Univerzita Pardubice
Fakulta elektrotechniky a informatiky

Backend pro firemní systém zpracování chyb

Bakalářská práce

2024 Maxim Joska

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2023/2024

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Maxim Joska**
Osobní číslo:	**I21162**
Studijní program:	**B0688A140009 Informační technologie**
Téma práce:	**Backend pro firemní systém zpracování chyb**
Zadávající katedra:	**Katedra informačních technologií**

Zásady pro vypracování

Cílem práce je vytvořit serverovou část pro systém zpracovávající chybové zprávy. Práce se zabývá současnou problematikou zpracování chyb ve firmě a následně nabízí na základě daných požadavků řešení. Praktická část práce je zpracována v programovacím jazyce Kotlin s pomocí knihovny Ktor.

Rozsah pracovní zprávy:	**min. 30 stran**
Rozsah grafických prací:	
Forma zpracování bakalářské práce:	**tištěná/elektronická**

Seznam doporučené literatury:

ECKEL, Bruce a Svetlana ISAKOVA. Atomic Kotlin. Mindview, 2021. ISBN 978-0981872551.
JEMEROV, Dmitry a Svetlana ISAKOVA. Kotlin in action. Shelter Island: Manning, [2017]. ISBN 978--161-7293-290.

Vedoucí bakalářské práce:	**Ing. Jan Panuš, Ph.D.** Katedra informačních technologií
Datum zadání bakalářské práce:	**15. prosince 2023**
Termín odevzdání bakalářské práce:	**10. května 2024**

L.S.

Ing. Zdeněk Němec, Ph.D. v.r.
děkan

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

V Pardubicích dne 28. února 2024

Prohlašuji:

Práci s názvem Backend pro firemní systém zpracování chyb jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

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

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

V Pardubicích dne 25. 04. 2024

Maxim Joska

Poděkování

Rád bych poděkoval vedoucímu své práce Ing. Janu Panušovi, Ph.D. za jeho ochotu a cenné rady při vedení práce. Také děkuji kolegům z práce za jejich trpělivost a odbornou pomoc ve firemních tématech. Dále děkuji své rodině a všem ostatním, kteří mě při studiu a psaní bakalářské práce podporovali.

Anotace

Cílem práce je vytvořit serverovou část pro systém zpracovávající chybové zprávy. Práce se zabývá současnou problematikou zpracování chyb ve firmě a následně nabízí na základě daných požadavků řešení. Praktická část práce je zpracována v programovacím jazyce Kotlin s pomocí knihovny Ktor.

Klíčová slova

Zpracování chyb, backend, webová aplikace, REST API, Kotlin, Ktor

Title

Backend for company error processing system

Annotation

The goal of the work is to create the server part for an error processing system. The work deals with the current problematics of error handling in the company and consequently offers a solution based on set requirements. The practical part is built using Kotlin programming language and Ktor framework.

Keywords

Error processing, backend, web application, REST API, Kotlin, Ktor

OBSAH

Seznam ilustrací a ukázek kódu 8
Seznam zkratek a značek 9
Úvod 10
1 RESTful API 11
1.1 API 11
1.2 REST 11
1.3 RESTful API 13
2 Použité technologie 15
2.1 Kotlin 15
2.2 Ktor 16
3 Další použité technologie 16
3.1 Vývojové nástroje 16
4 Představení firmy 17
5 Analýza problému 18
5.1 Existující řešení 19
6 Návrh RESTful API 19
7 Schéma databáze 23
8 Ktor server 25
8.1 Gradle 25
8.2 Konfigurace 26
8.3 Dependency Injection 27
8.4 Logování 28
8.5 Routing 29
8.6 Pluginy 29
8.7 Datová vrstva 31
8.8 Autentizace a autorizace 33
8.9 Instalace 35
8.10 Testování 37
Závěr 40
Použitá literatura 41

Seznam ilustrací a ukázek kódu

Obrázek 1: Derivace architektonických stylů pro REST [5] 12
Obrázek 2: Ukázka OpenAPI specifikace (vlevo) v aplikaci Swagger Editor [10] 15
Obrázek 3: Struktura IS VERA (Zdroj: VERA, spol. s r.o.) 17
Obrázek 4: Příklad chybového emailu (Zdroj: VERA, spol. s r.o.) 19
Obrázek 5: Schéma databáze v Enterprise Architect 25
Ukázka kódu 1: Část konfiguračního souboru a jemu odpovídající třída 27
Ukázka kódu 2: Inicialize a použití dependency injection 28
Ukázka kódu 3: Inicializace a použití routing pluginu pro cestu */vpms/v1/info* 29
Ukázka kódu 4: Využití pluginu pro validaci verze API 31
Ukázka kódu 5: Získání poslední chyby z databáze 32
Ukázka kódu 6: Registrace poskytovatele autentizace a jeho použití 34
Ukázka kódu 7: Aktualizace databáze pomocí nástroje Liquibase 36
Ukázka kódu 8: Test repository pro přidání chyby 38
Ukázka kódu 9: Test autentizace pomocí ticketu 39

Seznam zkratek a značek

IS Informační systém

REST Representational State Transfer

API Application Programming Interface

SDK Software development kit

HTTP Hypertext Transfer Protocol

SOAP Simple Object Access Protocol

RPC Remote procedure call

IDE Integrated development environment

DI Dependency injection

CRUD Create, Read, Update, Delete

DAO Data access object

JWT JSON Web Token

JVM Java virtual machine

JDK Java Development Kit

CLI Command line interface

SSL Spisová služba

NSWS Národní Standard Web Services

Úvod

Systém pro zpracování chyb je v moderních softwarových systémech velmi užitečným nástrojem. Poskytuje centralizovaný sběr, sledování a řešení chyb v různých částech aplikace. Jeho funkce zahrnují agregaci chyb a jejich filtraci, oznámení o nových chybách a umožňuje efektivně analyzovat a diagnostikovat chyby a příčinu jejich vzniku. Díky těmto vlastnostem mají vývojáři či analytici možnost rychle reagovat na incidenty plynoucí z chyb aplikací, což přispívá k lepší dostupnosti a spolehlivosti softwarových systémů. Ze strany uživatele tyto výhody znamenají větší spokojenost s aplikací, pro vývojáře dané aplikace spokojení zákazníci znamenají např. větší zisky nebo popularitu.

Cílem bakalářské práce je navrhnout backend pro takový systém, který odpovídá jak výše zmíněným obecným požadavkům, tak i konkrétním požadavkům a omezením ze strany firmy. Práce se zabývá analýzou firemního prostředí, výběrem a představením použité technologie a následně jejím využitím pro implementaci aplikace.

1 RESTful API

Práce se zabývá pouze serverovou částí aplikace, která musí s částí klientskou komunikovat a přenášet data. K tomuto slouží právě API, v případě této aplikace konkrétně RESTful API, jejichž význam a fungování přiblíží následující kapitoly.

1.1 API

Aplikační programové rozhraní, zkráceně API z anglického Application Programming Interface, je soubor definicí, pravidel či protokolů, které umožňují komunikaci mezi dvěma různými aplikacemi. [1] Poskytuje standardizovaný způsob, jak aplikace přistupují k funkcím, datům nebo službám poskytovaným jinými aplikacemi. To umožňuje aplikacím bezpečně a jednoduše poskytovat jejich funkcionalitu třetím stranám, jako jsou vývojáři či klienti. Mezi hlavní přínosy API patří, že je pro uživatele „black box", tedy se uživatel nemusí starat o implementaci, zajímají ho pouze vstupy a výstupy, které API poskytuje. API se převážně dělí na 2 druhy: systémové a webové. [2]

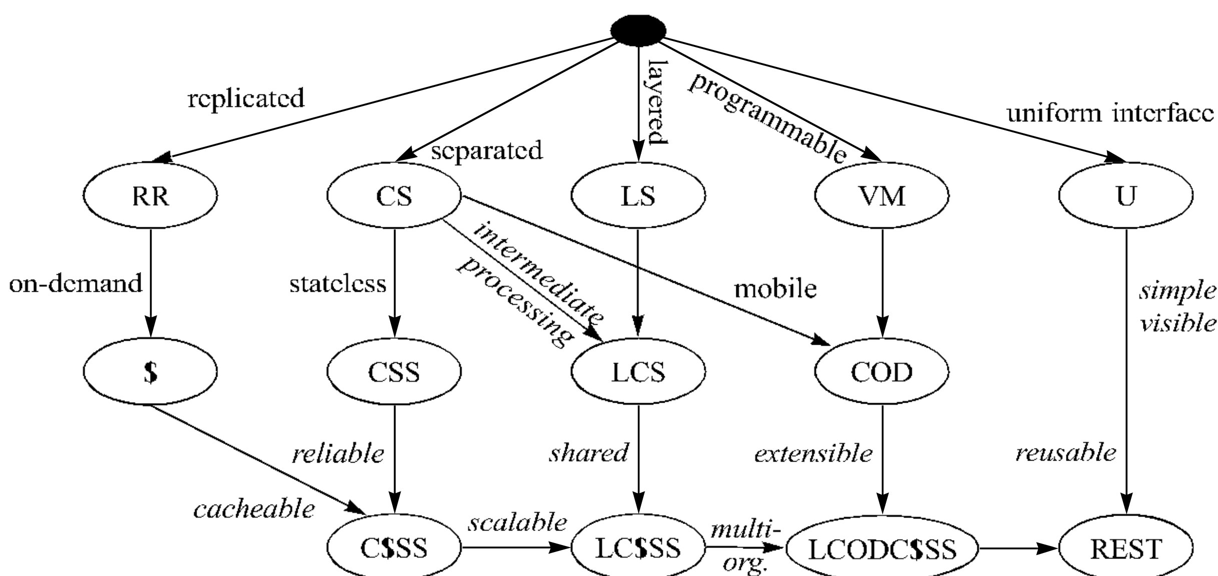
Systémové API jsou vlastně původním využitím API prvně zmíněném roku 1968 ve článku „Data structures and techniques for remote computer graphics" autorů Ira W. Cotton a Frank S. Greatorex. Jedná se o API komunikující ve stejném lokálním systému, jako jsou knihovny nebo SDK. V případě knihoven unixových systémů se podoba API definuje např. v hlavičkových souborech programovacího jazyka C, v objektověorientovaných jazycích se může jednat třeba o tzv. „interface" obsahující signaturu a kontrakt.

