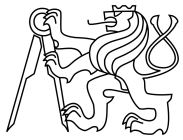
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Systém pro správu zákaznické podpory
Student:	Bc. Jiří Kocourek
Vedoucí:	Ing. Jan Herzán
Studijní program:	Informatika
Studijní obor:	Počítačové systémy a sítě
Katedra:	Katedra počítačových systémů
Platnost zadání:	Do konce letního semestru 2016/17

Pokyny pro vypracování

Navrhněte, implementujte a nasaďte informační systém pro zvýšení efektivity a přehlednosti zpracování příchozích e-mailových požadavků na zákaznickou podporu společnosti. Systém bude seskupovat související e-maily do případů a umožní případy přiřazovat zaměstnancům či skupinám zaměstnanců. Bude evidovat stav řešení, podporovat přidávání poznámek a příloh, využití štítků a šablon odpovědí. Serverová část systému musí zprostředkovávat svoji funkcionalitu prostřednictvím webových služeb, aby bylo možno k systému přistupovat jednotně pomocí různých klientských aplikací (např. webový klient, desktopová aplikace, mobilní aplikace). Předmětem zadání je vytvoření serverové části a desktopové aplikace. Podstatnou součástí práce bude nasazení serverové části aplikace tak, aby byla zajištěna vysoká dostupnost služby (replikace dat, klastrování).

Seznam odborné literatury

Dodá vedoucí práce.

L.S.

prof. Ing. Róbert Lórencz, CSc.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 18. února 2016

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra počítačových systémů

Diplomová práce

Systém pro správu zákaznické podpory

Bc. Jiří Kocourek

Vedoucí práce: Ing. Jan Herzán

10. května 2016

Poděkování

Děkuji vedoucímu práce Ing. Janu Herzánovi za umožnění vzniku této práce a vznesené podněty. Děkuji své rodině za podporu při procesu tvorby práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. Dále prohlašuji, že jsem s Českým vysokým učením technickým v Praze uzavřel dohodu, na základě níž se ČVUT vzdalo práva na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona. Tato skutečnost nemá vliv na ust. § 47b zákona č. 111/1998 Sb., o vysokých školách, ve znění pozdějších předpisů.

V Praze dne 10. května 2016 .....

Odkaz na tuto práci

Kocourek, Jiří. *Systém pro správu zákaznické podpory.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstrakt

Tato práce se zabývá návrhem a vytvořením informačního systému pro zpracování vznesených požadavků na zákaznickou podporu společnosti prostřednictvím e-mailu. Obsahem textu je analýza požadavků, návrh architektury a popis implementace serverové části systému za použití technologií Java a Spring, stejně jako grafické klientské aplikace. Práce navrhuje komunikační rozhraní za pomocí webových služeb typu REST, respektující všechny požadavky tohoto architektonického stylu. Dále jsou diskutovány možnosti nasazení aplikace za využití technologií pro zajištění vysoké dostupnosti služby a připraveny plány nasazení zajišťující nepřerušené poskytování služeb za situace výpadku libovolného serveru systému.

Klíčová slova zákaznická podpora, REST, HATEOAS, Spring, vysoká dostupnost.

Abstract

In this master thesis, we describe design and implementation of an information system, that manages e-mail requests sent co company's customer support line. Content of this text is made up of requirement analysis, architecture design and implementation description of system server module using Java and

Spring technology, as well as the graphical client application. The thesis designs a communication interface using REST web services, with respect to all requirements of this architectural style. The discussion of deployment possibilities, using high availability technology, follows and plans for deployment with regard to uninterrupted service availability in case of any server failure are prepared.

Keywords customer support, REST, HATEOAS, Spring, high availability.

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Téměř každá společnost, nabízející své zboží nebo služby přes internet, v dnešní době poskytuje svým zákazníkům podporu prostřednictvím e-mailu. S přibývajícím množstvím požadavků na zákaznickou podporu je potřeba tyto požadavky evidovat a kategorizovat nad rámec schopností obvyklých e-mailových aplikací. Práce si klade za cíl analyzovat stávající systémy a služby, které danou funkcionalitu nabízejí, identifikovat požadavky subjektů z kategorie malých a středních firem a posoudit vhodnost těchto služeb pro předmětné společnosti. Na základě uskutečněných zjištění je proveden návrh informačního systému zpracovávajícího příchozí e-mailové požadavky. Systém umožňuje jednoznačné rozdělení práce pomocí přiřazování požadavků zaměstnancům a sledování postupu vyřízení, čímž usnadňuje zaměstnancům rychlé posouzení zákazníkova problému a vytvoření adekvátní odpovědi za pomocí rychlého přehledu všech minulých požadavků daného zákazníka a šablon odpovědí. Vzhledem k nákladům, potřebným k provozování takovéhoto systému, počítá architektura aplikace s možností jejího provozování nezávislým provozovatelem pro mnoho společností a systém tak může být nabízen jako služba. Tento návrh je v souladu se současnými trendy, kdy roste obliba využití modelu nasazení *Software as a Service (SaaS)*, při kterém je systém hostován provozovatelem služby, namísto jeho nasazení konkrétní společností pouze pro vlastní účely.

Následně je implementována serverová část systému, zajišťující aplikační logiku, uložení dat a poskytující rozhraní pro přístup k datům klientským aplikacím. Z důvodu specifických požadavků různých společností, stejně jako vzrůstající potřeby přístupu k informacím i mimo kancelářské prostory, návrh serveru počítá s možností vytvořit různé klientské aplikace, od klasického počítačového programu, nabízejícího optimální odezvy a možnosti integrace s ostatními podnikovými systémy, přes webovou aplikaci, skýtající nejširší možnosti přístupu k systému v různých situacích, až po mobilní aplikaci, poskytující data i pracovníkům na cestách. Z důvodu zmíněné rozmanitosti jsou zvoleným rozhraním webové služby, eliminující závislost na použité technologii a platformě a umožňující snadnou komunikaci všem zařízením s podporou zá-

kladních internetových protokolů. Dále je provedena implementace multiplatformní desktopové grafické klientské aplikace, umožňující uživatelům přístup a manipulaci s daty v informačním systému. Ostatní klientské aplikace nejsou pokryty rozsahem této práce.

Jedním z klíčových parametrů zákaznické podpory je bezpochyby rychlost vyřízení vzneseného požadavku. Toho si jsou společnosti vědomy a svěří-li svá data do systému pro správu zákaznické podpory, požadují záruky, aby systém byl odolný proti hardwarovým, softwarovým i síťovým chybám a byl dostupný za všech okolností. Z tohoto důvodu práce obsahuje plány nasazení pro zajištění vysoké dostupnosti služby a minimalizace rizika ztráty dat v případě havárie, nicméně v rámci zaměření na malé a střední podniky, a tedy nutnosti udržet akceptovatelnou cenu provozování systému, přichází s řešeními na zajištění vysoké dostupnosti i s limitovaným počtem serverů a zdrojů. Dále mapuje nutnou konfiguraci serveru s ohledem na množství připojených klientů a uložených souborů, konfiguraci replikované databáze a nasazení distribuovaného souborového systému.

Kapitola 1

Rešerše

1.1 Funkční požadavky

Z nejčastěji uváděných potřeb společností byly identifikovány tyto základní funkční požadavky na informační systém pro správu e-mailové zákaznické podpory:

- Přiřazování požadavků zaměstnancům a oddělením
  - Společnosti, jejichž tým zákaznické podpory čítá více osob, řeší problém rozdělení příchozích požadavků jednotlivým členům. Problém je nejlépe patrný při vyřizování centrálních e-mailových schránek pro zákaznickou podporu (např. info@doména, podpora@doména apod.) skupinou zaměstnanců, kdy je potřeba zajistit, aby jeden požadavek nebyl vyřizován více pracovníky, nebo naopak nebyl přehlédnut, a to bez nutnosti explicitní domluvy mezi pracovníky. Systém musí zajistit jednoznačné a zpětně dohledatelné přiřazení odpovědnosti za požadavek konkrétnímu zaměstnanci.
- Mechanismus vnitrofiremního předávání požadavků
  - Vedle přiřazování požadavků konkrétnímu zaměstnanci v rámci týmu je potřeba zajistit mechanismus předávání požadavků v rámci firmy, ať už jinému oddělení, bez ohledu na specifického zpracovatele, nebo konkrétnímu zaměstnanci. Zde je vhodné zajistit podporu pro skupiny uživatelů a výchozí skupinu pro jednotlivé e-mailové schránky. Uživatel systému má jednoznačně odlišit, která agenda je přiřazena konkrétně jemu, a která skupině, do které náleží.
- Širší vnitrofiremní komunikace
  - Platforma by vedle správy zákaznických požadavků měla umožňovat i vytváření interních požadavků jednotlivými zaměstnanci,

např. pro přiřazování práce podřízeným či kolegům a sledování postupu řešení. Cílem by mělo být odstranit potřebu využití e-mailu pro vnitrofiremní komunikaci.

- Centralizovaná správa různých e-mailových schránek
  - Pro vyhodnocení aktuálního stavu zákaznické podpory je vhodné, aby existoval centrální přehled všech vznesených požadavků, bez ohledu na adresu, na kterou byly zaslány. Pokud společnost provozuje mnoho webových stránek s různými kontaktními údaji, je centrální bod nutností i pro efektivní vyřizování korespondence. Přístup k jednotlivým schránkám musí být podmíněn dostatečnou úrovní oprávnění uživatele.
- Přehled stavu všech požadavků
  - Systém musí monitorovat stav požadavku, tedy odlišovat situace, kdy je požadavek nově přijatý, přiřazený zpracovateli či vyřízený, včetně historie vývoje tohoto stavu. Tyto údaje může být vhodné využít k auditování a vyhodnocení úspěšnosti a rychlosti odezvy na vznesené dotazy, efektivity zaměstnanců či oddělení.
- Přidávání interních komentářů a příloh
  - Systém má podporovat přidávání komentářů zaměstnanců k jednotlivým případům, s jasným označením kdo a kdy jej vytvořil a čeho se týká. Musí být zabráněno možnosti, aby se interní komentář dostal do odpovědi zákazníkovi. Při předávání požadavku v rámci firmy je také požadována možnost připojit k případu přiložené soubory, ať už k odeslání zákazníkovi, či pouze pro firemního zpracovatele.
- Šablony odpovědí a podpisy zaměstnanců
  - Vhodně využité šablony zpráv dokáží výrazně urychlit vyřizování nejčastějších požadavků. Podobně předdefinovaný podpis eliminuje duplicitní práci a riziko uvedení nesprávného údaje.
- Zachování osobního a individuálního přístupu k zákazníkovi
  - Je důležité zajistit, aby informační systém nevynucoval jiný přístup k zákazníkovi, než na který jsou obě strany komunikace zvyklé. Toto je problém mnoha tiketovacích systémů známých převážně subjektům působících v oblasti informačních technologií a jiných technických oborů. Zákazník nemá být vystaven číslům tiketu, nadbytečným automatizovaným zprávám, nerelevantním informacím

o předávání požadavku a internímu stavu vyřízení. Zároveň je potřeba poskytnout zpracovateli přehled historie požadavků konkrétního zákazníka, aby byla zajištěna individualita odpovědi.

- Integrace s CRM systémy a dalšími podnikovými aplikacemi
  - Obvyklým požadavkem společností, které systém pro řízení vztahů se zákazníky využívají, je integrace s tímto systémem, minimálně v oblasti přenosu informací o zákazníkovi mezi systémy. Je žádoucí poskytnout rozhraní pro import a export detailů o zákaznících v některém z běžně používaných textových formátů. Těsnější integraci je vhodné řešit individuálně dle provozovaného podnikového systému.
- Uživatelská přívětivost a jednoduchost zavedení systému
  - Uživatelé dosahují nejvyšší efektivity práce, mají-li k dispozici povědomé intuitivní prostředí, nejsou rozptylováni dlouhými odezvami aplikace, mohou používat rychlé ovládací prvky, např. připojení či uložení přílohy tažením myši (Drag and Drop), fulltextové vyhledávání a jiné. Před administrátory, zodpovědné za konfiguraci systému, nesmí být kladeny překážky, vyžadující časově náročné nastavování a ladění parametrů a individualizace nasazeného řešení.

1.2 Nefunkční požadavky

Na systém jsou kladeny následující nefunkční požadavky:

- Dostupnost aplikace přes web i desktopovou klientskou aplikaci
  - Uživateli musí umožněn přístup k systému za každé situace a bez zbytečné instalace. K tomu je vhodná webová aplikace, prostřednictvím které se dostane ke všem údajům systému z široké škály zařízení. Pro maximalizaci efektivity práce má být také dostupná desktopová klientská aplikace, která dosahuje nízké doby odezvy, vysokého výkonu a rychlosti zpracování požadavků z kancelářských strojů společnosti, není vázána na webový prohlížeč a z něj vyplývající omezení. Aplikace je lépe integrována do operačního systému a může být propojena s ostatními systémy společnosti.
- Zajištění vysoké dostupnosti v případě havárie
  - Systém musí být odolný proti výpadku libovolného serveru ze své infrastruktury. Uživatelům musí být poskytnuta možnost nerušené práce i v případě selhání některé komponenty systému.

- Bezpečnost
  - Veškerá komunikace mezi klientskou aplikací a servery systému musí probíhat prostřednictvím zabezpečeného spojení. Vysoké bezpečnostní standardy musí být aplikovány i na servery aplikace. Hesla v databázi nesmí být uložena v podobě, z níž je možné zjistit původní znění hesla či toto znění v akceptovatelném čase nalézt útokem hrubou silou. Uživateli musí být zamezeno získat zdroje nedostupné pro jeho úroveň oprávnění, stejně jako zdroje jiných uživatelů.
- Rozšiřitelnost systému
  - Systém je navrhován tak, aby bylo možné v budoucnu přidat či modifikovat jeho funkcionalitu. Vhodný návrh objektového modelu eliminuje provázanost jednotlivých komponent. Kód využívá technik programování proti rozhraním pro oddělení jednotlivých vrstev aplikace.

1.3 Existující řešení

Tržně používaná řešení se rozdělují na tři hlavní směry – klasické e-mailové klienty, služby zaměřené na zákaznickou podporu a komplexní CRM (Customer relationship management) systémy.

1.3.1 E-mailový klient

Mnoho menších společností spoléhá pouze na funkcionalitu klasického e-mailu a e-mailových klientů, ať už webových rozhraní či aplikací typu Microsoft Outlook nebo Mozilla Thunderbird. Tato řešení, ač mohou nabízet mnoho funkcionalit pro tvorbu individuálních e-mailů, šablony e-mailů, podpisy, kategorizace zpráv či centrální přehled více schránek, nesplňují požadavky v bodě rozdělování zpráv mezi více osob. Spolupracuje-li více zaměstnanců na vyřizování požadavků z jedné schránky, je problematické zabránit vícenásobnému zpracování stejného požadavku, hrozí riziko opomenutí některé zprávy či její nechtěné smazání či přesunutí, nejedná se tedy o vhodné nástroje pro týmovou spolupráci. Tyto problémy vyvstávají nejen z nutnosti precizní domluvy na rozdělení práce, ale v případě desktopových aplikací, vytvářejících lokální kopie zpráv, i z technických důvodů způsobujících porušení synchronicity lokálních schránek u jednotlivých zaměstnanců. Předávání požadavku probíhá pomocí přeposílání zprávy bez přenesení jednoznačné odpovědnosti a interní komentáře se objevují v textu samotné předávané zprávy, vedoucí k riziku jejich inkluze v odpovědi zákazníkovi. Podobně obtížný je monitoring doby potřebné k odpovědi či počet vyřízených případů na zaměstnance. Subjekty do-

sud využívající klasické e-mailové aplikace by ze zavedení systému pro správu zákaznické podpory zaznamenaly největší přínosy.

1.3.2 Služby zaměřené na zákaznickou podporu

Na trhu existuje několik cloudově zaměřených služeb, nabízejících řešení pro zákaznickou podporu. Mezi nejvýznamnější poskytovatele se řadí Desk.com společnosti Salesforce, dále firmy Freshdesk a Zendesk. Společnost Zendesk od svého založení v roce 2006 do třetího čtvrtletí roku 2015 se svým stejnojmenným produktem získala 64 tisíc platících zákazníků, včetně firmy Uber a Airbnb, a vybudovala si tak významné postavení v kategorii malých a středních podniků [1]. Desk.com je reakcí na tento úspěch od významného dodavatele informačních systémů Salesforce, jehož doménou jsou jinak spíše podniky z kategorie velkých firem, na které v této oblasti míří produktem Salesforce Sales Cloud, spadajícím mezi komplexní CRM systémy. Desk.com je oproti tomu spíše služba pro dynamičtější podniky, zaměřená specifičtěji na zákaznickou podporu [2].

Všechny tyto služby nabízejí výrazně širší pokrytí odlišných kanálů pro zákaznickou podporu, kromě e-mailu například sociální sítě Facebook a Twitter, chat nebo telefonní linku. Poskytují možnosti dělení práce mezi zaměstnance, přidávání komentářů, využití šablon a monitoring stavu požadavků. Dále nabízejí mnoho doplňkových analytických funkcí, zákaznické portály, znalostní báze, komunitní fóra a mnoho dalšího. Problémem těchto systémů je, kromě delší iniciální konfigurace a nutnosti zákazníka přizpůsobit se mechanismům použití, omezená podpora pro vnitrofiremní komunikaci. Uživateli systému jsou tzv. agenti, což jsou operátoři vyřizující požadavky zákaznické podpory. Pro předání požadavku jiným zaměstnancům společnosti, kteří nejsou agenty, je potřeba jej odeslat prostřednictvím jiného kanálu. Systém zajišťující kompletní vnitrofiremní komunikaci by tak měl významnou tržní výhodu. Většina těchto poskytovatelů nabízí pouze webový přístup a mobilní aplikace, desktopové aplikace zpravidla chybí. U webových aplikací se však projevuje problém s dobou odezvy a pohodlností práce s přílohami. Také platební schémata, odvíjející se od počtu agentů v systému, mohou být pro některé společnosti odrazující.

1.3.3 Komplexní CRM systémy

Každý z velkých poskytovatelů CRM systémů, jako je SAP, Microsoft, Oracle nebo Salesforce nabízí jako součást svých systémů modul pro správu zákaznické podpory. Překážky pro zavedení těchto systémů v kategorii malých a středních podniků jsou obvykle vysoké, finanční hledisko činí tyto systémy nedosažitelnými a i v případě zavedení systému není adopce daného modulu společností a splnění uvedených požadavků zdaleka jisté. V případě úspěšného nasazení mohou být zabudované nástroje pro správu zákaznické podpory uspo-

kojivé. Dále vyvíjený systém však nemá být konkurencí těchto modulů velkých CRM systémů.

Z provedené analýzy je patrné, že je na trhu mnoho připravených řešení, v konkrétním ohledu funkcionálně přesahujících potřeby typického zákazníka, ovšem každá skupina identifikovaných požadavků je pokryta jiným typem systému a jejich průnik nikdy není stoprocentní.

Kapitola 2

Návrh

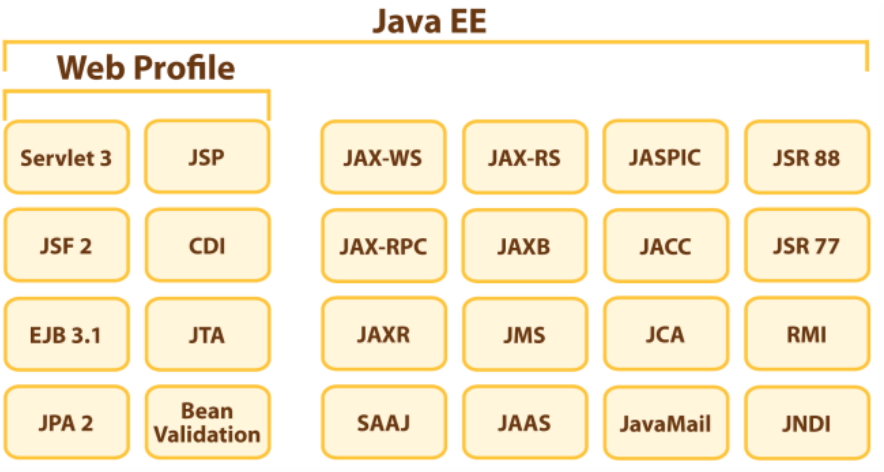
2.1 Obecné zásady

Z cílové skupiny zákazníků a nefunkčních požadavků na aplikaci vyplývají obecné zásady, které je nutné při návrhu serverové části aplikace zohledňovat. Jedná se zejména o nezávislost komunikačních protokolů a přenosových formátů na konkrétní použité technologii, eliminace využití proprietárního software a snaha o užití bezplatných řešení či aplikací s otevřeným zdrojovým kódem (open-source). Z požadavku na vysokou dostupnost aplikace vyplývá výhoda návrhu bezstavového serveru, tedy takového, který nebude uchovávat žádné informace o přihlášených klientech mezi jejich jednotlivými požadavky. Toto řešení zjednodušuje realizaci odolnosti aplikace při chybě a eliminuje potřebu replikovat tyto informace mezi více serverů při zachování úplného odstínění uživatele v případě havárie. Bezstavovost aplikace také podporuje jednoduchost škálování aplikace pro využití velkým množstvím současně připojených uživatelů. Pro zamezení vazby na konkrétního dodavatele softwarových technologií a prostředí pro nasazení je aplikace vytvářena jako multiplatformní.

2.2 Použité technologie

2.2.1 Java

Vývoj aplikace je proveden v programovacím jazyce Java. Jedná se o silně typovaný objektově orientovaný jazyk vhodný pro vývoj velkých podnikových aplikací. Jazyk splňuje vznesené požadavky na nezávislost na platformě díky implementaci běhového prostředí pro nejrůznější operační systémy. Využitá verze jazyka je *Java EE 7 Full Platform*. Java EE (*Java Platform, Enterprise Edition*) je podnikovou variantou Java platformy, která na rámec standardní edice přináší specifikace užitečné při vývoji podnikových aplikací a informačních systémů.



Obrázek 2.1: Srovnání Java EE 6 Web Profile a Full Platform [3]

2.2.2 Aplikační server

Běhovým prostředím pro aplikace vytvořené pro platformu Java EE je aplikační server. Pro tuto implementaci je zvolen aplikační server společnosti Oracle GlassFish 4. Jedná se o referenční implementaci platformy Java EE, přinášející implementaci nových standardů jako první a demonstrativní řešení pro ostatní aplikační servery. Na rozdíl od často používaného serveru Apache Tomcat, který je pouze HTTP serverem a servlet kontejnerem (tedy prostředím pro běh servletů, zpracovávajících HTTP požadavky), tedy realizací specifikace *Java EE Web Profile*, má GlassFish plnou podporu specifikace *Java EE Full Platform* a přináší tak všechny technologie dostupné této platformě.

2.2.3 Spring Framework

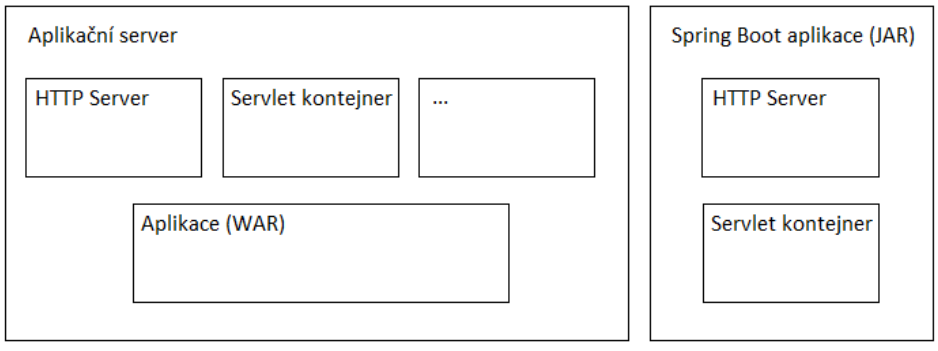
Spring Framework je aplikačním rámcem určeným k usnadnění vývoje enterprise aplikací v jazyce Java. Zaměřuje se na eliminaci nutnosti programově zajišťovat vzájemné závislosti jednotlivých komponent za pomocí vkládání závislostí (Dependency Injection), čímž odstraňuje mnoho propojovacího kódu a umožňuje preciznější oddělení jednotlivých částí aplikace a jednodušší testovatelnost. Nabízí tak komplexní, kontejnerem spravovanou a centrálně konfigurovatelnou infrastrukturu, umožňující vývojářům zaměřit se na obchodní logiku aplikace. Také poskytuje mnoho jednoduše použitelných řešení častých požadavků ve webových a jiných podnikových aplikacích. Využití frameworku usnadňuje škálovatelnost aplikace v případě potřeby nasazení více serverů.

2.2.4 Spring IO Platform

Spring IO Platform je jednotnou harmonizovanou platformou zastřešující moduly jádra Spring frameworku, stejně jako ostatních specifických projektů z rodiny Spring a dalších často používaných knihoven třetích stran. Hlavním přínosem jednotné platformy je správce závislostí, spravující verze využitých podporovaných knihoven bez nutnosti specifikace jejich verzí programátorem. Ten tak má zajištěn bezproblémový vývoj vzájemně kompatibilních a testovaných komponent a jejich verzí, podobně jako jednodušší přechod na novější verzi platformy. Související technologií je projekt Spring Boot, umožňující vytvářet aplikace nezávislé na tradičním aplikačním serveru, za pomocí generování přímo spustitelných (stand-alone) JAR (Java Archive) aplikací. Spring Boot podporuje integraci webového serveru, servlet kontejneru a dalších komponent do finální soběstačné aplikace. Využívá tak opačného principu než v případě nasazení aplikace do aplikačního serveru, který tyto komponenty standardně obsahuje. Spring Boot ve spolupráci se Spring IO Platform zjednodušuje provedení úvodní konfigurace a automaticky konfiguruje jednotlivé Spring komponenty pro umožnění rychlého započetí vývoje.

2.2.5 Spring MVC

Spring MVC je technologie implementující tradiční architekturu uživatelské aplikace model-view-controller v prostředí webu. Je založena na servletu, který dle konfigurace deleguje požadavky na odpovídající kontroléry k jejich obsloužení. Spring MVC poskytuje vysokou míru volnosti ve využití modelu i ostatních komponent bez nutnosti implementace předdefinovaných rozhraní frameworku. Obsluhující kontroléry nejsou vázány na konkrétní formát zobrazení a mohou univerzálně poskytovat data pro JSP (Java Server Pages) stránky či generovat data ve formátech XML (Extensible Markup Language), JSON (JavaScript Object Notation) a mnoha dalších, a mohou tak být vyu-



Obrázek 2.2: Porovnání nasazení do aplikačního serveru a Spring Boot aplikace

žity k vytváření webových služeb architektury REST (Representational State Transfer).

2.2.6 Spring Security

Spring Security je framework poskytující služby autentizace i autorizace požadavků obsluhovaných Spring aplikací. Umožňuje transparentně ověřovat identitu uživatele před zavoláním jakékoli vrstvy obchodní logiky bez nutnosti zásahu do této vrstvy. Podporuje zabezpečení na úrovni metod pomocí požadovaných rolí i omezení přístupu dle pravomocí ke konkrétnímu zpracovávanému doménovému objektu. Obsahuje ověřovací moduly pro nejčastěji používané metody zabezpečení a rozhraní pro implementaci vlastních poskytovatelů autentizace.

2.2.7 Spring Data

Spring Data cílí na zajištění jednotného přístupu k persistentním datovým úložištím bez ohledu na typ uložení informací – od relačních databází, přes NoSQL databáze a MapReduce frameworky, k datovým službám provozovaným v cloudu [4]. Poskytuje koncept tzv. repozitáře, který zajišťuje manipulaci s persistentní entitou. Repozitář je tvořen pouhým rozhraním, implementace je dynamicky doplněna a databázové dotazy vytvořeny odvozením od názvu metody pro přístup k datům. Vývojář je tak odstíněn od nutnosti definovat konkrétní způsob získání požadovaných dat. Technologie také obsahuje podporu pro uchovávání záznamů o manipulaci s persistentním objektem.

2.2.8 Komunikační rozhraní

Nejširší podporu pro poskytování služeb pro různé typy zařízení a aplikací v prostředí internetu mají webové služby typu SOAP a REST. SOAP (Simple Object Access Protocol) je komunikační protokol pro výměnu informací předem definované struktury pomocí formátu XML. K přenosu informace je nejčastěji použit protokol HTTP, SOAP však není závislý na konkrétním transportním protokolu. Jako následník protokolů pro vzdálené volání procedur je vhodný pro přenos zprávy či příkazu s libovolnou, konkrétní aplikací definovanou sémantikou. Oproti tomu REST je architektonickým stylem, navrženým disertační prací Roye Fieldinga [5], vyžadujícím dodržování jednotného rozhraní, definovaného pomocí sady jednoznačně identifikovaných zdrojů (resources) a skupiny metod pro manipulaci s nimi. U webových služeb typu REST odpovídají tyto metody standardním HTTP metodám GET, PUT, POST a DELETE, odpovídající CRUD (Create, Read, Update, Delete) operacím pro manipulaci s daty. Význam jednotlivých metod je tak dán samotnou specifikací protokolu. Další vlastností této architektury je oddělení zdroje od jeho reprezentace. Reprezentace zdroje může být realizována libovolným formátem,

nejčastěji využívané formáty jsou JSON a XML. Díky práci klientů s touto reprezentací, bez nutnosti znát způsob skutečné podoby zdroje na serveru, a díky jednotnému rozhraní vzniká architektura klient-server s volnou vazbou (loosely-coupled), umožňující vývoj a pozdější změny interní podoby klientské i serverové aplikace navzájem nezávisle. Protokol je také inherentně bezstavový a podporuje ukládání výsledků dotazu v mezipaměti.

2.2.9 HATEOAS

Jeden z nejčastěji diskutovaných principů architektury REST definuje hypermédia jako zdroj aplikačního stavu (Hypermedia as the Engine of Application State, HATEOAS). Tento princip spočívá v komunikaci pomocí dynamicky sestavovaného hypermédia (tedy média obsahujícího odkazy) za předpokladu, že klient nenese žádnou znalost o tom, jak s poskytovanou službou komunikovat. HATEOAS princip implikuje nutnost klienta být schopen využívat službu pouze se znalostí vstupního bodu služby (adresy serveru) a podporovaného formátu reprezentace, jako je JSON nebo XML. HATEOAS princip je však široce porušován většinou webových služeb, které jsou označovány za služby architektury REST, a to zejména nemožností vyhledat požadovaný zdroj bez znalosti jeho unikátního identifikátoru (Uniform Resource Identifier, URI). Správnost interpretace HATEOAS principu dále potvrdil Roy Fielding ve svém článku *REST API musí být řízeny hypertextem* [6], ve kterém uvádí: *„REST API by mělo být využito bez předchozí znalosti s výjimkou iniciální URL a sady standardizovaných typů médií, které jsou adekvátní pro zamýšlené příjemce. Od této chvíle musí být všechny změny aplikačního stavu prováděny pomocí klientova výběru ze sady serverem nabízených možností, které jsou přítomny v přijaté reprezentaci nebo implikovány uživatelskou manipulací s touto reprezentací. [...] Nesplnění tohoto principu implikuje užívání nepřenášených informací pro interakce namísto hypertextu“* (překlad vlastní). Míru splnění jednotlivých architektonických principů v používaných REST API kategorizuje tzv. *Richardson Maturity Model* [7][8]:

- Úroveň 0
  - Použití HTTP protokolu jako transportní systém a tunelovací mechanismus pro vlastní metodu vzdáleného ovládání, typicky založenou na vzdáleném volání procedur. Nejsou využity standardní principy webu.
- Úroveň 1 – Zdroje
  - API identifikuje jednotlivé zdroje a přiřazuje každému zdroji vlastní unikátní URI. Klient zná tyto URI a vznáší požadavek na adresu, která odpovídá jím požadovanému zdroji. Obvykle je použita pouze jedna HTTP metoda nebo je jejich význam identický.

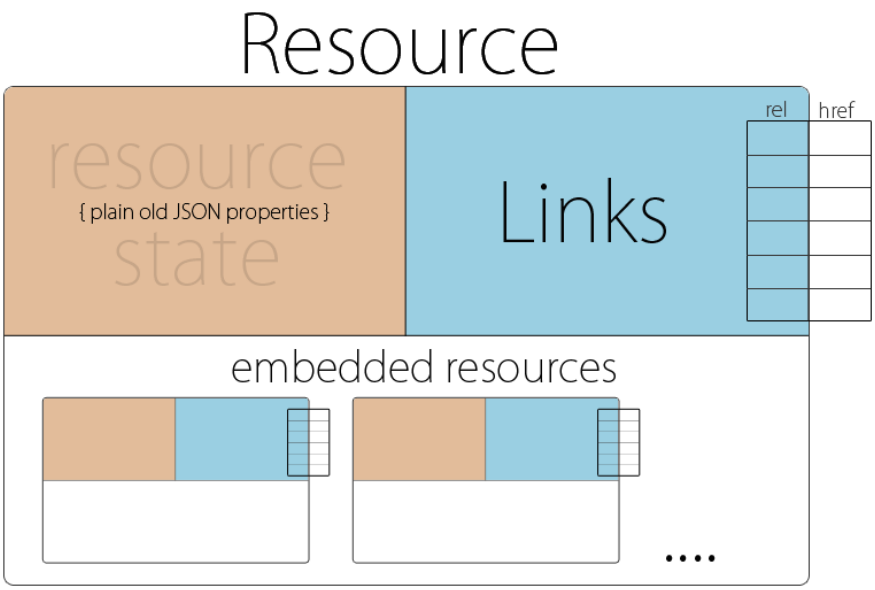
HTTP metoda	Idempotentní	Bezpečná
GET	ano	ano
HEAD	ano	ano
PUT	ano	ne
POST	ne	ne
DELETE	ano	ne
PATCH	ne	ne
OPTIONS	ano	ano

Tabulka 2.2: Idempotentní a bezpečné metody

- Úroveň 2 – HTTP metody
  - Služba respektuje sémantiku HTTP metod určených REST standardem a klient takové služby určuje požadovanou operaci uvedením adekvátní HTTP metody. API vrací korektní HTTP stavové kódy při vytváření odpovědi a respektuje idempotenci metod. Idempotentní metoda je taková, která může být opakovaně volána, aniž by se opakovanými voláními měnil počet vyvolaných vedlejších efektů. Bezpečná metoda je taková, která nemění reprezentaci zdroje.
- Úroveň 3 – Hypermédia
  - Nejvyšší úroveň REST API umožňuje klientům sdělit, jaké operace mohou v každé situaci provádět a na jakých adresách. Odpovědi na dotazy tedy obsahují odkazy na ostatní, v dané situaci související akce. Klient nepotřebuje znát konkrétní adresy předem a nemá tak problém s dynamickou změnou struktury služby. Klient je schopen dynamicky nalézt nové funkce služby a přizpůsobit se změnám bez narušení zpětné kompatibility.

2.2.10 HAL

Principy HATEOAS splňuje pouze nejvyšší úroveň výše uvedeného modelu a nižší úrovně tak nejsou kompatibilní s požadavky architektury REST, přestože jsou jako REST API označovány [9]. Snaha o splnění všech architektonických požadavků však naráží na omezenou podporu ze strany používaných reprezentací. Jak zmiňuje W3C, konsorcium pro vývoj webových standardů: *„JSON nemá žádnou zabudovanou podporu pro hyperlinky, které jsou fundamentálními stavebními bloky na webu“* [10] (překlad vlastní). Pro jednoduché strojové zpracování odkazů je potřeba definovat rozšíření standardních reprezentací o podporu odkazů. Takovým rozšířením je formát HAL, jehož formální specifikace JSON Hypertext Application Language je definována internetovým



Obrázek 2.3: Struktura formátu HAL [12]

draftem organizace Internet Engineering Task Force (IETF) [11]. Formát je rozšířením formátu JSON o sekci s odkazy, podporu pro vnořování reprezentací zdrojů do jiných, šablonové odkazy a zkracování URI (*curie*) [12]. Syntaxe jazyka JSON je plně dodržena a formát HAL tak může být zpracováván standardními JSON parsery. Označení typu internetového média (Internet Media Type, též MIME Type) je *application/hal+json*.

