VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

REENGINEERING DVOUVRSTVÝCH APLIKACÍ

REENGINEERING OF TWO-TIER APPLICATIONS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. MICHAL OBERREITER**
AUTHOR

VEDOUCÍ PRÁCE — **doc. Ing. RADEK BURGET, Ph.D.**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2021/2022

Zadání diplomové práce

Student: **Oberreiter Michal, Bc.**
Program: Informační technologie
Obor: Softwarové inženýrství
Název: **Reengineering dvouvrstvých aplikací**
Reengineering of Two-Tier Applications
Kategorie: Softwarové inženýrství

Zadání:

1. Seznamte se s problematikou architektury dvouvrstvých a třívrstvých aplikací a s problematikou reengineeringu.
2. Identifikuje obecné principy, činnosti a možné problémy při reengineeringu dvouvrstvých aplikací na třívrstvé.
3. Po dohodě s vedoucím zvolte vhodnou reálnou aplikaci, analyzujte požadavky a navrhněte způsob reengineeringu.
4. Implementujte přepracovanou aplikaci podle návrhu.
5. Proveďte testování výsledného řešení.
6. Zhodnoťte dosažené výsledky.

Literatura:

- Pinto de Matos, C. M.: Reengineering Software to Three-tier Applications andServices, Ph.D. thesis, University of Leicester, 2011
- Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, Third edition, Addison-Wesley, 2015

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Burget Radek, doc. Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 18. května 2022
Datum schválení: 11. října 2021

Abstrakt

Tato práce se zabývá analýzou problematiky reengineeringu v kontextu modernizace dvouvrstvých aplikací. Na základě dostupné literatury je popsána metodologie, přístupy a možné strategie k realizaci reengineeringu. Z pohledu softwarových architektur, které práce popisuje a vzájemně porovnává, jsou diskutována konkrétní řešení vybraných problémů v různých architekturách. Pro aplikaci principů reengineeringu je zvolena ve spolupráci se společností I&C Energo reálná aplikace Systém správy kabeláže. Stávající řešení je analyzováno, nedostatky identifikovány. Na základě zmapovaných technik je navrhnuto cloud-native řešení v architektuře mikroslužeb, které je následně implementováno a popsáno. Výstupem práce je případová studie aplikace reengineeringu na Systému správy kabeláže.

Abstract

This thesis deals with an analysis of issues regarding reengineering of two-tier applications. Based on the available literature, a comprehensive description of methodology, strategies and approaches is compiled. Relevant software architectures are described, mutually compared and selected problems and their solutions are discussed. As a demonstration of the described methodology, a real-world application was chosen in collaboration with company I&C Energo. This application named Cable Management System is thoroughly analysed and issues concerning the current implementation are identified. Based on the gathered knowledge, a new reengineered cloud-native microservice-based solution is designed and implemented. The result of this thesis is a case study of reengineering application on concrete example of Cable Management System.

Klíčová slova

reengineering, dvouvrstvé, případová studie, architektura, mikroslužby, kontejnery, cloud, cloud-native, orchestrace, OpenShift, Docker, Kubernetes

Keywords

reengineering, two-tier, case study, architecture, microservices, containers, cloud-native, cloud, orchestration, OpenShift, Docker, Kubernetes

Citace

OBERREITER, Michal. *Reengineering dvouvrstvých aplikací*. Brno, 2022. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Ing. Radek Burget, Ph.D.

Reengineering dvouvrstvých aplikací

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením doc. Ing. Radka Burgeta, Ph.D. Informace a materiály k reálnému systému mi poskytl Ing. Václav Hajšman, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Michal Oberreiter
18. května 2022

Poděkování

Chtěl bych tímto poděkovat vedoucímu práce docentu Burgetovi za asistenci a konzultace, doktoru Hajšmanovi, který mi poskytl cenné informace ohledně stávajícího řešení systému a konzultoval se mnou návrh nového řešení. Také bych chtěl vyjádřit dík společnosti I&C Energo, že mi umožnila demonstrovat reengineering na reálném systému.

Obsah

1 Úvod **6**

2 Softwarové architektury a principy reengineeringu **7**
2.1 Softwarové architektury 7
2.1.1 Klient-server architektury 7
2.1.2 Service Oriented Architecture 13
2.1.3 Mikroslužby 18
2.2 Software reengineering 24
2.2.1 Východiska a cíle reengineeringu 24
2.2.2 Business aspekty reengineeringu 25
2.2.3 Rizika 26
2.2.4 Metodologie 27
2.2.5 Strategie 30
2.2.6 Přístupy k náhradě existujícího řešení 31
2.2.7 Metody reengineeringu v kontextu architektury služeb 32

3 Reengineering aplikace Systém správy kabeláže **34**
3.1 Účel a základní popis systému 34
3.1.1 Moduly systému 35
3.1.2 Podobná a konkurenční řešení 36
3.2 Architektura a implementace výchozího řešení 38
3.3 Datová a procesní integrace 41
3.4 Východiska reengineeringu 41
3.5 Identifikace nedostatků stávajícího řešení 41
3.5.1 Externí nedostatky 42
3.5.2 Interní nedostatky 43
3.5.3 Technologické nedostatky 44
3.5.4 Integrační nedostatky 44
3.6 Požadavky na cílové řešení 44
3.7 Stanovení cílového řešení 46
3.8 Návrh cílového řešení 46
3.8.1 Aplikace reengineeringu 46
3.8.2 Architektura 47
3.8.3 Uživatelské rozhraní 58

4 Implementace řešení **62**
4.1 Celková koncepce 62
4.2 Logování 63

4.3 Škálovatelnost . . . 64
4.4 Autorství . . . 66
4.5 Realizované služby . . . 66
4.5.1 Služba autentizace . . . 66
4.5.2 Služba Gateway . . . 68
4.5.3 Služba Reporting . . . 69
4.5.4 Služba Core API . . . 70
4.5.5 Služba Graphics . . . 80
4.5.6 Služba FileStorage . . . 82
4.6 Klientská aplikace . . . 83
4.7 Testování systému . . . 87
4.8 Instalace a spuštění . . . 87

5 Vyhodnocení aplikovaných technik reengineeringu **89**

6 Závěr **91**

Literatura **92**

Přílohy **96**

A Instalace a spuštění **97**
A.1 Prerekvizity . . . 97
A.2 Spuštění načtením images . . . 97
A.3 Spuštění sestavením ze zdrojů . . . 98
A.4 Start aplikace . . . 98
A.5 Provozování v Red Hat OpenShift . . . 98

B Testovací scénář **99**

Seznam obrázků

2.1 Ukázka rozložení layers mezi tiers – vrstvy – ve dvouvrstvé architektuře, inspirováno a doplněno [5] 9
2.2 Porovnání modelů škálování databázových serverů, inspirováno [11] 11
2.3 Ukázka rozložení layers mezi tiers – vrstvy – ve třívrstvé architektuře, inspirováno a doplněno [5] 12
2.4 Schéma protokolu OpenID Connect, inspirováno [40] 13
2.5 Diagram komunikace publikace, vyhledání a volání služby, inspirováno [13] . 14
2.6 Příklad realizace ESB, převzato z [8] 16
2.7 Schéma vzájemné komunikace při absenci ESB, převzato z [8] 17
2.8 Schéma komunikace přes message oriented middleware, převzato z [8] 17
2.9 Schéma kolekce, zpracování, uložení a vizualizace dat v Elastic stacku, inspirováno [12] 19
2.10 Schéma komunikace synchronní a asynchronní, inspirováno [47] 21
2.11 Fáze a kroky analýzy rizik dle Boehma [6] 26
2.12 Ukázka mapy rizik, volně inspirováno [10] 27
2.13 Model podkovy reengineeringu, převzato a inspirováno z [34, str. 30] a [27] . 28
2.14 Kontext refaktorizace podle Ivkovic a Kontogiannis, převzato z [26] 30
2.15 Big-bang přístup k náhradě systému, inspirováno [44] 31
2.16 Inkrementální přístup k náhradě systému, inspirováno [44] 32
2.17 Evoluční přístup k náhradě systému, inspirováno [44] 32

3.1 Základní moduly SSK, zdroj [23] 35
3.2 Vstupy a výstupy algoritmu trasování, autor I&C Energo 36
3.3 Prostředí nástroje AVEVA.NET pro prohlížení 1D, 2D i 3D dat, zdroj [2] . 37
3.4 Architektura SSK z pohledu jedné lokality 38
3.5 Modul zobrazení 3D schématické grafiky SSK 39
3.6 Okno náhledu reportu v SSK 2.0 40
3.7 Hlavní obrazovka modulu v SSK 42
3.8 Dialogové okno zadání filtru SSK 43
3.9 Architektura cílového řešení SSK 48
3.10 Sekvenční diagram OpenID Connect autentizace v SSK 50
3.11 Sekvenční diagram získání reportu 51
3.12 Ukázka GraphQL 53
3.13 Porovnání paralelní editace bez/s řešením souběhu 54
3.14 Sekvenční diagram získání 3D dat 56
3.15 Sekvenční diagram obnovy cache stavby 57
3.16 Sekvenční diagram nahrání souboru službou FileStorage 58
3.17 Wireframe obrazovky modulu 59

3.18 Prototyp filtrovacího dialogu 60
3.19 Wireframe obrazovky detailu záznamu 60
3.20 Wireframe obrazovky modulu grafiky 61

4.1 Vizualizace sesbíraných logů v prostředí Kibana pod Elastic 64
4.2 Ilustrace zasílání zpráv přes SignalR s Redis backplane, převzato z [35] . . . 65
4.3 Přihlašovací okno Keycloak 67
4.4 Administrace klienta SSK 67
4.5 Ukázkový výstup reportu 69
4.6 Integrace návrháře reportů do SSK 70
4.7 Kontrola stavu reportu a kontextová akce otevření šablony reportu 70
4.8 Základní obrazovka modulu s filtry 84
4.9 Modul grafiky 84
4.10 Okno detailu entity s integrovaným modulem grafiky 85

Seznam tabulek

3.1 Nefunkční požadavky na nové řešení SSK 45
3.2 Funkční požadavky na nové řešení SSK 46
3.3 Návrh vybraných endpointů rozhraní služby Core API 52
3.4 Návrh rozhraní služby Graphics 55
3.5 Návrh rozhraní služby FileStorage 58

4.1 Rychlost odpovědi podů služby Core API dle zátěže 65
4.2 Porovnání rychlosti načítání 3D objektů staveb 82

Kapitola 1

Úvod

Výzvy dnešní doby, kdy práce z domova se stává samozřejmostí, kladou zvýšené nároky na fungování infrastruktury a systémů na ní provozované. Aplikace se přesouvají do cloudu a škálovatelnost s integrovatelností je dnes klíčovou vlastností. Organizace dodnes provozují klasické dvouvrstvé aplikace typu klient–databáze, které v prostředích virtualizovaných pracovních stanic nebo VPN nevykazují optimální funkčnost.

Tato práce se pokusí zmapovat problematiku reengineeringu a identifikovat konkrétní problémy při převodu aplikací z dvouvrstvé architektury. Jednotlivé architektury budou porovnány a budou diskutovány specifické aspekty řešení některých problémů, jakými je například bezpečnost, škálovatelnost či distribuce.

Pro demonstraci aplikace zmapovaných principů, metodologie a strategie bude navrhnuto nové řešení reálné aplikace Systém správy kabeláže. Tento systém byl vybrán ve spolupráci se společností I&C Energo, která tento produkt dodává pro správu kabelového systému na několika českých elektrárnách, včetně Jaderné elektrárny Dukovany a Temelín. Systém je v současné době implementován právě jako dvouvrstvá aplikace. Modernizace systému otevře dveře budoucímu rozvoji, který umožní zakomponování nové funkcionality a dovolí lepší integraci na jiné systémy. Principy popsané a aplikované při reengineeringu tohoto systému budou šířeji uplatnitelné pro obecně jakékoliv dvouvrstvé aplikace.

Kapitola 2 bude popisovat a porovnávat softwarové architektury a diskutovat problematiku reengineeringu z pohledu přístupu, metodologie, ale i rizik a důvodů pro započetí procesu reengineeringu. V kapitole 3 bude analyzováno stávající řešení, identifikovány nedostatky, stanoveny požadavky, navrhnuto a detailně popsáno nové řešení v architektuře mikroslužeb. Konkrétní volby technologií a detaily implementace budou rozvedeny v kapitole 4. Kapitola 5 pak vyhodnotí užití technik reengineeringu a pokusí se o zobecnění pro reengineering dvouvrstvých aplikací.

Kapitola 2

Softwarové architektury a principy reengineeringu

Tato kapitola popisuje vybrané softwarové architektury a diskutuje je v kontextu reengineeringu jako procesu. Dále identifikuje principy, oblasti a možné problémy, které mohou vzniknout při reengineeringu dvouvrstvých aplikací na třívrstvé.

2.1 Softwarové architektury

Softwarová architektura je systém struktur potřebných k popsání elementů, relací a jejich vlastností [3]. Architektura slouží k dekompozici systému do bloků a tím dále usměrňuje činnost v rámci návrhu řešení. Výběr architektury hraje tedy klíčovou roli a patří společně s výběrem technologie k rozhodnutím s největším dopadem při pozdější změně ve fázi implementace. Rozhodnutí použít danou architekturu by mělo být podloženo analýzou na základě konkrétních funkčních, ale i nefunkčních požadavků.

2.1.1 Klient-server architektury

Architektury typu klient-server umožňují sdílet data napříč vícero klienty prostřednictvím serveru. Typickým příkladem této architektury je web, kde webový prohlížeč – klient – přistupuje a žádá server o data. Takový server obsluhuje požadavky mnoha klientů [46].

Klient-server se řadí mezi jedny z nejstarších softwarových architektur. Jedním z prvních použití této architektury je služba emailu, která datuje svůj původ do roku 1965 na Massachusetts Institute of Technology (MIT) [49]. K realizaci této služby byl využit jeden z prvních time-sharing operačních systémů – Compatible Time-Sharing System (CTSS) – vyvinutý na stejné univerzitě. Základní myšlenka příkazu `MAIL` je připisována Louisovi Pouzionovi, Glendě Schroeder a Patovi Crismanovi, prvotní implementace pak Tom Van Vleckovi a Noeli Morrisovi [49]. Služba nahradila předávání vzkazů mezi uživateli CTSS pomocí souborů pojmenovaných uživateli dohodnutou konvencí. Nyní zaběhnutá konvence `@` znaku a formátu `user@hostname`, respektive `user@hostname.domain`, je dílem Raye Tomlinsona [45], který jako první implementoval systém emailu na síti ARPANET, která je obecně vnímána jako předchůdce moderního internetu.

V klasickém pojetí požadavky iniciuje pouze klient, server tedy nemůže zahájit činnost bez vzniklého požadavku [18]. Toto omezení se v praxi obchází zavedením služby na straně serveru, která může či nemusí být součástí serverové aplikace, lze tedy hovořit o *hostovaných* a *systémových* službách.

Hostovaná služba je taková, která je přímou součástí sestavení serverové aplikace a její životní cyklus je svázán s cyklem serverové aplikace [21]. Příkladem těchto služeb je implementace ASP.NET Core `BackgroundService` či Quartz.NET[1]. U tohoto typu služeb je taktéž nutné se zabývat zajištěním běhu tzv. *aplikačních poolů* (izolované procesy hostující jednu či více aplikací), u kterých je třeba pro správný chod služeb zajistit inicializaci aplikace při startu poolu[2]. Systémová služba je pak taková služba, která běží na úrovni operačního systému a není součástí aplikace.

Klient-server architektury lze dělit do vrstev – tiers – na dvouvrstvé, třívrstvé a obecně n-vrstvé. Každá vrstva je typicky reprezentována požadavky na prostředí a může být umístěna na oddělených výpočetních jednotkách propojených sítí. Nezávisle na počtu vrstev lze v architektuře typu klient-server identifikovat následující části – layers:

- **prezentační**
- **aplikační**
- **datová**

Tier označuje hardwarovou vrstvu, tedy například různé stanice, zatímco layer označuje softwarovou „vrstvu“/část. Způsob, jakým se tyto části dělí, následně určuje počet vrstev, který výsledná architektura bude mít.

Dvouvrstvá architektura

Architektura dvouvrstvého typu je taková architektura, kde se prezentační část nachází na straně klienta, data na serveru a aplikační logika je rozdělena mezi klienta a server. Míra dělení aplikační logiky mezi klienta a server je určena faktory, jakými jsou například:

- **datová intenzita operací** – operace náročná na data se typicky vykonávají na serveru a své výsledky zasílají na klienta
- **sdílení funkcionality mezi různými typy klientů** – například přístup k datům serveru pomocí nástrojů typu MS Access, Toad for Oracle, JetBrains DataGrip apod.
- **zajištění integrity dat** – server může vynucovat integritu dat namísto spoléhání se na korektnost chování klientů

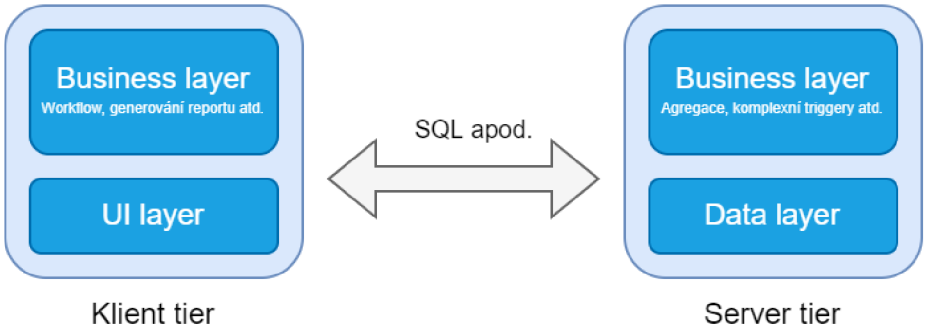
Řešení této problematiky je závislé na konkrétním projektu a je ovlivněno nejenom technickými, ale i business požadavky.

Typicky je klientem uživatelská stanice v podobě osobního počítače, kde je nainstalována klientská aplikace daného systému, která přes síťové rozhraní s pomocí příkazů SQL komunikuje se serverem, kde běží serverová služba poskytující relační databázi – RDBMS. V rámci RDBMS mohou být implementovány funkce, procedury, triggery apod., které vykonávají aplikační logiku nad daty.

Dvouvrstvá architektura přináší určité benefity, ovšem také určité limitace, překážky či nebezpečí [18]. Mezi hlavní výhody patří rychlejší počáteční vývoj a menší technologický *stack*, jelikož jedna vývojová platforma typicky zvládne pokrýt tvorbu uživatelského

[1] viz https://www.quartz-scheduler.net

[2] viz https://www.quartz-scheduler.net/documentation/faq.html#scheduler-keeps-stopping-when-application-pool-gets-recycled



Obrázek 2.1: Ukázka rozložení layers mezi tiers – vrstvy – ve dvouvrstvé architektuře, inspirováno a doplněno [5]

rozhraní, práci s daty a komunikaci s databází, příkladem může být například Windows Presentation Foundation (WPF) na platformě .NET či JavaFX pro Java. Pro práci s daty může být využito základních SQL klientů, kteří jsou povětšinou standardní součástí těchto platforem, například `System.Data.SqlClient` pro .NET či `java.sql` pro Java. Mezi další výhody za určitých podmínek patří taktéž nižší technické požadavky na provoz takového systému v rámci dané infrastruktury, jelikož je vyžadován pouze běh databázového serveru.

Mezi hlavní problémy spojené s dvouvrstvou architekturou patří problematika **distribuce** verzí klientské aplikace, která je instalována na uživatelských stanicích[3]. Proces distribuce přidává komplexitu a znesnadňuje rychlé nasazení nových verzí. Pro nasazení takové verze lze uvažovat několik možností, mezi které například patří distribuce skrze aplikační obchody (pokud existují na daném systému, např.: Microsoft Store pro Windows, Google Play pro Android či Apple Store na iOS) nebo balíčkovací systémy (`choco`[4] pro Windows nebo `apt` pro Debian). Výše zmíněné nástroje/služby podporují privátní repozitáře aplikací, tudíž je možné tímto způsobem distribuovat i neveřejné dvouvrstvé aplikace. Aplikace lze taktéž distribuovat pomocí externího aktualizačního nástroje, který zajistí kontrolu nové verze a její instalaci, tímto způsobem probíhá aktualizace u některých ERP[5] systémů, například Helios Green[6]. V poslední řadě aplikace také může sama obsloužit stažení nové verze a vyvolání systémového dialogu instalace, příkladem může být knihovna pro Android *AppUpdater*[7].

Zajištění **bezpečnosti** je dalším problémem, kterému je třeba čelit. V případě komunikace se serverem pomocí SQL se uživatel klientské aplikace skrze tuto aplikaci přihlašuje do databáze, tudíž je nutné řídit uživatelské účty a jejich role/oprávnění přímo na straně DBMS. Fakt, že přihlášení probíhá přímo do databáze znamená, že uživatelé se svými přihlašovacími údaji mohou přistupovat do databáze i pomocí jiných nástrojů (MS Access apod.), které mohou narušit integritu dat, jelikož tyto nástroje neobsahují případnou aplikační logiku sloužící k zajištění integrity dat, která je implementována v klientské aplikaci. Dnešní RDBMS umožňují napojení uživatelských účtů na doménové kontrolery (Active Di-

[3] Tato problematika je aplikovatelná i v případě vícevrstvých architektur, a to tehdy, kdy klientská aplikace komunikuje s aplikačním serverem, a ne s databází. V případě dvouvrstvé arch. je však tento problém nevyhnutelný.

[4] https://chocolatey.org/

[5] Enterprise Resource Planning (ERP) – systém pro plánování podnikových zdrojů

[6] Zdroj: vlastní observace aktualizace systému.

[7] https://github.com/javiersantos/AppUpdater

rectory apod.), tudíž účet do Oracle Database[8] či MSSQL je taktéž možné autentizovat vůči doméně. Dalším aspektem bezpečnosti je riziko, že potenciální útočníci mají k dispozici celou klientskou aplikaci a mohou s pomocí dekompilace aplikace odhalit bezpečnostně citlivé informace (pevné adresy, hesla apod.), které mohou využít při dalších útocích. Možným opatřením je obfuskace kompilovaného kódu přejmenováním proměnných, tříd či úpravou toku prováděného kódu. Obfuskace nezabrání získání potenciálně citlivých informací, pouze ztíží jejich získání.

Škálovatelnost celého řešení je limitována na databázový server, který kromě správy dat vykonává také některé business operace, které mohou dále zatěžovat server. Jelikož klienti se přímo pojí do databáze, nejsou zde žádné cachovací vrstvy, které by snížili zátěž serveru alespoň o `read` operace. Škálování lze obecně dělit na dva typy: vertikální a horizontální [4]. Vertikálním škálováním se rozumí takové škálování, kde prostředky serveru (CPU, RAM apod.) jsou dostatečně navýšeny. Jelikož výkon jednoho CPU (či kombinace několika v multi-CPU serverových platformách) je omezený, hovoří se taktéž o škálování horizontálním. Tohoto škálování je dosaženo rozložením vykonávané činnosti napříč vícero servery. Zásadní výhodou je možnost dosažení větší prostupnosti systému, jelikož počet serverů nemusí být striktně omezen. Nevýhodou v případě databázových řešení je potřeba zajištění synchronizace/replikace mezi danými servery. V takových distribuovaných systémech (DDBMS) potom namísto o ACID[9] vlastnostech hovoříme o BASE[10] – eventuální konzistenci. Této konzistence je dosaženo v momentě, kdy všechny uzly *konvergují* k jednomu navzájem konzistentnímu stavu. V době, než je dosaženo konzistence v rámci distribuovaného systému, může nastat *replikační lag*. Tento lag (zpoždění) vzniká, když klient zapisuje, přičemž požadavek zápisu obslouží uzel A a následně požadavek na zpětné čtení obslouží uzel B, který ještě nedostal zprávu z uzlu A o změně [30].

Replikaci databází je pak možné řešit synchronizováním vícero databázových serverů mezi sebou. Typicky se primární server označuje jako *Publisher* a podřízené instance jako *Subscriber* [39]. V závislosti na konkretní implementaci mohou i subscribeři propagovat změny na publishera. Alternativou replikace je pak *sharding*, tedy tříštění, kdy části databáze až na úroveň tabulek jsou rozmístěny napříč vícero servery. Jedna část dat – logický shard – může být umístěn napříč vícero *fyzickými shardy* – servery/uzly. Data mohou být rozdělena horizontálně (oddělením řádků) či vertikálně (rozříznutím přes sloupce) [11].

Jak bylo zmíněno výše, ve dvouvrstvé architektuře lze efektivně škálovat pouze serverovou vrstvu. Klienti typicky využívají různě výkoné prostředky a náhrada těchto výpočetních prostředků při změně požadavků na výkonost v podnikovém prostředí znamená nemalé finanční náklady.

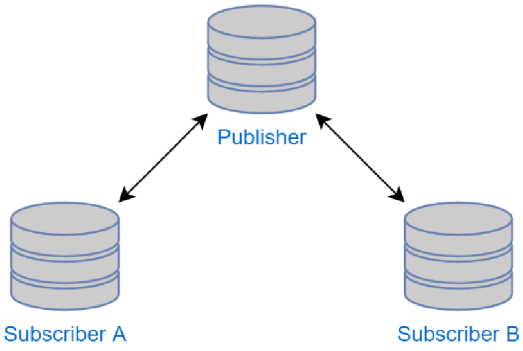
Jedním z dalších úskalí dvouvrstvých aplikací je jejich **odezva** přes rychlostně omezená vysoko-latenční prostředí. Typickým příkladem takového prostředí je používání aplikace přes VPN[11] při práci z domova či tunelování napříč vícero geografickými lokalitami. Problematika latence mezi vrstvou, kde se vykonává business logika a tou, kde jsou uložena data, se týká obecně jakékoliv více-vrstvé architektury. U dvouvrstvé architektury (na rozdíl od třívrstvé) nelze kontrolovat blízkost klienta k serveru, jelikož klientská aplikace je typicky přímo instalována na uživatelské stanici. Aplikace pracující s databází v rámci jedné business operace mohou provádět několik SQL dotazů, přičemž každý tento dotaz je zatížen

[8] https://docs.oracle.com/database/121/NTQRF/active_dir.htm#NTQRF270

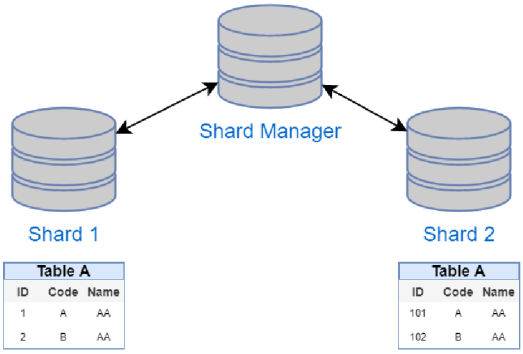
[9] A – Atomicita, C – konzistence, I – izolace, D – trvalost

[10] BA – basically available/základně dostupné, S – soft-state, nejistota stavu, z důvodu konvergence napříč distribuovaným systémem, E – eventually consistent/eventuálně konzistentní

[11] Virtual Private Network (VPN) – propojení prívátních sítí pře internet



(a) Model replikace, kdy subscribeři mohou propagovat data publisherovi



(b) Model shardingu s horizontálním dělením

Obrázek 2.2: Porovnání modelů škálování databázových serverů, inspirováno [11]

délkou *round-trip time* (RTT)[12] Autor práce definuje délku vykonání požadavku v rámci dvouvrstvé architektury následovně:

$$t = t_{client} + query\_count * (RTT + t_{server})$$

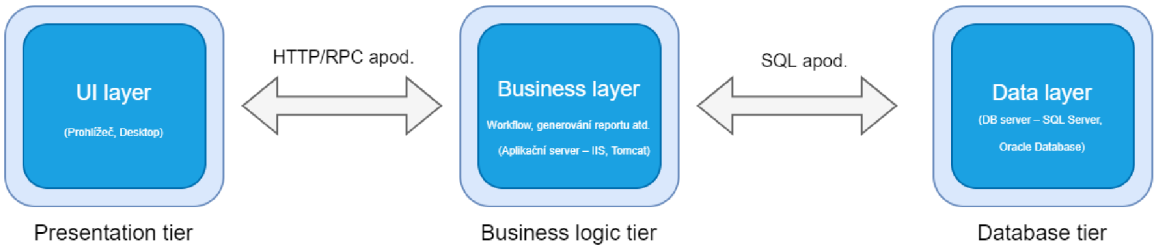
Pro ilustraci rozdílu odezvy dvouvrstvé aplikace uvažujme prostředí *local* (klient i server jsou v rámci jedné LAN), kde $RTT_{local} = 1\,\text{ms}$ a prostředí *remote* (např. klient je připojen přes VPN tunel k serveru), kde $RTT_{remote} = 50\,\text{ms}$. Dále uvažujme, že počet dotazů na server $query\_count = 20$ a dobu vykonání klientské logiky $t_{client} = 10\,\text{ms}$ a na straně serveru $t_{server} = 5\,\text{ms}$. Po aplikaci vzorce dostáváme $t_{local} = 130\,\text{ms}$ a $t_{remote} = 1110\,\text{ms}$. Z času t_{local} připadá 110 ms na zpracování dat (t_{client} a t_{server}), 20 ms pak na síťové zpoždění, tedy cca 15 % času, u t_{remote} je tato hodnota cca 90 %. V případě, že většina času je tvořena RTT, lze konstatovat, že latence je zásadním faktorem v degradaci výkonu aplikace [48]. Situace, kdy latence není problémem, může být například tehdy, kdy vypočtený celkový čas t_{remote} je zásadně menší než reálná doba vykonání požadavku. V takových případech lze uvažovat, že problémem může být nedostatečná prostupnost sítě, která prodlužuje posílání požadavku či stahování odpovědi.

Třívrstvá architektura

Třívrstvá architektura je architektura, která mapuje jednotlivé části – layers – aplikace do samotných vrstev - tierů. Typicky se jedná o webové aplikace, kde prezentační část běží ve webovém prohlížeči, business logika je provozována na aplikačním serveru a data jsou uložena na databázovém serveru. Alternativně prezentační část může být reprezentována desktopovou či jinou aplikací.

Separace jednotlivých částí umožňuje oddělit vývojářské technologie a týmy [18] s cílem snížení vzájemné provázanosti na nezbytné minimum. Tato architektura dovoluje nahradit či modernizovat prezentační vrstvu bez nutnosti například reimplemenovat kompletní business logiku aplikace. Rozdělení taktéž vynucuje lepší oddělení na úrovni kódu, kde lze snadněji udržovat princip jedné zodpovědnosti (single-responsibility principle, viz Martinova SOLID pravidla [32] pro psaní kódu). Separace na druhou stranu ovšem může přinášet duplikaci aplikační logiky, například validace dat. Příkladem může být webová třívrstvá aplikace, kde

[12]RTT je doba, která trvá signálu z bodu A doputovat do bodu B a zpět [9].



Obrázek 2.3: Ukázka rozložení layers mezi tiers – vrstvy – ve třívrstvé architektuře, inspirováno a doplněno [5]

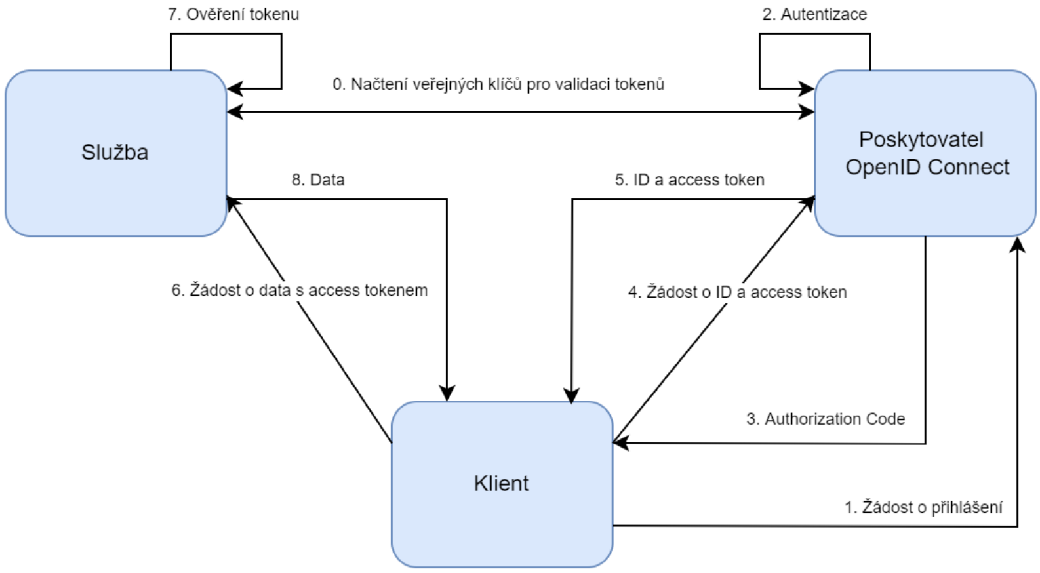
je vhodné provádět validaci dat na úrovni prohlížeče – prezentační vrstvy – pomocí jazyka JavaScript, dále pak validace na straně aplikačního serveru, jelikož s validátory na straně prohlížeče může být manipulováno či mohou být odstraněny. V poslední řadě je typicky validace na straně databáze, kde kromě standardních prostředků referenční integrity mohou být přítomna další omezení.

Odsunutí business logiky do aplikační vrstvy přináší výše zmíněné výhody, ovšem také i určité nevýhody, jmenovitě problematika přístupu alternativních nástrojů přímo k datovým zdrojům. Z důvodů historických, businessových či návrhových nelze vždy využít aplikační server pro napojení externích nástrojů, proto dochází k fenoménu, kdy nástroje jsou propojovány na úrovni databází[13]. Obejití části business logiky, která je umístěna na aplikačním serveru, může vyústit v datovou nekonzistenci či poruchu aplikace způsobenou externím přístupem do databáze. Je proto tedy nutné buď duplikovat nezbytnou business logiku na úrovni databáze, nebo vynutit napojení na úrovni aplikačních serverů. Pro takové napojení je nutné poskytnou odpovídají rozhraní, které bude schopno plně nahradit rozsah přístupu předcházejících externích nástrojů. Příkladem takového rozhraní může být vystavení odpovídajících datových zdrojů přes REST rozhraní.

Z hlediska bezpečnosti třívrstvé aplikace nabízejí komplexnější řešení a větší možnosti v **autentizaci** a autorizaci uživatelů. U dvouvrstvých aplikací je tato problematika řešena na úrovni DBMS, kde záleží na možnostech proprietární integrace s různými LDAP službami či jinými externími způsoby autentizace. Třívrstvá architektura v tomto typicky dovoluje integrovat jakékoliv externí poskytovatele autentizace či autorizace (např. OpenID Connect[14]). V českém prostředí se pak lze setkat s tzv. *Bankovní identitou*[15], která slouží jako prostředek autentizace skrze bankovní přihlášení do portálů veřejné správy. Typickým příkladem využití OpenID Connect je přihlášení skrze Google, Facebook či Apple do aplikací třetích stran. Mezi česká komerční řešení se řadí taktéž *mojeID*[16], které právě poskytuje služby autentizace na protokolu OpenID Connect [40]. Příkladem využití těchto protokolů je například přihlášení a získání informací o uživateli při registraci do systému postaveném na tří a více-vrstvé architektuře.

Problematika **distribuce** třívrstvých aplikací a její složitost závisí na zvolené platformě prezentační vrstvy. V případě webových třívrstvých aplikací je distribuce realizována načtením konkrétní webové stránky či obnovou cachovaných scriptů v paměti prohlížeče. U nativních aplikací jsou možnosti a řešení distribuce identické s dvouvrstvými aplikacemi.

[13] Zdroj: vlastní observace.
[14] https://openid.net/connect/
[15] https://bankovni-identita.cz/
[16] https://www.mojeid.cz/



Obrázek 2.4: Schéma protokolu OpenID Connect, inspirováno [40]

Škálování třívrstvých aplikací na rozdíl od dvouvrstvých dovoluje horizontálně škálovat provádění business logiky aplikace, která je typicky umístěna v rámci třívrstvé architektury na aplikačním serveru, který může být škálován vertikálně (podobně jako klient u dvouvrstvé architektury), ale také i horizontálně. Konkrétní podoba i možnosti škálování jsou však závislé na specifických vlastnostech aplikace a prostředí.

Na rozdíl od dvouvrstvých aplikací, tři a více-vrstvé aplikace typicky netrpí na zhoršení odezvy systému v případě vysoko-latenčního spojení mezi klientem a aplikačním serverem. Toto je dáno tím, že jeden požadavek klienta, který je zatížený touto latencí, realizuje komplexnější operaci. Taková komplexní operace se pak může skládat z vícero databázových volání, které už nejsou zatíženy vysokou latencí, přinejmenším je tato latence identická pro všechny klienty.

Další n-vrstvé architektury

Kromě zmíněné dvouvrstvé a třívrstvé architektury lze obecně hovořit o n-vrstvých architekturách. Další vrstvy mohou být tvořeny dodatečnými servery s business logikou, službami či prostředky vyvažování výkonu (load balancery) nebo skupinou serverů sloužící pro rychlejší doručení obsahu (Content Delivery Network – CDN).

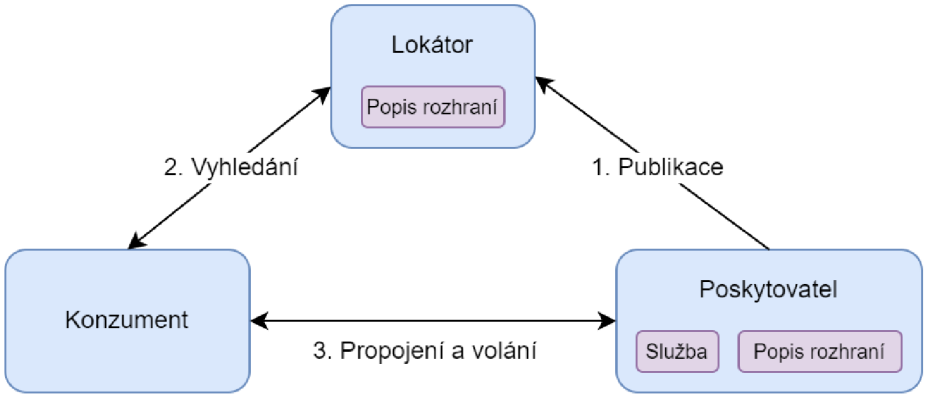
2.1.2 Service Oriented Architecture

Service Oriented Architecture je přístup k návrhu systému pomocí skládání a propojování separovaných služeb do většího celku [22]. Mezi základní principy SOA se dle [13] řadí: samostatnost, modularita, volné propojení (*loose coupling*) a interoperabilita. Cílem je u jednotlivých služeb maximalizovat znovupoužitelnost a nahraditelnost, což následně může umožnit lepší testovatelnost (zavedením některých fiktivních poskytovatelů služeb), lepší rozdělení činností mezi jednotlivé týmy vývojářů, a také potenciální diverzifikaci technologií napříč službami. V rámci SOA lze identifikovat tyto entity [13]:

- **služby** – logické entity systému definované rozhraním

- **poskytovatel služeb** (*service provider*) – konkrétní softwarová entita implementující rozhraní služby
- **konzument služby** (*service consumer*) – softwarová entita (klient) může být uživatelská aplikace či jiná služba
- **lokátor služeb** (*service locator*) – poskytovatel služby, která uchovává registr poskytovatelů služeb a jejich umístění
- **zprostředkovatel služeb** (*service broker*) – poskytovatel služby, která předává požadavky jedné či více službám

V klasickém pohledu na SOA služby existují v jediné instanci na rozdíl od mikroslužeb, u kterých je běžné provozování vícero instancí paralelně, viz kapitola 2.1.3.