Webové API, často také označované jako webová služba, umožňuje komunikaci aplikací skrze internet. Většinou toho dosahuje za pomoci webového standardu HTTP. Dle dostupnosti se dají rozdělit do několika kategorií. Veřejné API je volně dostupné k použití všem uživatelům internetu. Partnerské API je dostupné pouze autorizovaným uživatelům. Webová API často bývají kombinací obou zmiňovaných typů. Mohou poskytovat částečnou funkcionalitu či omezený počet dotazů ve veřejné části, a tím získat potenciální zákazníky, kteří se pro plné využití následně zaregistrují a za poskytované služby zaplatí. Poslední kategorií je vnitřní API, které je dostupné pouze v rámci firmy či určité skupiny. [1]

1.2 REST

REST, z anglického Representational state transfer, je styl softwarové architektury určený pro distribuované hypermédiové systémy. Poprvé tento termín použil a definoval Roy Fielding v roce 2000 ve své doktorské práci „Architectural Styles and the Design of Networkbased Software Architectures". Má za cíl definovat omezující pravidla pro lepší vývoj a chování

moderních webových systémů, z většiny kombinací již existujících architektonických stylů (Obrázek 1). [3] Skládá se z následujících 6 omezení.



Obrázek 1: Derivace architektonických stylů pro REST [5]

Architektura klientserver

Cílem tohoto omezení je rozdělení odpovědností jednotlivých komponentů. Rozdělením uživatelského rozhraní (klienta) od skladování dat (serveru) se dosáhne lepší platformní přenositelnosti uživatelského rozhraní a zlepší se škálovatelnost zjednodušením serverových komponent. Velmi důležitý důsledek je také, že rozdělení umožňuje na sobě nezávislý vývoj klienta a serveru, dokud rozhraní mezi nimi zůstává stejné.

Bezstavovost

Komunikace mezi klientem a serverem musí být bezstavová. Veškeré dotazy z klienta na server musí obsahovat všechny informace potřebné k porozumění dotazu. Server neudržuje žádné stavy relace, ty musí být uchovány výhradně na straně klienta. Motivací je zlepšení transparentnosti, spolehlivosti a škálovatelnosti. Nevýhodou je potenciální snížení výkonu, protože se na server musí opakovaně zasílat stejná data.

Využití mezipaměti

Vyžaduje, aby data v odpovědi na dotaz byla implicitně či explicitně označena jako uložitelná nebo neuložitelná do mezipaměti. Pokud byla data označena za uložitelná, může je klient znovupoužít pro další ekvivalentí dotazy. Mezipamět zlepšuje výkonnost a efektivitu částečným či úplným přeskočením některých kroků při dotazu. Může však nastat situace, kdy

využití mezipaměti způsobí značnou zastaralost dotazovaných dat oproti těm, které jsou na serveru.

Jednotné rozhraní

Skládá se z několika dílčích omezení. REST je zdrojově orientován. V dotazech jsou jednotlivé zdroje jasně identifikovány pomocí identifikátorů zdrojů. Zdrojem může být jakákoli informace, například textový dokument nebo obrázek. Zdrojem se rozumí koncept entity, ne její konkrétní reprezentace v nějakém bodě v čase. Například tedy server nezasílá část databáze, ale reprezentaci řádků ve formátu JSON. Pokud má klient reprezentaci zdroje, může zdroj na serveru modifikovat. Každý dotaz má dostatek informací na to, jak má být zpracován. Využívá principu Hypermedia as the Engine of Application State (HATEOAS). To znamená, že klient poskytuje stav pomocí těla, parametrů a hlaviček dotazu a požadovaného indentifikátoru zdroje, server poskytuje stav pomocí těla, kódu a hlaviček odpovědi.

Vrstvený systém

Systém je sestaven z hierarchických vrstev tak, aby každá vrstva viděla pouze na další nejbližší vrstvu, se kterou komunikuje. Architektura umožňuje využití load balancingu a jednoduché vyloučení a nahrazení zastaralých komponent. Tímto způsobem se sníží celková komplexnost systému a podpoří se nezávislost jednotlivých komponent. Vrstvení způsobuje zhoršení výkonu a odezvy z důvodu většího množství mezikroků, tomu se však dá částečně zabránit využitím sdílené mezipaměti, které vrstvený systém umožňuje.

Kód na požádání

Umožňuje rozšířit klientskou funkcionalitu stažením a spuštěním kódu ve formě skriptů či appletů. Na jednu stranu toto omezení zvyšuje rozšiřitelnost systému, ale zároveň snižuje jeho transparentnost, proto je jako jediné z omezení volitelné. [3]

1.3 RESTful API

Aplikováním architektonického stylu REST na webové API vzniká RESTful API, někdy je také zkráceně označováno jako REST API. Protože REST je zamýšlen pro webové systémy, předpokládá se, že RESTful API bude vždy webové a ne systémové. Zároveň se také předpokládá, že RESTful API funguje na protokolu HTTP. Samotný REST je sice na protokolu nezávislý, ale byl zamýšlen v úzkém spojení s protokolem HTTP a v současné době je tak výhradně používán. Pro identifikaci zdrojů se využívají URI, respektive URL či cesty. Je zdaleka nejpoužívanější architekturou pro vývoj webových API. [5]

Velká část API označovaných jako RESTful však plně nedodržuje všechna omezení REST. Richardson Maturity Model, dále jen RMM, klasifikuje webová API do 4 úrovní dle jejich

dodržování omezení REST. Úroveň 0 jsou webová API stylu XMLRPC či SOAP, kde je dotazována pouze 1 URI, typicky pouze metodou POST. Úroveň 1 zavádí různá URI pro různé zdroje, stále se však dotazuje pouze jednou HTTP metodou. Úroveň 2 umožňuje dotazovat různá URI více HTTP metodami. Úroveň 3 rozšiřuje předešlou úroveň o HATEOAS, tedy odpovědi na dotazy obsahující hypermédiové odkazy. [6] Většina RESTful API se v RMM nachází na úrovni 2, porušují tedy omezení jednotného rozhraní REST. I přes to je však možné a, velmi výhodné, využít ostatní omezení stylu pro lepší vývoj API.

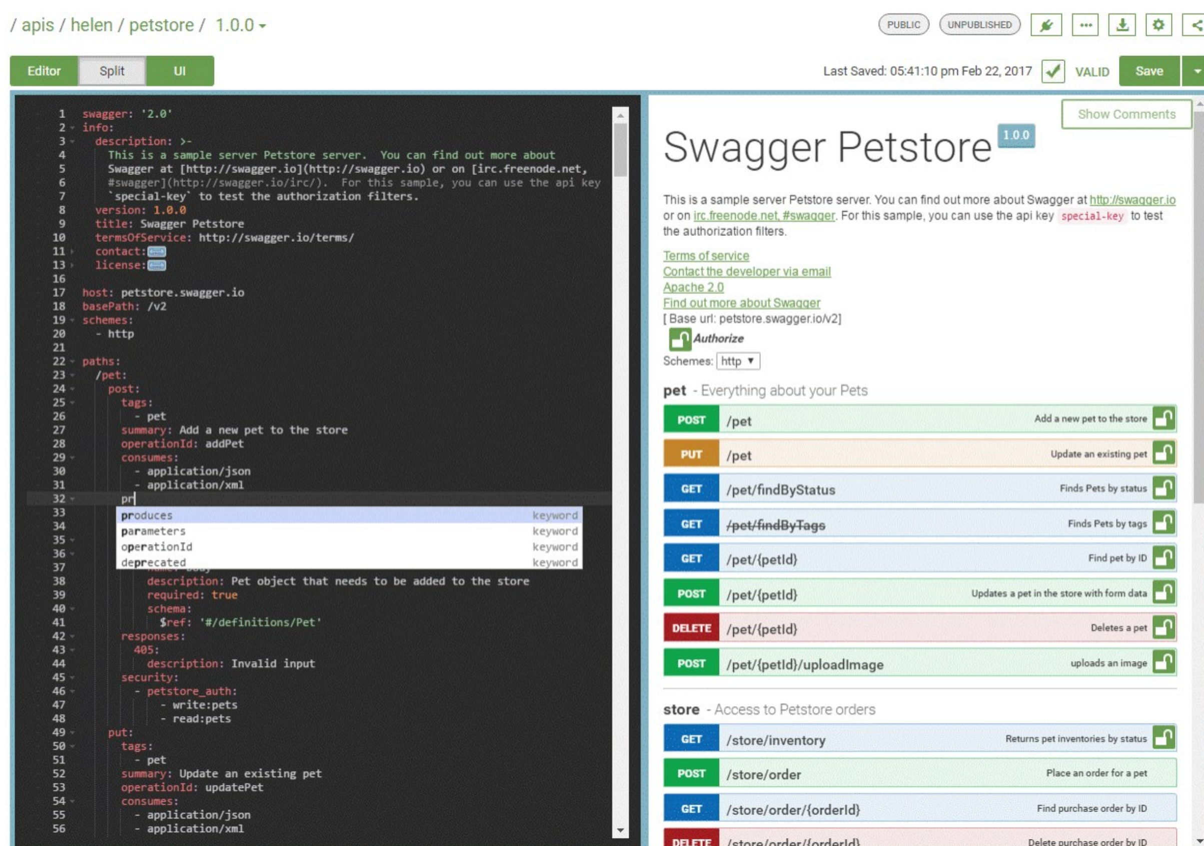
Pro vývoj webových API existují i alternativní přístupy. SOAP, zkráceno z Simple Object Access Protocol, je protokol pro komunikaci mezi webovými aplikacemi. K přenosu dat nejčastěji využívá protokol HTTP a jazyk XML. Oproti REST je složitější, méně flexibilní a pomalejší, ale za to robustnější, např. pro vývoj v bankovnictví. GraphQL je opensource dotazovací jazyk pro tvorbu API původně od společnosti Facebook. Jeho cílem je umožnit klientovi specifikovat konkrétní data, o která žádá, a tím eliminovat přenos nadbytečných dat. [7] Na rozdíl od REST využívá pouze jednu URI. Dalšími příklady webových API jsou webhooks, websockets, gRPC či SSE.

1.3.1 OpenAPI a Swagger

OpenAPI specifikace, původně Swagger specifikace, je specifikace určená pro standardizovaný a nezávislý popis a definici rozhraní webových, většinou RESTful, API. Umožňuje lidem i počítačům porozumět službám bez nutnosti znalosti zdrojového kódu nebo dokumentace. Čtenář by tedy měl být schopen jednoduše a efektivně využívat popsané API. Specifikace může být dále použita pro generaci dokumentace, generaci kódu pro klienty a servery a testování služeb. [8] V roce 2010 specifikaci pod původním názvem vytvořil Tony Tam a v roce 2015 byla přejmenována a převedena pod Linux Foundation, konkrétně pod iniciativu OpenAPI.

Popisuje se buď v JSON nebo v YAML dokumentech. Základními a povinnými poli dokumentu jsou *openapi*, *info* a *paths*. *openapi* určuje použitou verzi OpenAPI specifikace. *info* obsahuje obecné informace o dokumentu, jako je např. jméno či verze. *paths* je nejdůležitějším polem a obsahuje samotné definice všech zdrojů. Zde je možné popsat konkrétní URI, HTTP metody, parametry, těla dotazu i odpovědí a jejich stavové kódu. Další volitelné pole dokumentu je *components*, které umožňuje centralizovat a znovupoužít schémata těl dotazů a odpovědí přesunutím z konkrétních zdrojů. [9] Specifikace umožňuje detailně popsat všechny možné proměnné, které mohou nastat při komunikaci mezi serverem a klientem.

