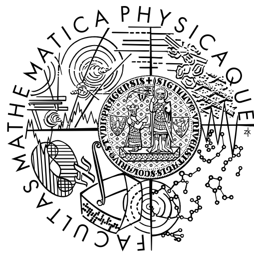


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Michael Ševčík

SaaS platforma pro BI nástroj Metabase

Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Martin Svoboda, Ph.D.
Studijní program: Informatika

Praha 2024

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V .... dne ....

Podpis autora

Rád bych poděkoval všem, bez kterých by tato práce nevznikla. V první řadě mnohokrát děkuji vedoucímu práce, panu RNDr. Martinu Svobodovi, Ph.D. za vstřícnost a trpělivost při konzultacích a všechen čas, který s tímto projektem během posledních tří semestrů strávil. Velký dík taktéž patří mé přítelkyni a rodině za obrovskou podporu a potřebnou gramatickou a stylistickou kontrolu.

Velmi děkuji vám všem!

Název práce: SaaS platforma pro BI nástroj Metabase

Autor: Michael Ševčík

Department: Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Martin Svoboda, Ph.D.

Abstrakt: Tato bakalářská práce se zaměřuje na návrh a implementaci SaaS (Software as a Service) platformy pro poskytování Business Intelligence (BI) nástroje Metabase. Hlavním cílem je usnadnit integraci zákaznických ERP (Enterprise Resource Planning) systémů s Metabase, čímž se urychlí procesy analýzy a tvorby reportů.

Platforma nabízí nástroj pro mapování dat, který zákazníkům umožňuje připojit jejich ERP databáze ke generickému datovému modelu, který následně může být použit nástrojem Metabase pro generování reportů a nástěnek. Systém také využívá Kubernetes pro automatizaci nasazení instancí Metabase, což zajišťuje škálovatelnost a flexibilitu.

Vyvinuté řešení umožňuje snížit čas a náklady spojené s manuální integrací dat a tvorbou skriptů, a tím poskytnout efektivní způsob, jak zákazníci mohou využívat svá data pro business intelligence. Budoucí vylepšení by se mohla zaměřit na zlepšení uživatelského rozhraní a rozšíření podpory pro další databázové systémy.

Klíčová slova: SaaS, Business Intelligence, Metabase, mapování dat

Title: SaaS Platform for Metabase BI Tool

Author: Michael Ševčík

Department: Department of Software Engineering

Supervisor: RNDr. Martin Svoboda, Ph.D.

Abstract: This thesis focuses on the design and implementation of a SaaS (Software as a Service) platform for the Metabase Business Intelligence (BI) tool. The primary objective is to facilitate easy integration of customer ERP (Enterprise Resource Planning) systems with Metabase, thereby accelerating the data analysis and reporting processes.

The platform offers a user-friendly data mapping tool that allows customers to connect their ERP databases to a generic data model, which can then be used by Metabase for generating reports and dashboards. Additionally, the system employs Kubernetes for automating the deployment of Metabase instances, ensuring scalability and flexibility.

The developed solution aims to reduce the time and cost associated with manual data integration and script creation, providing an efficient way for customers to leverage their data for business intelligence. Future enhancements could focus on improving the user interface and extending support for additional database systems.

Keywords: SaaS, Business Intelligence, Metabase, Data Mapping

Obsah

1 Úvod **8**
1.1 Struktura práce 9

2 Analýza existujících řešení **11**
2.1 Integrace dat 11
2.1.1 MS-SQL Server Integration Services (SSIS) 12
2.1.2 Talend Open Studio 15
2.1.3 Srovnání 17
2.2 BI nástroje 19
2.2.1 Microsoft Power BI 19
2.2.2 Tableau 20
2.2.3 Metabase 21
2.2.4 Srovnání 22
2.3 Závěr 24

3 Specifikace **25**
3.1 Pojmy 25
3.2 Požadavky 26
3.2.1 Funkční požadavky 26
3.2.2 Nefunkční požadavky 27
3.3 Koncept 31

4 Design systému **33**
4.1 Architektura systému 33
4.2 UI/UX design 41
4.2.1 Mapovací nástroj 41
4.2.2 Správní aplikace 43

5 Použité technologie **49**
5.1 *Správní aplikace* 49
5.1.1 Backend 49
5.1.2 Frontend 50
5.2 Nasazení 53
5.2.1 Kubernetes 53
5.3 Testování 54

6 Implementace **56**
6.1 Generický model a Metabase nástěnky 56
6.2 Mapování dat 58
6.2.1 Reprezentace relačních modelů 58
6.2.2 Reprezentace mapování relačních modelů 58
6.3 Mapovací nástroj – Mapper 59
6.4 Správní aplikace 63
6.4.1 Implementace modulů 68
6.4.2 Testování 74

Závěr 75

Literatura 76

Seznam obrázků 82

Seznam tabulek 83

Seznam použitých zkratek 84

A Přílohy 85
A.1 Spuštění 85
A.2 Obsah elektronických příloh 90

1 Úvod

Pro jakoukoliv firmu nabízející vzájemně provázané služby je důležité mít v rámci svého ekosystému takový produkt, který usnadní potenciálním zákazníkům cestu k využití dalších služeb. Cílem této práce je návrh a implementace systému, který zastane roli onoho produktu.

Systém vzniká pro konkrétní společnost, kterou pro jednoduchost budeme nazývat obecně jen *Firma*. *Firma*, jejíž zákazníci jsou z většiny výrobní společnosti, se zabývá optimalizací výroby a její hlavní produkty jsou postaveny kolem tohoto problému, proto je pro ni důležité mít přístup ke kvalitním zákaznickým datům. Potenciální zákazníci již většinou využívají nějaký ERP[1] [1] systém, tudíž jsou i jejich vnitřní procesy zmapované a uložené v jejich databázi.

Zpřístupnění dat zákazníků bylo původně řešeno vytvářením importovacích skriptů na míru zákazníkovi, což bylo časově náročné a obtížně škálovatelné. Dnes je pro nové zákazníky zadefinována sada databázových pohledů, které si musí sami implementovat a poskytnout *Firmě*, která na ně napojí nabízené nástroje pro plánování a optimalizaci výroby. Tento proces v reálných situacích mnohdy ústí ve dva problematické scénáře:

1. Nový zákazník nemá odborné znalosti potřebné k vytvoření zmíněných databázových pohledů, proto osloví dodavatele jejich ERP systému s žádostí o nacenění potřebné úpravy. Odhadovaná cena za úpravy dodavatelem je příliš vysoká a zákazník o *Firmou* nabízený produkt ztrácí zájem.

2. Nový zákazník databázové pohledy implementuje, ale v datech má velké množství chyb (jako duplicitní záznamy, nesprávné formáty, chybějící hodnoty atd.), ať už z důvodu chybně nastavených vnitřních procesů či chybné implementace pohledů. Tyto chyby jsou v rozporu s výše uvedeným požadavkem na kvalitní zákaznická data, a tím pádem musí být nejprve odstraněny, což vytváří zdržení v nasazování produktů *Firmy*.

Pro omezení negativních důsledků výše zmíněných scénářů se *Firma* rozhodla vytvořit nový produkt, který by usnadnil vytváření potřebné sady databázových pohledů pro import dat, umožnil vizualizaci takto importovaných dat a zároveň poskytl zákazníkovi okamžitou přidanou hodnotu.

Těmto požadavkům dle *Firmy* nejlépe odpovídala kombinace nástrojů pro mapování dat a BI[2] [2]. Odtud vychází zadání této bakalářské práce, jejíž cílem je navržení a implementace systému, který umožní poskytování předkonfigurovaného webového BI nástroje *Metabase*[3] [3] formou SaaS[4].

[1] Plánování podnikových zdrojů (ve zkratce ERP z anglického *Enterprise Rresource Planning*) je označení systému, jímž podnik (nebo jiná organizace) za pomoci počítače řídí a integruje všechny nebo většinu oblastí své činnosti, jako jsou plánování, zásoby, nákup, prodej, marketing, finance, personalistika, atd.

[2] Označení pro aplikace na podporu rozhodování. Tyto aplikace umožňují vytvářet obchodní, finanční, či jiné analýzy, které slouží jako podpůrný materiál v rozhodovacích procesech.

[3] Open-source BI nástroj umožňující jednoduchou vizualizaci dat, vytváření nástěnek a reportů.

[4] Software jako služba, anglicky *Software as a Service*.

Nástroj bude používat generický datový model vhodný pro většinu zákazníků, což umožní vytvoření základních reportů, které zrychlí adaptaci nástroje zákazníkem. Systém zákazníkům poskytne uživatelsky přívětivý nástroj pro mapování dat z databáze jejich ERP systému na generický datový model a přístup k BI nástroji. Správci systému umožní správu zákazníků, kontrolu mapování jejich dat a snadné nasazení instancí nástroje v Kubernetes klastru[5].

Po zavedení zamýšleného systému by tedy proces vstupu zákazníka do ekosystému *Firmy* vypadal následovně:

1. Správce systému vytvoří zákazníkovi účet, který opravňuje zákazníka k přístupu k nástroji pro mapování dat.

2. Zákazník umožní systému přístup k jeho ERP databázi a téměř samostatně, preferovaně zcela samostatně, vytvoří potřebné mapování v nástroji pro mapování dat.

3. Systém mapování verifikuje a poskytne zákazníkovi přístup k instanci nástroje *Metabase*, který pracuje nad daty zákazníka.

4. Zákazník začne využívat základní předvytvořené reporty v nástroji *Metabase* a ověří, zda ukazují správná data. Tím potvrdí logickou správnost mapování.

5. Zákazníkem vytvořené mapování je využito pro nasazení hlavních produktů *Firmy*, které slouží k plánování a optimalizaci výroby.

Pro zákazníka je nový proces postupnější, jelikož uvádí mezistupeň k využití plánování výroby v podobě analytického nástroje *Metabase*. Taktéž omezuje nutné odborné znalosti k jeho zvládnutí (viz první problematický scénář výše). Z pohledu *Firmy* si žádá méně lidské práce a lze jednodušeji škálovat.

1.1 Struktura práce

Abychom popsali design, funkcionalitu a vývoj námi vytvářeného systému, rozdělíme dále práci na následujících 6 kapitol.

V kapitole 2 *Analýza existujících řešení* se budeme věnovat již existujícím SW systémům z oblasti, pro kterou je náš systém navrhován. Prozkoumáme jejich klíčové funkce a vlastnosti, srovnáme jejich funkcionalitu s požadavky na náš systém a analyzujeme uživatelské zkušenosti.

V kapitole 3 *Specifikace* detailně specifikujeme funkční i nefunkční požadavky. Jasně definujeme všechny funkce a rozdělíme je do modulů. Stanovíme požadavky na výkon, udržitelnost, škálovatelnost, bezpečnost, použitelnost, testovatelnost, interoperabilitu a portabilitu

Kapitola 4 *Design systému* popisuje průběh návrhu tohoto systému, jaká rozhodnutí ho ovlivnila a jak reflektuje požadavky, které jsou na systém kladené. Představíme zde architekturu systému a jednotlivých modulů, přičemž také

[5] Více informací v podsekci 5.2.1 nebo na `https://kubernetes.io/docs/concepts/architecture/`.

ukážeme zohlednění aspektů jako bezpečnost, použitelnost, škálovatelnost a udržovatelnost systému. V závěru kapitoly si představíme design systému z pohledu UX[6] a UI[7].

V kapitole 5 *Použité technologie* budeme prezentovat popis technologií, které byly použity při vývoji systému. Popíšeme použité programovací jazyky, frameworky, knihovny a další nástroje. Zdůrazníme důvody pro výběr daných technologií a jejich vliv na design a implementaci systému.

Kapitola 6 *Implementace* se zaměřuje na technické aspekty implementace. Vysvětlíme detaily klíčových funkcí a modulů a zmíníme se o řešených problémech a výzvách.

V poslední kapitole, *Závěr*, prozkoumáme, jak jsme splnili vytyčené cíle a požadavky kladené na systém, a shrneme hlavní přínosy naší práce. Také diskutujeme o možných budoucích rozšířeních a vylepšeních systému, a navrhujeme směry pro další výzkum.

[6]UX design (z anglického *User Experience*) definuje způsob použití uživatelem.
[7]UI desing (z anglického *User Interface*) popisuje návrh uživatelského prostředí.

2 Analýza existujících řešení

Funkcionalita zamýšleného systému není v oblasti software zcela unikátní a samotný systém by bez integrace s existujícími produkty *Firmy* neměl pro *Firmu* dostatečnou hodnotu. Jak už jsme si ale představili v úvodu, *Firma* si žádá vlastní implementaci systému, kvůli požadavkům na jednoduchost použití a zmíněné pozdější propojení s produkty *Firmy*.

Přesto se pojďme podívat na možnosti řešení tohoto problému v mírně obecnější rovině, abychom dokázali vyvíjený systém zasadit do kontextu již existujícího software a mohli se inspirovat klady nalezených řešení, případně vylepšit jejich nedostatky. Budeme se snažit najít již existující softwarový systém či jejich kombinaci, která se v rámci nabízené funkcionality blíží následujícím třem vybraným funkčním požadavkům *Firmy*. Systém tedy umožní:

1. grafické mapování dat mezi zákaznickou databází a generickým datovým modelem, aby bylo možné vytvořit sadu databázových pohledů, které transformují zákaznická data na daný model,

2. vytváření základních reportů, které lze přenášet mezi zákazníky, aby tato činnost nemusela být vykonávána manuálně,

3. běžnou BI funkcionalitu (vizualizaci dat, vytváření reportů, atd.).

Jelikož se nepodařilo najít jeden systém, který by splňoval všechny tři výše uvedené požadavky najednou, budeme muset uvažovat o kombinaci dvou či více systémů. Požadavek číslo 1 nejlépe odpovídá kategorii nástrojů pro integraci dat, zbylé dva odpovídají právě BI nástrojům. Tyto dvě kategorie si představíme v následujících dvou sekcích – Integrace dat a BI nástroje. Závěrem kapitoly si provedenou analýzu možných řešení shrneme.

2.1 Integrace dat

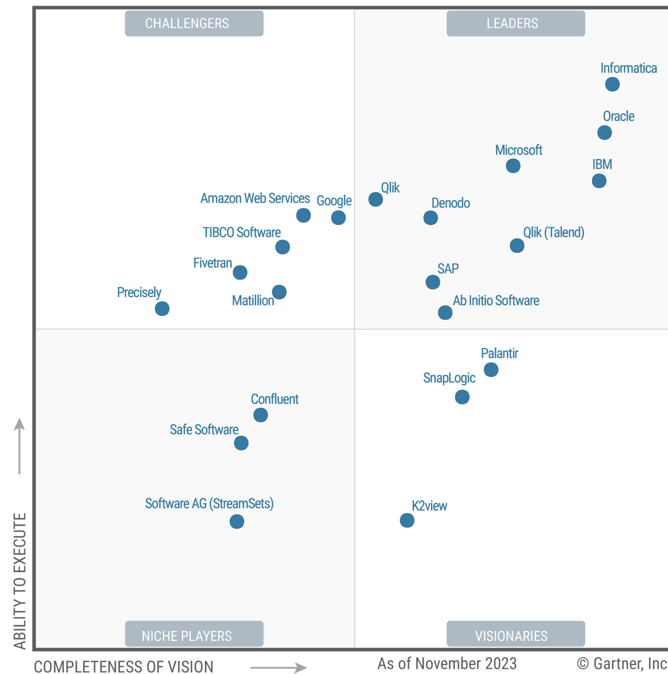
Nástrojů pro integraci dat je mnoho a některá řešení jsou kvalitnější než jiná. Využijeme-li srovnání nástrojů na integraci dat od poradenské společnosti Gartner [4], které je ilustrované na obrázku 2.1, a zohledníme-li požadavek číslo 1 z úvodního textu této kapitoly, můžeme výběr zúžit na nástroje:

- MS-SQL Server Integration Services (SSIS) [5],

- Talend Open Studio [6].

Obrázek 2.1 znázorňuje *Gartner Data Integration Magic Quadrant*, kde jsou nástroje hodnoceny dle kompletnosti vize a schopnosti jí dosáhnout (anglicky *completeness of vision and ability to execute*). My jsme z výběru vybrali SSIS a Open Studio, jelikož jsou dostatečně rozšířené a bylo u nich možné ověřit zmíněný požadavek číslo 1.

Tyto nástroje si podrobněji rozebereme a vzájemně porovnáme, včetně jejich schopnosti splnit zmíněný požadavek číslo 1.



Obrázek 2.1 Gartner Data integration Magic Quadrant, zdroj: Gartner, 2023 [4]

2.1.1 MS-SQL Server Integration Services (SSIS)

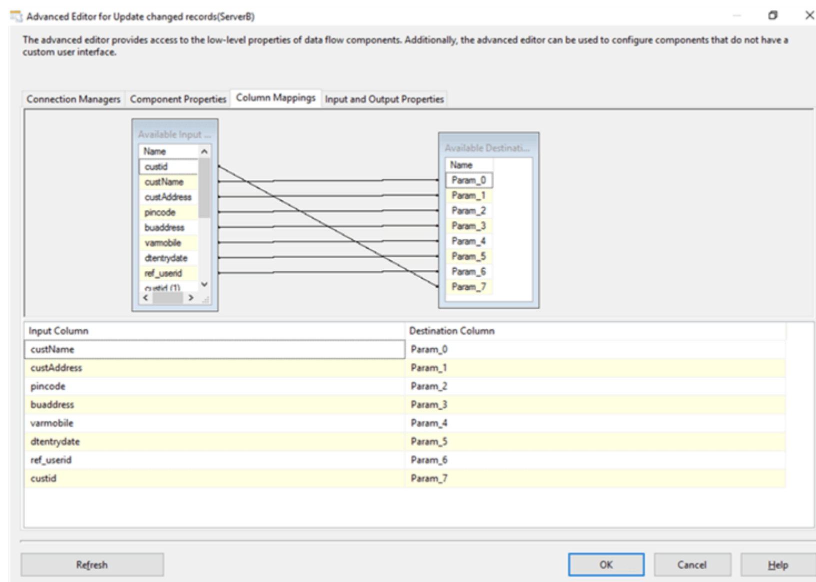
Microsoft SQL Server Integration Services (SSIS) je platforma pro vývoj podnikových řešení pro extrakci, transformaci a načítání dat (ETL[1]). SSIS je součástí produktu Microsoft SQL Server [7], což je databázový a analytický systém.

Extrakce dat

SSIS nabízí širokou škálu možností pro extrakci dat z různých zdrojů. Mezi nejběžnější typy podporovaných zdrojů dat patří [8]:

- Relační databáze – SSIS podporuje přímé připojení k mnoha relačním databázovým systémům, včetně SQL Server, Oracle[2], MySQL[3] a PostgreSQL[4]. Pomocí komponent SSIS, jako je Data Flow Task[5], lze z těchto databází extrahovat data do datových sad, které lze dále zpracovávat.
- XML soubory – Také umožňuje načítat data z XML souborů a transformovat je do požadovaného formátu. To je zvláště užitečné pro integraci dat z externích systémů, které umožňují export dat pouze ve formátu XML.
- Webové služby – SSIS podporuje přímé připojení k webovým službám a extrakci dat z nich. To umožňuje integraci dat z cloudových aplikací a webových API[6].

[1]Jedná se o formu integrace dat, tedy extrakce, transformace a nahrání dat z různých zdrojů (anglicky *Extract, Transform and Load*).
[2]Více informací na `https://www.oracle.com/database/`.
[3]Více informací na `https://www.mysql.com/`.
[4]Více informací na `https://www.postgresql.org/`.
[5]Více informací na `https://learn.microsoft.com/en-us/sql/integration-services/control-flow/data-flow-task`
[6]Webové rozhraní (z anglického *Application Programming Interface*).



Obrázek 2.2 Mapování sloupců v SSIS, zdroj: Patel Bhavesh, 2017 [12]

Kromě výše uvedených typů zdrojů dat SSIS umožňuje extrahovat data z dalších zdrojů, jako jsou například soubory CSV[7], textové soubory a soubory Excel[8] [8].

Tento proces je zásadní pro shromažďování dat z různých zdrojů do jednoho úložiště ve formě, která je vhodná pro další zpracování.

Transformace dat

Po extrakci dat SSIS umožňuje čistit data a odstraňovat chyby v podobě například duplicitních záznamů, chybějících hodnot a nedodržení standardů formátů. Dále umožňuje normalizovat data, aby se eliminovala redundance a zlepšila se konzistence dat. Umožňuje také agregovat data a vytvářet statistické reporty [9].

Uvedené činnosti zahrnují operace jako sloučení, rozdělení a mapování dat, či nahrazení hodnot záznamů [10]. Pro mapování SSIS nabízí vizuální editor, jak ilustruje obrázek 2.2. Na obrázku vidíme mapování mezi sloupci dvou tabulek. Jednotlivé sloupce zdrojové tabulky jsou se svými protějšky spojeny čarami pro znázornění mapování.

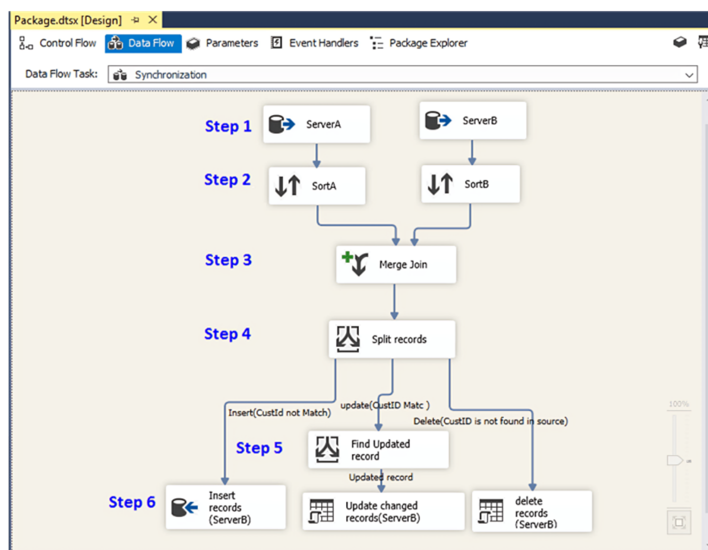
Celý integrační/transformační proces lze shrnout do tzv. balíčků (anglicky *packages*). Balíčky lze snadno vytvářet pomocí „drag-and-drop“ (lze přeložit jako „táhni a pusť“) komponent a intuitivního rozhraní – SSIS Designer [11], což ilustruje obrázek 2.3.

Automatizace a integrace s dalšími nástroji

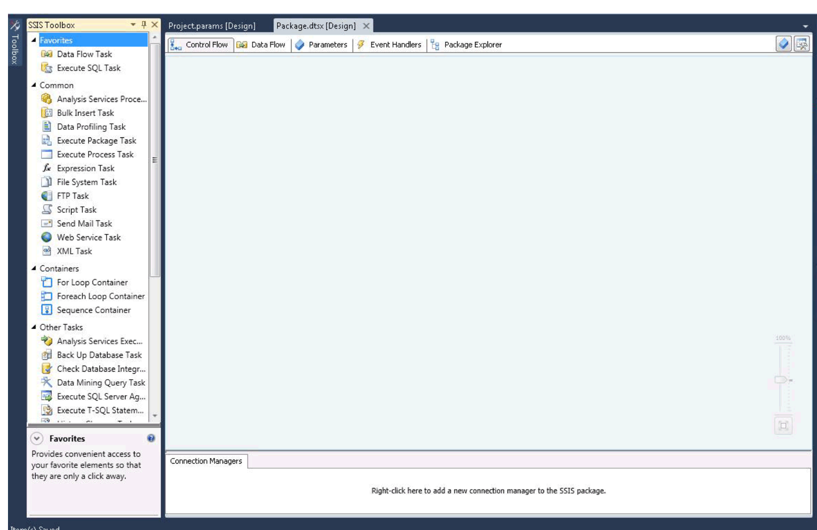
Automatizace úloh zahrnuje plánování spuštění úloh ETL v konkrétní čas nebo v opakujících se intervalech, definování závislostí mezi úlohami pro vytváření komplexních workflow a odesílání e-mailových upozornění o stavu úloh [14].

[7]Jednoduchý textový řádkově orientovaný formát s hodnotami oddělenými čárkami (z anglického *Comma-Separated Values*)

[8]Soubory v proprietárním formátu tabulkového procesoru Microsoft Excel.



Obrázek 2.3 SSIS Designer integračních balíčků, zdroj: Patel Bhavesh, 2017 [13]



Obrázek 2.4 Prostředí nástroje SSIS Designer, zdroj: Microsoft, 2023 [18]

Jako součást ekosystému Microsoft SQL Server SSIS se dobře integruje s dalšími nástroji firmy Microsoft, jako jsou SQL Server Analysis Services (SSAS) a SQL Server Reporting Services (SSRS) [15]. Kromě toho je možné balíčky rozšiřovat o vlastní kód díky podpoře platformy .NET Framework [16], která je také vytvořena Microsftem.

Vytvoření databázových pohledů

Pomocí grafického editoru SSIS Designer můžeme přehledně vytvářet integrační balíčky. Prostředí SSIS Designer ilustruje obrázek 2.4. Tyto balíčky lze dle návodu z edukačních stránek firmy Microsoft publikovat jako databázové pohledy [17].

Uvedeným postupem lze splnit požadavek číslo 1 definovaný v úvodním textu této kapitoly.

Shrnutí

MS-SQL Server Integration Services je silný nástroj pro podnikové ETL řešení a mimo jiné i pro náš případ. Jeho schopnosti extrakce, transformace a načítání dat, spolu s funkcemi pro automatizaci a integraci s dalšími produkty firmy Microsoft z něj činí jeden z klíčových nástrojů pro správu a analýzu dat.

2.1.2 Talend Open Studio

Talend Open Studio je open-source platforma pro ETL. Je široce používán, obzvláště v oblasti financí a IT, a patří mezi nejlepší open-source nástroje na integraci dat [19].

Extrakce dat

Open studio nabízí robustní funkce pro extrakci dat z široké škály zdrojů, čímž umožňuje efektivní shromažďování dat pro pozdější zpracovaní. Mezi podporované zdroje dat patří [20]:

- Relační databáze – Open studio se bezproblémově připojuje k většině databázových systémů včetně MySQL, PostgreSQL, Oracle a Microsoft SQL Server. Pomocí komponent, jako je *tDBInput*, lze z těchto databází extrahovat data do datových sad pro další zpracování.
- Aplikace pro správu a analýzu dat – open studio podporuje extrakci dat z různých aplikací, jako jsou CRM[9] systémy (Salesforce), ERP systémy (SAP[10]) a cloudové úložiště (Amazon S3[11]). Specifické komponenty pro dané aplikace usnadňují proces extrakce.
- Webové služby – Talend Open Studio umožňuje přímé připojení k webovým službám a extrakci dat z nich.

Prostředí nástroje ilustruje obrázek 2.5. Můžeme vidět designer (prostředí pro návrh) integračních úloh (aglicky *jobs*). Tento designer umožňuje návrh a vizualizaci ETL procesu, tedy toku dat mezi jednotlivými komponentami procesu. Návrh integrační úlohy je překládán do kódu v jazyce Java[12], což umožňuje úpravy nad rámec schopností vizuálního editoru [21].

Transformace dat

Podobně jako SSIS i Open Studio má celou řadu komponent pro čištění, normalizaci, agregaci a další transformace dat. Na rozdíl od SSIS Open Studio je vybaveno širší škálou připravených komponent implementovaných na míru externím systémům.

