledek	Každý požadavek je zpracován nejpozději 30 sekund po zafrontování

Tabulka 5.6: Testovací scénář 5 - Prioritizace analytických požadavků

5.3.2.3 Velké množství současně připojených uživatelů

Systém musí zvládat relativně velké množství požadavku v krátkém intervalu.

ID	TC 6
Akce	5 000 požadavků během jedné minuty
Zdroj akce	500 uživatelů
Prostředí	Standardní provoz
Artefakt	Systém
Reakce	Zpracuje všechny požadavky
Měřitelný výsledek	Průměrná latence je pod 1.7 s

Tabulka 5.7: Testovací scénář 6 - Velké množství uživatelů

5.4 Škálovatelnost

5.4.1 Definice

Škálovatelnost je vlastnost, která definuje, jak si je systém schopen poradit pod zvyšující se zátěží. Systém by si měl se zvyšující zátěží stále udržovat výkonnost v daných mezích. Může se jednat o dramatické zvýšení počtu uživatelů nebo o zpracovávání většího množství dat.

Škálování systému můžeme rozdělit do 2 základních kategorií: vertikální a horizontální. Vertikální škálovaní zvyšuje výpočetní výkon(zpravidla CPU a RAM) jednotlivých uzlů. Horizontálním škálováním se zvyšuje počet uzlů systému a tím se může celkový objem práce rozdělit mezi více "workerů". Horizontální škálování lze rozdělit dle modelu Scale Cube[2] do 3 kategorií:

- klonování - klonujeme stále stejné uzly a tím přidáváme systému výkonnost,
- funkční dekompozice - rozdělíme aplikaci do několika menších systému, kde každá část má na starost jinou funkcionalitu,
- datová dekompozice - rozdělíme aplikaci do několika menších systému, kde každá část má na starost rozdílnou část dat.

5.4.2 Požadavky vlastníků systému

Navrhovaný systém musí být horizontálně škálovatelný a musí být schopen distribuovat práci mezi jednotlivými uzly tak, aby byl systém schopen zpracovat velké množství dat. Škálování by mělo probíhat automaticky.

[2] `https://microservices.io/articles/scalecube.html`

5.4.3 Testovací scénáře

5.4.4 Automatická orchestrace počtu spuštěných instancí na základě metrik

Pro zachování maximalizace efektivity systému a minimalizace nákladů je potřeba automaticky orchestrovat spouštění/vypínání instancí na základě metrik získaných z podpůrných systémů.

ID	TC 7
Akce	Spusť N nových instancí komponenty
Zdroj akce	Orchestrační systém
Prostředí	Zvýšený provoz
Artefakt	Businessová komponenta
Reakce	Spustí N nových komponent
Měřitelný výsledek	Bylo úspěšně spuštěno N nových komponent a přijímají požadavky

Tabulka 5.8: Testovací scénář 7 - Automatická orchestrace instancí

5.5 Udržovatelnost

5.5.1 Definice

Většina ceny softwarového systému nepochází z počátečního vývoje, ale z jeho provozování (opravování chyb, udržování v operačním režimu, prověřování výpadků atd.) a přidávání nových funkcionalit. Správně navržená architektura by měla provoz co nejvíce ulehčit a být připravena na změny. Tím je samotná architektura schopna minimalizovat nejen provozní cenu, ale i cenu za změnové požadavky. Udržovatelnost můžeme rozdělit do tří specifických kategorií[10]:

- Operativnost - Tým zodpovědný za provoz by měl vždy mít přehled o celkovém stavu systému (metriky, selhání). DevOps by měl být maximálně automatizován a zdokumentován. Změny konfigurací jednotlivých komponent by měly být rychlé a jednoduché.

- Jednoduchost - Návrh systému a jednotlivé komponenty by se měly řídit principem „high coupling and low cohesion“ a využívat v maximální míře abstrakci pro snížení komplexity a jednodušší správu.

- Rozšiřovatelnost - Systém by měl být jednoduše modifikovatelný a přizpůsobený častým změnám v jakékoli jeho části.

5.5.2 Požadavky vlastníků systému

Z rozhovoru s vlastníky systému je zřejmé, že toto je slabý článek současného řešení a nově navrhovaný systém musí v maximální míře podporovat všechny body zmíněné v předchozí části. Vždy musí být vidět současný stav systému a architektura musí být otevřena jakýmkoli změnám.

5.5.3 Testovací scénáře

5.5.3.1 Celkový stav systému

ID	TC 8
Akce	Zobrazení celkového stavu systému
Zdroj akce	Člen operačního týmu
Prostředí	Standardní provoz
Artefakt	Systém
Reakce	Systém zobrazí informace o všech částech systému
Měřitelný výsledek	Systém zobrazí následující informace o jednotlivých komponentách: • Stav (běží/neběží) • Technické informace (kde běží, IP adresa atd.) • Zátěž

Tabulka 5.9: Testovací scénář 8 - Celkový stav systému

5.5.3.2 Podrobnosti chybového stavu

ID	TC 9
Akce	Chybový stav
Zdroj akce	Neznámý
Prostředí	Standardní provoz
Artefakt	Jakákoli komponenta
Reakce	Uložení logů chyby
Měřitelný výsledek	Podrobnosti o chybě jsou dostupné v systému spravující a vizualizující logy celé aplikace

Tabulka 5.10: Testovací scénář 9 - Podrobnosti chybového stavu

5.5.3.3 Změna konfigurace komponenty

ID	TC 10
Akce	Změna konfigurace komponenty
Zdroj akce	Operační tým
Prostředí	Standardní provoz
Artefakt	Konfigurace
Reakce	Změna je distribuována mezi všechny vybrané komponenty
Měřitelný výsledek	Změna je úspěšně provedena do 5 minut bez manuální intervence

Tabulka 5.11: Testovací scénář 10 - Změna konfigurace

5.5.3.4 Dekompozice komponenty

ID	TC 11
Akce	Rozdělení komponenty do N menších komponent
Zdroj akce	Vývojář
Prostředí	Standardní provoz
Artefakt	Businessová komponenta
Reakce	• Nasadí se nové komponenty, původní běží dále • Po úpravě ostatních komponent pro používání nových verzí se stará verze vypne
Měřitelný výsledek	• Úspěšná registrace nového typu komponent • Nedošlo k výpadku • Nejsou hlášeny žádné chyby

Tabulka 5.12: Testovací scénář 11 - Dekompozice komponenty

5.6 Bezpečnost

5.6.1 Definice

Bezpečnost je měřítkem schopnosti systému chránit data a informace před neoprávněným přístupem a zároveň umožnit přístup oprávněným osobám a systémům. Útok, tj. akce vedená proti počítačovému systému s úmyslem způsobit škodu, může mít řadu podob. Může se jednat o neoprávněný pokus o přístup k datům nebo službám, nebo o změnu dat, nebo může být jeho cílem odepřít služby oprávněným uživatelům. [2]

Základní charakteristiky bezpečnosti jsou popsány písmeny CIA:

- Confidentiality (Důvěrnost) - ochrana dat a služeb před neoprávněným přístupem.
- Integrity (Integrita) - ochrana dat před změnou neoprávněnou osobou.
- Availability (Dostupnost) - systém je oprávněným uživatelům vždy dostupný.

5.6.2 Požadavky vlastníků systému

Navrhovaný systém má standardní požadavky na zabezpečení. Systém neobsahuje citlivé uživatelské údaje, přesto by bezpečnost měla být na velmi vysoké úrovni. Systém by měl být chráněn proti DoS[3] útokům a všechna externí a interní komunikace by měla probíhat přes zabezpečená spojení. Všechny neoprávněné přístupy by měly být zaznamenávány.

Systém by měl být chráněn před vytěžováním dat/nadměrném užívání skrz webovou aplikaci. Musí být tedy zavedeny uživatelská omezení na počet dotazů za nějaký časový interval.

5.6.3 Testovací scénáře

5.6.3.1 Neoprávněný přístup

ID	TC 12
Akce	Neoprávněný požadavek na systém
Zdroj akce	Neznámý
Prostředí	Standardní provoz
Artefakt	Systém
Reakce	Požadavek není vykonán a podrobnosti přístupu jsou zaznamenány.
Měřitelný výsledek	• Požadavek není vykonán • Podrobnosti nepovedené autentizace zaznamenány • V případě velmi častých pokusů ze stejného zdroje je notifikován operační tým

Tabulka 5.13: Testovací scénář 12 - Neoprávněný přístup

[3]DoS - Denial of service

5.6.3.2 Detekce vytěžování dat

ID	TC 13
Akce	1000 požadavků na API systému od stejného uživatele v rámci 1 hodiny
Zdroj akce	Přihlášený uživatel
Prostředí	Standardní provoz
Artefakt	Systém
Reakce	Událost je detekována a uživateli odepřen přístup do systému. Obě strany jsou notifikovány.
Měřitelný výsledek	• Notifikace uživatele • Notifikace správců systému • Zaznamenány podrobnosti o uživateli • Uživatel je dočasně zablokován

Tabulka 5.14: Testovací scénář 13 - Detekce vytěžování dat

Kapitola 6

Analýza možností distribuovaných architektur

6.1 Architektonické styly

6.1.1 Vrstevnatá architektura

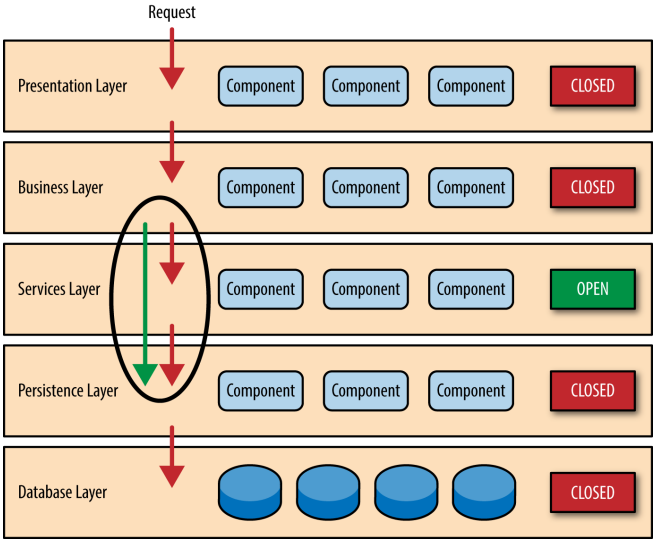
Tento architektonický styl je velmi známý a rozšířený. Prakticky se stal standardem ve většině enterprise aplikací.

Komponenty aplikace jsou uspořádány do separátních horizontálních vrstev. Každá z těchto vrstev má v systému specifickou roli a může komunikovat pouze s vrstvou o jednu pozici níže. Mezi klíčové koncepty tohoto architektonického stylu patří [25]:

- Izolace vrstev. Obecně jakékoli změny provedené v jedné z vrstev nemají dopad na komponenty v jiných vrstvách.
- Rozdělení odpovědnosti. Komponentám je povoleno komunikovat pouze mezi komponentami ze stejné vrstvy nebo z vrstvy přesně o jednu úroveň níže. Tím docílíme efektivního rozdělení rolí a odpovědností mezi jednotlivé vrstvy usnadňující jak vývoj, tak i následnou údržbu.

Princip komunikace lze v případě potřeby (např. výkon nebo sdílená logika) relaxovat a dovolit vrstvám komunikovat nejen striktně o jednu úroveň níže, ale o libovolnou vzdálenost. Stále ale platí předpoklad, že komunikace probíhá pouze směrem dolů. Vrstvy, které „propouštějí“ požadavky o úroveň níže značíme jako *otevřené* viz Obrázek 6.1. Samotná architektura nedefinuje konkrétní vrstvy, ale většinou jsou použity následující: prezentační, businessová, perzistentní a datová.

Tento styl je jednoduchý na implementaci i testování a je vhodný pro menší aplikace. Přestože použití není limitováno na monolitické systémy, tak v drtivé většině případů tomu tak je.[25] Jakýkoli monolitický systém není dobře škálovatelný- Je obtížné a drahé v něm dělat větší změny. Pokud jsou jednotlivé vrstvy nasazené separátně, je možné vrstvy horizontálně škálovat. Stále je ale granularita příliš široká pro efektivní fungování. Obecně architektura nevyniká ani ve výkonnosti kvůli neefektivnímu procházení všech vrstev.



Obrázek 6.1: Průběh požadavku ve vrstevnaté architektuře [25]

6.1.2 Microservices

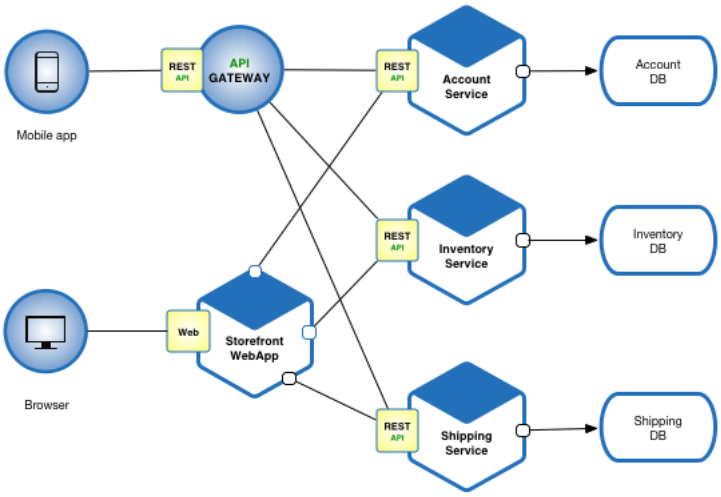
Microservices (Mikroslužby) se stávají čím dál tím více populárnějším architektonickým stylem, který je alternativou ke klasickým monolitickým aplikacím. Cílí především na agilní vývoj systému a flexibilní škálování. Mezi klíčové koncepty tohoto architektonického stylu patří [25]:

- Samostatně nasazovatelné komponenty. Každá komponenta v mikroslužbách je nasazena jako samostatná jednotka, což umožňuje snadnější a efektivnější nasazovací proces. Při vysoké míře automatizace se zvýší i stabilita celé aplikace.

- Komponenty služeb. Základním kamenem celé architektury jsou tzv. služby. Služba je komponenta starající se o jednu specifickou část. Ta může mít různou granularitu (objednávky vs objednávky specifické kategorie).

- Distribuovaná architektura. Microservices nejsou aplikovatelné pro monolitické systémy. Jde vždy o naprosto nezávislé komponenty komunikující mezi sebou pomocí jakéhokoli komunikačního média (např. REST, SOAP, RMI).

Nejdůležitější části při návrhu systému používající microservices je správné určení granularity služeb. Pokud zaměření zvolíme příliš malé, tak budou kladeny vysoké nároky na komunikaci a vznikne obrovský overhead. Pokud zaměření služeb zvolíme široce, tak se připravíme o výhody, které nám tento architektonická styl přináší a přiblížíme se více k monolitické aplikaci. Na obrázku 6.2 je uveden příklad velmi jednoduchého eshopu.



Obrázek 6.2: Microservices - Jednoduchý eshop
Zdroj: *https://microservices.io/patterns/microservices.html*

Tento styl řeší mnoho běžných problémů, které se vyskytují v monolitických aplikacích. Vzhledem k tomu, že hlavní komponenty aplikace jsou rozděleny do menších, samostatně nasazovaných jednotek, jsou aplikace vytvořené pomocí vzoru architektury mikroslužeb obecně robustnější. Poskytují lepší škálovatelnost, podporu pro automatické nasazování a jsou lépe připraveny na změnové požadavky.[25]

Protože jsou jednotlivé služby vyvíjeny separátně a v menších týmech, tak je vývoj obecně rychlejší a obsahuje menší pravděpodobnost zanesení chyb. Menší týmy zároveň zmenšují potřebu mezi-týmové komunikace, čímž snižují cenu za management. Nezávislost komponent pomáhá i při testování, kdy menší části aplikace jsou testeři schopni otestovat lépe a rychleji, než jeden velký monolit.

Mezi nevýhody patří menší výkonnost. Celkově se tento vzor kvůli distribuované povaze nehodí pro aplikace, pro které je kritický výkon. Zároveň distribuované systémy přináší nové problémy, které v monolitické aplikaci neexistují, např. datová konzistence napříč službami nebo jejich nedostupnost.

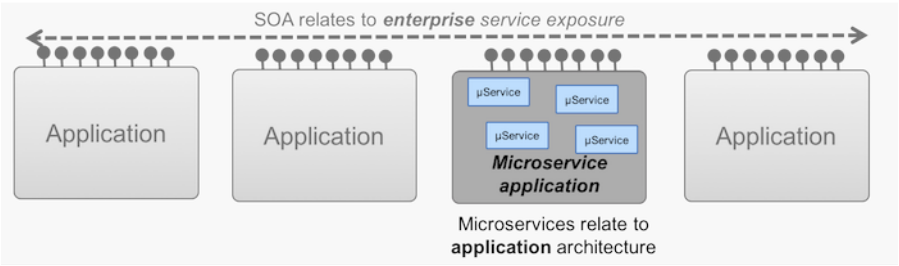
6.1.3 SOA

SOA (Service-oriented architecture) je distribuovaný architektonický styl zaměřený na nízké provázání jednotlivých služeb a jejich přepoužívávání. Mezi hlavní koncepty SOA patří:

- Samostatné služby - Jednotlivé služby jsou vyvíjeny a spravovány samostatně. Každá služba je zodpovědná za právě jeden businessový okruh a má přístup ke všem datům, které jsou potřeba pro její fungování.
- Společný komunikační kanál (ESB) - Přestože je možné implementovat SOA bez společného komunikačního kanálu, tak to není doporučované, protože by každá služba musela mít speciální integraci na všechny ostatní služby, které volá. ESB (Enterprise service bus) je architektonický vzor zprostředkovávající komunikaci mezi aplikacemi. Provádí

transformace datových modelů, zpracovává zprávy, provádí směrování a převádí komunikační protokoly. [6]

Velmi často je SOA přirovnáváno k mikroslužbám a přestože mají společné rysy, tak zásadní rozdíl je v jejich rozsahu. Microservices je architektura používaná v rámci jedné aplikace, zatímco SOA je architektura pro propojení různých systému v rámci jedné organizace.[29] Jak můžeme vidět na obrázku 6.3, tak aplikace, které jsou propojeny právě SOA architekturou, mohou být klidně tvořeny mikroslužbami.



Obrázek 6.3: SOA vs Microservices [29]

Hlavní výhodou SOA architektury je možnost přepoužívávání jednotlivých služeb. Jedna služba tím pádem může být součástí více aplikací. Zároveň SOA není obtížná ani na vývoj či údržbu. Jelikož za každou komponentou se skrývá samostatná aplikace, tak atributy jako je škálovatelnost a dostupnost si řeší jednotlivé aplikace sami.

