UNIVERZITA PARDUBICE

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

BAKALÁŘSKÁ PRÁCE

2024 Pavel Kireev

Univerzita Pardubice

Fakulta elektrotechniky a informatiky

Vývoj messengeru na bázi reaktivních a noSQL technologiích

Pavel Kireev

Bakalářská práce
2024

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2021/2022

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Pavel Kireev**
Osobní číslo:	**I21326**
Studijní program:	**B0688A140009 Informační technologie**
Téma práce:	**Vývoj messengeru na bázi reaktivních a noSQL technologiích**
Zadávající katedra:	**Katedra informačních technologií**

Zásady pro vypracování

Cílem bakalářské práce je vývoj aplikace typu messengeru založeného na bázi reaktivních a noSQL technologiích.
V teoretické části budou popsány použité technologie spojené s řešeným tématem bakalářské práce a důvod jejich výběru. Výstupem praktické části bude serverová a klientská aplikace, které budou využívat JAVA, Spring Framework, Angular, Ejabberd a dále databáze PostgreSQL a Apache Cassandra. Serverová část aplikace bude používat neblokující volání pomocí Spring WebFlux. Součástí práce bude detailní popis tvorby praktické části včetně analýzy zadání, návrhu systému, popis implementace a způsob ověření výsledků. V příloze práce bude uveden postup pro nasazení praktického výstupu včetně uvedení nutných systémových a provozních prostředků pro správný běh aplikace.

Rozsah pracovní zprávy: **40**
Rozsah grafických prací:
Forma zpracování bakalářské práce: **tištěná**

Seznam doporučené literatury:

SCHILDT, Herbert. Java: the complete reference. Eleventh edition. New York: McGraw-Hill Education, [2019]. ISBN 9781260440232.
SAINT-ANDRÉ, Peter, Kevin T. SMITH a Remko TRONCON. XMPP: the definitive guide : building real-time applications with Jabber technologies. [Sebastopol, CA]: O'Reilly, c2009. ISBN 059652126x.
Pro spring 5: an in-depth guide to the spring framework and its tools. New York, NY: Springer Science+Business Media, 2017. ISBN 9781484228074.
FOWLER, Martin. Patterns of enterprise application architecture. Boston: Addison-Wesley, c2003. ISBN 0321127420.

Vedoucí bakalářské práce: **Ing. Monika Borkovcová, Ph.D.**
Katedra informačních technologií

Datum zadání bakalářské práce: **17. prosince 2021**
Termín odevzdání bakalářské práce: **13. května 2022**

L.S.

Ing. Zdeněk Němec, Ph.D. v.r.
děkan

Ing. Jan Panuš, Ph.D. v.r.
vedoucí katedry

V Pardubicích dne 28. února 2022

Prohlašuji:

Práci s názvem Vývoj messengeru na bázi reaktivních a noSQL technologiích jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 7/2019 Pravidla pro odevzdávání, zveřejňování a formální úpravu závěrečných prací, ve znění pozdějších dodatků, bude práce zveřejněna prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 10. 05. 2024

Pavel Kireev v.r.

PODĚKOVÁNÍ

Rád bych poděkoval své rodině za finanční a morální podporu během celé doby mého studia. Rád bych také vyjádřil své velké poděkování vedoucí bakalářské práce Ing. Monice Borkovcové Ph.D. za její pomoc a cenné rady.

ANOTACE

Cílem bakalářské práce je vývoj aplikace typu messengeru založeného na bázi reaktivních a noSQL technologiích. V teoretické části budou popsány použité technologie spojené s řešeným tématem bakalářské práce a důvod jejich výběru. Výstupem praktické části bude serverová a klientská aplikace, které budou využívat JAVA, Spring Framework, Spring Web MVC, Project Reaktor, Ejabberd a dále databáze PostgreSQL a Apache Cassandra. Součástí práce bude detailní popis tvorby praktické části včetně analýzy zadání, návrhu systému, popis implementace a způsob ověření výsledků. V příloze práce bude uveden postup pro nasazení praktického výstupu včetně uvedení nutných systémových a provozních prostředků pro správný běh aplikace.

KLÍČOVÁ SLOVA

Java, Spring, XMPP, messenger, Angular, reaktivni programováni, databáze

TITLE

Vývoj messengeru na bázi reaktivních a noSQL technologiích

ANNOTATION

The aim of the bachelor thesis is to develop a messenger application based on reactive and NoSQL technologies. The theoretical part will describe the used technologies related to the topic of the bachelor thesis and the reason for their selection. The output of the practical part will be a server and client application that will use JAVA, Spring Framework, Spring Web MVC, Project Reaktor, Ejabberd, as well as PostgreSQL and Apache Cassandra databases. The thesis will include a detailed description of the creation of the practical part, including the analysis of the assignment, system design, implementation description and how to verify the results. The appendix of the thesis will include the procedure for deploying the practical output, including the necessary system and operational resources for the correct running of the application.

KEYWORDS

Java, Spring, XMPP, messenger, Angular, reactive programming, databases

OBSAH

SEZNAM ILUSTRACÍ A TABULEK....11
SEZNAM ZKRATEK....13
ÚVOD....14
1 Ejabberd XMPP Server....15
1.1 XMPP....15
1.1.1 Zobecněná architektura....16
1.1.2 XML Stream....17
1.1.3 BOSH....18
1.1.4 Přítomnost....18
1.1.5 Multi-User-Chat....19
1.2 Ejabberd....19
1.2.1 Architektura Ejabberd....20
1.2.2 Nasazení a správa....21
2 Reaktivní backend....23
2.1 Reactive Manifesto....23
2.2 Project Reactor....25
2.2.1 Flux....25
2.2.2 Mono....26
2.3 Spring WebFlux....27
3 Angular....29
3.1 Komponenty....29
3.1.1 Šablony, směrnice a datové vazby....30
3.2 Service a Dependency Injection....30
3.2.1 Směrování....31
3.3 Knihovna RxJS....31
4 Docker....33
4.1 Výhody kontejnerizace....33
4.2 Základní nástroje Docker....34
4.2.1 Dockerfile....34
4.2.2 Obrazy Docker....34

4.2.3 Docker Hub ... 35
4.3 Výhody Dockeru ... 35
5 PostgreSQL ... 38
5.1 Architektura ... 38
5.1.1 Proces na straně klienta ... 38
5.1.2 Proces Postmaster Daemon ... 39
5.1.3 Back-end proces ... 40
5.1.4 Shared Pool ... 40
5.2 Funkce ... 41
5.2.1 Syntaxe ... 41
5.3 Triggery ... 42
5.3.1 Syntaxe ... 43
5.4 Výhody a nevýhody PostgreSQL ... 43
5.4.1 Výhody ... 43
5.4.2 Nevyhody ... 45
6 Apache Cassandra ... 47
6.1 Vlastnosti ... 47
6.2 Provozování ... 48
6.3 Replikace dat ... 49
7 Messenger ... 51
7.1 Použité technologie ... 51
7.2 Požadavky ... 51
7.2.1 Funkční požadavky ... 52
7.2.2 Nefunkční požadavky ... 53
7.3 Bezpečnostní požadavky ... 53
7.4 Implementace ... 54
7.4.1 Ejabberd XMPP ... 54
7.4.2 Java Spring WebFlux Backend ... 56
7.4.3 Konfigurace databáze ... 65
7.4.4 Angular WebApp ... 67
7.5 Architektura ... 75

ZÁVĚR 77
POUŽITÁ LITERATURA 78
Přílohy..80

SEZNAM ILUSTRACÍ A TABULEK

Obrázek 1: Flux (zdroj [12]) 26

Obrázek 2: Mono (zdroj [11]) 27

Obrázek 3: Architektura PostgreSQL (zdroj: [19]) 39

Obrázek 4: Architektura aplikace (zdroj [22]) 41

Obrázek 5: Syntaxe triggeru (zdroj [23]) 43

Obrázek 6: Webová stránka administrátora: Seznam registrovaných uživatelů (zdroj vlastní) 55

Obrázek 7: Příklad Message Stanza (zdroj vlastní) 55

Obrázek 8: Příklad Presence Stanza (zdroj vlastní) 56

Obrázek 9: Příklad voláni metody Subscribe (zdroj vlastní) 58

Obrázek 10: : Implementace metody getRoster (zdroj vlastní) 58

Obrázek 11: Implementace metody signUp (zdroj vlastní) 59

Obrázek 12: Implementace metody getEnrichedGroups (zdroj vlastní) 60

Obrázek 13: Metoda findAllFriendsByUserId (zdroj vlastní) 61

Obrázek 14: Metoda findAllByJidToAndJidFromAndDate (zdroj vlastní) 61

Obrázek 15: Entita Group (zdroj vlastní) 62

Obrázek 16: Abstraktní entita (zdroj vlastní) 63

Obrázek 17: Entita GroupMessage (zdroj vlastní) 64

Obrázek 18: Cassandra klič GroupMessageKey (zdroj vlastní) 64

Obrázek 19: Implementace metody getCandidates (zdroj vlastní) 65

Obrázek 20: Schéma databáze PostgreSQL (zdroj vlastní) 66

Obrázek 21: Vytvořeni Docker clusteru pro Apache Cassandra (zdroj vlastní) 66

Obrázek 22: Schéma databáze Apache Cassandra (zdroj vlastní) 67

Obrázek 23: Implementace metody login (zdroj vlastní) 68

Obrázek 24: Implementace připojeni k Ejabberd serveru (zdroj vlastní) 69

Obrázek 25: Ukládáni souborů pomoci FileService (zdroj vlastní) 70

Obrázek 26: Autorizační stránka (zdroj vlastní) 71

Obrázek 27: Stránka pro vytvoření účtu (zdroj vlastní) 72

Obrázek 28: Ověřování přihlašovacích údajů (zdroj vlastni) 72

Obrázek 29: Logika metody autorizace uživatele (zdroj vlastní) 73

Obrázek 30: Soukromý chat (zdroj vlastní) 74

Obrázek 31: Architektura aplikace (zdroj vlastni) 75

Tabulka 1: Funkční požadavky Messengeru (zdroj vlastní) 52
Tabulka 2: Nefunkční požadavky k frontendové části messengeru (zdroj vlastní) 53
Tabulka 3: Nefunkční požadavky k backendové části messengeru (zdroj vlastní) 53
Tabulka 4: Struktura aplikaci (zdroj vlastní) 57
Tabulka 5: Typy tříd (zdroj vlastní)..68
Tabulka 6: Typy souborů komponent (zdroj vlastní) 71

SEZNAM ZKRATEK

XMPP	Extensible Messaging and Presence Protocol
YAML	YAML Ain't Markup Language
BOSH	Bidirectional-streams Over Synchronous HTTP
API	Application Programming Interface
RestAPI	Representational State Transfer Application Programming Interface
SQL	Structured Query Language
noSQL	Not only SQL
CQL	Cassandra Query Language
RFC	Request for Comments
DNS	Domain Name System
TCP/IP	Transmission Control Protocol/Internet Protocol
VM	Virtual Machine
LDAP	Lightweight Directory Access Protocol
RAM	Random Access Memory
GB	Gigabyte
MVC	Model-View-Controller
HTML	Hypertext Markup Language
DOM	Document Object Model
DI	Dependency Injection
URL	Uniform Resource Locator
CI/CD	Continuous Integration/Continuous Deployment
CLI	Command Line Interface
LXC	Linux Containers
ACID	Atomicity, Consistency, Isolation, Durability
JDBC	Java Database Connectivity
Perl DBD	Perl Database Driver
ODBC	Open Database Connectivity
PL/pgSQL	Procedural Language/PostgreSQL Structured Query Language
IANA	Internet Assigned Numbers Authority
AIM	AOL Instant Messenger
MSN Messenger	Messenger: Microsoft Network Messenger
XML	eXtensible Markup Language
SASL	Simple Authentication and Security Layer
MUC	Multi-User Chat
s2s	Server to Server
c2s	Client to Server

ÚVOD

V důsledku současných trendů roste tendence přesouvat většinu práce a procesů online. Aby bylo možné tento přechod úspěšně realizovat, jsou nutné určité změny v infrastruktuře. Pro podniky roste potřeba řešení, která mohou zaměstnancům poskytnout pohodlné a bezpečné nástroje pro co nejrychlejší spolupráci online. Vzhledem k tomu, že zaměstnanci často sdílejí důležité firemní informace, stává se klíčovým požadavkem na tyto nástroje bezpečnost. Často si každá společnost, která si může dovolit mít vlastní infrastrukturu, chce uspořádat ukládání těchto informací na vlastních serverech, což umožňuje omezit alespoň fyzický přístup k datům. Jednou z nejžádanějších je možnost zasílání okamžitých zpráv mezi zaměstnanci. Vzhledem k tomu, že zaměstnanci mohou ke své práci používat různá zařízení, je dalším důležitým požadavkem univerzálnost nástroje a jeho nezávislost na platformě. V této bakalářské práci bude vyvinut messenger, který lze rychle a pohodlně nasadit na firemní servery a který částečně splňuje požadavky firmy na bezpečnost dat. Bude k němu možné přistupovat prostřednictvím prohlížeče, což splňuje požadavky na univerzálnost a platformovou nezávislost.

Teoretická část je věnována popisu základních požadavků na firemní messenger a technologiím, které se používají k jejich splnění. Pro pochopení celého řetězce zpracování, ukládání a prezentace dat je tato část věnována především celkové architektuře aplikace. Rovněž pro pochopení fungování systému jako celku je popsána role a funkce jednotlivých modulů.

Praktická část obsahuje vývoj, konfiguraci a přizpůsobení jednotlivých modulů aplikace. Tato část zahrnuje vytvoření, konfiguraci a popis struktury databází SQL a noSQL, které jsou nezbytné pro správný chod aplikace a perzistenci dat. Spojovacím článkem mezi databázemi a ostatními moduly je reaktivní Java Spring WebFlux server. Klíčovým modulem pro zasílání okamžitých zpráv a oznámení o přítomnosti mezi uživateli je server Ejabberd, jehož instalace a správná konfigurace je rovněž provedena v praktické části. Posledním modulem je frontendová aplikace založená na platformě Angular, která poskytuje uživateli rozhraní pro interakci s aplikací prostřednictvím prohlížeče

1 Ejabberd XMPP Server

Ejabberd je celosvětově považován za jeden z nejpoužívanějších serverů XMPP.

Na základě potřeb podniků je ejabberd vytvořen tak, aby byl odolný proti chybám, dokázal využívat více clusterových strojů a podle potřeby jej lze jednoduše rozšiřovat přidáním dalších serverů nebo virtuálních strojů. [16]

1.1 XMPP

Protokol XMPP (eXtensible Messaging and Presence Protocol) se používá pro instant messaging. Poskytuje také indikaci přítomnosti v síti, skupinové chaty, hlasové a videohovory. XMPP je potomkem open source technologie Jabber. Hlavní výhody protokolu XMPP:

Otevřenost: Protokol XMPP je zdarma a má otevřený zdrojový kód. Existuje mnoho možných implementací (hotové servery, serverové komponenty, klienti a knihovny).

Standardizace: IETF (Internet Engineering Task Force) standardizovala protokol XMPP v roce 2004 a popsala jej v RFC 3920 a RFC 3921. V roce 2011 byla aktualizována v RFC 6120, RFC 6121 a RFC 7622.

Spolehlivost: Protokol XMPP má velkou komunitu vývojářů, kteří jej dodnes vyvíjejí, udržují a aktualizují. Tuto technologii používají tisíce serverů, což zajišťuje správný a spolehlivý chod mnoha aplikací. Zabezpečení: Servery XMPP mohou pracovat v izolovaných prostředích (například v podnikových intranetech) a obsahují silné bezpečnostní prvky včetně protokolů SASL a TLS. Úsilí o end-to-end šifrování nadále zvyšuje bezpečnost.

Rozšiřitelnost: Díky XML lze protokol XMPP rozšířit tak, aby podporoval přizpůsobené funkce a zároveň byla zajištěna interoperabilita. Zatímco běžná rozšíření jsou zdokumentována v řadě XEP, pro specifické potřeby organizace lze vyvinout i soukromá rozšíření.

Všestrannost: Kromě instant messagingu se protokol XMPP používá pro různé aplikace včetně správy sítě, distribuce obsahu, spolupráce, sdílení souborů, hraní her, vzdáleného monitorování systému, cloudových služeb a dalších, a nabízí tak flexibilitu svých aplikací.

Rozmanitost: Nástroje XMPP se používají v různých organizacích, od korporací po open source projekty, k vývoji systémů, které pracují v reálném čase a zajišťují, že všichni uživatelé dostanou zprávy včas.

XMPP zjednodušuje přenos síťového provozu XML, takže lze vyvíjet bezpečné, otevřené a decentralizované systémy pro okamžité zasílání zpráv. Protokol zahrnuje mimo jiné šifrování, silné ověřování a podporu Unicode. [2]

1.1.1 Zobecněná architektura

Ačkoli XMPP není omezen na konkrétní síťovou strukturu, běžně se používá v modelu klient-server. V tomto uspořádání se klient připojuje k serveru pomocí spojení TCP a servery se propojují podobně prostřednictvím TCP.

Server: V protokolu XMPP funguje server jako klíčový element mezi všemi ostatními prvky aplikace. Mezi jeho klíčové povinnosti patří správa toku XML a navazování relací s klienty, jinými servery a dalšími subjekty a správné směrování XML-Stanza do těchto uzlů a z nich. Mnoho serverů XMPP je také schopno zpracovávat a ukládat různá data, například seznamy kontaktů pro zasílání zpráv a sledování přítomnosti uživatelů, přičemž tato data zpracovává v rámci aplikace, bez nutnosti odesílat tato data někam jinam.

Klient: naváže přímé spojení TCP se serverem, aby měl přístup k jeho funkcím, službám a datům a aby mohl prostřednictvím serveru komunikovat s ostatními klienty. Jeden uživatel má možnost připojit se k serveru z více klientů, používat různá zařízení a měnit místa, přičemž každé připojení má v adrese XMPP jedinečný identifikátor zdroje. Organizace IANA doporučuje pro připojení klienta k serveru port 5222.

Brána: funguje jako překladač na straně serveru mezi protokolem XMPP a ostatními protokoly jiných systémů pro zasílání zpráv. Převádí data ve formátu XMPP do jiných formátů. Brány umožňují komunikaci protokolu XMPP s komunikačními systémy, které mají jiný formát zpráv, včetně e-mailu, IRC, SIMPLE, SMS a dalších systémů rychlého zasílání zpráv, jako jsou AIM, ICQ, MSN Messenger a Yahoo!