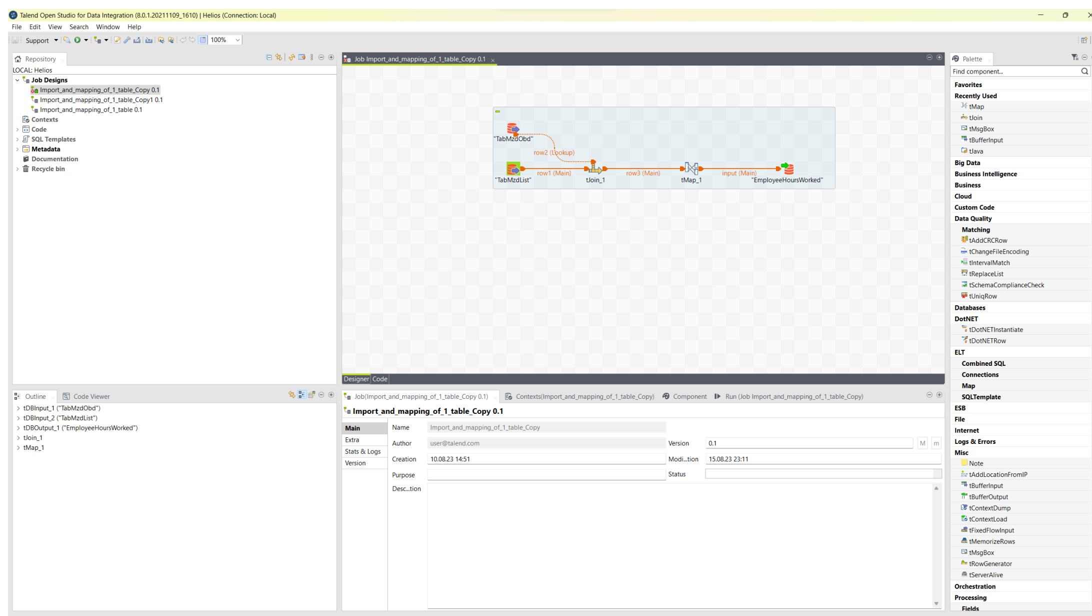
Open Studio například podporuje specifické komponenty pro integraci s Salesforce, SAP a dalšími populárními systémy, jak bylo zmíněno výše. Tyto komponenty

[9] CRM znamená řízení vztahů se zákazníky (anglicky *Customer Relationship Management*). Nástroje pro CRM pomáhájí se správou zákazníků a interakcí s nimi.

[10] Více informací na `https://www.sap.com/`.

[11] Více infromací na `https://aws.amazon.com/s3/`.

[12] Objektově orientovaný programovací jazyk. Více informací na `https://www.java.com/`.



Obrázek 2.5 Prostředí nástroje Talend Open Studio – Job Designer

umožňují využít specifické funkce daných systémů pro optimalizaci transformačních procesů a dosažení vyššího výkonu.

V oblasti mapování dat Open Studio nabízí grafický editor, jak ukazuje obrázek 2.6. Můžeme vidět, že se grafické rozhraní v tomto případě poměrně podobá SSIS (obrázek 2.2), avšak krom prostého znázornění mapování nabízí vyšší uživatelský komfort v podobě možnosti prohledávat názvy sloupců a informacích o datových typech.

Automatizace a integrace s cloudem

Talend Open Studio podporuje automatizaci celého ETL procesu. Jednotlivé úlohy lze rozdělit do tzv. *jobs* (jednotek práce), jejich exekuce může být periodicky plánována či podmíněna nějakou událostí [22].

Talend Open Studio je kompatibilní s cloudovými službami, jako je AWS[13] nebo Azure[14], což umožňuje snadnou škálovatelnost a zjednodušuje údržbu takového řešení [23].

Vytvoření databázových pohledů

V Open Studiu za pomocí ELT[15] komponent dokážeme také vytvořit databázové pohledy [24], avšak ne tak jednoduše jako v případě SSIS. Jak upozorňuje dokumentace [24], pro použití ELT komponent je nutným předpokladem znalost SQL[16].

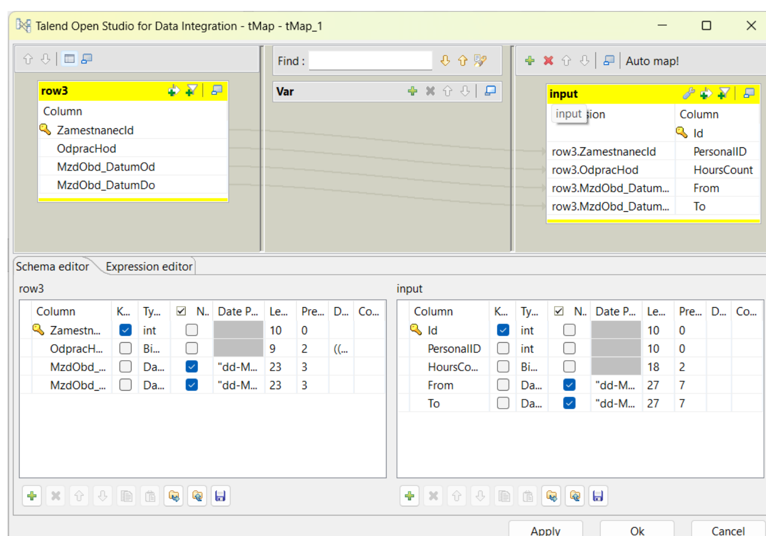
---

[13] Více informací na `https://aws.amazon.com/`

[14] Více informací na `https://azure.microsoft.com/`

[15] Jedná se o alternativní formu integrace dat, při které fáze načítání předchází transformaci dat, tedy extrakce, nahrání a transformace dat z různých zdrojů (anglicky *Extract, Load and Transform*).

[16] Strukturovaný dotazovací jazyk (z anglického *Structured Query Language*).



Obrázek 2.6 Mapování sloupců v Talend Open Studiu

Zde má výhodu SSIS, jelikož je navrženo primárně pro Microsoft SQL Server. Co se týče Open Studia, je znát, že je navrženo obecněji.

Shrnutí

Talend Open Studio je silný nástroj pro podnikové řešení ETL. Jeho schopnosti extrakce, transformace a načítání dat, spolu s funkcemi pro automatizaci a integraci s cloudovými službami, z něj činí jeden z klíčových open-source nástrojů pro správu a analýzu dat. Je tak ideální pro firmy s omezeným rozpočtem hledající komplexní a škálovatelné řešení pro integraci jejich dat.

2.1.3 Srovnání

Jak můžeme vidět v tabulce 2.1, která srovnává základní vlastnosti výše popsaných nástrojů, oba jsou velmi podobné. Nyní se tedy zaměříme na jejich nuance.

Podpora komunitních komponent dělá z Open Studia versatilní nástroj, který působí uživatelsky přívětivějším a přehlednějším dojmem nežli SSIS. V neposlední řadě je nemalým benefitem možnost využít open-source verzi tohoto nástroje.

Na druhou stranu SSIS je velmi robustní nástroj, který nabízí obdobnou funkcionalitu jako Open Studio. Jeho přidanou hodnotou je velmi snadná integrovatelnost s rodinou nástrojů firmy Microsoft, což z něj dělá perfektní volbu při využití s Power BI (viz podsekce 2.2.1).

Z výše uvedeného nelze definitivně říct, který nástroj je obecně lepší. Volba tak závisí hlavně na kontextu užití. Jedním z faktorů může být znalost ekosystému a jazyka pro skriptování srovnávaných nástrojů – C# a VS.NET v případě SSIS a Java v případě Open Studia.

Vlastnost	Microsoft SSIS	Talend Open Studio
Typ nasazení	Desktop[a]	Desktop[a]
Operační systém	Windows	Multiplatformní
Podpora zdrojů dat	Rozsáhlá, dobrá integrace s ekosystémem Microsoft	Rozsáhlá, snadno využitelné komunitní konektory
Prostředí mapování dat	Vizuální	Vizuální
Možnosti transformace dat	Robustní	Rozsáhlé
Automatizace ETL	Ano	Ano
Kvalita dokumentace	Rozsáhlá a podrobná	Dobrá – mix oficiální a komunitní
Vytváření databázových pohledů	Ano	Ano, ale ne tak jednoduše
Jednoduchost použití	Jednodušší integrace s nástroji Microsoft	Přehlednější prostředí, obecně jednodušší
Oficiální podpora	Dostupná	Pouze komunitní; možná u placené verze
Typ licence	Proprietární	Open-source (Apache 2.0)
Cena	V ceně licence SQL Serveru	Základní verze zdarma

Pozn: [a] Talend i Microsoft nabízí podobné produkty s možností cloudového nasazení. Těmito nástroji jsou Talend Data Fabric, respektive Azure Data Factory.

Tabulka 2.1 Porovnání důležitých vlastností nástrojů pro integraci dat

Obrázek 2.7 Gartner BI Magic Quadrant, zdroj: Gartner, 2021

2.2 BI nástroje

Sektor BI nástrojů je velmi rozsáhlý a probíhá v něm neustálý vývoj. Inspirujme se tedy výběrem nástrojů z oblasti BI od firmy Gartner Inc [25], který ilustruje obrázek 2.7.

Výběr označuje nástroje *Microsoft Power BI* [26] a *Tableau Desktop* [27] jako lídry daného sektoru. Krom těchto dvou je pro kontext naší práce také zajímavý nástroj *Metabase*, který budeme integrovat do našeho systému.

Zmíněné nástroje si podrobněji rozebereme v následujících podsekcích a uvedeme, jak splňují požadavky na funkcionalitu z úvodu kapitoly.

2.2.1 Microsoft Power BI

Microsoft Power BI je nástroj pro business intelligence. Nabízí interaktivní vizualizace dat a schopnosti business analytics[17] s rozhraním dostatečně jednoduchým pro koncové uživatele k vytváření svých vlastních reportů a nástěnek.

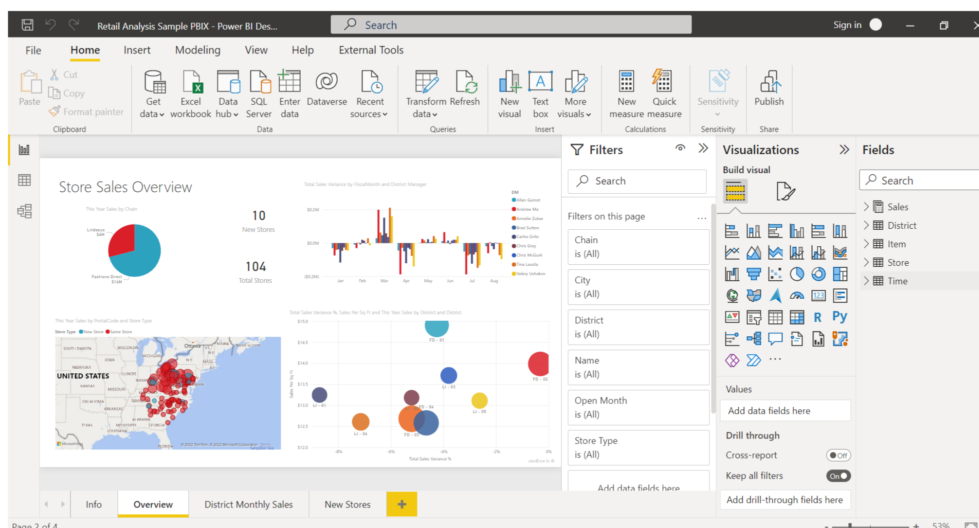
Power BI je nejpoužívanějším nástrojem pro business intelligence na trhu [28] a má hodnocení 4,4 hvězdiček z celkových 5 na základě více než 3 000 recenzí na platformě firmy Gartner Inc. [29] (únor 2024).

Nástroj umožňuje uživatelům připojit se k různým datovým zdrojům, včetně souborů Excel, systému SQL Server, či k Microsoft SharePoint[18] a mnoha dalším [30].

Mezi výhody Power BI dokáže zpracovat velké datové sady [31]. Také nabízí pokročilé analytické funkce, jako jsou Quick Insights, AI Insights a Analyze feature, které umožňují uživatelům snadno najít zajímavé informace a trendy v datech [32]. Dále umožňuje vytvářet vizualizace pomocí předem vytvořených šablon nebo

[17] Řešení manažerských a obchodních problému za pomoci analýzy dat.

[18] Více informací na `https://www.microsoft.com/cs-cz/microsoft-365/sharepoint/collaboration`.



Obrázek 2.8 Prostředí Power BI – vytváření nástěnky, zdroj: Microsoft, 2023, dostupné z: `https://learn.microsoft.com/en-us/power-bi/fundamentals/media/desktop-what-is-desktop/what-is-desktop-01.png`

vlastních návrhů. Uživatelé mohou také vytvářet interaktivní nástěnky, které nabízejí rychlý přehled o klíčových ukazatelích výkonnosti [31].

Interaktivní nástěnky ilustruje obrázek 2.8. Můžeme vidět, že Power BI nabízí celou plejádu možností vizualizace dat zabalenou do známého designového stylu firmy Microsoft. Uživatelé si mohou vybírat vizualizace od nejjednodušších, jako jsou koláčové grafy, až po vizualizace složitější, jako jsou například geografická data.

Power BI je zejména vhodný pro střední až větší firmy[19], v nichž je potřeba rychle vytvářet vizualizace a nástěnky bez nutnosti rozsáhlého školení. To neznamená, že by byl Power BI z hlediska funkcionality omezený, jelikož pro zkušenější uživatele nabízí řadu pokročilejších funkcí [34].

Shrnutí

Microsoft Power BI je velmi oblíbený nástroj pro business intelligence, který nabízí jak základní funkcionalitu většiny BI nástrojů, tak pokročilé analytické funkce, jako jsou Quick Insights, AI Insights a Analyze feature.

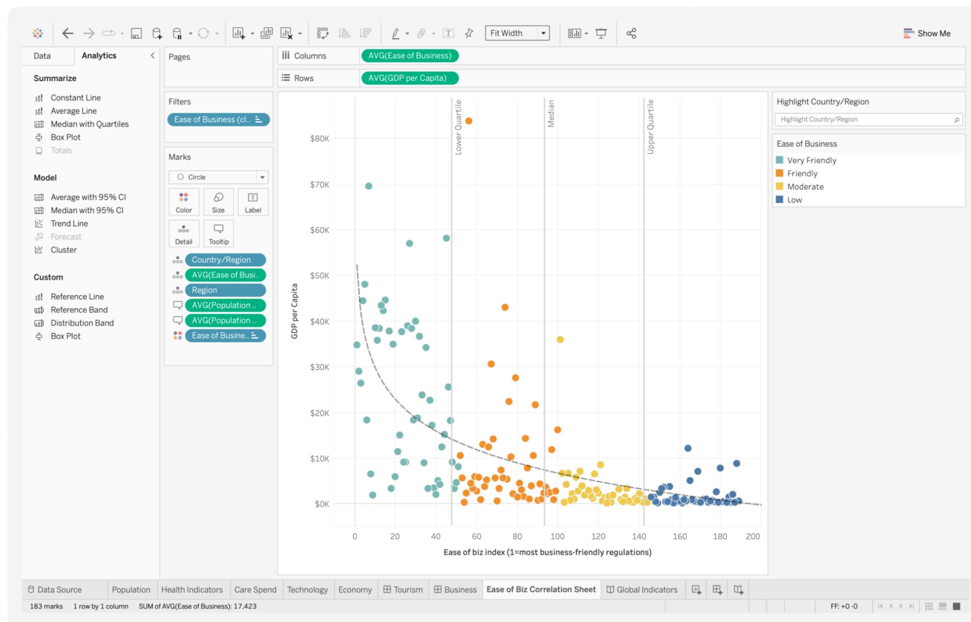
V rámci našich požadavků na funkcionalitu, Power BI rozhodně oba dva splňuje, přičemž přenositelnost je zajištěna možností vytvářet a využívat zmíněné předem vytvořené šablony.

2.2.2 Tableau

Tableau je další populární nástroj pro BI. Umožňuje uživatelům s různou úrovní zkušeností analyzovat a sdílet firemní data interaktivním způsobem. Je druhým nejpoužívanějším nástrojem pro BI hned za Power BI [28] a také získal na základě téměř 4 000 recenzí 4,4 hvězdiček z celkových 5 [35].

Tableau, podobně jako Power BI, lze připojit k široké škále datových zdrojů. V oblasti vizualizace dat, uživatelé reportují, že Tableau nabízí větší míru přizpů-

[19] Pro jednoduchost firmu považujeme za střední od 50 zaměstnanců a větší od 250.



Obrázek 2.9 Prostředí nástroje Tableau, zdroj: Talend, 2022 [38]

sobitelnosti. Na druhou stranu začínající uživatelé se v záplavě možností mohou ztrácet [36].

Prostředí nástroje Tableau Desktop ilustruje obrázek 2.9. Můžeme vidět, že také Tableau nabízí moderní design. Ten se však značně liší od Power BI a stylu aplikací, jako je známý Microsoft Excel. Možná i proto uživatelé hlásí strmější křivku učení (míra porozumění v závislosti na délce učení).

Co se týče splnění požadavku číslo 2 z úvodu této kapitoly, Tableau nabízí přenositelné pracovní knihy (anglicky *workbooks*). Tyto knihy lze přenášet mezi instancemi nástroje Tableau, přičemž po přenosu je nutné aktualizovat zdroje dat [37].

Z výše uvedeného vyplývá, že cílovou skupinou nástroje Tableau budou větší firmy, které nejsou citlivé na cenu a mají možnost využít jeho pokročilé funkce.

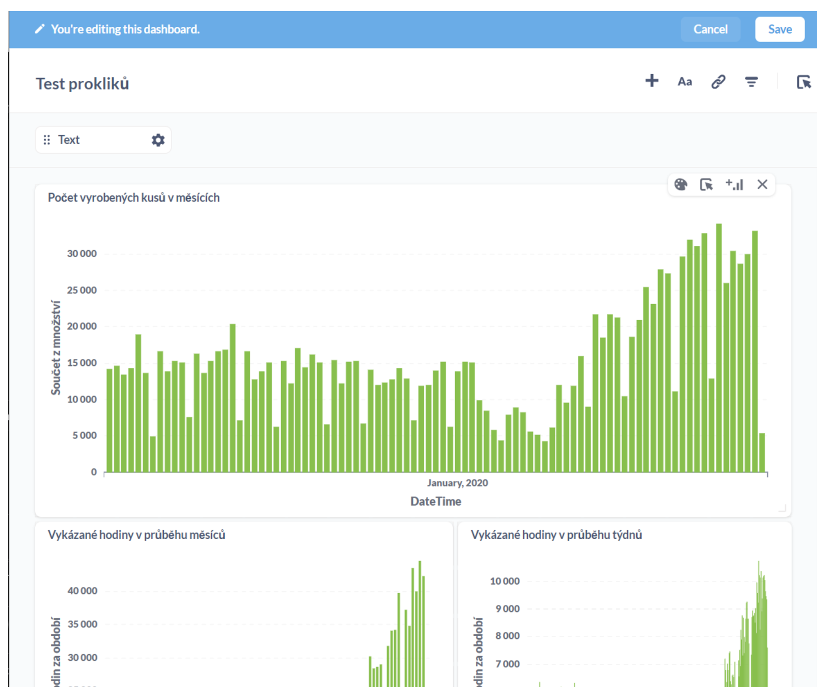
2.2.3 Metabase

Metabase je třetím analyzovaným nástrojem. Na rozdíl od předchozích dvou není tak rozšířený, ani nenabízí tak širokou škálu pokročilých funkcí, navíc nenabízí desktopové rozhraní. Má však i několik výhod. Jednou z nich je open-source licence základní verze, druhou jednoduchost použití, za kterou vděčí omezenější funkcionalitě.

Z hlediska podporovaných zdrojů dat je Metabase spíše zaměřený na databázové systémy a cloudové datové služby [39]. Je tak ideální pro malé a střední podniky, startupy nebo týmy s omezenými znalostmi BI, které nepotřebují integrovat velké množství zdrojů dat jiných, než jsou podporované.

Prostředí nástroje ilustruje obrázek 2.10. Můžeme vidět, že je oproti Tableau a SSIS značně jednodušší, resp. pokročilejší funkce jsou schované za rozklikávacími menu.

Zatímco Tableau v rámci požadavku číslo 2 z úvodu této kapitoly nabízí pracovní knihy, Metabase nedisponuje žádným standardizovaným systémem pro



Obrázek 2.10 Prostředí nástroje Metabase

tvorbu šablonových reportů. My se s tímto faktem budeme muset vypořádat implementací vlastního řešení pro sdílení reportů.

2.2.4 Srovnání

Pojďme se nyní podívat na vzájemné srovnání BI nástrojů, kterým jsme se věnovali. Jak můžeme vidět v tabulce 2.2, první dva zkoumané nástroje jsou velmi podobné. Oba dva podporují připojení k široké škále datových zdrojů, nabízejí velkou míru přizpůsobitelnosti vizualizací a cílí na střední až větší firmy. Není tak divu, že je i Gartner umístil ve svém výběru blízko (obrázek 2.7).

Nevýhodou Power BI může být omezení podpory desktopového rozhraní pouze na operační systém Windows. To však Microsoft vynahrazuje nižší cenou a prostředím podobným dalším jeho produktům, jako je široce užívaný Microsoft Excel. Největší výhodou Tableau je nabídka pokročilých vizualizací, za které si uživatelé musí připlatit.

Třetí nástroj – Metabase – se od výše zmíněných liší výrazněji. Mezi největší rozdíly patří absence desktopového rozhraní, omezenější nabídka podporovaných zdrojů dat a dostupných možností vizualizace. Zmíněné rozdíly nutně nemusí znamenat zásadní nevýhodu pro menší firmy s omezeným rozpočtem hledající zejména základní funkcionalitu BI. Velkou výhodou je open-source licence v případě základní verze.

Ani zde nemůžeme označit konkrétního vítěze. Power BI se jeví jako jasná volba, když firma již využívá produkty firmy Microsoft. V případě, kdy pokročilé možnosti přizpůsobení hrají zásadnější roli než cena, lze doporučit Tableau. Naopak Metabase je dobrou volbou pro cenově dostupný vstup do světa BI nástrojů.

Naší volbou je stále Metabase, jednak si to *Firma* žádá a jednak lze Metabase díky open-source licenci bez dalších problémů integrovat do našeho SaaS modelu.

Vlastnost	Power BI	Tableau	Metabase
Nasazení v cloudu	Ano	Ano	Ano
Desktopové rozhraní	Ano	Ano	Ne
Operační systém[a]	Windows	Windows a MacOS	—
Podpora zdrojů dat	Rozsáhlá	Rozsáhlá	Zejména SQL databáze
Sdílení Reportů	Pokročilé sdílení; přesná kontrola oprávnění přístupu	Jednoduché sdílení na základě odkazů	Jednoduché sdílení na základě odkazů
Vizualizace	Pokročilé, vysoce přizpůsobitelné	Propracované a přizpůsobitelné	Základní vizualizace, oproti zbylým nepříliš přizpůsobitelné
Kolaborace	Real-time kolaborace	Sdílení nástěnek a reportů; možnosti komentářů	Sdílení nástěnek; méně zaměřeno na kolaboraci
Typ licence	Proprietární	Proprietární	Open-source (AGPL)
Cena	Omezená verze zdarma, u vyšších poplatek měsíčně za uživatele[b]	Placené různé formy	Základní verze zdarma

Pozn: [a] Týká se nástrojů s desktopovým rozhraním.
[b] Porovnávaná verze odpovídá typu *Premium*.

Tabulka 2.2 Porovnání důležitých vlastností nástrojů pro business intelligence

2.3 Závěr

V předchozích sekcích jsme si představili několik nástrojů, jejichž kombinací lze splnit požadavky na funkcionalitu definované v úvodním textu této kapitoly. Ovšem kombinace těchto nástrojů by vytvářela poměrně nesourodý celek, který by bylo složité automatizovat a integrovat do jednoho systému.

Další problém představují nástroje na integraci dat. Jsou primárně určeny pro použití experty na integraci či analýzu dat, proto je u nich kladen důraz na obecnost a širokou škálu využití, a tak v rámci vyvíjeného systému mnohonásobně převyšují požadovanou funkcionalitu. Tato nepotřebná funkcionalita zvyšuje komplexitu těchto nástrojů a tím vytváří bariéru pro použití ve spolupráci se zákazníkem – laikem.

V dalších kapitolách se tedy budeme soustředit na navržení a implementaci vlastního řešení, které minimalizuje nabízenou funkcionalitu na nezbytně nutnou úroveň, jež splňuje funkční požadavky, za účelem zachování jednoduchosti použití. Zároveň se pokusíme vyhovět dalším požadavkům *Firmy* na vlastní řešení.

V rámci vlastního řešení se můžeme inspirovat některými vlastnostmi analyzovaných nástrojů, zejména z oblasti integrace dat, kterou budeme do velké míry sami implementovat. Mezi tyto vlastnosti zajisté patří vizuální přístup k práci – grafický editor s „drag and drop“ komponentami – a možnost mapovat sloupce tabulky natažením čáry. Takovýto přístup se jeví jako menší bariéra pro přijetí uživatelem než je psaní kódu. Vizuální přístup ilustrují obrázky 2.2, 2.4, 2.5 a 2.6.

Kdybychom závěrem přeci jen měli vybrat nejlepší hledanou kombinaci z představených nástrojů, byly by jí nástroje *SSIS* a *Power BI* od firmy Microsoft, které díky jednotnému původu nabízí nejsnadnější integrovatelnost.

3 Specifikace

Tato kapitola se zabývá specifikací systému, která slouží jako podrobný popis požadované funkcionality a vlastností. Jak uvádí Tina Pringle [40], detailní specifikace požadavků je stále důležitá pro:

- jasnou definici cílů a funkcí systému,
- zajištění souladu s potřebami uživatelů a zadavatelů,
- usnadnění efektivního designu a implementace,
- posouzení a testování funkčnosti a kvality systému.

My si nejprve zadefinujeme klíčové pojmy týkající se vyvíjeného systému, abychom zajistili jednotné chápání této specifikace. Následně si shrneme požadavky na systém ve dvou kategoriích – funkční a nefunkční požadavky.

3.1 Pojmy

V této sekci definujeme klíčové pojmy používané napříč specifikací. To poslouží k přesné definici požadavků a usnadnění komunikace mezi všemi zúčastněnými stranami.

Nejprve definujeme, jaké uživatele bude mít námi vyvíjený systém. Tyto uživatele lze rozdělit na dva typy:

- **Zákazník** – je uživatel využívající systém pro mapování a vizualizaci dat.
- **Správce systému** – je zaměstnanec provozovatele systému, *Firmy*, který má na starost správu zákazníků a poskytování asistence s jejich problémy.

Dále si upřesníme některé pojmy, jež budeme v požadavcích na systém užívat:

- **Nový zákazník** – je zákazník, s nímž byla nově domluvena spolupráce a ještě pro něj není vytvořený uživatelský účet.
- **Stávající zákazník** – je zákazník, pro něhož je vytvořený účet a může používat všechny části systému, tzn. již pro něj je dostupná instance nástroje *Metabase*.
- **Bývalý zákazník** – je zákazník, s nímž byla spolupráce rozvázána.
- **Mapování dat** – je obecně proces vytváření mapování mezi elementy dvou datových modelů, např. za účelem pozdější transformace dat mezi danými datovými modely [41]. My se omezíme na mapování mezi relačními modely.
- **Generický datový model** – je datový model, který je dostatečně zobecněný tak, abychom na něj mohli převést, v našem případě, data od různých zákazníků.

- **Mapovací nástroj** – je jedna z komponent našeho systému, která umožňuje zákazníkům mapovat data na generický datový model systému.
- **Mapovací projekt** – označení pro mapování dat jednoho zákazníka na generický datový model systému.
- **Klastr** – pojmem klastr (anglicky *cluster*) je myšlen Kubernetes klastr[1].
- **Dashboard** – někdy též nástěnka, je typ grafického UI, který umožňuje jednoduše na jednom místě sledovat hlavní ukazatele výkonnosti[2] dané organizace ve strukturované podobě [42]. V našem případě se bude jednat o dashboardy z nástroje *Metabase*.

3.2 Požadavky

Abychom navrhli a implementovali v praxi použitelný systém, musíme nejprve podrobněji rozebrat, jaké požadavky jsou na systém kladeny.

3.2.1 Funkční požadavky

Jednotlivé funkční požadavky si představíme ve formě uživatelských příběhů (anglicky *user stories*). Jak uvádí Fred Health, jedná se o formu zaznamenávání funkčních požadavků na systém ve strukturované podobě krátkých příběhů většinou vypadajících následovně:

„Jako *typ uživatele* chci, nebo potřebuji *nějakou funkcionalitu*, aby *nějaký benefit*." [43] (překlad autora)

Tato forma zápisu nám pomůže konzistentně strukturovat informace o tom, jakou funkcionalitu potřebuje jaký typ uživatele a proč ji daný typ uživatele potřebuje, což zužitkujeme při tvorbě designu systému (kapitola 4).