2.2.11 Spring HATEOAS

Spring HATEOAS je framework, poskytující rozhraní pro vytváření REST API, které splňují princip Hypermedia as the Engine of Application State. Poskytuje podporu pro vytváření odkazů na jednotlivé zdroje odkazováním na metody obslužných tříd. Vývojář tak není nucen pevně určovat odkazy na související zdroje – místo toho poskytne referenci na metodu třídy, která zdroj poskytuje. Framework určí odpovídající adresu dle toho, na kterou destinaci je odkazovaná třída i metoda namapována. Dále definuje návrhový vzor pro sestavování REST zdrojů z databázových entit.

2.2.12 JAX-RS a Jersey

Java API for RESTful Web Services (JAX-RS) je specifikace rozhraní jazyka Java pro vytváření webových služeb typu REST a jejich klientů. Poskytuje možnost jednoduše definovat mapování metod na zdroje přístupné přes web,

```
{
    "_links": {
        "self": { "href": "/orders" },
        "curies": [{ "name": "ea",
            "href": "http://example.com/docs/rels/{rel}",
            "templated": true }],
        "next": { "href": "/orders?page=2" },
        "ea:find": {
            "href": "/orders{?id}",
            "templated": true
        },
        "ea:admin": [{
            "href": "/admins/2",
            "title": "Fred"
        }, {
            "href": "/admins/5",
            "title": "Kate"
        }]
    },
    "currentlyProcessing": 14,
    "shippedToday": 20,
    "_embedded": {
        "ea:order": [{
            "_links": {
                "self": { "href": "/orders/123" },
                "ea:basket": { "href": "/baskets/98712" },
                "ea:customer": { "href": "/customers/7809" }
            },
            "total": 30.00,
            "currency": "USD",
            "status": "shipped"
        }, {
            "_links": {
                "self": { "href": "/orders/124" },
                "ea:basket": { "href": "/baskets/97213" },
                "ea:customer": { "href": "/customers/12369" }
            },
            "total": 20.00,
            "currency": "USD",
            "status": "processing"
        }]
    }
}
```

Obrázek 2.4: Příklad dokumentu ve formátu HAL [12]

dojednat formát komunikace a zpracovávat parametry požadavku z adresy, těla i hlaviček HTTP požadavku. Jersey je knihovnou poskytující referenční implementací specifikace JAX-RS. Poskytuje intuitivní rozhraní pro implementaci konvertorů z/do libovolných formátů pro přenos mezi oběma komunikující stranami.

2.2.13 Jackson

Jackson je knihovna pro zpracování formátu JSON, jeho převod na objekty jazyka Java a naopak. Poskytuje také anotace pro provedení mapování a označení požadovaných atributů, které mají být serializovány. Může být využit knihovnou Jersey pro implementaci webových služeb, využívajících formát JSON.

2.2.14 Autentizace

Pro řízení přístupu k webovým službám architektury REST je možné použít několik autentizačních technologií:

- Základní HTTP autentizace
  - Základní HTTP autentizace (HTTP Basic authentication), definovaná standardy organizace IETF RFC 2617 [13] a RFC 7235 [14], spoléhá na přenos uživatelského jména a hesla HTTP hlavičce Authorization s prefixem Basic. Základní autentizace byla dříve používána k zabezpečení webových stránek, z důvodu široké podpory webových prohlížečů, které automaticky reagovaly na autentizační požadavek a odstiňovaly webovou stránku od nutnosti explicitního ověřování uživatelských informací. Uživatelské údaje nejsou před přenosem žádným způsobem kryptograficky zabezpečeny, pouze je na ně aplikováno kódování Base64, reprezentující binární data jako ASCII řetězec. Zabezpečení přenášených informací je tak ponecháno na přenosovém protokolu, což implikuje nutnost využití zabezpečeného protokolu HTTPS. Bez jeho využití je autentizační mechanismus také zranitelný útokem typu man-in-the-middle či opakováním dříve zachyceného požadavku. Protože architektura REST vyžaduje bezstavovost, musí být autorizační údaje opakovány v každém klientském požadavku na webovou službu a serverem při každém volání ověřovány. Na obdobném principu funguje Digest autentizace, s rozdílem, že na přihlašovací údaje před jejich odesláním aplikuje hašovací funkci MD5. Tato metoda v dnešní době neposkytuje zásadním způsobem vyšší úroveň zabezpečení oproti základní HTTP autentizaci.
- Relace pomocí cookies

  - Obvyklé využití cookies spočívá v uložení identifikátoru konkrétní relace tak, aby byl server schopen identifikovat v rámci které relace je klientský požadavek vznesen. Takovéto využití implikuje server, který si musí pamatovat stav jednotlivých relací. Pro eliminaci této potřeby je nutné do cookie umístit všechny informace o konkrétní relaci namísto jejího identifikátoru. Klientovi pak není žádným způsobem znemožněno neoprávněně manipulovat s těmito údaji. Využití cookies pro implementaci bezstavového serveru je tak téměř nemožné.

- Autentizační token

  - Princip autentizačního tokenu spočívá ve vygenerování řetězce, který je klientem přikládán v hlavičce každého HTTP požadavku. Řetězec je vygenerován na straně serveru po úspěsném přihlášení uživatele a zaslán klientovi pro další komunikaci. Přiložený token může být ověřen proti databázi vydaných tokenů v případě stavového serveru, či může být technologicky zaručena autenticita, integrita a nepopiratelnost vystaveného tokenu, který pak může být ověřen bezstavovým serverem. Autentizační token je využíván například protokolem OAuth 2.0, který odděluje autorizační server od serveru s požadovanými zdroji a umožňuje tak přihlášení pomocí účtu u poskytovatele identity třetí strany, jako je Google či Facebook, aniž by klientská aplikace měla přístup k uživatelem poskytnutým utajeným údajům. Protokol podporuje selektivní určení rozsahu přístupu, který má být uživateli poskytnut, a umožňuje využití různých typů a formátů autentizačních tokenů.

- Podepisování požadavků

  - Další možností pro bezstavové ověření identity uživatele je připojení podpisu ke každému požadavku jako součást URL či hlavičky požadavku. Generováním podpisu se obvykle rozumí výpočet HMAC (Keyed-hash Message Authentication Code) řetězce, obsahujícího všechny parametry požadavku, za využití sdíleného tajemství, kterým může být API klíč či uživatelské údaje [15]. Server dokáže verifikovat uživatelovu identitu pomocí identického výpočtu podpisu přijatého požadavku a jeho porovnání s obdrženou hodnotou. Problémem může být identická podoba podpisu při podepisování požadavků neobsahujících žádné parametry [15]. Dále řešení naráží na nedostatečnou standardizovanost formátu přenášené identifikace uživatele a podpisu v hlavičce požadavku či nevýhody plynoucí z inkluze podpisu do URL.

2.2.15 JWT

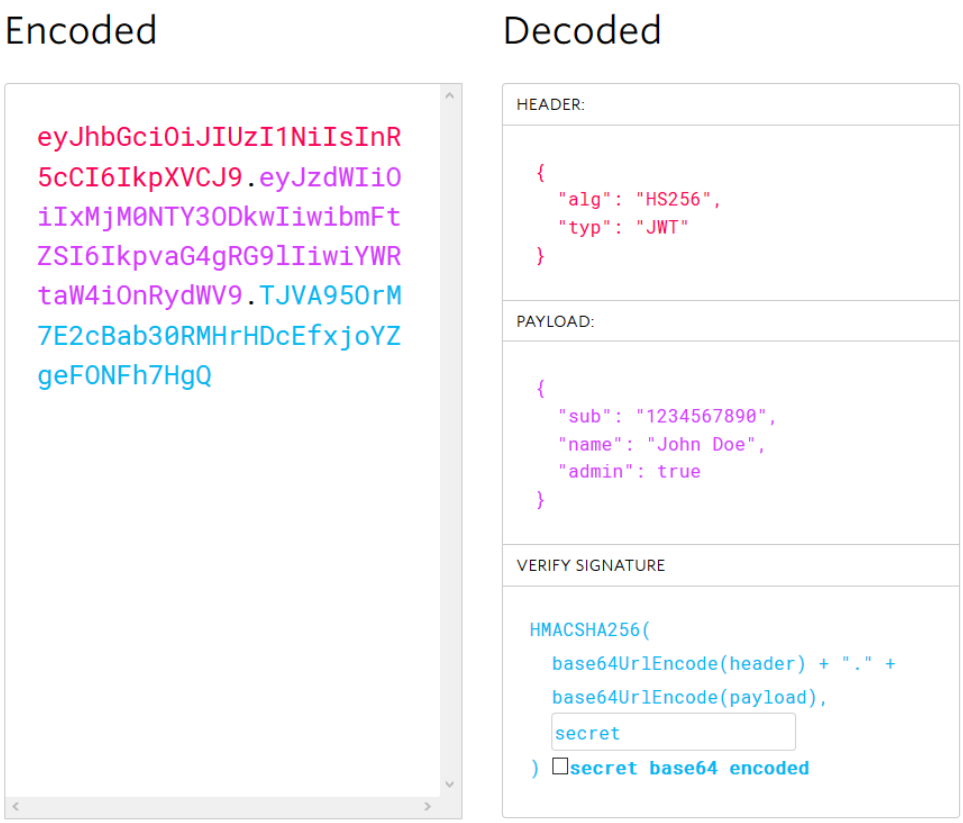
JWT, či JSON Web Token, je standard, definovaný v RFC 7519 [16], pro bezpečný přenos informací mezi komunikujícími stranami prostřednictvím JSON objektu [17]. JWT může být využit jako autentizační token při ověřování identity uživatele. Struktura tokenu je následující:

- Hlavička (Header)
  - Obsahuje typ tokenu a algoritmus použitý k podpisu tokenu
- Tělo (Payload)
  - Obsahuje jednotlivé informace (claims) o dané entitě, tedy data, jejichž důvěryhodnost je potřeba zaručit. Informace se dělí na vyhrazené, veřejné a soukromé. Mezi nejdůležitější vyhrazené informace se řadí vydavatel tokenu, ID tokenu, datum vydání, datum expirace, předmět a příjemce. Veřejné informace mohou být definovány uživateli, přičemž by měly být obsaženy v k tomu určenému registru organizace IANA. Soukromé informace jsou libovolné informace, které komunikující strany využívají.
- Podpis (Signature)
  - Podpis obsahuje výsledek výpočtu podpisového algoritmu, na jehož vstupu je enkódovaná hlavička a tělo tokenu, spolu s privátním klíčem. Podpis zaručuje, že zprávu podepsal držitel soukromého klíče a zpráva nebyla po jejím podepsání změněna.

Praktická podoba enkódované i dekódované reprezentace tokenu je patrná z obrázku 2.5. Výhodou JWT je možnost umístit do tokenu libovolné aplikačně-specifické údaje, jako jsou například uživatelské role (rozsah oprávnění), kterým pak server díky důvěryhodnému podpisu může důvěřovat bez nutnosti ověřovat platnost těchto údajů. JWT tak umožňuje nejen implementaci bezstavového autentizačního a autorizačního schématu, ale může též eliminovat nutnost databázových dotazů při ověřování identity a pravomocí uživatelů a dalších údajů. Nevýhodou JWT je nemožnost jednoduché revokace platnosti tokenu bez použití stavových černých listin (blacklistů) a je nutno je tak vystavovat na omezenou dobu platnosti a implementovat mechanismus obnovení tokenu. Při použití JWT pro autentizaci je token klientem po jeho vystavení vkládán do HTTP hlavičky Authorization s prefixem Bearer při každém prováděném požadavku.

2.2.16 PostgreSQL

PostgreSQL je open-source objektově relační databázový systém s podporou jazyka SQL. Má za sebou více než 15 let aktivního vývoje a nabízí podporu



Obrázek 2.5: Podoba JWT tokenu [17]

všech hlavních operačních systémů. Prezentuje se jako nejpokročilejší open-source databázový systém na světě [18]. Z hlediska podporovaných funkcí se kromě standardních vlastností relačních databází zaměřuje na enterprise segment a nabízí tak podporu obnovy ke konkrétnímu stavu v minulosti, asynchronní replikace, vnořených transakcí, podobně jako sofistikovaný optimalizátor dotazů [18].

2.2.17 Umístění dokumentů

Kontroverzní otázku představuje umisťování souborů, spravovaných serverovou aplikací a poskytovaných klientům, do databáze oproti jejich ukládání na souborový systém. Obě řešení mají své výhody a nevýhody. Výhodou ukládání souborů jako binárních objektů do databáze je zajištění konzistence se záznamy v databázi díky podpoře transakčního zpracování i pro tyto soubory, jednotný přístup ke všem datům a eliminace potřeby zálohování a replikace dodatečného externího úložiště. Nevýhodou je vyšší cena za jednotku kapacity v databázi než na souborovém systému, problematičtější správa řádově větší databáze, omezení maximální velikosti souborů a další. Za zmínku také

stojí delší doba obnovy po selhání – v případě oddělení velkých binárních dat na souborový systém může být menší kriticky důležitá databáze rychleji obnovena a zprovozněna, zatímco delší doba obnovy binárních souborů nemusí být pro chod služby kritická. Výkonová stránka je výrazně závislá na velikosti souborů. V minulosti studie ukázaly výkonovou převahu databáze při práci se soubory menšími než 1 MB a ztrátu při zpracování souborů větších [19], komplexní výkonové testy dnešních databází však nejsou dostupné. Mnoho databázových strojů nicméně dnes podporuje uložení souborů ve speciálních datových typech, umisťujících data mimo standardní datové soubory tabulek. Díky tomu se některé rozdíly zmenšují. Výhodou souborového systému může být podpora komprese a deduplikace dat, při využití více serverů můžou být data lépe rozdělena za použití distribuovaného souborovému systému. Rozhodnutí o využití specifického řešení tak závisí na konkrétních požadavcích vyvíjené aplikace, velikosti souborů a četnosti přístupů k nim. Další alternativou je využití některé z NoSQL databází, která může mít nativní podporu ukládání neformátovaných dat. Dané řešení může být vhodné spíše za předpokladu využití nerelační databáze i na zbylá aplikační data.

Pro potřeby navrhované aplikace, kdy spravovanými binárními soubory jsou přílohy e-mailů, řádově přesahující velikost zprávy samotné, bylo zvoleno umístění těchto dat mimo relační databázi, na souborovém systému. Pro zajištění vysoké dostupnosti služby, jejíž realizace vyžaduje běh na serverech v geograficky oddělených datacentrech, byl zvolen souborový systém distribuovaný. Volba konkrétního souborového systému je diskutována v kapitole Nasazení.

2.2.18 Hibernate

Hibernate je framework zajišťující objektově-relační mapování. Zprostředkovává tedy uložení stavu objektů (entit) do persistentního úložiště, reprezentovaného relační databází, a jeho opětovné načítání. Framework odstiňuje programátora od nutnosti psát kód v jazyce relační databáze (SQL) a převádět data z objektů do databázových tabulek. Také do značné míry abstrahuje konkrétní použitou databázovou technologii podporou různých databázových dialektů. Konfiguraci mapování na databázové tabulky je možné provádět pomocí konfiguračního souboru ve formátu XML či obvykleji za užití anotací přímo v entitních třídách. Hibernate splňuje specifikaci standardu Java Persistence API (JPA) a může tak být využit jako poskytovatel persistence pro technologii Spring Data.

2.2.19 JDBC

Java Database Connectivity je rozhraní pro přístup k relačním databázím. Umožňuje aplikaci provádět SQL dotazy nad databází a získávat data z výsledků těchto dotazů. Vzhledem k tomu, že každý databázový stroj používá

do jisté míry odlišnou variantu dotazovacího jazyka, datových typů a dalších specifických parametrů, je pro konkrétní databázi vyžadován specifický JDBC ovladač. Hibernate interně používá JDBC API se specifickým ovladačem pro přístup k relační databázi.

2.2.20 SMTP

SMTP (Simple Mail Transfer Protocol) je textový protokol pro přenos e-mailů. Klientskými stanicemi je využíván pro odesílání zpráv. Protokol ve výchozím nastavení využívá TCP port 25, v případě zabezpečeného spojení, také označovaného jako SMTPS, port 465. SMTP server může vyžadovat přístupové údaje pro provedení autentizace přihlašovaného uživatele. Protokol v takovém případě ověřuje oprávnění uživatele odesílat zprávy, nicméně nevyžaduje, aby uživatel byl majitelem schránky, z jejíž adresy se e-mail pokouší zaslat. Z tohoto důvodu vznikají techniky jako SPF (Sender Policy Framework) či DKIM (DomainKeys Identified Mail) pro vyhrazení oprávněných odesílatelů z dané domény či podepisování e-mailu doménovým klíčem.

2.2.21 POP3

POP3 (Post Office Protocol v. 3) je protokolem pro získávání zpráv z e-mailových stránek. Klientské e-mailové aplikace si prostřednictvím tohoto protokolu vytvářejí lokální kopie zpráv dostupných na serveru. Nejčastějším průběhem POP3 relace je připojení k serveru, stažení všech zpráv, jejich lokální uložení a následné odstranění ze serveru. Ačkoli klient může ponechávat stažené zprávy na serveru, protokol neposkytuje žádnou unikátní identifikaci jednotlivých zpráv ani podporu složek, klient je tak nucen pro zjištění přítomnosti nových zpráv při každém připojení porovnávat hlavičky zpráv na serveru s lokálními zprávami. Tato vlastnost činí protokol nevhodným pro uchovávání velkého množství zpráv na serveru. Vyhrazeným portem pro POP3 protokol je TCP port 110, zabezpečená varianta používá obvykle port 995.

2.2.22 IMAP

IMAP (Internet Message Access Protocol) je protokol pro vzdálený přístup k e-mailové schránce. Někdy je uváděn také jako IMAP4, dle aktuálně dominantní verze protokolu. V mnoha ohledech se jedná o následníka POP3 protokolu, který překonává v možnostech manipulace se zprávami na serveru, podpoře složek a systémových příznaků. Principiální rozdíl při komunikaci protokolem IMAP je ten, že obvykle bývají všechny zprávy ponechány na poštovním serveru, ke kterému je klientská aplikace trvale připojena a lokálně stahuje jen aktuálně požadované informace. Každé zprávě je přiřazen unikátní identifikátor (UID), podle kterého je možné rozlišit identitu e-mailové zprávy. Každá složka také poskytuje informaci o příštím vydaném UID, díky čemuž je

možné monitorovat přijetí nové zprávy do složky bez nutnosti procházet existující zprávy. Protokol je tak vhodný i pro manipulaci s vyšším množstvím e-mailů v jedné schránce. Vyhrazeným portem pro IMAP je TCP port 143, případně může probíhat zabezpečeným spojením na portu 993.

Některé e-mailové servery podporují příkaz *IDLE* protokolu IMAP. Ten umožňuje klientské aplikaci signalizovat, že je připravena přijímat zprávy hned jakmile jsou přijaty serverem, který tak okamžitě upozorní klienta prostřednictvím otevřeného spojení na změnu monitorované složky. To umožňuje udržovat synchronicitu lokální složky se vzdálenou složkou téměř v reálném čase. Není-li použit příkaz *IDLE*, musí se klientská aplikace, podobně jako u protokolu POP3, v pravidelných intervalech opakovaně dotazovat serveru, zdali nejsou k dispozici nové zprávy.

2.3 Architektura serverové aplikace

Návrh využívá vícevrstvou architekturu, kdy datová vrstva je reprezentována repozitáři technologie Spring Data, které prostřednictvím poskytovatele persistence Hibernate komunikují s PostgreSQL relační databází. Vrstva obchodní logiky zpřístupňuje data jako zdroje webových služeb typu REST, obsahuje logiku pro manipulaci s těmito daty a provádí automatizované úkony aplikace, jakými je kontrola nakonfigurovaných e-mailových schránek a manipulace se zprávami. Prezentaci dat uživatelům zajišťuje klientská aplikace, komunikující s vrstvou obchodní logiky prostřednictvím webových služeb.

2.3.1 Datový model

Nejprve je nutné identifikovat jednotlivé entity systému, které je potřeba evidovat. Jedná se o následující:

Company společnost reprezentující subjekt, který poskytuje zákaznickou podporu

Customer zákazník společnosti, který by mohl vznést dotaz na zákaznickou podporu

User zaměstnanec společnosti, vyřizující jednotlivé požadavky

Group skupina uživatelů; skupina může být identifikována názvem a/nebo e-mailovou adresou

EmailAccount konfigurace e-mailové schránky; každé skupině může být přiřazena jedna výchozí e-mailová schránka

Email přijatá či odeslaná e-mailová zpráva

Ticket vznesený požadavek na zákaznickou podporu; může obsahovat jednotlivé záznamy, spočívající v souvisejících přijatých i odeslaných e-mailech, komentářích a přílohách a evidenci historie přiřazení jednotlivým zpracovatelům a změn stavu

Attachment přiložený dokument; dokument může být přiložen u konkrétní e-mailové zprávy či obecně u požadavku na zákaznickou podporu

Email přijatá či odeslaná e-mailová zpráva

Tag štítek; uživatelsky definovatelné štítky, umožňující kategorizaci požadavků jejich seskupováním

TextTemplate šablona textu zprávy; přes tuto entitu jsou implementovány šablony zpráv i podpisů

Permission oprávnění; jedná se o základní sadu systémových oprávnění pro provádění jednotlivých úkonů, které nejsou specifické pro konkrétní společnost

Role uživatelské role; role může být specifická pro konkrétní společnost a skládá se ze skupiny systémových oprávnění – reprezentuje tak logické postavení uživatele, ve kterém se nachází

TicketEntry záznam, spadající pod požadavek na zákaznickou podporu; reprezentuje abstraktního rodiče konkrétních typů záznamů

AttachmentCreateEntry záznam reprezentující přidání jedné či více příloh k požadavku

AttachmentDeleteEntry záznam reprezentující odstranění jedné čí více příloh požadavku

CommentEntry záznam reprezentující interní komentář požadavku

GroupAssignmentEntry záznam reprezentující přiřazení požadavku skupině

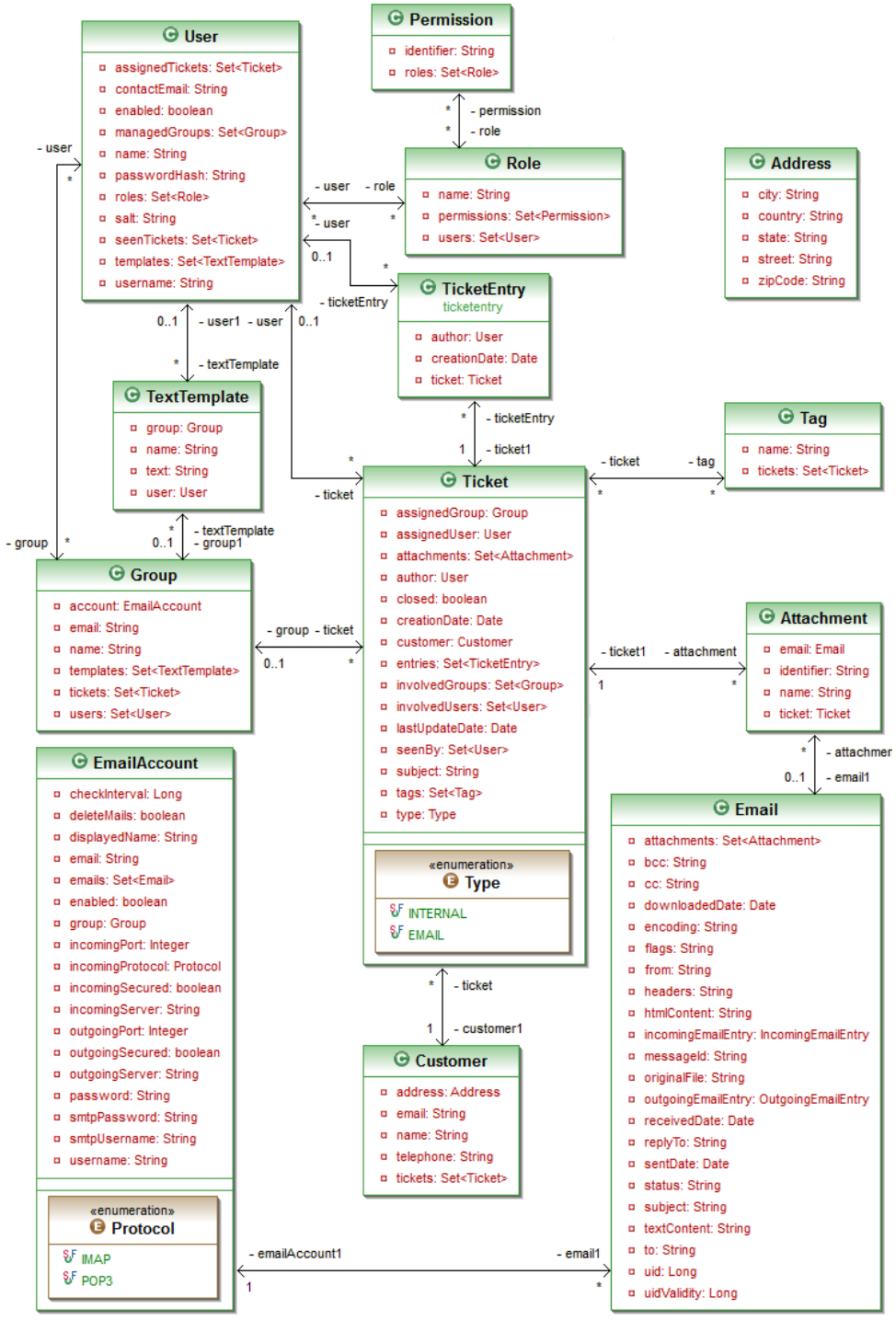
IncomingEmailEntry záznam reprezentující přijatou e-mailovou zprávu

OutgoingEmailEntry záznam reprezentující odchozí e-mailovou zprávu

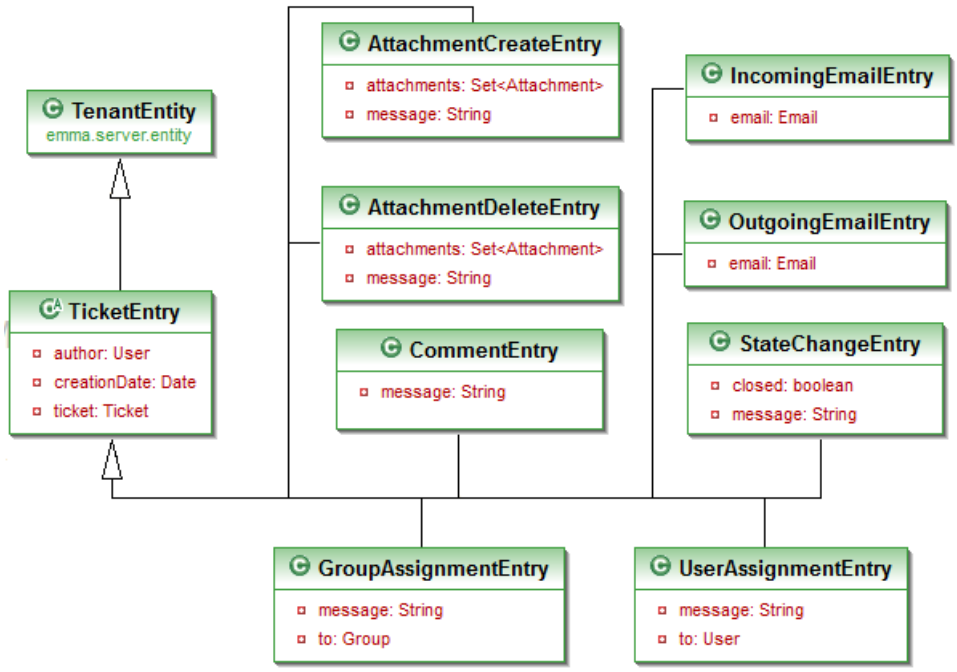
StateChangeEntry záznam reprezentující provedenou změnu stavu požadavku

UserAssignmentEntry záznam reprezentující přiřazení požadavku konkrétnímu uživateli

Diagram entitních tříd je vyobrazen na obrázcích 2.6 a 2.7.



Obrázek 2.6: Entitní třídy a jejich vztahy



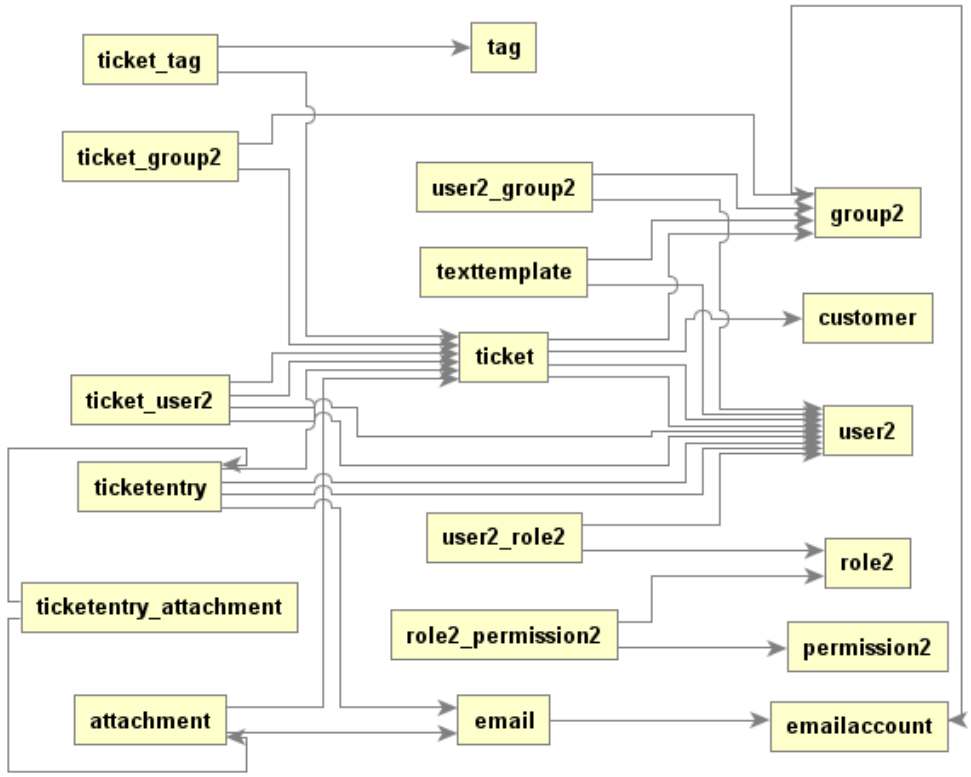
Obrázek 2.7: Entitní třídy balíku *emma.server.entity.ticketentry*

2.3.2 Databázové tabulky

Identifikovaná sada entit je realizována databázovými tabulkami dle obrázku 2.8. Obrázek pro zachování přehlednosti neobsahuje tabulku *company*, kterou cizími klíči referují všechny tabulky s výjimkou vztahových tabulek a tabulky *permission2*. Relace m:n mezi libovolnými objekty je reprezentována vztahovou tabulkou pojmenovanou dle obou stran relace, s cizími klíči mapovanými na primární klíče obou tabulek.

Každá tabulka (s výjimkou vztahových tabulek) obsahuje sloupec *id* typu *BIGSERIAL*, reprezentující unikátní identifikátor záznamu, který je primárním klíčem záznamu, a sloupec *version* typu *TIMESTAMP*, využívaný poskytovatelem persistence k zajištění optimistického zamykání a zabránění přepsání dat při editaci více uživateli mimo rozsah jedné transakce. Dále tyto tabulky obsahují sloupec *deleted* typu *BOOLEAN*, používaný pro indikaci odstranění záznamu místo nutnosti jej fyzicky odebrat. To umožňuje částečně evidovat historii záznamu a obnovit záznam v případě potřeby.

Protože vztah mezi třídou *TicketEntry* a jednotlivými konkrétními typy záznamů reprezentuje dědičnost, nabízí se tři varianty databázové implementace tohoto vztahu. První možností je využití jedné tabulky, reprezentující všechny typy. Taková tabulka obsahuje sloupce všech konkrétních potomků a navíc sloupec, dle kterého je rozlišen typ záznamu (*discriminator column*).



Obrázek 2.8: Vztahy mezi databázovými tabulkami

Slabinou řešení je nemožnost implementovat integritní omezení *NOT NULL* nad sloupci potomků a nutnost filtrovat všechny záznamy tabulky v případě dotazu na záznamy konkrétního potomka. Alternativní variantou je vytvoření tabulky pro každý konkrétní typ i pro abstraktního předka. Konkrétní typy odkazují na primární klíč předka. Nevýhodou je rozdělení jednoho logického záznamu do dvou tabulek a nutnost tyto tabulky spojovat při každém dotazu na celý záznam. Poslední možností je vytvořit tabulku pouze pro každý konkrétní typ. V důsledku nutnosti spojovat výsledky mnoha dotazů (či využití databázového sjednocení) v případě dotazu na všechny záznamy nehledě na typ není toto řešení pro navrhovanou aplikaci vhodné. Hibernate je schopen konfigurace pro komunikaci s libovolnou podobou realizace dědičnosti pouhou změnou anotace abstraktního typu. Pro dosažení nejvyššího výkonu, za situace, kdy žádná operace neprovádí dotaz jen na konkrétní typ, je zvoleno řešení pomocí jedné tabulky s rozlišovacím sloupcem. Takovouto tabulkou je tabulka *ticketentry*. Požadovaná integritní omezení na hodnoty jednotlivých sloupců jsou zajištěna aplikační logikou.

2.3.3 Spring Data repozitáře

K informacím v databázi je přistupováno pomocí repozitářů technologie Spring Data. Pro každou entitní třídu je definován vlastní repozitář se všemi požadovanými metodami pro hledání a manipulaci s daty. Každý repozitář je tvořen rozhraním, které je potomkem rozhraní *JpaRepository*, a získává tak metody pro přístup, modifikaci a odstranění entity: *findAll()*, *getOne()*, *save()*, *delete()* a další. Do těchto rozhraní jsou doplněny hlavičky metod pro hledání dle požadovaných parametrů. Spring Data doplní konkrétní implementaci, spolu s dotazy vznesenými na databázi, dynamicky při běhu aplikace. Databázové dotazy jsou odvozeny dle názvu metody. Například metoda *User findByUsername(String)* z repozitáře *UserRepository* vyhledá záznam z tabulky *user2* s požadovanou hodnotou ve sloupci *username*. Podporovány jsou i logické operátory, řazení výsledku a filtrování dle parametrů vztahové entity. Například metoda *List<Ticket> findByCompanyIdAndDeletedFalseOrderByLastUpdateDateDesc(Long)* vráti všechny požadavky specifikované společnosti, které nejsou smazány, seřazené dle data poslední editace sestupně. Kompletní implementace databázové vrstvy tak probíhá pouhým vytvořením rozhraní s korektně pojmenovanými metodami.

2.3.4 REST zdroje

Prostřednictvím webových služeb typu REST jsou klientům zpřístupněny jednotlivé datové zdroje. Každý zdroj je reprezentován jako třída, která je potomkem třídy *ResourceSupport* definované frameworkem, jež přidává zdrojům schopnost vkládat odkazy na jiné zdroje v souladu s formátem HAL. Navrhovaná aplikace přidává do tohoto řetězce dvě další úrovně dědičnosti – třídu *HALResource*, která je potomkem třídy *ResourceSupport* a přidává metody pro vnořování (embedding) zdrojů, a třídu *VersionedResource*, která je potomkem třídy *HALResource*, přidávající všem zdrojům časovou značku, pro zabránění přepsání dat při paralelní editaci stejného zdroje více uživateli. Zdroje do značné míry odpovídají databázovým entitám, s výjimkou takových dat, která jsou interní pro využití serverem a nemají být klientům prezentována. Konkrétní popis všech zdrojů je uveden v Příloze B.