Síť: V reálném světě servery XMPP často tvoří síť serverů, které spolu komunikují a umožňují uživatelům připojeným k různým serverům vzájemně komunikovat (např. posílat zprávy a vyměňovat si informace o přítomnosti). Síť serverů XMPP je podobná sítím používaným jinými protokoly pro zasílání zpráv, které používají standardy síťového adresování. Síťování mezi servery není povinné, ale pokud je implementováno, je nutné dodržet používání protokolově definovaných XML streamů přes TCP spojení. Podle organizace IANA je doporučený port pro spojení mezi servery 5269. [2] [17]

1.1.2 XML Stream

Rychlá a asynchronní výměna strukturovaných informací mezi prvky aplikace vytvořené pomocí protokolu XMPP se provádí pomocí dvou hlavních technologií: XML Streams a XML Stanza. Tyto koncepty jsou vysvětleny níže:

XML Stream slouží jako kanál pro přenos elementů XML mezi libovolnými dvěma síťovými objekty. Začátek XML streamu je označen počáteční tagem XML <stream>, doplněným potřebnými atributy a deklaracemi jmenného prostoru, a jeho ukončení je označeno uzavíracím tagem XML </stream>. Jakmile je stream inicializován, může iniciační objekt předat více XML elementů buď za účelem definování různých parametrů streamu (např. pro iniciaci TLS nebo SASL), nebo pro odeslání XML Stanza (<message/>, <presence/> nebo <iq/>). Stream začíná u iniciátora (obvykle klienta nebo serveru) a je směrován k příjemci (obvykle serveru). Tento stream zajišťuje jednosměrnou komunikaci od iniciátora k příjemci. Pro uskutečnění obousměrné komunikace musí příjemce vyjednat zpětný stream ("response stream").

Stanza je blok strukturovaných informací předávaných pomocí XML streamu z jednoho objektu do druhého. Stanza je zabalena přímo do kořenového elementu <stream/> a její struktura musí odpovídat standardům obsahu XML definovaným v protokolu XMPP. Začátek a konec stanzy je označen značkami jejího začátku a konce v kořenovém elementu streamu (např. <přítomnost> a </přítomnost>). Stanza může obsahovat interní elementy, atributy a znaková data XML potřebná k předání specifických informací. Elementy odesílané pro vyjednávání TLS nebo SASL nebo pro zpětnou vazbu nejsou stanzami XML.

Například když klient iniciuje spojení se serverem, začíná otevřením streamu XML pomocí tagu <stream> odeslaného serveru, případně včetně textové deklarace definující verzi XML a kódování znaků. Server musí podle svého nastavení odpovědět klientovi vzájemným streamem, kterému v případě potřeby rovněž předchází textová deklarace. Jakmile je vyjednávání SASL dokončeno, může klient začít odesílat XML-Stanza pomocí streamu na libovolný objekt v síti. Pro ukončení spojení odešle klient serveru uzavírací značku </stream>, která vyjadřuje jeho záměr ukončit spojení (obvykle TCP).

Pro ty, kteří znají XML orientovaný na dokumenty, může být mylná představa relace klient-server jako dvou běžících dokumentů XML – jednoho od klienta k serveru a druhého od serveru ke klientovi, přičemž kořenový element <stream/> funguje jako element pro oba "dokumenty".

Tyto "dokumenty" se rozšiřují, když se v obou proudech vyměňují strofy XML. Tato reprezentace je sice pohodlná, ale není zcela správná. Protokol XMPP se zaměřuje na přenos a výměnu streamů XML a stanz, nikoliv dokumentů jako takových. XML Stream v podstatě zahrnuje všechny řetězce XML přenášené během relace a nabízí zjednodušený model tohoto procesu výměny. [2] [17]

1.1.3 BOSH

BOSH (Bidirectional Streams over Synchronous HTTP) je technologie vyvinutá pro obousměrnou komunikaci přes HTTP (Hypertext Transfer Protocol). BOSH se dobře hodí pro aplikace, které vyžadují podporu odesílání a přijímání zpráv vhodným využitím některých vlastností protokolu TCP (Transmission Control Protocol). Díky tomu je BOSH lepší než mnoho podobných metod obousměrné komunikace HTTP (např. Ajax), pokud jde o efektivitu šířky pásma a rychlou dobu odezvy. Tyto výhody jsou dosaženy tím, že odpadá potřeba častých HTTP requestů a nejsou používány segmentované HTTP odpovědi, což je technika známá jako Comet. Původně byl BOSH vyvinut v rámci XMPP pro výměnu zpráv mezi klienty a servery Jabber/XMPP. V podstatě bylo potřeba, aby BOSH zlepšil interakci webových a mobilních klientů se serverem XMPP v sítích s proměnlivou konektivitou. Díky výše popsaným výhodám se však jeho použití rozšířilo i mimo XMPP. [2] [17]

1.1.4 Přítomnost

Přítomnost je funkce protokolu XMPP, která umožňuje uživatelům sdělovat stav dostupnosti svých kontaktů. Je to něco jako digitální indikátor, který vás informuje o aktuálním stavu lidí v seznamu kontaktů. O tom, jestli je někdo právě online, mimo domov, není schopen odpovědět nebo je offline. Tato funkce také umožňuje uživatelům přidat k oznámení o stavu další kontext, například přidat zprávu typu "na schůzce" nebo "na dovolené". Informace o přítomnosti umožňují uživatelům činit informovaná rozhodnutí o tom, kdy a jak zahájit konverzaci, tedy zda uživateli napsat zprávu v chatu, nebo zda je vhodnější jej oslovit jiným způsobem. Tato funkce je navržena tak, aby poskytovala vývojářům prostor pro další rozšíření, takže použití této funkce není omezeno na prosté zasílání zpráv. Lze ji použít pro oznamování přítomnosti v různých aplikacích. [2] [17]

1.1.5 Multi-User-Chat

Chat s více uživateli (MUC) je doplňková funkce protokolu XMPP, která umožňuje vytvářet skupiny, v nichž může současně komunikovat více uživatelů.

Aby mohli uživatelé tuto funkci využít, mohou vytvářet virtuální "místnosti" nebo "kanály" a posílat zprávy v těchto víceuživatelských skupinách několika adresátům současně. MUC má množinu všech potřebných funkcí víceuživatelského chatu. Například nastavení tématu diskuse nebo zasílání pozvánek a také nástroje správce pro správu místnosti. Tyto nástroje poskytují možnosti odstraňování nebo blokování uživatelů, přidělování moderátorů a správců a nastavení podmínek pro připojení k MUC, například členství nebo heslo. Vzhledem k tomu, že MUC je součástí protokolu XMPP, mohou členové chatu posílat nejen jednoduché textové zprávy, ale také různá data založená na XML (například obrázky nebo dokumenty). [2] [17]

1.2 Ejabberd

Původně vyvinutý v době, kdy desktopoví klienti používali především metodu dotazování HTTP známou jako BOSH, se ejabberd vyvinul tak, aby zahrnoval podporu moderních technologií, jako jsou WebSockets, vylepšený BOSH a robustní mobilní stack. Aplikace ejabberd vznikla v době, kdy se XMPP ještě označoval jako "Jabber", a neustále se vyvíjela tak, aby vyhovovala různým RFC XMPP. Podporuje širokou škálu rozšíření vytvořených XSF.

Tuto přizpůsobivost a růst usnadňuje modulární architektura, která je založena na jádru směrovače, rozšířeném o výkonný a stále se rozšiřující mechanismus pluginů.

Ejabberd nabízí různé verze pro použití, z nichž nejpopulárnější je ejabberd Community Edition. Tato verze je open-source, vysoce škálovatelný a flexibilní standard. [16]

Integrace ejabberdu s aplikacemi je jednoduchá a může být realizována pomocí:

- REST API a nástroje příkazového řádku ejabberdctl .
- Mobilních knihoven pro iOS, včetně XMPPFramework a Jayme REST API .
- Mobilních knihoven pro Android, například Smack a Retrofit.
- Webových knihoven, které podporují WebSocket a poskytují nouzový přístup k BOSH, například Strophe.

1.2.1 Architektura Ejabberd

Ejabberd zajišťuje konfigurovatelnost, škálovatelnost a odolnost proti chybám u základní funkce protokolu XMPP - směrování zpráv. Jeho konstrukce zahrnuje řadu modulárních komponent, které aktivují různé funkce, včetně:

- Soukromé zasílání zpráv.
- Ukládání a předávání pro offline zprávy.
- Seznamy a informace o přítomnosti.
- Chat pro více uživatelů (MUC) pro skupinové diskuse.
- Archivace zpráv prostřednictvím správy archivu zpráv (MAM).
- Rozšíření pro přítomnost uživatelů, například protokol PEP (Personal Event Protocol) a indikátory psaní na klávesnici.
- Kontrola soukromí prostřednictvím seznamů soukromí a jednoduchých blokovacích mechanismů.
- Profily uživatelů spravované pomocí karet vCardsKomplexní podpora webu prostřednictvím BOSH a websocketů.
- Stream management pro zvýšení spolehlivosti zpráv na mobilních zařízeních, označovaný jako XEP-0198.
- Potvrzení o doručení zprávy, známé jako XEP-184.
- Sledování poslední aktivity uživatele.
- Metriky a rozsáhlé nástroje pro správu příkazového řádku.
- Mnoho dalších funkcí.

Podrobný seznam všech podporovaných protokolů a rozšíření naleznete na stránce "Protokoly podporované systémem ejabberd".

Tato modulární struktura zajišťuje snadné přizpůsobení a jednoduchý přístup k potřebným funkcím.

Aplikace ejabberd podporuje ověřování uživatelů prostřednictvím externích nebo interních databází (Mnesia, SQL), LDAP nebo externích skriptů a podle potřeby také usnadňuje připojení anonymních uživatelů.

Pro trvalé ukládání dat ejabberd primárně používá Mnesia, distribuovanou interní databázi Erlang, s možností integrace jiných SQL databází, například MySQL nebo PostgreSQL.

Díky přizpůsobitelnému rozhraní API lze ejabberd přizpůsobit tak, aby bezproblémově spolupracoval s databází podle volby uživatele. [16]

1.2.2 Nasazení a správa

Ejabberd lze nastavit pro různé případy použití, přizpůsobené potřebám koncových uživatelů, vývojářů nebo zákazníků. Jeho výchozí konfigurace obsahuje jediný uzel ejabberd využívající Mnesia, který nevyžaduje žádné další nastavení. Toto základní uspořádání je vhodné pro rychlé nasazení a připojení klientů XMPP a obvykle stačí pro většinu malých až středně velkých nasazení.

Pro zvýšení škálovatelnosti lze ejabberd integrovat s externí databází pro správu trvalých dat. Vzhledem k tomu, že Mnesia ukládá část svých dat do mezipaměti ejabberdu (konkrétně v rámci uzlu Erlang VM), zvyšuje tato konfigurace škálovatelnost a obecně zjednodušuje integraci s běžnými databázemi.

Pro operace většího rozsahu lze ejabberd provozovat jako cluster aktivních uzlů. Tento clusterový režim poskytuje nejen odolnost proti chybám, ale také zvyšuje kapacitu nasazení ejabberdu.

Provoz do uzlů clusteru lze řídit pomocí jednoho z několika řešení pro vyrovnávání zátěže:

- Tradiční vyvažování zátěže TCP/IP (zvažte náklady, protože typická spojení XMPP jsou trvalá).
- Vyrovnávání zátěže DNS.
- Vlastní metoda, která zahrnuje spolupráci klientů.

Na hardwaru, jako je stroj s 16 GB RAM a alespoň 4 jádry, může jeden uzel ejabberdu obvykle podporovat 200-300 tisíc online uživatelů. Takové nastavení je proveditelné pro systémy do 10 uzlů.

Aplikace ejabberd také podporuje propojování různých clusterů do větších systémů, což je funkce vlastní protokolu XMPP známá jako server-to-server (nebo s2s). Tato schopnost je klíčová pro geograficky lokalizované služby, které zpracovávají značný globální provoz.

Nástroj příkazového řádku ejabberdctl je nezbytný pro správu uživatelů, seznamů, zpráv a obecných nastavení. Umožňuje také shromažďovat systémové metriky pro sledování, pochopení a případné předcházení problémům.

Významnou výhodou ejabberd je dostupnost příkazového řádku pro přímé provádění příkazů Erlang. Tato funkce je velmi užitečná pro řešení složitých problémů, aktualizaci a načítání kódu bez nutnosti zastavení služby. [16]

2 Reaktivní backend

Pro pochopení reaktivního programování je zásadní pochopit, co znamená "reaktivní". Reaktivní znamená reakci na něco – v tomto kontextu na události. Reaktivní programování je tedy programování, které reaguje na události, což z něj činí přístup orientovaný na události.

Reaktivní programování se soustředí na vytváření softwaru, který pracuje asynchronně a bez blokování a využívá strukturu řízenou událostmi. [2] [4]

Asynchronní a neblokující programování

Na rozdíl od tradičních aplikací, kde jsou volání blokována a mohou zastavit vlákno nebo celou aplikaci během provádění operací, jako jsou dotazy do databáze nebo API, asynchronní provádění probíhá bez čekání na dokončení těchto operací. Neblokující přístup používá futures a zpětná volání k obsluze operací, které by jinak zastavily vykonávání. [1]

Tok dat řízený událostmi/zprávami

V reaktivním programování jsou data reprezentována jako spojitý stream. Při reaktivním zpracování program čeká na událost a poté vyvolá příslušnou reakci. Podobným způsobem fungují například streamy v jazyce Java, které byly zavedeny ve verzi 1.8. Při tradičním zpracování dat se data načítají najednou, ale ve streamech řízených událostmi se data načítají postupně, po částech, tak jak se události posílají příjemci.

Kód ve funkčním stylu

Reaktivní programování často využívá lambda výrazy, které jsou základním kamenem funkcionálního programování v jazyce Java. Tyto výrazy představují funkční styl kódování.

Zpětný tlak

Pokud během konzumace dat v reaktivních proudech konzument (aplikace) nestíhá rychlost dat producenta, může požádat producenta o zpomalení. Tento mechanismus řízení toku dat je znám jako zpětný tlak. [4]

2.1 Reactive Manifesto

V roce 2013 vydal Jonas Boner se skupinou vývojářů dokument Reactive Manifesto, který popisuje klíčové principy reaktivního programování.

Nutnost vzniku takového dokumentu se objevila kvůli změně požadavků na moderní aplikace v důsledku zvýšené zátěže. Zatímco dříve velké aplikace běžely většinou na desítkách serverů, měly dobu odezvy do několika sekund, hodinové prostoje v provozu, v dnešní době aplikace musí komunikovat s výrazně větším počtem zařízení od mobilních až po cloudové clustery s tisíci vícejádrovými procesory, což výrazně zvyšuje zátěž serveru a zároveň se přidává nutnost dodržet dobu odezvy v rozumných mezích. V dnešní době se očekávají milisekundové odezvy a nepřetržitá dostupnost, přičemž objemy dat mohou dosahovat petabajtů. Architektonické standardy minulosti neodpovídají dnešním požadavkům.

Dnes existuje potřeba nového obecného přístupu k architektuře systémů, který zdůrazňuje důležitost reaktivních, odolných, flexibilních a zprávami řízených systémů, označovaných společně jako reaktivní systémy. Nová architektura na bázi reaktivních technologií zajišťuje větší flexibilitu, škálovatelnost a neblokující interoperabilitu, což usnadňuje vývoj a přizpůsobení. Prokazují výraznou odolnost vůči selháním a šetrně řeší případná selhání. Takové systémy podporují vysokou odezvu a poskytují uživateli účinnou zpětnou vazbu.

Mezi vlastnosti reaktivních systémů patří:

Reaktivnost: Jejich cílem jsou pohotové reakce, které zajišťují použitelnost a funkčnost a umožňují rychlé odhalení a řešení problémů. Udržováním rychlé a konzistentní doby odezvy nabízejí spolehlivou kvalitu služeb, což zjednodušuje správu chyb, zvyšuje důvěru uživatelů a podporuje jejich zapojení.

Odolnost: Zůstávají pohotové i přes poruchy, což platí jak pro kritické, tak pro standardní systémy. Odolnosti je dosaženo pomocí strategií, jako je replikace, omezení, izolace a delegování, které zajišťují, že selhání jednotlivých komponent neohrozí celkovou integritu systému. Vysoká dostupnost je udržována prostřednictvím replikace podle potřeby, aniž by byl klient nadměrně zatěžován správou výpadků.

Pružnost: Přizpůsobují odezvu podle pracovního zatížení, jsou schopny škálovat zdroje nahoru nebo dolů na základě vstupních rychlostí. Tato flexibilita je navržena bez centrálních úzkých míst a umožňuje sharding komponent nebo replikaci a distribuci vstupů. Pružnost je podporována prediktivním i reaktivním škálováním, optimalizovaným pro nákladovou efektivitu na standardním hardwaru a softwaru.

Řízení zprávami: Využívají asynchronní předávání zpráv k zajištění oddělení komponent, izolace a transparentnosti umístění, což napomáhá delegování a správě výpadků. Explicitní předávání zpráv usnadňuje správu zátěže, pružnost a řízení toku a podle potřeby uplatňuje zpětný tlak pro efektivní provoz. Tento přístup umožňuje konzistentní techniky správy poruch v různých prostředích, minimalizuje spotřebu aktivních prostředků a tím snižuje režii.

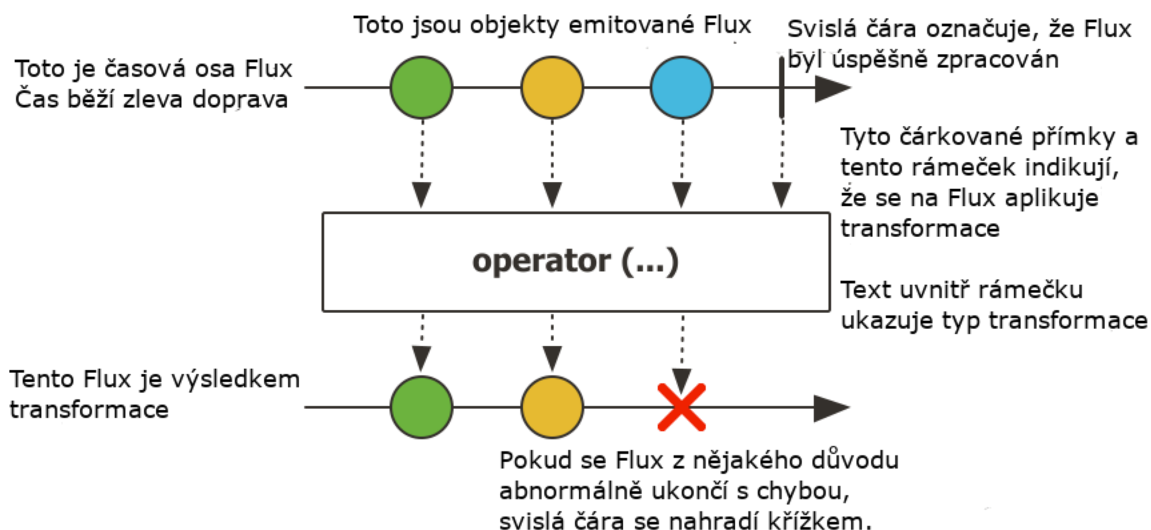
Reaktivní systémy jsou strukturovány tak, aby tyto vlastnosti byly vlastní všem úrovním, což umožňuje škálovatelnost a kompozici. Na tyto principy spoléhají největší a nejsložitější globální systémy, které denně obsluhují miliardy lidí. Přijetí těchto principů návrhu od samého počátku, spíše než jejich znovuobjevování, představuje strategický posun ve vývoji systémů. [5]

2.2 Project Reactor

Project Reactor je uznávanou reaktivní knihovnou pro Javu. Jako reaktivní knihovna obsahuje zcela neblokující proud, který podporuje zpětný tlak. Bez problémů spolupracuje s funkčními rozhraními API Javy 8. Srdcem Project Reactor je "reactor-core", který zahrnuje několik klíčových myšlenek, které stojí za to prozkoumat.