Swagger je řada nástrojů pro jednodušší práci s OpenAPI specifikací od původních vývojářů specifikace. Swagger Editor je webová aplikace, která umožňuje psaní OpenAPI dokumentů s kontrolou syntaxe, doplňováním kódu a živým náhledem. Swagger UI nabízí možnost prohlížení lidsky čitelného zobrazení OpenAPI dokumentů ve webovém prohlížeči a volání dotazů na API pomocí vestavěného klienta.



Obrázek 2: Ukázka OpenAPI specifikace (vlevo) v aplikaci Swagger Editor [10]

2 Použité technologie

2.1 Kotlin

Kotlin je moderní multiparadigmatický kompilovaný programovací jazyk poprvé představený v roce 2011. Je primárně vyvíjen společností JetBrains, ale zároveň je opensource pod licencí Apache License 2.0. Původně byl zamýšlen pouze jako lepší Java, avšak v současnosti se rozvinul daleko nad tento rámec. Může být kompilován jak do bytecode pro JVM, tak do JavaScriptu, WebAssembly nebo dokonce nativního kódu. Cílem Kotlinu je být expresivní, pragmatický, jednoduchý na využití a co nejméně „upovídaný“. Toho dosahuje inspirací a vypůjčováním již otestovaných a úspěšných konceptů z ostatních programovacích jazyků a vyhýbáním se konceptům nepraktickým. [11]

Z Javy přebírá sémantiku, virtuální stroj JVM a garbage collector. Zároveň je Kotlin s Javou velmi dobře interoperabilitní. [11] Pro rozšíření již existujících tříd o nové metody a vlastnosti je možné využít tzv. „extension functions“, převzaté z jazyka C#. Z jazyků Groovy a Ruby podporuje návrhový vzor delegace zároveň s dědičností, pojmenované argumenty a podporu vytváření typově bezpečných DSL. Stejně jako Scala umožňuje přetěžování operátorů, avšak striktnější, a využívá kombinace funkčního a objektového paradigma. [12]

2.2 Ktor

Ktor je opensource framework pro vytváření webových aplikací napsaný pouze v Kotlinu. Je vyvíjen společností JetBrains. Umožňuje snadný a flexibilní vývoj s důrazem na modularitu a rozšiřitelnost. Toho dokáže docílit plným využitím všech výhodných vlastností implementujícího jazyka. Skládá se ze 2 částí: Ktor Server a Ktor Client.

I přes to, že se Ktor označuje za framework, neomezuje uživatele ve výběru doplňujících technologií a knihoven. Volba typu logování, serializace, perzistence či dependency injection záleží čistě na vývojáři. Mnoho knihoven jazyka Kotlin s frameworkem nabízí možnost přímé integrace. Pokud knihovna tuto možnost nenabízí, je možné její funkce jednoduše vlastnoručně integrovat pomocí poskytnutých mechanismů. Pro samotné hostování lze využít jakýkoli ze servletů podporujících Servlet API, např. Netty, Jetty, CIO či Tomcat. Ktor využívá coroutines všude, kde je to možné. Většina procesů je tedy asynchronní a je schopná plně a efektivně využívat všechna vlákna. Testování aplikace je zjednodušeno díky speciální testovací knihovně. Ta umožňuje emulovat webový server bez jakýchkoli síťových volání. Například pro integrační testy lze samozřejmě testovat i oproti klasickému webovému serveru. [13]

3 Další použité technologie

3.1 Vývojové nástroje

IntelliJ IDEA je multiplatformní IDE pro kódování jazyků pro JVM, jako jsou Java a Kotlin. Je vyvíjeno společností JetBrains. Verze Community edition je zdarma k použití, avšak nabízí pouze omezené možnosti jak samotného IDE, tak i jeho doplňků. Verze Ultimate je sice placená, ale nabízí plné využití možností kódovacího prostředí. Navíc podporuje další programovací jazyky a frameworky včetně frameworku Ktor a nástroje pro profilování JVM softwaru. [14]

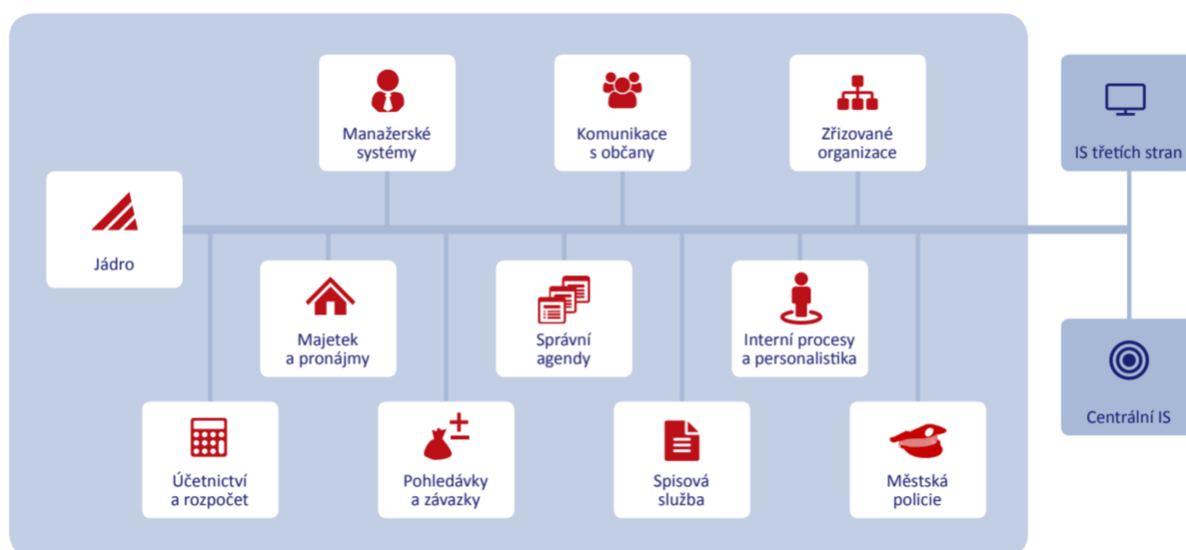
Git je opensource distribuovaný systém správy verzí. Používá se převážně pro sledování změn v kódu při vývoji softwaru. Byl vytvořen Linusem Torvaldsem v roce 2005 při práci na Linux kernelu. Jeho primárním cílem je rychlost, jednoduchost, dobrá podpora nelineárního

vývoje, být plně distribuovaný a efektivně zvládat velmi rozsáhlé projekty. Umožňuje vývojářům pracovat paralelně na různých verzích kódu a efektivně spravovat jeho změny, včetně jejich sledování, vracení a slučování. [15]

4 Představení firmy

Společnost VERA, spol. s r.o., dále jen VERA, se od roku 1994 věnuje vývoji a implementaci komplexních informačních systémů pro instituce veřejné správy. Poskytuje nejenom technologická řešení, ale také kompletní podporu v oblasti poskytovaných technologií jak v průběhu celého procesu implementace, tak při následném provozu systému. Firma vyvíjí software v souladu s legislativou České republiky a je držitelem certifikátu ISO 9001:2015 (Quality management System) a ISO 27001:2013 (Information Security Management System). Je také členem ICT UNIE a spolupracuje na rozvoji eGovernmentu.

Firma primárně nabízí 2 produkty: informační systém pro obce VERA Radnice a informační systém pro kraje a ministerstva VERA Dimenze. Oba systémy jsou složeny z mnoha vzájemně komunikujících modulů a zároveň také komunikují s různými externími informačními systémy (Obrázek 3).



Obrázek 3: Struktura IS VERA (Zdroj: VERA, spol. s r.o.)

Z hlediska technologií se ve firmě nejvíce používá programovací jazyk 4GL Genero společnosti Four Js, z menší části pak také Java 11 společnosti Oracle. Pro uchování dat se využívají databáze Oracle Database 21c, Microsoft SQL Server 2019 a PostgreSQL 14.

5 Analýza problému

Právě při komunikaci rozhraní pro propojení informačních systémů spravujících dokumenty dle národního standardu pro elektronické systémy spisové služby může dojít k sledovaným chybám. Stává se, že při zasílání dat, tzv. událostí, rozhraní externího systému požadavek odmítne, a to z mnoha různých příčin. Samozřejmě může k odmítnutí dojít i ve směru opačném. Požadavky lze zasílat buď formou synchronní, to je v reálném čase, nebo formou asynchronní, to je v dávkách.

U synchronních požadavků je validita události na straně přijímajícího rozhraní ověřena okamžitě a obratem je navrácena platnost či neplatnost požadavku. Je tedy možné zasílat a přijímat požadavky nezávisle na sobě. Asynchronní požadavky zasílájí události ve formě dávek, kde jedna dávka zpravidla obsahuje několik událostí. Pokud během zpracování události v dávce dojde k chybě, je zpracování celé dávky, a v ní obsažených událostí, pozastaveno. Pro pokračování zpracování dávky je nutné nastalou chybu opravit či chybovou událost vyjmout. Dokud není dávka dokončena, není možné zpracovávat, respektive odesílat, další dávky. Může tedy nastat situace, kdy je rozhraní delší dobu nepoužitelné, pokud není chyba řádně odstraněna. To ohrožuje proces byznys kontinuity, tzn. nečinnost uživatelů systému. Stav daného rozhraní aplikace tedy můžeme sledovat podle poslední provedené dávky, respektive zda v poslední dávce došlo či nedošlo k chybě a v jakém stavu se konkrétní dávka nachází.

V současné době je pro chybové hlášení z rozhraní využívána emailová komunikace (Obrázek 4). Z důvodu časté chybovosti dochází k zahlcení chybovými zprávami, což znemožňuje efektivní vyhodnocování. Chyby může způsobovat například rozdílnost metodik, nesoulady v číselnících, výpadky sítě či neplatné certifikáty. Cílem je tedy vytvořit aplikaci, která bude schopna zobrazovat jednoduchý přehled o chybách v rozhraní a stavech úřadů v reálném čase, čímž eliminuje potřebu manuálního procházení emailů. Nová aplikace tak umožní rychlejší a efektivnější identifikaci a opravu chyb, což přispěje k hladšímu provozu a lepšímu výkonu celého systému.

Vážený uživateli

Nepodařilo se zpracovat dávku pořadí: 246848 událost ID: 60
Zpráva: UID: magirv23o0304y není v držení externí aplikace.