Mezi nevýhody musíme zařadit počáteční investici a rozsáhlou komunikaci mezi službami. Protože jsou typicky se SOA spojeny větší počáteční investice, míří tento architektonický styl spíše na větší podniky. Všechna komunikace mezi službami probíhá prostřednictvím ESB, která navíc provádí nad zprávami různé transformace, což přidává relativně velký overhead ke každé zprávě. Tím se snižuje výkon systému.[6]

6.1.4 Event-driven architektura

Event-driven architektura je distribuovaný asynchronní architektonický styl. Soustředí se především na vysokou škálovatelnost. Jejími hlavními koncepty jsou[25]:

- Asynchronní komunikace. Odesílatel a příjemce požadavku na sebe nemusí vzájemně čekat a mohou pokračovat dalšími úkoly.

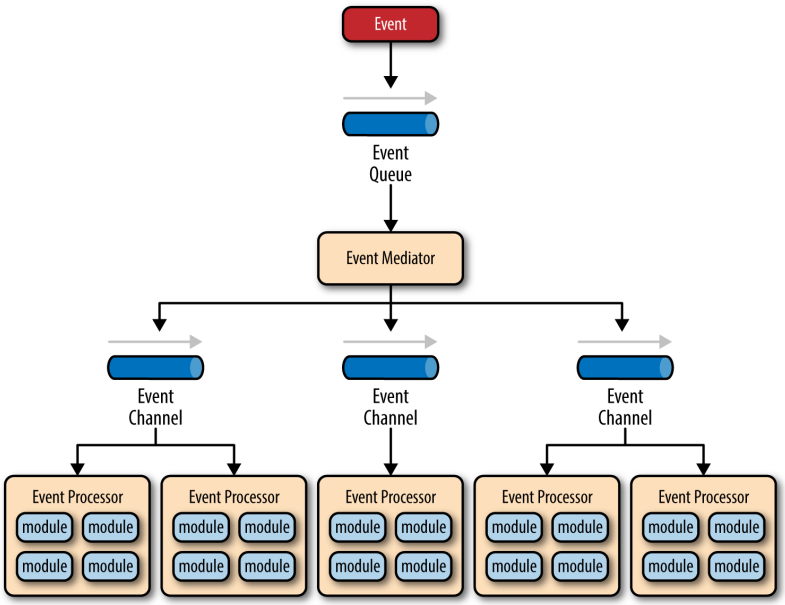
- Nezávislé a jednoúčelové procesory. Komponenty, které zpracovávají události, by měly být zaměřeny pouze na jeden typ událostí. Granularita typu může být rozdílná. Záleží na architektovi, aby posoudil, jak široké zaměření by procesor měl mít. To je podobné mikroslužbám.

Tento styl můžeme dále rozdělit do dvou hlavních topologií: *mediator* a *broker*.

6.1.4.1 Topologie Mediator

Tato topologie se hodí pro události, které mají více kroků a potřebují být nějakým způsobem organizovány, aby došlo k jejich úspěšnému zpracování. Můžeme si například představit dekompozici události do několika úkolů, které musí být vykonány ve specifickém pořadí a s omezenou paralelizací. Z obrázku 6.4 můžeme vidět tyto komponenty typické pro Mediator topologii [25]:

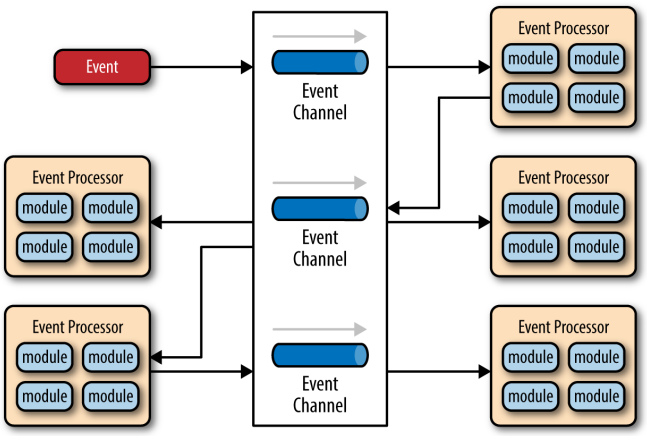
- Event queue - Komponenta přijímající tzv. počáteční události, které jsou následně distribuovány k dalšímu zpracování. Může to být např. message queue nebo jednoduchý REST endpoint. Systém jich může mít libovolné množství.
- Event Mediator - Komponenta zodpovědná za orchestraci kroků obsažených v počáteční události. Každý krok odpovídá tzv. pracovní události, která je poslána k samotnému zpracování. Stará se o správné pořadí a paralelizaci jednotlivých kroků.
- Event Channel - Komponenta pro asynchronní posílání jednoho typu požadavku ke zpracování.
- Event Processor - Komponenta vykonávající samotnou práci.



Obrázek 6.4: Event-driven architektura - Mediator Topology [25]

6.1.4.2 Topologie Broker

Tato topologie narozdíl od mediatoru neobsahuje žádnou centrální řídící komponentu jako je *Event Mediator*. Samotnou orchestraci si jednotlivé procesory řídí samy založením nové události. Topologie je ilustrována na obrázku 6.5. Hodí se především pro řetězení jednotlivých událostí.



Obrázek 6.5: Event-driven architektura - Broker Topology [25]

6.1.4.3 Hodnocení

Event-driven architektura exceluje především ve škálovatelnosti. Ta je dosažena nezávislými a oddělenými procesory a tím, že každou událost jsme schopni škálovat separátně. Na základě těchto vlastností tento styl dobře reaguje na změnové požadavky. Vzhledem k asynchronní povaze celé architektury, dosahuje i velmi dobré výkonnosti.

Na druhou stranu je právě kvůli asynchronnímu chování tento styl náročný na implementaci i testování. Zároveň je velmi obtížné zajistit silnou datovou konzistenci.

6.2 Architektonické návrhové vzory

6.2.1 Databáze

6.2.1.1 Database per service

Database per service je návrhový vzor, kde je jedna databáze sdílena pouze mezi komponentami stejného typu. Můžeme tedy říct, že pro N typů komponent existuje N databází. To znamená, že každá komponenta má přímý přístup pouze k datům, která jsou nutně potřebná pro její funkcionalitu. Tímto jsme schopni docílit menší závislosti mezi komponentami, lepší robustnosti (databáze není single point of failure) a škálovatelnosti. Zároveň není celá aplikace vázána na jeden typ databáze, ale architekt může pro každou komponentu zvolit takový typ databáze, který se pro daný use-case nejvíce hodí. Mezi nevýhody patří absence klasických transakcí zajišťujících datovou konzistenci. V případě potřeby je nutné implementovat komplexní transakční mechanismy mezi více komponentami.

6.2.1.2 Shared database

Tento návrhový vzor je přesným opakem předchozího vzoru. V celé aplikaci existuje pouze jedna databáze, ke které mají přístup všechny komponenty. Mezi největší výhodu

patří ACID[1] transakce zajišťující datovou konzistenci. Zároveň je tento způsob mnohem jednodušší a levnější na údržbu. Na druhou stranu tím vznikne vývojová závislost mezi komponentami. Úpravy databáze musí být koordinovány se všemi komponentami. Pokud se rozhodneme pro jednu velkou sdílenou databázi, musíme zároveň počítat s obtížnou škálovatelností a potencionálním úzkým hrdlem celé aplikace.

6.2.1.3 Replikace

V distribuovaných systémech, bez ohledu na to jaký databázový vzor zvolíme, z pravidla neběží čistě jedna instance databáze, ale je tvořena clusterem mnoha instancí. Jak již bylo několikrát zmiňováno, v distribuovaných systémech je selhání komponent nutné brát jako věc, která se děje běžně. V tomto ohledu je důležitý pojem „Replikace dat“. Replikace dat je proces vytváření kopií stejných dat napříč různými uzly clusteru. Hlavní motivací pro zavedené replikace je prevence ztráty dat. Současně může dobře navržená replikace zlepšit rychlost přístupu k datům (datová lokalita) nebo zlepšit jejich dostupnost. Způsob replikace můžeme rozdělit do dvou základních kategorií[10]:

- Master slave - Existuje jeden hlavní uzel, který celou replikace řídí a zbylé uzly pouze uchovávají data. Jakékoli zápisy dat musí jít přes master uzel, zatímco čtení je možné z jakéhokoli uzlu. Tím se tato architektura hodí pro aplikace, kde převažuje čtení dat.
- Peer to peer - Všechny uzly mají stejná práva. Mohou tedy jak zapisovat, tak číst data. Tím je tento způsob škálovatelnější, ale vyžaduje synchronizaci zápisů kvůli prevenci konfliktů.

6.2.1.4 Sharding

Sharding je databázový návrhový vzor, který rozděluje základní struktury (tabulky, key-value pairs, dokumenty...) databáze do několika menších částí nazývaných *shards*. Typicky se data rozdělují na základě předem definovaného klíče tak, aby data která jsou čtena typicky najednou byly uloženy ve stejné části. Hlavním důvodem pro implementaci shardingu je možnost horizontálního škálování a potencionální zrychlení přístupu k datům. Přestože tento způsob přináší do aplikace netriviální komplexitu, tak u větších distribuovaných systému je jeho zavedení velmi vhodné.

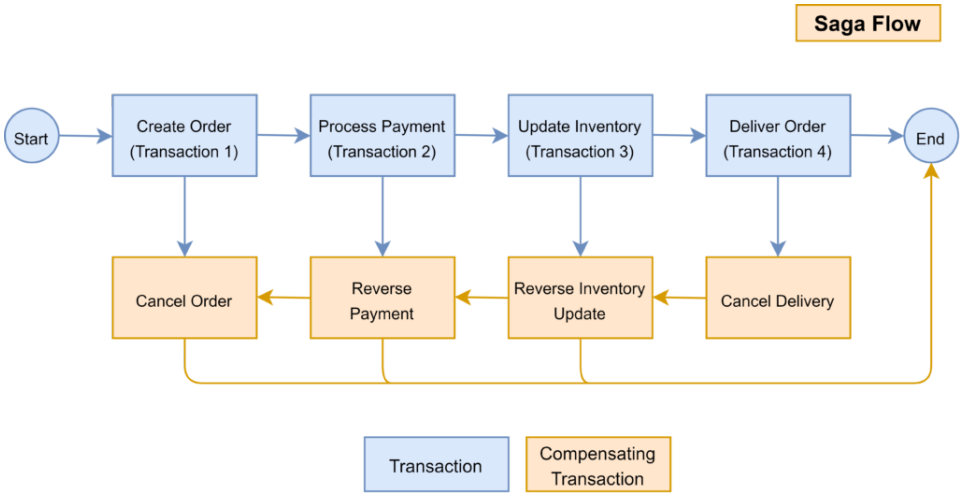
6.2.2 Datová konzistence

V architekturách s jedinou databází je dosažení datové konzistence jednoduché. Stačí použít lokální transakce. U distribuovaných architektur s mnoha databázemi ovšem dosažení tohoto stavu může být velmi komplexní problém. Typicky se jeden požadavek skládá ze vzájemné komunikace několika komponent a kdykoli se může tento požadavek přerušit. Nemusí se jednat o fyzické selhání některé z komponent, ale může se například porušit jedna z businessových podmínek. Proto je v některých systémech potřeba nahradit lokální transakce něčím jiným.

[1] ACID - atomicita, konzistence, izolovanost, trvalost

6.2.2.1 Saga

Saga je architektonický návrhový vzor, který implementuje businessové transakce pomocí sekvence lokálních transakcí tzv. saga. Každá lokální transakce provede lokální operaci a vytvoří událost, která spustí následující transakci v jiné komponentě. Pokud z jakéhokoli důvodu jedna z transakcí selže, jsou všechny předchozí transakce anulovány pomocí nové sekvence kroků skládajících se z lokálních transakcí vracejících stav do původní podoby. Aby tento způsob mohl fungovat, musí všechny operace být možné anulovat. Na obrázku 6.6 je ilustrován základní princip pomocí příkladu online objednávky.



Obrázek 6.6: Saga pattern

Zdroj: *https://www.baeldung.com/cs/saga-pattern-microservices*

Existují dva způsoby jak koordinovat sekvenci transakcí[33]:

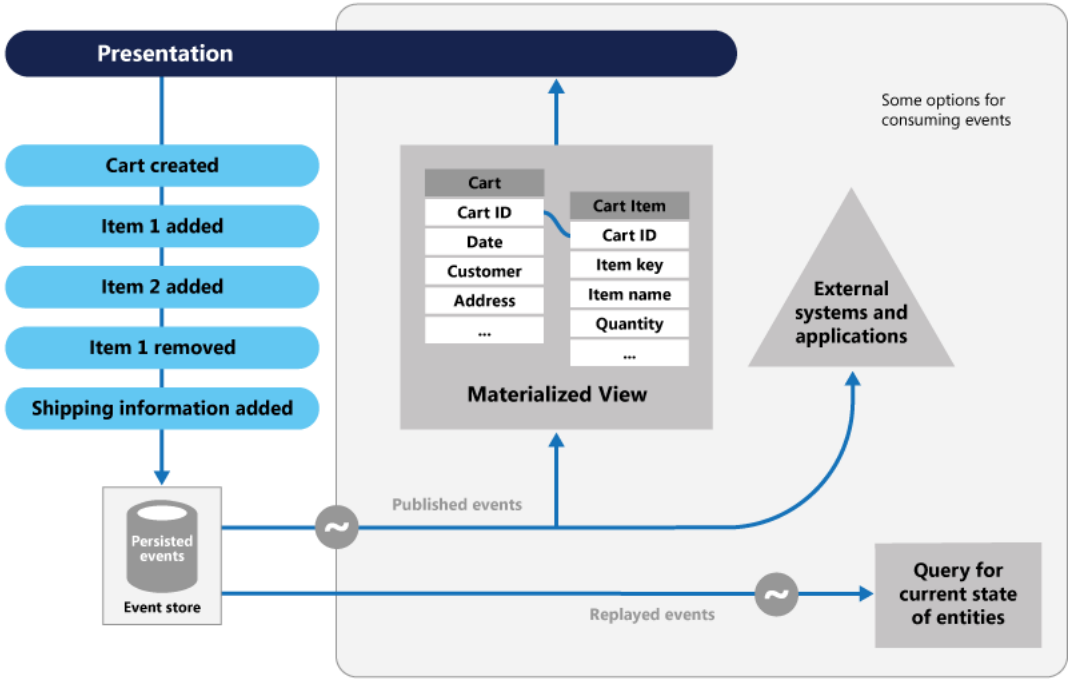
- Choreography - každá lokální transakce sama vytváří a publikuje událost na základě které je spuštěna další transakce, která dodržuje stejný princip. Tento přístup je flexibilní, ale při netriviálních úkolech se může stát velmi nepřehlednou.
- Orchestration - celá saga je řízena centrální komponentou, která postupně spouští jednotlivé lokální transakce. Přestože orchestrační komponenta značně zjednodušuje práci s komplexními úkoly, tak zavadí do systému single point of failure.

6.2.3 Event sourcing

Event sourcing řeší lehce jiný problém než Saga. V lokální transakci komponenty velmi často potřebujeme poslat nějakou zprávu nebo událost do jiné komponenty. Nemáme ovšem jistotu toho, že po odeslání události transakce skončí úspěšně. Právě jedním z řešení tohoto problému je použití návrhového vzoru Event Sourcing.

Typicky je uložený v databázi pouze nejnovější stav entit. Použitím Event sourcing patternu přestaneme ukládat do databáze nejnovější stav a budeme místo toho ve správném

pořadí ukládat všechny události, které entitu jakkoli upravují. Abychom dostali aktuální stav entity, musíme provést všechny události od posledního snapshotu entity [18]. Na obrázku 6.7, kde je ilustrován Event sourcing je vidět, že databáze událostí dovoluje komponentám odebírat události, které jsou do ní vloženy a tím jim umožňuje upravovat jejich reprezentaci entit, která je použita pro jejich čtení.



Obrázek 6.7: Event sourcing pattern [18]

Event sourcing pattern nám kromě řešení problému s atomicitou odesílání zpráv kompletně vyřeší i audit. Zároveň umožňuje dobrou škálovatelnost a výkonnost. Na druhou stranu je nutné se smířit s vyšší komplexností a eventuální konzistencí(trvá než se změny projeví ve čtecí reprezentaci nebo-li v materialized view).

6.2.4 Komunikace

V distribuovaných systémech je komunikace mezi komponentami velmi důležitou částí architektury. Zde jsou zmíněné 3 velmi populární styly. Je jich samozřejmě mnohem více, ale v rámci práce není důležité je uvádět.

6.2.4.1 REST API

REST je architektonický styl fungující na klasickém HTTP principu request/response. Jedná se o synchronní komunikaci. REST API je jedním z nejrozšířenějších způsobů komunikace v posledních letech. Je jednoduchý na používání a má relativně malý overhead. Nevýhodou může být právě synchronní komunikace z hlediska výkonu a nutnost použití „Service discovery“ pro zjištění IP adres komponent, se kterými je potřeba komunikovat.

6.2.4.2 Events

U tohoto stylu komponenty komunikují pomocí tzv. Domain events. Domain event je událost nad jakoukoli business entitou v systému. Může se jednat například o událost „objednávka vytvořena“. Tyto události jsou vytvořeny a odeslány do sdílené fronty, ze kterých si mohou ostatní komponenty tuto událost vyzvednout a následně ji zpracovat.

6.2.4.3 Messaging

Jedná se o asynchronní způsob komunikace mezi komponentami, které si mezi sebou vyměňují zprávy. Messaging můžeme dále rozdělit do několika dalších kategorií jako je „Publish/Subscribe“ nebo „Request/Asynchronous response“. Tento způsob podporuje výkonnost, dostupnost i škálovatelnost. Přidává ovšem do systému větší míru komplexnosti a je nutné využít „message broker“ komponentu.

6.2.5 Service discovery

V distribuovaných aplikacích komunikují aplikace přes síť. Problémem je, že každá komponenta musí mít přehled o IP adresách ostatních komponent, aby bylo možné navázat spojení. Tyto IP adresy není možné staticky konfigurovat jednak z hlediska použitelnosti v různých prostředích, a jednak se IP adresy dynamicky mění (například při selhání hardware). Právě tento problém řeší architektonický vzor „Service disovery“. Existuje několik způsobů implementace.

6.2.5.1 Client-side service discovery

V této verzi implementace figurují dvě komponenty: klient(komponta, která chce komunikovat s jinou) a server (databáze obsahující lokaci všech komponent označována jako „service registry“). V případě, že chce kdokoli komunikovat s jinou komponentou, tak se nejprve dotáže service discovery komponenty na jakých adresách požadovaná komponenta běží. Pak je už na klientovi jaký load-balancing mechanismus použije. Jednotlivé adresy v service discovery serveru jsou dynamicky přidávány, upravovány a odebírány na základě periodických heartbeatů z klientů. Výhodou tohoto přístupu je relativně jednoduchá implementace a žádná statická konfigurace. Nevýhodou může být závislost všech klientů na jednom typu service registry.[16]

6.2.5.2 Server-side service discovery

Tato implementace je velmi podobná té předchozí. Obsahuje navíc komponentu routeru/proxy, na kterou komunikující komponenty posílají všechny své dotazy a ta následně provede stejný mechanismus jako u client-side discovery. Velkou výhodou je, že se klient nemusí vůbec o nic starat a neobsahuje žádnou funkcionalitu navíc. Nevýhodou je, že se musíme starat o další komponentu aplikace, která musí být vysoce dostupná.[17]

6.2.6 Infrastruktura

V distribuovaných systémech musíme uvažovat také fyzické umístění jednotlivých komponent na dostupný hardware. Následuje výčet možných strategií pro toto přiřazení.