2.2.1 Flux

Flux je pojem, který označuje asynchronní řadu, která může obsahovat od nuly do mnoha položek. Představte si jej jako potrubí schopné zpracovávat posloupnost položek, od žádné až po potenciálně mnoho, umožňující různé transformace této posloupnosti na jiný typ nebo formu posloupnosti.



Obrázek 1: Flux (zdroj [12])

Obrázek ukazuje, jak jsou položky (označované jako N položek) přiváděny do tohoto potrubí k odběratelům. Tyto položky jsou součástí proudu Flux a ilustrace končí řádkem označujícím konec této sekvence Flux. V procesu je funkce, označená jako pole operátora, která je zodpovědná za určitou modifikaci toku.

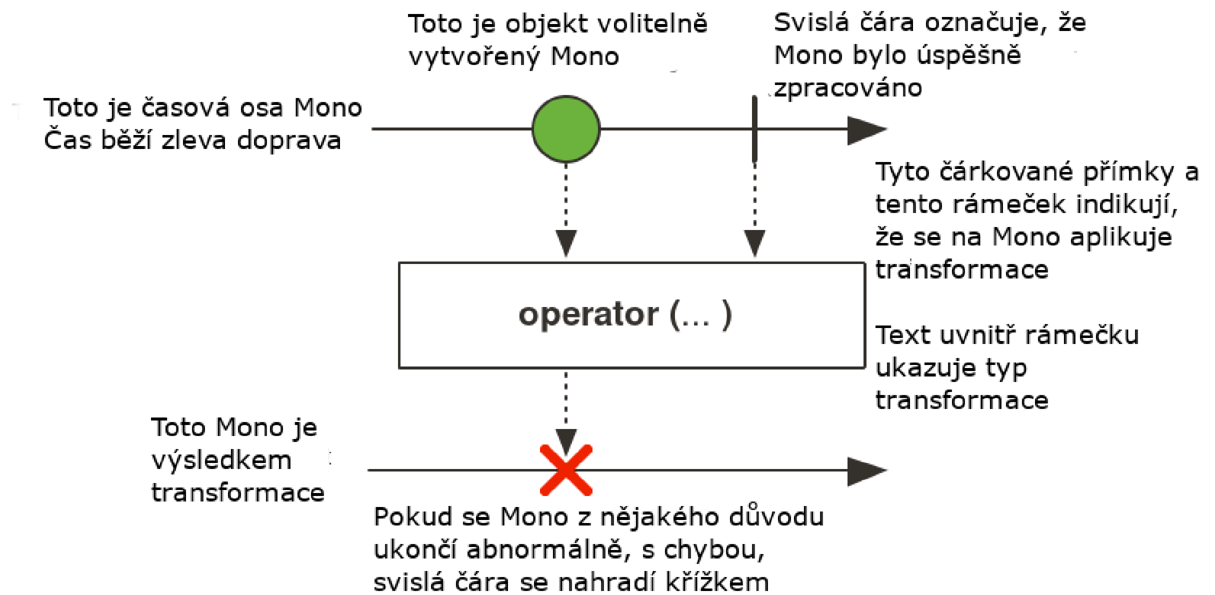
Tato modifikace se provádí položku po položce prostřednictvím metody nazvané onNext(), což je klíčová funkce, o které jsme se již zmínili. Transformované položky jsou pak emitovány jako nový proud Flux, jak ukazuje řada čísel v dolní části.

Chybu během procesu transformace symbolizuje červená ikona X, která upozorňuje na selhání transformace konkrétní položky. Tento scénář spouští metodu onError(), další funkci, kterou jsme již dříve probrali.

Za předpokladu, že proces nenarazí na žádnou chybu, je transformace aplikována na všechny položky, což je završeno metodou `onComplete()`. Tento závěrečný krok znamená dokončení transformace pro celou sekvenci. [4] [12]

2.2.2 Mono

Mono je koncept, který se zabývá proudem, který vysílá jedinou položku nebo žádnou. Umožňuje různé transformace tohoto prvku, včetně jeho úplné změny.



Obrázek 2: Mono (zdroj [11])

Diagram ukazuje, že v proudu Mono je účastníkovi odesílána pouze jedna položka. Sekvence je zakončena zápisem označujícím konec proudu. Specifická část procesu, známá jako operátorské pole, má za úkol změnit jedinou položku v proudu Mono.

Tato změna se provádí na položce proudu Mono prostřednictvím metody onNext(), o které jsme hovořili dříve.

Červená ikona X slouží k označení chyby, která se vyskytla během transformace položky, a aktivuje metodu `onError()`, což je další téma, kterému jsme se věnovali dříve.

Pokud proces proběhne bez chyb, je položka úspěšně transformována, což je zakončeno použitím metody `onComplete()`. Tato metoda znamená úspěšné dokončení procesu transformace položky proudu Mono. [4] [11]

2.3 Spring WebFlux

Spring Framework původně přišel s webovým frameworkem Spring Web MVC, který byl navržen speciálně pro použití s rozhraním Servlet API a kontejnery Servlet. Později, s uvedením verze 5.0, společnost Spring představila Spring WebFlux, nový webový framework postavený na reaktivním zásobníku. Spring WebFlux se vyznačuje tím, že je zcela neblokující, podporuje zpětný tlak z reaktivních proudů a je kompatibilní se serverovými technologiemi, jako jsou Netty, Undertow a tradiční kontejnery Servlet. [3]

Tyto dva rámce, Spring Web MVC a Spring WebFlux, jsou pojmenovány podle příslušných modulů v rámci frameworku Spring (spring-webmvc a spring-webflux) a jsou k dispozici k současnému použití. Jsou modulární, což znamená, že vývojáři mohou pro svou aplikaci flexibilně zvolit jeden z těchto frameworků nebo v určitých scénářích využít oba. Příkladem takového hybridního přístupu je použití řadičů Spring MVC společně s reaktivním WebClientem. [4] [6]

3 Angular

Angular slouží jako platforma i jako framework pro tvorbu jednostránkových aplikací využívajících HTML a TypeScript. Je vyvinutý v jazyce TypeScript a nabízí sadu knihoven TypeScript, které obsahují základní a volitelné funkce a které mohou vývojáři začlenit do svých projektů.

Jádrem architektury Angularu jsou základní koncepty, jejichž základními prvky jsou komponenty. Tyto komponenty jsou zodpovědné za vytváření view – kolekcí prvků obrazovky, které Angular dynamicky vybírá a aktualizuje na základě logiky a dat aplikace.

Komponenty využívají služby k provádění úloh nesouvisejících s přímým zobrazením, jako je například načítání dat. Tyto služby lze integrovat do komponent jako závislosti, což zvyšuje modularitu, znovupoužitelnost a efektivitu kódu.

Komponenty i služby jsou definovány jako třídy ozdobené dekorátory, které poskytují systému Angular potřebná metadata o jejich využití.

U komponent tato metadata propojují třídu se šablonou, která nastiňuje zobrazení. Šablona slučuje standardní HTML se směrnicemi a syntaxí vazeb jazyka Angular, což umožňuje jazyku Angular upravit HTML před jeho zobrazením.

Metadata tříd služeb informují modul Angular o tom, jak tyto služby využít pro komponenty prostřednictvím vstřikování závislostí (DI), což usnadňuje bezproblémovou integraci.

Aplikace Angular se obvykle skládá z několika view uspořádaných do hierarchické struktury. Služba Angular Router pomáhá definovat navigační cesty mezi těmito view a nabízí pokročilé možnosti navigace v prohlížeči. [7]

3.1 Komponenty

V každé aplikaci Angular je vždy minimálně jedna komponenta, známá jako kořenová komponenta, která slouží jako most mezi hierarchií komponent a objektovým modelem dokumentu (DOM) stránky. Tato komponenta je definována třídou, v níž jsou uložena data a logika aplikace, a je propojena se šablonou HTML, která nastiňuje zobrazení, jež se má zobrazit v určeném prostředí.

Dekorátor `@Component()` se používá k označení třídy, které předchází, jako komponenty a dodává šablonu a další metadata vztahující se ke komponentě.

Dekorátory jsou v podstatě funkce určené ke změně tříd jazyka JavaScript. Angular zavádí několik dekorátorů určených k přidávání určitých typů metadat ke třídám. Tento přídavek informuje framework o rolích těchto tříd a určuje jejich funkčnost. [7] [8]

3.1.1 Šablony, směrnice a datové vazby

Šablona Angular integruje jazyk HTML s anotacemi specifickými pro Angular, které umožňují změny prvků HTML před jejich zobrazením. Pomocí direktiv šablony zavádí programovací logiku, zatímco vazební anotace propojují data aplikace s DOM. Datové vazby se dělí na dva typy:

Vazba na události: Umožňuje aplikaci reagovat na interakce uživatele tím, že odpovídajícím způsobem aktualizuje svá data.

Vazba na vlastnosti: Usnadňuje vkládání vypočtených hodnot z dat aplikace do HTML.

Angular zpracovává směrnice a aplikuje vazební anotace v rámci šablony, aby změnil prvky HTML a DOM na základě dat a logiky vaší aplikace před zobrazením zobrazení. Je také vybaven obousměrnou vazbou dat, která zajišťuje, že veškeré změny v DOM, jako jsou uživatelské výběry, jsou současně aktualizovány v datech vaší aplikace.

Pro zvýšení uživatelského komfortu mohou šablony využívat roury, které transformují hodnoty pro prezentaci. Trubice mohou například formátovat data a měnové údaje tak, aby odpovídaly regionálnímu nastavení uživatele. Angular je vybaven řadou vestavěných rour pro běžné transformace dat a umožňuje také vytváření vlastních rour. [8]

3.2 Service a Dependency Injection

Pokud máte data nebo logiku, které se netýkají konkrétního zobrazení, ale musí být přístupné ve více komponentách, měli byste vytvořit třídu Service. Dekorátor @Injectable(), umístěný přímo před definici třídy služby, dodává potřebná metadata, která umožňují do této třídy vstřikovat další poskytovatele jako závislosti.

Díky injekci závislostí (DI) zůstává architektura tříd vašich komponent přehledná a efektivní. Místo přímého provádění činností, jako je načítání dat ze serveru, ověřování uživatelských vstupů nebo přihlašování do konzoly, jsou tyto povinnosti přiřazeny službám. [8]

3.2.1 Směrování

Angular Router je služba, která umožňuje vytvářet navigační cesty přes různé stavy a hierarchie zobrazení v rámci aplikace, přičemž vychází ze známých konvencí navigace v prohlížeči:

Zadání adresy URL do adresního řádku vede prohlížeč na konkrétní stránku.

Kliknutím na odkaz přejdete na jinou stránku v prohlížeči.

Pomocí tlačítek zpět a vpřed prohlížeče lze procházet historii stránek prohlížeče.

Namísto navigace na různé stránky směrovač přiřazuje cesty podobné URL k součástem. Pokud by akce, jako je kliknutí na odkaz, obvykle vedla k načtení nové komponenty, směrovač tento proces převezme a rozhodne, zda zobrazí nebo skryje komponentu a její potomky.

Pokud směrovač zjistí, že požadovaná komponenta pro aktuální stav aplikace ještě není načtena, má možnost líně načíst potřebnou komponentu a její závislosti.

Směrovač zpracovává adresy URL odkazů na základě navigačních pravidel a stavu dat aplikace. Umožňuje navigaci do nových zobrazení vyvolaných uživatelskými akcemi, jako je kliknutí na tlačítko nebo výběr z rozbalovací nabídky, nebo v reakci na jiné typy podnětů. Směrovač také uchovává záznam o této navigaci v historii prohlížeče a zajišťuje správnou funkci tlačítek zpět a vpřed.

Pravidla navigace se nastavují propojením navigačních cest s komponentami s využitím syntaxe podobné URL, která zahrnuje programová data podobně, jako syntaxe šablon spojuje view s programovými daty. To umožňuje použití programové logiky k řízení viditelnosti zobrazení v souladu s akcemi uživatele a specifickými pokyny pro přístup. [8]

3.3 Knihovna RxJS

Reaktivní programování je styl programování zaměřený na řízení asynchronních datových toků a automatické šíření změn. RxJS, zkratka pro Reactive Extensions for JavaScript, je knihovna určená k usnadnění reaktivního programování pomocí pozorovatelných objektů (Observables). To usnadňuje psaní asynchronního kódu nebo kódu založeného na zpětných voláních.

RxJS zavádí typ Observable pro správu asynchronních datových toků do doby, než bude tento typ nativně podporován jazykem JavaScript a webovými prohlížeči. Vedle toho knihovna nabízí řadu užitečných funkcí pro:

- Transformaci asynchronního operačního kódu na pozorovatelné objekty.
- Iteraci nad hodnotami proudu.
- Transformaci hodnot proudu na různé typy.
- Filtrování proudů pro výběr konkrétních hodnot.
- Kombinování více proudů do jednotného datového toku. [8]

4 Docker

Docker je bezplatná platforma s otevřeným zdrojovým kódem, která pomáhá vývojářům vytvářet, nasazovat, spouštět, aktualizovat a spravovat kontejnery. Tyto kontejnery jsou standardizované spustitelné jednotky, které obsahují zdrojový kód aplikace spolu se všemi knihovnami operačního systému a závislostmi potřebnými ke spuštění kódu v libovolném prostředí.

Kontejnery jsou klíčem k zefektivnění vývoje a distribuce distribuovaných aplikací a získávají na popularitě s tím, jak společnosti přecházejí na vývoj v cloudu a hybridní multicloudová nastavení. Vývojáři mohou vytvářet kontejnery pomocí nativních funkcí v Linuxu a dalších operačních systémech, Docker však tento proces zjednodušuje a zabezpečuje, takže je rychlejší a efektivnější. V době zpracování této zprávy využívalo platformu Docker více než 13 milionů vývojářů.

Termín „Docker“ označuje také společnost Docker, Inc., která prodává komerční verzi nástroje Docker. Tato společnost je součástí širšího open source projektu Docker, do kterého přispívá společnost Docker, Inc. A řada dalších organizací a jednotlivců. [10]

4.1 Výhody kontejnerizace

Kontejnery umožňují funkce izolace procesů a virtualizace vestavěné v jádře Linuxu.

Tyto funkce, jako jsou řídicí skupiny (Cgroups) pro rozdělování prostředků mezi procesy a jmenné prostory pro omezení přístupu procesu k určitým systémovým prostředkům nebo oblastem, jsou pro jejich funkčnost nezbytné.

Tato technologie umožňuje různým aplikačním komponentám využívat prostředky jedné instance hostitelského operačního systému. Podobně jako hypervizor umožňuje více virtuálním počítačům (VM) využívat procesor, paměť a další prostředky jednoho hardwarového serveru, kontejnery efektivně sdílejí prostředky.

Technologie kontejnerů proto poskytuje všechny výhody virtuálních počítačů, včetně izolace aplikací, nákladově efektivní škálovatelnosti a snadné disponibility. Kromě toho nabízí několik dalších důležitých výhod:

- **Snížení režie:** Kontejnery se od virtuálních strojů liší tím, že nevyžadují plnou instanci operačního systému a hypervizor. Místo toho obsahují pouze základní procesy

operačního systému a závislosti potřebné ke spuštění kódu. Kontejnery jsou obecně menší, často se měří v megabajtech na rozdíl od gigabajtů, které vyžadují některé virtuální počítače. Tato menší velikost vede k efektivnějšímu využití hardwaru a rychlejšímu spuštění.

- **Zvýšená efektivita pro vývojáře:** Aplikace umístěné v kontejnerech jsou navrženy tak, aby fungovaly konzistentně v jakémkoli prostředí. Kontejnery se rychleji a jednodušeji nastavují, nasazují a restartují než virtuální počítače, a proto jsou vhodné pro kontinuální integraci a kontinuální dodávání (CI/CD). Díky tomu jsou obzvláště výhodné pro vývojové týmy, které používají agilní metodiky a metodiku DevOps.
- **Lepší využití prostředků**: Kontejnery umožňují vývojářům provozovat více instancí aplikace na stejném hardwaru, mnohem více než je možné u virtuálních strojů. [10]

4.2 Základní nástroje Docker

Docker obsahuje řadu klíčových nástrojů pro vytváření, konfiguraci a správu kontejnerů. Nejčastěji používané z těchto nástrojů budou popsány níže.

4.2.1 Dockerfile

Každý kontejner Docker začíná základním textovým souborem, který popisuje kroky pro vytvoření obrazu kontejneru Docker. Tento soubor, známý jako Dockerfile, zjednodušuje vytváření obrazů Docker. V podstatě se skládá z řady příkazů command-line interface (CLI), které Docker Engine provede při sestavování obrazu. Řada příkazů dostupných v nástroji Docker je rozsáhlá a zároveň standardizovaná, což zajišťuje konzistentní fungování funkcí nástroje Docker bez ohledu na obsah, infrastrukturu nebo proměnné prostředí. [10]

4.2.2 Obrazy Docker

Obrazy Docker obsahují spustitelný zdrojový kód aplikace spolu se všemi potřebnými nástroji, knihovnami a závislostmi, které jsou nutné pro provoz aplikace v kontejneru.

Po spuštění se obraz Docker transformuje do jedné nebo více instancí kontejneru.

I když je možné vytvořit obraz Docker od základu, většina vývojářů je raději stahuje z široce používaných úložišť. Z jednoho základního obrazu lze odvodit několik obrazů Docker, které sdílejí základní prvky svého zásobníku.

Obrazy Docker jsou strukturovány do vrstev, kde každá vrstva představuje určitou verzi obrazu. Při změnách obrazu se přidá nová vrchní vrstva, která nahradí předchozí vrchní vrstvu jako nejnovější verzi obrazu. Dřívější vrstvy jsou zachovány pro případné vrácení nebo budoucí použití v různých projektech.