Je nutné rozhodnout, zda událost reklamovat, či zda je potřeba provést nápravu na straně IS Radnice VERA.
Nastalé potíže je žádoucí řešit co nejdříve, jinak může dojít k ochromení systému.

Zpráva byla odeslána automatickým upozorňovacím systémem VERA Radnice.

Prostředí: vera

Obrázek 4: Příklad chybového emailu (Zdroj: VERA, spol. s r.o.)

5.1 Existující řešení

Nástrojů pro sledování a agregaci chyb či logů existuje na trhu spousta. I přes to, že jsou velmi podobné, existují mezi nimi dostatečné rozdíly, které by mohly být pro výběr správné platformy klíčové. Velmi populární jsou například Grafana s Grafana Loki, Splunk, Sentry, Datadog, Loggly či New Relic.

Důvodů pro vlastní implementaci existuje několik. Žádný z výše zmíněných nástrojů přímo nepodporuje Genero, ve firmě nejvíce používaný programovací jazyk. Mnoho nástrojů je sice možné využívat zdarma, avšak pouze v omezeném provozu, a plnou verzi je nutné zaplatit. Protože je požadováno řešení relativně specifického problému, nabízí ostatní nástroje zbytečně velké množství funkcionalit, které budou nevyužity. Jednodušší a konkrétnější řešení také napomáhá lepší uživatelské přívětivosti.

6 Návrh RESTful API

Základním cílem aplikace je přijímat data z externích zdrojů a poskytovat zpracovaná data pro frontend. API tedy poskytuje řadu endpointů pro obě z těchto využití. Všechny cesty, , dle firemního zvyku začínají názvem rozhraní, za kterým následuje verze rozhraní a poté samotné dílčí cesty. Pro tuto aplikaci se konkrétně jedná o prefix „vpms/v1/“. Všechny z nich vyžadují zaslání hlavičky „XVeraAPIVersion“, která nese aktuální verzi dotazujícího klienta. Pokud není verze klienta kompatibilní s verzí serveru, vrátí server stavový kód 412. Tento proces vychází z již existujícího firemního zvyku. Data jsou zasílána i přijímána výhradně ve formátu JSON, pokud se nejedná pouze o velmi jednoduchá data. K návrhu RESTful API backendové aplikace je použito specifikace OpenAPI. Ta je následně umístěna na firemní server pro přístup ostatním vývojářům pomocí Swagger UI.

Pro přijímání dat jsou použity následující cesty:

- POST */vpms/v1/external/chyby* slouží pro uložení chyby rozhraní SSL. Informace o chybě jsou zaslány v těle požadavku. Tělo má následují položky:
  - *urad* je objekt, který obsahuje detaily úřadu. Skládá se z prefixu, IČO, názvu, typu a verze SSL VERA.
  - *extapl* obsahuje kód konkrétní externí aplikace, ve které k chybě došlo.
  - *verzeNsws* obsahuje verzi rozhraní NSWS.
  - *typ* obsahuje typ externí aplikace. Nabývá buď hodnoty „A“, což je externí agenda, nebo hodnoty „S“, což je externí spisová služba.
  - *casVzniku* obsahuje čas vzniku chyby ve formátu dle standardu RFC 3339.
  - *kod* obsahuje kód chyby.
  - *popis* obsahuje krátký popis chyby.
  - *udalostNazev* obsahuje název události, při které k chybě došlo.
  - *provedlKdo* obsahuje jméno a příjmení osoby, která provedla s chybou spojenou událost.
  - *davkaPoradi* obsahuje číselné pořadí dávky, při které k chybě došlo.
  - *udalostId* obsahuje ID události, při které k chybě došlo.
  - *zpravaSmer* obsahuje směr asynchronní zprávy. Nabývá buď hodnoty „I“, což je příchozí směr, nebo hodnoty „O“, což je odchozí směr.
  - *vznikSmer* obsahuje směr, ve kterém se na chybu přišlo. Nabývá buď hodnoty „I“, což je příchozí směr, nebo hodnoty „O“, což je odchozí směr.

  Položky *urad*, *extapl*, *verzeNsws*, *typ*, *casVzniku*, *kod*, *popis* a *vznikSmer* jsou povinné.
- POST */vpms/v1/external/stavy* slouží pro uložení stavu rozhraní SSL. Informace o stavu jsou zasílány v těle požadavku. Tělo má následují povinné položky:
  - *urad* je objekt, který obsahuje detaily úřadu. Skládá se z prefixu, IČO, názvu, typu a verze SSL VERA.
  - *cas* obsahuje čas, ve kterém byl stav rozhraní naposledy aktualizován, ve formátu dle standardu RFC 3339.
  - *extapls* obsahuje pole všech externích aplikací, které jsou na rozhraní vázány. Objekt externí aplikace se skládá z:
    - *extapl* obsahuje kód externí aplikace.
    - *verzeNsws* obsahuje verzi rozhraní NSWS.
    - *typ* obsahuje typ externí aplikace. Nabývá buď hodnoty „A“, což je externí aplikace, nebo hodnoty „S“, což je externí spisová služba.

– *smer* obsahuje směr, ve kterém je pro danou externí aplikaci využíváno rozhraní. Nabývá buď hodnoty „IN“, což je příchozí směr, hodnoty „OUT“, což je odchozí směr, nebo „INOUT“, což znamená oboustrannou komunikaci.
– *prichozi* obsahuje stav zpracování příchozí dávky. Tato položka je platná a povinná pouze, pokud není *smer* hodnoty „IN“. Nabývá hodnot „N“ (Neevidovaná), „E“ (Evidovaná), „P“ (Potvrzená), „O“ (Odmítnutá), „K“ (K posouzení), „R“ (Reklamovaná), „D“ (Dokončená) nebo „S“ (Stornovaná).
– *odchozi* obsahuje stav zpracování odchozí dávky. Tato položka je platná a povinná pouze, pokud není *smer* hodnoty „OUT“. Nabývá hodnot „E“ (Evidovaná), „G“ (Vygenerovaná), „S“ (Odeslaná), „P“ (Potvrzená), „O“ (Odmítnutá), „R“ (Reklamovaná), „D“ (Dokončená) nebo „Z“ (Omezení).
– *textOmezeni* obsahuje informace o omezení. Tato položka je platná a povinná pouze, pokud je *odchozi* hodnoty „Z“.

Položky *extapl*, *verzeNsws*, *typ* a *smer* jsou povinné, ostatní jsou povinné pouze při již zmiňovaných podmínkách. Výčtové hodnoty pro příchozí a odchozí stavy jsou již existující ustálená označení pro stavy použité v dalších firemních aplikacích.

Pokud některá z povinných položek není vyplněna nebo je vyplněna chybně, vrátí server v odpovědi stavový kód 400. Pokud dotaz proběhl v pořádku, vrátí server v odpovědi stavový kód 200.

Další skupinou cest jsou cesty určené pro GUI klienta:

- GET */vpms/v1/internal/chyby* vrací seznam chyb dle zadaných parametrů. Parametry jsou povinné ID úřadu a nepovinný kód externí aplikace, kód chyby, čas od a čas do. Objekty chyb mají formát velmi podobný chybám přijímaným.
- GET */vpms/v1/internal/chyby/souhrn* vrací souhrny chyb dle zadaných parametrů. Parametry jsou povinné ID úřadu a nepovinný kód externí aplikace, kód chyby, čas od a čas do. Pro každý kód chyby vrátí nejaktuálnější chybu společně s počtem chyb daného kódu, které by odpovídaly zadanému filtru.
- GET */vpms/v1/internal/urady* vrátí seznam úřadů s jejich souhrnným stavem. Objekty úřadů mají formát velmi podobný úřadům přijímaným s přidaným polem pro stav. Stav úřadu nábývá hodnoty buď „O“ (OK), „W“ (Warning) nebo „E“ (Error).
- GET */vpms/v1/internal/urady/{uradId}/nsws* vrací souhrnný stav NSWS úřadu. Jediným parametrem je *uradId*, který se nachází v cestě. Výstupem je počet chyb za den, týden a celkem pro chyby, u kterých není známá konkrétní externí aplikace.
- GET */vpms/v1/internal/urady/{uradId}/extapls* vrací seznam souhrnných stavů rozhraní externích aplikací. Jediným parametrem je *uradId*, který se nachází v cestě. Jedná se o

konkrétní stav rozhraní, poslední evidovanou chybu a počet chyb za poslední den, 7 dní a celkový počet chyb od počátku.

- GET */vpms/v1/internal/urady/{uradId}/statistika/data* vrací data pro souhrnnou statistiku chyb, primárně určená pro grafy. Povinnými parametry jsou *uradId* v cestě, kód externí aplikace a rok, nepovinným parametrem je kód chyby. Vždy vrací pole párů typ a data. Data je pole párů měsíc a počet chyb. Pokud je zadán parametr kódu chyby, výstupem je pole jednoho páru s typem „CH“ a příslušnými daty. Pokud není zadán parametr kódu chyby, výstupem je pole 4 párů, a to typu „PV“ pro všechny příchozí chyby, typu „OV“ pro všechny odchozí chyby, typu „PZ“ pro všechny příchozí asynchronní chyby a typu „OZ“ pro všechny odchozí asynchronní chyby.
- GET */vpms/v1/internal/urady/{uradId}/extapls* vrací seznam všech externích aplikací dle úřadu. Jediným parametrem je *uradId*, který se nachází v cestě.
- GET */vpms/v1/internal/urady/{uradId}/roky* vrací seznam všech roků, pro které existují na daném úřadě a pro danou externí aplikaci záznamy. Povinnými parametry jsou *uradId* v cestě a kód externí aplikace.

Pro autentizaci dosud zmíněných cest je použito hlaviček „XVeraClientIPAddr“ obsahující IP adresu klienta, „XVeraAuthorization“ obsahující autentizační token a „XVeraFceZarId“ obsahující zvolené funkční zařazení uživatele pro autorizaci. Zvolená forma autentizace i autorizace je firemním standardem. I přes to, že pro přihlašování a odhlašování již existují endpointy v jiném z firemních RESTful API, tato aplikace obsahuje pro tyto účely vlastní rozhraní. Důvodem je z velké části větší kontrola, lepší kompatibilila s klientskou částí a postupný přechod na nový způsob autentizace. Jedná se o následující cesty:

- POST */vpms/v1/login* slouží pro přihlášení uživatele. V těle formátu *application/xwww formurlencoded* jsou v dotazu zaslány povinné parametry *login* a *password*, volitelně také *domain* a *idVoj*. Pokud je kombinace *login* a *password* neplatná, vrátí server stavový kód 401. Pokud je kombinace *login* a *password* platná, ale *idVoj* není zadáno, vrátí server stavový kód 300 a v těle odpovědi seznam všech možných VOJ, ve kterých daný uživatel existuje, se všemi funkčními zařazeními pro dané VOJ. Pokud je kombinace *login* a *password* platná, a je poskytnuto *idVoj*, vrátí server přidělený autentizační token uživatele. Přihlášení uživatele tedy vlastně probíhá ve 2 krocích. V prvním kroku pomocí přihlašovacího jména a hesla uživatel získá „identity“, pod kterými se může přihlásit, a v kroku druhém jednu konkrétní „identitu“ využije pro získání autentizačního tokenu. Autentizační token dále uživatel využívá pro autorizaci požadavků na server, zvolené funkční zařazení pak pro autorizaci.