Uživatelské příběhy rozdělíme do dvou částí podle typu uživatele.

Příběhy správce systému

P1 Jako správce systému potřebuji přidávat *nové zákazníky* (vytvořit jim uživatelský účet), aby mohli začít používat systém.

P2 Jako správce systému potřebuji, aby systém, v případě přidání *nového zákazníka*, zaslal zákazníkovi e-mail s odkazem na vytvoření hesla, aby jej zákazník mohl začít bezpečně používat.

P3 Jako správce systému potřebuji mazat *bývalé zákazníky* (zrušit jejich uživatelský účet), aby jen *stávající zákazníci* měli přístup do systému.

P4 Jako správce systému potřebuji nahlížet do *mapovacích projektů* zákazníků, abych mohl poskytnout asistenci s *mapováním* nebo mohl provést kontrolu *mapování*.

[1] Více informací v podsekci 5.2.1.

[2] Více na `https://en.wikipedia.org/wiki/Performance_indicator`.

P5 Jako správce systému potřebuji zajistit spuštění nové instance nástroje *Metabase* v *klastru*, aby jej zákazník mohl začít používat.

P6 Jako správce systému potřebuji, aby systém v případě spuštění nové instance nástroje *Metabase* automaticky provedl zpracování *mapování dat* a zpřístupnil vizualizaci dat daného zákazníka pomocí nové instance.

P7 Jako správce systému potřebuji, aby systém na základě autentizované identity autorizoval akce uživatelů, aby bylo zajištěno, že uživatelé mohou přistupovat jen ke svým datům.

P8 Jako správce systému v případě, kdy zákazník odejde od *Firmy*, potřebuji, aby systém ukončil instanci nástroje *Metabase* příslušící danému uživateli, aby jej již nemohl používat.

Příběhy správce jsou shrnuty formou use case diagramu na obrázku 3.1.

Zákaznické příběhy

P9 Jako zákazník potřebuji připojit svoji Microsoft SQL Server[3] databázi obsahující data mého ERP systému k vyvíjenému systému, abych mohl začít mapovat svoje data na *generický datový model*.

P10 Jako zákazník potřebuji používat uživatelsky přívětivý *nástroj pro mapování dat*, abych mohl snadno a rychle převést svoje data na generický datový model.

P11 Jako zákazník potřebuji mít možnost zadat přístupové údaje ke své databázi, aby systém mohl připravit mapování dat.

P12 Jako zákazník potřebuji mít přístup k nástroji *Metabase*, abych mohl vizualizovat a analyzovat svoje data pomocí různých reportů a dashboardů.

P13 Jako zákazník potřebuji mít možnost sdílet nebo exportovat svoje reporty a dashboardy, abych mohl prezentovat nebo sdílet svoje výsledky s ostatními.

Výše uvedené požadavky zjednodušeně ilustruje use case diagram 3.2.

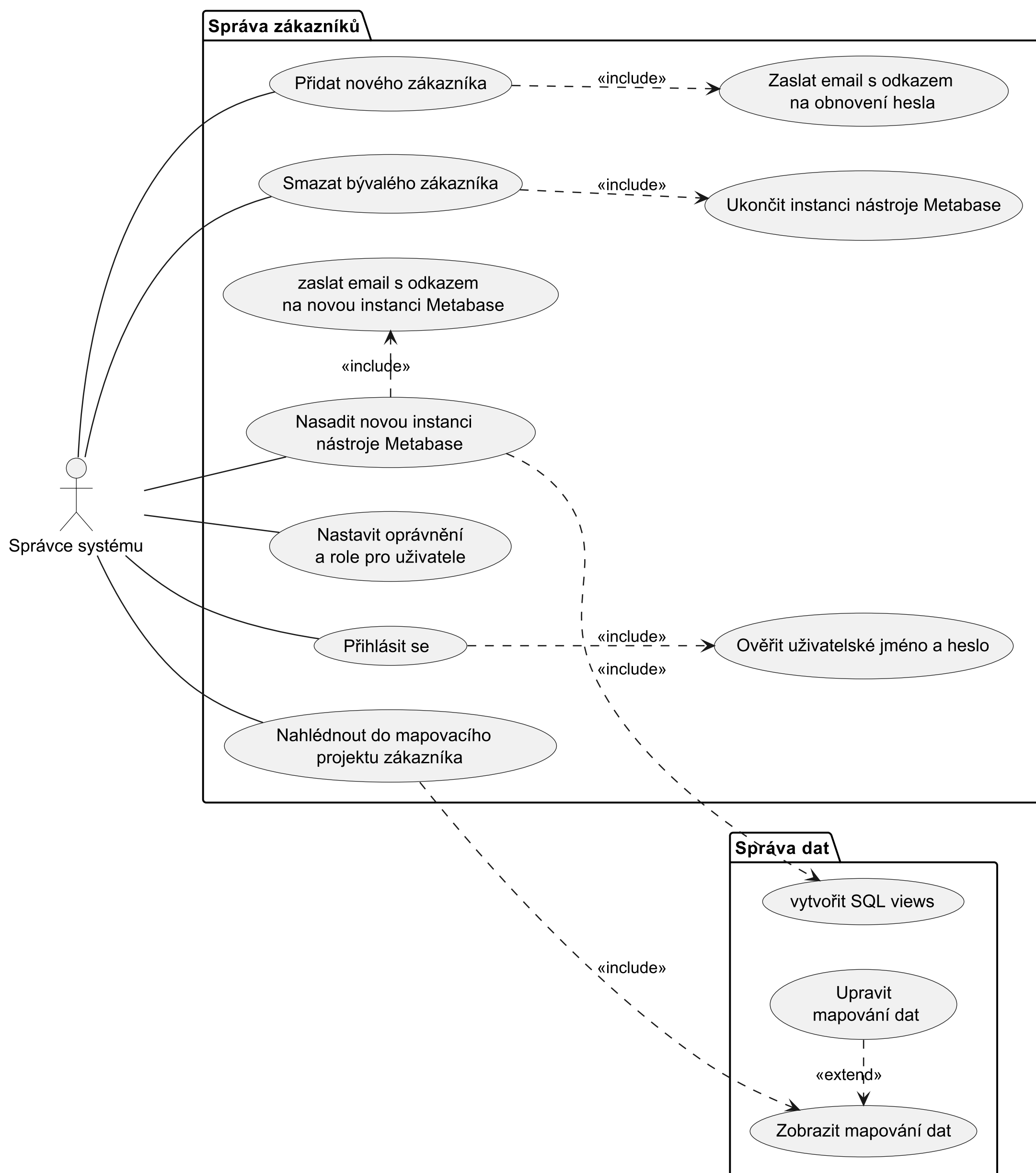
3.2.2 Nefunkční požadavky

Podle Sommervilla jsou nefunkční požadavky[4] omezení kladená na systém jako celek spíše než požadavky na jednotlivé funkce systému. Sommerville dále uvádí, že tyto požadavky jsou mnohdy zásadnější než mnohé funkční požadavky [44](strana 85-88), tudíž i my si patřičně rozebereme, jaké nefunkční požadavky musí námi vyvíjený systém splňovat.

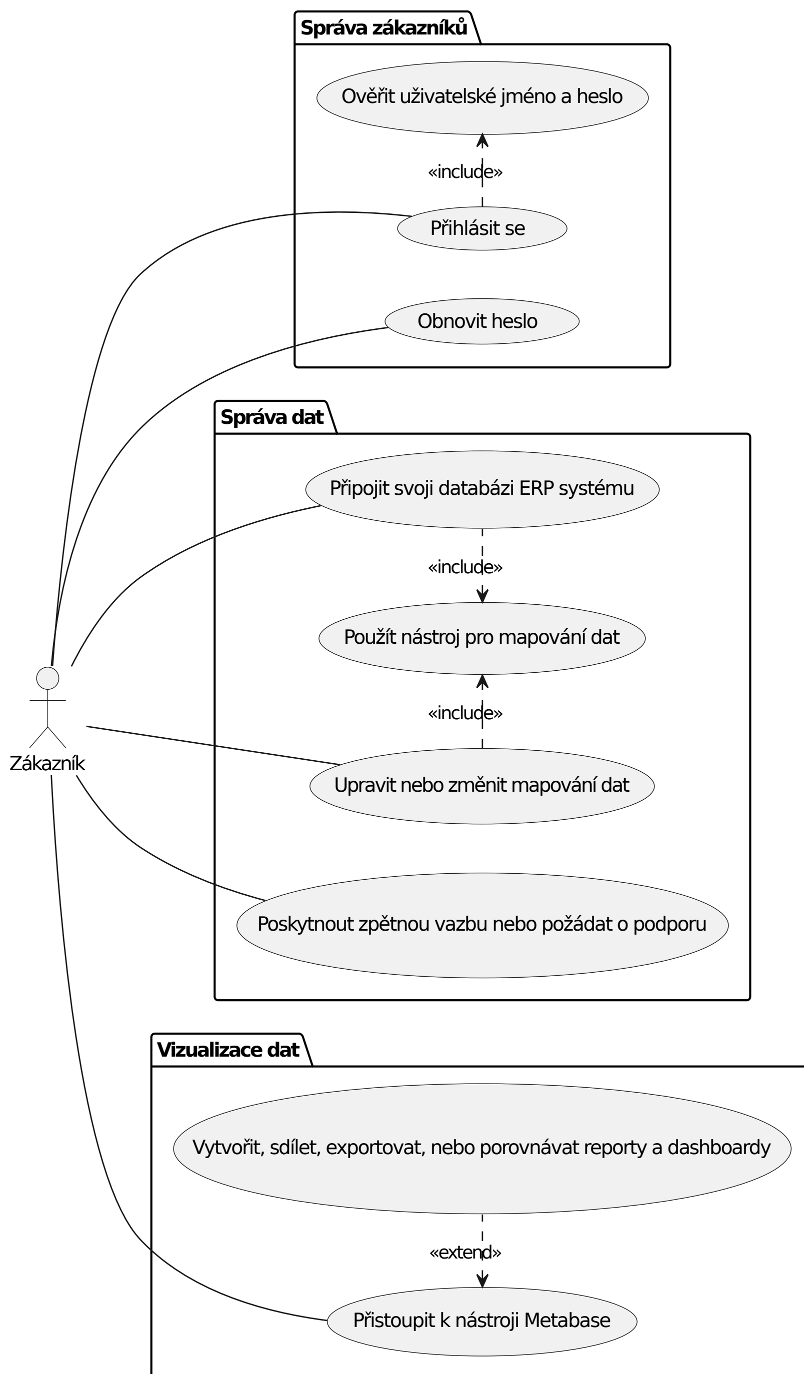
Typů nefunkčních požadavků existuje celá řada [45]. My se v následujících podsekcích zaměříme jen na několik hlavních a pro nás důležitých.

[3]Více na `https://www.microsoft.com/cs-cz/sql-server`.

[4]Z anglického *non-functional requirements*.



Obrázek 3.1 Diagram případů užití systému správcem



Obrázek 3.2 Diagram případů užití systému zákazníkem

Výkonnost

Jelikož se neočekává, že systém budou využívat stovky či tisíce uživatelů současně, požadavky na výkon jsou zaměřeny spíše na náročnější akce, které by měl systém vykonávat.

- Nasazení nové instance nástroje Metabase nesmí trvat déle než 90 sekund.
- Vytvoření nového projektu zákazníka nesmí trvat déle než 30 sekund.

Použitelnost

Tento aspekt systému je zejména z pohledu zákazníka klíčový. Konfigurace přístupových údajů k zákaznické databázi a mapování dat musí být co nejintuitivnější. 70 % zákazníků musí tento proces zvládnout samostatně.

Udržitelnost

- Modulární design – implementace systému musí mít jasně definované rozhraní mezi moduly pro snadnou výměnu a údržbu. Dále jednotlivé moduly musí být samostatně testovatelné pro zjednodušení identifikace a opravy chyb.
- Vývoj – při implementaci systému je nutné využívat nástroj pro verzování kódu (Git[5]) pro sledování změn a snadný návrat k předchozím verzím.
- Externí závislosti – systém by měl záviset pouze na knihovnách a technologiích, u kterých je předpoklad dlouhodobé podpory.
- Kvalita kódu – kód systému musí používat nejlepší praktiky použitých programovacích jazyků, musí využívat principy enkapsulace pro oddělení funkcí a usnadnění testování jednotlivých komponent a musí být dokumentován tak, aby bylo snadné jej pochopit a modifikovat.

Bezpečnost

Jelikož pracujeme se zákaznickými daty, je naprosto zásadní, aby tato data zůstala chráněna.

- Všichni uživatelé musí být autentizováni.
- Přístupové údaje daného zákazníka si může zobrazit a modifikovat pouze daný zákazník.
- Zákazníci si mohou zobrazit pouze vlastní *mapování dat* a užívat pouze pro ně vytvořenou instanci nástroje Metabase.
- Systém bude vyžadovat použití silných hesel[6] a umožní použití vícefázového ověření[7].

---

[5] Více informací na `https://git-scm.com/`.

[6] Podobně jako popisuje `https://support.microsoft.com/en-us/windows/create-and-use-strong-passwords-c5cebb49-8c53-4f5e-2bc4-fe357ca048eb`.

[7] Více informací na `https://learn.microsoft.com/en-us/aspnet/core/security/authentication/mfa`.

Interoperabilita

- Systém musí pro export a import *mapování dat* využívat formát JSON[8], zároveň systém musí zajistit validaci importovaných dat pro zajištění integrity *mapování.*
- Systém musí podporovat databázový systém *Microsoft SQL Server*, jelikož jej využívá většina zákazníků *Firmy.* Dále musí být připraven na rozšíření podpory pro další databázové systémy.

Škálovatelnost

Jelikož možnosti vertikálního škálování jsou omezené, požadujeme, aby bylo možné provoz instancí nástroje Metabase škálovat horizontálně přidáváním dalších serverů.

Portabilita

Systém musí být nasaditelný jako Docker kontejner[9] pro snadnou instalaci. Systém tedy musí podporovat jakoukoliv platformu, která umožňuje běh Docker kontejnerů.

Pro snadnou orchestraci a správu více kontejnerů při vývoji musí systém podporovat Docker Compose[10], což je využíváno ve vývojových prostředích (zkráceně IDEs) pro spuštění jedním klikem [46].

Testovatelnost

Systém musí být vybaven sadou testovacích dat, která pokrývá klíčovou funkcionalitu. Testovací data by měla odpovídat reálnému použití systému. Systém musí umožňovat běh v testovacím prostředí za účelem integračního testování.

3.3 Koncept

Díky výše uvedené specifikaci si již můžeme vytvořit představu o základních rysech systému. Celý systém se bude skládat ze 3 částí. Hlavní částí je webová aplikace sloužící pro správu zákazníků, *mapování dat* a nasazování instancí nástroje Metabase. Pro jednoduchost si ji pojmenujeme *správní aplikace.* Dvě vedlejší externí části jsou poštovní server a Kubernetes klastr.

Správní aplikace vedlejší části využívá – v případě poštovního serveru k odesílání e-mailových zpráv a v případě *klastru* pro nasazení instancí nástroje Metabase. *Správní aplikace* bude mít frontedovou a backendovou část.

Zákazník od frontendu potřebuje zejména možnost nakonfigurovat přístup k jeho ERP databázi a vytvořit *mapování* na *generický datový model.* Správce systému potřebuje spravovat zákazníky, kontrolovat nasazení instancí nástroje Metabase a nahlížet na vytvořené *mapování.*

[8]Textový formát pro výměnu dat založený na objektovém zápisu z jazyka JavaScript. Více informací na `https://www.json.org/`.

[9]Více informací na `https://docs.docker.com/`.

[10]Více informací na `https://docs.docker.com/compose/`.

Backendová část všechnu funkcionalitu zajišťuje. Víme, že nepotřebujeme implementovat webové API a tak si můžeme zjednodušit implementaci využitím modelu server-side rendering[11], kdy jsou všechny požadavky, včetně generování HTML pro UI (tzn. generování frontendové části), vykonány serverem. Backend bude dle nefunkčních požadavků členěn na moduly, což už naznačovaly use case diagramy (obrázek 3.1 a 3.2)

[11]Někdy překládáno jako *skriptování na straně serveru* (zkráceně SSR).

4 Design systému

V této kapitole si popíšeme proces navrhování vyvíjeného systému. Budeme zkoumat klíčová rozhodnutí, která ovlivnila výslednou architekturu, a vysvětíme, jak design reflektuje funkční a nefunkční požadavky stanovené pro tento projekt.

Architektonický přehled povede k podrobnějšímu zkoumání jednotlivých modulů a jejich propojení. Nakonec se zaměříme na návrh UI a UX.

4.1 Architektura systému

Abychom mohli systém snadno dle požadavků implementovat, je vhodné si nejdříve vytvořit návrh architektury. Pro jeho představení využijeme notaci C4 modelu [47] vytvořenou Simonem Brownem. Tato notace nám poskytuje čtyři úrovně abstrakce.

Nejvyšší úrovní je softwarový systém, ten se skládá z kontejnerů, což jsou nasaditelné jednotky jako aplikace nebo databázové systémy. Kontejnery jsou složené z komponent, které představují sdružení související funkcionality, která je dostupná přes dobře definované rozhraní. Pro poslední úroveň – diagramy tříd – C4 model notaci nedefinuje, a tak se většinou využívá jazyk UML.

Jelikož je vhodné při implementaci udržet stejné názvosloví jako v návrhu architektury, budeme při jejím popisu a v diagramech, které ji ilustrují, užívat anglické názvy.

Nejprve si představíme **kontext vyvíjeného systému**, ilustrovaný diagramem 4.1. Diagram ukazuje, jak vyvíjený systém využívají jednotlivé typy uživatelů a jaké má externí závislosti. Zákazníci používají mapovací nástroj a přistupují ke své instanci nástroje Metabase. Také můžeme vidět, že systém ke svému fungování potřebuje poštovní server, přes který posílá uživatelům notifikace, a připojení ke Kubernetes klastru (na diagramu pojmenovaném *Metabase Deployment System*) využívanému pro nasazení instancí nástroje Metabase.

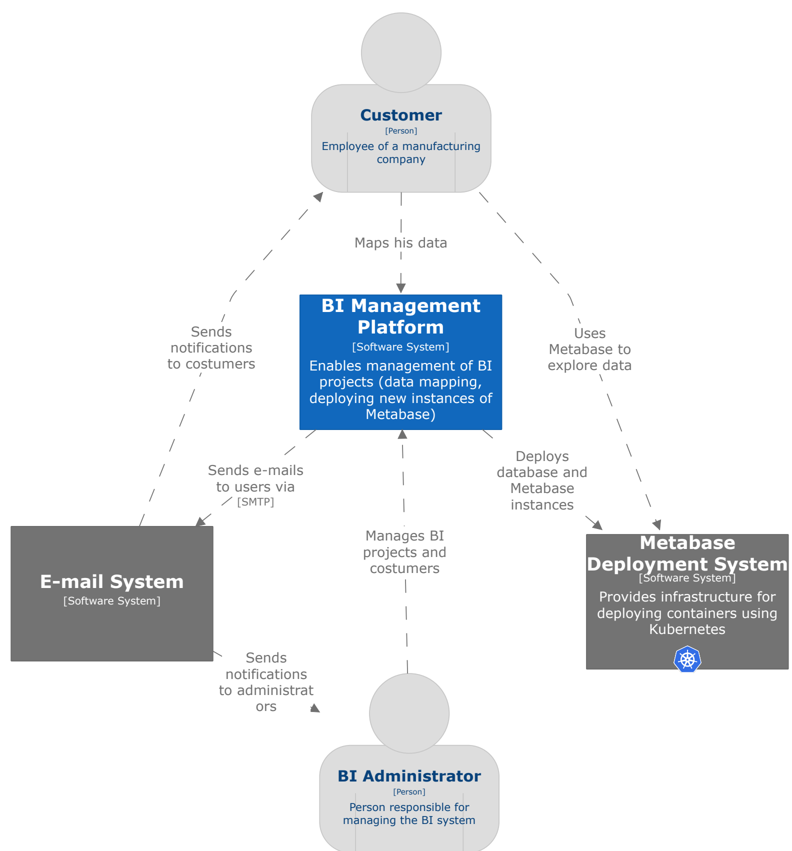
Pro každého zákazníka je nasazena jedna instance nástroje Metabase pracující nad jeho daty. Instance je pro zákazníka nasazena v moment, kdy dokončí *mapování*, případně až po schválení správcem systému. Jak je na diagramu vidět, správce systému platformu využívá pro správu zákazníků a BI projektů (souhrnně *mapování dat* a příslušných instancí nástroje Metabase).

Dále si představíme **kontejnerový diagram systému**, tedy z jakých kontejnerů se systém skládá (obrázek 4.2). Můžeme vidět, že se skládá ze 3 kontejnerů: *Management HTML*, *BI Management App*[1]. a *BI Management Database*.

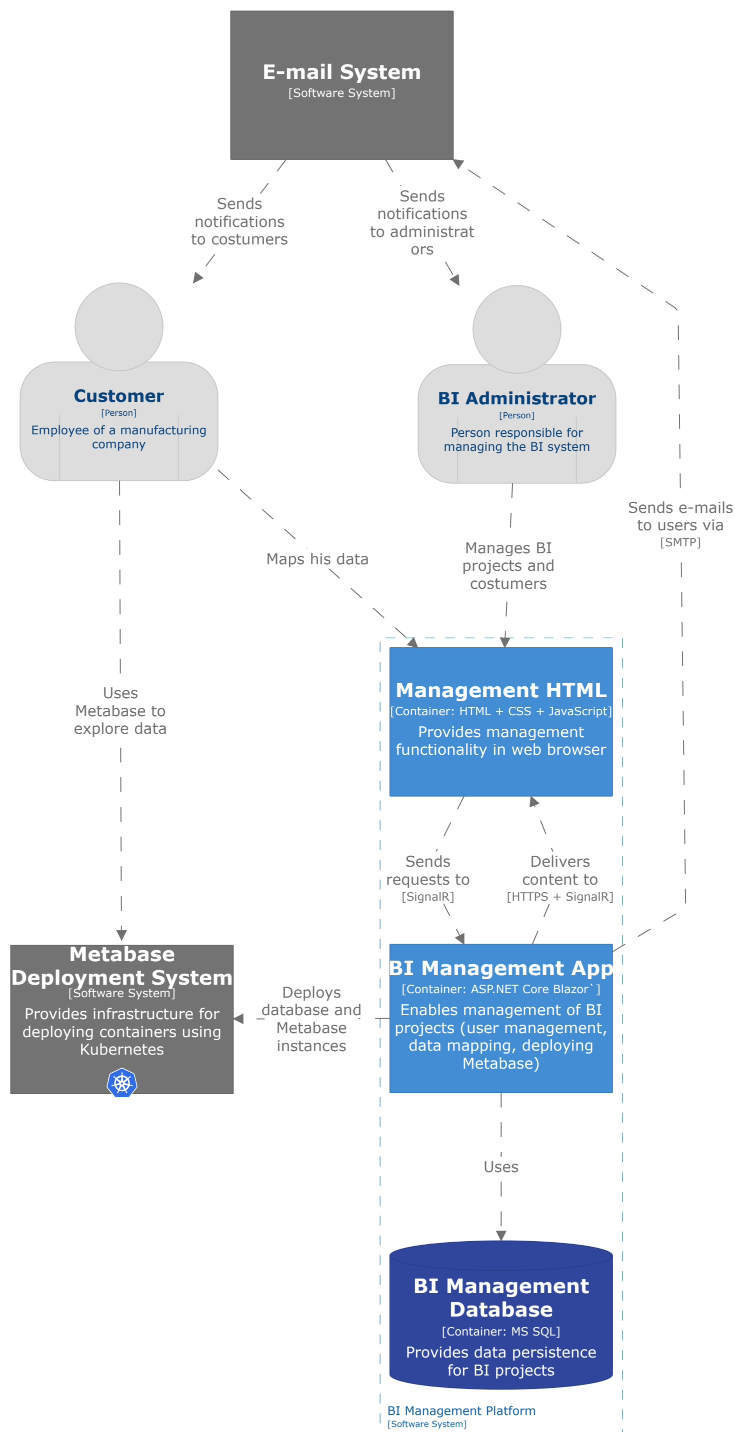
Uživatelé interagují s frontendovou částí *Management HTML*, kterou generuje a doručuje uživatelům kontejner *BI Management App* Ten představuje backendovou část. Frontend při běhu zasílá backendu požadavky, ten je, dle zmíněného modelu server-side rendering (sekce 3.3), zpracuje a aktualizuje frontend.

Backend vykonává veškerou business logiku a také zajišťuje integraci s externími systémy. Pro perzistenci všech dat backend využívá kontejner *BI Management Database*.

[1]Kontejner *BI Management App* jsme si v konceptu (sekce 3.3) představili jako *správní aplikaci* a *Management HTML* jako jeho frontendovou část.



Obrázek 4.1 C4 diagram zachycující kontext systému



Obrázek 4.2 C4 kontejnerový diagram systému

Dle požadavku na modulární design (podpodsekce 3.2.2) je backend členěn na moduly, což ilustruje obrázek 4.3. Vidíme **rozpad kontejneru *BI Management App* na komponenty**: *Blazor Server*[2] [48], *User Management Module*, *Metabase Deployment Module*, *Notification Module*, *Data Integration Module* a k němu příslušící *Mapper Library*.

Původně se pro propojení modulů v architektuře projektu zvažovala událostmi řízená architektura založená na asynchronním zasílání zpráv. Nabízela by nižší provázanost modulů a usnadnila by tak jejich rozšiřitelnost. Nicméně z důvodu náročnosti implementace byla tato myšlenka opuštěna.

V současné verzi architektury moduly navzájem komunikují synchronně prostřednictvím svých veřejných rozhraní, což vede k jejich běhové závislosti[3].

Vstupní branou celé aplikace je *Blazor server*. Generuje frontend a přijímá od něj požadavky, které dále předává jednotlivým modulům. Mimo celistvosti aplikace zajišťuje také její bezpečnost – autentizaci uživatelů. Ověřenou identitu pak využívají další komponenty pro autorizaci požadavků.

User Management Module využije zejména správce systému, jelikož slouží pro správu zákazníků – jejich přidávání, odstraňování atd. Pro správné fungování potřebuje *Notification Module*, jelikož zákazníkům, pro něž byl nově vytvořen uživatelský účet, zasílá notifikaci s odkazem na vytvoření hesla (požadavek P2).

Notification Module, jak už název napovídá, slouží pro notifikaci uživatelů. Ty jsou řešeny formou e-mailových zpráv, proto se *Notification Module* potřebuje připojovat k poštovnímu serveru (*E-mail System*).

Data Integration Module poskytuje funkcionalitu spojenou s *mapováním dat*, což obnáší získávání dat o databázi zákazníka jako jsou přístupové údaje a informace o jejím schématu, dále integraci a poskytování UI mapovacího nástroje (*Mapper Library*).

Posledním komponentou je *Metabase Deployment Module*, který zajišťuje nasazení instancí nástroje Metabase využívaných zákazníky. Pro nasazení nové instance potřebuje informace o *mapování* a databázi zákazníka z *Data Integration Module*.