2.3.5 Resource assemblers

Resource assemblers jsou třídy zodpovědné za sestavování zdrojů. Třídy jsou potomky abstraktní generické třídy *ResourceAssemblerSupport* a obsahují metody pro zajištění převodu entitních tříd odpovídajícího typu na REST zdroje. Sestavovací třídy také vkládají odkazy na ostatní zdroje a provádějí vnořování zdrojů. Tato architektura je doporučena frameworkem Spring HATEOAS. Návrh počítá i s využitím těchto tříd i pro opačný směr převodu, tedy z REST zdrojů na entity při vytváření či editaci zdrojů klienty. Pro ověření bezkoliznosti prováděné editace zde existuje třída *VersionedResourceAssemblerSup-*

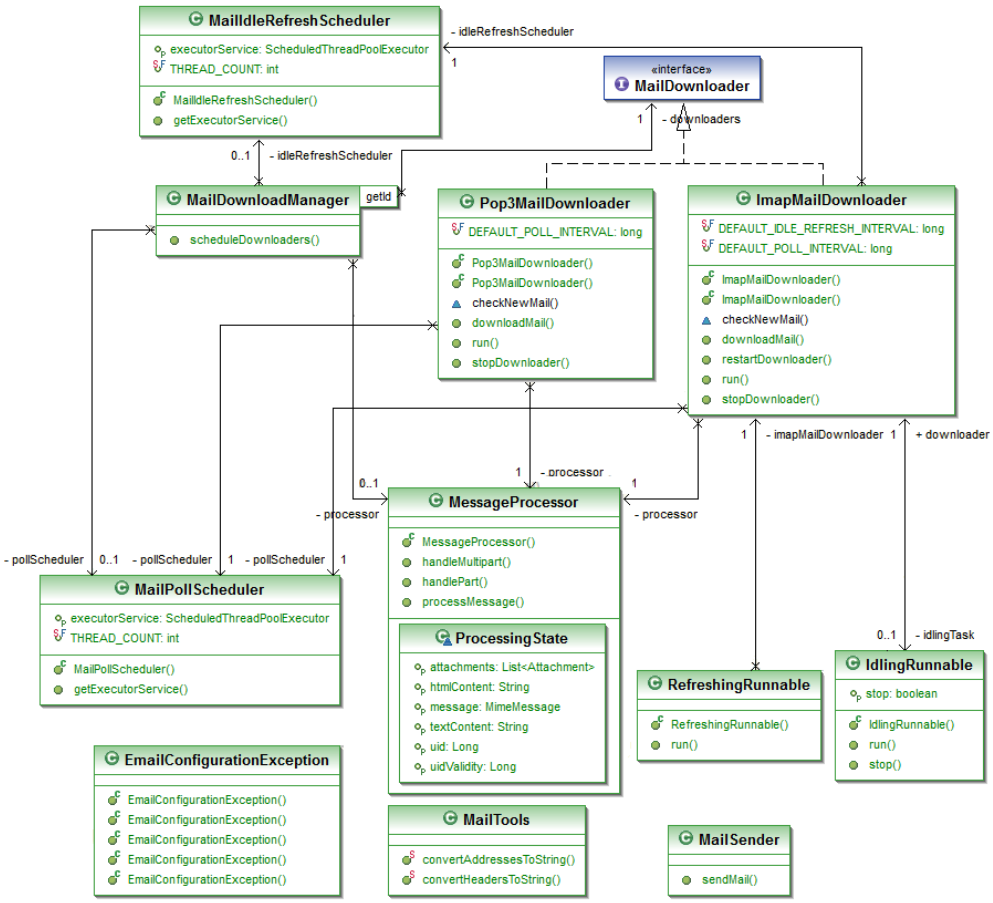
port, kontrolující, zda modifikovaná reprezentace zdroje je identické verze jako aktuální entita v databázi.

2.3.6 Kontroléry

Kontroléry jsou třídy obsahující anotaci *@Controller*, značící, že objekt slouží k vyřizování požadavků vznesených klienty. Anotace je univerzální pro jednotlivé prezentační vrstvy podporované frameworkem Spring MVC a zde je využita pro vytvoření REST služeb. Každá třída zpřístupňuje jeden zdroj, metody těchto tříd odpovídají jednotlivým HTTP metodám pro konkrétní URL. Obvyklými metodami jsou *get()*, *getAll()*, *add()*, *replace()*, *update()*, *delete()*, případně další, jsou-li dané zdroje dostupné i na jiných adresách dle závislosti na ostatních zdrojích. Kontroléry obsahují aplikační logiku, zajišťující kontrolu požadavku a jeho zpracování, je-li validní. Argumentem upravujících metod je objekt, reprezentující upravovaný zdroj. Spring MVC ověřuje syntaktickou správnost požadavku a konvertuje přijatou reprezentaci zdroje (např. ve formátu JSON) na objekt třídy reprezentující zdroj. V případě neúspěchu vrací klientovi chybový kód *400 Bad Request* a k zavolání metod kontroléru vůbec nedojde. Přehled všech URL a podporovaných metod je dostupný v dokumentaci API v Příloze B.

2.3.7 Zpracování e-mailů

Systém pro zpracování e-mailů, obsažený v balíku *emma.server.mail*, sestává z třídy *MailDownloadManager*, která je zodpovědná za vytvoření jednotlivých modulů provádějících stahování e-mailových zpráv, pro každý nakonfigurovaný e-mailový účet v databázi. Také zajišťuje naplánování jednotlivých úkonů kontroly schránky, prostřednictvím tříd *MailPollScheduler* a *MailIdleRefreshScheduler*, které reprezentují plánovače úloh pro pravidelnou kontrolu schránky, respektive pro opakované vydávání *IDLE* příkazu. Jednotlivé moduly pro stahování zpráv implementují rozhraní *MailDownloader* – aplikace obsahuje dva moduly, *Pop3MailDownloader* a *ImapMailDownloader*. Tyto moduly obsahují protokolově specifickou logiku pro připojení ke schránce, kontrolu a identifikaci nových zpráv a odpojení od schránky. V případě přijetí nové zprávy je tato zpráva předána objektu typu *MessageProcessor* pro zpracování. Ze zprávy jsou vyextrahovány potřebné údaje a zpráva je uložena do databáze. Komponenty *MailDownloadManager*, *MessageProcesor* a oba plánovače jsou spravovány kontextem a jsou do ostatních tříd injektovány pomocí vkládání závislostí. Moduly pro stahování zpráv jsou klasickými objekty, jejichž instance existuje pro každý účet, a jsou vytvářeny a spravovány komponentou *MailDownloadManager*. Dále se v balíku nachází komponenta *MailSender* pro zajištění odesílání e-mailů prostřednictvím protokolu SMTP a třída *MailTools*, poskytující pomocné statické metody pro práci se zprávami a hlavičkami.



Obrázek 2.9: Diagram tříd a asociací balíku mail

2.3.8 Zabezpečení

Pro autentizaci a autorizaci požadavků je využit framework Spring Security. Protože však ve výchozím stavu neobsahuje podporu pro zpracování JWT tokenů, je tato podpora implementována. To spočívá ve vytvoření několika souvisejících tříd – primárně se jedná o poskytovatele autentizace, který je schopen provést autentizaci a při zadání autentizačního tokenu vrátit objekt, reprezentující autentizovaného uživatele. Implementovaným poskytovatelem autentizace je třída *JwtAuthenticationProvider*, vracející autentizovaného uživatele v podobě objektu typu *AuthenticatedUser*, který obsahuje všechny informace o autentizovaném uživateli. Tento objekt je potomkem třídy *org.springframework.security.core.userdetails.User* a je tak frameworkem rozeznán jako nositel oprávnění. Dále je vytvořena třída *JwtAuthenticationToken*, reprezentující kódovanou podobu JWT tokenu (řetězec).

První komponentou, která je zavolána při ověřování identity uživatele, je

JwtAuthenticationFilter. Ten extrahuje JWT token z hlavičky požadavku a vytvoří instanci třídy *JwtAuthenticationToken*. Tento objekt je předán poskytovateli autentizace, který zaručí jeho dekódování komponentou *JwtTokenParser*, provádějící samotný převod formátu, a navrácení autentizovaného uživatele – instance *AuthenticatedUser*. Spring Security dále vyžaduje definici komponenty, zde pojmenované *JwtAuthenticatoinSuccessHandler*, implementující chování po úspěšné autentizaci. Například v případě využití Spring MVC pro implementaci webové stránky by bylo podobnou komponentou provedeno přesměrování na stránku zobrazenou pro přihlášení. V případě REST služby není potřeba vykonávat po provedení autentizace žádný úkon, zpracování má být navráceno kontroléru, a tak je implementace této komponenty prázdná. Balík také definuje výjimky pro případ chybného (*JwtTokenInvalidException*) či žádného (*JwtTokenMissingException*) autentizačního tokenu v klientském požadavku. Architektura mechanismu autentizace je inspirována článkem [20].

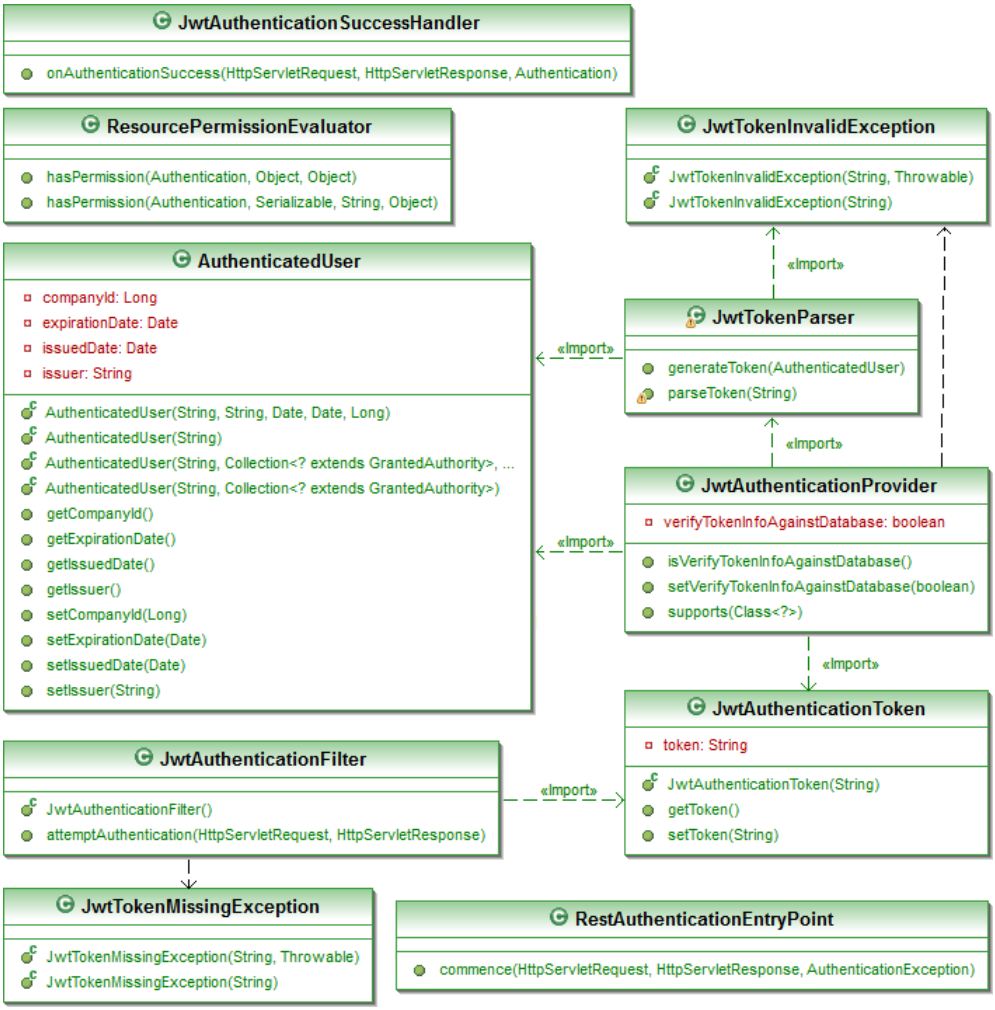
2.4 Architektura klientské aplikace

Desktopová klientská aplikace vzniká z důvodu maximalizace rychlosti práce se systémem a míry integrace s jinými uživatelskými aplikacemi. V rámci projektu je vyvíjena i webová aplikace, zajišťující kromě primárního uživatelského použití i konfiguraci uživatelů, skupin, pravomocí a e-mailových účtů. Společnost se pro využití systému také registruje přes webovou stránku. Tyto komponenty jsou vyvíjeny odlišnými autory a nejsou součástí této práce. Na desktopovou aplikaci ale díky tomu nejsou kladeny požadavky na uživatelské rozhraní pro konfiguraci těchto entit a je tak zaměřena výhradně na hlavní uživatelské použití aplikace, spočívající ve vyřizování požadavků zákaznické podpory.

Klientská aplikace je vyvíjena v jazyce Java a je založena na grafické knihovně JavaFX. Architektura aplikace vychází z architektonického vzoru MVC (*Model-view-controller*), oddělujícího zobrazení dat od aplikační logiky programu. Vzor je složen ze tří komponent: *model* reprezentuje zpracovávané informace, *view* (zobrazení, pohled) definuje grafické uživatelské prostředí aplikace a podobu zobrazení informací, a *controller* (kontrolér, řadič) zpracovává události vyvolané uživatelem a manipuluje s daty modelu či modifikuje zobrazení.

2.4.1 Zobrazení

Primární funkcionalita aplikace je realizována dvěma obrazovkami: přehledem požadavků na zákaznickou podporu a detailem požadavku. Obrazovka s přehledem požadavků je rozdělena na tři hlavní sekce – v levé části se nachází přehled agendy přihlášeného uživatele a jeho skupin, spolu s kategoriemi jednotlivých požadavků (nepřiřazené, nevyřízené, odeslané a vyřízené). V pravé horní části se nachází přehled jednotlivých požadavků ve vybrané kategorii,



Obrázek 2.10: Diagram tříd a závislostí balíku security

zatímco v pravé dolní části je detail vybraného požadavku a informace o zákazníkovi.

Po otevření detailu požadavku je zobrazena scéna, rozdělená na dvě sekce. V hlavní, pravé sekci se nachází informace o požadavku (předmět, datum vytvoření, stav, ...) a přehled jednotlivých záznamů požadavku, jako jsou příchozí zprávy, komentáře či historie změn stavu a přiřazení. V levém segmentu jsou jednotlivé akce, které může uživatel s konkrétním požadavkem činit. Klikne-li uživatel na některou z akcí či ji táhnutím myši přesune do hlavní části zobrazení, dojde k vysunutí další sekce ze spodní části obrazovky s poli náležícími ke konfiguraci dané akce. Uživatel tak může např. psát odpověď na e-mail v dolní časti obrazovky a přitom mít přístup ke všem záznamům požadavku v části horní. Těchto akcí je možné vybrat více, všechny akce jsou pak provedeny

najednou při stisknutí tlačítka Provést.

Vrstva zobrazení je implementována pomocí souborů ve formátu FXML, definující jednotlivé grafické kontejnery a umístění komponent. Zobrazení se skládá z následujících souborů:

MailboxPane.fxml definuje kontejner pro zobrazení přehledu agendy a skupin s kategoriemi

RequestListPane.fxml definuje kontejner pro zobrazení seznamu požadavků v konkrétní kategorii

RequestPane.fxml definuje kontejner pro zobrazení informací o konkrétním požadavku a jeho událostí

CustomerPane.fxml definuje kontejner pro zobrazení informací o zákazníkovi a jeho požadavcích

ToolbarPane.fxml definuje kontejner pro zobrazení horní lišty s nástroji

TaskListPane.fxml definuje kontejner pro zobrazení přehledu dostupných akcí k provedení s požadavkem

PendingTaskPane.fxml definuje kontejner pro zobrazení akcí vybraných uživatelem pro provedení

AttachmentPane.fxml definuje kontejner pro zobrazení seznamu příloh požadavku

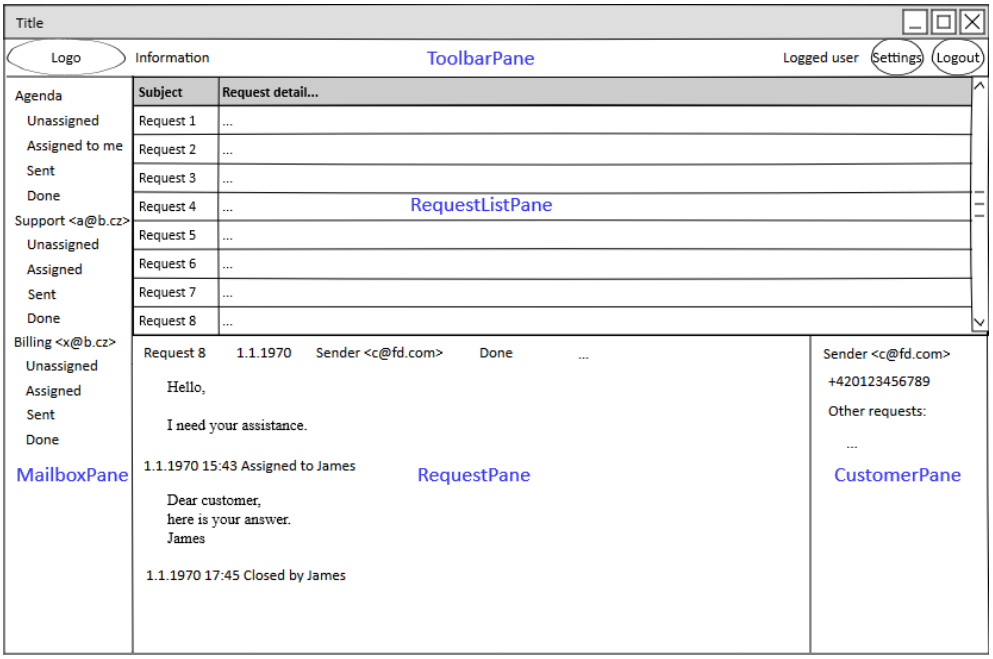
MainScene.fxml definuje scénu s přehledem požadavků, dle obr. 2.11; k tomu využívá ostatních FXML dokumentů

DetailScene.fxml definuje scénu s detailem požadavku, dle obr. 2.12; k tomu využívá ostatních FXML dokumentů

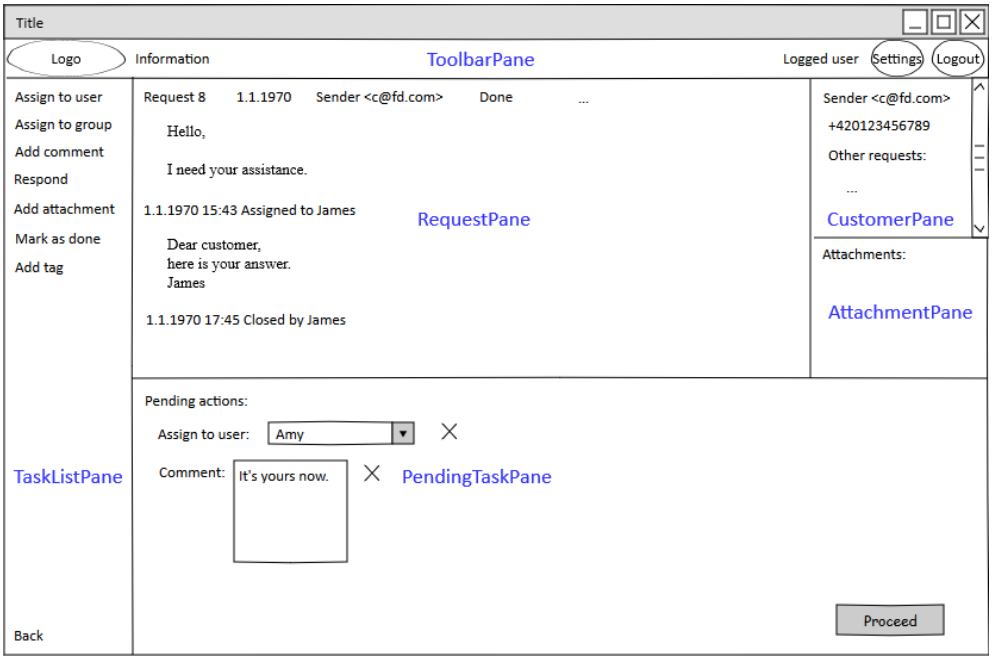
Zobrazení je tvořeno i dalšími FXML soubory, reprezentujícími kontejnery pro jednotlivé typy akcí a záznamů případu. V FXML souborech nejsou definovány kontroléry pro jednotlivé kontejnery, vrstva zobrazení tak není explicitně závislá na žádné z ostatních vrstev aplikace. Názvy proměnných a metod v kontrolérech však musí odpovídat hodnotám *fx:id* jednotlivých komponent a řetězcům, označujícím cíle zpracování událostí.

2.4.2 Kontrolér

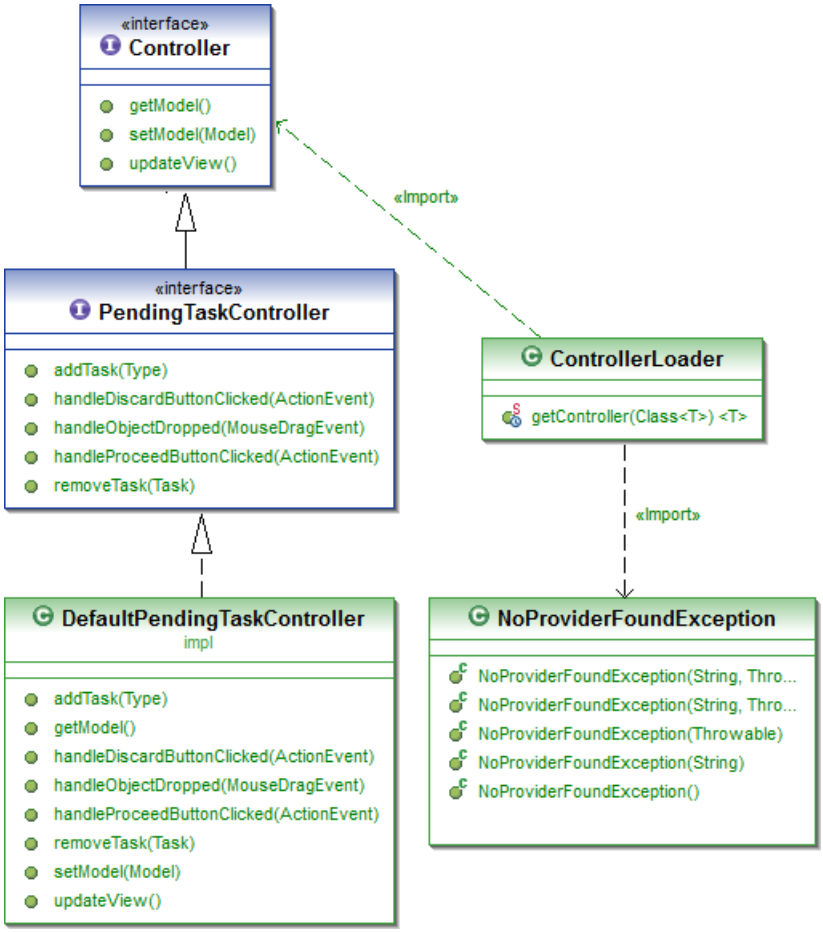
Každému z výše uvedených kontejnerů odpovídá kontrolér, zpracovávající události definované pro komponenty daného kontejneru. Pro oddělení jednotlivých vrstev aplikace se realizace kontroléru skládá z rozhraní, definujícího metody pro zpracování událostí a nastavování vlastností grafických komponent, a třídy



Obrázek 2.11: Nákres obrazovky s přehledem požadavků



Obrázek 2.12: Nákres obrazovky s detailem požadavku



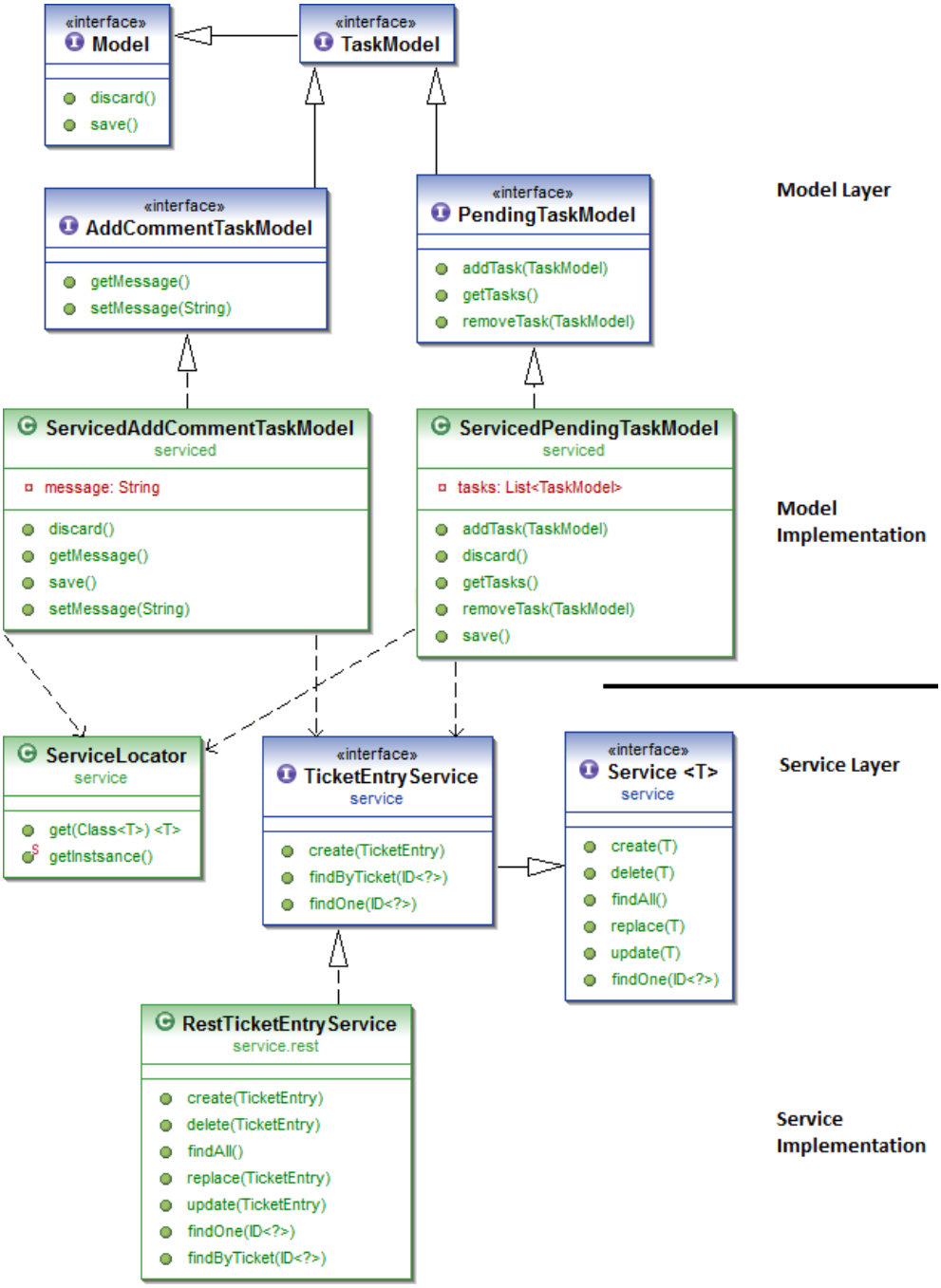
Obrázek 2.13: Příklad kontroléru a třídy ControllerLoader

poskytující výchozí implementaci těchto metod. Pro umožnění dynamické výměny implementace kontrolérů je navržena třída *ControllerLoader*, definující generickou metodu *T getController(Class<T>)*, která načítá konkrétní implementaci pomocí metod třídy *java.util.ServiceLoader* a ta tak může být nahrazena pouhou změnou konfigurace bez nutnosti upravovat kód. Příklad kontroléru pro jeden kontejner a ostatních souvisejících tříd je uveden na obrázku 2.13.

2.4.3 Model

Komponenty reprezentující model jsou rozděleny na dvě vrstvy. První vrstva reprezentuje zobrazovaná data, v podobě vhodné pro prezentaci uživateli, a tvoří tak samotný model. S touto vrstvou komunikuje kontrolér – žádá ji o data k zobrazení, či jí upravená data předává při odeslání formulářových informací. Třídy ale neobsahují logiku pro persistentní uchovávání informací.

K tomu slouží vrstva služeb, která poskytuje operace pro získávání, vytváření, editaci a mazání dat. Konkrétní implementací rozhraní této služby je klient webových služeb typu REST, komunikující se serverovou částí aplikace. Vrstva služeb také obsahuje třídy reprezentující REST zdroje, podobné, jako obsahuje serverová komponenta aplikace. Obě zmíněné vrstvy využívají techniku programování proti rozhraním pro lepší oddělení jednotlivých částí aplikace, podobně jako v případě kontrolérů. Klient webových služeb je realizován pomocí technologie JAX-RS a její referenční implementace Jersey. Pro zpracování formátu JSON je využit parser Jackson. Díky němu probíhá převod výsledku volání webové služby na doménový objekt plně automaticky. Pro zpracování odkazů a vnořených objektů jsou všechny tyto objekty potomky třídy *HALResource*, podobně jako v serverové části aplikace. V aplikaci nejsou pevně uvedeny adresy jednotlivých služeb. Klient je schopen za pomocí kořenové URL následovat odkazy k požadovaným zdrojům; k tomu vyžaduje jen znalost názvu zdroje, který vyhledává. Objevené adresy si nicméně v rámci běžící instance uchovává pro rychlejší zpracování následných volání.



Obrázek 2.14: Vybrané komponenty modelu

Kapitola 3

Implementace

3.1 Entity

Entitní třídy, reprezentující persistentní objekty, jsou implementovány pomocí balíků *emma.server.entity* a *emma.server.entity.ticketentry*. Jako poskytovatel persistence je využit framework Hibernate, pro anotování entitních tříd a proměnných jsou nicméně využity anotace balíku *javax.persistence* pro zajištění nezávislosti kódu na konkrétním poskytovateli persistence. Abstraktní třída *DatabaseEntry*, označená anotací *@MappedSuperclass*, tvoří předka všech ukládaných entit. Definuje identifikátor objektu typu *Long*, použitý jako primární klíč, označení verze entity typu *Timestamp*, a logickou hodnotu, reprezentující, zdali byla entita odstraněna. Identifikátor objektu je definován následujícím způsobem:

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Access(AccessType.PROPERTY)
Long id;
```

Anotace *@Id* označuje primární klíč objektu, dále je definována strategie generování primárního klíče, která využívá databázové sekvence, specifické pro každý entitní typ. Generování klíčů je tak ponecháno na databázi a u nových objektů je hodnota *id* před jejich persistencí nastavena na *null*. Poslední anotace definuje způsob přístupu persistenčního poskytovatele k jednotlivým proměnným třídy – zde je využit přístup k instančním proměnným objektu, tzv. vlastnostem (property) za pomocí přístupových metod *getId()* a *setId()*. Alternativní řešení za využití přímého přístupu k proměnným objektu za pomocí reflexe skýtá nevýhodu v nutnosti vyžádat entitu, referovanou z jiné entity, při přístupu k primárnímu klíči. Příkladem budiž entita *Ticket*, obsahující proměnnou author typu *User*. Chceme-li zjistit primární klíč autora, můžeme provést:

```
Ticket ticket = session.get(Ticket.class, 1);
Long authorId = ticket.getAuthor().getId();
```

Při využití reflexe k přístupu k primárnímu klíči je entita typu *User* z databáze vyžádána v okamžiku zavolání metody *getAuthor()*, přestože to není nutné – přistupujeme pouze ke klíči, který je uložen i v tabulce reprezentující entitu *Ticket* jako cizí klíč. Jak autoři knihy *Java Persistence with Hibernate* [21] k problematice uvádí: *„Pokud přistupujete pouze k vlastnosti tvořící databázový identifikátor, není zapotřebí inicializovat proxy objekt. (Toto není pravdou, pokud mapujete identifikátor pomocí přímého přístupu k proměnné; Hibernate pak ani neví, že metoda getId existuje. Pokud ji zavoláte, proxy musí být inicializována)“* (překlad vlastní). Je-li tedy využito mapování pomocí přístupových metod, Hibernate je schopen toto rozlišit a entitu typu *User* nevyžádá, dokud neproběhne přístup k jiným proměnným. To tak umožňuje zvýšit výkon aplikace pomocí minimalizace počtu databázových dotazů v případě práce pouze s primárními klíči.

Při práci s entitami je využito optimistické zamykání záznamů – při vyžádání entity není odpovídající databázový řádek uzamčen a může jej v průběhu editace uživatelem vyžádat jiný uživatel a provést úspěšnou editaci. Při uložení nové podoby objektu prvního uživatele by tak došlo k přepsání změn již provedených. Tomu je zabráněno pomocí proměnné *version*, označené anotací *@Version*, dedikující tuto proměnnou k využití poskytovatelem persistence k optimistickému zamykání. Proměnná obsahuje časovou známku konkrétní verze objektu – nesouhlasí-li při ukládání časová známka s aktuální hodnotou uloženou v databázi, byl záznam mezitím změněn a je vyvolána výjimka *javax.persistence.OptimisticLockException*.

Veškeré entitní třídy jsou potomkem třídy *TenantEntity*, která přidává odkaz na konkrétní společnost, ke které se záznam váže. Toto neplatí pro entitní typ *Permission*, jehož záznamy jsou univerzální pro všechny společnosti. Třída *TenantEntity* je potomkem třídy *DatabaseEntry*. Jednotlivé entity jsou označeny anotací *@Entity*, v případě kolize vyhrazenými názvy databázového stroje i anotací *@Table* s definovaným jménem. Veškeré časové údaje jsou označeny anotací *@Temporal(TemporalType.TIMESTAMP)* pro indikaci nutnosti uložení data i času. U textových proměnných, které mohou obsahovat více znaků než 255, je využit databázový typ *TEXT*. Hierarchie záznamů požadavku je realizována pomocí mapování do jedné tabulky, entitní třída je ale vytvořena pro každý typ záznamu vlastní. Konkrétní typy záznamu jsou potomky abstraktní třídy *TicketEntry*, která je definována takto:

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "DTYPE")
public abstract class TicketEntry extends TenantEntity {
```

Anotace nastavují metodu mapování všech potomků do jedné tabulky a určují jméno sloupce s hodnotou rozlišující typ záznamu. Entitní třídy konkrétních typů záznamů jsou označeny anotací *@DiscriminatorValue* s názvem typu.

3.2 Spring Data repozitáře

Kompletní implementace repozitářů je poskytnuta frameworkem Spring Data. Dodefinovány jsou pouze metody pro hledání dat dle požadovaných parametrů. Pro každou metodu, která vrací kolekci entit, je definována i metoda, vracející stránku entit typu *org.springframework.data.domain.Page<T>*, pro podporu stránkování ve výsledném dotazu a omezení nutnosti vracet z databáze všechny záznamy. Všechny repozitáře jsou označeny anotací *@Transactional* pro zajištění zpracování jednotlivých metod uvnitř transakce. V případě neúspěchu při některém kroku transakce je transakce automaticky zrušena a jsou anulovány všechny změny provedené touto transakcí.