Obrázek 2.5: Diagram komunikace publikace, vyhledání a volání služby, inspirováno [13]

Web Services

V konkrétní podobě *webové služby* (web services) jsou takové služby, které jako transportní protokol využívají HTTP/HTTPS, kde jsou informace přenášeny pomocí XML či JSON. W3C Web Service Working Group definuje webové služby takto [50]:

> Webová služba je softwarový systém navrhnutý pro interoperabilní síťovou komunikaci mezi stroji. Taková webová služba má své rozhraní popsané strojově čitelným formátem (WSDL). Ostatní systémy interagují s takovou webovou službou pomocí SOAP zpráv, které jsou typicky doručeny prostřednictvím HTTP protokolu a jsou serializované pomocí XML.

Web Service Description Language (WSDL) je jazyk založený na XML sloužící pro popis rozhraní poskytované webovou službou. V rámci WSDL souboru lze definovat služby, endpointy a jejich *binding* na HTTP či SOAP.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <definitions xmlns="http://www.w3.org/ns/wsdl"
3              xmlns:tns="http://www.example.com/thesis-sample"
4              xmlns:whttp="http://schemas.xmlsoap.org/wsdl/http/"
5              xmlns:wsoap="http://schemas.xmlsoap.org/wsdl/soap/"
6              targetNamespace="http://www.example.com/thesis-sample">
7
```

```
<!-- Abstract type -->
   <types>
      <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  xmlns="http://www.example.com/thesis-sample"
                  targetNamespace="http://www.example.com/wsdl20sample">

         <xsd:element name="request"> ... </xsd:element>
         <xsd:element name="response"> ... </xsd:element>
      </xsd:schema>
   </types>

<!-- Abstract interfaces -->
   <interface name="StockItem">
      <fault name="ErrorDescription" element="tns:response"/>
      <operation name="GetStockPrice" pattern="http://www.w3.org/ns/wsdl/in-out">
         <input messageLabel="In" element="tns:request"/>
         <output messageLabel="Out" element="tns:response"/>
      </operation>
   </interface>

<!-- Concrete Binding Over HTTP -->
   <binding name="HttpBinding" interface="tns:StockItem"
            type="http://www.w3.org/ns/wsdl/http">
      <operation ref="tns:GetStockPrice" whttp:method="GET"/>
   </binding>

<!-- Concrete Binding with SOAP-->
   <binding name="SoapBinding" interface="tns:StockItem"
            type="http://www.w3.org/ns/wsdl/soap"
            wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/"
            wsoap:mepDefault="http://www.w3.org/2003/05/soap/mep/request-response">
      <operation ref="tns:GetStockPrice" />
   </binding>

<!-- Web Service offering endpoints for both bindings-->
   <service name="ServiceOne" interface="tns:StockItem">
      <endpoint name="HttpEndpoint"
                binding="tns:HttpBinding"
                address="http://www.example.com/rest/"/>
      <endpoint name="SoapEndpoint"
                binding="tns:SoapBinding"
                address="http://www.example.com/soap/"/>
   </service>
</definitions>
```

Výpis 2.1: Ukázka WSDL souboru

Simple Object Access Protocol (SOAP) je protokol pro zasílání strukturovaných informací mezi webovými službami. Využívá principu *remote procedure calling* (RPC) pro volání konkrétních metod poskytnutých rozhraním popsaným prostřednictvím WSDL. Jako formát zprávy je využito XML.

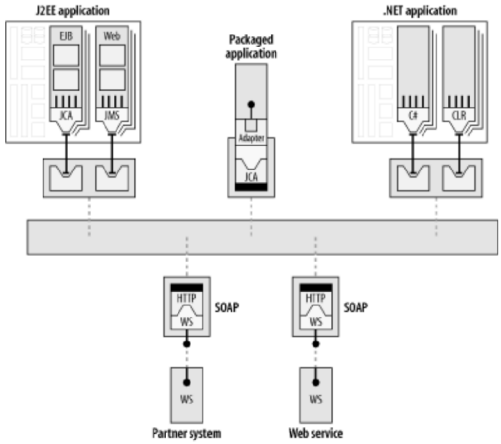
```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope" xmlns:m="http://www.
    example.com">
  <soap:Header>
  </soap:Header>
  <soap:Body>
    <m:GetStockPrice>
      ...
```

```
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```

Výpis 2.2: Ukázka volání SOAP

Enterprise Service Bus

Enterprise service bus (ESB) je architektonický návrhový vzor reprezentovaný sběrnicí vzájemně propojující decentralizovanou sadu služeb/aplikací. ESB nalézá využití při procesu známém jako *enterprise application integration* (EAI), tedy integrace podnikových aplikací. V prostředí podnikových informačních systémů je typicky nutné spolu integrovat systémy od různých dodavatelů postavených mnohdy na zcela odlišných technologiích. Pro realizaci této integrace se pak využívá konceptu ESB, respektive konkrétní implementace, jakými jsou například Oracle Service Bus[17] nebo Azure Service Bus[18].



Obrázek 2.6: Příklad realizace ESB, převzato z [8]

Chappell [8] jmenuje několik charakteristik ESB:

- **adaptovatelnost** – ESB může naplňovat požadavky na integraci v rámci jedné i napříč více organizacemi
- **distribuovaná SOA založená na událostech** – služby mohou být geograficky oddělené, přitom ale propojené a dostupné na sběrnici
- **oddělení a škálovatelnost** – jednotlivé komponenty/adaptéry do ESB mohou být odděleně nasazovány a škálovány

ESB tedy tvoří páteř daného řešení, na kterou se napojují nejrůznější aplikace a služby. Napojení na samotné ESB je pak realizováno prostřednictvím *ESB kontejnerů*. Kontejner v kontextu ESB je izolované prostředí, které poskytuje implementaci rozhraní pro komunikaci s ESB. V rámci jednoho kontejneru obecně může fungovat vícero služeb/aplikací.

ESB se snaží řešit problematiku těsného provázání služeb, která může postihovat ad-hoc architektury. Těsné provázání v kontextu ESB znamená, že v nejhorším případě všechny

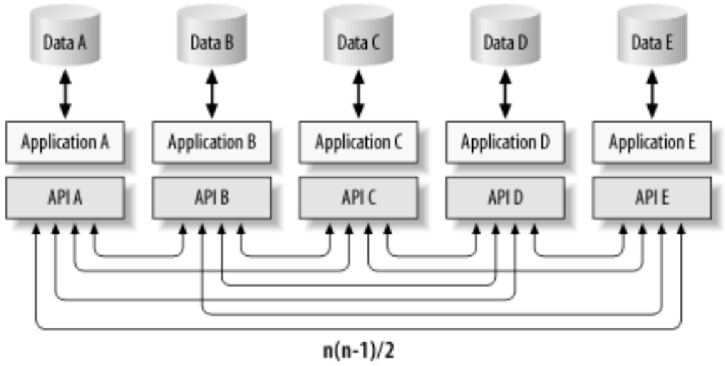
[17] https://www.oracle.com/middleware/technologies/service-bus.html
[18] https://azure.microsoft.com/en-us/services/service-bus/

služby spolu komunikují vzájemně. Celkový počet rozhraní v takovém systému lze určit na základě této formule [8, str. 117]:

$$n(n-1)/2$$

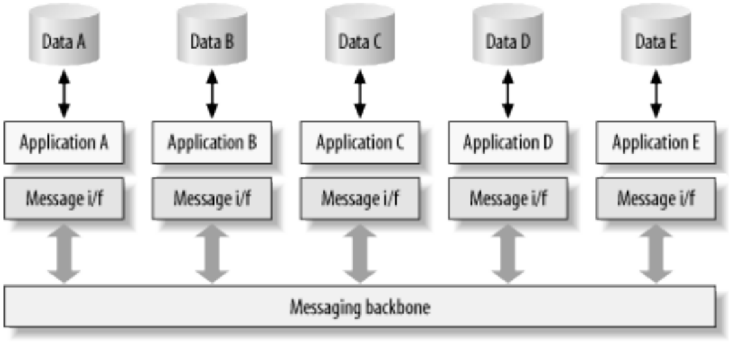
kde n je počet aplikací/služeb.

Počet rozhraní přímo souvisí s náročností správy takového systému. Chyby a selhání napříč systémem mohou být hůře dohledatelné a systém k nim celkově může být náchylnější z pohledu celkového množství rozhraní a nutnosti zachování jejich kompatibility. Změny v takovém prostředí mohou být značně komplikované, jelikož změny na rozhraní mohou zasáhnout značné množství navázaných služeb.



Obrázek 2.7: Schéma vzájemné komunikace při absenci ESB, převzato z [8]

ESB přináší pro tyto případy řešení v podobě konceptu *message oriented middleware* (MOM), který abstrahuje komunikaci mezi službami do jednotného doručovatele zpráv. Komunikace tedy probíhá vždy buď mezi odesílatelem a ESB, nebo ESB a příjemcem. Počet rozhraní v takovém systému je potom n, kde n odpovídá počtu služeb. V rámci MOM komunikace pak lze hovořit o komunikaci typu *point-to-point* (jeden odesílatel, jeden příjemce) a *publish-subscribe* (jeden odesílatel, více dynamických příjemců). Komunikace je asynchronní a spolehlivá, tedy zprávy jsou perzistentně ukládány mechanismem *store-and-forward* [8, str. 123].



Obrázek 2.8: Schéma komunikace přes message oriented middleware, převzato z [8]

2.1.3 Mikroslužby

Microservices – mikroslužby – je architektura, která (stejně jako SOA) je založena na službách. Na první pohled je základním rozdílem velikost konkrétních služeb, ovšem pohledem druhým se mikroslužby odlišují i v celkovém přístupu k dělení a implementaci služeb.

Vznik mikroslužbového přístupu vychází z reálných potřeb [41, str. 9]. Na rozdíl od SOA se tento přístup snaží o nastínění vhodné velikosti služeb, korektnějšího dělení a taktéž o abstrakci od konkrétních komunikačních protokolů a technologií, viz SOAP a WSDL u SOA. I přestože se mikroslužby typicky implementují s využitím HTTP/REST, celková koncepce není na tyto technologie nijak vázána.

Newman jmenuje několik klíčových výhod [41, str. 19–27] mikroslužeb. Mezi tyto výhody řadí: technologickou heterogenitu, odolnost proti poruchám, škálování, jednoduchost nasazení a nahraditelnost.

Mezi hlavní výhody mikroslužeb se řadí jejich **odolnost proti poruchám** celého systému. V případě, že jedna služba selže (a její selhání nemá za následek kaskádovitý kolaps systému), zbytek systému může nadále fungovat ačkoliv jeho funkcionalita/výkon bude degradována. Porucha u monolitického systému ovlivní všechny jeho součásti a může vést k celkovému kolapsu systému. Vyšší množství procesů/komponent v rámci mikroslužeb jednak, jak již bylo zmíněno, zvyšuje komplexitu, ale taktéž i náchylnost k poruchám, je tedy vhodné při návrhu architektury tyto aspekty zvážit a zdržet přespříliš komplexních modelů interakce služeb.

Mikroslužby umožňují volit **odlišné technologie** pro implementaci služeb. Technologie tedy může být zvolena na základě specifických požadavků na službu a může se lišit od převládajících technologických řešení u jiných služeb. Příkladem může být volba lightweight frameworku (např. Flask[19]) oproti robustnímu (např. .NET či Spring) pro implementaci jednoduchého REST API. Při volbě odlišné technologie je ale také nutné brát v potaz i udržovatelnost, zvlášť v případech menších vývojářských týmů, kde ne všichni členové mohou být znalí daných technologií. Přílišná *technologická heterogenita* pak může vést k přílišným nárokům na případné nové členy týmu, pokud na vývojáře připadá více jak jedna služba u konkrétního projektu.

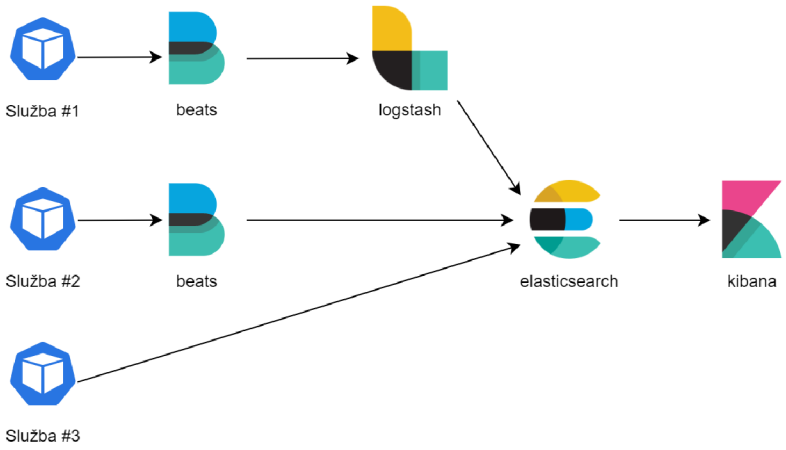
Velikost mikroslužeb taktéž podporuje koncept **nahraditelnosti**. Služba může být snadno nahrazena (reimplementována), jelikož její rozhraní je jasně specifikované a oddělené od ostatních služeb systému. Rozhodnutí o nahrazení služby může vycházet ze zásadní změny funkčních požadavků, které nedovolují využití stávající implementace či technologické zastaralosti.

Škálování je jednou z hlavní předností mikroslužeb [41, str. 23]. Rozdělením systému na mikroslužby je možné *on-demand* horizontálně škálovat jednotlivé služby. Služba tedy pracuje ve více instancích na jednom či více strojích. Více instancí jedné služby znemožňuje použití IP adresy pro přístupu ke službě a je potřeba pokročilejších nástrojů, jakým je například orchestrátor Kubernetes[20]. Tato platforma z hlediska škálování poskytuje *load-balancing* (automatická dělba zátěže mezi více uzlů) a také *service discovery* (dohledávání lokace služeb)[21]. Kuberenetes výše zmíněný problém řeší pomocí DNS záznamu pro služby, při jehož překladu Kubernetes automaticky provádí load-balancing, zároveň tak řeší i service discovery, jelikož je služba pojmenována.

[19] https://flask.palletsprojects.com/en/2.0.x/
[20] https://kubernetes.io/
[21] https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/

Mikroslužby taktéž mohou umožnit **snazší nasazení** změn do produkčního prostředí za předpokladu, že změna je izolovaná v rámci dané služby. Změna jedné služby na rozdíl od přenasazení celého monolitického systému nemá typicky takové dopady, tudíž nenese vysoká rizika [41, str. 24], což podporuje častější distribuce a flexibilnější vývoj. Z tohoto důvodu je právě u architektury mikroslužeb typická přítomnost automatizace procesu sestavení, integrace a nasazení, tedy *continuous integration* a *continuous deployment*. Jednotlivé služby mohou být nezávisle na sobě sestaveny, otestovány a posléze nasazeny například do produkčního Kuberenetes clusteru.

Architektura mikroslužeb dává vyšší nároky na sledování stavu, logování a diagnostiku problémů (či předcházení jejich vzniku) [41, str. 310]. Všechny aplikace/běhová prostředí typicky produkují nějakou formu logů, kterou je vhodné zaznamenávat a uchovávat. U monolitických aplikací lze, dle názoru autora, povětšinou vystačit s logováním do systémových událostí, souboru či databáze, ovšem v případě mikroslužeb by takový přístup znamenal přílišnou fragmentaci logů mezi mnoha místy, a tudíž značně stíženou diagnostiku problémů, nehledě na nemožnost centrálně monitorovat zdraví daných služeb.



Obrázek 2.9: Schéma kolekce, zpracování, uložení a vizualizace dat v Elastic stacku, inspirováno [12]

Pro tyto účely existuje mnoho řešení, mezi nimi například tzv. **Elastic stack**[22] – kombinace několika technologií pro účely sběru, zpracování, ukládaní a vizualizace dat. Elastic pro sběr dat ze služeb může využívat tzv. *beats* – sběrače dat pro Elasticsearch/Logstash. Příkladem beatů může být *filebeat* pro streaming souborových logů, *winlogbeat* pro odesílání Windows Event Logů či *metricbeat* pro monitorování systémových prostředků. Další součástí stacku je *Logstash*, což je nástroj pro zpracování dat, který poskytuje možnosti transformace a filtrace dat. Centrální součástí stacku je pak *Elasticsearch*, který plní roli databáze, vyhledávače a také poskytuje nástroje datové analýzy. Nástroj *Kibana* vizualizuje data uložená v Elasticsearch a poskytuje dashboardové uživatelské rozhraní s možností hledání a datové analýzy.

Velikost služeb

Pro určení funkcionality, která může být oddělena do mikroslužby, lze použít *principu jedné zodpovědnosti* (*single responsibility principle* – SRP, autorem je Robert C. Martin). Tento

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

princip se běžně aplikuje na kód, ale v tomto případě jej lze vztáhnout i na separaci služeb. Jelikož cílem služby v rámci ekosystému mikroslužeb by mělo být samostatnou, izolovanou a nahraditelnou komponentou, tak lze předpokládat, že taková funkcionalita by měla plnit unikátní zodpovědnost [41, str. 17].

Správnou velikost lze obecně těžko stanovit bez znalosti konkrétních podmínek a vlastností prostředí. Společnost Amazon v rámci svých týmů identifikuje maximální velikost služby dle tzv. *two pizza* pravidla, tzn. celý vývojářský tým služby by se měl být schopen najíst ze dvou pizz, tedy cca tucet vývojářů [31]. Alternativou je přistoupit k této problematice komplexněji a užít techniky **domain-driven designu** (DDD), která se zaměřuje na hloubkové porozumění procesů a pravidel dané domény [17]. Technika vychází z knihy E. Evanse *Domain-Driven Design: Tackling Complexity in the Heart of Software* [14] Zásadním pojmem v rámci DDD z hlediska problematiky velikosti služeb je tzv. *bounded context*, tedy ohraničený kontext. U rozsáhlých systémů/organizací může být problematické vytvořit jeden unifikovaný model pro celý systém, a to z důvodu rozdílných pojmenování entit napříč skupinami uživatelů, specifických atributů apod. Řešením je pak rozdělení modelu do ohraničených kontextů, v rámci kterých lze uchopitelněji popsat danou poddoménu. Mezi ohraničenými kontexty se jasně definují vzájemné vztahy a mapování. Příkladem může být doména e-shopu, kde hovoříme o uživatelích v kontextu používání webové stránky e-shopu a kupujících v kontextu procesu nákupu zboží. Stejná entita v různých kontextech nabývá různých pojmenování a atributů, ale její identita je zachována. Podél hranic těchto ohraničených kontextů pak lze navrhnout dělení do daného systému do služeb.

Model komunikace

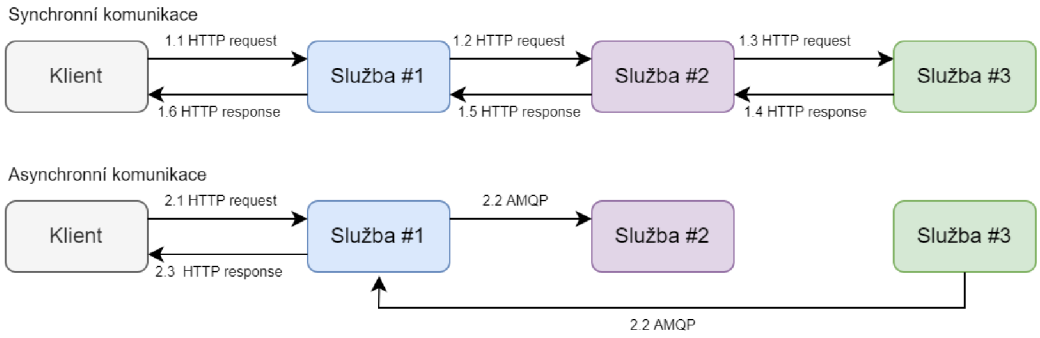
Způsob komunikace mezi službami lze obecně dělit na synchronní a asynchronní [37]. **Synchronní** model je založen na principu požadavek/odpověď, kdy klient čeká na vyřízení odpovědi serverem, kde toto vyřízení může zahrnovat i například komunikaci mezi vícero službami. Komunikace se označuje jako synchronní i tehdy, pokud kód vykonání požadavku je neblokující, tedy i když je metoda (v terminologii jazyků C# nebo JavaScript) označena `async`.

Synchronní komunikace je citlivá na dostupnost všech služeb zúčastněných dané komunikace. Při obsluze požadavku, který vyžaduje součinnost většího množství služeb (zanořených volání služeb) a případné poruše (nedostupnost, chyba) na jedné či více služeb, může dojít ke kaskádovité propagaci napříč službami [37]. V případě, že služba je nedostupná, proces obsluhy požadavku čeká na vypršení *timeoutu* u volaných služeb, což značně prodlužuje dobu vyřízení požadavku. Za účelem zkrácení doby vyřízení požadavku v této situaci se používá technika *circuit breakers*, jejíž účelem je zajistit tzv. *fail fast*, tedy rychlé selhání. Pokud vyřízení několika po sobě jdoucích požadavků na vzdálenou službu selže, je aplikován circuit breaker, který namísto čekání a následného selhání, selže okamžitě. Po přednastavené době bude dovoleno vykonání několika testovacích požadavků. Pokud jsou požadavky úspěšné, normální fungování volání služby je obnoveno, pokud tyto požadavky opět selžou, znovu se aplikuje timeout. Rychlým selháním je možné snížit celkovou zátěž systému způsobenou blokací vláken čekajících na odpověď [42]. Nástroj resilience4j[23] je příkladem implementace circuit breakers. Technika circuit breakerů je jednou z mnoha při návrhů systémů odolným vůči poruchám (*fault-tolerant systems*), jejichž jedním z aspektů je i tzv. *graceful degradation*. Tato vlastnost umožňuje, že v případě poruchy v rámci systému bude funkcionalita degradována kontrolovaným způsobem, mluvíme potom o tzv. *degrado-*

[23] https://github.com/resilience4j/resilience4j

vané funkcionalitě. Příkladem může být například nefunkčnost tvorby reportů v podnikovém systému následkem poruchy reportovací služby.

Za **asynchronní** model komunikace se primárně[24] považuje tzv. *event-based* komunikace. Tedy taková komunikace, u které se povětšinou nečeká na odpověď, avšak pokud se na ni čeká, tak se kód, který se vykoná po odpovědi, označuje jako *callback* [37]. Asynchronní komunikace může používat protokol AMQP[25]. V rámci asynchronní komunikace se typicky používá i tzv. *message broker*, tedy zprostředkovatel zpráv. Tato mezivrstva, *middleware*, plní funkci fronty a doručovatele zpráv. Pro čtení určitého typu zprávy od *publishera* se může přihlásit více *consumerů*. Doručení zprávy N consumerům by bez tohoto prostředníka znamenalo zaslání N zpráv od publishera namísto jedné. Dále také tito consumeři se mohou dynamicky přihlašovat k odběru těchto zpráv. Přítomnost prostředníka dále také znamená, že v případě, že služba consumera nebude dostupná, může být služba při svém oživení vyrozuměna o všech proběhlých událostech/doručených zprávách [41, str. 55]. Příkladem těchto message brokerů je například RabbitMQ[26] nebo Apache Kafka[27].



Obrázek 2.10: Schéma komunikace synchronní a asynchronní, inspirováno [47]

Problematika databází

Při návrhu mikroslužeb je nutné řešit otázku **(ne)sdílení databáze** napříč službami. Mimo případů, kde business domény daných služeb jsou zcela disjunktní, existuje mnoho případů, kde se domény vážou přes sdílené entity. Konkrétním příkladem pak může být vazba mezi entitami `Customer` a `Order` v modelovém e-shopu, kde každá tato entita může být udržována jinou službou, tedy `CustomerService` a `OrderService`. V případě, že by tyto entity sídlily v jedné databázi, budou zřejmě propojeny vazbou cizího klíče, například `Order` bude obsahovat atribut `CustomerId`, který bude reprezentovat vazbu mezi těmito entitami.

Možným přístupem k ukládaní dat je tyto data oddělit a ukládat v oddělených databázích/schématech/tabulkách. Tento přístup se označuje jako *database-per-service* [43]. Podstatou je, že data jsou od sebe zcela oddělena a zásah do jedné služby nemá dopady na fungování druhé. Oddělením je taktéž možné vybrat databázovou technologii, která nejlépe odpovídá struktuře dat a požadavkům na prohledávání, např. volba Elasticsearch pro datové analýzy nebo NOSQL databáze pro nestrukturovaná data. Zásadní nevýhodou pak je prohledávání a dotazování či komplexita realizace business scénářů napříč vícero databázemi [43]. Při takových scénářích je nutné *orchestrovat* proces realizace daných procesů.

[24] Požadavek/odpověď lze taktéž použít v asynchronní komunikaci [41, str. 89].
[25] `https://www.amqp.org/`
[26] `https://www.rabbitmq.com/`
[27] `https://kafka.apache.org/`

Jelikož se jedná o operace nad více databázemi, nelze zde zajistit ACID vlastnosti. Namísto toho zde mluvíme o BASE[28], kde se primárně aplikuje tzv. *soft state* a *eventuální konzistence*. Jelikož systém se dočasně může nacházet v nekonzistentním stavu (například při čekání na potvrzení operace z jiné služby), hovoříme o soft stavu systému, tedy stavu, který se v čase mění tak, aby dosáhnul eventuální konzistence (po potvrzení operace z jiné služby). Brewer [7] pro takovéto systémy s distribuovaným datovým skladem definuje tzv. *CAP teorém*, který tvrdí, že pro takový datový sklad není možné zajistit více dvě z těchto vlastností [19]:

- **konzistence** (*consistency*) – musí existovat lineární uspořádání všech operací takové, u kterého každá operace byla vykonána tak, že mohla být vykonána za nulový čas
- **dostupnost** (*availability*) – pro distribuovaný kontinuálně dostupný systém musí platit, že každý požadavek z funkční služby musí vyústit v odpověď
- **odolnost k přerušení** (*partition tolerance*) – systém je odolný vůči ztrátě zpráv zasílaných mezi síťovými uzly

V případě mikroslužeb s oddělenými databázemi lze typicky zajistit odolnost k přerušení v kombinaci s dostupností *nebo* konzistencí. Zajištění všech třech vlastností není možné, což bylo formálně dokázáno Gilbert a Lynch [19].

Jak již bylo zmíněno, komplexní dotazování vyžadující součinnost vícero databází může být problematické z hlediska výkonu. Pro tyto účely lze použít vzor *Command Query Responsibility Segregation* (CQRS) [16]. Jedná se o vzor, kde pro čtení (tedy i dotazování) je použit odlišný model než pro editaci dat. Model pro editaci dat (tzv. *Command model*) může být jednoduchou implementací uložení dat do databáze, zatímco model pro dotazování (tzv. *Query model*) může být implementován tak, aby optimalizoval rychlost vyhledávání. Taková implementace se může lišit například přítomností odvozených atributů, agregací primárních dat či celkovou transformací a denormalizací dat. V případě oddělených databází můžeme tento vzor aplikovat vytvořením sdílené databáze určené pouze pro účely dotazování nad daty [47, str. 32].

Alternativou pro ukládání dat je využití **sdílené databáze** napříč službami. Tento přístup zachovává ACID vlastnosti a zjednodušuje implementaci. Zásadními nevýhodami je pak úzká vazba mezi službami (změna schématu může způsobit selhání jiné služby), tudíž změna v jedné službě má vedlejší dopady do druhé. Úzké provázaní se pak může projevovat jak ve fázi vývoje, tak i za provozu, kdy operace jedné služby mohou uzamykat tabulky, které jiná služba potřebuje pro svoji činnost. Sdílení databází taktéž může znamenat zvýšené výkonnostní nároky. V takovém případě je pak nutno aplikovat techniky vertikálního i horizontálního škálování, viz kapitola 2.1.1. Druhý jmenovaný typ škálování pak s sebou přináší identickou problematiku popsanou výše, tedy úskalí distribuovaného datového skladu.

Technologické řešení

Jak již bylo zmíněno, koncept mikroslužeb se přímo neváže na žádné konkrétní technologie, ovšem běžně ustálené řešení je využití REST rozhraní nad HTTP komunikací.

REST, tedy *Representational State Transfer*, je přístup k návrhu typicky webových rozhraní. RESTové rozhraní je takové, které splňuje následující podmínky [15]: klient-server,

[28] BASE – viz kapitola 2.1.1 – Škálovatelnost

oddělení zodpovědnosti, bezestavovost, cachování, jednotné rozhraní, vícevrstevnatost (load balancery, cache apod.). Volitelně je součástí REST i tzv. code-on-demand funkcionalita, tedy stahování spustitelného kódu ze serveru do klienta za účelem zjednodušení implementace klienta (applety, skripty apod.).

Vazba mezi REST a HTTP není vyžadovaná, ale HTTP nabízí vhodné metody[29] a mechanizmy (např. caching) pro implementaci RESTových operací [41, str. 50]. Konkrétní návrh podoby RESTového rozhraní se řídí mnoha konvencemi [33].

```
POST /api/v1/customer HTTP/1.1
HOST: example.com
Content-Type: application/json
Accept: application/json

{
    "name": "Michal Oberreiter",
    "email": "xoberr00@fit.vutbr.cz",
    "address": "Bozetechova 1/2",
    "city": "Brno",
    "countryId": 420
}
```

Výpis 2.3: Příklad volání REST rozhraní přes HTTP (vytvoření nové entity `customer`)

Při implementaci mikroslužeb je taktéž nutné zvážit do jaké míry aplikovat princip DRY – *Don't repeat yourself*, tedy omezení duplikace kódu. U monolitických aplikací obecně platí, že chceme maximalizovat znovupoužití kódu a minimalizovat prosté kopírování z důvodů zachování udržitelnosti. Deduplikace kódu umožňuje sdílet chování, rychleji přidávat nové funkcionality při zachování rozhraní apod. Newman ovšem argumentuje [41, str. 59], že přespřílišné sdílení kódu mezi službami vede k jejich provázání takovým způsobem, že je porušen *loose coupling* – jedna ze základních vlastností mikroslužeb. Těsným provázáním na úrovni kódu, např. pomocí sdílených knihoven, může dojít k problémům při interakci služeb využívající různé verze sdílených knihoven. Newman doporučuje dodržovat princip DRY na úrovni jedné mikroslužby, ale pozdržet se v jeho aplikaci napříč službami.

[29] HTTP metody, například: GET, POST, PUT, PATCH, DELETE

2.2 Software reengineering

Reengineering je proces studia, analýzy a následné transformace existujícího softwaru do nové podoby. Tento proces typicky zahrnuje fáze *reverzního engineeringu*, *transformace* a *dopředného engineeringu*. Cílem reengineeringu je realizovat nefunkční i funkční požadavky, které z různých důvodů nebylo možné naplnit prostou modifikací původního softwaru. K dosažení cíle je nutné analyzovat původní software (často s absencí dokumentace či popisu), navrhnout řešení a následně tyto řešení implementovat při zachování původní funkcionality s možným případným rozšířením určeným novými požadavky či přípravou na integraci nové funkcionality v budoucnu [44, str. 2].

2.2.1 Východiska a cíle reengineeringu

Výchozí motivací pro spuštění procesu reengineeringu může být mnoho, obvykle se jedná o zásadní problémy současného řešení softwarového systému [1]. Na systém jsou tedy kladeny *nefunkční požadavky*, které není možné současným řešením naplnit. Tyto požadavky se vlivem času a změnou prostředí (růst společnosti, změna business cílů) mohou měnit, tudíž systém je v průběhu času může přestat naplňovat. Cílem reengineeringu je odstranit nedostatky stávájícího systému. Na základě Rosenbergové [44] a Abbase et. al. [1] lze cíle shrnout do následujících bodů:

- příprava na funkční vylepšení
- zlepšení udržitelnosti
- zlepšení spolehlivosti
- zlepšení interoperability
- zlepšení výkonu/škálovatelnosti
- zlepšení testovatelnosti
- migrace na nové technologie

Typickým problémem u starších systémů může být například technologická zastaralost znemožňující naplnění požadavků uživatelů, absence softwarové podpory pro vyžadovanou platformu daného softwaru nebo konec podpory komerčních komponent/řešení využitých v současném řešení. Dalším aspektem může být z hlediska aktuálního pohledu nevhodná architektura. S nástupem webových aplikací a později cloudových řešení se aplikace navržené pro desktop s přímou komunikací do databáze stávají rychle zastaralé a překážkou pro jejich provoz v moderních virtualizovaných prostředích či prostředích využívajících vzdálený přístup. Kromě samotné zastaralosti můžou být motivací pro reengineering i vysoké náklady na údržbu současného řešení. Mezi tyto náklady může například patřit placení speciální komerční podpory pro zastaralé komponenty nebo celková obtížnost/časová náročnost udržování funkčnosti řešení. Na druhé straně nové řešení může vyžadovat značně menší náklady na údržbu, než jakého bylo původní řešení kdykoliv schopno dosáhnout.

Vylepšení výkonu či škálovatelnosti pro moderní prostředí může vést k reengineeringu. Abbas et. al. [1] shledávají, že právě proces reengineeringu dává značný prostor pro zakomponování požadavků na výkonnost na rozdíl od pozdějšího ladění, které nemusí nikdy

dosáhnout stejných výsledků, doporučují tedy proaktivní přístup k řešení výkonnostních otázek.

Zlepšení interoperability je klíčové pro systémy fungující v kontextu organizací, kde různé entity mají různé vlastníky (softwarové systémy). Typickým příkladem může být entita zaměstnance, která se vyskytuje většinou ve všech interních systémech, ale vlastníkem této entity je ERP[30] systém. V pojetí služeb je interoperabilitou myšleno schopnost služby se vzájemně propojit s jinými službami za účelem vzájemné výměny informací pomocí standardizovaných rozhraní.

Nevhodné členění kódu a jeho špatná udržovatelnost mohou být klíčovým faktorem při rozhodování o reengineeringu. Tento problém není nutně spojen s dobou vzniku daného softwarového řešení, jelikož i relativně nový systém se může stát neakceptovatelným z důvodu nespolehlivosti a/nebo neudržovatelnosti. Implementace se může stát problematickou, pokud nejsou například dodrženy zásady SOLID nebo cyklomatická složitost[31] přesahuje vhodnou mez[32]. Moderní metody a návrhové vzory mohou snížit frekvenci výskyt těchto defektů, ale konečná podoba implementace je do určité míry odrazem prostředí, ve kterém řešení vznikalo. Reengineering taktéž dává příležitost významně vylepšit testovatelnost kódu. Oddělení prezentační a business logiky, využití rozhraní [1] umožňují snáze vytvářet jednotkové i integrační testy, které následně mohou vylepšit spolehlivost systému při jeho změnách v průběhu životního cyklu.

Ve své podstatě nezahrnutí implementace nových funkcionalit v rámci engineeringu je konceptuálně podobné jako u souvisejícího pojmu *refaktorizace*. Refaktorizace na rozdíl od reengineeringu zahrnuje pouze část systému (tedy systém není nahrazen) a typicky je prováděna kontinuálně v průběhu aktivního vývoje.

2.2.2 Business aspekty reengineeringu

Na reengineering je nutné nahlížet i z hlediska businessu, jelikož se jedná o časově a technologicky náročný proces, který s sebou může nést značné investice, jejichž návratnost není bez rizika. Cílem reengineeringu je tedy zvýšit *business value*. Autor této práce definuje tyto faktory při rozhodování, zda realizovat reengineering:

- současné náklady na provoz řešení
- dostupnost zaměstnanců s potřebnými znalostmi pro údržbu stávajícího řešení
- konkurenceschopnost na trhu
- odhad nákladů na reimplementaci a následný provoz řešení
- dostupnost zaměstnanců s potřebnými znalostmi pro vývoj nového řešení
- míra nahraditelnosti současného řešení z hlediska procesu přechodu
- doba návratnosti investice (ROI)[33]
- možné nové příležitosti na trhu s novým řešením

[30] ERP – *Enterprise Resource Planning* systém
[31] Metrika určující počet lineárně nezávislých cest v *Control Flow Grafu* – graf cest průchodu programem.
[32] Vhodnost nelze přesně stanovit, Microsoft doporučuje komplexitu menší než 25 [36].
[33] ROI – *return of investment*

Jednotlivým faktorům může být přiřazena různá důležitost v závislosti na konkrétních podmínkách organizace a strategické pozici systému/produktu v rámci portfolia firmy či pro interní systémy jeho důležitost v rámci podnikových procesů. Zhodnocení vhodnosti reengineeringu je tedy komplexní úkon vyžadující analýzu jak z pohledu technického, tak i obchodního a strategického.

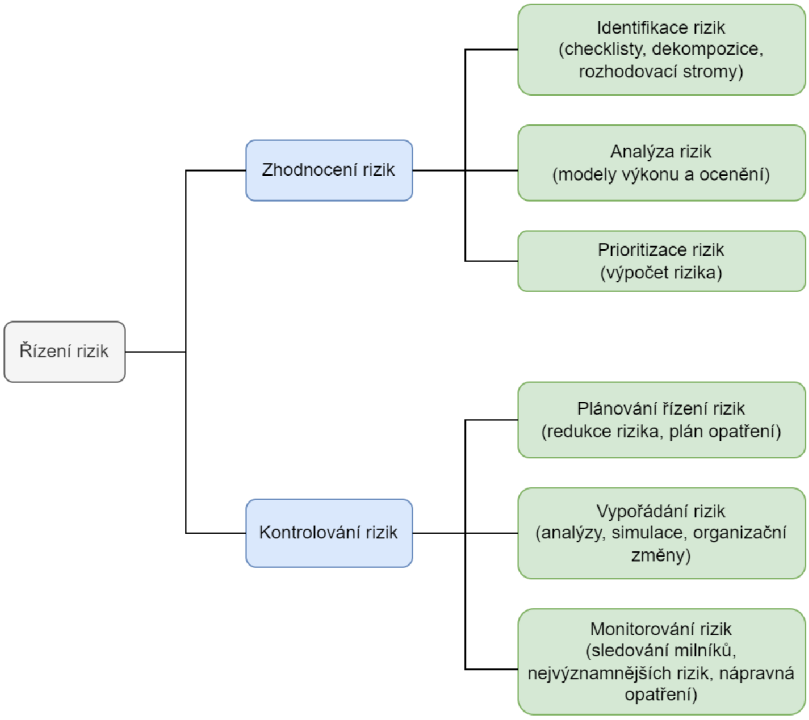
2.2.3 Rizika

Stejně jako i u nově vyvíjených systémů, proces reengineeringu s sebou nese mnoho rizik, které by měly být odpovědnými osobami (typicky projektový/produktový manažer) řízeny. Řízení rizik je důležité pro úspěšnost daného projektu, jelikož dává jasný přehled o dopadech a pravděpodobnosti jejich vzniku. Identifikací je možné včasně udělat opatření pro jejich minimalizaci. Obecně riziko lze definovat následovně [6]:

$$R = D_R * P_R$$

kde R je riziko, D_R je dopad daného rizika (vyjádřený buď relativně či absolutně v peněžních jednotkách) a P_R je pravděpodobnost vzniku.

Boehm [6, str. 34] dělí proces řízení rizik na následující fáze: zhodnocení rizik a kontrolování rizik. Zhodnocení rizik v sobě zahrnuje procesy identifikace rizik (vytvoření registru rizik například s využitím SWOT[34] analýzy), analýzy rizik (zjištění dopadů, pravděpodobnosti, spouštěčů rizika). Druhá fáze zahrnuje plánování řízení rizik, které spočívá v přípravě nápravných opatření pro případ vzniku rizika, dále zahrnuje vypořádání rizik, kde je snahou některá rizika zcela eliminovat z projektu. Fáze taktéž obsahuje monitorování rizik, kde jsou rizika průběžně sledována za účelem vypořádání či aplikace nápravných opatření.