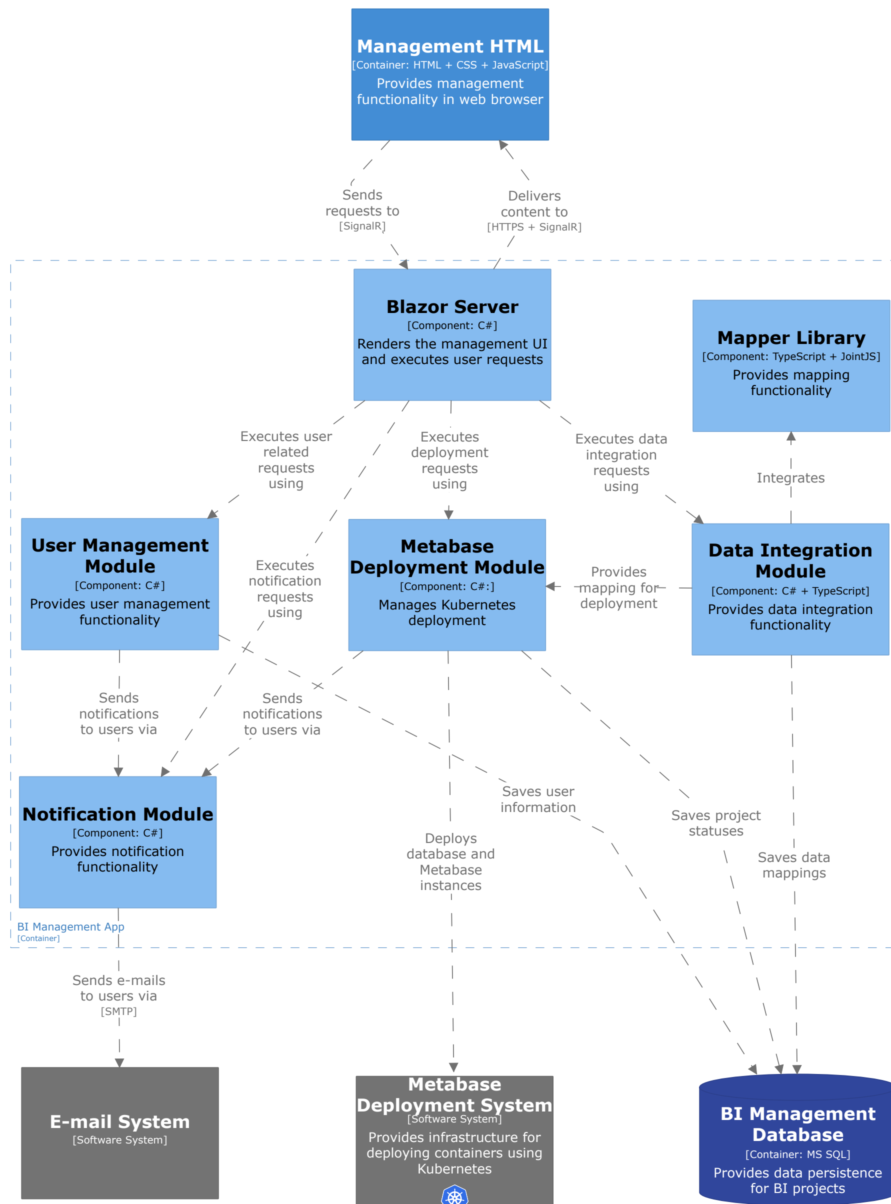
Mapování jednotlivých funkčních požadavků správce systému z podsekce 3.2.1 na moduly aplikace ukazuje tabulka 4.1. Tabulka 4.2 ukazuje obdobné mapování pro funkční požadavky zákazníka.

V levém sloupci obou tabulek se nachází označení požadavků. V dalších sloupcích velké písmeno *X* značí průnik žádané funkcionality a funkcionality modulu, který daný sloupec reprezentuje. Funkcionalita žádaná některými požadavky se týká více modulů, tudíž na jednom řádku můžeme vidět i více než jedno označení průniku (například v případě P2).

Také můžeme vidět, že požadavky zákazníka jsou koncentrované zejména v modulu *Data Integration*, kdežto požadavky správce systému jsou více rozprostřeny mezi jednotlivé moduly. Zákazníkův cíl je totiž právě jen začít používat nástroj Metabase. Správce systému se musí jednak starat o správu zákazníků a jednak jim poskytovat asistenci s mapováním.

[2]Pojmenované podle UI framework umožňující full stack vývoj (frontend i backend vývoj v jednom prostředí).

[3]Komunikující moduly musí fungovat ve stejnou chvíli. Není tedy beze změn možné moduly rozdělit na samostatně nasaditelné jednotky.



Obrázek 4.3 C4 diagram komponent webové platformy

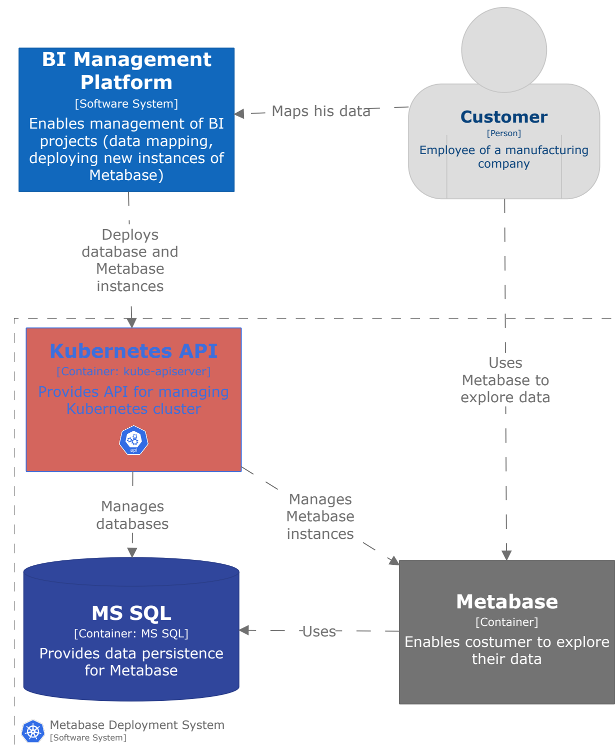
Požadavek	Blazor Server	User Manag.	Metab. Deploy	Data Integ.	Notification
P1: Přidání zákazníka		X			
P2: Link pro heslo na mail		X			X
P3: Odebírání zákazníků		X			
P4: Náhled *mapování*				X	
P5: Nasazení Metabase			X	X	
P6: Zpracování *Mapování*			X	X	
P7: Autorizace	X				
P8: Ukončení instance Metabase		X	X		

Tabulka 4.1 Mapování funkčních požadavků správce systému na moduly systému

Požadavek	Blazor Server	User Manag.	Metab. Deploy	Data Integ.	Notif.	Metabase[a]
P9: Připojení databáze				X		
P10: Mapovací nástroj				X		
P11: Přístupové údaje				X		
P12: Přístup k Metabase			X			
P13: Sdílení Metabase						X

Pozn: [a] Sice se nejedná přímo o modul vyvíjeného systému, přesně splňuje jeden z funkčních požadavků.

Tabulka 4.2 Mapování funkčních požadavků zákazníka na moduly systému



Obrázek 4.4 C4 kontejnerový diagram *klastru*

Nyní se vraťme o úroveň výše a podívejme se na **kontejnerový diagram *Metabase Deployment System*** – obrázek 4.4. Ten reprezentuje Kubernetes klastr, ve kterém jsou nasazeny instance nástroje Metabase. Můžeme vidět, že zákazník interaguje s nasazenou instancí nástroje a že jeho nasazení zajistí *BI Management Platform* komunikací s *Kubernetes API*.

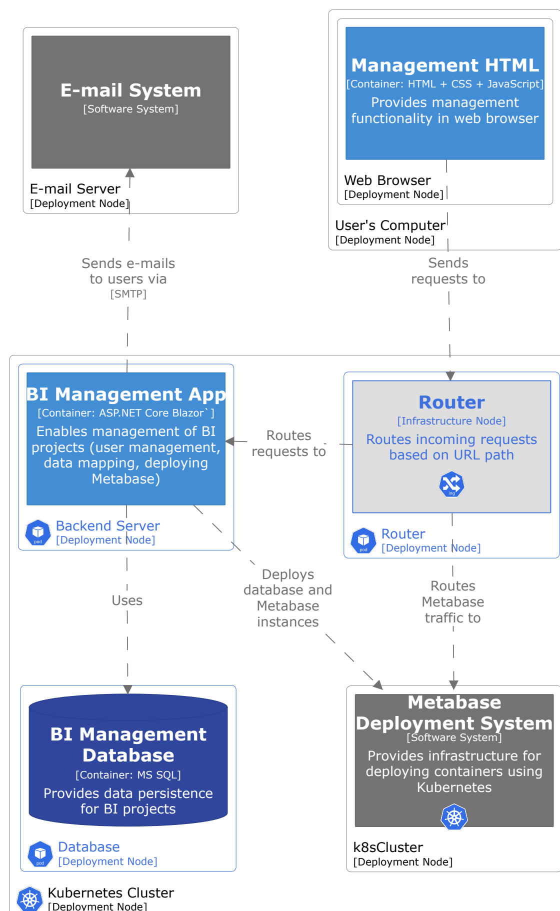
Z hlediska interní struktury systému *Metabase Deployment* se v diagramu nachází ještě kontejner databáze, který bude nasazen společně s každou instancí nástroje Metabase. Metabase databázi potřebuje pro ukládání aplikačních a uživatelských dat.

Z pohledu architektury je také důležitý plán nasazení systému, což ilustruje obrázek 4.5. Můžeme vidět, že celý systém je nasazen v Kubernetes klastru a že diagram není nijak složitý, přesto zachycuje řešení jednoho z klíčových problémů pro nasazování instancí nástroje Metabase – směrování požadavků na jednotlivé instance.

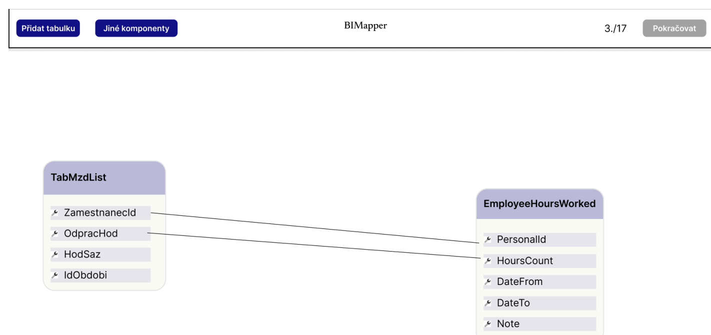
Infrastrukturní uzel[4] *Router* směruje na základě URL cesty požadavky do Kubernetes podů[5]. Mezi pody patří instance nástroje Metabase a backend aplikace. My tedy můžeme nastavit v konfiguraci *Routeru* unikátní cesty pro jednotlivé pody, což umožní uživatelům přístup jak k *BI Management App*, tak k instancím nástroje Metabase na jedné doméně druhého řádu.

[4]Jedná se o podpůrný prvek nasazení systému (anglicky *infrastructure node*), prvek, který přímo nevyvíjíme, ale například jen nakonfigurujeme.

[5]Pod je nejmenší a základní jednotkou nasazení v Kubernetes. Více informací v oficiální dokumentaci [49].



Obrázek 4.5 C4 diagram nasazení systému



Obrázek 4.6 Návrh prostředí nástroje Mapper

4.2 UI/UX design

Nyní jsme získali bližší představu o architektuře vyvíjeného systému a můžeme se přesunout k designu UI/UX.

4.2.1 Mapovací nástroj

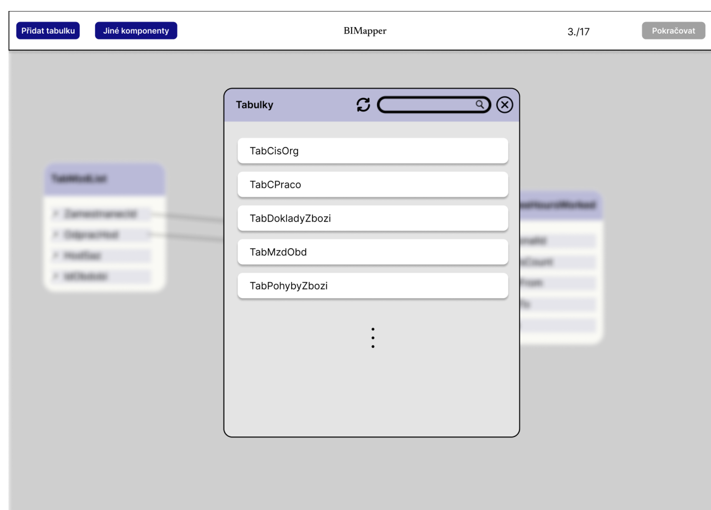
Již v zadání této práce jsme si vytyčili za cíl navrhnout a implementovat uživatelsky přívětivý nástroj pro *mapování dat*. Tento nástroj budou využívat zejména zákazníci, a tak je nutné řádně promyslet jeho design a fungování.

Návrh jeho prostředí ilustruje obrázek 4.6. Ve střední části návrhu můžeme vidět reprezentaci dvou tabulek, které mezi sebou mají natažené úsečky. Tyto úsečky značí ono mapování, tzn. sloupec zdrojové tabulky *TabMzdList ZamestnanecId* je mapován na sloupec *PersonalId* cílové tabulky *EmployeeHoursWorked* atd. Zdrojovou tabulkou myslíme tabulku z ERP databáze zákazníka a cílovou tabulkou entitu generického datového modelu.

Na horní liště se nacházejí 3 tlačítka: *Přidat tabulku*, *Jiné komponenty* a *Pokračovat*. Na rozdíl od prvních dvou je tlačítko *Pokračovat* šedé, což značí, že ještě nelze pokračovat na mapování další cílové tabulky. Důvodem je, že není dokončené mapování *EmployeeHoursWorked* – zatím nebyly namapovány všechny povinné sloupce tabulky.

Tlačítko *Přidat tabulku* slouží pro otevření menu pro výběr první, či další zdrojové tabulky k přidání. Menu ilustruje obrázek 4.7. Většinu menu zabírá seznam dostupných tabulek k přidání. V horní části se nachází tlačítko s ikonou šipek jdoucích dokola, které slouží pro znovu načtení seznamu. Napravo od tlačítka pro znovu načtení se nachází vyhledávací pole, které umožňuje prohledávání seznamu tabulek. Poslední tlačítko s ikonou kříže je určeno k zavření tohoto menu, a to bez přidání tabulky.

Kliknutím na nějakou z tabulek ze seznamu se otevře menu pro volbu výrazu pro spojení tabulek, jako je tomu v SQL. Menu ilustruje obrázek 4.8. Můžeme vidět, že výraz podmínky spojení zatím lze zvolit ve formátu rovnosti, nerovnosti, případně vztahu menší, nebo větší dvojice sloupců, kde každý pochází z jiné



Obrázek 4.7 Výběr další tabulky v nástroji Mapper

Obrázek 4.8 Definice výrazu pro připojení další tabulky v nástroji Mapper

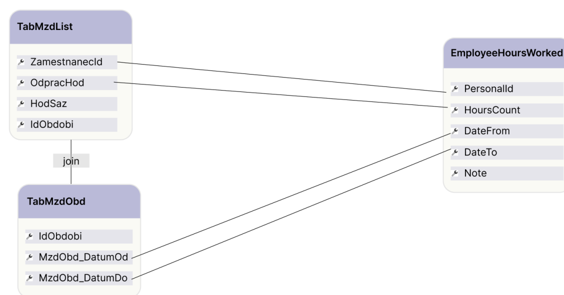
tabulky. Dvojice sloupců musí mít porovnatelný typ.

Po zadání validního výrazu se zobrazí reprezentace žádané tabulky. Stav po přidání druhé tabulky a namapování jejich sloupců ilustruje obrázek 4.9. Úsečka s popiskem *Join* mezi zdrojovými tabulkami *TabMzdList* a *TabMzdObd* reprezentuje jejich spojení. Dvojitým kliknutím na tuto úsečku lze opětovně otevřít menu pro definici příslušného výrazu spojení (obrázek 4.8). Povšimněme si také aktivace tlačítka *pokračovat*, kterou představuje aktuální zelené podbarvení.

Použitím tlačítka z navigační lišty *Jiné komponenty* se otevře menu pro přidání jiné komponenty do diagramu (obrázek 4.10). Mezi jiné komponenty patří *Výchozí hodnota*, *Filtr* a *Vlastní SQL skript*.

Výchozí hodnota slouží pro vyplnění nějakého sloupce hodnotou, která nepochází ze zdrojových tabulek, ale je uměle definována. Obrázek 4.11 ukazuje její použití při mapování, kdy je řetězec *Příklad mapování* použit jako hodnota sloupce *Note*.

Komponenta *Filtr* reprezentuje *WHERE* klauzuli z SQL, slouží tedy pro výběr podmnožiny dostupných dat splňující určitou podmínku. Tato podmínka se



Obrázek 4.9 Mapování po přidání druhé zdrojové tabulky

Obrázek 4.10 Přidání další komponenty do mapování

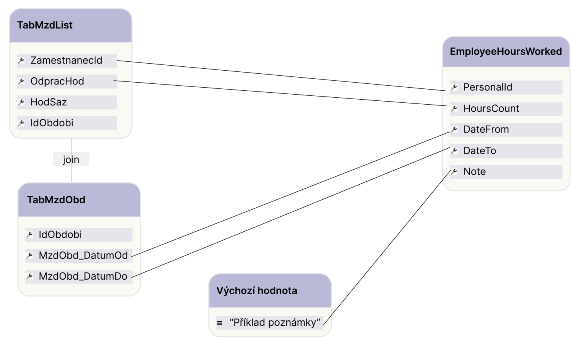
definuje v menu komponenty *Filtr*.

Na rozdíl od předchozích dvou komponenta *Vlastní SQL skript* poskytuje mnohem více volnosti využití. Jelikož poskytuje možnost zadání skriptu v jazyce SQL a umožňuje tak využít jeho plnou sílu. Tato komponenta slouží pro pokročilejší uživatele, kteří potřebují pro své mapování využít operace nepodporované vyvíjeným vizuálním editorem.

4.2.2 Správní aplikace

Další větší částí systému je webová *správní aplikace*. Ta musí umožnit vykonání všech funkčních požadavků a integrovat výše představený mapovací nástroj. Funkční požadavky jsme si rozdělili mezi několik modulů, a tak je vhodné přenést toto rozdělení také do návrhu UI. Návrh si představíme nejprve z pohledu *správce systému* a poté z pohledu *zákazníka*.

Na obrázku 4.12 je znázorněna stránka pro správu zákazníků, která přísluší *správci aplikace*. V levé části vidíme navigační menu, které krom odkazu na domovskou obrazovku a správy přihlášení (spodní část) nabízí podmenu jednotlivých



Obrázek 4.11 Mapování s výchozí hodnotou

ManagamentApp

Home
Data Integration
Deployment
Users
Customers
Admins

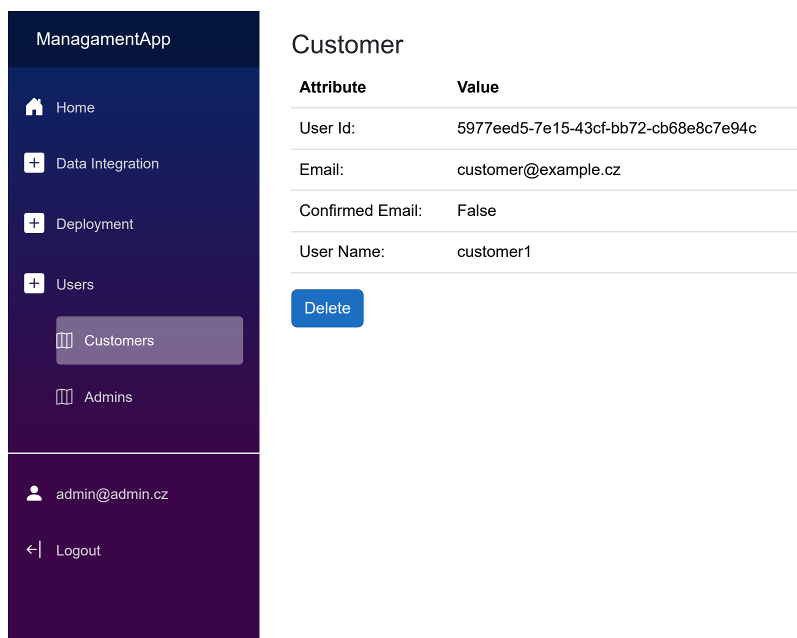
admin@admin.cz
Logout

Customers

Add Customer

ID	User name	Full name
5977eed5-7e15-43cf-bb72-cb68e8c7e94c	customer@example.cz	customer1
ca0dbd5e-eca5-4419-bbd1-f23ae700c142	customer2@example.cz	customer 2

Obrázek 4.12 Přehled zákazníků ve správní aplikaci.



Obrázek 4.13 Detail zákazníka ve správní aplikaci a možnost smazání jeho účtu.

modulů. Moduly s UI jsou pouze tři, jelikož modul *Notifications* vlastní UI nepotřebuje. V pravé, hlavní, části již vidíme samotný přehled zákazníků, přičemž jednotlivé identifikátory zákazníků lze rozkliknout.

Kliknutím na identifikátor se dostaneme na detail zákazníka, který ilustruje obrázek 4.13. Vidíme vybrané detaily zákazníka a tlačítko *Delete*, po jehož stisknutí a potvrzení záměru je daný zákazník smazán společně s pro něj nasazenou instancí nástroje Metabase (v případě, že už pro něj instance nasazena byla).

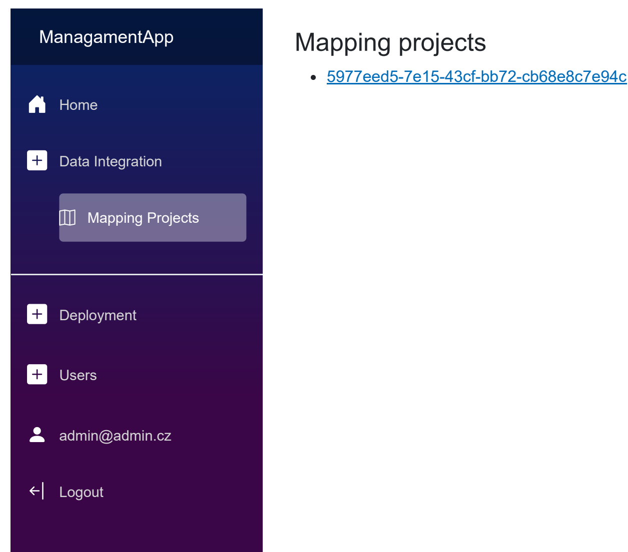
V podmenu *Data Integration* nalezneme stránku *přehled mapovacích projektů*, který je znázorněn na obrázku 4.14. V seznamu vidíme identifikátory jednotlivých zákazníků. Kliknutím na některý z identifikátorů, otevřeme mapovací nástroj s daty příslušícími danému zákazníkovi. Integraci tohoto nástroje ilustruje obrázek 4.17.

Nyní se přesuňme do části systému dostupné zákazníkům. Po přidání *nového zákazníka správcem systému* (tlačítko *Add Customer* na obrázku 4.12) je zákazníkovi zaslán mail s odkazem na vytvoření nového hesla. Rozkliknutím zaslaného odkazu se zákazník dostane na stránku pro zadání nového hesla znázorněnou obrázkem 4.15.

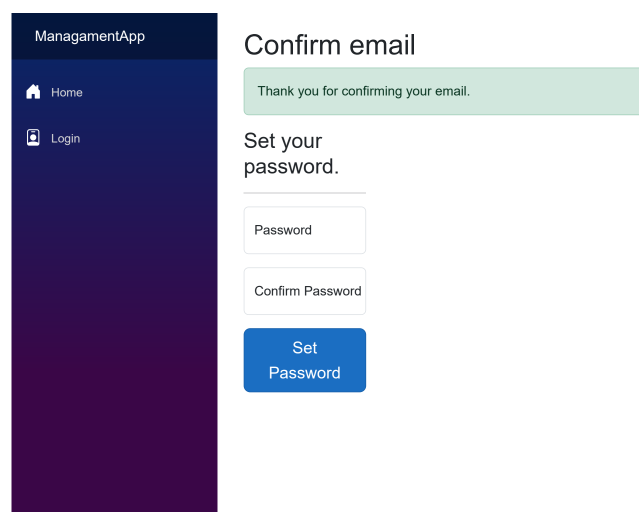
Na obrázku můžeme vidět formulář se vstupními poli pro zadání hesla a jeho opětovné zadání pro potvrzení shodnosti a tlačítko *Set Password* pro odeslání formuláře. Po zadání hesla a odeslání formuláře je zákazník (či obecně uživatel, jelikož je stránka využívaná také pro nové správce systému) přihlášen pod svým účtem.

Navigujeme-li se jako zákazník na stránku *Database connection configuration* přes podmenu *Data Integration*, je nám zobrazena stránka pro zadání přístupových údajů k zákaznické databázi, což ilustruje obrázek 4.16. Na obrázku můžeme vidět formulář pro zadání potřebných údajů a tlačítko *Save* pro odeslání formuláře. Po úspěšném odeslání formuláře a otestování připojení k databázi pomocí zadaných údajů, se zobrazí tlačítko *Create model* a hláška *Connection saved succesfully*.

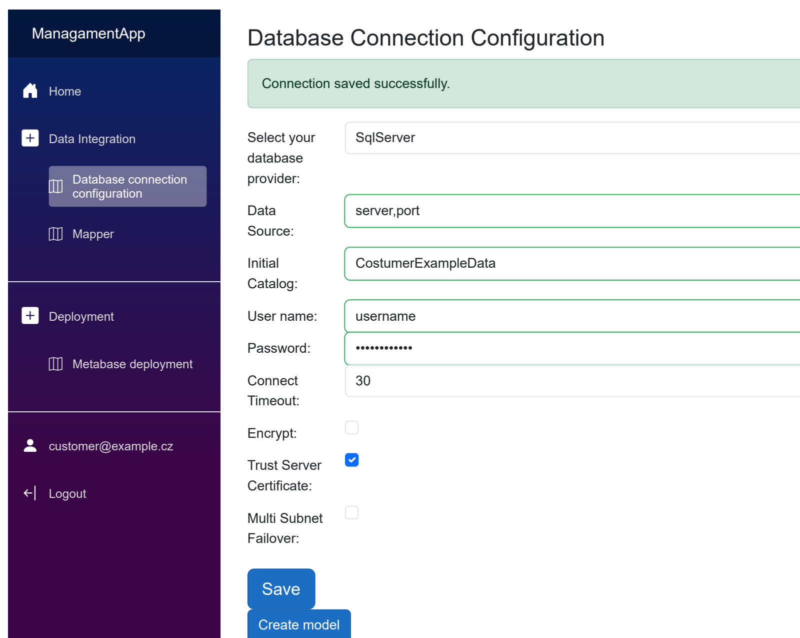
Po stisku tlačítka *Create model* oskenuje databázi a vytvoří si model jejího schématu. Poté je zobrazena hláška *Model created succesfully*.



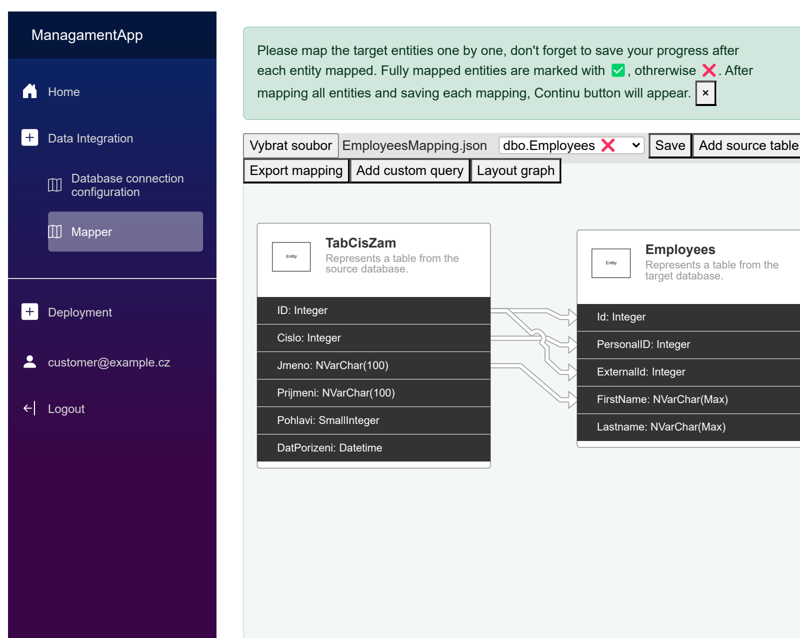
Obrázek 4.14 Přehled mapovacích projektů ve správní aplikaci.