3.3 Práce s e-maily

Pro práci z e-maily je využita knihovna JavaMail API, tedy balík *javax.mail*. Stahování e-mailů pomocí protokolu POP3 probíhá pravidelným připojením ke schránce a kontrolou nových e-mailů. Nejprve dojde ke stažení obálek (envelope) zpráv, obsahujících hlavičky zprávy, např. odesílatele, předmět, datum odeslání apod. Z důvodu neexistence unikátního identifikátoru zprávy ve schránce je pro každou zprávu vyžádán obsah hlavičky *Message-ID* a dojde k porovnání, zda daný identifikátor zprávy již je v databázi obsažen. Pokud ne, je zpráva předána komponentě *MessageProcessor* pro zpracování. Dle nastavení konkrétního e-mailového účtu je pak zpráva ze serveru odstraněna či je ponechána. Pro minimalizaci zátěže databáze při ponechávání zpráv na serveru, kdy by docházelo k identickým dotazům na *Message-ID*, je do komponenty stahující zprávy zavedena mezipaměť (cache), do které jsou ukládány již zpracované identifikátory *Message-ID*. Ty jsou při následujících kontrolách ignorovány. I tak může v případě vysokého počtu zpráv ve schránce (v řádu milionů) nastat výkonový problém z důvodu nutnosti stahovat všechny obálky zpráv při každé kontrole schránky a je vhodné pro takto rozsáhle schránky využít protokolu IMAP.

Protokol IMAP oproti tomu definuje unikátní identifikátor každé zprávy, označený *UID*. Tento identifikátor je unikátní v rámci konkrétní složky na serveru. Standard připouští možnost, že se *UID* zprávy v průběhu jejího životního cyklu změní – obvykle se tak děje v případě, kdy server ztratí vytvořené indexy, dojde k poškození složky či migraci na jiný server. Pro tento případ je pro každou složku definována hodnota *UIDValidity*, identifikující sérii vydávaných *UID*.

Prvotní kontrola schránky prostřednictvím protokolu IMAP probíhá podobným způsobem. Po připojení ke schránce je otevřena výchozí složka *INBOX* a jsou staženy obálky zpráv. Aplikace si uchovává u každého e-mailu dvojici *UIDValidity* a *UID* pro jeho identifikaci. Od otevřené složky je vyžádána hodnota *UIDValidity*, pokud se neshoduje s dřívější hodnotou *UIDValidity*, pak není na *UID* brán zřetel a je vyčištěna cache zpracovaných *UID*. V takovém případě jsou e-maily spárovány dle *Message-ID*, podobně jako u protokolu POP3. V opačném případě jsou staženy e-maily s *UID*, která ještě nejsou obsažena v systému.

Následně dojde k identifikaci podpory příkazu *IDLE* ze strany serveru. Příkaz umožňuje za pomocí déle trvajícího otevřeného spojení okamžitou notifikaci ze strany serveru na příchozí zprávy a tak rychlejší doručení zprávy aplikaci. Protože knihovna neposkytuje metodu zjišťující podporu příkazu *IDLE*, je tato zjištěna pomocí zaslání příkazu *IDLE* a sledování, zda dojde k výjimce, způsobené chybovou zprávou serveru. Je-li příkaz *IDLE* podporován, je okamžitě ukončen prostřednictvím vydání příkazu *NOOP*, reprezentující prázdnou operaci, aby nedošlo k blokování zpracovávajícího vlákna. Aktuální vlákno je vláknem plánovače úloh pro pravidelnou kontrolu zpráv (polling), která jsou využívána pro úlohy kontroly různých schránek, zatímco notifikace na nově příchozí zprávy (mail push) musí probíhat ve vlastním vlákně pomocí blokujícího volání. V případě podpory *IDLE* příkazu je tedy vytvořeno nové vlákno, které v cyklu iniciuje *IDLE* příkaz a v případě notifikace zajistí zpracování nové zprávy.

Není-li příkaz *IDLE* poštovním serverem podporován, dojde k odpojení od serveru a naplánování pravidelné kontroly, podobně jako v případě protokolu POP3. Následné kontroly složky však provedou pouze porovnání hodnoty *UIDNext*, značící, jaké *UID* bude serverem přiřazeno následující příchozí zprávě. Pokud je tato hodnota stejná jako při poslední kontrole, nedošlo k přijetí žádné zprávy a není tak nutné stahovat obálky existujících zpráv. To umožňuje kontrolovat i složky s téměř neomezeným množstvím zpráv. Pro maximalizaci kompatibility s různými poštovními servery je metoda *getUIDNext()* IMAP složky volána při zavřené složce [22].

Systém obsahuje dva plánovače úloh – *MailPollScheduler* a *MailIdleRefreshScheduler*. Oba obalují instanci třídy *ScheduledThreadPoolExecutor*, která vytváří skupinu vláken (thread pool), pomocí nichž je schopna vykonávat úlohy naplánované na určený čas. *MailPollScheduler* zajišťuje pravidelnou kontrolu schránek (polling), ať už využívajících protokolu POP3, či používajících protokol IMAP bez podpory příkazu *IDLE*. Druhý plánovač slouží k zasílání příkazu *NOOP* serverům, které jsou monitorovány pomocí příkazu *IDLE*. To je nutné z důvodu možnosti serveru odpojit klienta, vydávajícího *IDLE* příkaz v důsledku jeho domnělé neaktivity. Jak uvádí standard RFC 2177, popisující IMAP4 *IDLE* příkaz [23]: *„Server může považovat klienta za neaktivního, pokud má běžící IDLE příkaz, a pokud server obsahuje časový limit pro detekci neaktivity, může odhlásit klienta implicitně na konci časového*

limitu. Z tohoto důvodu je klientům, používajícím IDLE příkaz, doporučeno ukončit IDLE a znovu jej vydat přinejmenším každých 29 minut, aby bylo zabráněno jejich odhlášení. Toto stále umožňuje klientům přijímat okamžité aktualizace schránky i přes to, že se potřebují „dotazovat" pouze jednou za půl hodiny" (překlad vlastní). Plánovač tedy v pravidelných intervalech zasílá serveru příkaz *NOOP*, čímž demonstruje svoji aktivitu a zabrání tak svému odhlášení. Tím dojde ke zrušení *IDLE* příkazu, který je díky jeho cyklickému vydávání ihned obnoven. Interval je nastaven na 5 minut, z důvodu maximalizace kompatibility s různými e-mailovými servery – například poštovní server služby Gmail navzdory citovanému standardu provádí odpojení klienta již po cca 10 minutách domnělé neaktivity.

Využití těchto plánovačů zaručuje omezení maximálního počtu vláken, která budou zpracovávat úlohy daného typu, a znovuvyužití již existujících vláken, šetřící zdroje serveru aplikace. V případě, že počet vláken nedostačuje k provádění všech naplánovaných úloh v daný čas, jsou úlohy vykonávány v pořadí plánu zpožděně, tedy interval mezi jednotlivými kontrolami schránky se zvětšuje. Toto řešení je preferováno oproti nekontrolovanému počtu nárůstu vláken, vedoucího k vyčerpání zdrojů a prudkého růstu doby odezvy. Pro účely kontroly zpráv existuje maximálně $p+r+i$ vláken, kde p je maximální počet vláken plánovače *MailPollScheduler*, r je maximální počet vláken plánovače *MailIdleRefreshScheduler* a i je počet vláken s vydaným příkazem *IDLE*, obvykle odpovídající počtu konfigurovaných schránek v systému s podporou *IDLE* příkazu.

Komponenta *MessageProcessor*, zpracovávající přijaté zprávy, provádí extrakci obvyklých hlaviček (odesílatel, adresy příjemců, předmět, datum odeslání, adresu pro odpověď, *Message-ID* a další) a obsahu zprávy. Obsah zprávy může být tvořen přímo typem *text/plain* či *text/html*, ale i tělem o více částech pomocí typu *multipart*. Algoritmus prochází jednotlivými části *multipart* zprávy rekurzivně a dle konkrétního podtypu (*multipart/mixed*, *multipart/alternative* nebo *multipart/signed*) určí souřadnost či prioritu konkrétního typu obsahu. Komponenta také dokáže zpracovávat přílohy zprávy. Všechny vyextrahované údaje jsou uloženy do entity typu *Email*, případně dojde k vytvoření entit *Attachment*. Pro případné další využití je uložena i kompletní podoba hlaviček zprávy.

Přiřazení zprávy k již existujícímu požadavku je prováděno pomocí hlavičky *In-Reply-To*, v případě její nepřítomnosti dle hlavičky *References*. Hlavička *In-Reply-To* obsahuje identifikátor e-mailové zprávy (*Message-ID*), na kterou zpráva odpovídá. Hlavička *References* obsahuje identifikátory zpráv, na které zpráva odkazuje; obvykle ji tvoří část či všechny zprávy odeslané oběma stranami v rámci této konverzace. Dle extrahovaných identifikátorů jsou vyhledány odeslané i přijaté zprávy s těmito identifikátory a zpráva je přiřazena k tomuto požadavku. V případě, že v konfiguraci společnosti není povoleno znovuotevírat jednou uzavřené požadavky a požadavek je již vyřízen, je vytvořen požadavek nový. To umožňuje společnostem indiviualizovat si proces

vyřizování požadavků dle vlastních preferencí. Ze stejného důvodu je možné konfigurovat, zda požadavek při jeho otevření má být přiřazen původnímu zpracovateli, ponechán přiřazené skupině či přiřazen skupině, která odpovídá e-mailovému účtu, na který byla zpráva přijata.

3.4 Autentizace a autorizace

Identita uživatele je ověřována a autentizace prováděna pomocí JWT tokenu, přiloženého ke každému HTTP požadavku. Zabezpečeny jsou všechny zdroje, s výjimkou adres */* a */login*, které jsou dostupné bez přihlášení. Tyto výjimky jsou definovány v konfiguračním souboru *security-config.xml*, spolu s dalšími bezpečnostními nastaveními, jako je nastavení poskytovatele autentizace na komponentu *JwtAuthenticationProvider* a nastavení autentizačního filtru. Autentizační filtr zajistí extrahování JWT tokenu, jeho dekódování a ověření podpisu, a případné dodatečné autentizační kontroly před vytvořením instance, reprezentující autentizovaného uživatele. Není-li požadavek tímto filtrem odmítnut, pokračuje v řetězci zpracování na jednotlivé kontroléry.

Postup vydání autentizačního tokenu spočívá v zaslání požadavku typu POST na adresu */login*, s parametry *login* a *password*. Heslo je přeneseno v textové podobě, komunikace spoléhá na šifrování díky zabezpečenému spojení protokolem HTTPS. Server ověří poskytnuté údaje proti databázi a vytvoří a vrátí JWT token v případě úspěchu, či stavový kód *401 Unauthorized* při poskytnutí nesprávných údajů. Hesla v databázi jsou uložena v podobě výsledku šifrovací funkce pro odvození klíče (key derivation function) Bcrypt. Ta je odvozena od symetrické šifry Blowfish a je vhodná k bezpečnému ukládání hesel. Její výhodou je možnost dynamicky měnit složitost výpočtu pomocí nastavení počtu iterací výpočtu. Výsledek výpočtu má v sobě také zabudovanou sůl pro zamezení útoku pomocí předpočítaných tabulek klíčů pro jednotlivé řetězce, zvané rainbow tables. Oproti obvyklým hašovacím algoritmům, jako je rodina funkcí SHA-2, je funkce Bcrypt v obvyklém nastavení o několik řádů složitější na výpočet. To je vhodné pro zvýšení odolnosti proti útokům hrubou silou – i kvatlitní hašovací funkce, pro níž není známá žádná zranitelnost, nemusí být dostatečně odolná kvůli vysoké rychlosti výpočtu, zvláště při použití grafických karet. Výpočetní stroj s osmi grafickými kartami nVidia Titan X dosahuje při výpočtu hašovací funkce SHA256 hodnoty více než 14 miliard vypočtených hašových kódů za sekundu [24]. Díky tomu dokáže vypočíst hašový kód pro všechna alfanumerická hesla anglické abecedy o délce 8 znaků během čtyř hodin. Oproti tomu funkce Bcrypt bývá obvykle konfigurována tak, aby běžný počítač byl schopen vypočíst pouze desítky hodnot za sekundu. Vzhledem k jejímu výpočtu aplikací pouze při přihlášení uživatele není vyšší výpočetní náročnost problematická.

JWT token je vytvořen v následujícím formátu:

```
{
  "alg": "HS512"
}
{
  "sub": "john.doe\emph{@domain.com",
  "iat": 1461434729,
  "iss": "EMMA 1.0.0 Server 1",
  "exp": 1463234729,
  "cid": 1,
  "permissions": [
    "get_UserResource",
    "list_UserResource"
  ]
}
```

Význam jednotlivých položek je následující:

sub uživatelské jméno reprezentovaného uživatele

iat časová značka vystavení tokenu

iss vydavatel tokenu; obvykle spočívá v názvu a verze aplikace a označení vydávajícího serveru

exp časová značka okamžiku expirace tokenu

cid identifikátor společnosti, do které uživatel náleží

permissions kolekce oprávnění uživatele

Tento token je poté podepsán privátním klíčem, který je společný pro všechny aplikační servery aplikace. Při přijetí tokenu v následných požadavcích aplikace ověří platnost podpisu, díky čemuž může věřit datům obsaženým v tokenu (byla vydána serverem se znalostí privátního klíče a nebyla uživatelem či útočníkem pozměněna). To umožňuje ověřit autenticitu uživatele bez nutnosti ověřovat jeho údaje v databázi. Také to serveru poskytuje možnost uložit si do tokenu informace pro zjednodušení manipulace s daty – zde jde o identifikátor společnosti a kolekci oprávnění uživatele. Ani tyto informace tak nemusí být vyžádány z databáze. Poskytovatel autentizace (*JwtAuthenticationProvider*) zařídí vytvoření oprávnění (objektů typu *org.springframework.security.core.GrantedAuthority*) z obsahu JWT tokenu po pozdější provedení autorizace.

Implementace serverové aplikace nicméně umožňuje zapnout ověřování všech údajů tokenu dle aktuálních hodnot v databázi. To je vhodné za situace, kdy je nutné dynamicky reagovat na změny uživatelského účtu. Dojde-li k editaci uživatele, například úrovně jeho oprávnění, a hodnoty nejsou ověřovány, pak

se provedené změny neprojeví, dokud uživatel používá starší vydaný token, tj. po dobu aktuální relace. Je-li nutné tomu zamezit, je možné povolit ověřování hodnot za cenu nutnosti provádět více databázových dotazů při každém požadavku. Ze stejného důvodu je token vydáván s platností pouhých třiceti minut, po kterých klientská aplikace (bez nutnosti spolupráce uživatele) iniciuje nový požadavek na přihlášení a je jí vygenerován token aktuální.

Alternativním přístupem by bylo zařadit token uživatele, jehož údaje se změnily, na černou listinu a vynutit tím nové přihlášení. Tato listina by však musela být replikována mezi všemi instancemi serveru aplikace a došlo by k potlačení výhod bezstavového serveru.

Dříve než proběhne ověření podpisu JWT tokenu není žádným informacím v tokenu obsažených důvěřováno. Principiální výjimkou z tohoto pravidla musí být informace o podpisovém algoritmu v hlavičce tokenu, která je nutná pro ověření platnosti podpisu. Implementace proto musí odmítat takové tokeny, jejichž použitý algoritmus podpisu není dostatečně bezpečný. Specifikace JWT umožňuje nastavit hodnotu podpisového algotmu na *'none'*, ověření takových tokenů pro zajištění bezpečnosti musí skončit neúspěchem [25]. Do tokenu také nejsou ukládány žádné citlivé údaje, které by měly zůstat před klientem skryty – podoba tokenu není zašifrovaná a klient je schopen zobrazit obsažené údaje. Dále může být vhodné zajistit, aby nebyl vstupem podpisové funkce vždy identický, předem známý řetězec; to je v případě JWT tokenu zajištěno pomocí data expirace generovaného serverem při sestavování tokenu.

Autorizace je založena na omezení přístupu ke konkrétním metodám pomocí anotací *@PreAuthorize*, *@PreFilter*, *@PostAuthorize* a *@PostFilter* jejichž parametrem je výraz pro ověření uživatelských oprávnění, například *@PreAuthorize("hasPermission(id, 'TicketResource', 'get')")*. Před/po volání metody s některou z těchto anotací framework Spring Security ověří, zda uživatel má dostatečná oprávnění pro její invokaci. Pro vyhodnocení této kontroly je vytvořena třída *ResourcePermissionEvaluator*, implementující rozhraní *PermissionEvaluator*, poskytnutého frameworkem. Tento přístup má oproti zabezpečení pomocí anotace *@Secured* s názvem role výhodu ve vyšší granularitě využitých pravomocí – zabezpečené třídy nejsou vázané na logické role, při jejichž změně či přidání je nutné editovat kód zabezpečených tříd, ale pouze na pravomoc, která je typicky platná pouze pro danou metodu. Mapování na role je tak odděleno do třídy *ResourcePermissionEvaluator*, která provádí samotné ověření. Druhou výhodou přístupu je možnost vyhodnocovat oprávnění nejen dle metody, kterou klient volá, ale i dle objektu, který ji předává jako parametr či který metoda vrací. Tím může být jednoduše zajištěno, aby uživateli bylo umožněno např. editovat požadavky, které jsou mu přiřazeny, ale odmítnout upravit ostatní, aniž by tato kontrola probíhala v aplikačním kódu. Podobně lze anotací *@PostFilter* filtrovat zdroje při výpisu seznamu všech zdrojů daného typu – uživateli mohou být poskytnuty pouze ty, ke kterým má přístup; tento mechanismus ale není v aplikaci využíván, vzhledem k narušení stránkování při následné filtraci.

3.5 Webové služby

Realizace webových služeb se skládá z kontroléru, tříd reprezentujících REST zdroje a komponent provádějících transformaci databázových entit na tyto zdroje a sestavování odkazů. Kontrolér je tvořen třídou, jejíž definice má následující podobu:

```
@Controller
@RequestMapping(produces = "application/hal+json")
@ExposesResourceFor(RoleResource.class)
public class RoleController {
```

Uvedené anotace definují kontrolér, na který můžou být směrovány klientské požadavky, a definují výstupní formát HAL. Anotace *@ExposesResourceFor* je poskytována frameworkem Spring HATEOAS a určuje, že kontrolér poskytuje REST zdroje definovaného typu. Systém je pak schopen vytvářet odkazy na odpovídající zdroje při referování k metodám této třídy. Jednotlivé hlavičky metod kontroléru mají obvykle tuto syntaxi:

```
@RequestMapping(value="/roles/{id}", method=RequestMethod.GET)
@ResponseBody
@Transactional
@PreAuthorize("hasPermission(#id, 'RoleResource', 'get')")
public ResponseEntity<RoleResource> get(@PathVariable Long id,
    @AuthenticationPrincipal AuthenticatedUser user) {
```

Anotace *@RequestMapping* nastavuje mapování na konkrétní URL a využitou HTTP metodu, *@ResponseBody* definuje, že návratová hodnota metody má být využita jako tělo odpovědi na požadavek, *@Transactional* zajišťuje, že celé volání metody je provedeno v jedné transakci a *@PreAuthorize* ověřuje dostatečná oprávnění uživatele, jak bylo diskutováno výše. Argumenty metod díky vkládání závislostí obsahují parametry uvedené v URL a autentizovaného uživatele provádějícího požadavek. Podobně je pro HTTP metody obsahující tělo požadavku toto tělo injektováno jako argument metody za pomocí anotace *@RequestBody* v podobě objektu třídy reprezentující REST zdroj. Podporované HTTP metody odpovídají specifikaci REST a tvoří odlišnou množinu pro adresy reprezentující kolekce a jednotlivé zdroje. Pro kolekce jsou definovány obvykle následující metody:

GET vrátí seznam požadovaných zdrojů, v podobě stránky s n prvky. Ve výchozím nastavení odpovídá hodnota n = 20

POST vytvoří nový záznam v dané kolekci, odpovědí je kompletní reprezentace vytvořeného zdroje

Pro zdroje jsou poskytovány tyto metody:

GET vrátí reprezentaci požadovaného zdroje

PUT upraví zdroj na dané adrese kompletní reprezentací zdroje poskytnuté při volání metody. Přestože se jedná o nahrazení všech hodnot zdroje, nejedná se o nahrazení zdroje samotného a jsou tak vyžadována integritní omezení: nelze například změnit uživatelské jméno uživatele, které je neměnné po celý životní cyklus zdroje. Při opomenutí některé z položek zdroje je předpokládána výchozí hodnota *null*. Operace není definována pro zdroje, které nejsou editovatelné.

PATCH upraví zdroj na dané adrese nekompletní reprezentací zdroje poskytnuté při volání metody. Nahrazeny jsou všechny hodnoty, které jsou v požadavku přítomny a nejsou rovny hodnotě *null*. Protože aplikace ztotožňuje hodnoty *null* s nepřítomnými hodnotami, nelze pomocí metody PATCH nastavovat hodnoty položek na *null* – v takovém případě je nutné provést operaci PUT. Operace není definována pro zdroje, které nejsou editovatelné.

DELETE odstraní zdroj na dané adrese. Sémantika odstranění může být odlišná pro jednotlivé zdroje, v některých případech může dojít k pouhému označení zdroje jako smazaného. Konkrétní implementace pro každý zdroj je definována specifikací API v Příloze B. Operace není definována pro zdroje, které nejsou editovatelné.

Pojmenování adres jednotlivých zdrojů dodržuje pravidla, navržená v [26], zejména jsou všechny adresy pojmenovány v plurálu, v adresách nejsou využita žádná označení operací a vytváření závislých zdrojů je umožněno pouze přes zdroj, ve kterém jsou obsaženy, např. vytvoření záznamu požadavku probíhá pomocí volání metody POST na adrese */tickets/1/ticketentries*.

Zdroje, které vůči sobě mají vztah *m:n*, není možné vytvářet pomocí podobného volání. Pro zamezení nutnosti definovat specifické URL, zajišťující přiřazení zdroje k jinému, které by porušovalo sémantiku definovanou standardem REST, jsou v takovýchto zdrojích přítomny kolekce odkazů na přiřazené zdroje pro každou relaci. Klient poté provádí přiřazení (či jej ruší) pomocí editace zdroje s jiným obsahem kolekce. Aby nebylo nutné propagovat změnu verze zdroje oběma entitám tvořícím vztah, je vždy jeden ze zdrojů označen jako vlastnící strana vztahu. Editace vztahu pak může probíhat jen pomocí editace tohoto zdroje. Například přiřazení uživatele do skupiny je tak možné provést pouze pomocí editace uživatele a přidání odkazu na skupinu do jeho kolekce *managedGroups*; není možné přiřazení uživatele provést editací skupiny. Kompletní specifikace navrženého rozhraní, včetně všech poskytovaných zdrojů, jejich adres a podporovaných metod, je uvedena v Příloze B.

Dále jsou (nezávisle na typu zdroje či kolekce) definovány HTTP metody HEAD a OPTIONS. Metoda HEAD provede identické volání jako GET, ovšem nevrátí žádné tělo. V souladu se standardem jsou všechny hlavičky, včetně

Content-Length, identické jako při odpovídajícím volání metody GET. Realizace této metody je provedena filtrem *HttpHeadFilter*, kterým prochází každý HTTP požadavek, a jehož implementace při volání metody HEAD převede volání na metodu GET, a výsledné tělo odpovědi zahodí. Implementace tak nešetří žádné zdroje serveru, požadavek musí být obsloužen a kompletní odpověď stále musí být vytvořena. Dochází nicméně k úspoře síťových přenosů a není nutné podporu HEAD v aplikačním kódu explicitně přidávat.

Univerzální podporu metody OPTIONS nelze realizovat pomocí filtru, který by vyžádal podporované metody pro konkrétní adresu, mechanismus vrácení podporovaných metod není frameworkem Spring aplikaci poskytován. Přesto v něm samotný algoritmus pro jejich zjištění obsažen je – v případě dotazu pomocí metody, která není podporována, server vrátí v rámci chybové stránky *405 Method Not Allowed* v hlavičce *Allow* korektně určený seznam podporovaných metod. Toho aplikace využívá pomocí vlastní implementace servletu – třídy *AutomaticOptionsServingServlet*. Ta je využita jako hlavní servlet aplikace a překrývá metodu *doOptions()* třídy *DispatcherServlet*. V rámci její implementace volá metodu *doOptions()* rodičovské třídy, která v případě, že metoda OPTIONS není explicitně aplikací implementována, vyvolává výjimku *HttpRequestMethodNotSupportedException*. Tato výjimka obsahuje metodu *getSupportedHttpMethods()*, vracející seznam podporovaných metod. Implementace tedy zachycuje tuto výjimku a do hlavičky *Allow* odpovědi vyplňuje hodnoty vrácené touto metodou. Tím je dosažena globální podpora metody OPTIONS napříč aplikací bez nutnosti specifikovat podporované metody pro jednotlivé adresy.

Technologie také podporuje jednoduchou práci s chybovými stavy při zpracování požadavku. Kontrolér (či libovolný kód z něj volaný) může vyvolat uživatelsky definovanou výjimku. Pokud je třída této výjimky označena anotací *@ResponseStatus*, definující HTTP stavový kód odpovědi a chybovou zprávu, pak framework při vyvolání výjimky automaticky vytvoří tělo odpovědi s uvedenou zprávou a stavovým kódem. To umožňuje nastavení chybových kódů a zpráv specifických pro HTTP požadavky centralizovat a oddělit od aplikační logiky kontroléru.

Posledním článkem implementace webových služeb jsou komponenty, transformující databázové entity na objekty reprezentující REST zdroje, tzv. resource assemblers. Pro každý zdroj je implementována vlastní komponenta, která vytvoří nový objekt reprezentující zdroj, nastaví mu odkaz na něj samotný a nastaví všechny odpovídající proměnné dle vzorové entity. Vztahy k jiným entitám jsou reprezentovány pomocí odkazů na odpovídající zdroje. Komponenta také může zajišťovat vnořování jiných zdrojů.

Vytváření odkazů probíhá pomocí referencí na kontrolér, poskytující daný zdroj. Například odkaz na zdroj samotný v případě sestavování zdroje *EmailAccountResource* komponentnou *EmailAccountResourceAssembler* proběhne pomocí volání:

```
resource.add(linkTo(
    methodOn(EmailAccountController.class).get(
        entity.getId(), null)
).withSelfRel());
```

Metody *linkTo()* a *methodOn()* jsou staticky importované ze třídy frameworku *org.springframework.hateoas.mvc.ControllerLinkBuilder*. Metoda *methodOn()* vytvoří proxy instanci pro uvedenou třídu, na níž je možné „volat“ odpovídající metodu třídy. Takovéto volání je ovšem proxy instancí zachyceno a není provedeno, pouze je vytvořena návratová hodnota, která umožňuje metodě *linkTo()* určit adresu, na kterou je daná metoda mapována. Metoda *withSelfRel()* či *withRel()* specifikuje logickou relaci, kterou odkazovaný zdroj s aktuálním zdrojem tvoří. V konkrétním případě by tak byl vložen odkaz na metodu *get()* třídy *EmailAccountController* s parametrem *id* zpracovávané entity, v relaci *self*, tedy například */emailaccounts/1*. Zdroje nicméně mohou být libovolně zanořené a resource assembler díky mechanismu není vázán na konkrétní URL a v případě změny URL je potřeba pouze upravit mapování kontroléru a není potřeba činit žádné změny v ostatních komponentách a metodách sestavujících odkazy. Klíčovým bodem, využívajícím princip HATEOAS, je schopnost komponent vkládat pouze takové odkazy, které jsou v konkrétní situaci pro daný zdroj relevantní.

3.6 Konfigurace

Konfigurace aplikace se skládá z následujících souborů:

pom.xml definuje informace o Maven artefaktu a závislosti na ostatních modulech a knihovnách

webapp/WEB-INF/web.xml hlavní konfigurační soubor pro webovou část aplikace (REST služby), obsahuje informace o webové aplikaci, odkazuje na ostatní konfigurační soubory, nastavuje třídu použitého servletu a jeho parametry; definuje jednotlivé filtry pro zpracování požadavků (např. pro Spring Security či automatickou podporu metody HEAD).

webapp/WEB-INF/config/application-config.xml obsahuje obecné nastavení aplikace, jako je vymezení balíků, ve kterých má probíhat sken komponent pro vkládání závislostí a odkaz na jiné konfigurační soubory

webapp/WEB-INF/config/dao-config.xml definuje balík s repozitáři, EntityManager, neboli komponentu pro manipulaci s persistentním úložištěm, správce transakcí, persistentní kontext a datový zdroj

webapp/WEB-INF/config/security-config.xml obsahuje nastavení týkající se zabezpečení, autentizace a autorizace; jedná se především o de-

finování filtru pro zpracování JWT, poskytovatele autentizace, konfigurace třídy vyhodnocující oprávnění, vynucení využití zabezpečeného spojení a nastavení výjimek pro kořenovou a přihlašovací adresu.

webapp/WEB-INF/config/servlet-config.xml hlavní konfigurační soubor servletu; obsahuje konfiguraci neprovedenou specializovanými konfiguračními soubory

resources/META-INF/persistence.xml definuje persistentní kontext (entity, které mají být ukládány), persistentního poskytovatelel a vlastnosti nutné pro přístup k databázi.

Protože konfigurace některých parametrů, zvláště u nových projektů, jako je Spring HATEOAS, se provádí pomocí XML souborů problematicky a neexistují odpovídající alternativy konfigurace pomocí anotací, je vytvořena třída *AppConfig*, označená anotací *@Configuration*. Třída se omezuje na anotace povolující a autokonfigurující jednotlivé technologie, obsahuje tak anotace *@EnableWebMvc*, *@EnableSpringDataWebSupport*, *@EnableHypermediaSupport*, *@EnableEntityLinks* a *@EnableWebSecurity*. Také nastavuje výchozí formát komunikace REST služeb. Projekt v ostatních případech využívá konfigurační anotace jenom tehdy, když mohou být obsaženy přímo v konfigurované komponentě.

Kapitola 4

Nasazení

4.1 Požadavky

Jedním ze základních nefunkčních požadavků na aplikaci je zajištění vysoké dostupnosti a odolnosti aplikace v případě havárie. Hlavní podmínkou splnění tohoto požadavku je, aby veškerá uživatelská data systému byla replikována na více fyzických serverech a aby byla dostupná i v takovém okamžiku, kdy je některý ze serverů data obsahujících nefunkční či nedosažitelný. Dále musí být za každých okolností dostupná samotná aplikace pro zprostředkování těchto dat. Nesmí existovat žádný bod, mající roli při vyřizování požadavku, jehož nedostupnost by způsobila nedostupnost celého systému. Toho je docíleno pomocí redundance kritických serverů a služeb. Zároveň však musí být zajištěna integrita všech datových replik pomocí adekvátní koordinace přístupu k datům a měly by být přítomny mechanismy pro uvedení dat do konzistentního stavu po odstranění chybového stavu serveru. Kvalitní řešení vysoké dostupnosti by mělo zabraňovat nejen výpadku v případě havárie, ale mělo by také umožňovat systém bez výpadku spravovat, udržovat a aktualizovat. Protože výpadek může být způsoben i selháním síťové konektivity či infrastruktury datového centra, je pro zajištění skutečné vysoké dostupnosti nutné provozovat systém ve více geograficky oddělených datových centrech, nesdílejících žádné součásti infrastruktury. To přináší problém s oddělenými síťovými segmenty (network partitioning), kdy mohou být například ze stanice klienta dostupné všechny servery systému, ale spojení mezi nimi samotnými nemusí existovat. Korektně navržená aplikace je schopna zajistit fungování i na takto odděleném systému a zabránit vzniku konfliktů spojených s potenciálními změnami dat na více serverech najednou. Součástí zajištění vysoké dostupnosti by měl být i mechanismus detekce chyb, upozorňující administrátora systému na nestandardní chování některých komponent systému.

Pro účely vyvíjené aplikace je důležité dosáhnout takového stavu systému, aby se zápis na jeden server projevil na ostatních redundantních serverech v co nejkratším časovém intervalu pro minimalizaci ztráty dat v případě havárie,

nicméně bez nutnosti čekat na zápis všemi servery synchronně před potvrzením prováděné operace, aby bylo možné udržet krátkou dobu odezvy služby. V případě havárie některého ze serverů musí být všechna data systému nadále dostupná bez nutnosti rekonfigurovat zbylé součásti systému, ideálně s nulovým dopadem na výkon systému. Po zprovoznění havarovaného serveru by mělo řešení pokud možno automaticky obnovit konzistenci a synchronicitu dat mezi uzly.

4.2 Architektonické možnosti

Vysokou dostupnost systému je nutné zajistit na všech vrstvách provozované aplikace, tedy je třeba zavést redundantní prvky na úrovni databázového i aplikačního serveru. Dále je nutné v případě využití komponent zajišťujících rozložení zátěže (load-balancer) duplikovat i tyto komponenty. Přesto je aplikace na jednotlivých vrstvách možné rozdělit na ty, které aktivně zajišťují a koordinují replikaci dat, a ty, které znalost a logiku pro replikaci nepotřebují.

4.2.1 Replikace na úrovni aplikace

Replikace dat může být kompletně zajištěna a spravována nasazenou aplikací. Aplikace může přeposílat veškerá data a požadavky všem ostatním instancím aplikace a žádná nižší vrstva nemusí podporovat žádný replikační mechanismus a nemusí si být vědoma toho, že se data nacházejí i na jiném serveru. Tento replikační mechanismus potřebuje rozsáhlou podporu aplikace, která s tímto požadavkem musí být navrhována, a vyžaduje časové investice na straně vývojářů aplikací. V důsledku rozsáhlosti typických aplikací může být toto řešení náchylné k chybě či implementace mechanismu datové konzistence může zvyšovat odezvy a snižovat maximální počet vyřízených požadavků za časový úsek. V případě chyby komponenty na kterékoli vrstvě systému obvykle dojde k odstavení celé horizontální instance aplikace, např. selhání databáze způsobí znefunkčnění této instance aplikační i prezentační vrstvy, protože mezi nimi existuje vztah 1:1, tj. v každé instanci aplikace je nakonfigurován přístup k pouze jednomu databázovému serveru.

4.2.2 Replikace na úrovni aplikačního serveru

Některé aplikační servery obsahují podporu clusterování, tedy vytvoření skupin aplikačních serverů, na nichž běží různé instance stejné aplikace. Cílem je zajistit redundanci aplikace bez nutnosti aplikaci upravovat pro toto použití. Naplnění tohoto cíle se liší dle použitého aplikačního serveru a rozsahu požadovaných služeb, obvykle je ale nutné aplikaci do jisté míry modifikovat pro běh v takovémto prostředí. Implementace clusterování v aplikačních serverech sahá od předřazení komponenty zpracovávající požadavky, která je směruje na různé instance serverů, které nemají žádnou znalost o ostatních instancích, až

po vysoce propojené servery, mající přístup ke stavu a objektům na ostatních serverech [27]. Výhodou využití tohoto prostředí je možnost implementace stavové (stateful) aplikace, protože i uživatelská relace může být clusterovacím mechanismem replikována na ostatní servery. Clusterované aplikační servery se také prezentují vysokou a jednoduchou škálovatelností aplikace, obsahují zabudované load-balancery a jsou schopny automaticky řešit výpadek některé instance. Nevýhodou je různící se rozsah podpory jednotlivých funkcí aplikačními servery a vysoká cenová a konfigurační náročnost pokročilých řešení. Chyba v implementaci či konfiguraci clusteru může mít za následek nedostupnost celé aplikace, v případě přístupu ke sdíleným zdrojům může též dojít k uváznutí aplikace při čekání na tyto zdroje, zvláště při použití globálního správce daného zdroje. Pro bezproblémové využití clusterování jsou nutné rozsáhlé znalosti konkrétního aplikačního serveru ze strany administrátora clusteru i vývojáře aplikace.