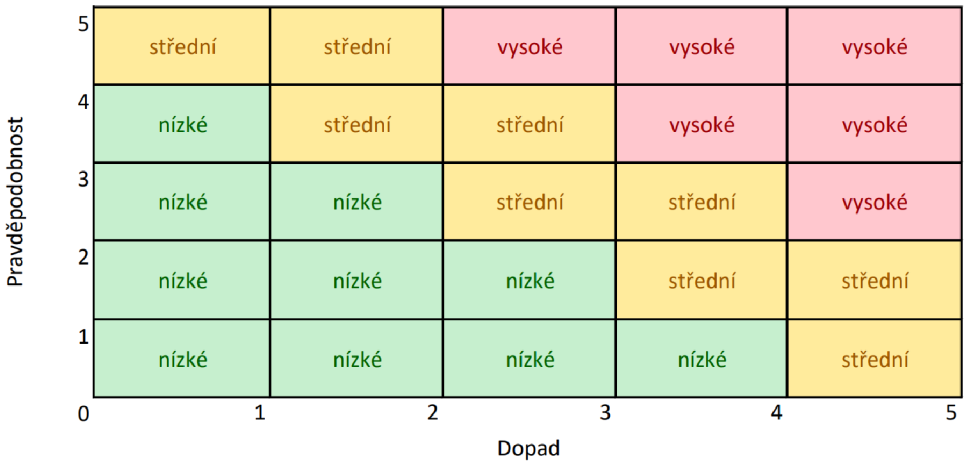
Obrázek 2.11: Fáze a kroky analýzy rizik dle Boehma [6]

[34]SWOT (*Strengths, Weaknesses, Opportunities, Threats*) analýza identifikující rizika v projektu

Jako nejčastější rizika softwarových projektů uvádí Boehm [6, str. 35] tyto položky: nepřesné odhady rozpočtu a plánu, nedostatky zaměstnanců, vývoj nekorektních funkcionalit, vlastností a uživatelského rozhraní, problémy externích komponent, výkonnostní nedostatky, neustále měnící se požadavky.

Konkrétně pro reengineering se z pohledu autora této práce jeví jako nejproblematičtější odhady ceny a plánu, jelikož pro přesné určení je nutné provést komplexní a přesnou analýzu celé business funkcionality původního řešení, a to včetně rozhovoru s uživateli, jelikož systém v průběhu svého používání mohl být používán nestandardně pro naplnění specifických scénářů, které nejsou nijak dokumentované. Objevení této skryté funkcionality až v průběhu reengineeringu může mít značné dopady do plánu a rozpočtu. Dalším neméně závažným rizikem pro reengineering je výběr komerčních/externích komponent pro naplnění určité funkcionality dříve realizované vlastním kódem. U těchto komponent je důležité provést detailní průzkum, zda naplňují funkční i nefunkční požadavky. Z vlastní zkušenosti autora je pak vhodné se zaměřit na míru a způsob přizpůsobitelnosti (a to i nad rámec aktuálně vyžadované úrovně) a výkonnostní stránku externího řešení. Bohužel výkon těchto komponent se bez alespoň prototypové implementace těžko ověřuje, což z tohoto požadavku tvoří jeden z nejrizikovějších, jelikož pro jeho ověření je nutné investovat nemalé prostředky před samotným výběrem, což vzhledem k celkové náročnosti analýzy nemusí být vždy naplněno.

Komplexní řízení rizik v projektu, ať už se jedná o projekt reengineeringu či běžný nový projekt, je silně odvislé od procesů dané společnosti a schopnostech konkrétního projektového manažera. Nástrojem pro takové řízení může být například sestavení 10 největších rizik [6, str. 40] či vytvoření mapy rizik, též matice pravděpodobnosti a dopadu. Matice jednoduše vizualizuje vypočtené riziko na základě jeho pravděpodobnosti a dopadu. Pomocí matice mohou být jednoduše identifikována položky s vysokým rizikem, u kterých následně může být vyvinuto úsilí na jejich mitigaci.



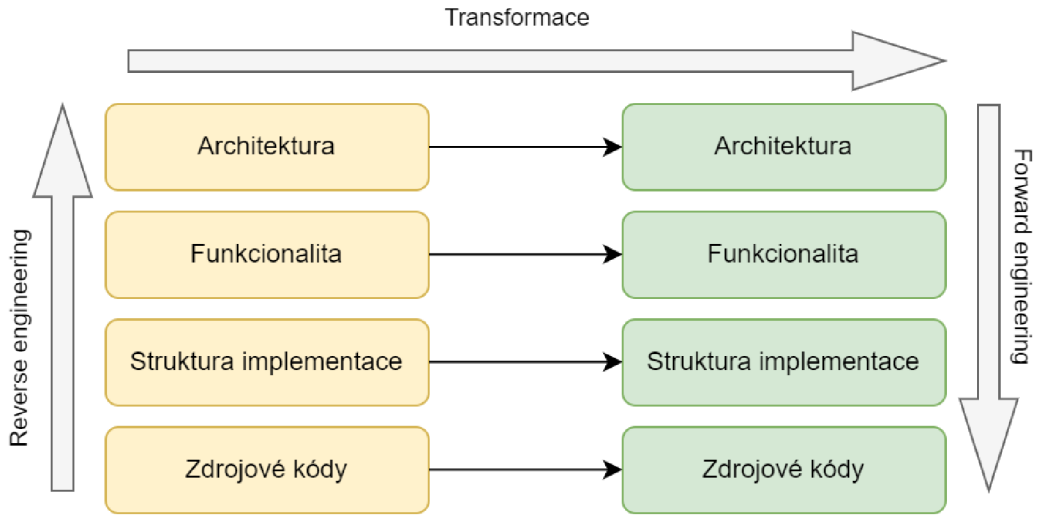
Obrázek 2.12: Ukázka mapy rizik, volně inspirováno [10]

2.2.4 Metodologie

Proces reengineeringu se ve své podstatě skládá ze třech fází: reverzního engineeringu, transformace a forward engineeringu. Reverzní engineering si klade za cíl zanalyzovat a porozu-

mět existujícímu řešení, transformace pak aplikaci požadavků a forward engineering tvorbě finálního řešení/implementace. Jednotlivé fáze budou hlouběji diskutovány v následujících podkapitolách.

Pro názorný popis procesu reengineeringu Kazman et. al. [27] používá tzv. *model podkovy* (viz obrázek 2.13), tedy metaforické znázornění procesu ve formě podkovy. Podkova se skládá z několika úrovní abstrakce (stejně jako reálný systém): zdrojové kódy, struktura implementace, funkční úroveň a architektura. Každá z těchto částí je pak procesem transformace pozměněna, jehož výsledkem je pak nový systém.



Obrázek 2.13: Model podkovy reengineeringu, převzato a inspirováno z [34, str. 30] a [27]

Reverzní engineering

Reverzní engineering je Rosenbergovou [44] definován jako proces analýzy systému za účelem identifikace komponent systému a jejich vzájemných vztahů s cílem přetvořit systém do jiné podoby. V rámci reverzního engineeringu je zkoumáno chování systému z pohledu zdrojového kódu i chování samotného systému. V určitých případech zdrojové kódy nemusí být dostupné důsledkem jejich ztráty, nemožnosti přístupu apod., potom je nutné opírat se pouze o zkompilovanou variantu systému. Výstupem reverzního engineeringu je sada znalostí o systému, a to ve formě buď control-flow diagramů (CFG) či jiné abstraktní reprezentace nebo anotovaného kódu. Alternativně může taktéž být výstupem i dokument popisující systém *as-is*. Znalosti extrahované tímto procesem mohou obsahovat ztracené informace (skrytá nebo nedokumentovaná funkcionalita), seznam postranních efektů či závislosti a vzájemná propojení.

Jako možné techniky pro reverzní engineering Matos [34, str. 39] jmenuje tzv. *program slicing* a *software reconnaissance*. Slicing spočívá v izolací vybraných částí systému za účelem porozumění dané části. Kritérium slicingu je definováno jako sada proměnných. Výpočet řezu/slice je pak proces, jehož výstupem je sada příkazů/metod, které interagují s danými proměnnými. Techniku ve své neformální podobě využívají i vývojáři při běžné činnosti vývoje a debuggingu, kde se slicing uplatní ve snaze izolovat konkrétní chybu programu.

Software reconnaissance, neboli průzkum softwaru, je technika využívající automatický průzkum vlastností softwarů s pomocí instrumentace a testování za účelem objevení kódu přímo souvisejícího s danou vlastnosti, tzv. *feature-specific* kódu [51]. Průzkum probíhá

zmíněnou instrumentací, kde jsou porovnávány stromy volání při volání dané vlastnosti a stromy bez volání vlastnosti. Tímto způsobem je možné snadno lokalizovat konkrétní části programu, které se podílí na realizaci dané vlastnosti. Tuto znalost pak lze využít pro další fáze reengineeringu.

V případech, kdy zdrojový kód není dostupný, je nutné buď zkoumat systém pouze z pohledu uživatelského, nebo se pokusit o dekompilaci binárních souborů. Úspěšnost dekomplikace z hlediska čitelnosti kódu silně závisí (z autorových zkušeností) na aplikovaných optimalizacích a obfuskačních technikách aplikovaných v době kompilace kódu. Kód, který byl kompilován a instrumentován pro debugging, lze typicky dekompilovat do téměř původní podoby, u produkčních sestavení či obfuskovaných je čitelnost omezená či téměř nemožná, ovšem i v těchto případech je možné identifikovat vedlejší efekty a závislosti, zvláště jedná-li se o interakci s databází či jinými službami. Příkladem nástrojů pro dekompilaci je JetBrains dotPeek[35] pro .NET nebo Java Decompiler[36] pro jazyk Java.

Transformace

Transformace v kontextu reengineeringu je proces přetvoření původního modelu systému na nový [34]. Proces může být aplikován na různých úrovních – od zdrojových kódů po úroveň architektury, viz obrázek 2.13.

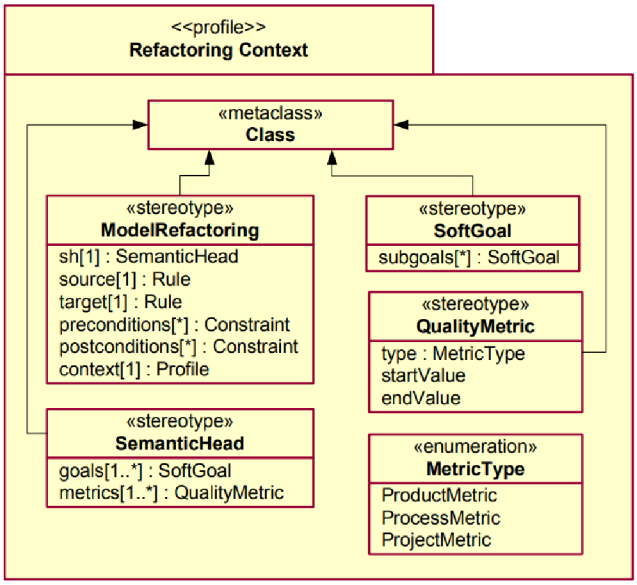
Na úrovni zdrojových kódů, jejich struktury a funkcionality se typicky jedná o automatické transformace s pomocí anotací. Matos [34] tuto problematiku řeší modelováním původního anotovaného kódu s pomocí grafu. Následně na tento graf aplikuje refaktorizační transformace. Matos abstrahuje aplikaci pravidel na takovou úroveň, která je relevantní pro vykonání daných transformací (bloky kódu či celé třídy). Transformace dále rozlišuje na technologické a funkční. Prvně jmenované se primárně zabývají oddělením business logiky a logiky uživatelského rozhraní. Funkční transformace pak řeší změny kódu v souvislosti s přeuspořádáním jeho struktury.

Z hlediska problematiky této práce je nejvýznamnější transformace architektonická. Ivkovic a Kontogiannis [26] ve své práci představují framework pro systematickou transformaci architektury. Framework řeší formalizaci a standardizaci nefunkčních (kvalitativních) požadavků s využitím technik UML. Autoři definují koncept *kontextu refaktorizace*, který modelují jako UML profil[37]. Dále jmenují následující komponenty kontextu refaktorizace:

- `ModelRefactoring` – konkretní anotovaný model transformace skládající se ze `SemanticHead`, výchozího a cílového vzoru, vstupních a výstupních podmínek
- `SemanticHead` – sémantická anotace stanovující cíle (`SoftGoal`) a metriky (`QualityMetric`) transformace
- `SoftGoal` – reprezentuje graf cílů refaktorizace, vychází z tzv. *soft-goal interdependence grafu*, který reprezentuje vzájemnou závislost mezi jednotlivými cíli
- `QualityMetric` – kvalitativní metrika obsahující vypočtené hodnoty pro vychozí a cílový vzor transformace

V případech, kdy transformace systému zahrnuje nejenom zásadní změnu architektury, ale i diverzifikaci technologií spojenou s jiným řešením naplnění funkčních požadavků, je

[35] https://www.jetbrains.com/decompiler/
[36] http://java-decompiler.github.io/
[37] Standardní nástroj UML pro rozšíření a adaptaci na konkrétní doménovou problematiku



Obrázek 2.14: Kontext refaktorizace podle Ivkovic a Kontogiannis, převzato z [26]

automatické či poloautomatické řešení transformace, dle pohledu autora této práce, značně obtížné. Je tedy nutné zvážit, zda-li nepřistoupit k transformaci neformální cestou, kdy nový systém bude navrhnut tzv. *top-down* a následné mapování původních komponent/bloků na nové, bude provedeno ad-hoc, kdy bude zároveň rozhodnuto, které části budou transformovány, nahrazeny či zaobaleny.

Forward engineering

Finální fázi reengineeringu je tzv. *forward engineering*. Tato fáze spočívá v generování výstupů předchozí fáze transformace. Výstupem je typicky zdrojový kód – nová implementace – na základě pravidel transformace, která splňuje výstupní podmínky [34]. V závislosti na použitém přístupu může být fáze forward engineeringu reprezentována celým vývojovým cyklem (náhrada), obalením minimální vrstvou (wrapping) nebo právě přegenerovaným kódem z transformovaného grafu. Posledně jmenovaný případ je realizován pomocí nástrojů generujících kód. Příkladem takového nástroje může být modelovací nástroj Visual Paradigm[38], který umožňuje generování kódu (např. do Java či C#) z diagramů tříd. Podobnou funckionalitu pak také nabízí IBM Engineering Systems Design Rhapsody[39].

2.2.5 Strategie

K reengineeringu lze obecně přistoupit několika způsoby. Matos [34, str. 79] pak konkretně jmenuje následující způsoby: top-down, bottom-up a hybridní. Strategie se liší přístupem, ze které strany bude proces reengineeringu v hierarchii abstrakce (viz Obrázek 2.13) započat.

Strategie **top-down** nejprve analyzuje business doménu problematiky, navrhuje architekturu, interakce mezi high-level komponentami/službami, stanovuje kontrakty a orchestraci mezi články systému. Postupně je návrh konkretizován a pro jednotlivé části je

[38] https://www.visual-paradigm.com/
[39] https://www.ibm.com/products/systems-design-rhapsody

stanoveno, na které je možné aplikovat techniky reengineeringu a případně i zvolit jeho konkrétní způsob. Z pohledu autora této práce je tento způsob vhodné aplikovat v takových případech, kde výsledná architektura řešení je zásadní vlastností systému, jelikož taková architektura může vyplývat z konkretních požadavků zákazníka, nebo může být klíčová pro naplnění nefunkčních požadavků na výkon, škálovatelnost či modularitu systému.

V rámci **bottom-up** strategie nejdříve probíhá analýza zdrojového kódu, ze kterého jsou následně určeny kandidátní služby či modulární komponenty. Na rozdíl od top-down, kde primárním podnětem může být nutná změna architektury vyplývající z externích požadavků, tato strategie může být vhodná při řešení požadavků, které vycházejí z nedostatků současné implementace. Proces refaktorizace může být příkladem iniciátora procesu bottom-up reengineeringu, kde namísto lokalizovaného zásahu – reimplementace/restrukturalizace komponenty – se změna dotýká celkové architektury systému.

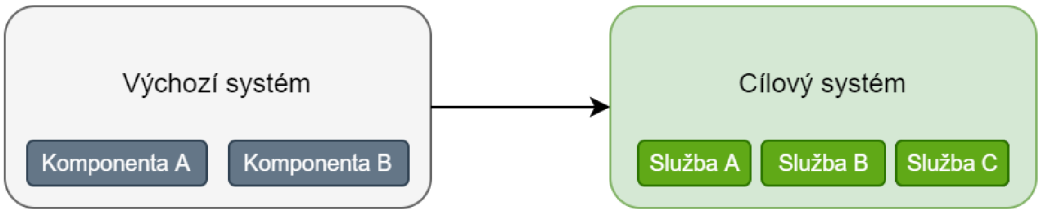
Hybridní přístup pak kombinuje techniky top-down a bottom-up. Návrh systému je tedy realizován kombinací analýzy domény a návrhu architektury se znalostmi získanými analýzou zdrojového kódu. Lze tedy mluvit o informovaném návrhu architektury na základě znalosti existující implementace. Výhodou může být dosáhnutí většího překryvu mezi existujícím a cílovým systémem, což umožňuje lépe aplikovat transformace reengineeringu.

Volba konkrétního přístupu silně závisí na omezujících podmínkách cílového řešení, nefunkčních požadavcích, prioritách a možnostech v daném prostředí.

2.2.6 Přístupy k náhradě existujícího řešení

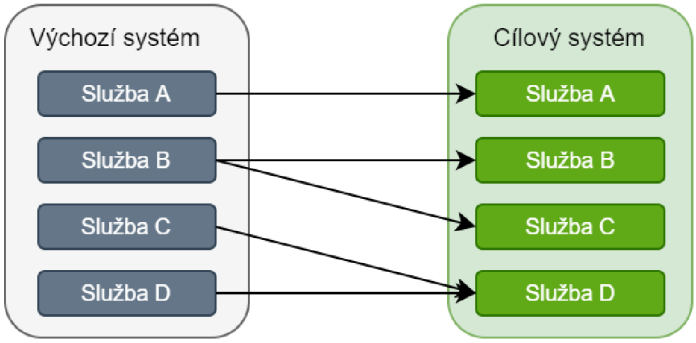
Z hlediska procesu náhrady existujícího řešení za cílové Rosenbergová [44, str. 7–9] uvádí tři možné přístupy: *big-bang*, inkrementální a evoluční. Přístupy se liší v tom, jak velká část systému a jakým způsobem je nahrazena v průběhu přechodu na cílové řešení.

Přistup tzv. **big-bangu** provádí náhradu celého systému v jednom kroce. Přístup se typicky aplikuje v situacích, kdy z pohledu zásadní změny architektury není možné paralelně provozovat části starého a nového systému. Mezi výhody patří fakt, že není třeba vytvářet vrstvy zpětné kompatibility a testovat vzájemnou interakci mezi původními a novými komponentami. Nevýhodou je typicky vysoká náročnost skokovitého přechodu mezi systémy, kdy je potřeba zajistit dostatečné proškolení, uskutečnit migrace a validovat, že všechny kritické funkcionality budou novým systémem naplněny již od samotného spuštění. V praxi (vypozorované autorem této práce) se tyto rázovité přechody mohou řešit tvorbou a provozováním testovacího prostředí, na kterém je provozován cílový systém s obrazem namigrovaných dat z produkčního prostředí. Na tomto prostředí pak probíhají akceptační testy zákazníka, který před nasazením systému do produkce schvaluje/akceptuje jednotlivé funkcionality systému. Při nasazení/náhradě je pak původní systém odstaven, čerstvá data jsou namigrována a nový systém je uveden do produkčního prostředí.



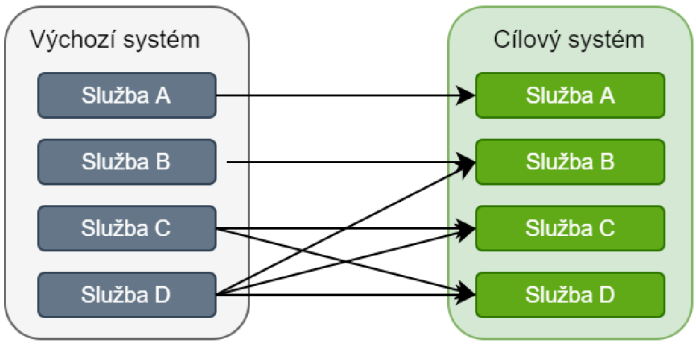
Obrázek 2.15: Big-bang přístup k náhradě systému, inspirováno [44]

Inkrementální přístup náhrady systému umožňuje postupný *phase-out* stávajících komponent systému, kdy jednotlivé části jsou postupně nahrazeny za nová řešení. Rosenbergová uvádí, že značnou výhodou oproti kompletní náhradě je snížené riziko ztráty některé funkcionality původního systému a taktéž snazší dohledání chyb. Zákazník může jasně pozorovat postup prací na náhradě systému. Jelikož jsou v průběhu náhrady vytvářeny nové přechodné verze systému, je nutné validovat chování a funkcionalitu celého systému, což může znamenat značné časové i finanční výdaje. Zásadní omezující nevýhodou je fakt, že celková architektura systému nemůže být do značné míry změněna. Inkrementální přístup dovoluje rozdělovat a slučovat původní služby.



Obrázek 2.16: Inkrementální přístup k náhradě systému, inspirováno [44]

Posledním přístupem, který Rosenbergová zmiňuje, je přístup **evoluční**. Na rozdíl od přístupu inkrementálního podmnožina služeb, která bude nahrazena, je určena na základě funkcionality cílového systému nezávisle na tom, kde se tato funkcionalita nachází ve stávajícím systému. Předpokladem je tedy nalezení nahrazované funkcionality napříč existujícím systémem. Výhodou je jasné oddělení funkcionality v cílovém systému. Na druhou stranu je nutné brát v potaz možnost, že bude nutné vytvořit dočasná rozhraní v původních službách a celkový výkon systému může být dočasně degradován [44, str. 9].



Obrázek 2.17: Evoluční přístup k náhradě systému, inspirováno [44]

2.2.7 Metody reengineeringu v kontextu architektury služeb

Reengineering monolitických řešení na drobné služby přináší specifické výzvy a problémy z hlediska toho, jakým způsobem uchopit stávající funkcionalitu, která má být přenesena do cílového řešení. Koutsoukos et al. [29] identifikují tyto následující metody: náhrada, zaobalení a restrukturalizace.

V případě plné **náhrady** stávající funkcionality za novou implementaci může být využito nejmodernějších technologií a přístupů bez vazby na původní řešení. Stejně tak lze novou službu zobecnit, případně použít již existující řešení, a to buď interní, nebo i komerční. Náhrada nese značné časové a finanční náklady, jelikož se jedná o klasický software engineering. Reimplementací také může být ztracena určitá funkcionalita systému, což v důsledku může vést k selhání projektu [34, str. 72].

Zaobalení funkcionality do služby může vést krátkodobě k dosažení dobrých výsledků, jelikož je zachováno původní jádro řešení. Z dlouhodobého pohledu je tento přístup problematický z hlediska udržitelnosti a následného rozvoje původní funkcionality. Primárním motivem pro výběr tohoto přístupu může být rychle a s nízkými náklady systém přetransformovat do nové podoby a následně se zaobírat náhradou či restrukturalizací jednotlivých služeb.

Poslední zmíněnou metodou je pak **restrukturalizace**. V rámci této metody je původní funkcionalita zaobalena do služby a následně její implementace modifikována tak, aby byla zajištěna dlouhodobá udržitelnost a bylo možné funkcionalitu do budoucna dále rozvíjet. Z hlediska časových i finančních nákladů se jedná o náročnější metodu než zaobalení.

Kapitola 3

Reengineering aplikace Systém správy kabeláže

Za účelem demonstrace principů a postupů reengineeringu popsaných v kapitole 2 společnost I&C Energo[1] poskytla modelový případ. Autor této práce je v pracovním poměru se společností I&C Energo na pozici programátor analytik. Předmětem reengineeringu je aplikace Systém správy kabeláže (SSK). Následující kapitola se věnuje popisu, analýze existujícího řešení a návrhu nového. Budou objasněny výchozí motivace, požadavky funkční i nefunkční a celková vize nového řešení.

Na systém bude nahlíženo především z pohledu reengineeringu a změny architektury cílového řešení. Předmětem není hloubkový popis ani analýza pokročilé business funkcionality, která je duševním vlastnictvím společnosti I&C Energo. Výsledná implementace, která bude odevzdána v rámci této práce, bude zbavena funkcionality, která není podstatná z hlediska tématu diplomové práce, a také většiny funkcionality, jejíž autorem není autor této práce. V případech, ve kterých budou převzaty části původní implementace, bude autorství explicitně zmíněno.

3.1 Účel a základní popis systému

Systém správy kabeláže je CAD[2]/CAE[3] aplikace určená k podpoře projektování, realizace a správy kabelových systému komplexních technologických celků, jakými jsou například jaderné, uhelné či paroplynové elektrárny. Aplikace byla vyvíjena v letech 2004–2010 v rámci společného projektu I&C Energo a Západočeské univerzity v Plzni pod vedením Ing. Václava Hajšmana, Ph.D. Projekt byl realizován za podpory Ministerstva průmyslu a obchodu [24] [25]. Doktor Hajšman v současné době působí ve společnosti I&C Energo jako vedoucí oddělení „Správa projektových dat“ a zároveň je produktovým manažerem aplikace SSK. Mezi hlavní přínosy Systému správy kabeláže se z hlediska jeho provozování řadí [23]:

- centralizovaná správa informací o prvcích kabelového systému (kabely, kabelové trasy, koncová zařízení, spojky, průchody atd.)
- úspora nákladů na projektování, realizace a správu kabelového systému s pomocí automatizovaného trasování kabeláže

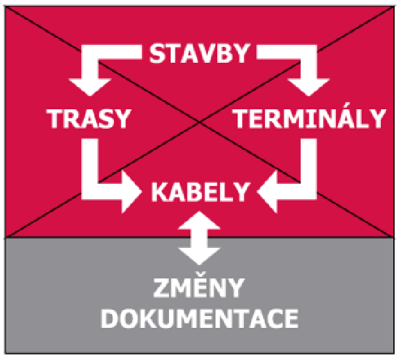
[1] https://www.ic-energo.eu/
[2] Computer-aided design (CAD) – software pro projektování
[3] Computer-aided engineering (CAE) – software pro inženýrskou analýzu

- sledování a řízení změn kabelového systému od návrhu přes provozování po rušení
- zajištění a prokazování dodržení technických a legislativních norem[4] při návrhu a provozování kabelového systému
- podpora zajištění kvality procesů správy prvků kabelového systému v souladu se systémem řízení jakosti

V současné době je SSK provozováno na Jaderné elektrárně Dukovany (EDU) a Temelín (ETE), kde naplňuje výše zmíněné přínosy. Systém byl dále použit při realizaci komplexních obnov na klasických elektrárnách Tušimice II, Prunéřov II a výstavbě nových zdrojů v lokalitách Ledvice a Počerady. V systému jsou napříč těmito lokalitami spravovány vysoké stovky tisíc kabelů.

3.1.1 Moduly systému

Systém je logicky členěn do několika modulů, z nichž každý tvoří jednu část informačního modelu daného technologického celku. Moduly společně tvoří digitální popis prvků kabelového systému.



Obrázek 3.1: Základní moduly SSK, zdroj [23]

Modul **staveb** eviduje stavební dispozice technologického celku s pomocí definice stavebních objektů, řezů, místností a zaměřených bodů. Stavba je část technologického celku zaměřená v daném bodě, typicky odpovídá fyzické budově. Řez přestavuje horizontální průřez stavbou v určité výškové hladině, který může a nemusí odpovídat stavebnímu podlaží budovy. Místnost je nejmenší jednotka budovy nalézající se na jednom či více řezech. Posledně pak body představují konkrétní zaměřenou pojmenovanou souřadnici uvnitř budovy, na kterou se dále zaměřují jednotlivé trasy a terminály.

Modul **terminálů** uchovává informace o koncových zařízeních připojených na kabeláž. Koncovými zařízeními mohou být například rozvaděče, spotřebiče (čerpadla, armatury) nebo hermetické stěny, na nichž jsou kabely ukončeny (terminály mají zaměření a definují body počátku/konce při trasování kabelů). Terminál je dále dělen na segmenty, které reprezentují jednotlivé části terminálu, např. svorkovnice nebo hermetické průchodky v případě hermetických stěn. Segment lze pak dále dělit na prvky, které identifikují např. svorky nebo žíly hermetických průchodek, na které jsou připojeny kabely.

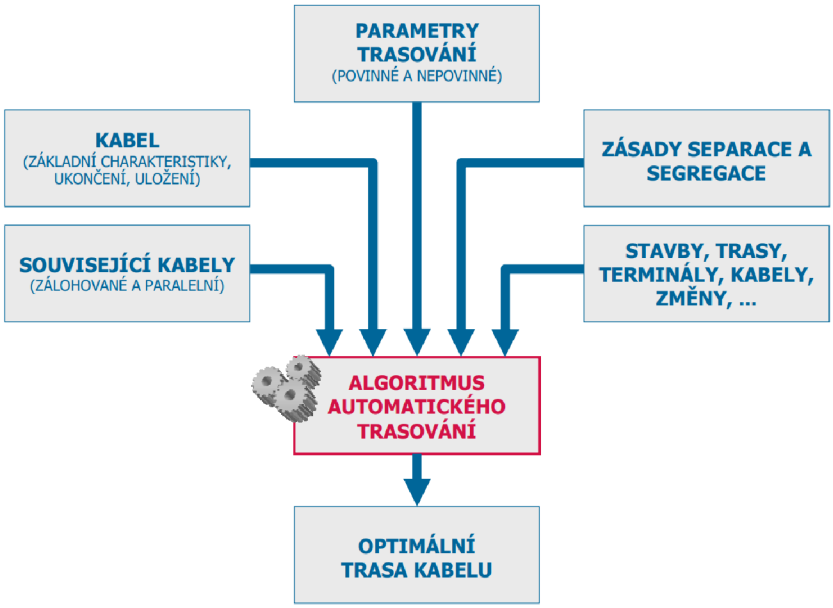
Modul **tras** popisuje vedení a umístění stavebních a konstrukčních prvků sloužících pro uložení kabeláže. Jedná se například o lávky, žlaby, trubky nebo stavební prostupy (otvory).

[4] V případě ČR je dozorujícím orgánem Statní úřad pro jadernou bezpečnost (SÚJB).

Každá trasa je charakterizována svoji separační skupinou, která společně se separační klasifikací místností tvoří základ pro komplexní řešení zajištění separace. Trasa je dělena na segmenty, kde každý segment tvoří neměnnou část trasy z hlediska uspořádání prvků pro uložení kabelů, změny dispozice trasy (spojení, rozbočení lávek trasy) či odbočení významného množství kabelů. Segment je dále tvořen prvky popisujícími jednotlivé žlaby, lávky, trubky, ze kterých se daná část/segment trasy skládá.

Konečně pak modul **kabelů** eviduje kabely napříč výše zmíněnými moduly. Kabel je definován svým počátečním a koncovým terminálem, posloupností segmentů/prvků trasy (tedy uložením) a kabelovými spojkami. Kabely jsou v systému předmětem změnového řízení, které mapuje celkový životní cyklus kabelu od jeho návrhu, provoz po odstranění. Změnové řízení je definováno stavovým modelem kabelu. Systém dále umožňuje generování řízené dokumentace na základě změnového řízení.

Zásadní přidanou hodnotou je pak funkcionalita **automatického trasování kabelu**. Algoritmus na základě informací z modulů, konkrétních charakteristik kabelu, jeho souvisejících kabelů (paralelní a zálohované) a parametrů trasování je schopen navrhnout optimální trasu kabelu tak, že budou zachovány zásady separace, segregace a zároveň trasa kabelu bude cenově optimální (délka kabelu, jakož i míra splnění všech požadavků a zásad, je cenově ohodnocena/penalizována). Alternativně lze hovořit o tom, že optimální trasa je trasa s nejnižší celkovou penalizací.



Obrázek 3.2: Vstupy a výstupy algoritmu trasování, autor I&C Energo

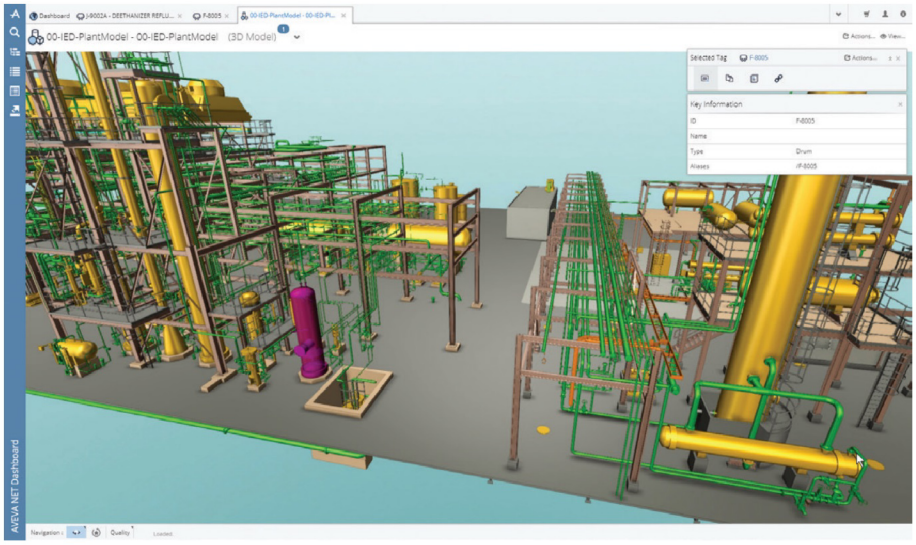
3.1.2 Podobná a konkurenční řešení

V oblasti správy a trasování kabeláže se na trhu vyskytuje několik typů produktů. Prvním z nich jsou obecná softwarová řešení pro správu informačního modelu staveb[5] a technologických celků. Tato řešení poskytují komplexní možnosti projektování a správy objektů těchto celků. Příkladem tohoto typu řešení je AVEVA Electrical & Instrumentation[6]. Tato řešení

[5] Taktéž známo pod názvem *building information model* – BIM.

[6] https://www.aveva.com/en/products/electrical-instrumentation/

typicky vyžadují existenci plně digitálních modelů pro existující lokality. Tento fakt značně znemožňuje nasazení těchto systémů na lokalitách, kde se data historicky evidovala pouze tzv. alfa-numerickou formou, tedy s absencí prostorového popisu technologického celku.



Obrázek 3.3: Prostředí nástroje AVEVA.NET pro prohlížení 1D, 2D i 3D dat, zdroj [2]

Druhým typem řešení jsou specializované softwarové produkty pro návrh kabeláže, mezi které se řadí i SSK. Tyto produkty se vyznačují užším záběrem, nespravují celé informační modely, ale jsou zodpovědné pouze za příslušnou oblast, tedy kabeláž. U ostatních datových objektů (stavby, zařízení) je vlastníkem dat jiné softwarové řešení, které může spadat i do prvně zmíněného typu produktů. V oblasti specializovaných řešení na trhu působí následující produkty: Bentley Raceway and Cable Management[7] a Traceman[8] od společnosti KIP Brno. Obě řešení poskytují služby automatického trasování kabeláže na základě mnoha kritérií (separace, segregace, zatížení, zaplnění apod.), tedy podobně jako SSK. Dále také obě řešení umí pracovat ve 2D i 3D režimech, přičemž řešení od Bentley se primárně zaměřuje na projektování a editaci kabeláže z 3D grafického režimu, jedná se tedy o řešení, které silně závisí na existenci těchto prostorových dat.

Specifické podmínky starších technologických celků mají značné dopady na využitelnost výše zmíněných softwarových řešení. V mnoha případech byl kabelový systém projektován v dobách před existencí jakýchkoliv softwarových řešení a jeho dokumentace musela být v průběhu let digitalizována. Původní evidence v papírové podobě a následná digitalizace mnohdy odhalila značné dokumentační neshody, které v rámci nasazení digitální správy kabelového systému nebylo možno ihned odstranit, tedy nekritická část kabeláže mohla obsahovat nevalidní data, například neexistující reference mezi prvky kabelového systému či duplicitní značení. V některých případech nevalidita dat mohla vzniknout absencí mechanismů referenční a datové integrity v historických systémech. Systém správy kabeláže je schopen s těmito nevalidními daty pracovat, a to tak, že je volitelně umožněno vypnout mechanizmy referenční integrity.

[7] https://www.bentley.com/en/products/product-line/plant-design-software/bentley-raceway-and-cable-management

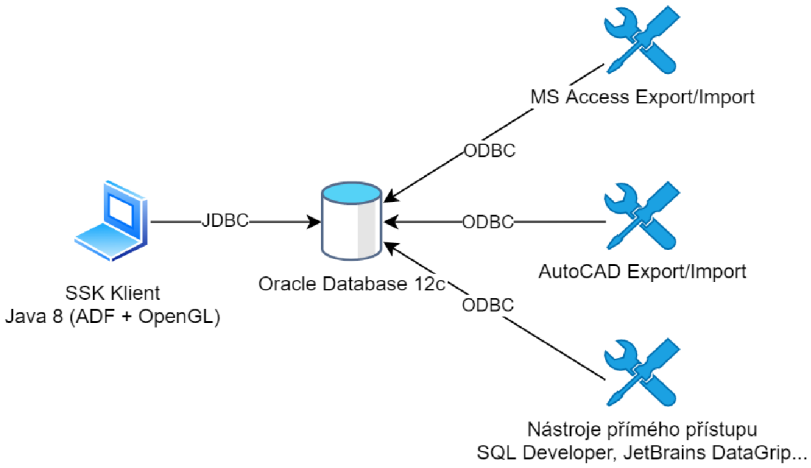
[8] https://www.kipbrno.cz/produkty/trasovani-a-sprava-kabelaze-kabelovy-system-7.html

3.2 Architektura a implementace výchozího řešení

Výchozí řešení označované jako SSK 2.0[9] je dvouvrstvá aplikace s tlustým klientem pro operační systém Microsoft Windows a databází Oracle hostovanou na serveru. Klient je implementován v jazyce Java ve verzi 8 s využitím Oracle ADF[10] komponent. Pro grafické zobrazení aplikace využívá OpenGL rendereru. Server je tvořený Oracle databází ve verzi 12c.

Na databázi Oracle jsou přímo připojeny nástroje zajišťující hromadný export či import stavebních dispozic, terminálů, tras a kabelů v podobě seznamů a souborů ve formátu DWG/DXF[11]. Nástroje SSK pro export/import souborů DWG/DXF jsou přímo závislé na AutoCAD .NET API, které vyžaduje plnou licenci softwaru AutoCAD[12]. Existence těchto nástrojů je nutná z důvodů funkčních omezení způsobených současnou implementací systému.

V kontextu provozovaných řešení u zákazníka je pak pro každou jeho lokalitu/technologický celek vytvořena samostatná databáze. Jedna klientská aplikace může být za předpokladu kompatibility verzí používána pro připojení na vícero lokalit. Volba lokality je realizována pomocí parametrizace řetězce pro připojení k databázi.



Obrázek 3.4: Architektura SSK z pohledu jedné lokality

Z důvodu, že k databázi přistupují i jiné nástroje než samotná aplikace SSK, je business logika řešená implementována převážně v databázi formou triggerů a PL/SQL procedur. Datová integrita je tedy zajištěna i při přímém přístupu do databáze z nástrojů jakými je například SQL Developer nebo JetBrains DataGrip. Umístěním business funkcionality přímo do PL/SQL procedur je umožněno tyto procedury využívat i ostatním připojeným nástrojům.