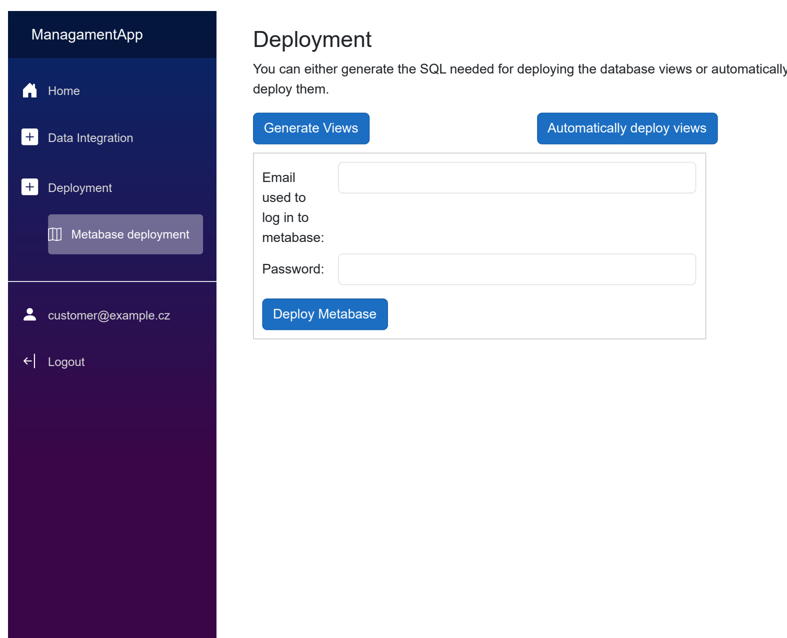
Obrázek 4.15 Potvrzení emailu a nastavení hesla u nového zákazníka ve správní aplikaci.



Obrázek 4.16 Konfigurace údajů potřebných pro připojení k zákaznické databázi.



Obrázek 4.17 Integrace mapovacího nástroje a jeho použití pro mapování zákaznických dat.



Obrázek 4.18 Stránka pro nasazení databázových pohledů a instance nástroje Metabase

Po vytvoření modelu můžeme pokračovat na stránku *Mapper*, která se nachází ve stejném podmenu *Data Integration*. Jedná se o integraci mapovacího nástroje představeného v podsekci 4.2.1 do správní aplikace. Stránku ilustruje obrázek 4.17.

Můžeme vidět, že na rozdíl od návrhu 4.6 se na liště nástrojů nenachází tlačítko *Pokračovat* a tlačítko *Jiné komponenty* ale pouze *Add custom query* pro přidání vlastního SQL dotazu. Další komponenty z menu *jiné komponenty* (vizte obrázek 4.10) zatím využívat nebudeme.

Na liště se navíc nachází tlačítka *Vybrat soubor*, které slouží pro vložení uloženého mapování, *Export mapping* pro export mapování do souboru, *Save* pro uložení mapování na server a *Layout graph* pro automatické rozložení mapování.

Po úplném namapování cílové tabulky a stisku tlačítka *Save*, je mapování uloženo na server. Ikona červeného křížku u názvu aktuální mapované entity v liště nástrojů se změní na zelenou fajfku. Kliknutím na název aktuální mapované komponenty v liště nástrojů se zobrazí nabídka pro cílové tabulky k mapování.

Po namapování všech cílových tabulek se zobrazí nad lištou nástrojů tlačítko *Deploy metabase*, to po stisku přesměruje uživatele na stránku pro nasazení databázových pohledů a instance nástroje Metabase. Návrh stránky je ilustrován obrázkem 4.18.

Na obrázku můžeme vidět tři tlačítka a formulář pro specifikaci přihlašovacích údajů k instanci nástroje Metabase. Tlačítko *Generate Views* slouží pro vygenerování a zobrazení SQL příkazů pro vytvoření potřebných databázových pohledů. Po stisku tlačítka *Automatically deploy views správní aplikace* potřebné pohledy nasadí v databázi zákazníka. Zákazník v případě potřeby může databázové pohledy vytvořit i manuálně použitím vygenerovaných příkazů.

Po vyplnění formuláře a stisku tlačítka *Deploy Metabase správní aplikace* vytvoří a nakonfiguruje instanci nástroje Metabase. Uživateli je zobrazena hláška pro vyčkání a po dokončení operace nasazení je zobrazen odkaz na připravenou instanci nástroje Metabase (v případě chyby je zobrazena chybová hláška).

5 Použité technologie

V předchozí kapitole jsme si představili návrh systému z architektonického pohledu i z pohledu UI a UX. Nyní se podíváme, jaké technologie pro realizaci navrženého systému budeme potřebovat. Nejprve si představíme technologie použité ve *správní aplikaci*, poté technologie určené pro nasazení a závěrem technologie pro testování.

5.1 *Správní aplikace*

Ze specifikace a představení architektury již víme, že cílíme na modulární architekturu a chceme využít model server-side rendering, kdy backend (serverová část) generuje frontend, přičemž od něj poté přijímá požadavky, které dále zpracovává.

5.1.1 Backend

Pro vývoj backendu byla vybrána platforma .NET, konkrétně jazyk C# [50]. C# je moderní objektově orientovaný jazyk, který je neustále rozvíjen původním tvůrcem, firmou Microsoft, a komunitou dobrovolníků.

Jazyk je obvykle překládaný do strojového mezikódu a běží na virtuálním stroji CLR[1], který zajišťuje JIT[2] kompilaci. Běh ve virtuálním prostředí zajišťuje přenositelnost kódu na jakoukoliv platformu podporující CLR [50].

Díky dlouhodobému rozvoji jazyk disponuje rozsáhlým ekosystémem jak standardních knihoven, tak rozšiřujících komunitních knihoven distribuovaných pomocí praktického správce balíčků NuGet [52].

C# byl vybrán právě pro jeho rozsáhlý ekosystém knihoven a přenositelnost. Navíc s ním má autor práce nejvíce zkušeností a je již široce užívaný ve *Firmě*.

Blazor

Blazor [53] je webový UI framework fungující nad ASP.NET core[3] aplikacemi. Blazor umožňuje vytvářet interaktivní (zejména webové) UI za použití jazyka C#, čímž umožňuje sdílení logiky napříč backendovou a frontendovou částí, tedy full stack vývoj.

Komponenta frameworku Blazor Server [48] navíc umožňuje generování UI dle zmíněného modelu server-side rendering.

Připravenost užití modelu server-side rendering a možnost tvorby UI v C# splňuje naše požadavky a dobře zapadá do kontextu znalostí autora a technologií používaných ve *Firmě*,

[1] Zkratka (z anglického *Common Language Runtime*) označuje virtuální běhové prostředí jazyků .NET [51].

[2] Zkratka z anglického *Just-in-Time* označuje formu překladu mezikódu na instrukce specifické systému, na kterém běží, za běhu programu.

[3] ASP.NET core je framework pro modulárních vývoj webových aplikací na platformě .NET [54].

ASP.NET Core SignalR

Blazor Server pro zajištění komunikace s jím vygenerovaným frontendem využívá knihovnu ASP.NET Core SignalR [48, 55, 56] (dále jen SignalR).

SignalR je moderní knihovna založená na protokolu WebSocket[4] určená pro komunikaci mezi serverem a klientem v reálném čase. Je široce využívaná pro implementaci chatů či jiných aplikací vyžadujících časté aktualizace ze strany serveru, jako je webová aplikace dle modelu server-side rendering.

Nutným požadavkem pro správné fungování SignalR spojení je kvalitní internetové spojení, což je tranzitivně také nevýhodou modelu server-side rendering. Na druhou stranu vykonání veškeré logiky na straně serveru omezuje nutnost poskytovat zdrojové kódy klientovi a umožňuje zachovávat stav aplikace klienta. Klient má tak možnost rozpracovanou práci přenášet mezi zařízeními, v případě chyby na jeho zařízení o rozpracovaný stav nepřijde.

Spojení pomocí SignalR je znázorněno na architektonickém diagramu 4.2 mezi kontejnery *BI Management App*, tedy správní aplikací, a *Management HTML*, tedy generovaným frontendem.

Perzistence

Abychom o klientská a aplikační data nepřišli, potřebujeme nějaký způsob jejich perzistence. Pro ukládání dat využijeme Microsoft SQL Server [7] (zkráceně MSSQL). MSSQL je léty ověřený databázový systém, který je již užíván *Firmou*. Jakožto produkt firmy Microsoft dobře funguje s dalšími jejími produkty, jako je například právě prostředí .NET.

Pro přístup k databázi z programovacího jazyku C# použijeme v prostředí .NET velmi rozšířenou knihovnu Entity Framework core [58] (zkráceně EF core). EF core slouží pro objektově relační mapování, tedy usnadňuje konverzi dat z relačních databází na objekty v C# (obecně v objektově orientovaných programovacích jazycích).

5.1.2 Frontend

V předchozí sekci jsme se dozvěděli, že Blazor Server bude zajišťovat generování frontendu. Nyní si pojďme představit z čeho frontend sestává a jaké technologie jsou použity pro jeho definici a fungování.

Generovaný frontend sestává z kombinace HTML[5] a JavaScript[6]. HTML v našem případě definuje základní strukturu webu. JavaScript zajišťuje přes SignalR klienta [61] komunikaci se serverem a zobrazení přijatých dat uživateli pomocí manipulace HTML DOM[7]

[4]Internetový protokol pro obousměrnou komunikaci založený na TCP [57](strana 1).

[5]Označuje jednoduchý značkovací jazyk (anglicky *Hypertext Markup Language*), který slouží pro tvorbu hypertextových dokumentů. Tyto dokumenty se obvykle používají na webu [59](strana 1).

[6]JavaScript (zkráceně JS) je interpretovaný skriptovací jazyk nejčastěji používaný pro tvorbu interaktivních webových stránek [60].

[7]Objektový model HTML dokumentu (z anglického *Document Object Model*) si vytváří prohlížeč při načítání webové HTML stránky. Prohlížeče pro jeho manipulaci nabízí API, které využívá JavaScript. [62]

Pro tvorbu frontendu nebudeme z většiny přímo psát HTML či JavaScript, ale použijeme silnější jazyk, který bude serverem následně přeložen do prostšího HTML.

Razor Pages

Razor Pages [63] jsou dokumenty či komponenty webové stránky napsané v syntaxi značkovacího jazyka Razor [64]. Razor umožňuje kombinovat C# a HTML kód, kde HTML definuje strukturu stránky či komponenty a C# umožňuje dynamické generování obsahu, řízení toku dat či implementaci business logiky.

Zmíněné Razor komponenty představují znovupoužitelné části uživatelského rozhraní, jako jsou tlačítka nebo formuláře. Mohou mít definované parametry, které mění jejich vzhled či chování. Komponenty se stávají stránkou, když je jim definovaná URL cesta pomocí atributu *page*.

Pomocí Razor komponent můžeme vytvářet modulární UI tak, že v jednotlivých modulech případné potřebné ovládací prvky implementujeme jako komponenty. Ty poté můžeme integrovat do správní aplikace.

JointJS

Jak můžeme vidět na obrázku 4.6, v mapovacím nástroji *Mapper* budeme potřebovat vytvářet diagramy znázorňující *mapování dat*. Abychom si ušetřili jejich kompletní implementaci, můžeme si pomoci knihovnou, která se na tvorbu diagramů specializuje. Vybrána byla knihovna JointJS [65].

JointJS je open-source JavaScript knihovna pro tvorbu a interaktivní práci s diagramy. Vývojářům nabízí nástroje pro snadnou implementaci vizuálních prvků do webových aplikací. Architektura knihovny umožňuje snadnou tvorbu vlastních tvarů dle specifických potřeb projektu, čehož při implementaci využijeme.

Důvod výběru knihovny JointJS je její široké rozšíření, bohatá funkcionalita a open-source licence. Knihovna je navíc stále udržovaná, což představuje velkou výhodu. V neposlední řadě nabízí sadu definic typů pro využití jazyka TypeScript [66].

TypeScript

TypeScript je programovací jazyk, který přidává statické typování do jazyka JavaScript (TypeScript je jeho nadmnožinou). Jeho výhodou oproti jazyku JavaScript je plná podpora objektově orientovaného programování se stylem syntaxe podobným C#, proto jej využijeme.

Také podobně jako v případě C# je třeba TypeScript kompilovat. Na rozdíl od C# je TypeScript kompilován, přesněji transpilován, do jazyka JavaScript. Při kompilaci je nejprve provedena statická typová kontrola, poté je TypeScript přeložen do syntaxe JavaScript, čímž se typová informace ztrácí. [67]

TypeScript dokáže se statickým typováním využívat JS knihovny, které externě definují zmíněné definice typů jako JointJS. Tyto definice jsou obvykle v souborech *.d.ts* [68].

npm

npm [69] je správce balíčků v ekosystému JavaScript běhového prostředí Node.js[8]. V projektech Node.js jsou v souborech *package.json* definovány vlastnosti aplikace, skripty pro spuštění a ovládání aplikace a závislosti na externích balíčcích [70].

Pomocí *npm* můžeme nainstalovat všechny závislosti projektu, jak přímo ve zdrojovém kódu použité balíčky (JS knihovny), tak i vývojové nástroje. Právě podle účelu použití se závislosti *npm* balíčku dělí na pouhé závislosti (anglicky *dependencies*), které jsou nutné pro použití vyvíjeného balíčku, v našem případě mapovacího nástroje, a vývojové závislosti (anglicky *devDependencies*) sloužící například pro testování, či transpilaci kódu do JS.

Mezi přímé závislosti mapovacího nástroje *Mapper* bude patřit zmíněný JointJS v podobě hlavního balíčku knihovny *@joint/core*[9] a přídavné komponenty *@joint/layout-directed-graph*[10] umožňující, jak název napovídá, automaticky rozložit DAG[11] diagram.

Mezi vývojové závislosti bude patřit *TypeScript* balíček[12] využívaný zejména pro transpilaci zdrojových kódů, JS testovací framework *Jasmine*[13] a balíček *@types/jasmine*[14] definující statické typování nad frameworkem *Jasmine*. Dále využijeme *webpack*[15] a jeho pluginy distribuované jako balíčky *ts-loader*[16] a *css-loader*[17]. Krom uvedeného budeme potřebovat ještě pomocné balíčky pro multiplatformní správu prostředí *rimraf*[18] (mazání souborů) a *cross-env*[19] (nastavování proměnných prostředí).

Webpack

Webpack [71] využijeme pro vytvoření tzv. *bundle* zdrojových kódů v jazyce JavaScript[20] (koncentrace potřebného JS kódu do jednoho souboru). Webpack také provádí řadu optimalizací, jako je kombinace použitých modulů do jednoho nebo vynechání nepoužitého kódu a znaků.

Webpack také využijeme přímo při vývoji mapovacího nástroje. Jeho komponenta webpack-dev-server [72] funguje jako jednoduchý webový server. Poskytuje základní funkcionalitu pro spuštění JavaScript, respektive TypeScript, aplikací.

Jednou z výhod webpack-dev-server je možnost využití opětovného načítání za provozu (známější anglicky jako *live reload*), kdy se stránka poskytovaná serverem

[8]Více informací na `https://nodejs.org/en`.

[9]Více na `https://www.npmjs.com/package/@joint/core`.

[10]Více na `https://www.npmjs.com/package/@joint/layout-directed-graph`.

[11]DAG je zkratka z anglického *Directed Acyclic Graph* a označuje orientovaný acyklický graf.

[12]Více na `https://www.npmjs.com/package/typescript`.

[13]Více na `https://www.npmjs.com/package/jasmine`.

[14]Více na `https://www.npmjs.com/package/@types/jasmine`.

[15]Více na `https://www.npmjs.com/package/webpack`.

[16]Plugin pro načítání TypeScript kódu do nástroje webpack. Více na `https://www.npmjs.com/package/ts-loader`.

[17]Umožňuje do bundle zahrnout CSS soubory (CSS slouží pro stylizaci stránek). Více na `https://www.npmjs.com/package/css-loader`.

[18]Více na `https://www.npmjs.com/package/rimraf`.

[19]Více na `https://www.npmjs.com/package/cross-env`.

[20]V našem případě jazyka TypeScript předchází vytvoření bundle transpilace do jazyka JavaScript.

při uložení jakékoliv změny ve zdrojových kódech znovu načte se změněným obsahem či chováním.

Microsoft JSInterop

Pro zajištění integrace vyvíjeného mapovacího nástroje v jazyce TypeScript do zbytku UI běžícího na Blazor serveru a využívajícího syntaxi Razor použijeme knihovnu Microsoft JSInterop [73] (dále jen JSInterop).

Razor komponenty umožňují za pomoci JSInterop, konkrétně objektu s rozhraním *IJSRuntime*[21] načíst, spustit a ovládat JS kód. Tímto způsobem lze funkcionalitu poskytovanou JS kódem integrovat do Razor komponenty, potažmo do Blazor aplikace.

IJSRuntime po načtení JS kódu, v našem případě *bundle* generovaný pomocí balíčku *webpack*, obvykle vrací objekt s rozhraním *IJSObjectReference*[22], který reprezentuje načtený modul. Přes rozhraní *IJSObjectReference* lze volat funkce z načteného modulu.

5.2 Nasazení

Výše jsme si představili technologie, které budeme potřebovat pro implementaci správní aplikace. Nyní se pojďme podívat, jaké technologie použijeme pro její nasazení.

Některé již naznačil diagram 4.5. Můžeme vidět, že celý systém bude nasazen v Kubernetes klastru a že infrastrukturní uzel *Router* je tvořen Ingress ovladačem[23] [74].

5.2.1 Kubernetes

Kubernetes [75] (dále jen K8S) je moderní nástroj pro správu a orchestraci kontejnerizovaných aplikací[24]. Usnadňuje jejich nasazení v K8S klastru, škálování a mnohé další.

K8S klastr sestává z K8S uzlů [77] (anglicky *Nodes*). Uzly klastru reprezentují buď fyzické, nebo virtuální stroje a jsou složeny z K8S infrastruktury[25] a podů [49]. Pod je nejmenší jednotka nasazení v K8S. Skládá se z jednoho či více kontejnerů vytvořených z *OCI Image* [76] (zjednodušeně šablona pro vytvoření kontejneru).

Abychom mohli vyvíjený systém nasadit pomocí K8S musí být všechny jeho části (správní aplikace, databáze a instance nástroje Metabase) nasaditelné jako K8S pod, potažmo kontejner.

---

[21] Rozhraní abstrahující JS běhové prostředí. Více na `https://learn.microsoft.com/en-us/dotnet/api/microsoft.jsinterop.ijsruntime`.

[22] Více na `https://learn.microsoft.com/en-us/dotnet/api/microsoft.jsinterop.ijsobjectreference`.

[23] Anglicky *Ingress Controller*.

[24] Kontejnerizovanou aplikací rozumějme aplikaci složenou z kontejnerů vytvořených na základě *OCI Image* [76].

[25] Více informací v oficiální dokumentaci [77].

Nginx Ingress ovladač

Ingress ovladač je objekt v K8S, který slouží jako styčný bod mezi vnitřní a vnější sítí. Umožňuje tak směrovat externí požadavky na pody běžící v K8S klastru. Také se používá pro vyvažování zátěže mezi jednotlivými pody a pro směrování požadavků na základě URL cesty.

V tomto projektu využijeme Nginx Ingress ovladač [78], což je jedna z nejrozšířenějších implementací Ingress ovladače. Hlavní úlohou Ingress ovladače v našem systému bude směrování požadavků na základě URL cesty. Toto užití je znázorněno na obrázku 4.5, kde infrastrukturní uzel *Router* směruje požadavky do K8S klastru pomocí Ingress ovladače.

KubernetesClient pro .NET

Pro správu K8S ze správní aplikace použijeme oficiální klientskou knihovnu pro K8S KubernetesClient [79]. Tato knihovna nám umožní objektově interagovat s K8S API, čímž získáme plnou kontrolu nad nasazením a provozem kontejnerů. Mezi funkce knihovny patří:

- Vytváření, čtení, aktualizace a mazání K8S objektů (např. podů nebo ingress konfigurací).
- Získávání informací o stavu K8S klastru.
- Zpracovávání událostí generovaných K8S.
- Provádění operací na K8S API serveru.

Díky KubernetesClient můžeme zautomatizovat úlohy správy K8S a zefektivnit tak vývoj a provoz aplikací.

5.3 Testování

Pro testování různých částí našeho systému využijeme dva testovací frameworky: pro testování mapovacího nástroje framework Jasmine [80] a pro zbytek správní aplikace běžící na platformě .NET NUnit [81].

NUnit je testovací framework pro .NET. Umožňuje nám psát jednotkové testy a integrační testy. Jedná se o robustní a snadno použitelný framework, který nám pomůže zajistit kvalitu našeho kódu. Navíc je již využíván *Firmou*, a tak s ním má i autor nejvíce zkušeností.

Jasmine je jednoduchý testovací framework pro JavaScript (použitelný také pro TypeScript). Je populární zvláště pro testování webových aplikací a frontendových komponent. Výhodou je, že umožňuje psát testy v téměř přirozeném jazyce, tudíž se snadno čtou a udržují (viz ukázka testu ze stránek frameworku – Výpis kódu 1).

Výpis kódu 1 Ukázka testu využívajícího framework Jasmine [80]

```
describe("A suite is just a function", function() {
  let a;

  it("and so is a spec", function() {
    a = true;

    expect(a).toBe(true);
  });
});
```

6 Implementace

Nyní si pojďme představit, jak jsme při implementaci systému postupovali. Postup byl značně ovlivněn požadavky *Firmy*, která si žádala:

1. Vytvořit velmi jednoduchý generický datový model, na který lze snadno transformovat podmnožinu dat od většiny zákazníků a modeluje nějakou část výrobní firmy.

2. Prozkoumat nástroj Metabase a vytvořit základní nástěnku, která by byla užitečná pro většinu zákazníků *Firmy*.

3. Navrhnout datový model pro vytvoření a ukládání mapování relačních modelů, přičemž pro vytvoření mapování bude zapotřebí nejprve zvolit reprezentaci pro relační modely a poté navrhnout reprezentaci jejich mapování.

4. Vytvořit demo verzi[1] mapovacího nástroje *Mapper*.

5. Implementovat správní aplikaci a integrovat do ní mapovací nástroj.

Kvůli velikosti projektu[2] si implementaci představíme abstraktněji a zmíníme jen nejdůležitější detaily.

6.1 Generický model a Metabase nástěnky

Pro účely vývoje byla vybrána podmnožina dat modelující vykazování práce zaměstnanců firmy na různých pracovištích. Model dat je znázorněn na obrázku 6.1. Můžeme vidět tři entity: *Employee*, *WorkReport* a *Worplace*.

Employee reprezentuje zaměstnance firmy, *WorkReport* pracovní výkaz s možností namapovat další informace jako *OrderId* (Id příkazu, který vedl k danému výkonu práce) a *ProductionOperationId* (pro nás také nepodstatné; slouží pro zpětnou identifikaci dat zákazníkem). Cizí klíč *WorkerId*, respektive *WorkPlaceId*, odkazuje na *Employee.Id*, respektive *Workplace.Id*. Poslední entita *Workplace* reprezentuje pracoviště, na kterém byl výkon práce vykázán.

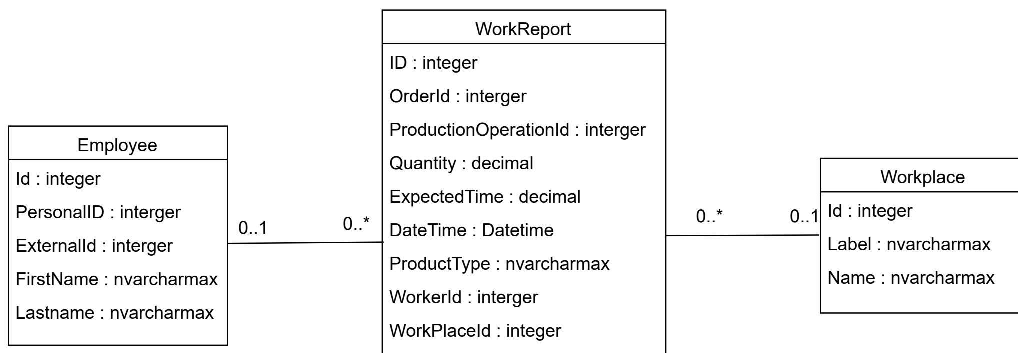
Co se Metabase nástěnky využívající představený datový model týče, pro její vytvoření jsme využili atributů *Quantity* (množství), *ExpectedTime* (očekávaný čas pro vykonání) a *ProductTime* (typ vzniklého produktu) entity *WorkReport*. Nástěnku ilustruje obrázek 6.2.

Můžeme vidět 4 grafy: součet vykázaných hodin v měsících, součet vykázaných hodin v měsících proložený součtem vyrobených jednotek v měsících, obdobný proložený graf s týdenním intervalem a koláčový graf znázorňující poměr počtu výkazů práce s pozitivním očekávaným časem pro vykonání vůči počtu všech výkazů. Poslední zmíněný graf slouží pro ilustraci validity dat (ne každý je poctivý ve vyplňování).

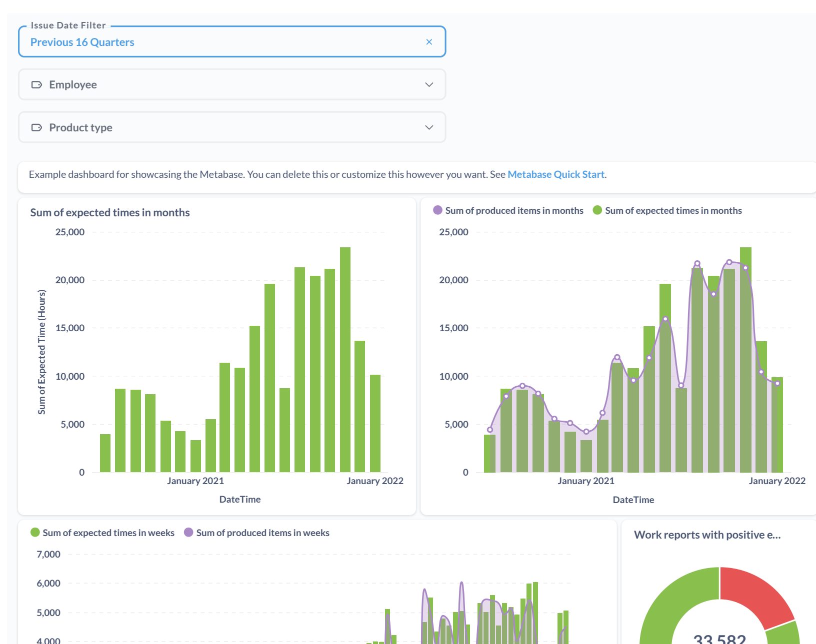
Takto implementovaná nástěnka uvítá každého zákazníka, který použije pro něj nasazenou instanci nástroje Metabase. Zákazník v může filtrovat pro vytvoření

[1]Demo verzí bývá označována základní neprodukční verze produktu sloužící zejména pro účely ukázky.

[2]Zdrojové kódy mají přes 700 KB, ve kterých je přes 20 000 řádků.



Obrázek 6.1 UML diagram generického modelu



Obrázek 6.2 Předkonfigurovaný Metabase – základní nástěnka

grafů použité výkazy podle zaměstnance, pracoviště či typu produktu, a tak sledovat jak se v konkrétních částech společnosti vyvíjí produkce a produktivita.

Metabase s popsanou nástěnkou, jedním vytvořeným administrátorským účtem a jedním API klíčem k rozhraní Metabase jsme jako Docker Image umístili do registru kontejnerů Docker Hub[3].

6.2 Mapování dat

Dalším krokem bylo navrhnout model reprezentující schéma relační databáze a poté model reprezentující jejich mapování. Oba modely si představíme jako UML diagramy C# tříd.

6.2.1 Reprezentace relačních modelů

Pro reprezentaci relačních modulů byl navržen model ilustrovaný obrázkem 6.3. Můžeme vidět, že základní entitou je *DbModel*. Její atribut *Name* reprezentuje jméno příslušné databáze, a kolekci tabulek (*Table*).