- POST */vpms/v1/logout* slouží pro odhlášení uživatele. V hlavičce „XVeraAuthorization“ klient zašle svůj autentizační token, který je následně zneplatněn a není ho tudíž možné využít pro žádnou další komunikaci se serverem.

Pod cesty určéné pro GUI klienta patří také administrátorské cesty:

- POST */vpms/v1/admin/login* slouží pro přihlášení administrátora. Přihlášení probíhá pomocí standardní kombinace přihlašovacího jména a hesla. Pokud je kombinace platná, server v odpovědi vrátí autentizační token pro použití v následujících požadavcích.
- GET */vpms/v1/admin/logy* vrací seznam logů, které se nacházejí na serveru. Odpovědí je pole párů název a název souboru.
- GET */vpms/v1/admin/logy/{nazev}* vrací poslední řádky daného logového souboru. Ten je získán z povinného parametru *nazev*. Pomocí dalšího volitelného parametru je také možné vyžádat konkrétní počet řádků. Pokud je žádaný počet řádků větší než počet řádků logového souboru, vrátí všechny řádky.
- POST */vpms/v1/admin/instalace* slouží pro aktualizaci schématu databáze a aktualizaci práv aplikace v organizační struktuře. Instalace je spuštěna na serveru na pozadí. Při úspěšném spuštění instalace vrátí server stavový kód 200. Při opakovaném dotazu, pokud již instalace probíhá, vrátí server stavový kód 425.
- GET */vpms/v1/admin/instalace/stav* vrací současný stav instalace a část instalačního logu. Volitelným parametrem je také možné vyžádat konkrétní počet řádků logu. Instalace může být ve 4 stavech, a to „I“, pokud instalace probíhá, „F“, pokud je instalace dokončena, „E“, pokud nastala chyba při instalaci, a „N“, pokud je nutná instalace. Jestliže nastala chyba při instalaci, položka *chybovaZprava* obsahuje její znění.

Administrátorské cesty jsou na rozdíl od cest pro normální uživatele autentizovány vlastním způsobem. Protože nepodléhají firemní přihlašovací struktuře, je možné pro autentizaci využít standardní HTTP hlavičku „Authorization“. Autentizace je řešena pomocí JWT, což nese výhody i nevýhody s metodou spojené, avšak pro dané použití je dostačující.

Poslední cestou je GET */vpms/v1/info*. Ta vrací informace o aplikace, jako je její verze, název, prefix, kódování, Java verze a serverový čas.

7 Schéma databáze

Aplikace využívá SQL databázi PostgreSQL, verze 12.5. Veškerá tvoření a migrace databáze jsou zprostředkována pomocí knihovny Liquibase. Ta se v aplikaci volá skrze poskytované Java API. Schéma se skládá ze 4 tabulek: *vpmsurad*, *vpmsstav*, *vpmsstavextapl* a *vpmschyba*. Názvy tabulek jsou dle firemních konvencí prefixovány zkratkou názvu aplikace malými písmeny, tedy „vpms“. Tabulky velmi blízce odrážejí podobu přijímaných dat tak, jak je

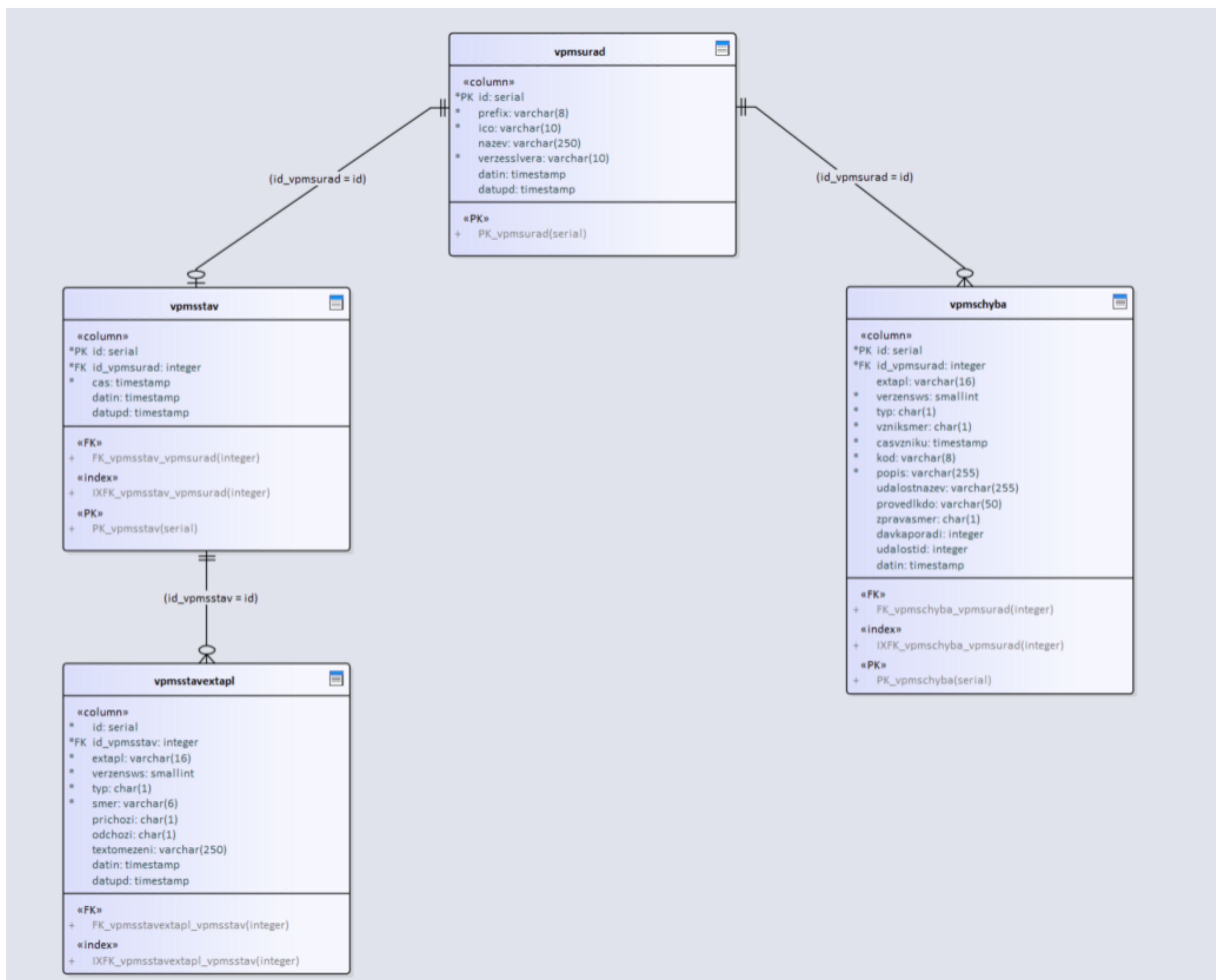
uvedeno v předešlé kapitole, kde jsou konkrétněji popsané jednotlivé položky. Každá tabulka má také sloupec *datin*, který označuje datum prvního vložení záznamu, a *datupd*, který označuje datum poslední úpravy záznamu.

Tabulka *vpmsurad* má sloupce *id*, *prefix*, *ico*, *nazev*, *verzesslvera*, *typ*, *datin* a *datupd*. Povinné sloupce jsou *id*, *prefix*, *ico*, *verzesslvera typ*. Sloupec *id* je primární klíč. Sloupce *id* a *ico* jsou indexy a také jsou unikátní.

Tabulka *vpmschyba* má sloupce *id*, *id_vpmsurad*, *extapl*, *verzensws*, *typ*, *vzniksmer*, *casvzniku*, *kod*, *popis*, *udalostnazev*, *provedlkdo*, *zpravasmer*, *davkaporadi*, *udalostid* a *datin*. Povinné sloupce jsou *id*, *id_vpmsurad*, *verzensws*, *typ*, *vzniksmer*, *casvzniku*, *kod* a *popis*. Sloupec *id* je primární klíč, sloupec *id_vpmsurad* je cizí klíč spojující tabulku *vpmsurad*. Sloupce *id*, *id_vpmsurad*, *extapl* a *kod* jsou indexy.

Tabulka *vpmsstav* má sloupce *id*, *id_vpmsurad*, *cas*, *datin* a *datupd*. Povinné sloupce jsou *id*, *id_vpmsurad* a *cas*. Sloupec *id* je primární klíč, sloupec *id_vpmsurad* je cizí klíč spojující tabulku *vpmsurad*. Sloupce *id* a *id_vpmsurad* jsou indexy.

Tabulka *vpmsstavextapl* má sloupce *id*, *id_vpmsstav*, *extapl*, *verzensws*, *typ*, *smer*, *prichozi*, *odchozi*, *textomezeni*, *datin* a *datupd*. Povinné sloupce jsou *id*, *id_vpmsstav*, *extapl*, *verzensws*, *typ*, *smer*. Sloupec *id* je primární klíč, sloupec *id_vpmsstav* je cizí klíč spojující tabulku *vpmsstav*. Slupce *id* a *extapl* jsou indexy.



Obrázek 5: Schéma databáze v Enterprise Architect

8 Ktor server

Nejdůležitější částí aplikace je samotný webový server. Pro tuto funkci byl zvolen framework Ktor programovacího jazyka Kotlin. Základní struktura projektu byla vygenerována pomocí nástroje Ktor Project Generator [16]. Tu je také možné obdobně vygenerovat v placené verzi IDE IntelliJ Idea. Pro build systém byla zvolena výchozí možnost, a to Gradle s konfigurací pomocí Kotlin Scriptu. Původně měl být zvolen serverový engine Tomcat, avšak z důvodu chyb při zpracování požadavků byl nahrazen enginem Netty. Pro konfiguraci je použito kombinace konfigurace v kódu a externí konfigurační knihovny. Konfigurace a konkrétní využité pluginy jsou blíže popsány ve vlastních kapitolách.

V následujících kapitolách jsou přiblíženy jednotlivé části implementace serveru.