Při každém použití obrazu Docker k vytvoření kontejneru se vytvoří další vrstva známá jako kontejnerová vrstva. Změny provedené během běhu kontejneru, například přidání nebo odebrání souborů, jsou zaznamenány v této kontejnerové vrstvě a zůstávají zachovány pouze po dobu, kdy je kontejner aktivní.

Tento vrstvený přístup k vytváření obrazů zvyšuje efektivitu, protože umožňuje, aby několik aktivních instancí kontejneru fungovalo z jednoho základního obrazu a využívalo sdílený zásobník. [10]

4.2.3 Docker Hub

Docker Hub, který je považován za *"největší světovou knihovnu a komunitu pro obrazy kontejnerů*", slouží jako veřejné úložiště obrazů Docker.

Obsahuje více než 100 000 obrazů kontejnerů od komerčních dodavatelů, open-source projektů a jednotlivých vývojářů. Kolekce obsahuje obrazy vytvořené společností Docker, Inc., certifikované obrazy z důvěryhodného registru Docker a tisíce dalších možností.

Všichni uživatelé služby Docker Hub mohou své obrazy sdílet podle vlastního uvážení. Mají také přístup k předdefinovaným základním obrazům ze souborového systému Docker, které slouží jako užitečné výchozí body pro jakoukoli kontejnerizaci.

Kromě služby Docker Hub mají důležitou roli i další úložiště obrazů, například GitHub. GitHub je známý svou službou hostování repozitářů, která je klíčová pro nástroje pro vývoj aplikací a podporu platformy pro spolupráci a komunikaci. Uživatelé služby Docker Hub mohou vytvářet úložiště neboli "repozitáře", které mohou obsahovat několik obrazů. Tyto repozitáře lze nastavit jako veřejné nebo soukromé a propojit je s účty GitHub nebo BitBucket. [10]

4.3 Výhody Dockeru

Docker se dnes stal takovým synonymem pro kontejnerovou technologii, že se pojmy *„Docker“* a *„kontejnery“* často používají zaměnitelně. Technologie související s kontejnery

však existovaly již léta, a dokonce i desetiletí před veřejným vydáním nástroje Docker v roce 2013.

Významné je, že v roce 2008 byly do linuxového jádra integrovány LinuX kontejnery (LXC), které umožnily kompletní virtualizační schopnosti pro jednu instanci Linuxu. Zatímco LXC se používá i nadále, v současné době existují další modernější technologie, které rovněž využívají jádro Linuxu k podobným účelům. Například Ubuntu, současný open-source operační systém Linux, tyto virtualizační funkce podporuje.

Docker poskytuje vývojářům snadno použitelné rozhraní k těmto neodmyslitelným kontejnerizačním funkcím a umožňuje jim jednoduše provádět příkazy a automatizovat procesy prostřednictvím rozhraní pro programování aplikací (API), které šetří práci. Ve srovnání s LXC Docker nabízí:

- **Zmenšená velikost a detailnější aktualizace:** Docker umožňuje seskupení několika procesů do jednoho kontejneru. Tato schopnost umožňuje vytvářet aplikace, které zůstávají funkční i v době, kdy je některá komponenta dočasně odebrána kvůli aktualizacím nebo opravám.
- **Vylepšená a snadná mobilita kontejnerů:** Na rozdíl od kontejnerů LXC, které často závisí na konfiguraci konkrétního počítače, kontejnery Docker jsou navrženy tak, aby mohly běžet na různých platformách, včetně desktopů, datových center a cloudů, bez nutnosti jakýchkoli úprav.
- **Správa verzí pro kontejnery:** Docker má možnost sledovat různé verze obrazu kontejneru, vrátit se k dřívějším verzím a zjistit, kdo verzi vytvořil a jakým způsobem. Navíc dokáže nahrát pouze změny mezi stávající a novou verzí.
- **Automatizované vytváření kontejnerů:** Docker je schopen automaticky vytvořit kontejner ze zdrojového kódu aplikace.
- **Opakované použití kontejnerů:** Existující kontejnery mohou být použity jako šablony pro vytvoření nových kontejnerů.
- **Společná úložiště kontejnerů:** Vývojáři mají přístup k tisícům kontejnerů připravených k použití.

V současné době je Docker kompatibilní také se systémy Microsoft Windows a Apple MacOS. Hlavní poskytovatelé cloudových služeb poskytují služby, které pomáhají vývojářům při vytváření, nasazování a správě aplikací kontejnerizovaných pomocí nástroje Docker. [10]

5 PostgreSQL

PostgreSQL je robustní objektově-relační databázový systém s otevřeným zdrojovým kódem, který rozšiřuje jazyk SQL o řadu funkcí určených k bezpečnému zpracování a škálování složitých datových úloh. Jeho vývoj byl zahájen v roce 1986 jako součást projektu POSTGRES na Kalifornské univerzitě v Berkeley a na jeho základní platformě probíhá aktivní vývoj již více než 35 let.

Databáze PostgreSQL je známá svou pevnou architekturou, spolehlivostí, integritou dat, rozsáhlou sadou funkcí, rozšiřitelností a angažovaností komunity open-source, která si vybudovala silnou reputaci. Tento databázový systém je kompatibilní se všemi hlavními operačními systémy, od roku 2001 vyhovuje standardu ACID a obsahuje výkonná rozšíření, jako je například známý rozšiřující modul pro geoprostorové databáze PostGIS. [15]

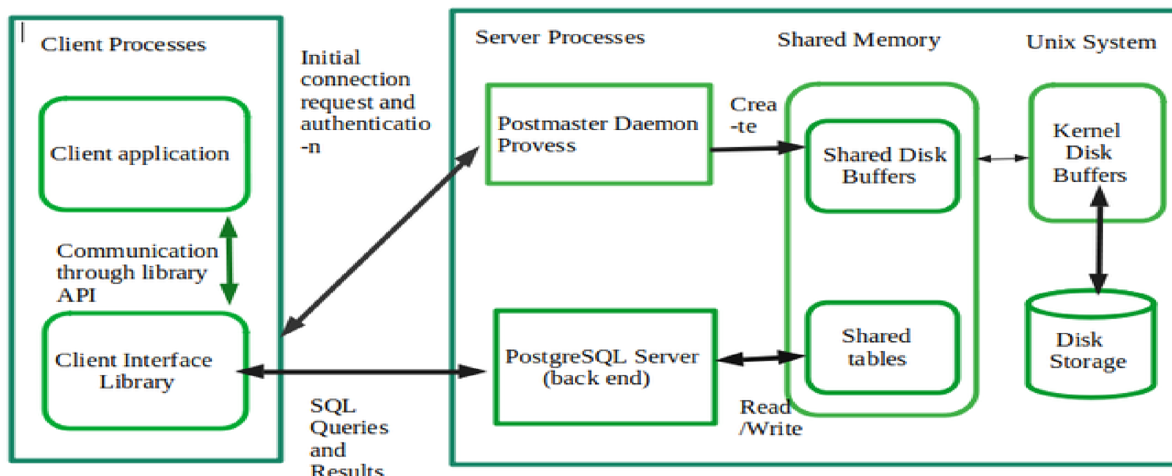
5.1 Architektura

PostgreSQL funguje v rámci klient-server architektury. Služba PostgreSQL se v podstatě skládá ze dvou hlavních komponent:

- Proces na straně serveru: Tento proces, známý jako aplikace Postgres, se stará o správu připojení, operací a statických i dynamických zdrojů.
- Proces na straně klienta (Front-end aplikace): Tyto aplikace jsou určeny pro interakci uživatele s databází. Obvykle mají jednoduché uživatelské rozhraní a usnadňují komunikaci mezi uživatelem a databází, obvykle prostřednictvím rozhraní API. [19] [21]

5.1.1 Proces na straně klienta

Když uživatel provádí dotazy na PostgreSQL, klientská aplikace může navázat spojení se serverem PostgreSQL (označovaným jako proces Postmaster Daemon) a posílat dotazy pomocí některého z podporovaných klientských aplikačních rozhraní PostgreSQL, jako jsou JDBC, Perl DBD, ODBC a další, která poskytují knihovny pro klientskou stranu. [19] V klientském procesu je komunikace mezi klientskou aplikací a klientskou aplikační knihovnou realizována pomocí API rozhraní knihovny, jak je zobrazeno na obrázku níže:



Obrázek 3: Architektura PostgreSQL (zdroj: [19])

5.1.2 Proces Postmaster Daemon

Architekturní framework PostgreSQL je navržen na základě modelu Process-Per-Transaction (model klient/server). Provoz instalace PostgreSQL je řízen Postmasterem, který je klíčovým koordinačním procesem označovaným také jako Serverový proces.

Mezi zodpovědnosti procesu démona Postmaster patří:

- Inicializace serveru.
- Vypnutí serveru.
- Správa nových požadavků na připojení klientů.
- Provádění obnovení systému.
- Provádění procesů na pozadí.

Sdílená paměť: Jedná se o paměť, ke které může přistupovat více programů současně, aby se dosáhlo rychlých a efektivních výsledků s nízkou redundancí. V PostgreSQL je tato paměť alokována pro ukládání do mezipaměti databáze a do mezipaměti protokolu transakcí.

Sdílená disková vyrovnávací paměť: Její primární funkcí je snížit počet vstupních a výstupních operací na disku. Bez ní by byly diskové operace pomalejší, což by vedlo k redundanci a neefektivitě systému. Výhodou implementace sdílené vyrovnávací paměti je zkrácení doby zpracování, snadnější přístup k velkým datovým sadám a snížení rizika přehřátí při současném přístupu více uživatelů.

Sdílené tabulky: Tato strategie využívá společnou množinu tabulek pro ukládání dat od více klientů. Hlavními výhodami tohoto přístupu jsou nižší náklady na hardware, nižší náklady na zálohování a možnost zpracování velkých datových sad v rámci jedné databáze.

Systém UNIX: V systému UNIX Kernel Disk Buffer spravuje vyrovnávací paměť a poskytuje fyzické úložiště pro data na diskovém úložišti. Kromě toho jsou příkazy PostgreSQL kontrolovány, aby byla zajištěna správná syntaxe, přičemž chybová hlášení vysvětlují případné chybějící prvky nebo problémy. [19]

5.1.3 Back-end proces

Postmaster hraje klíčovou roli při správě počátečních klientských připojení neustálým sledováním příchozích připojení na zadaném portu. Po přijetí spojení provede inicializační proces včetně ověření uživatele a poté spustí nový proces backendového serveru, který bude příchozího klienta spravovat. Klienti pak komunikují pouze s tímto procesem backendového serveru, který zpracovává úlohy, jako je zpracování dotazů a doručení výsledků, což demonstruje implementaci modelu Process-per-transaction v PostgreSQL.

Úkolem backendového serveru je provádět dotazy zadané klientem prostřednictvím určitých operací. Každý backendový server zpracovává v jednom okamžiku pouze jeden dotaz, přesto může současně pracovat několik backendových serverů díky tomu, že se k systému současně připojuje více klientů. Tyto backendové servery načítají data z vyrovnávací paměti hlavní paměti umístěné ve sdílené paměti. [19]

5.1.4 Shared Pool

Shared Pool je část paměti RAM v rámci haldy RAM, která se vytvoří při spuštění. Je součástí SGA (System Global Area). Bez dostupnosti nebo použití Shared Pool v paměti RAM by došlo k nárůstu obnovování knihovní a řádkové mezipaměti.

Na rozdíl od mnoha databázových systémů, jako je Oracle, kde je Shared Pool důležitou součástí, PostgreSQL Shared Pool nemá. Místo toho PostgreSQL nabízí funkci, která umožňuje sdílet informace SQL na úrovni procesu. To znamená, že pokud uživatel provede stejný dotaz SQL několikrát v rámci stejného procesu, stačí, aby byl dotaz tvrdě analyzován pouze jednou. To je efektivnější ve srovnání s jinými databázovými systémy, které využívají Shared Pool, kde je nutné provést hard-parse při každém načítání jednoho příkazu SQL z Shared Pool. Pokud je

jeden SQL dotaz prováděn opakovaně ve stejnou dobu, může to v systémech využívajících Shared Pool vést ke zvýšení zátěže. [19]

5.2 Funkce

PostgreSQL nabízí funkce, často označované jako uložené procedury, které umožňují provádět operace, které by obvykle vyžadovaly více dotazů a interakcí, pouze jednou operací v rámci databáze. Tyto funkce usnadňují opakované použití kódu tím, že umožňují ostatním aplikacím přímo využívat uložené procedury, čímž se eliminuje potřeba prostřední vrstvy nebo replikace kódu.

Tyto funkce lze vytvářet v různých programovacích jazycích, například v SQL, PL/pgSQL, C a Pythonu. [21] [22]

5.2.1 Syntaxe

Základní struktura pro vytvoření funkce je popsána níže.

```
CREATE [OR REPLACE] FUNCTION function_name (arguments)
RETURNS return_datatype AS $variable_name$
  DECLARE
    declaration;
    [...]
  BEGIN
    < function_body >
    [...]
    RETURN { variable_name | value }
  END; LANGUAGE plpgsql;
```

Obrázek 4: Architektura aplikace (zdroj [22])

- Function-name označuje název funkce.
- [OR REPLACE] slouží k úpravě existující funkce.
- Funkce musí povinně obsahovat příkaz return.
- Výraz RETURN určuje typ dat, která funkce vrátí. Tento return_datatype může být základní, složený nebo doménový typ nebo může odpovídat typu sloupce v tabulce.
- Tělo funkce obsahuje spustitelný segment funkce.
- Klíčové slovo AS slouží k vytvoření samostatné funkce.

- plpgsql označuje programovací jazyk, ve kterém je funkce napsána. V případě PostgreSQL to může být alternativně SQL, C, interní nebo jiný uživatelsky definovaný procedurální jazyk. Pro zajištění zpětné kompatibility může být název jazyka uzavřen v jednoduchých uvozovkách. [21] [22]

5.3 Triggery

Triggery PostgreSQL fungují jako zpětné volání v rámci databáze, které se automaticky provádí nebo vyvolává v reakci na konkrétní události v databázi.

Aktivaci triggerů lze nakonfigurovat:

- Před provedením operace na řádku (například INSERT, UPDATE nebo DELETE). Předtím, než jsou zkontrolována omezení a než je proveden samotný pokus o operaci.
- Po ukončení operace, po kontrole omezení a dokončení INSERT, UPDATE nebo DELETE.
- Místo skutečné operace, zejména v případech, kdy se jedná o vkládání, aktualizace nebo mazání ve view.

Mezi klíčové aspekty triggerů PostgreSQL patří:

- Trigger definovaný jako FOR EACH ROW se spouští jednotlivě pro každý řádek, kterého se operace týká. Naopak trigger FOR EACH STATEMENT se spouští pouze jednou pro každou operaci bez ohledu na počet řádků.
- Triggery mohou používat podmínku WHEN a akce triggeru k interakci s prvky vkládaného, odstraňovaného nebo aktualizovaného řádku prostřednictvím odkazů jako například NEW.column-name nebo OLD.column-name, kde „column-name“ odkazuje na sloupec v tabulce propojené s triggerem.
- Pokud je zadána podmínka WHEN, zadané příkazy PostgreSQL se provedou pouze pro řádky, u kterých jsou splněny podmínky WHEN. Bez zadání podmínky WHEN se příkazy použijí na všechny řádky.
- Pokud existuje více triggerů stejného typu pro stejnou událost, spouštějí se v abecedním pořadí podle svých názvů.
- Klíčová slova BEFORE, AFTER nebo INSTEAD OF určují časování spouštěcích akcí vůči úpravě, přidání nebo odebrání příslušného řádku.

- Triggery jsou automaticky odstraněny, když je zrušena tabulka, která je s nimi spojena.
- Tabulka, která má být změněna, musí být ve stejné databázi jako tabulka nebo view, ke kterému je trigger přiřazen, a měl by se použít pouze název tabulky bez předpony názvu databáze.
- Zadáním parametru CONSTRAINT se vytvoří omezující trigger, který funguje stejně jako běžný trigger, ale umožňuje řídit časování provedení triggeru pomocí SET CONSTRAINTS. Očekává se, že omezující triggery budou generovat výjimku, pokud dojde k porušení implementovaných omezení.[21] [23]

5.3.1 Syntaxe

Struktura triggeru v obecném případě je následující

```
CREATE  TRIGGER trigger_name [BEFORE|AFTER|INSTEAD OF] event_name
ON table_name
[
 -- Trigger logic goes here....
];
```

Obrázek 5: Syntaxe triggeru (zdroj [23])

V tomto kontextu může event_name odkazovat na databázové operace. Například INSERT, DELETE, UPDATE nebo TRUNCATE, v zadané tabulce table_name. Volitelně lze za název tabulky uvést klíčový výraz FOR EACH ROW. [21] [23]

5.4 Výhody a nevýhody PostgreSQL

Jako každá technologie má i PostgreSQL své výhody a nevýhody, které je třeba zvážit při výběru databáze pro konkrétní účel.

5.4.1 Výhody

PostgreSQL má mnoho výhod oproti ostatním databázím a některé z nich jsou klíčové.

- **Transakce:** V případě, že transakce neexistují, je třeba zvážit dodatečné kódování nutné pro ošetření chyb. Opakované začlenění takového ošetření chyb do programů by vyžadovalo knihovnu na straně klienta a pravděpodobně by vyžadovalo použití transakčních labelů. To se však s databází, která transakce podporuje, stává zbytečným. PostgreSQL nabízí transakční DDL, které rozšiřuje typické operace INSERT, UPDATE nebo DELETE. Zahrnuje akce, jako je vytváření, rušení nebo změna tabulek, což

výrazně liší od toho, jak s transakcemi pracují některé jiné databáze, například Oracle. Například jakákoli změna tabulky v PostgreSQL je okamžitě zaznamenána jako součást transakce. Tato vlastnost je klíčová ve složitých relačních aplikacích, kde se často mění aplikace a základní databázové schéma současně. Implementace změn tímto způsobem umožňuje zpracovat všechny aktualizace v rámci jediné transakce, což zefektivňuje proces aktualizace v celé aplikaci.