Každá tabulka má také své jméno (atribut *Name*) a dále má své schéma[4] (*Schema*), kolekci sloupců (*Columns* – kolekce entit *Column*) a může mít popis (*Description*).

Jednotlivé sloupce tabulek (entita *Column*) mají také svůj název a dále datový typ (atribut *DataType*). Datový typ sloupce je reprezentován abstraktní třídou[5] *DataTypeBase*, která má pouze atribut *IsNullable* indikující, zda daný sloupec může obsahovat *null* hodnoty.

Její konkretizace[6] NVarChar a NVarCharMax reprezentují řetězce Unicode znaků. Konkretizace *SimpleType* reprezentuje typy, které nepotřebují další parametry (oproti např. *NVarChar*, který potřebuje mít specifikovanou maximální délku), anebo je zanedbáváme. Konkretizace *UnknownDataType* představuje typy, které zatím nerozeznáváme, a tak jejich název při vytváření modelu databáze pouze uložíme jako atribut *StoreType*.

6.2.2 Reprezentace mapování relačních modelů

Mapování relačních modelů reprezentujeme datovým modelem ilustrovaným obrázkem 6.4. Hlavní třídou modelu je *EntityMapping*, která zapouzdřuje mapování sloupců tabulek ze zdrojové databáze (v našem případě databáze zákazníka) na sloupce jedné tabulky z cílové databáze (či entit našeho generické modelu – podsekce 6.1).

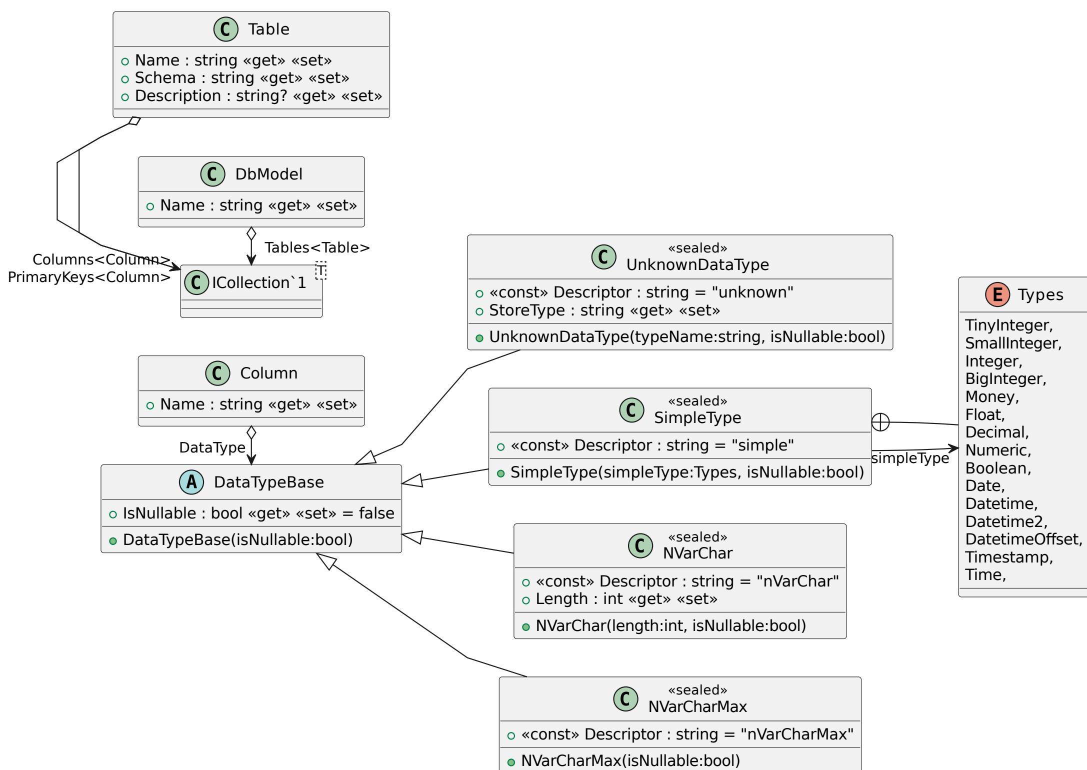
EntityMapping má tedy název (*Name*), respektive schéma (*Schema*), což je název, resp. schéma, dané cílové tabulky. Dále má slovník *ColumnMappings*, jehož

[3]Docker Image můžeme najít na `https://hub.docker.com/r/michaelsevcik/preconfigured-metabase`.

[4]Schéma rozděluje tabulky v databázi do skupin.

[5]pojem třída a entita budeme v popisu modelů zaměňovat, jelikož entita modelu je reprezentována vždy C# třídou.

[6]Konkrétní neabstaktní třída.



Obrázek 6.3 Model databáze reprezentovaný C# třídami

klíče jsou názvy sloupců dané cílové tabulky a hodnoty jsou příslušející sloupce (entity *SourceColumn*[7]) zdrojových entit (*ISourceEntity*).

Mezi zdrojové entity patří *SourceTable*, *CustomQuery* a *Join*. *SourceTable* reprezentuje konkrétní tabulku ze zdrojové databáze. *CustomQuery* představuje pojmenovaný vlastní SQL dotaz (atribut *Query*), jemuž jsme přiřadili sloupce, které jsou výsledkem vykonání definovaného vlastního dotazu ve zdrojové databázi.

Poslední zdrojovou entitou je *Join*, který představuje spojení zdrojových entit (zdrojových tabulek či vlastních dotazů) na základě podmínky *JoinCondition* podobně, jako je tomu v případě *SQL JOIN*.

JoinCondition reprezentuje podmínku, která musí být pro spojení splněna. Podmínka sestává ze dvou zdrojových sloupců, přičemž sloupce musí být z různých zdrojových entit, a vztahu mezi nimi (výčtový typ *Operator*). Podmínky se mohou také vázat pomocí *ConditionLink*, vztah mezi dvěma podmínkami představuje výčtový typ *LinkRelation*.

Celý model tak vlastně představuje zjednodušený SQL dotaz, který může mít vícero úrovní zanoření. Mapování mezi dvěma databázemi je kolekce entit *EntityMapping*.

6.3 Mapovací nástroj – Mapper

V předchozí sekci jsme se seznámili s datovými modely, na kterých bude mapovací nástroj *Mapper* pracovat. UI nástroje a proces jeho použití již také

[7]Entita *SourceColumn má navíc také datový typ (atribut DataType) se stejnou hierarchii, jako jsme si představili u entity Column v podsekci 6.2.1.*

Obrázek 6.4 Model mapování relačních tabulek

známe z podsekce 4.2.1. Před popisem samotné implementace je třeba si ujasnit, jaká je role této komponenty.

Mapper slouží pro vytvoření entity *EntityMapping* za pomoci informací ze zdrojové a cílové databáze a interakce (v podobě přidávání zdrojových entit a tahání čar) s uživatelem. Vstupem *Mapperu* tedy jsou:

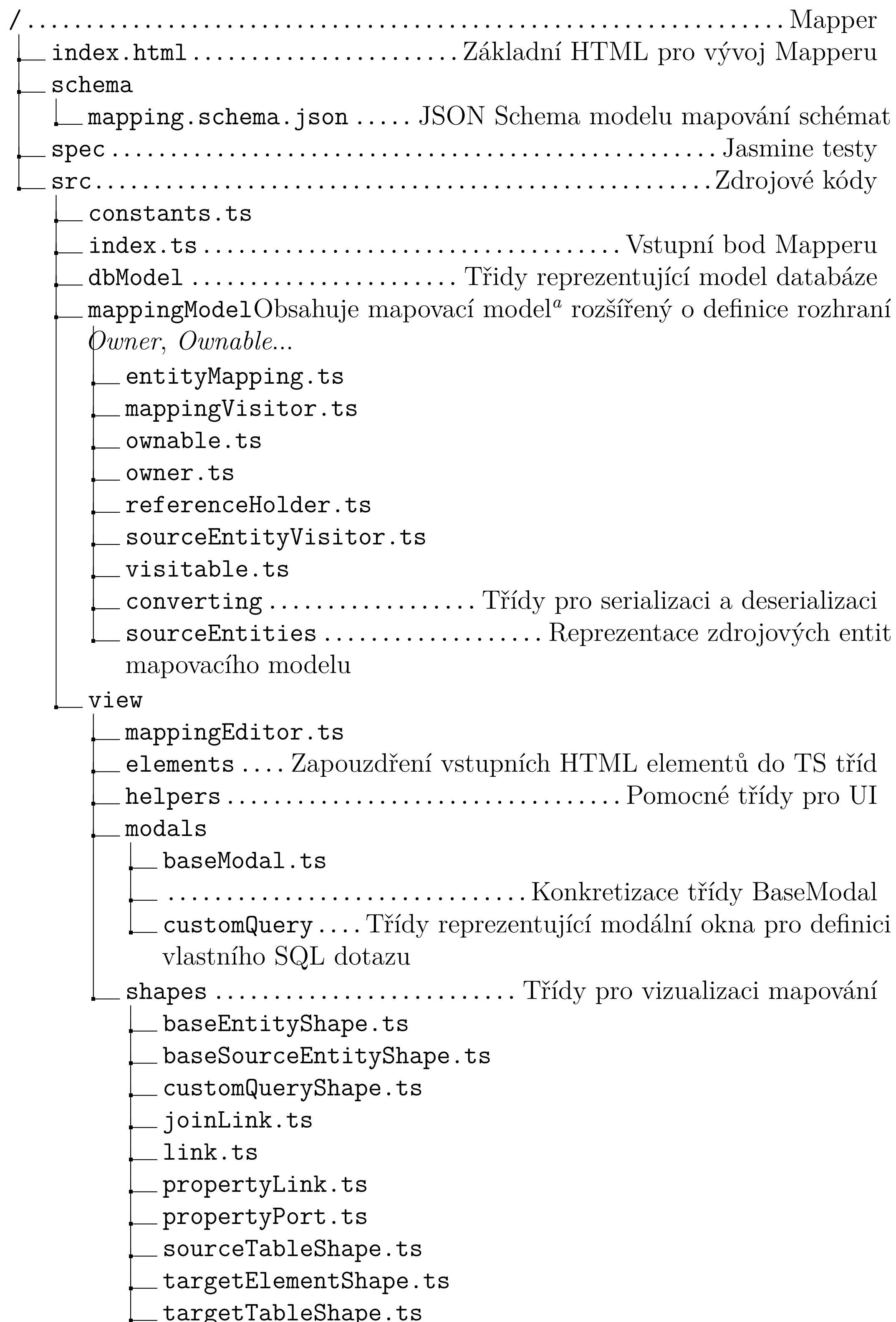
- serializovaný[8] model zdrojové databáze[9],
- serializovaná tabulka z cílové databáze[10],
- popřípadě již vytvořené serializované mapování dané tabulky z cílové databáze, které bude upravováno.

Výstupem je serializované mapování (*EntityMapping*) dané cílové tabulky daty ze zdrojové databáze. Nyní se pojďme podívat na to, jak jsme nástroj implementovali.

Nejprve se seznámíme se základním fungováním komponenty *Mapper* a přitom si představíme její zjednodušenou strukturou, kterou zachycuje obrázek 6.5. Vstupní branou do *Mapperu* je soubor *index.html*. Ten načte JS bundle[11] *Mapperu*. Načtením se *Mapper* spustí.

V případě samostatného spuštění, tzn. v případě, kdy *Mapper* není integrován do správní aplikace, se voláním funkce *loadDevelopmentView* ze souboru

[8]Serializovanými daty myslíme data ve formátu JSON.
[9]Databáze, z níž se data brát.
[10]Databáze, na jejíž strukturu se budou data mapovat.
[11]Bundle je nutné vygenerovat pomocí *npm*.



Obrázek 6.5 Zjednodušená struktura zdrojových souborů mapovacího nástroje *Mapper*.

[a]Obdobný již představenému v podsekci 6.2.2.

src/index.ts[12] načte vývojové prostředí.

Funkce inicializuje instanci třídy *MappingEditor*, což je třída sloužící pro správu mapování. Zajišťuje zpracování vstupních dat, inicializaci potřebných objektů knihovny JointJS[13] , vytváření UI objektů z View/shapes reprezentující objekty modelu mapování a registraci obsluhy událostí.

MappingEditor umožňuje buďto vytvořit nové mapování tabulky z cílové databáze (metoda *createFromTargetTable*), nebo načíst již vytvořené mapování v podobě entity *EntityMapping* (metoda *loadEntityMapping*).

V prvním případě *MappingEditor* vytvoří pouze odpovídající instance *EntityMapping*, která nemá žádné zdrojové entity, a *TargetTableShape* nacházející se v adresáři */src/view/shapes*[14]. V druhém případě *MappingEditor* využije pomocnou třídu *SourceEntititesToShapesTransformer* (z adresáře */src/view/helpers*), která hierarchii zdrojových entit z *EntityMapping.SourceEntity* převede do odpovídajících tvarů z adresáře *shapes*.

SourceEntititesToShapesTransformer využívá toho, že náš model mapování implementuje návrhový vzor *Visitor*, tedy třídy modelu mapování implementují rozhraní *Visitable* a samotný transformer rozhraní *SourceEntityVisitor*. Návrhového vzoru *Visitor* využíváme také pro serializaci mapování, umožňuje nám totiž zapouzdření různé aplikační logiky do *Visitor* tříd, díky čemuž ji nenecháváme pronikat do datového modelu.

Při vytváření instancí tvarů jsou jejich konstruktorům předány odpovídající instance tříd modelu mapování, čímž mezi nimi vzniká datové provázání (anglicky *data-binding*). Díky provázání mohou tvary při běhu *Mapperu* obsluhovat jednodušší uživatelské požadavky (vstupy uživatele) jako je natažení nového spojení dvou sloupců ap. Při obsluze tvary přímo aktualizují model mapování.

Složitější požadavky, jako přidání další zdrojové entity či požadavek na export mapování, obsluhuje přímo *MappingEditor*. Při přidávání zdrojové entity, ať už se jedná o tabulku zdrojové databáze (*SourceTable*), nebo *CustomQuery*, je totiž v případě, kdy přidáváme druhou či další zdrojovou entitu, nutné zařadit přidávanou entitu do hierarchie již existujících zdrojových entit. To zajišťuje metoda *handleSourceEntityShapeAddition*.

Při přidání v případě, kdy ještě nemáme žádnou zdrojovou entitu, metoda vytvoří odpovídající tvar a přidá zdrojovou entitu do *EntityMapping*. V opačném případě vytvoří nejprve instanci třídy *Join*, ve které je *Join.LeftSourceEntity = stávající EntityMapping.SourceEntity* a *Join.RightSourceEntity = přidávané zdrojové entitě*. Poté je vytvořena instance třídy *JoinModal*, která slouží pro definici podmínky *JoinCondition*. Po definici podmínky je podmínka přidána do *Join*, dále je vytvořen tvar odpovídající přidávané entitě a tvar *JoinLink*, znázorňující propojení nové zdrojové entity se zbytkem.

Dvojitým kliknutím na JoinLink může uživatel znovu otevřít *JoinModal* a upravit *JoinCondition*, to už ale zajistí samotný *JoinLink* díky *data-binding*.

Dalším složitějším požadavkem, který obsluhuje přímo *MappingEditor* je export mapování. To zajišťuje metoda *createSerializedMapping*, která nejprve převede

[12]Pro integraci do správní aplikace je nachystaná funkce *getMappingEditor(serializedSourceDbModel : string)*, která vrací inicializovanou instanci třídy *MappingEditor* definovanou v souboru *src/view/mappingEditor.ts*

[13]JointJS jsme si představili v podsekci 5.1.2.

[14]Třídy z adresáře *shapes* reprezentují tvary, které vidí uživatel, a jsou odvozeny ze tříd z knihovny *JointJS*, která zajišťuje jejich vykreslení.

EntityMapping na obyčejný JS objekt a poté využije *JSON.stringify* pro převedení do textové reprezentace. Struktura výsledné JSON reprezentace je popsána pomocí *JSON Schema*[15], které se nachází v */schema*.

Specifikem převodu na obyčejný JS objekt je, že potřebujeme zachovat reference na entity *SourceColumn*, které jsou nutné pro definici *JoinCondition* a *EntityMapping.ColumnMappings*. Zachování referencí řešíme přidáním identifikátorů (atribut *$id* s hodnotou unikátního identifikátoru) k prvnímu výskytu a nahrazením dalších výskytů objektem obsahujícím pouze *$ref* s hodnotou odpovídajícího unikátního identifikátoru. Samotný převod zajišťuje třída *MappingToPlainConvertorVisitor*[16].

Posledním složitějším požadavkem, který však řeší samotné tvary, je mazání tvaru z mapování, kdy musí být aktualizován model mapování a odstraněny navázané tvary jako je *PropertyLink*. Při aktualizaci modelu musí být odstraňovaná entita nahrazena ve vyšší vrstvě hierarchie zdrojových entit případným potomkem (potomky mají entity *Join* a *EntityMapping*).

Jelikož je model mapování hierarchický byl v kontextu *Mapperu* rozšířen o zpětné propojení (v terminologii grafů *zpětné hrany*) pomocí implementace rozhraních *Owner* a *Ownable* (definovaných v */src/mappingModel*). *Owner* je například *EntityMapping* a *Join*. *Ownable* jsou všechny zdrojové entity. Při odstraňování tedy odstraňovaná entita zavolá metodu na jejím vlastníkovi (*Owner) replaceChild* a předá ji svého potomka, nebo hodnotu *null*.

Nyní jsme získali představu o struktuře *Mapperu* a jeho fungování a můžeme se podívat na jeho architekturu. Jelikož *MappingEditor* vytváří objekty z *view* a obsluhuje složitější požadavky, přičemž jednodušší požadavky jsou obslouženy přímo tvary pomocí *data-binding* s modelem mapování, nabízí se přirovnání architektury *Mapperu* k architektuře *MVP* (*model-view-presenter*), konkrétně návrhovému vzoru *supervising controller*[17].

Můžeme totiž říct, že *MappingEditor* plní roli *supervising controller*, tedy je *presenter*, a *view* z *MVP* tvoří třídy tvarů.

6.4 Správní aplikace

V předchozí sekci jsme se seznámili s komponentou *Mapper*, kterou správní aplikace integruje. Krom poskytování *Mapperu* musí aplikace umožňovat správu zákazníků, získávání přístupových údajů k databázím zákazníků, generování modelu databází zákazníků, převod vytvořených mapování na příkazy k vytvoření potřebných databázových pohledů a nasazení, konfiguraci a správu instancí nástroje Metabase.

Nejdříve se podobně jako v předchozí sekci pojďme seznámit se strukturou správní aplikace a jejím fungováním. Obrázek 6.6 zachycuje její základní adresářovou strukturu.

Můžeme vidět, že implementace správní aplikace je rozdělená do dvou adresářů *test* (obsahující příslušné, zejména integrační testy) a *src*, kde se nachází většina

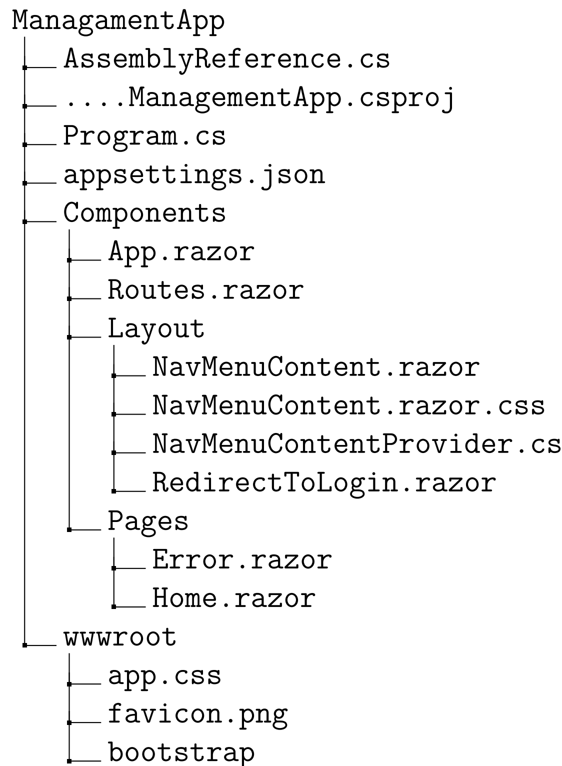
[15]Více na `https://json-schema.org/`.

[16]Ve správní aplikaci můžeme díky bohaté nabídce knihoven platformy *.NET* využít knihovnu System.Text.Json, která práci s referencemi umožňuje, a tak ji nemusíme implementovat znovu sami.

[17]Více na `https://zdrojak.cz/clanky/prezentacni-vzory-zrodiny-mvc/`.

```
SaaSPlatform
├── Dockerfile
├── SaaSPlatform.sln ..................Soubor solution pro vývoj ve VS
├── docker-compose.yaml … docker compose file pro spuštění
│                         aplikace se všemi závislostmi.
├── Architecture
├── ExampleMappings ... Obsahuje vzorové mapování vzorové cílové databáze
├── src.............................................Zdrojové kódy
│   ├── Common
│   ├── Images.........................Docker Images sloužící pro vývoj
│   │   ├── ingress-nginx-k3s
│   │   ├── mssql-example-customer-db
│   │   └── mssql-server-db
│   ├── ManagamentApp … Samotná Blazor aplikace integrující
│   │                   všechny moduly, zároveň jediný přímo
│   │                   spustitelný projekt.
│   └── Modules.....................................Moduly aplikace
│       ├── DataIntegration
│       ├── Deployment… Každý modul je rozdělen na C#
│       │   │           projekty podle rozšířené Clean
│       │   │           Architecture.
│       │   ├── Api
│       │   ├── Application
│       │   ├── Domain
│       │   ├── Infrastructure
│       │   ├── Pages ...................Projekt s Razor Pages – UI modulu
│       │   └── Persistence
│       ├── Notifications
│       └── Users
└── test ..................................................... Testy
    └── Modules
        ├── DataIntegration
        └── Deployment
```

Obrázek 6.6 Zjednodušená struktura zdrojových souborů *SaaSPlatform*.



Obrázek 6.7 Zjednodušená struktura zdrojových souborů aplikace *ManagementApp*.

zdrojových kódů. Adresář *src* obsahuje adresář *Images*, kde jsou uloženy Docker files definující vývojové a ukázkové závislosti projektu jako Docker Images.

Můžeme si povšimnout, že modul *Deployment* má na obrázku 6.6 znázorněný rozpad na jednotlivé C# projekty, podobně jsou rozděleny všechny moduly. Tento rozpad odpovídá mírně upravené vícevrstvé doménově orientované architektuře Clean Architecture[82] (zkráceně CA).

Zjednodušeně by se dalo říct, že podle CA rozdělujeme projekt do několika vrstev. Základní vrstvou je *Domain*, kde definujeme hodnotové třídy reprezentující naši doménu a rozhraní pro práci s nimi. jako jsou repositories[18]. *Domain* je obalena vrstvou *Application*, ve které definujeme a implementujme naši business logiku. Závěrem vrstvy *Application* společně obalují vrstvy *Infrastructure* (implementace interakce s vnějším světem) a *Presentation* (UI).

Naše úprava spočívá v tom, že vrstvu *Presentation* nazýváme *Pages* a přidáváme vrstvu *Persistence*, kde implementuje repositories, a *API*, ve které definujeme rozhraní pro integraci modulů.

V adresáři *Common* se nachází knihovny[19] společné pro všechny moduly, konkrétně: *Shared*, *Application*, *Infrastructure*, *Persistence* a *Components*.

Základ komponenty *Common* vychází z kurzu Milana Janoviče o modulárních monolitických aplikacích[20] Knihovna *Shared* definuje třídu *Result* (česky výsledek), *Result<T>* a *Error* společně s *extension* metodami, které slouží pro *fluent* práci

[18] Návrhový vzor repository – více na `https://medium.com/@pererikbergman/repository-design-pattern-e28c0f3e4a30`.
[19] Knihovnami myslíme v tomto případě C# projekty typu *class library*, či *Razor class library*.
[20] Více na `https://youtu.be/5dilYMii9T4?si=dpxbTpIiPrcDAmdO`.

Výpis kódu 2 Ukázka *fluent* použití *Result* objektů pro konfiguraci Metabase. Ukázka pochází ze třídy *MetabaseConfigurator*.

```
public async Task<Result> ConfigureMetabase(
    string CustomerId,
    string metabaseRootUrl,
    DefaultAdminSettings adminSettings)
{
    using var client = metabaseClientFactory.Create(metabaseRootUrl)

    // Cascade of configurations that will be fully executed
    // only if all of them succeed.
    var result = await client.WaitForMetabaseToStartResponding()
        .Bind(() => client
            .ChangeDefaultAdminEmail(adminSettings.Email))
        .Bind(() => client
            .ChangeDefaultAdminPassword(adminSettings.Password))
        .Bind(() => databaseSettingsAccessor
            .GetDatabaseSettings(CustomerId))
            .Map(client.ConfigureDatabaseAsync)
        .Bind(() => client
            .ConfigureSmtpAsync(smtpOptions.Value))
        .Bind(client.DeleteDefaultTokenAsync);

    if (result.IsFailure)
    {
        // log error - comment for the sake of brevity
    }

    return result;
}
```

s výsledky.

Použití výsledků nejlépe ilustruje Výpis metody 2, který ukazuje metodu *MetabaseConfigurator.ConfigureMetabase* z modulu *Deployment*. Můžeme vidět použití *extension* metod *Bind* a *Map*. Obzvláště zajímavé je použití kombinace obou metod, které můžeme vidět na řádcích 15 - 17.

Zde pomocí metody *Bind* navazujeme přístup k nastavení databáze zákazníka na úspěšné vykonání změny hesla admin účtu instance nástroje *Metabase* (řádek 14). V případě chyby při změně hesla navazující metoda *Bind* pouze přepošle chybový *Result*. Vrátí-li *databaseSettingsAccessor* úspěšný výsledek, metoda *Map* jej předá jako parametr metodě *client.ConfigureDatabaseAsync*.

V knihovně *Application* definujeme pomocné rozhraní *ISingleton*, *IScoped*, *ITransient*, které využíváme pro rozlišení životních cyklů objektů při registraci pomocí *DependencyInjection* (zkráceně DI). Tyto rozhraní využíváme ve společné knihovně *Infrastructure*, konkrétně v třídě *ServiceCollectionExtensions* pomocí metody *AddServicesWithLifetimeAsMatchingInterfaces(this IServiceCollection services, Assembly assembly)*. Ta při zavolání na *ServiceCollection* a předání *Assembly*[21], ve které se zamýšlené *Services* (služby) označené rozhraními *ISingleton*, *IScoped*, či *ITransient*, služby v *ServiceCollection* zaregistruje jako rozhraní

[21]Více na `https://learn.microsoft.com/en-us/dotnet/standard/assembly/`.

s odpovídajícím názvem[22].

Knihovna *Infrastructure* dále definuje pro náš modulární systém důležité rozhraní *IModuleInstaller* (instalátor modulu), které každý modul ve své vrstvě *Infrastructure* implementuje. Díky instalátorům modulů můžeme v jednotlivých modulech koncentrovat nutnou práci s DI, tedy *ServiceCollection*. Tyto instalátory využijeme v *ManagementApp* pro integraci všech modulů.

Společná knihovna *Persistence* obsahuje konstanty pro definici omezení velikosti různých typů používaných v aplikaci (např. *userId*), dále registruje třídu *ConnectionStringOptions*[23], která zapouzdřuje konfiguraci připojení k databázi aplikace.

Persistence dále obsahuje abstraktní třídu *BaseRepository<TDerived, TEntity, TContext>*, která slouží pro zjednodušení implementace jednotlivých *Repository* (s využití knihovny *Entity Framework Core*[24]) napříč moduly. Repository definujeme pro každý typ, který potřebujeme ukládat.

Poslední společnou knihovou je knihovna *Razor* tříd *Components*. Ta krom pomocných komponent, jako je *Alert*, definuje *MainLayout* (rozložení celé aplikace – menu v levé části a obsah v pravé) a *NavMenu*, což je komponenta reprezentující ono navigační menu.