8.1 Gradle

Celý projekt aplikace využívá ke kompilaci, podobně jako většina Kotlin projektů, nástroj Gradle. Ten je nastaven v build skriptu v kořenovém adresáři projektu. Spravuje všechny

závislosti aplikace, a to jak z repository Maven Central, tak z repository firemního. Další velmi důležitou součástí je automatická generace modelů z OpenAPI specifikace. Pro tento účel je vytvořeno několik vlastních tasků, tj. Gradle skripty. První nastaví správné firemní SSL certifikáty, a následně stáhne specifikaci z firemního SVN serveru. Druhý ze staženého specifikačního souboru pomocí Gradle pluginu *OpenAPI Generator* vygeneruje do zdrojové složky dle zvoleného balíku modelové třídy. Tyto tasky jsou poté sloučeny pro jednodušší použití do dalšího tasku, který ve správném pořadí provede oba dva postupně. Také je vytvořen task pro smazání všech vygenerovaných modelových tříd. Projekt je z důvodu kompatibility s ostatními firemními aplikacemi kompilován do JDK verze 11.

8.2 Konfigurace

Konfigurace logování je popsána v kapitole věnující se logování. Konfigurace samotné aplikace je primárně realizována pomocí knihovny Hoplite. Ke konfiguraci se využívá formát HOCON, hlavně díky jeho jednoduchosti, čitelnosti a dobré podpoře ve využité knihovně. Distribuční adresář v podadresáři *etc* obsahuje příkladový konfigurační soubor. Ten obsahuje všechny povinné i nepovinné položky s příklady, které jsou detailně popsány. Tento soubor je při nasazení aplikace upraven systémovými administrátory, aby vyhovoval danému prostředí. Pro vývoj programátor zkopíruje tento soubor do projektového adresáře a přejmenuje ho na „localconf.conf“. Aplikace pro získání konfiguračního souboru využívá JVM parametru s názvem „config“, ve kterém je zapsána plná cesta k danému souboru. Pokud JVM parametr není zadán, soubor neexistuje nebo je formát souboru neplatný, aplikace informuje uživatele a ukončí se.

Konfigurace obsahuje následující položky:

- *server* nastavuje port a host Ktor serveru. Port je také možné změnit pomocí volitelné proměnné prostředí *VERA_VPMS_PORT*.
- *auth* nastavuje prostředky autentizace a autorizace. Pro firemní organizační strukturu je to URL autentizačního serveru, cesta k certifikátu a heslo k certifikátu. Pro autentizaci administrátore je to cesta k souboru s přihlašovacími údaji.
- *cors* nastavuje povolené hosty pro CORS.
- *database* nastavuje JDBC URL databáze a příslušné přihlašovací jméno a heslo.
- *client* nastavuje interval pro vypisování chyb za časový úsek.

Aplikace obsahuje datové třídy, které jsou přímým odrazem zmíněných konfigurovaných položek. Knihovna při spuštění aplikace z konfiguračního souboru naplní a poskládá jednotlivé datové třídy a jejich vlastnosti do instance konfigurační třídy. Ta je následně zaregistrována do DI a využita pro typově bezpečný přístup k jednotlivým položkám.

```
application-sample.conf
01 server {
02   port = 8088
03   port = ${?VERA_VPMS_PORT}
04   host = "127.0.0.1"
05 }
06 ...
```

```
Config.kt
01 data class Config(
02     val server: Server,
03     val auth: Auth,
04     val cors: Cors,
05     val database: DatabaseConfig,
06     val client: Client,
07 ) {
08     data class Server(
09         val host: String,
10         val port: Int,
11     )
12     ...
13   }
```

Ukázka kódu 1: Část konfiguračního souboru a jemu odpovídající třída

8.3 Dependency Injection

Pro dependency injection je použita knihovna KodeinDI. I přes to, že s frameworkem Ktor se dá velmi dobře pracovat i bez dependency injection, jeho využití značně usnadňuje vytváření a správu instancí používaných tříd. Kodein také skrze pomocnou knihovnu přímo podporuje Ktor a rozšiřuje ho o specifickou funkcionalitu pro webové servery.

Samotný DI kontejner je inicializován ještě před spuštěním webového serveru v hlavní třídě. Je do něj zaregistrována instance konfigurační třídy, poskytovatelé autentizace, manažer databáze a connection pool databáze, poskytovatelé DAO a repository a další potřebné pomocné třídy. Kontejner je poté předán pro využití samotnému serveru při jeho inicializaci. Následně je možné kdekoli v aplikaci pomocí rozšiřujících metod a delegátů zažádat z dependency injection kontejneru o instance zaregistrovaných tříd. Toho je využíváno především v cestách pro poskytnutí obslužných tříd (Ukázka kódu 2).

```kotlin
Application.Kt
01 val kodeinModule = DI {
02     bindInstance { config }
03
04     bindProviderOf<IChybaDAO>(::ChybaDAO)
05     bindProviderOf<IStavDAO>(::StavDAO)
06     ...
07 }
08
09 fun Application.module() {
10     di {
11         extend(kodeinModule)
12     }
13     ...
14 }
```

```kotlin
UradyRoutes.kt
01 fun Route.uradyRoutes() {
02     val uradRepository by closestDI().instance<IUradRepository>()
03     ...
04 }
```

Ukázka kódu 2: Inicializace a použití dependency injection

8.4 Logování

Logování je obstaráno kombinací knihoven Log4j2, SLF4J a kotlinlogging. Konfigurační soubor pro logování je umístěn stejně jako aplikační konfigurace v podadresáři distribuce *etc*. Ten obsahuje několik popsaných příkladů různých nastavení appenderů a loggerů. Nejdůležitějším z appenderů je typu RollingRandomAccessFile, který umožňuje ukládat logy do souborů, které jsou dle nastavených pravidel rotovány. Je zde také nakonfigurováno několik základních loggerů, většinou knihoven. Velmi důležitá je zde nastavená vlastnost „VERA_VPMS_LOG_DIR“, která určuje adresář pro logové soubory. Tato vlastnost je totiž v aplikaci využívána pro poskytování logových souborů v API. Konfigurační soubor je při nasazení aplikace upraven systémovými administrátory, aby vyhovoval danému produkčnímu prostředí. Pro vývoj programátor zkopíruje tento soubor do projektového adresáře a přejmenuje ho na „log4j2local.xml“. Aplikace pro získání logovací konfigurace využívá JVM parametru s názvem „log4j2.configurationFile“, ve kterém je zapsána plná cesta k danému souboru. Pokud JVM parametr není zadán, soubor neexistuje nebo je formát souboru neplatný, aplikace informuje uživatele a ukončí se.

Logování v samotné aplikaci obstarává převážně knihovna kotlinlogging. Většina tříd má vlastní privátní logger definovaný ve svém companion objectu. Tento logger je pak v

konkrétní třídě využíván dle potřeby. Mohou však nastat situace, kdy není možné využít logger třídy. V hlavním aplikačním souboru je použito několik top level funkcí, které by bylo vhodné logovat. Je tedy vytvořen logger jako privátní top level proměnná, a ten je využíván v celém souboru hromadně. Logger místo jména třídy přebírá jméno souboru a dá se tedy obdobně a velmi dobře konfigurovat.

8.5 Routing

Routing je plugin, který obstarává obsluhu příchozích dotazů na webový server. Je jednou z nejdůležitějších částí celé aplikace. Jednotlivé cesty a jejich obsluha jsou rozděleny dle svých kategorií do několika podbalíků. Ty jsou sjednoceny v souboru „Routing“, který pomocí extension funkce instaluje samotný plugin a zaregistruje do něj dílčí cesty a jejich obsluhy. Konkrétní cesty a jejich vstupy a výstupy byly již zmíněny v kapitole týkající se návrhu RESTful API.

```
Routing.kt
01 fun Application.configureRouting() {
02     routing {
03         route("/vpms/v1") {
04             infoRoutes()
05             ...
06             }
07         }
08     }
09 }
```

```
InfoRoutes.kt
01 @Resource("/info")
02 class Info
03
04 fun Route.infoRoutes() {
05     get<Info> {
06         ...
07         call.respond(InfoModel(serverTime, apiVersion, ...))
08     }
09 }
```

Ukázka kódu 3: Inicializace a použití routing pluginu pro cestu */vpms/v1/info*

8.6 Pluginy

Pluginy jsou nedílnou součástí aplikací ve frameworku Ktor. Podobně jako cesty routingu je každý plugin instalován ve vlastním souboru, a následně jsou všechny pluginy pomocí

extension funkcí pro jednodušší použití sjednoceny do souboru „ConfigurePlugins“. Samotný framework poskytuje řadu základních pluginů, které jsou v aplikaci využívány.

„CallLogging“ umožňuje pomocí knihovny SLF4J logovat příchozí dotazy a odpovědi. Je nastaven na výpis HTTP metody, cesty, hlaviček a parametrů dotazu a stavového kódu odpovědi. Úroveň logování je nastavena na „INFO“. Plugin kvůli neoptimálnímu vyhodnocování logů není zcela vhodný pro využití ve vytíženějších aplikacích, při většímu rozvoji aplikace by tedy bylo nutné implementovat vlastní plugin s obdobnou funkcionalitou.

„Compression“ umožňuje komprimovat odchozí obsah. Server implementuje kompresi pomocí formátů *gzip* a *DEFLATE*. *DEFLATE* má vyšší prioritu a nastavenou minimální velikost obsahu 1024 bytů.

„ContentNegotiation“ umožňuje vyjednávání typů obsahu a jejich serializaci a deserializaci. K tomu využívá hlavičky „Accept“ a „ContentType“. Aplikace využívá pouze formát JSON, který je serializován knihovnou kotlinx.serialization. Aby nevznikaly chyby při změnách RESTful API a byla zaručena zpětná kompatibilita, serializátor je nastaven, aby ignoroval neznámé klíče.

„CORS“ umožňuje nastavení obsluhy crossorigin požadavků. Povolení hosté jsou načteni z konfigurace. Povolené HTTP metody jsou *OPTIONS*, *GET* a *POST*. Povolené hlavičky jsou standardní „ContentType“ a „Authorization“ a vlastní „XVeraAPIVersion“, „XVera ClientIPAddr“, „XVeraAuthorization“ a „XVeraFceZarId“.

„DefaultHeaders“ umožňuje přidávat hlavičky ke každé odpovědi. Plugin automaticky přidává hlavičky „Server“ a „Date“. K nim je navíc přidána vlastní hlavička „XVeraAPI Version“ s hodnotou aktuální verze aplikačního rozhraní.

„Resources“ umožňuje využívat typově bezpečné cesty. Pokud je třída anotována pomocí „@Resource(„…“)“, je možné ji použít jako handler cesty. Parametry požadavku jsou pomocí pluginu serializovány do konkrétní instance třídy, která je předána handleru jako parametr. Z instance lze následně číst typově bezpečné parametry. Veškeré cesty v aplikaci jsou zapsány konvenčně tímto způsobem, i když některé cesty žádné parametry nevyžadují.