4.2.3 Replikace na úrovni databáze

Data je možné replikovat na úrovni databázového stroje. Podporu pro databázovou replikaci poskytují všichni hlavní výrobci tradičních relačních databází, v případě NoSQL databází byl požadavek vysoké dostupnosti zohledněn mnohdy už při jejich vzniku a vychází z podstaty jejich návrhu. Využití replikace databází se liší dle míry konzistence dat, kterou je potřeba zajistit. V případě synchronní replikace není potřeba žádné podpory aplikace pro replikaci či zajištění integrity. Pro případ asynchronní replikace je nutné zajistit některá omezení na úrovni aplikace, nebo musí být aplikace schopná zpracovávat data s porušenou integritou. Využívá-li aplikace jiného úložiště než databázi, například souborový systém, je nutné zajistit dostupnost tohoto úložiště separátně.

4.2.4 Replikace na úrovni operačního systému

Za pomocí virtualizace je možné spustit více instancí operačního systému na jednom fyzickém stroji. V rámci každého operačního systému běží nezávislý aplikační server s instancí aplikace. V případě selhání instance aplikace, aplikačního serveru či operačního systému běží druhá instance aplikace nedotčeně. Pro směrování požadavku na odpovídající instanci je nutná aplikační podpora či komponenta směrování zajišťující. Toto řešení nechrání před dopady selhání hardware a samo o sobě není schopné zajistit vysokou dostupnost služby. Může ovšem zajistit jednoduší aktualizovatelnost a údržbu aplikace nebo prostředí či vyšší dostupnost v případě výskytu softwarových chyb.

4.2.5 Replikace na úrovni úložiště

Téměř každý server využívá replikaci dat na úrovni fyzického úložiště zcela transparentně pro spuštěné aplikace, ať už pomocí RAID pole u fyzických

disků serveru, či na úrovni SAN sítí a diskových polí. Takové řešení chrání proti hardwarovému selhání disku či úložiště. Odolnost proti selhání ostatních komponent je nutné zajistit na jiné vrstvě systému.

4.3 CAP teorém

Při vytváření každého distribuovaného systému je nutné mít na paměti, že jednotlivá řešení v jistých situacích musí obětovávat konzistenci dat pro dosažení vysoké dostupnosti či naopak. O tom pojednává CAP teorém, který tvrdí, že není možné v distribuovaném systému najednou zajistit konzistenci dat (Consistency), dostupnost (Availability) a odolnost proti rozdělení síťových segmentů (Partition tolerance). Konzistence dat je definovaná tak, že každý požadavek na čtení obdrží poslední zapsanou hodnotu. Dostupnost v tomto kontextu pojednává o tom, že každý uzel (s výjimkou jeho selhání) je schopen vyřizovat požadavky. Odolnost proti rozdělení síťových segmentů říká, že ostatní vlastnosti systému nejsou ovlivněny výpadkem spojení mezi uzly. Navzdory obvyklému výkladu, tvrdícímu, že je nutné zvolit libovolné dva z těchto parametrů, se nabízí skutečná volba pouze mezi konzistencí a dostupností, vzhledem k tomu, že síťovým výpadkům není možné zabránit a bez schopnosti tolerovat výpadek spojení se nejedná o distribuovaný systém [28].

4.4 Replikace databáze

Existuje mnoho principů pro škálování a replikaci databází. Řešení je možné rozdělovat pomocí okamžiku přenosu dat, směru přenosu dat, podoby přenášených informací a množství sdílených prostředků.

4.4.1 Okamžik přenosu dat

Způsob replikace může být synchronní, kdy jsou data přenášena z jednoho serveru na ostatní nejpozději v okamžiku potvrzení (commit) transakce, a klientovi není provedení transakce potvrzeno, dokud její provedení nepotvrdí všechny replikované servery. Alternativou k tomuto přístupu je replikace asynchronní, kdy jsou transakce potvrzeny hned po jejich provedení jedním serverem, a k předání informace ostatním instancím může docházet až poté – ať už co nejdříve, nebo dávkově po dosažení určité velikosti přenášených informací či dosažení časového intervalu.

V případě synchronního přenosu je zajištěna konzistence všech replik; všechny instance mají v každém okamžiku potvrzena stejná data a klient dostane stejný výsledek, ať provede dotaz na libovolné databázi (za předpokladu, že neselhalo spojení mezi databázemi). Při využití asynchronní replikace existuje časové okno, kdy každá databáze obsahuje jiná data a při editaci stejných dat více databázovými servery může docházet ke konfliktům.

4.4.2 Směr přenosu dat

Data mohou být přenášena pouze v jednom směru, od jedné hlavní databáze k jejím replikám (master-slave replikace). To zajišťuje udržování aktuálních kopií originální databáze. Tyto kopie mohou být použity pro provádění dotazů nemodifikujících data, veškeré úpravy dat jsou prováděny na hlavní databázi. V případě havárie hlavní databáze je kopie uvedena do stavu pro čtení i zápis a zastává roli hlavní databáze. Po zprovoznění původního systému je nutné databázovým administrátorem obnovit data na havarované databázi z aktuální instance a poté vrátit databázím jejich původní role. Kopie databáze také může sloužit pro provádění déle trvajících analytických a statistických dotazů nad transakčními daty bez zatěžování produkční databáze.

Alternativou je obousměrný přenos dat (master-master či multi-master replikace), kdy mají všechny databázové instance stejné postavení. Každý databázový stroj si obvykle udržuje spojení se všemi ostatními (full mesh) a zasílá jim všechny provedené změny. V případě editace stejných dat různými stroji dochází ke konfliktům, které nemusí být replikačním mechanismem řešitelné, je tedy na aplikaci, aby takovýmto situacím zabránila či byla ochotna akceptovat jistou ztrátu dat, nebo transakce jiným způsobem koordinovat. V případě výpadku není nutný žádný administrátorský zásah a v okamžiku zprovoznění havarované databáze může být tato databáze také automaticky uvedena do konzistentního stavu bez nutnosti manuálních úkonů.

4.4.3 Podoba přenášených informací

Mezi databázemi je možné přenášet data v podobě fyzické, obsahující skutečně změněné datové bloky, nebo v podobě logické, reprezentující příkazy, které mají být nad databází vykonány. Logická podoba je reprezentována pomocí příkazů v jazyce SQL či změnových vektorů a cílová databáze je vykonává nezávisle. Komunikace ve fyzické podobě může probíhat prostřednictvím přenosu diskových bloků nebo řádků tabulek, které jsou přímo aplikovány na cílové databázi. Další možností je přenášení WAL logu. WAL (Write-ahead Log, různí výrobci databází jej označují odlišně) je soubor, do kterého jsou zapsány změny, prováděné každým příkazem transakce dříve, než jsou provedeny v samotném datovém souboru tabulky. Vynucení zápisu WAL logu na disk před každým potvrzením transakce zajišťuje persistenci provedených změn i v případě okamžité havárie databáze po jeho zápisu a umožňuje rychlejší potvrzení transakcí oproti nutnosti čekat na zapsání změn do datových souborů. Tento log může obsahovat fyzické i logické údaje, dle výrobce databáze i aktuální konfigurace.

4.4.4 Množství sdílených prostředků

Hlavní odlišností v míře sdílených prostředků mezi jednotlivými řešeními je otázka přítomnosti centralizovaného koordinátora transakcí. Ten zajišťuje glo-

bální zamykací mechanismus a správu transakcí pro všechny databázové stroje v clusteru. V případě jeho přítomnosti je vhodné mluvit o databázovém clusteru namísto pouhé replikované databáze; databázový cluster se chová stejně, jako by se choval jeden databázový server, nemůže dojít k žádné nekonzistenci či vzájemným blokacím. Takovýto systém může kromě replikace jednotlivých tabulek databáze umožňovat i vytvoření distribuovaných tabulek, tedy že každý databázový stroj může mít lokálně uchovanou jen podmnožinu veškerých dat v systému. Databázový cluster je však kvůli těsné koordinaci činnosti jednotlivých strojů citlivý na latenci při komunikaci mezi servery a obvykle není vhodný k nasazení na geograficky oddělené servery.

Standardně replikované databáze zahrnují také různou míru sdílených prostředků, může jít například o uživatele, sekvence, triggery a další databázové zdroje. Mimo tuto kategorizaci stojí databáze využívající sdíleného disku či replikace na úrovni souborového systému.

4.5 Replikační řešení databáze PostgreSQL

4.5.1 Warm Standby a Hot Standby

Databázový stroj PostgreSQL podporuje replikaci na téměř-aktuální kopie pomocí zasílání WAL logu. Poskytuje variantu realizovanou pomocí souborově orientovaného zasílání logu (file-based log shipping), kdy primární server operuje v režimu kontinuálního archivování a ostatní servery v režimu kontinuální obnovy, při které čtou WAL log z primárního serveru. Tento log je přenášen po naplnění jednotlivých segmentů (16 MB) [29], zpoždění tedy může být zvláště při méně četných manipulacích s daty výrazné. Druhou variantou je využití kontinuálního přenosu (streaming replication), kdy jsou repliky s primární databází stále spojeny a jsou jim zasílány WAL záznamy okamžitě. Jedná se ale stále o asynchronní přenos dat se zaznamenatelnou prodlevou mezi aktuálností jednotlivých serverů. Přenos dat je jednosměrný a data jsou editována pouze na primárním serveru, díky tomu je zajištěna eventuální konzistence na straně repliky. Hot Standby se oproti Warm Standby liší v možnosti provádět čtecí dotazy nad replikou.

4.5.2 Jednosměrná replikace pomocí triggerů

Master-slave replikaci je možné realizovat pomocí triggeru definovaného na tabulkách primární databáze, který zašle změněná data replikám. Takovýmto replikačním systémem pro PostgreSQL je například Slony-I. Obsahem přenášených dat jsou jednotlivé řádky tabulky. Výhodou oproti zabudované replikaci je možnost interagovat mezi různými verzemi PostgreSQL serverů, umožnění replikace pouze částí databáze a možnosti specifikace speciálního chování replik [30], v ostatních parametrech je řešení podobné variantě Hot Standby.

4.5.3 Middleware zasílající dotazy

Před databází je předřazena komponenta, která přijímá klientské dotazy a zasílá je na všechny replikované instance databáze v případě modifikujících dotazů či pouze na jednu v případě čtecích dotazů. Protože instance databáze vykonávají dotaz nezávisle, funkce pracující s náhodnými čísly či aktuálním časem a sekvence mohou mít na jednotlivých serverech různé hodnoty, pokud middleware tyto situace nezpracovává odlišně [29]. Na tomto systému jsou založeny replikační nástoje Pgpool-II a Sequoia.

4.5.4 Asynchronní multi-master replikace

Jedná se o oboustrannou replikaci se všemi databázemi ve stejné roli. Mechanismus je specifický dle využitého nástroje. Nástroj Bucardo používá k replikaci triggery a zasílá řádky tabulky, BDR pracuje na úrovni databáze a zasílá modifikované záznamy WAL logu jakmile jsou dostupné. Technologie musí obsahovat mechanismy pro řešení konfliktů vyvstávajících z editací dat více servery bez použití centralizovaného transakčního koordinátora.

4.5.5 Synchronní multi-master replikace

Požadavky jsou synchronně zaslány všem ostatním instancím databáze. Při tomto řešení je nutné minimalizovat latenci mezi servery; i přes možnost provádět dotaz na libovolné instanci je výkon takovéto databáze obvykle nižší než výkon jedné samostatné databáze. Projektem využívajícím toto schéma je cluster Postgres-XL, který obsahuje i centralizovaného koordinátora transakcí. Dělí jednotlivé servery na datové uzly, koordinátory a globální transakční monitory, zajišťující transakční konzistenci. Jedná se o rozsáhlé řešení vhodné pro nasazení na velký počet serverů v jednom datacentru pro škálování aplikací vyžadujících velký počet databázových zápisů.

4.5.6 Zvolené řešení

Z důvodu nasazení na servery v geograficky oddělených datacentrech a nutnosti zachování krátké doby odezvy není možné využít synchronní replikační schéma. Pro minimalizaci množství nedostupných dat v případě výpadku je ale potřeba použít schéma, ve kterém je časové okno mezi potvrzením transakce a jejím provedením na replice minimální. Vzhledem k práci v homogenním prostředí je možné použít synchronizační mechanismus fungující přímo na databázi bez využití externích nástrojů. Maximální míru identity dat mezi databázemi zajistí replikace na fyzické úrovni či pomocí WAL logu. Tato omezení ponechávají variantu Hot Standby s manuální změnou role databáze v případě výpadku a manuálním mechanismem obnovy, a variantu asynchronní multi-master replikace pomocí mechanismu BDR. Replikace Hot Standby cílí na

zkrácení doby nedostupnosti v případě havárie. Ač může být ve většině situací záložní databáze povýšena do role primární databáze externími nástroji, jako je PgPool-II či Repmgr, robustnost takového řešení je nižší než v případě multi-master replikace. V důsledku vyšší míry automatizace mechanismů zajišťujících obnovení konzistence a vyšší odolnosti proti nahodilým chybám je zvolen systém BDR, jehož cílem je úplně eliminovat nedostupnost aplikace bez nutnosti administrátorského zásahu. Na aplikaci je ponechána nutnost zajistit provádění dotazů ve stejnou chvíli vždy pouze na jedné databázi, aby nedocházelo k modifikačním konfliktům.

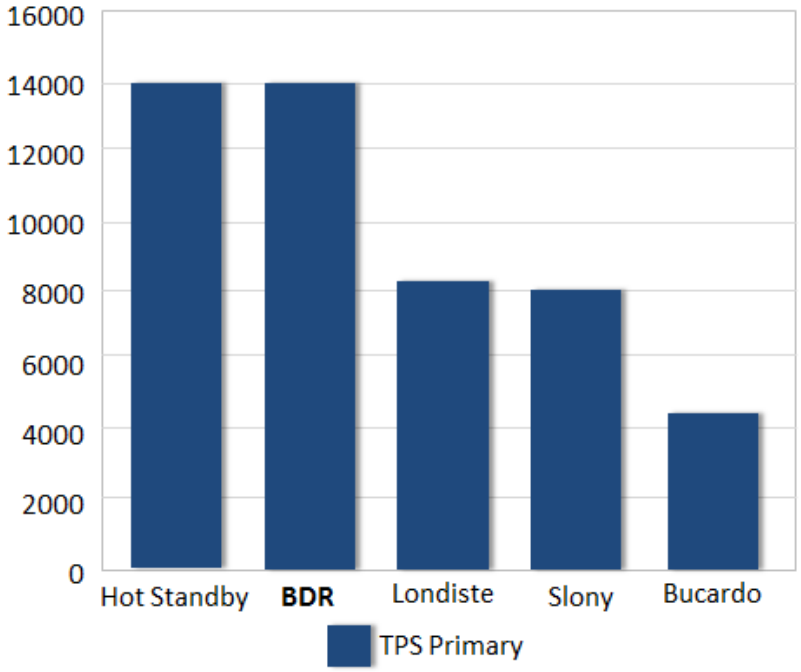
4.5.7 BDR

BDR je replikační systém pro PostgreSQL, vyvinutý společností 2ndQuadrant, specificky navržený pro umožnění replikace geograficky oddělených databázových uzlů [31]. BDR ve verzi 0.9.3 využívá modifikované varianty databáze PostgreSQL 9.4. Tyto modifikace ovlivňují mimo jiné formát datových souborů na disku, modifikovaná verze databáze tak nemůže přistupovat k datovým adresářům vytvořeným originální verzí a naopak. Technologie vyvinuté v rámci tohoto projektu jsou postupně integrovány do samotného databázového stroje PostgreSQL a v příštích vydáních se očekává, že bude BDR pracovat i nad standardní instalaci PostgreSQL databáze. Jedná se o novou technologii, jejíž design byl představen v roce 2012 a zatím není použit v mnoha produkčních nasazeních. Systém neobsahuje centrální zamykání a transakčního koordinátora pro DML dotazy (Data manipulation language, příkazy pro manipulaci s daty v tabulce -– SELECT, INSERT, UPDATE a DELETE), nicméně obsahuje globální DDL zámek pro příkazy modifikující schéma databáze (Data definition language, příkazy CREATE, DROP, ALTER, RENAME).

Řešení se zaměřuje na dosažení minimálního vlivu replikace na výkon a propustnost databáze. Výkonnostní testy ukazují výsledky srovnatelné s Hot Standby replikací a počtem zpracovaných transakcí za sekundu BDR překonává ostatní poskytovatele replikace, jak je patrné z obrázku 4.1. Dle obrázku 4.2 patří i mezi replikační schémata s nejnižší latencí na transakci, s hodnotami v řádu nízkých jednotek milisekund.

Při běhu aplikace v prostředí replikovaném systémem BDR je třeba mít na paměti několik omezení, vyplývajících z architektury a implementace systému. Souvisí zejména s těmito oblastmi:

- Globální sekvence
  - SQL standard definuje objekt *SEQUENCE* pro generování unikátních hodnot. Tyto sekvence nejsou systémem BDR synchronizovány mezi servery a existují tak na každém serveru nezávisle. Vzhledem k tomu, že v PostgreSQL databázi jsou pseudotypy *SERIAL* a *BIGSERIAL*, často používané pro generování unikátních identifikátorů,

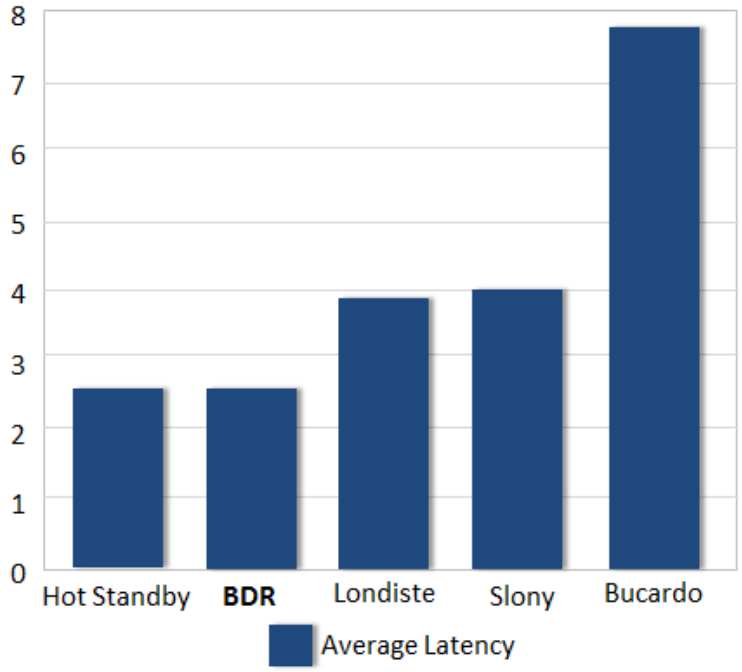


Obrázek 4.1: Počet zpracovaných transakcí za sekundu

využitých jako primární klíče tabulek, implementovány pomocí sekvencí, docházelo by při vkládání dat do stejné tabulky více servery k neustálým kolizím na primárním klíči. Tento problém je řešitelný i za pomoci klasické sekvence, která je na každém serveru nastavena tak, aby se vždy inkrementovala o stejnou hodnotu, alespoň ve výši počtu serverů, a začínala s různou hodnotou. Uzel 1 tak může generovat posloupnost 1, 11, 21, 31, ..., zatímco uzel 2 generuje posloupnost 2, 12, 22, 32, ..., aniž by mohlo dojít ke kolizi. Toto řešení spoléhá na korektní a individuální inicializaci sekvencí na každém serveru a v případě zvýšení počtu replikovaných uzlů nad počet ponechaných míst v sekvenci způsobuje obtížnou migraci.

– Alternativou je využití globálních sekvencí systému BDR. Globální sekvence zaručují generování unikátní hodnoty v rámci všech uzlů replikované skupiny. Globální sekvenci je možné vytvořit příkazem *CREATE SEQUENCE seq_name USING bdr;*, či je možné v konfiguračním souboru databáze nastavit automatické vytváření sekvencí jako globálních pomocí parametru *default_sequenceam = 'bdr'*. V takovém případě je využití globálních sekvencí pro aplikaci naprosto transparentní.

– Pro zajištění unikátnosti hodnoty generované sekvencí napříč všemi



Obrázek 4.2: Latence na transakci (v milisekundách)

servery bez nutnosti centrálního přístupu ke sdílenému objektu jsou hodnoty sekvencí serverům přidělovány po blocích o velikosti 1000 hodnot. Počet jednorázově přidělovaných bloků je konfigurovatelný pro každou sekvenci. Rozdělování bloků probíhá pomocí volby, pro kterou je nutné, aby byla k dispozici nadpoloviční většina všech serverů v replikované skupině. Pokud by tedy došlo, například v důsledku dlouhodobého výpadku více než poloviny uzlů, k vyčerpání hodnot přidělených zbylým instancím, žádost o novou hodnotu by skončila chybou. Po vytvoření sekvence je také nutné vyčkat na proběhnutí prvního přidělení bloků, do té doby není sekvence připravena k použití. Globální sekvence mají v současné implementaci omezení na hodnotu inkrementu rovnou jedné a neumožňují manuálně restartovat sekvenci na požadovanou hodnotu.

- DDL replikace
  - BDR zajišťuje replikaci příkazů modifikujících databázové schéma pomocí globálního DDL zámku. DDL příkazy může provádět pouze uzel, který je držitelem zámku. K získání zámku je zapotřebí potvrzení všemi uzly v replikované skupině, není tedy možné modifikovat schéma v okamžiku nedostupnosti některého ze serverů. Pokud je

některý uzel držitelem DDL zámku, žádné jiné uzly nesmí provádět DDL příkazy ani modifikovat data v tabulkách. Z důvodu nutnosti kooperace všech uzlů dosahují transakce modifikující schéma nízkého výkonu, v řádu jednotek za sekundu, oproti až tisícům transakcí za sekundu pro transakce obsahující pouze příkazy DML na totožné skupině strojů.

- Tabulky bez primárního klíče
  - Systém není schopen provádět příkazy UPDATE a DELETE nad tabulkami, které nemají definovaný primární klíč, z důvodu závislosti mechanismu replikace a řešení konfliktů na primárním klíči. Toto omezení většinu aplikací výrazně nelimituje – definice primárního klíče nad každou tabulkou je dobrou návrhovou praxí. Existují však aplikace využívající vztahové tabulky bez primárních klíčů, v takové situaci je nutné primární klíč dodefinovat, například jako dvojici cizích klíčů referujících k oběma stranám vztahového záznamu.
- Omezení příkazů
  - Některé typy příkazů nejsou kvůli problematické replikaci povoleny. Jedná se například o příkazy *CREATE TABLE AS SELECT*, *SELECT INTO* či *REFRESH MATERIALIZED VIEW*. Aplikace používající pokročilejší příkazy musí jejich podporu ověřit v dokumentaci systému.
- Individuální zamykání
  - Vzhledem k neexistenci sdíleného správce transakcí pracují transakce na jednotlivých uzlech systému nezávisle a zámky na libovolných úrovních dat nejsou propagovány. Ostatní uzly tedy mohou modifikovat data, která jsou v rámci transakce na vyřizujícím uzlu uzamčena. Tato vlastnost nemá vliv na transakční konzistenci; replikační mechanismus nezmění data zamčená na konkrétním uzlu před jejich odemčením.
- Řešení konfliktů
  - Jsou-li modifikována stejná data ve stejný čas na různých uzlech, dochází ke konfliktu. Typy konfliktů a jejich výchozí způsob řešení je patrný z tabulky 4.2. BDR umožňuje definovat uživatelské procedury pro řešení jednotlivých typů konfliktů, pokud výchozí metoda není uspokojivá. Každý konflikt může být zaznamenán do tabulky konfliktů pro manuální inspekci.

Typ konfliktu	Výchozí řešení
INSERT/INSERT – více záznamů se stejným primárním klíčem	ponechání pozdějšího záznamu
INSERT porušující více UNIQUE omezení	neexistuje — dojde k chybě, nutno řešit manuálně
UPDATE/UPDATE – modifikace stejného řádku	ponechání pozdější změny
UPDATE konflikty na primárním klíči	neexistuje – dojde k chybě, nutno řešit manuálně
UPDATE porušující více UNIQUE omezení	neexistuje – dojde k chybě, nutno řešit manuálně
UPDATE/DELETE – modifikace a smazání stejného řádku	ignorování UPDATE příkazu
INSERT/UPDATE – uzel získá UPDATE příkaz dříve než odpovídající INSERT příkaz	ignorování UPDATE příkazu, může vést k odlišným datům na jednotlivých uzlech, nutno řešit manuálně
DELETE/DELETE – záznam je smazán dvěma požadavky	jeden z příkazů je ignorován

Tabulka 4.2: Typy replikačních konfliktů systému BDR

– Vedle těchto konfliktů může dojít k porušení omezení cizích klíčů. Jak vysvětluje BDR dokumentace: *„Zatímco je dodrženo striktní pořadí aplikování [transakcí] pro každý uzel, neexistuje žádné vynucování pořadí transakcí mezi dvěma různými uzly, takže je možné, aby (například) uzel 1 vložil řádek do T1, a tato změna byla aplikována na uzlu 2. Uzel 2 vloží řádek do T2, který cizím klíčem referuje na řádek z T1. Na uzlu 3, za předpokladu že transakce z uzlu 2, vkládající řádek do T2, je přijata dříve než transakce z uzlu 1, vkládající řádek do T1, dojde k chybě při aplikaci transakce z uzlu 2. Tato chyba zaznamená zrušení transakce do bdr.pg_stat_bdr a chybu s detaily do PostgreSQL chybového logu na aplikujícím uzlu (uzel 3). V tomto případě se bude BDR pokoušet periodicky opakovat transakci z uzlu 2, tak, že jakmile je aplikována transakce z uzlu 1, na které tato transakce závisí, bude transakce úspěšně potvrzena“* [32] (překlad vlastní). Výchozí řešení tohoto problému je tak ve většině případů schopné eventuálně dosáhnout korektního stavu automaticky. Může však existovat takové pořadí transakcí, kdy nebude možné aplikovat změny žádného uzlu na žádném jiném uzlu a dojde k uváznutí celého replikačního mechanismu, vyžadující administrátorský zásah.

– Je vhodné navrhnout aplikaci tak, aby zabránila vzniku konfliktů,

nebo aby byla schopna tyto konflikty tolerovat. Vhodnými nástroji pro eliminaci významné části potenciálních konfliktů může být využívání globálních sekvencí a zamezení změny primárního klíče.

- Nereplikované objekty
  - Některé objekty nejsou replikovány mezi databázemi a existují nezávisle na každém serveru. Kromě zmíněných sekvencí se jedná zejména o uživatele, role a skupiny. Dále nejsou replikovány příkazy vytvářející či modifikující tabulkové prostory a databáze. Je tak nutné spravovat tyto zdroje individuálně.
- Administrace a monitoring
  - Pro zabránění uváznutí v důsledku sdíleného DDL zámku je vhodné monitorovat výpadky jednotlivých uzlů. Tabulka obsahující konflikty vyplývající ze souběžné modifikace dat by měla být analyzována se snahou nalézt řešení pro zabránění provádění takovéto kombinace operací klientskou aplikací. Je-li nutné zajistit stoprocentní integritu dat, musí být tato tabulka po havárii zkontrolována na nové záznamy, vycházející z možnosti včasného nezaslání modifikovaných dat havarovanou databází replikám. V případě dlouhodobé nedostupnosti některého uzlu dochází na ostatních uzlech k hromadění WAL segmentů, čekajících na odeslání nedostupnému uzlu. Je nutné monitorovat a zajistit dostatek místa na disku pro tato nadbytečná data a při odebrání některého z databázových serverů provést adekvátní úkony vyřazující tento uzel z replikované skupiny.

4.6 Vyplývající aplikační omezení

Z těchto vlastností zvoleného řešení vyplývají některá omezení na činnost aplikace. Vyvíjený informační systém pro správu zákaznické podpory je navržen se znalostí nasazení do replikovaného prostředí. Existuje tak jedna instance aplikace, která je hlavní instancí a provádí úkony, které musí být vykonávány právě jednou instancí aplikace, jako je stahování e-mailů z e-mailových schránek. Před aplikací se nachází (duplikovaná) komponenta (failover manager), která zpracovává klientské požadavky a přeposílá je hlavní instanci aplikace. Tím je zajištěna manipulace s daty pouze jednou instancí aplikace a tedy jedním databázovým uzlem. Díky tomu odpadá jakákoli možnost výskytu konfliktu za situace, kdy jsou všechny databázové uzly dostupné. V případě výpadku hlavní instance aplikace je failover manager schopen povýšit jinou instanci aplikace do role hlavní aplikace. Databázi díky roli multi-master není nutno o provedené změně informovat.

Databáze je nakonfigurována pro plošné využívání globálních sekvencí a aplikace používá pseudotyp *BIGSERIAL* pro generování primárních klíčů pomocí těchto sekvencí. Aplikace definuje primární klíče na všech tabulkách a neprovádí žádné změny primárních klíčů.

V kontextu CAP teorému se zvolené řešení vzdává dostupnosti ve prospěch konzistence dat. To neznamená, že by data za některé situace byla nedostupná; dostupnost dat je zajištěna v případě výpadku libovolného serveru. Pouze není zaručeno, že na požadavek bude schopen odpovědět libovolný ze zbylých uzlů. Pokud instance aplikace není primární instancí a není zaručeno, že má přístup k nejnovějším datům, není schopná vyřizovat požadavky a ty jsou směrovány na jiný uzel systému. Toto řešení je podrobněji diskutováno v sekci s jednotlivými plány nasazení. Navržená aplikace za jistých podmínek relaxuje i podmínku absolutní konzistence dat – v důsledku asynchronního zápisu mohou být v případě výpadku poskytována starší data než ta, která byla zapsána na havarovaný uzel. Rozvolnění této podmínky je preferováno z důvodu výrazného snížení latence provedení transakce za použití asynchronního zápisu. Konzistence dat je porušena pouze v případě výpadku a po jeho odstranění je obnovena, pokud nedošlo k fyzické ztrátě dat na úložišti uzlu.

4.7 Zajištění vysoké dostupnosti souborového systému

Vyvíjená aplikace na souborový systém ukládá z uživatelských dat pouze přílohy e-mailů a požadavků. Přístup k těmto datům je definován velmi specifickým případem užití. Přílohy jsou vytvářeny při stahování zpráv z e-mailových schránek a při nahrávání souboru uživatelem systému. V některých případech může být požadováno odstraňování příloh. Stahování příloh musí být vždy dostupné. Tyto dokumenty však v žádném případě nejsou upravovány, díky čemuž mohou být umístěny v asynchronně replikovaném či distribuovaném prostředí bez rizika kolize vyplývající ze souběžné modifikace dat. Předpokládané využití aplikace také neklade vysoké požadavky na výkon ani odezvy takovéhoto úložiště – počet přístupů k přílohám je o několik řádů nižší než k transakčním datům systému. Oproti tomu předpokládaná velikost všech uložených příloh řádově přesahuje velikost relační databáze. Je tedy vhodné navrhnout takové řešení souborového systému, které v případě nutnosti škálovat nasazený systém umožní umístit dokumenty na více serverů distribuovaně, tedy tak, aby každý server měl přístup k veškerým datům, uloženým na distribuovaném svazku, aniž by byl nucen uchovávat kompletní data systému lokálně. I za takové situace je ale nutné, aby byl každý datový blok duplikován na jiném serveru. Souborový systém by tak měl být distribuovaný, s podporou replikace a nasazení přes geograficky oddělená datacentra, se schopností automatického zotavení při výpadku některého z uzlů.

4.7.1 HDFS

Hadoop Distributed File System (HDFS) je distribuovaný souborový systém, vyvinutý v rámci projektu Apache Hadoop, zaměřeného na vytvoření frameworku pro zpracování velkého množství nestrukturovaných dat. HDFS používá dva druhy uzlů – jmenné uzly (NameNode) a datové uzly (DataNode). Jmenný uzel zastává roli hlavního serveru, který spravuje jmenný prostor souborového systému a řídí přístup klientů k datům. Datové uzly obsahují (či spravují) úložiště, na kterých se nacházejí data systému. Klient zasílá požadavky na souborové operace jmennému uzlu, ten pověřuje odpovídající datové uzly, z nichž pak jsou data klientovi vrácena přímo.

Oproti prvním verzím HDFS je nyní možné definovat druhý jmenný uzel, fungující jako záložní, díky čemuž systém neobsahuje žádný bod, jehož selhání by způsobilo výpadek systému, a může tak být nasazen do prostředí vyžadujícího vysokou dostupnost. Systém je nativně vytvářen pro podporu modelu write-once read-many (WORM), tedy počítá s vytvořením souboru s daty, která nejsou v průběhu jeho životního cyklu upravována. Výhodou HDFS je schopnost konfigurovat počet replik, které musí být zapsány před potvrzením zápisové operace, díky čemuž je možné individualizovat systém pro zajištění odolnosti a konzistence, nebo naopak nízkých latencí. Vytváření ostatních replik probíhá asynchronně, svazek tak může být nasazen i mezi servery s vyšší síťovou latencí.

Přístup k souborovému systému je obvykle realizován prostřednictvím frameworku přímo z aplikace, systém tak nebývá dostupný operačnímu systému. Existují však projekty, jako například Fuse-DFS nebo HDFS NFS Proxy, které umožňují připojit HDFS souborový systém k Unixovým operačním systémům pomocí příkazu mount. Nasazení a správa Hadoop clusteru je komplexním úkonem a vyvíjená aplikace, ve které je uložení příloh jen vedlejší funkcionalitou a data nejsou dále nijak zpracovávána, by využila jen zlomek nabízených vlastností. Investice do nasazení takového clusteru by tak nebyla odůvodněná. Jmenný uzel si navíc ponechává adresářový strom a mapování souborů na datové uzly v paměti RAM, což může v případě uložení mnoha malých souborů vykazovat vysokou paměťovou náročnost; HDFS je optimalizován na práci s velkými soubory.

4.7.2 Lustre, CephFS, XtreemFS

Tyto systémy zastupují klasické distribuované souborové systémy. Lustre je nejčastěji nasazovaným distribuovaným souborovým systémem do velkých výpočetních clusterů a superpočítačů, zaměřený na dosažení vysokého výkonu spíše než odolnosti při chybě. CephFS používá Ceph Storage Cluster, distribuované úložiště objektů k ukládání souborů. XtreemFS reprezentuje zvažovanou variantu, díky zaměření na replikaci a vysokou dostupnost a explicitní podpoře needitovatelných soborů. Jeho architektura definuje tři typy uzlů –

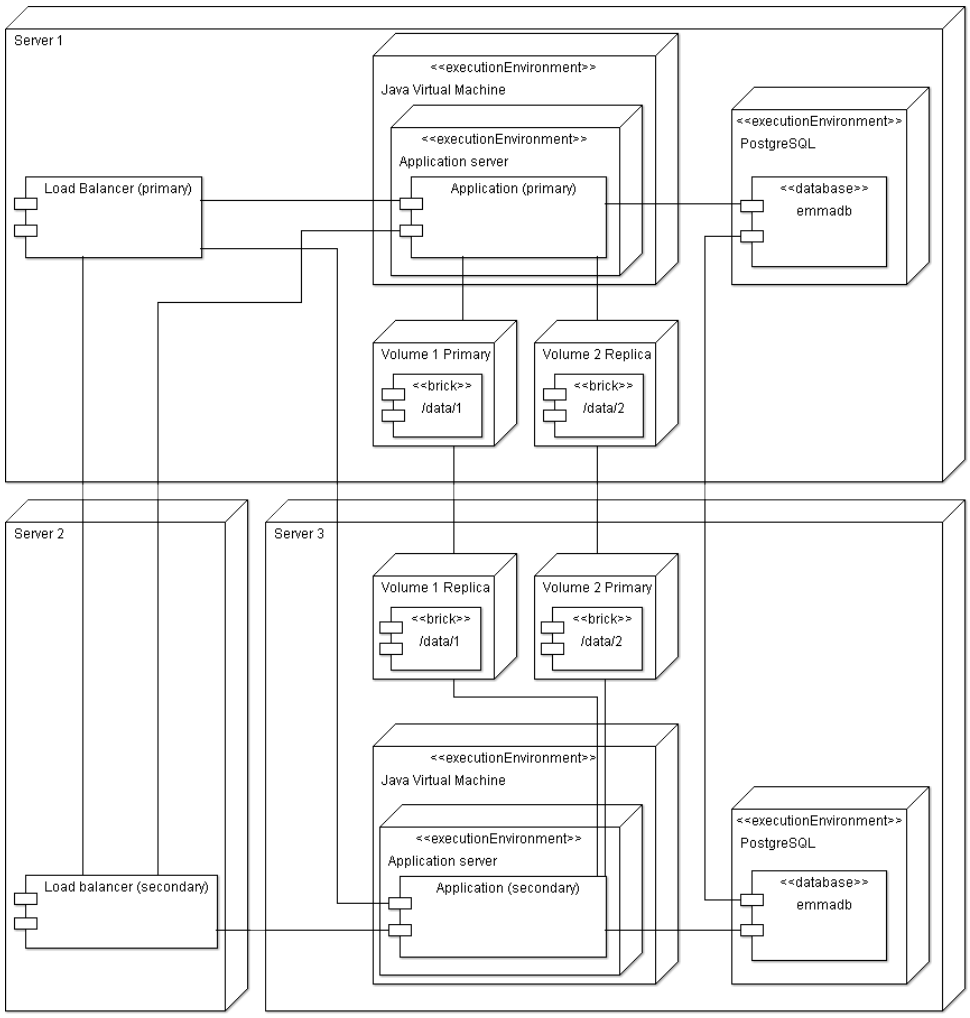
katalog metadat a replik (MRC, Metadata and Replica Catalog), adresářovou službu (DIR, Directory Service) a úložné zařízení (OSD, Object Storage Device). Zajištění vysoké dostupnosti vyžaduje nutnost replikovat nejen uložená data, ale i katalog metadat. Tato metadata jsou uložena v databázi a pro zajištění konzistence jsou synchronně replikována, v důsledku čehož by odezva systému při nasazení do geograficky oddělených datacenter nebyla akceptovatelná.