- **Komentáře:** PostgreSQL se od ostatních databázových aplikací liší tím, že nabízí komentáře ke kódu. Tyto komentáře poskytují přehled o tom, co konkrétní kód splňuje nebo nesplňuje, než je dokončen pro použití v aplikaci. Tato transparentnost umožňuje hlubší pochopení struktury kódu a usnadňuje otevřený inženýrský proces. Přístupnost a srozumitelnost, kterou komentáře ke kódu poskytují, zvyšuje bezpečnost a kvalitu a podporuje růst komunity. Více přispěvatelů se může zapojit a pochopit vývoj v kódu, což podporuje dynamickou a aktivní open-source komunitu namísto pouze statického kódu hostovaného na GitHubu.
- **Parametry:** Úkolem parametrů v databázi je umožnit konfigurovatelná nastavení, která lze podle potřeby upravovat. V případě nutnosti změny těchto nastavení existuje možnost podívat se do manuálu, zjistit jejich funkci a provést potřebné úpravy. PostgreSQL je vynikající díky mnoha laditelným parametrům. Na rozdíl od většiny databázových systémů, kde se parametry prostředí nastavují v rámci celé databáze, PostgreSQL umožňuje provádět úpravy na podrobnějších úrovních. Počet parametrů není tak podstatný jako flexibilita při jejich úpravách podle konkrétních požadavků. Například PostgreSQL obsahuje interní parametry, které lze přizpůsobit podle způsobu využití systému, například přidělení požadovaného množství paměti serveru. Tyto parametry jsou dokumentovány v rámci databáze. Flexibilita se kromě toho rozšiřuje na místa, kde lze tyto parametry nastavit - od celé relace až po jednotlivé transakce a funkce.
- **Rozšiřitelnost:** Klíčovou vlastností PostgreSQL je její vysoká rozšiřitelnost. Struktura databáze není pevná a umožňuje přidávat nové funkce podle potřeby. Tato flexibilita není u jiných databázových systémů častá a zahrnuje různé aspekty PostgreSQL. Umožňuje integraci nových funkcí, datových typů a programovacích jazyků jednoduše pomocí příkazu CREATE EXTENSION, který se postará o zbytek integračního procesu. PostgreSQL slouží jako univerzální platforma pro vylepšování databází a

podporuje širokou řadu programovacích jazyků včetně PL/pgSQL, PL/Python, Java a dokonce i JavaScript. Systém poskytuje rozhraní, které umožňuje každému vyvíjet a spouštět vlastní skriptovací jazyky v prostředí databáze. Při spouštění funkcí zůstává základní jazyk transparentní, což zjednodušuje uživatelské prostředí tím, že maskuje složitost použitého programovacího jazyka. Tato vlastnost rozšiřitelných jazykových rozhraní umožňuje vývoj rozsáhlých knihoven kódu, které v databázi fungují bez problémů. Tato schopnost je velmi cenná zejména pro velké organizace, kterým umožňuje vyvíjet na míru sestavené funkční balíčky, které vyhovují konkrétně jejich potřebám.

- **Bezpečnostní funkce**: PostgreSQL obsahuje vestavěné bezpečnostní funkce a další rozšíření, která mohou zlepšit bezpečnostní opatření. Je mezinárodně uznávaná pro své robustní bezpečnostní funkce. PostgreSQL zajišťuje bezpečnost prostřednictvím konfigurovatelných parametrů a ochran na úrovni aplikace. Pokud jde o zabezpečení parametrů, umožňuje podrobné konfigurace na úrovni operačního systému, které zabezpečují prostředí kolem databáze. V oblasti zabezpečení aplikací spravuje přístup na základě uživatelských oprávnění a rozděluje účty do kategorií, jako je například pouze pro čtení, čtení/zápis nebo jiné specifické akce. Kromě samotného přidělování oprávnění uživatelům pro přístup umožňuje také stanovit průběžná oprávnění, čímž zvyšuje kontrolu a bezpečnost. [20]

5.4.2 Nevyhody

V PostgreSQL je poměrně málo kritických nedostatků, ale některé aspekty mohou být rozhodujícím faktorem při výběru databáze pro určité účely.

- **Struktura databáze:** Databáze v Postgresu se obvykle používá jako relační databáze. To znamená, že máme databázi, která pracuje s určitými předpoklady nebo určitým způsobem. A samozřejmě pouziva jazyk SQL. Taková databáze pracuje s tabulkami jako s datovým košem nebo úložným kontejnerem. Takže ve světě SQL, když se budeme dotazovat na data pomocí SQL, budeme mít velmi přísné požadavky na data, která ukládáme do naší databázové tabulky. S jasným schématem, která data mohou do tabulky patřit, a toto schéma je definováno poli. Každá nová entita nebo záznam, který přidáváme, má hodnotu těchto polí. Důležité ale je, že nesmí mít více polí, než kolik jich pro tabulku definujeme. Není možné, aby jeden záznam měl název a popis ceny a

další záznam měl také název a popis ceny. Pokud chceme ke stávajícím údajům přidat nějaké další údaje jako například komentáře nebo atributy údajů, nelze to udělat jen tak, že k nim přidáme samostatné pole, ale pak musí mít všechny záznamy nebo všechny záznamy další pole. A ať už je máte, nebo ne, musíte u všech těchto polí uvést nějaké informace. Na druhou stranu databáze NoSQL, jako je například MongoDB, ukládá data jako dokument. Dokumenty mohou mít různé atributy, takže neexistuje žádné pevné schéma.

- **Open source:** Postgres open source databázová aplikace, kterou nevlastní žádný subjekt. Nabízí množství funkcí a působivou funkčnost, ale zpočátku se těžko rozšiřoval ve srovnání s proprietárním softwarem, který si nad svými produkty ponechává výhradní kontrolu a autorská práva. V důsledku toho neposkytuje žádnou záruku a je osvobozen od jakéhokoli krytí odpovědnosti nebo odškodnění. Další nevýhodou otevřeného softwaru je, že se o jeho správu často stará více komunitních skupin. Proto musí být základní kód obzvláště transparentní, aby v případě, že je odpovědnost za software převedena na jinou skupinu, zůstal přístupný a srozumitelný. To může vést k potenciálním problémům s příjemností uživatelského rozhraní nebo k tomu, že funkce, které mohou být dnes známé, zastarají. Kromě toho mohou nastat problémy s kompatibilitou s některými systémy a v některých případech může být k efektivnímu běhu programu zapotřebí specifický software nebo hardware.
- **Problémy s výkonem**: Uživatelé Postgresu se často setkávají s potížemi souvisejícími s výkonem a obnovou záloh. Běžně se stává, že dříve rychlý dotaz se výrazně zpomalí, což znamená pokles výkonu v databázovém prostředí. Vzhledem ke své relační konstrukci Postgres začíná vyhledávat od prvního řádku a prohledává celou tabulku, aby našel potřebná data. Tento proces vede ke zpomalení výkonu, zejména pokud se jedná o tabulky, které obsahují velké množství dat v mnoha řádcích a sloupcích a obsahují mnoho polí s rozsáhlými informacemi pro porovnání. [20]

6 Apache Cassandra

Apache Cassandra je open-source NoSQL databáze, která funguje v distribuované síti. Využívá přístup rozděleného ukládání do širokých sloupců a zajišťuje případnou konzistenci.

Databáze Apache Cassandra, původně vyvinutá ve společnosti Facebook, kombinuje prvky Dynamo od společnosti Amazon a Bigtable od společnosti Google. Zatímco Dynamo poskytovalo robustní rámec pro distribuované úložiště a replikaci a Bigtable nabízel pokročilý úložný engine, oba systémy měly nedostatky, které bylo třeba odstranit, aby splňovaly požadavky na škálovatelná, spolehlivá a vždy dostupná úložná řešení.

Cassandra představuje optimalizovanou syntézu těchto technologií, která byla vytvořena tak, aby splňovala požadavky na správu rozsáhlých dat a vysoké objemy dotazů. Když globální aplikace začaly vyžadovat komplexní globální replikaci a trvale nízkou latenci přístupu k datům, tradiční relační databáze již nedokázaly tyto rozsáhlé potřeby uspokojit.

Cassandra je speciálně navržena tak, aby tyto výzvy splňovala, a snaží se dosáhnout několika klíčových cílů návrhu:

- Kompletní replikace mezi více předlohami
- Celosvětový přístup se sníženou latencí
- Rozšíření pomocí standardního, cenově dostupného hardwaru
- Úměrné zvýšení propustnosti s každým přidaným procesorem
- Rozdělování zátěže a rozšiřování clusteru v reálném čase
- Dotazy založené na rozdělených klíčích
- Přizpůsobitelná struktura databáze [9] [13]

6.1 Vlastnosti

Cassandra nabízí jazyk Cassandra Query Language (CQL), který je podobný jazyku SQL a slouží k vytváření schémat, aktualizacím a vyhledávání dat. Jazyk CQL umožňuje uživatelům spravovat data v clusteru uzlů Cassandra pomocí:

- **Klíčový prostor (Keyspace)**: Určuje strategii replikace datové sady v rámci každého datového centra. Replikace se týká počtu kopií udržovaných v rámci clusteru. Klíčové prostory zahrnují tabulky.

- **Tabulka**: Stanoví strukturované schéma pro sadu oddílů. Tabulky obsahují oddíly, které obsahují řádky, a tyto řádky se skládají ze sloupců. Tabulky Cassandra umožňují bezproblémové přidávání nových sloupců bez výpadků.
- **Rozdělení (Partition)**: Určuje základní komponentu primárního klíče, který každý řádek v Cassandře vyžaduje k určení uzlu v clusteru, kde je řádek uložen. Efektivní dotazy vždy obsahují rozdělovací klíč (partition key).
- **Řádek**: Uchovává kolekci sloupců jednoznačně identifikovaných primárním klíčem složeným z klíče rozdělení a případně dalších klíčů klastrování.
- **Sloupec**: Představuje jednotlivý datový prvek v rámci řádku, charakterizovaný jeho typem.

Jazyk CQL podporuje řadu pokročilých funkcí nad rozdělenou datovou sadou, jako jsou:

- Atomické transakce v rámci jednoho oddílu s použitím funkce compare-and-set.
- Přizpůsobitelné typy, funkce a agregáty definované uživatelem.
- Typy kolekcí, jako jsou množiny, mapy a seznamy.
- Pomocné indexy lokalizované do konkrétních oblastí.
- Materializované view (v experimentální fázi).

Cassandra se úmyslně vyhýbá implementaci funkcí, které potřebují koordinaci mezi oddíly, protože ty jsou často pomalé a náročné na udržení vysoké dostupnosti v globálním rozsahu. Cassandra například nepodporuje:

- Transakce, které se provádí na několika oddílech
- Spojení (join) rozdělené mezi uzly
- Cizí klíče nebo omezení referenční integrity [9] [13]

6.2 Provozování

Konfigurace pro Apache Cassandra je řízena pomocí souboru cassandra.yaml, který lze editovat ručně nebo měnit pomocí nástrojů pro správu konfigurace. Některá nastavení lze měnit za běhu pomocí online rozhraní, jiná však vyžadují restart databáze, aby se uplatnila.

Cassandra nabízí různé nástroje pro správu clusteru. Příkaz 'nodetool' poskytuje přístup k rozhraní pro ovládání Cassandry a umožňuje upravovat řadu nastavení uvedených v souboru cassandra.yaml za běhu. Příkaz 'auditlogviewer' slouží ke kontrole auditních protokolů a příkaz

'fqltool' umožňuje prohlížení, přehrávání a porovnávání úplných protokolů dotazů. Auditlogviewer a fqltool jsou doplňky v Apache Kasandra 4.0.

Kromě toho Cassandra obsahuje integrovanou funkci atomických snímků (snapshot), která nabízí okamžitý snímek dat pro integraci s řadou zálohovacích řešení. Rovněž podporuje inkrementální zálohování, což umožňuje zálohovat data v okamžiku, kdy jsou zaznamenána.

Apache Cassandra 4.0 přináší několik nových funkcí, například virtuální tabulky, experimentální přechodnou replikaci, protokolování auditů, úplné protokolování dotazů a kompatibilitu s Javou 11 (plně podporovanou od verze 4.0.2). [9] [13]

6.3 Replikace dat

Cassandra zajišťuje spolehlivost a odolnost proti chybám pomocí ukládání kopií dat ve více uzlech. Umístění těchto replik je určeno replikační strategií a celkový počet replik v rámci clusteru je označován jako replikační faktor. Replikační faktor 1 znamená, že v rámci clusteru existuje jediná kopie každého datového řádku, což vede ke ztrátě dat v případě selhání hostitelského uzlu. Replikační faktor 2 zajišťuje, že jsou dvě kopie každého řádku, z nichž každá je uložena v jiném uzlu. Všechny repliky mají rovnocenný význam; žádná replika není považována za primární nebo hlavní. Obvykle replikační faktor nesmí být vyšší než počet uzlů v clusteru, nicméně je možné nejprve zvýšit replikační faktor a poté odpovídajícím způsobem rozšířit počet uzlů.

Dvě hlavní strategie replikace jsou:

- **SimpleStrategy**: Je vhodná pro použití pouze v rámci jednoho datového centra. Pokud existuje pravděpodobnost budoucího rozšíření do více datových center, měla by se místo toho použít strategie NetworkTopologyStrategy.
- **NetworkTopologyStrategy**: Doporučuje se pro většinu nastavení, zejména pokud se předpokládá rozšíření do dalších datových center. Tato strategie umožňuje zadat požadovaný počet replik na datové centrum. Strategicky umisťuje repliky do stejného datového centra a pokud možno do různých serverových stojanů, aby se zmírnila současná selhání, ke kterým může dojít v důsledku problémů, jako jsou poruchy napájení, chlazení nebo sítě.

Při plánování počtu replik na datové centrum je třeba uvažovat dva hlavní faktory: schopnost provádět lokální čtení bez zpoždění způsobeného komunikací mezi datovými centry a možné scénáře selhání. Mezi běžné konfigurace clusterů s více datovými centry patří:

- Dvě repliky v každém datovém centru: Tato konfigurace dokáže odolávat selhání jednoho uzlu pro replikační skupinu a přitom stále podporovat lokální čtení s úrovní konzistence ONE.
- Tři repliky v každém datovém centru: To dovoluje selhání jednoho uzlu v replikační skupině se silnou úrovní konzistence LOCAL_QUORUM nebo selhání více uzlů v každém datovém centru s úrovní konzistence ONE.

Je také možno použít asymetrická nastavení replikace. Například tři repliky mohou být uloženy v jednom datovém centru, aby zvládly obsluhování požadavků aplikací v reálném čase, zatímco jedna replika v jiném datovém centru se používá pro analytické účely. [13] [14]

7 Messenger

V této bakalářské práci byl vyvinut Messenger založený na výše popsaných technologiích. V následujícím textu budou popsány klíčové aspekty architektury aplikace a její implementace

7.1 Použité technologie

Důležitým faktorem při výběru firemního messengeru je jeho škálovatelnost. S rostoucím počtem uživatelů roste především zatížení modulů aplikace zodpovědných za zasílání okamžitých zpráv a oznámení o stavu mezi uživateli a databáze zodpovědné za ukládání historie zpráv. Ve vyvíjené aplikaci se jedná o server Ejabberd (přenos zpráv a oznámení o stavu) a databázi Cassandra (ukládání zpráv). Klíčovou vlastností těchto modulů je možnost vytvoření clusteru, který umožňuje škálování aplikaci a je omezen pouze kapacitou infrastruktury společnosti.

S rostoucím počtem uživatelů roste zatížení ostatních aplikačních služeb v menší míře, přesto však vyžaduje určité nástroje pro zvýšení spolehlivosti a odolnosti proti chybám. Zatímco aplikace Angular běží přímo na zařízení uživatele a nevyžaduje zvláštní změny v infrastruktuře, backend Java Spring je klíčovým článkem mezi databázemi a překročení přípustné zátěže této služby vyvolá nestabilitu v práci jednotlivých modulů i aplikace jako celku. Především backendový server je omezen počtem současných spojení, jejichž požadavky je schopen správně zpracovat. Pro znásobení počtu povolených současných spojení a zmírnění následků jejich překročení používá backend v Javě reaktivní webový server Netty, jehož hlavní vlastností je možnost používat neblokující volání. Vzhledem k tomu, že tradiční řešení nejsou pro správnou funkci reaktivního serveru vhodná vzhledem k tomu, že jsou původně navržena pro používání blokujících volání, používá tato aplikace reaktivní verzi frameworku Spring, která je vyčleněna do samostatného modulu Spring WebFlux, což byl původně samostatný projekt s názvem Project Reactor. Kromě neblokujících volání prostřednictvím rozhraní RestAPI používá Spring WebFlux speciální reaktivní ovladače pro připojení k databázím PostgreSQL a Apache Cassandra, díky čemuž jsou volání do databáze rovněž neblokující a zvyšuje se maximální přípustná zátěž.

7.2 Požadavky

Aby messenger splňoval minimální požadavky podniku, musí mít implementovány určité funkce. Stanovené požadavky pro Messenger jsou popsány v tabulkách níže.

7.2.1 Funkční požadavky

Tabulka 1: Funkční požadavky Messengeru (zdroj vlastní)

F1	Přidávání nových uživatelů	Aplikace musí umožňovat novým uživatelům registrovat účty a přidávát všechny potřebné záznamy o nových uživatelích do všech modulů aplikace jedním požadavkem z jediného formuláře.
F2	Správa osobních údajů	Aplikace musí umožňovat uživateli zobrazovat a měnit některé osobní údaje (jméno, příjmení, avatar). Kritické údaje (jid), které jsou důležité pro správné fungování aplikace, měnit nelze.
F3	Odesílání a přijímání soukromých zpráv	Aplikace musí umožňovat uživateli zobrazovat seznam svých kontaktů, otevírat s nimi chat, zobrazovat historii soukromých zpráv a posílat nové soukromé zprávy.
F4	Odesílání a přijímání skupinových zpráv	Aplikace musí umožňovat uživateli zobrazovat seznam svých skupin, prohlížet historii zpráv skupiny a posílat nové zprávy do skupinového chatu.
F5	Odesílání souborů	Aplikace musí umožňovat uživateli přiložit soubor ke zprávě a odeslat jej jinému uživateli nebo skupině. V případě, že souborem je obrázek, musí být tento obrázek zobrazen v chatu.
F6	Správa seznamu kontaktů	Aplikace musí umožňovat uživateli prohlížet seznam zaregistrovaných uživatelů, odesílat jim žádosti o přidání do seznamu kontaktů pro další zasílání zpráv a také přijímat nebo odmítat příchozí žádosti o přidání do seznamu kontaktů.
F7	Správa seznamu účastníků skupinového chatu	Aplikace musí umožňovat vlastníkům skupiny přidávat nové členy a odstraňovat stávající členy ze skupiny.
F8	Přítomnost	Aplikace musí uživateli zobrazovat, kdo z jeho seznamu kontaktů je právě online.

7.2.2 Nefunkční požadavky

Tabulka 2: Nefunkční požadavky k frontendové části messengeru (zdroj vlastní)

N1	Připojení k serveru Ejabberd	Aplikace se musí připojit přímo k serveru Ejabberd pomocí BOSH.
N2	Komunikace s backendovým serverem	Aplikace musí komunikovat s backendovým serverem prostřednictvím RestAPI.
N3	Osobní údaje	Aplikace nesmí umožňovat změnu údajů účtu jiných uživatelů.
N4	Přístupová oprávnění	Aplikace nesmí umožňovat správu skupin a kontaktů jiných uživatelů.