Proporční dispozice business systému jsou prezentovány v modulu grafiky. Tento modul zobrazuje v 3D prostoru rozložení staveb a jejich bodů, terminálů, tras a kabelů. Business objekty jsou reprezentovány zástupnými symboly/objekty, jedná se tedy o schématické 3D zobrazení. Modul je možné vyvolat z kontextu aplikace (business objekt a příslušné řezy

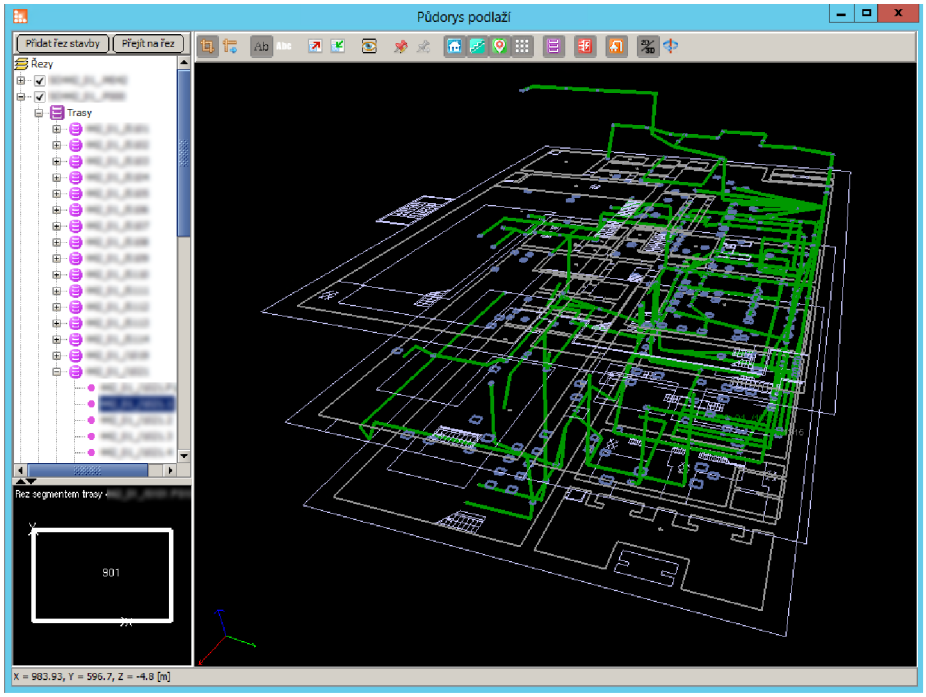
[9] Verze 2.0 bylo evolucí verzí 1.x, oproti které 2.0 přinesla vylepšení v oblasti změnového řízení a grafického zobrazování dat.

[10] https://www.oracle.com/database/technologies/developer-tools/adf/

[11] Formáty CAD systémů pro ukládání 2D/3D dat.

[12] https://www.autodesk.cz/products/autocad/overview

stavby), kde se následně otevře nové okno, ve kterém se grafická data zobrazí. V modulu je možné dynamicky donačítat další řezy či kabely. Modul taktéž poskytuje 2D režim, který imituje výkresové zobrazení, včetně odpovídacího symbolického značení např. stoupaček. Pro segmenty tras jsou v tomto modulu zobrazovány i jejich schématické řezy znázorňující jednotlivé konstrukční prvky (žlaby, lávky apod.) a jejich zaplnění/zatížení. Současné řešení využívá OpenGL, konkretně pak knihovnu JOGL[13], která poskytuje Java rozhraní přímo na OpenGL API.



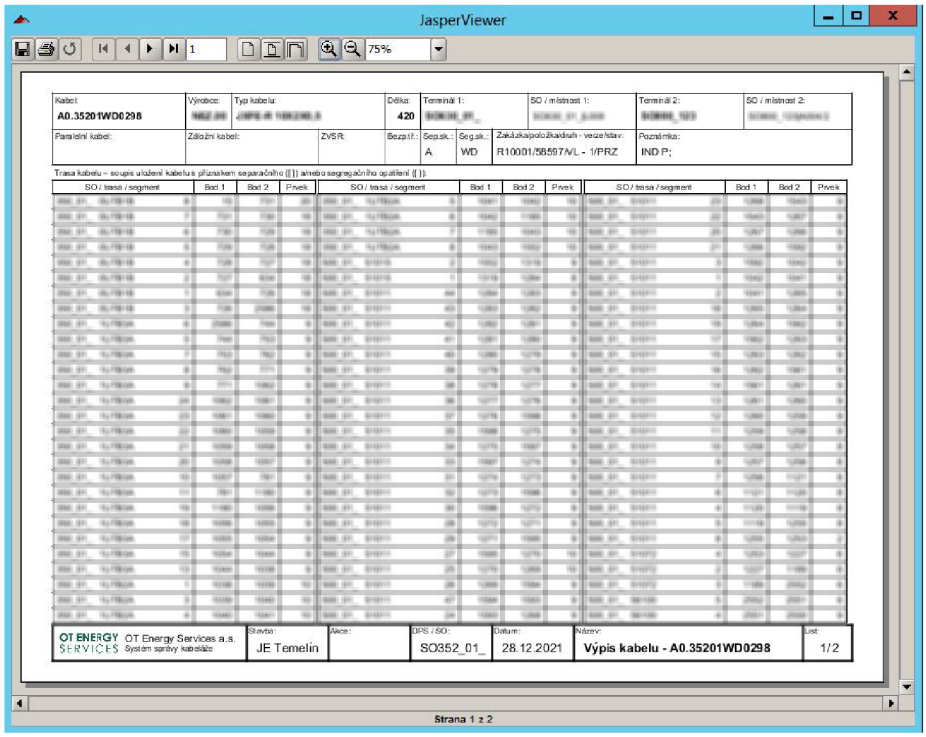
Obrázek 3.5: Modul zobrazení 3D schématické grafiky SSK

Podstatnou funkcionalitou systému je automatické trasování kabeláže v rámci datového modelu technologického celku. Algoritmus trasování si klade za cíl nalézt optimální cestu pro kabel mezi dvěma koncovými zařízeními s přihlédnutím ke všem omezujícím podmínkám separace, segregace, zatížení apod. Cenově optimální trasa je taková, kde je minimalizována délka kabelu jakož i penalizace nesplnění požadavků. Algoritmus trasování kabeláže je časově náročná úloha. Úloha vyžaduje značné systémové prostředky a je citlivá na rychlost přístupu k datům. V současném řešení je algoritmus trasování z těchto důvodů implementován v databázi Oracle prostřednictvím procedur a funkcí. Algoritmus je parametrizován pomocí vytvoření *dávky*, v rámci které jsou stanoveny konkretní podmínky a způsoby trasování.

Systém dále umožňuje generování reportů/tiskových sestav na základě předdefinovaných šablon. Reporty slouží jako dokumentace pro zákazníka např. při řízených změnách kabelů, výpisu segmentů tras pro pokládku apod. Mezi podporované formáty výstupu se řadí PDF, DOCX, XLS, HTML atd. Reporty jsou generovány na základě vstupních parametrů, např. kód kabelu. Výstup je zobrazen v okně prohlížeče reportů. Implementace je provedena s pomocí nástroje Jasper Reports[14]. Definice šablon pro reporty je prováděna vývojářem

[13] https://jogamp.org/jogl/www/
[14] https://community.jaspersoft.com/

v návrhovém prostředí a koncový uživatel nemůže provádět v prostředí systému změny těchto šablon. Místem uložení těchto šablon je databáze.



Obrázek 3.6: Okno náhledu reportu v SSK 2.0

Pro podporu práce v terénu, tedy například v oblastech primárního okruhu jaderných elektráren nebo obecně místech bez pokrytí internetu, je aplikace provozována v offline režimu. Současná aplikace tento požadavek naplňuje prostřednictvím vytvoření obrazu databáze a jejím následném provozování na lokální stanici. Tento režim provozu je v současné době umožněn omezené skupině pokročilých uživatelů. Zpětná propagace změn do centrální databáze není nijak řešena, jelikož se předpokládá, že uživatelé budou na data pouze nahlížet při práci v terénu.

Aplikace umožňuje připojování souborů k business objektům. Ke každému objektu je možno připojit více souborů. Soubory jsou nahrávány a ukládány přímo do databáze jako `BLOB`. Soubory je možné po jednom exportovat ze systému.

Autentizace a autorizace probíhá na úrovni databáze, kam se jednotliví uživatelé přihlašují pod vytvořenými SQL účty. Každý uživatel je přihlášen do svého schématu, ke schématu `SSK` je vyžadováno přidělené oprávnění. Práva jsou přiřazována prostřednictvím databázových rolí.

Logování je odděleně realizováno na straně klienta i databáze. Klient do souborů v instalační složce zaznamenává chybové události včetně výjimek. Databáze provádí logování vytváření, modifikací a mazání dat do speciální logovací tabulky. Záznamy o těchto změnách slouží pro zpětné auditování a dohledatelnost.

3.3 Datová a procesní integrace

Business entity popsané v kapitole 3.1.1 jsou součástí širšího informačního managementu daného technologického celku. Některé entity jsou tedy vlastněny jinými systémy a SSK k nim eviduje dodatečné informace. V případě provozu na EDU a ETE je SSK vlastníkem entit tras a kabelů. Primárním datovým zdrojem pro entitu stavby je systém typu GIS[15]. U terminálu se jedná o tzv. *master equipment list* (MEL), kde jsou centralizována data o všech zařízeních dané elektrárny. Data jsou z těchto zdrojů synchronizována ad-hoc.

Funkcionalita trasování je používána pro projektování nové kabeláže nebo modifikací stávající kabeláže, v případě potřeby pro doplnění tras, segmentů a/nebo prvků jako míst pro uložení kabeláže. V systému je tedy vytvářen *as-designed* stav, jedná se o budoucí – projektovaný – stav datového modelu. Typicky se může jednat o projektování nových uložení kabeláže při stavebních úpravách. Tyto úpravy či podobné změny by z principu vyžadovaly existenci as-designed stavu i u všech externích vlastníků entit, což ne všichni tito vlastníci aktuálně evidují. Docílení přítomnosti as-designed stavu v primárních datech by bylo možné změnou podnikových a mezipodnikových procesů zákazníků. V současném stavu je tedy v SSK umožněno i nad externě vlastněnými entitami provádět libovolné operace a eventuální datová konzistence mezi SSK a GIS/MEL je zajištěna ručně.

3.4 Východiska reengineeringu

Implementace stávajícího řešení je v současné době více než 12 let v zásadě nezměněné podobě, což s sebou nese jistá úskalí při provozování na současné infrastruktuře, a to jak na straně klienta, tak i serveru. Z hlediska využitých technologií je nutné dbát zvýšenou pozornost na končící životní cykly a podporu zajištěnou ze strany vydavatele technologií. Za dobu provozu taktéž došlo ke značném pokroku v centralizaci a správě informačního modelu staveb, což přináší požadavky na integraci s jinými systémy a jejich vzájemnou synchronizaci pomocí standardních rozhraní REST API.

S integrací souvisí i celková konsolidace nástrojů pro správu kabeláže pod SSK. Integrace přímo do aplikace tyto funkcionality standardizuje a umožní optimalizaci souvisejících podnikových procesů.

Z pohledu business cílů je třeba alespoň udržovat krok s podobnými či konkurenčními řešeními. Na základě konkurenčních řešení zmíněných v kapitole 3.1.2 je zřejmé, že modernizace SSK dává jedinečný prostor nabídnout zákazníkům moderní a flexibilní řešení, které bude v některých ohledech napřed.

Reengineering SSK je tedy z jednoho pohledu nutnost pro udržení kroku s technologiemi a konkurencí, ale také příležitost systém rozvinout a integrovat s moderními řešeními nabízejícími komplexní správu informačního modelu.

3.5 Identifikace nedostatků stávajícího řešení

Za účelem korektního návrhu cílového řešení bylo nutné sesbírat zpětnou vazbu, která se nahromadila za dobu provozu systému od jeho poslední větší aktualizace v roce 2012. Dále na systém bylo na systém pohlíženo v kontextu současných trendů a aktuálních požadavků na realizaci softwarových produktů napříč potenciálními i stávajícími zákazníky. V neposlední řadě taktéž proběhla analýza interních procesů v rámci společnosti I&C Energo, které jsou

[15]Geographic Information System (GIS) – systém spravující prostorová a mapová data

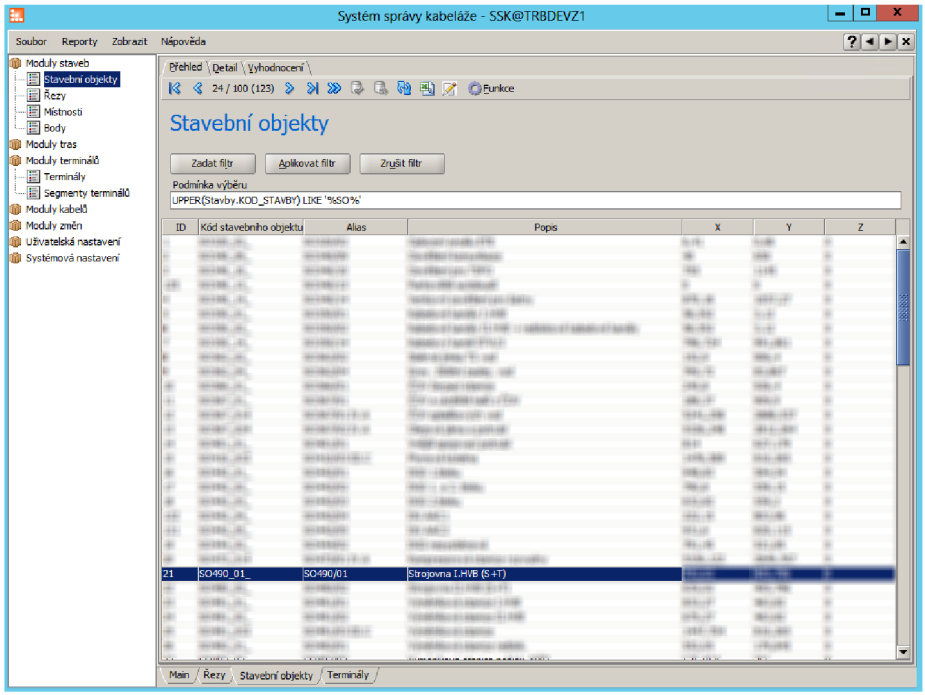
přímo vázané na činnosti související s poskytováním služeb správy kabeláže a trasování pro zákazníky. Celkově tedy zdroje nedostatků lze kategorizovat následovně:

- **externí** – nedostatky identifikované zákazníkem
- **interní** – nedostatky z hlediska interního používaní SSK pro potřeby poskytování služeb zákazníkům
- **technologické** – nedostatky vyplývající z provozu v moderních prostředích a end-of-lifecycle technologií
- **integrační** – potřeby na integraci s jinými řešeními

Jednotlivé typy nedostatků budou níže podrobněji popsány a v následujících kapitolách bude z těchto nedostatků sestaven konkrétní seznam požadavků na cílové řešení z nichž bude vytvořen návrh systému.

3.5.1 Externí nedostatky

Nedostatky tohoto typu byly získány na základě zpětné vazby uživatelů v průběhu provozu stávající aplikace.

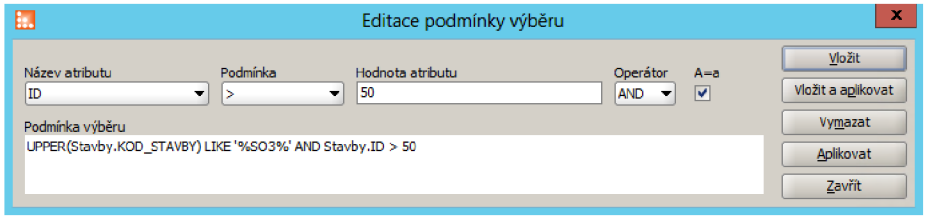


Obrázek 3.7: Hlavní obrazovka modulu v SSK

Ačkoliv uživatelské rozhraní (UI) aplikace je z hlediska použití grafických prvků minimalistické, což napomáhá základní orientaci, určité části vyžadují netriviální znalosti či značnou investici do zorientování. Tyto aspekty značně zhoršují prvotní uživatelskou zkušenost (UX) a vedou ke strmé učící křivce, která negativně ovlivňuje čas nutný k zaučení se v práci se softwarem.

Jedním ze zmiňovaných nedostatků uživatelského rozhraní je způsob zadávání a interakce s filtry. V současném řešení jsou filtry zadávány pomocí dialogového okna, které staví

`WHERE` klauzuli, která je následně přímo použita pro SQL dotaz. Z hlediska uživatelské interakce je problematické, že uživatel tento způsob zadávání provádí v okně, což přidává zbytečnou interakci s UI, jelikož funkcionalita filtrování je nutná pro téměř jakoukoliv práci s přehledy záznamů, a to z důvodu vysokého počtu záznamů. Přímé zadávání filtru bez dialogového je možné ovšem pouze formou ručního vypsání `WHERE` klauzule do prostého textového pole. Zadáváním tímto způsobem nutně předpokládá znalost jazyka SQL, stejně tak interpretace již aplikovaného filtru. Možnost zadávání pomocí SQL oceňují pokročilí uživatelé, ale pro běžného uživatele je tento způsob značně komplikovaný.



Obrázek 3.8: Dialogové okno zadání filtru SSK

Dalším nedostatkem je pak absence možnosti přizpůsobovat a ukládat uživatelsky definovaná zobrazení nad přehledy. Uživatelé z jiných aplikací tuto funkcionalitu již očekávají jako standardní a její absence v SSK snižuje možnosti efektivní práce pokročilých uživatelů.

Uživatelské rozhraní pro práci s 2D a 3D grafickými entitami v současné aplikaci vykazuje problémy v oblasti ovladatelnosti. K efektivnímu ovládání je třeba kombinace myši a klávesnice, která na první pohled nemusí být zřejmá. Rotace je dosažena držením levého tlačítka myši nebo šipkami na klávesnici. Pro posun ve scéně je nutné použít klávesy WASD, posun ovšem neprovádí přesun středu otáčení, což může způsobit snadné ztracení se ve scéně. Podržení pravého tlačítka provádí přiblížení/oddálení dle směru pohybu. Kolečko myši skokově oddálí scénu, uživatelé si stěžují na přílišnou citlivost. Celkově ovládání není uzpůsobeno pro touchpady ani dotyková zařízení.

Současná podoba modulu kabely, kde probíhá komplexní stavové řízení kabelu taktéž vykazuje určité deficity ve zřetelnosti prezentace dat. Existující UI zobrazuje tři typy business objektů: kabel (základní metadata), kabely verze (změnově řízená metadata) a kabely stavy (přiřazení stavů – schvalování apod.). Z pohledu datového modelu se jedná o zcela korektní řešení, ovšem prezentace uživateli v tří-členné hierarchii znesnadňuje porozumění.

3.5.2 Interní nedostatky

Původní autoři systému byli schopni v průběhu času používáním daného softwaru odhalit nedostatky či problémy, které by běžní uživatelé přehlédli nebo by ani neměli šanci je identifikovat, jelikož daná část funkcionality jim nemusí být zpřístupněna.

Zásadním nedostatkem z pohledu interních potřeb zaměstnanců společnosti je roztříštěnost nástrojů podporujících fungování SSK. Tyto nástroje doplňují funkcionalitu samotného systému v oblastech, kde současné technologické řešení nebylo schopno tyto funkcionality integrovat nebo naplnění těchto funkcionalit bylo snazší mít v externích nástrojích. Konkrétně se jedná o hromadné importy/exporty tabulkových dat a DWG/DXF souborů, tvorba pokročilých dotazů apod. I přestože UI poskytuje možnosti pokročilého filtrování nad atributy entit, není možno docílit filtrování nad přivázanými objekty a provádět tzv. projekci dat.

3.5.3 Technologické nedostatky

Z pohledu použitých technologií se jako jeden z hlavních problémů jeví ukončení podpory pro některé využívané technologie. Oracle Database 12c bude mít v 31. červenci 2022 ukončenou podporu, migrace systému na novější je možná, ale není testovaná. Starší verze aplikace dosud provozované na některých lokalitách běží na Java 7, které rozšířená podpora končí taktéž v červenci 2022. Řešení by tedy bylo i nadále provozovatelné při recertifikaci pro nové verze.

V kontextu posledních let a častější práce z domova či vzdálených lokací se uživatelé častěji připojují přes VPN či do virtuálních terminálových stanic. Oba scénáře přinášejí pro architekturu současného SSK určité výzvy. Jak bylo popsáno v kapitole 2.1.1, vzdálenost mezi klientem a serverem, zvláště u dvouvrstvé architektury, má negativní vliv na responzivitu systému. Sekvenční vykonání většího množství SQL dotazů, které je typické pro aplikace tohoto typu, nemusí být z hlediska responzivity problematické, pokud klient i server sdílí jednu interní síť. V případě provozu přes VPN, kde se odezvy namísto maximálně nízkých jednotek milisekund pohybují v řádu desítek milisekund, systém může přestávat reagovat či celková uživatelská zkušenost může být značně degradována. Stejně tak provoz ve virtuálních stanicích může být problematický, pokud tyto stanice jsou provozovány v datovém centru v odlišné lokalitě než je umístěna databáze systému. Tyto problémy jsou přímo aplikovatelné i na současné řešení SSK, kde výše uvedené nedostatky vychází ze zkušenosti provozování tohoto systému v těchto prostředích.

Způsob autentizace přes SQL účty není vhodný pro situace, kde přístupy jsou řízeny skrze centrální politiku, například Active Directory. Administrátoři aplikace musí vytvářet a přidělovat oprávnění přímo přes databázový server. Konkrétně Oracle Database umožňuje napojení na Active Directory až od verze 18c, tedy v době aktivního vývoje stávajícího řešení nebyla tato funkcionalita dostupná.

3.5.4 Integrační nedostatky

Jelikož server je v tomto případě tvořen pouze databází, jediný možný způsob integrace s ostatními databázemi je přes SQL, tedy dotazy, dočasnými tabulkami apod. Ostatní aplikace integrující se na SSK tedy musí podporovat komunikaci s danou verzí Oracle databáze. Odstínění od implementačních detailů databázových struktur je možné dosáhnout tvorbou databázových pohledů s možností zápisu. Možnosti integrace a případné prováděné logiky předzpracování dat jsou limitovány možnostmi procedurálních nástaveb jazyka SQL, v tomto případě PL/SQL.

3.6 Požadavky na cílové řešení

Na základě výše zmíněný nedostatků byl sestaven seznam požadavků na cílové řešení. Požadavky reflektují existující nedostatky, obchodní stránku a dávají rámec, ze kterého následně bude zvolen přístup k reengineeringu u jednotlivých částí řešení.

Název	Kategorie	Způsob ověření
Intuitivní a moderní UI	Nefunkční – UI	Uživatelské testování
Uživatelské rozhraní bude moderní, responzivní a intuitivní.		
Touch-ready UI	Nefunkční – UI	Uživatelské testování
Všechny moduly systému budou podporovat ovládání na dotykových zařízeních		
Výkonnost UI	Nefunkční – UI	Uživatelské testování
Uživatelské rozhraní, zvláště pak modul grafiky, bude provozovatelný noteboocích s integrovaným čipem GPU, tedy běžných klientských stanicích.		
Cloud-ready řešení	Nefunkční – Systém	Návrh
Systém bude provozovatelný v cloudovém kontejnerizovaném prostředí.		
Modularita řešení	Nefunkční – Systém	Validace návrhu
Jednotlivé služby budou nahraditelné pro přizpůsobení prostředí zákazníka a integraci na jeho systémy.		
Zachování databází	Nefunkční – Systém	Validace návrhu
Bude zachován model jedna lokalita – jedna databáze. Databáze budou taktéž provozovány mimo orchestrační platformu.		
Odolnost proti SQL Injection a XSS	Nefunkční – Systém	Penetrační testování
Systém bude odolný proti základním typům útoků.		
Výkonnost systému	Nefunkční – Systém	Uživatelské testování
Nový systém nebude zásadně limitovaný při provozu přes VPN či virtuální stanice.		
Optimistická konkurence	Nefunkční – Systém	Automatizované testování
Systém bude hlídat paralelní modifikace dat prostřednictvím mechanizmu optimistické konkurence.		
Dokumentovaný kód	Nefunkční – Vývoj	Revize kódu
Implementace nového řešení bude řádně dokumentovaná.		
Aplikace návrhových vzorů	Nefunkční – Vývoj	Revize kódu
V případě náhrady částí systému bude ve vhodných situacích využito standardních návrhových vzorů pro implementaci.		
Architektura služeb	Nefunkční – Návrh	Validace návrhu
Cílové řešení bude využívat architekturu služeb.		
Využití metod reengineeringu	Nefunkční – Návrh	Validace návrhu
Pro jednotlivé části bude zváženo, která z metod reengineeringu (náhrada, zaobalení, restrukturalizace) bude použita.		
Ověření skrz Active Directory	Nefunkční – Návrh	Validace návrhu
Navržené řešení bude umožňovat autentizaci s využitím Active Directory.		
Logování	Nefunkční – Návrh	Validace návrhu
Logy z jednotlivých služeb budou možno ukládat v centrálním skladu.		

Tabulka 3.1: Nefunkční požadavky na nové řešení SSK

Název	Kategorie	Způsob ověření
Zjednodušené filtrování	Funkční – UI	Uživatelské testování
Vyhledávat bude možné z jednoho pole napříč celým přehledem. Filtry nebudou vyžadovat znalost SQL pro stavbu pokročilých podmínek.		
Rychlé vyhledávání systémem	Funkční – UI	Uživatelské testování
Vyhledávací pole napříč celým systémem, tzn. obdoba vyhledávače.		
Vícejazyčné UI	Funkční – UI	Uživatelské testování
Bude podporováno vícero jazyků, minimálně čeština a angličtina.		
REST a GraphQL rozhraní	Funkční – Služby	Testování
Systém bude poskytovat REST a GraphQL rozhraní pro napojení a komunikaci s jinými systémy.		
Trasování na pozadí	Funkční – Služby	Testování
Funkcionalita trasování bude dovolovat pokračovat v práci s aplikací v průběhu trasování.		

Tabulka 3.2: Funkční požadavky na nové řešení SSK

3.7 Stanovení cílového řešení

Cílové řešení bude moderní cloud-native systém sestávající se z navzájem propojených mikroslužeb. Služby budou kontejnerizované a proces jejich nasazení bude plně automatizovatelný. Separace služeb bude navrhnuta tak, aby přinášela praktické výhody oproti monolitické architektuře a doručovala konkrétní hodnotu zákazníkovi. Řešení bude taktéž navrhnuto tak, aby je bylo možné provozovat v prostředích, kde nejsou dostupné prostředky orchestrace, a to včetně provozu na lokální stanici. Prezentační vrstva systému bude moderní uživatelsky přivětivá webová aplikace využívající standardní designový jazyk se schopností přizpůsobení se dotykovým, a tedy i mobilním zařízením. Oproti výchozímu řešení nové řešení bude klást důraz na uživatelskou zkušenost a snadnou ovladatelnost. Cílem je zákazníkům poskytnout řešení, které je moderní, adaptovatelné a poskytující celou původní funkcionalitu s vybranými oblastmi, kde funkcionalita bude rozšířena. K doručení tohoto řešení budou použity metody reengineeringu pro snížení celkových nákladů na realizaci řešení. Kde to bude nutné a vhodné z hlediska budoucího rozvoje, bude zvolena metoda náhrady. V ostatních případech bude aplikováno zaobalení na původní řešení.

3.8 Návrh cílového řešení

Na základě stanovených požadavků vycházejících z určeného cílového prostředí, zpětné vazby od zákazníka a definice cílového řešení byl vytvořen návrh cílového řešení, který bude prezentován v následující kapitole. Popis návrhu bude zaměřen na architektonickou stránku a popis vzájemné interakce služeb.

3.8.1 Aplikace reengineeringu

Na základě popisu procesu reengineeringu v kapitole 2.2 lze reengineering SSK charakterizovat jako top-down big-bang. Řešení reengineeringu bylo provedeno top-down, tedy nejprve analýzou business požadavků a postupným návrhem řešením počínaje architekturou a konče

specifikací rozhraní jednotlivých služeb. Výhodou tohoto přístupu je zaměření na business požadavky a celkovou architekturu, což v případě reengineeringu SSK jsou klíčové faktory, jelikož stávající funkcionalita bude nahrazena či zaobalena. Big-bang přístup, tedy jednokrokové nahrazení celého systému, je jediné možné, a to z důvodu současné architektury aplikace. Jelikož aplikační funkcionalita bude přesunuta do služeb, nelze prakticky uvažovat situace, kde by se části stávající aplikace odstraňovaly a nahrazovaly novým řešením. Analýza času a nákladů na možný souběžný provoz starého a nového řešení ukázala, že takový souběh by byl značně náročný a vyžadoval by značné investice.

Na úrovni jednotlivých služeb bude především využito metod náhrady a zaobalení původní funkcionality. Náhrada bude zvolena tam, kde změny technologií přinesou zásadní benefity, které umožní další rozvoj nových funkcionalit. Zaobalení bude zvoleno u funkcionalit, kde se naopak z pohledu nákladů nevyplatí tuto funkcionalitu kompletně přepisovat či ji nelze transformovat.

Reverse engineering aplikace SSK probíhal seznamováním se s funkcionalitou aplikace prostřednictvím práce v testovacím prostředí, studiem technické dokumentace systému a také analytickými diskuzemi s doktorem Hajšmanem. Studiu taktéž byly podrobeny i zdrojové kódy. Pro krok transformace byly zanalyzovány metody automatické konverze/transpilace mezi programovacími jazyky. Tyto metody nejsou pro reengineering SSK vhodné, jelikož produkují ne vždy čistý kód a nebylo by možné využít značné pokroky ve vývoji frameworků a knihoven od data vzniku stávajícího řešení SSK. Fáze forward engineeringu je v případě SSK reprezentována návrhem, implementací a aplikací metod reengineeringu služeb. Tyto aspekty budou v následujících podkapitolách a kapitolách podrobně diskutovány.

3.8.2 Architektura

Pro realizaci cílového řešení Systému správy kabeláže byla zvolena architektura mikroslužeb. Tato architektura dovoluje jasně oddělit jednotlivé funkční komponenty systému do jasně definovaných a oddělených služeb, které mohou být pro různá nasazení systému upraveny či nahrazeny bez nutnosti modifikovat systém jako takový.

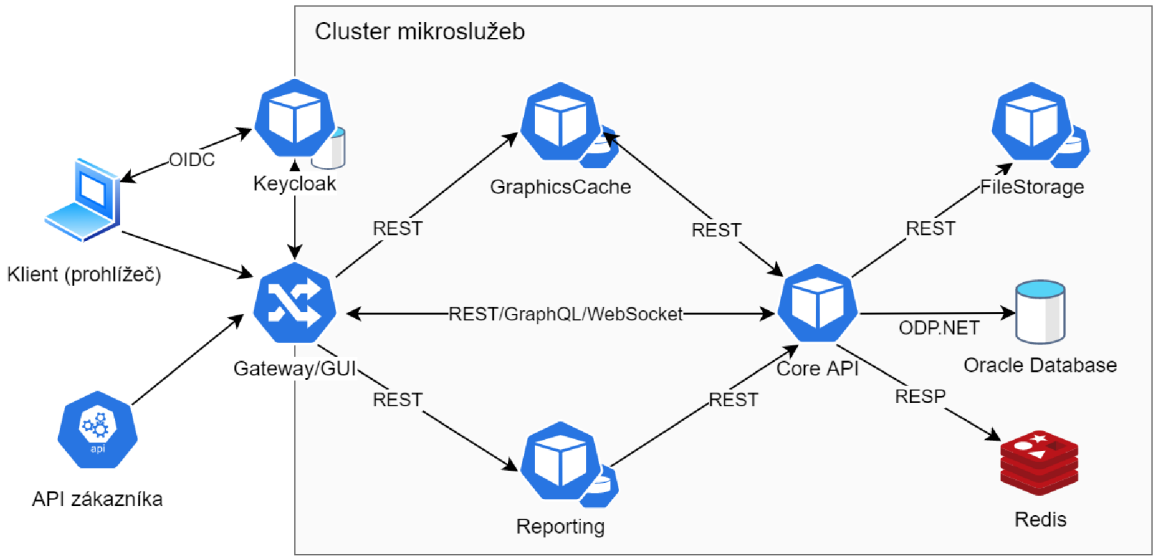
Oproti původně zvažované monolitické třívrstvé architektuře byl tento přístup zvolen z následujících důvodů:

- **vyšší modularita** – přizpůsobitelnost prostředí zákazníka výměnou některých služeb
- **diferenciace technologií** – rozdílné technologie pro implementaci funkcionalit
- **škálovatelnost** – vybraná funkcionalita může být nezávisle škálována
- **zaobalení** – snazší zaobalení původních funkcionalit systému

Celková změna architektury poskytuje tedy zásadní výhody, které se přímo promítají do možností dalšího rozvoje systému a jsou v souladu business strategií pro produkt SSK. Změna cílové architektury byla konzultována s vedoucím této práce.

Základním konceptem návrhu cílového řešení je rozdělení funkcionalit do služeb. Tyto služby budou nahrazovat či standardizovat funkcionalitu přítomnou ve stávajícím řešení. Komunikace mezi službami bude probíhat pomocí REST a GraphQL nad HTTP, případně s využitím WebSocket protokolu[16] pro zasílání průběžných informací o stavu. Jak zmiňuje kapitola 2.1.3, využití metodiky REST neznamená nutnost používat HTTP protokol,

[16] https://datatracker.ietf.org/doc/html/rfc6455



Obrázek 3.9: Architektura cílového řešení SSK

ale právě HTTP nabízí prostřednictvím svých metod ideální vazbu pro mapování operací REST. Komunikace pomocí WebSocketů bude využita tam, kde bude vyžadováno real-time zasílání zpráv mezi službou a klientem, konkrétně pak při reportování stavu algoritmu trasování. Model komunikace mezi službami bude synchronní, nebude tedy využito message brokerů. Komunikace v rámci navrhovaného řešení bude nanejvýš zahrnovat tři služby – Gateway, Core API a služba naplňující business funkcionalitu.

Cílové řešení bude obsahovat následující služby:

- **Core API** – centrální API poskytující CRUD[17], autorizaci, trasování a obecné business operace nad centrální Oracle databází
- **Gateway** – výchozí brána zastřešující ostatní služby a poskytující soubory pro GUI
- **Keycloak** – služba implementující OpenID Connect autentizaci prostřednictvím řešení Keycloak
- **Graphics** – ukládá a distribuuje zoptimalizované 3D grafické objekty a příslušná metadata
- **Reporting** – služba poskytující API pro generování reportů
- **FileStorage** – zapouzdřuje operace nad skladem dokumentů a souborů
- **Oracle databáze** – centrální databáze pro ukládání dat SSK
- **Redis** – Redis cache pro podporu škálování

Všechny služby budou dokumentované OpenAPI specifikací[18], všeobecně známou jako *Swagger*. Dokumentace rozhraní bude klíčová pro případné integrace na zákaznické systémy. Existence specifikace také usnadní vývoj klientské aplikace SSK.

[17] Create, Read, Update, Delete (CRUD) – základní datové operace
[18] https://swagger.io/resources/open-api/

Součástí řešení bude i Redis[19] cache, která bude sloužit jako podpora při škálování v clusterech. Redis je *in-memory* databáze pro cachování dat. Této funkcionality bude využito pro komunikaci mezi běžícími instancemi stejné služby, viz bližší popis v kapitolách 3.8.2, 4.5.4 a 4.3.

Logování a monitoring

Nové řešení SSK bude klást bude důraz na auditovatelnost a monitoring stavu služeb a datových operací. Cílem nového řešení je centralizovat logování do jednoho místa, aby bylo možné s minimální pracností identifikovat problémy. Všechny kontejnery služeb bude možné instrumentovat sondami, které budou reportovat využití prostředků a živost daného kontejneru. Služby svoje výstupy budou zapisovat minimálně na `stdout` a chyby na `stderr`. Návrh, nastavení a dodání služeb logování a monitoringu není součástí dodávky SSK ani této diplomové práce, jelikož se jedná o řešení centralizovaná na úrovni daného clusteru. SSK jako řešení bude implementováno tak, aby bylo integrovatelné s těmito řešeními.

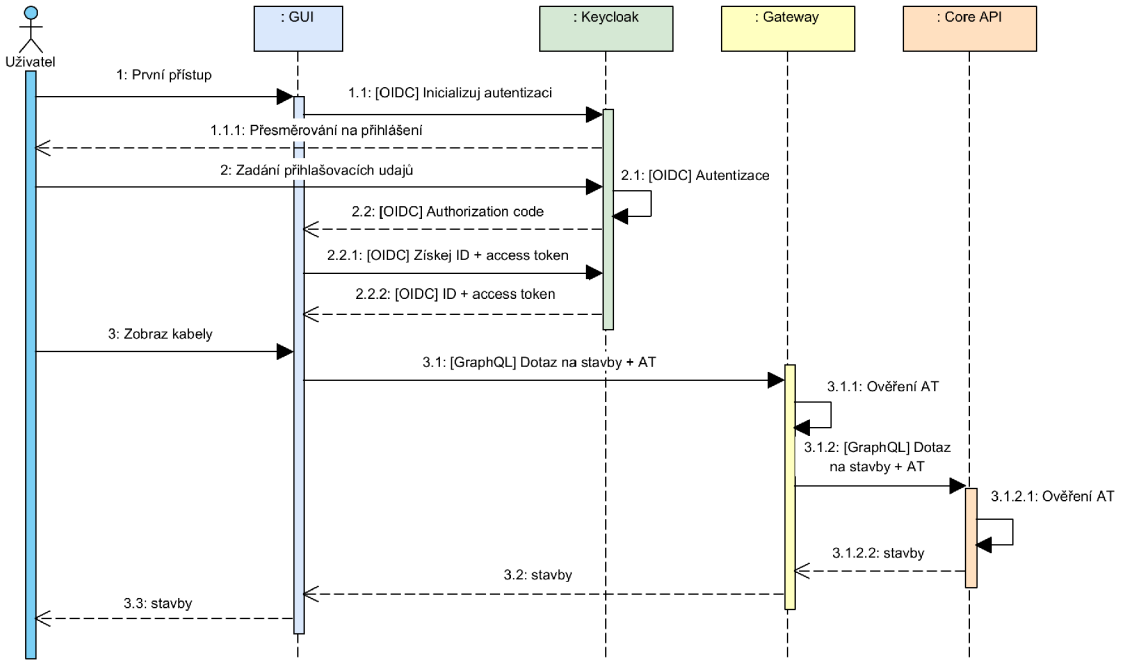
Autentizace a autorizace

Koncept autentizace v novém řešení bude využívat protokolu OpenID Connect[20]. Služba autentizace bude reprezentována řešením Keycloak[21], které poskytuje prostředky správy identit a přístupů. Služba Keycloak bude v produkčním prostředí pravidelně synchronizovat svoji databázi uživatelů s Active Directory prostřednictvím LDAP protokolu. Do databáze Keycloak se budou synchronizovat relevantní AD atributy. Synchronizace hesla nebude prováděna, jelikož jeho ověření se bude provádět přímo vůči LDAP serveru. Autentizace bude využívat access tokeny ve formě JSON Web Tokenu (JWT)[22] pro přenos *user claims*. Součástí výstupů této práce nebude konfigurace napojení na Active Directory.

Výhodou tohoto řešení autentizace je odstínění služeb (v případě SSK se jedná o Gateway) od konkrétních specifik implementace autentizace. Při rozdílných způsobech autentizace napříč vícero prostředími není nutné modifikovat služby, které tuto autentizaci vyžadují. Vypořádání se s různými prostředími je pak řešeno na straně služby Keycloak, kde administrátoři mohou případně provádět plnohodnotnou správu uživatelů. V konkrétním případě SSK nebude na straně služby Keycloak prováděna žádná administrace, uživatelé budou pouze automaticky synchronizováni. Vytvořením oddělené autentizační služby bude taktéž možné docílit jejího znovupoužití v jiných projektech/řešeních.

Autorizace v rámci řešení SSK bude prováděna na službě Core API, která bude ukládat mapování uživatelů na role. Obecně bude moci být uživateli přiřazeno více rolí. Ke každé roli bude možné definovat sadu konkrétních oprávnění. V prvotní fázi nebudou role synchronizovány s Keycloak claims (potažmo Active Directory rolemi), do budoucna se počítá s volitelnou konfigurovatelnou synchronizací. Autor práce si je vědom nevýhod spojených s oddělením autentizace (Keycloak) a autorizace (Core API). Důvodem, který stojí za tímto návrhem, je fakt, že administrace uživatelských práv bývá typicky prováděna administrátory aplikace a nikoliv administrátory Active Directory apod. Je možné se zamýšlet na bezpečností a spravovatelností takových prostředí, ale z hlediska naplnění požadavků zákazníka se jedná o nezbytné rozhodnutí.