„StatusPages“ umožňuje odpovídat na dotazy, při nichž nastala nějaká chyba. Odpovědi je možné nastavit dle stavového kódu nebo vyjímky. V aplikaci se primárně využívá jako mechanismus pro centrální zachycování nečekaných vyjímek a jejich standardizované řešení. Sekundárně také pro zpracování vyjímek vlastních pluginů, jako jsou ověření verze API či autorizace.

Ktor také poskytuje API pro vytváření vlastních pluginů. Aplikace musí při každém dotazu ověřit, zda je verze API klienta kompatibilní s verzí API serveru. Pro tuto potřebu je vytvořen plugin typu *RouteScopedPlugin*, který se místo na celou aplikaci, jak tomu bylo v pluginech již zmiňovaných, vztahuje pouze na určitou cestu. Tento plugin je součástí obecné knihovny společné pro všechny budoucí aplikace napsané ve frameworku Ktor. Verze API je rozdělena do 3 částí: kompatibilita, rozšíření a revize. Je uváděna ve tvaru *KOMPATIBILITA.ROZŠÍŘENÍ.REVIZE*. Pokud se klientovi a serveru liší kompatibilita, verze nejsou kompatibilní. Pokud je kompatibilita shodná, ale rozšíření má klient vyšší než server, verze nejsou kompatibilní. Ostatní možné kombinace se za kompatibilní považují. Kontrola verzí pluginem probíhá již v prvním kroku zpracování dotazu. Toho je dosaženo hookem „CallSetup“. Plugin získá verzi API klienta z hlavičky „XVeraAPIVersion“ a srovná ji s verzí API serveru. Pokud nejsou verze kompatibilní, vyvolá vyjímku obsahující obě verze. Plugin se využívá stylem DSL, kdy se jím obalí všechny cesty, které ho mají využívat (Ukázka kódu 4).

```
Routing.kt
01 fun Application.configureRouting() {
02     routing {
03         route("/vpms/v1") {
04             ...
05             apiVersionValidation(
06                 ApiVersion.parse(Constants.VPMS_API_VERSION)!!
07             ) {
08                 externalRoutes()
09                 internalRoutes()
10                 ...
11             }
12         }
13     }
14 }
```

Ukázka kódu 4: Využití pluginu pro validaci verze API

8.7 Datová vrstva

Pro komunikaci s databází aplikace využívá framework Exposed s connection poolem HikariCP. HikariCP je inicializován při spuštění aplikace a zaregistrován do DI kontejneru. JDBC URL a přihlašovací údaje načítá z konfiguračního souboru. Výchozí hodnota velikosti connection poolu je 10, což zhruba odpovídá doporučení knihovny. Funkce vytváření spojení z connection poolu je následně předána do Exposed. Ten z ní vytvoří novou instanci reprezentující databázi a tu zaregistruje do globálního manažeru databází. První registrovanou

databázi také nastaví jako výchozí. Vzhledem k tomu, že aplikace pracuje pouze s jednou databází, není při samotných dotazech nutné konkrétní databázi specifikovat.

Exposed pro práci s tabulkami využívá objekty, které dědí ze základní třídy *Table*. Pro každou tabulku, se kterou aplikace pracuje, je vytvořen právě jeden objekt, který jí odpovídá. Objekt má stejný název jako daná tabulka a vlastnosti adekvátního typu dle sloupců tabulky. Aplikace tyto objekty využívá pouze pro CRUD operace, ale je možné jejich pomocí dané tabulky vytvářet i mazat. Skrze DSL metody třídy *Table* je vytvořen a proveden SQL dotaz, který z databáze vrací řádek v případě dotazu na entitu či například číslo z dotazu *count*. Řádek je následně pomocnou metodou převeden na instanci reprezentující datové třídy. (Ukázka kódu 5)

Aplikace pro správu dat využívá návrhové vzory repository a DAO. DAO slouží především k obsluze jednoduchých CRUD operací a každá tabulka má svou vlastní DAO třídu. Repository obsluhují složitější požadavky, obsahují tedy většinu byznys logiky aplikace. Využívají DAO tříd, kterým poskytují připojení ke konkrétní databázi. Samotné repository jsou pomocí DI injektovány do cest, ve kterých jsou pro obsluhu potřeba.

```
VpmsChyba.kt
01 object VpmsChyba : Table() {
02     val id = integer("id").autoIncrement()
03     val extapl = varchar("extapl", 16)
04     ...
05 }
```

```
ChybaDAO.kt
01 ...
02 VpmsChyba
03     .select {
04         (VpmsChyba.casVzniku lessEq currentTime) and
05                 (VpmsChyba.idVpmsurad eq id) and
06                 (VpmsChyba.extapl eq extapl) and
07                 (VpmsChyba.davkaPoradi eq null)
08     }
09     .orderBy(VpmsChyba.casVzniku to SortOrder.DESC)
10     .limit(1)
11     .map(::mapRowToEntity)
12     .singleOrNull()
13 ...
```

Ukázka kódu 5: Získání poslední chyby z databáze

8.8 Autentizace a autorizace

Autentizace a autorizace jsou nedílnou součástí každé webové aplikace. V případě posuzované aplikace jsou metody autentizace dvě, a to každá pro jinou část aplikace.

Jednodušší z nich je autentizace administrátorské části. Skládá se z přihlášení uživatele pomocí přihlašovacího jména a hesla, a poté kontroly JWT. Stejně jako vlastní plugin je možné ve frameworku relativně jednoduše implementovat vlastního poskytovatele autentizace. K tomu stačí vytvořit potomka třídy *AuthenticationProvider*. Ta obsahuje abstraktní metodu *onAuthenticate* s parametrem typu *AuthenticationContext*, která je použita pro samotnou autentizaci. Aby byl uživatel autentizován, musí být v metodě do kontextu přidán alespoň jeden principál. Principál může obsahovat informace o uživatel, například pro použití v autorizaci. Přihlašovací údaje jsou v dotazu zaslány v těle formátu *application/x wwwformurlencoded*. Nejprve jsou deserializovány, a poté ověřeny proti platným přihlašovacím údajům. Pokud jsou platné, je do kontextu přidán principál obsahující přihlašovací jméno uživatele. V konfiguraci autentizačního poskytovatele je možné nastavit názvy parametrů pro přihlašovací jméno a heslo, přidat platné přihlašovací údaje a nastavit funkci pro odmítnutí neplatných údajů. Přihlašovací údaje se přidávají buď pomocí metody s parametry přihlašovacího jména a hashe hesla, nebo ze souboru. Soubor obsahuje právě jedny přihlašovací údaje na řádek ve formátu *jméno*:*hash_hesla*. Všechna hesla jsou hashována funkcí *bcrypt* verze $2b$. Tento typ autentizace je použit nad přihlašovací cestou, která vrací uživateli JWT pro využití v následujícím typu autentizace.

Pro jednodušší práci s JWT je vytvořena pomocná třída. Ta poskytuje ověřovatele JWT a metodu pro generaci JWT. Využívá jednotný algoritmus a klíč, který je náhodně generován při inicializaci instance třídy. To zjednodušuje nastavení a, vzhledem k zamýšlenému použití, nepřináší žádné nevýhody. Pomocí konfigurace je možné nastavit vydavatele tokenů a jejich čas vypršení. Pokud čas vypršení není zadán, token je platný do restartování aplikace. Ktor nabízí poskytovatele autentizace pro JWT. Tomu stačí předat pouze ověřovatele JWT z pomocné třídy a validační funkci. V aplikaci se JWT využívá pouze pro autentizaci, ve validační funkci tedy stačí pouze vytvořit libovolný principál, protože verifikace tokenu již proběhla. Samotný token je zasílán ve hlavičce „Authorization“ s typem Bearer. Tento typ autentizace je použit na všechny administrátorské cesty.

```kotlin
Authentication.kt
01 fun Application.configureAuthentication() {
02     ...
03     authentication {
04         localStorageLogin(AuthType.ADMIN_LOGIN.name) {
05             addCredentialsFromFile(
06                 config.auth.adminCredentialsPath.toFile()
07             )
08         }
09
10         jwt(AuthType.ADMIN_JWT.name) {
11             verifier(jwtTokenManager.verifier)
12
13             validate {
14                 JWTPrincipal(it.payload)
15             }
16         }
17         ...
18     }
19 }
```

```kotlin
AutentizaceRoutes.kt
01 fun Route.adminAutentizaceRoutes() {
02     ...
03     authenticate(AuthType.ADMIN_LOGIN) {
04         post<Login> {
05             call.respond(
06                 AdminLoginResponse(jwtTokenManager.generateToken())
07             )
08         }
09     }
10 }
```

Ukázka kódu 6: Registrace poskytovatele autentizace a jeho použití

Druhou variantou autentizace je ověřování oproti firemní jednotné organizační struktuře, tzv. JOS či JS. K tomu je možné z části využít již existující firemní knihovny napsané v jazyce Java. Z důvodu lepší kompatibility pro Kotlin a jednodušší práce s autentizačními poskytovateli je obalena vlastní třídou *JosAuthenticator*. Ta poskytuje všechny potřebné metody pro autentizaci a autorizaci. První poskytovatel autentizace pro tyto účely využívá přihlašovací jméno, heslo, doménu, ID VOJ a IP adresu hosta. Tyto parametry jsou kromě hosta přečteny z těla dotazu ve formátu *application/xwwwformurlencoded*. IP adresu hosta zasílá klient v hlavičce „XVeraClientIPAddr“. Pomocí zaslaných údajů dojde k pokusu o přihlášení do organizační struktury. Pokud se podaří uživatele přihlásit, ale není zadán ID VOJ, je zavolána příslušná obsluhující funkce, které jsou předány možné VOJ a funkční

zařazení uživatele. Pokud se podaří uživatele přihlásit a je zadán ID VOJ, v kontextu je nastaven principál obsahující údaje o uživateli. V konfiguraci je nutné nastavit instanci již zmiňované třídy pro autentizaci a funkci obsluhující případ nezadaného ID VOJ. Tento typ autentizace je použit nad přihlašovací cestou, která vrací uživateli ticket z organizační struktury pro využití v následujícím typu autentizace.

Autentizaci pomocí ticketu, což v JOS odpovídá standardnímu tokenu, využívá naprostá většina cest. Ticket zasílá klient v hlavičce „XVeraAuthorization“. Ticket je pomocí autentizační třídy ověřen proti organizační struktuře. Pokud je platný, je nastaven do principálu kontextu pro použití v autorizaci. V konfiguraci je nutné nastavit instanci již zmiňované třídy pro autentizaci.