Tabulka 3: Nefunkční požadavky k backendové části messengeru (zdroj vlastní)

N1	Umístěni	Všechny moduly aplikace musí být nezávislé na platformě a musí běžet na jakémkoli moderním operačním systému (Windows, MacOS, Linux atd.).
N2	Jazyk a framework	Aplikace je napsána v jazyce Java a používá reaktivní verzi frameworku Spring.
N3	RestAPI	Aplikace musí poskytovat koncové body RestAPI pro zpracování příchozích požadavků.
N4	Komunikace s Ejabberd serverem	Aplikace musí používat koncové body RestAPI serveru Ejabberd pro příjem a odesílání dat.
N5	Validace	Aplikace musí před provedením jakýchkoli akcí s daty přijatými z frontendu zkontrolovat jejich platnost a oprávnění uživatele k provádění operací s těmito daty.

7.3 Bezpečnostní požadavky

Vzhledem k tomu, že prostřednictvím messengeru mohou být přenášena citlivá firemní data, existují určité bezpečnostní požadavky. Pro vyvíjenou aplikaci byly stanoveny následující bezpečnostní požadavky:

- Možnost instalace a konfigurace aplikace na vlastní infrastruktuře
- Databáze ukládající informace mohou být nainstalovány a nakonfigurovány na vlastní infrastruktuře společnosti
- Nástroje pro správu, které slouží k prohlížení statistik, stavu aplikací a správě dat

- Hesla by měla být ukládána pouze v zašifrované podobě

7.4 Implementace

Tato část je věnována přímé implementaci aplikace, přizpůsobení, vývoji a instalaci jejích jednotlivých modulů.

7.4.1 Ejabberd XMPP

Klíčovým modulem aplikace je server XMPP Ejabberd. Ejabberd poskytuje všechny potřebné funkce pro instant messaging, správu přítomnosti uživatelů a skupinové chaty.

Server Ejabberd má základní konfiguraci v jazyce YAML. Přestože je server schopen běžet s původní konfigurací, byla původní konfigurace mírně upravena, aby splňovala požadavky aplikace.

V konfiguračním souboru byly provedeny následující změny:

- Vypnutá podpora fronty offline zpráv. K ukládání offline zpráv se používá databáze Apache Cassandra.
- Zapnuta podpora BOSH.
- Zapnutá podpora vestavěné webové stránky administrátora pro správu serveru a prohlížení statistik
- Zapnuta podpora rozhraní REST Api.

Aktuální konfigurační soubor je v přílohách tohoto dokumentu.

Webová stránka správce umožňuje zobrazovat a upravovat následující informace.

- Virtuální hostitelé.
- Zobrazení seznamu uživatelů a registrace nových uživatelů.
- Seznam online uživatelů.
- Seznam skupinových chatů a základní informace o nich

Zobrazení seznamu uživatelů a registrace nových uživatelů. Přestože tato funkce je k dispozici, tento způsob registrace uživatelů se nedoporučuje. Pro správný běh aplikace je nutné přidat do databáze PostgreSQL další informace o uživateli.

Obrázek 6: Webová stránka administrátora: Seznam registrovaných uživatelů (zdroj vlastní)

Zbývající stránky tohoto modulu nejsou pro vyvíjenou aplikaci relevantní a nejsou předmětem této práce.

Přenos dat mezi uživateli Ejabberdu ve vyvíjené aplikaci se provádí pomocí spojení BOSH mezi webovou aplikací Angular a serverem Ejabberd ve formě XMPP Stanza.

XMPP Stanza může být třech typů (message, presense a iq). Iq Stanza se používají k odesílání požadavků na server, ale v této aplikaci se všechny potřebné interakce se serverem provádějí z backendového serveru Java prostřednictvím rozhraní Rest API, protože požadují synchronní zpracování dat.

Message Stanza obsahuje zprávy odeslané mezi uživateli nebo odeslané do skupinového chatu. Vyvinutá aplikace používá modifikovanou verzi Message Stanza pro podporu výměny souborů mezi uživateli. Kromě obsahu zprávy může uživatel získat také cestu do souboru na serveru, která je obsažena v dodatečném tagu XML "attachment". V aplikaci Messenger server Ejabberd neukládá soubory interně, tuto funkci plní souborový systém backend serveru.

```
Stanza received: Stanza<message xml:lang="en" to="pavelk@localhost"      messaging.service.ts:73
from="pavel@localhost/13356598713755733455236 82" type="chat" xmlns="jabber:client"><archived
by="pavelk@localhost" id="1714418724942475" xmlns="urn:xmpp:mam:tmp"/><stanza-id
by="pavelk@localhost" id="1714418724942475" xmlns="urn:xmpp:sid:0"/>
<attachment>/Users/pavelkireev/messenger-storage/attachment/a06d4c89-a2e3-401b-9c66-
ef5ac0bd20b5.jpg</attachment><body>Hey, how's it going?</body></message>
```

Obrázek 7: Příklad Message Stanza (zdroj vlastní)

Druhým typem Stanza použitým v aplikaci je Presense Stanza. Tento typ Stanza se používá k vzájemnému informování uživatelů o jejich statusech.

```
Stanza received: Stanza<presence xml:lang="en"                    messaging.service.ts:73
to="pavelk@localhost/3361481873441811602278" from="pavelk@localhost/17381041269938051455316"
xmlns="jabber:client"><x xmlns="vcard-temp:x:update"/></presence>
```

Obrázek 8: Příklad Presence Stanza (zdroj vlastní)

Přestože uživatel může posílat zprávy jakémukoli registrovanému uživateli na serveru. Aby mohl Ejabberd začít informovat uživatele o jejich vzájemné přítomnosti, musí být do serveru přidán příslušný Roster Item.

Roster Item je záznam na serveru Ejabberd, který indikuje, že dva uživatelé chtějí být vzájemně informováni o jejich přítomnosti v síti. Ve vyvíjené aplikaci jsou tyto záznamy přidávány pomocí volání REST Api z Java backendu obsahujícího potřebné informace pro přidání záznamu.

Jak bylo popsáno výše, všechny požadavky na server jsou odesílány pomocí rozhraní Ejabberd Rest API z backendového serveru Java. Pomocí těchto volání lze provádět následující akce:

- Registrace uživatele.
- Správa skupinových chatů.
- Přidání Roster Item.

7.4.2 Java Spring WebFlux Backend

Hlavní úlohou serverové aplikace Java je správa uživatelských dat a interakce s rozhraním Ejabberd Rest API. Klíčovým aspektem Spring WebFlux je použití neblokujících volání. Proto by všechna volání měla být zpracována asynchronně, aby byl dosažen optimální výkon aplikace. Ve vyvinuté aplikaci Spring WebFlux umožňuje asynchronně zpracovávat databázové požadavky, odchozí požadavky HTTP, příchozí požadavky HTTP.

Zdrojový kód aplikace je rozdělen do jednotlivých složek, z nichž každá obsahuje třídy podle jejich účelu. Všechny potřebné konfigurační soubory jsou uloženy v samostatném adresáři.

Navrženo podle [24]. Typy tříd a jejich popisy jsou uvedeny v následující tabulce.

Tabulka 4: Struktura aplikaci (zdroj vlastní)

Název	**Popis**
Client	Třídy potřebné k interakci se serverem Ejabberd v roli klienta
Configuration	Konfigurační třídy pro kontejner Spring. Obsahuje konfigurace pro webový server, autorizaci uživatelů, databázová připojení
Controller	Třídy kontrolérů, které obsahují logiku zpracování požadavků na webový server
DB	Obsahuje entity definující vztahy s tabulkami databází Apache Cassandra a Postgres a také odpovídající třídy – repositáře obsahující logiku databázových dotazů
DTO	Reprezentační třídy pro vytváření a zpracování těl HTTP POST requestů
Handler	Třídy zpracovávající různé scénáře kontrolérů, např. zpracování chyb
Model	Třídy reprezentace formuláře z frontendu, které se předávají v těle HTTP requestů
Service	Třídy serverové vrstvy mezi řadiči a ostatními třídami serveru. V této aplikaci kombinují logiku z tříd repozitářů a klientů.
Validator	Třídy ověřující korektnost dat přijatých z frontendu.

Interakce se serverem Ejabberd probíhá prostřednictvím rozhraní Rest API. Pro všechny interakce existuje jediná třída, která obsahuje logiku volání sevreru. Všechny interakce jsou definovány v rozhraní `cz.upce.messenger.client.EjabberdClient` a implementované ve třídě `cz.upce.messenger.client.EjabberdClientImpl`. Všechna volání jsou prováděna asynchronně pomocí třídy `WebClient`, což je reaktivní verze RestTemplate. Navrženo podle [24]

Při asynchronním přístupu se volání provádí pomocí metody subscribe. Pokud se při volání neočekává žádná odpověď, volá se metoda subscribe přímo uvnitř metody třídy klienta.

```
2 usages  pavelkireev
@Override
public void registerUser(String username, String password) {
    ejabberdWebClient.post() RequestBodyUriSpec
                    .uri(ejabberdBaseUrl + "/api/register") RequestBodySpec
                    .bodyValue(new RegisterUserRequest(username, password, "localhost"))
                    .retrieve() ResponseSpec
                    .bodyToMono(Void.class) Mono<Void>
                    .subscribe();
}
```

Obrázek 9: Příklad voláni metody `Subscribe` *(zdroj vlastní)*

Pokud se však od serveru Ejabberd očekává odpověď, která vyžaduje zpracování, bude odpovědnost za přímé volání předána servisní třídě ve formě připraveného objektu Flux nebo Mono. Následující obrázek ukazuje příklad implementace `getRoster`. Tato metoda se dotazuje na roster konkrétního uživatele podle jeho `jid`. Po úspěšném provedení požadavku metoda vrací objekt `RosterResponseDto` zabalený do objektu Flux, který je určen k vyvolání a zpracování jinými metodami.

```
1 usage  pavelkireev
@Override
public Flux<RosterResponseDto> getRoster(String username) {
    return ejabberdWebClient.get() RequestHeadersUriSpec<capture of ?>
                    .uri(ejabberdBaseUrl + "/api/get_roster?user=" + username + "&server=localhost")
                    .retrieve() ResponseSpec
                    .bodyToFlux(RosterResponseDto.class);
}
```

Obrázek 10: : Implementace metody `getRoster` *(zdroj vlastní)*

Z pohledu kontejneru Spring je klientská třída službou.

Jak bylo popsáno výše, servisy představují doplňkovou vrstvu mezi kontrolérem a ostatními třídami aplikace. V této aplikaci servisní třídy kombinují logiku tříd repozitáře a tříd klientů.

Při interakci s metodami klientské třídy existují dva scénáře v závislosti na tom, jestli klientská metoda vrací nějaký objekt (Mono nebo Flux), nebo ne. Pokud třída klient nevrací objekt a volání provede sama, není nutné žádné další zpracování. Tento scénář je vidět na obrázku na příkladu registrace nového uživatele. Metoda signUp kombinuje logiku tříd EjabberdClient a UserService voláním jejich metod s příslušnými parametry. Vzhledem k tomu, že metoda

registerUser v EjabberdClient `může` vyhodit výjimku, je celá logika metody zabalena do bloku `try-catch`, abychom v případě problémů s připojením k Ejabberdu nevytvářeli zbytečné záznamy v databázi PostgreSQL. Pokud je vyhozena výjimka, provádění logiky metody je přerušeno a v logu aplikace je vytvořen odpovídající záznam o chybě.

```java
@Override
public void signUp(RegistrationModel model) {
    try {
        ejabberdClient.registerUser(model.getUsername(), model.getPassword());
        userService.create(model);
    } catch (
        Exception e
    ) {
        log.error(e.getMessage());
    }
}
```

Obrázek 11: Implementace metody `signUp` *(zdroj vlastní)*

Pokud požadujeme `nějaká` data a očekáváme, že klientská metoda vrátí objekt Flux nebo Mono, je služba povinna definovat logiku volání a zpracování tohoto objektu.

Příkladem takového `zpracování` je metoda, která načítá seznam skupinových chatů uživatele v `cz.upce.messenger.service.ejabberd.GroupChatServiceImpl` na obrázku. Tato metoda zároveň kombinuje data získaná ze serveru Ejabberd s daty uloženými v databázi PostgreSQL, která získává voláním příslušných metod třídy repozitáře příslušné entity.

```
1 usage  new *
@Override
public Flux<EnrichedGroupDto> getEnrichedGroups(String userJid) {
    return ejabberdClient.getUserRooms(userJid) Flux<UserGroupSubscriptionDto>
                         .flatMap(group -> groupRepository.findByMucJid(group.getRoomjid())) Flux<Group>
                         .flatMap(group -> memberGroupRepository.findAllMembersByMucJid(group.getMucJid()) Flux<String>
                                       .collectList() Mono<List<...>>
                                       .zipWith(Mono.just(group))) Flux<Tuple2<...>>
                         .flatMap(pair -> {
                             List<String> member = pair.getT1();
                             Group group = pair.getT2();
                             return userRepository.findByJidList(member) Flux<User>
                                         .collectList() Mono<List<...>>
                                         .zipWith(Mono.just(group));
                         }).map(pair -> {
                             List<User> users = pair.getT1();
                             Group group = pair.getT2();

                             List<MemberDto> members = users.stream() Stream<User>
                                              .map(user -> MemberDto.builder()
                                                           .jid(user.getJid())
                                                           .firstName(user.getFirstName())
                                                           .lastName(user.getLastName())
                                                           .build()) Stream<MemberDto>
                                              .toList();

                             EnrichedGroupDto enrichedGroupDto = EnrichedGroupDto.builder()
                                                          .mucJid(group.getMucJid())
                                                          .name(group.getName())
                                                          .build();
                             enrichedGroupDto.setMembers(members);
                             return enrichedGroupDto;
                         });
}
```

Obrázek 12: Implementace metody `getEnrichedGroups` *(zdroj vlastní)*

Tato metoda zároveň kombinuje data získaná ze serveru Ejabberd s daty uloženými v databázi PostgreSQL, která získává voláním příslušných metod třídy repozitáře příslušné entity.

Na obrázku je uveden také příklad volání metody třídy repozitáře. Z hlediska servisní třídy se příliš neliší od klientské třídy a vrací objekty Mono nebo Flux, které jsou volány a zpracovávány přímo v servisní třídě

Pro komunikaci s databázemi se v reaktivní verzi Spring Frameworku používají speciální reaktivní drivery. Všechna data přijatá z databáze jsou proto také vracena jako objekty Flux a Mono a zpracovávají se asynchronně.

Reaktivní třída repozitáře neprovádí přímá volání do databáze, ale pouze konstruuje objekty Mono nebo Flux a deleguje odpovědnost za volání a zpracování na servisní vrstvu.

V tomto případě jsou třídy úložiště Apache Cassandra a PostreSQL z hlediska kódu prakticky nerozeznatelné a všechny rozdíly v implementaci řeší framework Spring.

V obou případech stačí definovat metodu v rozhraní a napsat přímý dotaz SQL nebo CQL do anotace `@Query` pro příslušnou metodu.

```
1 usage   pavelkireev
@Query("SELECT DISTINCT u.* " +
       "FROM subscription_request sr " +
       "JOIN users u ON u.jid = sr.user_jid_from OR u.jid = sr.user_jid_to " +
       "WHERE sr.status = 'ACCEPTED' AND :userJid IN (sr.user_jid_from, sr.user_jid_to) AND :userJid <> u.jid;")
Flux<User> findAllFriendsByUserId(String userJid);
```

Obrázek 13: Metoda `findAllFriendsByUserId` (zdroj vlastní)

Příklad na obrázku ukazuje metodu, která zkonstruuje dotaz do databáze PostgreSQL na seznam přátel uživatele a vrátí jej ve formě objektu Flux, přičemž přímé volání a zpracování přijatých dat je delegováno na servisní vrstvu.

Dotazy pro databázi Apache Cassandra vypadají podobně.

```
1 usage   pavelkireev
@Query(value = "select * " +
               "from private_messages " +
               "where participants_jid = :participantsJid " +
               "and date = :date")
Flux<PrivateMessage> findAllByJidToAndJidFromAndDate(String participantsJid, LocalDate date);
```

Obrázek 14: Metoda `findAllByJidToAndJidFromAndDate` (zdroj vlastní)

Obrázek ukazuje příklad sestavení dotazu na soukromé zprávy mezi dvěma uživateli pro zadané datum. V tomto případě metoda také vrací objekt Flux, jehož další volání a zpracování je delegováno na servisní vrstvu.

Databázové tabulky jsou v aplikaci Java definovány jako entity. Entity obsahují všechny atributy odpovídající sloupcům tabulky. Mezi entitami PostgreSQL a entitami Apache Cassandra je však zásadní rozdíl. PostgreSQL je relační databáze a v této aplikaci jsou všechny primární klíče definovány jako id s typem BIGINT, jehož analogem v Javě je typ Long. Například entita group, která je reprezentací tabulky groups v PostgreSQL (název group je rezervován pro potřeby databáze).

```java
@Entity
@Getter
@Setter
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Table(name = "groups")
public class Group extends AbstractEntity<Long> {

    private String mucJid;
    private String name;
    private String ownerJid;

}
```

Obrázek 15: Entita Group (zdroj vlastní)

Vzhledem k tomu, že některé atributy jsou společné pro všechny entity aplikace, byly začleněny do entity AbstractEntity, která je předkem všech PostgreSQL entit aplikace.

Tato třída obsahuje parametry entit, jako jsou:

- Primární klíč.
- Datum vytvoření.
- Datum poslední úpravy .

Primární klíč je označen pomocí anotace `@Id`, což je klíčový požadavek pro správnou práci s databází a správné fungování aplikace.

Čas vytvoření a čas poslední úpravy se vytvoří automaticky před uložením nové entity do databáze. Pro přípravu těchto parametrů před uložením entity do databáze byla vytvořena metoda prePersist, která je rovněž označena anotací @PrePersist, jejímž účelem je explicitně uvést, že tato metoda má být volána výhradně před uložením dat entity.

```java
6 usages   4 inheritors   pavelkireev
public abstract class AbstractEntity<P extends Serializable> implements Persistable<P>, Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    protected P id;

    2 usages
    private LocalDateTime createdAt = LocalDateTime.now();
    4 usages
    private LocalDateTime updatedAt = LocalDateTime.now();

    pavelkireev
    @PrePersist
    public void prePersist() {
        createdAt = LocalDateTime.now();
        updatedAt = LocalDateTime.now();
    }

    pavelkireev
    @PreUpdate
    public void onPreUpdate() { updatedAt = LocalDateTime.now(); }

    pavelkireev
    @Override
    @Transient
    public boolean isNew() { return null == getId(); }
```

Obrázek 16: Abstraktní entita (zdroj vlastní)

Vzhledem k tomu, že Apache Cassandra je noSQL databáze a je rozdělena podle klíče rozdělení, skládá se klíč každé entity z několika atributů a je samostatnou třídou.