[19] https://redis.io/
[20] https://openid.net/connect/
[21] https://www.keycloak.org/
[22] https://datatracker.ietf.org/doc/html/rfc7519



Obrázek 3.10: Sekvenční diagram OpenID Connect autentizace v SSK

Služba Gateway

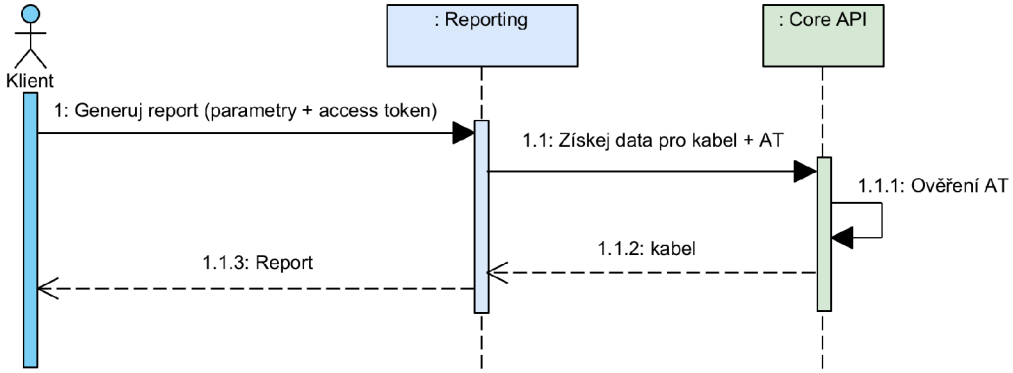
Gateway zastřešuje služby ekosystému mikroslužeb pod jednotné rozhraní – API. Cílem tohoto API je odstínit interní akce služeb od veřejného rozhraní. Sjednocené rozhraní také zjednoduší implementaci klientských aplikací. V případě SSK bude Gateway komunikovat se službami Core API, Reportingu a Graphics. Tyto služby a jejich části společně tvoří veřejné rozhraní SSK. Rozhraní bude dokumentované OpenAPI specifikací. Specifikace interních služeb budou transformovány a publikovány na službě Gateway. Pro volání služby bude vyžadována autentizace access tokenem, který bude získán ze služby Keycloak protokolem OIDC. Gateway tento token ověří a zašle připojeným službám s požadavkem. Obrázek 3.10 znázorňuje popsanou sekvenci.

Tato služba bude taktéž klientům distribuovat soubory pro vykreslení uživatelského rozhraní. Jelikož se bude jednat o statické soubory klientské aplikace, není na spojení služby Gateway s poskytováním uživatelského rozhraní nahlíženo jako problematické z pohledu minimalizace velikosti služeb.

Služba Reporting

Služba Reportingu bude poskytovat prostředky návrhu a generování uživatelských reportů. Součástí služby bude REST API, které bude umožňovat úpravu šablon reportů a také jejich generování nad daty. Napojení na datové zdroje bude realizováno prostřednictvím integrace na Core API, které bude poskytovat všechny potřebné end-pointy pro vytváření takových reportů. Služba nebude obsahovat žádnou komplexnější nadstavbovou logiku nad reporty, bude se tedy jednat o jednoduchou službu (z pohledu velikosti rozhraní) pro tvorbu reportů. Z pohledu reengineeringu je tato služba *náhradou* existující funkcionality.

Obrázek 3.11 znázorňuje proces získání reportu pro daný kabel. Účastníky této sekvence je Reporting a Core API. Aktérem, který může tuto sekvenci vyvolat, může být prostřed-



Obrázek 3.11: Sekvenční diagram získání reportu

nictvím Gateway přímo uživatel, případně externí API. Každý report bude parametrizován access tokenem aktuálního uživatele, na základě kterého budou vyhodnocena práva pro zobrazení obsahu reportu. Tímto je zaručeno, že uživatel si nezobrazí report na žádná data, která by jinak nemohl nahlížet skrze klientskou aplikaci.

Služba bude ukládat šablony do souborového systému. Uložené šablony budou mechanismem tzv. *volumes* trvale uloženy mimo běžící kontejnery/pody dané služby. Varianta ukládání do databáze byla zvažována, ale z důvodů zvýšené komplexity v už tak komplexním systému bylo přistoupeno na řešení skrze volumes. Jelikož se bude jednat o jednotky reportů, jejichž úprava bude spíše ojedinělá, je varianta souborového systému vhodnější. Příkladem reportů generovaných touto službou může být například kladečský list (posloupnost segmentů tras pro pokládku daného kabelu) či report o trasování (přehled výstupu trasování). Pro implementaci této služby bude vybrán vhodný komerční reportovací systém.

Služba Core API

Core API bude centrální službou SSK. Jejím úkolem bude poskytovat rozhraní nad centrálním datovým skladem – databází Oracle – pro realizaci operací CRUD a business funkcionality. Ostatní služby SSK i klienti budou využívat tuto službu pro přístup k primárním datům.

Kromě této zmíněné funkcionality bude služba poskytovat i autorizační endpointy. Vyčlenění této funkcionality do samostatné služby by přinášelo značné problémy při možné budoucí implementaci[23] objektových oprávnění[24]. Integrací Core API bude možno na úrovni datového modelu/databáze zajistit referenční integritu mezi uživateli a metadaty o provedených změnách. Core API bude udržovat seznam uživatelů, ve kterém bude možné editovat pouze SSK specifické atributy a vazby. Nebude tedy možné uživatele zakládat ani mazat a nebude dovolena editace jména, emailu, uživatelského jména apod. Při odstranění uživatele na straně Keycloaku (nebo deaktivaci na AD) bude možné uživatele v seznamu označit za neplatného, tudíž jím provedené činnosti budou i nadále evidovány. Uživatel bude moci být ve více rolích, přičemž role bude sdružovat sadu oprávnění, kde oprávnění bude představovat právo vykonat konkrétní činnost, tedy např. čtení seznamu kabelů. Pro

[23] O této funkcionalitě se uvažuje a z pohledu návrhu autorizace je tento požadavek zásadní.
[24] Oprávnění na úrovní konkrétního záznamu.

čtení a evaluaci oprávnění budou vytvořeny speciální zjednodušené endpointy navrhnuté pro použití v jiných službách.

Název	**HTTP metoda a URL**
Metadata	`GET /viewmodels/{viewmoldel}/metadata`
Získá metadata (popis) viewmodelu včetně výpisu jeho properties s datovými typy.	
Seznam	`GET /viewmodels/{viewmoldel}/list?p1=p1&...`
Vrátí seznam vyfiltrovaných a seřazených záznamů na základě query parametrů.	
Export seznamu	`GET /viewmodels/{viewmoldel}/export?p1=p1&...`
Exportuje seznam vyfiltrovaných a seřazených záznamů na základě query parametrů.	
Získej záznam	`GET /viewmodels/{viewmoldel}?id1=id1&id2=id2...`
Vrátí záznam odpovídající mající jednoduchý/kompozitní primární klíč.	
Vytvoř záznam	`POST /viewmodels/{viewmoldel}`
Vytvoří záznam na základě těla požadavku. Vrátí vytvořený záznam.	
Uprav záznam	`PUT /viewmodels/{viewmoldel}?id1=id1&id2=id2...`
Upraví záznam mající jednoduchý/kompozitní primární klíč na základě těla požadavku. Vrátí upravený záznam.	
Smaž záznam	`DELETE /viewmodels/{viewmoldel}?id1=id1&id2=id2...`
Smaže záznam mající jednoduchý/kompozitní primární klíč.	
Našeptávač	`GET /viewmodels/{viewmoldel}/{propertyName}?p1=p1&...`
Vrátí našeptávané hodnoty pro `propertyName` odpovídající query parametrům.	
Backend akce	`PATCH /viewmodels/{viewmoldel}/backendAction`
Vykoná business akci. Název a parametry akce budou zaslané v těle požadavku, vrátí formátovanou odpověď. Akce má vedlejší efekt uložení upravených do datového skladu.	
Frontend akce	`PATCH /viewmodels/{viewmoldel}/frontendAction`
Vykoná business akci. Název a parametry akce budou zaslané v těle požadavku, vrátí formátovanou odpověď. Akce nemá vedlejší efekt do datového skladu.	
GraphQL	`POST /viewmodels/graphql`
GraphQL endpoint. Dotaz je zaslán v těle požadavku.	
Je v roli	`GET /user/has-permission/{tableName}/{permission}`
Ověří, jestli aktuální uživatel má právo `permission` pro tabulku `tableName`.	
Hub trasování	`WS /ws/v1/tracing`
Reportuje stav trasování spuštěného aktuálním uživatelem. Komunikuje protokolem WebSocket.	

Tabulka 3.3: Návrh vybraných endpointů rozhraní služby Core API

Jelikož základní datový model SSK (stavby, trasy, terminály, kabely) je mezi sebou úzce provázán, rozdělení této funkcionality mezi vícero služeb/databází by přineslo značné překážky v možnostech vzájemného referencování a zajištění konzistence. Současný datový model implementuje vazby statické (tedy vazby realizované formou cizích klíčů). Varianta dynamických vazeb, kde typ vazby je uložen jako dato, byla taktéž zvažována, ale z důvodů extrémních zásahů do datového modelu byla tato varianta prozatím odložena. Z hlediska použití je SSK primárně read-only systém (zápisy nejsou časté), tudíž existence jedné centrální služby se z pohledu výkonu nejeví jako nedostatek, zvláště když je službu možné v prostředí cloudu škálovat. Lze uvažovat, že pokud by se do budoucna tato situace změnila a výkon služby/databáze by byl nedostatečný, bylo by možné tuto službu dále rozmělnit

po hranicích současných modulů. Realizace tohoto rozdělení by však nebyla triviální a musel by tedy existovat závažný business případ, který by právě navrhovaným řešením nebyl realizovatelný.

Služba bude poskytovat REST rozhraní pro plnou práci s business entitami SSK. Pro každou business entitu bude existovat akce pro její čtení, vytvoření, úpravu a smazání. V kontextu implementace bude tato entita reprezentována *viewmodelem*. Viewmodel je v tomto případě veřejným rozhraním business objektu, které kromě vlastností taktéž publikuje specifickou business funkcionalitu implementovanou nad viewmodelem. Viewmodel typicky odpovídá jedné databázové tabulce či pohledu.

Core API bude taktéž zpřístupňovat endpointy pro práci s trasováním. Algoritmus trasování, který i nadále zůstane implementován v PL/SQL procedurách, bude zapouzdřen do modulu služby Core API. V kontextu reengineeringu se bude jednat o *zaobalení* původní funkcionality. Trasování bude možné spouštět prostřednictvím REST API. Jelikož algoritmus trasování je spouštěn po dávkách (tedy skupinách kabelů), může se doba vykonání této dávky pohybovat od sekund po hodiny. Je tedy nutné uživatele z hlediska UX informovat o průběhu trasování. Informace o průběžných výsledcích a dokončení zpracování dávky budou zpřístupněny přes WebSocket rozhraní. Klientská aplikace bude moci odposlouchávat aktuální stav a notifikovat uživatele o dokončení.

Pokročilá validace dat obsluhovaných Core API bude probíhat na úrovni databáze, stejně jako v původním řešení. Součástí DB jsou a zůstanou pravidla a validační logika změn atributů entit. Příkladem takové logiky může být např.: podmíněné hlídání referenční integrity, tvorba verze kabelu apod. Vzhledem ke komplexnosti této logiky, která by vyžadovala značné úsilí na reimplementaci, bude přistoupeno k zaobalení. Core API bude zprávy triggerů odchytávat a srozumitelně je uživateli prezentovat. V rámci budoucího rozvoje bude uvažováno o přesunu a reimplementaci této logiky do Core API.

```
type Query {
  me: User
}

type User {
  id: ID
  name: String
}
```

Výpis (3.1) Definice schématu GraphQL

```
{
  me {
    name
  }
}
```

Výpis (3.2) Příklad GraphQL dotazu

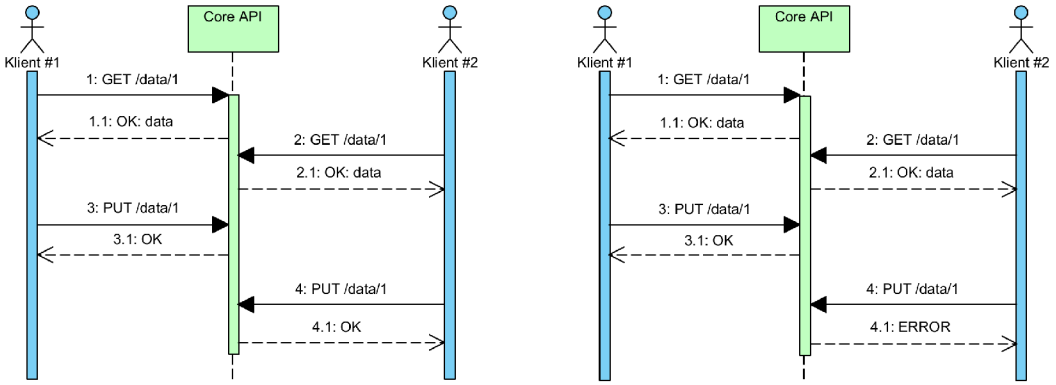
```
{
  "me": {
    "name": "Michal"
  }
}
```

Výpis (3.3) Příklad GraphQL odpověďi

Obrázek 3.12: Ukázka GraphQL

Nad rámec standardního REST rozhraní bude služba poskytovat i rozhraní GraphQL[25]. GraphQL je dotazovací jazyk pro API umožňující selekci, aktualizaci a především projekci dat. GraphQL pro popis dotazovatelných dat definuje *schéma*, v rámci kterého jsou dále popsány *typy* reprezentující jednotlivé typy požadavků/odpovědí. Typ se skládá z pojmenovaných *polí*. Datový typ pole pak může být jeden z vestavěných skalárních typů[26] nebo uživatelsky definovaný typ[27]. Na základě tohoto typového systému je možné provádět projekci polí ze zanořených typů. Právě tato funkcionalita zanořeného dotazování je klíčovou

[25] https://graphql.org/
[26] Typ, u kterého nelze specifikovat podvýběr.
[27] Rekurzivní zanoření typů je podporováno.



(a) Paralelní editace bez řešení souběhu (b) Paralelní editace s optimistickým zamykáním

Obrázek 3.13: Porovnání paralelní editace bez/s řešením souběhu

výhodou pro datový model SSK. Poskytnutí mechanizmu, kde klienti si sami určí, která data potřebují, zbavuje Core API nutnosti mít přítomny specifické endpointy pro tyto klienty. API tedy může být jednotné a obecné. GraphQL rozhraní SSK nebude podporovat modifikace (mutace) dat. Z pohledu modifikace zcela dostačuje REST rozhraní a implementace mutací by přinášela zvýšenou komplexitu implementace s nevýznamnými či žádnými přínosy. Zároveň taktéž lze říci, že tento způsob dodržuje princip CQRS popsaný v kapitole 2.1.3.

Jelikož se jedná o centrální službu pro datový model SSK, je nutné uvažovat situace paralelní práce s jedním záznamem, a to buď mezi uživateli, nebo souběhu mezi službou a uživatelem, nebo exkluzivně službami. Při této paralelní editaci je nutné zabránit ztrátě dat, která by byla způsobena výhrou (přepsáním provedených změn) posledního ze série zápisů. Core API bude v těchto situacích aplikovat metodu optimistického zamykání záznamu, kde před každou aktualizací bude zkontrolováno, že daný klient pracuje se záznamem, který nebyl od doby čtení klientem pozměněn. Pokud byl záznam změněn, bude zabráněno aplikaci nově žádané změny. Tento přístup je v kontrastu s pasivním zamykáním, kdy klient obdrží exkluzivní zámek pro aktualizaci dat. Optimistické zamykání je pro SSK vhodnější, jelikož záznam může mít pro editaci otevřeno mnoho uživatelů a ne všichni se rozhodnou pro provedení změn. Podobně tak uživatel může od stanice odejít nebo může ztratit spojení. Optimistické zamykání bude implementováno porovnáním *hash* původního a aktuálního záznamu. Hash původního záznam je takový hash, který byl vypočten v době zaslání záznamu klientovi. Tento hash bude společně s požadovanou modifikací zaslán na Core API. Alternativy v podobě inkrementace čítadla či přímého porovnání sloupců byly taktéž zvažovány, ale byly vyloučeny pro horší flexibilitu či komplikovanější implementaci. Použití databázového verzování je typicky silně závislé na konkrétním DBMS systému, u Oracle se jedná o `ORA_ROWSCN`[28] a u SQL Serveru o `rowversion`. Vázáním se na tyto specifické funkcionality by se do budoucna ztěžovala možná migrace na jiné DBMS[29].

Připojení k databázi bude specifikováno URL parametrem `database`, který bude součástí každého požadavku. Hodnota parametru bude název jedné z předdefinovaných databází, konkrétní řetězec pro připojení bude součástí konfigurace Core API. Současný model

[28] https://docs.oracle.com/en/database/oracle/oracle-database/21/sqlrf/ORA_ROWSCN-Pseudocolumn.html

[29] V kontextu této práce změna DBMS není požadovaná ani chtěná, ale navrhované řešení se snaží být DBMS-agnostickým pro budoucí možné změny.

Název	**HTTP metoda a URL**
Získej 3D data	`GET /exporter/graphics/{database}/{kod}`
Vyvolá proces získání dat a vrátí komprimovaná 3D data pro stavbu identifikovanou `kod` z databáze `database`.	
Získej popisky	`GET /exporter/labels/{database}/{kod}`
Vyvolá proces získání dat a vrátí popisky pro stavbu identifikovanou `kod` z databáze `database`.	
Získej metadata	`GET /exporter/metadata/{database}/{kod}`
Vyvolá proces získání dat a vrátí metadata pro stavbu identifikovanou `kod` z databáze `database`.	
Obnov stavbu	`GET /exporter/internal/refreshAll/{database}/{kod}`
Vyvolá obnovení 3D dat, metadat a popisků pro stavbu identifikovanou `kod` z databáze `database`.	
Ob. všechny st.	`GET /exporter/internal/refreshAll/{database}`
Vyvolá obnovení 3D dat, metadat a popisků pro všechny stavby z databáze `database`.	
Ob. chybějící st.	`GET /exporter/internal/refreshAll/{database}`
Vyvolá obnovení 3D dat, metadat a popisků pro všechny stavby z databáze `database`, ke kterým neexistuje záznam v cache.	

Tabulka 3.4: Návrh rozhraní služby Graphics

jedna lokalita – jedna databáze bude zachován a stejně tak i možnost připojení jedné aplikace (v tomto případě služby) na vícero databází[30].

Služba Graphics

Tato služba vytváří komprimované a optimalizované 3D objekty pro schématické zobrazení na základě dat Core API. Potřeba pro toto API vznikla z faktu, že získání všech dat pro vykreslení tras, terminálů a bodů u velkých staveb (reaktor, strojovna apod.) je časově a prostředkově náročná operace[31]. Načtením, sestavením a komprimováním těchto dat v oddělené službě s využitím cachovacích mechanizmů je možno dosáhnout značné časové a paměťové úspory oproti řešení v klientovi – prohlížeči.

Vznik této služby je důsledkem reengineeringu modulu grafiky metodou *náhrady* stávajícího řešení. Služba je nutná pro naplnění nefunkčních požadavků na modul grafiky.

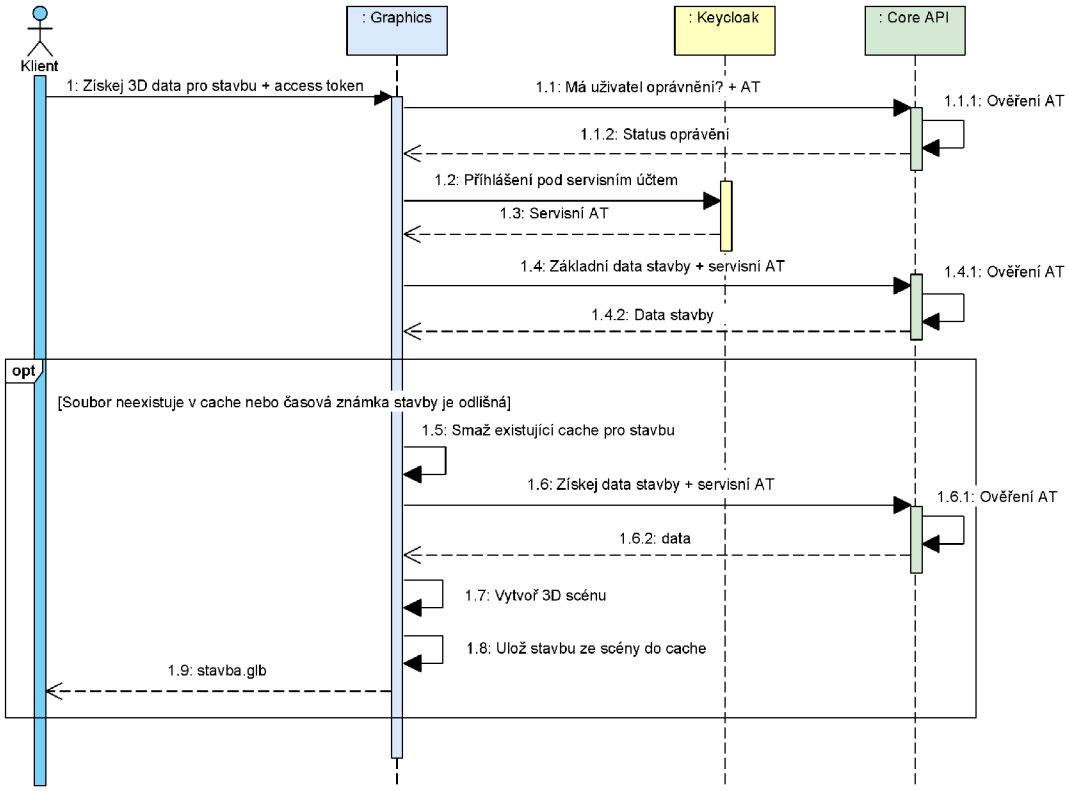
Databáze Oracle v současné i cílové verzi bude ukládat grafické objekty v podobě kolekce bodů. Graphics tyto data extrahuje pomocí příslušného GraphQL dotazu a získá je ve formátu JSON. Z dat bude sestaven 3D objekt, který bude následně exportován, optimalizován a komprimován. Cílem této *pipeline* je vylepšit renderovací výkon, zátěž paměti a záběr místa na disku při ukládání. Zkomprimovaný model následně bude uložen do souborového systému služby, která bude pomocí *volume* sdílena do perzistentního úložiště mimo kontejner.

Data budou agregována do 3D objektů podle staveb. Do budoucna budou objekty děleny i podle typu (stavební objekty, trasy, terminály), jelikož frekvence změn těchto typů je odlišná. Lze předpokládat, že stavební dispozice budou měněny minimálně, méně častá bude úprava terminálů a nejčastější bude editace tras.

[30] Tato funkcionalita je samozřejmě podmíněna kompatibilitou databáze a verze služby.

[31] Řádově vysoké desítky vteřin a desítky MB dat.

Mimo grafické objekty bude služba poskytovat i doprovodná data přímo související s danou stavbou potřebné pro efektivní práci v modulu grafiky. Konkrétně se jedná o popisky v rámci objektu a základní informace o zobrazených business objektech. Popisek je 2D textem v rovinně definovaný textem popisku a příslušnou souřadnicí. Popisky budou seskupeny podle kategorií business objektu, kterému náleží. Základní informace o business objektech budou sloužit pro identifikaci a zobrazení relevantních dat při uživatelské interakci nad modelem stavby, a to například při propichu stavbou. Cachováním těchto dat bude zvýšena rychlost načtení dat do modulu grafiky.

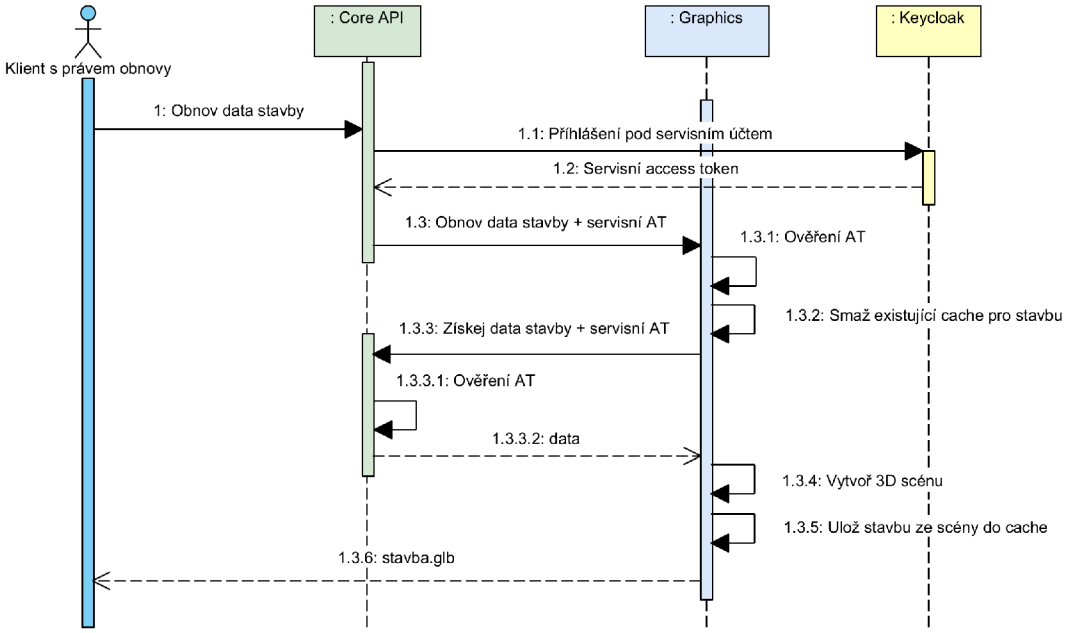


Obrázek 3.14: Sekvenční diagram získání 3D dat

Služba při každém požadavku zkontroluje případné změny od doby posledního vygenerování vůči Core API. V případě, že nastala změna, bude spuštěna výše popsaná pipeline, jejíž výstup bude kromě zapsání na disk okamžitě zaslán jako odpověď. V tomto případě bude doba obsluhy požadavku ekvivalentní době obsluhy bez přítomnosti cache. Pro následující požadavky už bude využito nacachovaných dat.

Pro zobrazení dat bude vyžadováno explicitní uživatelské právo. Jeho vlastnictví bude ověřeno u služby Core API. Pro získání dat z Core API při neexistenci dat v cache se služba Graphics přihlásí vůči Keycloak servisním účtem, který ji na úrovni Core API oprávní k roli administrátora, tudíž i ke čtení dat staveb, tras či terminálů.

Služba bude taktéž podporovat i obnovu již cachovaných dat. Tyto akce budou spustitelné jenom pod servisním účtem. Jejich volání bude prováděno z Core API, kde mohou být iniciovány na základě požadavku uživatele s příslušným oprávněním, či v budoucnu i automaticky plánované akce obnovy bez kontextu uživatele. Diagram na obrázku 3.15 znázorňuje výše popsaný proces.



Obrázek 3.15: Sekvenční diagram obnovy cache stavby

Služba FileStorage

Služba FileStorage slouží jako jednoduchý *document management system* (DMS) pro potřeby SSK. Oddělením této funkcionality do služby je dosaženo flexibility z hlediska nasazení v různých zákaznických prostředích, kde zákazník typicky klade požadavky na uschování a správu dokumentů ve vlastním DMS řešení. Tato služba tedy u zákazníků majících vlastní DMS (např. IBM FileNet[32]) bude řešena jako wrapper nad API poskytnutým daným DMS. U zákazníků, kde DMS není přítomno, nebo zákazník nevyžaduje centralizaci dokumentů, bude služba využívat souborový systém pro ukládání těchto dokumentů. Dokumenty budou mechanismem tzv. *volumes* trvale uloženy mimo běžící kontejnery/pody této služby. API se nesnaží být plnohodnotným DMS, jelikož jeho účelem je pouze být minimálním rozhraním do perzistentního dokumentového úložiště.

FileStorage služba je *náhradou* existující funkcionality pro připojování souborů.

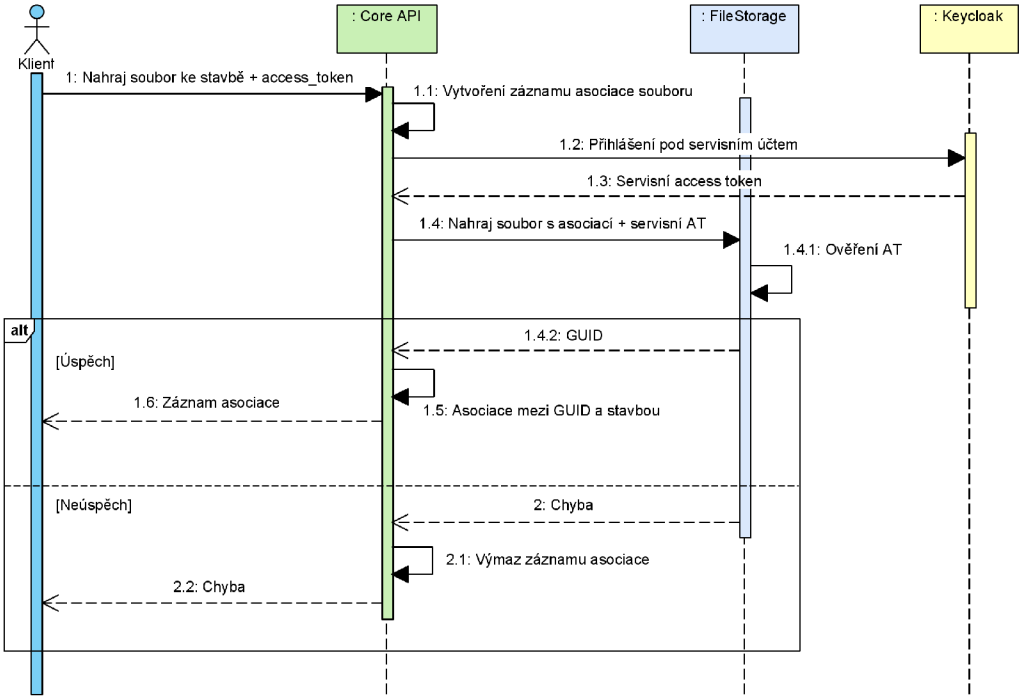
Z pohledu SSK budou dokumenty odlišeny dle typu entity/business objektu, např.: stavba, řez, kabel apod. Ke každé entitě bude možné získat seznam přiřazených souborů. Tento seznam bude obsahovat metadata souborů, např.: identifikátor, název, formát, velikost, autor, datum změny apod. Tyto informace bude možné získat pro každý soubor jednotlivě. Funkcionalita FileStorage bude naplňovat minimální požadavky pro SSK, tedy vyhledání, nahrání a smazání souboru.

Pro získání souboru ze služby FileStorage bude Core API využívat servisní účet, jehož access token služba získá přihlášením vůči Keycloak. Nahrávání nových souborů bude probíhat vytvořením záznamu o asociaci souboru s entitou SSK. Tento záznam bude obsahovat metadata nahrávaného souboru, která budou zobrazována v uživatelském rozhraní SSK. Jedná se například o název souboru, jeho typ, a taktéž i unikátní identifikátor (GUID – *Globally unique identifier*) souboru ze služby FileStorage, který bude zapsán při úspěšném uložení ve službě FileStorage. V případě neúspěchu bude celý záznam asociace smazán.

[32] https://www.ibm.com/cz-en/products/filenet-content-manager

Název	**HTTP metoda a URL**
Seznam souborů	`POST /fileStorage/{database}/findAll`
Vrátí seznam metadat souborů příslušící databázi `database` vyhovující dotazu v těle požadavku.	
Metadata souboru	`POST /fileStorage/{database}/findOne`
Metadata nalezeného souboru příslušící databázi `database` vyhovující dotazu v těle požadavku.	
Data souboru	`GET /fileStorage/{database}/{fileId}/file`
Vrátí data souboru identifikovaného `fileId` příslušící databázi `database`.	
Nahraj soubor	`POST /fileStorage/{database}`
Nahraje nový soubor. Vrátí GUID vytvořeného záznamu.	
Smaž soubor	`DELETE /fileStorage/{database}/{fileId}`
Smaže soubor identifikovaný `fileId` příslušící databázi `database`.	

Tabulka 3.5: Návrh rozhraní služby FileStorage



Obrázek 3.16: Sekvenční diagram nahrání souboru službou FileStorage

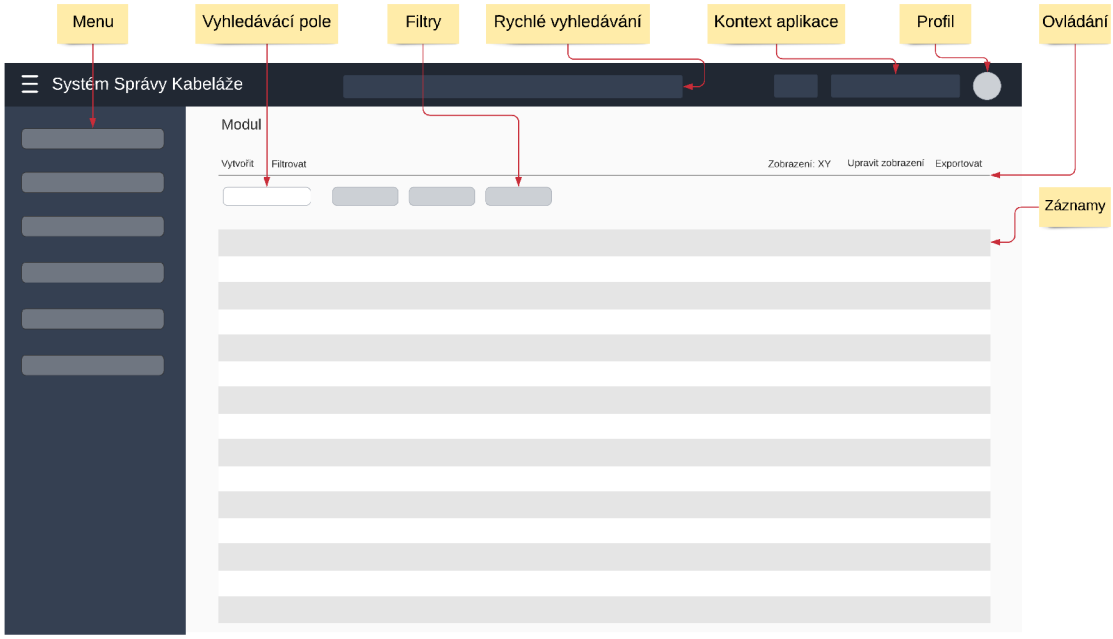
3.8.3 Uživatelské rozhraní

Uživatelské rozhraní bude implementováno jako tzv. *progressive web application* (PWA). Jedná se tedy o webovou aplikaci, kterou bude možné instalovat přímo do operačního systému skrze podporované prohlížeče[33]. Nainstalovaná aplikace je v takovém případě tenký obal nad danou webovou aplikací. Režim PWA obecně podporuje i práci offline, ale ta

[33]V době psaní této práce se jedná o Safari a prohlížeče založené na Chromiu – Google Chrome, Microsoft Edge atd.

nebude pro SSK využita, jelikož SSK je datově intenzivní aplikace, kde téměř celá business funkcionalita existuje pouze na straně API.

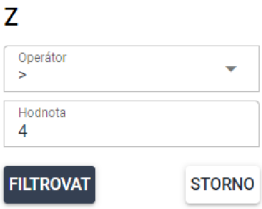
Rozhraní bude moderní, vícejazyčné, responzivní a s podporou dotykových zařízení. Ovladatelnost bude reflektovat nefunkční požadavky popsané v tabulce 3.1. Kde to bude vhodné, bude UI obsahovat názorné ikony pro lepší zřetelnost. Z hlediska funkcionality bude uživatelské rozhraní kopírovat akce Core API. Součástí uživatelského rozhraní bude taktéž i vykreslování 3D grafiky pro naplnění funkcionality modulu grafiky. 3D grafika bude využívat hardwarové akcelerace WebGL. Pro vykreslování grafiky bude datovým zdrojem služba Graphics. Do rozhraní taktéž budou taktéž integrovány komponenty náhledu a designu reportů.



Obrázek 3.17: Wireframe obrazovky modulu

Uživatelské rozhraní aplikace je členěno na vrchní navigační lištu, postranní panel a část obsahu. Navigační lišta obsahuje pole rychlého vyhledávání. Pole umožňuje vyhledávat napříč systémem pomocí jediného řetězce. Výsledky budou asynchronně načítány do interaktivního našeptávače. Tento našeptávač bude data strukturovat dle modulů a bude umožňovat rychlý přechod do modulu. Navigační lišta dále bude obsahovat přepínače kontextu aplikace (databáze, provozovaný/projektovaný stav) a odkaz do profilu, kde uživatel bude moci měnit své preference. Postranní panel v levé části obrazovky bude obsahovat menu se seznamem modulů aplikace. Panel bude možno přepínat mezi třemi stavy: skrytý, zobrazený a mini. Přepínání bude obslouženo tlačítkem v horní navigační liště. V režimu mini bude panel pouze zobrazovat ikony příslušných modulů. Po interakci s mini panelem bude panel odkryt. Výchozí podoba panelu bude měněna v závislosti na velikosti obrazovky (tedy na mobilním zařízení skrytý) na tabletech režim mini a na velkých obrazovkách odkrytý.

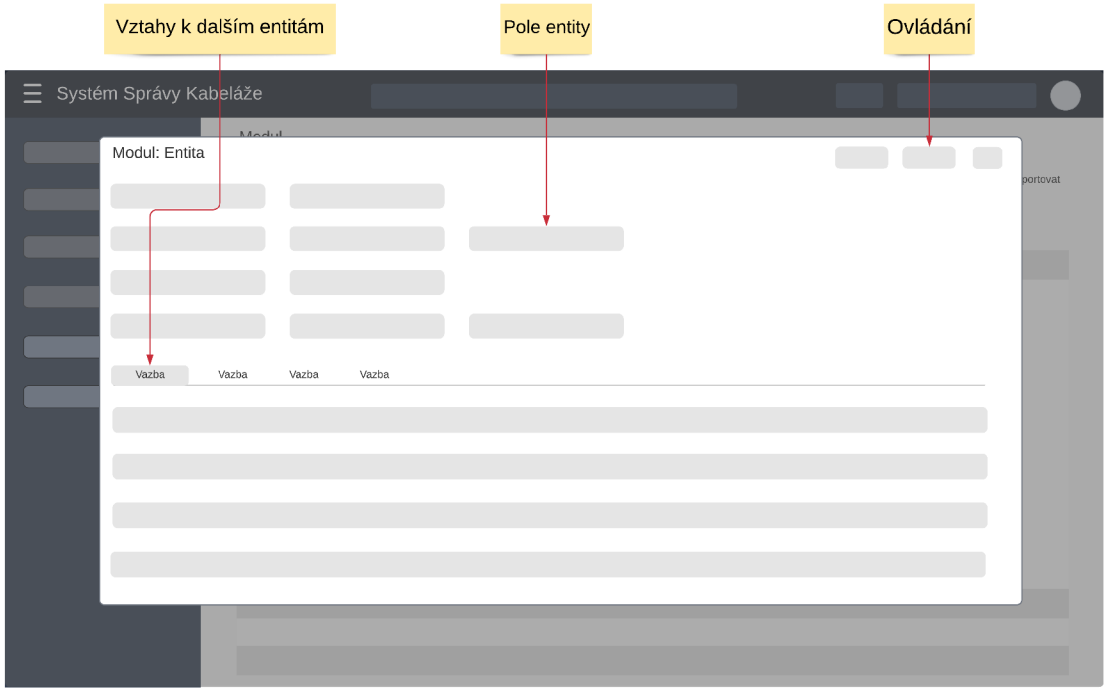
Každý modul bude umožňovat řazení, vyhledávání, filtrování a exportování nad záznamy. Stejně tak bude možné upravit aktuální zobrazení modulu přizpůsobením zobrazených sloupců a přeskládání pořadí jejich zobrazení. Tato zobrazení bude možné ukládat a označovat buď jako globální (sdílená mezi všemi uživateli), nebo uživatelská – privátní.