4.7.3 GlusterFS

GlusterFS je distribuovaný souborový systém, implementovaný nad tradičními souborovými systémy, jakými jsou ext4 či XFS. Z jednotlivých oddílů je vytvořen tzv. storage pool, nad nímž je možné tvořit logické svazky. Tyto svazky jsou spravovány *glusterd* démonem a je možné je připojit jako běžný lokální souborový systém či je exportovat v podobě síťového souborového systému NFS. GlusterFS podporuje mnoho režimů nasazení. Kromě základní varianty distribuované, replikované a jejich kombinací, nabízí možnost uložení jednotlivých částí souboru na různé uzly, pro podporu uchovávání velkých souborů a paralelizaci čtecích operací. Dále umožňuje vytvářet samoopravné kódy, zajišťující dostupnost souboru i při výpadku některého z úložišť, v konfiguraci konceptuálně podobné, jako je RAID 5 pro fyzická disková pole. Systém je široce dynamicky konfigurovatelný, je tak možné přidávat i odebírat jednotlivé uzly systému bez dopadu na dostupnost dat. Zápis na jednotlivé uzly v klasických módech činnosti probíhá synchronně, souborový systém ale poskytuje možnost nastavit geo-replikaci, při které je obsah jednoho svazku jednosměrně replikován na druhý svazek v odlišném datacentru asynchronně. Díky tomu je možné využívat výhod synchronního zápisu a konzistence dat v rámci distribuovaného svazku v jednom datacentru, a zároveň asynchronní replikace do datacentra jiného pro zajištění vysoké dostupnosti. Nevýhodou je nutnost konfigurovat další svazky pro opačný směr replikace a upravit aplikaci tak, aby zapisovala do svazku, který je v daném datacentru v roli primárního svazku (master). Je nutné poznamenat, že tato varianta nasazení je umožněna pouze díky zaručené neměnnosti dat, z důvodu nemožnosti zapisovat do svazku na straně repliky. Protože aplikace toto omezení splňuje, je použit GlusterFS jako distribuovaný souborový systém pro ukládání příloh.

4.8 Diagram nasazení

V rámci práce byly navrženy dva plány nasazení. První přestavuje minimální konfiguraci, tedy takovou, která využívá co nejnižší počet serverů při zachování vysoké dostupnosti. Druhý reprezentuje plnou konfiguraci, kde každá komponenta běží na vlastním fyzickém serveru. Tato varianta přináší možnost škálování aplikace na více strojů pro dosažení vyššího maximálního počtu obsluhovaných klientů.



Obrázek 4.3: Minimální schéma nasazení

4.8.1 Minimální konfigurace

Diagram nasazení pro minimální konfiguraci je na obrázku 4.3. Systém využívá tří fyzických serverů – na jednom (na obrázku označen jako server 1) se nachází kompletní instance systému: load balancer, aplikace, databáze a souborové úložiště. Na druhém (server 2) běží pouze záložní load balancer, a na stroji server 3 je spuštěna téměř kompletní instance systému bez load balanceru. Všechny servery by měly být umístěny v geograficky oddělených lokalitách, nesdílejících žádný prvek infrastruktury. Každá instance aplikace je nakonfigurována pro využití právě jedné (lokální) databáze a souborového systému. Load balancer v této konfiguraci působí pouze jako tzv. failover manager, komponenta, která v pravidelných intervalech kontroluje dosažitelnost ostatních serverů a dle výsledků povyšuje konkrétní instanci aplikace do role primární instance. Všechny požadavky klientů pak směruje na tuto instanci.

Za standardní situace je primární instancí aplikace běžící na serveru 1. Oba load balancery předávají požadavky na tuto instanci. Dojde-li k výpadku serveru 3, či konektivity serveru 3 k některému z load balancerů, nedochází k žádné změně. Pokud dojde k výpadku serveru 2 (tedy spojení mezi oběma load balancery), primární load balancer tuto změnu zaregistruje a povýší instanci aplikace na serveru 3 do role primární instance. Zároveň degraduje lokální instanci do role sekundární aplikace. Toto chování je nutné pro zamezení tzv. split brain syndromu, kdy se každá instance aplikace považuje za primární, což vede k paralelní modifikaci dat. K němu by mohlo dojít proto, že primární load balancer není schopen rozeznat, zda došlo k výpadku serveru 2, nebo je naopak server 1 částečně odříznut od konektivity. Ponechal-li by primární load balancer vlastní instanci jako primární a jednalo-li by se pouze o výpadek konektivity, měl by sekundární load balancer dvě možnosti – ponechat hlavní instanci na serveru 1, či povýšit instanci serveru 3. V případě ponechání by mohla být ohrožena vysoká dostupnost služby tím, že by byly směrovány všechny požadavky na server s omezenou konektivitou. V případě povýšení sekundární instance by docházelo ke split brain syndromu, jelikož by klienti dostávali odpovědi z různých serverů dle toho, na který load balancer by vznesli dotaz. Díky tomu, že primární load balancer povýší instanci serveru 3 a sekundární load balancer (běží-li) provede totéž, míří požadavky všech klientů na bezproblémový server 3.

V případě výpadku serveru 1 povýší sekundární load balancer zbývající instanci. Podobně se zachovají oba load balancery za situace, že havaruje aplikace běžící na serveru 1. Oba load balancery tak jsou za všech okolností řízeny stejnými logickými podmínkami a provádějí identické úkony. Za každé situace je tak zajištěna vysoká dostupnost (v rozsahu zajištění plnohodnotného běhu služby v případě havárie libovolného jednoho serveru) a odolnost proti rozštěpu systému v případě výpadku konektivity a vzniku nezávislých síťových segmentů. Při havárii primárního uzlu může dojít k nekonzistenci dat – uživatelům nebudou dostupná data, která v okamžiku výpadku ještě nebyla uložena

na sekundárním uzlu v důsledku asynchronní replikace. Toto množství nedostupných dat však bude odpovídající pouze zlomku sekundy, který replikace typicky trvá. Po obnovení primární instance dojde automaticky k replikaci těchto dat a obnovení dostupnosti chybějících informací.

4.8.2 Plná konfigurace

Plná konfigurace vyhrazuje vlastní servery pro load balancery, aplikační servery, databázové stroje a souborové úložiště. Architektura systému je patrná z obrázku 4.4. Oproti minimální konfiguraci nejsou jednotlivé instance aplikace vázány na jednu instanci databáze, místo toho zde existuje vrstva databázových load balancerů, spravujících primární instanci databáze. Díky tomu mohou být klientské požadavky vyřizovány oběma instancemi aplikace a databázové load balancery jen směrují požadavky obou aplikačních instancí na identický databázový uzel. Webové load balancery pak plní roli skutečného vyvažování zátěže a umožňují aplikaci škálovat na úrovni aplikačního serveru. Jejich role při řízení výpadku však není eliminována, jedna z aplikačních instancí stále musí být v nadřazené roli, pro vykonávání systémových úkonů, např. stahování zpráv z e-mailových schránek. Logika řízení výpadku je podobná jako u minimální konfigurace. Škálování souborového systému je umožněno díky možnosti přidat jednotlivé servery s úložišti do distribuovaných svazků. Nasazení počítá s využitím dvou geograficky oddělených datacenter.

4.9 Praktické nasazení

4.9.1 Parametry systému

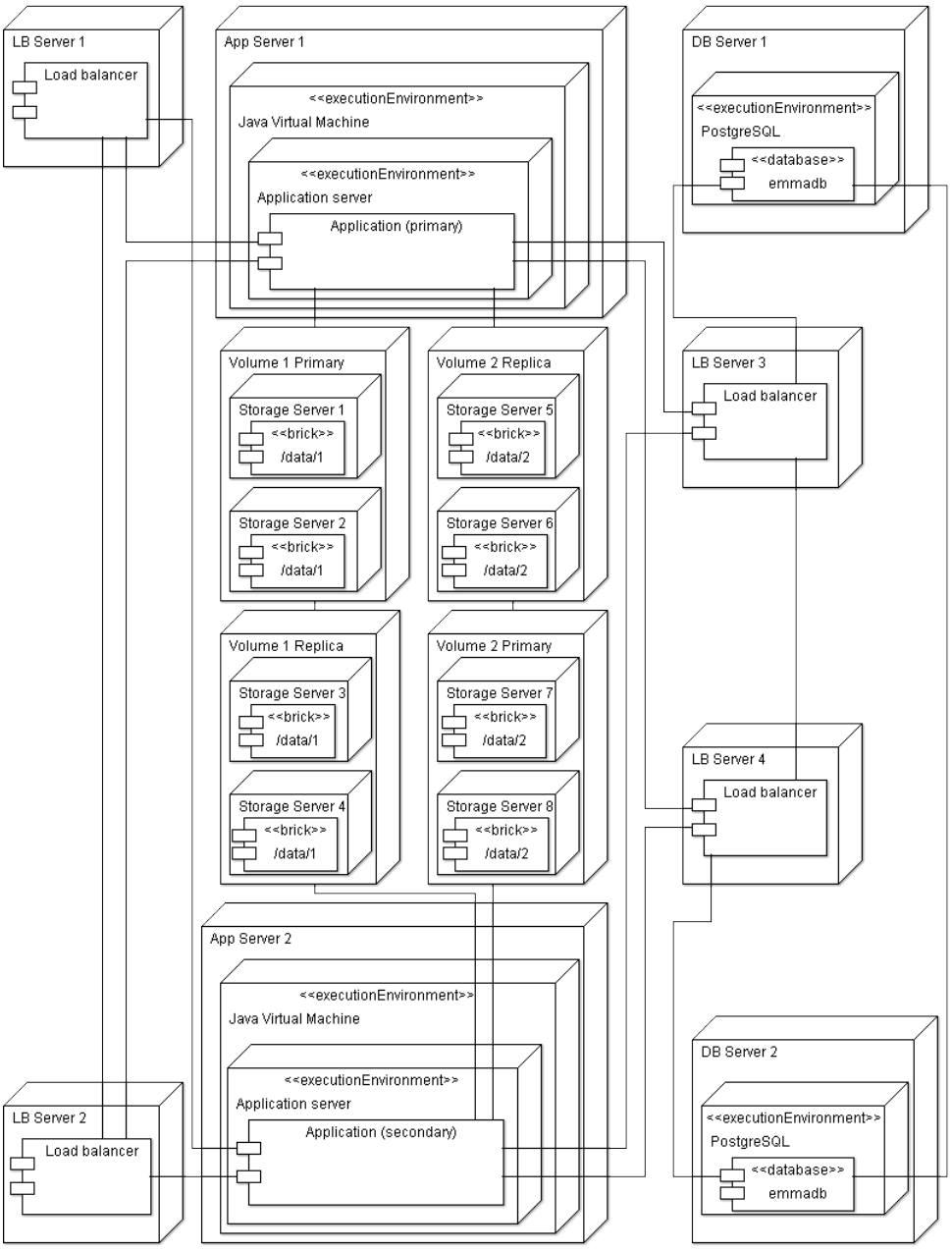
Nasazení aplikace v počátcích projektu probíhá dle plánu minimální konfigurace. Jednotlivé servery jsou vybaveny operačním systémem Linux, konkrétně distribucí Debian 8.0 (Jessie). Primární server aplikace se nachází v datacentru společnosti OVH ve Francii. Sekundární instance aplikace běží na serveru umístěném v datacentru v České republice. Odezva mezi těmito servery činí průměrně 19 milisekund.

4.9.2 Konfigurace BDR

Pro využití BDR replikace je nutné nebo vhodné nastavit následující hodnoty v hlavním konfiguračním souboru databáze *postgresql.conf* na všech databázových uzlech:

```
shared_preload_libraries = 'bdr'
```

Zajistí načtení BDR knihovny při startu serveru. Tato knihovna je nutná pro zajištění funkčnosti replikace a její přednačtení umožňuje spustit procesy přijímající data od ostatních uzlů.



Obrázek 4.4: Plné schéma nasazení

```
wal_level = 'logical'
```

Nastavuje, jaké informace budou ukládány do WAL logu. PostgreSQL definuje úroveň *minimal*, kdy jsou do logu ukládány pouze informace nutné pro zotavení při pádu databáze, úroveň *archive*, která přidává informace nutné pro archivaci WAL logu a rekonstrukci dat při obnově databáze ze zálohy, úroveň *hot_standby*, obsahující navíc data pro rekonstrukci stavu běžící transakce, a úroveň *logical*, která kromě informací úrovně *hot_standby* ukládá informace o identitě řádku v rámci repliky a další data potřebná k provedení logické replikace. Úroveň *logical* je potřeba nastavit na všech uzlech, ze kterých se data mají replikovat.

```
track_commit_timestamp = on
```

Umožňuje sledování časových známek potvrzení jednotlivých transakcí. Ty jsou důležité pro mechanismy řešení konfliktů založené na principu uchování poslední provedené změny. Tento parametr je specifický pro BDR plugin.

```
max_connections = 100
```

Definuje maximální počet připojených klientů k databázi. Nastavení této hodnoty je specifické dle použití konkrétní aplikací. V kontextu replikačního systému BDR je nutné vzít v úvahu, že replikační procesy jsou také připojenými klienty a je tak potřeba tuto hodnotu navýšit o počet uzlů v replikační skupině. Pro potřeby této aplikace je výchozí hodnota 100 spojení maximálně dostačující.

```
max_wal_senders = 10
```

Definuje maximální počet WAL sender procesů. Tyto procesy jsou zodpovědné za odesílání WAL logu ostatním replikovaným uzlům. WAL sender procesy jsou také potřebné pro některé metody zálohování databáze, je tedy vhodné nastavit hodnotu vyšší než je počet replikovaných uzlů.

```
max_replication_slots = 10
```

Určuje maximální počet replikačních slotů. Replikační sloty zajišťují, že WAL segment nebude odstraněn dříve, než jej přijmou všechny repliky. Slot je vyhrazen pro každý uzel odesílající nebo přijímající repliku. Replikační sloty jsou zabrány i replikami, které nejsou v aktuálním okamžiku dostupné. Jejich počet v případě multi-master replikace musí dosahovat alespoň počtu replikovaných uzlů ve skupině.

```
max_worker_processes = 10
```

Nastavuje maximální počet externích procesů běžících na pozadí. Hodnota je nastavena s ohledem na počet replikačních procesů, případně na použití jiných zásuvných modulů do databáze.

```
bdr.log_conflicts_to_table = on
```

Povoluje zaznamenávání konfliktů vzniklých při replikaci do tabulky s názvem *bdr.bdr_conflict_history*. Tabulka konfliktů je unikátní pro každý uzel. Pro ověření adekvátní funkce systému a řešení konfliktů způsobených v důsledku selhání některého z uzlů je pro potřeby práce zaznamenávání konfliktů povoleno.

```
bdr.conflict_logging_include_tuples = on
```

Zaznamenává kompletní data kolizních řádků tabulky při logování konfliktu. Je-li vypnuto, jsou zaznamenány pouze údaje popisující objekt, okamžik a umístění vzniku konfliktu bez modifikovaných hodnot.

```
bdr.synchronous_commit = off
```

Zapnutí synchronního potvrzení vynucuje fyzické zapsání dat na disk pomocí volání fsync na replice hned jakmile je na ní transakce z jiného uzlu aplikována. Ovlivňuje pouze nastavení fyzického uložení již přijatých a aplikovaných dat, nejedná se o vynucení synchronní metody replikace. Při nastavení volby off může proběhnout zápis na disk později – aplikování transakce je uzlu, který transakci zaslal, potvrzeno až po jejím zápisu na disk. Takové řešení zvyšuje výkon uzlu přijímajícího repliku, aniž by implikovalo vyšší pravděpodobnost ztráty dat při výpadku primárního uzlu, který transakci zpracovával – na něm již je transakce potvrzena a data uložena a replice již data byla také odeslána. Větší množství ztracených dat v důsledku této volby by bylo zaznamenáno pouze za situace fyzické ztráty dat na úložišti primárního uzlu a zároveň pádu uzlu, na který je replikováno dříve, než byla transakce zapsána na disk tohoto uzlu. Taková varianta však odpovídá dvěma nezávislým haváriím v rozsahu zlomku sekundy či několika sekund.

```
default_sequenceam = 'bdr'
```

Konfiguruje vytváření všech sekvencí jako globálních sekvencí. Uživatel nemusí zajišťovat vytvoření sekvencí na všech uzlech manuálně a má zajištěnu bezkoliznost generovaných hodnot.

Dále je nutné povolit přístup k databázi oprávněným uživatelům a uzlům pomocí souboru *pg_hba.conf*, například:

```
local   replication   postgres                    trust
host    replication   postgres     127.0.0.1/32 trust
host    replication   postgres     ::1/128      trust
host    replication, emmadb   postgres    37.157.XXX.XX/32 trust
host    replication, emmadb   postgres    5.135.XXX.XXX/32 trust
```

Každý záznam označuje databázi, uživatele, adresu, na které je spojení akceptováno a metodu ověření pro jednotlivé uzly. Je důležité, aby byl povolen přístup každému uzlu replikované skupiny. Při konfiguraci replikační skupiny je potřeba, aby byly uzly dosažitelné i v nereplikačním módu na adrese, prostřednictvím které jsou při replikaci adresovány. Z toho důvodu je kromě hodnoty *replication*, která zajišťuje přístup všem replikačním spojením bez ohledu na databázi, povolen přístup i na konkrétní databázi.

Pro vyšší úroveň zabezpečení přenosu je vhodné omezit oprávnění pro přístup pouze prostřednictvím zabezpečeného spojení (záznam *hostssl*) a uživatele vyhrazeného k provádění replikace. V tuto chvíli bohužel replikační uživatel musí být držitelem pravomocí *superuser* pro korektní funkčnost replikace.

Posledním krokem je vytvoření replikační skupiny a připojení všech uzlů do této skupiny. Toho je dosaženo prostřednictvím SQL relace. Na prvním uzlu je vytvořena replikační skupina pomocí následujících příkazů:

```
CREATE EXTENSION btree_gist;
CREATE EXTENSION bdr;
SELECT bdr.bdr_group_create(
      local_node_name := 'node1',
      node_external_dsn :=
            'host=5.135.XXX.XXX port=5432 dbname=emmadb'
);
```

local_node_name reprezentuje logické jméno uzlu, *node_external_dsn* řetězec identifikující daný uzel, prostřednictvím kterého se ostatní uzly připojují ke skupině. Na ostatních uzlech je stroj připojen do skupiny pomocí těchto příkazů:

```
CREATE EXTENSION btree_gist;
CREATE EXTENSION bdr;
SELECT bdr.bdr_group_join(
      local_node_name := 'node2',
      node_external_dsn :=
            'host=37.157.XXX.XXX port=5432 dbname=emmadb',
      join_using_dsn :=
            'host=5.135.XXX.XXX port=5432 dbname=emmadb'
);
```

join_using_dsn reprezentuje uzel, ke kterému se má aktuální uzel připojit. Tím může být libovolný z uzlů, které už se ve skupině nacházejí. První uzel, který skupinu vytvářel, nemá žádné speciální postavení mezi ostatními uzly skupiny.

Po úspěšném připojení jsou všechny změny provedené libovolným serverem replikovány na ostatní uzly. Konfigurace a nasazení BDR je díky kvalitní dokumentaci jednoduchým úkonem a nebyly při ní zaznamenány žádné problémy.

Funkčnost obosuměrné replikace byla oveřena simulovaným pádem jednoho z uzlů, kdy po jeho obnovení došlo k sesynchronizování změn provedených na druhém uzlu v době výpadku, a výsledná data byla na obou uzlech identická.

4.9.3 Konfigurace GlusterFS

Pro konfiguraci svazku je nejprve nutné přidat všechny servery, na kterých se bude svazek nacházet, do tzv. storage pool, pomocí příkazu: [33]

```
gluster peer probe server2
```

Dále je svazek vytvořen, jako čistě distribuovaný:

```
gluster volume create testvol replica 2 server1:/exp1
    server2:/exp2
```

Čistě replikovaný:

```
gluster volume create testvol server1:/exp1 server2:/exp2
```

Distribuovaně replikovaný:

```
gluster volume create testvol replica 2 server1:/exp1
    server2:/exp2 server3:/exp3 server4:/exp4
```

V této konfiguraci závisí na pořadí uvedených parametrů – v tomto případě bude server1:/exp1 replikován zdrojem server2:/exp2, a server3:/exp3 zdrojem server4:/exp4, a nad těmito dvěma skupinami bude vytvořen distribuovaný svazek, jak je patrné z obrázku 4.5. Poté je svazek spuštěn:

```
gluster volume start testvol
```

A může být připojen klienty:

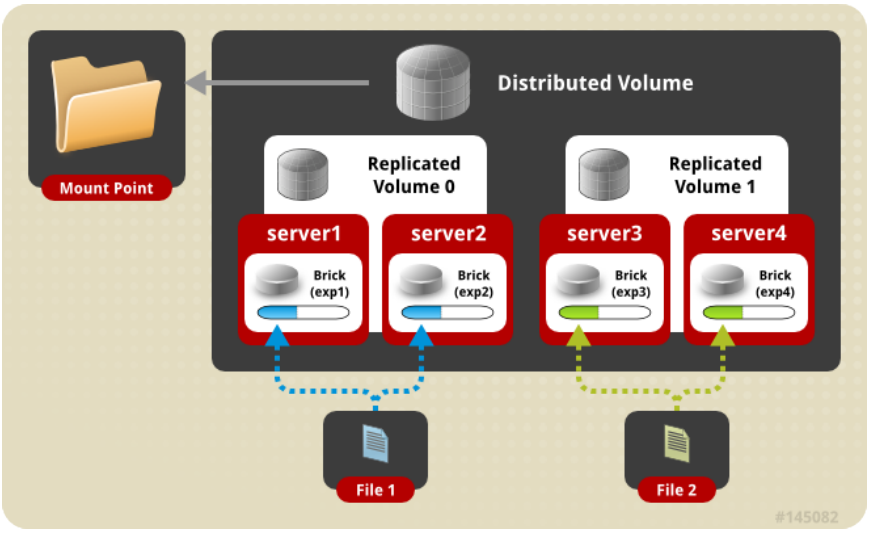
```
mount.glusterfs server1:/exp1 /mnt/gluster
```

Vzhledem k nasazení dle minimální konfigurace není využita replikace v rámci svazku, svazek je místo toho geo-replikován do svazku jiného. Je tedy potřeba připravit oba svazky. Svazky jsou nastaveny pro využití distribuovaného úložiště (pro budoucí škálování) pouze s jedním zdrojem:

```
gluster volume create gl1_primary 5.135.XXX.XXX:/data/1
gluster volume start gl1_primary
mount.glusterfs 5.135.XXX.XXX:/gl1_primary /mnt/emma_data/1
```

Podobná konfiruace je provedena na uzlu, na který bude probíhat geo-replikace:

```
gluster volume create gl1_replica 37.157.XXX.XX:/data/1
gluster volume start gl1_replica
mount.glusterfs 37.157.XXX.XX:/gl1_replica /mnt/emma_data/1
```



Obrázek 4.5: Schéma replikovaně-distribuovaného GlusterFS svazku [33]

Předpokladem pro úspěšné nasazení geo-replikace souborového systému je nastavení stejného aktuálního času na všech uzlech master svazku. Dále je nutné povolit spuštění Gluster démona pomocí SSH přístupu bez hesla (např. uložením veřejného klíče do *~/.ssh/authorized_keys*) z některého uzlu master svazku na stanici, kam má replikace probíhat. Poté jsou na straně master svazku provedeny příkazy:

```
gluster volume geo-replication gl1_primary
    5.135.XXX.XXX::gl1_replica create push-pem
gluster volume geo-replication gl1_primary
    5.135.XXX.XXX::gl1_replica start
```

Stav replikace je možné monitorovat příkazem:

```
gluster volume geo-replication gl1_primary
    5.135.XXX.XXX::gl1_replica status
```

Obdobná konfigurace dvojice svazků je provedena i pro opačný směr replikace. Aplikace je nastavena tak, aby její instance na serveru 5.135.XXX.XXX zapisovala do adresáře */mnt/emma_data/1*, zatímco instance uzlu 37.157.XXX.XX do adresáře */mnt/emma_data/2*. Čtení dat je umožněno oběma instancím z obou adresářů.

Škálování aplikace (v rámci ustanovených geografických lokalit) probíhá pomocí připojení dalších zdrojů do existujících svazků. V rámci těchto svazků může být nastavena i standardní (synchronní) replikace pro zamezení nutnosti převzetí role jiné instance aplikace v případě výpadku některého z datových uzlů. Konfigurace GlusterFS, ač minimalistická, na některých instancích

skončila chybovým stavem replikace a nutností upravovat konfigurační soubor geo-replikace *gsyncd.conf*.

Závěr

V rámci této práce byla provedena analýza požadavků na systém, využitelný pro správu zákaznické podpory společnosti. Dle těchto požadavků byly navrženy a implementovány dvě komponenty systému – serverová aplikace a desktopová klientská aplikace. Pro serverovou část aplikace byly zvoleny technologie, umožňující zachování funkčnosti systému i v případě havárie některé z jeho komponent. Práce vytvořila různé plány nasazení dle dostupného počtu serverů, vhodně alokující komponenty systému na jednotlivé stroje. Systém byl prakticky nasazen a byla ověřena funkčnost a odolnost zvoleného řešení.

Nemalou částí práce se ukázal být korektní návrh rozhraní webových služeb REST v jeho autorem zamýšlené podobě, respektující veškeré předložené principy. Dynamické sestavování odkazů a objevitelnost rozhraní skýtá výhody v rozšiřitelnosti služby a nezávislosti klientů služeb na struktuře a umístění služby, nicméně je vykoupeno nutností využít rozsáhlejší aparát technologií a způsobů práce se službou na serverové i klientské straně služby. Vyhnutí se využití adres nereprezentujících zdroje, ale vykonání operace či přiřazení zdrojů, vynucuje specifickou architekturu zdrojů, která nemusí být plně vhodná k reprezentaci rozhraní služeb. V tomto kontextu je pochopitelné využití nižší úrovně REST, resp. porušení některého z touto architekturou definovaných principů naprostou většinou veřejně dostupných webových služeb v prostředí internetu.

Naopak je na místě vyzdvihnout přínosy frameworku Spring, jehož jednotlivé moduly výrazně urychlily vývoj aplikace a práce s většinou z nich byla bezproblémová. Množství nutné konfigurace nepřesahovalo vhodnou mez a využití značkovacích anotací eliminovalo velké množství propojovacího kódu. Zejména na projektu Spring HATEOAS je však patrné jeho relativní mládí a lze očekávat vylepšení podpory vytváření služeb s odkazy v příštích verzích projektu.

Systém vzešlý z této práce je plně použitelný pro produkční nasazení a počítá s průběžnou integrací dalších funkcí dle nových požadavků společností.

Literatura

[1] SUN, Leo. Better Buy: Zendesk Inc or Salesforce.com Inc? *The Montley Fool* [online]. 2015 [cit. 2016-05-09]. Dostupné z: <http://www.fool.com/investing/general/2015/11/30/better-buy-zendesk-inc-or-salesforcecom-inc.aspx >

[2] KANARACUS, Chris. Salesforce takes a shot at Zendesk, Freshdesk with Desk.com upgrade. *IT World* [online]. 2014 [cit. 2016-05-09]. Dostupné z: <http://www.itworld.com/article/2695764/enterprise-software/salesforce-takes-a-shot-at-zendesk--freshdesk-with-desk-com-upgrade.html >

[3] RAHMAN, Reza. CAUCHO TECHNOLOGY, INC. *RESIN APPLICATION SERVER JAVA EE 6 WEB PROFILE.* San Diego, 2011.

[4] Spring Data. *Spring* [online]. [cit. 2016-05-09]. Dostupné z: <http://projects.spring.io/spring-data/ >

[5] FIELDING, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures.* Irvine, 2000. Disertace. University of California.

[6] FIELDING, Roy. REST APIs must be hypertext-driven. *Untangled* [online]. 2008 [cit. 2016-05-09]. Dostupné z: <http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven >

[7] FOWLER, Martin. Richardson Maturity Model. *MartinFowler.com* [online]. 2010 [cit. 2016-05-09]. Dostupné z: <http://martinfowler.com/articles/richardsonMaturityModel.html >

[8] JIM WEBBER, Savas Parastatidis and Ian Robinson. *REST in practice.* Farnham: O'Reilly, 2010. ISBN 978-059-6805-821.

[9] PALMER, Dan. Your API is not RESTful. *DanPalmer.me* [online]. [cit. 2016-05-09]. Dostupné z: <https://danpalmer.me/blog/your-api-is-not-restful >

[10] SPORNY, Manu, Dave LONGLEY, Gregg KELOGG, Markus LANTHALER a Niklas LINDSTRÖM. JSON-LD 1.0. *W3C.org* [online]. 2014 [cit. 2016-05-09]. Dostupné z: <https://www.w3.org/TR/json-ld/#basic-concepts >

[11] KELLY, Mike. JSON Hypertext Application Language. *IETF Tools* [online]. 2015 [cit. 2016-05-09]. Dostupné z: <https://tools.ietf.org/html/draft-kelly-json-hal-07 >

[12] KELLY, Mike. HAL – Hypertext Application Language. *Stateless API Consulting* [online]. 2011 [cit. 2016-05-09]. Dostupné z: <http://stateless.co/hal_specification.html >

[13] FRANKS, J., P. HALLAM-BAKER, J. HOSTETLER, S. LAWRENCE, P. LEACH, A. LUOTONEN a L. STEWART. HTTP Authentication: Basic and Digest Access Authentication. *IETF.org* [online]. 1999 [cit. 2016-05-09]. Dostupné z: <https://www.ietf.org/rfc/rfc2617.txt >

[14] FIELDING, R. a J. RESCHKE. Hypertext Transfer Protocol (HTTP/1.1): Authentication. *IETF tools* [online]. 2014 [cit. 2016-05-09]. Dostupné z: <https://tools.ietf.org/html/rfc7235 >

[15] REESE, George. Principles for Standardized REST Authentication. *O'Reilly Community* [online]. 2009 [cit. 2016-05-09]. Dostupné z: <http://broadcast.oreilly.com/2009/12/principles-for-standardized-rest-authentication.html >

[16] JONES, M., J. BRADLEY a N. SAKIMURA. JSON Web Token (JWT). *IETF tools* [online]. 2015 [cit. 2016-05-09]. Dostupné z: <https://tools.ietf.org/html/rfc7519 >

[17] Introduction to JSON Web Tokens. *JWT.io* [online]. [cit. 2016-05-09]. Dostupné z: <https://jwt.io/introduction/ >

[18] About. *PostgreSQL* [online]. [cit. 2016-05-09]. Dostupné z: <http://www.postgresql.org/about/ >

[19] SEARS, Russell, Catharine VAN INGEN a Jim GRAY. To BLOB or Not To BLOB: Large Object Storage in a Database or a Filesystem. *Microsoft Research* [online]. 2006, 2006(MSR-TR-2006-45), 10 [cit. 2016-05-09]. Dostupné z: <http://research.microsoft.com/apps/pubs/default.aspx?id=64525 >

[20] MILOSEVIC, Dejan. REST Security with JWT using Java and Spring Security. *Toptal* [online]. 2015 [cit. 2016-05-09]. Dostupné z: <https://www.toptal.com/java/rest-security-with-jwt-spring-security-and-java >

[21] BAUER, Christian a Gavin. KING. *Java persistence with Hibernate.* Rev. ed. London: Pearson Education [distributor], 2007. ISBN 978-193-2394-887.

[22] JavaMail API documentation: Class IMAPFolder. *JavaMail API* [online]. [cit. 2016-05-09]. Dostupné z: <https://javamail.java.net/nonav/docs/api/com/sun/mail/imap/IMAPFolder.html#getUIDNext-- >

[23] LEIBA, B. IMAP4 IDLE command. *IETF Tools* [online]. 1997 [cit. 2016-05-09]. Dostupné z: <https://tools.ietf.org/html/rfc2177 >

[24] OclHashcat. *Hashcat.net* [online]. [cit. 2016-05-09]. Dostupné z: <http://hashcat.net/oclhashcat/ >

[25] ABBOTT, Tom. Use JWT The Right Way! *Stormpath* [online]. 2014 [cit. 2016-05-09]. Dostupné z: <https://stormpath.com/blog/jwt-the-right-way >

[26] Learn REST: A RESTful Tutorial: Resource Naming. *REST API Tutorial* [online]. [cit. 2016-05-09]. Dostupné z: <http://www.restapitutorial.com/lessons/restfulresourcenaming.html >

[27] KANG, Abraham. J2EE clustering, Part 1. *JavaWorld* [online]. 2001 [cit. 2016-05-09]. Dostupné z: <http://www.javaworld.com/article/2075019/jndi/j2ee-clustering--part-1.html >

[28] MESSINGER, Lior. Better explaining the CAP Theorem. *DZone* [online]. 2013 [cit. 2016-05-09]. Dostupné z: <https://dzone.com/articles/better-explaining-cap-theorem >

[29] PostgreSQL 9.4.7 Documentation: Comparison of Different Solutions. THE POSTGRESQL GLOBAL DEVELOPMENT GROUP. *PostgreSQL* [online]. [cit. 2016-05-09]. Dostupné z: <http://www.postgresql.org/docs/9.4/static/different-replication-solutions.html >

[30] Slony-I. *Slony-I* [online]. [cit. 2016-05-09]. Dostupné z: <http://slony.info/ >

[31] BDR. *2ndQuadrant* [online]. [cit. 2016-05-09]. Dostupné z: <http://2ndquadrant.com/en/resources/bdr/ >

[32] BDR 0.10.0 Documentation: Chapter 9. Multi-master conflicts. *bdr-project.org* [online]. 2016 [cit. 2016-05-09]. Dostupné z: <http://bdr-project.org/docs/next/conflicts-types.html >

[33] GlusterFS Documentation: Setting up GlusterFS Server Volumes. *Gluster Docs* [online]. [cit. 2016-05-09]. Dostupné z: <http://gluster.readthedocs.io/en/latest/Administrator%20Guide/Setting%20Up%20Volumes/ >

Příloha **A**

Seznam použitých zkratek

API Application programming interface

ASCII American standard code for information interchange

CRM Customer relationship management

CRUD Create, retrieve, update, delete

DDL Data definition language

DIR Directory service

DKIM DomainKeys identified mail

DML Data manipulation language

HAL Hypertext application language

HATEOAS Hypermedia as the engine of application state

HDFS Hadoop distributed file system

HMAC Keyed-hash message authentication code

HTTP Hypertext transfer protocol

HTTPS Hypertext transfer protocol secure

IANA Internet Assigned Numbers Authority

IETF Internet Engineering Task Force

IMAP Internet message access protocol

JAR Java archive

Java EE Java Platform, Enterprise Edition

Java SE Java Platform, Standard Edition

JAX-RS Java API for RESTful Web Services

JDBC Java Database Connectivity

JPA Java Persistence API

JSON JavaScript object notation

JSP Java server pages

JWT JSON web token

MIME Multipurpose internet mail extensions

MRC Metadata and replica catalog

MVC Model-view-controller

OSD Object storage device

POP Post office protocol

RAID Redundant array of independent disks

RAM Random access memory

REST Representational state transfer

RFC Request for comments

SaaS Software as a service

SHA Secure hash algorithm

SMTP Simple mail transfer protocol

SMTPS Simple mail transfer protocol secure

SOAP Simple object access protocol

SPF Sender policy framework

SQL Structured query language

TCP Transmission control protocol

UID Unique identification number

URI Uniform resource identifier

URL Uniform resource locator

WAL Write-ahead log

WORM Write-once read-many

XML Extensible markup language

Příloha B

Specifikace REST API

B.1 Popis rozhraní

B.1.1 Úvod

Tento dokument popisuje specifikaci REST rozhraní pro komunikaci mezi klientem a serverem projektu EMMA. API respektuje principy HATEOAS a všechny zdroje jsou tak dostupné za pomocí série odkazů z kořenového zdroje.