Obrázek ukazuje jako příklad entitu reprezentující skupinové zprávy v databázi Apache Cassandra. Parametr sloužící jako dělený klíč v entitě, která představuje konkrétní tabulku Apache Cassandra, má anotaci `@PrimaryKey` a je primárním klíčem entity. Některé parametry jsou navíc označeny anotací `@Column`, protože v případě, že se název tabulky skládá z několika slov, je nutné v této anotaci explicitně uvést název sloupce, aby se parametry správně navázaly na sloupce tabulky. Celá třída je označena anotací `@Table` s názvem příslušné tabulky uvnitř, která explicitně označuje, že entita patří do konkrétní tabulky.

```java
@Table("group_messages")
public class GroupMessage {
    @PrimaryKey
    private GroupMessageKey key;
    @Column("jid_from")
    private String jidFrom;
    private String content;
    private String attachment;
    @Column("message_id")
    private String messageId;
```

Obrázek 17: Entita `GroupMessage` (zdroj vlastní)

Klíč je reprezentován třídou GroupMessageKey, která obsahuje klíč oddílu a klíč seskupení. Klíč oddílu je `JID` skupinového chatu. Všechny zprávy skupinového chatu budou uloženy v jednom oddílu a seřazeny podle klíče seskupení, který se skládá ze dvou atributů (datum a čas odeslání zprávy). Rozdělený klíč musí být anotován pomocí anotace `@PrimaryKeyColumn`, přičemž v parametrech je uvedeno pořadové číslo parametru, název příslušného sloupce tabulky a typ klíče, v rámci této aplikace bude typ vždy odpovídat příslušnému výčtu `PrimaryKeyType.PARTITIONED.`

Kromě klíče pro rozdělení obsahuje `GroupMessageKey` dva atributy, které definují klíč pro klastrování, podle kterého budou záznamy v tabulce v budoucnu seskupovány a tříděny.

U tohoto klíče také určíme jeho pořadové číslo, název odpovídajícího sloupce v tabulce, typ klíče `PrimaryKey.CLUSTERED` a výchozí pořadí řazení `Ordering.DESCENDING.`

```java
@PrimaryKeyClass
@NoArgsConstructor
@AllArgsConstructor
public class GroupMessageKey {
    @PrimaryKeyColumn(ordinal = 0, value = "muc_jid", type = PrimaryKeyType.PARTITIONED)
    private String mucJid;
    @PrimaryKeyColumn(ordinal = 1, value = "date", type = PrimaryKeyType.CLUSTERED, ordering = Ordering.DESCENDING)
    private LocalDate date;
    @PrimaryKeyColumn(ordinal = 2, value = "time", type = PrimaryKeyType.CLUSTERED, ordering = Ordering.DESCENDING)
    private LocalTime time;
}
```

Obrázek 18: Cassandra klíč GroupMessageKey (zdroj vlastní)

Vzhledem k tomu, že vyvíjená aplikace využívá reaktivní verzi frameworku Spring, mají i kontroléry své rozdíly oproti tradiční nereaktivní verzi. Stejně jako kontroléry v nereaktivní verzi frameworku Spring zpracovávají kontroléry v reaktivní verzi příchozí volání HTTP.

Hlavní rozdíl je v tom, že jejich metody mohou vracet objekty Flux a Mono. Přestože to není povinný požadavek, je to doporučený způsob implementace pro správné fungování reaktivního webového serveru. Příkladem takové implementace je metoda getCandidates v GroupChatController. V této metodě vrací servisní třída seznam všech uživatelů, kteří mohou být pozváni do skupinového chatu, jako objekt Flux, další zpracování je delegováno na framework Spring a závisí na konkrétním webovém serveru, ve kterém je aplikace spuštěna.

```
pavelkireev
@GetMapping("candidates")
public Flux<SubscribableUserDto> getCandidates(
    Authentication authentication
) {
    Jwt jwt = ((Jwt) authentication.getPrincipal());
    return userService.findAllFriendsByUserJid(jwt.getClaim("jid"));
}
```

Obrázek 19: Implementace metody `getCandidates` *(zdroj vlastní)*

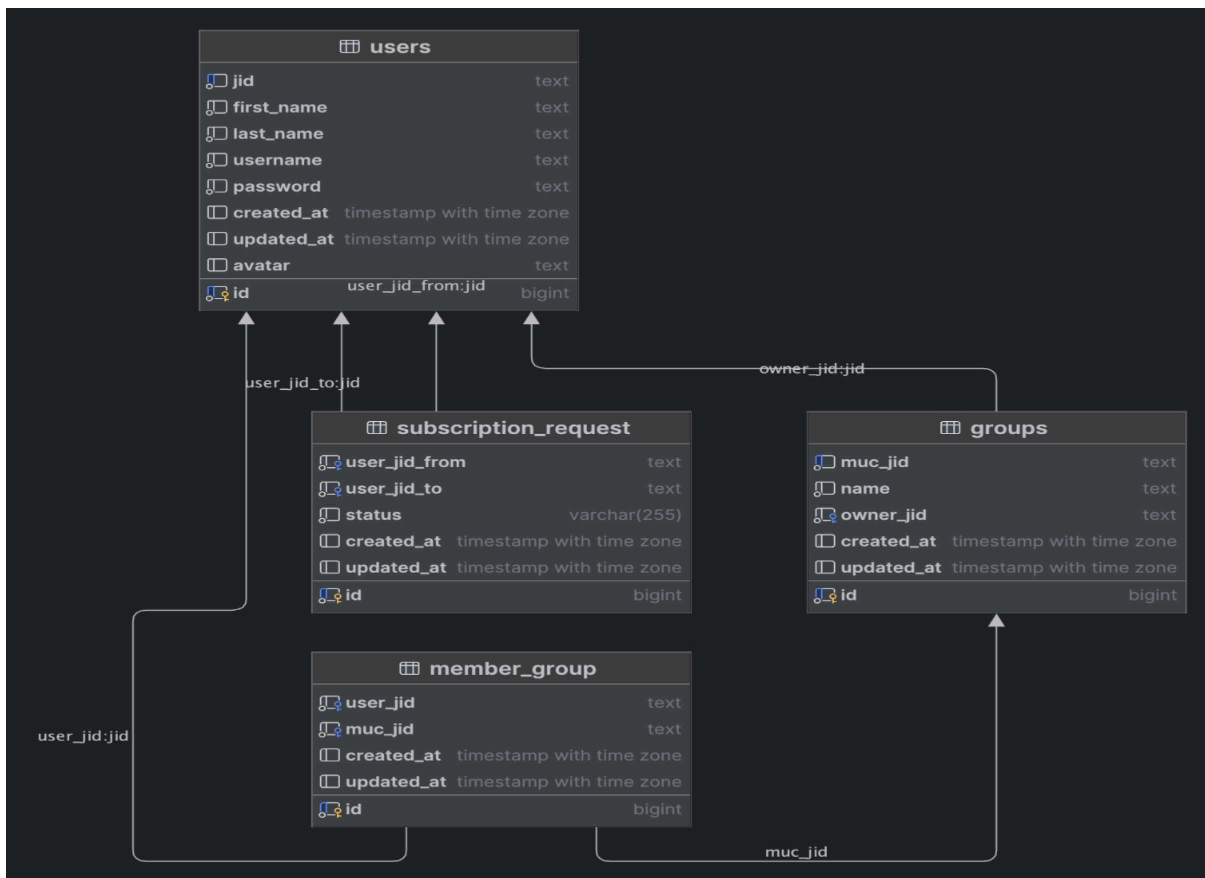
7.4.3 Konfigurace databáze

Ukládání dat ve vyvíjené aplikaci probíhá pomocí dvou různých databází. Data o uživatelích a skupinových chatech jsou uložena v relační databázi PostgreSQL. Historie zpráv soukromých a skupinových chatů je uložena v noSQL databázi Apache Cassandra.

Tato aplikace používá relační databázi PostgreSQL. Vytváření tabulek je implementováno přímo v aplikaci Java pomocí nástroje Flyway.

Pro aplikaci jsou definovány migrace, které se kontrolují při každém spuštění aplikace, a pokud jsou nalezeny nové nepoužité migrace, spustí se SQL skripty popsané v migračním souboru, které vytvoří tabulky nebo je upraví podle skriptu. Příklad takového skriptu se nachází v migračním souboru s názvem `V1_1681525852__Create_user_tables.sql.`

Po prvním spuštění aplikace se vytvoří všechny potřebné tabulky se všemi potřebnými relacemi. Schéma databáze PostgreSQL ve vyvíjené aplikaci je následující.



Obrázek 20: Schéma databáze PostgreSQL (zdroj vlastní)

Aplikace používá databázi Apache Cassandra, která nahrazuje standardní mechanismus ukládání zpráv Ejabberd. Jednou z výhod databáze Apache Cassandra je snadné vytvoření clusteru. Pro vyvíjenou aplikaci byl vytvořen lokální cluster Cassandra o třech uzlech v kontejnerech Docker. Cluster byl vytvořen ručně pomocí bash skriptu.

Tento skript určuje název kontejneru, lokální port, verzi Apache Cassandra a instance databáze, které mají být sdruženy do clusteru.

```
docker run --name xmpp_messenger_cassandra_node_1 -p 9042:9042 -d cassandra:latest
INSTANCE1=$(docker inspect --format="{{ .NetworkSettings.IPAddress }}" xmpp_messenger_cassandra_node_1)
echo "Instance 1: ${INSTANCE1}"

docker run --name xmpp_messenger_cassandra_node_2 -p 9043:9042 -d -e CASSANDRA_SEEDS=$INSTANCE1 cassandra:latest
INSTANCE2=$(docker inspect --format="{{ .NetworkSettings.IPAddress }}" xmpp_messenger_cassandra_node_2)
echo "Instance 2: ${INSTANCE2}"

docker run --name xmpp_messenger_cassandra_node_3 -p 9044:9042 -d -e CASSANDRA_SEEDS=$INSTANCE1,$INSTANCE2 cassandra:latest
INSTANCE3=$(docker inspect --format="{{ .NetworkSettings.IPAddress }}" xmpp_messenger_cassandra_node_3)
echo "Instance 3: ${INSTANCE3}"
```

Obrázek 21: Vytvořeni Docker clusteru pro Apache Cassandra (zdroj vlastní)

Jakmile jsou vytvořeny všechny potřebné instance Cassandry, stačí v jedné z nich vytvořit tabulky potřebné pro aplikaci, které budou replikovány i v ostatních uzlech.

Při vytváření tabulek Apache Cassandra je třeba kromě požadovaných sloupců zadat také primární klíč a klíč clusteru. zpráv uživatelů ve vyvinutém messengeru.

Po provedení všech nastavení a vytvoření tabulek je schéma databáze Apache Cassandra pro aplikaci následující.

Obrázek 22: Schéma databáze Apache Cassandra (zdroj vlastní)

Databáze messengerů je poměrně malá a skládá se pouze ze dvou tabulek:

- `private_messages` ukládá historii zpráv mezi dvěma uživateli s použitím řetězce vytvořeného z jejich JID jako klíče pro oddělení a času odeslání zprávy jako klíče pro seskupení.
- `group_messages` ukládá historii zpráv skupiny uživatelů s použitím muc_jid (identifikátoru místnosti) jako klíče rozdělení a času odeslání zprávy jako klíče seskupení.

Po vytvoření clusteru a potřebných tabulek bude Apache Cassandra automaticky provádět replikaci dat a správu clusteru podle zadané konfigurace.

7.4.4 Angular WebApp

Hlavním způsobem interakce koncového uživatele s aplikací je frontendová aplikace vytvořená pomocí frameworku Angular. Webová aplikace v Messengeru se skládá ze tří hlavních typů tříd.

Tabulka 5: Typy tříd (zdroj vlastní)

Typ třídy	**Popis**
Module	Konfigurační třídy pro definici jednotlivých modulů, jejich závislostí a obecných nastavení. Aplikace obsahuje jediný modul.
Service	Servisní vrstva, která spravuje komunikaci se servery Ejabberd a Java.
Component	Prezentace a logika jednotlivých stránek webové aplikace a jejich součástí

V aplikaci webového messengeru servisní vrstva plní následující funkce:

- Správa připojení k serveru Ejabberd
- Odesílání zpráv a oznámení o přítomnosti na server Ejabberd
- Zpracování příchozích zpráv a oznámení o přítomnosti
- Interakce s backendovým serverem Java
- Autentizace uživatelů
- Stahování a nahrávání souborů ze serveru

Dva hlavní komunikační kanály mezi webovou aplikací a ostatními moduly aplikace jsou spojení BOSH pro komunikaci se serverem Ejabberd a Rest API pro výměnu dat s aplikací Java.

Uživatel se musí nejprve přihlásit nebo zaregistrovat, poté aplikace obdrží od serveru Java token potřebný k autorizaci.

Metody pro autorizaci uživatelů jsou zahrnuty ve třídě `AuthService`.

```
login(loginForm: LoginFormDto): Observable<string> {
  const httpOptions = {
    headers: {
      Authorization: 'Basic ' + window.btoa( data: loginForm.username + ':' + loginForm.password)
    },
    responseType: 'text' as 'text',
  };
  return this.http.post( url: "api/auth/sign-in", loginForm , httpOptions);
}
```

Obrázek 23: Implementace metody `Login` (zdroj vlastní)

Metoda login asynchronně posílá požadavek HTTP na autorizaci pomocí metody Basic Auth, přičemž údaje zadané uživatelem v přihlašovacím formuláři jsou odeslány na server jako hlavička HTTP Authorization. Jako odpověď od serveru pak aplikace obdrží jedinečný token

JWT, který je uložen v session storage a bude přidáván do hlaviček Authorization při odesílání HTTP požadavků na backendový server. Tato metoda vrací objekt Observable, který obdrží token jako parametr při volání své metody `subscribe` v komponentě Angular.

Spojení BOSH je navázáno ihned po autorizaci uživatele voláním metody connect služby MessagingService. Současně s navázáním spojení jsou definovány zpracovatelé událostí, kteří zpracovávají zprávy Stanza přicházející ze serveru Ejabberd.

```
1+ usages  pavelkireev
public connect(): void {
    const username = String(sessionStorage.getItem( key: "jid"));
    const password = String(localStorage.getItem( key: 'password'));
    const url = "http://localhost:5280/bosh/";

    this.client = new BoshClient(username, password, url,  route: 'web');

    this.client.on( event: "error",  listener: (data) => this.handleError(data));
    this.client.on( event: "stanza",  listener: (data) => this.handleStanza(data));
    this.client.on( event: "online",  listener: (data) => this.handleOnline(data));
    this.client.on( event: "offline",  listener: (data) => this.handleOffline(data));

    this.client.connect();
    console.log(`XMPP connection: ${JSON.stringify(this.client)}`)
}
```

Obrázek 24: Implementace připojeni k Ejabberd serveru (zdroj vlastní)

V kódu aplikace messenger jsou před odesláním požadavku na připojení definovány metody obsluhy příchozích objektů Stanza. Příchozí objekty Stanza mohou být 4 typů, což vyžaduje čtyři různé metody obsluhy pro každý typ události.

- **Error:** Příchozí oznámení o chybách, které se vyskytují při komunikaci se serverem Ejabberd. V webové aplikaci messengeru uživatel tato chybová hlášení nevidí, jsou pouze zaznamenána v konzole prohlížeče a pokud je v případě chyby přerušeno spojení se serverem, aplikace se pokusí otevřít nové BOSH spojení.
- **Stanza:** Může obsahovat jak zprávy od ostatních uživatelů, tak různá oznámení ze serveru Ejabberd. Metoda obsluhy objekt analyzuje, a pokud se jedná o novou zprávu nebo oznámení o změně stavu uživatele ze seznamu kontaktů, přidá novou zprávu do příslušného chatu nebo změní stav uživatele v seznamech kontaktů.

- **Online:** Oznámení, že aplikace úspěšně navázala BOSH spojení se serverem Ejabberd.
- **Offline:** Oznámení, že spojení BOSH se serverem Ejabberd bylo uzavřeno z důvodu chyby nebo požadavku klienta.

Uživatel pak musí informovat svůj seznam kontaktů, že se přihlásil do sítě. K tomu aplikace okamžitě po navázání spojení se serverem odešle serveru Ejabberd zprávu Presence Stanza.

`MessagingService` také poskytuje metody pro odesílání zpráv do soukromých i skupinových chatů. Současně s odesláním zprávy přes spojení BOSH odešle aplikace tuto zprávu na server Java prostřednictvím rozhraní Rest API, aby byla uložena do databáze.

Kromě již popsaných servisních tříd existuje také třída `FileService`, která je zodpovědná za stahování a nahrávání avatarů a příloh ke zprávám.

Všechny soubory nahrané uživateli jsou uloženy v souborovém systému serveru a jsou rozděleny do dvou oddělení.

- **Attachment**: Do tohoto oddělení se přidávají všechny soubory uživatelů messengeru, které byly odeslány ve skupinových nebo soukromých zprávách.
- **Avatar**: Do tohoto oddělení jsou přidány všechny profilové obrázky uživatelů messengeru, které byly nahrány prostřednictvím stránky pro úpravu profilu.

Každému souboru je jako název přiřazen identifikátor UUID.

```
  upload(file: File, type: string): Observable<any> {
    const formData: FormData = new FormData();
    formData.append( name: 'file', file);
    return this.http.post( url: `api/file/upload?fileType=${type}`, formData,  options: {
      reportProgress: true,
      observe: 'events'
    })
  }

  1+ usages  pavelkireev
  download(filePath: string): Observable<any> {
    return this.http.get( url: `api/file/download?filePath=${filePath}`,  options: {responseType: 'blob'});
  }
}
```

Obrázek 25: Ukládáni souborů pomoci `FileService` (zdroj vlastní)

Komponenty frameworku Angular jsou reprezentací webových stránek nebo komponent webových stránek, se kterými uživatel interaguje. V Messengeru se každá komponenta skládá

ze tří samostatných souborů, z nichž každý odpovídá za určitý aspekt reprezentované stránky nebo její části.

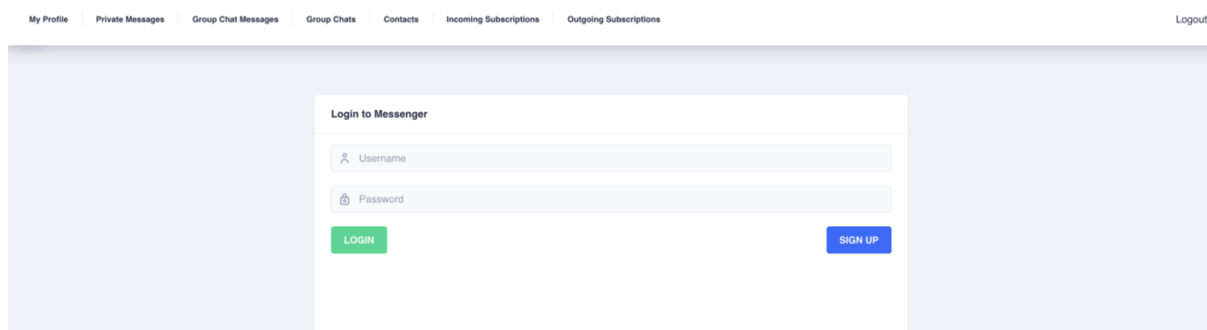
Tabulka 6: Typy souborů komponent (zdroj vlastní)

Typ souboru	**Popis**
TypeScript	Tento soubor popisuje logiku konstrukce stránky, logiku získávání a zpracování dat pro reprezentaci a zpracování událostí stránky.
HTML	Soubor, který definuje strukturu HTML stránky a vztah mezi komponentami HTML, daty a obslužnými funkcí souboru TypeScript.
SCSS	Definuje styly CSS, které se aplikují na prvky HTML komponenty.