Obrázek 3.18: Prototyp filtrovacího dialogu

Jednotnost ovládání a funkcionality přispívá ke zlepšení uživatelské zkušenosti a zplošťuje učící křivku.

Filtrování bude podporovat uživatelsky jednoduché zadávaní vstupu včetně možnosti volby operátoru. Aplikované filtry budou maximálně vizuálně kompaktní a jejich editace bude možná pomocí dialogu. Mezi takto zadanými filtry bude implicitní logická spojka `AND`. Pokud filtrované pole umožňuje zadání vícero hodnot, bude platit implicitní logická spojka `OR`. Tyto pravidla implicitních logických spojek většině uživatelům dávají možnost v jednoduché formě vytvářet poměrně komplexní filtry. Pro uživatele a případy, kde tyto implicitní spojky jsou nevhodné, bude filtrování umožňovat speciálním UI i konstrukci libovolně hlubokého stromu filtrů. Uživatelské rozhraní bude jednak umožňovat konstrukci tohoto stromu, ale také i vypsání/vložení odpovídajícího JSON objektu filtru.

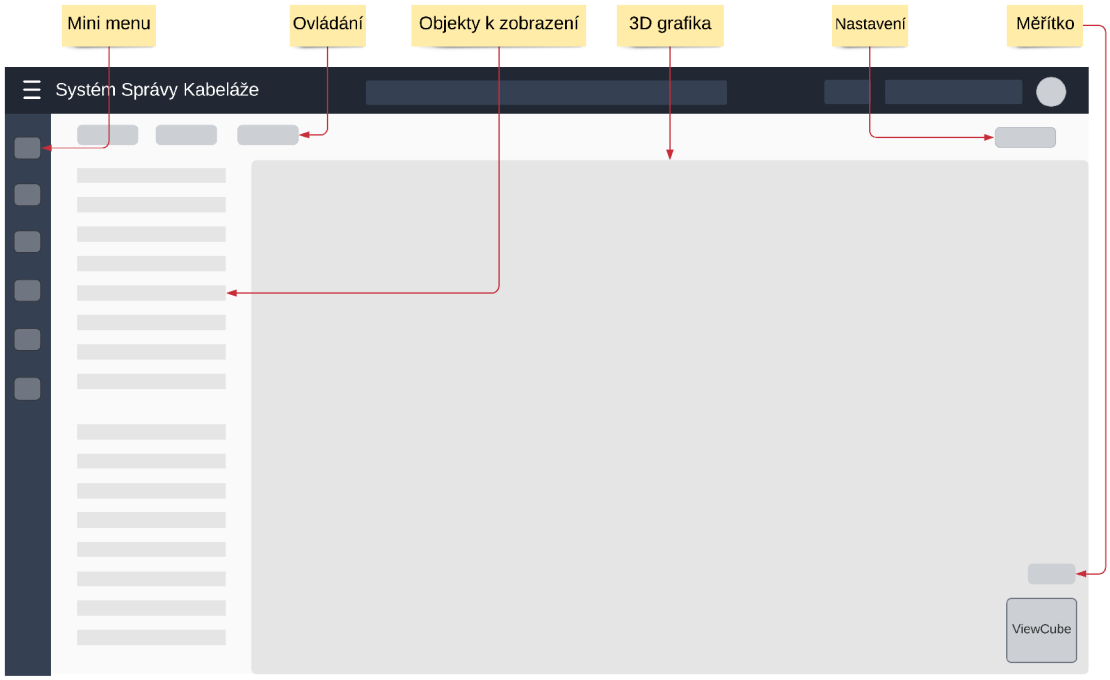


Obrázek 3.19: Wireframe obrazovky detailu záznamu

Detail záznamu bude zobrazován v dialogovém okně zakrývající výchozí obrazovku modulu. Dialogové modální okno taktéž umožňuje detaily záznamu jednotně prezentovat napříč systémem, například při otevření z jiného modulu. Dialogové okno bude obsahovat editační formulář pro daný záznam, ovládání a přehled vazeb ve formě podřazených tabulek. V rámci

ovládání bude možné záznam ukládat, rušit změny, přecházet dopředu či dozadu (pouze nad kontextem seznamu). Vazby budou prezentovat vztahy 1:N či M:N daného záznamu. Pod záložkou vztahů bude možné provádět standardní operace jako u každého modulu. V případě modulů, kde to je relevantní, bude zobrazena i záložka *Grafika*, která bude zobrazovat kompaktní verzi modulu grafiky, tedy interaktivní 3D grafiku.

Modul grafiky bude taktéž možné použít mimo kontext detailu záznamu. Modul bude možné otevřít z odkazu v postranním panelu. Tato verze modulu grafiky bude zobrazena na celou plochu obrazovky aplikace. V levé části modulu se bude nalézat panel, který bude umožňovat zaškrtávání objektů pro zobrazení. V horní části se bude nalézat ovládací lišta s možností přepínání zobrazení a konfiguraci nastavení zobrazovače. V pravém dolním rohu bude umístěna i tzv. *viewcube* – navigační kostka umožnující rotaci scénou. Bude se zde taktéž nalézat i měřítko. Modul grafiky bude podporovat interakci se zobrazenými modely formou *raycastu*, tedy vrhnutím paprsku/provedením propichu scénou. Tímto propichem bude možné označit a zachytit vícero vzájemně překrývajících se objektů. Při zachycení propichem budou business objekty zvýrazněny kontrastní barvou. Propíchnutí vyvolá kontextové menu, které zobrazení seznam zachycených objektů. Z tohoto seznamu bude možné objekty zacentrovat nebo zobrazit jejich detail.



Obrázek 3.20: Wireframe obrazovky modulu grafiky

Kapitola 4

Implementace řešení

Na základě návrhu popsaného v kapitole 3.8 bylo implementováno nové řešení SSK. Implementované řešení naplňuje kladené požadavky a z hlediska architektury představuje významnou část finální podoby produktu SSK. Z pohledu naplnění business scénářů a správy dat je odevzdávaná implementace fragmentem celého systému. Odevzdaná část reprezentuje vybrané části systému, které jsou významné z pohledu reengineeringu – tématu diplomové práce – a nenesou, z pohledu práce, podstatné know-how společnosti I&C Energo, které je předmětem duševního vlastnictví.

V následující kapitole budou popsány implementační detaily nového řešení a budou diskutovány použité technologie včetně důvodu jejich využití.

4.1 Celková koncepce

Řešení bylo implementováno jako sada služeb s cílovou platformou běhu v Red Hat OpenShift[1] a Docker (skrze nástroj Docker Compose[2]).

OpenShift je *hybridní* cloudové řešení založené na orchestrační platformě Kubernetes[3] umožňující *multi-tenantní* provoz na privátní či veřejné infrastruktuře. Více tenantů – zákazníků – je schopno tedy provozovat svá řešení na sdílené platformě. Platforma je provozována v rámci společnosti I&C Energo.

Docker Compose byl využit jako vývojová platforma a také bude sloužit jako produkční prostředí u zákazníků, kteří nemají vlastní cloudové řešení ani OpenShift.

Odevzdané řešení obsahuje předpisy pro spuštění v obou platformách, ovšem pro účely diplomové práce se předpokládá Docker Compose jako primární platforma, a to z důvodu jednoduchosti spuštění, nižších nároků na hardware a také minimálních prerekvizit. Produkční prostředí produktu SSK bude v OpenShift, kde budou zákazníkům dodány sestavené *images* a potřebné předpisy pro spuštění. Odevzdání obsahuje tyto předpisy, a navíc i dodatečné předpisy pro sestavení ze vzdáleného Gitového repozitáře. Tyto předpisy jsou interně používány pro nasazení do testovacího prostředí na korporátní infrastruktuře.

Z hlediska odlišnosti mezi dodanými přepisy je nutné zmínit, že předpisy pro OpenShift neobsahují službu pro hostování databáze, která v korporátním prostředí je typicky provozována na dedikovaném serveru, kde vlastnosti jako dostupnost, škálovatelnost a hlavně

[1] https://www.redhat.com/en/technologies/cloud-computing/openshift
[2] https://docs.docker.com/compose/
[3] https://kubernetes.io/

zálohování jsou zajišťovány přímo mechanizmy daného DBMS. Pro snadnost spuštění a demonstrace realizovaného řešení předpisy pro Docker Compose službu s databází obsahují.

4.2 Logování

Všechny implementované služby logují své výstupy do konzole, souborů a případně do služby Elastic. Každý záznam logu je opatřen časovou známkou, úrovní důležitosti, zprávou a dalšími specifickými atributy. Logování a archivace logů je důležitá z hlediska zpětného auditu aktivit a především diagnostiky problémů, což je v případě komplexní interakce služeb klíčové.

V případě služeb implementovaných v .NET je použit framework NLog[4]. Všechny tyto služby mají identickou konfiguraci logování. Soubory logů jsou děleny dle data, aby se předešlo vzniku velkých souborů, které by byly obtížné na zpracování. Služba Graphics, která je založena platformě Node.js, využívá framework Winston[5], který taktéž zapisuje do konzole a souborů dělených po dnech.

Služby Gateway, Core API, FileStorage, Graphics a Reporting podporují přesměrování svých logů do Elastic. Řešení Elastic umožňuje agregovat aplikační logy všech zmíněných služeb do jednoho místa, což podporuje auditovatelnost a zesnaďnuje monitoring celého řešení. Služby implementované v .NET s NLog používají knihovnu `NLog.Targets.ElasticSearch`[6]. V případě služby Graphics na platformě Node.js se jedná o knihovnu `winston-elasticsearch`[7]. Elastic není součástí odevzdaných služeb, předpokládá se existence externího nasazení. Přesměrování logů do Elastic je ve výchozí odevzdávané konfiguraci deaktivované, pro konfiguraci přesměrování je nutné upravit proměnné prostředí z výpisu 4.1 u zmíněných služeb.

```
1 ELASTIC_USERNAME=elastic
2 ELASTIC_PASSWORD=password
3 ELASTIC_URL='http://elasticsearch:9200'
4 ELASTIC_DISABLE=false
```

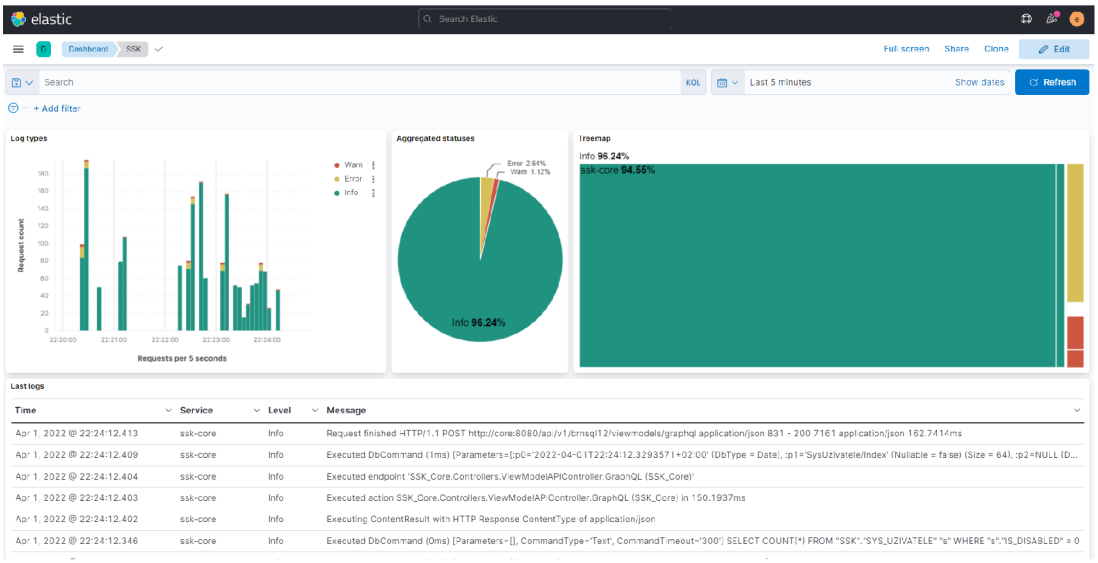
Výpis 4.1: Proměnné prostředí pro konfiguraci Elastic

Výstupní logy jsou zasílány přímo do Elasticsearch služby, transformace pomocí Logstash není vyžadována, jelikož standardizace zasílané zprávy probíhá na úrovni služeb. Autor nevylučuje, že v budoucnu nebude Logstash využit pro předzpracování či filtraci logů.

Podpora pro zasílání aplikačních logů do Elastic byla implementována zvláště kvůli případům provozování SSK v Docker kontejnerech v nejrůznějších prostředích, kde nemusí být zajištěn sběr logů do centrálních řešení. Zajištěním zasílání logů do centrálního řešení na úrovni služeb se usnadňuje nasazení a monitoring.

Red Hat OpenShift podporuje celo-clusterové logování do vlastního clusterového Elasticsearch. Tato funkcionalita je volitelná a je možné jí doinstalovat[8]. Pro nasazení v rámci OpenShift se očekává sběr logů právě tímto způsobem, kde logy jsou sbírány ze standardního výstupu kontejneru a následně zasílány do interního Elasticsearch. Výše popsaná konfigu-

[4] https://nlog-project.org/
[5] https://github.com/winstonjs/winston
[6] https://github.com/markmcdowell/NLog.Targets.ElasticSearch
[7] https://github.com/vanthome/winston-elasticsearch
[8] https://docs.openshift.com/container-platform/4.10/logging/cluster-logging-deploying.html



Obrázek 4.1: Vizualizace sesbíraných logů v prostředí Kibana pod Elastic

race sběru logů do Elastic je v případě OpenShift relevantní pouze tehdy, pokud OpenShift Logging není nainstalován, například z důvodu nedostatečných hardwarových prostředků.

4.3 Škálovatelnost

Reengineering do služeb, kromě již detailně popsaných výhod, taktéž přináší lepší možnosti ve škálovatelnosti vybraných části řešení. Nasazením v clusteru OpenShift je možné služby jednoduše a účinně škálovat dle potřeby.

V rámci této práce bylo experimentováno se škálováním vybraných služeb. Konkrétně se experimentování týkalo služeb Core API a Graphics, které při naplňování business scénářů podléhají největšímu zatížení. Jelikož běh v Docker Compose je v případě SSK primárně určen pro demonstrační a testovací provoz na pracovních stanicích, experimentace v tomto prostředí nebyla prováděna, ovšem lze konstatovat, že principiálně všechny výsledky dosažené v rámci OpenShift lze přenést i na případná nasazení pod Docker Compose.

V případě OpenShift/Kubernetes je škálování možné řešit pomocí objektu `Horizontal-PodAutoscaler`[9] nebo ruční konfigurace *replik*. Prvně jmenovaný přístup nabízí prostředek automatického škálování běžících podů podle aktuálního vytížení. Druhý přístup pak pevně definuje počet souběžně běžících podů jedné služby. Pod je v tomto případě instance služby, tedy kontejner. Výběr podu probíhá na základě algoritmu round-robin.

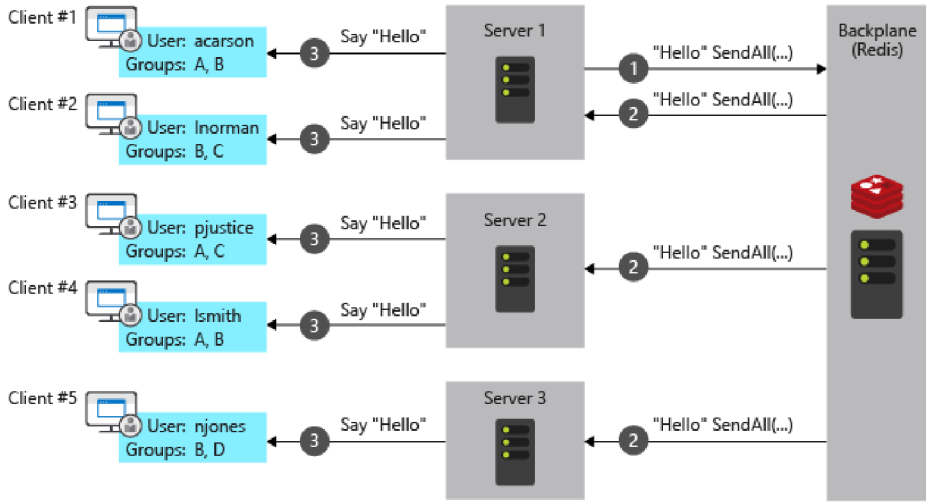
Služby, které jsou bezestavové je možné typicky libovolně škálovat. U služeb, které udržují kontext spojení (*session*), je nutné zajistit, že klienti komunikující s instancemi služby budou vždy komunikovat s jedním z daných podů po celou dobu existence kontextu spojení. V případě SSK je bezestavová služba Graphics a stavová Core API. U Core API je stavovost dána použitím SignalR spojení, které je inicióváno *handshakem*, po kterém si klient a server udržují perzistentní spojení. Přepínáním mezi pody bude toto spojení vždy ukončeno.

Docílit pevné asociace mezi klientem a serverem chování lze nastavením tzv. *afinity*, která v tomto konkrétním případě byla na úrovni Kubernetes `Service` objektu nastavena

[9] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

následovně: `sessionAffinity: ClientIP`. Afinita je tedy nastavena podle IP adresy klienta, tudíž všechny spojení z dané IP adresy budou směřovány na jeden pod.

Samotné nastavení afinity však neřeší problém sdílené SignalR zpráv napříč pody. Z tohoto důvodu je jako páteřní spojnice (*backplane*) použita Redis cache. Redis[10] je *in-memory* databáze pro cachování dat, kterou právě SignalR dokáže používat jako svůj backplane pro ukládání všech připojených klientů a eviduje seznam podů, kam jsou daní klienti připojeni [35]. Je tedy možné zasílat zprávy identickým způsobem jako v případě jediného podu.



Obrázek 4.2: Ilustrace zasílání zpráv přes SignalR s Redis backplane, převzato z [35]

Redis backplane se transparentně integruje se SignalR přidáním tzv. *extension*. Příklad nastavení SignalR s Redis backplane je znázorněn ve výpisu 4.2. Redis je v rámci SSK integrován jako samostatná služba. Služba je pro úplnost zahrnuta i v rámci konfigurace pro Docker Compose.

```
services.AddSignalR().AddStackExchangeRedis("{connection_string}");
```

Výpis 4.2: Konfigurace SignalR s Redis backplane

Při experimentování byl nastaven počet replik u služby Graphics na dvě a u Core API na tři. Tabulka 4.1 porovnává naměřené časy odpovědí podů služby Core API v závislosti na počtu podů a zátěži. Škálováním je dosaženo měřitelného zlepšeních průměrných časů a především významného snížení směrodatné odchylky, které právě lze přisuzovat distribuci provozu mezi pody. Celkově škálování v případě SSK zvyšuje predikovatelnost rychlosti odpovědi a do budoucna umožní produkční provoz, kde větší množství uživatelů bude přistupovat k jednomu nasazení SSK připojenému na vícero databází.

	1 pod bez zátěže	**1 pod se zátěží**	**3 pody se zátěží**
Prům. doba odpovědi [s]	1,37	2,61	1,95
Směrodatná odchylka [s]	0,24	1,29	0,57

Tabulka 4.1: Rychlost odpovědi podů služby Core API dle zátěže

[10] https://redis.io/

4.4 Autorství

Součástí odevzdaného řešení je obraz databáze SSK, která obsahuje schéma, triggery a Oracle packages. Tyto objekty nejsou předmětem výstupu autora této práce a jsou použity se souhlasem společnosti I&C Energo. Řešení taktéž obsahuje části, které byly předmětem dřívějšího výstupu zaměstnanecké činnosti autora práce, jmenovitě mikro-framework WebController. Pokud není uvedeno jinak (ať už v textu či zdrojových kódech), odevzdané výstupy jsou dílem autora této práce.

4.5 Realizované služby

V následujících podkapitolách bude popsaná implementace či konfigurace služeb, které jsou jsou realizované novém řešení SSK. Budou popsány jednotlivé problémy a výzvy, které bylo nutné řešit v rámci implementace.

4.5.1 Služba autentizace

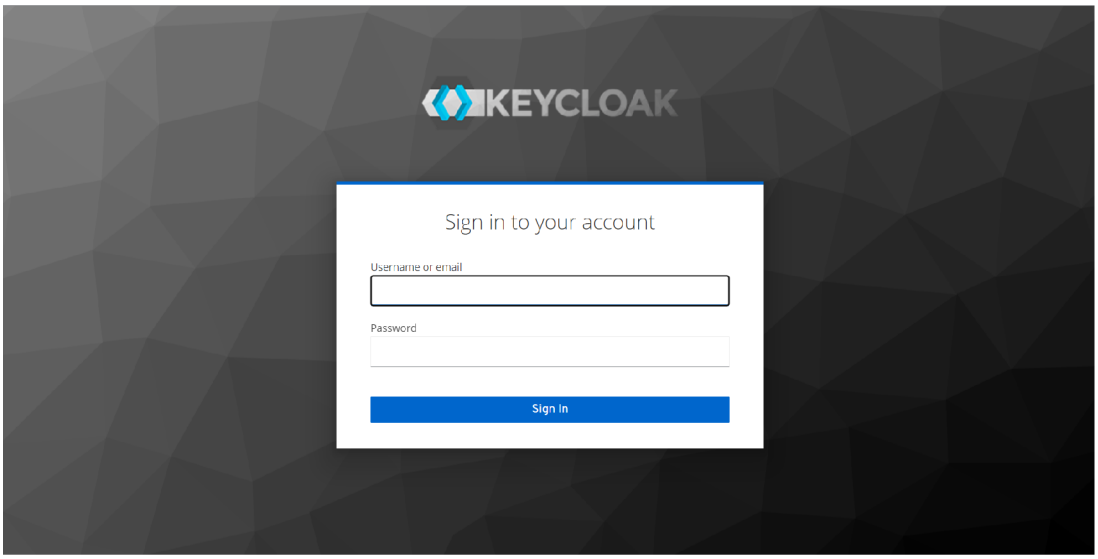
Tato služba zprostředkovává OpenID Connect autentizaci uživatelům či službám při vzájemné komunikaci. Služba je realizovaná řešením Keycloak[11]. Toto řešení poskytuje správu identit a řízení přístupu skrze protokoly OpenID Connect (OIDC) či SAML 2.0. Keycloak může fungovat v několika základních režimech: lokální uživatelé, federace uživatelů a *identity brokering*. V případě lokálních uživatelů se uživatelé mohou sami registrovat do databáze Keycloak, toto řešení je vhodné pro izolované aplikace/systémy. V kombinaci s identity brokeringem, který umožňuje autentizaci pomocní externích poskytovatelů (s využitím OIDC/SAML) identity (Google, Apple apod.), lze dosáhnout řešení vhodného pro běžné koncové uživatele, kde daný uživatel může volit mezi klasickým nebo externě autentizovaným účtem.

Pro korporátní prostředí je vhodné využít federaci uživatelů z LDAP/Active Directory serverů. Keycloak federaci realizuje periodickou synchronizací LDAP stromu uživatelů a případným mapováním atributů z LDAP do Keycloak. Uživatelé a jejich namapované atributy jsou následně uloženy přímo v databázi Keycloak. Ověření správnosti hesla vždy probíhá přímo vůči LDAP [28]. V testovacím prostředí I&C Energo Keycloak využívá synchronizace s Active Directory v režimu *read-only*, tedy ze strany Keycloak není možné provádět změny atributů na AD.

Pro Keycloak byla v rámci řešení dedikována vlastní databáze PostgreSQL. Keycloak navíc podporuje i MariaDB, MSSQL, MySQL a Oracle databáze. PostgreSQL byla vybrána pro menší zátěž systémových prostředků než ostatní databáze. Integrace pod hlavní databázi SSK, která využívá Oracle technologii, byla zvažována, ale takové řešení přináší několik úskalí:

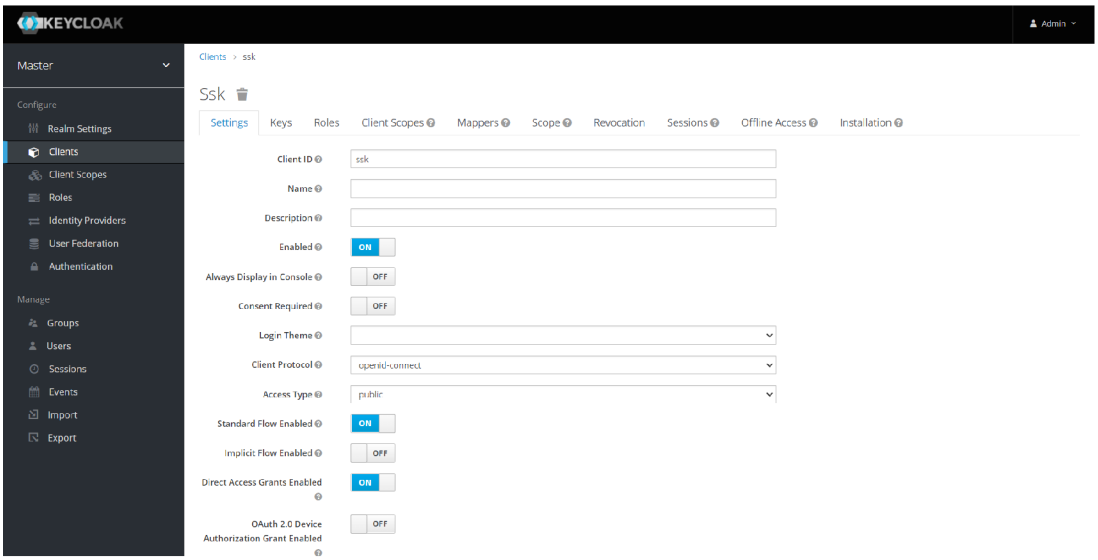
- **nemožnost využít autentizační službu pro jiné systémy** – Keycloak je strukturován do *realmů* (oddělených prostředí s vlastní konfigurací), tudíž ze své podstaty počítá s využitím napříč vícero systémy jako centrální autentizační a autorizační služba
- **zátěž jedné databáze** – přímo v rozporu s konceptem database-per-service, snížení výkonu systému

[11] https://www.keycloak.org/



Obrázek 4.3: Přihlašovací okno Keycloak

V odevzdaném řešení se Keycloak při prvním spuštěním inicializuje importem JSON konfigurace. V Keycloak se vytvoří realm SSK se dvěma SSK specifickými klienty: `ssk` a `ssk-services`. Klienti jsou aplikace, které mohou jménem uživatele žádat o přihlášení. První z jmenovaných je nakonfigurován pro přihlášení skrze koncovou klientskou aplikaci, tedy webový prohlížeč. Z pohledu OIDC se jedná o *Authorization Code Flow*. Klient `ssk-services` slouží pro komunikaci služba–služba (např. core–graphics–core). Zde je využito *Client Credentials Flow*.



Obrázek 4.4: Administrace klienta SSK

4.5.2 Služba Gateway

Služba Gateway plní roli směrovače, ověřovatele přístupového tokenu a také hostitele sestavených statických souborů uživatelského rozhraní. Služba je implementována na platformě .NET ve verzi 6. K naplnění funkcionality směrovače se využívá knihovny Ocelot[12], která na základě předdefinovaných pravidel mapuje a transformuje přijmuté externí požadavky, které dále rozesílá na interní služby. Ocelot pro svou konfiguraci využívá JSON soubor konvencí pojmenovaný `ocelot.json`.

Ukázka možné konfigurace jednoho pravidla je nastíněna ve výpisu 4.3. V ukázce je demonstrováno pravidlo překlad komunikace technologií WebSocket mapováním *upstream* cesty (URL cesta dostupná zvenčí) na *downstream* (cesta na konkrétní skrytou službu). Dále je ve výpisu konfigurován poskytovatel autentizace pro danou cestu. Poskytovatel `Bearer` je definován v rámci `Startup` třídy jako `JwtBearerDefaults.AuthenticationScheme`. Ocelot taktéž umožňuje překládat JWT `claims` (informace o autentizovaném subjektu) na hlavičky pro downstream služby.

```
{
  "DownstreamPathTemplate": "/ws/v1/{everything}",
  "DownstreamScheme": "ws",
  "DownstreamHostAndPorts": [
    {
      "Host": "core",
      "Port": 8080
    }
  ],
  "UpstreamPathTemplate": "/ws/core/v1/{everything}",
  "UpstreamHttpMethod": [ "GET", "POST", "PUT", "DELETE", "OPTIONS" ],
  "AuthenticationOptions": {
    "AuthenticationProviderKey": "Bearer",
    "AllowedScopes": []
  },
  "AddHeadersToRequest": {
    "users": "Claims[preferred_username] > value[0] > |",
    "roles": "Claims[resource_access][roles] > |"
  }
}
```

Výpis 4.3: Příklad konfigurace `ocelot.json`

Autentizace na úrovni Gateway je zajištěna ověřením tokenu vůči veřejným klíčům autentizační služby. Tokeny jsou akceptované z HTTP `Authorization` hlavičky a také `access_token` URL parametru. Zasílání tokenů v URL je méně bezpečné, jelikož URL je běžně předmětem logování, tudíž hrozí prozrazení tokenu, ovšem některé nástroje nepodporují nastavování HTTP hlaviček, tudíž je to jediná možnost, jak s takovými klienty v opodstatněných případech komunikovat.

Služba taktéž jednotně zprostředkovává přístup k OpenAPI (Swagger) dokumentaci spojením nakonfigurovaných endpointů pod jediné uživatelské rozhraní.[13]

[12] https://github.com/ThreeMammals/Ocelot

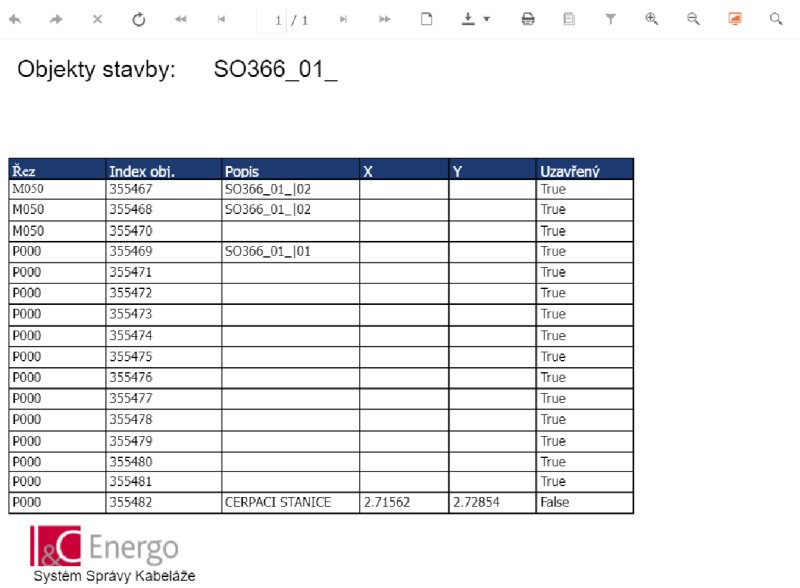
[13] Rozhraní je dostupné z adresy http://localhost:8081/swagger.

4.5.3 Služba Reporting

Tato služba poskytuje v rámci SSK funkcionalitu generování a návrhu reportovacích sestav. Pro její implementaci bylo využito produktu Telerik Reporting[14]. Tento produkt byl společností I&C Energo zakoupen pro účely tohoto a jiných projektů. Pro demonstraci sestavení služby v rámci diplomové práce je nutné aktivovat alespoň *trial* verzi vývojářské licence tohoto produktu. Dodání sestavených výstupů, které jsou taktéž součástí odevzdání, nepodléhá platbě za vývojářskou licenci.

Implementace je realizována na platformě .NET 6. Telerik Reporting je integrován vytvořením dvou controllerů dědících od `ReportsControllerBase` a `ReportDesignerControllerBase` tříd z knihoven Teleriku. První jmenovaná poskytuje potřebné endpointy pro prohlížeč reportů a druhá pro webový návrhář. Integrace na straně uživatelského rozhraní (součástí klientské aplikace) je realizována pomocí JavaScriptových knihoven využívajících knihovnu jQuery[15].

Ukládání reportů a konfigurace služby je realizována na souborový systém kontejneru, odkud jsou perzistovány mimo běžící kontejner přes *volumes* (Docker Compose) nebo *Persistent Volume Claim* (PVC, OpenShift). Skrze návrhář je možné do tohoto úložiště přidávat a nahrávat nové šablony reportů.

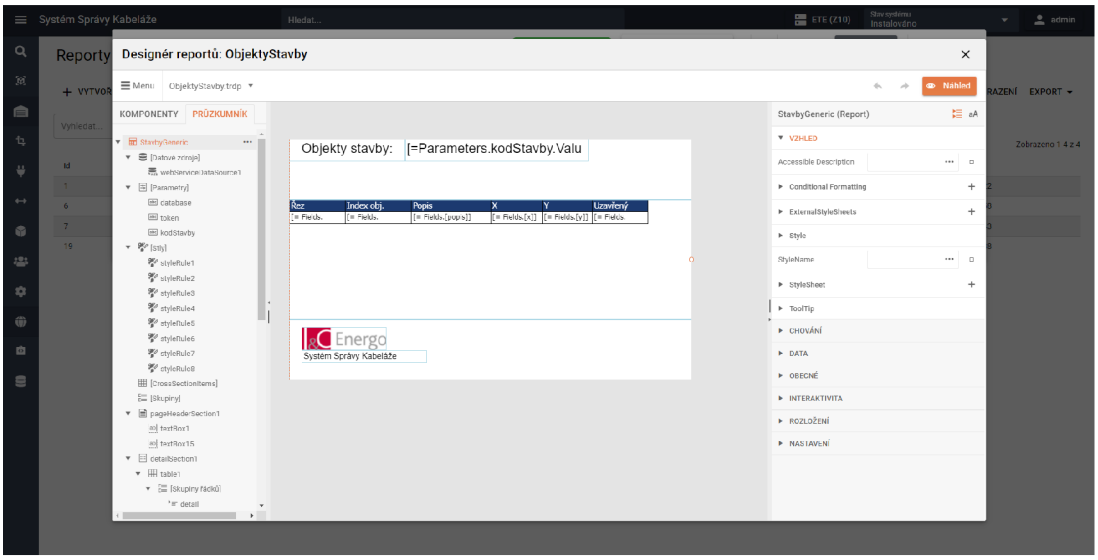


Obrázek 4.5: Ukázkový výstup reportu

Přístup k funkcionalitě reportování je podmíněn držením uživatelského práva nad objektem `SYS_SESTAVY_REPORTY` s akcemi `SELECT` pro zobrazení reportů a `UPDATE` pro vytváření a úpravu šablon reportů. Tato autorizace probíhá vůči službě Core API, která drží mapování mezi uživateli, rolemi a oprávněními na jednotlivé typy objektů. Podrobné schéma komunikace bylo uvedeno v kapitole návrhu služby reporting 3.8.2. Tato práva zajišťují pouze obecný přístup k funkcionalitě reportingu. Zobrazení konkrétních reportů je pak závislé právech daného uživatele, protože pro získání dat reportu se využívá přímo tokenu aktuálního uživatele, který je společně s názvem cílové databáze standardizovaným skrytým

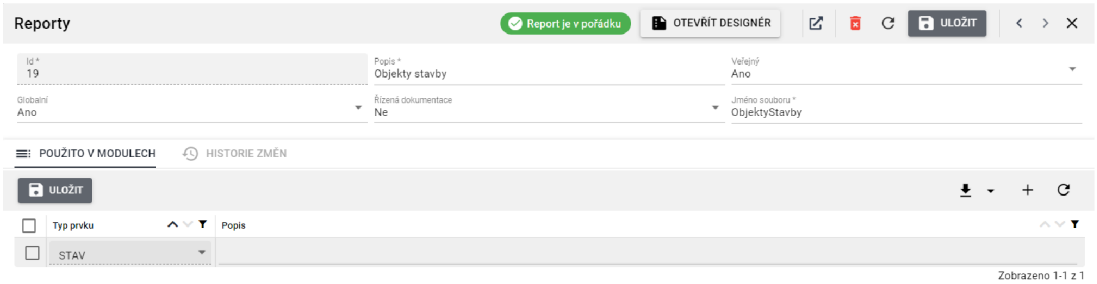
[14] https://www.telerik.com/products/reporting.aspx
[15] https://jquery.com/

parametrem reportu. Standardizované parametry jsou automaticky zasílány z uživatelského rozhraní při generování jakéhokoliv reportu. V rámci definice modulu na úrovni služby Core API lze zadefinovat další parametry, které se budou (například na základě hodnot aktuálního záznamu) předvyplňovat.



Obrázek 4.6: Integrace návrháře reportů do SSK

Samotné soubory šablon reportů jsou potom se samotnou aplikací integrovány na základě jmenné konvence se záznamy v tabulce `SYS_SESTAVY_REPORTY`, konkrétně polem `JmenoSouboru`, které musí odpovídat názvu šablony souboru ze služby Reportingu. Uživatelské rozhraní poskytuje prostředek kontroly existence reportu přímo z detailu záznamu a také přímý přechod do editoru šablony. Jednotlivé reporty jsou pak v aplikaci přiřazovány konkrétním modulům (např. stavby, místnosti apod.). Přiřazené reporty je možné z detailu záznamů daných modulů zobrazovat, případně tisknout či stahovat ve formátech PDF, CSV, XLSX, PPTX, RTF a DOCX.



Obrázek 4.7: Kontrola stavu reportu a kontextová akce otevření šablony reportu

4.5.4 Služba Core API

V rámci služby Core API je implementována klíčová funkcionalita řešení SSK, tedy správa hlavního datového skladu realizace business scénářů nad ním. Služba je implementována ve frameworku .NET 6.

Přístup do databáze je realizován prostřednictvím knihovny Entity Framework[16] (EF) s adaptérem pro Oracle databázi. Entity Framework je objektově-relační mapper (ORM), který umožňuje mapovat databázové objekty (tabulky, pohledy) do tříd a provádět dotazování skrze technologii *LINQ to SQL*, která za běhu překládá lambda výrazy (či SQL-like C# syntaxi) a generuje SQL dotazy v příslušném SQL dialektu.

Kromě EF se v rámci služby Core API významně využívá mikro-frameworku *WebController*, který byl pro účely rychlého vývoje webových aplikací vyvinut v minulosti autorem této práce. WebController byl a je vyvíjen ve společnosti I&C Energo. Autor práce stojí za hlavní myšlenkou projektu a je autorem takřka celé současné implementace. WebController poskytuje mechanizmy automatického mapování EF modelů do/z *viewmodelů*, provádění CRUD operací, exportů, filtrování, řazení, správu souborů a definování vlastních operací. Součástí implementace je taktéž i nativní podpora pro optimistické zamykání záznamu.