Problém definice obsahu *NavMenu* ve společné knihovně by byla nutnost zavedení závislostí na jednotlivých modulech (v navigačním menu musí být odkazy na stránky UI modulů). Tento problém je vyřešen definicí a užitím rozhraní *INavMenuContentProvider*, jehož implementací a registrací v DI můžeme do již definovaného navigačního menu vkládat vlastní obsah z jiného projektu.

Nyní jsme si již představili společný základ pro implementaci aplikace. Ještě než si popíšeme implementaci jednotlivých modulů, pojďme si představit, co a jak dělá *ManagementApp* (její strukturu zachycuje obrázek 6.7).

ManagementApp integruje moduly, definuje obsah navigačního menu a domovskou a chybovou stránku. Integrace modulů spočívá ve využití jejich instalátorů, registraci jejich *Razor* stránek a přidání *Razor* komponent, které definují jejich podmenu, do komponenty reprezentující navigační menu celé aplikace.

Pro definici obsahu navigačního menu *ManagementApp* implementuje rozhraní *INavMenuContentProvider* třídou *NavMenuContentProvider*, kterou registruje v DI.

Veškerá práce s DI je soustředěna do vstupní brány celé aplikace – souboru */src/ManagementApp/Program.cs*. Zde je nejprve vytvořen *WebApplicationBu-*

[22]Odpovídajícím názvem rozumíme název prefixovaný velkým *I*, jak je v C# zvykem, tedy pro službu *ExampleService* je odpovídající rozhraní *IExampleService*. Pro jasnost např. označení *ExampleService* rozhraním *ISingleton* vyústí ve volání *ServiceCollection.AddSingleton<IExampleService, ExampleService>()*.

[23]Hodnota, kterou třída *ConnectionStringOptions* zapouzdřuje je konfigurovaná třídou *ConnectionStringSetup*. *Setup* třída dědí od *IConfigureOptions<T>*, což je rozhraní označující podobné třídy sloužící pro práci s konfigurací. Tento proces vychází z návrhového vzoru **options pattern**, který využíváme hojně napříč implementací. Více informací na `https://learn.microsoft.com/en-us/aspnet/core/fundamentals/configuration/options`.

[24]Knihovnu jsme si představili v podpodsekci Perzistence 5.1.1.

ilder[25] , zaregistrovány všechny služby a moduly, načež je vytvořena instance *WebApplication*[26]. Poté jsou zaregistrovány všechny *Razor* stránky a aplikace je spuštěna (metoda *WebApplication.Run()*).

6.4.1 Implementace modulů

Výše jsme již došli až ke spuštění samotné aplikace, pro pochopení celé její implementace je ještě nezbytné si představit implementaci jednotlivých modulů. Moduly si představíme v pořadí, ve kterém s nimi budou většinou uživatelé interagovat.

Users

Modul uživatelů slouží pro správu uživatelů, potažmo zákazníků. Oproti původnímu návrhu architektury také poskytuje rozhraní pro autentizaci a autorizaci.

Pro zajištění opravdu bezpečné a robustní autentizace a autorizace modul využívá knihovnu *Identity ASP.NET Core*[27]. Základ použití knihovny vychází ze startovacího projektu pro Blazor aplikace, který nabízí IDE *Visual Studio*. Jedná se o Razor stránky[28], které jsou potřebné pro přihlášení a správu samotného uživatelského účtu. My jsme tento základ upravili tak, aby odpovídal požadavkům na vyvíjený systém – přístup do aplikace na principu *jen na pozvání*.

Úpravy spočívaly v deaktivaci registrace a implementaci zadání hesla po rozkliknutí zvacího odkazu, který je zasílán e-mailem pomocí modulu *Notifications*. Generování odkazů, ověřovacích tokenů ap. zajišťuje knihovna *Identity*.

Modul uživatelů správcům systému nabízí také správu zákazníků. Ta je reprezentována rozhraním *IUserManager* z projektu *Domain*, Implementace rozhraní interně využívá *Identity ASP.NET Core*. Abychom odlišili uživatele podle jejich typu, v *Domain* také definujeme tzv. role. UI pro správu zákazníků se nachází v projektu *Pages*[29], jedná se o jednoduché formuláře předávající požadavky objektu typu *IUserManager*.

Abychom mohli přistupovat k informacím o přihlášeném uživateli a autorizaci založenou na rolích uživatelů, modul *Users* v projektu *Api* definuje rozhraní *IUserAccessor* a třídy sloužící pro autorizaci.

IUserAccesor umožňuje při zpracování požadavků modulům získat identifikátor aktuálně přihlášeného zákazníka či jeho e-mailovou adresu. Pro autorizaci definuje atributy *AuthorizeOnlyAdmins* a *AuthorizeOnlyCustomers* a *Razor* komponenty *AuthorizeAdminView* a *AuthorizeCustomerView*.

Atributy omezení přístupu k *Razor* stránce na základě role zákazníka. Oba atributy jsou odvozeny od třídy *AuthorizeAttribute* z knihoven *Microsoft*[30]. Komponenty *AuthorizeAdminView* a *AuthorizeCustomerView* jsou odvozeny od kom-

[25]Více na `https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.webapplicationbuilder`.

[26]Více na `https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.webapplication-`

[27]Více na `https://learn.microsoft.com/en-us/aspnet/core/security/authentication/identity`.

[28]Nachází se v */src/Modules/Users/Pages/Account*.

[29]Jak UI vypadá jsme si představili v podsekci 4.2.2.

[30]Více na `https://learn.microsoft.com/en-us/aspnet/core/security/authorization/simple#authorize-attribute-and-razor-pages`.

Výpis kódu 3 Ukázka definice domovské stránky v syntaxi *Razor*.

```
1  @page "/"
2  <PageTitle>Home</PageTitle>
3  <AuthorizeAdminView>
4      <Authorized>
5          <h1>Welcome Admin</h1>
6          <p>
7              Use the side bar to navigate through the application.
8          </p>
9      </Authorized>
10     <NotAuthorized>
11         <h1>
12             Log in to see more
13         </h1>
14     </NotAuthorized>
15 </AuthorizeAdminView>
16
17 <AuthorizeCustomerView>
18     <Authorized>
19         @*
20         // Customer welcome text ...
21         // Comment for the sake of brevity.
22         *@
23     </Authorized>
24 </AuthorizeCustomerView>
```

ponenty *AuthorizeView*[31] a na rozdíl od zmíněných atributů neomezují přístup k celé stránce ale pouze jejím částem. Jejich užití ukazuje výpis kódu 3.

DataIntegration

V předchozím modulu mohl správce systému přidat zákazníka a zákazník si mohl vytvořit přihlašovací heslo, a tak se přihlásit do aplikace. Nyní si představíme, jak funguje získávání informací o zákaznické databázi, generování jejího modelu[32], integrace mapovacího nástroje *Mapper* a vytvoření SQL příkazů pro vytvoření databázových pohledů na základě mapování z *Mapperu*.

Co se týče potřebných informací o zákaznické databázi, potřebujeme získat informace v rozsahu sestavení *connection string*[33]. Informace získáváme pomocí *Razor* stránky *DatabaseConnectionConfigurationForm* s formulářem[34].

Formulář po přijetí dat otestuje jejich validitu navázáním spojení se zákaznickou databází. Pro sestavení *connection string* formulář využívá objekt typu

[31]Více na `https://learn.microsoft.com/en-us/aspnet/core/blazor/security/#authorizeview-component`.

[32]model jsme si představili v podsekci 6.2.1.

[33]Více informací na `https://learn.microsoft.com/en-us/dotnet/framework/data/adonet/connection-string-syntax`.

[34]Formulář je ilustrován obrázkem 4.16.

DbConnectionStringBuilder [35]. Po sestavení formulář využije ADO.NET[36] k pokusu o připojení se k dané databázi. Pokud je připojení navázáno, zpracování formuláře *connection string* uloží pomocí rozhraní *CustomerDatabaseConnectionConfigurationManager* a zobrazí tlačítko *Create Model*.

Po stisku tlačítka *Create Model* stránka *DatabaseConnectionConfigurationForm* předá požadavek na vytvoření modelu databáze objektu implementujícímu rozhraní *ICustomerDbModelManager*. *CustomerDbModelManager* využije typu databáze odpovídající implementaci rozhraní *IDbModelFactory*, a tak vytvoří model databáze, který dále uloží.

Implementaci rozhraní *IDbModelFactory* máme zatím pouze jednu a to *MSSQLDbModelFactory*. Ta využívá implementaci rozhraní *IDatabaseModelFactory*[37], která pochází z knihovny pro perzistenci *Entity Framework Core*. *MSSQLDbModelFactory* tedy pouze převádí informace z *IDatabaseModelFactory* na naši reprezentaci modelu databáze. Díky tomu, že *EF Core* umožňuje pracovat s různými typy databází, můžeme v budoucnu snadněji přidat podporu dalších typů databází.

Po vytvoření modulu databáze se zákazník může přesunout do prostředí nástroje *Mapper*. Jak už jsme zmínili, modul *DataIntegration Mapper* integruje do správní aplikace. Pro integraci *Mapperu* je vytvořen v adresáři modulu *DataIntegration* další projekt tříd *Razor – MapperComponent*, podobně jako jsme navrhli v sekci 4.1[38].

Jelikož je *Mapper* psán v TS, musíme využít, pro tyto účely firmou Microsoft vytvořený, *JSRuntime*[39]. Použití JSInterop zapouzdřuje třída *MapperJsInterop*. Tu využívá *Razor* komponenta *Mapper*. Při vytváření instance *MapperJsInterop* dojde k načtení JS bundlu *Mapperu* a vytvoření JS objektu *MappingEditor*.

Poté *Razor* komponenta *Mapper* interaguje s *MappingEditorem* za použití *MapperJsInterop* jako prostředníka. *Razor* komponenta *Mapper* tedy zajišťuje předání modelů databází, výběr tabulky z cílové databáze, která je zrovna mapovaná, a perzistenci mapování. Pro perzistenci využívá rozhraní *ISchemaMappingRepository*.

Jelikož *MapperJsInterop* potřebuje načíst bundl *Mapperu*, projekt *MapperComponent* obsahuje zdrojové soubory *Mapperu* ve formě, jakou jsme si představili v předchozí sekci, a v konfiguračním souboru projektu je nastaveno, aby se při kompilaci také vytvořil potřebný JS bundl ze zdrojových souborů.

Po vytvoření mapování může zákazník přejít k nasazení instance nástroje *Metabase*. K jejímu nasazení potřebuje modul *Deployment* vytvořit potřebné databázové pohledy v databázi zákazníka, pro což potřebuje z modulu *DataIntegration* získat *connection string* k zákaznické databázi a SQL příkazy vytvářející databázové pohledy, proto modul *DataIntegration* v projektu *Api* definuje rozhraní

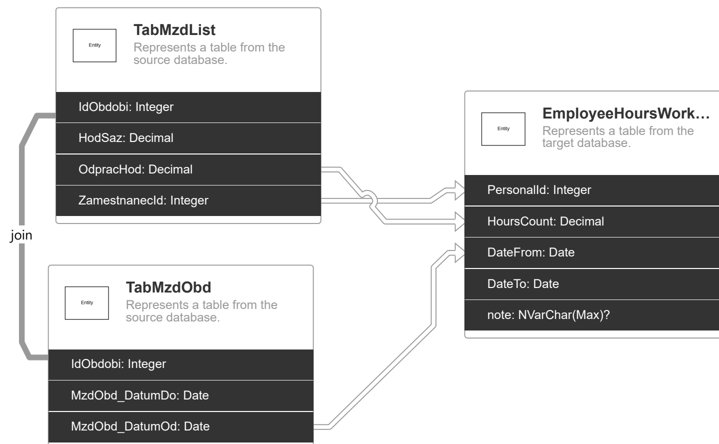
[35]Jelikož zatím podporujeme pouze MS-SQL server, využíváme implementaci přímo pro něj určenou SqlConnectionStringBuilder. Více informací na `https://learn.microsoft.com/en-us/dotnet/api/system.data.common.dbconnectionstringbuilder` a na `https://learn.microsoft.com/en-us/dotnet/api/microsoft.data.sqlclient.sqlconnectionstringbuilder`.

[36]Knihovna pro jednotný přístup k různým druhům databází. Více informací na `https://learn.microsoft.com/en-us/dotnet/framework/data/adonet/ado-net-overview`.

[37]Více informací na `https://learn.microsoft.com/en-us/dotnet/api/microsoft.entityframeworkcore.scaffolding.idatabasemodelfactory`.

[38]Konkrétně na obrázku 4.3.

[39]*JSInteop* a s ním spojený *JSRuntime* jsme představili v podsekci 5.1.2.



Obrázek 6.8 Ukázka mapování v Mapper

IDataIntegrationService.

Pro implementaci rozhraní jsme v modulu *DataIntegration* implementovali třídu *EntityMappingViewGenerator*, která generuje potřebné SQL příkazy. Získání *connection string* je jednoduché – přístup do *CustomerDbConnectionConfigurationRepository*.

EntityMappingViewGenerator využívá pro generování příkazů vytvářejících databázové pohledy třídu *SqlViewVisitor*, která prochází jako návštěvník z návrhové vzoru *Visitor* entity modelu mapování od *EntityMapping* níže po hierarchii zdrojových entit. Pro každou zdrojovou entitu vygeneruje SQL dotaz. Dotazy se na základě hierarchie zanořujeme do sebe.

Zanořování využíváme pro entity *EntityMapping*, *Join* a v jedné úrovni pro *CustomQuery*. Pro jasnost uvedeme příklad mapování cílové tabulky *EmployeeHoursWorked* zdrojovými tabulkami *TabMzdList* a *TabMzdObd*, které znázoňuje obrázek 6.8. Výsledný SQL příkaz je uveden ve výpisu kódu 4.

Deployment

Předpokládejme, že nyní již zákazník vytvořil mapování, a tak se může přesunout k nasazení instance nástroje Metabase. Před samotným nasazení instance musí modul *Deployment* ještě v zákaznické databázi vytvořit sadu databázových pohledů a po nasazení instance ji nakonfigurovat. Pojďme se podívat, jak to vše modul *Deployment* dělá.

Zákazník interaguje s modulem přes stránku *MetabaseDeployment*, která jsou jsme si představili na obrázku 4.18.

Tlačítko *Generate views* je přímo propojeno s rozhraním *IDataIntegationService* představeným výše a po stisku pouze zobrazí získané SQL příkazy. Tlačítko *Automatically deploy views* obdobně získá SQL příkazy společně s *connection string* k databázi zákazníka, poté pomocí třídy *DatabaseViewDeployer* rozhraní nasadí.

DatabaseViewDeployer, stejně jako modul *DataIntegration*, využívá k připojení knihovnu *ADO.NET*. Pomocí takto vytvořeného připojení *DatabaseViewDeployer* předá SQL příkazy databázi zákazníka k vykonání, načež vrátí výsledek operace.

Výpis kódu 4 Ukázka SQL příkazu pro vytvoření jednoho databázového pohledu.

```
CREATE VIEW [dbo].EmployeeHoursWorked AS
SELECT
    join1.TabMzdList__ZamestnanecId AS PersonalId,
    join1.TabMzdList__OdpracHod AS HoursCount,
    join1.TabMzdObd__MzdObd_DatumOd AS DateFrom,
    NULL AS DateTo,
    NULL AS note
FROM
    (SELECT
        TabMzdList.TabMzdList__IdObdobi,
        TabMzdList.TabMzdList__HodSaz,
        TabMzdList.TabMzdList__OdpracHod,
        TabMzdList.TabMzdList__ZamestnanecId,
        TabMzdObd.TabMzdObd__IdObdobi,
        TabMzdObd.TabMzdObd__MzdObd_DatumDo,
        TabMzdObd.TabMzdObd__MzdObd_DatumOd
    FROM
        (SELECT
            TabMzdList.ZamestnanecId AS TabMzdList__ZamestnanecId,
            TabMzdList.OdpracHod AS TabMzdList__OdpracHod,
            TabMzdList.HodSaz AS TabMzdList__HodSaz,
            TabMzdList.IdObdobi AS TabMzdList__IdObdobi
        FROM
            [dbo].TabMzdList TabMzdList
        ) TabMzdList
    INNER JOIN
        (SELECT
            TabMzdObd.MzdObd_DatumOd AS TabMzdObd__MzdObd_DatumOd,
            TabMzdObd.MzdObd_DatumDo AS TabMzdObd__MzdObd_DatumDo,
            TabMzdObd.IdObdobi AS TabMzdObd__IdObdobi
        FROM
            [dbo].TabMzdObd TabMzdObd
        ) TabMzdObd
    ON
       TabMzdObd.TabMzdObd__IdObdobi = TabMzdObd.TabMzdObd__IdObdobi
    ) join1
```

Po úspěšném vytvoření databázových pohledů může zákazník vyplnit přístupové údaje, které bude využívat pro přihlašování k jeho instanci nástroje Metabase, a stiskem tlačítka *DeployMetabase* spustit proces nasazování a konfigurace nové instance.

Proces je spuštěn voláním metody *DeployMetabaseAsync* třídy *MetabaseDeployer*. Nasazení instance probíhá v Kubernetes klastru. Pro jeho obsluhu využívá *MetabaseDeployer* rozhraní *IKubernetes* z knihovny *KubernetesClient*[40].

MetabaseDeployer při nasazení postupuje následovně:

1. Vytvoří záznam o nasazení Metabase pomocí *DeploymentRepository*.
2. Vytvoří a nasadí instanci nástroje v Kubernetes klastru. Metabase nasazujeme jako předpřipravenou Docker Image popsanou v sekci 6.1.
3. Předá Ingress Nginx ovladači Kubernetes konfiguraci URL cesty, přes kterou bude nasazená instance dostupná vnějšímu světu. Nejprve volí náhodnou URL cestu, aby mohl provést konfiguraci (zejména změnit přihlašovací údaje a odstranit API klíč).
4. Počká než Kubernetes klastr instanci Metabase nasadí a než je kontejner Metabase spuštěn.
5. Provede konfiguraci pomocí rozhraní IMetabaseConfigurator.
6. Změní konfiguraci přístupové URL cesty, aby odpovídala vzoru */metabase-{metabaseId}*, kde *metabaseId* je identifikátor záznamu o nasazení.
7. Pomocí rozhraní *IIntegrationNotifier* pošle notifikaci o nasazení s odkazem na novou instanci Metabase.
8. Vrátí výsledek nasazení.

Rozhraní *IMetabaseConfigurator* implementujeme třídou *MetabaseConfigurator*, jejíž jedinou metodu jsme si již ukázali ve výpisu kódu 2. Můžeme vidět, že metoda využívá factory *metabaseClientFactory*, která s předanou URL adresou instance Metabase vyrobí objekt třídy *MetabaseClient*. *MetabaseClient* pouze zapouzdřuje API nástroje Metabase[41]. *MetabaseConfigurator* pomocí *MetabaseClient* provede konfiguraci a vrátí její výsledek.

Poslední, co nám chybí k pochopení nasazení Metabase, je popis implementace *IIntegrationNotifier*. Ta se nachází v Api projektu *DataIntegration* a pouze předává notifikaci o nasazení modulu *Notifications*, který odešle notifikaci jako e-mail příslušnému zákazníkovi.

Nyní již víme, jak je Metabase nasazován. Pojďme si tedy představit, jak a kdy je instance Metabase mazána. V případě, že v modulu Users odstraníme zákazníka, je modulu *DataIntegration* předána zpráva o smazání uživatele přes volání metody z Api modulu *IIntegrationService.HandleCustomerDeletionAsync*. Implementace metody využívá *MetabaseDeployer.DeleteMetabase* pro odstranění instance nástroje z Kubernetes klastru a pomocí *DeploymentRepository* odstraní záznam nasazení.

[40] Knihovnu *KubernetesClient* jsme si představili v podsekci 5.2.1.

[41] O API Metabase lze nalézt více informací na `https://www.metabase.com/docs/latest/api-documentation`.

Notifications

Posledním a nejjednodušším modulem správní aplikace je modul *Notifications*. Jak jsme již zmínili, slouží pro zasílání notifikací zákazníkům. Tyto notifikace zasílá formou e-mailových zpráv, pro jejichž odeslání používá protokol *SMTP* skrze rozšířenou knihovnu *MailKit*[42].

Jediným problémem modulu *Notifications* je, že potřebuje e-mailové adresy uživatelů, které drží modul *Users*. Tento problém je vyřešen pomocí rozhraní *IUserEmailAccessor*, které je definované v *Api* projektu modulu *Notifications* a implementováno modulem *Users* v jeho *Api* projektu.

Popisem modulu *Notifications* jsme uzavřeli popis implementace správní aplikace. Abychom ji mohli použít, je nejprve nutné zajistit všechny její externí závislosti (přístup k *SMTP* serveru, přístup k databázi, přístup ke Kubernetes klastru...). Celý popis spuštění je popsán v příloze A.1.

6.4.2 Testování

Závěrem si uvedeme, jak jsme správní aplikaci testovali. Co se týče komponenty *Mapper*, implementovali jsme několik testů serializace a deserializace modelu mapování a modelu databází. Testy *Mapperu* využívají knihovnu *Jasmine*, představenou v sekci 5.3.

Složitější je testování samotné správní aplikace, kde jsme potřebovali ověřit správnou interakci s externími systémy. Pro definici všech testů je využit testovací framework *NUnit*. Pro implementaci integračních testů jsme využili framework *TestContainers*[43] a mokovací knihovnu *Moq*[44], díky které můžeme závislosti testované komponenty či třídy simulovat[45].

Framework *TestContainers* umožňuje vytvářet skutečné závislosti jejich spuštěním v podobě Docker kontejnerů, čehož využíváme při testování konfigurace instancí Metabase a testování nasazování a mazání instancí Metabase pomocí *MetabaseDeployer*.

V prvním případě spustíme Metabase skrz *TestContainers* jako Docker kontejner a pomocí PreconfiguredMetabaseClient zkusíme udělat změnu konfigurace. V druhém případě nám *TestContainers* poslouží pro vytvoření testovacího Kubernetes klastru, ve kterém nejprve *MetabaseDeployer* zkusí nasadit instanci Metabase a poté ji smazat.

Krom výše uvedeného testování je správní aplikace také ověřena experimentálně ve vývojovém a ukázkovém prostředí, které popisujeme v příloze A.1. Příloha A.2 představuje elektronické přílohy této práce.

[42] Více informací na `https://github.com/jstedfast/MailKit`.

[43] Více informací na `https://testcontainers.com/`.

[44] Více informací na `https://github.com/devlooped/moq`.

[45] Knihovna *Moq* vytvoří pro dané rozhraní simulovaný objekt, kterému můžeme zadefinovat chování.

Závěr

V úvodu této práce jsme si stanovili za cíl navrhnout a implementovat systém, který umožní poskytování předkonfigurovaného webového BI nástroje Metabase formou SaaS. Tento cíl jsme si upřesnili definicí funkčních a nefunkčích požadavků v kapitole 3.

Implementací systému dle jeho návrhu designu z kapitoly 4 se nám podařilo splnit všechny stanovené funkční požadavky, a tak vyvinout plně funkční systém, který je již postupně zaváděn ve *Firmě*. Co se týče nefunkčních požadavků, byly ověřeny všechny krom požadavku na použitelnost, jelikož uživatelské testování bude probíhat nad rámec této bakalářské práce.

Jako hlavní přínosy této práce vnímáme:

1. **Efektivní integraci dat**: Vyvinutý systém umožňuje zákazníkům snadno propojit jejich ERP systémy s Metabase, což výrazně zkracuje čas a snižuje náklady spojené s ručním vytvářením importovacích skriptů či vlastní implementací databázových pohledů.

2. **Automatizaci nasazení**: Implementace využívající Kubernetes umožňuje rychlé a spolehlivé nasazení instancí Metabase pro jednotlivé zákazníky, což zvyšuje škálovatelnost a flexibilitu celého řešení.

Budoucí výzkum a vývoj by se mohl zaměřit zejména na postupné zlepšování přívětivosti uživatelského rozhraní. Další možným směrem je implementace podpory více typů databázových systémů.

Literatura

1. *Plánování podnikových zdrojů – Wikipedie* [online]. [cit. 2023-12-29]. Dostupné z: https://cs.wikipedia.org/wiki/Pl%C3%A1nov%C3%A1n%C3%AD_podnikov%C3%BDch_zdroj%C5%AF.
2. ELENA, Cebotarean et al. Business intelligence. *Journal of Knowledge Management, Economics and Information Technology*. 2011, roč. 1, č. 2, s. 1–12.
3. METABASE, Inc (ed.). *Metabase documentation* [online]. [cit. 2024-03-19]. Dostupné z: https://www.metabase.com/docs/latest/.
4. INC, Gartner. *Magic Quadrant for Data integration*. 2023.
5. MICROSOFT. *SQL Server Integration Services* [online]. [cit. 2024-04-16]. Dostupné z: https://learn.microsoft.com/en-us/sql/integration-services/sql-server-integration-services.
6. TALEND, Qlik. *Talend Open Studio: Open-source ETL and Free Data Integration | Talend* [online]. [cit. 2024-03-22]. Dostupné z: https://www.talend.com/products/talend-open-studio/.
7. MICROSOFT. *Microsoft Data Platform – SQL Server* [online]. [cit. 2024-04-17]. Dostupné z: https://www.microsoft.com/en-us/sql-server.
8. MICROSOFT (ed.). *Integration Services (SSIS) Connections* [online]. [cit. 2024-01-25]. Dostupné z: https://learn.microsoft.com/en-us/sql/integration-services/connection-manager/integration-services-ssis-connections?view=sql-server-ver16.
9. MICROSOFT (ed.). *Integration Services Transformations - SQL Server Integration Services (SSIS)* [online]. 2023. [cit. 2024-03-24]. Dostupné z: https://learn.microsoft.com/en-us/sql/integration-services/data-flow/transformations/integration-services-transformations.
10. PATEL, Bhavesh (ed.). *Synchronize Table Data Using a Merge Join in SSIS* [online]. [cit. 2024-01-25]. Dostupné z: https://www.mssqltips.com/sqlservertip/5082/synchronize-table-data-using-a-merge-join-in-ssis/.
11. MICROSOFT (ed.). *SSIS Designer - SQL Server Integration Services (SSIS)* [online]. [cit. 2024-01-25]. Dostupné z: https://learn.microsoft.com/en-us/sql/integration-services/ssis-designer?view=sql-server-ver16.
12. PATEL, Bhavesh. *SSIS Mapping* [online]. [cit. 2024-01-25]. Dostupné z: https://www.mssqltips.com/tipimages2/5082_synchronization-jtk.012.png.
13. PATEL, Bhavesh. *SSIS Package* [online]. [cit. 2024-01-25]. Dostupné z: https://www.mssqltips.com/tipimages2/5082_synchronization-jtk.001.png.