6.2.6.1 Service per host

Jak již název napovídá, na každý fyzický/virtuální server nasazujeme maximálně jednu komponentu. Výhodami může být maximální možná izolovanost, jednoduchá správa a monitoring. Tento způsob je ale velmi neefektivní a drahý v případě většího množství komponent.

6.2.6.2 Multiple Services per host

Tímto architektonickým vzorem sice zefektivníme použití našich zdrojů a zjednodušíme automatizaci, ale služby už na sobě nejsou kompletně nezávislé. Sdílejí spolu prostředí i knihovny. Chyby vzniklé z konfliktu jednotlivých komponent v jednom prostředí jsou hůře reprodukovatelné.

6.2.6.3 Service per container

Populárním trendem dnešní doby je kontejnerizace[4]. Toho využívá vzor „Service per container“. Kontejner je software, který zabalí dohromady aplikaci a všechny její závislosti. Umožňuje ji spouštět izolovaně a nezávisle na prostředí.[5] Nasazováním komponent pomocí kontejnerů jsme schopni dosáhnout nejen velmi dobré efektivity ve využití zdrojů, ale také jednoduché správy a pokročilé automatizace.

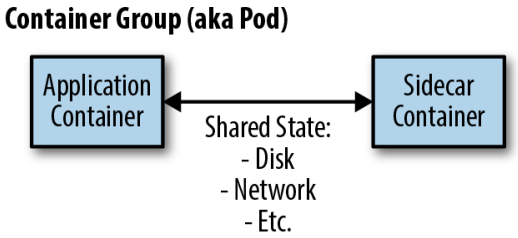
6.2.6.4 Serverless

Serverless je architektonický vzor, někdy označován také jako architektura, ve kterém je celý systém nasazen v cloudu. Kompletní infrastruktura systému je delegována na poskytovatele služeb a tým se stará pouze o samotný kód. Cena závisí na počtu příchozích požadavků a požadované výkonnosti. Serverless je založený na reakcích na specifické události a jejich zpracování pomocí tzv. funkcí. To znamená, aby systém mohl být takto nasazen, musí být bezestavový a fungovat pouze na principu request/response. Velkou výhodou tohoto způsobu nasazení je naprostá izolace od nižších úrovní infrastruktury a automatické škálování dle potřeby.[26]

6.2.6.5 Sidecar

Architektonický vzor Sidecar je způsob nasazení rozšiřující „Service per container“. Každá komponenta je nasazena stále v samostatném kontejneru, ale k tomu je na stejném fyzickém hostu nasazen další kontejner obsahující čistě podpůrnou funkcionalitu (Sidecar kontejner).[3] Mezi podpůrnými funkcionalitami si můžeme představit např. service discovery, logování nebo monitoring. Kontejnery obsahující komponenty aplikace se sidecar kontejnerem komunikují a nemusí se o podpůrné funkcionality starat. Dohromady tvoří tzv. Pod nebo Container

Group. Velkou výhodou tohoto vzoru je jednoduchost použití i v případě použití heterogenních jazyků a technologií pro aplikační komponenty.



Obrázek 6.8: Sidecar pattern [3]

6.2.7 Observability

6.2.7.1 Agregace logů

V distribuovaných systémech je velmi obtížné nalézt příčinu jakýchkoli chyb. Jednak z důvodu velkého množství komponent mající separátní logy a za druhé z obrovského množství dat. Řešením je agregace logů na jednom centrálním místě. Logy se tam posílají ze všech komponent systému. Zpravidla jsou ještě před tím nějakým způsobem upraveny a zformátovány. Správce systému pak může logy prohledávat a provádět nad nimi různé analýzy.

6.2.7.2 Agregace aplikačních metrik

Obdobně jako u agregace logů, tak bychom měli využít stejný způsob i u aplikačních metrik. Aplikačních metrik může být obrovské množství, např. počet příchozích požadavků nebo průměrná latence odpovědí. Uchováváním těchto informací na centrálním místě získáme ucelený pohled na stav celého systému a zároveň jsme schopni detekovat případné problémy či zvýšení zátěže a adekvátně na to reagovat.

6.2.7.3 Distributed tracing

V případě implementované agregace logů v nich nejsme velmi často schopni efektivně vyhledávat. Jeden externí požadavek typicky jde přes několik komponent, což analýzu celého požadavku značně komplikuje. Metoda „Distributed tracing“ toto řeší pomocí unikátních identifikátorů každého externího požadavku. Toto ID se pak předává mezi všemi komponentami, které jsou součástí požadavku a zapisují ho do logů. Následně je možné v centrálním úložišti logů velmi jednoduše dohledat celý průběh požadavku.

6.2.8 Nasazení

Pro dosažení důležitých architektonických požadavků, jako je vysoká dostupnost a dobrá udržitelnost, je důležité zvolit správný postup při nasazování nové verze komponenty. Zde se zaměříme na kompletní nahrazení N komponent služby A.

6.2.8.1 Blue/green deployment

V případě tohoto způsobu je vytvořeno N nových instancí služby A - zelené instance. Následně je veškerý provoz přesměrován na tyto nové instance, ale původní (modré) instance stále běží. Během nastaveného intervalu od přepnutí na novou verzi je nutné zjistit, jestli vše funguje správně a následně se modré komponenty vypnou. V případě jakýchkoli problémů se provoz přesměruje zpět na původní verze. [2]

6.2.8.2 Rolling upgrade

Na rozdíl od typu blue/green nejsou při zvolení tohoto způsobu nasazení nahrazeny instance komponent najednou, ale postupně. Celý proces probíhá následovně [2]:

1. Vytvoř novou instanci komponenty A.
2. Začni novou komponentu používat.
3. Vyber jednu z původních instancí komponenty, vyčkej na dokončení všech jejích úkolů a vypni ji.
4. Opakuj, dokud není kompletně nahrazeno N instancí.

Výhodou tohoto přístupu je, že nevyžaduje dvojnásobné prostředky čistě pro nasazení.

6.2.9 Bezpečnost - Autentizace

6.2.9.1 Access token

Access token je standardní způsob autentizace oproti externímu API. Typicky to bývá JWT, které jednoznačně identifikuje uživatele a běžně obsahuje i doplňující informace(jméno, email, práva atd.). Tento token se předává mezi všemi komponentami, které se účastní vyřizování požadavku. Takto každá komponenta ví, o jakého uživatele jde.

6.2.9.2 Klientské certifikáty

V některých případech může být pro systém důležité ověřit zdrojový systém požadavku. Pro takové ověření můžeme využít tzv. klientské certifikáty. Každé komponentě se vygeneruje TLS certifikát a při navázaní spojení se klasickým způsobem ověří jeho platnost.[20]Pro úspěšné ověření musí mít každá komponenta v „trust store“ uložený nadřazený certifikát(případně samotný klientský certifikát) nebo příslušný veřejný klíč.

6.3 Technologie

6.3.1 Framework

V distribuovaných systémech zpravidla není nutné psát celý systém v jednom jazyce či frameworku. Přestože existuje obrovské množství technologií, ve kterých tyto systémy

můžeme psát, tak zde se omezíme na technologie v jazyce Java. Tento výběr je založen na preferenci a znalostí současného vývojového týmu. Tato kapitola se soustředí především na technologie vhodné pro psaní business komponent.

6.3.1.1 Spring

Spring je nejpopulárnější a nejvíce rozšířený Java framework současnosti[30]. Konkrétně by mělo jít o Spring Boot 2[2] jako framework pro vývoj aplikací a Spring Cloud[3], který obsahuje velké množství „out of the box" architektonických vzorů zmiňovaných v kapitole 6.2. Spring má velmi dobrou dokumentaci a běžné problémy jsou podrobně diskutovány v internetových diskuzích.

6.3.1.2 Vert.x

Vert.x[4] je framework od firmy Eclipse pro psaní reaktivních aplikací bežících na JVM[5]. Vert.x podporuje event-driven a neblokující model, což znamená, že aplikace je výkonnější a lepé škálovatelná.[13] Oproti frameworku Spring Boot je Vert.x opravdu „lightweight" a neobsahuje v sobě aplikační server. Podobně jako Spring Cloud, tak obsahuje i různé architektonické vzory „out of the box"

6.3.1.3 Quarkus

Tento framework je poměrně nový a dobře se hodí při integrací s Kubernetes. Quarkus[6] je cloude native, kontejnerově orientovaný framework od společnosti Red Hat pro psaní aplikací v jazyce Java. Quarkus je přizpůsobený pro GraalVM a HotSpot. Jeho cílem je vytvořit Java platformu v prostředí Kubernetes a serverless a zároveň nabídnout vývojářům jednotný reaktivní a imperativní programovací model, který optimálně řeší širší škálu distribuovaných aplikačních architektur [21]. Jednou z klíčových vlastností frameworku je rychlý start systému, kde oproti Spring Boot aplikacím může být rychlejší až o 70%. Na druhou stranu není tak dobře zdokumentován.

6.3.1.4 MicroNaut

Micronaut[7] je softwarový framework založený na JVM pro vytváření lehkých modulárních aplikací a mikroslužeb. Micronaut je známý svou schopností vytvářet aplikace s malou paměťovou stopou a krátkou dobou spouštění. Důležitou výhodou frameworku je, že doba spouštění a spotřeba paměti nejsou vázány na velikost kódové základny aplikace. Velký rozdíl mezi frameworkem Micronaut a ostatními frameworky spočívá v tom, že Micronaut analyzuje metadata při kompilaci aplikace. Během této fáze kompilace Micronaut vygeneruje další sadu tříd, které reprezentují již přednastavený stav aplikace. To umožňuje mnohem efektivněji

[2] `https://spring.io/projects/spring-boot`
[3] `https://spring.io/projects/spring-cloud`
[4] `https://vertx.io`
[5] JVM - Java Virtual Machine
[6] `https://quarkus.io`
[7] `https://micronaut.io`

aplikovat dependency injection a aspektově orientované programování (AOP) při konečném spuštění aplikace [23]. Zároveň poskytuje nativní podporu pro některé architektonické vzory jako jsou například service discovery nebo distribuovaná konfigurace.

6.3.2 Kontejnerizace

Kontejnery jsou velmi rozšířeným způsobem pro vývoj a nasazení aplikací. Umožňují kompletní izolaci od vnějšího prostředí při zachování velmi dobré výkonosti a malém overheadu. Zde je uveden výčet dvou nejpopulárnějších zástupců této technologie.

6.3.2.1 Docker

Docker je v dnešní době v zásadě standardem kontejnerizace. Umožňuje spustit kontejnery na jakémkoli operačním systému a jeho nástroje zvládnou všechny úkoly spojené s orchestrací kontejnerů, od vyrovnávání zátěže až po síťování. Celý Docker běží pomocí démon procesu: procesu na pozadí, který pomáhá spravovat/vytvářet kontejnery, sítě a datové volumes [7]. Zde mohou nastat drobné problémy. Jelikož Docker daemon může běžet pouze jeden, tak se z něho stává „single point of failure“. Všechny běžící kontejnery jsou podprocesy daemon procesu, takže jeho selhání má fatální následky.

6.3.2.2 Podman

Podman je kontejnerový engine bez démona a bez roota vyvinutý společností RedHat jako alternativa ke službě Docker. Modulární konstrukce umožňuje Podmanu používat jednotlivé systémové komponenty pouze v případě potřeby. Jeho přístup ke správě kontejnerů bez nutnosti root práv umožňuje, aby kontejnery spouštěli i neprivilegovaní uživatelé. Právě tento přístup odstraňuje „single point of failure“, který je u služby Docker. [7] Výhodou služby Podman je velmi jednoduchá integrace s Kubernetes. Umí vytvářet tzv. pods, jejichž definice lze exportovat do YAML souboru kompatibilního s Kubernetes clusterem.

6.3.3 Orchestrace

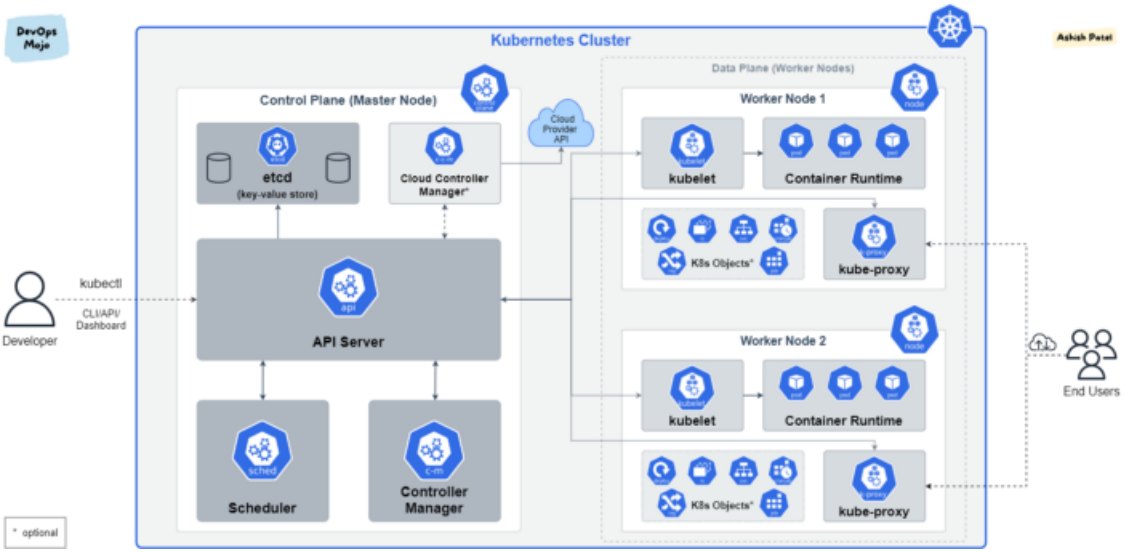
6.3.3.1 Kubernetes

Kubernetes[8] je přenosná, rozšiřitelná open-source platforma pro správu kontejnerových úloh a služeb, která umožňuje deklarativní konfiguraci i automatizaci. Má rozsáhlý, rychle rostoucí ekosystém. V produkčním prostředí je třeba spravovat kontejnery, v nichž jsou aplikace spuštěny a zajistit, aby nedocházelo k výpadkům. Pokud například dojde k výpadku kontejneru, musí se spustit jiný. Tuto situaci a mnoho dalších Kubernetes řeší. Seznam funkcionalit, které Kubernetes s sebou přináší je následující [11]:

- service dicovery a load balancing,
- orchestrace úložišť,

[8] `https://kubernetes.io/`

- automatické nasazování,
- automatické rozdělování úloh mezi dostupné zdroje,
- automatické opravy nefunkčních kontejnerů,
- management konfigurace a citlivých informací.



Obrázek 6.9: Komponenty Kubernetes [19]

6.3.3.2 Docker Swarm

Docker Swarm[9] je orchestrační nástroj pro správu skupiny Docker kontejnerů běžících napříč fyzickými nebo virtuálními servery, také označováno jako cluster. Jakmile je cluster nakonfigurován, mohou se používat pro jeho správu klasické Docker příkazy. Činnost clusteru řídí tzv. Swarm manager a servery, které se do clusteru zapojily, se označují jako uzly. Docker Swarm za nás podobně jako Kubernetes řeší i další funkcionality jako je Service discovery, load balancing nebo škálování. Docker Swarm se kvůli své jednoduchosti hodí spíše pro menší až středně velké projekty. Na rozdíl od Kubernetes nedisponuje tak rozsáhlým množstvím funkcionalit a automatizace, která se ve větších systémech může hodit.[22]

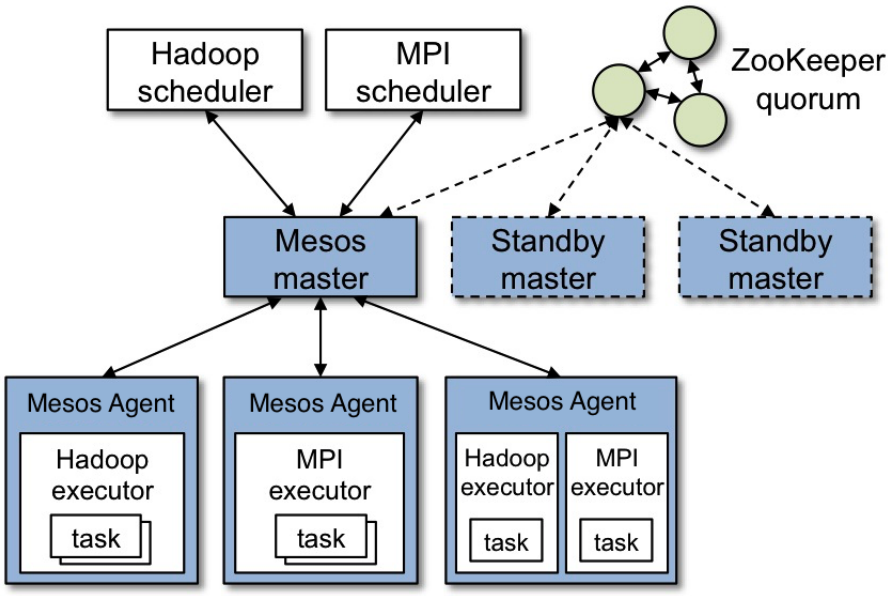
6.3.3.3 Apache Mesos a Marathon

Apache Mesos[10] je open-source správce clusteru, který řídí pracovní zátěž v distribuovaném prostředí prostřednictvím dynamického sdílení a izolace prostředků. Mesos je vhodný pro nasazení a správu aplikací v rozsáhlých clusterových prostředích. Mesos sdružuje prostředky strojů/uzlů v clusteru do jednoho fondu, z něhož lze využívat různé pracovní zátěže.