Autorizace je řešena pouze pro autentizaci pomocí ticketu. Pro účely autentizace je implementován vlastní plugin způsobem obdobným pluginu pro kontrolu verze API. K autorizační fázi dochází až po autentizaci uživatele. Toho je docíleno pomocí hooku „AuthenticationChecked“. Zde je důležité upozornit, že hook je vykonán i při volitelné autentizaci či dokonce bez autentizace, což je nutné zohlednit při samotném využití pluginu. Autorizuje se pomocí ticketu a ID funkčního zařazení. Ticket je dostupný z principálu již proběhlé autentizace a ID funkčního zařazení je zasíláno v hlavičce „XVeraFceZarId“. Autorizace dle nastavení ověřuje, zda má uživatel všechna požadovaná práva, nebo zda má uživatel alespoň jedno z požadovaných práv. Pokud se uživatele nepodaří autorizovat, vyvolá plugin vyjímku „UnauthorizedAccessException“. Tu je následně možné zachytit pomocí pluginu „StatusPages“ a dále ji zpracovat.

Všechny vlastní autentizační i autorizační metody jsou součástí obecné knihovny, která je společná pro všechny budoucí aplikace napsané ve frameworku Ktor, a to včetně pomocné třídy pro generaci a ověřování JWT.

8.9 Instalace

V případě této aplikace se instalací rozumí aktualizace či vytvoření databáze a práv v organizační struktuře. Samotná aplikace je distribuována ve formátu JAR a není ji tedy nutné instalovat v klasickém slova smyslu.

Pro správu databáze je využíván nástroj Liquibase. Liquibase je primárně volán z CLI jako samostatně spustitelný program, ale nabízí také Java knihovny pro volání přímo v kódu, což právě aplikace využívá. Pro práci s databází je určena třída *DatabaseManager*, která pomocí Liquibase umožňuje aktualizaci databáze a zjištění stavu databáze. Aktualizace je řešena třídou *CommandScope*, která se používá jako fasáda pro tvorbu příkazů (Ukázka kódu 7). Je

vytvořena instance s názvem příkazu „update“ pro aktualizaci. Do instance jsou přidány argumenty obsahující naslouchače změn, databázi a soubor s changelogem. Nakonec je příkaz spuštěn metodou *execute*. Pro zjištění aktuálního stavu databáze Liquibase poskytuje třídu *StatusCommandStep* s metodou *listUnrunChangeSets*. Ta pomocí parametrů databáze a changelogu zjistí, zda jsou v dané databázi všechny změny changelogu provedeny, a vrátí změny neprovedené. Tato metoda je značně jednodušší než stavění celého příkazu, jak tomu bylo v případě aktualizace databáze. Volání Liquibase by mělo probíhat vždy ve vlastním „scope“. Ty jsou však vázány na konkrétní vlákno, což při opakovaném volání může způsobovat problémy. Coroutines v Kotlinu totiž nezaručují stejné vlákno. Je tedy nutné jednotlivá volání řádně synchronizovat.

```
DatabaseManager.kt
01 fun install() = synchronizedScopedLiquibase {
02     CommandScope(*UpdateCommandStep.COMMAND_NAME).apply {
03         addArgumentValue(
04             ChangeExecListenerCommandStep.CHANGE_EXEC_LISTENER_ARG,
05             installChangeListener
06         )
07         addArgumentValue(
08             DbUrlConnectionCommandStep.DATABASE_ARG,
09             it.database
10         )
11         addArgumentValue(
12             UpdateCommandStep.CHANGELOG_FILE_ARG,
13             it.changeLogFile
14         )
15
16         execute()
17     }
18 }
```

Ukázka kódu 7: Aktualizace databáze pomocí nástroje Liquibase

Pro možnosti autorizace vytváří aplikace v organizační struktuře vlastní práva. Ta jsou definována jako číselník v kódu. Třída *JsPravoManager* umožňuje aktualizaci práv a zjištění stavu práv. Pro tento účel využívá firemní knihovny. Pro aktualizaci práv se jednoduše pomocí knihovny přidají všechna práva z číselníku do organizační struktury. Duplikáty práv organizační struktura řeší sama způsobem, že pokud již právo existuje, není přidáno. Pro zjištění aktuálnosti práv v organizační struktuře je nejprve nutné zažádat o všechna práva organizační struktury pro danou aplikaci pomocí vrcholového práva. Ta jsou následně porovnána exkluzivní disjunkcí s právy z číselníku.

Tyto manažerské třídy jsou spojeny a využívány třídou *InstallationManager*. Ta spouští tyto instalace paralelně. Zároveň spravuje řádnou synchronizaci instalací a ošetřuje jejich chybové stavy. Také zaznamenává stav instalace v daném okamžiku a kontroluje, zda proběhly instalace bez chyb.

8.10 Testování

Serverová část aplikace je testována převážně unit testy, aplikace jako celek je testována i ručně. Testovány jsou i obecné knihovny s autorizací a pluginy. Pro testování jsou využity knihovny JUnit, kotlintest, ktorservertest a MockK.

Repository jsou testovány mockováním DAO tříd pomocí knihovny MockK. Před spuštěním každého testu jsou vytvořeny mockovaná instance, které jsou použity pro novou instanci daného repository. Po skončení každého testu jsou všechny mockované instance, včetně repository, uvolněny. Vzhledem k tomu, že repository standardně pracuje s databází, je nutné i tuto operaci nahradit. Toho je docíleno statickým mockem třídy *ThreadLocalTransactionManager* a metody *transaction*. V testech je většinou ověřována funkčnost jednotlivých metod třídy a ne komplexní volání několika metod. (Ukázka kódu 8)

```
ChybaRepositoryTest.kt
01 @Test
02 fun `pridani chyby, pokud neni urad v DB`() {
03     val uradDaoAddReturn = 1
04
05     val urad = UradDbModel.fromRequest(uradModel)
06     val chyba = Chyba.fromRequest(chybaRequest).apply {
07         idVpmsurad = uradDaoAddReturn
08     }
09
10     every { uradDao.getByIco(any()) } returns null
11     every { uradDao.add(any()) } returns uradDaoAddReturn
12     every { chybaDao.add(any()) } returns Unit
13
14     chybaRepository.save(chybaRequest)
15
16     verifySequence {
17         uradDao.getByIco(chybaRequest.urad.ico)
18         uradDao.add(
19           match { matchObjectsByPropertyValues(it, urad) }
20         )
21         chybaDao.add(
22           match { matchObjectsByPropertyValues(it, chyba) }
23         )
24     }
25 }
```

Ukázka kódu 8: Test repository pro přidání chyby

Testy poskytovatelů autentizace a pluginů využívají knihovny ktorservertest. Ta pro testování serverové části poskytuje funkci *testApplication*. Ve funkci je jednoduchým DSL možné vytvořit ekvivalent webového serveru a do něj instalovat požadované pluginy či autentizátory. Také obsahuje Ktor klienta, ze kterého je možné server dotazovat. Pro testování každého pluginu je vytvořena třída, která obsahuje co nejminimalističtější konfigurovaný server potřebný pro jeho otestování. Většinou také obsahuje vlastního klienta s pluginy potřebnými pro testování, jako je například „ContentNegotiation“. (Ukázka kódu 9)

```
JosTokenAuthenticationProviderTest.kt
01 @Test
02 fun token() = baseTestApplication {
03     val token = generateToken()
04
05     client.get("/private") {
06         headers.append(authorizationHeaderName, token)
07     }.apply {
08         assertEquals(HttpStatusCode.OK, status)
09         assertEquals(token, bodyAsText())
10     }
11 }
```

Ukázka kódu 9: Test autentizace pomocí ticketu

Závěr

Cílem této bakalářské práce bylo navrhnout a implementovat serverovou část pro firemní systém zpracování chyb. Byly představeny využité návrhové vzory a techniky, také byl přiblížen implementující programovací jazyk a framework. Byl uveden firemní kontext, ve kterém byla práce psána, a analýza daného problému. Dále se práce věnovala návrhu webového rozhraní aplikace, což se dá považovat za nejdůležitější část. Na závěr byla přiblížena konkrétní implementace aplikace v jazyku Kotlin, webovém frameworku Ktor a několika souvisejících knihovnách.

Pro firmu byla vytvořená aplikace přínosná natolik, že již zvažuje její rozšíření o další původně neplánovanou funkcionalitu. Také firmu posunula z hlediska využívaných technologií a nových vývojových postupů. Tím tedy splnila i svůj vedlejší cíl, který přímo nebyl předmětem této bakalářské práce. Práce snad také poslouží k dokumentaci implementace a využití vytvořené aplikace.

Použitá literatura

[1] IBM. *What is an application programming interface (API)?* [online]. [vid. 20240101]. Dostupné z: https://www.ibm.com/topics/api

[2] KRISTOPHER SANDOVAL. *How Are Web APIs Different Than System APIs?* [online]. [vid. 20240101]. Dostupné z: https://nordicapis.com/howarewebapisdifferentthan systemapis/

[3] FIELDING, Roy Thomas a Richard N. TAYLOR. *Architectural styles and the design of networkbased software architectures*. B.m., 2000. University of California, Irvine.

[4] *REST (Representational State Transfer)* [online]. [vid. 20240101]. Dostupné z: https:// fullstack.wiki/architecture/rest

[5] *2023 State of the API Report* [online]. [vid. 20240101]. Dostupné z: https://www. postman.com/stateofapi/apitechnologies/#apitechnologies

[6] *Justice Will Take Us Millions Of Intricate Moves: Act 3* [online]. [vid. 20240101]. Dostupné z: https://www.crummy.com/writing/speaking/2008QCon/act3.html

[7] *GraphQL* [online]. [vid. 20240101]. Dostupné z: https://graphql.org/

[8] *OpenAPI Specification v3.1.0* [online]. [vid. 20240101]. Dostupné z: https://spec. openapis.org/oas/latest.html

[9] *Structure of an OpenAPI Description* [online]. [vid. 20240101]. Dostupné z: https:// learn.openapis.org/specification/structure.html

[10] *Introducing the New SwaggerHub Editor: A Smarter Editor for Faster API Design* [online]. [vid. 20240101]. Dostupné z: https://swagger.io/blog/apidevelopment/new swaggerhubeditor/

[11] ECKEL, Bruce a Svetlana ISAKOVA. *Atomic Kotlin*. B.m.: Mindview LLC, 2021. ISBN 9780981872551.

[12] SUBRAMANIAM, Venkat. *Programming Kotlin: Create Elegant, Expressive, and Performant JVM and Android Applications*. B.m.: Pragmatic Bookshelf, 2019. ISBN 1680506358.

[13] *Ktor* [online]. [vid. 20240101]. Dostupné z: https://github.com/ktorio/ktor

[14] *IntelliJ IDEA – the Leading Java and Kotlin IDE* [online]. [vid. 20240101]. Dostupné z: https://www.jetbrains.com/idea/

[15] CHACON, Scott a Ben STRAUB. *Pro Git*. 2nd ed. B.m.: Apress, 2014. ISBN 1484200772.

[16] *Ktor Project Generator* [online]. [vid. 20240101]. Dostupné z: https://start.ktor.io/#/settings