14. Khanna, Vinay (ed.). *What Does Automating Your Integration Workflow With SSIS Entail* [online]. 2023. [cit. 2024-03-24]. Dostupné z: https://medium.com/@noel.B/what-does-automating-your-integration-workflow-with-ssis-entail-067232842de4.
15. Zagade, Ashutosh (ed.). *Streamlining Your Data Workflow: How SSIS, SSAS, and SSRS can Help* [online]. 2023. [cit. 2024-03-25]. Dostupné z: https://www.linkedin.com/pulse/streamlining-your-data-workflow-how-ssis-ssas-ssrs-can-zagade/.
16. Microsoft (ed.). *Integration Services Programming Overview* [online]. 2023. [cit. 2024-03-25]. Dostupné z: https://learn.microsoft.com/en-us/sql/integration-services/integration-services-programming-overview?view=sql-server-ver16.
17. Microsoft (ed.). *Walkthrough: Publish an SSIS Package as a SQL View - SQL Server Integration Services (SSIS)* [online]. [cit. 2024-01-25]. Dostupné z: https://learn.microsoft.com/en-us/sql/integration-services/data-flow/walkthrough-publish-an-ssis-package-as-a-sql-view?view=sql-server-ver16.
18. Microsoft. *SSIS Designer Toolbox* [online]. 2023. [cit. 2024-03-19]. Dostupné z: https://learn.microsoft.com/en-us/sql/integration-services/media/denali-designerandtoolbox.gif.
19. Gartner, Inc (ed.). *Talend Open Studio review in 'Data and Analytics - Others'* [online]. 2023. [cit. 2024-01-30]. Dostupné z: https://www.gartner.com/reviews/market/data-and-analytics-others/vendor/qlik-talend/product/talend-open-studio/review/view/4942970.
20. Talend, Qlik (ed.). *Supported systems, databases, and business applications by Talend components* [online]. [cit. 2024-01-30]. Dostupné z: https://help.talend.com/r/en-US/7.3/installation-guide-mac/supported-systems-databases-and-business-applications-by-talend-components.
21. Talend (ed.). *Talend component-runtime: Talend Component Kit (implementation repository)* [online]. [cit. 2024-03-25]. Dostupné z: https://github.com/Talend/component-runtime.
22. Talend, Qlik (ed.). *Building Jobs* [online]. [cit. 2024-01-30]. Dostupné z: https://help.talend.com/r/en-US/7.3/open-studio-user-guide/building-jobs.
23. Talend, Qlik (ed.). *Data Integration in an AWS Environment* [online]. [cit. 2024-01-30]. Dostupné z: https://www.talend.com/resources/data-integration-aws/.
24. Talend, Qlik (ed.). *What is ELT - SQL templates* [online]. [cit. 2024-01-30]. Dostupné z: https://help.talend.com/r/en-US/8.0/open-studio-user-guide/what-is-elt.
25. Inc, Gartner. *Magic Quadrant for Analytics and Business Intelligence Platforms.* 2021.
26. Microsoft. *Power BI - Data Visualization* [online]. [cit. 2024-04-17]. Dostupné z: https://www.microsoft.com/en-us/power-platform/products/power-bi.

27. Tableau. *Tableau Desktop | Connect, analyze, and visualize any data* [online]. [cit. 2024-04-17]. Dostupné z: https://www.tableau.com/products/desktop.

28. TrustRadius (ed.). *49 Shocking Business Intelligence Statistics for 2021* [online]. [cit. 2024-02-02]. Dostupné z: https://solutions.trustradius.com/vendor-blog/business-intelligence-statistics-and-trends/.

29. *Microsoft Power BI Reviews, Ratings & Features 2024 | Gartner Peer Insights* [online]. [cit. 2024-02-02]. Dostupné z: https://www.gartner.com/reviews/market/analytics-business-intelligence-platforms/vendor/microsoft/product/microsoft-power-bi.

30. Microsoft (ed.). *Data sources in Power BI Desktop* [online]. [cit. 2024-02-03]. Dostupné z: https://learn.microsoft.com/en-us/power-bi/connect-data/desktop-data-sources.

31. Naba, Ijaz (ed.). *Handling Big Data: Performance and Limits of Power BI* [online]. [cit. 2024-02-03]. Dostupné z: https://www.alphabold.com/handling-big-data-performance-and-limits-of-power-bi/.

32. Microsoft (ed.). *Find Insights in your reports - Power BI* [online]. [cit. 2024-02-03]. Dostupné z: https://learn.microsoft.com/en-us/power-bi/create-reports/insights.

33. Microsoft (ed.). *What is Power BI Desktop?* [online]. [cit. 2024-02-04]. Dostupné z: https://learn.microsoft.com/en-us/power-bi/fundamentals/desktop-what-is-desktop.

34. Abhishek, Jain (ed.). *Why Are Big Companies Using Microsoft Power BI* [online]. [cit. 2024-02-03]. Dostupné z: https://datafloq.com/read/why-are-big-companies-using-microsoft-power-bi-2/.

35. Gartner, Inc. (ed.). *Salesforce Tableau Reviews, Ratings & Features 2024* [online]. [cit. 2024-03-26]. Dostupné z: https://www.gartner.com/reviews/market/analytics-business-intelligence-platforms/vendor/salesforce-tableau/product/tableau.

36. Skillmea (ed.). *Power BI nebo Tableau? Který BI nástroj si vybrat v roce* [online]. 2023. [cit. 2024-03-26]. Dostupné z: https://skillmea.cz/blog/power-bi-vs-tableau-co-si-vybrat-v-roce-2023.

37. Tableau (ed.). *Workbooks and Sheets | Tableau Desktop* [online]. [cit. 2024-03-26]. Dostupné z: https://help.tableau.com/current/pro/desktop/en-us/environ_workbooksandsheets.htm.

38. Tableau. *Tableau Desktop Dashboard* [online]. [cit. 2024-01-25]. Dostupné z: https://www.tableau.com/sites/default/files/2022-05/Products_Desktop_Intro.png.

39. Metabase (ed.). *Data Sources | Metabase* [online]. [cit. 2024-03-26]. Dostupné z: https://www.metabase.com/data_sources/.

40. Pringle, Tina (ed.). *15 reasons why specifications are still important* [online]. [cit. 2024-03-21]. Dostupné z: https://manufacturers.thenbs.com/resources/knowledge/15-reasons-why-specifications-are-still-important.

41. *Data mapping - Wikipedia* [online]. [cit. 2024-01-20]. Dostupné z: https://en.wikipedia.org/wiki/Data_mapping.
42. *Dashboard (business) - Wikipedia* [online]. [cit. 2024-01-22]. Dostupné z: https://en.wikipedia.org/wiki/Dashboard_(business).
43. HEATH, Fred. *Managing Software Requirements the Agile Way*. 1st edition. Packt Publishing, 2020. ISBN 1-80020-646-1.
44. SOMMERVILLE, Ian. *Software engineering*. Software engineering. Tenth edition. Boston: Pearson, 2016. ISBN 978-1-292-09613-1.
45. KHOMENKO, Tetiana (ed.). *Non-Functional Requirements Examples: Your Comprehensive Guide with Definitions - Testomat.io* [online]. [cit. 2024-01-18]. Dostupné z: https://testomat.io/blog/non-functional-requirements-examples-definition-complete-guide/.
46. MICROSOFT (ed.). *Work with multiple containers using Docker Compose - Visual Studio (Windows)* [online]. [cit. 2024-03-19]. Dostupné z: https://learn.microsoft.com/en-us/visualstudio/containers/tutorial-multicontainer?view=vs-2022.
47. BROWN, Simon (ed.). *The C4 model for visualising software architecture* [online]. 2018. [cit. 2024-03-27]. Dostupné z: https://c4model.com/.
48. MICROSOFT. *ASP.NET Core Blazor hosting models* [online]. [cit. 2024-04-19]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/blazor/hosting-models?view=aspnetcore-8.0.
49. KUBERNETES. *Pods* [online]. 2024-04. [cit. 2024-04-24]. Dostupné z: https://kubernetes.io/docs/concepts/workloads/pods/.
50. MICROSOFT. *C# | Modern, open-source programming language for .NET* [online]. [cit. 2024-04-19]. Dostupné z: https://dotnet.microsoft.com/en-us/languages/csharp.
51. MICROSOFT. *(.NET glossary - .NET)* [online]. [cit. 2024-04-19]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/standard/glossary.
52. MICROSOFT. *What is NuGet and what does it do?* [online]. [cit. 2024-04-19]. Dostupné z: https://learn.microsoft.com/en-us/nuget/what-is-nuget.
53. MICROSOFT. *ASP.NET Core Blazor* [online]. [cit. 2024-04-21]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-8.0&WT.mc_id=dotnet-35129-website.
54. MICROSOFT. *What is ASP.NET?* [online]. [cit. 2024-04-21]. Dostupné z: https://dotnet.microsoft.com/en-us/learn/aspnet/what-is-aspnet.
55. MICROSOFT. *Overview of ASP.NET Core SignalR* [online]. [cit. 2024-04-22]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/signalr/introduction?view=aspnetcore-8.0&WT.mc_id=dotnet-35129-website.

56. Microsoft. *Use ASP.NET Core SignalR with Blazor* [online]. [cit. 2024-04-22]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/blazor/tutorials/signalr-blazor?view=aspnetcore-8.0&tabs=visual-studio.
57. Fette, I.; Melnikov, A. *The WebSocket Protocol* [Internet Requests for Comments]. RFC Editor, 2011-12. RFC, 6455. RFC Editor. issn 2070-1721. Dostupné také z: http://www.rfc-editor.org/rfc/rfc6455.txt. http://www.rfc-editor.org/rfc/rfc6455.txt.
58. Microsoft. *Overview of Entity Framework Core - EF Core* [online]. 2021-05. [cit. 2024-04-22]. Dostupné z: https://learn.microsoft.com/en-us/ef/core/.
59. Berners-Lee, T.; Connolly, D. *Hypertext Markup Language - 2.0* [Internet Requests for Comments]. RFC Editor, 1995-11. RFC, 1866. RFC Editor. issn 2070-1721.
60. Network, Mozilla Developer. *JavaScript* [online]. [cit. 2024-04-22]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript.
61. Appel, R. *ASP.NET Core SignalR JavaScript client* [online]. 2022-06. [cit. 2024-04-22]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/signalr/javascript-client?view=aspnetcore-8.0&tabs=visual-studio.
62. Network, Mozilla Developer. *Introduction to the DOM - Web APIs | MDN* [online]. [cit. 2024-04-22]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction.
63. R. Anderson D. Brock, K. Larkin. *Introduction to Razor Pages in ASP.NET Core | Microsoft Learn* [online]. 2023-10. [cit. 2024-04-22]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/razor-pages/?view=aspnetcore-8.0&tabs=visual-studio.
64. R. Anderson T. Mullen, D. Vicarel. *Razor syntax reference for ASP.NET Core* [online]. 2023-11. [cit. 2024-04-22]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/mvc/views/razor?view=aspnetcore-8.0.
65. clientIO. *JointJS - JavaScript diagramming library powering exceptional UIs* [online]. 2024-03. [cit. 2024-04-23]. Dostupné z: https://github.com/clientIO/joint/blob/master/packages/joint-core/README.md.
66. Microsoft. *TypeScript: JavaScript With Syntax For Types* [online]. [cit. 2024-04-23]. Dostupné z: https://www.typescriptlang.org/.
67. Goldberg, J. *Using the Compiler API · TypeScript Wiki* [online]. 2024-10. [cit. 2024-04-23]. Dostupné z: https://github.com/Microsoft/TypeScript/wiki/Using-the-Compiler-API.
68. Microsoft. *Type Declarations* [online]. 2021-09. [cit. 2024-04-23]. Dostupné z: https://microsoft.github.io/TypeScript-New-Handbook/chapters/type-declarations/.
69. npm, Inc. *npm/cli: the package manager for JavaScript* [online]. 2023-09. [cit. 2024-05-08]. Dostupné z: https://github.com/npm/cli.

70. GABRIEL T., Marian Villa. *The Basics of Package.json - NodeSource* [online]. 2022-02. [cit. 2024-05-08]. Dostupné z: https://nodesource.com/blog/the-basics-of-package-json/.
71. WEBPACK. *Getting Started | webpack* [online]. [cit. 2024-05-08]. Dostupné z: https://webpack.js.org/guides/getting-started/.
72. WEBPACK. *Development | using webpack dev server* [online]. [cit. 2024-05-08]. Dostupné z: https://webpack.js.org/guides/development/#using-webpack-dev-server.
73. MICROSOFT. *ASP.NET Core Blazor JavaScript interoperability (JS interop)* [online]. 2024-04. [cit. 2024-05-08]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/blazor/javascript-interoperability/.
74. KUBERNETES. *Ingress* [online]. 2024-04. [cit. 2024-04-27]. Dostupné z: https://kubernetes.io/docs/concepts/services-networking/ingress/.
75. KUBERNETES. *Kubernetes Overview* [online]. 2023-09. [cit. 2024-04-25]. Dostupné z: https://kubernetes.io/docs/concepts/overview/.
76. OPEN CONTAINER INITIATIVE. *The OpenContainers Image Spec* [online]. 2024-04. [cit. 2024-04-25]. Dostupné z: https://specs.opencontainers.org/image-spec/.
77. KUBERNETES. *Nodes* [online]. 2024-01. [cit. 2024-04-25]. Dostupné z: https://kubernetes.io/docs/concepts/architecture/nodes/.
78. NGINX. *NGINX Ingress Controller* [online]. [cit. 2024-04-25]. Dostupné z: https://docs.nginx.com/nginx-ingress-controller/overview/about/.
79. KUBERNETES-CLIENT. *Officially supported dotnet Kubernetes Client library* [online]. [cit. 2024-05-09]. Dostupné z: https://github.com/kubernetes-client/csharp.
80. JASMINE. *Jasmine Documentation* [online]. [cit. 2024-05-09]. Dostupné z: https://jasmine.github.io/.
81. C. POOLE, R. Prouse. *NUnit* [online]. 2024. [cit. 2024-05-09]. Dostupné z: https://nunit.org/.
82. JOVANOVIC, M. *How To Approach Clean Architecture Folder Structure* [online]. 2024-09. [cit. 2024-07-16]. Dostupné z: https://www.milanjovanovic.tech/blog/clean-architecture-folder-structure.

Seznam obrázků

2.1 Gartner Data integration Magic Quadrant, zdroj: Gartner, 2023 [4] 12
2.2 Mapování sloupců v SSIS, zdroj: Patel Bhavesh, 2017 [12] 13
2.3 SSIS Designer integračních balíčků, zdroj: Patel Bhavesh, 2017 . . 14
2.4 Prostředí nástroje SSIS Designer, zdroj: Microsoft, 2023 14
2.5 Prostředí nástroje Talend Open Studio – Job Designer 16
2.6 Mapování sloupců v Talend Open Studiu 17
2.7 Gartner BI Magic Quadrant, zdroj: Gartner, 2021 19
2.8 Prostředí Power BI – vytváření nástěnky, zdroj: Microsoft, 2023 . 20
2.9 Prostředí nástroje Tableau, zdroj: Talend, 2022 21
2.10 Prostředí nástroje Metabase 22

3.1 Diagram případů užití systému správcem 28
3.2 Diagram případů užití systému zákazníkem 29

4.1 C4 diagram zachycující kontext systému 34
4.2 C4 kontejnerový diagram systému 35
4.3 C4 diagram komponent webové platformy 37
4.4 C4 kontejnerový diagram *klastru* 39
4.5 C4 diagram nasazení systému 40
4.6 Návrh prostředí nástroje Mapper 41
4.7 Výběr další tabulky v nástroji Mapper 42
4.8 Definice výrazu pro připojení další tabulky v nástroji Mapper . . 42
4.9 Mapování po přidání druhé zdrojové tabulky 43
4.10 Přidání další komponenty do mapování 43
4.11 Mapování s výchozí hodnotou 44
4.12 Přehled zákazníků ve správní aplikaci. 44
4.13 Detail zákazníka ve správní aplikaci a možnost smazání jeho účtu. 45
4.14 Přehled mapovacích projektů ve správní aplikaci. 46
4.15 Potvrzení emailu a nastavení hesla u nového zákazníka ve správní aplikaci. 46
4.16 Konfigurace údajů potřebných pro připojení k zákaznické databázi. 47
4.17 Integrace mapovacího nástroje a jeho použití pro mapování zákaznických dat. 47
4.18 Stránka pro nasazení databázových pohledů a instance nástroje Metabase 48

6.1 UML diagram generického modelu 57
6.2 Předkonfigurovaný Metabase – základní nástěnka 57
6.3 Model databáze reprezentovaný C# třídami 59
6.4 Model mapování relačních tabulek 60
6.5 Zjednodušená struktura zdrojových souborů mapovacího nástroje *Mapper*. 61
6.6 Zjednodušená struktura zdrojových souborů *SaaSPlatform*. 64
6.7 Zjednodušená struktura zdrojových souborů aplikace *ManagementApp*. 65
6.8 Ukázka mapování v Mapper 71

Seznam tabulek

2.1 Porovnání důležitých vlastností nástrojů pro integraci dat 18
2.2 Porovnání důležitých vlastností nástrojů pro business intelligence 23

4.1 Mapování funkčních požadavků správce systému na moduly systému 38
4.2 Mapování funkčních požadavků zákazníka na moduly systému . . 38

Seznam použitých zkratek

SW	Software
UI	Uživatelské rozhraní (anglicky *User Interface*)
UX	User experience
ETL	extrakce, transformace a nahrání (anglicky *Extract, Transform and Load*)
ELT	extrakce, nahrání a transformace (anglicky *Extract, Load and Transform*)
SSR	Server-side rendering (někdy překládáno jako *skriptování na straně serveru*)
DI	Dependency Injection

A Přílohy

A.1 Spuštění

Pro spuštění správní aplikace je potřebný přístup k databázi sloužící pro uchovávání dat správní aplikace, přístup k poštovnímu *SMTP* serveru a přístup ke Kubernetes klastru.

Databáze správní aplikace navíc musí mít vytvořené používané tabulky a základní data jako definice rolí z modulu *Users* nebo model cílové databáze z modulu *Dataintegration*. Potřebnou inicializaci databáze zajišťuje samotná knihovna *Entity Framework Core*[1].

Přístupové údaje definujeme pomocí konfigurace aplikace. Konfigurace se nachází v souboru *appsettings.json* v projektu *ManagementApp* či je předána pomocí proměnných prostředí[2].

Modul *Notifications* potřebuje přístupové údaje k poštovnímu serveru a základní adresu URL, ze které je správní aplikace dostupná. Modul *Users*, potřebuje jméno a e-mailovou adresu prvního správce systému, aby mu mohla vytvořit uživatelský účet. Modul *Deployment* potřebuje adresu, ze které je přístupný Kubernetes klastr, také přístupové údaje k poštovnímu serveru, aby je mohl předat instancím nástroje Metabase při konfiguraci.

Pro účely vývoje a ukázky je možné správní aplikaci spustit pomocí Docker Compose, což jsme definovali jako jeden z nefunkčních požadavků v podsekci 3.2.2. Soubor *docker-compose.yaml* definuje všechny závislosti jako propojené Docker kontejnery. Navíc přidává dvě instance MS-SQL databází představující databáze zákazníků

S využitím vývojového prostředí si můžeme představit kompletní průběh užití aplikace krok po kroku.

Spuštění aplikace

Pokud chcete aplikaci spustit na Windows, je třeba před klonováním repozitáře nastavit git tak, aby nekonvertoval LF na CRLF (`git config --global core.autocrlf false`), jinak dojde k narušení shellových skriptů.

Spusťte v kořenovém adresáři repositáře:

```
docker compose up --build
```

To může trvat několik minut, ale ne více než 20, v závislosti na výkonu vašeho počítače a rychlosti internetového připojení.

Počkejte, až bude kontejner `server-1` spuštěn, což je server, který čeká, až budou připraveny všechny ostatní kontejnery. Kvůli velké velikosti příkladových dat zákazníků (kontejnery `customer-db-1-1` a `customer-db-2-1`), trvá inicializace příkladových databází poměrně dlouho.

[1]Více na https://learn.microsoft.com/en-us/ef/core/managing-schemas/migrations/.

[2]Více na https://learn.microsoft.com/en-us/aspnet/core/fundamentals/configuration/.

Vytvoření prvního zákazníka a nasazení Metabase

Admin:
email: `admin@admin.cz`
heslo: `Admin123*`

customer1:
email: `customer@example.cz`
heslo: `Customer123*`

Otevřete `http://localhost:8080/`

1. Přihlaste se jako Admin, přejděte na Users -> Customers -> Add customer a zadejte údaje zákazníka (např. `customer@example.cz` a `customer1` jako jméno)

2. Odhlaste se a přejděte na `http://localhost:5080/`, abyste viděli odeslané e-maily.

3. Otevřete zvací e-mail a klikněte na zvací odkaz.

4. Zadejte heslo, např. `Customer123*`, a klikněte na `Set password`. Nyní byste měli být přihlášeni jako zákazník.

5. Přejděte na Data Integration -> Database connection configuration a vyplňte následující údaje:

```
Database connection configuration:
    Select your database provider: SqlServer
    Data Source: 10.5.0.3,1433
    Initial Catalog: CostumerExampleData
    User name: sa     # must be entered
    Password: password123!
    Connect Timeout: 30
    Encrypt: false
    Trust Server Certificate: true
    Multi Subnet Failover: false
```

7. Klikněte na Save, měli byste vidět zprávu: Connection established successfully

8. Klikněte na `Create model`, měli byste vidět zprávu: Model created successfully.

9. Přejděte na Data Integration -> Mapper, měli byste vidět zelenou zprávu: Please map the target entities one by one, don't forget to save your progress after each entity mapped. Fully mapped entities are marked with green flag, otherwise red cross. After mapping all entities and saving each mapping, Continue button will appear

10. Pokud chcete, zprávu zavřete.

11. Přesuňte entitu Workplaces na pravou stranu přetažením jejího záhlaví.

12. Klikněte na `Add source table` a vyberte `TabCisZam`, klikněte na `Continue`.

13. Namapujte sloupce z `TabCisZam` na sloupce cílové tabulky `Employees` kliknutím na sloupce zdrojové tabulky a přetažením na cílové sloupce:

    ```
    Mapování:
        ID -> Id
        Cislo -> PersonalID
        ID -> ExternalId
        Jmeno -> FirstName
        Prijmeni -> Lastname
    ```

14. Dvojklikem na `TabCisZam` zrušte výběr nepoužitých sloupců (volitelné).

15. Klikněte na `Save`, měli byste vidět: Mapping was saved.

16. Klikněte na výběrový vstup `dbo.Employees`, vyberte `dbo.Workplaces`. Měli byste vidět entitu cílové tabulky `Workplaces`.

17. Klikněte na `Vybrat soubor`.

18. Vyberte `{RepositoryRoot}/ExampleMappings/WorkPlacesMapping.json`

19. Měli byste vidět mapování Workplaces, klikněte na save.

20. Pokračujte na `dbo.WorkReports`

21. Přidejte zdrojovou tabulku `TabPrikazMzdyAZmetky`.

22. Namapujte sloupce:

    ```
    Mapování:
        ID -> ID
        IDPrikaz -> OrderId
        DokladPrPostup -> ProductionOperationId
        IDPracoviste -> WorkPlaceId
        kusy_odv -> Quantity
        Zamestnanec -> WorkerId
        Nor_cas -> ExpectedTime
        datum -> DateTime
    ```

23. Klikněte na `Add custom query`

24. Vyplňte podrobnosti vlastního dotazu:

    ```
    Unique name: WorkReportTypeCustomQuery
    Query:
    ```

```
SELECT [TabPrikazMzdyAZmetky].ID as TabPrikazMzdyAZmetkyID,
CAST(
    CASE
    -- Forbidden prefixes for [TabKmenZbozi].RegCis
    -- "17", "18", "37", "38"
    -- and [TabKmenZbozi_EXT]._pracoviste_filtr equals one of:
        -- "R", "R+cell", "R+podia", "T+R",
    -- "R+Přípraváři"
    WHEN ([TabKmenZbozi].RegCis LIKE '17%'
        OR [TabKmenZbozi].RegCis LIKE '18%'
        OR [TabKmenZbozi].RegCis LIKE '37%'
        OR [TabKmenZbozi].RegCis LIKE '38%')
        AND [TabKmenZbozi_EXT]._pracoviste_filtr IN
            ('R', 'R+cell', 'R+podia', 'T+R', 'R+Přípraváři')
    THEN 'Type1'
    -- Forbidden prefixes for [TabKmenZbozi].RegCis
        -- "17", "18", "37", "38"
    -- and [TabKmenZbozi_EXT]._pracoviste_filtr equals one of:
        -- "TR", "TR+podia", "T+R", "TR+Přípraváři"
    WHEN ([TabKmenZbozi].RegCis LIKE '17%'
        OR [TabKmenZbozi].RegCis LIKE '18%'
        OR [TabKmenZbozi].RegCis LIKE '37%'
        OR [TabKmenZbozi].RegCis LIKE '38%')
        AND [TabKmenZbozi_EXT]._pracoviste_filtr IN
            ('TR', 'TR+podia', 'T+R', 'TR+Přípraváři')
    THEN 'Type2' ELSE 'Other' END AS nvarchar(100)
) AS ProductType
FROM [CostumerExampleData].[dbo].[TabPrikazMzdyAZmetky]
LEFT JOIN [TabKmenZbozi]
ON TabPrikazMzdyAZmetky.IDTabKmen =
    [TabKmenZbozi].ID LEFT JOIN [TabKmenZbozi_EXT]
    ON TabPrikazMzdyAZmetky.IDTabKmen = [TabKmenZbozi_EXT].ID
```

```
Selected columns:
    name: TabPrikazMzdyAZmetkyID
    type: Integer
    Is nullable: false

    name: ProductType
    type: nvchar(length)
    Is nullable: false
    length: 100
```

25. Klikněte na `Continue`, mělo by se zobrazit modální okno `Join definition`

26. Zadejte podrobnosti spojení:

```
Left column: TabPrikazMzdyAZmetky_ID
Operator: Equals
```

```
Right column:
    WorkReportTypeCustomQuery_TabPrikazMzdyAZmetkyID
```

27. Klikněte na `Continue`, zdrojová entita `WorkReportTypeCustomQuery` by se měla objevit.
28. Přetáhněte entitu na volné místo.
29. Namapujte sloupec zdrojové entity *CustomQuery* `ProductType` na sloupec cíle `ProductType`.
30. Klikněte na `Save`, nyní by mělo být mapování dokončeno a v horní části stránky by se mělo zobrazit tlačítko `Continue to deployment`.
31. Klikněte na `Continue to deployment`. Měli byste být přesměrováni na stránku `http://localhost:8080/Deployment/DeployMetabase`
32. (volitelné) Klikněte na `Generate Views`, abyste viděli, jaké SQL příkazy budou provedeny pro nasazení databázových pohledů.
33. Klikněte na `Automatically deploy views`, měli byste vidět zprávu: `Views deployed successfully.`
34. Zadejte přihlašovací údaje správce Metabase, náhodný e-mail a silné heslo, např.:

    ```
    Email used to log in to metabase: customer@example.cz
    Password: Customer123*
    ```

34. Klikněte na `Deploy Metabase`, měli byste vidět točící se ikonu a zprávu: This could take several minutes. You can close this page and you will be notified when the metabase is successfully deployed. Tato operace by neměla trvat déle než 10 minut, pro první nasazení je třeba stáhnout asi 1 GB.
35. Buď počkejte, až se točící se ikona přestane točit, nebo můžete průběžně kontrolovat `http://localhost:5080/` pro e-mail s odkazem
36. Pokračujte odkazem na Metabase a použijte zadané přihlašovací údaje správce Metabase pro přihlášení. Po přihlášení byste měli vidět nástěnku s grafy. Pokud se nic nezobrazuje, zkuste stránku znovunačíst.
37. Odtud můžete začít používat vlastní předkonfigurovanou instanci Metabase

Nasazení druhé instance Metabase pro druhého zákazníka

Proces je stejný, mění jen zdroj dat v `Database connection configuration`:

```
Database connection configuration 2:
    Select your database provider: SqlServer
    Data Source: 10.5.0.7,1433
    Initial Catalog: CostumerExampleData
    User name: sa     # must be entered
```

```
Password: password123!
Connect Timeout: 30
Encrypt: false
Trust Server Certificate: true
Multi Subnet Failover: false
```

Nasazení druhé instance Metabase by nemělo trvat déle než 2 minuty. Průměrně jednu minutu.

Smazání zákazníka

1. Přihlaste se jako admin.
2. Přejděte na Users -> Customers a klikněte na ID zákazníka, kterého chcete smazat.
3. Klikněte na `Delete` a potvrďte svůj záměr kliknutím na `Confirm`.
4. Měli byste vidět, že zákazník již není v seznamu a během několika sekund se jeho instance Metabase, pokud nějakou měl, stane nedostupnou.

A.2 Obsah elektronických příloh

V přiloženém souboru `ZIP` nalezneme soubor `Readme.md` popisující projekt a jeho spuštění, dále `Dockerfile` pro vytvoření Docker Image správní aplikace a soubor `docker-compose.yaml`, který slouží pro orchestraci vývojového prostředí popsaného v příloze A.1. Zbytek souborů jsme si již představili v kapitole 6.