WebController je členěn do dvou separátních projektů: `WebController.Standard` a `WebController.MVCCore`. První z jmenovaných poskytuje obecnou funkcionalitu bez závislosti na konkrétním webovém či jiném frameworku. V současné době projekt cílí na .NET 6, v minulosti byl cílovou platformnou .NET Standard, tudíž bylo možné používat WebController jak v aplikacích postavených na .NET 4.x, tak i v moderním .NET Core, z kterého vychází dnešní .NET 5/6. Projekt `WebController.MVCCore` obsahuje konkrétní implementace vztahující se k webovému frameworku ASP.NET Core MVC. `WebController.Standard` abstrahuje tyto implementace pomocí rozhraní. Oddělení i v budoucnu může být výhodné v případě implementace pro technologie Blazor či Xamarin a jejich budoucí iterace. WebController poskytuje REST a GraphQL[17] rozhraní. REST rozhraní umožňuje nad každým nadefinovaným viewmodelem provádět následující operace:

- `list` – kolekce záznamů vyhovující zadaným filtrů, seřazená a stránkovaná podle zaslaného požadavku
- `detail` – jeden záznam identifikovaný jedním či vícero klíči (může či nemusí být primární klíč, postačuje unikátnost ze zadaných hodnot, tudíž všechny kandidátní klíče mohou být použity)
- `create` – vytvoření jednoho záznamu
- `edit` – editace jednoho záznamu, provádí parciální aktualizaci záznamu ze zaslaného slovníku obsahující klíč/hodnotu na daném záznamu
- `editBatch` – editace či vytvoření vícero záznamů, záznam pro vytvoření je odlišen přítomností klíče `__create` ve slovníku pro daný záznam
- `delete` – smazání jednoho záznamu
- `deleteBatch` – smazání vícero záznamů
- `toggleDisabled` – (de)aktivace záznamu, nativní podpora pro soft-mazání záznamů
- `autocomplete` – našeptávač číselníku definovaného např. na základě cizího klíče
- `export` – export vyfiltrovaných nebo explicitně vybraných záznamů
- `fileUpload` – nahrání souboru přiřazeného k entitě

[16] https://docs.microsoft.com/en-us/ef/
[17] https://graphql.org/

- `fileDownload` – stáhnutí souboru přiřazeného k entitě
- `fileDelete` – smazání souboru přiřazeného k entitě
- `metadata` – serializovaný výstup konfigurace viewmodelu
- `frontendAction` – speciální pojmenována business akce bez vedlejšího efektu do datového skladu
- `backendAction` – speciální pojmenována business akce s vedlejším efektem do datového skladu

WebController umožňuje volání předem nadefinovaných pojmenovaných business akcí (metod), které mohou provádět vývojářsky definovanou aplikační logiku (tedy nad rámec standardu WebControlleru). Business akce označována jako `frontendAction` akceptuje aktuální hodnoty formuláře, provádí transformaci těchto hodnot a vrací upravené hodnoty. Tento typ akce je vhodný pro případy, kde je třeba provést akci s ohledem na databázi, ale zároveň finální uložení ponechat na uživateli. Akce typu `backendAction` akceptuje aktuální hodnoty formuláře a její transformace nad těmito hodnotami jsou okamžitě ukládány do databáze. Vlastní business akce byly standardizovány na úrovni WebControlleru, aby poskytly možnosti snadného dodefinování aplikační logiky na základě pojmenovaných uživatelsky volaných akcí.

Středobodem implementace WebControlleru do cílového projektu je zmíněný viewmodel, který je své podstatě kopií EF modelu, kde jednotlivé *properties* jsou odekorované specifickými atributy, které definují jejich chování a viditelnost. Kromě standardních typu properties (`string`, `int` apod.) je WebController schopen pomocí speciálních typů zpřístupnit vztahy přes cizí klíče, včetně kompozitních. Mezi další patří typ pro booleanovské hodnoty (remapování pravda/nepravda na vlastní texty), typ pro transpozici řádků na sloupce či typ pro automatické zobrazení vztahu 1:N.

Z hlediska filtrování je implementována plná podpora stromových filtrů s logickými a hodnotovými operátory, je tedy možné konstruovat libovolně zanořené filtry s logickými spojkami `AND` a `OR` a každá listová hodnota stromu, která filtruje specifikované pole, může dle kontextu obsahovat operátory, jakými jsou například `EQUAL`, `SUBSTRING` apod. Stromová struktura filtru je rekurzivně vyhodnocována, přičemž je využíváno knihovny Dynamic LINQ[18], která umožňuje *runtime* překlad z textových řetězců do LINQ výrazů. Rozhraní GraphQL podporuje pouze čtení dat, editace skrze GraphQL *mutations* byla zvažována, ale vzhledem k malé přidané hodnotě oproti REST rozhraní byla tato funkcionalita zavrhnuta. Autor práce spatřuje hlavní výhodu GraphQL v selektivitě polí a navigací napříč grafem, kde WebController obě tyto vlastnosti podporuje. V případě výběru pouze některých polí skrze GraphQL je optimalizována konstrukce speciálních typů objektů na úrovni viewmodelu, kde (pokud není explicitně vynuceno) se tyto objekty nenaplňují z databáze/repository.

```
{
  {
  stavbyRezy(
    startIndex: 0
    amount: 25
    sort: [{ field: kodRezuStavby, dir: DESC }]
    filter: { childrenOperator: AND, children: [
```

[18] https://dynamic-linq.net/

```
      { leafProperty: kodStavby, leaf: { value: "SO350_01_", operator: EQUAL}}
      ]}
  ){
    totalCount
    sort {
      field
      dir
    }
    items{
      kodStavby
      kodRezuStavby
      popis
      stavbyMistnostis(startIndex: 0, amount: -1){
       kodMistnostiStavby
     }
    }
  }
}
```

Výpis 4.4: GraphQL dotaz na objekt vůči Core API

Výpis 4.4 dává příklad GraphQL dotazu vůči Core API, které pro jeho vyhodnocení využívá WebController. Dotaz je prováděn nad entitami typu `stavbyRezy`, ze kterých je žádán 1. až 25. záznam vyhovující filtru, kde pole `kodStavby` je rovno hodnotě `SO350_01_`, záznamy jsou sestupně řazeny dle pole `kodRezuStavby`. Z dotazu je na výstupu žádán celkový počet záznamů vyhovující filtru, aplikované řazení (řazení mohlo být automaticky aplikováno v případě nepoužití žádného vstupního řazení) a samotná kolekce záznamů, ze které budou načteny a zobrazeny pouze specifikovaná pole. Kromě základních polí lze také získávat záznamy s přivázaných záznamů (např. vztahem 1:N skrze cizí klíč), konkrétně v příkladu je tato funkcionalita demonstrována skrze `stavbyMistnostis`, což je kolekce asociovaných místností u daného řezu, ze které jsou žádány pole `kodMistnostiStavby` všech záznamů. Kromě dotazu na jednotlivé viewmodely, je podporován i dotaz metadata viewmodelu, která reprezentují dynamicky vyčtenou konfiguraci z daného viewmodelu. Zkrácený příklad takového dotazu je uveden ve výpisu 4.5.

```
{
    metadata(viewModel:"stavbyRezyObjekty"){
        settings{
            optionFlags
            featureFlags
            uIName
        }
        businessActions
        props{
            displayName
            primaryKey
            propertyName
            propertyType
        }
    }
}
```

Výpis 4.5: GraphQL dotaz na metadata vůči Core API

Integrace na databázi/perzistentní úložiště je řešena skrze návrhový vzor *repository*, kde `WebController.Standard` interaguje pouze s rozhraním `IModelRepository`, jehož výchozí implementace vůči Entity Framework je umístěna v projektu `WebController.MVCCore`. Na

úrovni Core API projektu se pak pouze dědí od této implementace a je dodávána či měněna specifická funkcionalita. Oproti standardní implementaci `ModelRepository`, vlastní SSK implementace (`SSKModelRepository`) mění způsob ukládání souborů (metody `GetBlob`, `SetBlob` a `DeleteBlob`) z původního přímého ukládání do databáze na volání služby FileStorage. Volání služby FileStorage je prováděno pod autentizací přes již zmíněný Client Credentials Flow, tedy obdobou servisního účtu. Použití Client Credentials Flow řeší problém, kdy by autentizační token mohl v průběhu komunikace služba–služba expirovat, což by vyústilo v selhání volání. Stejně tak tento flow umožňuje volat jiné služby bez kontextu uživatele, což v může být v budoucnu vhodné pro nejrůznější plánované dávkové úlohy. Redefinice (`override`) metod `ModelRepository` tedy transparentně mění chování původní implementace a demonstruje flexibilitu řešení WebController.

Kromě zmíněného `IModelRepository` se WebController s cílovým projektem integruje skrze následující rozhraní:

- `IContext` – platformě nezávislé úložiště pro *scoped* data (referenčně implementováno přes `HttpContext.Items`), parsování dat souboru a získávání virtuální URL cesty
- `IViewModelLocator` – rozhraní pro získání typu viewmodelu na základě jeho jména a naopak (využívá reflekci pro získání konkrétní třídy v rámci *assembly*)
- `IUserServices` – rozhraní pro vyhodnocení práv uživatele v daném kontextu (typ akce a viewmodel)
- `IWebControllerLogger` – obecné rozhraní pro logování z WebControlleru, typicky obsahuje volání projektově specifického logovacího frameworku

Core API se společně se službou Graphics podílí na realizaci obnovy cache 3D objektů. Kromě automatické kontroly aktuálnosti 3D objektu, které nemusí komplexitou datového modelu SSK (případně přímých přístupů mimo do databáze v okrajových případech) postihnout všechny boční efekty, které mohou změnit atributy a entity vstupující do 3D objektu, je umožněno manuálně vyvolat obnovu cache skrze Core API. V takovém případě Core API volá endpoint na Graphics s autentizací pomocí Client Credentials Flow, který spustí obnovu jednoho či více 3D objektů. Služba Graphics následně s přijatým tokenem volá nazpět Core API a získává aktuální data pro sestavení potřebných 3D objektů.

```
[VMSettings(UIName = "Stavby", UINameResourceType = typeof(SSKAttributes), FeatureFlags =
    Features.StandardDeleting)]
public class StavbyRezy_VM : CoreModuleBaseViewModel<StavbyRezy>
{
    public StavbyRezy_VM(ViewModelParameters parameters) : base(parameters)
    {
        AfterLoad = () =>
        {
            ...
        };
    }

    [VMPrimaryKey(Order = 1), Required]
    [Display(Order = 1, ResourceType = typeof(SSKAttributes))]
    [VMDatabaseColumn, VMVisible(Visibility = AttrVisibility._All, IsEditable = false)]
    public VMDatabaseDropdown KodStavby { get; set; } =
                                    new VMDatabaseDropdown(typeof(Stavby), typeof(String),
                                    nameof(Stavby.KodStavby), nameof(Stavby.KodStavby));

```

```
18      [VMPrimaryKey(Order = 2), Required]
19      [Display(Order = 2, ResourceType = typeof(SSKAttributes))]
20      [VMDatabaseColumn, VMVisible(Visibility = AttrVisibility._All, IsEditable = false)]
21      public string KodRezuStavby { get; set; }
22
23      ...
24  }
```

Výpis 4.6: Ukázka definice viewmodelu

Výpis 4.6 obsahuje ukázku konfigurace viewmodelu, v tomto případě se jedná o značně zjednodušený příklad konfigurace pro modul stavebních řezů. Třída `StavbyRezy_VM` dědí od generické třídy `CoreModuleBaseViewModel`, která obsahuje společné chování pro hlavní moduly SSK. Tato třída sama dědí od `GenericViewModel`, což je bázová třída WebControlleru. U každého implementovaného viewmodelu je vyžadováno, aby byl dekorován atributem `VMSettings`, který definuje základní nastavení viewmodelu. Zde je možné konfigurovat jeho název v uživatelském rozhraní (včetně zdrojové *resource* pro lokalizaci), podporované vlastnosti (vytváření, editace, mazání, export, soft-mazání apod.), výchozí nastavení pro kopírování properties, multithreading zpracování viewmodelů apod. Konstruktor viewmodelu musí akceptovat jeden argument typu `ViewModelParameters`, který obsahuje specifické argumenty, pod kterými je podle požadavku konstruována aktuální instance. V rámci konstruktoru lze také přidat *hooks* pro napojení do životního cyklu viewmodelu za účelem přizpůsobení chování, nastavování vlastní properties apod. Mezi podporované hooks se řadí:

- `BeforeLoad/AfterLoad` – invokován před/po načtení hodnot z instance `IModelRepository` modelu (EF model)
- `BeforeParseDataFromForm/AfterParseDataFromForm` – invokován před/po parsování zaslaného slovníku při editaci či vytváření
- `BeforeCreateEntity/AfterCreateEntity` – invokován před/po instanciaci modelu pro `IModelRepository`
- `BeforeUnload/AfterUnload` – invokován před/po překopírování hodnot z viewmodelu do asocivoaného modelu
- `BeforeCreate/AfterCreate` – invokován před/po přidání modelu do `IModelRepository`
- `BeforeUnloadDependant/AfterUnloadDependant` – invokován po uložení hlavního záznamu a před/po ukládáním závislých záznamů (záznamy, které vyžadují existenci hlavního v DB)
- `BeforeFinalSave/AfterFinalSave` – invokován před/po konečném uložení

V případě editace je životní cyklus následující: `Load`, `ParseDataFromForm`, validace, `Unload`, `UnloadDependant`, `FinalSave`. Pro vytváření je posloupnost: `ParseDataFromForm`, validace, `CreateEntity`, `Unload`, `Create`, `UnloadDependant`, `FinalSave`

Mimo tyto hooks specifické pro viewmodel lze na úrovni `ViewModelCollection`, což je nadřazený objekt zapouzdřující všechny aktuálně načítané viewmodely (bez ohledu na to, zda se jedná o operaci `list` nebo `detail` aj.), definovat i hooks pro zpracování celé této kolekce. Jedná se o následující:

- `Collection.Prefetch` – invokován pouze jednou v rámci právě načítané kolekce viewmodelu, vhodný pro donačítání vlastních dat z databáze
- `Collection.Filtering.CustomFiltering` – invokace před vyhodnocením zadaných filtrů, umožňuje předfiltrování záznamů

Instanciované a naplněné viewmodely se následně podle výstupního rozhraní (REST, GraphQL či export) transformují na výsledné objekty. V případě REST se viewmodel transformuje do slovníku, kam je přidán i unikátní hash vypočtený z hodnot aktuálního záznamu, který slouží pro kontrolu při zpětném ukládání z důvodu optimistického zamykání. V rámci GraphQL je každá hodnota reprezentována objektem sestávajícího se z polí `value` a `text`, kde `text` je řetězcovou reprezentací hodnoty podle aktuálního nastavení lokalizace. Pro export se data viewmodelů extrahují do `ExportDataTable`, která je následně dle zvoleného typu exportu (XLSX, CSV) exportována do souboru skrze rozhraní `IExporter`, jehož implementace se liší právě v závislosti na zvoleném typu exportu. Konkrétní implementace je získaná podle návrhového vzoru *factory*, konkrétně zde `IExporterFactory`.

Součástí Core API je taktéž i autorizace pro celé řešení SSK. Umístění autorizace do této služby je dáno úzkým propojením na objekty datového modelu. Integrací (nebo spíše neoddělením) je možné do budoucna ponechat možnost snadno implementovat objektová práva[19]. Jelikož je Core API je dominantním uživatelem funkcionality vyhodnocení práv uživatele a zajištění trvalé konzistence je důležité pro existující datový model, oddělení autorizace do zvláštní služby by v konečném důsledku nepřinášelo znatelné benefity. V implementovaném řešení služby, které jsou dostupné z vnějšku (Reporting a Graphics), provádějí autorizaci právě vůči Core API. Četnost požadavků na tyto služby bude řádově nižší než v případě služby Core API, která obstarává téměř celou funkcionalitu alfa-numerické části aplikace.

Služba Core API taktéž podporuje práci s více databázemi, a to prostřednictvím specifikace názvu databáze v URL cestě. Povolené hodnoty jsou definovány v rámci konfigurace aplikace v souboru `appsettings.json`. Ke každé databázi (kromě jejího klíče) je nutné definovat její název v uživatelském rozhraní, obrázek pro dlaždici databáze a řetězec pro připojení, kde je podporována interpolace argumentů (např. `{}`), kde první argument je uživatelské jméno a druhý heslo. Tyto údaje lze zadat jako další pole pod danou sekci konfigurace. Taktéž je podporováno načítání proměnných prostředí ve formátu `SSK_{database}_{Username|Password}`, tedy např. `SSK_local_Username`.

```
"AvailableDatabases": {
  "local": {
    "DisplayName": "Local",
    "Uri": "DATA SOURCE=oracle:1521/XEPDB1;PERSIST SECURITY INFO=True;USER ID={0};PASSWORD=
    {1}",
    "PhotoUri": "/images/imageLocal.jpg"
  }
  ...
}
```

Výpis 4.7: Konfigurace databází v `appsettings.json`

[19] Práva pro konkrétní záznamy namísto jejich typu.

Wrapping funkcionality trasování

Součástí implementace Core API je taktéž i wrapping funkcionality trasování, kde v původním řešení byl napřímo volán Oracle package, jehož volání způsobovalo zneresponzivnění celé aplikace. Nově je tento problém řešen voláním package na separátním vlákně, provádění je tudíž asynchronní. Spuštění trasování běží nad vlastním Entity Framework contextem, který na rozdíl od toho, který podléhá *dependency injection*, není uvolněn po dokončení požadavku.

```
Thread thread = new Thread(new ParameterizedThreadStart(ExecTracing));
thread.Start(new TracingThreadParams(existingContext, davkaId, ex =>
{
    logger.LogError("Internal error in procedure", ex);
}));

...

private static void ExecTracing(object param)
{
    var threadParams = param as TracingThreadParams;
    try
    {
        using (var db = new ModelContext(
                    threadParams.Context.ConnectionString, threadParams.Context.UserLogin,
                    threadParams.Context.DatabaseVersion, threadParams.Context.Schema))
        {
            db.ExecuteProcedure($"BEGIN PG_ATRS.TRASOVANI({threadParams.DavkaId}); END;");
        }
    }
    catch (Exception ex)
    {
        threadParams.ExceptionHandler(ex);
    }
}
```

Výpis 4.8: Spuštění algoritmu trasování

Nad rámec původní funkcionality je implementováno i sledování průběhu trasování skrze technologii SignalR[20]. Tato technologie je využívána pro real-time komunikaci s klienty a používá protokol WebSocket, alternativně *server-sent events* či *HTTP long polling*, pokud WebSocket nejsou podporované [38].

Sledování a notifikace uživatele je zahájena po nalezení prvního záznamu v tabulce `DAVKY_ZAZNAMY`, ve kterém sloupec `INDEX_ULOHY_DAVKY` je větší než poslední index zapsaný do tabulky `DAVKY_ULOHY`, kam trasování zapisuje své výsledky až po svém skončení. Zmíněný sloupec není záměrně provázán referenční integritou s `DAVKY_ULOHY`, aby algoritmus mohl ve svém průběhu již zapisovat informace o provádění. Toto chování bude v budoucnu upraveno v závislosti na hlubším přepracování způsobu logování z algoritmu trasování. Po nalezení prvního odpovídajícího záznamu v `DAVKY_ZAZNAMY` je zaplánována úloha na pozadí skrze frontu `BackgroundTaskQueue`. Tato fronta je implementována pomocí semaforu (`SemaphoreSlim`), kdy s každou přijatou úlohou je semafor inkrementován a odebranou úlohou dekrementován. Semafor zajistí, že vyzvednutí sledovací úlohy se odblokuje až v momentě, kdy byla nějaká úloha zaplánována.

[20] https://dotnet.microsoft.com/en-us/apps/aspnet/signalr

```
private ConcurrentQueue<JobInfo> jobs = new ConcurrentQueue<JobInfo>();
private SemaphoreSlim signal = new SemaphoreSlim(0);

public void EnqueueJob(JobInfo job)
{
    if (job == null)
        throw new ArgumentNullException(nameof(job));

    jobs.Enqueue(job);
    signal.Release();
}

public async Task<JobInfo> DequeueAsync(CancellationToken cancellationToken)
{
    await signal.WaitAsync(cancellationToken);
    jobs.TryDequeue(out var job);
    return job;
}
```

Výpis 4.9: Fronta pro spouštění sledování úlohy

Vyzvedávání a spouštění úloh sledování řeší služba na pozadí, tzv. *background service*. ASP.NET Core MVC poskytuje mechanismy pro hostování služby v rámci webového serveru formou třídy `BackgroundService`, od které dědí právě služba sledování `TracingWatcher`. Služba běží jako *singleton*, tudíž mezi požadavky je sdílena identická instance. V metodě `ExecuteAsync` běží smyčka, ve které je pomocí `await` čekáno na uvolnění semaforu z `BackgroundTaskQueue`. Po získání zaplánované sledovací úlohy je spuštěno samotné sledování, které už právě využívá technologii SignalR.

```
public class TracingWatcher : BackgroundService
{
    ...
    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            var job = await queue.DequeueAsync(stoppingToken);
            try
            {
                StartJob(job, stoppingToken);
            }
            catch (Exception ex)
            {
                logger.LogError("Error executing tracing watcher", ex);
            }
        }
    }
    ...
}
```

Výpis 4.10: Služba na pozadí pro sledování trasování

SignalR se do ASP.NET integruje pomocí tzv. *hubů*, kde každý hub má svoji URL, v tomto případě `TracingHub` s relativní URL `/ws/v1/trasovani`. Huby, stejně jako služby na pozadí, jsou instanciovány jako singletony a běží po celou dobu života serverové služby. Instance klientské aplikace se při svém startu připojují k tomuto hubu, kde je invokována metoda `OnConnectedAsync`, ve které jsou na základě přihlášeného uživatele (uživatelská

identita je zjištěna na základě přístupového tokenu) klienti rozřazeni do skupin. Výpis 4.11 demonstruje zařazení uživatele na základě jeho identity do příslušné skupiny.

```
public class TracingHub : Hub {
    ...
    public override async Task OnConnectedAsync()
    {
        string name = Context.User.Identity.Name;
        if (name != null)
            await Groups.AddToGroupAsync(Context.ConnectionId, name);

        await base.OnConnectedAsync();
    }
}
```

Výpis 4.11: Zařazení uživatele do SignalR skupiny

Všechny otevřené záložky a okna prohlížeče daného uživatele tedy sdílí jednu skupinu. Zprávy o stavu trasování jsou zasílány skupině, která odpovídá uživateli, který algoritmus trasování spustil. Komunikace směrem klient-server není využívána, jedná se tedy o jednostranné zasílání zpráv ze serveru přes trvalé spojení. Příklad zaslání je uveden ve výpisu 4.12.

```
await hubContext.Clients.Group(userName).SendAsync("TracingStart", data);
```

Výpis 4.12: Zaslání zprávy klientům SignalR skupiny

Reportování o stavu trasování je realizováno smyčkou s periodou 1 sekunda, v rámci které je kontrolována tabulka `DAVKY_ZAZNAMY` na nově vzniklé záznamy pro danou úlohu trasování. Všechny tyto záznamy za poslední sekundu jsou zaslány v jedné zprávě klientům. Smyčka se může přerušit na základě těchto podmínek:

- je vyžádáno ukončení služby (služba se vypíná apod.)
- byl nalezen záznam úlohy `DAVKY_ULOHY` a sloupec `SYS_DAT_UKONCENI` obsahuje neprázdnou hodnotu, tedy algoritmus trasování doběhnul (úloha je zapsána až po dokončení)
- po dobu 5 minut nebyl detekován žádný nový záznam

Pro případ zamrznutí či fatální chyby trasování je nastaven i timeout 5 minut, který zamezí hromadění sledovacích úloh. Pokud algoritmus z nespecifikovaných důvodů po dobu 5 minut nic nezapsal, ale jeho chování je stále korektní, ukončení sledování nemá na jeho provádění vliv. Služba zasílá tři typy zpráv: `TracingStart` (zahájení sledování), `TracingStatus` (zpráva o stavu) a `TracingEnd` s příznakem `WasSuccess`. Úspěch či neúspěch algoritmu je vyhodnocován podle porovnávání řetězců ze sdíleného slovníku na specifikované chybové hlášky. V budoucnu bude algoritmus trasování upraven tak, aby srozumitelněji reportoval svůj výsledek.

```
this.connection = new HubConnectionBuilder()
  .withUrl("/ws/core/v1/tracing", {
    accessTokenFactory: () => this.token,
  })
  .configureLogging(LogLevel.Information)
  .build();
this.connection.start()
this.connection.on("TracingStart", (json) => { ... });
```

Výpis 4.13: Registrace a příjem zprávy v SignalR JavaScript klientovi

Na straně klienta je použita knihovna `@microsoft/singlar`[21]. Knihovna podporuje zasílání přístupového tokenu pro autentizaci uživatele, což právě dovoluje seskupování připojených klientů do skupin podle uživatelů. Výpis 4.13 dává příklad konfigurace nového spojení, jeho spuštění a definici obsluhy pojmenované zprávy ze serverové služby.

4.5.5 Služba Graphics

Tato služba plní účel cache 3D objektů a asociovaných metadat a popisků staveb. Služba je implementovaná v prostředí Node.js s webovým frameworkem Express[22] a využitím knihoven Three.js[23] pro rendering a export 3D modelů a glTF Pipeline[24] pro kompresi těchto modelů.

Endpointy služby, které podléhají akcím nad jednou stavbou, je možné volat klientskými aplikacemi skrze Gateway. Akce pro vynucené sestavení vícero 3D objektů staveb je možné volat pouze pod účtem mající *claim* `ssk-master`, který je ve výchozí konfiguraci přiřazen pouze Keycloak klientovi `ssk-services`, jež využívá Client Credentials Flow. Jedná se tedy o akce dostupné pod servisním účtem, běžní uživatelé/klienti je nemohou volat.

Kontrola aktuálnosti 3D objektu, který agreguje všechny objekty[25] podřízené stavbě (trasy, terminály, body apod.), je prováděna na základě porovnání aktuální časové známky na sloupci `SYS_DAT_ZMENY`. V rámci modulu staveb je zajištěno, že změna na řezech, místnostech či bodech aktualizuje i zmíněný sloupec na stavbě. Sestavené soubory jsou ukládány na souborovém systému, kde každá z dostupných databází má vlastní podsložku a název souboru je dán formátem `{stavba}_export_{timestamp}{specifier}.{extension}`, kde:

- `stavba` – kód stavby
- `timestamp` – časová značka
- `specifier` – podtyp souboru (3D objekt, metadata, popisky)
- `extension` – přípona souboru (`glb`/`gltf`, `json`)

Při požadavku na získání některých dat o stavbě je dotázáno Core API na aktuální hlavičku záznamu stavby, následně je z této hlavičky sestaven očekávaný název souboru. Pokud soubor s takovým názvem existuje (je aktuální), tak je vrácen. Pokud neexistuje, tak buď existuje starší verze souboru, nebo neexistuje žádná. V takovém případě se instanciuje třída `ThreeRenderer`, která sestaví a exportuje scénu s danou stavbou.

Pro renderování stavby jsou načítána data Core API, konkrétně se jedná popisné texty a body pro vykreslení polyline řezů a segmentů, boxů terminálů a značených bodů staveb. Dotazy jsou realizovány skrze GraphQL rozhraní.

Následující text uvede pouze stručný popis implementace sestavení 3D scény a jejího exportu, jelikož tato problematika značně přesahuje rámec této diplomové práce a zaobírá se tématy spojenými s 3D modely, optimalizací vykreslování a komprese modelů, které nejsou součástí tématu této práce.

[21] https://www.npmjs.com/package/@microsoft/signalr
[22] https://expressjs.com/
[23] https://threejs.org/
[24] gltf-pipeline
[25] V budoucnu budou 3D objekty děleny i podle tras a terminálů, tudíž každá tato vrstva bude mít vlastní 3D model. Tímto bude i lépe zajištěna kontrola aktuálnosti modelu.

Pro geometrii je využito bázové třídy `THREE.BufferGeometry` a jejích specifických implementací konkrétně pak: `THREE.CircleGeometry` (značené body), `THREE.TubeBufferGeometry` (spojitá trubka pro reprezentaci tras), `THREE.BoxGeometry` (terminály) a metody `SVGLoader.pointsToStroke`, která vykresluje posloupnost bodů do SVG cesty uložené ve formátu `THREE.BufferGeometry`. Za účelem optimalizace vykreslování je nutné minimalizovat počet zobrazených objektů v dané scéně, což v případě komplexních budov technologických celků (reaktor, strojovna apod.) nelze dosáhnout ořezáním *frustra*[26], jelikož i při značném přiblížení je možné, že se v zobrazované části scény budou nacházet stovky či tisíce objektů. Je tedy nutné tyto objekty slučovat na úrovni geometrií. Naivním přístupem ke sloučení se však ztratí možnost identifikovat jednotlivé objekty v rámci scény (kontextová akce apod.), tudíž je nutné ke každému vertexu (vrcholu) zapsat i atribut (*vertex attribute*, zde pojmenovaný `ssk_id`), který jasně identifikuje příslušnost vertexu k danému objektu SSK. Atribut `ssk_id` je čtveřice 32-bitových čísel, kde první identifikuje typ objektu (objekt řezu, terminál, segment trasy atd.) a 3 další čísla jsou rezervována pro kompozitní klíč identifikující konkrétní objekt. Tento atribut je čten při vyvolání kontextové akce v UI (pravé tlačítko), kde tato akce spustí vrh paprsku (`THREE.Raycaster`), který vrátí index intersektované stěny, ze které je vypočten index vertexu a následně je dohledána v atributu konkrétní čtveřice. Pomocí atributů je taktéž nastavována i barva jednotlivých částí slučovaných objektů. Výsledná *mesh* je sestavena z této geometrie a materiálu (např. `THREE.MeshBasicMaterial`), který je konfigurován pro načítání zmíněných vertexových barev (`vertexColors`). Slučování probíhá pro každý typ (objekty řezu, trasy, terminály, značené body) na úrovni řezu stavby, tedy všechny trasy na řezu jsou reprezentovány jedinou meshí.

Export je prováděn do formátu glTF skrze `THREE.GLTFExporter`. Výstup exportéru je následně zoptimalizován pomocí knihovny glTF Pipeline, která model komprimuje technologií Draco[27]. Technologie je schopná komprimovat a optimalizovat meshe, tudíž snižovat nároky na ukládání a přenos těchto souborů. Experimentálně bylo zjištěno, že úspora velikosti na disku pro modely SSK je průměrně 8–10 násobná.

Z hlediska rychlosti načítání dat nové řešení přináší zásadní zrychlení. Tohoto zrychlení je dosaženo jednak reimplementací dotazování načítání řezu, kde nová implementace provádí jeden dotaz na všechny prvky daného typu (segmenty tras, terminály, značené body a objekty řezu) v dané stavbě. Tímto se minimalizuje počet vykonávaných dotazů, který, jak bylo popsáno v kapitole 2.1.1, významně zhoršuje dobu načtení dat přes vysokolatenční spojení. I přestože přesunutím načítání na server je možné se vyhnout vysoké latenci, která může být mezi klientem a serverem, serverová služba může přistupovat k databázím fyzicky umístěným v jiném regionu. Společnost I&C Energo a její zákazníci běžně provozují geograficky distribuovaná řešení napříč Českou a Slovenskou republikou, tudíž se jedná o validní případ použití.

Čistě optimalizací způsobu načítání bylo dosaženo až **20-násobného zrychlení** na přístupu k databázi v jiném regionu[28]. Do tohoto zrychlení není započteno odhadované 50% zrychlení nad daty načítanými z databáze ve stejném regionu, kde je odezva menší než 1 ms. Toto zrychlení lze přisuzovat reimplementaci. Nad rámec optimalizace načítání je dalšího zrychlení dosaženo cachováním výstupních modelů. V případě přístupu do cache (situace *cache hit*) je zrychlení až 7-násobné oproti načítání z primárního datového zdroje

[26] Viz https://learnopengl.com/Guest-Articles/2021/Scene/Frustum-Culling
[27] https://google.github.io/draco/
[28] Odezva mezi lokalitami byla testována a typicky se pohybuje kolem 10–15 ms.

	Počet prvků stav.	Původní (lokální)	Původní (vzdálený)	Nové (cache)	Nové (sestavení lok.)	Nové (sest. vzd.)
#1	1 500	9 s	1 m 5 s	3 s	5 s	7 s
#2	12 000	21 s	11 m 40 s	4 s	14 s	30 s
#3	45 000	59 s	31 m 27 s	6 s	45 s	1 m 2 s

Tabulka 4.2: Porovnání rychlosti načítání 3D objektů staveb

(Core API). **Celkové testované zrychlení** v případě práce s databází v jiném regionu je tedy **až 300-násobné**. V rámci společného regionu je zrychlení až 10-násobné.

Tabulka 4.2 uvádí konkrétní experimentálně naměřené časy nad reálnými daty systému mezi lokalitami Brno a Třebíč. Původní řešení SSK bylo testováno přímo na instalaci Windows Server, kde je hostována jedna z testovaných databází. Od reálných stavebních objektů je uveden i přibližný počet prvků, ze kterých se daný objekt skládá – segmenty trasy, terminály apod.

4.5.6 Služba FileStorage

Služba FileStorage poskytuje funkcionalitu jednoduchého *document management systému* (DMS). Implementace služby je realizována v .NET 6 s využitím LiteDB[29], což je *embedded* NoSQL databáze, která ke svému běhu nepotřebuje server. Databáze je využívána pro ukládání metadat souborů, které jsou uložené souborovém systému služby.

Pouze servisní účty mající *claim* `ssk-master` mohou volat tuto službu. V odevzdaném řešení se jedná o klienta `ssk-services`, který je autentizován metodou Client Credentials Flow.

Služba složkuje ukládané soubory podle databáze, modulu systému a konkrétního záznamu. Složkováním je předejito omezením souborových systémů na počet souborů ve složce, a taktéž výkonnostním problémům pojících se s velkým množstvím souborů. Pro každý nový soubor je vygenerován globálně unikátní identifikátor (třída `Guid`), v případě .NET se jedná o 128-bitový integer. Do LiteDB jsou ukládány následující metadata:

- `Id` – GUID souboru
- `FileName` – původní název souboru
- `FileSize` – velikost souboru
- `MimeType` – typ souboru (*MIME type*)
- `Module` – příslušnost modulu SSK
- `Entity` – textová reprezentace (složeného) primárního klíče příslušící entity

V případě napojení SSK na některý z komerčních DMS se předpokládá reimplementace této služby, přičemž rozhraní služby bude zachováno a všechny metadata budou ukládány příslušnými prostředky daného DMS (vlastní metadata či využití standardních metadat souboru v daném DMS).

[29] https://www.litedb.org/

4.6 Klientská aplikace

Webová klientská aplikace je implementována ve frameworku Vue.js[30] ve verzi 2[31]. Dále je použit Quasar Framework[32] jako komponentová knihovna a také poskytuje možnosti sestavování aplikace pro různé módy jakými jsou například: PWA[33], SSR[34], hybridní mobilní aplikace[35] či desktop aplikace[36]. Dále se také například využívá knihoven axios[37] (HTTP klient), microsoft/signalr[38] a stejně jako u služby Graphics i Three.js, která je zde použita pro vykreslování sestavených 3D modelů.

Uživatelské rozhraní je responzivní, dodržuje zásady Material designu[39] a je připraveno na ovládání dotykem. V budoucnu bude finalizováno i uzpůsobení zobrazení pro tablety. Mezi podporované prohlížeče se řadí Chrome, Edge, Opera, Safari a Firefox, přičemž funkcionalita byla testována v Edge a Chrome ve verzích 99.0. Uživatelské rozhraní je plně internacionalizované a v současné době lokalizované do češtiny a angličtiny. Uživatelé si mohou svůj preferovaný jazyk kdykoliv změnit v uživatelském profilu. Internacionalizace a lokalizace dat není podporována ani žádána z hlediska projektu SSK.

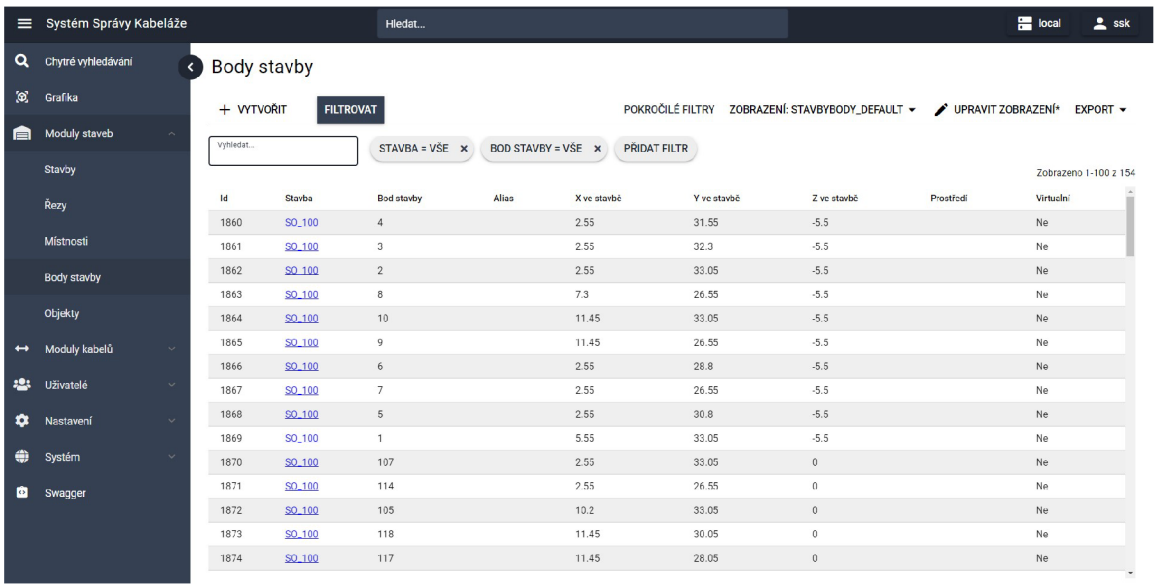
Klientská aplikace podporuje režim PWA[40], tudíž je možné ji skrze podporované prohlížeče nainstalovat do systému (desktop či mobilní OS). Pro implementaci tohoto režimu bylo nutné zavést tzv. *service worker*. Service worker je mezivrstva napsaná v jazyce JavaScript mezi webovým serverem a prohlížečem [20]. Jedná se o službu na pozadí, jejímž cílem je zprostředkovat komunikaci, notifikovat o změnách a ošetřovat přechod mezi online a offline režimy, jelikož PWA aplikace mohou fungovat právě i režimu offline, podporuje-li to daná aplikace.

Implementované uživatelské rozhraní naplňuje stanové požadavky. Rozhraní umožňuje nahlížet, filtrovat, vytvářet, editovat a mazat všechny entity. U vybraných oken je dostupná i takzvaná *inline* editace, tedy editace záznamu přímo v řádku tabulky.

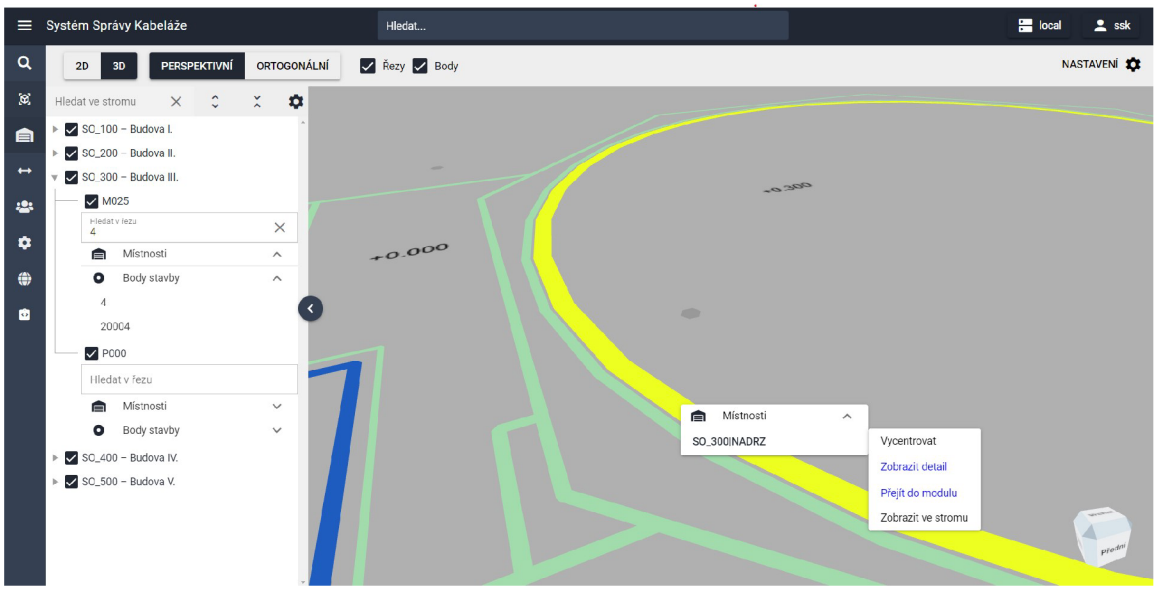
Součástí editačních formulářů je i základní klientská validace, která je nastavena na základě konfigurace viewmodelů služby Core API. Jelikož uživatelské rozhraní (ani služba Core API) v současné době nedisponuje pokročilými validacemi (stavem podmíněné povinnosti sloupců, kontroly souvisejících objektů při mazání apod.), jsou výjimky databázových triggerů a procedur propagovány do uživatelského rozhraní. Jelikož tyto výjimky jsou ošetřované na úrovni databáze a následně jsou dále propagovány s uživatelsky srozumitelnou hláškou, není tato situace považována za zásadně problematickou z důvodu existence plné validace dat na úrovni databáze. V budoucnu bude maximum této validace migrováno směrem do klientské aplikace a služby Core API.