B.1.2 Formát komunikace

Kromě výjimek uvedených v této specifikaci server poskytuje odpovědi na požadavky ve formátu HAL (Hypertext Application Language) a JSON. Internet media type odpovědí je *application/hal+json*. Klient musí být schopen zpracovávat tento formát a musí schopnost jej akceptovat propagovat pomocí HTTP hlavičky *Accept*. Data jsou kódována ve znakové sadě *UTF-8*.

Klient musí své požadavky činit ve formátu JSON, Internet media type *application/json*. Data musí být kódována ve znakové sadě *UTF-8*.

B.1.3 Autentizace

Přístup ke všem zdrojům kromě kořenového vyžaduje autentizovaného uživatele. Ověření uživatelské identity je prováděno při každém požadavku; klient je povinen ke každému požadavku připojit HTTP hlavičku ve formátu:

```
Authorization: Bearer {data}
```

kde *{data}* je hodnota enkódovaného JWT tokenu, reprezentujícího přihlášeného uživatele. Tento token je možno získat na adrese */login* (viz sekce B.3). Klient není povinen rozumět obsahu JWT tokenu ani jej být schopen dekódovat; pokud obsahu tokenu rozumí, může použít data v něm obsažená pro rozhodování o podobě dotazu (např. iniciovat nový požadavek na přihlášení,

pokud datum platnosti tokenu již uplynulo).

Struktura dekódovaného JWT tokenu je následující:
Hlavička:

```
{
  "alg": "HS512"
}
```

Payload (příklad):

```
{
  "sub": "john.doe@domain.com",
  "iat": 1461434729,
  "iss": "EMMA 0.0.1",
  "exp": 1463234729,
  "cid": 1,
  "permissions": [
    "get_TicketResource",
    "get_UserResource",
    "list_TicketResource",
    "list_UserResource"
  ]
}
```

Používáný algoritmus podpisu je HS512, tedy HMAC využívající SHA-512 hashovací algoritmus. Popis položek tokenu:

Název položky	Typ proměnné	Popis
sub	řetězec	uživatelské jméno reprezentovaného uživatele
iat	časová značka	časová značka vystavení tokenu
iss	řetězec	vydavatel tokenu; obvykle spočívá v názvu a verze aplikace a označení vydávajícího serveru
exp	časová značka	časová značka okamžiku expirace tokenu

Poznámka: Jiné než výše popsané položky obsahu tokenu mohou být přiloženy dle uvážení serveru. Klient jim nesmí přikládat žádný význam.

B.1.4 Zabezpečení

API je přístupné pouze prostřednictvím zabezpečeného spojení protokolem HTTPS. Klient nesmí činit žádné požadavky nezabezpečeným protokolem HTTP a nesmí důvěřovat serveru službu HTTPS neposkytující.

B.2 /

Podporované HTTP metody: GET, HEAD, OPTIONS

B.2.1 GET

Vrátí kořenový zdroj aplikace s odkazy na ostatní zdroje.

Obsah odpovědi: zdroj typu Root

Název položky	Typ proměnné	Popis
_links	sekce s odkazy	sekce s odkazy pro navigování

Obsah sekce s navigačními odkazy (_links):

Název položky	Typ proměnné	Popis
login	odkaz	odkaz na adresu pro přihlášení (získání JWT tokenu)
me	odkaz	odkaz na zdroj odpovídající autentizovanému uživateli
attachments	odkaz	odkaz na (stránkovanou) kolekci všech příloh společnosti
customers	odkaz	odkaz na (stránkovanou) kolekci všech zákazníků společnosti
emailAccounts	odkaz	odkaz na (stránkovanou) kolekci všech e-mailových účtů společnosti
groups	odkaz	odkaz na (stránkovanou) kolekci všech skupin společnosti
permissions	odkaz	odkaz na (stránkovanou) kolekci všech oprávnění, které v systému existují
roles	odkaz	odkaz na (stránkovanou) kolekci všech rolí společnosti
tickets	odkaz	odkaz odkaz na (stránkovanou) kolekci všech tiketů společnosti
users	odkaz	odkaz na (stránkovanou) kolekci všech uživatelů společnosti
tags	odkaz	odkaz na (stránkovanou) kolekci všech štítků společnosti
textTemplates	odkaz	odkaz na (stránkovanou) kolekci všech šablon textů společnosti
companies	odkaz	odkaz na kolekci společností

B.3 /login

Podporované HTTP metody: POST

B.3.1 POST

Vrátí JWT token reprezentující autentizovaného uživatele. Internet media format odpovědi je *application/jwt*. Požadavek je vznesen ve formátu *application/x-www-form-urlencoded*.

Parametry požadavku:

Název položky	Typ proměnné	Popis
login	řetězec	uživatelské jméno přihlašovaného uživatele
password	řetězec	heslo přihlašovaného uživatele v plaintextu

Poznámka: Klient musí indikovat schopnost akceptovat formát *application/jwt* pomocí *Accept* hlavičky požadavku.

Obvyklé stavové kódy:
200 OK pokud byl uživatel úspěšně autentizován
401 UNAUTHORIZED pokud zadané uživatelské jméno nebo heslo není správné

B.4 /me

Podporované HTTP metody: GET, HEAD, OPTIONS

B.4.1 GET

Vrátí aktuálně přihlášeného uživatele.

Obsah odpovědi: zdroj typu User (dle sekce B.6.1)

B.5 /users

Podporované HTTP metody: GET, POST, HEAD, OPTIONS

B.5.1 GET

Vrátí kolekci všech uživatelů dané společnosti.

Obsah odpovědi: stránka (dle sekce B.32), obsahující kolekci zdrojů typu User (dle sekce B.6.1) pod názvem *users*

B.5.2 POST

Vytvoří nového uživatele.

Tělo požadavku:

Název položky	Typ proměnné	Popis
deleted	logická hodnota	reprezentuje, zda byl tento zdroj smazán; musí být *false*
name	řetězec	reprezentuje skutečné jméno a příjmení uživatele
username	řetězec	reprezentuje přihlašovací jméno uživatele (e-mailovou adresu)
contactEmail	řetězec	reprezentuje kontaktní e-mailovou adresu uživatele
password	řetězec	reprezentuje uživatelovo heslo v plaintextu
managedGroups	kolekce	kolekce odkazů na skupiny, do kterých uživatel náleží
roles	kolekce	kolekce odkazů na role, které uživatel má

Obvyklé stavové kódy:
200 OK pokud byl zdroj úspěšně vytvořen
422 UNPROCESSABLE ENTITY pokud uživatelské jméno již existuje, uživatelské jméno je prázdné, heslo je prázdné, některá z odkazovaných skupin či rolí neexistuje, společnost autentizovaného uživatele není nastavena nebo neexistuje, nebo *deleted* je nastaveno na *true*

B.6 /users/{id}

Podporované HTTP metody: GET, PUT, PATCH, DELETE, HEAD, OPTIONS

B.6.1 GET

Vrátí požadovaného uživatele.

Obsah odpovědi: zdroj User

Název položky	Typ proměnné	Popis
version	časová značka	časová značka aktuální verze zdroje, využita pro concurrency control

deleted	logická hodnota	reprezentuje, zda byl tento zdroj smazán
name	řetězec	reprezentuje skutečné jméno a příjmení uživatele
username	řetězec	reprezentuje přihlašovací jméno uživatele (e-mailovou adresu)
contactEmail	řetězec	reprezentuje kontaktní e-mailovou adresu uživatele
password	řetězec	vrací vždy prázdný řetězec, uživatelovo heslo není navráceno
managedGroups	kolekce	kolekce odkazů na skupiny, do kterých uživatel náleží, včetně smazaných; je zodpovědností klienta profiltrovat nesmazané skupiny
roles	kolekce	kolekce odkazů na role, které uživatel má
_links	sekce s odkazy	sekce s odkazy pro navigování
_embedded	sekce se zdroji	sekce vnořených zdrojů

Poznámka: Hodnota položky password v odpovědi je vždy prázdným řetězcem, uživatelovo heslo není navráceno

Obsah sekce s navigačními odkazy (_links):

Název položky	Popis
self	odkaz na tento zdroj
unassignedTickets	odkaz na tikety, které jsou přiřazené některé skupině, do které uživatel náleží, a nejsou přiřazeny žádnému uživateli
assignedTickets	odkaz na tikety přiřazené uživateli
sentTickets	odkaz na tikety přiřazené uživateli sentTickets odkaz na tikety, na které uživatel odpověděl nebo je předal
closedTickets	odkaz na uzavřené tikety, které náleží tomuto uživateli, nebo na ně uživatel odpověděl nebo je předal
textTemplates	odkaz na šablony textů, náležející tomuto uživateli

Obsah sekce vnořených zdrojů (_embedded):

Název položky	Typ proměnné	Popis
managedGroups	kolekce	kolekce skupin, do kterých uživatel náleží, včetně smazaných; je zodpovědností klienta profiltrovat nesmazané skupiny

roles	kolekce	kolekce rolí, které uživatel má
assignedTickets	kolekce	kolekce uživateli přiřazených tiketů

Poznámka: Volba, zda budou tyto zdroje vnořeny, je na uvážení serveru. Klient nemůže spoléhat na přítomnost těchto vnořených zdrojů. Vnořené zdroje obvykle dále neobsahují další vnořené zdroje.

B.6.2 PUT

Edituje kompletní reprezentaci požadovaného uživatele.

Tělo požadavku:

Název položky	Typ proměnné	Popis
version	časová značka	časová značka aktuální verze zdroje, musí být shodná s obdrženou časovou značkou tohoto zdroje
deleted	logická hodnota	reprezentuje, zda byl tento zdroj smazán; hodnota musí být shodná s obdrženou hodnotou tohoto zdroje
name	řetězec	reprezentuje skutečné jméno a příjmení uživatele
username	řetězec	reprezentuje přihlašovací jméno uživatele (e-mailovou adresu); hodnota musí být shodná s obdrženou hodnotu tohoto zdroje
contactEmail	řetězec	reprezentuje kontaktní e-mailovou adresu uživatele
password	řetězec	reprezentuje uživatelovo heslo v plaintextu
managedGroups	kolekce	kolekce odkazů na skupiny, do kterých uživatel náleží
roles	kolekce	kolekce odkazů na role, které uživatel má

Obvyklé stavové kódy:
200 OK pokud byl zdroj úspěšně upraven
404 NOT FOUND pokud uživatel s touto URL neexistuje
412 PRECONDITION FAILED pokud má server novější verzi zdroje, než je upravována
422 UNPROCESSABLE ENTITY pokud uživatelské jméno neodpovídá původnímu uživatelskému jménu, heslo je prázdné, některá z odkazovaných skupin či rolí neexistuje, společnost autentizovaného uživatele není nastavena

nebo neexistuje, nebo je uživatel smazán (nastavena položka *deleted* na hodnotu *true*)

Poznámka: Nelze editovat smazaného (*deleted*) uživatele.

B.6.3 PATCH

Edituje nekompletní reprezentaci požadovaného uživatele.

Syntaxe je shodná s metodou PUT (sekce B.6.2), s výjimkou možnosti v požadavku neuvádět needitované položky. Tato výjimka se netýká položky *version*, která musí být vždy přítomna.

B.6.4 DELETE

Smaže požadovaného uživatele.

Obvyklé stavové kódy:
200 OK pokud byl uživatel úspěšně smazán
404 NOT FOUND pokud uživatel s touto URL neexistuje

Poznámka: Smazáním uživatele dojde k nastavení položky *deleted* na hodnotu *true*, nedojde k odstranění samotného zdroje. Požadavek nicméně stále může vyvolat vedlejší efekty, např. změnu vztahů tohoto zdroje s jinými zdroji.

B.7 /groups

Podporované HTTP metody: GET, POST, HEAD, OPTIONS

B.7.1 GET

Vrátí kolekci všech skupin dané společnosti.

Obsah odpovědi: stránka (dle dle sekce B.32), obsahující kolekci zdrojů typu Group (dle sekce B.8.1) pod názvem *groups*

B.7.2 POST

Vytvoří novou skupinu.

Tělo požadavku:

Název položky	Typ proměnné	Popis
deleted	logická hodnota	reprezentuje, zda byl tento zdroj smazán; musi být *false*

name	řetězec	reprezentuje jméno skupiny
email	řetězec	řetězec reprezentuje e-mailovou adresu skupiny
emailAccount	odkaz	reprezentuje odkaz na e-mailový účet náležící této skupině

Obvyklé stavové kódy:
200 OK pokud byl zdroj úspěšně vytvořen
422 UNPROCESSABLE ENTITY pokud je jméno i email prázdný, odkazovaný e-mailový účet neexistuje, společnost autentizovaného uživatele není nastavena nebo neexistuje, nebo je položka *deleted* nastavena na hodnotu *true*

Poznámka: Přiřazování uživatelů náležící do skupiny se provádí pomocí editace uživatele, není možno je specifikovat při vytváření skupiny. Uživatel je vlastnící zdroj tohoto vztahu.

B.8 /groups/{id}

Podporované HTTP metody: GET, PUT, PATCH, DELETE, HEAD, OPTIONS

B.8.1 GET

Vrátí požadovanou skupinu.

Obsah odpovědi: zdroj Group

Název položky	Typ proměnné	Popis
version	časová značka	časová značka aktuální verze zdroje, využita pro concurrency control
deleted	logická hodnota	reprezentuje, zda byl tento zdroj smazán
name	řetězec	reprezentuje jméno skupiny
email	řetězec	řetězec reprezentuje e-mailovou adresu skupiny
emailAccount	odkaz	reprezentuje odkaz na e-mailový účet náležící této skupině
users	kolekce	kolekce odkazů na uživatele, kteří do této skupiny náleží, včetně smazaných; je zodpovědností klienta profiltrovat nesmazané uživatele
_links	sekce s odkazy	sekce s odkazy pro navigování

_embedded	sekce se zdroji	sekce vnořených zdrojů

Obsah sekce s navigačními odkazy (_links):

Název položky	Popis
self	odkaz na tento zdroj
unassignedTickets	odkaz na tikety, které jsou přiřazené této skupině, a nejsou přiřazeny žádnému uživateli
assignedTickets	odkaz na tikety, které jsou přiřazené této skupině, a jsou přiřazeny nějakému uživateli
sentTickets	odkaz na tikety, na které bylo z této skupiny odpovězeno nebo byly předány jiné skupině
closedTickets	odkaz na uzavřené tikety, které jsou přiřazeny této skupině, nebo na ně bylo z této skupiny odpovězeno či byly předány jiné skupině
textTemplates	odkaz na šablony textů, náležející této skupině

Obsah sekce vnořených zdrojů (_embedded):

Název položky	Typ proměnné	Popis
users	kolekce	kolekce uživatelů, kteří náleží do této skupiny, včetně smazaných; je zodpovědností klienta profiltrovat nesmazané uživatele

Poznámka: Volba, zda budou tyto zdroje vnořeny, je na uvážení serveru. Klient nemůže spoléhat na přítomnost těchto vnořených zdrojů. Vnořené zdroje obvykle dále neobsahují další vnořené zdroje.

B.8.2 PUT

Edituje kompletní reprezentaci požadované skupiny.

Tělo požadavku:

Název položky	Typ proměnné	Popis
version	časová značka	časová značka aktuální verze zdroje, musí být shodná s obdrženou časovou značkou tohoto zdroje
deleted	logická hodnota	reprezentuje, zda byl tento zdroj smazán; hodnota musí být shodná s obdrženou hodnotou tohoto zdroje
name	řetězec	reprezentuje jméno skupiny

email	řetězec	řetězec reprezentuje e-mailovou adresu skupiny
emailAccount	odkaz	reprezentuje odkaz na e-mailový účet náležící této skupině

Obvyklé stavové kódy:
200 OK pokud byla skupina úspěšně upravena
404 NOT FOUND pokud skupina s touto URL neexistuje
412 PRECONDITION FAILED pokud má server novější verzi zdroje, než je upravována
422 UNPROCESSABLE ENTITY pokud je jméno i email prázdný, odkazovaný e-mailový účet neexistuje, společnost autentizovaného uživatele není nastavena nebo neexistuje, nebo je skupina smazána (nastavena položka *deleted* na hodnotu *true*)

Poznámka: Přiřazování uživatelů náležící do skupiny se provádí pomocí editace uživatele, není možno je specifikovat při editaci skupiny. Uživatel je vlastnící zdroj tohoto vztahu. Nelze editovat smazanou (*deleted*) skupinu.

B.8.3 PATCH

Edituje nekompletní reprezentaci požadované skupiny.

Syntaxe je shodná s metodou PUT (sekce B.8.2), s výjimkou možnosti v požadavku neuvádět needitované položky. Tato výjimka se netýká položky *version*, která musí být vždy přítomna.

B.8.4 DELETE

Smaže požadovanou skupinu.

Obvyklé stavové kódy:
200 OK pokud byla skupina úspěšně smazána
404 NOT FOUND pokud skupina s touto URL neexistuje

Poznámka: Smazáním skupiny dojde k nastavení položky *deleted* na hodnotu *true*, nedojde k odstranění samotného zdroje. Požadavek nicméně stále může vyvolat vedlejší efekty, např. změnu vztahů tohoto zdroje s jinými zdroji.

B.9 /customers

Podporované HTTP metody: GET, POST, HEAD, OPTIONS

B.9.1 GET

Vrátí kolekci všech zákazníků dané společnosti.

Obsah odpovědi: stránka (dle sekce B.32), obsahující kolekci zdrojů typu Customer (dle sekce B.10.1) pod názvem *customers*

B.9.2 POST

Vytvoří nového zákazníka.

Tělo požadavku:

Název položky	Typ proměnné	Popis
email	řetězec	řetězec reprezentuje e-mailovou adresu zákazníka
name	řetězec	reprezentuje jméno zákazníka
telephone	řetězec	reprezentuje telefonní číslo zákazníka
address	Address	reprezentuje adresu zákazníka

Obvyklé stavové kódy:
200 OK pokud byl zdroj úspěšně vytvořen
422 UNPROCESSABLE ENTITY pokud společnost autentizovaného uživatele není nastavena nebo neexistuje

B.10 /customers/{id}

Podporované HTTP metody: GET, PUT, PATCH, DELETE, HEAD, OPTIONS

B.10.1 GET

Vrátí požadovaného zákazníka.

Obsah odpovědi: zdroj Customer

Název položky	Typ proměnné	Popis
version	časová značka	časová značka aktuální verze zdroje, využita pro concurrency control
email	řetězec	reprezentuje e-mailovou adresu zákazníka
name	řetězec	reprezentuje jméno zákazníka
telephone	řetězec	reprezentuje telefonní číslo zákazníka
address	Address	reprezentuje adresu zákazníka

_links	sekce s odkazy	sekce s odkazy pro navigování
_embedded	sekce se zdroji	sekce vnořených zdrojů

Obsah sekce s navigačními odkazy (_links):

Název položky	Popis
self	odkaz na tento zdroj
tickets	odkaz na všechny tikety tohoto zákazníka

Obsah sekce vnořených zdrojů (_embedded):

Název položky	Typ proměnné	Popis
tickets	kolekce	kolekce všech tiketů tohoto zákazníka

B.10.2 PUT

Edituje kompletní reprezentaci požadovaného zákazníka.

Tělo požadavku:

Název položky	Typ proměnné	Popis
version	časová značka	časová značka aktuální verze zdroje, musí být shodná s obdrženou časovou značkou tohoto zdroje
email	řetězec	reprezentuje e-mailovou adresu zákazníka
name	řetězec	reprezentuje jméno zákazníka
telephone	řetězec	reprezentuje telefonní číslo zákazníka
address	Address	reprezentuje adresu zákazníka

Obvyklé stavové kódy:
200 OK pokud byl zákazník úspěšně upraven
404 NOT FOUND pokud zákazník s touto URL neexistuje
412 PRECONDITION FAILED pokud má server novější verzi zdroje, než je upravována
422 UNPROCESSABLE ENTITY pokud společnost autentizovaného uživatele není nastavena nebo neexistuje

B.10.3 PATCH

Edituje nekompletní reprezentaci požadovaného zákazníka.

Syntaxe je shodná s metodou PUT (sekce B.10.2), s výjimkou možnosti v požadavku neuvádět needitované položky. Tato výjimka se netýká položky *version*, která musí být vždy přítomna.

B.10.4 DELETE

Odstraní požadovaného zákazníka.

Obvyklé stavové kódy:
200 OK pokud byl zákazník úspěšně odstraněn
404 NOT FOUND pokud zákazník s touto URL neexistuje
409 CONFLICT pokud jsou tomuto zákazníkovi přiřazeny některé tikety

Poznámka: Je možné odstraňovat pouze zákazníky, kterým nejsou přiřazeny žádné tikety.

B.11 /tickets

Podporované HTTP metody: GET, POST, HEAD, OPTIONS

B.11.1 GET

Vrátí kolekci všech tiketů dané společnosti.

Obsah odpovědi: stránka (dle sekce B.32), obsahující kolekci zdrojů typu Ticket (dle sekce B.12.1) pod názvem *tickets*

B.11.2 POST

Vytvoří nový tiket.

Tělo požadavku:

Název položky	Typ proměnné	Popis
subject	řetězec	reprezentuje předmět tiketu
type	řetězec	reprezentuje typ tiketu; akceptovatelná hodnota je pouze *INTERNAL*
closed	logická hodnota	reprezentuje, zda je tiket vyřízen
tags	kolekce	kolekce štítků, které mají být k tiketu přiřazeny

Obvyklé stavové kódy:
200 OK pokud byl zdroj úspěšně vytvořen
422 UNPROCESSABLE ENTITY pokud typ vytvářeného tiketu není nasta-

ven na hodnotu *INTERNAL*, některý z odkazovaných štítků neexistuje nebo společnost autentizovaného uživatele není nastavena nebo neexistuje

B.12 /tickets/{id}

Podporované HTTP metody: GET, PUT, PATCH, DELETE, HEAD, OPTIONS

B.12.1 GET

Vrátí požadovaný tiket.

Obsah odpovědi: zdroj Ticket

Název položky	Typ proměnné	Popis
version	časová značka	časová značka aktuální verze zdroje, využita pro concurrency control
subject	řetězec	reprezentuje předmět tiketu
type	řetězec	reprezentuje typ tiketu, validní hodnoty jsou *EMAIL* a *INTERNAL*
closed	logická hodnota	reprezentuje, zda je tiket vyřízen
creationDate	časová značka	reprezentuje datum vytvoření tiketu
lastUpdateDate	časová značka	reprezentuje datum poslední změny tiketu či vytvoření nového záznamu tiketu
involvesMe	logická hodnota	reprezentuje, zda autentizovaný uživatel odpověděl na tento tiket nebo jej předal jinému uživateli či skupině
seenByMe	logická hodnota	reprezentuje, zda autentizovaný uživatel viděl poslední změnu tiketu
tags	kolekce	kolekce štítků, které jsou k tiketu přiřazeny
_links	sekce s odkazy	sekce s odkazy pro navigování
_embedded	sekce se zdroji	sekce vnořených zdrojů

Obsah sekce s navigačními odkazy (_links):

Název položky	Popis
self	odkaz na tento zdroj
customer	odkaz na zákazníka, kterého se tiket týká, může být *null*
author	odkaz na uživatele, který tiket vytvořil; může být *null*, pokud byl tiket vytvořen systémem
assignedUser	odkaz na uživatele, kterému je tiket přiřazen; může být *null*

assignedGroup	odkaz na skupinu, které je tiket přiřazen; může být *null*
ticketEntries	odkaz na jednotlivé záznamy tohoto tiketu
attachments	odkaz na všechny přílohy tohoto tiketu, včetně smazaných; je zodpovědností klienta profiltrovat nesmazané přílohy

Obsah sekce vnořených zdrojů (_embedded):

Název položky	Typ proměnné	Popis
customer	Customer	zákazník, kterému je tiket přiřazen
author	User	uživatel, který tiket vytvořil
assignedUser	User	uživatel, kterému je tiket přiřazen
assignedGroup	Group	skupina, které je tiket přiřazen
entries	kolekce	záznamy tohoto tiketu
attachments	kolekce	přílohy tohoto tiketu, včetně smazaných
tags	kolekce	štítky, které jsou k tiketu přiřazeny

Poznámka: Volba, zda budou tyto zdroje vnořeny, je na uvážení serveru. Klient nemůže spoléhat na přítomnost těchto vnořených zdrojů. Vnořené zdroje obvykle dále neobsahují další vnořené zdroje.

B.12.2 PUT

Edituje kompletní reprezentaci požadovaného tiketu.

Tělo požadavku:

Název položky	Typ proměnné	Popis
version	časová značka	časová značka aktuální verze zdroje, musí být shodná s obdrženou časovou značkou tohoto zdroje
subject	řetězec	reprezentuje předmět tiketu
type	řetězec	reprezentuje typ tiketu, hodnota musí být shodná s obdrženou hodnotou tohoto zdroje
closed	logická hodnota	reprezentuje, zda je tiket vyřízen, hodnota musí být shodná s obdrženou hodnotou tohoto zdroje
creationDate	časová značka	hodnota je při editaci ignorována
lastUpdateDate	časová značka	hodnota je při editaci ignorována
involvesMe	logická hodnota	hodnota je při editaci ignorována

seenByMe	logická hodnota	reprezentuje, zda autentizovaný uživatel viděl poslední změnu tiketu
tags	kolekce	kolekce štítků, které jsou k tiketu přiřazeny

Obvyklé stavové kódy:
200 OK pokud byl tiket úspěšně upraven
404 NOT FOUND pokud tiket s touto URL neexistuje
412 PRECONDITION FAILED pokud má server novější verzi zdroje, než je upravována
422 UNPROCESSABLE ENTITY pokud hodnota položky closed nebo type neodpovídá aktuální hodnotě, některý z odkazovaných štítků neexistuje, společnost autentizovaného uživatele není nastavena nebo neexistuje

Poznámka: Většina změn v tiketu je prováděna pomocí vytvoření nového záznamu tiketu, nikoli přímou editací položek tiketu. Aktualizace jakékoli položky tiketu, s výjimkou položek version a *seenByMe* zapříčiní nastavení hodnoty *seenByMe* pro všechny ostatní uživatele na hodnotu *false* a aktualizování hodnoty položky *lastUpdateDate* na okamžik provedení požadavku.

B.12.3 PATCH

Edituje nekompletní reprezentaci požadovaného tiketu.

Syntaxe je shodná s metodou PUT (sekce B.12.2, s výjimkou možnosti v požadavku neuvádět needitované položky. Tato výjimka se netýká položky *version*, která musí být vždy přítomna.

B.12.4 DELETE

Odstraní požadovaný tiket.

Obvyklé stavové kódy:
200 OK pokud byl tiket úspěšně odstraněn
404 NOT FOUND pokud tiket s touto URL neexistuje

B.13 /tickets/{id}/ticketEntries

Podporované HTTP metody: GET, POST, HEAD, OPTIONS

B.13.1 GET

Vrátí kolekci všech záznamů daného tiketu.

Obsah odpovědi: stránka (dle sekce B.32), obsahující kolekci zdrojů typu TicketEntry (dle sekce B.14.1) pod názvem *ticketEntries*

B.13.2 POST

Vytvoří nový záznam daného tiketu.

Tělo požadavku:

Název položky	Popis
type	řetězec, reprezentující typ záznamu; akceptované hodnoty jsou *ATTACHMENT_CREATE_ENTRY*, *ATTACHMENT_DELETE_ENTRY*, *STATE_CHANGE_ENTRY*, *COMMENT_ENTRY*, *GROUP_ASSIGNMENT_ENTRY*, *OUTGOING_EMAIL_ENTRY*, *USER_ASSIGNMENT_ENTRY*
attachmentCreateEntry	obsahuje data záznamu o vytvoření přílohy, typu AttachmentCreateEntryContent
attachmentDeleteEntry	obsahuje data záznamu o odstranění přílohy, typu AttachmentDeleteEntryContent
stateChangeEntry	obsahuje data záznamu o změně stavu tiketu, typu StateChangeEntryContent
commentEntry	obsahuje data záznamu, reprezentující poznámku, typu CommentEntryContent
groupAssignmentEntry	obsahuje data záznamu o přiřazení tiketu skupině, typu GroupAssignmentEntryContent
outgoingEmailEntry	obsahuje data záznamu o odchozím e-mailu, typu OutgoingEmailEntryContent
userAssignmentEntry	obsahuje data záznamu o přiřazení tiketu uživateli, typu UserAssignmentEntryContent

Obvyklé stavové kódy:
200 OK pokud byl zdroj úspěšně vytvořen
404 NOT FOUND pokud tiket s danou URL neexistuje
422 UNPROCESSABLE ENTITY pokud typ vytvářeného záznamu není platný, data záznamu nejsou platná či nesplňují specifická omezení daného typu, nebo společnost autentizovaného uživatele není nastavena nebo neexistuje

Poznámka: Záznam typu *INCOMING_EMAIL_ENTRY* není možné vytvořit, vytváří jej výhradně server při stahování nových e-mailů. Detaily o uvedených datových typech a jejich omezení jsou k dispozici v sekci B.31.

B.14 /tickets/{tid}/ticketEntries/{eid}

Podporované HTTP metody: GET, HEAD, OPTIONS

B.14.1 GET

Vrátí požadovaný záznam tiketu.

Obsah odpovědi: zdroj TicketEntry

Název položky	Popis
version	časová značka aktuální verze zdroje, využita pro concurrency control
type	řetězec, reprezentující typ záznamu; akceptované hodnoty jsou *ATTACHMENT_CREATE_ENTRY*, *ATTACHMENT_DELETE_ENTRY*, *STATE_CHANGE_ENTRY*, *COMMENT_ENTRY*, *GROUP_ASSIGNMENT_ENTRY*, *OUTGOING_EMAIL_ENTRY*, *USER_ASSIGNMENT_ENTRY*
attachmentCreateEntry	obsahuje data záznamu o vytvoření přílohy, typu AttachmentCreateEntryContent
attachmentDeleteEntry	obsahuje data záznamu o odstranění přílohy, typu AttachmentDeleteEntryContent
stateChangeEntry	obsahuje data záznamu o změně stavu tiketu, typu StateChangeEntryContent
commentEntry	obsahuje data záznamu, reprezentující poznámku, typu CommentEntryContent
groupAssignmentEntry	obsahuje data záznamu o přiřazení tiketu skupině, typu GroupAssignmentEntryContent
incomingEmailEntry	obsahuje data záznamu o příchozím e-mailu, typu IncomingEmailEntryContent
outgoingEmailEntry	obsahuje data záznamu o odchozím e-mailu, typu OutgoingEmailEntryContent
userAssignmentEntry	obsahuje data záznamu o přiřazení tiketu uživateli, typu UserAssignmentEntryContent
_links	sekce s odkazy pro navigování

_embedded	sekce vnořených zdrojů

Obsah sekce s navigačními odkazy (_links):

Název položky	Popis
self	odkaz na tento zdroj
author	odkaz na uživatele, který záznam vytvořil; může být *null*, pokud byl záznam vytvořen systémem

Obsah sekce vnořených zdrojů (_embedded):

Název položky	Typ proměnné	Popis
author	User	uživatel, který záznam vytvořil
userTo	User	reprezentuje uživatele, kterému byl přiřazen tiket záznamem typu *USER_ASSIGNMENT_ENTRY*
groupTo	Group	reprezentuje skupinu, kterému byl přiřazen tiket záznamem typu *GROUP_ASSIGNMENT_ENTRY*
attachments	kolekce	reprezentuje přílohy náležící k záznamu typu *ATTACHMENT_CREATE_ENTRY*, *ATTACHMENT_DELETE_ENTRY*, *INCOMING_EMAIL_ENTRY*, *OUTGOING_EMAIL_ENTRY*

Poznámka: Volba, zda budou tyto zdroje vnořeny, je na uvážení serveru. Klient nemůže spoléhat na přítomnost těchto vnořených zdrojů. Vnořené zdroje obvykle dále neobsahují další vnořené zdroje.

B.15 /tickets/{tid}/ticketEntries/{eid}/data

Podporované HTTP metody: GET, HEAD, OPTIONS

B.15.1 GET

Vrátí binární reprezentaci požadovaného e-mailu, je-li záznam typu *INCOMING_EMAIL_ENTRY* a binární reprezentace e-mailu je dostupná. Formát odpovědi je *application/octet-stream*.

Obvyklé stavové kódy:
200 OK pokud byla data poskytnuta

404 NOT FOUND pokud záznam s touto URL neexistuje nebo data nejsou dostupná

B.16 /roles

Podporované HTTP metody: GET, POST, HEAD, OPTIONS

B.16.1 GET

Vrátí kolekci všech rolí dané společnosti.

Obsah odpovědi: stránka (dle sekce B.32), obsahující kolekci zdrojů typu Role (dle sekce B.17.1) pod názvem *roles*

B.16.2 POST

Vytvoří novou roli.

Tělo požadavku:

Název položky	Typ proměnné	Popis
name	řetězec	časová značka aktuální verze zdroje, využita pro concurrency control
permissions	kolekce	kolekce odkazů na oprávnění, které tato role obsahuje

Obvyklé stavové kódy:
200 OK pokud byl zdroj úspěšně vytvořen
422 UNPROCESSABLE ENTITY pokud některé z odkazovaných oprávnění neexistuje, společnost autentizovaného uživatele není nastavena nebo neexistuje

B.17 /roles/{id}

Podporované HTTP metody: GET, PUT, PATCH, DELETE, HEAD, OPTIONS

B.17.1 GET

Vrátí požadovanou roli.

Obsah odpovědi: zdroj Role

Název položky	Typ proměnné	Popis
version	časová značka	časová značka aktuální verze zdroje, využita pro concurrency control
name	řetězec	reprezentuje název role
permissions	kolekce	kolekce odkazů na oprávnění, které tato role obsahuje
_links	sekce s odkazy	sekce s odkazy pro navigování
_embedded	sekce se zdroji	sekce vnořených zdrojů

Obsah sekce s navigačními odkazy (_links):

Název položky	Popis
self	odkaz na tento zdroj

Obsah sekce vnořených zdrojů (_embedded):

Název položky	Typ proměnné	Popis
permissions	kolekce	kolekce oprávnění této role

Poznámka: Volba, zda budou tyto zdroje vnořeny, je na uvážení serveru. Klient nemůže spoléhat na přítomnost těchto vnořených zdrojů. Vnořené zdroje obvykle dále neobsahují další vnořené zdroje.