[9] `https://docs.docker.com/engine/swarm/`
[10] `https://mesos.apache.org/`

To je také známé jako abstrakce uzlů a tento způsob odstraňuje potřebu přidělovat konkrétní stroje pro různé pracovní zátěže.[1]

Marathon je platforma pro orchestraci kontejnerů postavená nad Apache Mesos. Podobně jako ostatní orchestrační platformy přidává mnoho funkcionalit jako je Service discovery, load balancing, automatické škálování nebo sbírá metriky. [15]



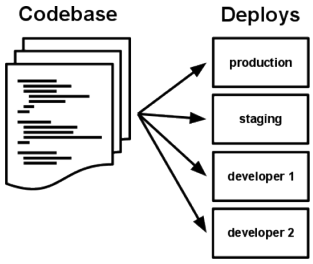
Obrázek 6.10: Architetura Apache Mesos
Zdroj: *https://mesos.apache.org/documentation/latest/architecture/*

6.4 The Twelve-Factor App

The Twelve-Factor App (12F App) je metodologie a sada principů pro vytváření kvalitních služeb SaaS (software-as-a-service)[32]. Zavádí dvanáct principů a doporučeních, které míří na vytváření škálovatelných, přenositelných a cloudových aplikací. Zároveň podporuje v maximální míře automatizaci procesů vývoje a nasazení. Tuto metodologii lze použít na aplikacích napsaných v libovolném programovacím jazyce, které využívají libovolnou kombinaci podpůrných služeb. Následuje seznam pravidel a jejich vysvětlení na základě zdroje [32].

I. Codebase
Aplikace musí být verzována ve verzovacím systému (Git, SVN atd.). Každá aplikace musí mít právě jeden repozitář, který je použit k nasazení ve více prostředích. Ilustrace viz Obrázek 6.11.

II. Dependencies
Aplikace nesmí být závislá na implicitní existenci systémových balíčků. Musí explicitně de-



Obrázek 6.11: Twelve factor app - One codebase, many deploys [32]

klarovat a izolovat všechny své závislosti. Příkladem může být použití Maven v jazyce Java nebo NPM v jazyce Javascript.

III. Config
Konfigurace aplikace musí být uložena v prostředí aplikace(environment). Konfigurace aplikace se typicky velmi liší na typu prostředí a tím pádem nesmí být žádná konfigurace uložena natvrdo v kódu. Všechnu potřebnou konfiguraci si aplikace načítá z proměnných prostředí (environment variables).

IV. Backing Services
Backing service je jakákoli služba, kterou aplikace používá při svém fungování. Může se jednat například o databázi nebo message broker. Aplikace musí brát tyto služby jako připojené zdroje dostupné přes URL. Zdroje by mělo být možné kdykoli změnit bez vlivu na funkčnost aplikace.

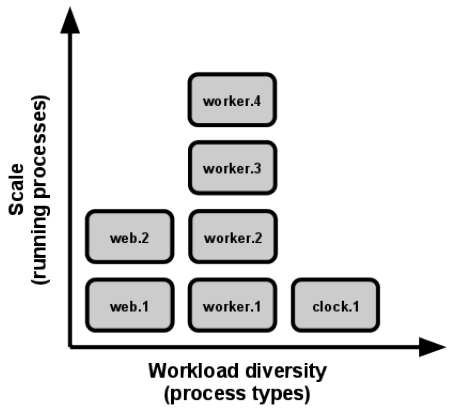
V. Build, release, run
Principy 12F vyžadují, aby nasazení aplikace bylo rozděleno do tří fází:

- Build - transformace kódu do spustitelného artefaktu.
- Release - zkombinuje vytvořený artefakt s aktuální konfigurací prostředí.
- Run - samotné spuštění aplikace.

VI. Processes
Aplikace je spouštěna jako jeden nebo více procesů. Dle principů 12F by procesy měly být bezestavové a neměly by mezi sebou nic sdílet. Jakékoli data, které je potřeba ukládat by se měla dávat do databáze. Žádná data by se neměla ukládat v paměti aplikace.

VII. Port binding
Aplikace musí být kompletně samostatná. Často jsou webové aplikace závislé na webovém serveru nainstalované v daném prostředí. To porušuje princip toho pravidla. Aplikace musí být samostatná a pouze vystavovat své služby na nějakém portu.

VIII. Concurrency
Toto pravidlo říká, že by se měla celá aplikace rozdělit do více procesů/instancí stejného typu místo toho, aby běžel jeden masivní systém. Můžeme si to představit jako preferenci horizontálního škálování oproti vertikálnímu.



Obrázek 6.12: Twelve factor app - Škálování [32]

IX. Disposability
Procesy aplikace musí být robustní. Jakékoli zapínání/vypínání instance by nemělo mít žádný dopad na stav celého systému. Důležité je, aby procesy byly vždy ukončeny tzv. „graceful shutdown“. Ten zajišťuje, aby systém byl vždy v korektním stavu.

X. Dev/prod parity
Různá prostředí aplikace by si měla být co nejvíce podobná. Tím se významně snižuje pravděpodobnost nalezení chyb pouze ve specifickém prostředí. Velmi důležité je používat stejné podpůrné komponenty (databáze, message broker atd.).

XI. Logs
Logování je naprosto klíčové v monitorování a provozování aplikace. Principy 12F říkají, že samotná aplikace se nestará o ukládání a zpracovávání logů, jen je generuje na standardní výstup. Logy jsou následně vyzvednuté a zpracované jiným procesem.

XII. Admin processes
V aplikacích typicky kromě standardních procesů bývají i administrativní procesy, které je nutné provést pouze jednou (např. migrace databáze). Aplikace 12F by měla mít tyto procesy součástí repozitáře. Měly by být plně automatizovány, aby nedocházelo k chybám nebo zapomenutí.

Kapitola 7

Návrh finální architektury

V této kapitole bude představen kompletní návrh nové architektury, včetně způsobu vývoje a nasazení. Součástí návrhu není způsob migrace dat z původního systému.

7.1 Použitá metodika

Při návrhu architektury byla použita metodika „Domain Driven Design“ (dále jen DDD). Jedná se o způsob návrhu architektury softwaru, při které se klade důraz na rozdělení jednotlivých komponent podle businessových kontextů. DDD zavádí následující tři pojmy[31]:

- Doména
- Subdoména
- Ohraničený kontext (Bounded context)

Doména představuje celou oblast podnikání, ve které se firma pohybuje. Každá doména se skládá z několika subdomén, kde každá představuje oddělenou logickou část domény. Ohraničený kontext je spojení jedné nebo více subdomén do jednoho logického celku, který následně definují jednotlivé komponenty systému.

Současně s metodikou DDD byly aplikovány i prvky metodiky „Attribute Driven Design“ (dále jen ADD). Základním stavebním kamenem ADD jsou testovací scénáře[2] definované v kapitole 5. Tyto scénáře jsou při návrhu použity jako nástroj pomáhající k usměrnění návrhu ke správnému výsledku a jako následná verifikace.

7.2 Architektura

Zvolená architektura je spojením dvou architektonických stylů zmiňovaných v kapitole 6.1: Microservices a Event-driven. Spojení těchto stylů je také někdy označováno jako Event-driven microservices. Asynchronní komunikací mezi jednotlivými službami docílíme nezávislosti jednotlivých mikroslužeb pro zlepšení udržitelnosti systému a zjednodušení nasazovacích procesů. Zároveň jsme schopni celý systém lépe škálovat a zajistit vysokou dostupnost systému, která byla požadována vlastníky systému.

Kompletní architektura je znázorněna v diagramu 7.1.Systém poskytuje externím uživatelům (web nebo integrující se systém) REST API. Jedinou vstupní bránou do systému je API Gateway, která požadavky přesměrovává na příslušné komponenty. Systém robotů není v rozsahu práce, proto je v diagramu znázorněn mimo systém, ale reálně by byl jeho součástí. V případě externích požadavků jsou použity pro komunikaci synchronní HTTPS dotazy. Samotná komunikace mezi komponentami probíhá asynchronně přes messaging. Specifickou vlastností některých služeb je, že jejich databáze je přístupná i komponentě Apache Spark z důvodů jednoduchého přístupu k datům a analýzami nad nimi. U těchto služeb byly zvoleny databáze sloupcového typu kvůli jejich optimalizaci přístupu k velkému množství dat v rámci analýz. Předpokladem architektury je, že systém robotů je schopen úkoly jednoduše deduplikovat a nedochází tak ke spuštění stejných úkolů najednou.

7.2.1 Hlavní komponenty

Rozsah jednotlivých mikroslužeb byl určen na základě výstupů z metodiky DDD. Dle entit systému budou existovat mikroslužby čtyř domén: uživatelů, webových domén, zpětných odkazů a klíčových slov. Speciálním případem je analytická mikroslužba, která má čistě funkcionální ohraničení, nikoli doménové.

7.2.1.1 User Service

User Service je mikroslužba spravující uživatele aplikace. Neslouží k autentizaci ani k autorizaci. Jedná se čistě o službu, která uchovává data uživatelů a jejich preference. Tyto údaje jsou dostupné přes vystavené REST API.

Vzhledem ke struktuře dat a jejich relativně malému množství (řádově deseti tisíce uživatelů), je pro ukládání dat zvolena relační databáze PostgreSQL. Databáze poběží jako cluster, do kterého lze libovolně přidávat další uzly dle potřeby. Jako prevence ztráty dat je zvolena WAL (Write-Ahead Logging) streamová replikace s replikačním faktorem 3. Pro zajištění vysoké dostupnosti databáze je použit nástroj Postgres Operator [1] od firmy Zalando. Ten je schopný v Kubernetes spustit celý cluster s out-of-the-box nastavenou replikací nebo load balancerem.

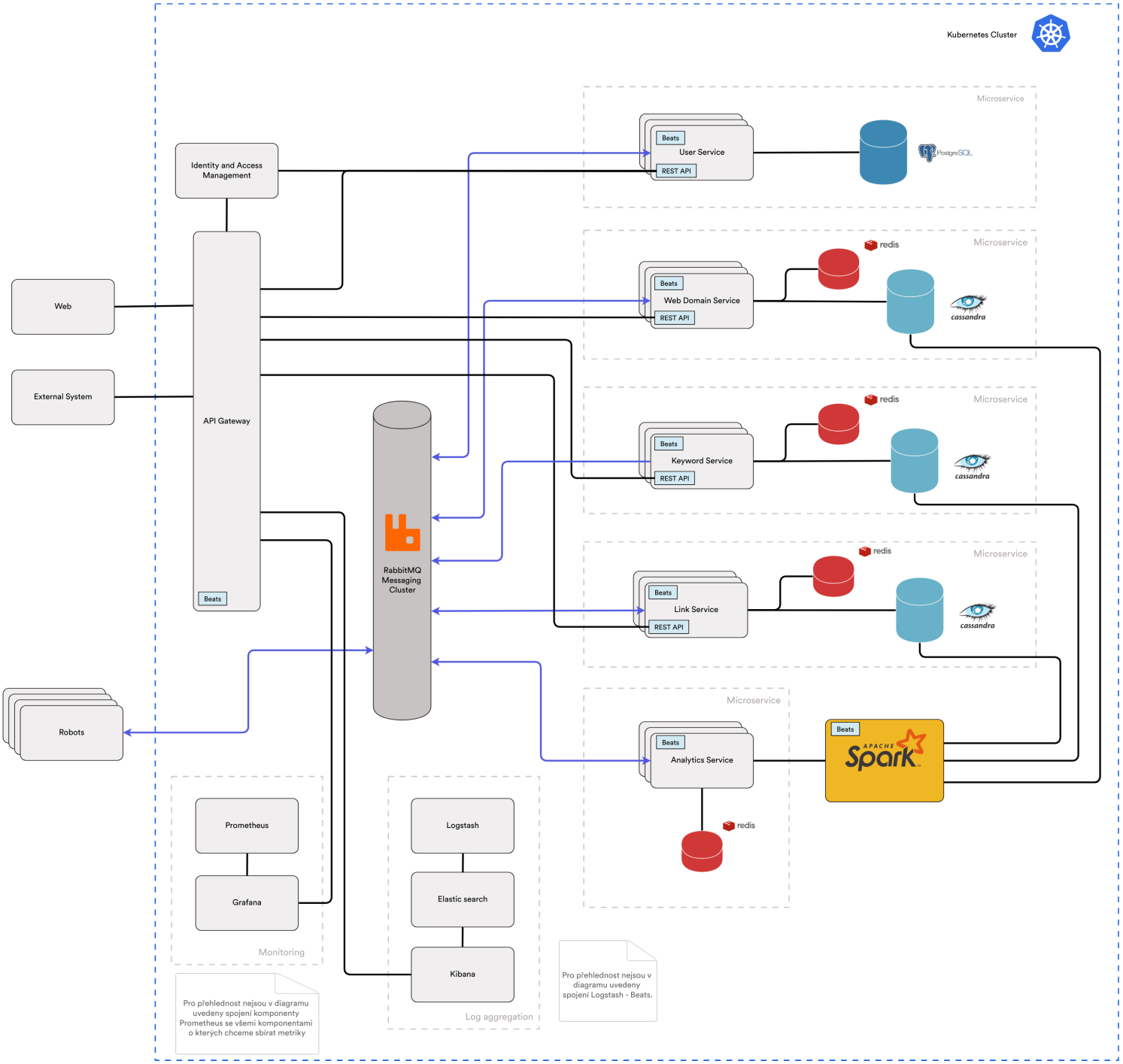
Pro vytvoření a úpravy schématu databáze bude použit nástroj Liquibase[2], který tyto akce plně automatizuje. Definice databáze bude uložena v repozitáři služby.

7.2.1.2 Web Domain Service

Web Domain Service je mikroslužba spravující informace o internetových doménách. Může se jednat o základní data jako je současný stav webu nebo informace o vlastnících domény. Zároveň jsou zde uloženy informace o různých kategoriích, do kterých mohou domény patřit. Současně je na základě svých dat schopna generovat úkoly pro roboty. Služba potřebuje ke správné funkcionalitě dvě databáze: Apache Cassandra a Redis.

Apache Cassandra je NoSQL databáze sloupcového typu. Ukládá všechny potřebná data o doménách a zároveň jsou zde dostupné výsledky analytických prací nad souvisejícími daty.

[1] `https://github.com/zalando/postgres-operator`
[2] `https://liquibase.org`



Obrázek 7.1: Diagram komponent navrhované architektury

Databáze je nasazena v clusteru s replikační strategií „NetworkTopologyStrategy“ a replikačním faktorem 3. Primární funkcionalitou je uchovávání webových domén. Tabulka proto bude velmi rozsáhlá a je potřeba ji rozdělit na menší části viz kapitola 6.2.1.4. Je potřeba zvolit takový klíč, který zajistí, že jednotlivé části nebudou ani moc velké a ani moc malé. Z toho důvodu je zde zvolen jako „partitioning key“ první písmeno jména domény.

Redis je NoSQL databáze typu key-value. Funguje jako distribuovaná cache předpočítaných souhrnů o jednotlivých doménách (doména →souhrn). Tyto souhrny jsou dostupné z vystaveného REST API. Databáze je nasazena v clusteru s replikačním faktorem 3. Redis se sám stará o sharding pomocí hashů jednotlivých klíčů.

Pro vytvoření a úpravy schématu databáze Apache Cassandra je stejně jako u předchozí služby použit nástroj Liquibase pro automatizaci.

7.2.1.3 Keyword Service

Komponenta Keyword Service je mikroslužba, která má na starosti výsledky vyhledávačů po vyhledání klíčového slova. Klíčová slova jsou předdefinována v databázi a jsou předávány robotům pro zjištěni výsledků v podporovaných vyhledávačích. Tato služba podobně jako ostatní služby vystavuje REST API. Konkrétně poskytuje souhrn všech dostupných informací ohledně klíčového slova a umožňuje administrátorům systému přidávat nová klíčová slova. Služba potřebuje ke správné funkcionalitě dvě databáze: Apache Cassandra a Redis.

Apache Cassandra je provozována v clusteru se stejným nastavením, které je zmiňováno v sekci 7.2.1.2. V databázi výsledků vyhledávání dle klíčových slov nás typicky zajímá nějaký časový rozsah, proto jako „partition key“ zvolíme datum vyhledávání. Pro správu schématu je využit nástroj Liquibase.

Redis funguje jako distribuovaná cache pro zmiňované souhrny klíčových slov (slovo →souhrn).

7.2.1.4 Link Service

Link service je mikroslužba zpracovávající odkazy mezi doménami. Jedná se o nejjednodušší strukturu dat ze všech služeb. V zásadě pouze zpracovává, ukládá záznamy zpětných odkazů a výsledky analýz nad nimi. Služba externím komponentám vystavuje REST API. Služba potřebuje ke správné funkcionalitě dvě databáze: Apache Cassandra a Redis.

Apache Cassandra je provozována v clusteru se stejným nastavením, které je zmiňováno v sekci 7.2.1.2. V původním systému obsahuje databáze se zpětnými odkazy nejvíce záznamů z celého systému a s obrovskou datovou zátěží se počítá i nadále. Je tedy nutné tuto tabulku rozdělit do partitions tak, aby bylo efektivní dotazovat se nad tak velkým počtem dat. Při různých analýzách nás typicky zajímá čas získání informace a o jaké domény jde. Proto bude partitioning key tvořen právě datem získání odkazu a mezi clustering keys(určuje řazení v rámci partion) bude patřit zdrojová i cílová doména. Pro správu schématu je využit nástroj Liquibase.

Redis funguje jako distribuovaná cache pro data dostupná v REST API. Může se například jednat o strukturu: doména X →list domén, které na doménu X odkazují

7.2.1.5 Analytics Service

Analytics Service je mikroslužba poskytující možnost spouštět rozsáhlé analytické dotazy nad daty systému. Tato služba má jedinou funkcionalitu: odebírá z RabbitMQ fronty události, na základě kterých spouští ve Sparku nové analytické práce. V případě volných prostředků si služba z prioritizované fronty v RabbitMQ vytáhne analytický úkol nahraný jinou komponentou a na základě informací v tomto úkolu rozhodne, který „job“ spustit. Pomocí Redis databáze zároveň zajišťuje, aby úkoly stejného typu měly mezi sebou nakonfigurovaný časový rozestup.

7.2.1.6 Apache Spark

Apache Spark je open-source distribuovaný systém pro zpracování a analýzu velkých objemů dat. Poskytuje vysokoúrovňové API rozhraní v jazycích Java, Scala, Python, R a optimalizovaný engine, který podporuje obecné prováděcí grafy. Podporuje také bohatou sadu nástrojů jako je Spark SQL, strukturované zpracování dat, MLlib pro strojové učení, GraphX pro zpracování grafů a Structured Streaming pro inkrementální výpočty a streamové zpracování[27]. V našem systému by měl sloužit k jakýmkoli analýzám nad větším množstvím dat. Využijeme tedy pouze dávkový(batch) přístup. Je přímo napojen na databáze služeb: Link Service, Keyword Service a Web Domain Service. Spuštěné „joby“ mohou z těchto databází libovolně čerpat data, analyzovat je a následně jsou výsledky uloženy zpět do těchto databází, kde k tomu mají přístup příslušné služby. V systému jsou práce výhradně spouštěny z Analytics Service.

Důležité je použít **Apache Spark 3**. V této verzi je významně zvýšena nejen výkonnost zpracování dat, ale také počet optimalizací nad napsanými dotazy a mnoho dalších zlepšení. K zajištění dobré škálovatelnosti a jednoduché konfigurace v rámci Kubernetes bude použit speciálně navržený operátor „spark-on-k8s“ od firmy Google[3].

7.2.1.7 RabbitMQ messaging

Pro veškerou asynchronní interní komunikaci mezi komponentami je použit message broker RabbitMQ. Jedná se o open-source systém implementující AMQP protokol[24]. Splňuje požadavky na vysokou dostupnost a lehkou horizontální škálovatelnost. V systému použijeme 2 typy front: klasické FIFO (first in, first out) fronty pro události emitované jednotlivými službami a prioritizované fronty pro zadávání analytických a robotických úkolů.

7.2.1.8 API Gateway

API gateway je vstupní bránou pro všechny externí požadavky. Funguje jako reverzní proxy, která rozděluje požadavky na příslušné komponenty. Zároveň se stará o autentizaci a autorizaci všech požadavků. Dle požadavků na systém je potřeba zavést nějaké limity na používání systému. Takovou logiku a sledování limitů na každého uživatele by obstarávala také API Gateway. V případě potřeby je možné i odpovědi na některé dotazy cachovat. Pro zajištění vysoké dostupnosti systému by mělo běžet více instancí této komponenty.

[3] `https://github.com/GoogleCloudPlatform/spark-on-k8s-operator`

7.2.1.9 Identity and access management

Identity and access management, zkráceně IAM, je komponenta systému starající se o autentizaci a autorizaci uživatelů. Může se jednat o on-premise řešení (např. Keycloak) nebo o externí systém (např. Auth0 nebo Okta). IAM se stará o kompletní flow autentizace a odhlášení, umožňuje správu uživatelů, jejich rolí a typicky podporuje SSO(Single Sign-On). V našem systému bude IAM komunikovat s komponentou User Service, která slouží jako zdroj uživatelů. Můžeme tedy IAM také chápat jako prostředníka, který obaluje uživatele informacemi navíc a přidává další funkcionality. Pro zajištění vysoké dostupnosti celého systému je klíčové, aby IAM byl vysoce dostupná komponenta.

7.2.2 Podpůrné komponenty

7.2.2.1 Logování

Logování v celém systému je implementací architektonického vzoru Agregace logů popsaného v kapitole 6.2.7.1. Konkrétně budeme používat skupinu systému známých jako ELK stack[4]. Ten se skládá ze 4 komponent: Beats, Logstash, Elasticsearch a Kibana.

Beats je rodina „lightweight“ produktů schopná sbírat nejrůznější možná data o běžících komponentách. V systému budeme používat pouze jeden z nich: Filebeat. Jedná se o specializovaný „sběrač“ souborů a logů. Filebeat bude v rámci Kubernetes nasazován jako DaemonSet, což znamená, že na každém uzlu systému bude běžet právě jeden požadovaný Pod. Filebeat sbírá v rámci jednoho uzlu všechny logy z lokálně nasazených Podů a následně je v pravidelné intervalu odesílá komponentě Logstash k dalšímu zpracování.

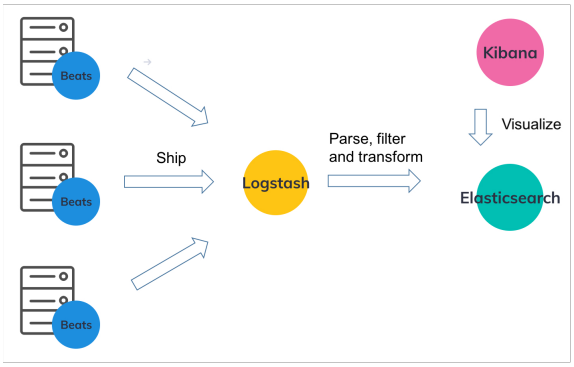
Logstash funguje jako agregátor všech logů celého systému, který data filtruje, transformuje nebo přidává dodatečné informace na základě nakonfigurovaných pipeline. Data jsou transformavána do takového formátu, aby je bylo možné poslat do komponenty ElasticSearch. ElasticSearch všechny příchozí data indexuje a ukládá do databáze. Právě komplexní indexace dovoluje provádět rozsáhlou analýzu nad logy systému. Pro provádění takových analýz či jiných dotazů nad databází logů je použita Kibana. Jedná se o software, který umožňuje uživateli vše vizualizovat a dotazovat se nad daty z uživatelsky přívětivého webového rozhraní. Kibana zároveň umožňuje nastavit tzv. watchers, které opakovaně posílají nakonfigurované dotazy do Elasticsearch a na základě výstupů mohou notifikovat zodpovědné osoby. V případě zakoupení licencí je možné nakonfigurovat i detekce anomálií pomocí strojového učení. Celé flow zpracovávání logů je znázorněno na obrázku 7.2.

Zároveň je do systému zapracován vzor „Distributed tracing“ popsaný v kapitole 6.2.7.3. Vztahuje se na 3 entity: uživatelské požadavky na REST API, generované úkoly pro roboty a analytické úkoly nad daty. Uživatelským požadavkům toto ID přiřazuje API gateway, zatímco u generovaných úkolů jim jsou přiřazeny v rámci vytvoření.

7.2.2.2 Monitoring

Monitorování celého systému je implementací architektonického vzoru Agregace aplikačních metrik popsaného v kapitole 6.2.7.2. Pro implementaci budeme používat dvě kompo-

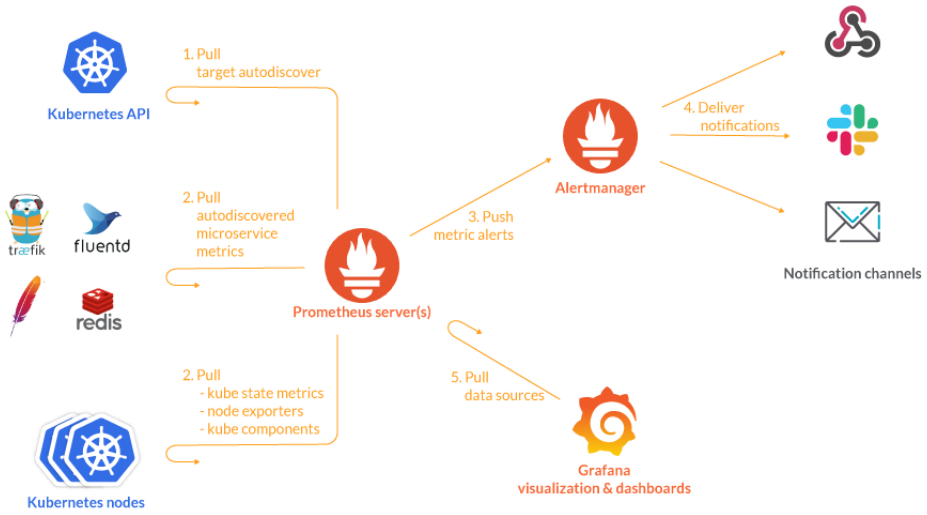
[4] `https://www.elastic.co/what-is/elk-stack`



Obrázek 7.2: Zpracování logů v systému
Zdroj: *https://logz.io/blog/filebeat-vs-logstash/*

nenty: Prometheus a Grafana. Průběh sběru aplikačním metrik je znázorněn na obrázku 7.3.

Prometheus sbírá aplikační metriky přímo z běžících služeb podporující Prometheus API nebo specializovaných exportéru pomocí modelu HTTP pull. Pro identifikaci těchto služeb v rámci Kubernetes lze použít tzv. labels. Všechny tyto metriky Prometheus ukládá po konfigurovanou dobu (typicky 15 dnů) a sestavuje z nich časové řady. Dovoluje nastavit různá pravidla, která v případě porušení jsou schopná administrátorům poslat varování (mail, slack...). Pro vizualizaci těchto dat je využita Grafana. Ta bere Prometheus jako zdroj dat, nad kterým je schopná se efektivně dotazovat a uživateli zobrazuje stav celého systému v rámci intuitivního uživatelského rozhraní.



Obrázek 7.3: Sběr aplikačních metrik v systému
Zdroj: *https://sysdig.com/blog/kubernetes-monitoring-prometheus/*

7.3 Vývoj

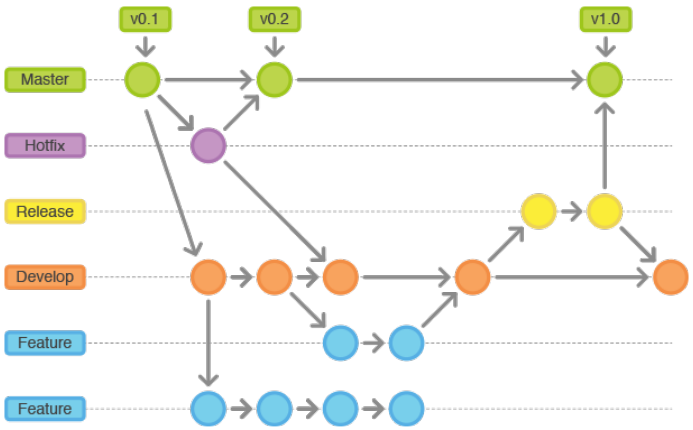
Tato sekce je věnována popisu procesu vývoje jednotlivých služeb. Všechen kód a potřebná konfigurace je verzována v systému Git.

7.3.1 Mikroslužby

Každá mikroslužba bude mít svůj vlastní repozitář. V rámci vývoje bude použit model GitFlow znázorněn na obrázku 7.4. Pro vývoj jedné služby v distribuovaném systému je typicky potřeba mít lokálně spuštěné i jiné služby. V našem případě minimálně databázi a message brokera, ale v některých případech může být potřeba vetší část celého systému. Zde má vývojář tři možnosti:

- Lokální Docker Compose - vývojář pomocí docker-compose a předem definovaných konfiguračních souborů spustí všechny potřebné služby. Jedná se o nejjednodušší způsob hodící se pro vývoj s minimálním množstvím závislých komponent.
- Lokální Kubernetes cluster - vývojář si spustí „ligtweight“ verzi Kubernetes zvanou minikube[5] na svém lokálním počítači. Následně si spustí všechny potřebné služby pomocí nástroje Helm, o kterém bude řeč v dalších kapitolách, a speciálně nakonfigurovaných „chartů“ pro lokální prostředí.
- Kubernetes cluster na vývojovém prostředí - vývojář použije nástroj, který mu umožní se z lokálního vývojového prostředí připojit na služby běžící v clusteru vývojového prostředí. Může se jednat např. o nástroj Telepresence[6].

Jakou možnost zvolit závisí čistě na dané situaci. Samozřejmě existuje pro lokální vývoj mnoho dalších přístupů pomocí nástrojů jako Tilt, Kompose nebo Skaffold. V principu by ale mělo jít o podobné mechanismy.



Obrázek 7.4: Model GitFlow
Zdroj: *https://leanpub.com/git-flow/read*

[5] `https://minikube.sigs.k8s.io/docs/`
[6] `https://www.telepresence.io`

7.3.2 Podpůrné služby

Podpůrnými službami se zde myslí všechny komponenty systému, které systém vyžaduje, ale zároveň se jedná o software třetích stran, který je potřeba pouze nakonfigurovat. Konkrétně se jedná o komponenty monitoringu, logování, messagingu, IAM, API Gateway a Apache Spark. Konfiguraci celé této infrastruktury budeme mít v jednom Git repozitáři tak, aby konfigurace byla verzována. Z tohoto repozitáře následně budou vést všechny DevOps „pipeliny“ pro nasazování jednotlivých komponent do systému.

Každé prostředí bude mít svůj separátní Git repozitář.

7.3.3 Apache Spark

Apache Spark je specifickou komponentou, která je součástí obou zmíněných částí. Je to software třetí strany, ale zároveň je potřeba pro něj vyvíjet v rámci Analytics Service tzv. joby. Samotná konfigurace a proces nasazení bude uložen v repozitáři pro podpůrné služby, ale joby se budou vyvíjet v rámci Analytics Service. Apache Spark podporuje řadu populárních jazyků a v zásadě nezáleží na tom, který se použije. Osobně bych doporučil psát Spark úkoly v jazyce Python (PySpark).

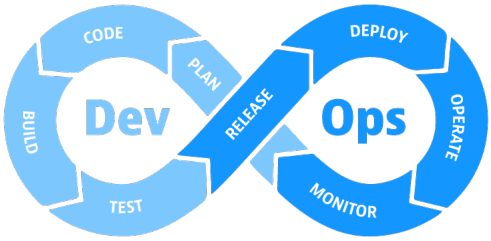
Apache Spark podporuje několik tzv. deploy módů. Ten říká, kde jsou prostředky používané v rámci pak Spark aplikace fyzicky lokalizovány. V produkčním prostředí se typicky používá *cluster mode*, v případě lokálně spuštěné instance Spark aplikace speciálně pro vývoj existuje *local mode*. V rámci něj běží všechny procesy na lokálním zařízení a je jednoduché „joby“ vyvíjet, debugovat a testovat.

7.4 DevOps

DevOps je pojem označující praktiky a procesy organizace pro vytvoření, nasazení a používání softwaru [8]. Pro maximálního využití Devops by měla být většina těchto procesů plně automatická. Typicky se DevOps rozděluje do osmi částí znázorněných charakteristickou „osmičkou“ na obrázku 7.5.

Tato sekce je věnována popisu DevOps v navrhované architektuře. Právě dobře zvládnuté DevOps procesy mohou dramaticky zmenšit čas od vývoje k nasazení a získat tím konkurenční výhodu na trhu. Fáze operací a monitoringu byly probrány již v dřívějších kapitolách. Fáze plánování není předmětem architektury, ale spíše projektového řízení. Zbylé fáze rozdělíme do tří částí: *Continuous integration*, *Continuous delivery* a *Continuous deployment*, které jsou specifikovány v následujících sekcích.

Počítá se s využitím třech prostředí: Vývojové, Testovací a Produkční



Obrázek 7.5: Devops - Grafické znázornění [8]

7.4.1 Continuous integration

Continuous integration je pojem obsahující tři fáze: Code, Build a Test. Jedná se o automatizaci procesu analýzy kódu, vytvoření spustitelného artefaktu a jeho testování. V našem systému se tato fáze týká pouze mikroslužeb. Každá mikroslužba bude obsahovat „pipelinu“ s následujícími částmi:

- Build - Vytvoření spustitelného artefaktu a zjištění, zda je vůbec možné artefakt vytvořit.
- Unit Testing - Spuštění automatických jednotkových testů (Všechny musí projít).
- Integration Testing - Spuštění automatických integračních testů. Ostatní služby mohou být buď mockované nebo být v prostředí „pipeliny“ spušteny pomocí dockeru.
- Static code analysis - Statická analýza nového kódu zda vyhovuje standardům nastavených organizací. Populární volbou je použití systému Sonarqube.

Tento proces bude spuštěn při každém commitu do jakékoli git větve.

7.4.2 Continuous delivery

Continuous delivery je nadstavbou nad continuous integration. Obsahuje navíc fázi Release. Release fáze vytvoří novou verzi softwaru a zajistí, aby byl vytvořený artefakt dostupný k použití. V našem systému se tato fáze týká pouze mikroslužeb. Ve fázi Release se z artefaktu vytvořeného v rámci předchozích kroků vytvoří Docker image s názvem příslušné služby a označením (tag) vycházející verze. Tento image je následně nahrán do interního repozitáře určeného speciálně pro Docker images (Container registry). V případě vytvoření produkční verze (master) se také automaticky v repozitáři vytvoří git tag označující tuto verzi. Pro číslování verzí je použit model Sémantického verzování[7].

Tento proces bude puštěn při každém commitu nebo merge requestu do následujících větví:

- Develop - Verze k nasazení na vývojová prostředí. Formát X.Y.Z-SNAPSHOT.
- Release - Verze k nasazení na testovacím prostředí. Formát X.Y.Z-RC.x.
- Master - Produkční verze X.Y.Z.

[7] `https://semver.org`

7.4.3 Continuous deployment

Continuous deployment je nadstavbou nad continuous delivery. Přidává navíc fázi nasazení (Deploy). Jak již z názvu vyplývá, tato fáze nasadí vytvořenou verzi do běžícího prostředí. Tato část není limitována pouze na naše mikroslužby, ale vztahuje se i na podpůrné systémy.

Celý proces nasazení je postaven kolem softwaru Helm 3[8]. Jedná se o package manager pro Kubernetes[14]. Umožňuje zabalit k sobě více YAML souborů s definicemi různých komponent v Kubernetes a jednoduše je distribuovat pomocí specializovaných Helm repozitářů. Takto sdruženým YAML souborům, které spolu tvoří jeden systém se říká „Helm Charts“. Helm charts můžeme vytvářet pro vlastní software nebo používat již vytvořené a veřejně dostupné charts pro software třetích stran (např. ELK stack nebo Prometheus)[14]. Helm je zároveň i tzv. "templating engine". Umožňuje v rámci vytvořených YAML souborů používat placeholdery, které se při použití nahradí konkrétními hodnotami definovanými v externím souboru (typicky values.yml). Helm 3 naplno využívá Kubernetes API. Defaultním způsobem nasazování je Rolling upgrade popsaný v kapitole 6.2.8.2.

Každá mikroslužba bude mít ve svém repozitáři definována svůj vlastní Helm Chart, který se bude používat na různých prostředích. Jeho konfigurace bude definována pomocí hodnot v nasazovací procesu. Proces nasazení bude spuštěn automaticky po commitu nebo merge requestu do příslušných git větví.

Jak již bylo zmíněno v sekci 7.3.2, tak konfigurace podpůrné infrastruktury bude definována v jednom separátním repozitáři pro každé prostředí zvlášť. Tento repozitář bude rozdělen do podadresářů dle použitých externích komponent. Každý podadresář bude obsahovat soubor *values.yml* s konfigurací příslušné komponenty v rámci daného prostředí. Každý repozitář bude obsahovat pro každou komponentu tlačítko "Deploy", které komponentu nasadí do Kubernetes clusteru.

7.4.4 Orchestrace

Celý systém je postaven okolo nástroje Kubernetes popsaného v kapitole 6.3.3.1. Ten kompletně spravuje všechny kontejnery v rámci jednoho clusteru. Kromě zmiňovaných funkcionalit, které Kubernetes řeší, se zaměříme na detekci selhání a automatické škálování nazývané „Horizontal Pod Autoscaling“ (HPA). Počítá se s použitím kontejnerů Docker.

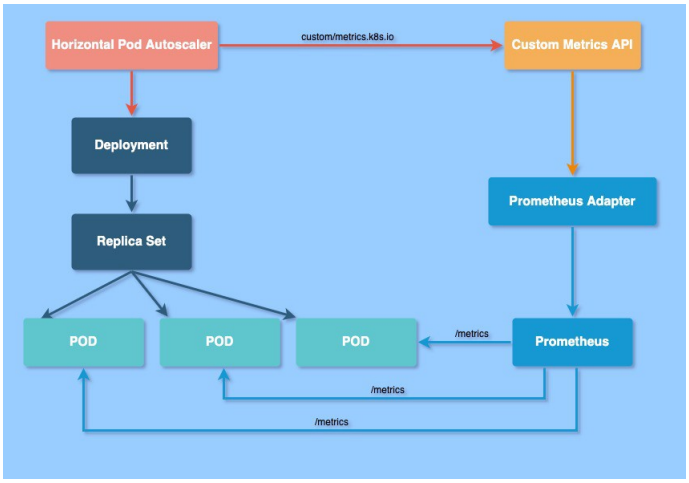
Kubernetes již v základní konfiguraci detekuje selhání „podů“ a následně nasadí novou instanci komponenty a starou smaže. Zároveň umožňuje konfigurovat tzv. liveliness probes. Jedná se o vlastní implementaci kontroly, zda Pod běží/funguje správně. Pokud je Pod označen jako nefunkční je automaticky restartován.

Automatické škálování v případě zvýšení zátěže spustí nové „Pody“ příslušné služby. V případě snížení zátěže a počtu „Podů“ nad minimálním nakonfigurovaným počtem jsou nadbytečné komponenty vypnuty. Dle dokumentace Kubernetes[12] se počet spuštěných replik řídí následujícím jednoduchým vzorcem:

$$desiredReplicas = ceil[currentReplicas * (currentMetricValue/desiredMetricValue)] \tag{7.1}$$

[8] `https://helm.sh`

Pro sběr metrik nutných pro automatické škálování využijeme již zmiňovaného monitoringu pomocí systému Prometheus viz obrázek 7.6. Pro každý Deployment/StatefulSet v aplikaci, který chceme automaticky škálovat vytvoříme HPA, kde definujeme požadované metriky. Abychom mohli využít vlastní metriky ze systému Prometheus, tak je potřeba využit Kubernetes poskytované Custom Metrics API, které bude využívat data z komponenty Prometheus Adapter spuštěné speciálně pro tento účel.



Obrázek 7.6: Architektura automatického škálování
Zdroj: `https://towardsdatascience.com/kubernetes-hpa-with-custom-metrics-from-prometheus-9ffc201991e`

7.4.5 Infrastruktura

Navrhovaná architektura nevyžaduje specifickou infrastrukturu. Všechny komponenty musejí být kontejnerizovány a jejich fyzická lokace je determinována Kubernetes. Z pohledu funkčnosti ale není důležité zda vše běží on-premise, v cloudu nebo kombinovaně.

Je důležité zmínit, že z pohledu ceny je preferovanější varianta on-premise. Kvůli charakteristice systému zpracovávat obrovské množství dat (big data) a následně je analyzovat v Apache Spark, bude docházet k relativně velkým přenosům dat mezi službami. Cena služeb u poskytovatelů cloudu se z velké části odvíjí právě od množství přenášených dat. Při větších přenosech může tedy cena rapidně růst. Ve variantě on-premise je na rozdíl od cloudu potřeba velká počáteční investice, ale v dlouhodobém měřítku se to vyplatí více.

7.5 Kontrola testovacích scénářů

V této sekci je vyhodnoceno, zda architektura pokrývá všechny testovací scénáře definované v kapitole 5 Nefunkční požadavky systému.

ID Scénáře	Splněno	Podrobnosti
TC 1	Ano	V případě selhání softwarové komponenty v Kubernetes clusteru (Pod), je selhání automaticky detekováno a je nasazena nová instance dané komponenty. Případně je komponenta pouze restartována. Záleží na konfiguraci atributu *restartPolicy*. V případě selhání na straně softwarové komponenty, jsou chybové logy směrovány na standardní výstup, kde jsou odchyceny komponentou FileBeat a poslány k dalšímu zpracování. V případě selhání na straně Kubernetes je log dostupný v logovacím streamu příslušného uzlu, který je opět monitorován komponentou FileBeat.
TC 2	Ano	Load balancing za nás řeší Kubernetes. Všechny instance jedné komponenty jsou abstrahovány do tzv. Service, která automaticky distribuuje požadavky mezi jednotlivé služby. Defaultním mechanismem je Round Robin.
TC 3	Ano	Nasazení nové verze komponenty v Kubernetes probíhá pomocí stylu rolling upgrade. Automaticky je zajištěno, aby se vytvořil požadovaný počet nových instancí a staré se smazaly. Při jakémkoli problému se vše automaticky vrátí do původního stavu. V rámci definované CI/CD pipeline musí úspěšně proběhnout všechny testy.
TC 4	Ano	Oprava kritické chyby v produkčním prostředí (tzv. hotfix) postupuje dle principů GitFlow. Následně je nasazení naprosto stejné jako u TC 3.
TC 5	Ano	Fronta úloh v rámci message brokeru je realizována pomocí prioritizované fronty. Tím jsme schopni odbavovat události s vysokou prioritou velmi rychle. Dle návrhu je použit Spark operator, který v zásadě poskytuje neomezené škálování. Jediným omezením jsou zdroje clusteru.
TC 6	Ano	Uživatelé v rámci aplikace neprovádí žádné náročné operace. V drtivé většině jde o operace GET k získání cachovaných údajů. Systém je na základě informací ze systému Prometheus schopen zjistit, zda je požadovaný počet spuštěných instancí komponent dostačující. Pokud není, tak zafunguje proces automatického horizontálního škálování.

TC 7	Ano	Kubernetes automaticky spouští nové instance na základě nastavených počtu replik.
TC 8	Ano	Ke zobrazení celkového stavu systému slouží Grafana. Pomocí webového rozhraní vizualizuje data ze systému Prometheus, který monitoruje celý systém.
TC 9	Ano	Všechny logy jsou zpracovávány a ukládány pomocí skupiny systému ELK stack. Logy jsou dostupné v aplikaci Kibana, která poskytuje webové rozhraní pro vizualizaci všech logů.
TC 10	Ano	Všechny konfigurace jsou ukládány v Kubernetes objektech ConfigMap a Secret. Všechny změny jsou dostupné v rámci několik sekund. Ve běžících kontejnerech, kde se konfigurace používá pomocí „environemnt variables“ se tato změna neprojeví dokud je nerestartujeme. Pokud chceme změny konfigurace zobrazit v běžících kontejnerech máme dvě možnosti: použít ConfigMap jako volume (doba propagace se poté rovná *synchronizační perioda + doba propagace do lokální cache*) nebo poslouchat na události změn konkrétních objektů typu ConfigMap.
TC 11	Ano	Vzhledem ke zvolené mikroservisní architektuře by dekompozice mikroslužeb neměl být žádný problém. Jednoduše se jen nové mikroslužby přidají do systému po vzoru ostatních a stará mikroslužba se buď upraví v její další verzi nebo se úplně ze systému smaže.
TC 12	Ano	Neoprávněné přístupy jsou zachyceny již v komponentě API Gateway. Podrobnosti takových požadavků jsou logovány a dostupné v aplikaci Kibana. Zároveň API gateway umožňuje při častých neoprávněných dotazech ze stejné IP adresy notifikovat operační tým.
TC 13	Ano	API gateway si uchovává v omezené časové době přístupy jednotlivých uživatelů a je schopna při překročení nastavených limitů uživatele zablokovat.

Tabulka 7.1: Přehled splnění testovacích scénářů

7.6 Kontrola principů Twelve-factor app

V této sekci je vyhodnoceno, zda navržená architektura splňuje principy Twelve-Factor app popsaných v kapitole 6.4.

Princip	**Výsledek**	**Popis**
I. Codebase	Splněno	Každá mikroslužba má právě jeden vlastní repozitář a je nasazená na více prostředích.
II. Dependencies	Splněno	Přestože architektura nespecifikuje zvolenou technologii, předpokládá, že všechny služby jsou samostatné a nevyžadují závislosti definované infrastrukturou.
III. Config	Splněno	Kubernetes poskytuje mechanismy, které do běžících „podů“ nastaví „environment variables“. Konkrétně se jedná o objekty ConfigMaps a Secrets. Jejich hodnoty jsou vždy nastavovány dle prostředí.
IV. Backing Services	Splněno	Všechny služby komunikují s podpůrnými systémy pomocí URL adres nastavených v „environment variables“. Ty jsou konfigurované přes objekty ConfigMaps a Secrets.
V. Build, release, run	Splněno	CI/CD pipeline je rozdělena do tří fází(ta se může skládat z několika kroků). První fází je sestavení spustitelného artefaktu, následně je vytvořen Docker Image, který je nahrán do interního repozitáře a v poslední fázi se nová verze nasadí.
VI. Processes	Splněno	Všechny mikroslužby jsou bezstavové Docker kontejnery. Pro ukládání dat jsou využity databáze.
VII. Port binding	Splněno	Všechny služby v systému jsou Docker kontejnery. Tím pádem je automaticky toto pravidlo splněno. Navíc Kubernetes zavádí nad skupinou kontejnerů stejného typu abstrakci nazvanou Service. Takže v rámci systému jsou ostatní komponenty dostupné pod jménem objektu Service.
VIII. Concurrency	Splněno	Kubernetes zajišťuje, aby v systému vždy běžel nakonfigurovaný počet komponent daného typu. Systém zároveň využívá automatického horizontálního škálování na základě metrik ze systému Prometheus.
IX. Disposability	Splněno	Vzhledem k tomu, že všechny mikroslužby jsou kontejnerizované a bezstavové, tak je možné je libovolně zapínat a vypínat bez vlivu na stav systému.

X. Dev/prod parity	Splněno	Všechna prostředí by měla běžet nad stejnou verzí Kubernetes. Tím dosáhneme maximální možné podobnosti mezi prostředími.
XI. Logs	Splněno	Pro logování je využit ELK stack. Ten kompletně vyhovuje požadavkům tohoto pravidla.
XII. Admin processes	Splněno	Mikroslužby používají pro migraci schématu databáze technologii Liquibase. V případě neočekávaných jednorázových úkolů je možné také použít KubernetesJob.

Tabulka 7.2: Přehled splnění principů Twelve-Factor app

Kapitola 8

Proof of concept

V této kapitole je popsán vytvořený prototyp architektury včetně některých procesů. Je vyhodnoceno, zda prototyp splňuje nároky systému definovaných testovacími scénáři. Splnění koncepčních reakcí architektury na podněty testovacích scénářů jsou již vyhodnoceny na konci předchozí kapitoly. Zde se zaměříme na měřitelné podmínky scénářů a jejich splnění.

Speciálně pro účely prototypu byla pořízena doména *vodvarka-diplomka.cz*. Na této veřejně dostupné doméně je celý prototyp provozován. Pro testování zátěže REST API a kontroly dostupnosti služeb je použit nástroj Locust[1].

8.1 Prostředí

Vzhledem k tomu, že vytvoření infrastruktury pro účely provozu aplikace není předmětem práce, ale zároveň je potřeba ke správnému otestování použít více než jeden uzel, jsou použity tzv. Managed Kubernetes. Jedná se o pronajatý Kubernetes cluster o jehož infrastrukturu se stará cloudový provider. Pro tento prototyp byly využity Managed Kubernetes od cloudového providera DigitalOcean[2] s následující konfigurací:

- Kubernetes v1.22.8
- 7 uzlů
- 14 vCPUs (2 vCPU na každém uzlu)
- 28 GB paměti (4 GB na každém uzlu)
- 560 GB místa na disku (80 GB na každém uzlu)

8.2 Technologie

Návrh architektury z velké části říká jaké technologie použít (ELK stack, Promethues atd.). Obsahuje ale i komponenty, u kterých je popsaná funkcionalita, ale není vyžadována

[1] `https://locust.io`
[2] `https://www.digitalocean.com/products/kubernetes`

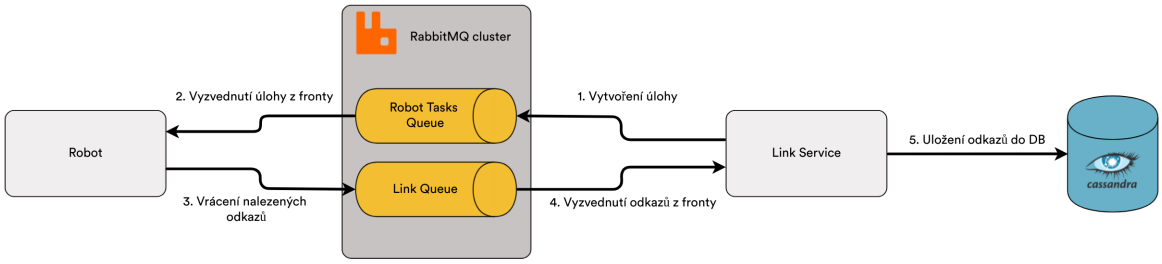
specifická technologie. Zároveň není nutné v rámci návrhu specifikovat jazyk, ve kterém jsou psané jednotlivé mikroslužby. Konkrétní technologie použité v prototypu jsou následující:

- API Gateway - Kong[3],
- Identity and Access Management - Keycloak[4],
- Mikroslužby - Java a Spring Boot 2. Java a Spring byly zvoleny především kvůli mým předchozím zkušenostem s těmito technologiemi,
- Git - Gitlab[5],
- CI/CD - Gitlab CI/CD[6].

8.3 Implementované procesy

8.3.1 Získávání odkazů z domén

Tento proces simuluje získávání dat z internetu pomocí robotů. V rámci prototypu je v Link Service periodicky generována úloha pro roboty "Najdi všechny odkazy vedoucí z domény X". Služba má v sobě statický seznam domén, ze kterého se náhodně vybírá. Jako mock systému robotů byla vytvořena další Spring Boot komponenta, která při zahájení úkolu náhodně vybere nějaké množství domén ze statického seznamu. Tyto domény následně „robot“ po menších dávkách posílá do messaging komponenty.

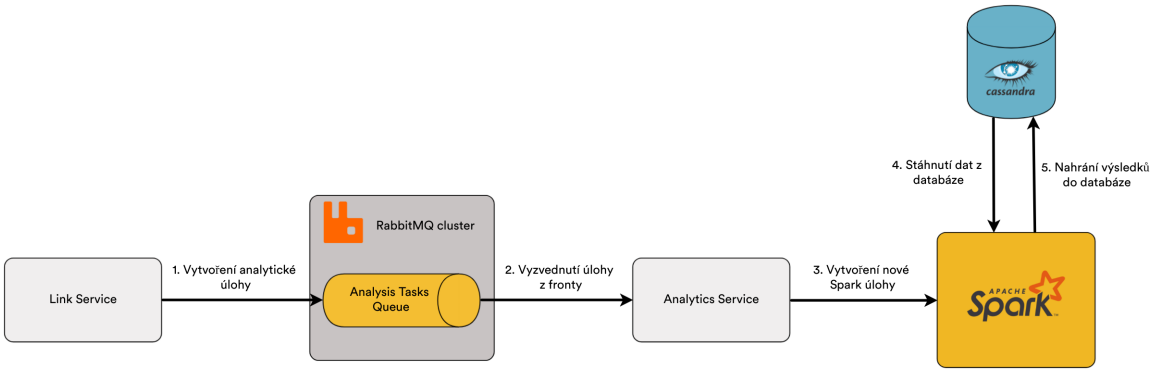


Obrázek 8.1: Diagram procesu získávání odkazů konkrétní domény

8.3.2 Analýza zpětných odkazů

Tento proces simuluje analýzy dat pomocí komponenty Spark. V rámci prototypu jsou úlohy periodicky generovány v Link Service. Vygenerované úlohy jsou vyzvednuté z fronty v Analytics Service, která spustí novou Spark úlohu. Data se stahují přímo z databáze Apache Cassandra a následně se tam nahrávají i samostatné výsledky.

[3] `https://konghq.com/kong`
[4] `https://www.keycloak.org`
[5] `https://gitlab.com`
[6] `https://docs.gitlab.com/ee/ci/`



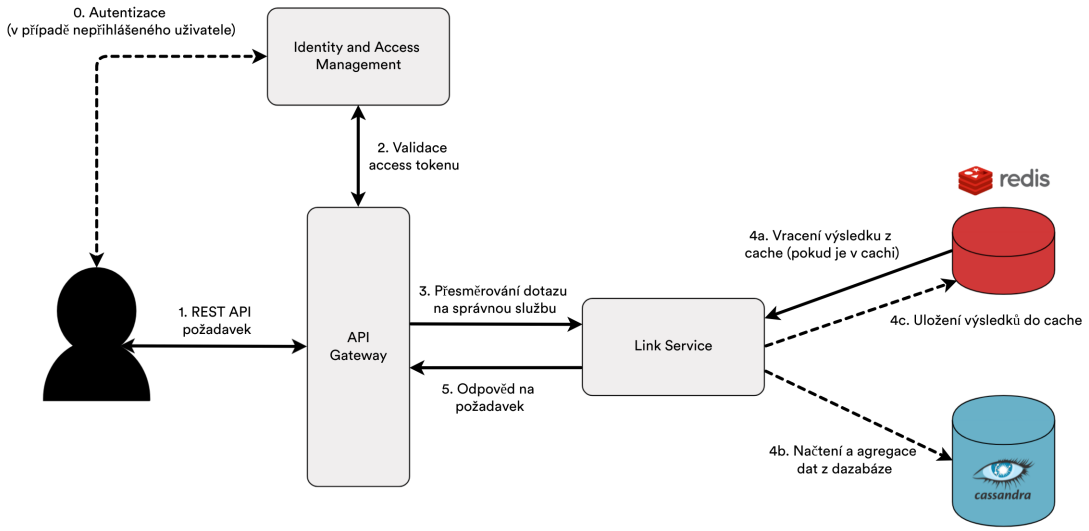
Obrázek 8.2: Diagram procesu analýzy zpětných odkazů

V rámci prototypu je implementován velmi jednoduchý Spark job napsaný pomocí technologie PySpark. Analyzuje všechny odkazy nalezené v daném datu (v prototypu vždy dnešek) a počítá kolik odkazů vede na jakou doménu.

```
today = datetime.now().strftime("%Y-%m-%d")

# Load links gathered today from Apache Cassandra
links = spark.read.format("org.apache.spark.sql.cassandra") \
    .options(table="links", keyspace="links") \
    .load()
links =links.filter(F.col("date") ==F.lit(today))

# Count how many links go to every domain
links_agg = links.groupBy("target").count()\
    .toDF("domain", "incoming_links_count")\
    .withColumn("date", F.lit(today))

# Save data to Apache Cassandra
links_agg.write.format("org.apache.spark.sql.cassandra")\
    .options(table="link_domain_aggregation", keyspace="links")\
    .mode("append")\
    .save()
```

8.3.3 API pro získání nejpopulárnějších webových domén

Tento proces simuluje uživatelský požadavek na externí REST API systému. Uživatel musí být autentizovaný. Jedná se o jednoduchý dotaz, při kterém uživatel chce vrátit padesát nejpopulárnějších domén(čím více odkazů na doménu vede, tím je populárnější) na základě výsledných dat spočítaných v předchozím procesu. Při prvním požadavku se data načtou ze sloupcové databáze Apache Cassandra (relativně náročná operace) a uloží se do Redis databáze. Zde slouží databáze jako cache k rapidnímu zrychlení dotazů.



Obrázek 8.3: Diagram procesu REST API požadavku uživatele

8.4 Ověření testovacích scénářů

8.4.1 TC 1 - Selhání komponenty

8.4.1.1 Popis testu

Speciálně pro tento test byly do komponenty Link Service přidány dva nové REST API endpointy: *POST /api/links/admin/shutdown* a *GET /api/links/admin/health*. Prvně jmenovaný endpoint danou instanci vypne a druhý pouze vrací jednoduchý text říkající, že komponenta je naživu. V systému běží tři repliky komponenty Link Service. V rámci testu budeme posílat požadavky na endpoint *health* v přibližné kadenci dvacet požadavků za sekundu. Na začátku testu jednu repliku vypneme a budeme pozorovat, zda všechny požadavky proběhnou v pořádku a zda komponenta bude automaticky restartována. Použitý Python program je uveden v příloze B.1.

8.4.1.2 Naměřené výsledky

Instance komponenty Link Service prošla čtyřmi následujícími fázemi, viz výstupy příkazu *kubectl get pods -n link-service*[7]:

```
NAME                            READY   STATUS    RESTARTS   AGE
link-service-6cd8ddd758-f7vwp   1/1     Running   0          3m12s
link-service-6cd8ddd758-rfjvl   1/1     Running   0          3m15s
link-service-6cd8ddd758-vk2s5   1/1     Running   0          3m30s

NAME                            READY   STATUS     RESTARTS   AGE
```

[7]kubectl - CLI příkaz umožňující ovládat Kubernetes cluster

```
link-service-6cd8ddd758-f7vwp  0/1    Completed  0          3m16s
link-service-6cd8ddd758-rfjvl  1/1    Running    0          3m19s
link-service-6cd8ddd758-vk2s5  1/1    Running    0          3m34s

NAME                           READY  STATUS     RESTARTS     AGE
link-service-6cd8ddd758-f7vwp  0/1    Running    1 (3s ago)   3m18s
link-service-6cd8ddd758-rfjvl  1/1    Running    0            3m21s
link-service-6cd8ddd758-vk2s5  1/1    Running    0            3m36s

NAME                           READY  STATUS     RESTARTS     AGE
link-service-6cd8ddd758-f7vwp  1/1    Running    1 (16s ago)  3m31s
link-service-6cd8ddd758-rfjvl  1/1    Running    0            3m34s
link-service-6cd8ddd758-vk2s5  1/1    Running    0            3m49s
```

Výstupy příkazu *kubectl* jasně říkají, že komponenta byla bez problému restartována. Ze shrnutí testování REST API požadavku viz Obrázek 8.4 je možné vidět, že všechny požadavky proběhly v pořádku.

Type	Name	# Requests	# Fails	Median (ms)	90%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/api/links/admin/health	694	0	31	40	120	35	25	624	15	18.3	0

Obrázek 8.4: Výsledek měření TC 1 v programu Locust

8.4.1.3 Hodnocení

Z výsledků testování je možné odvodit, že detekce selhání komponenty trvala přibližně **5 vteřin** a nová instance byla úspěšně spuštěna do **20 vteřin**. Tyto hodnoty vyhovují požadavkům testovacího scénáře a můžeme ho tedy prohlásit za **splněný**.

8.4.2 TC 2 - Load balancing

8.4.2.1 Popis testu

Pro účely tohoto testu přidáme do služby Link Service endpoint, který vrátí název Kubernetes Podu, ve kterém aplikace běží. Všech deset požadavků by mělo být rovnoměrně distribuováno mezi instance služby. V rámci tohoto testu použijeme tři repliky.

8.4.2.2 Naměřené výsledky

Postupně spuštěné dotazy vrátily následující výsledky:

1. link-service-6cbf445457-svjqg
2. link-service-6cbf445457-qm62f
3. link-service-6cbf445457-d5pw5
4. link-service-6cbf445457-svjqg
5. link-service-6cbf445457-qm62f

6. link-service-6cbf445457-d5pw5
7. link-service-6cbf445457-svjqg
8. link-service-6cbf445457-qm62f
9. link-service-6cbf445457-d5pw5
10. link-service-6cbf445457-svjqg

8.4.2.3 Hodnocení

Z naměřených výsledků je vidět, že požadavky jsou rovnoměrně distribuovány mezi jednotlivé instance služby. Testovací scénář můžeme označit za **splněný**.

8.4.3 TC 3, TC 4 - Nasazení nové verze komponenty

Vzhledem k naprosto stejnému procesu pro nasazení nových verzí komponent do různých prostředí, je vyhodnocení scénáře TC 3 a TC 4 spojeno do jedné sekce.

8.4.3.1 Popis testu

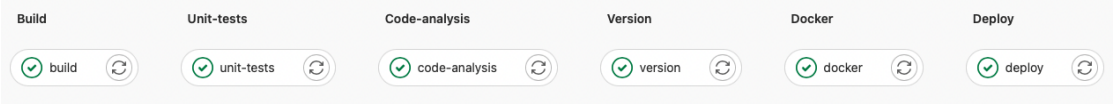
V rámci testovacího scénáře vznikne nová verze mikroslužby Link Service. Automatická CI/CD pipeline se spustí po „pushnutí“ do příslušné Git větve. Během celého procesu je pro otestování dostupnosti systému periodicky (25 požadavků/s) provoláván endpoint *health* zmíněný v předchozích scénářích. Použitý program je uveden v příloze B.2.

8.4.3.2 Naměřené výsledky

Komponenta Link Service byla úspěšně aktualizována na nejnovější verzi (0.0.14 →0.0.15). Důkazem jsou výsledky příkazu *kubectl get deployments -o wide -n link-service* vykonaném před spuštěním aktualizace a po jejím dokončení:

```
NAME         READY  UP-TO-DATE  AVAILABLE  AGE    CONTAINERS    IMAGES
link-service 3/3    3           3          7d20h  link-service  .../link-service:0.0.14

NAME         READY  UP-TO-DATE  AVAILABLE  AGE    CONTAINERS    IMAGES
link-service 3/3    3           3          7d20h  link-service  .../link-service:0.0.15
```

Na obrázku 8.5 jsou znázorněné všechny kroky, které byly při vydávání nové verze udělány. Součástí bylo provedení jednotkových testů a statické analýzy kódu nástrojem SonarCloud. Integrační testy nejsou součástí prototypu, ale dle návrhu by měly být provedeny mezi fází jednotkových testů a analýzou kódu. Z obrázku 8.6 je patrné, že nasazení nové verze nemělo vliv na dostupnost aplikace.



Obrázek 8.5: Úspěšný běh CI/CD pipeliny

Type	Name	# Requests	# Fails	Median (ms)	90%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/api/links/admin/health	12053	0	30	42	160	36	22	603	15	25	0

Obrázek 8.6: Výsledek měření TC 3,4 v programu Locust

8.4.3.3 Hodnocení

Celkový čas nasazení nové verze byl přibližně 7 minut a 30 sekund. Tento čas se odvíjí od množství jednotkových a integračních testů. V reálném provozu by vše trvalo déle. Rozhodně by to ale nemělo přesáhnout požadovaných 60 minut. Úspěšně se nahradily všechny instance a nedošlo k žádnému výpadku funkcionality. Scénář můžeme označit za **splněný**.

8.4.4 TC 5 - Prioritizace analytických požadavků

Vzhledem k použití Spark Operatoru[8] je škálovatelnost zpracování analytických požadavků v zásadě neomezena. Při každém spuštění nové Spark aplikace se vytvoří nový izolovaný kontejner se spark driverem. Ten následně komunikuje s Kubernetes API serverem a požaduje vytvoření worker kontejnerů speciálně pro tuto aplikaci. Po dokončení úlohy jsou všechny kontejnery smazány. Počet worker kontejnerů může být určen staticky nebo alokován dynamicky na základě zátěže konkrétní aplikace. V případě používání systému v cloudu je vhodné zapnout automatické škálování celého clusteru. Tím zajistíme dostatečné prostředky pro všechny Spark aplikace.[28]

Jelikož jsou úlohy vyzvedávány z prioritizované fronty, tak je zpracování požadavků s prioritou „HIGH“ velmi rychlé. Spuštění Spark aplikace je realizováno přes Kubernetes API, které vše zařídí za nás. Samotné spuštění je tedy velmi rychlé, ale počet běžících aplikací paralelně je omezen zdroji clusteru. Testovací scénář můžeme označit za **splněný**. Z důvodu velmi omezených zdrojů v clusteru použitého pro prototyp je testovací scénář ověřen pouze logickou úvahou v předchozím odstavci.

8.4.5 TC 6 - Velké množství připojených uživatelů

8.4.5.1 Popis testu

Test probíhá pomocí nástroje Locust, který simuluje specifikaci testovacího scénáře: 1000 připojených uživatelů a 5000 požadavků v jedné minutě. Testován bude proces dotazu na 500 nejpopulárnějších domén popsaný v sekci 8.3.3. V Kubernetes clusteru poběží tři repliky. V testu je ignorována autentizace. Použitý program je uveden v příloze B.3.

8.4.5.2 Naměřené výsledky

Naměřené výsledky jsou zobrazené na obrázku 8.7.

[8] `https://github.com/GoogleCloudPlatform/spark-on-k8s-operator`

Type	Name	# Requests	# Fails	Median (ms)	90%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/api/links/popular-domains	5052	0	57	240	7300	367	21	11279	0	100	0

Obrázek 8.7: Výsledek měření TC 6 v programu Locust

8.4.5.3 Hodnocení

Přestože jednotky případů zaznamenaly velmi pomalé zpracování viz položka „Max (ms)“, tak 90% percentil času zpracování je **240 ms**. Scénář tedy můžeme označit za **splněný**.

8.4.6 TC 7 - Automatické škálování

8.4.6.1 Popis testu

V rámci testu vytvoříme vlastní metriku „http_server_requests_seconds_average_rate“. Ta nám říká, kolik požadavků průměrně dorazilo na komponentu v rámci jedné sekundy. Průměr se bere z dat za poslední jednu minutu. Metrika je založená na předkonfigurované metrice „http_server_requests_second_count“ a je definována v komponentě Prometheus Adapter.

```
custom:
    - seriesQuery: '{__name__=~"^http_server_requests_seconds_.*"}'
      resources:
        overrides:
          kubernetes_namespace:
            resource: namespace
          kubernetes_pod_name:
            resource: pod
      name:
        matches: "^http_server_requests_seconds_count(.*)"
        as: "http_server_requests_seconds_average_rate"
      metricsQuery: sum(rate(<<.Series>>{<<.LabelMatchers>>}[1m])) by (<<.GroupBy>>)
```

Tuto metriku použijeme pro automatické škálování komponenty Link Service pomocí Kubernetes HPA. Nastavíme minimální množství instancí na dva a maximální množství na osm. *TargetAverageValue* nastavíme na pět požadavků za sekundu. V průběhu testu budeme pomocí nástroje Locust různě měnit zátěž na cílovou komponentu a sledovat, zda se automaticky přidávají nebo odebírají instance.

8.4.6.2 Naměřené výsledky

Následuje znázornění výsledků příkazu *kubectl get hpa -n link-service*, který informuje o stavu HPA. Jsou zde vyobrazeny postupně 4 fáze:

1. Bez zátěže
2. Průměrně 20 požadavků za sekundu
3. Průměrně 30 požadavků za sekundu

4. Bez zátěže

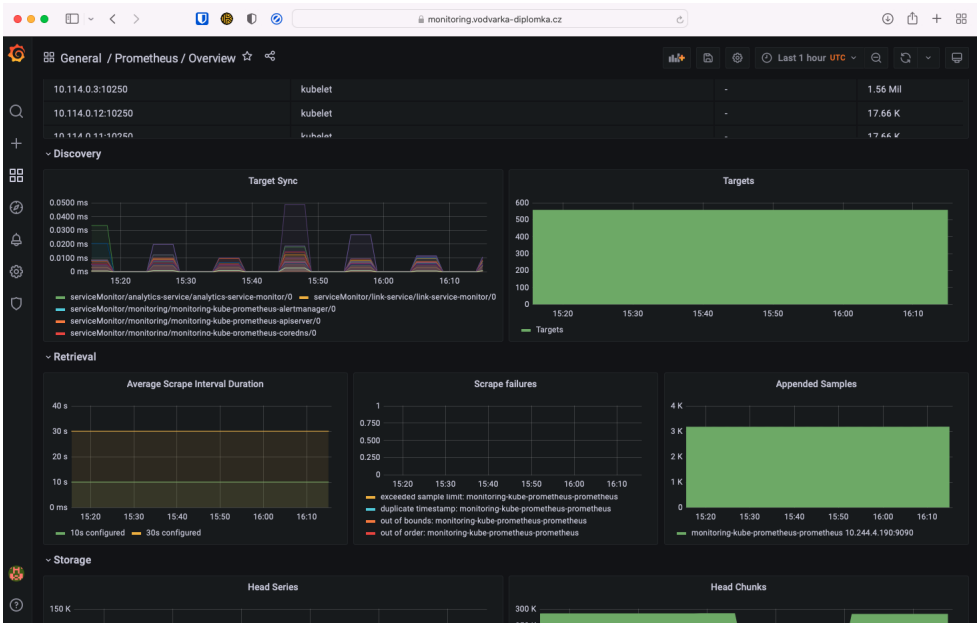
```
NAME          REFERENCE                TARGETS   MINPODS   MAXPODS   REPLICAS
link-service  Deployment/link-service  1/5       2         8         2

NAME          REFERENCE                TARGETS   MINPODS   MAXPODS   REPLICAS
link-service  Deployment/link-service  20/5      2         8         4

NAME          REFERENCE                TARGETS   MINPODS   MAXPODS   REPLICAS
link-service  Deployment/link-service  30/5      2         8         6

NAME          REFERENCE                TARGETS   MINPODS   MAXPODS   REPLICAS
link-service  Deployment/link-service  1/5       2         8         2
```

8.4.6.3 Hodnocení

Z naměřených výsledků je vidět, že dochází k automatickému škálování na základě námi definovaných metrik. Můžeme tedy označit testovací scénář za **splněný**.

8.4.7 TC 8 - Celkový stav systému

O zobrazení stavu celého systému na jednom místě se stará nástroj Grafana. Případně je možné využít i Kubernetes Dashboard, který je také schopen zobrazit relevantní informace ohledně stavu clusteru. V prototypu běží Grafana na veřejně dostupné adrese `https://monitoring.vodvarka-diplomka.cz`. V této aplikaci je možné procházet všechny metriky dostupné v systému a vizualizovat je. Na obrázku 8.8 je snímek obrazovky jednoho z dashboardů zobrazující stav systému Prometheus.



Obrázek 8.8: Snímek obrazovky nástroje Grafana

Jelikož Grafana plní všechny funkce požadované testovacím scénářem, tak ho můžeme označit za **splněný**.

8.4.8 TC 9 - Podrobnosti chybového stavu

8.4.8.1 Popis testu

Speciálně pro tento scénář byl do služby Link Service přidán endpoint *POST /api/-links/admin/error*, který jen vyhodí výjimku. V průběhu testu tento endpoint několikrát provoláme a budeme zkoumat, zda je dostupný v systému Kibana. Kibana v prototypu běží na adrese *https://logs.vodvarka-diplomka.cz*.

8.4.8.2 Naměřené výsledky

Na obrázku 8.9 můžeme vidět, že log chyby se úspěšně dostal až do nástroje Kibana, kde je dostupný uživatelům včetně nejrůznějších metadat.

t kubernetes.node.labels.region	fra1
t kubernetes.node.labels.topology_kubernetes_io/region	fra1
t kubernetes.node.name	pool-5l8d1xere-cobwd
t kubernetes.node.uid	daaca01a-aef2-46cd-b67c-3c4bd30acdd8
t kubernetes.pod.ip	10.244.3.4
t kubernetes.pod.name	link-service-7b75849d66-rdnwn
t kubernetes.pod.uid	ba61294c-8afe-4799-81d4-a9383c867a72
t kubernetes.replicaset.name	link-service-7b75849d66
t log.file.path	/var/log/containers/link-service-7b75849d66-rdnwn_link-service_link-service-6747fe95e8f8f284d63511c2bba9e7502d7e73ad5bd5c94f7a7a806bcf431535.log
# log.offset	10,478
t message	2022-05-08 16:53:03.800 ERROR 1 --- [nio-8080-exec-7] o.a.c.c.C.[.[.[/].[dispatcherServlet] : Servlet.service() for servlet [dispatcherServlet] in context with path [] threw exception [Request processing failed; nested exception is java.lang.RuntimeException: Mysterious exception thrown on purpose] with root cause

Obrázek 8.9: Log chyby v nástroji Kibana

8.4.8.3 Hodnocení

Vytvořené chyby byly v pořádku zalogovány a jsou dostupné v systému Kibana včetně doplňujících informací. Testovací scénář je **splněn**.

8.4.9 TC 10 - Změna konfigurace

8.4.9.1 Popis testu

Pro tento test vytvoříme v Link Service další endpoint: *GET /api/links/admin/favourite-animal*. Jak již název napovídá, tak endpoint vrací název oblíbeného zvířete. Ten je nastaven externí konfigurací. V průběhu testu tuto konfiguraci změníme a budeme pozorovat změny.

V rámci prototypu je všechna externí konfigurace zavedena do služeb pomocí environment proměnných, které v Kubernetes nejde měnit dynamicky. Proto je potřeba po změně konfigurace danou službu restartovat.

8.4.9.2 Naměřené výsledky

- Výsledek před změnou konfigurace: „Dog“.
- Výsledek po změně konfigurace bez restartu: „Dog“.
- Restart systému: *kubectl rollout restart -n link-service deployment link-service*.
- Výsledek po změně konfigurace a po restartu: „Cat“.
- Trvání restartu: cca 35 sekund.

8.4.9.3 Hodnocení

Změna se úspěšně provedla do požadovaného časového limitu. Celkový čas se odvíjí především od trvání restartu všech služeb. V testovacím scénáři je požadována plná automatizace, ale v tomto případě byl vyžadován manuální restart. Pokud by bylo opravdu nutné vše propagovat automaticky, je potřeba zvolit jiný způsob distribuce konfigurace, než jsou environment proměnné. Testovací scénář označíme jako **částečně splněný**.

8.4.10 TC 11 - Dekompozice komponenty

8.4.10.1 Popis testu

Předpokladem testu je existující endpoint *GET /api/test/hello* ve službě Link Service. Speciálně pro tento test vytvoříme novou mikroslužku Test Service, která převezme funkcionalitu endpointu. Po nasazení nové komponenty přesměrujeme všechny požadavky začínající */api/test* na komponentu Test Service. Následně odstraníme danou funkcionalitu ze služby Link Service. Po celou dobu testu probíhají v nástroji Locust požadavky na tuto funkcionalitu pro kontrolu dostupnosti viz příloha B.4.

8.4.10.2 Naměřené výsledky

Úspěšně proběhlo nasazení nové komponenty Test Service a následného přesměrování požadavků. Úspěšně proběhlo i smazání funkcionality z původní mikroslužby. Na obrázku 8.10 je vidět, že všechny požadavky proběhly úspěšně a nedošlo k žádnému výpadku.

Type	Name	# Requests	# Fails	Median (ms)	90%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/api/test/hello	19220	0	27	45	160	35	20	2179	11	9.2	0

Obrázek 8.10: Výsledek měření TC 11 v programu Locust

8.4.10.3 Hodnocení

Všechny podmínky testovacího scénáře byly splněny a můžeme tedy i celý testovací scénář označit za **splněný**.

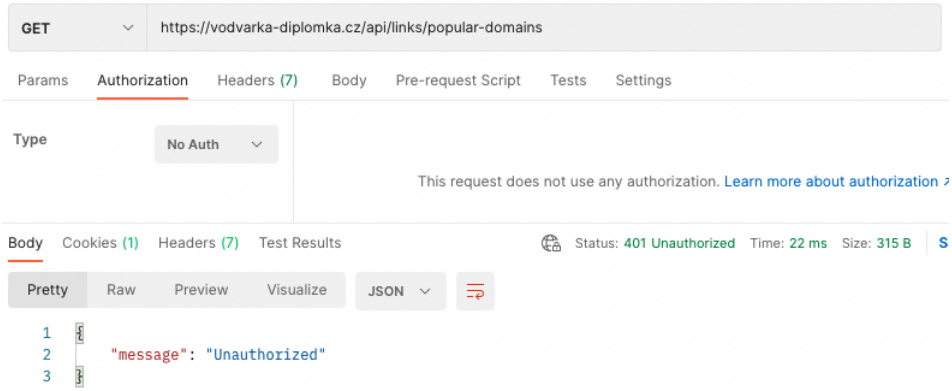
8.4.11 TC 12 - Neoprávněný přístup

8.4.11.1 Popis testu

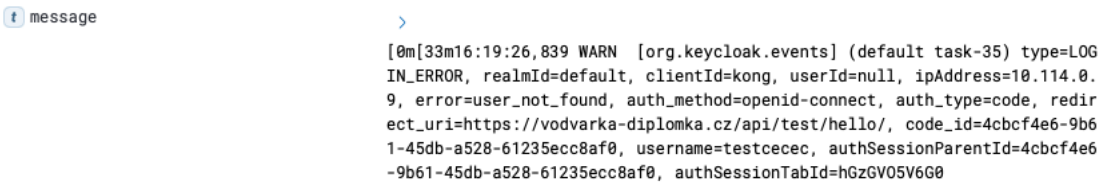
Princip testu je velmi jednoduchý. Provedeme neautorizovaný požadavek na REST API systému a budeme zkoumat jeho reakci. Zároveň se pokusíme do systému přihlásit nesprávnými údaji. Předpokladem celého testu je implementovaná autentizace alespoň pro jednu službu. Pro zjednodušení nebudeme v rámci prototypu implementovat notifikační mechanimus v případě častých pokusů o přihlášení. Propojení API Gateway a Keycloku je realizováno pomocí pluginu Kong OIDC[9].

8.4.11.2 Naměřené výsledky

Na obrázku 8.11 můžeme vidět, že neautorizovaný request se neprovedl, vrátil stavový kód 401. Na obrázku 8.12 jsou vidět podrobnosti nepovedeného přihlášení z centrální vizualizace logů.



Obrázek 8.11: Neautorizovaný požadavek v aplikaci Postman



Obrázek 8.12: Podrobnosti nepovedeného pokusu o přihlášení

[9] https://github.com/revomatico/kong-oidc

8.4.11.3 Hodnocení

Všechny kritéria testovacího scénáře implementované v prototypu byly splněny. Pro kompletní splnění je potřeba doplnit automatické notifikace při podezřelém chování. Testovací scénář můžeme označit jako **částečně splněný**.

8.4.12 TC 13 - Detekce těžení dat

8.4.12.1 Popis testu

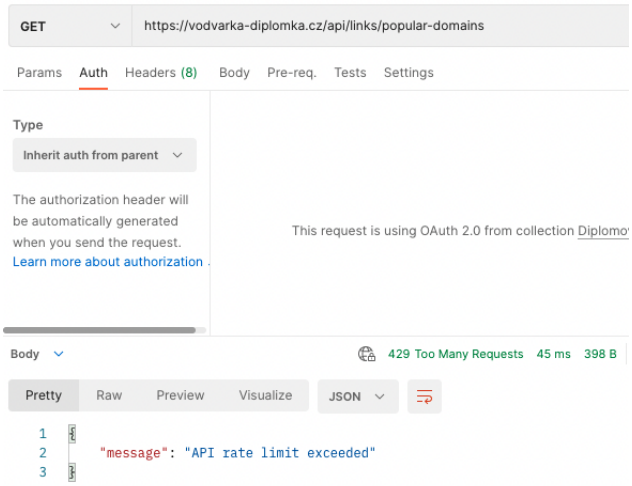
Pro realizaci testu je použit API Gateway plugin Rate Limiting[10]. V průběhu testu je nastaven limit na každého uživatele 1000 požadavků za jednu hodinu. Pomocí nástroje Locust tento limit překročíme a budeme zkoumat reakci systému. Použitý program je uveden v příloze B.5.

8.4.12.2 Naměřené výsledky

Na obrázku 8.13 můžeme vidět 1365 provedených požadavků. Z toho přesně 365 požadavků skončilo chybou. Konkrétněji skončily chybou všechny požadavky, které měly pořadové číslo větší než tisíc. Na obrázku 8.14 je zobrazena odpověď na požadavek, který byl poslán po překročení limitu.

Type	Name	# Requests	# Fails	Median (ms)	90%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/api/links/popular-domains	1365	365	39	60	220	49	28	381	11	10	10
POST	/auth/realms/default/protocol/openid-connect/token	1	0	204	200	200	204	204	204	3510	0	0

Obrázek 8.13: Výsledek měření TC 13 v programu Locust



Obrázek 8.14: Odpověď na požadavek po překročení limitu v aplikaci Postman

[10] `https://docs.konghq.com/hub/kong-inc/rate-limiting/`

8.4.12.3 Hodnocení

Z naměřených výsledků je zřejmé, že všechny požadavky, které jsou daným uživatelem poslány nad definovaný limit se neprovedou. API Gateway se tedy stará o to, aby uživatel byl dočasně zablokován do doby, než se limit obnoví. Odpovědí na takové požadavky je stavový kód 429 a jasná zpráva o tom, co se děje. Použitá API Gateway Kong bohužel neumí posílat notifikace, takže správce systému nelze upozornit na překročení limitu API. Zároveň použitý plugin neumí zaznamenávat podrobné informace o uživateli, který limit překročil. Hlavní funkcionalita scénáře je tedy splněna, ale podpůrné mechanismy splněny nejsou. Testovací scénář tedy označíme jako **částečně splněný**.

8.5 Nalezené nedostatky

Přestože prototyp splňuje drtivou většinu požadavků definovaných testovacími scénáři, tak v průběhu se objevily drobné nedostatky. Největším nedostatkem bylo spojení kódu služeb s definicí jejich Helm tabulek do jednoho repozitáře. To znamenalo, že nemohlo dojít k upravení drobností v Helm definicích bez vydání nové verze celé služby. Vhodnějším řešením by bylo kód a Helm definice verzovat odděleně.

Kapitola 9

Závěr

Cílem této práce bylo navržení distribuované softwarové architektury pro práci s daty online trhu a jejich analýzu. Architektura byla navržena ve stylu *event-driven microservices* s kombinací synchronní (REST API) a asynchronní (messaging) komunikace. Celý systém je orchestrován pomocí nástroje Kubernetes. Byla vytvořena prototypová implementace architektury obsahující kompletní business proces získání a analýzy zpětných odkazů včetně podpůrných DevOps procesů. Navržená architektura podporuje v maximální míře horizontální škálování, automatizaci procesů a další rozvoj. V rámci práce proběhla analýza architektonických požadavků vlastníků systému a identifikace problémů současné architektury. Na základě těchto informací byly vytvořeny testovací scénáře nejen pro formální ověření funkčnosti navržené architektury, ale i pro podporu správného rozhodování v samotném návrhu. Vytvořený prototyp byl testovacími scénáři ověřen a splňuje drtivou většinu požadavků.

9.1 Možnosti dalšího postupu

Práce splňuje vše, co bylo definováváno v požadavcích zadání práce. Jedná se o novou architekturu již existujícího systému, proto by mohl být systém implementován v praxi. V případě rozhodnutí pro reálné využití návrhu definovaného v této práci by bylo vhodné implementovat v rámci prototypu více business procesů a přidat testovací scénáře. Tím bychom maximálně architekturu ověřili a případně vyřešili potencionální problémy.

Samotný produkt je stále v začátcích a metodiky pro sběr a analýzu dat se jistě budou v průběhu času vyvíjet. Možností rozvoje, vylepšení nebo rozšíření architektury o další komponenty je jistě mnoho a je proto důležité, aby tyto akce softwarová architektura maximálně podporovala.

Literatura

[1] ABDELRAZIK, A. *Docker vs. Kubernetes vs. Apache Mesos: Why What You Think You Know is Probably Wrong* [online]. [cit. 6. 4. 2022]. Dostupné z: https://d2iq.com/blog/docker-vs-kubernetes-vs-apache-mesos.

[2] BASS, L. – CLEMENTS, P. – KAZMAN, R. *Software Architecture in Practice, 4th Edition.* Boston : Addison-Wesley Professional, 2021. ISBN 9780136885979.

[3] BURNS, B. *Designing Distributed Systems* [online]. [cit. 16. 3. 2022]. Dostupné z: https://www.oreilly.com/library/view/designing-distributed-systems/9781491983638/ch02.html.

[4] DATADOG. *10 Trends in real-world container use* [online]. [cit. 14. 5. 2022]. Dostupné z: https://www.datadoghq.com/container-report/.

[5] DOCKER. *Use containers to Build, Share and Run your applications* [online]. [cit. 16. 3. 2022]. Dostupné z: https://www.docker.com/resources/what-container.

[6] EDUCATION, I. C. *SOA (Service-Oriented Architecture)* [online]. 2021. [cit. 5. 3. 2022]. Dostupné z: https://www.ibm.com/cloud/learn/soa#toc-what-is-so-pGpP4Puh.

[7] GAMELA, A. – FIGUEIREDO, R. *Podman vs Docker: What are the differences?* [online]. [cit. 6. 4. 2022]. Dostupné z: https://www.imaginarycloud.com/blog/podman-vs-docker/.

[8] GUNJA, S. *What is DevOps? Unpacking the rise of an IT cultural revolution* [online]. [cit. 18. 4. 2022]. Dostupné z: https://www.dynatrace.com/news/blog/what-is-devops/.

[9] JEVTIC, G. *What is High Availability Architecture? Why is it Important?* [online]. 2018. [cit. 22. 2. 2022]. Dostupné z: https://phoenixnap.com/blog/what-is-high-availability.

[10] KLEPPMANN, M. *Designing Data-Intensive Applications.* Sebastopol, CA : O'Reilly Media, Inc., 2017. ISBN 9781449373320.

[11] KUBERNETES. *What is Kubernetes?* [online]. [cit. 30. 3. 2022]. Dostupné z: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

[12] KUBERNETES. *Horizontal Pod Autoscaling* [online]. [cit. 17. 4. 2022]. Dostupné z: `https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/`.

[13] KURMI, A. *Top 10 Microservices frameworks for 2022* [online]. [cit. 26. 3. 2022]. Dostupné z: `https://medium.com/microservices-architecture/top-10-microservices-framework-for-2020-eefb5e66d1a2`.

[14] MERRON, D. – IDOWU, T. *Introduction to Kubernetes Helm Charts* [online]. [cit. 19. 4. 2022]. Dostupné z: `https://www.bmc.com/blogs/kubernetes-helm-charts/`.

[15] MESOSPHERE, I. *Marathon* [online]. [cit. 30. 3. 2022]. Dostupné z: `https://mesosphere.github.io/marathon/#features`.

[16] MICROSERVICES.IO. *Pattern: Client-side service discovery* [online]. [cit. 15. 3. 2022]. Dostupné z: `https://microservices.io/patterns/client-side-discovery.html`.

[17] MICROSERVICES.IO. *Pattern: Server-side service discovery* [online]. [cit. 15. 3. 2022]. Dostupné z: `https://microservices.io/patterns/server-side-discovery.html`.

[18] MICROSOFT. *Event Sourcing pattern* [online]. [cit. 14. 3. 2022]. Dostupné z: `https://docs.microsoft.com/cs-cz/azure/architecture/patterns/event-sourcing`.

[19] PATEL, A. *Kubernetes — Architecture Overview* [online]. [cit. 30. 3. 2022]. Dostupné z: `https://medium.com/devops-mojo/kubernetes-architecture-overview-introduction-to-k8s-architecture-and-understanding-k8s-cluster-components-90e11eb34ccd`.

[20] PENHALE, C. *Patterns in Microservices Authentication with Client Certificates* [online]. [cit. 6. 4. 2022]. Dostupné z: `https://www.openlogic.com/blog/patterns-microservices-authentication-client-certificate`.

[21] PENHALE, C. *What is Quarkus?* [online]. [cit. 6. 4. 2022]. Dostupné z: `https://quarkus.io/about/`.

[22] POWELL, R. *Docker Swarm vs Kubernetes: how to choose a container orchestration tool* [online]. [cit. 30. 3. 2022]. Dostupné z: `https://circleci.com/blog/docker-swarm-vs-kubernetes/`.

[23] PRATT, M. *Introduction to Micronaut Framework* [online]. [cit. 6. 4. 2022]. Dostupné z: `https://www.baeldung.com/micronaut`.

[24] RABBITMQ. *Queues* [online]. [cit. 13. 4. 2022]. Dostupné z: `https://www.rabbitmq.com/queues.html#basics`.

[25] RICHARDS, M. *Software Architecture Patterns*. Sebastopol, CA : O'Reilly Media, Inc., 2015. ISBN 9781449373320.

[26] ROBERTS, M. *Serverless Architectures* [online]. [cit. 6. 4. 2022]. Dostupné z: `https://martinfowler.com/articles/serverless.html`.

[27] SPARK, A. *Spark Overview* [online]. [cit. 12. 4. 2022]. Dostupné z: `https://spark.apache.org/docs/latest/`.

[28] STEPHAN, A. J.-Y. – DUMAZERT, A. J. *Running Apache Spark on Kubernetes: Best Practices and Pitfalls* [online]. [cit. 10. 5. 2022]. Dostupné z: `https://databricks.com/session_na20/running-apache-spark-on-kubernetes-best-practices-and-pitfalls`.

[29] TEAM, I. C. *SOA vs. Microservices: What's the Difference?* [online]. 2021. [cit. 5. 3. 2022]. Dostupné z: `https://www.ibm.com/cloud/blog/soa-vs-microservices`.

[30] VERMEER, B. *Spring dominates the Java ecosystem with 60% using it for their main applications* [online]. [cit. 16. 5. 2022]. Dostupné z: `https://snyk.io/blog/spring-dominates-the-java-ecosystem-with-60-using-it-for-their-main-applications/`.

[31] VERNON, V. *Implementing Domain-Driven.* Massachusetts : Pearson Education, 2013. ISBN 9780321834577.

[32] WIGGINS, A. *The Twelve-Factor App* [online]. [cit. 19. 4. 2022]. Dostupné z: `https://12factor.net`.

[33] ÖZKAYA, M. *Saga Pattern for Microservices Distributed Transactions* [online]. 2021. [cit. 14. 3. 2022]. Dostupné z: `https://medium.com/design-microservices-architecture-with-patterns/saga-pattern-for-microservices-distributed-transactions-7e95d0613345`.

Příloha A

Seznam použitých zkratek

DDD	Domain Driven Design
PoC	Proof of concept
PPC	Pay per click
SEO	Search engine optimization
VCS	Version Control System
VPS	Virtuální privátní server

Příloha B

Python kód zátěžových testů pro nástroj Locust

B.1 Testovací scénář 1

```
from locust import HttpUser, task, constant_throughput

class TestCaseOne(HttpUser):

    host ="https://vodvarka-diplomka.cz"
    wait_time =constant_throughput(20)

    @task
    def health_check(self):
        self.client.get("/api/links/admin/health")

    def on_start(self):
        self.client.post("/api/links/admin/shutdown")
```

B.2 Testovací scénář 3,4

```
from locust import HttpUser, task, constant_throughput

class TestCaseThreeFour(HttpUser):

    host ="https://vodvarka-diplomka.cz"
    wait_time =constant_throughput(5)

    @task
    def health_check(self):
        self.client.get("/api/links/admin/health")
```

B.3 Testovací scénář 6

```
from locust import HttpUser, task, constant_throughput
```

```
class TestCaseThreeFour(HttpUser):

    host ="https://vodvarka-diplomka.cz"
    wait_time =constant_throughput(0.2)
    fixed_count =500

    @task
    def popular_domains(self):
        self.client.get("/api/links/popular-domains")
```

B.4 Testovací scénář 11

```
from locust import HttpUser, task, constant_throughput

class TestCaseEleven(HttpUser):

    host ="https://vodvarka-diplomka.cz"
    wait_time =constant_throughput(10)

    @task
    def health_check(self):
        self.client.get("/api/test/hello")
```

B.5 Testovací scénář 13

```
from locust import HttpUser, task, constant_throughput

class TestCaseThirteen(HttpUser):
    host ="https://vodvarka-diplomka.cz"
    wait_time =constant_throughput(10)
    auth_url ="https://keycloak.vodvarka-diplomka.cz
               /auth/realms/default/protocol/openid-connect/token"
    client_id ="kong"
    client_secret ="<<< client secret >>>"
    username ="test"
    password ="<<< user password >>>"

    @task
    def popular_domains(self):
        self.client.get("/api/links/popular-domains")

    def on_start(self):
        self.login()

    def login(self):
        response =self.client.post(self.auth_url, {
            'client_id': self.client_id,
            'client_secret': self.client_secret,
            'scope': 'openid',
            'grant_type': 'password',
            'username': self.username,
            'password': self.password
        }).json()
        self.client.headers.update({'Authorization': 'Bearer ' +response['access_token']})
```

Příloha C

Seznam digitálních příloh

Analytics_Service - Zdrojový kód analytické mikroslužby
CI_CD_Scripts - Skripty společné pro CI/CD pipelines v různých repozitářích
Kubernetes_Gitlab_Agents - Soubor konfigurující Gitlab Kubernetes agenta
Kubernetes_POC_Infrastructure - Soubory konfigurující podporující infrastrukturu
Link_Service - Zdrojový kód mikroslužby odkazů
Robot - Zdrojový kód simulující robota
Test_Service - Zdrojový kód testovací mikroslužby speciálně pro účely PoC