Ve webové aplikaci messengeru jsou všechny komponenty umístěny v samostatném adresáři a mají výše popsanou strukturu.

Komponenta `SignInComponent` v aplikaci tvoří autorizační stránku uživatele a zpracovává na ní události uživatele.

Jedná se v podstatě o standardní formu přihlášení, ale její klíčová vlastnost je skryta uvnitř implementace, protože pro úspěšné přihlášení musí webová aplikace úspěšně navázat spojení se serverem Ejabberd pomocí metody `connect` z výše popsané třídy MessagingService.



Obrázek 26: Autorizační stránka (zdroj vlastní)

Pokud uživatel nemá účet v Messengeru, může se zaregistrovat kliknutím na tlačítko Sign Up. Po kliknutí na tlačítko Sign Up se bude vygenerována příslušná událost, která bude zpracována příslušnou obslužnou rutinou, a uživatel bude přesměrován na stránku pro vytvoření profilu.

Obrázek 27: Stránka pro vytvoření účtu (zdroj vlastní)

Na této stránce bude muset zadat klíčové údaje o svém profilu, všechny údaje kromě username lze později na stránce profilu změnit.

Pokud je profil úspěšně vytvořen, vytvoří se záznamy uživatele v interní databázi Ejabberdu a v databázi PostgreSQL a poté bude uživatel přesměrován zpět na přihlašovací stránku, kde se může přihlásit pomocí vytvořeného účtu. Ověřování se provádí jak na straně webové aplikace, tak na straně backendového serveru. Některá data se kontrolují dvakrát, aby se předešlo situacím, kdy uživatel úmyslně odešle formulář s neplatnými údaji, což by mohlo potenciálně ohrozit celou aplikaci. Webová aplikace se zabývá především validací formulářů, zatímco backendová aplikace kontroluje data vůči databázím, například při ukládání stavu objektu do databáze kontroluje zadaná data vůči již existujícím záznamům, které musí být unikátní.

Pokud již uživatel účet má nebo si ho právě zaregistroval, musí zadat své přihlašovací jméno a heslo. V případě, že účet neexistuje nebo jsou údaje zadány nesprávně, zobrazí se uživateli příslušná oznamovací zpráva.

Obrázek 28: Ověřování přihlašovacích údajů (zdroj vlastni)

Pokud jsou přihlašovací údaje správné, aplikace obdrží autorizační token z backendové aplikace Java, uloží jej do lokálního úložiště prohlížeče. V budoucnu bude uložený token odeslán v hlavičce každého HTTP požadavku na server Java. Po odhlášení bude tento token z úložiště vymazán a server při příští autorizaci uživatele vygeneruje nový token. Tato logika je implementována přímo v komponentě v metodě login.

Na obrázku níže je vidět, že aplikace vytvoří formulář na základě údajů zadaných uživatelem na přihlašovací stránce a předá tento formulář výše popsané metodě `login` ze třídy `AuthService`.

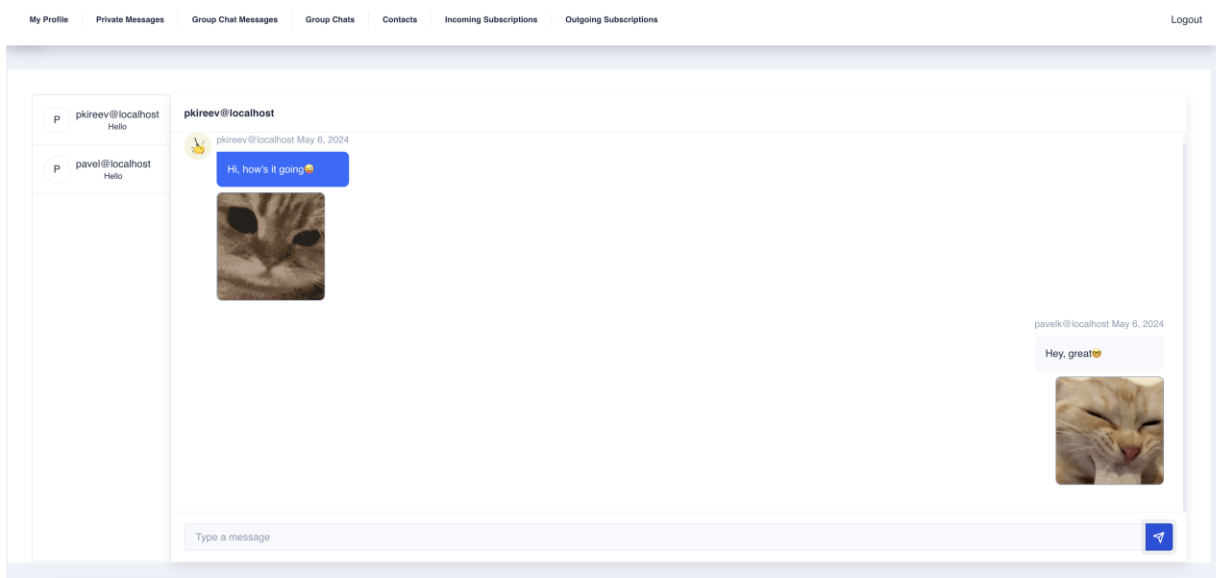
```
public login(loginForm: NgForm): void {
  const values = loginForm.value as any;
  this.showError = false;

  const loginFormDto: LoginFormDto = {
    username: values.username,
    password: values.password
  }
  sessionStorage.removeItem( key: "app.token");
  this.authService.login(loginFormDto)
                 .subscribe( observerOrNext: {
                   next: (token) => {
                     sessionStorage.setItem("app.token", token);
                     const decodedToken = jwtDecode<JwtPayload>(token);
                     if (typeof decodedToken.aud === "string") {
                       sessionStorage.setItem("app.roles", decodedToken.aud);
                     }
                     sessionStorage.setItem("jid", `${loginFormDto.username}@localhost`);
                     localStorage.setItem("jid", `${loginFormDto.username}@localhost`);
                     localStorage.setItem("password", loginFormDto.password);
                     this.messagingService.connect();
                     this.invalidLogin = false;
                     this.router.navigate( commands: ['/chat']);
                   },
                   error: (error) => {
                     this.invalidLogin = true;
                     this.snackBar.open( message: `Login failed: ${error.status}`,  action: "OK");
                   }
                 });
}
```

Obrázek 29: Logika metody autorizace uživatele (zdroj vlastní)

Pokud jsou data úspěšně odeslána a token je přijat jako parametr metody subscribe, token je dekódován a některá jeho data jsou uložena do localStorage a sessionStorage. Poté se vyvolá výše popsané spojení se serverem Ejabberd a uživatel je přesměrován na stránku se svými soukromými zprávami. Celá tato logika je popsána v obsluze „next“, v případě chyby při práci s objektem Observable se zavolá logika ošetření chyby z obsluhy „error“. V této metodě jde o vykreslení zprávy o nesprávně zadaných údajích na přihlašovací stránce a zápis chybových protokolů do konzoly prohlížeče.

Po úspěšné autorizaci bude uživatel přesměrován na stránku se svými soukromými chaty. Stránka se soukromými chaty je uvedena na obrázku níže.



Obrázek 30: Soukromý chat (zdroj vlastní)

Na této stránce si uživatel může zobrazit historii zpráv se svými kontakty, zjistit, kdo z jeho kontaktů je právě online, a posílat nové zprávy.

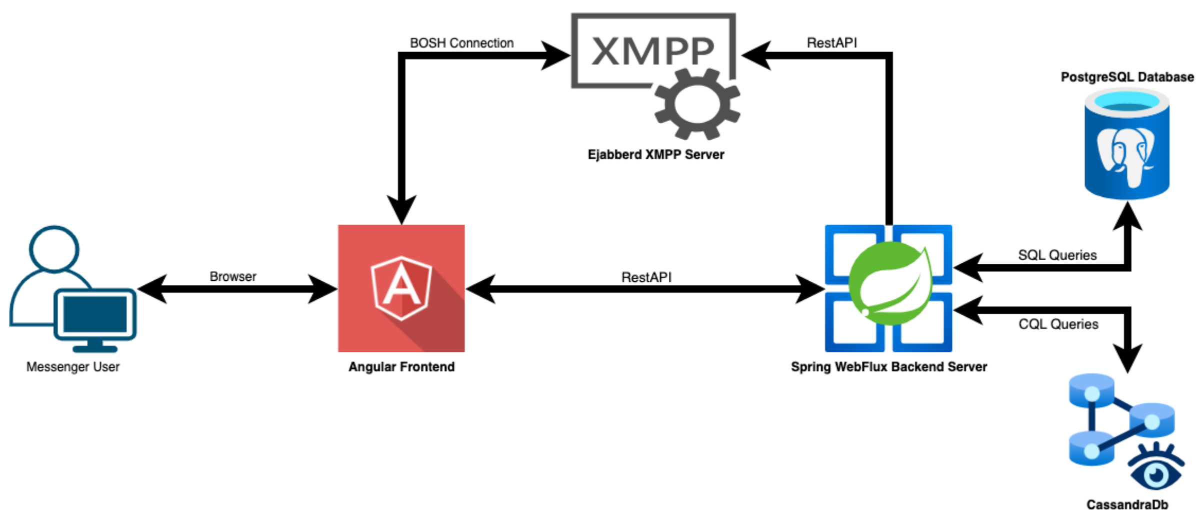
V horní části stránky se nachází navigační panel, který uživateli umožňuje přepínat mezi základními stránkami aplikace. Na navigačním panelu může uživatel přejít na tyto stránky:

- **My Profile:** Stránka pro prohlížení a upravování osobního profile.
- **Private Messages:** Stránka s kontakty uživatele a jeho soukromými chaty s nimi.
- **Group Chat Messages:** Stránka skupinových chatů uživatele.
- **Group Chats:** Stránka pro správu skupinových chatů patřících uživateli a vytváření nových skupinových chatů.

- **Contacts:** Zobrazení registrovaných uživatelů a odesílání žádostí o jejich přidání do seznamu kontaktů.
- **Incoming Subscriptions:** Příchozí žádosti o přidání do seznamu kontaktů, které může uživatel přijmout nebo odmítnout.
- **Outgoing Subscriptions:** Zobrazení požadavků na přidání do seznamu kontaktů odeslaných jiným uživatelům
- **Logout:** Odhlášení a přesměrování na přihlašovací stránku.

7.5 Architektura

Výše popsané aplikační moduly a jejich komunikaci můžeme shrnout do schématu na následujícím obrázku.



Obrázek 31: Architektura aplikace (zdroj vlastni)

Z této architektury můžeme odvodit následující hlavní body:

- Aplikace se skládá ze dvou implementovaných modulů: webové aplikace Angular a serverové aplikace Java Spring WebFlux.
- Komunikace mezi webovou aplikací a aplikací Java na straně serveru probíhá pomocí rozhraní RestAPI.
- Důležitou součástí aplikace je také server Ejabbard XMPP, který je nakonfigurován tak, aby mohl komunikovat s klientskou aplikací Angular a aplikací Spring WebFlux.
- Komunikace se serverem Ejabberd probíhá pomocí spojení BOSH v případě webové aplikace Angular a RestAPI v případě serverové aplikace Java.

- Aplikace využívá databáze Apache Cassandra a PostreSQL, které spravuje serverová Java aplikace.
- Komunikace backendové aplikace s PostgreSQL probíhá pomocí dotazů SQL a s databází Apache Cassandra pomocí dotazů CQL.

ZÁVĚR

Implementace vlastního messengeru ukázala, že softwarové řešení pro zasílání zpráv pro firemní účely má určité výhody oproti použití univerzálních hotových řešení jako jsou například Slack, Discord, Zoom chat, Signal, apod. Především poskytuje větší flexibilitu při změně nastavení a funkcí aplikace podle požadavků konkrétního podniku. Jednou z hlavních překážek při vytváření vlastního messengeru v rámci firmy je složitost implementace a vysoké časové a finanční náklady. Vytvořená implementace messengeru využívá řadu hotových řešení a open source knihoven, které umožňují výrazně snížit náklady na jeho vytvoření a podporu. Aplikace je postavena na serveru XMPP Ejabberd, který i v základní konfiguraci poskytuje všechny funkce potřebné pro většinu podniků. Ejabberd má mimo jiné širokou komunitu, která aktivně spolupracuje, aby si vzájemně pomohla najít nejlepší řešení při implementaci aplikace založené na Ejabberdu. Reaktivní backend aplikace umožňuje výrazně snížit zatížení aplikace při nárůstu počtu uživatelů, což je také kritický aspekt při výběru řešení pro podnikání, který umožňuje mít určitou rezervu v případě růstu počtu zaměstnanců.

Během vývoje aplikace se také objevily některé potíže. Protokol XMPP je poměrně starý a přenáší data ve formátu XML, který vyžaduje implementaci dodatečné logiky zpracování zpráv na straně klienta aplikace, na rozdíl od vhodnějšího formátu JSON. Kromě toho je backendová část aplikace založena na reaktivní verzi frameworku Spring, která je zcela odlišná od tradiční verze a vyžaduje odlišný přístup při implementaci aplikace, což vývojáři, který nemá s reaktivními aplikacemi zkušenosti, ztěžuje práci.

Existují také určité možnosti zlepšení výkonu a rozšíření aplikace. Především je to podpora několika domén pro flexibilnější nastavení jednotlivých uzlů aplikace. Ejabberd také obsahuje potřebné funkce pro realizaci skupinových a soukromých hovorů a videohovorů, které lze při rozšiřování aplikace také implementovat. Mimo jiné existuje řada standardních funkcí moderního messengeru, které ve vyvinuté aplikaci chybí, například výměna audio a video zpráv.

POUŽITÁ LITERATURA

[1] SCHILDT, Herbert. Java: the complete reference. Eleventh edition. New York: McGraw-Hill Education, [2019]. ISBN 9781260440232.

[2] SAINT-ANDRÉ, Peter, Kevin T. SMITH a Remko TRONÇON. XMPP: the definitive guide : building real-time applications with Jabber technologies. [Sebastopol, CA]: O'Reilly, c2009. ISBN 059652126x.

[3] Pro spring 5: an in-depth guide to the spring framework and its tools. New York, NY: Springer Science+Business Media, 2017. ISBN 9781484228074.

[4] Reactive Systems in Java. In *baeldung.com* [online]. [cit. 2024-05-12]. Dostupné z: https://www.baeldung.com/java-reactive-systems

[5] The Reactive Manifesto. In *reactivemanifesto.org* [online]. [cit. 2024-05-12]. Dostupné z: https://www.reactivemanifesto.org/

[6] Spring WebFlux. In *docs.spring.io* [online]. [cit. 2024-05-12]. Dostupné z: https://docs.spring.io/spring-framework/reference/web/webflux.html

[7] What is Angular? In *angular.io* [online]. [cit. 2024-05-12]. Dostupné z: https://angular.io/guide/what-is-angular

[8] Introduction to Angular concepts. In *angular.io* [online]. [cit. 2024-05-12]. Dostupné z: https://angular.io/guide/architecture

[9] The Apache Cassandra Beginner Tutorial. In *freecodecamp.org* [online]. [cit. 2024-05-12]. Dostupne z: https://www.freecodecamp.org/news/the-apache-cassandra-beginner-tutorial/

[10] What is Docker? In *ibm.com* [online]. [cit. 2024-05-12]. Dostupné z: https://www.ibm.com/topics/docker

[11] Mono. In *projectreactor.io* [online]. [cit. 2024-05-12]. Dostupné z: https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html

[12] Flux. In *projectreactor.io* [online]. [cit. 2024-05-12]. Dostupné z: https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html

[13] Cassandra Documentation. In *cassandra.apache.org* [online]. [cit. 2024-05-12]. Dostupné z: https://cassandra.apache.org/doc/stable/index.html

[14] Cassandra Data distribution and replication. In *docs.datastax.com* [online].

[cit. 2024-05-12]. Dostupne z: https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/architecture/archDataDistributeAbout.html

[15] About PostgreSQL. In *postgresql.org* [online]. [cit. 2024-05-12]. Dostupné z: https://www.postgresql.org/about/

[16] Ejabberd Docs. In *docs.ejabberd.im* [online]. [cit. 2024-05-12]. Dostupné z: https://docs.ejabberd.im/get-started/

[17] An overview of XMPP. In *xmpp.org* [online]. [cit. 2024-05-12]. Dostupné z: https://xmpp.org/about/technology-overview/

[18] Extensible Messaging and Presence Protocol (XMPP): Core. In xmpp.org [online]. [cit. 2024-05-12]. Dostupné z: https://xmpp.org/rfcs/rfc6120.html

[19] PostgreSQL - Systém Architecture. In *geeksforgeeks.org* [online]. [cit. 2024-05-12]. Dostupné z: https://www.geeksforgeeks.org/postgresql-system-architecture/

[20] PostgreSQL Advantages and Disadvantages. In *aalpha.net* [online]. [cit. 2024-05-12]. Dostupné z: https://www.aalpha.net/blog/pros-and-cons-of-using-postgresql-for-application-development/

[21] PostgreSQL 16.3 Documentation. In *postgresql.org* [online]. [cit. 2024-05-12]. Dostupné z: https://www.postgresql.org/docs/16/index.html

[22] PostgreSQL Functions. In *javatpoint.com* [online]. [cit. 2024-05-12]. Dostupné z: https://www.javatpoint.com/postgresql-functions

[23] PostgreSQL Triggers. In *tutorialspoint.com* [online]. [cit. 2024-05-12]. Dostupné z: https://www.tutorialspoint.com/postgresql/postgresql_triggers.htm

[24] FOWLER, Martin. Patterns of enterprise application architecture. Boston: Addison-Wesley, c2003. ISBN 0321127420.

PŘÍLOHY

Příloha A – Messenger.zip 81

PŘÍLOHA A – MESSENGER.ZIP

Messenger.zip obsahuje:

1. Zdrojový kód serverové aplikace
2. Zdrojový kód webové aplikace
3. Konfigurační soubor serveru Ejabberd
4. Instrukce pro počáteční konfiguraci a spuštění aplikace