Uživatelské rozhraní taktéž obsahuje i modul grafiky, který je implementován zmíněnou knihovnou Three.js. Modul jako svůj datový zdroj využívá služby Graphics, která mu dodává jednak samotné 3D modely, tak i cachovaná metadata zobrazených elementů (pro identifikaci elementů v grafice) a datové podklady pro vykreslování textových popisků. Modul grafiky je vykreslován prostřednictvím WebGL a nevyžaduje tedy žádné externí

[30] https://vuejs.org/
[31] Eventuální migrace na verzi 3 bude předmětem dalšího vývoje SSK.
[32] https://quasar.dev/
[33] PWA (Progressive Web App) – instalovatelná webová aplikace
[34] SSR (Server Side Rendering) – předsestavení uživatelského rozhraní na serveru.
[35] Sestavení skrze frameworky Cordova či Capacitor do mobilní aplikace.
[36] Realizováno skrze framework Electron.
[37] https://github.com/axios/axios
[38] https://www.npmjs.com/package/@microsoft/signalr
[39] https://material.io/design
[40] PWA režim vyžaduje komunikaci přes protokol HTTPS, která není součástí odevzdání této práce.

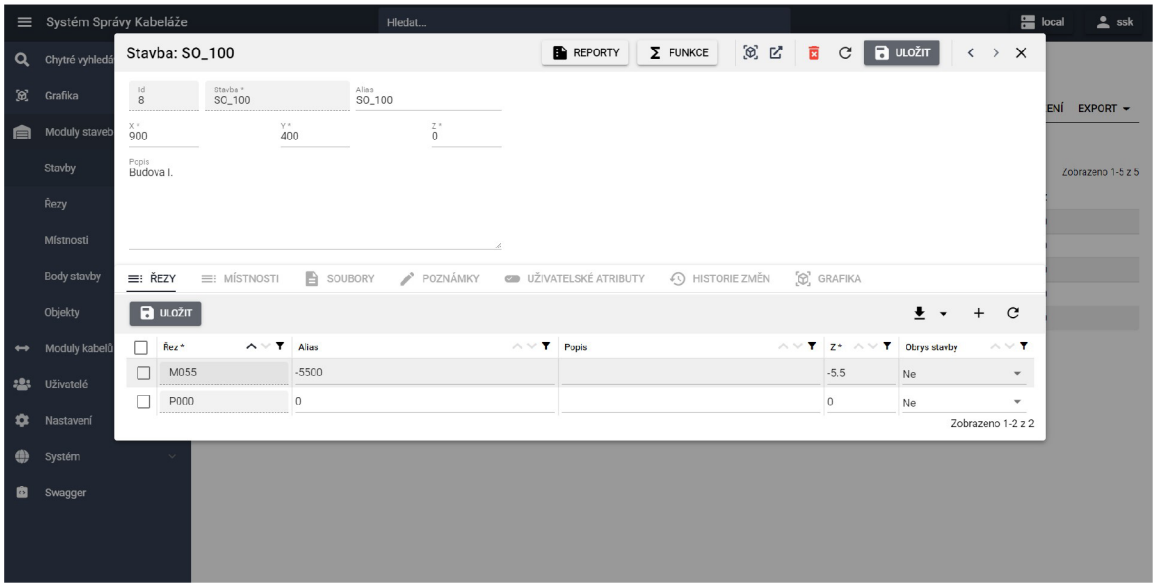


Obrázek 4.8: Základní obrazovka modulu s filtry



Obrázek 4.9: Modul grafiky

závislosti. Interaktivita s 3D modely je realizována skrze čtení vertex atributů, viz podkapitola 4.5.4. Stisknutím pravého tlačítka nad daným elementem je vyvoláno kontextové menu se seznanem všech protnutých elementů (*raycast*). Od těchto elementů lze zobrazit jejich detail, navigovat se do jejich modulu, zacentrovat je v zobrazení a taktéž zobrazit ve stromové struktuře v levé části obrazovky. Součástí modulu je navigační kostka umožňující rychlou rotaci scény kliknutím na danou plochu této kostky. Modul umožňuje 2D i 3D zobrazení. 2D zobrazení zafixuje kameru do pohledu směrem dolů, znemožní rotaci a přepne zobrazení do ortogonální projekce. V rámci 3D zobrazení uživatel může volit mezi perspektivní a ortogonální projekcí. Ovládání scény je uzpůsobitelné i mobilním zařízením s dotykovou obrazovkou, kde tento režim je automaticky detekován na základě detekce platformy frameworku Quasar. Dotykové ovládání je automaticky zapnuté pro platformu `mobile`[41]. V případě hybridních zařízení (notebooky 2v1 apod.) uživatel může v záložce nastavení ovládání ručně přepnout. Modul grafiky je taktéž integrovaný do záložek modulů v alfa-numerické části aplikace, kde je možné spustit kompaktní verzi modulu grafiky parametrizovanou daným záznamem, která se po spuštění automaticky načte.



Obrázek 4.10: Okno detailu entity s integrovaným modulem grafiky

Jednotlivé moduly uživatelského rozhraní (tabulky, detaily a záložky) jsou implementované obecnými komponentami (`DataTable`, `DataDetail`, `DataDetailInput` atd.), které jsou parametrizované metadaty entit získaných z endpointu `metadata` služby Core API. Parametrizací je zabráněno duplikaci kódu, jelikož jednotlivé obrazovky modulů jsou pouze výstupem složeným z parametrizovaných komponent. Pro konfiguraci hlavní tabulky záznamů (stejně tak grafické rozložení detailu záznamu) jsou využity atributy viewmodelů Core API, například `VMSettings` či `VMVisible`.

Přizpůsobení zobrazení nad rámec parametrizace je dosahováno skrze načítání souboru `customization.json`, který pro dané moduly definuje přizpůsobení v předem dovolených rozsazích. V současné implementaci je dovoleno pomocí Vue.js *slotu* dodávat vlastní obsah do hlavičky detailu a definovat nové záložky detailu s vykreslením obsahu právě přes slot. Do

[41] viz https://quasar.dev/options/platform-detection

budoucna bude míra přizpůsobení rozšířena na sloty v těle formuláře. Přizpůsobení záložek definuje název slotu a název souboru komponenty, která bude dynamicky načítána a zobrazována. U záložek je taktéž podporována i ikona a pojmenování záložky z lokalizačního souboru.

```
{
    "any": {
        "headerDetailSlot": [
            "ReportHeaderDetailSlot"
        ]
    },
    "stavby": {
        "customTabs": [
            {
                "displayOrder": 22,
                "name": "custom.tabs.stavbyRezy",
                "icon": "toc",
                "slot": "RezyTabSlot"
            }
        ]
    }
}
```

Výpis 4.14: Přizpůsobení okna detailu skrze `customization.json`

Skrze zmíněné přizpůsobení je realizována integrace menu asociovaných reportů k danému modulu. Integrace reportů (na rozdíl od business akcí, viz podkapitola 4.5.4) je implementována mimo standard WebControlleru, jelikož se jedná o SSK-specifickou funkcionalitu.

4.7 Testování systému

Za účelem ověření komplexní funkcionality systému bylo přistoupeno k uživatelskému testování. Vzhledem k rozsahu systému by bylo nutné vyvinout extrémní úsilí na tvorbu jednotkových, komponentních a integračních testů, které by bylo z ekonomického pohledu nepřínosné. Na systému se po jeho dokončení nepředpokládá intenzivní rozvojová činnost, která by úsilí spojené s tvorbou těchto testů ospravedlnila. Autor nerozporuje smysluplnost testů, pouze konstatuje, že je podstatné nahlížet na testy i ekonomickým pohledem vzhledem k vysokým nákladům na jejich tvorbu a především udržování. Autor nevylučuje, že v budoucnu nebudou pro klíčovou business funkcionalitu napsány jednotkové testy, avšak tato činnost (z pohledu autora) musí následovat ustálení business požadavků a finálního návrhu integrace SSK do procesů information managementu zákazníků.

Uživatelské testy systému jsou formalizovány dokumenty testovacích scénářů, které popisuji jednotlivé body testování, testy a jejich konkrétní kroky. Testovací scénáře vypracované pro SSK jsou zpracovány podle standardní šablony pro testovací scénář společnosti I&C Energo. Jeden testovací scénář se typicky mapuje na konkrétní *business scénář* popsaný v dokumentu cílového konceptu. Scénář je identifikovaný jednoznačným projektově specifickým kódem. V rámci scénáře jsou uvedeny konkrétní body testování, které se následně dělí na jednotlivé testy sestávající se z kroků. Příkladem testovacího bodu je např. ovládání tabulky, příkladem testu pak např. vytvoření záznamu a příkladem kroku např. kliknutí na tlačítko.

Dokument testovacího scénáře slouží jako šablona pro konkrétní testovací běh. V rámci běhu je do dokumentu zaevidováno datum, jméno testera a výsledky jednotlivých testů s případným identifikátorem do systému *issue trackingu*. Dokumenty testovacích scénářů jsou předmětem řízené dokumentace.

Příloha B obsahuje konkrétní ukázku vypracovaného testovacího scénáře pro SSK.

V průběhu celé doby vývoje byl systém průběžně testován jednotlivými členy projektového týmu SSK. Kromě verifikace byla funkcionalita taktéž i validována s vybranými koncovými uživateli systému. Uživatelé byli průběžně seznamováni s vyvíjenou funkcionalitou, která byla v případě potřeby na základě jejich vstupu upravována. V rámci odevzdaného řešení byla většina klíčové systémové funkcionality (chování tabulek, filtrů, interakce mezi moduly) validována s těmito uživateli. Před finálním předáním systému zákazníkům bude systém se svými přizpůsobeními podroben akceptačním testům, kterými bude systém koncovými uživateli a managementem zákazníka schválen pro nasazení.

4.8 Instalace a spuštění

Ke spuštění odevzdaného řešení je vyžadován Docker Compose s Docker Engine ve verzi 20.10.11 a novější. Řešení může být funkční na starších i novějších verzích, ale funkcionalita byla testována na zmíněné verzi Docker Engine.

Pro sestavení ze zdrojů je vyžadována alespoň zkušební verze knihovny Telerik Reporting[42]. Zkušební verzi je možné aktivovat na 30 dní bezplatně. Součástí odevzdání jsou taktéž i sestavené *images*, v případě kterých již není nutné se registrovat na zkušební licenci Telerik.

Spuštění včetně sestavení images je možné provést příkazem `docker-compose up` v kořenovém adresáři projektu. Do Keycloak je zavedena demo konfigurace, ve které jsou výchozí

[42] https://www.telerik.com/download

přihlašovací údaje `ssk` a heslo `ssk`. Pro administrátora Keycloak je pak uživatelské jméno `admin` a heslo `admin`.

Po úspěšné inicializaci služeb a databází je klientská aplikace dostupná na adrese http://localhost:8081/app. Při prvním přístupu bude uživatel přesměrován na přihlašovací okno služby Keycloak, kam zadá výše zmíněné údaje uživatele `ssk`.

Dokumentace rozhraní REST je poskytována skrze Swagger, který je dostupný na adrese http://localhost:8081/swagger/index.html. V rámci definic je zpřístupněno REST rozhraní služeb Core API, Graphics a Gateway. Schéma GraphQL rozhraní je dostupné na adrese http://localhost:8081/api/core/v1/local/viewmodels/graphql. K zobrazení dokumentace lze použít aplikaci `graphiql-app`[43]. Do konfigurace HTTP headeru lze zadat získaný `access_token`, který se zadává ve formátu `Authorization: Bearer {access_token}`.

Odevzdané řešení je taktéž možné provozovat na zmíněném Red Hat OpenShift clusteru. Konfigurace pro nastavení projektu jsou součástí odevzdaného řešení, ale vyžadují existence vzdáleného Gitového repozitáře pro spuštění sestavení ze zdrojů. Nasazení je realizované skrze Helm charts[44]. Řešení je na platformě OpenShift v rámci společnosti I&C Energo nasazané i testované. Pro účely diplomové práce je tento způsob nasazení pouze ilustrační vzhledem k náročnosti přípravy prostředí OpenShift i hardwarovým nárokům, které téměř vylučují provoz clusteru s řešením SSK na běžné pracovní stanici.

Databázové služby jsou dodávány s předkonfigurovaným obsahem, který je perzistován skrze *volumes*. Pro obnovu lze vymazat obsah složek volumes, bližší popis se nachází v `README.MD` v kořenovém adresáři projektu. Po výmazu se při prvním startu obnoví databáze ze zálohy, kterou se inicializuje schéma s demo daty. Obnova zálohy může trvat několik minut, dokončení obnovy je reportováno do konzole kontejnerem Oracle databáze. Dále je také inicializována služba Keycloak včetně jí příslušné databázi na technologii PostgreSQL. Popis kroků instalace je uveden v příloze A.

[43] https://github.com/skevy/graphiql-app
[44] https://helm.sh/

Kapitola 5

Vyhodnocení aplikovaných technik reengineeringu

Autor práce na základě popsaných technik reengineeringu, návrhu a zkušeností získaných implementací navrženého řešení dospívá k následujícím závěrům ohledně reengineeringu dvouvrstvých aplikací.

Reverzní engineering pohledem z vnějšku

Absencí kvalitní technické dokumentace[1] stávajících řešení je provádění reverzního engineeringu komplikované a jako nejefektivnější řešení autor práce vidí kombinaci používání samotného systému s druhotným náhledem do zdrojových kódů za účelem hledání implementace specifické vlastnosti. Interakce s uživateli v procesu reverzního engineeringu stávajícího systému je v očích autora naprosto kritická, v mnohých případech uživatelé používají systém jiným způsobem, než byl navržen.

Obtížná automatická transformace

Techniky automatické transformace popsané Matosem [34] nejsou pro případy takto zásadních technologických skoků vhodné, jelikož cílová implementace využívá typicky diametrálně odlišné knihovny, frameworky a vlastnosti jazyků, které v době původní implementace neexistovaly[2].

Neoddělitelnost od nového vývoje

Ačkoliv se reengineering zabývá přetvářením existujícího řešení, v praxi je vlivem požadavků zákazníků či managementu reengineering úzce spojen i s vývojem nové funkcionality systému. Z projektového pohledu je tato vazba naprosto zřejmá, že součástí přepracování stávajícího systému bude i vývoj nové funkcionality, avšak existující literatura ([27], [34] aj.) se výlučně zaměřuje na reengineering jako izolovaný problém.

Výhodnost strategie top-down

Strategie top-down, u které je nahlíženo na reengineering od vyšší míry abstrakce k nižší, je výhodná z hlediska postavení business scénářů na první místo, jelikož požadavky na

[1] Dokumentace mimo zdrojový kód popisující technické řešení s vyšší abstrakcí

[2] Například asynchronní kód, lambda funkce apod.

systém nejsou typicky stanoveny na úrovni implementace, ale na úrovni business scénářů definovaných business analytiky a specialisty.

Reengineering na služby metodou náhrady případně zaobalení

Z uvedených metod reengineeringu uvedených v kapitole 2.2.7 na základě [29] v kontextu reengineeringu do zcela odlišných technologií lze konstatovat, že pro většinu systému bude využita metoda náhrady. Autor se domnívá, že metoda bude vhodná tam, kde kompletní přepracování funkcionality je nezbytné či ekonomicky výhodné. Komponenty systému, kde toto neplatí, budou pravděpodobně využívat metodu zaobalení, která pro dané komponenty může být dočasným řešením, nežli bude existovat dostatečná motivace pro jejich náhradu. Z pohledu autora je restrukturalizace uplatnitelná pouze tehdy, kdy je technologie implementace totožná či podobná a existuje jasný převod mezi do nové podoby.

Použitím metod reengineeringu lze dosáhnout snížení nákladů na vývoj nového řešení a snížením celkového času, v kterém je možné nové řešení dodat zákazníkovi. Konkrétní úspora se liší projekt od projektu a je silně závislá na odlišnosti výchozí a cílové technologie, a taktéž na množství komponent systému, které lze transformovat či zaobalit.

V případě projektu SSK byla zvolena střední cesta, která kombinuje novou implementaci celé funkcionality, kterou poskytoval původní klient SSK, a zaobalení většiny funkcionality, kterou realizovala plná aplikační logika na straně databáze. Jelikož původní systém byl navržen tak, aby byl schopen zajistit validitu a ochranu dat i při přímém přístupu do databáze, nové řešení využívá existence této aplikační logiky jako poslední instance validace dat.

Kapitola 6

Závěr

Cílem této diplomové práce bylo se seznámit s problematikou reengineeringu, identifikovat možné činnosti při reengineeringu z dvouvrstvých aplikací na třívrstvé, zanalyzovat, navrhnout a implementovat nové řešení stávající dvouvrstvé aplikace.

Kapitola 2 popisuje softwarové architektury, konkrétně pak dvouvrstvé, třívrstvé, n-vrstvé, service-oriented architecture a především architekturu mikroslužeb. Architektury jsou vzájemně porovnávány a jsou diskutovány specifické problematiky a odlišnosti. Možné důvody volby a výzvy spojené s výběrem těchto architektur jsou analyzovány. Problematika reengineeringu je uchopena jak z pohledu metodologie, strategií a přístupů, ale také i z hlediska možných rizik a obecných důvodů pro jeho započetí.

Pro reengineering byla ve spolupráci se společností I&C Energo vybrána reálná aplikace Systému správy kabeláže, která je aktivně využívána pro správu kabelového systému na několika českých elektrárnách, včetně Jaderné elektrárny Dukovany a Temelín. Kapitola 3 se věnuje analýze stávajícího řešení SSK, diskutuje jeho problémy, připomínky uživatelů a mapuje používání SSK včetně pomocných nástrojů. S pomocí metod reengineeringu je navrhnuto nové cloud-native řešení v architektuře mikroslužeb, které je detailně popsáno.

V kapitole 4 je diskutována implementace, kde je popsáno řešení realizovaných služeb. U podstatných aspektů implementace jsou uvedeny i ukázky kódu a funkcionalita je detailně objasněna společně s problémy, které bylo nutné třeba řešit. Konkretně je diskutována integrace externích nástrojů reportingu, vlastní mikro-framework pro zpřístupnění a modifikaci dat přes REST API/GraphQL, zaobalení funkcionality trasování přes technologii SignalR a optimalizace grafického modulu skrze caching.

Aplikované techniky a obecné závěry z použití reengineeringu na převod dvouvrstvých aplikací jsou uvedeny v kapitole 5. Nové řešení kombinuje přístupy náhrady a zaobalení původní implementace. V rozsahu původní klientské aplikace je přistoupeno ke kompletní náhradě, na straně původních databázových funkcionalit je využito zaobalení a zpřístupnění přes služby.

Výstupem diplomové práce případová studie reengineeringu na konkretním příkladu Systému správy kabeláže. Na základě zmapovaných metod reengineeringu je navrhnuto nové řešení systému, které odpovídá současným trendům v návrhu modulárních systémů. Navrhnuté řešení je z hlediska architektury a základní funkcionality systému plně implementováno. Předmětem budoucího vývoje bude dosažení plné business funkcionality původního řešení. Odevzdané řešení obsahuje řez cílového systému, jehož funkcionalita je verifikována pomocí testovacích scénářů, z nichž jeden je součástí přílohy této práce.

Literatura

[1] Abbas, A., Jeberson, W. a Klinsega, V. The Need of Re-engineering in Software Engineering. *International Journal of Engineering and Technology 2049-3444*. Únor 2012, sv. 2.

[2] AVEVA. *AVEVA™ Asset Information Management* [online]. AVEVA [cit. 2021-12-23]. Dostupné z: https://www.aveva.com/en/products/asset-information-management/.

[3] Bass, L., Clements, P. a Kazman, R. *Software Architecture in Practice*. 3. vyd. Addison-Wesley, 2012. ISBN 978-0321815736.

[4] Beaumont, D. *How to explain vertical and horizontal scaling in the cloud* [online]. IBM [cit. 2021-10-15]. Dostupné z: https://www.ibm.com/blogs/cloud-computing/2014/04/09/explain-vertical-horizontal-scaling-cloud/.

[5] Bisbal, J., Lawless, D., Richardson, R., O'Sullivan, D., Wu, B. et al. A Survey of Research into Legacy System Migration. Duben 2022.

[6] Boehm, B. W. Software risk management: principles and practices. *IEEE software*. IEEE. 1991, sv. 8, č. 1, s. 32–41.

[7] Brewer, E. Towards robust distributed systems. In:. Leden 2000, s. 7. DOI: 10.1145/343477.343502.

[8] Chappell, D. A. *Enterprise Service Bus*. O'Reilly Media, Inc., 2004. 247 s. ISBN 9780596006754.

[9] Cloudflare. *What is round-trip time?* [online]. Cloudflare [cit. 2021-10-22]. Dostupné z: https://www.cloudflare.com/learning/cdn/glossary/round-trip-time-rtt/.

[10] Consulting, A. *Mapping Project Risk & Uncertainty* [online]. Alkira Consulting [cit. 2021-12-05]. Dostupné z: https://www.alkiraconsulting.com/mapping-project-risk-uncertainty/.

[11] Drake, M. *Understanding Database Sharding* [online]. Digital Ocean, 07. února 2019 [cit. 2021-10-15]. Dostupné z: https://www.digitalocean.com/community/tutorials/understanding-database-sharding.

[12] Elastic. *What are Beats?* [online]. Elastic [cit. 2022-04-21]. Dostupné z: https://www.elastic.co/guide/en/beats/libbeat/current/beats-reference.html.

[13] Endrei, M., Ang, J., Arsanjani, A., Chua, S., Comte, P. et al. *IBM: Patterns: service-oriented architecture and web services.* Leden 2004. 345 s.

[14] Evans, E. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison Wesley, August 2003. 560 s. ISBN 978-0-321-12521-7.

[15] Fielding, R. T. a Taylor, R. N. *Architectural Styles and the Design of Network-Based Software Architectures.* Disertační práce. ISBN 0599871180. AAI9980887.

[16] Fowler, M. *CQRS* [online]. Martin Fowler [cit. 2021-12-03]. Dostupné z: https://martinfowler.com/bliki/CQRS.html.

[17] Fowler, M. *DomainDrivenDesign* [online]. Martin Fowler [cit. 2021-12-03]. Dostupné z: https://martinfowler.com/bliki/DomainDrivenDesign.html.

[18] Gallaugher, J. M. a Ramanathan, S. C. Choosing a client/server architecture: a comparison of two-and three-tier systems. *Information Systems Management.* Taylor & Francis. 1996, sv. 13, č. 2, s. 7–13.

[19] Gilbert, S. a Lynch, N. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. New York, NY, USA: Association for Computing Machinery. jun 2002, sv. 33, č. 2, s. 51–59. DOI: 10.1145/564585.564601. ISSN 0163-5700. Dostupné z: https://doi.org/10.1145/564585.564601.

[20] Google. *Introduction to Service Worker* [online]. Google [cit. 2022-03-24]. Dostupné z: https://developers.google.com/web/ilt/pwa/introduction-to-service-worker.

[21] Huan, J. L. *Background tasks with hosted services in ASP.NET Core* [online]. Microsoft [cit. 2021-09-28]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/host/hosted-services?view=aspnetcore-5.0&tabs=visual-studio.

[22] IBM. *SOA (Service-Oriented Architecture)* [online]. IBM [cit. 2021-11-23]. Dostupné z: https://www.ibm.com/cloud/learn/soa.

[23] I&C Energo a.s.. *Systém správy kabeláže* [online]. I&C Energo [cit. 2021-12-23]. Dostupné z: https://www.ic-energo.eu/produkty-a-sluzby/inzenyring/cinnosti-a-aplikace/sprava-projektovych-dat/system-spravy-kabelaze.

[24] I&C Energo a.s., Západočeská univerzita v Plzni/Fakulta aplikovaných věd. *FI-IM3/173: Vývoj obecné metodiky a CAE systému pro podporu projektování a správy kabelových systémů.* 2006–2008. Dostupné z: https://backup.isvavai.cz/cep?s=jednoduche-vyhledavani&ss=detail&n=0&h=FI-IM3%2F173.

[25] I&C Energo a.s., Západočeská univerzita v Plzni/Fakulta aplikovaných věd. *FI-IM5/182: Vývoj otevřené metodiky a CAD systému pro podporu projektování nových kabelových systémů složitých technologických celků.* 2008–2010. Dostupné z: https://backup.isvavai.cz/cep?s=jednoduche-vyhledavani&ss=detail&n=0&h=FI-IM5%2F182.

[26] Ivkovic, I. a Kontogiannis, K. A framework for software architecture refactoring using model transformations and semantic annotations. In: *Conference on Software Maintenance and Reengineering (CSMR'06).* 2006, s. 10 pp.–144. DOI: 10.1109/CSMR.2006.3.

[27] Kazman, R., Woods, S. a Carrière, S. Requirements for Integrating Software Architecture and Reengineering Models: CORUM II. In:. Leden 1998, s. 154–163. DOI: 10.1109/WCRE.1998.723185.

[28] Keycloak. *Keycloak – Server Administration Guide* [online]. Keycloak [cit. 2022-03-19]. Dostupné z: https://www.keycloak.org/docs/latest/server_admin/index.html.

[29] Koutsoukos, G., Andrade, L., Gouveia, J. a ElRamly, M. *Technical Report D6.2a: Service extraction.* SENSORIA Project, srpen 2006.

[30] Lewin, M. *What Causes Replication Lag?* [online]. Oracle, 16. ledna 2018 [cit. 2021-10-15]. Dostupné z: https://blogs.oracle.com/mysql/what-causes-replication-lag-v2.

[31] Lewis, J. a Fowler, M. *Microservices* [online]. Martin Fowler, 25. března 2014 [cit. 2021-11-27]. Dostupné z: https://martinfowler.com/articles/microservices.html.

[32] Martin, R. C. *The Principles of OOD* [online]. Robert C. Martin [cit. 2022-01-11]. Dostupné z: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

[33] Massé, M. *REST API Design Rulebook.* O'Reilly Media, Inc, October 2011. 116 s. ISBN 978-1449310509.

[34] Matos, C. *Reengineering Software to Three-tier Applications and Services.* 2011. 141 s. Disertační práce. Department of Computer Science, University of Leicester.

[35] Microsoft. *ASP.NET Core SignalR hosting and scaling* [online]. Microsoft [cit. 2022-04-04]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/signalr/scale?view=aspnetcore-6.0.

[36] Microsoft. *CA1502: Avoid excessive complexity* [online]. Microsoft [cit. 2021-12-04]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/fundamentals/code-analysis/quality-rules/ca1502.

[37] Microsoft. *Communication in a microservice architecture* [online]. Microsoft [cit. 2021-11-28]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture.

[38] Microsoft. *Overview of ASP.NET Core SignalR* [online]. Microsoft [cit. 2022-03-21]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/signalr/introduction.

[39] Microsoft. *Replication Publishing Model Overview* [online]. Microsoft, 16. ledna 2018 [cit. 2021-10-15]. Dostupné z: https://docs.microsoft.com/en-us/sql/relational-databases/replication/publish/replication-publishing-model-overview?view=sql-server-ver15.

[40] MojeID. *Proces komunikace přes OpenID Connect* [online]. MojeID [cit. 2021-10-23]. Dostupné z: https://www.mojeid.cz/dokumentace/html/SeznameniSMojeid/ProcesKomunikacePresMojeid/OpenIDConnect/index.html.

[41] Newman, S. *Building Microservices*. 1. vyd. O'Reilly Media, Inc, 2015. ISBN 978-1-491-95035-7.

[42] Richardson, C. *Pattern: Circuit Breaker* [online]. Microservices.io [cit. 2021-12-03]. Dostupné z: https://microservices.io/patterns/reliability/circuit-breaker.html.

[43] Richardson, C. *Pattern: Database per service* [online]. Microservices.io [cit. 2021-11-28]. Dostupné z: https://microservices.io/patterns/data/database-per-service.html.

[44] Rosenberg, L. H. a Hyatt, L. E. Software re-engineering. *Software Assurance Technology Center*. Goddard Space Flight Center, NASA. 1996.

[45] Spicer, D. Raymond Tomlinson: Email Pioneer, Part 1. *IEEE Annals of the History of Computing*. IEEE. 2016, sv. 38, č. 2, s. 72–79.

[46] Sulyman, S. Client-Server Model. *IOSR Journal of Computer Engineering*. Leden 2014, sv. 16, s. 57–71. DOI: 10.9790/0661-16195771.

[47] Torre, C. de la, Wagner, B. a Rousos, M. *.NET Microservices: Architecture for Containerized .NET Applications*. Microsoft Developer Division, .NET and Visual Studio product teams, 2021. 350 s.

[48] Turner, D. *Why is My Database Application so Slow?* [online]. Redgate, 24. srpna 2017 [cit. 2021-10-22]. Dostupné z: https://www.red-gate.com/simple-talk/development/dotnet-development/database-application-slow/.

[49] Van Vleck, T. Electronic Mail and Text Messaging in CTSS, 1965-1973. *IEEE Annals of the History of Computing*. 2012, sv. 34, č. 1, s. 4–6. DOI: 10.1109/MAHC.2012.6.

[50] W3C Web Services Architecture Working Group. *SOA (Service-Oriented Architecture)* [online]. W3C [cit. 2021-11-23]. Dostupné z: https://www.w3.org/TR/ws-arch/#whatis.

[51] Wilde, N., Buckellew, M., Page, H., Rajlich, V. a Pounds, L. A comparison of methods for locating features in legacy software. *Journal of Systems and Software*. Únor 2003, sv. 65, s. 105–114. DOI: 10.1016/S0164-1212(02)00052-3.

Přílohy

Příloha A

Instalace a spuštění

Následující text popisuje postup spuštění SSK prostřednictvím `docker-compose`.

A.1 Prerekvizity

Seznam nutných prerekvizit:

- Docker a Docker Compose s podporou Linux kontejnerů (testována verze 20.10.11+)
- 15 GB volného místa na disku
- minimálně 16 GB RAM

Prerekvizity pro sestavení ze zdrojů:

- Telerik DevCraft účet (alespoň zkušební)

A.2 Spuštění načtením images

Pro snadnější a rychlejší spuštění bez nutnosti registrace trial účtu Telerik DevCraft je řešení dodáváno i se sestavenými Docker images.

1. Načtení images

   Příkaz `docker load` načte sestavené images pro služby Gateway, Core API, FileStorage, Reporting a Graphics.

```
1 docker load -i .\images\images.tar
```

Výpis A.1: Načtení images do Dockeru

2. Spuštění `docker-compose`

   Příkaz spustí sadu služeb. Pro služby Keycloak, Postgres a Redis budou staženy images z Docker hubu. Po stažení bude řešení automaticky spuštěno.

```
1 docker-compose up
```

Výpis A.2: Spuštění `docker-compose`

A.3 Spuštění sestavením ze zdrojů

1. Registrace na trial Telerik Devcraft

   Balíčky Teleriku jsou nutné pro sestavení Reporting služby. Registrace a následná aktivace zkušební verze je možná na stránkách Telerik[1].

2. Sestavení images

   Přihlašovací údaje získané v prvním kroku se zadají do těchto proměnných prostředí:

```
# Windows:
$env:TELERIK_USERNAME="uzivjmeno"; $env:TELERIK_PASSWORD="heslo"; docker-compose
    build

# Linux:
TELERIK_USERNAME="uzivjmeno" & TELERIK_PASSWORD="heslo" & docker-compose build
```

Výpis A.3: Proměnné prostředí pro instalaci Telerik Reporting

3. Spuštění `docker-compose`

   Příkaz spustí sadu sestavených služeb.

```
docker-compose up
```

Výpis A.4: Spuštění `docker-compose`

A.4 Start aplikace

Po spuštění je aplikace dostupná na adrese http://localhost:8081/app. Výchozí uživatel `ssk` má heslo `ssk`. Pro administraci Keycloak je výchozí uživatel `admin` s heslem `admin`.

Uživatel si může své výchozí heslo změnit přes rozhraní služby Keycloak na adrese http://localhost:8085/auth.

A.5 Provozování v Red Hat OpenShift

Dodatečně je součástí odevzdání i Helm chart pro deployment v prostředí Red Hat OpenShift. Tyto konfigurace předpokládají uložení zdrojových kódů v hostovaném Git repozitáři. Skripty se nalézají v adresáři `deployment`.

[1] https://www.telerik.com/download

Příloha B

Testovací scénář

Testovací scénář SSK pro klíčové uživatele
Základní práce s tabulkou záznamů

Body testování:

1. Ovládání modulu 2
2. Práce s daty v modulu 4

Vlastník (kdo bude zodpovědný za TS): .....

Datum běhu: .....

1. Ovládání modulu

Předpoklady: nejsou

Č.	Test	Kroky testu	Očekávaný výsledek testu	OK / CHYBA
1.	Vstup do aplikace	1. Uživatel si otevře aplikaci SSK. 2. Uživatel zadá své doménové přihlašovací údaje do okna Keycloak. 3. Uživatel se přihlásí. 4. Uživatel z nabízených databází zvolí jednu z dostupných.	Uživatel je přihlášen a nachází se na domovské stránce SSK.	
2.	Základní filtrování v modulu	1. Uživatel vybere jeden z nabízených modulů a přejde do něj. 2. Do pole „Vyhledat“ v horní liště modulu zadá libovolnou hodnotu (nebo její část), která se vyskytuje v některém ze zobrazených sloupců.	Vyfiltrovaná data, kde v každém vyfiltrovaném řádku je žlutě podbarvena vyfiltrovaný podřetězec.	
3.	Filtrování přes vícero polí	1. Uživatel přes tlačítko „Přidat filtr“ vybere název sloupce, který chce filtrovat. 2. Pokud daný typ sloupce umožnuje volbu operátoru, tak vybere některý z nabízených. 3. Do pole „Hodnota“ uživatel zadá filtrovanou hodnotu. 4. Uživatel stiskne „Přidat do filtru“. 5. Uživatel jednou zopakuje kroky 1.-4.	Vyfiltrovaná data, kde každý záznam splňuje obě zadané podmínky.	
4.	Vyčištění filtru	1. Uživatel stiskne tlačítko „Zrušit filtr“	Všechny aplikované filtry budou zrušeny.	
5.	Pokročilé filtrování	1. Na obrazovce modulu uživatel v horní liště stiskne tlačítko „Pokročilé filtry“. 2. Z výběru „Logický operátor“ uživatel vybere operátor „OR“. 3. V levé části zobrazeného okna vybere jeden z nabízených sloupců a stiskne tlačítko „Přidat podmínku“. 4. Uživatel zvolí operátor skrze výběrové pole „Operátor“. 5. Do pole „Hodnota“ zadá filtrovanou hodnotu. 6. Uživatel jednou zopakuje kroky 3.-5. 7. Uživatel stiskne tlačítko „Filtrovat“ v horní liště.	Vyfiltrovaná data, kde každý řádek splňuje alespoň jednu ze zadaných podmínek.	
6.	Změna zobrazení	1. Uživatel v horní liště modulu vybere jednu hodnotu z výběru aktivovaného kliknutím „Zobrazení: XXX“, kde XXX je název aktuálního zobrazení.	Změněné zobrazení indikované změnou v „Zobrazení XXX“ a případnou změnou zobrazených sloupců/filtrů/řazení.	

Č.	Test	Kroky testu	Očekávaný výsledek testu	OK / CHYBA
7.	Úprava zobrazení	1. Uživatel stiskne tlačítko „Upravit zobrazení“. 2. V zobrazeném modálovém okně uživatel zatrhne jeden ze sloupců, který nemá zatržený přepínač „Viditelnost“ a u jednoho již zatrženého zruší zatržení. 3. Uživatel vybere jednu položku pod sekcí „Zobrazené sloupce“, klikne na ni a při držení kliku přetáhne myší tuto položku nahoru/dolů mezi zobrazenými sloupci. 4. Uživatel kolečkem myši přejde do spodní části okna a změní hodnotu v „Název zobrazení“ a stiskne tlačítko „Uložit jako“.	Modul zobrazuje data na základě provedených úprav v horní liště se zobrazuje „Zobrazení: XXX“, kde XXX je zadaný název zobrazení z kroku 4.	
8.	Smazání zobrazení	1. Uživatel stiskne tlačítko „Upravit zobrazení“. 2. Uživatel kolečkem myši přejde do spodní části okna a stiskne tlačítko „Smazat“. 3. Uživatel potvrdí smazání v modálovém okně.	Zobrazení je smazáno.	
9.	Export aktuálního filtru	1. Uživatel stiskne tlačítko „Export“. 2. Uživatel přepne či ponechá přepínač „CSV“/“XLSX“. 3. Uživatel stiskne tlačítko „Export aktuálního filtru“.	Aktuálně vyfiltrované záznamy jsou vyexportovány v požadovaném formátu.	
10.	Export zobrazených	1. Uživatel stiskne tlačítko „Export“. 2. Uživatel přepne či ponechá přepínač „CSV“/“XLSX“. 3. Uživatel stiskne tlačítko „Export zobrazených“.	Aktuálně zobrazené záznamy indikované rozsahem „Zobrazeno 1-X z Y“ jsou vyexportovány v požadovaném formátu.	

2. Práce s daty v modulu

Předpoklady:

- Ovládání modulu

Poznámka:

Č.	Test	Kroky testu	Očekávaný výsledek testu	OK / CHYBA
1.	Tvorba nového záznamu	1. Stisk tlačítka „Vytvořit“ v levém horním rohu lišty. 2. Vyplnění údajů požadovaných pro nový záznam. Uživatel zadá údaje do editovatelných polí. 3. Po vyplnění údajů uloží uživatel záznam stiskem tlačítka „Vytvořit“.	Záznam je vytvořen a modálové okno s detailem vytvořeného záznamu je zobrazeno.	
2.	Zobrazení detailu záznamu.	1. V náhledu nad modulem uživatel klikne na řádek tabulky mimo modře podbarvené odkazy (pokud jsou přítomny). 2. Uživatel prohlíží záznam, v případě zájmu může pokračovat v jiné činnosti (editace). 3. Uživatel uzavře záznam stiskem tlačítka „X“ v pravém horním rohu okna, případně kliknutím mimo modálové okno.	Zobrazen detail záznamu v modálním okně (před jeho uzavřením).	
3.	Smazání záznamu	1. V zobrazeném detailu záznamu uživatel stiskem tlačítka „Smazat“ zadá pokyn ke smazání. 2. Systém vyvolá modálové okno pro potvrzení smazání. 3. Uživatel potvrdí smazání stiskem tlačítka „Smazat“, záznam je smazán.	Záznam je smazán.	
5.	Editace záznamu	1. Uživatel zobrazí detail záznamu. 2. Uživatel edituje potřebné parametry. 3. Uživatel stiskem tlačítka vrátí původní hodnoty záznamu před editací. 4. Uživatel opakuje krok 2. 5. Uživatel uloží změny pomocí tlačítka „Uložit“.	Změněny uživatelem editované hodnoty v záznamu.	

Schvalovací doložka o provedení testů v rozsahu celého testovacího scénáře:

VYHOVUJE	VYHOVUJE S PODMÍNKOU	NEVYHOVUJE

Za správnost odpovídá vlastník TS: …….

Datum: ……