B.17.2 PUT

Edituje kompletní reprezentaci požadované role.

Tělo požadavku:

Název položky	Typ proměnné	Popis
version	časová značka	časová značka aktuální verze zdroje, musí být shodná s obdrženou časovou značkou tohoto zdroje
name	řetězec	reprezentuje název role
permissions	kolekce	kolekce odkazů na oprávnění, které tato role obsahuje

Obvyklé stavové kódy:
200 OK pokud byla role úspěšně upravena
404 NOT FOUND pokud role s touto URL neexistuje
412 PRECONDITION FAILED pokud má server novější verzi zdroje, než je upravována
422 UNPROCESSABLE ENTITY pokud některé z odkazovaných oprávnění

neexistuje, společnost autentizovaného uživatele není nastavena nebo neexistuje

B.17.3 PATCH

Edituje nekompletní reprezentaci požadované role.

Syntaxe je shodná s metodou PUT (sekce B.17.2), s výjimkou možnosti v požadavku neuvádět needitované položky. Tato výjimka se netýká položky *version*, která musí být vždy přítomna.

B.17.4 DELETE

Odstraní požadovanou roli.

Obvyklé stavové kódy:
200 OK pokud byla role úspěšně odstraněna
404 NOT FOUND pokud role s touto URL neexistuje

B.18 /permissions

Podporované HTTP metody: GET, HEAD, OPTIONS

B.18.1 GET

Vrátí kolekci všech oprávnění, která v systému existují.

Obsah odpovědi: stránka (dle sekce B.32), obsahující kolekci zdrojů typu Permission (dle sekce B.19.1) pod názvem *permissions*

B.19 /permissions/{id}

Podporované HTTP metody: GET, HEAD, OPTIONS

B.19.1 GET

Vrátí požadované oprávnění.

Obsah odpovědi: zdroj Permission

Název položky	Typ proměnné	Popis
version	časová značka	časová značka aktuální verze zdroje, využita pro concurrency control

identifier	řetězec	textový identifikátor oprávnění
_links	sekce s odkazy	sekce s odkazy pro navigování

Obsah sekce s navigačními odkazy (_links):

Název položky	Popis
self	odkaz na tento zdroj

B.20 /attachments

Podporované HTTP metody: GET, POST, HEAD, OPTIONS

B.20.1 GET

Vrátí kolekci všech příloh dané společnosti.

Obsah odpovědi: stránka (dle sekce B.32), obsahující kolekci zdrojů typu Attachment (dle sekce B.21.1) pod názvem *attachments*

B.20.2 POST

Vytvoří novou přílohu.

Tělo požadavku:

Název položky	Typ proměnné	Popis
deleted	logická hodnota	eprezentuje, zda byl tento zdroj smazán; musí být *false*
name	řetězec	reprezentuje název přílohy včetně přípony

Obvyklé stavové kódy:
200 OK pokud byl zdroj úspěšně vytvořen
422 UNPROCESSABLE ENTITY pokud je položka *deleted* nastavena na *true* nebo společnost autentizovaného uživatele není nastavena nebo neexistuje

B.21 /attachments/{id}

Podporované HTTP metody: GET, PUT, PATCH, HEAD, OPTIONS

Poznámka: Mazání přílohy se provádí pomocí vytvoření záznamu tiketu reprezentující smazání přílohy, metoda DELETE není podporovaná.

B.21.1 GET

Vrátí požadovanou přílohu.

Obsah odpovědi: zdroj Attachment

Název položky	Typ proměnné	Popis
version	časová značka	časová značka aktuální verze zdroje, využita pro concurrency control
deleted	logická hodnota	reprezentuje, zda byl tento zdroj smazán
name	řetězec	reprezentuje název přílohy včetně přípony
_links	sekce s odkazy	sekce s odkazy pro navigování

Obsah sekce s navigačními odkazy (_links):

Název položky	Popis
self	odkaz na tento zdroj
data	odkaz na binární data reprezentující obsah této přílohy

B.21.2 PUT

Edituje kompletní reprezentaci požadované přílohy.

Tělo požadavku:

Název položky	Typ proměnné	Popis
version	časová značka	časová značka aktuální verze zdroje, musí být shodná s obdrženou časovou značkou tohoto zdroje
deleted	logická hodnota	reprezentuje, zda byl tento zdroj smazán, hodnota musí být shodná s obdrženou hodnotou tohoto zdroje
name	řetězec	reprezentuje název přílohy včetně přípony

Obvyklé stavové kódy:
200 OK pokud byla příloha úspěšně upravena
404 NOT FOUND pokud příloha s touto URL neexistuje
412 PRECONDITION FAILED pokud má server novější verzi zdroje, než je upravována
422 UNPROCESSABLE ENTITY pokud je hodnota položky *deleted true* anebo neodpovídá aktuální hodnotě nebo společnost autentizovaného uživa-

tele není nastavena nebo neexistuje

Poznámka: Nelze editovat smazanou (*deleted*) přílohu.

B.21.3 PATCH

Edituje nekompletní reprezentaci požadované přílohy.

Syntaxe je shodná s metodou PUT (sekce B.21.2), s výjimkou možnosti v požadavku neuvádět needitované položky. Tato výjimka se netýká položky *version*, která musí být vždy přítomna.

B.22 /attachments/{id}/data

Podporované HTTP metody: GET, POST, HEAD, OPTIONS

B.22.1 GET

Vrátí binární data požadované přílohy. Formát odpovědi je typu *application/octet-stream*.

B.22.2 POST

Nahraje binární data přílohy. Formát požadavku je typu *application/octet-stream*.

Obvyklé stavové kódy:
200 OK pokud byla data úspěšně nahrána
404 NOT FOUND pokud příloha s touto URL neexistuje
422 UNPROCESSABLE ENTITY pokud příloha již nějaká data má

Poznámka: Nelze přepsat data již nahrané přílohy.

B.23 /emailAccounts

Podporované HTTP metody: GET, POST, HEAD, OPTIONS

B.23.1 GET

Vrátí kolekci všech e-mailových účtů dané společnosti.

Obsah odpovědi: stránka (dle sekce B.32), obsahující kolekci zdrojů typu EmailAccount (dle sekce B.24.1) pod názvem *emailAccounts*

B.23.2 POST

Vytvoří nový e-mailový účet.

Tělo požadavku:

Název položky	Typ proměnné	Popis
email	řetězec	e-mailová adresa schránky; hodnota musí být přítomna, nesmí již existovat a nesmí být prázdný řetězec
incomingProtocol	řetězec	reprezentuje protokol pro stahování příchozí pošty; akceptované hodnoty jsou *POP3* a *IMAP*
username	řetězec	uživatelské jméno pro přihlášení k serveru příchozí pošty
password	řetězec	heslo k přihlášení k serveru příchozí pošty
smtpUsername	řetězec	uživatelské jméno pro přihlášení k serveru odchozí pošty
smtpPassword	řetězec	heslo k přihlášení k serveru odchozí pošty
incomingServer	řetězec	adresa serveru příchozí pošty, bez specifikace protokolu a portu
outgoingServer	řetězec	adresa serveru odchozí pošty, bez specifikace protokolu a portu
incomingSecured	logická hodnota	reprezentuje, zda má být použito zabezpečené spojení při přístupu k serveru příchozí pošty
outgoingSecured	logická hodnota	reprezentuje, zda má být použito zabezpečené spojení při přístupu k serveru odchozí pošty
incomingPort	celočíselná hodnota	port serveru příchozí pošty; musí být v rozsahu 1 - 65535
outgoingPort	celočíselná hodnota	port serveru odchozí pošty; musí být v rozsahu 1 - 65535
displayedName	řetězec	jméno odesílatele, zobrazované spolu s e-mailovou adresou při odesílání e-mailu v hlavičce From; může být *null*
enabled	logická hodnota	reprezentuje, zda tato e-mailová schránka má být kontrolována na nové zprávy

checkInterval	celočíselná hodnota	počet sekund mezi jednotlivými kontrolami nových e-mailů (není-li použit IMAP *IDLE* příkaz); server může tuto hodnotu ignorovat či nastavit jinou dle svého uvážení
deleteMails	celočíselná hodnota	reprezentuje, zda má být e-mail po jeho stažení do systému smazán z e-mailové schránky

Obvyklé stavové kódy:
200 OK pokud byl zdroj úspěšně vytvořen
422 UNPROCESSABLE ENTITY pokud hodnota položky *incomingProtocol* není platná, hodnota položky *email* není platná nebo již účet s touto e-mailovou adresou existuje, hodnota položek *incomingPort* nebo *outgoingPort* není platná nebo společnost autentizovaného uživatele není nastavena nebo neexistuje

B.24 /emailAccounts/{id}

Podporované HTTP metody: GET, PUT, PATCH, HEAD, OPTIONS

B.24.1 GET

Vrátí požadovaný e-mailový účet.

Obsah odpovědi: zdroj EmailAccount

Název položky	Typ proměnné	Popis
version	časová značka	časová značka aktuální verze zdroje, využita pro concurrency control
email	řetězec	e-mailová adresa schránky
incomingProtocol	řetězec	reprezentuje protokol pro stahování příchozí pošty; akceptované hodnoty jsou *POP3* a *IMAP*
username	řetězec	uživatelské jméno pro přihlášení k serveru příchozí pošty
password	řetězec	vrací vždy prázdný řetězec, heslo příchozí pošty není navráceno
smtpUsername	řetězec	uživatelské jméno pro přihlášení k serveru odchozí pošty
smtpPassword	řetězec	vrací vždy prázdný řetězec, heslo odchozí pošty není navráceno

incomingServer	řetězec	adresa serveru příchozí pošty, bez specifikace protokolu a portu
outgoingServer	řetězec	adresa serveru odchozí pošty, bez specifikace protokolu a portu
incomingSecured	logická hodnota	reprezentuje, zda má být použito zabezpečené spojení při přístupu k serveru příchozí pošty
outgoingSecured	logická hodnota	reprezentuje, zda má být použito zabezpečené spojení při přístupu k serveru odchozí pošty
incomingPort	celočíselná hodnota	port serveru příchozí pošty; musí být v rozsahu 1 - 65535
outgoingPort	celočíselná hodnota	port serveru odchozí pošty; musí být v rozsahu 1 - 65535
displayedName	řetězec	jméno odesílatele, zobrazované spolu s e-mailovou adresou při odesílání e-mailu v hlavičce From
enabled	logická hodnota	reprezentuje, zda tato e-mailová schránka má být kontrolována na nové zprávy
checkInterval	celočíselná hodnota	počet sekund mezi jednotlivými kontrolami nových e-mailů (není-li použit IMAP *IDLE* příkaz); nemusí se jednat o hodnotu, kterou server skutečně využívá
deleteMails	celočíselná hodnota	reprezentuje, zda má být e-mail po jeho stažení do systému smazán z e-mailové schránky
_links	sekce s odkazy	sekce s odkazy pro navigování

Obsah sekce s navigačními odkazy (_links):

Název položky	Popis
self	odkaz na tento zdroj

B.24.2 PUT

Edituje kompletní reprezentaci požadovaného e-mailového účtu.

Tělo požadavku:

Název položky	Typ proměnné	Popis

version	časová značka	časová značka aktuální verze zdroje, musí být shodná s obdrženou časovou značkou tohoto zdroje
email	řetězec	e-mailová adresa schránky; hodnota musí být přítomna, nesmí již existovat a nesmí být prázdný řetězec
incomingProtocol	řetězec	reprezentuje protokol pro stahování příchozí pošty; akceptované hodnoty jsou *POP3* a *IMAP*
username	řetězec	uživatelské jméno pro přihlášení k serveru příchozí pošty
password	řetězec	heslo k přihlášení k serveru příchozí pošty
smtpUsername	řetězec	uživatelské jméno pro přihlášení k serveru odchozí pošty
smtpPassword	řetězec	heslo k přihlášení k serveru odchozí pošty
incomingServer	řetězec	adresa serveru příchozí pošty, bez specifikace protokolu a portu
outgoingServer	řetězec	adresa serveru odchozí pošty, bez specifikace protokolu a portu
incomingSecured	logická hodnota	reprezentuje, zda má být použito zabezpečené spojení při přístupu k serveru příchozí pošty
outgoingSecured	logická hodnota	reprezentuje, zda má být použito zabezpečené spojení při přístupu k serveru odchozí pošty
incomingPort	celočíselná hodnota	port serveru příchozí pošty; musí být v rozsahu 1 - 65535
outgoingPort	celočíselná hodnota	port serveru odchozí pošty; musí být v rozsahu 1 - 65535
displayedName	řetězec	jméno odesílatele, zobrazované spolu s e-mailovou adresou při odesílání e-mailu v hlavičce From; může být *null*
enabled	logická hodnota	reprezentuje, zda tato e-mailová schránka má být kontrolována na nové zprávy
checkInterval	celočíselná hodnota	počet sekund mezi jednotlivými kontrolami nových e-mailů (není-li použit IMAP *IDLE* příkaz); server může tuto hodnotu ignorovat či nastavit jinou dle svého uvážení

deleteMails	celočíselná hodnota	reprezentuje, zda má být e-mail po jeho stažení do systému smazán z e-mailové schránky

Obvyklé stavové kódy:
200 OK pokud byl e-mailový účet úspěšně upraven
404 NOT FOUND pokud mailový účet s touto URL neexistuje
412 PRECONDITION FAILED pokud má server novější verzi zdroje, než je upravována
422 UNPROCESSABLE ENTITY pokud hodnota položky *incomingProtocol* není platná, hodnota položky *email* není platná nebo již účet s touto e-mailovou adresou existuje, hodnota položek *incomingPort* nebo *outgoingPort* není platná nebo společnost autentizovaného uživatele není nastavena nebo neexistuje

Poznámka: Hodnoty položek *password* a *smtpPassword* v odpovědi jsou vždy prázdným řetězcem, hesla ke schránce nejsou navrácena. Nejsou-li při editaci e-mailového účtu nastavována hesla, je nutné editovat účet metodou PATCH, z důvodu nemožnosti zjistit (a tedy poskytnout při volání metody PUT) aktuální hesla.

B.24.3 PATCH

Edituje nekompletní reprezentaci požadovaného e-mailového účtu.

Syntaxe je shodná s metodou PUT sekce B.24.2), s výjimkou možnosti v požadavku neuvádět needitované položky. Tato výjimka se netýká položky *version*, která musí být vždy přítomna.

B.24.4 DELETE

Odstraní požadovaný e-mailový účet.

Obvyklé stavové kódy:
200 OK pokud byl e-mailový účet úspěšně odstraněn
404 NOT FOUND pokud e-mailový účet s touto URL neexistuje

B.25 /tags

Podporované HTTP metody: GET, POST, HEAD, OPTIONS

B.25.1 GET

Vrátí kolekci všech štítků dané společnosti.

Obsah odpovědi: stránka (dle sekce B.32), obsahující kolekci zdrojů typu Tag (dle sekce B.26.1) pod názvem *tags*

B.25.2 POST

Vytvoří nový štítek.

Tělo požadavku:

Název položky	Typ proměnné	Popis
name	řetězec	reprezentuje název štítku

Obvyklé stavové kódy:
200 OK pokud byl zdroj úspěšně vytvořen
422 UNPROCESSABLE ENTITY pokud společnost autentizovaného uživatele není nastavena nebo neexistuje

B.26 /tags/{id}

Podporované HTTP metody: GET, PUT, PATCH, DELETE, HEAD, OPTIONS

B.26.1 GET

Vrátí požadovaný štítek.

Obsah odpovědi: zdroj Tag

Název položky	Typ proměnné	Popis
version	časová značka	časová značka aktuální verze zdroje, využita pro concurrency control
name	řetězec	reprezentuje název štítku
_links	sekce s odkazy	sekce s odkazy pro navigování

Obsah sekce s navigačními odkazy (_links):

Název položky	Popis
self	odkaz na tento zdroj
tickets	odkaz na tikety, které jsou označeny tímto štítkem

B.26.2 PUT

Edituje kompletní reprezentaci požadovaného štítku.

Tělo požadavku:

Název položky	Typ proměnné	Popis
version	časová značka	časová značka aktuální verze zdroje, využita pro concurrency control
name	řetězec	reprezentuje název štítku

Obvyklé stavové kódy:
200 OK pokud byl štítek úspěšně upraven
404 NOT FOUND pokud štítek s touto URL neexistuje
412 PRECONDITION FAILED pokud má server novější verzi zdroje, než je upravována
422 UNPROCESSABLE ENTITY pokud společnost autentizovaného uživatele není nastavena nebo neexistuje

B.26.3 PATCH

Edituje nekompletní reprezentaci požadovaného štítku.

Syntaxe je shodná s metodou PUT (sekce B.26.2), s výjimkou možnosti v požadavku neuvádět needitované položky. Tato výjimka se netýká položky *version*, která musí být vždy přítomna.

B.26.4 DELETE

Odstraní požadovaný štítek.

Obvyklé stavové kódy:
200 OK pokud byl štítek úspěšně odstraněn
404 NOT FOUND pokud štítek s touto URL neexistuje

B.27 /textTemplates

Podporované HTTP metody: GET, POST, HEAD, OPTIONS

B.27.1 GET

Vrátí kolekci všech šablon textů dané společnosti.

Obsah odpovědi: stránka (dle sekce B.32), obsahující kolekci zdrojů typu TextTemplate (dle sekce B.28.1) pod názvem *textTemplates*

B.27.2 POST

Vytvoří novou šablonu textu.

Tělo požadavku:

name	řetězec	reprezentuje název šablony
text	řetězec	reprezentuje text šablony
user	odkaz	reprezentuje odkaz na uživatele, kterému šablona textu náleží; může být *null*
group	odkaz	reprezentuje odkaz na skupinu, které šablona textu náleží; může být *null*

Obvyklé stavové kódy:
200 OK pokud byl zdroj úspěšně vytvořen
422 UNPROCESSABLE ENTITY pokud odkazovaný uživatel či skupina neexistuje nebo je smazána (*deleted*), nebo společnost autentizovaného uživatele není nastavena nebo neexistuje

B.28 /textTemplates/{id}

Podporované HTTP metody: GET, PUT, PATCH, DELETE, HEAD, OPTIONS

B.28.1 GET

Vrátí požadovanou šablonu textu.

Obsah odpovědi: zdroj TextTemplate

Název položky	Typ proměnné	Popis
version	časová značka	časová značka aktuální verze zdroje, využita pro concurrency control
name	řetězec	reprezentuje název šablony
text	řetězec	reprezentuje text šablony
user	odkaz	reprezentuje odkaz na uživatele, kterému šablona textu náleží; může být *null*
group	odkaz	reprezentuje odkaz na skupinu, které šablona textu náleží; může být *null*
_links	sekce s odkazy	sekce s odkazy pro navigování

Obsah sekce s navigačními odkazy (_links):

Název položky	Popis
self	odkaz na tento zdroj

B.28.2 PUT

Edituje kompletní reprezentaci požadované šablony.

Tělo požadavku:

Název položky	Typ proměnné	Popis
version	časová značka	časová značka aktuální verze zdroje, musí být shodná s obdrženou časovou značkou tohoto zdroje
name	řetězec	reprezentuje název šablony
text	řetězec	reprezentuje text šablony
user	odkaz	reprezentuje odkaz na uživatele, kterému šablona textu náleží; může být *null*
group	odkaz	reprezentuje odkaz na skupinu, které šablona textu náleží; může být *null*

Obvyklé stavové kódy:
200 OK pokud byla šablona úspěšně upravena
404 NOT FOUND pokud šablona s touto URL neexistuje
412 PRECONDITION FAILED pokud má server novější verzi zdroje, než je upravována
422 UNPROCESSABLE ENTITY pokud odkazovaný uživatel či skupina neexistuje nebo je smazána (*deleted*), společnost autentizovaného uživatele není nastavena nebo neexistuje

B.28.3 PATCH

Edituje nekompletní reprezentaci požadované šablony.

Syntaxe je shodná s metodou PUT (sekce B.28.2), s výjimkou možnosti v požadavku neuvádět needitované položky. Tato výjimka se netýká položky *version*, která musí být vždy přítomna.

B.28.4 DELETE

Odstraní požadovanou šablonu.

Obvyklé stavové kódy:
200 OK pokud byla šablona úspěšně odstraněna
404 NOT FOUND pokud šablona s touto URL neexistuje

B.29 /companies

Podporované HTTP metody: POST, HEAD, OPTIONS

B.29.1 POST

Vytvoří novou společnost.

Tělo požadavku:

Název položky	Typ proměnné	Popis
name	řetězec	reprezentuje název společnosti
address	Address	reprezentuje adresu společnosti
identificationNumber	řetězec	reprezentuje identifikační číslo společnosti (IČ), či jeho ekvivalent
taxIdentification-Number	řetězec	reprezentuje daňové identifikační číslo společnosti (DIČ), či jeho ekvivalent
type	řetězec	reprezentuje typ společnosti; akceptované hodnoty jsou *PERSON_NONREGISTERED*, *PERSON_REGISTERED* a *COMPANY*
reopeningAllowed	logická hodnota	hodnota, reprezentující zda je povoleno automatické znovuotevíraní již zavřených tiketů v případě nových příchozích zpráv k danému tématu; je-li nastaveno na hodnotu *false*, je pro příchozí zprávu vytvořen nový tiket

keepGroup-AssignmentOnReopen	logická hodnota	hodnota, reprezentující zda bude ponecháno přiřazení tiketu aktuální skupině při jeho znovuotevření; je-li nastavena hodnota *false*, je tiket přiřazen skupině, jejíž e-mailový účet zprávu přijal
keepUser-AssignmentOnReopen	logická hodnota	hodnota, reprezentující zda bude ponecháno přiřazení tiketu aktuálnímu uživateli při jeho znovuotevření; je-li nastavena hodnota *false*, nebude tiket přiřazen žádnému uživateli

Obvyklé stavové kódy:
200 OK pokud byl zdroj úspěšně vytvořen
409 CONFLICT pokud je současnému uživateli již přiřazena některá společnost
422 UNPROCESSABLE ENTITY pokud společnost s tímto názvem již existuje

Poznámka: Vytvoření společnosti je jednorázový úkon, prováděný při registraci prvního uživatele. Není možné vytvořit společnost, náleží-li již autentizovaný uživatel do některé společnosti. Při vytvoření společnosti dojde k automatickému přiřazení autentizovaného uživatele pod nově vytvořenou společnost.

B.30 /companies/{id}

Podporované HTTP metody: GET, PUT, PATCH, HEAD, OPTIONS

B.30.1 GET

Vrátí požadovanou společnost.

Obsah odpovědi: zdroj Company

Název položky	Typ proměnné	Popis
version	časová značka	časová značka aktuální verze zdroje, využita pro concurrency control
name	řetězec	reprezentuje název společnosti

address	Address	reprezentuje adresu společnosti
identificationNumber	řetězec	reprezentuje identifikační číslo společnosti (IČ), či jeho ekvivalent
taxIdentification-Number	řetězec	reprezentuje daňové identifikační číslo společnosti (DIČ), či jeho ekvivalent
type	řetězec	reprezentuje typ společnosti; akceptované hodnoty jsou *PERSON_NONREGISTERED*, *PERSON_REGISTERED* a *COMPANY*
reopeningAllowed	logická hodnota	hodnota, reprezentující zda je povoleno automatické znovuotevíraní již zavřených tiketů v případě nových příchozích zpráv k danému tématu; je-li nastaveno na hodnotu *false*, je pro příchozí zprávu vytvořen nový tiket
keepGroup-AssignmentOnReopen	logická hodnota	hodnota, reprezentující zda bude ponecháno přiřazení tiketu aktuální skupině při jeho znovuotevření; je-li nastavena hodnota *false*, je tiket přiřazen skupině, jejíž e-mailový účet zprávu přijal
keepUser-AssignmentOnReopen	logická hodnota	hodnota, reprezentující zda bude ponecháno přiřazení tiketu aktuálnímu uživateli při jeho znovuotevření; je-li nastavena hodnota *false*, nebude tiket přiřazen žádnému uživateli
_links	sekce s odkazy	sekce s odkazy pro navigování

Obsah sekce s navigačními odkazy (_links):

Název položky	Popis
self	odkaz na tento zdroj

Poznámka: Oprávnění uživatele neumožňují vyžádat jinou společnost, než je společnost, do které uživatel náleží.

B.30.2 PUT

Edituje kompletní reprezentaci požadované společnosti.

Tělo požadavku:

Název položky	Typ proměnné	Popis
version	časová značka	časová značka aktuální verze zdroje, musí být shodná s obdrženou časovou značkou tohoto zdroje
name	řetězec	reprezentuje název společnosti
address	Address	reprezentuje adresu společnosti
identificationNumber	řetězec	reprezentuje identifikační číslo společnosti (IČ), či jeho ekvivalent
taxIdentification-Number	řetězec	reprezentuje daňové identifikační číslo společnosti (DIČ), či jeho ekvivalent
type	řetězec	reprezentuje typ společnosti; akceptované hodnoty jsou *PERSON_NONREGISTERED*, *PERSON_REGISTERED* a *COMPANY*
reopeningAllowed	logická hodnota	hodnota, reprezentující zda je povoleno automatické znovuotevíraní již zavřených tiketů v případě nových příchozích zpráv k danému tématu; je-li nastaveno na hodnotu *false*, je pro příchozí zprávu vytvořen nový tiket

keepGroup-AssignmentOnReopen	logická hodnota	hodnota, reprezentující zda bude ponecháno přiřazení tiketu aktuální skupině při jeho znovuotevření; je-li nastavena hodnota *false*, je tiket přiřazen skupině, jejíž e-mailový účet zprávu přijal
keepUser-AssignmentOnReopen	logická hodnota	hodnota, reprezentující zda bude ponecháno přiřazení tiketu aktuálnímu uživateli při jeho znovuotevření; je-li nastavena hodnota *false*, nebude tiket přiřazen žádnému uživateli

Obvyklé stavové kódy:
200 OK pokud byla společnost úspěšně upravena
404 NOT FOUND pokud společnost s touto URL neexistuje
412 PRECONDITION FAILED pokud má server novější verzi zdroje, než je upravována
422 UNPROCESSABLE ENTITY pokud jiná společnost s tímto názvem již existuje

Poznámka: Oprávnění uživatele neumožňují editovat jinou společnost, než je společnost, do které uživatel náleží.

B.30.3 PATCH

Edituje nekompletní reprezentaci požadované společnosti.

Syntaxe je shodná s metodou PUT (sekce B.30.2), s výjimkou možnosti v požadavku neuvádět needitované položky. Tato výjimka se netýká položky *version*, která musí být vždy přítomna.

B.31 Definice datových typů

B.31.1 Address

Typ reprezentující fyzickou poštovní adresu subjektu.

Název položky	Typ proměnné	Popis
street	řetězec	reprezentuje ulici včetně případné specifikace budovy

city	řetězec	reprezentuje město
zipCode	řetězec	reprezentuje poštovní identifikátor lokality
state	řetězec	reprezentuje stát, může být nevyplněn v případě zemí, které nejsou členěny na státy
country	řetězec	reprezentuje zemi

B.31.2 AttachmentCreateEntryContent

Typ obsahující data záznamu tiketu typu *ATTACHMENT_CREATE_ENTRY*, reprezentující přidání přílohy.

Název položky	Typ proměnné	Popis
attachments	kolekce	kolekce odkazů na přílohy, které byly tímto záznamem přidány
message	řetězec	doplňkový komentář k tomuto záznamu

Omezení při vytváření nebo editaci: Všechny odkazy v kolekci odkazů na přílohy musí být platné a musí odpovídat existujícím přílohám. Žádná příloha nesmí mít nastavenu položku *deleted* na hodnotu *true*.

B.31.3 AttachmentDeleteEntryContent

Typ obsahující data záznamu tiketu typu *ATTACHMENT_DELETE_ENTRY*, reprezentující smazání přílohy.

Název položky	Typ proměnné	Popis
attachments	kolekce	kolekce odkazů na přílohy, které byly tímto záznamem smazány
message	řetězec	doplňkový komentář k tomuto záznamu

Omezení při vytváření nebo editaci: Všechny odkazy v kolekci odkazů na přílohy musí být platné a musí odpovídat existujícím přílohám. Žádná příloha nesmí mít již nastavenu položku *deleted* na hodnotu *true*.

B.31.4 StateChangeEntryContent

Typ obsahující data záznamu tiketu typu *STAGE_CHANGE_ENTRY*, reprezentující změnu stavu (otevření či uzavření) tiketu.

Název položky	Typ proměnné	Popis

closed	logická hodnota	reprezentuje, zda byl stav tiketu změněn tímto záznamem na uzavřený či otevřený
message	řetězec	doplňkový komentář k tomuto záznamu

Omezení při vytváření nebo editaci: Tiket se nesmí již nacházet ve stavu, do kterého je požadována změna.

B.31.5 CommentEntryContent

Typ obsahující data záznamu tiketu typu *COMMENT_ENTRY*, reprezentující přidání poznnámky k tiketu.

Název položky	Typ proměnné	Popis
message	řetězec	text poznámky

B.31.6 GroupAssignmentEntryContent

Typ obsahující data záznamu tiketu typu *GROUP_ASSIGNMENT_ENTRY*, reprezentující přiřazení tiketu skupině.

Název položky	Typ proměnné	Popis
groupTo	odkaz	odkaz na skupinu, které byl tiket tímto záznamem přiřazen; nesmí být *null*
message	řetězec	doplňkový komentář k tomuto záznamu

Omezení při vytváření nebo editaci: Odkaz na skupinu musí být platný a musí odkazovat na existující skupinu. Skupina nesmí mít nastavenu položku *deleted* na hodnotu *true*.

B.31.7 IncomingEmailEntryContent

Typ obsahující data záznamu tiketu typu *INCOMING_EMAIL_ENTRY*, reprezentující příchozí e-mailovou zprávu.

Název položky	Typ proměnné	Popis
subject	řetězec	předmět e-mailu
from	řetězec	internetová adresa odesílatele e-mailu ve formátu dle standardu RFC822
textContent	řetězec	text e-mailu ve formátu *text/plain*; může být *null*, je-li použit pouze HTML formát e-mailu

htmlContent	řetězec	text e-mailu ve formátu *text/html*; může být *null*, je-li použit pouze textový formát e-mailu
to	řetězec	internetové adresy příjemců e-mailu ve formátu dle standardu RFC822, oddělené čárkou
cc	řetězec	internetové adresy příjemců kopie e-mailu ve formátu dle standardu RFC822, oddělené čárkou
headers	řetězec	kompletní hlavičky e-mailu; znak(y) pro odřádkování může být různý dle platformy
receivedDate	časová značka	časová značka okamžiku přijetí e-mailu
replyTo	řetězec	internetová adresa ve formátu dle standardu RFC822, reprezentující adresu, na kterou má příjemce e-mailu odpovídat
encoding	řetězec	reprezentuje kódování e-mailu
attachments	kolekce	kolekce odkazů na přílohy, které jsou součástí přijatého e-mailu
data	odkaz	odkaz na binární originál přijatého e-mailu; binární originál nemusí být dostupný

Poznámka: Typická hodnota e-mailové adresy je ve formátu *"user@host.domain"* nebo *"Personal Name <user@host.domain>"*.

B.31.8 OutgoingEmailEntryContent

Typ obsahující data záznamu tiketu typu *OUTGOING_EMAIL_ENTRY,* reprezentující odchozí e-mailovou zprávu.

Název položky	Typ proměnné	Popis
subject	řetězec	předmět e-mailu
from	řetězec	internetová adresa odesílatele e-mailu ve formátu dle standardu RFC822
textContent	řetězec	řetězec text e-mailu ve formátu *text/plain*; může být *null*, je-li použit pouze HTML formát e-mailu; text musí být kódován ve formátu UTF-8

htmlContent	řetězec	řetězec text e-mailu ve formátu *text/html*; může být *null*, je-li použit pouze textový formát e-mailu; text musí být kódován ve formátu UTF-8
to	řetězec	internetové adresy příjemců e-mailu ve formátu dle standardu RFC822, oddělené čárkou
cc	řetězec	internetové adresy příjemců kopie e-mailu ve formátu dle standardu RFC822, oddělené čárkou
bcc	řetězec	internetové adresy příjemců slepé kopie e-mailu ve formátu dle standardu RFC822, oddělené čárkou
status	řetězec	serverem nastavená zpráva reprezentující stav odeslání e-mailu; klientem nastavená hodnota je ignorována
headers	řetězec	serverem nastavené kompletní hlavičky e-mailu; znak(y) pro odřádkování může být různý dle platformy; klientem nastavená hodnota je ignorována
attachments	kolekce	kolekce odkazů na přílohy, které jsou součástí odeslaného e-mailu
data	odkaz	odkaz na binární originál odeslaného e-mailu; binární originál nemusí být dostupný; klientem nastavená hodnota je ignorována

Poznámka: Typická hodnota e-mailové adresy je ve formátu *"user@host.domain"* nebo *"Personal Name <user@host.domain>"*. Požadavek může obsahovat neprázdný *textContent*, *htmlContent* či obojí, v takovém případě server zašle e-mail v MIME formátu *multipart/alternative* s vyšší prioritou obsahu ve formátu HTML. Pokud je neprázdná pouze jedna z těchto položek, server zašle e-mail v jí odpovídajícím formátu.

Omezení při vytváření nebo editaci: Všechny odkazy v kolekci odkazů na přílohy musí být platné a musí odpovídat existujícím přílohám. Žádná příloha nesmí mít nastavenu položku *deleted* na hodnotu *true*.

B.31.9 UserAssignmentEntryContent

Typ obsahující data záznamu tiketu typu *ATTACHMENT_CREATE_ENTRY*, reprezentující přiřazení tiketu uživateli.

Název položky	Typ proměnné	Popis
userTo	odkaz	odkaz na uživatele, kterému byl tiket tímto záznamem přiřazen; nesmí být *null*
message	řetězec	doplňkový komentář k tomuto záznamu

Omezení při vytváření nebo editaci: Odkaz na uživatele musí být platný a musí odkazovat na existujícího uživatele. Uživatel nesmí mít nastavenu položku *deleted* na hodnotu *true*.

B.32 Podoba stránky

Každá kolekce, dostupná pod samostatnou URL, využívá schopnosti stránkování. Podoba stránky je následující:

Název položky	Typ proměnné	Popis
_links	sekce s odkazy	sekce s odkazy pro navigování
_embedded	sekce se zdroji	sekce vnořených zdrojů
page	stránka	informace o aktuálně zobrazené stránce

Obsah sekce s navigačními odkazy (_links):

Název položky	Popis
self	odkaz na tuto stránku
first	odkaz na první stránku; počet prvků na stránku je zachován
prev	odkaz na předchozí stránku; počet prvků na stránku je zachován
next	odkaz na následující stránku; počet prvků na stránku je zachován
last	odkaz na poslední stránku; počet prvků na stránku je zachován

Obsah sekce vnořených odkazů je specifický pro zobrazovanou kolekci zdrojů a obsahuje kolekci zdrojů pod názvem specifikací definovaným pro každý zdroj.

Klient může nastavit aktuální pozici a velikost stránky pomocí následujících proměnných, zakódovaných do URL:

Název položky	Typ proměnné	Popis
page	celočíselná hodnota	číslo požadované stránky; číslo první stránky je 0

size	celočíselná hodnota	počet prvků na stránku; rozsah hodnot je 1 - 2000

Poznámka: Příklad syntaxe adresy je */users?page=1&size=20*

Příloha C

Obsah přiloženého CD

- `readme.txt` ..... popis obsahu CD
- `src`
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF