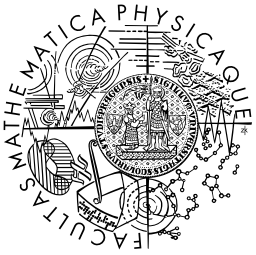


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Michal Půlpán

REST API pro SAP Business One a jeho využití v praxi

Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: doc. RNDr. Jan Kofroň, Ph.D.
Studijní program: Informatika
Studijní obor: Softwarové a datové inženýrství

Praha 2021

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

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 chci mnohokrát poděkovat vedoucímu práce panu doc. RNDr. Janu Kofroňovi, Ph.D. za všechny rady a podporu, kterou jsem od něho během posledních tří semestrů získal. Velmi si vážím veškerého času, který s tímto projektem, a hlavně se mnou, strávil. Taktéž patří veliký dík mému otci, který mi byl velikou oporou a pomáhal mi se správným návrhem celého projektu. Mé matce a sestře Veronice děkuji za veškerou podporu při psaní práce a následných, velmi potřebných, jazykových a stylistických korekturách.

Název práce: REST API pro SAP Business One a jeho využití v praxi

Autor: Michal Půlpán

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: doc. RNDr. Jan Kofroň, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Cílem této práce je vytvořit dva projekty. Prvním z nich bude tvorba REST API umožňující jednodušší komunikaci externích aplikací se SAP Business One. Druhým bude tvorba webové aplikace sloužící jako využití vytvořeného API v praxi. Ta bude zajišťovat předávání a získávání data od přepravních společností (Zásilkovna, Česká Pošta a v případě potřeby dalších) a prohlížet záznamy o zásilkách dle různých filtrů v uživatelsky přívětivém prostředí. Celá práce se bude zaobírat nasazením v jedné konkrétní firmě. Naše API bude implementováno pomocí webového frameworku Flask a na základě požadavků firmy bylo napojení na SAP zajištěno prostřednictvím SAP DI API. Aplikace pro komunikaci s přepravními společnostmi naopak bude implementováno pomocí frameworku Django a doprovodných technologií (HTML, CSS a JavaScript). Oba projekty k odevzdání této práce byly několik měsíců nasazeny v produkčním prostředí (za využití technologie Docker) a využívány jejich uživateli.

Klíčová slova: REST API SAP Business One webová aplikace Packeta Ceska Posta Flask Django

Title: REST API for SAP Business One in practice

Author: Michal Půlpán

Department: Department of Distributed and Dependable Systems

Supervisor: doc. RNDr. Jan Kofroň, Ph.D., Department of Distributed and Dependable Systems

Abstract: The aim of this thesis is to develop two projects. The first project is development of REST API allowing simpler and more comfortable communication of external applications and SAP Business One. The other project is development of web application serving as a practical implementation of the API. It will serve as a communication layer for acquisition and transmission of data between one particular company and shipping companies (Packeta, Czech Post and, if necessary, others) allowing for simple data preview about parcels with different filters in a user friendly interface. Whole project will be deployed in one particular company. Our API will be implemented using web framework Flask and, based on requirements of the company, SAP connection will be established using SAP DI API. Application for communication with shipping companies will be implemented using framework Django in conjunction with other technologies (HTML, CSS and JavaScript). Both projects for this thesis were deployed in a production environment for several months (using Docker technology) and used by their target users.

Keywords: REST API SAP Business One web application Packeta Ceska Posta Flask Django

Obsah

1. Úvod

Pro vývoj jakéhokoli software ve firmě je z ekonomických i praktických důvodů nezbytné mít rychlý a na použití jednoduchý způsob, jak přistoupit k interním datům v informačním systému. V kontextu této práce se bude jednat o SAP Business One. To, co může uživatel či programátor chtít získat, jsou například data k nabízeným produktům, zakázkách či jiným informacím, se kterými chce pracovat. Pokud takovou možnost nemáme, je nutné, aby programátor při tvorbě jakéhokoli programu vytvořil řešení vlastní. A to buď sofistikovanější, ale pravděpodobně nákladnější, kdy využije nějaké cesty nabízené dodavatelem konkrétního informačního systému, a nebo řešení využívající přímé komunikace s interní databází. To není vždy úplně bezpečné, jak z hlediska ochrany databáze, kdy zbytečně vystavujeme přístupové údaje k ní, tak i vzhledem k samotným datům. Takový přístup však v některých situacích může znamenat i porušení záručních podmínek dodavatele informačního systému a následnou ztrátu jakékoli podpory. Nicméně přístup, kdy se pro každý firemní program řeší vlastní napojení, ať už prvním či druhým způsobem, není optimální.

Cílem této bakalářské práce je navrhnout a vyvinout dvě aplikace dle požadavků zadavatelské firmy, kterou pro jednoduchost budeme značit *Firma*. Jedná se o českou společnost založenou v roce 1993, jenž se v průběhu let stala největším výrobcem a dodavatelem spodního prádla v České republice. Zhruba okolo roku 2007 byl ve *Firmě* zaveden SAP Business One plnící funkci takzvaného *Enterprise resource planning (ERP) software*, který sdružuje podniková data o nabízených produktech, jejich skladové dostupnosti, zákaznících, objednávkách a více. *Firma* funguje jak na maloobchodní, tak i velkoobchodní bázi. Je tedy nutné, aby *Firma*, ale i její odběratelé, měli možnost se zavedeným SAP pohodlně komunikovat.

Mějme dva uživatele, kteří si budou potřebovat vyměňovat se SAP informace. Popišme to následujícími scénáři:

(i) *Firma* potřebuje pro její internetové podnikání získávat data o nabízených produktech. Těmi daty by měly být: skladová dostupnost, zařazení do konkrétní produktové kategorie, vícejazyčné názvy a ceny pro dané skupiny uživatelů. Touto formou je možné naplnit daty jak velkoobchodní, tak i maloobchodní internetový obchod.

(ii) *Firma* potřebuje získávat data o vytvořených objednávkách, aby mohla jednoduše tyto informace předávat přepravním společnostem, které zásilku doručí zákazníkovi.

(iii) *Firma* potřebuje zapisovat data o nově (i dříve) odeslaných objednávkách jako je například sledovací číslo zásilky a její aktuální stav dle daného přepravce.

(iv) *Firma* potřebuje mít možnost zapisovat informace o nových objednávkách formou předběžných dokladů[1].

[1]Předběžný doklad můžeme pro naše účely považovat za objednávku. Jedná se totiž o „předstupeň“ objednávky v procesu fakturace, kdy jsou údaje dokladu kontrolovány v rámci interního procesu.

(v) Společně s tím, kdy *Firma* vytvoří objednávku, potřebuje i do SAP zapsat informace o zákazníkovi[2], který objednávku vytvořil.

(vi) Velkoobchodní odběratel *Firmy* potřebuje získávat informace o produktech, které může v rámci maloobchodního prodeje nabízet svým zákazníkům. Mělo by se jednat o skladovou dostupnost, potřebná data z karty produktu jako je složení, barva, velikost, název, popis, kategorie a samozřejmě i doporučená maloobchodní cena.

(vii) Velkoobchodní odběratel *Firmy* fungující na bázi toho, že například přes internetový obchod nabízí produkty *Firmy* formou dropshipping[3], kdy využívá služby nabízené *Firmou* v podobě skladování a přímé distribuce zboží k zákazníkovi odběratele. Ten tak potřebuje zapsat informace o vytvořené objednávce a samotném zákazníkovi kvůli adrese, na kterou má být zboží doručeno.

Obecným cílem aplikací je usnadnit práci s interními daty. První je REST API pro bezpečnou a rychlou komunikaci se SAP Business One. To bude využíváno jak pro potřeby interní, například pro získávání a aktualizaci dat o objednávkách a zákaznících, tak i externí, jako například dropshipping internetové obchody, jenž využijí možností jako zápis nové objednávky a možnost získat data o produktech - například skladovou dostupnost.

Druhou částí této práce je vývoj aplikace využívající zmíněné REST API v praxi. Jedná se o program pro odesílání dat o objednávkách k přepravním společnostem jako je Česká Pošta, Zásilkovna a v budoucnu i PPL. Generování štítků na zásilky a průběžné aktualizace stavu při přepravě objednávky. Program by měl zaměstnancům výrazně ulehčit jejich práci a integrovat data do jednoho prostředí, které budou při práci využívat.

1.1 Složitý přístup k datům

Jednou z hlavních motivací vývoje vlastního API byla skutečnost, že se *Firma* velice často potýkala s potřebou poskytovat, ale i získávat informace od třetích stran. Ať už se jednalo o veškerá data k zakázkám z internetových obchodů obsahující kompletní nákupní košík, dodací, fakturační i kontaktní údaje, a nebo k již zmíněným zásilkám připraveným pro přepravní společnosti. Procesy byly velice časově náročné a přispívaly ke vzniku lidských chyb, kterým se nešlo vyvarovat, ať už se jednalo o chyby z nepozornosti či nesprávného dodržení postupu.

1.1.1 Složité exporty dat

Problémem zmíněným v sekci 1.1 byla jakákoli komunikace s externími službami. Když totiž *Firma* potřebovala získat informace, například o svých objednávkách, či produktech pro třetí stranu, nebylo možné tyto operace jakkoli „centrálně“ automatizovaně řešit a data tak přímo předat jejich zájemci, aniž by musel jakkoli zasahovat příslušný zaměstnanec. Pokaždé bylo nutné, aby *Firma*

[2] V terminologii SAP se jedná o tzv. obchodního partnera resp. Business Partner

[3] Dropshipping je forma maloobchodního prodeje, při kterém prodejce funguje pouze jako prostředník mezi zákazníkem a jeho velkoobchodním dodavatelem

vytvořila proces pro extrakci dat v potřebném formátu, zaměstnanec úlohu spustil (pokud se nejednalo o časovanou úlohu) a předal data třetí straně. Získat tak informace z databáze až tak veliký problém nebyl. Avšak bylo téměř nemožné nějakým elegantním způsobem data předat třetí straně, aniž by k tomu nebyla potřeba netriviální práce zaměstnance. Například obsluha ve skladu, která se denně věnovala nahrávání dat k přepravním společnostem, tak byla nucena skončit s přípravou zásilek výrazně dříve, než by bylo nutné, a věnovat se ručnímu importu informací o zásilkách v datovém formátu např. CSV. Po importu musela nastat opětovná ruční kontrola, kdy bylo nutné ověřit, zda-li se všechny zásilky nahrály ke konkretnímu přepravci. Poté musela ručně exportovat přiřazená sledovací čísla, která se následně nahrají, opět ručním importem do SAP. Toto se však opakovalo tolikrát, kolik v danou chvíli *Firma* využívala přepravních společností. A obsluha tak musela pracovat s několika různými prostředími, která zpravidla poskytovala různou funkcionalitu, či v lepším případě byly pouze klíčové ovládací prvky odlišně rozmístěné po uživatelském prostředí.

Není to samozřejmě problém nastavených procesů zaměstnanců *Firmy*, kteří by si z vlastního zájmu ztěžovali práci, ale faktu, že není možné jednoduchým způsobem propojit SAP s externí služnou resp. libovolným „příjemcem“ dat.

1.1.2 Problémy s napojením na externí služby

Sekce 1.1.1 a samotný úvod práce již napovídá, že se *Firma* potýkala i s problémy napojení na externí služby. Pokud totiž měl odběratel získat informace o produktech, které měl zájem nabízet na svém internetovém obchodě, obvyklým řešením byla tvorba XML feedu, ke kterému dostal odběratel přístup. Nicméně sdílení odkazu bylo prakticky nekontrolovatelné a škálovatelnost tohoto řešení byla téměř nemožná. Druhým hlavním problémem byl i zápis dat do databáze, například o nově příchozí objednávce do SAP Business One. Tato operace byla možná pouze velice zastaralou a ne příliš uživatelsky přívětivou cestou.

1.2 Cíl práce

Samotným cílem práce je vyhotovení a nasazení dvou téměř izolovaných aplikací. Jednou z nich bude zmíněné API čerpající a zapisující data ze SAP Business One. Druhou aplikací bude můstek s přívětivým uživatelským rozhraním mezi *Firmou* a přepravními společnostmi, která umožní jednoduchý a moderní přehled odešlých objednávek a jejich průběžnou kontrolu. Odboural se tím tak nutnost, aby obsluha ve skladu pracovala s tolika prostředími pro nahrávání objednávek, kolik je přepravních společností. Mimo to, že můstek výrazně ulehčí práci zaměstnancům, bude i sloužit jako příklad využití vytvořeného API.

1.2.1 REST API

Abychom mohli s daty pracovat, je nutné vytvořit prostředí pro komunikaci mezi *Firmou* a uživatelem, či aplikací, která má být na *Firmu* napojena. REST API vytvořené v této práci bude ve své základní verzi umožňovat zápis dat o zákaznících a objednávkách. Získat informace skrze API bude možné například k dostupnosti produktů na centrálním skladě, základní informace o produktové

skupině, ve které se nacházejí a nebo jejich detailní karty s názvy v různých jazykových mutacích a odkazy na produktové obrázky. Všechny uživatelské funkce API jsou rozebrány v kapitole 3.1.1. API tak *Firmě* umožní i rozvoj v oblasti internetového podnikání ve formě B2C[4] a nebo již zmíněného dropshippingu, kdy bude externí prodejce schopný přímo ze svého internetového obchodu zapisovat přijaté objednávky do SAP a získávat přehled o skladové dostupnosti produktů a podobných informacích. API bude z programátorského hlediska rozdělené na dvě závislé komponenty. Hlavní z nich pojmenujeme *master aplikace*. Ta bude izolovaně, v rámci firemní sítě, řešit komunikaci s Microsoft SQL databází a instancí SAP Business One. Uživatel však nebude komunikovat přímo s *master aplikací*, ale s (v naší terminologii), *uživatelskou aplikací*. Ta bude „veřejně“ dostupná bude zajišťovat veškerou autentizaci. *Uživatelská aplikace* tak bude jednoduše po ověření uživatele a kontroly jeho práv na požadovanou funkcionalitu předávat požadavek *master aplikaci*, jejíž odpověď vrátí v jednotné přehledné struktuře. Bude plnit funkci určité „proxy“.

Řešení tohoto API je sice silně ovlivněné potřebami a způsobem jakým *Firma* využívá SAP, avšak při splnění všech požadavků, které si představíme v dalších částech práce, je toto řešení do jisté míry obecné a aplikovatelné. Pokud bychom v projektu nepracovali s uživatelskými tabulkami a poli, aplikace by měla být přenositelná.

1.2.2 Aplikace pro komunikaci s přepravci

Již sekce 1.1.2 mírně poukazovala na podobu aktuálního stavu procesů přenosu dat mezi *Firmou* a například Českou Poštou. Složitost, a s tím i spojená zvýšená chybovost procesu pro obsluhu s rapidním nárůstem objednávek v prvním kvartálu roku 2020, ukázala, že se jedná o dlouhodobě neudržitelné řešení. Nejednalo se jen o maloobchodní internetové objednávky, které se napříč e-shopy v České republice pohybovaly na úrovni předvánočního období s dvojciferným růstem proti srovnatelnému období roku 2019, jak je zmíněno v článku [2]. V tomto konkrétním případě se však jednalo i o návazný zvýšený nárůst tuzemských i zahraničních velkoobchodních objednávek. Je proto nutné přijít s řešením, které by práci zaměstnancům částečně zautomatizovalo a bylo tak možné pohodlně a rychle přistupovat k objednávkám. Současně potřebujeme sledovat aktuální stavy zásilek během přepravy, aniž by se obsluha musela přihlašovat do několika různých webových prostředí a zabývat se zdlouhavými importy a exporty datových souborů. Požadavek na aplikaci je, aby byla jednoduše rozšiřitelná o další přepravní služby, neboť se jejich nabídka ze strany *Firmy* v čase mění. V této práci bude rozebráno napojení na API České Pošty, Packeta. Součástí práce je i napojení na přepravce Uloženka (dnes již WE|DO), avšak *Firma* s přepravcem rozvázala spolupráci, a tak napojení není více rozebráno.

[4] Z anglického „Business-to-Customer“, kdy velkobchod prodává přímo zákazníkovi. Obdoba zažitého konceptu B2B (neboli „Business-To-Business“), kde z celého procesu odpadá „prostředník“ ve formě dalšího prodejce.

2. Analýza problému

Jak vyplývá z úvodní kapitoly, *Firma* se potýká s problémem, jehož řešení není tak jednoduché, jak by se v první chvíli mohlo zdát. Stojí totiž v situaci, kdy každý rozvoj, ať už v oblasti internetového podnikání či vlastních interních systémů, je velice nákladný. Mimo vývoje hlavního produktu se musí vždy programátoři zaměřit i na napojení do zavedeného firemního ERP řešení, SAP Business One [3]. Což mělo za následek velice ztížené výběrové řízení při tvorbě nového software. Zpravidla drtivá část firem, která měla o zakázku vývoje nějaké poptávané aplikace zájem, se již při čtení zadání zalekla napojování na SAP. Ti, kteří přetrvali, byli velice nákladní a dlouhodobě se *Firmě* spolupráce nevyplácela. Řešení SAP Business One je velikým přínosem v byznys odvětví. Nicméně, jakmile přijde na řešení nějakého vývoje komplexnějších aplikací, zpravidla je implementace SAP ve *Firmě* nemalou překážkou. V této kapitole si detailně popíšeme problémy a procesy, které přímo i nepřímo souvisely s tím, že rozvoj systémů byl, díky zavedenému ERP řešení, velmi nákladný a v určitých situacích i naprosto nerentabilní.

2.1 Získání a zápis dat

V první chvíli se nezasvěcenému čtenáři může zdát, že řešíme již vyřešený či velice jednoduchý problém. Vzhledem k tomu, že SAP Business One ve verzi, ve které je v době vzniku této práce nasazený, potřebuje pro svou práci a uchování dat instanci databáze Microsoft SQL.

2.1.1 Přímé připojení na Microsoft SQL server

Popis

Mělo by nám tedy z naprosto logické a plně opodstatnitelné úvahy stačit každou vyvíjenou aplikaci obohatit o Microsoft SQL konektor do databáze. Takovému konektoru se zpravidla říká ODBC *Open Database Connectivity* [4]. Jedná se standardizované API pro přístup k datům SQL serveru. Takový ovladač, resp. konektor, pro MS SQL společnost Microsoft poskytuje pro většinu operačních systémů. Těmi jsou Linux, macOS a samozřejmě Windows s podporou velikého množství programovacích jazyků. Jmenovitě:

- **C#**[5] Jazyk z platformy .NET.
- **C++** [6] Původně rozšířením jazyka C s podporou několika programovacích stylů.
- **PHP** [7] Skriptovací programovací jazyk především pro tvorbu dynamických webových aplikací.
- **Python**[8] Skriptovací programovací jazyk, který se v posledních letech zařadil na první místa v popularitě.
- **Perl**[9] Velice benevolentní dynamicky typovaný jazyk.

Přímé napojení Microsoft SQL databáze na aplikaci by technický problém nebyl.

Problém tohoto řešení

Jedinou překážkou v technickém návrhu mohl být fakt, že databáze není z bezpečnostních důvodů přístupná k připojení mimo vnitřní firemní síť. To by bylo možné vyřešit připojením přes tunel, který by mohl k tomuto účelu sloužit. Nicméně, toto řešení přináší hned několik problémů. Vznikla by obtíž se správou velikého množství uživatelů databáze a kontroly jejich práv. Druhý problém, který toto řešení přináší, je potencionální bezpečnostní slabina využitelná k útoku za účelem vniknutí do vnitřní sítě. Doposud jsme však nezmínili velice důležitou stránku věci. SAP Business One nedovoluje žádnou přímou manipulaci s databází skrze SQL dotazy. V praxi by se toto řešení při vytváření interního software nabízelo, ale zákazník tím porušuje podmínky stanovené firmou SAP.

2.1.2 Oficiální řešení SAP

Variantu, kdy by se každý program přímo připojoval do databáze, jsme zavrhli hned z několika důvodů. Jednalo by se o možné narušení bezpečnosti sítě *Firmy*, vznikl by veliký tlak na údržbu uživatelských účtů databáze a SAP jako takový nepovoluje svým uživatelům manipulovat s databází za využití INSERT, UPDATE či DELETE příkazů. Měl by tedy, když vytvoří tak veliké omezení na svoje softwarové řešení, přinést i způsob, kterým jejich zákazníci mohou úpravy v databázi provádět. Toto řešení existuje a nazývá se DI API.

Popis

Jedná se o C# knihovnu, pomocí které je možné libovolně manipulovat s oficiálními i uživatelskými tabulkami v databázi SAP. DI API je poměrně přímočaré řešení, kterým se dají záznamy v databázi upravovat velice podobným způsobem, jako kdyby uživatel pracoval s grafickým rozhraním SAP. V první chvíli se může programátorovi API zdát relativně složité, ale jakmile pronikne do logiky SAP a jeho způsobu, kterým jsou data vedeny, není až tak složité s API pracovat.

Omezení kladené na programátora

Ve svém oficiálním řešení klade knihovna poměrně veliké omezení na programátora aplikace tím, že je dostupná pouze v C#. To je, jak bylo popsáno výše, programovací jazyk z platformy .NET vyvíjený Microsoftem. Naštěstí ale vzhledem k velikému počtu zákazníků SAP a programátorů, kteří právě na omezení na jazyk C# narazili, existuje několik takzvaných wrapper libraries, neboli knihoven zaobalujících, respektive překládajících původní DI API do prostředí kompatibilního s jazykem, ve kterém programátor chce, či píše aplikaci. Většina těchto knihoven, ať už oficiálních či neoficiálních, přenášejí syntaxi původního DI API do svého prostředí. Je tak tedy jen pozměněna o specifika daného jazyka. Například se jedná o:

- **SAP Business One Java Connector**[10] Oficiální wrapper DI API pro jazyk Java.
- **phpDIServer**[11] PHP interface pro SAP Business One DI API.
- **SAP RFC**[12] Poskytuje možnost, jak napojit C/C++ aplikaci k SAP.

- **Flask-SAPB1**[13] Knihovna pro Python umožňující napojení na SAP a přímé užití objektů z rozhraní SAP DI API.

Problém tohoto řešení

Kdyby se ale přistoupilo k řešení takovému, že každá aplikace bude napojena na DI API (libovolnou knihovnou), stále jsme nevyřešili problém, při kterém musejí programy žít, či přistupovat do vnitřní sítě, ve které je jediná možnost, jak se k instanci SAP Business One a Microsoft SQL databáze připojit. Druhý problém tkví ve věci takové, že se DI API připojuje prostřednictvím uživatele SAP. Je tedy nutné, aby každý, kdo se prostřednictvím API připojuje, byl vedený jako uživatel SAP. Což opět přináší nevýhodu v tom, že každý, kdo by se chtěl k API připojit, by musel znát „svého“ uživatele SAP. Což si *Firma* nepřeje a raději by vytvořila „alias“ pravého uživatele SAP tak, aby nemusela externím programátorům dávat přímé přístupy. Nemluvě o tom, že je neudržitelné, jak z ekonomického, tak i časového hlediska, aby každý vývojář, například e-shopu či jakékoli externí služby, musel znát strukturu a hierarchii objektů dostupných v SAP. S tím jsou i spojena pole v konkrétních objektech či tabulkách.

2.1.3 Cesta nejmenšího odporu

Vzhledem ke všem výše zmíněným problémům, kdy je:

1. Nemožné díky podmínkám instalace SAP Business One zapisovat do databáze prostřednictvím ovladače pro Microsoft SQL databázi. Nehledě na to, že by každý program musel přistupovat a nebo být ve vnitřní síti *Firmy*, aby mohl s databází vůbec komunikovat.

2. Víceméně nesmyslné, aby každý programátor vyvíjející libovolný software pro *Firmu*, či např. napojoval externí internetový obchod, musel znát objektů v SAP Business One a pracovat s DI API.

se *Firma* uchýlila v několika situacích jít tzv. cestou „nejmenšího odporu“. To v praxi znamenalo, že jediné, co bylo potřeba, byla tvorba SSIS procesu, nebo-li SQL Server Integration Services, pomocí kterého je možné vytvořit jednoduchou komponentu sloužící k importu dat do tabulek SAP formou již zmiňovaných SQL INSERT. Byly tak importovány CSV soubory a nebo, v krajních nejojedinělejších situacích, se dokumenty zadávaly ručně. Neefektivita obou těchto řešení, i v malém měřítku, je zřejmá. Ale v několika situacích se stalo, že projekt, od kterého se tak veliká časová nákladnost neočekávala, přerostl přes hlavu a zaměstnanci, kteří se věnovali občasnému ručnímu importu dat, již nestíhali svoji hlavní pracovní náplň. Obecně CSV importy a exporty až tak špatným a nákladným řešením nejsou. Využívaly se zejména v případě odesílání dat o zásilkách k přepravcům, importu přiřazených sledovacích čísel a průběžné aktualizace stavů zásilek. Nicméně toto řešení má jeden veliký problém, kterým není jen to, že to je poměrně „těžkopádný“ přístup, který prakticky nelze rozumně automatizovat a nebo před i po něm vytvořit nějakou formu automatizace. Každý se tak daný proces musí naučit, a ne vždy se jedná o tak jasnou a jednoduchou úlohu. Spousta uživatelských prostředí, se kterými tak musí zaměstnanec pracovat, jsou

úplně odlišná, poskytují odlišnou funkcionalitu a je v tomto procesu snadné udělat chybu. Pojďme si představit doposud nezmíněný možný přístup k věci, který by *Firmě* umožnil některé procesy automatizovat. Vyřešil by dokonce i problém s přístupem z venkovní sítě. Avšak, jako vše doposud zmíněné, s sebou přináší určitá negativa, která zapříčinila nevyužívání této služby.

2.1.4 SOAP API VCZ.WebService

Využití tzv. VCZ.WebService by mohlo být do této chvíle nezmíněnou variantou řešení představeného problému. Jedná se o službu poskytovanou českou společností Versino CZ s.r.o. Ta se mimo jiné zabývá instalacemi SAP Business One do firem a jejich následné údržby a podpory. Dává tedy veliký smysl, aby přímo zprostředkovatel SAP poskytoval i možnost, skrze kterou by se dala integrace externí aplikace do SAP Business One provést.

Popis

Tato služba se nazývá *Versino WebService pro SAP B1i* a *Firma* má její licenci již od doby, kdy se integrace SAP prováděla. Tato služba je vedena jako SOAP API, což v praxi znamená, že se právě snaží řešit problém, na který narážely samotné implementace DI API do každého programu zvlášť. Vzhledem k tomu, že se jedná o webovou službu protokolem SOAP, je možné k funkcím tohoto software přistupovat vzdáleně po síti.

SOAP

Pojďme si ve zkratce představit protokol SOAP, výhody a nevýhody, které přináší implementace webové služby tímto způsobem. **SOAP** neboli *Simple Object Access Protocol* [14] je důležitým protokolem vyvinutým za podpory Microsoft v roce 1998. Protokol stál a víceméně dal za vznik API využívajících pro svou komunikaci HTTP. Pro výměnu zpráv využívá značkovací jazyk XML, který měl za cíl vytvořit jazyk pro formátování dat srozumitelný jak pro člověka, tak i stroj, a dal tak základ dnešním trendům v přenosu dat přes *World Wide Web*. Bohužel tím, že je SOAP založený na XML, nese i pár nevýhod, kterými jsou například: i) Je relativně složité na parsování. ii) Bez podpory datových typů. iii) Je poměrně složitý na validaci. Ty se přenášejí i do samotného SOAP. Díky formátování XML pak vzniká i poměrně veliká redundance v datech a odpovědi, alternativně požadavky na API, tak nabývají.

Problém tohoto řešení

Vraťme se opět k VCZ.WebService. Jaký je tedy důvod, že ho *Firma* k těmto zdánlivě vzorovým úlohám nevyužívá? Jedním z důvodů je fakt, že toto API opět neřeší problém s tím, že každý externí program napojený skrze VCZ.WebService opět potřebuje vlastního SAP uživatele a není tedy možné, jako ve všech výše zmíněných případech, oddělit uživatelskou správu API od SAP. Je tedy velmi složitě šiřitelné třetím stranám. Druhým a více důležitým problémem je složitá rozšiřitelnost tohoto API. VCZ.WebService působí jako velice dobře navržené řešení, které má ale sloužit všem. API přináší řadu funkcí, které dokáží nahradit

užití uživatelského prostředí SAP, neboť ho skvěle kopíruje ve formě XML komunikace. Webová služba poskytuje dva možné přístupy ke svým službám.

(i) **Přístup SQL**, který je, z výše zmíněných důvodů, pouze pro čtení, jelikož by zápis porušoval smluvní podmínky služby.

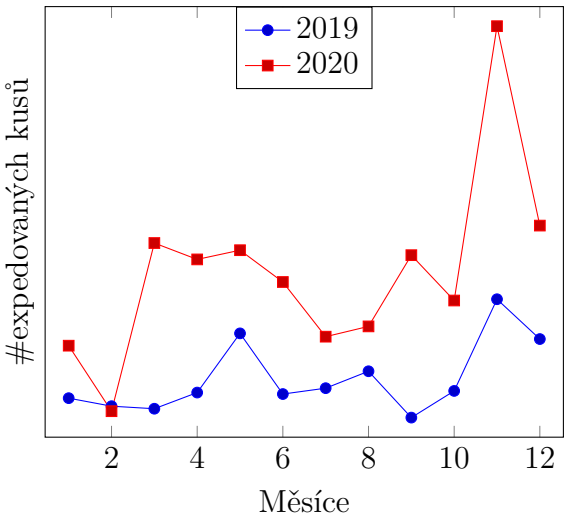
(ii) **DI přístup**, jenž slouží jak k načtení, tak i k zápisu dat. Jak si však ukážeme v nadcházejících částech textu, využívat DI pro čtení je velice neefektivní.

Komplikace služby popisované v této sekci však spočívá v její univerzálnosti. Z důvodu obecného navržení, i když velice složitou cestou, umožňuje většinu potřebné funkcionality, která je v případě velikého počtu dotazů velmi časově i výkonové náročná. Například v situaci, kdy by se chtěl internetový obchod dotázat na dostupnost několika produktů, není možné zaslat jeden dotaz se seznamem více artiklů. Je nutné načíst detail každého produktu zvlášť. Což v praxi může znamenat u několika tisíců artiklů, které má *Firma* skladem, desítky minut postupného zasílání dotazů a zpomalování jak SAP, tak i samotné aplikace, která požadavky zasílá. V průběhu let, kdy se služba zkoušela, se tak zjistilo, že API samozřejmě poskytuje velice rozsáhlou funkcionalitu, ale kolikrát až tak obecně, že chtěný proces brzdí. Firma Versino samozřejmě ke svému produktu poskytuje i podporu a možnost rozšíření jeho funkcionality. To by však v tomto případě znamenalo velikou spoustu rozšíření, a to by bylo jak finančně, tak i časově pro obě strany velice nákladné. Pojďme se v rychlosti podívat na přehled několika funkcí, které služba přináší a nebo naopak nepřináší a pro účely *Firmy* by byly vyhovující. Můžeme vidět, že i přes velikou škálu nabízených funkcí VCZ.WebService, požadavky *Firmy* jsou jiné *(viz Tabulka 2.1)*.

2.2 Datová komunikace s přepravními společnostmi

V této části textu si přiblížíme problém, který mimo jiné úzce souvisí se sekcí 2.1. *Firma*, jejíž podnikaní se čím dál tím více posouvá do formy *Business-To-Customer* z původního *Business-To-Business*, je nucena pozměnit i interní logistické postupy. Zvětšuje se totiž absolutní počet zásilek, ale protiváhou v prodeji koncovému zákazníkovi je to, že se objem, resp. počet kusů, v zásilce zmenšuje. Narůstající počet zásilek sklad při svých procesech zvládá velice dobře. Problém však nastal v jejich expedici. Rok 2020 přinesl veliký zlom v podobě dlouhodobého uzavření maloobchodu, a tak i odstřižení velké části *B2B* odběratelů. Růst *B2C* byl však ještě strmější a více než kdy jindy bylo potřeba revitalizovat postup, při kterém jsou informace o zabaleném balíčku předány přepravci. Nárůst počtu expedovaných kusů produktů v daném měsíci v letech 2019 a 2020 nastiňuje graf 2.1.

Aktuální situace byla totiž dost problematická. Postup, kterým se zásilky expedovaly, byl velice zdlouhavý a relativně náročný pro každodenní obsloužení v několika na sebe navazujících krocích.



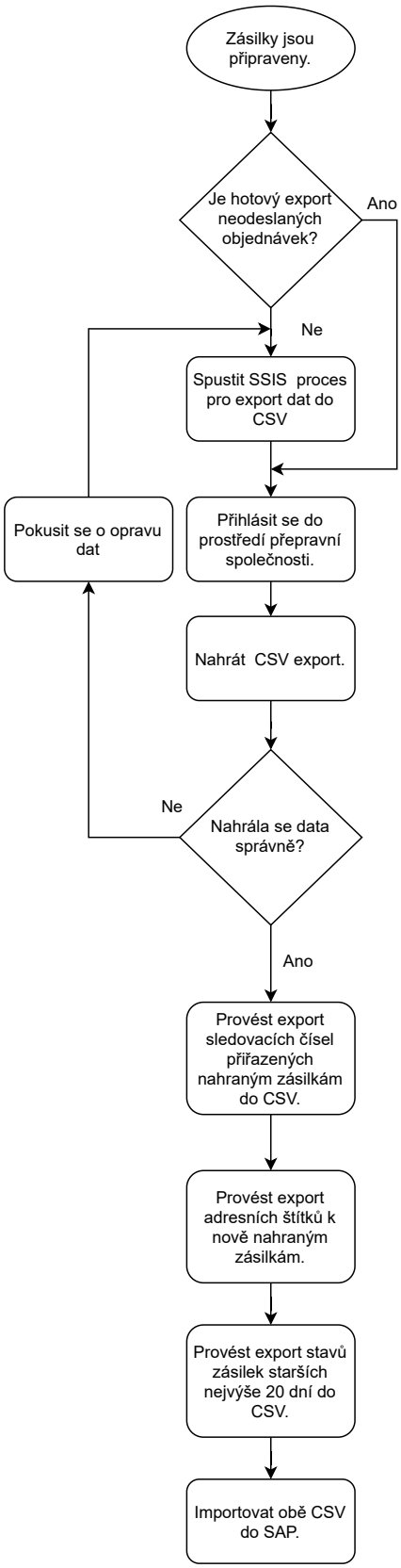
Obrázek 2.1: Porovnání expedovaných kusů/měsíc v letech 2019 a 2020

2.2.1 Původní neefektivní postup

Jak je v předchozích částech postupně nastiňováno, postup, kterým byla odesílána data k přepravním společnostem, byl značně neefektivní a velice náchylný ke vzniku lidské chyby. Proces byl v prvopočátcích relativně udržitelný, protože *Firma* spolupracovala s jedním, nanejvýš dvěma přepravci. Jakmile však počet vzrostl na stabilní tři, v některých chvílích i čtyři přepravce, bylo pro obsluhu již velmi náročné celý postup tolikrát denně opakovat. Nemluvě o tom, že byli zaměstnanci vázáni na využívání uživatelského prostředí SAP. To obecně není samozřejmě špatně. SAP má velice propracované a rozsáhlé možnosti zobrazení tabulkových dat, a tak s poskytovanými informacemi problém v žádném případě nebyl. Ten vznikal z důvodů toho, že se tento proces, v závislosti na době, kdy kurýři přepravních společností zásilky vyzvedávají, spouští zhruba mezi 12. a 13. hodinou. V tuto dobu je ve *Firmě* SAP i samotná databáze velmi vytížená, a tak se doba načítání dat a čekání obsluhy na potřebný import/export prodlužuje. Se zvyšujícím počtem objednávek tento problém rostl, neboť nebyla vytížená jen obsluha ve skladu, ale i oddělení fakturace, které pak právě aktivněji využívalo SAP. Řešení, které totiž *Firma* v tuto chvíli využívala, bylo nastíněno v sekci 2.1.3. Tento proces je znázorněný na obrázku 2.2. Z pochopitelných důvodů bylo s narůstajícím počtem objednávek toto řešení dlouhodoběji neudržitelné.

Výhody tohoto přístupu

Doposud jsme na toto řešení problému komunikace s přepravními společnostmi nahlíželi pouze negativně. Nelze však pohlížet jen na stinnou stránku věci. Je nutné vzít v úvahu i to, že toto řešení přinášelo do určité úrovně relativně velikou flexibilitu v možnostech doplňovat nové přepravní společnosti. Pokud tedy, a ve velké většině tomu tak opravdu je, přepravní společnost nabízí možnost importu dat v podobě CSV či jiného podobného formátu, *Firmě* stačilo pouze vytvořit správný export formou *SQL SELECT* dotazu a data pro import byla velice jednoduše na světě. Proces nahrání a uložení dat o zásilkách k přepravní společnosti je zpravidla velice přímočarý. Poté pouze stačí správně nastavit kom-



Obrázek 2.2: Univerzální vysokoúrovňový pohled na neefektivní proces předávání dat k přepravním společnostem.

ponentu pro import do SAP tak, aby mapovala příslušné sloupce tabulky. Jak je však patrné z obrázku 2.2, těchto importů a exportů dat je v celém procesu 5.

Nevýhody tohoto přístupu

Nevýhody tohoto přístupu byly zmíněny v předchozích částech textu. Jsou totiž tak důležité, že tvořily valnou většinu motivace stojící za návrhem a tvorbou aplikace, která má obsluze práci výrazně ulehčit. Obecný problém tohoto řešení spočívá, mimo jiné, v naprosté nemožnosti ho jakkoli efektivně škálovat. Ať už z pozice nárůstu objednávek, či samotných smluvních přepravců. Kvůli tomu, že obsluha nemá žádné unifikované prostředí, ve kterém by mohla jednoduše pracovat, naráží tento přístup na velikou různorodost uživatelských prostředí aplikací poskytovaných přepravními společnostmi. Vše je zřejmě odrazem pro-zákaznického přístupu jednotlivých přepravců. Lze totiž vidět paralelu v návrhu a celkové přehlednosti uživatelského prostředí se samotnou přehledností nabídky služeb, které daný přepravce poskytuje. Cílem této kapitoly samozřejmě není hodnotit kvalitu zákaznické podpory ani spolehlivost kurýrních služeb. Ale nahlédnout na fakt, že si některé společnosti zřejmě neuvědomují, že spokojenost odesílatele nelze měřit až od chvíle předání zásilky kurýrovi, ale během celého procesu, i během odeslání dat o konkrétních zásilkách. Velmi často se obsluze totiž stávalo, například u odesílání zásilek Českou Poštou, že se některé zásilky zkrátka nenahrály. To není nutně chybou daného systému, neboť mohla být chyba ve vstupních datech. Ale taková informace by měla být v uživatelském prostředí velice výrazně zobrazena. Obsluha tak díky tomu musela ručně porovnávat počet zásilek, které byly exportovaný ze SAP a těch, které se zobrazily v prostředí ČP. Další značnou nevýhodou tohoto přístupu byl samotný SAP. Kvůli jeho rychlosti v nejvytíženějších částech pracovního dne bylo pro obsluhu velice těžké kontrolovat, zda-li se po následném exportu a nahrání dat do firemního ERP všechny zásilky spárovaly správně a zda-li jsou všechna exportovaná data přiřazena k odesílaným zásilkám. A jako poslední slabinou tohoto řešení je fakt, že tento proces strádá z lidské lenosti. Velmi často totiž docházelo k tomu, že obsluha ve skladu zapomínala, či v některých případech nedělala pořádně, aktualizace stavů již odeslaných zásilek. Některá uživatelská prostředí totiž nedovolovala přímočarým způsobem vybírat rozsahy datumů s překryvem dvou a více měsíců. Docházelo tak k tomu, že zásilky odeslané koncem měsíce nebyly zpravidla aktualizovány v potřebném rozsahu, standardně 20 dní po odeslání.

Funkce	VCZ.WebService	Využitelné
Neodeslané objednávky ČP	NE	ANO
Odeslané objednávky ČP	NE	ANO
Aktualizace objednávky ČP	NE	ANO
Neodeslané objednávky Packeta	NE	ANO
Odeslané objednávky Packeta	NE	ANO
Aktualizace objednávky Packeta	NE	ANO
Dostupnost artiklu	ANO	ANO
Dostupnost pole artiklů	NE	ANO
Cena pole artiklů	NE	ANO
Detail artiklu	ANO	ANO
Seznam artiklů dle data změny	ANO	NE
Seznam artiklů dle ItemCode	ANO	ANO
Seznam artiklů dle ItemCode	ANO	ANO
Počet artiklů dle data změny	ANO	NE
Detail OP dle ID	ANO	NE
Seznam OP dle jména	ANO	NE
Aktualizovat OP dle ID	ANO	NE
Vrátit seznam obchodních příležitostí	ANO	NE
GDPR status OP dle emailu	NE	ANO
GDPR statusy pole OP dle emailu	NE	ANO
Zápis OP	ANO	ANO
Seznam zakázek OP	ANO	NE
Zápis pole OP	NE	ANO
Zápis dodacího listu	ANO	NE
Zápis předběžného dokladu	ANO	ANO
Zápis pole předběžných dokladů	NE	ANO
Vrátit sazbu DPH dle kódu země	ANO	NE
Vrátit seznam pole doprav	ANO	NE
Správa uživatelů	NE	ANO
Informace k podnikové databázi	ANO	ANO
Informace k podnikové databázi	ANO	ANO
Správa uživatelů API	NE	ANO
Správa rolí uživatelů v API	NE	ANO

Tabulka 2.1: Přehled funkcí, které poskytuje VCZ.WebService API a funkcí, které by *Firma* využila.

3. Analýza řešení - API

Výsledkem práce a řešením problémů popsaných v předešlých sekcích jsou dva ve své podstatě izolované projekty, kde však jeden představuje praktické využití toho druhého. Jedná se o realizaci REST API jakožto řešení problému ze sekce 2.1 a vytvoření aplikace, která zautomatizuje většinu úloh z procesu popsaného v pasáži 2.2. V této kapitole zanalyzujeme řešení REST API.

Samotné API bude vytvořeno s využitím několika knihoven, které budou postupně představovány v nadcházejících pasážích tohoto textu. Základní stavební jednotkou však bude WSGI[1] microframework Flask pro programovací jazyk Python. API bude v *nízkoúrovňovém* pohledu rozděleno na dvě separátní aplikace. Nástin řešení je zobrazený na obrázku 3.1.

1. **Master aplikace** hlavní část API sloužící pro přímou komunikaci s SQL databází a instancí SAP. Bude spuštěna na Windows Server 2012 a uzavřena pouze pro komunikaci v rámci vnitřní sítě.

2. **Uživatelská aplikace**, jenž bude sloužit pro vnější přístup k API a tudíž i jako část ověřující uživatele, kteří se k API přihlašují. Bude nasazena na separátním serveru a její komunikace s *master aplikací* bude pouze uvnitř firemní sítě.

3.0.1 Master aplikace

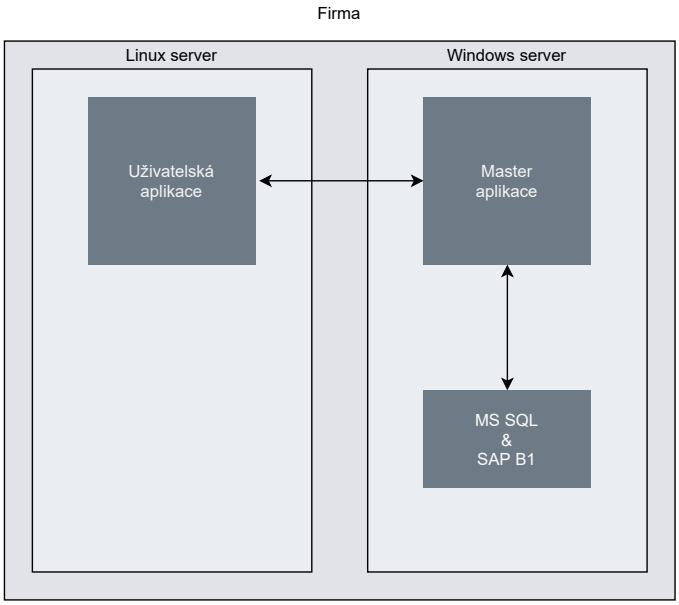
Toto rozhraní pro komunikaci s *Firmou* bude navrženo tak, aby například umožňovalo plné napojení drop-shipping e-shopu. Konkrétní příklad napojení bude rozebrán v kapitole 7.1.1 Vyhodnocení, neboť již v době psaní této práce existuje internetový obchod, jenž toto API využívá pro kontrolu skladových zásob produktů a zápis dat o objednávkách do SAP.

3.1 Požadavky na API

Řešením problému, kdy bylo pro *Firmu* velice obtížné efektivně moderně a dle svých požadavků získat data o produktech, zákaznících, ale i objednávkách vedených v databázi SAP Business One, byla tvorba REST API. Hlavními požadavky na realizaci tohoto API bylo:

- Moderně navrženo pro rychlé a jednoduché napojení na libovolnou externí službu.
- Správa uživatelů by měla být řešena na úrovni API.
- Jednoduše rozšiřitelné o další požadované funkce, na které se v průběhu využívání přijde.

[1] Web Server Gateway Interface



Obrázek 3.1: Diagram API

Dalším z požadavků, resp. z rozumných přístupů, k návrhu aplikace bylo oddělit logiku získávání a zápisu dat. Tak, aby se k dotazování využíval konektor do SQL databáze, a naopak pro zápis dat se využívalo, z důvodu zmíněných v předešlých částech textu, SAP DI API. Neboť využívat DI API ke čtení je velice neefektivní kvůli dlouhé inicializaci a načítání objektů SAP.

3.1.1 Funkcionalita API

Požadavky na metody dostupné v API nebyly příliš rozsáhlé. Ač je SAP Business One velmi veliký a obsáhlý software, *Firmě* v tuto chvíli stačí pro aktuální verzi API popisovanou v této práci přístup pouze k pár základním tabulkám, kterými jsou například - tabulka s předběžnými doklady (ODRF), tabulka s objednávkami (ODRD), uživatelská tabulka Pošta, tabulka obchodní partnerů (OCRD). Obecně se funkce v API dají rozdělit na dvě skupiny.

Dotazovací funkce

První a zároveň nejobsáhlejší skupinou jsou funkce komunikující přes ODBC driver pro Microsoft SQL databázi. Jedná se o metody, které přímo přistupují k databázi bez využití prostředníka, kterým je SAP. Protože se jedná pouze o dotazovací funkce, není nutné mít obavu o jakoukoli ztrátu podpory či záruky. Mezi specifikované patří:

- Získání informací o nových zásilkách pro ČP.
- Získání informací o zásilkách odeslaných před X dny ČP.
- Získání informací o nových zásilkách pro Zásilkovnu.
- Získání informací o zásilkách odeslaných před X dny Zásilkovnou.
- Získání informací o nových zásilkách pro Uloženku.

- Získání informací o zásilkách odeslaných před X dny Uloženkou.
- Získat dostupnost artiklu.
- Získat detail artiklu.
- Získat všechny artikly přiřazené prefixu.
- Získat seznam všech prefixů.
- Získat cenu artiklu ve specifikovaném ceníku (dle ID).
- Získat souhlas ke zpracování osobních údajů za účelem marketingu ke specifikovanému emailu.

Zapisovací funkce

Druhou skupinou jsou funkce komunikující přes SAP DI API. Jedná se o metody, které nepřímo přistupují k databázi a využívají abstraktní vrstvu nad databází, kterou vytvářejí objekty SAP. Tyto objekty jsou zprostředkovány využitím DI API. Každá funkce tak musí využít tohoto konektoru a inicializovat správné SAP objekty. Ač je možné DI API využít k dotazovacím funkcím, jedná se o velmi neefektivní přístup. I přes to, že je i klasický SQL UPDATE/INSERT do Microsoft SQL databáze výrazně rychlejší, než-li použití DI API, bohužel není jiná možnost, než využít tohoto oficiální SAP konektoru. Předchozí způsob je totiž v rozporu s podmínkami výrobce tohoto software a do databáze by se nemělo zapisovat jiným způsobem. Funkce, které manipulují s daty v databázi jsou:

- Zápis dat k objednávce expedované Českou Poštou v tabulce Posta.
- Zápis dat k objednávce expedované Packeta v tabulce Posta.
- Zápis dat k objednávce expedované Uloženka v tabulce Posta.
- Vytvoření nového předběžného dokladu.
- Vytvoření nového obchodního partnera.
- Změna souhlasu ke zpracování osobních údajů za účelem marketingu ke specifikovanému emailu.

3.2 Implementace jako webová služba

Již několikrát bylo v této práci zmíněno, přímo i nepřímo, že API, které bude využíváno pro získávání a zápis dat, bude, z pohledu architektury, implementováno jako **REST API**. V této sekci rozebereme důvody pro tuto formu implementace. Jedním z hlavních požadavků na aplikaci bylo, aby umožnila přístup ověřeným třetím stranám k datům *Firmy*. To, jak jsme si ukázali v části 2.1, se dá vyřešit několika způsoby. Patrně nejméně bezpečnou variantou by bylo napojit aplikace nějakým „tunelem“ do vnitřní sítě *Firmy*. Toto řešení bylo ihned zavrhnuto a nápady se ubíraly směrem vytvoření veřejného komunikačního interface. K této úloze se skvěle hodí využít HTTP. Neboť pro téměř kterýkoli moderní

programovací jazyk existuje knihovna umožňující HTTP komunikaci. Jmenovitě pro Python se může jednat o knihovnu *Requests* [15], pro C++ *libcurl*[16] a nebo *Guzzle*[17] pro PHP. Možnosti jsou tedy veliké a externí programy by mělo být velice jednoduché napojit na naše webové API. Jednoduše tak umožní napojení dropshipping e-shopů k zápisu předběžných dokladů či získání dat o produktech. Nevýhody implementace formou webové služby vycházejí převážně ze samotného HTTP. Jediným z hlavních teoretických problémů může být bezstavovost samotného protokolu. Díky této koncepci celý proces funguje pouze tak, že uživatel zašle požadavek našeho API, to ve chvíli přijetí zprávy požadavek zpracuje a odpoví. Jakmile odpoví, o požadavku samotném nemůže v podstatě už nic říct. A tak všechna data musí být, jak ze strany klienta, tak i serveru, zaslána v jedné dávce. Za použití správných technik se jedná o řešitelný problém, avšak tím se v této práci zaobírat nebudeme.

3.2.1 REST API

[18] Již během analýzy problému zápisu a získávání dat jsme popsali jedno z možných řešení implementovaných technologií SOAP (viz 2.1.4). Tento protokol byl ve své době vzniku velmi dobrým a lukrativním řešením webové služby. Nicméně, pro implementaci této služby si autor práce vybral přístup formou **Representational state transfer (REST)**. Ten, na rozdíl od přístupu SOAP, je orientovaný více datově, než-li procedurálně. Navíc není vázaný na jeden datový formát jdoucí v těle požadavku. REST totiž například podporuje:

- **XML (Extensible Markup Language)** [19] značkovací jazyk užívaný pro formátování datových dokumentů. Hlavní motivací vzniku bylo vytvořit rozhraní pro komunikaci člověka a stroje tak, aby bylo rozumně čitelné pro obě strany. Nevýhodou však je, že se všechna data reprezentují jako textový řetězec.

- **JSON (JavaScript Object Notation)** [20], je datový formát založený na principu slovníku, neboli párů klíč-hodnota. Na rozdíl od XML se jedná o silně typovaný formát využívající mapování s užitím různých datových struktur.

- **YAML (YAML Ain't Markup Language)** [21], je datový formát pojmenovaný rekurzivním akronymem. Formátem připomíná JSON (pozn. JSON je validní YAML), avšak neodděluje bloky složenými závorkami, ale odsazením. Jako například jazyk Python.

API popisované v této práci využívá pro komunikaci formát JSON. Motivací pro jeho volbu byla jednoduchá čitelnost a větší kompaktnost na rozdíl od XML. Ta je zapříčiněna nižší redundancí elementů. Díky tomu by i syntaktická analýza takového souboru měla být rychlejší než-li u XML.

Samotný REST je díky tomu velice jednoduše rozšiřitelný. Základní model použití je například *GET*. Tyto metody tak lze v nejjednodušších případech volat přímo z webového prohlížeče. Jedná se tak o velice jednoduchou webovou aplikaci, ve které uživatel volá funkce podle zvolené metody:

- GET

- POST
- PUT
- DELETE

Metody jsou dostupné na konkrétních URL. Kombinace těchto dvou „parametrů“ resp. metody a URL pak dají za vznik unikátní cestě, na které může být dostupná metoda daného API.

3.3 Technologie

Již jsme si ukázali, že naše API bude implementováno architekturou REST. Protože je již z předchozích částí známo, že je naše aplikace naprogramována v jazyce Python s využitím frameworku Flask, ukážeme si, že se nejednalo o řešení, které by omezovalo, či výrazně ztěžovalo, vývoj funkcionality samotné aplikace.

3.3.1 Programovací jazyk

Začněme samotným programovacím jazykem, ve kterém je, resp. jsou, aplikace napsány. Zvolen byl *Python*. Jelikož autor této práce měl s návrhem webových aplikací v jazyce Python základní zkušenosti, pohlížel na to jako výzvu, jak si je rozšířit. Dalším z argumentů, proč byl zvolen tento jazyk, je jeho vysokoúrovňovost. Vzhledem k charakteru aplikace, která převážně slouží k předávání a zpracování dat přes HTTP, nebyla hlavním požadavkem ani cílem co nejmenší paměťová, či obecně výkonová náročnost. Ale spíše velmi dobrá čitelnost a co nejjednodušší rozšiřitelnost aplikace o další API funkce během jejího života. Všechny tyto parametry Python skvěle splňoval. Mimo jiné i díky velké komunitě programátorů ovládajících tento jazyk zajišťující bezproblémovou online podporu, či poměrně jednoduché a nenákladné rozšíření jiným programátorem. V předchozích částech jsme již zmiňovali i existenci knihoven obalujících C# SAP DI API knihovnu. Python, jak bylo uvedeno, není výjimkou a též pro něj existuje tzv. wrapper. Tudíž ani v tomto pro nás není volba tohoto dynamicky typovaného jazyka omezením.

3.3.2 Framework

V této části si představíme možnosti webových frameworků, které by bylo možné využít k vyhotovení API.

Flask

[22] Začněme velice oblíbenou knihovnou *Flask* pro jazyk Python. Flask se zpravidla označuje jako tzv. webový micro-framework. Jak píší samotní autoři knihovny viz. [22], „micro“, v tomto smyslu, neznamená „ochuzené“, ba naopak „jednoduché“ a „snadno rozšiřitelné“. A to je pravda. Flask je totiž sám o sobě velice kompaktní framework s rozsáhlou uživatelskou základnou, která pro téměř každý případ použití vytvořila rozšíření. Jak totiž přímo stojí v úvodním slově

dokumentace „Flask may be “micro”, but it’s ready for production use on a variety of needs.“.

Z jakých dalších knihoven bychom mohli vybírat?

Pistache

[23] Pokud se odloučíme od omezení na použití jazyka *Python*, *Pistache* je zajímavým příkladem frameworku pro tvorbu REST API v jazyce C++. Obsahuje vícevláknový HTTP server a je implementovaný v C++17. Ale jeho komunita a tím i spojená dostupnost uživatelských návodů či implementací je velice omezená.

Tornado

[24] Pokud se přesuneme zpět k jazyku Python, dalším možným frameworkem bylo například *Tornado*. To bohužel opět strádá z výrazně menší komunity než té, kterou má *Flask*. To samozřejmě není nutně důvod pro okamžité zavrhnutí dané knihovny, ale budoucí rozšiřování programu by mohlo být složitější z důvodu nižšího počtu uživatelů. Nemluvě o tom, že se *Tornado* zdálo svou obsáhlostí externích knihoven už příliš obsáhlé pro specifičnost této aplikace.

FastAPI

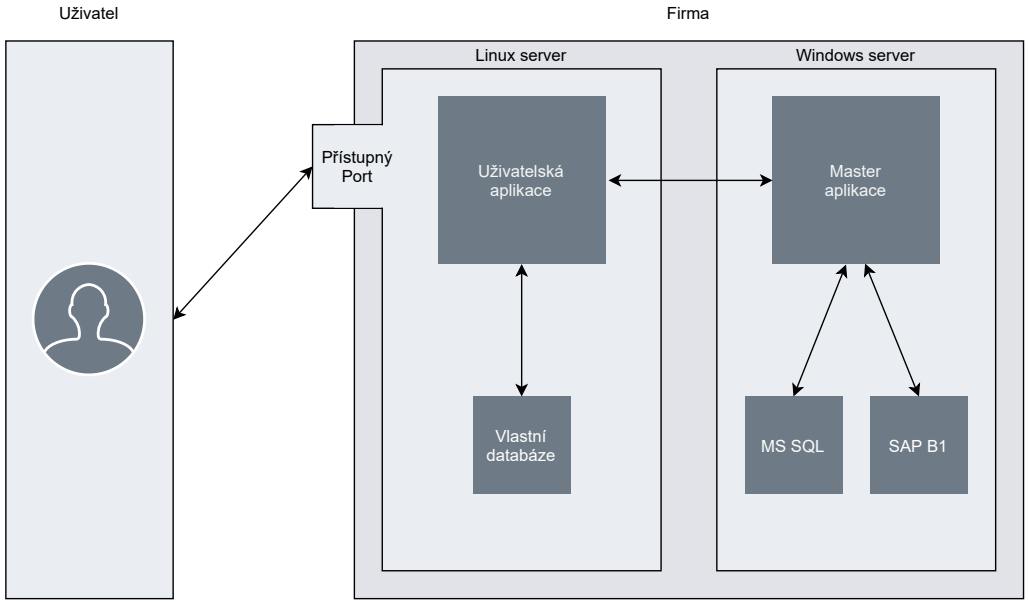
[25] Poslední alternativou bylo *FastAPI*. Opět se jedná o framework pro Python, který, jak již název vypovídá, by měl primárně sloužit ke tvorbě API. Díky tomuto frameworku je možné poměrně rychle vytvořit fungující REST API s automaticky generovanou dokumentací *Swagger* [26]. Syntaxe je poměrně přímočará a jednoduchá na pochopení, ale opětovně strádá z nízkého počtu zdrojů.

Výsledek

Autor aplikace se tedy rozhodl pro tvorbu aplikace s využitím *Flask* jakožto frameworku s velikou uživatelskou základnou a velmi dobrou dostupností informací a návodů. Framework byl rozšířen o knihovnu *flask_restx* [27] umožňující mimo jiné automatizovanou *Swagger* [26] dokumentaci jako v případě frameworku *FastAPI*. Subjektivním důvodem pro volbu tohoto frameworku byl i zájem o rozšíření již existujících znalostí. I přes značné nevýhody v paměťové náročnosti či obecné rychlosti tato webová služba z volby Python frameworku nijak nestrádá. Nekladla příliš vysoké nároky na rychlost, které by bylo z pozice REST API nutné dohánět. DI API samo o sobě je v inicializaci všech objektů velice pomalé, jak bude popsáno v další části textu. Výhoda čitelnosti a dostupné podpory k Flasku tak byla převažující.

3.4 Návrh REST API

Již jsme si uvedli, jaké funkce má naše API mít, v jakém programovacím jazyce bude napsané a jaký hlavní framework k tomu bude využitý. Co jsme však doposud přecházeli, byl nízkoúrovňový návrh API. Již jsme zmínili, že bude rozděleno do dvou samostatných aplikací, které budou spolu komunikovat v rámci vnitřní sítě *Firmy*. Do hlavní aplikace, tedy té, která bude přímo napojena skrze



Obrázek 3.2: Diagram komunikace uživatele a API

DI API k SAP B1 a databázového konektoru k MS SQL, kde bude pro zápis informací využito DI API a pro získávání informací a dotazování se bude využitý konektor. Pro účely této práce si ji označíme jako **Master aplikace**. Druhou aplikací, která bude součástí tohoto API, je uživatelský „interface“ pro komunikaci s *Master aplikací*. Nazveme ho **Uživatelská aplikace**. Schéma návrhu API je znázorněno diagramem 3.2.

Tento přístup byl zvolen ze dvou hlavních důvodů. Prvním z nich byla bezpečnost. Hlavně díky tomu, že bude uživatelská aplikace umístěna na separovaném serveru, na kterém bude otevřený port namapovaný na 80. Uživatel tak bude komunikovat pouze se serverem s nainstalovanou distribucí Linux a nikoli přímo na server s instancí SAP i databáze k němu napojené. Druhým důvodem, ač méně důležitým v porovnání s bezpečností sítě, proč bylo API takto rozděleno, byl fakt, že si *Firma* nepřála, z opodstatnitelných důvodů, aby se na server s instancí SAP instalovalo co nejméně balíčků a knihoven. Mohlo by totiž dojít ke kolizi se SAP. Totiž, během testování možných řešení, z neznámých důvodů, instalace knihovny *Flask-JWT* ve verzi 0.3.2 zapříčinila pád celého SAP. Proto se od řešení, kdy bude veškerá logika API, včetně uživatelské autentizace, odstoupilo a nastavil se cíl, aby část aplikace fungující na Windows byla co nejjednodušší a opravdu pouze vyřizovala nejnutnější věci, kterými samozřejmě byla komunikace se SAP a databází. O zbytek, jako je sledování požadavků, přiřazování rolí a kontroly přistupujících uživatelů, se postará *uživatelská aplikace*.

3.4.1 Master aplikace

Jak již bylo uvedeno, „master“ aplikace funguje jako hlavní komunikační část API se SAP a jeho databází. Z toho důvodu musí být umístěna přímo na serveru s instalací SAP, neboť se DI API připojuje pouze k lokálně pracující instanci tohoto ERP software. Z bezpečnostních důvodu je tato část API izolována od venkovní sítě a je tak možné s *Master aplikací* komunikovat pouze v rámci vnitřní sítě

Firmy.

Funkcionalita

Tato část je zodpovědná za připojení k Microsoft SQL databázi a samotnému SAP. Přímé připojení k SQL databázi je využíváno ke čtení dat, protože je výrazně rychlejší než-li využití DI API. To je používáno pouze k zápisu a aktualizaci v databázi. Kvůli velice zdlouhavé inicializaci SAP objektů je výhodnější zapisovat data ve větších dávkách, poněvadž by se při každém zápisu musely opětovně načíst příslušné SAP objekty. To je i hlavní důvod, proč je ke čtení využito přímé SQL připojení k databázi.

Prostředí

Prostředí, kde *Master aplikace* pracuje, je Windows Server 2012. Avšak to je i jeden z důvodů, proč bylo přistoupeno k řešení, kde *Master aplikace* nezastřešuje žádnou složitou logiku a pouze tak přijímá jednoduché požadavky a vykonává SQL dotazy či využívá DI API. Požadavek ze strany *Firmy* byl, aby se na server hostující SAP instalovaly pouze ty nejnutnější knihovny sloužící ke komunikaci API. Na základě dřívějších zkušeností, které *Firma* měla, se „zásahy“ resp. instalace jakékoli software třetí strany do serveru musely co nejvíce omezit, protože docházelo k situacím, kdy z nejasných důvodů došlo ke přímé kolizi se SAP či významného zpomalení celé aplikace. Vzhledem k výše popsanému, kdy *Master aplikace* využívá ke své práci knihovny fungující na Windows a potřebuje přímý přístup k instanci SAP, ani nebylo možné API umístit do izolovaného kontejneru, ve kterém by fungovalo téměř izolovaně.

Nasazení

Problém nastal i ve chvíli, kdy bylo nutné aplikaci připravit ke spuštění. Za normálních okolností by byl samozřejmě rozumný přístup přistupovat k API prostřednictvím webovým serverem (např. Apache [28] či Nginx [29]). To však opět nebylo možné, a tak se přistoupilo k využití časovaných úloh kontrolující „živost“ aplikace. To je zařízeno prostřednictvím velmi jednoduchého skriptu kontrolujícího, zda-li je na specifikovaném portu aplikace dostupná. Pokud není, skript jednoduše provede pokus o její spuštění. V části popisující *Uživatelskou aplikaci* si předvedeme „správný“ nebo alespoň efektivnější a z programátorského i administrátorského hlediska přívětivější způsob, kterým se dá zařídit tzv. „deployment“ webové aplikace.

3.4.2 Uživatelská aplikace

Není možné, aby se uživatel API připojoval přímo k *Master aplikaci* hned z několik důvodů. Primárně z bezpečnostního hlediska a sekundárně kvůli prostředí pro hostování této aplikace. To na nás klade natolik veliká omezení, že by vývoj aplikace byl zbytečně složitý. Pro naši uživatelskou aplikaci totiž budeme potřebovat i nějaké vlastní perzistentní úložiště dat, ve kterém budeme schopni uchovávat informace potřebné k ověření uživatelů, kontroly jim přiřazených rolí a sledování jejich zasílaných požadavků do API.

Funkcionalita

Naše *Uživatelská aplikace* tak bude jediným „klientem“ *Master aplikace*. Neboť všechny dotazy, které *Uživatelská aplikace* odešle směrem k *Master aplikaci*, by již měly mít dostatečně ověřený původ. V aplikaci bude možnost vytvářet uživatelské účty s přiřazenými „rolemi“. Při tvorbě každé funkce, jenž bude prostřednictvím REST API dostupná, budeme schopni poměrné jednoduše přiřadit množinu uživatelských rolí, které k ní mohu přistupovat. Role budou jednoduše rozšiřitelné a API bude umožňovat i funkce, pomocí kterých bude možné uživatelské účty kompletně spravovat. Včetně přidání nových uživatelů, změny jejich rolí či hesel a správy samotných rolí. Bude tedy primárně zajišťovat správu a ověřování uživatelů. Funkce pro administrátora uživatelů našeho API budou:

- Vytvoření nového uživatele.
- Smazání stávajícího uživatele.
- Přiřazení role uživateli.
- Odebrání role uživateli.
- Výpis přiřazených rolí konkrétnímu uživateli.
- Výpis uživatelů.
- Výpis rolí.
- Výpis požadavků zaslaných všemi uživateli.
- Výpis požadavků zaslaných konkrétním uživatelem.

Databáze

Již jsme uvedli, že aplikace bude ke své práci potřebovat perzistentní úložiště dat. Možností je hned několik. Můžeme zvolit klasickou relační databázi a nebo se vydat cestou „nerelační“ databáze neboli NoSQL [30]. To přináší jisté výhody i nevýhody. Nerelační databáze dávají možnost ukládat data v jiném formátu než relační databáze. Databáze má tak velikou výhodu v jednoduchém růstu do šířky. Avšak pro nás bude vhodnější růst do výšky, neboť schéma našich tabulek se měnit nebude a data, která budeme ukládat, budou téměř identická. Zvolili jsme tady relační model. Z jakých databází můžeme volit? Hlavními třemi databázovými management systémy, které se aktuálně využívají, jsou *MySQL* [31], *SQLite* [32] a *PostgreSQL* [33] [34]. Začněme od té nejjednodušší a představme si *SQLite*. Jedná se bezserverové řešení databáze ve formě souboru uloženého na disku. Je to velice jednoduché a snadno přenositelné, avšak ne příliš připravené jako produkční řešení. K databázi nelze vzdáleně přistupovat pomocí nějakého prohlížeče tabulek a neobsahuje žádnou správu uživatelů přistupujících k databázi. Přesuneme se tedy k serverovým řešením, kterými jsou MySQL a PostgreSQL. Obě jsou poměrně jednoduché na nasazení a pro naše účely mezi nimi není až takový rozdíl. K databázi bude standardně přistupovat pouze jeden uživatel, kterým bude naše API a veliké omezení na výkonnost a rychlost též neklademe, poněvadž neočekáváme veliký nápor uživatelů. PostgreSQL je paměťově mírně náročnější, ale ze

subjektivního hlediska má autor práce více zkušeností s prací s PostgreSQL, a to byl jeden z hlavních důvodů, proč také byla zvolena.

Nasazení

Jakým způsobem můžeme program nasadit do produkčního prostředí? Zvolíme moderní a aktuálně velmi populární způsob „deploymentu“. *Firma* nám vytvoří virtuální stroj na jednom z jejich serverů, ve kterém bude nainstalované Ubuntu Server ve verzi 18.04 [35]. Nasazení aplikace bude provedeno do Docker kontejneru. To samo o sobě samozřejmě nestačí, neboť bychom tímto způsobem dali uživatelům API přístup do kořenové složky naší *Uživatelské aplikace*. Vytvoříme si tedy druhý kontejner, ve kterém bude nasazený webový server Nginx. Ten nám bude sloužit jako taková proxy směrující požadavky na náš server do *Uživatelské aplikace*. Samozřejmě by bylo možné aplikaci nasadit podobným způsobem jako naši *Master aplikaci*. Zde by již ale mohlo hrozit nebezpečí, protože by *Uživatelská aplikace* měla být přístupná požadavkům z vnější sítě. Velikou výhodou řešení, kdy aplikaci nasadíme do kontejnerů a budeme schopni ji spravovat jako *docker-compose* [36] aplikaci. *Docker-compose* je totiž nástroj sloužící ke spojení více kontejnerů do jednoho celku využitím jednoduchého konfiguračního souboru. Díky tomu budeme schopni naši aplikaci, ač tvořenou ze tří kontejnerů, spouštět jedním příkazem. Hlavní výhodou tohoto přístupu je to, že naše *Uživatelská aplikace* bude naprosto nezávislá na prostředí, ve kterém je spouštěna. Jakmile bude v rámci sítě *Firmy* a na hostu bude nainstalovaný nástroj Docker [37], budeme schopni aplikaci spustit v řádu jednotek minut. Mimo jiné nám to umožní aplikaci lépe škálovat a ulehčí případný přechod na Kubernetes [38]. Bližší informace o administraci kontejnerů jsou popsány v přiložené administrátorské příručce C.

Poskytovat aplikaci bude námi zvolený web server nasazený jako „proxy“. Víceméně máme dvě možnosti, mezi kterými můžeme volit. Lze zvolit *Apache*[28], jenž byl vytvořený v devadesátých letech minulého století. Obecně se jedná o nejvyužívanější variantu webového serveru, neboť je zpravidla ve většině výchozích instalací různých hostingových služeb. Druhou a více „Docker-friendly“ variantou je *Nginx*[29] publikovaný v roce 2004 jako odpověď na neduhy *Apache* resp. jeho výkonnostním omezení. Pro *Nginx* existuje velice dobrá dokumentace nasazení Flask aplikace a nese s sebou mnohem kompaktnější konfiguraci, než právě zmiňované *Apache*.

4. Analýza řešení - aplikace pro komunikaci s přepravci

Jak již bylo v úvodu této práce zmíněno, jedním z výsledků této práce bude reálná aplikace vytvořeného REST API pro *Firmu*. Tento projekt se bude snažit zautomatizovat většinu úloh z procesu, popsaného v sekci 2.2, a výrazně tak zjednoduší každodenní práci obsluze ve skladu *Firmy*. Bude mít tak možnost se více věnovat své další práci a tuto velmi důležitou, ale zbytečně složitě řešenou úlohu, jí pomůžeme urychlit a zpříjemnit.

Aplikace pro komunikaci s přepravci bude navržena jako webová aplikace s vlastní databází zásilek, ve které budou uchovávány jejich průběžné stavy a data, jenž byla odeslána k přepravní společnosti. Aplikace ve formě, jak je popsána v této práci, má dokončené napojení na Českou Poštu rozebráno ve vlastní sekci 6.2.14. Napojení na Zásilkovnu je též rozebráno v samostatné sekci. 6.2.13. Původní návrh aplikace počítal i s napojením na API Uloženka[1]. S tou však *Firma* rozvázala spolupráci, a tak napojení nebylo plně dokončeno. Nicméně, zásadním důvodem, který dělá tuto aplikaci zajímavou z pozice tohoto textu je fakt, že slouží jako názorná ukázka přímého využití našeho API v praxi v každodenním procesu ve *Firmě*.

4.1 Návrh aplikace pro komunikaci s přepravci

Návazným problémem na složitou možnost napojení externího software bylo i obtížné zřízení komunikačního mostu mezi *Firmou* a jejími smluvními přepravci. Jak jsme si již představili v předchozích částech textu, součástí této práce je i návrh a tvorba programu sloužícího k automatizovanému přenosu dat o zásilkách mezi *Firmou* a přepravní společností. Program tak pomůže v práci obsluhy ve skladu a názorně ukáže možné využití našeho API představeného v předchozích sekcích.

Aplikace by ve své základní verzi, jež je součástí tohoto textu, měla plně umožňovat předávat data o odchozích zásilkách a generovat štítky a soupisku zásilek[2] pro přepravce Česká Pošta (balíčky[3] i dopisy[4]) a Zásilkovna (Packeta). Automatizovaně by aplikace měla umožňovat zapisovat data o zásilkách do SAP prostřednictvím našeho API a průběžnou aktualizaci stavů zásilek. I přes to, že jsou standardně zásilky doručeny nejvýše do druhého pracovního dne, průběžné aktualizace by se měly na žádost *Firmy* provádět i u zásilek odeslaných před dvaceti dny. To však úměrně prodlužuje celkovou dobu aktualizace. Ze zkušenosti *Firmy* občas docházelo ke zdržení zásilek při přepravě a bylo tak nutné jejich stavy získávat i v delším časovém horizontu, než je obvyklá doba doručení. Velikou výhodou tohoto programu bude i uživatelské prostředí, ve kterém bude moci obsluha jednoduše a rychle prohlížet jednotlivé zásilky a kontrolovat jejich stav,

[1] Od 2. 11. 2020 byla Uloženka přejmenována na WE|DO.

[2] Jedná se o jednoduchý A4 dokument, ve kterém je souhrn všech zásilek. Obsluha předává dokument k potvrzení kurýrovi přebírajícímu balíčky.

[3] DR - vnitrostátní zásilka „Balík Do ruky“

[4] RR - vnitrostátní „Doporučené psaní“

neboť u každé zásilky bude uchována i historie jejích stavů. Abychom si tedy funkcionalitu shrnuli, v tomto seznamu jsou vypsány klíčové funkce, které jsou požadavkem ze strany *Firmy*:

- Přehled všech zásilek v tabulce.
  - Včetně možnosti exportovat aktuální tabulku do CSV.
- Možnost zobrazit historii stavů získaného od přepravce u konkrétní zásilky.
- Jednoznačně označit zásilky, které nebyly doposud nahrány k přepravci.
- Filtrování zásilek
  - Dle unikátní identifikátoru SAP.
  - Dle variabilní symbolu.
  - Dle jména příjemce.
  - Dle emailu příjemce.
  - Dle zvoleného přepravce.
  - Dle interního stavu v programu.
    - Úspěšně odesláno k přepravci.
    - Úspěšně nahráno do SAP.
    - Vygenerováno na soupisce.
    - Vygenerováno na štítku.
- Odeslat data o neodeslaných zásilkách k České Poště.
- Získat aktuální stav zásilek odeslaných nejvýše před X dny Českou Poštou.
- Odeslat data o neodeslaných zásilkách k Zásilkovně.
- Získat aktuální stav zásilek odeslaných nejvýše před X dny Zásilkovnou.
- ~~Odeslat data o neodeslaných zásilkách k Uložence (WE|DO).~~[5]
- ~~Získat aktuální stav zásilek odeslaných nejvýše před X dny Uloženkou (WE|DO).~~[5]
- Průběžně aktualizovat stavy zásilek u přepravců.
- Průběžně kontrolovat, zda-li úspěšně proběhlo nahrání do SAP.
- Vygenerovat adresní štítky pro danou přepravní společnost.
- Vygenerovat soupisku zásilek pro danou přepravní společnost.

[5] Jak již bylo zmíněno, spolupráce s tímto přepravcem byla v průběhu realizace této práce rozvázána.

4.2 Uživatelské prostředí

V této části si popíšeme uživatelské prostředí naší aplikace. Obsahuje tři hlavní stránky, které se starají o vykreslování všech potřebných informací a dostupných ovládacích prvků.

4.2.1 Hlavní panel

Ihned po spouštění bude našemu uživateli zobrazena stránka zobrazující tabulku s přehledem všech zásilek, které byly do aplikace nahrány. Tento hlavní panel by měl obsahovat základní informace o zásilce, pomocí kterých ji bude obsluha schopna rychle identifikovat. Patří mezi ně takzvané „Docentry“, nebo-li unikátní identifikátor v tabulce SAP, variabilní symbol, samozřejmě přepravce, kterým je zásilka dopravována, základní kontaktní údaje a datum. Na řádku by mělo být i tlačítko, které nás přenese na detail zásilky popsaný v příští podsekci 4.2.2.

Abychom práci obsluze zpříjemnili ještě více, měli bychom analyzovat situace, kdy bude tabulku prohlížet nejčastěji. První a nejdůležitější chvíle bude ihned po nahrání zásilek. Pokud obsluha obdrží během nahrávání informaci o tom, že se některé zásilky nepodařilo nahrát, je nutné je co nejpohodlněji identifikovat. Proto každý řádek, jenž má příznak značící chybu při nahrávání zásilky prostřednictvím externího API přepravce, výrazně označíme. Například celý řádek obarvíme červenou barvou. Druhým důležitým parametrem je datum nahrání zásilky. Zejména tak potřebujeme odlišit řádky, jejichž datum nahrání je aktuální den, aby bylo možné jednoduše zvolit ty, u kterých je potřeba například vygenerovat adresní štítky.

Nad tabulkou obsahující výpis zásilek by měla být umístěna textová a rozbalovací pole, pomocí kterých bude možné mezi zásilkami rychle filtrovat. Filtrovat by mělo jít dle docentry, variabilního symbolu, emailu, jména, přepravní společnosti a chybového stavu, ve kterém se může nyní zásilka nacházet na úrovni naší aplikace. Takovými stavy máme na mysli zásilky, které se z nějakého důvodu nepodařilo nahrát k přepravci do SAP a nebo nebyly doposud vygenerovány na adresní štítek či soupisku.

Protože se očekává, že bude aplikace zpracovávat relativně vysoké počty objednávek denně, je rozumné vyřešit i stránkování. To bude zabezpečeno klasickým číselníkem pod tabulkou zobrazující jednotlivé stránky.

Jak již bylo uvedeno, naše aplikace bude umožňovat i vygenerování adresních štítků a soupisek. Bylo by však dobré, aby aplikace umožňovala toto generování u uživatelem vybraných zásilek. Pokud si tedy uživatel pomocí dostupného filtru zvolí konkrétní přepravní společnost, mělo by se na každém řádku objevit zaškrtávací políčko. Společně s tím bude pod tabulkou ovládací komponenta, prostřednictvím které bude možné vygenerovat štítky/soupisku pro označené řádky. Pokud bychom se neomezili na konkrétní přepravní společnost, toto řešení by zjevně nefungovalo a nebo bylo pro uživatele zbytečně náročné. Každá přepravní společnost totiž vyžaduje jiný formát a podobu adresních štítků či soupisky.

Na obrázku 4.1 můžeme vidět wireframe sloužící k lepšímu pochopení a vizualizaci popisovaného dashboardu.

4.2.2 Detail zásilky

Abychom aplikaci zpřehlednili, hlavní panel se omezil pouze na ty „nejdůležitější“ informace o zásilce. Resp. ty, které nám ji pomohou rychle identifikovat. Aby uživatel zjistil všechny podrobnosti, bude k tomu moci využít stránku poskytující všechny informace o detailní zásilce včetně všech stavů, kterými během přepravy prošla.

Uživatelské prostředí detailu zásilky tak bude rozděleno na dvě části. První z nich bude jednořádková tabulka obsahující všechny informace, které máme v databázi dostupné. Od základních dodacích údajů, které byly přepravci předány, až po sledovací číslo s odkazem a příznaky, pomocí kterých můžeme i v hlavním panelu filtrovat. Vzhledem k obsáhlosti a šíři tabulky, bude horizontálně pohyblivá, aby počet zobrazených informací nebyl na úkor jejich čitelnosti.

Pod vodorovným výpisem informací bude umístěna tabulka, s chronologicky seřazenými stavy, ve kterých se zásilka kdy nacházela. V této tabulce bude na každém řádku umístěno pořadí, konkrétní stav, jak z přepravy, tak i nahrávání včetně chybových hlášení, které bylo možné z externího API získat. V posledním sloupci bude umístěno datum, kdy zásilka daného stavu nabyla.

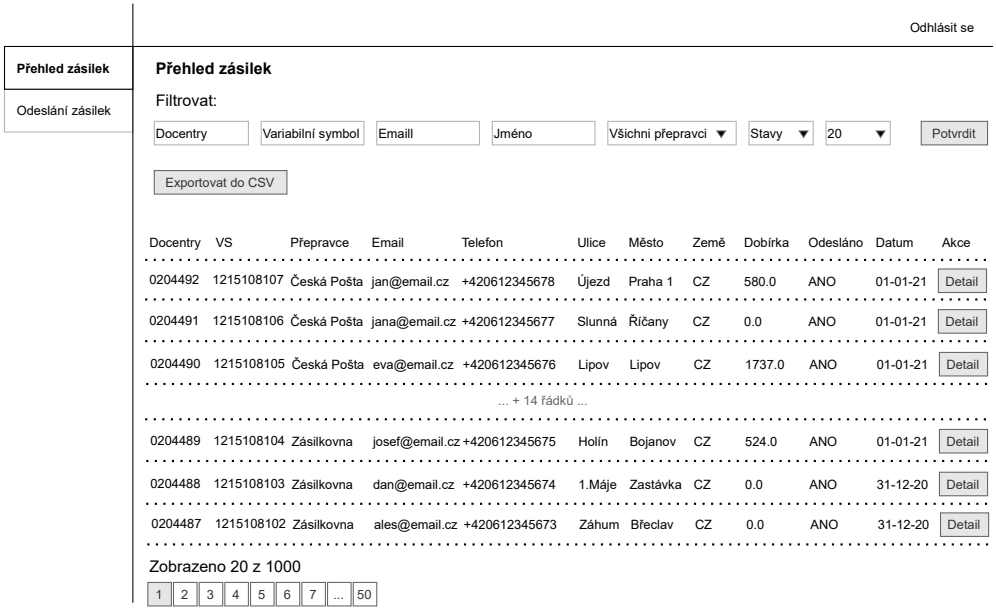
Opět, jako v u hlavního panelu byla vytvořena skica uživatelského prostředí, která je přiložena na obrázku 4.2.

4.2.3 Okno pro odeslání zásilek

Hlavní motivací za vznikem této aplikace však nebylo prohlížení zásilek či kontrola jejich historických stavů. Bylo to však pohodlné odesílání a předávání dat prostřednictvím API přepravních společností. Všechny předchozí části uživatelského prostředí byly primárně pro doplnění přehlednosti, aby obsluha nemusela nutně využívat SAP pro rychlé prohlížení zásilek či, v horším případě, log programu. Cílem tedy bylo urychlit proces při předávání informací o zásilkách. O to se stará okno, popsané v této podsekci.

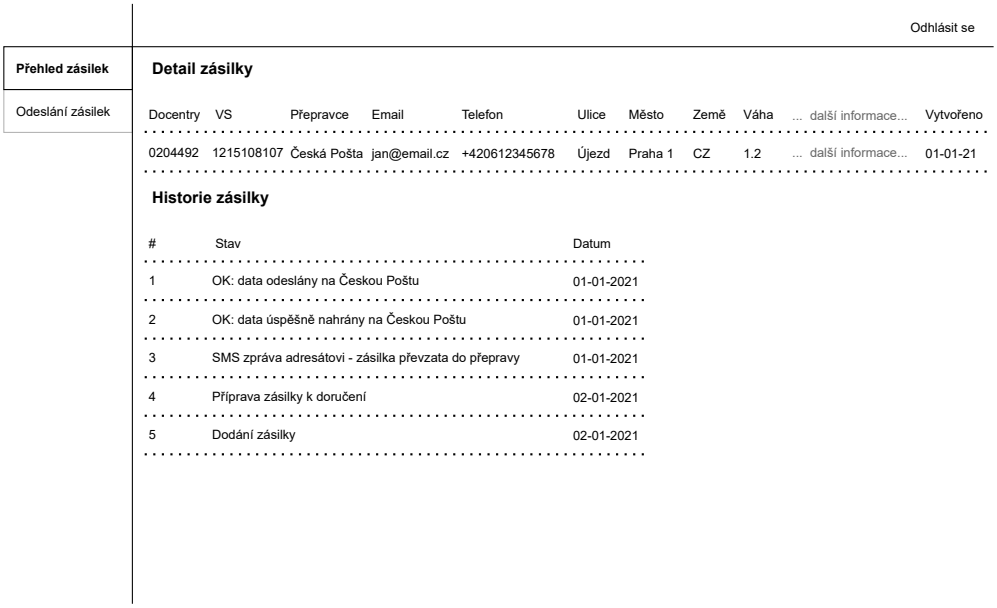
Abychom proces co nejvíce zjednodušili, je cílem, aby obsluze stačilo stisknout, téměř vždy, jen jedno tlačítko. Tento proces je totiž předem velmi jasně daný. V podstatě jediné, co se děje. je to, že si program nahraje do své databáze zásilky k odeslání, předá data přepravci, ten vrátí odpověď, *Firma* ji nahraje do SAP a zažádá o adresní štítky + vytvoří soupisku balíčků. Vystačíme si tedy s tlačítkem, které spustí řetězec funkcí našeho programu. Protože naše funkce budou muset být specifické pro každého přepravce, bude stránka rozdělena do sekcí dle přepravní společnosti a každá tak tedy bude mít vlastní sadu tlačítek. Je to i součástí procesu, neboť obsluha předávání dat upřednostňuje dle času, kdy během dne kurýři vyzvedávají zásilky. Pod každou sadou tlačítek se po jejich stisku zobrazí tabulka, ve které budou chronologicky vypsány jednotlivé kroky programu v uživatelsky přívětivé formě. Budeme tak zobrazovat počet zásilek, které program nahrál a odeslal. Nakonec po zpracování odeslání uživateli zobrazíme odkaz, na kterém si stáhne adresní štítky nyní podaných zásilek i jejich soupisku. Pokud budou tedy čísla odeslaných a úspěšně přijatých zásilek souhlasit, nebude tak nucený vybírat jednotlivé řádky v tabulce hlavního panelu. Pro lepší pochopení byl přiložený wireframe popisovaného okna na obrázku 4.3.

Vzhledem k charakteru API České Pošty, budeme potřebovat druhé ovládací tlačítko „Získat výsledek“. Funkce pro odeslání dat je totiž asynchronní, a tak

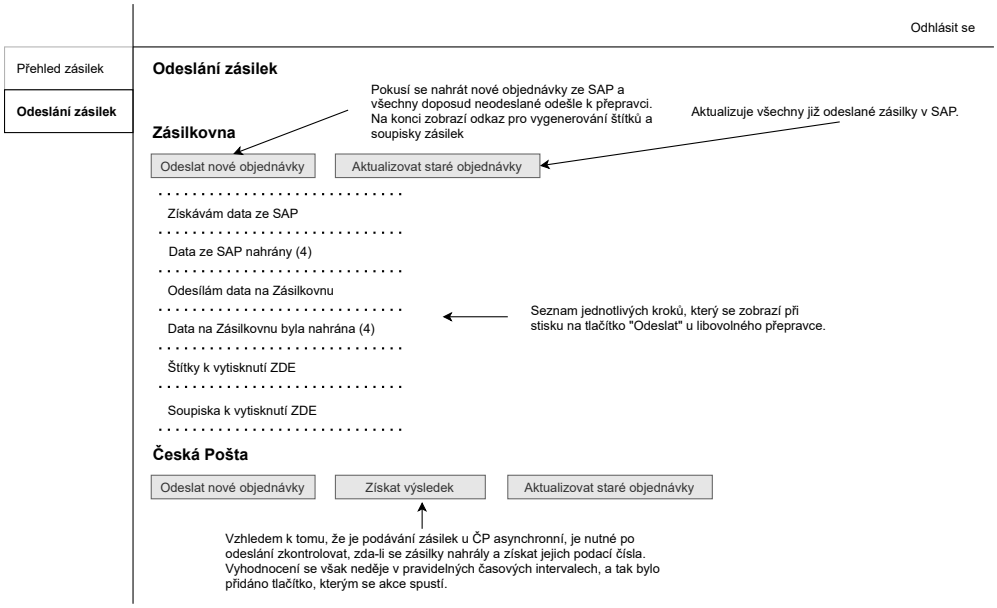


Obrázek 4.1: Wireframe Okna s hlavním panelem

se obsluha musí na výsledek zpracování dotázat až několik vteřin po úspěšném odeslání dat. Protože doba, kdy je odpověď dostupná,je velice různorodá, bude právě zmíněné tlačítko umožňovat obsluze vyvolat dotaz na odpověď opakovaně.



Obrázek 4.2: Wireframe Okna detailu zásilky



Obrázek 4.3: Wireframe Okna pro odeslání zásilky

4.3 Možnosti implementace aplikace

Vzhledem k tomu, že uživatelem naší aplikace je obsluha ve skladu využívající Windows a SAP, způsobů, jak přistupovat k implementaci naší aplikace, je několik. Nabízela by se totiž varianta aplikaci rovnou implementovat jako komponentu do SAP, nativní[6], webovou či snad dokonce mobilní aplikaci. V této sekci si představíme každou ze zmíněných variant a ukážeme si přístup, který jsme nakonec zvolili.

4.3.1 Integrace do SAP

Představme si na úvod variantu, která by nejspíše rozporovala většinu požadavků *Firmy* na tuto aplikaci. Zakomponování komunikačního můstku s přepravci by byla patrně „nejčistší“, pro nezasvěcené, nejrozumnější varianta. Neboť, pokud je možnost rozšířit nějaký stávající systém, je to samozřejmě rozumnější než vytvářet další. Začne pak vznikat veliká spousta programů, které se musí separovaně udržovat a spravovat. Jeden z problémů, na který obsluha při své práci narážela, bylo to, že ji SAP vlastně „brzdí“ při práci. Jedná se o velice dobrý systém, který má firma zaběhnutý. Pro tyto účely bylo lepší se od „all-in-one“ řešení odloučit. Základní filtrování a proklikávání se uživatelským prostředím SAP má poměrně velikou odezvu a i z pohledu „user-experience“ je v hodnější konkrétně tuto aplikaci od SAP odloučit. Pokud bychom však potřebovali velice dobrý databázový editor, byl by SAP rozumnou volbou, avšak pro naše účely bude stačit pouze „prohlížeč“. Nehledě na to, že integrace do SAP s sebou přináší potřebu znalosti celého prostředí i SAP SDK[7]. A jakákoli rozšiřitelnost funkčnosti aplikace by byla silně vázána na možnosti poskytované SAP.

4.3.2 Mobilní aplikace

Tato varianta byla již v počátcích vyloučena, protože obsluha při tomto procesu nemá potřebu být jakkoli mobilní a není nutné implementovat aplikaci např. do tabletu. Nehledě na to, že aplikace má umožňovat i pohodlné prohlížení dat, což by na malém displeji, když to není vyžadováno např. prací z terénu, bylo zbytečné.

4.3.3 Desktopová aplikace

Vývoj nativní aplikace by zřejmě splňoval většinu požadavků. Aplikace by byla spouštěna jako externí nezávislá na SAP a obsluha by mohla pohodlně prohlížet data na celé své obrazovce. Jednou z nevýhod, které skýtá toto řešení je nemožnost rychle doručit aktualizaci či opravu chyby. Občas se totiž může stát, že se vyskytne velice ojedinělá forma zásilky, a protože je všechna logika aplikace postavena pouze na předávání a získávání dat využitím externích API, je vysoce pravděpodobné, že se občas vyskytne chyba, kterou je nutno velice rychle opravit a ideálně umožnit co nejrychleji obsluze dále pracovat a expedovat zásilku. To nám bohužel tato varianta v rozumné míře neposkytuje, neboť by každá změna

[6] Myšleno Windows desktopová aplikace

[7] Zkratka anglického *sousloví software development kit*

vyžadovala kompilaci a následnou instalaci na počítač obsluhy. Navíc, pokud by chtěl do aplikace přistupovat i někdo jiný z *Firmy*, musel by si aplikaci k sobě lokálně doinstalovat.

4.3.4 Webová aplikace

Toto řešení nám přinese velice jednoduchou možnost aktualizovat samotný program, jelikož se z pohledu obsluhy nebude dít nic jiného, než jen smazání mezipaměti v prohlížeči (pro aktualizaci stylů a skriptů) a obnova načtení stránky. A zároveň přinese jednoduchost i nám při, tvorbě například front-end, kdy se nebudeme muset vázat na omezení nějakého frameworku pro vývoj nativních aplikací, ale velmi jednoduše vytvoříme front-end využitím HTML[39], CSS[40] a JavaScript[41].

4.3.5 Volba způsobu implementace

Představili jsme si patrně většinu způsobů, kterými by šla aplikace realizovat. Všechny mají nějaké výhody a nevýhody. Patrně pro nás je nejdůležitější rychlá odezva samotné aplikace a flexibilita v jejích aktualizacích. Ač se webové aplikace mohou zdát poměrně těžkopádné kvůli zastaralému prostředí, které HTTP poskytuje a složitě udržitelné z programátorského pohledu, resp. nějakých „best practices“, jejich jednoduchá rozšiřitelnost a dynamičnost v možnostech změn a úprav se odráží v celém trendu, kdy spousta známého nativního software je přesouvána online a uživatelé k nim přistupují ze svého webového prohlížeče. Zaměstnanci navíc nebudou nuceni aplikaci doinstalovávat a pouze se podívají skrze prohlížeč.

4.4 Návrh webové aplikace

To, že bude aplikace koncipována jako webová, jsme se již rozhodli. Tento způsob implementace s sebou nese velikou škálu cest, kterými se může programátor vydat. V této sekci rozebereme možné programovací jazyky konkrétní frameworky, které by byly pro naše použití vhodné.

4.4.1 Programovací jazyk

Zde si představíme možné programovací jazyky pro vývoj back-end naší aplikace. Výběr je velice rozsáhlý, neboť pro téměř každý programovací jazyk existuje framework, se kterým lze rozumně vytvořit webová aplikace. My se však omezíme na tři, které by potencionálně připadaly v úvahu. Nemusí se nutně jednat o ty nejpoužívanější, ale budeme vycházet ze zkušeností s tvorbou webových aplikací autora práce.

PHP[7]

Zřejmě stále nejrozšířenějším a hojně užívaným jazykem pro vývoj serverové části webových aplikací, je **PHP** [42]. Jedná se o skriptovací jazyk využívaný na back-end zhruba 79% všech webových stránek [43], ke kterým je tato informace

dostupná. Toto číslo se však v čase snižuje. V roce 2017 bylo PHP zastoupeno na 83% webových stránek, jejíž vývojáři informaci o použitém jazyku na serveru zveřejnili. Samozřejmě je toto číslo velmi ovlivněno a nemůžeme ho brát jako fakt. Co však lze z tohoto průzkumu vidět, je snižující se trend využití PHP. To sklízelo ve svých předešlých verzích[8] velikou kritiku za špatný návrh a přístup k objektově orientovaným jazykům. Na samotném PHP by v dnešní době již nemělo být nic, co by programátora doslova odradilo od jeho využití. Velmi mu do karet i hraje fakt, že lze velice levně nasadit, protože většina webhostingových poskytovatelů nabízí podporu PHP v jejich základních verzích. Což pro jiné jazyky pravidlo nebývá, a tak se využití dále představovaných jazyků může prodražit, protože programátor musí sáhnout po řešení virtuálního serveru, který bývá výrazně dražší.

JavaScript[41]

Druhým jazykem, který si rozebereme je **JavaScript**. Ač ho většina čtenářů bude znát jako client-side/front-end programovací jazyk, v poslední době se výrazně zvyšuje oblíbenost jeho využití i pro vývoj back-end díky Node.js [44]. Jeho využití není tak rozsáhlé jako u PHP [45], avšak narozdíl od něj, během posledních 12 měsíců zaznamenal veliký nárůst ve využití pro vývoj webových aplikací. Proč se tedy bavíme o JavaScriptu, ale zmiňujeme popularitu Node.js? Jedná se totiž o běhové prostředí JavaScriptu vyvíjený pod záštitou Google, které nám umožňuje psát aplikace v JavaScript a spouštět je lokálně či na serveru. JavaScript však s sebou nese i pár zvláštností. Ač se jedná o objektově orientovaný jazyk, všechny objekty představují asociativní pole. Druhou zajímavostí je, že se funkce řadí mezi ostatní objekty. Můžeme si ji tak například přiřadit do proměnné.

Python [8]

Třetím a posledním jazykem, který připadal do úvahy při vývoji, je jazyk **Python**, představený již v předchozích částech tohoto textu a využitý pro naprogramování našeho API pro komunikaci se SAP. Jedná se dynamicky typovaný jazyk, který v posledních letech nabral na veliké oblibě. Jeho využití mezi webovými aplikacemi za posledních 12 měsíců víceméně stagnuje [46]. Což není samozřejmě nutně indikátorem problému. Velikou výhodou Pythonu, oproti všem ostatním zmíněným jazykům, je jeho veliká univerzálnost. Vzhledem k tomu, že má Python obrovský záběr mezi použitím jednotlivých aplikací, které jsou v něm napsány, je vyhledatelnost knihoven a implementací různých algoritmů velice jednoduchá a podpora od ostatních programátorů nemusí být, jako u PHP, ovlivněna světem webových aplikací, ale z úplně jiného úhlu, který může programátorovi webové aplikace výrazně pomoci či otevřít oči.

Zvolený programovací jazyk

Vzhledem k tomu, že aplikace byla vyvíjena prakticky paralelně se samotným API, byl pro vývoj webové aplikace využit jazyk Python. Nejen však z tohoto důvodu byl tento jazyk zvolený. Protože bude aplikace zpracovávat odpovědi z několika externích API, v Pythonu se, ze subjektivního pohledu autora, velmi příjemně dobře serializují objekty, které právě mohou představovat odpovědi z

[8] V době psaní práce je aktuální verze PHP 8

API. Ty potřebujeme ukládat a nebo si z nich převzít nějakou důležitou informaci. Další motivací, za kterou stála tato volba, byla chuť se naučit pracovat právě s Pythonem na úrovni webové aplikace s front-end. Neboť je možné, že pozici, kterou mezi webovými aplikacemi zabírá PHP budou postupně přebírat další jazyky. Myšlenka samozřejmě netkví v tom, že PHP zanikne, ale jeho mírný propad v posledních letech značí, že většina nově vznikajících aplikací již není vyvíjena čistě v PHP.

4.4.2 Framework pro vývoj webové aplikace v Python

Aby se vývoj obecně urychlil, používají se k vývoji aplikací frameworky, které mohou sloužit jako základ pro vytvářený software. Dá programátorovi či programátorům určitý směr například s konkrétním návrhovým vzorem a ulehčí práci, protože může zapouzdrovat některé knihovny, které by si tak museli zaopatřit sami [47]. Jelikož jsme se předchozí části textu omezili pouze na jazyk Python, pojďme si představit možné frameworky, sloužící k vývoji webových aplikací v tomto jazyce.

Flask[48]

Nejspíše nejznámějším webovým frameworkem pro Python je již zmiňovaný *Flask*. V první části jsme si ho představili jako ideálního kandidáta pro naše REST API. Nyní se však nacházíme v jiné situaci. Budeme potřebovat rozumně přistupovat ke statickým souborům, využívat šablonový jazyk [49] při renderování HTML a hlavně přistupovat k databázi, ideálně pomocí nějakého ORM. ORM, neboli objektově relační mapování, je technika, která nám dovolí k objektům uloženým v relační databázi přistupovat v programovacím jazyce jako ke třídám z daného jazyka. Flask víceméně vše ze zmíněného poskytuje. Ve své standardní verzi je dodáván s knihovnou *Jinja2*, která umožňuje právě šablonovou syntaxi. Jako ORM je možné zvolit populární knihovnu *SQLAlchemy* a pro migrace databáze zvolit *Flask-Migrate*.

Django[50]

Vše výše popsané by nám mělo stačit. Avšak pokud nám nebude vadit zvolit „těžkotonážní řešení“, které nám dá ve výsledku více, můžeme zvolit framework *Django*. To vše a mnoho více z výše popsaného obsahuje. Je založené na Flasku a mimo jiné má velmi dobře zpracovanou administraci, která funguje víceméně jako editor záznamů v databázi, a například integrovanou logiku pro správu relací v prohlížeči například kontroly přihlášení uživatele. K Django existuje veliká uživatelská základna a s tím i spousta knihoven rozšiřující jeho základní funkcionalitu. Vývojáři tohoto frameworku publikovali i skvěle zpracovanou dokumentaci, ve které je v příkladech popsáno téměř vše, na co může programátor narazit.

Pyramid[51]

Jako odlehčené Django bychom mohli považovat minimalistický framework *Pyramid*. Ten obsahuje jak ORM, tak i engine pro zpracování šablon. Co je však

mírně odrazující je celková uživatelská základna a ne natolik přehledná dokumentace, jako má například Django či Flask.

Volba frameworku

Rozhodnutí padlo na Django. Neboť má opravdu skvěle zpracované ORM a výbornou administraci. Pokud by se totiž obsluze do aplikace nahrály špatně zformátovaná data, může velmi jednoduše skrze */admin* stránku upravovat data v interní databázi aplikace. Nemusíme se tak starat o tvorbu vlastního prostředí, ve kterém by bylo možné data editovat a to nám usnadní spoustu práce. Další hlavní výhodou byla již připravená správa uživatelů vč. využitelných funkcí pro ověření a přihlášení uživatele. Obecně má Django velikou uživatelskou podporu a díky velice dobře zpracované dokumentaci se s ním rychle a jednoduše pracuje.

4.4.3 Úložiště dat

Vzhledem k požadavkům, které byly na naši aplikaci kladeny, je potřeba vytvořit nějaké perzistentní úložiště dat. Protože je naše aplikace velice závislá na chodu externích služeb, je rozumné, aby se mohla, v případě výpadku, kdykoli zotavit a chybu ohlásit. Nechávat aplikaci tohoto typu pracovat tzv. „na jeden průchod“ by bylo velmi náročné na správu. Ve chvíli, kdy obsluha odesílá data o zásilkách, nemá nikdy 100% jistotu, že přepravce data přijme. Ať už z důvodu výpadku či nevalidity konkrétních dat. Například Česká Pošta má poměrně vysoká omezení na adresy, které jí jsou zasílány. V lepším případě adresu upraví dle vlastního formátu a ohlásí informační zprávu „INFO_ADDRESS_WAS_MODIFIED“. V horším případě zahlásí chybu. Pokud bychom si nikde stavy a data k zásilce neukládali, bylo by nutné, aby si obsluha buď zapamatovala vždy dvojici (chyba, zásilka) a nebo složitě pročítala logy programu. Díky rozumné volbě perzistentního datového úložiště tak můžeme práci obsluze ulehčit a vytvořit panel, ve kterém bude vše, včetně chyb přímočaře trvale zobrazeno.

Databáze SAP

Samozřejmě by se nabízela varianta, při které využijeme stávající databázi SAP s uživatelskou tabulkou, ve které si budeme stavy zásilek uchovávat. Toto řešení by bylo rozhodně z návrhového hlediska lepší. Avšak, jak již bylo zmíněno v předchozích částech textu, práce se SAP není z programátorského hlediska nejpříjemnější a bohužel by nám zabránila ve využití zmiňovaného ORM.

Vlastní databáze

A tedy, i přes redundanci dat, zvolíme přístup, kdy budeme všechny informace o zásilkách, jako jsou adresy a kontaktní údaje příjemce, ukládat na dvou místech, tedy v SAP i naší vlastní databázi. Naše řešení díky tomu bude pracovat mnohem svižněji a práce s databází bude z programátorského hlediska výrazně jednodušší.

Možnosti vlastní databáze

Opět tedy stojíme před rozhodnutím ve volbě správného databázového management systému. Možnosti jsou prakticky identické jako v sekci 3.4.2. Pokud

bychom se rozhodli pro NoSQL databázi, konkrétně MongoDB, museli bychom pravděpodobně využít rozšíření frameworku Django třetí strany *djongo* [52]. Ač zmíněné rozšíření působí robustním dojmem, NoSQL databáze by nemusela nutně znamenat špatné rozhodnutí vzhledem k nekonzistentnosti našich dat, kdy každý přepravce vyžaduje jiné informace. Aby bylo možné k datům rozumně přistupovat a případně je i implementovat do nějakých interních firemních procesů, byl zvolen přístup s konvenční relační databází. Konkrétně se jednalo o PostgreSQL databázi, kdy byla její volba ovlivněna téměř identicky jako v sekci 3.4.2. Díky volbě relační databáze tak bude *Firma* případně schopna k datům přistupovat, jak jsou její více technicky založení zaměstnanci zvyklí.

4.4.4 Nasazení aplikace

Aplikace, stejně jako uživatelská část našeho API, bude nasazena v Docker kontejneru na vlastním virtuálním serveru ve *Firmě*. Nebude se však jednat pouze o tuto aplikaci, která na serveru bude pracovat v kontejneru. Budeme opět potřebovat webový server pro poskytování naší aplikace uživateli. K tomuto účelu zvolíme Nginx [29], stejně jako v části pojednávající o nasazení API. Tento webový server bude fungovat jako jistá proxy předávající všechny dotazy naší Django [50] aplikaci. Nasazení bude řešeno formou docker-compose [36], pomocí kterého bude spouštěna samotná aplikace, Nginx proxy a samozřejmě i zmiňovaná PostgreSQL [33] databáze.

Bližší informace o administraci kontejnerů jsou popsány v přiložené administrátorské příručce D.

5. Řešení REST API

V předchozích kapitolách jsme si prošli různá rozhodnutí a možné přístupy ke tvorbě našeho API. V této kapitole si představíme výsledek, který vzešel ze všech zmíněných rozhodnutí.

Jak jsme již zmínili, API bude rozděleno do dvou separátních aplikací implementovaných s využitím webového frameworku Flask pro jazyk Python a několika dalších knihoven, které nám například umožní využití SAP DI API. Vysoko úrovňové schéma je naznačeno na obrázku 3.2. Zároveň přikládáme i sekvenční diagram 5.1, na kterém je pro lepší pochopení naznačena komunikace uživatele volající `GET` funkci na získání dostupnosti produktů.

5.1 Návrh Master aplikace

Naše tzv. *Master aplikace* bude sloužit pouze jako hlavní komunikační jednotka s databází a SAP B1 DI API. Nebude se tedy muset starat o ověřování uživatelů a můžeme aplikaci vytvořit co nejkompaktněji. Tím splníme jeden z požadavků, neboť si *Firma* nepřeje instalovat jiné, než nezbytně nutné knihovny do serveru obsluhující SAP.

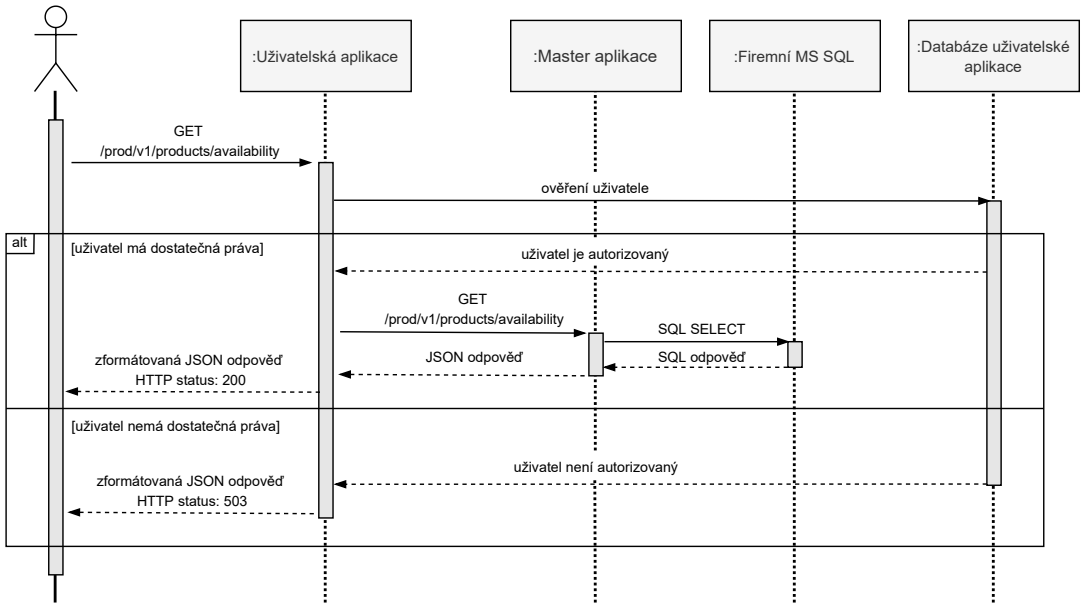
Aplikace bude tedy primárně využívat „wrapper“ DI API pro Python. Konkrétně se jedná o knihovnu `Flask-SAPB1` [53] pracující jako rozšíření `Flask`. Knihovna obsahuje jak adaptér do Microsoft SQL databáze (za využití knihovny `pymssql` [54]), tak i do SAP B1 s využitím „přeloženého“ DI API popsaného v programátorské části.

API by mělo i mimo jiné umožňovat připojení na testovací, ale i produkční databázi. Budeme chtít zprostředkovávat co nejjednodušším způsobem identické funkce. Vytvoříme tedy dvě instance `Flask` aplikace. To, která instance se zvolí, bude rozhodnuto na základě volané URL.

5.2 Návrh Uživatelské aplikace

Tato část API, tedy *Uživatelská aplikace*, bude primárně plnit úlohu komunikace s koncovým uživatelem API. Ověřuje uživatele a předává dotazy směřující na API *Master aplikaci*. K tomu, jak jsme dospěli v analýze řešení 3, využijeme relační databázi jako formu úložiště dat o našich uživatelích. Pro lepší práci s databází využijeme knihovnu `flask_sqlalchemy` [55], jež nám umožní vytvořit abstrakci formou *modelů* nad tabulkami a vztahy v databázi formou standardních Python tříd. Společně s knihovnou `flask_migrate` [56] nám tak usnadní správu a práci s databází.

Aplikace jako taková bude primárně využívat framework `Flask` s knihovnou `flask_restx` [57]. Flask RESTX nám totiž velice přímočaře umožní vytvořit REST API společně s automaticky vygenerovanou `Swagger` dokumentací.



Obrázek 5.1: Sekvenční diagram vysokoúrovňové komunikace API

5.3 Programátorská dokumentace Master aplikace

V této sekci bude popsáno, jak byla vytvořena *Master aplikace* našeho REST API.

5.3.1 Struktura projektu

Základní část projektu je založena na centrálním skriptu, který vytváří instance `Flask` objektů, soubor definující prostředí našeho API, a poté již samotné funkce využívající adaptéry do databáze a SAP. Celý projekt je rozdělený do souborů s následující strukturou:

```
SAPB1/
    flask_sapb1/
        __init__.py
        flask_sab1.py
        SAPbobsCOM90.py
    v1/
        __init__.py
        sapb1api.py
    app.py
config.py
config_dev.py
manage.py
server.py
```

Popišme si nyní strukturu aplikace po jednotlivých složkách a skriptech v nich umístěných.

Kořenový adresář

Tato část projektu obsahuje primárně hlavní spouštěcí a konfigurační skripty.

- **server.py** obstarává způsob, pomocí kterého jsme schopni prostřednictvím knihovny `CherryPy` [58] zpřístupnit testovací a produkční přístupy na základě parametru v URL při volání API. Spouští totiž samotné instance `Flask` aplikací popsané v další části.
- **config*.py** přináší konkrétní parametry připojení do databáze i SAP. Je rozdělený na produkční a testovací přístupy.

SAPB1

V této složce a jejích podsložkách je již definována logika samotného API. Obsahuje dvě podsložky, v jedné z nich je definováno prostředí API z pohledu „uživatele“ (tedy *Uživatelské aplikace*, která jako jediná bude s Master aplikací komunikovatna) a ve druhé již samotné funkce využívány k dotazovaní a zápisu dat.

- **app.py** vytváří instance `Flask` aplikací jenž jsou využívány ve vyšších úrovních projektu.

SAPB1/v1/

Je složka obsahující „uživatelské prostředí“ API. Definuje totiž funkce tak, jak je vidí uživatel.

- **___init___.py** definuje „URL dispatcher“ API. Tedy to, na jakou URL se mapuje která funkce.
- **sapb1api.py** obsahuje právě konkrétní funkce, které „URL dispatcher“ volá.

SAPB1/flask_sapb1/

Přináší definici adaptéru jak pro databázi, tak i pro SAP.

- **flask_sapb1.py** jedná se o rozšíření knihovny `Flask-SAPB1` poskytující oba adaptéry jak do MS SQL databáze, tak i do SAP. Obsahuje tak převážně funkce volané z `SAPB1/v1/sapb1api.py`.
- **SAPbobsCOM90.py** je zmiňovaný „wrapped“ SAP DI API. obsahuje všechny objekty SAP B1 převedené do Python tříd odvozených od objektů z knihovny nazývané `win32com` [59]. Díky tomu můžeme přímočaře přistupovat k SAP objektům a všem polím na daném objektu.

5.3.2 REST API

Logika REST API je implementována prostřednictvím frameworku `Flask` s využitím `CherrPy`. Jednotlivé API funkce jsou implementovány v souboru `SAPB1/v1/sapb1api.py`. Tyto API funkce jsou definovány jako HTTP metody `get`,

post, put, atp. Ty z HTTP Request, který je součástí Flask, v případě potřeby získají tělo s daty prostřednictvím funkce get_json(), jež je právě na objektu request definována.

Funkce v souboru SAPB1/v1/sapb1api.py pracují jen jako vstupní „brána“ do API. Hlavní logika se koná ve funkcích definovaných ve třídě SAPB1Adaptor v souboru SAPB1/flask_sapb1/flask_sapb1.py. Právě tato třída díky svým dalším funkcím implementuje veškerou logiku komunikace se SAP objekty a databází.

app.py

Skript app.py zajišťuje tvorbu instancí Flask objektů, ke kterým je následně přistupováno. Funkce create_app a create_app_dev načtou příslušný konfigurační soubor spustí se. Ke každé z nich poté díky cherrypy přistupujeme na odlišné URL.

SAPbobsCOM

Konkrétně soubor SAPB1/flask_sapb1/SAPbobsCOM90.py, který obsahuje hlavní logiku DI API, resp. objekty SAP převedené z SDK SAP B1 SAPbobsCOM90.dll. Jedná se o třídy odvozené od objektu DispatchBaseClass z knihovny win32com.
client. Mezi těmito objekty se nacházejí i obchodní partneři, či předběžné doklady využívané v API. Dává nám mimo jiné i přístup k uživatelským tabulkám i všem polím na SAP objektech, vč. těch uživatelských objektů jako kdyby se jednalo o proměnné na standardní Python třídě.

pywin32

Program využívá knihovnu pywin32, pomocí které je možné ovládat Microsoft aplikace prostřednictvím *Window's Component Object Model (COM)* [60] [61].

5.3.3 Připojení DI API

Připojení k SAP je zajištěno prostřednictvím DI API resp. Python „wrapperu“ v knihovně Flask-SAPB1 třídou SAPB1Adaptor definované v souboru SAPB1/flask_sapb1/flask_sapb1.py prostřednictvím funkce connect(con_type=None). Ta jednoduše předá data z config kontextu aktuální aplikace a vytvoří připojení díky comAdaptor().

5.3.4 Připojení MS SQL

Obdobně jako DI API, je připojení k SAP zajištěno prostřednictvím knihovny Flask-SAPB1 třídou MSSQLCursorAdaptor definované v souboru SAPB1/flask_sapb1/flask_sapb1.py opět využitím funkce connect(con_type=None). Jednoduše předá data z kontextu aktuální aplikace, konkrétně daného config, a vytvoří připojení pomocí cursorAdaptor().

5.3.5 Testovací a produkční prostředí

Naše API by mělo umožňovat výběr mezi přístupem do produkční a testovací databáze. To je zařízeno tak, že ve své podstatě vytváříme dvě `Flask` aplikace spravovány prostřednictvím `CherryPy` v souboru `./server.py`. Ten umožňuje tyto dvě instance připojit prostřednictvím funkce `graft`.

5.3.6 Zápis předběžného dokladu do SAP

Zápis předběžného dokladu, nebo-li „Draftu“ v terminologii SAP, je zajištěn třídou `DraftInsertAPI`. Tato třída implementuje metodu `put` volající funkci `SAPB1Adaptor.insert_business_partner`.

Zde máme implementovanou logiku pro zprostředkování `BusinessObject oDrafts` využitím `COM adapteru`. Na tomto objektu poté v cyklu přes vstupní slovník (deserializovaný JSON) s párem (klíč, hodnota) kontrolujeme jednotlivé klíče a v případě existence na objektu `oDrafts` přiřadíme danou hodnotu. Využíváme k tomu funkci `setattr` [62], která nám umožňuje dle klíčů v zaslaném JSON přiřazovat konkrétní hodnoty stejně označeným polím na objektu `oDrafts`.

Nakonec již zavoláme pouze naši funkci `_add()`, která v případě chyby vrátí námi definovanou výjimku `AdaptorError`, vracející string s chybou.

V případě kladného zápisu do SAP vrátíme tuto informaci s kódem `201`. V opačném případě se pokusíme vrátit konkretní chybu, která při zápisu může nastat.

5.3.7 Zápis obchodního partnera do SAP

Zápis obchodního partnera, nebo-li „BusinessPartnera“, v terminologii SAP je zajištěna třídou `BusinessPartnerCardAPI`. Tato třída implementuje metodu `put` volající funkci `SAPB1Adaptor.insert_business_partner`.

Zde máme implementovanou logiku, jež si prostřednictvím `COM adapteru` zprostředkuje `BusinessObject oBusinessPartners`. Na tomto objektu poté již v cyklu přes vstupní slovník (deserializovaný JSON) s párem (klíč, hodnota) kontrolujeme jednotlivé klíče a v případě existence na objektu `oBusinessPartners` přiřadíme danou hodnotu. Využíváme k tomu opět funkci `setattr` [62], která nám umožňuje dle klíčů v zaslaném JSON přiřazovat konkrétní hodnoty stejně označeným polím na objektu `oBusinessPartners`.

Nakonec již zavoláme pouze naši funkci `_add()`, která v případě chyby vrátí námi definovanou výjimku `AdaptorError` vracející string s chybou.

V případě kladného zápisu do SAP vrátíme odpověď s kódem `201`. Pokud se zápis nepodaří, pokusíme se vrátit konkretní chybu, která při zápisu nastala.

5.3.8 Příklad funkce GET

V předchozí podsekci jsme si představili funkce pro zápis do SAP. K tomu jsme využili DI API tak, jak bylo zamýšleno. Již v analýze našeho řešení 3.1.1 jsme zmínili, že k čerpání dat, nebo-li `GET` funkcím, budeme využívat přímé napojení na databázi prostřednictvím `ODBC` konektoru. Pojďme si ukázat příklad, jak je taková funkce implementována.

Opět si vytvoříme či rozšíříme příslušnou třídu v souboru `SAPB1/v1/sapb1api.py` po vzoru ostatních a přidáme příslušnou `get()` funkci, ze které budeme volat funkci v objektu `sapb1Adaptor`, který je definovaný v `SAPB1/flask_sapb1/flask_sapb1.py`. Ten rozšíříme o příslušnou funkci, ve které si jako parametrizovaný string zadefinujeme náš SQL dotaz. Poté nám stačí zavolat funkci `cursorAdaptor()` definovanou ve stejném objektu `SAPB1Adaptor`. Ta se pokusí připojit k databázi a nebo vrátit již existující připojení. Obojí je však reprezentováno objektem `MSSQLCursorAdaptor`, který nám zaobaluje funkce knihovny `pymssql`[54], kterou poté využíváme k přímému přístupu do databáze. Celý dotaz pak vypadá jako `self.cursorAdaptor.sqlSrvCursor.execute(dotaz)` [63], kde `dotaz` je zmíněný string. s „cursorem“, a s tím se poté pracuje již standardně jako při obyčejném použití knihovny `pymssql` [54]. Můžeme tedy standardně využívat funkce jako `fetchall()` [64] a podobné.

5.4 Programátorská dokumentace Uživatelské aplikace

V této části si popíšeme způsob, kterým byla uživatelská aplikace naprogramována. Jedná se opět o projekt ve frameworku `Flask` obstarávající komunikaci mezi uživatelem a *Master* aplikací. Tento projekt má tedy na starost ověření uživatele, správu rolí a volání správných funkcí, vč. předání těla požadavku při komunikaci uživatel → API nebo API → uživatel. V projektu jsou využity zajímavé knihovny usnadňující práci s databází, jako `Flask-SQLAlchemy` [55] a `Flask-Migrate` [56]. Jejich využití si popíšeme níže.

5.4.1 Struktura projektu

Náš projekt je rozdělený do logických celků primárně dle jejich účelu. V nejvyšší vrstvě dělíme projekt do tzv. `resources`, tvořící jednotlivé kategorie funkcí API a na část `server` starající se o o celkovou konfiguraci aplikace, napojení na databázi a definici objektů, resp. modelů využívaných v aplikaci. Celá struktura projektu vypadá následovně:

```
├── environment/
│   └── instance.py
├── migrations/
├── resources/
│   ├── authentication.py
│   ├── decorators.py
│   ├── drafts.py
│   ├── general.py
│   ├── orders.py
│   ├── partners.py
│   └── products.py
└── server/
    └── instance.py
```

```
    └── models.py
├── main.py
└── wsgi.py
```

Resources

Jednotlivé „`resources`“ tvoří sadu funkcí obstarávající funkcionalitu API. Starají se totiž o to, aby veškerá data přijata na konkrétní adresu představující funkci v API byla předána naší hlavní *Master* aplikaci a její odpověď byla opět vrácena uživateli. Resources máme rozdělené, až na jednu výjimku, do souboru představující jednotlivé skupiny funkcí API. Pojďme si je postupně představit.

- **Authentication.py** tvoří skupinu funkcí, zabezpečující logiku za tvorbou nových uživatelů, změnu hesel a přiřazování a odebírání uživatelských rolí.
- **Decorators.py** se svým zaměřením vymyká ostatním kategoriím, avšak jedná se o velice důležitou část přinášející dekorátory [65] funkcí. Čtenář by však neměl hledat podobnost s návrhovým vzorem „dekorátor“, představeným v tzv. Gang of Four knize [66]. V tomto kontextu se jedná o standardní Python funkce umožňující ovlivnit chování jiné funkce, které jsou volány prostřednictvím syntaxe decorator. Tímto způsobem funkci prakticky „obalíme“ do dekorátoru, rozšíříme její funkcionalitu a udržíme tím kód čistý bez nutnosti rozšiřování konkrétní funkce. V našem případě jsme jako „dekorátory“ využili funkce ověřující samotného uživatele, resp. jejich přihlašovací jméno a heslo, ověřující role uživatele a sledování jednotlivých požadavků uživatelů.
- **Drafts.py** - jak název napovídá, jedná se o část implemetující logiku pro zápis předběžného dokladu do SAP.
- **General.py** implementuje pomocné funkce sloužící k validaci JSON objektů zasílaných v těle požadavků a formátování odpovědí API. Přináší tak funkci `response_parse`, která upraví odpověď od *Master* aplikace tak, aby byla v jednotném a srozumitelném formátu čitelná uživatelem. Funkce `validate_body_model` naopak rekurzivním přístupem kontroluje JSON soubory zaslané v těle požadavku oproti definovanému schématu u každé funkce. Tuto funkci však využíváme pouze v případě, kdy je pevně daná struktura těla volání. Například u zápisu předběžného dokladu a obchodního partnera do SAP je struktura volání závislá na objektu v SAP, kde samotná validace proběhne při pokusu o zápis.
- **Orders.py** je prostředí přinášející možnosti, jak získat informace o odchozích objednávkách a zapsat aktualizace objednávek do SAP. Jedná se o prostředí využívané aplikací pro komunikaci s přepravními společnostmi, popisované v této práci.
- **Partners.py** je skupina funkcí zpracovávající požadavky pro zápis obchodního partnera a kontrolu GDPR stavu na kartě obchodního partnera.
- **Products.py**, jak název napovídá, je prostředí pro získávání informací o produktech *Firmy*.

Server

Je část projektu implementující konfigurace Flask aplikace a datovou strukturu databáze.

- **instance.py** máme popsanou kompletní konfiguraci aplikace vč. nastaveného „logování“, a samozřjemě i napojení do databáze za využití `SQLAlchemy` popsané dalším bodě.
- **models.py** definuje objekty primárně pro správu uživatelských účtů uložených v databázi a dává nám tak příjemný interface do databáze. Tomuto přístupu se říká `ORM` a popsali jsme si ho v části textu pojednávající o frameworku `Django`.

5.4.2 Modely

Jak jsme se již obeznámili, objekty, zejména pro správu uživatelských účtů, jsou definovány v souboru `/server/models.py`. Jedná se o způsob, pomocí kterého definujeme databázovou strukturu prostřednictvím Python tříd odvozených od objektu `SQLAlchemy.Model`. V práci jsou zavedeny následující modely:

- **Role** definující uživatelskou roli.
- **User** uživatel nesoucí informace jako `username`, `email`, `password_hash`, pole rolí `roles` a `card_code`, které může být využito v budoucnu pro ještě vyšší možnost ověření práv v případě většího rozšíření mezi koncové uživatele.
- **Request** je záznam požadavku na API udržující informace jako `user_id`, `ip_address`, `date` a `request_type` jakožto konkrétní funkci.

5.4.3 Migrace

Abychom mohli databázi určitým způsobem verzovat, využíváme konceptu migrací. K tomuto účelu jsme aplikaci doplnili o knihovnu `Flask-Migrate` [56], jenž spolupracuje přímo s `SQLAlchemy` databází a umožňuje nám zaznamenávat jednotlivé verze databáze. Tento přístup se nám může hodit ve chvíli, kdy dojde k problému v aplikaci a budeme se nejen potřebovat vrátit ve verzi „kódu“, ale i k verzi kompatibilního schématu tabulek databáze.

5.4.4 Autorizace a autentizace

Veškeré ověřování uživatelů je prováděno čtením informací v hlavičce každého požadavku na API. Flask hlavičku serializuje do objektu, ve kterém přistupujeme pouze k poli `username` a `password`. Za účelem ověření přihlašovacích údajů uživatele vznikl dekorátor `authorize` definovaný v souboru `resources/decorators.py`. Abychom mohli podobným způsobem ověřit i práva přistupujícího uživatele, vytvořili jsme ve stejném souboru dekorátor s parametrem `allowed_roles` ověřující dostatečnou shodu práv.

5.4.5 Struktura API funkcí

Obecně jsou API funkce velice podobné. Protože se jedná pouze o logiku „prostředníka“, nebylo nutné, aby *Uživatelská* aplikace implementovala jakoukoli větší funkcionalitu, než byla doposud zmíněna. K definici jednotlivých funkcí využíváme knihovnu `flask_restx` [57], která nám umožňuje přímo způsobem, kterým API funkce definujeme, postupně vytvářet `Swagger` dokumentaci. Každá ze skupin funkcí API začíná definicí tzv. `namespace` sdružující jednotlivou funkcionalitu v kategoriích a na konkrétních URL prefixech. Uživatelské „namespacy“ obsahují parametr `env` směřující dotazy na testovací či produkční databázi a SAP. Následně jsou jednotlivé skupiny funkcí definovány ve třídách odvozených od objektu `Resource`, definovaného ve zmíněné knihovně. V těchto třídách poté definujeme funkce jako `get`, `post` apod., do kterých jsou poté směřovány jednotlivé požadavky na API dělené dle HTTP metody. Knihovna obecně pracuje s konceptem dekorátorů tak, jak byly popsány výše. Můžeme tak jednoduše definovat jednotlivé odpovědi vč. struktury dané odpovědi prostřednictvím objektu `model`[1], ve kterém si definujeme vzorovou strukturu těla požadavku i odpovědi dané funkce. Na základě tohoto „modelu“ se poté vytvoří ve `Swagger` dokumentaci uživatelsky čitelný vzor. Velikým neduhem tohoto přístupu však je to, že je velice objemný a „modely“ nejde příliš dobře zanořovat do sebe. Ani samotné pole nejdou velmi dobře definovat, a tak jsou tyto modely programátorovi spíše na obtíž a jejich tvorba je poměrně časově náročná. Tento koncept v knihovně je totiž poměrně stručně zadokumentovaný a nelze k němu nalézt příliš mnoho informací.

Způsob volání API funkcí vč. zavedení nového uživatele je popsaný v uživatelské příručce A.

[1] Zde dochází k mírnému „křížení“ pojmů, nejedná se o stejné chápání pojmu „model“, jako přináší SQLAlchemy.

6. Řešení aplikace pro odesílání zásilek

V předchozích kapitolách jsme si navrhli možná řešení a přístupy ke tvorbě aplikace pro odesílání zásilek pro *Firmu*. Ze všech návrhů jsme si zvolili variantu, kdy bude aplikace koncipována jako webové s frameworkem Django. Umožní nám to mimo jiné jednodušší rozšiřitelnost v rámci *Firmy* a zároveň nám to pomůže s jednodušším doručování změn v programu. Aplikace bude sloužit i jako názorná ukázka použití vytvořeného API v praxi.

6.1 Architektura

U aplikací tohoto ale i podobného typu je vhodné, aby bylo uživatelské prostředí bylo co nejvíce separované od logiky aplikace a staralo se tak pouze o interakci s uživatelem. Umožní nám to tak lépe oddělit *back-end* a *front-end*. Změna v uživatelském prostředí tedy neovlivní logiku naší aplikace a naopak změna v logice resp. na pozadí aplikace nebude muset nutně vyžadovat úpravu uživatelského prostředí.

Abychom mohli co nejvíce od sebe logickou a uživatelskou vrstvu oddělit a zjednodušit tak vývoj aplikace, rozhodli jsme se využít koncepty návrhového vzoru **Model-View-Controller** [1], neboli ve zkratce MVC. Ten je v určité formě implementovaný ve frameworku Django, jenž nám právě s využitím a dodržením tohoto vzoru pomůže.

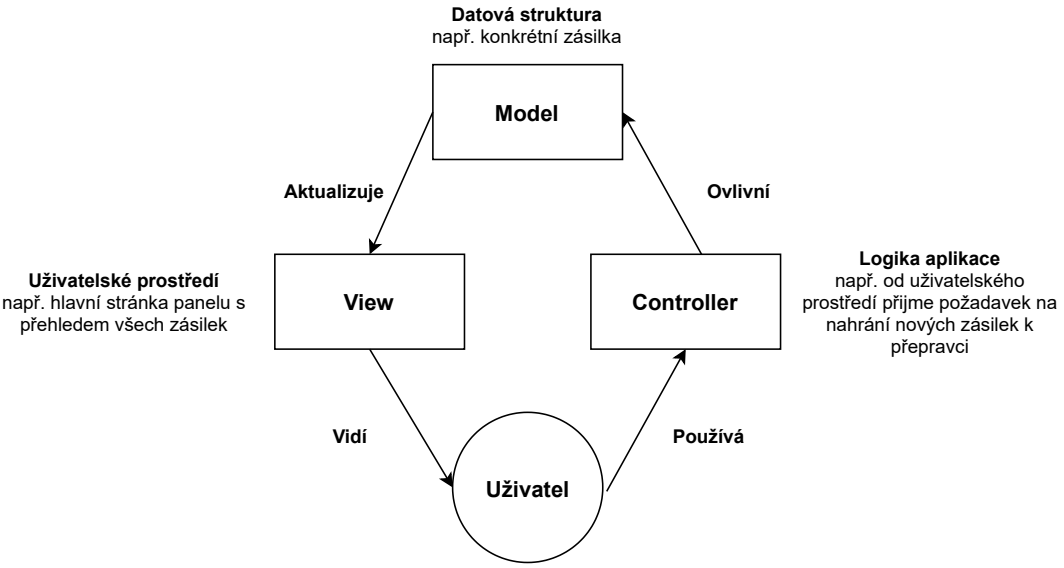
V této sekci si MVC včetně framewokru Django představíme blíže a ukážeme si, jakým způsobem je zmíněný vzor ve frameworku implementovaný a jak se liší od jeho „učebnicové“ implementace.

6.1.1 Model-View-Controller [1]

Stejně jako například architekturu významné stavby lze zpravidla zařadit pod nějaký obecnější umělecký směr, lze podobné tvrdit i o software (ve velmi přeneseném smyslu). Barokní kostely může například spojovat doba vzniku okolo 17. století, využití sloupů či kopulí. Obecně bude dané architektonické typy spojovat provedení, důvod, motivace či doba vzniku. Podobné by se dalo uvést i u software. Jelikož se myšlenka či možné řešení může jistým způsobem podobat, dá se návrh software občas řídit principem nějakého návrhového vzoru. Ten nám dává obecné řešení k tomu, jak samotnou aplikaci navrhnout.

Jak již bylo zmíněno na začátku této sekce, naše aplikace se bude řídit principy MVC. To nám umožní efektivně a čistě oddělit logiku aplikace a uživatelské prostředí. Ač se implementace mohou mírně lišit, logika vzoru Model-View-Controller zůstává stejná. Kód je rozdělený do třech separátních částí:

- **Model** je hlavní komponentou celého vzoru. V objektově orientovaném programování by se dal připodobnit ke konkrétním objektům či třídám v aplikaci. Pro nás bude model reprezentovat například konkrétní zásilku, přepravní společnost a stav konkrétní zásilky. Zpravidla bude model nejméně



Obrázek 6.1: Diagram MVC

měněnou komponentou MVC během života dané aplikace.

- **View** je část starající se o reprezentaci konkrétních dat modelu. Jedná se tak o to, co uživatel vidí, resp. co z našeho programu vystupuje. V našem případě se bude jednat o konkrétní stránky, které se uživateli zobrazí po zadání správné URL.
- **Controller** je část vzoru starající se o propojení Modelu a View. Získává uživatelský vstup či nějaké příkazy a ty poté formuluje dvěma zbylým částem. Ne vždy je nutná komunikace s oběma komponentami, například filtrování konkrétních zásilek nemusí nijak upravovat Model, pouze jen na své úrovni vybere ty odpovídající a prostřednictvím View je zobrazí uživateli.

Jak tyto tři části interagují, je ukázáno na klasickém schématu MVC přiloženém na obrázku 6.1.

6.1.2 Vztah MVC a frameworku Django

Jak samotní autoři frameworku uvádějí v dokumentaci [67], Django se nedrží názvů MVC a může to tak být matoucí, neboť *View* a *Controller* ve frameworku nejsou konkrétně pojmenovány. Django by proto spíše neslo akronym návrhového vzoru „MTV“. „Model“ nám zůstává, „Template“, který ve frameworku představuje „View“ z MVC. „View“ ve smyslu Django právě představuje část, kde by se měla nacházet logika naší aplikace. Zprvu se to může zdát matoucí, ale pojďme se podívat blíže na jednotlivé části Django. Autoři tvrdí, že pro ně nemělo smysl se držet názvů MVC, udělali to tak, jak se jim to zdálo rozumné.

- **Model (Django)** [68] je v Djangu to samé, jak jsme si již popsali v předchozí sekci.
- **Template (Django)** - jedná se o konkrétní prezentace toho, co uživatel vidí. Mohou to například být HTML soubory, které udávají nějakou formu toho, jak se data uživateli zobrazí.

- **View (Django)** Určuje, která data se zobrazí v „Template“. Ve „View“ se již standardně nachází jednoduchá logika a předávají se data do „Template“, který je poté zformátuje. Views zpracovávají příchozí HTTP requesty. Je zde již paralela s klasickým MVC, protože nám to přesně udává oddělení aplikační logiky od uživatelské prezentace.

Paralela je zřejmá, ale vynechali jsme „Controller“. Jak stojí v Django dokumentaci [69], víceméně celé Django představuje z určitého úhlu pohledu „Controller“. Jedná se o to, co se děje mezi tím, kdy uživatel pošle požadavek na určitou URL a nalezením toho správného Django „View“, které by se mělo spustit.

6.2 Programátorská dokumentace

V této sekci si popíšeme, jak a za pomocí čeho byla aplikace vyvíjena, strukturu projektu a způsob využití externích API přepravních společností.

6.2.1 Použité technologie

Hlavní technologií, pomocí které byla naše aplikace vytvořena, je bezesporu framework Django. Jak již bylo uvedeno, jedná se o sadu knihoven pro jazyk Python, umožňující rychlou a efektní tvorbu webových aplikací. Mezi další technologie, které jsme využili, patří:

- **PostgreSQL** zvolený objektově-relační databázový systém. Výhoda této volby spočívá i v plné podpoře Django ORM.
- **HTML** značkovací jazyk pro tvorbu šablon rozšířených šablonovacím enginem, který Django obsahuje.
- **CSS** jako jazyk pro popis zobrazení elementů v HTML.
- **JavaScript** skriptovací jazyk využitý pro dynamickou manipulaci s DOM a volání Django funkcí na views front-end.
- **Bootstrap** sada nástrojů pro CSS k rychlé a jednoduché tvorbě vzhledného a responzivního front-end. Pro vývoj samotné aplikace bylo využito IDE **Visual Studio Code** od Microsoft a jako grafické rozhraní pro prohlížení a úpravu dat v databázi aplikaci **TablePlus**.

6.2.2 Struktura projektu

Každý projekt Django začíná příkazem `django-admin startproject nazev-projektu`. Tento projekt vytvoří základní Django skripty a konfigurační soubory s následující hierarchií (pro lepší představu):

```
nazev-projektu/
├── manage.py
└── nazev-projektu/
    ├── __init__.py
    ├── settings.py
    └── urls.py
```

```
    └── wsgi.py
```

Rozeberme si postupně jednotlivé složky a skripty. Na názvu vnější kořenové složky `nazev-projektu/` nijak Djangu nezáleží. Skript `manage.py` je pro Django velice důležitý, neboť nám umožňuje s Djangem interagovat. Jedná se o nástroj na příkazové řádce, který nám umožňuje spouštět migrace databáze či samotný projekt, využívat tzv. **shell**, jenž nám dovoluje na příkazové řádce psát kód, jako kdyby byl součástí naší aplikace. Soubor `nazev-projektu/settings.py` slouží jako základní konfigurace celého projektu. Mimo jiné umožňují i určitým způsobem definovat globální proměnné, které mohou například reprezentovat některé důležité složky, soubory či konfugrace v našem projektu následně využitelné v kódu. Skript `nazev-projektu/urls.py` nám dovoluje definovat URL schéma využívající URL dispatcher obsažený v Django. Ke každé uvedené URL se tak může „napárovat“ nějaké view a na té adrese se uživateli zobrazí. Skript `nazev-projektu/wsgi.py` je vzorový soubor pro spuštění aplikace na WSGI kompaktibilním serveru. Základní struktura naší aplikace vypadá následovně. Nejdůležitější složky si popíšeme do detailu:

```
zasilky/
├── manage.py
├── zasilky/
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
├── app/
│   ├── management/
│   │   └── commands/
│   │       ├── remove_redundant_states.py
│   │       ├── update_new_ceskaposta.py
│   │       ├── update_new_zasilkovna.py
│   │       ├── update_old_ceskaposta.py
│   │       └── update_old_zasilkovna.py
│   ├── migrations/
│   ├── __init__.py
│   ├── admin.py
│   ├── models.py
│   ├── urls.py
│   └── views.py
├── ceskaposta/
├── zasilkovna/
├── ulozenka/
├── upload/
├── static/
├── static_files/
└── cert/
```

6.2.3 Django aplikace

Projekty Django jsou složeny z aplikací vytvořených pomocí příkazu `./manage.py startapp nazev-aplikace` [70]. Je to doporučený způsob, jak separovat kód do dílčích logických celků. V našem případě jsme tak byli schopni oddělit uživatelské prostředí formou aplikace `app` od částí komunikujících s externími API přepravců jako aplikace `ceskaposta`, `zasilkovna`, `ulozenka`.

Nyní si postupně rozeberme všechny „aplikace“, které jsou součástí našeho Django programu.

App

Je naše hlavní aplikace, která se stará o vyobrazení uživatelského prostředí. Jeho součásti uživatelského prostředí, o jehož zobrazení se stará `app/views.py`, jsou:

- **Přihlašovací stránka** využívající autentizaci Django. Jelikož framework přináší skvělé řešení pro uživatelské „sessions“ a obecně správu uživatelů, nebyl důvod pro „znovuobjevení kola“ a nevyužití prostředí pro přihlášení/odhlášení, které nám framework poskytuje. O zobrazení naší vlastní přihlašovací stránky a její funkcionality se stará třída `Login` s funkcí `loginView`.
- **Hlavní stránka** se po přihlášení zobrazí jako `/prehled`. Tato stránka je naznačena na obrázku 4.1 a obsahuje veškerou logiku pro filtrování zásilek. K jejímu zobrazení je nutné, aby byl uživatel přihlášený. O to se nám stará dekorátor `login_required` z knihovny `django.contrib.auth.decorators`.
- **Detail zásilky** se zobrazí na URL `/detail/<int:id_zasilky>`, kde `id_zasilky` je naše interní ID v databázi a se SAP nemá nic společného. Tato stránka je opět zobrazena pouze přihlášeným uživatelům ve velice podobném designu, jako je navržený na obrázku 4.2.
- **Odeslání zásilek** pracuje na URL `/odeslani`. Návrh stránky byl ukázán na obrázku 4.3. O většinu funkcionality této stránky se stará JavaScript, jenž volá funkce z `views` příslušných aplikací přepravních společností popsaných níže.

V rámci stejné aplikace jsou i všechny modely využívané v našem programu. Modely jsou definovány v `app/models.py` a budou popsány v následující podsekci.

Důležitou částí je i soubor `app/urls.py` definující všechny adresy našeho programu pro URL dispatcher. Jsou logicky rozděleny do 4 částí, kde první část definuje UI a další z nich se starají o volání funkcí aplikací popsaných dále. Definice URL vzoru je v Djangu velice jednoduchá. Ve své podstatě se jedná o pole `urlpatterns` obsahující v každém prvku funkci `path(route, view, kwargs=None, name=None)` z knihovny `django.urls`, jenž má za parametry `route` určující vzor adresy, se kterou může Django „počítat“, `view` definující konkrétní funkci, jenž má Django zavolat a po nalezení dané adresy a nepovinné parametry `kwargs` popsaný v [71].

`ceskaposta`

Další z našich aplikací je `ceskaposta` starající se o komunikaci s B2B API České Pošty. Hlavní částí této aplikace jsou `ceskaposta/views.py`. Naše aplikace je koncipována tak, že většina funkcí, jako je nahrání nových zásilek, jejich odeslání a opětovné nahrání do SAP, je přístupná na konkrétních URL a mohou se tak volat pohodlně z prohlížeče či JavaScriptu. Ve Views máme definované funkce obstarávající:

- Nahrání nových dat ze SAP.
- Nahrání nových zásilek k České Poště.
- Aktualizace stavu zásilek České Pošty (parametrizováno počtem dní jakožto „stářím“ objednávky.
- Získání adresních štítků pro zásilky České Pošty.
- Vygenerování soupisky zásilek České Pošty.
- Aktualizace stavu zásilek v SAP.

Protože je nepohodlné držet veškerou funkcionalitu ve skriptu `views.py`, byla většina aplikační logiky přesunuta do jednotlivých souborů nacházejících se ve složce `ceskaposta/logic/`. Těmi soubory jsou: `generate.py` starající se generování adresních štítků a soupisek, `prepareData.py`, které obstarává tvorbu XML odeslaných do API ČP, `retrieve.py`, jenž se stará o nahrávání zásilek ČP ze SAP a jejich ukládání do příslušného formátu v databázi, a nakonec `send.py` jako nejobsáhlejší soubor věnující se nahrávání nových zásilek k ČP a do SAP a průběžné aktualizaci starších zásilek.

`zasilkovna`

Aplikací věnující se komunikaci s API Packeta resp. Zásilkovna, je identicky pojmenovaná část našeho programu. Opět, hlavní částí aplikace je skript `zasilkovna/views.py`, jenž obstárává uživatelsky přístupnou funkcionalitu, kterou je:

- Nahrání nových dat ze SAP.
- Nahrání nových zásilek prostřednictvím PacketaAPI.
- Aktualizace stavu zásilek Zásilkovna (parametrizováno počtem dní jakožto „stářím objednávky“.
- Získání adresních štítků.
- Vygenerování soupisek.
- Aktualizace stavu zásilek v SAP.

Opět byla veškerá logika jednotlivých funkcí přesunuta do separátních souborů nacházejících se v `zasilkovna/logic/`. Jejich struktura víceméně kopíruje již popsanou logiku v části věnující se Django aplikaci `ceskaposta`. Více bude komunikace popsána v sekci věnující se PacketaAPI.

ulozenka

Rozvoj aplikace věnující se komunikaci s API Uloženka (v době psaní této práce již WE|DO) byl ukončen, neboť *Firma* rozvázala s přepravcem spolupráci. Součástí programu je základní a neupravená verze obstarávající veškerou potřebnou funkcionalitu jako u dvou výše popsaných přepravců. Refaktorování kódu nijak neprobíhalo, a tak se této aplikaci nebudeme vůbec věnovat.

6.2.4 Modely

Modely v Django jsou odvozené třídy od `django.db.models.Model`. Každý model může být reprezentovaný tabulkou v relační databázi a každý z atributů daného modelu představuje pole v dané tabulce. Díky využití modelů můžeme pomocí Django ORM, resp.automaticky vygenerovaného databázového API, jednoduše pracovat s daty v databázi prostřednictvím funkcí, které v kódu vypadají jako klasické funkce zapsané v jazyku Python. Django nám i dává velice dobrý „wrapper“ nad tzv. Many-To-Many vztahem. To je pro nás například vztah Zásilka + její stavy. Stačí nám, abychom si v Django vytvořili model reprezentující zásilku a model pro „stav“. Framework se pak postará o to, aby nám vytvořil párovací tabulku mezi konkretní zásilkou a jí přiřazeným stavem. Můžeme tak uživatelsky příjemně vytvářet jednotlivé třídy resp. modely, aniž bychom museli jakkoli řešit „omezení“ relačních databází. Django ORM nám dané vztahy přeloží tak, že pokud budeme mít konkrétní zásilku, její stavy se nám budou jevit jako její vlastní `list` obsahující prvky modelu reprezentující „stav“.

Modely, které v našem programu využíváme a jsou definovány v souboru `app/models.py`, jsou:

ShippingCompany

Jedná se o velmi jednoduchý model, pomocí kterého jsme schopni kategorizovat zásilky a filtrovat je tak vůči konkrétní přepravní společnosti. Tento model nese pole jako je název daného přepravce, parametrizovaný název využívaný pro filtrování (tj. místo „Česká Pošta“ hledáme jen „ceskaposta“) a pole RGB, jenž nám reprezentuje barvu přidělenou danému přepravci. RGB je poté využito v UI, aby bylo možné rychle a zřetelně opticky odlišit zásilky od jednotlivých přepravců. Definice modelu `ShippingCompany` je následující:

```
class ShippingCompany(models.Model):
    name = models.CharField(max_length=30)
    name_parameter = models.CharField(max_length=30, null=True)
    rgb = models.CharField(max_length=12)

    def __str__(self):
        return self.name
    class Meta:
        verbose_name = "Přepravní společnost"
    class Meta:
        verbose_name_plural = "Přepravní společnosti"
```

`Package`

Je naše „hlavní“ třída, představující danou zásilku nahranou ze SAP. Tento model nese pole:

- s kontaktními a dodacími údaji `customer_name`, `cusotmer_surname`, `customer_email`, `customer_phone`, `customer_street`, `customer_house_number`, `customer_town`, `customer_town`, `cusotmer_zip` a `customer_state`;
- informacemi o zásilce jako `parcel_weight`, `parcel_size`, `parcel_variable_num`, `parcel_cash_on_delivery`, `parcel_stated_price`, `parcel_insurance` a `parcel_currency`;
- informacemi o přepravní společnosti `shipping_company` a univerzálním proměnným jako `shipping_company_number` reprezentující sledovací číslo a `tracking_link` obsahující odkaz pro sledování zásilky u přepravní společnosti;
- a neboť je API každé přepravní společnosti odlišné a každý přepravce vyžaduje jiná specifická data, vytvořili jsme pro ČP pole `ceskaposta_prefix_parcel_code` reprezentující typ zásilky jako např. „DR“ či „RR“, `ceskaposta_service` obsahující doplňkovou službu ČP nahranou ze SAP, `ceskaposta_id_transaction` definující ID vrácené z asynchronní funkce pro nahrání zásilky k ČP;
- a pro PacketaAPI se pak jedná pouze o `zasilkovna_address_id` reprezentující id výdejny či zahraniční přepravní společnosti - v tomto smyslu je API Packeta velice příjemná na používání;
- binární příznaky zásilky `sent_to_sap`, `on_label`, `on_consignment_list`, `sent_to_api`, `error`;
- „pole“ stavů zásilky `status`, datum vložení do databáze `create_at` a datum aktualizace zásilky `update_at`;

`PackageStatus`

je opět velice jednoduchý model nesoucí časovou známku `create_at` a text obsahující „lidsky čitelný“ konkrétní stav zásilky `status`.

`Config`

je model nesoucí základní konfigurace pro přepravní společnosti, jako jsou například API klíče, apod. Tento přístup byl zvolený, aby bylo případně možné v budoucnu využívat více API klíčů k jednomu přepravci a nemuseli jsme je složitě ukládat v globálních proměnných na úrovni Django settings.

6.2.5 Autorizace a autentizace

Jak jsme již zmínili, pro přístup do aplikace je nutné přihlášení uživatele. Díky tomu, že Django ve své základní verzi obsahuje velice schopný systém pro uživatelské přihlášení, správu účtů a cookie sessions, nám, jakožto uživatelům

frameworku, pak stačí v případě potřeby definovat vlastní prostředí využívající předdefinovanou funkcionalitu z knihovny `django.contrib.auth`.

Přihlášení uživatele je tak zabezpečeno stránkou `/login` definovanou ve funkci `loginView` ve třídě `Login` se šablonou `templates/login.html`. Šablona obsahuje jednoduchý HTML formulář, pomocí kterého probíhá přihlášení zpracované zmíněnou funkcí. V případě chybně zadaných údajů je nad formulářem zobrazeno červené pole s textem informujícím o chybně zadaných přihlašovacích údajích.

Odhlášení uživatele pak probíhá požadavkem GET na adresu `/logout` reprezentovanou funkcí `logoutView` v třídě `Login`.

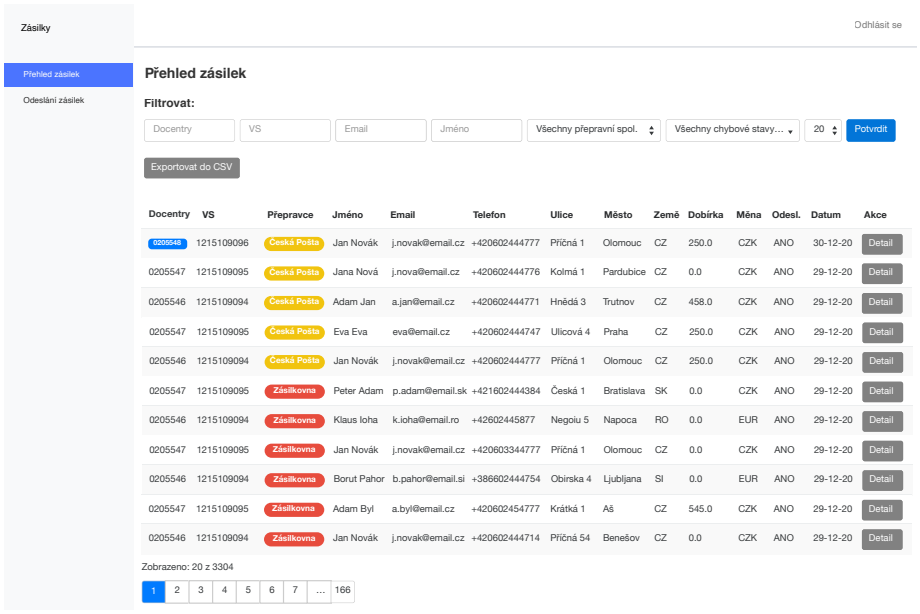
6.2.6 Úvodní stránka a filtrování

Po úspěšném přihlášení je uživateli zobrazena stránka `/prehled` (na stejnou adresu jsou přesměrovány i požadavky na adresu `/`). Jedná se o hlavní stránku reprezentovanou jako `View` funkce `prehledView` v `app/views.py`. Její základní šablona je v souboru `templates/prehled_zasilek.html` s jednotlivými komponentami definovanými ve složce `templates/components_prehled_zasilek`.

Na stránce se jako komponenta zobrazuje uživatelský filtr s parametry *Docentry*, *Jméno*, *VS*, *Email*, *přepravce*, *Chybové stavy* a *počet řádků na stránku*. Tato část stránky je definována v šabloně `templates/components_prehled_zasilek/filter_prehled_zasilek.html` Protože jsou na stránku všechny požadavky směrovány prostřednictvím požadavku GET, jsou filtry předávány v URL parametrech. Ve views parametry získáme z objektu `HttpRequest.GET`, který v Django představuje slovník všech HTTP GET parametrů. Pomocí těchto parametrů pak postupně řetězíme funkci `filter()` definovanou nad Django QuerySet, kterou aplikujeme jednotlivé filtry.

Pod hlavní tabulkou popsané v dalším odstavci, je v komponentě `templates/components_prehled_zasilek/pagination_prehled_zasilek.html` pak definovaný princip výpisu číslování stránek. Ve view si totiž, na základě velikosti výsledného QuerySet a parametrů `items_per_page` označující požadovanou velikost tabulky na stránku a `page` označující číslo aktuální stránky, jednoduchým algoritmem spočítáme, kolik je celkem stránek tabulky a které prvky QuerySet máme zobrazit. Manipulací s DOM stromem v JavaScriptu pak dokážeme zařídit to, že se nám výpis „dostupných“ stránek omezí jen na výpis, řekněme, předchozích 5 a následující 5 stránek. Algoritmus pro tuto úpravu stránky je popsaný v souboru `static_files/js/paginations.js` . Pro lepší představu je možné komponentu stránkování vidět dole pod tabulkou na obrázku 4.1. Smyslem této úpravy je to, aby se komponenta s přibývajícím počtem zásilek nerozšiřovala do šířky více než je šíře tabulky. Zbytek budou symbolizovat tři tečky viz 4.1.

Hlavní částí stránky je veliká tabulka obsahující základní informace o zásilkách. Obsah, resp. sloupce této tabulky je zbytečné detailně popisovat. Zobrazují totiž nutné informace k rychlé identifikaci zásilek. Vše ostatní je zobrazeno na detailu stránky. Čím je tabulka zajímavá je to, že na každém řádku je uvedený konkrétní přepravce, jehož název je obalený v `Boostrap pill badge` s přiřazenou barvou specifickou pro přepravce. Pomocí stejného prvku knihovny `Boostrap` označujeme i každý řádek, jenž byl do aplikace nahraný tentýž den. Avšak nyní ve výchozí modré barvě, která plně postačuje účelu a v prostředí se příjemně vyjímá.



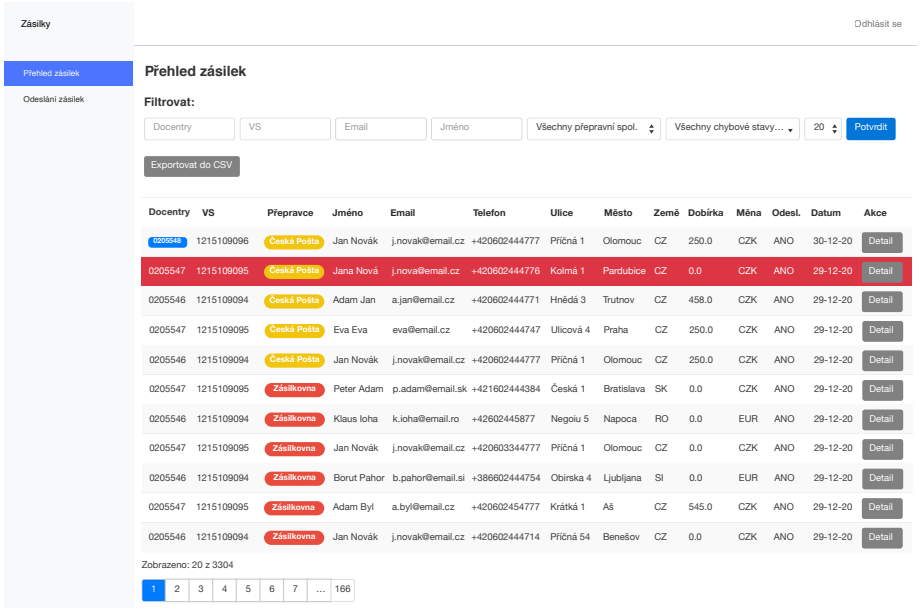
Obrázek 6.2: Uživatelské prostředí hlavní tabulky zásilek.

Pokud se uživatel rozhodne filtrovat konkrétního přepravce pomocí popsané komponenty, na každém řádku se zobrazí zaškrtávací pole. Toto pole nám umožňuje vybírat několik řádků najednou a provádět na nich operace specifikované komponentou nacházející se v souboru `templates/components_prehled_zasilek/actions_prehled_zasilek.html`. Zajímavostí, která stojí za zmínění v kontextu zaškrtávacích polí, bylo rozšíření funkcionality o výběr několika řádků najednou, jak jsou zvyklí uživatelé většiny tabulkových prohlížečů. Jednak v hlavičce tabulky je obsažené zaškrtávací pole, které vybere všechny řádky na stránce (viz funkce `toggle_all_checkboxes` v souboru `static_files/js/main.js`, a za druhé je možné, díky funkci `allow_group_select_checkboxes` ze stejného souboru, vybrat všechny zásilky ve specifikovaném intervalu. Stačí počáteční zásilku (nehledě na pořadí v tabulce) označit a za stisku tlačítka *SHIFT* označit poslední zásilku, jež má být součástí intervalu. Funkce se následně postará o výběr všech zásilek mezi nimi. S označenými zásilkami pak můžeme manipulovat prostřednictvím menu ve zmíněné komponentě `actions_prehled_zasilek.html`, která nám umožní pro vybrané zásilky vygenerovat adresní štítek či jejich soupisku. Musíme se samozřejmě omezit na konkrétního přepravce, protože nelze mít na jedné soupisce či adresním štítku kombinace zásilek od různých dopravců.

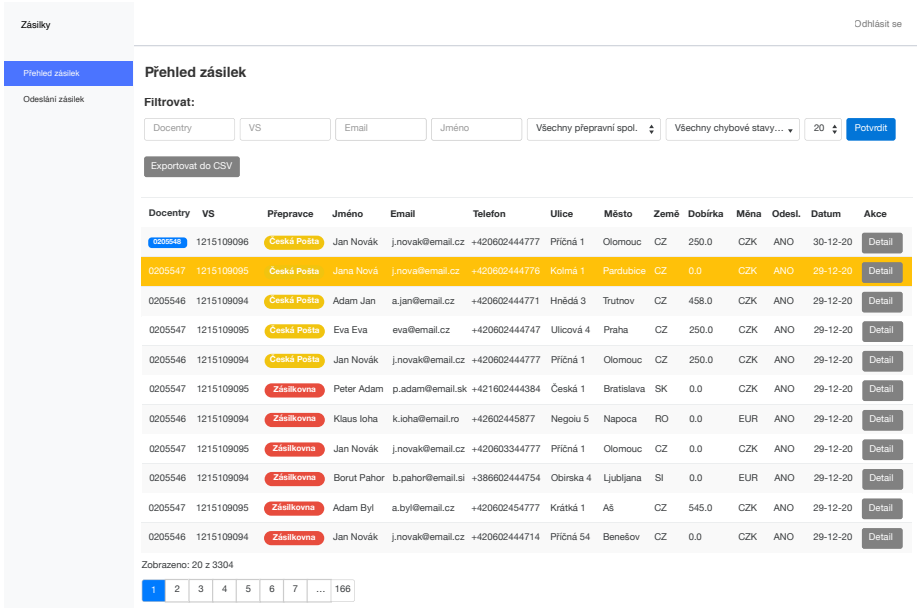
Finální návrh hlavní stránky je vyobrazený na obrázku 6.2. Vizualizace stavu, kdy se zásilka nepovede odeslat k přepravci, je zobrazena na obrázku 6.3. Zásilka, jež se nepovedla nahrát do SAP, je k vidění na obrázku 6.4. Více je uživatelské popsáno v jednotlivých situacích v uživatelské příručce v příloze B.

6.2.7 Export do CSV

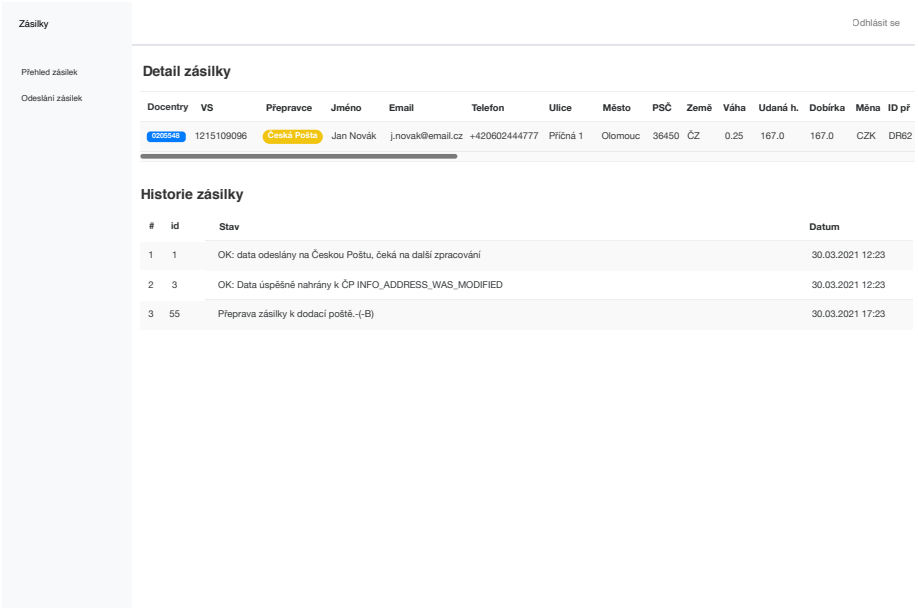
Jelikož byl jeden z požadavků, aby bylo možné aktuálně zobrazenou tabulku na hlavní stránce exportovat do CSV, je tlačítkem nad tabulkou odeslaný URL parametr `export=CSV`. Ten je zpracovaný ve funkci `prehledView`. Mezi částmi, kdy je QuerySet vyfiltrovaný na základě uživatelských parametrů a rozdělený do



Obrázek 6.3: Uživatelské prostředí hlavní tabulky zásilek s chybou nahrání zásilky k přepravci



Obrázek 6.4: Uživatelské prostředí hlavní tabulky zásilek s chybou nahrání zásilky do SAP



Obrázek 6.5: Uživatelské prostředí stránky pro detail zásilky

jednotlivých stránek, resp. jeho velikost omezena na chtěný počet, je za přítomnosti zmíněného parametru zavolána funkce `ExportHelper.dumpToCsv()` využívající knihovnu `djqcsv` umožnující vracení `HttpResponse` ve formě CSV s naším QuerySetem.

6.2.8 Stránka pro detail zásilky

Stránka zobrazující detail zásilky je dostupná na adrese `/detail/<int:id_zasilky>` reprezentující funkci `views.DetailDashboard.detailView` v `app/views.py`. Tato funkce zajišťuje zobrazení šablony stránky definované v souboru `templates/detail_zasilky.html`, do které je předána konkrétní zásilka. Jedná se v podstatě jen o výpis dat z databáze, proto je možné přenechat většinu práce template engine, který nám šablonu naplní specifikovanými daty.

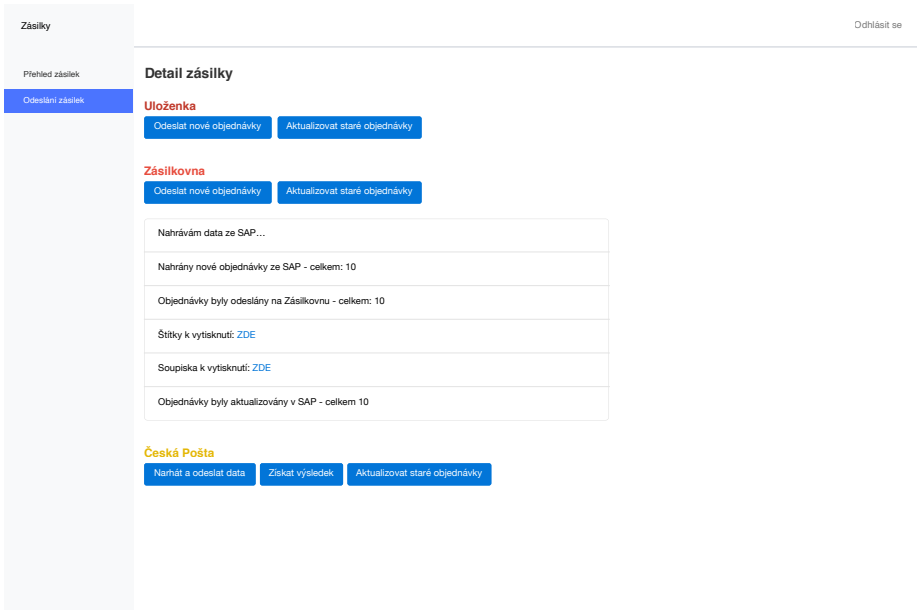
Vizualizace stránky je přiložena na obrázku 6.6.

6.2.9 Stránka pro odeslání zásilek

Stránka sloužící obsluze ke spuštění akce „odeslat data“. Back-end logika tohoto „view“ dostupného na adrese `/odeslani` je velice jednoduchá a volaná funkce `views.OdeslaniDashboard.odeslaniView` tak pouze zobrazuje stránku `templates/odeslani_zasilek.html`. Většina hlavní logiky se odehrává na front-end, odkud jsou prostřednictvím JavaScript knihovní funkce `fetch` funkce volány konkretní Django „views“ představující naše funkce pro nahrání zásilek ze SAP, odeslání zásilek k přepravci, atd.

Na stránce je poté, po spuštění většiny z funkcí, zobrazena tabulka mapující posloupnost jednotlivých akcí a odpovědí. Dává tak uživateli informace o tom, zda-li operace doběhla a co se během ní stalo, jako například kolik bylo nahraných zásilek ze SAP, kolik se podařilo odeslat zásilek k přepravci, atd.

Vizualizace stránky včetně zmíněné tabulky je přiložena na obrázku 6.6.



Obrázek 6.6: Uživatelské prostředí stránky pro odeslání zásilek

6.2.10 Nahrávání nových zásilek ze SAP

Program čerpá data prostřednictvím našeho API popsaného v předchozí části této práce 3. Pro každou přepravní společnost je vytvořena vlastní funkce v API, která vrací zásilky, u kterých doposud nebyl označený příznak „odesláno“. V našem programu poté musíme u každé přepravní služby udělat víceméně identické kroky. Implementace pro aplikaci `ceskaposta` a `zasilkovna` je až na detaily v jednotlivých polích identická. Můžeme se proto podívat pouze na implementaci funkce `Zasilkovna_retrieve_data.create_models(request)` v souboru `zasilkovna/views.py` volanou jako `/zasilkovna/create_models`. Prvním krokem je dotázat se API na nové zásilky (dle přepravce např. `/v1/orders/zasilkovna/new`) a poté již jednotlivé pole uložit do databáze, resp. vytvořit instance modelu `Package`. Každá zásilka má zmiňovaný unikátní SAP identifikátor `Docentry`, který je v naší databázové struktuře též uvedený jako `UNIQUE`. Tím zabráníme opakovanému nahrání identické zásilky v SAP a tvorby redundantních dat. Avšak v případě, kdy vznikne chyba v datech[1] a je nutné je v SAP opravit, program to v případě, kdy daná zásilka byla zamítnuta API přepravní společností (nemá přiřazené sledovací číslo nebo je příznak `sent_to_api` nepravda), umožní pole aktualizovat.

6.2.11 Generování soupisky zásilek

Generování soupisky muselo být pro našeho uživatele koncipováno co nejpohodlněji. Opět mezi implementací pro různé přepravce není téměř žádný rozdíl. Program umožňuje generovat soupisky pro všechny zásilky s příznakem `!on_consignment_list` a nebo dle specifikovaných `Docentry` v URL parametru. Jedná se tak opětovně o funkci volanou prostřednictvím views (např. `Zasilkovna_communication.create_consignment_list_new(request)` či `Zasilkovna_communi-`

[1] Občas se může stát, že procesem projde zásilka s neplatnou adresou, atp.

cation.create_consignment_list(request) v Python skriptu zasilkovna/views.py) na URL /zasilkovna/create_consignment_list/new resp. /zasilkovna/create_consignment_list/codes.

Využilo se hlavní „přednosti“ Django a generování soupisky probíhá jako generování každé jiné stránky, a to prostřednictvím funkce render(request, template_name, context=None, content_type=None, status=None, using=\ None) z knihovny django.shortcuts. Šablona je vytvořena v souboru templates/pdf/consignment_list_zasilkovna.html a její struktura je vytvoře záměrně pouze pro tisk pomocí CSS @media pravidla print. Díky tomu se nám stránka nastyluje jako A4 připravená k tisku. Pomocí JavaScript skritpu na konci stránky windows.print() vyvoláme ve většině moderních prohlížečů [72] okno, které naši stránku vytiskne.

6.2.12 Aktualizace zásilek v SAP

Aktualizace zásilek v SAP je rozdělena na dvě situace a v obou případech využíváme naše API, které prostřednictvím DI API provede aktualizaci dat v SAP.

1. Nově nahraná zásilka, u které je nutné změnit status *odesláno*.
2. Průběžná aktualizace stavu zásilky, u které se status *nemění*.

První i druhý bod jsou z vysoko-úrovňového pohledu velmi podobné pro všechny přepravní služby.

První situaci prezentuje například funkce Zasilkovna_return_data_to_sap.send_to_sap v souboru zasilkovna/views.py. Funkce je dostupná pod URL /zasilkovna/send_to_sap. V zásadě se jedná pouze o volání API funkce /v1/orders/zasilkovna/update, které je v těle požadavku předán list JSON objektů reprezentující jednotlivé objednávky.

Druhou situaci můžeme ukázat na funkci Zasilkovna_return_data_to_sap.update_old v souboru zasilkovna/views.py dostupná na URL /zasilkovna/update_old/<int:days>. Tato funkce využívá i PacketaAPI resp. B2B API České Pošty, práci s nimi si ukážeme v další části textu. Na úrovni odesílání dat do SAP se téměř nic nemění, neboť se opět při volání API funkce /v1/orders/zasilkovna/update předává v těle požadavků list JSON objektů s objednávkami. Nyní se však jedná o naparsovaná data z odpovědi externích API přepravců.

6.2.13 Packeta API

Abychom mohli komunikovat s přepravcem *Zásilkovna*, musíme využít jejich externí napojení prostřednictvím *Packeta API*.

Komunikace

Jedná se o REST API s rozumně zpracovanými funkcemi volanými dle struktury XML souboru zasílaného v těle požadavku. Jednotlivé funkce se tedy specifikují konkrétním XML elementem, jako např. createPacket. Ověření probíhá prostřednictvím hodnoty v elementu apiPassword specifikovaného v každém požadavku. URL API je https://www.zasilkovna.cz/api/rest.

Odeslání dat

Odesílání dat k přepravci je jedna pro nás z nejdůležitějších funkcí, jež může takové API poskytovat. V *Packeta API* se jedná o funkci `createPacket()`, [73] které se v těle požadavku zasílá struktura `PacketAttributes` [74]. Funkce je synchronní, a tak nám v případě kladného vyhodnocení požadavku a zápisu dat do jejich databáze vrátí v odpovědi strukturu `PacketIdDetail` [75] obsahující pole `barcode` značící sledovací číslo zásilky. V případě záporné odpovědi, API vrací strukturu `PacketAttributesFault` [76].

Získání stavu zásilky

Abychom mohli průběžně aktualizovat stavy odeslaných zásilek, využíváme funkci `packetStatus()` [77], do které se předává `packetId` obsahující sledovací číslo zásilky. Funkce vrací v odpovědi `response.result.codeText` aktuální status zásilky bez její historie.

Získání adresního štítku

Pro vygenerování adresního štítku využíváme funkci `packetsLabelsPdf()` [78]. Ta její návratová hodnota je opět XML s vygenerovanými adresními štítky dle specifikovaných parametrů (např. „A7 on A4“) pro pole zásilek identifikovaných jejich sledovacím číslem. V odpovědi funkce je vráceno PDF zakódované jako base64.

Vrácení PDF uživateli je řešeno pomocí Python knihoven `io` a `base64`. Django nám jako `HttpResponse` umožňuje vrátit strukturu `BytesIO` s dekódovaným PDF z base64 s hlavičkou HTTP Content-Type `application/pdf`.

6.2.14 B2B Brána České Pošty

Pro automatizovanou komunikaci s Českou Poštou je nutné použít jejich tzv. *B2B Bránu* nahrazující prostředí *Podání Online*. Jedná se o REST API, kterému se požadavky zasílají společně s poměrně komplexními XML dokumenty v těle.

Práce s tímto REST API, narozdíl do *Packeta API*, příliš příjemná není. Problémy, kterým jsme čelili při implementaci, si popíšeme níže.

Komunikace

Abychom vůbec s API mohli jakkoli komunikovat, bylo potřeba vyřešit autentizaci firmy při používání API. Ta je u *B2B Brány* řešená prostřednictvím certifikátu, který se přiloží ke každému HTTP požadavku na API. Certifikát vydává autorita *Postsignum* a k jeho zřízení, v době psaní práce, byla nutná návštěva pobočky ČP se službou Czech POINT zmocněncem *Firmy*.

Ke komunikaci s API nebyla využita standardní Python knihovna `requests`, ale její uživatelská nástavba `requests_pkcs12` [79]. Tato knihovna je implementována jako rozšíření oficiální knihovny s vlastním `TransportAdapter`, tak jak doporučují její autoři [80]. Z této knihovny využíváme funkci `post(*args, **kwargs)`, které do parametru `pkcs12_filename` předáme certifikát ve formátu `.p12` a do `pkcs12_password` heslo od certifikátu.

Bohužel má toto API dokumentaci sepsanou „horkou jehlou“, proto práce s ním nebyla příliš příjemná. Zajímavým zjištěním bylo, že všechny funkce, i ty, které nesou v názvu *GET*, jsou řešeny prostřednictvím HTTP POST. Občas nejsou v dokumentaci příliš dobře popsaná jednotlivá pole, a tak je uživatel odkázaný na podporu. K té však není jednoduché se propracovat a až po několika týdnech komunikace s obchodními zástupci ČP, se *Firma* dostala ke kontaktu na podporu API. Jinak bohužel ČP nikde přímý kontakt na IT oddělení neuvádí a doba implementace se tak výrazně prodlužuje.

Odpovědi tohoto API jsou formou XML. To však není vždy příliš jednoduché na analýzu, a tak jsme využili při implementaci knihovnu `xmltodict` [81], která dokáže XML odpověď uloženou jako text převést do struktury `OrderedDict`, k jehož položkám se přistupuje příjemněji, než s využitím DOM parsování.

Jelikož není dokumentace API dostupná online, je přiložena jako příloha.

Odeslání dat

B2B Brána má pro odesílání zásilek dvě funkce. Protože je logika předání dat asynchronní, je nutné, aby uživatel odeslal data funkcí `sendParcels`. Ta, v případě, že jsou přijatá data validní, předá informaci specifikované podací poště a vrátí unikátní identifikátor zásilky `idTransaction`[2]. Se znalostí `idTransaction` můžeme využít funkci `getResultParcels`, která nám na základě `idTransaction`, v případě kladného přijetí dat, vrátí sledovací číslo v poli `PO:parcelCode` a zásilku tak můžeme považovat za podanou.

Získání stavu zásilky

Podobně jako u Packeta API je získávání stavu zásilky poměrně přímočaré. Opět se pouze specifikuje v XML sledovací číslo zásilky. Bohužel ČP nemá funkci, kterou by vrátilo aktuální stav zásilky, ale pouze celou historii zásilky s, bohužel, nezaručeným pořadí jednotlivých stavů. Vytvořili jsme k tomuto účelu alespoň funkci `CeskaPosta_send.find_latest_status`, jež se pokusí vybrat poslední status zásilky.

Získání adresního štítku

Získání adresního štítku je opět téměř identické jako u Packeta API. Pošta vrátí base64 PDF s adresními štítky pro předané pole sledovacích čísel. Během vývoje se zjistilo, že *Firma* doposud využívala formát štítků, který v API nebyl vůbec specifikovaný a umožňoval generování štítků jak pro zásilky typu *DR*, tak i *RR* zároveň. To bohužel žádné formáty specifikované v API neumožňovaly a ani podpora nám při vývoji nebyla schopna poradit. Nicméně zkoušením různých možností (neboť jsou štítky označeny číselným identifikátorem) se nakonec správný formát podařilo nalézt.

[2]Dočasný identifikátor zásilky, na základě kterého můžeme zjistit sledovací číslo či důvod, proč se zásilky nepodařilo nahrát (neexistující adresa, atp.)

7. Vyhodnocení práce

Výstupem práce jsou dvě aplikace pro využití *Firmou*. První z nich je **REST API**, umožňující *Firmě* snazší rozvoj díky možnosti napojovat jednodušeji aplikace třetích stran na jejich hlavní ERP systém SAP Business One. Druhou částí práce bylo konkrétní využití zmíněného REST API v pracovně nazvaném projektu **Aplikace pro komunikaci s přepravními společnostmi**. Ta by měla ke své interakci se SAP využívat naše REST API a *Firmě* tak pomoci ke snazšímu předávání dat o zásilkách přepravním společnostem a výrazně zrychlit proces, jenž byl ve firmě nastavený.

Je důležité zmínit, že obě aplikace jsou již od počátku roku 2021 ve *Firmě* nasazené a naplno využívané. Tento „agilní“ přístup nám tak umožnil získat již během vývoje aplikace důležité poznatky, které se poté jednodušeji implementovaly. Přímá zpětná vazba od uživatelů byla velice přínosná. Protože je aplikace vyvíjena primárně pro ně, bylo důležité, aby se jim s aplikací pracovalo co nejpříjemněji. Více si popíšeme v následujících podkapitolách věnovaným oběma částem práce.

7.1 REST API

API sloužící ke komunikaci se SAP *Firma* potřebovala. Nejen že poskytuje velice jednoduchou rozšiřitelnost o další funkcionalitu v případě nutnosti, ale poskytuje i moderní a jednoduché prostředí, aby komunikace s API byla co nejjednodušší a nejméně nákladná.

7.1.1 Aktuální využití REST API

Jak již bylo v úvodu zmíněno, naše API je již plně nasazené. Uživatelská aplikace zaznamenala v době psaní této kapitoly téměř 14000 požadavků převážně od tří uživatelů, kteří API aktuálně využívají. Nejvíce službu využívá internetový obchod zmíněný v následující části. Ten zaslal více než 55% všech požadavků na API. Je to zejména dané častou kontrolou skladové dostupnosti. Téměř 40% veškerých dotazů vytvořila *Aplikace pro komunikaci s přepravními společnostmi*. Je samozřejmé, že spousta těchto dotazů byla testovacích. Každopádně je vidět, že je API poměrně hojně využívané a bez větších problémů zvládá relativně vysoký počet dotazů.

Napojení na aplikaci pro komunikaci s přepravními společnostmi

V průběhu práce jsme několikrát uvedli, že konkrétně *Aplikace pro komunikaci s přepravními společnostmi* má v našem API speciální *místo*. Dostala vlastní „namespace“ `/orders/`, ve kterém jsou všechny funkce využívané touto aplikací.

Aplikace jmenovitě využívá na denní bázi následující funkce:

- `/orders/zasilkovna/new`
- `/orders/zasilkovna/old`
- `/orders/zasilkovna/update`

- `/orders/ceskaposta/new`
- `/orders/ceskaposta/old`
- `/orders/ceskaposta/update`
- ~~`/orders/ulozenka/new`~~
- ~~`/orders/ulozenka/old`~~
- ~~`/orders/ulozenka/update`~~

Jednou z hlavních změn v API, ke které došlo po nasazení *Aplikace pro komunikaci s přepravními společnostmi*, bylo u funkcí končících `/new` rozdělení pole s adresou do dvou separátních polí „ulice“ a „číslo popisné“. V SAP se tyto hodnoty ukládají dohromady, ale některé přepravní společnosti (zejména ČP a v některých případech i Zásilkovna) doporučují zasílat hodnoty odděleně. Bylo tomu tak dosaženo tvorbou SQL funkce, jež toto s určitou heuristikou obstarává.

Druhou změnou, tentokrát s cílem výrazného zrychlení zápisu dat do SAP, byla úprava `/update` funkcí tak, aby umožňovaly zápis v dávce až 100 položek najednou. Více je tento problém rozebrán v sekci o úpravách po nasazení aplikace 7.1.2.

Napojení na internetový obchod

Jedním z hlavních cílů bylo, aby *Firma* mohla jednoduše poskytovat napojení internetového obchodu na svou instanci SAP Business One. To je nyní s implementací API možné a lze napojit jak externí drop-shipping e-shop, tak například nějaký interní e-shop. Vše spočívá pouze na dohodě mezi provozovatelem a *Firmou*.

Jeden z drop-shipping e-shopů, který s *Firmou* spolupracuje již od roku 2018, doposud využíval řešení *VCZ.WebService* zmíněné v sekci 2.1.4. Tento internetový obchod využíval API poměrně naplno. Zapisoval data o předběžných dokladech, zákaznících a získával prostřednictvím webové služby skladovou dostupnost produktů. Na začátku roku 2021 byl přizván k testování našeho API. Implementoval funkcionalitu pro zápis zákazníka a předběžného dokladu společně s funkcemi pro získávání dat o skladové dostupnosti a jednoduchého způsobu pro nahrávání nových produktů.

Programátor, který řešil implementaci komunikačního rozhraní mezi e-shopem a *Firmou*, se po napojení našeho API vyjádřil následovně:

> „Napojení eshopu bylo dříve řešeno přes VCZ webservice pomocí protokolu SOAP. Pro vytvoření spojení však bylo nutné vytvořit odvozenou třídu, kde bylo potřeba upravit zasílaný request. Což implementaci ztěžovalo.
>
> Největší problém s tímto způsobem spojení byl však response time. Jelikož služba nebyla připravena na přijímání batch requestů, například pro zjištění dostupnosti 100 jednotlivých produktů, bylo třeba zaslat 100 dotazů.
>
> Díky novému API se práce několikanásobně zrychlila a i zjednodušila vzhledem k možnosti přizpůsobení funkcí potřebám e-shopu.“

To však bylo i velikým přínosem při tvorbě API. Měli jsme možnost přímých konzultací s koncovými uživateli API a získat tak lepší přehled o tom, co uživatelé budou skutečně potřebovat.

Případné další využití

Mezi dalšími plánovanými aplikacemi našeho API je využití k párování plateb ve *Firmě*. Znamenalo by to pouze jednoduché rozšíření o dotazovací a zapisovací funkci.

Další služba, při které by API nalezlo uplatnění, by mohl být nástroj pro časované rozesílání notifikací zákazníků. Ať už o nějakém „postupu“ jejich objednávky či jako forma připomínky v případě neuhrazené platby.

7.1.2 Úpravy po nasazení aplikace

Jakmile se API povedlo ve *Firmě* nasadit (neboť se v očích autora jednalo o nemalou překážku), uchýlila se pozornost vývoje směrem, který činil aplikaci velmi složitě použitelnou.

Pomalý zápis

Jednalo se o pomalý zápis prostřednictvím DI API. Pomalým myslíme řádově nižší jednotky sekund během méně vytížených časů. Postupně se zjistilo, že největší překážkou je samotné načtení a inicializace DI API, resp. jeho objektů. Jedná se o veliké množství dat, které musí program zpracovat. Jak se zjistilo, problém vzniká až ve chvíli, kdy uživatel DI API (tedy některá z našich funkcí), odešle požadavek na první zápis. Do té chvíle je program velice rychlý, jak je vidět například na funkcích zajišťujících pouhé získávání dat.

Problém se prozatímně vyřešil formou, kdy je doporučováno zasílat data ve větších dávkách najednou, protože probíhá právě jedna inicializace. Zápis většího množství objektů je tak srovnatelně rychlý se zápisem pouze jednoho objektu.

Nabízelo se však i druhé řešení, které by uživateli poskytlo odpověď rychleji. Tím by mohlo být využití asynchronní logiky. Od tohoto řešení se však ustoupilo a do budoucna bude naši snahou zrychlit inicializaci DI API.

7.1.3 Zhodnocení

API by mělo ve své aktuální verzi plnit všechny požadavky, které byly v analýze práce 3.1 zmíněny. Pojďme se ve zkratce podívat, o které se jednalo:

- ☒ **Moderně navrženo pro rychlé a jednoduché napojení na libovolnou externí službu.**
  - Jak jsme si ukázali, API bylo využito k napojení aplikace pro komunikaci s přepravními službami.
  - Koncipováno jako REST, implementuje tedy obecně zavedenou logiku pro webové služby. Pro jakéhokoli programátora se základními zkušenostmi by tak napojení mělo být velice přímočaré.
  - API přijímá a odesílá požadavky s daty ve formátu JSON s minimální strukturou.

- Chybová hlášení jsou pokud možno odesílány v těle odpovědi pro jednodušší „ladění“ problému.
- Standardizované HTTP status kódy odpovědí.

☒ **Správa uživatelů by měla být řešena na úrovni API.**

- Uživatelská aplikace našeho API obsahuje logiku pro správu uživatelských účtů.
- Možnost jednoduché registrace či změny hesla administrátorem.
- Uživatelům mohou být přiřazovány role stanovující přístupné funkce.

☒ **Jednoduše rozšiřitelné o další požadované funkce, na které se v průběhu využívání přijde.**

- Jak zapisovací tak i dotazovací funkce je velmi jednoduché do API přidat. Stačí implementovat logiku funkce v *Master aplikaci* a vytvořit „proxy“ funkce v naší *Uživatelské aplikaci*. Implementace nových funkcí lze jednoduše odvodit z již vytvořených funkcí.

☒ **Nevyužívat DI API pro čtení.**

- Všechny implementované funkce pro dotazování jsou řešeny prostřednictvím SQL konektoru. Samozřejmě nám nic nebrání ve využití DI API pro čtení, ale lze využít i SQL konektor dostupný v *Master aplikaci*.

Ze samotných uživatelských funkcí API popsaných v 3.1.1 jsme implementovali všechny, jež *Firma* požadovala.

API se již naplno používá na denní bázi v internetovém obchodě a naší *Aplikaci pro komunikaci s přepravními společnostmi*. API se v případě výpadku do jedné minuty opětovně nastartuje a pracuje dále. Výpadek může občas nastat v situaci, kdy je SAP vytížený a DI API není schopné zapisovat.

Z pohledu autora se jedná se o úspěšný projekt, jenž pomohl *Firmě* překonat problém s potřebou jednoduchého způsobu, jak přistupovat k databázi téměř „odkudkoliv“. I když autor práce měl v prvopočátku minimální znalosti SAPB1, *Firma* mu poskytla přínosné konzultace, které pomohly v orientaci ve zkonkrétnění potřeb a samotné struktury objektů SAP.

7.2 Aplikace pro komunikaci s přepravními společnostmi

Abychom byli schopni funkcionalitu a schopnost API prezentovat, přišlo se s nápadem, který spojil potřebné s užitečným. *Firma* kvůli nemožnosti jednoduše komunikovat s využívaným SAPB1 neměla ani možnost bez nákladných úprav snadným způsobem předávat či získávat data od přepravních společností. Jelikož je využívání balíkových služeb pro *Firmu* klíčové k chodu jejího podnikání, bylo vhodné se vydat tímto směrem a pokusit se co nejvíce automatizovat procesy s tím spjaté.

Proto vznikl náš projekt umožňující jednoduché prohlížení zásilek, odesílání a přijímání dat od přepravců, generování adresních štítků a mnoho více. Aplikace byla vytvořena jako webová a v době psaní této práce zpracovala téměř 5000 unikátních zásilek zaslaných ke koncovým zákazníkům, které byly rozeslány prostřednictvím České Pošty, Uloženky nebo Zásilkovny.

7.2.1 Aktuální využití aplikace

Jak již bylo zmíněno, naše aplikace je již zhruba od konce ledna 2021 nasazena a využívána ve *Firmě*. Jako první byla nasazena s podporou pro přepravce Uloženka (dnes již WE|DO), se kterým však *Firma* po velice krátké době rozvázala spolupráci. Aplikace zpracovala 127 zásilek Uloženka. V jisté formě Uloženku nahradila Zásilkovna, která tvoří téměř 70% všech B2C zásilek.

7.2.2 Postřehy z vývoje aplikace

Po zkušenosti s napojením API Uloženka či Zásilkovna, se kterým se pracovalo velice příjemně, bylo podobné, ač naivní, očekávání i od B2B API České Pošty. Avšak předtím, než se vůbec se samotnou komunikací s API začalo, byl i management *Firmy* překvapený, kolik formalit je nutné vyřídit před tím, než je přístup do API (formou PostSignum certifikátu) vůbec udělen. Další nepříjemnosti nastaly ve chvíli, kdy se autor práce pokoušel testovat komunikaci s API. Prvním problémem bylo to, že z nějakého důvodu náš certifikát nebyl u České Pošty ověřený, a tak se necelý měsíc strávil pouze emailovou komunikací, jejíž předmětem byla aktivace certifikátu. Poté, co bylo s API možné komunikovat, nastal druhý problém tkvící v dokumentaci ČP. Jednak se v průběhu implementace narazilo na fakt, že spousta věcí v dokumentaci chybí, a nebo jsou velice mlhavě popsány s předpokladem, že na zbytek uživatel přijde.

Druhým a nyní již pozitivní postřehem bylo, jak příjemně se pracuje s frameworkem *Django*. Vývoj webové aplikace je velmi přímočarý, a ač se v porovnání s např. frameworkem *Flask* může zdát jako zbytečně „objemný“, přináší v základní instalaci již vše, co programátor při vývoji moderní aplikaci potřebuje. Navíc se snaží do určité míry dodržet koncepci návrhového vzoru MVC.

7.2.3 Postřehy po nasazení aplikace

Ačkoliv *Firma* poskytla relativně rozsáhlá testovací data (ve formě zásilek s různými parametry), nebylo možné postihnout všechny situace. Proto se předpokládalo, že první týden po nasazení konkrétního přepravce do aplikace bude probíhat testování, na kterém se otestuje diametrálně více situací, neboť se očekával minimální týdenní počet zásilek okolo 100 kusů.

U přepravní společnosti Zásilkovna k chybám nedocházelo a nasazení bylo téměř bezproblémové.

Očekávaný problematický prvek bylo B2B API České Pošty, kvůli kterému se musela několikrát upravovat samotná data (na úrovni různých „doplňkových“ služeb, které se v dávce se zásilkou odesílají). Dokumentace v době implementace totiž nepokrývala ani standardní způsoby využití API, a tak se spousta změn musela provádět způsobem „pokus/omyl“ a nebo se snažit heuristickým přístupem přijít na to, jak dosáhnout požadovaného výsledku.

U API České Pošty ještě chvíli zůstaneme. Po nasazení se ukázalo, že pošta při zpracování předaných dat o zásilkách kontroluje existenci adresy. Bohužel žádné API pro validaci zaslané adresy neposkytuje, a tak při odeslání dat musí uživatel počkat na odpověď zpracování. Ta v kontextu dodací adresy však může nabývat tří hodnot. 1) **OK** 2) **INFO_ADDRESS_WAS_MODIFIED** značící to, že Pošta adresu přijala, avšak nějakým způsobem upravila. Problém je v tom, že nevíme jak. 3) Zahlásí chybu, že adresa není validní a nelze upravit. V případě třetího bodu však nezbývá nic jiného, než postupně zkoušet upravovat adresu, dokud ji ČP neakceptuje. Problém je však velmi ojedinělý a zpravidla jsou v dodacím listu uložena již sanitizovaná data prostřednictvím aplikace Mapy.cz.

Návrhy a připomínky uživatelů

Poté, co byla aplikace nasazena, se s uživateli prováděly průběžné konzultace a zhodnocení. Ač byly v průběhu celého vývoje uskutečňovány postupné konzultace se zadavateli ve *Firmě*, ale i s uživateli, jenž budou aplikaci ke své práci užívat, vznikly ze strany *Firmy* i autora práce nějaké postřehy (primárně z uživatelského prostředí), které bylo vhodné zapracovat.

Pojďme si postupně v bodech představit všechny připomínky, které během prvních týdnů používání aplikace byly sepsány.

(P1) Přidat do filtrů variabilní symbol.

(P2) Zapisovat do soupisky velikost a váhu balíku.

(P3) Na přehledu zásilek by mělo být v případě výběru zaškrtávacími políčky dostupné i hlavní „vybrat vše“ políčko v hlavičce.

(P4) Na přehledu zásilek by mělo být v případě výběru zaškrtávacími políčky možnost i vybrat více řádku najednou za stisku klávesy „SHIFT“.

(P5) Do SAP by se v případě aktualizace stavu zásilky nemělo zapisovat aktuální datum, ale datum poskytnuté přepravcem (tedy kdy zásilka stavu skutečně nabyla).

(P6) Chyby odeslané z API Česká Pošta by měly být více uživatelsky srozumitelné.

(P7) Přesunout sloupec s datem v hlavním panelu tak, aby byl u variabilního symbolu. Je nutné rychle rozpoznat dnes podané zásilky.

(P8) S nabývajícím počtem zásilek komponenta přepínače stránek v hlavním panelu je širší než obrazovka.

Způsoby implementace jednotlivých bodů si ukážeme v následující podsekci.

Implementace připomínek a návrhů

Protože se počítalo, že se bude projekt po nasazení upravovat, zapracovaly se připomínky zpravidla ihned, jak vznikly. Popišme si způsoby, kterými se jednotlivé připomínky (P1-P8) z předchozí části vyřešily.

(P1) *Přidat do filtrů variabilní symbol.* Jedná se o poměrně jednoduché rozšíření. Nejdříve se upravila komponenta „filtru“ přidáním textového pole. Druhý krok spočíval v rozšíření back-end. Protože jsou filtry předávány jako URL parametry, stačilo pouze vytvořit „getter“ pro další parametr a přidání dalšího „WHERE“ prostřednictvím `filter` v Django ORM při získávání dat z databáze.

(P2) *Zapisovat do soupisky velikost a váhu balíku.* Jelikož se jedná o pole přístupné na každé objektu typu `parcel`, stačilo rozšířit HTML šablonu o další sloupce, do kterého se přidala proměnná definující velikost a váhu.

(P3) *Na přehledu zásilek by mělo být v případě výběru zaškrtávacími políčky dostupné i hlavní „vybrat vše“ políčko v hlavičce.* Protože se jednotlivé „check-boxy“ přidávají do DOM hlavního panelu dynamicky na základě výběru přepravní společnosti, stačilo pouze tuto logiku rozšířit o úpravu hlavičky tabulky. Na toto pole se poté navázala JavaScript funkce vybírající všechna zaškrtávací pole na stránce.

(P4) *Na přehledu zásilek by mělo být v případě výběru zaškrtávacími políčky možnost i vybrat více řádku najednou za stisku klávesy „SHIFT“.* Jelikož tato funkce není obecně prohlížeči podporována, bylo nutné vytvořit logiku popsanou v sekci 6.2.6.

(P5) *Do SAP by se v případě aktualizace stavu zásilky nemělo zapisovat aktuální datum, ale datum poskytnuté přepravcem (tedy kdy zásilka stavu skutečně nabyla).* Upravila se funkce na „parsování“ odpovědi od přepravních společností, které jak pro Zásilkovnu, tak i Českou Poštu obsahují pole s datem. To se poté převede do proměnné typu `Datetime`.

(P6) *Chyby odeslané z API Česká Pošta by měly být více uživatelsky srozumitelné.* Neboť obecně není známá přesná množina chyb a jejich významů, byl každý „chybový“ stav, kterého zásilka nabyla, uveden v historii stavů každé zásilky. Zpravidla se jedná o srozumitelné anglicky psané chyby, ze kterých se dá chyba určitým způsobem vyvodit.

(P7) *Přesunout sloupec s datem v hlavním panelu tak, aby byl u variabilního symbolu. Je nutné rychle rozpoznat dnes podané zásilky.* Protože by nám přesunutí relativně dlouhého data odsadilo ostatní sloupce s důležitými informacemi mimo „view-port“ na většině 16:9 monitorech, označilo se Docentry zásilek podaných tentýž den modrou barvou. Obsluha je tak schopná rychle odlišit, zda-li se jedná o dnes nahraný řádek.

(P8) *S nabývajícím počtem zásilek komponenta přepínače stránek v hlavním panelu je širší než obrazovka.* Vyřešeno algoritmem, který komponentu „zkrátí“. Více popsáno v programátorské dokumentaci 6.2.6.

7.2.4 Zpětná vazba

Vzhledem k tomu, že je naše aplikace denně využívána zaměstnanci *Firmy*, máme možnost získat zpětnou vazbu po více než 4 měsících používání programu. Přikládáme proto zpětnou vazbu od paní, která aplikaci denně využívá.

„Nahrávání balíčků k Zásilkovně, Uložence a České Poště probíhalo na odlišných místech (zasilkovna.cz, ulozenka.cz, postaonline.cz) . Předtím jsem si musela stáhnout excel pro konkrétního přepravce, nahrát ho do jejich aplikace (v případě Pošty zdlouhavě upravovat adresy). Potom jsem musela stahovat soubory s výsledky podání balíčků. Každý den ještě bylo nutné stáhnout soubor s podáními za posledních 20 dní a nahrávat zpět do SAPU. Nahrání České pošty zabralo určitě více času než Zásilkovna nebo Uloženka, jelikož jsem nahrála balíčky, a pak musela opravovat některé adresy, tak aby byly přesně v tom znění, jak je tam požadují. Takže mi to šetří hromadu času, teď vlastně jen párkrát kliknu a mám to hotové. Nemusím se přihlašovat do třech různých aplikací. Zároveň už není nutné se starat o nahrávání dat zpět do SAPU, což je taky určitě plus. Taky je skvělé, že když něco neprojde a nastane nějaká chyba, tak hned vidím, v čem je problém. Předtím to nebylo poznat a navíc, pokud byla chyba u jednoho balíku, nenahrály se ani ty ostatní. Nyní se nenahraje pouze ten chybný balík a ostatní projdou.“

7.2.5 Zhodnocení

Velikou výhodou tohoto celého projektu byla vidina faktu, že aplikace bude reálně „pomáhat“. Nejedná se o nějaký výplňový projekt, ale o něco, co *Firma* skutečně potřebovala. I přes všechny komplikace, které napojení na B2B API České Pošty s sebou přineslo, aplikace pracuje velmi dobře a stabilně.

Pojďme si postupně zopakovat požadavky na naši aplikaci stanovené v kapitole 4.1.

- ☒ **Přehled všech zásilek v tabulce. (vč. export do CSV)**
  - Jedná se o hlavní stránku `/prehled` s vytvořeným tlačítkem „Exportovat do CSV“.
- ☒ **Možnost zobrazit historii stavů získaného od přepravce u konkrétní zásilky.**
  - Na detailu každé zásilky je výpis stavů, kterých zásilka během svého „života“ nabyla.
- ☒ **Jednoznačně označit zásilky, které nebyly doposud nahrány k přepravci.**
  - V hlavní tabulce `/prehled` je každá nenahraná zásilka označena červeně.
- ☒ **Filtrování zásilek dle zadaných parametrů**
- ☒ **Odeslat data o neodeslaných zásilkách k České Poště.**
  - Základní funkcionalita zajištěna ze stránky `/odeslani`.
- ☒ **Získat aktuální stav zásilek odeslaných nejvýše před X dny Českou Poštou.**

  - Základní funkcionalita zajištěna ze stránky `/odeslani` nebo funkcí `/ceskaposta/update_old/<int:days>`.

☒ **Odeslat data o neodeslaných zásilkách k Zásilkovně.**

  - Základní funkcionalita zajištěna ze stránky `/odeslani`.

☒ **Získat aktuální stav zásilek odeslaných nejvýše před X dny Zásilkovnou.**

  - Základní funkcionalita zajištěna ze stránky `/odeslani` nebo funkcí `/zasilkovna/update_old/<int:days>`.

☒ **Průběžně aktualizovat stavy zásilek u přepravců.**

  - Vytvořeny skripty, které jsou spouštěny prostřednictvím `Crontab` na hostovacím počítači. Skripty jako takové jsou spouštěny uvnitř kontejneru.

☒ **Průběžně kontrolovat, zda-li úspěšně proběhlo nahrání do SAP.**

  - Opět zajištěno skripty spouštěnými prostřednictvím `Crontab` na hostovacím počítači. Skripty jako takové jsou spouštěny uvnitř kontejneru.

☒ **Vygenerovat adresní štítky pro danou přepravní společnost.**

  - Pro podporované přepravce jsou zavedeny funkce zajišťující generování adresních štítků. Je zde možnost vygenerovat štítek pro zásilky specifikované v URL parametru prostřednictvím jejich identifikátoru ze SAP a nebo prostřednictvím „zkratky“, jež vygeneruje adresní štítky pro všechny objednávky, které ho doposud neměly vytvořený.

☒ **Vygenerovat soupisku zásilek pro danou přepravní společnost.**

  - Pro podporované přepravce jsou zavedeny funkce zajišťující generování soupisku zásilek. Je zde možnost vygenerovat ji pro zásilky specifikované v URL parametru prostřednictvím jejich identifikátoru ze SAP a nebo prostřednictvím „zkratky“, jež vygeneruje soupisku pro všechny objednávky, které ji doposud neměly vytvořenou.

Protože odevzdáním této práce vývoj projektu rozhodně nekončí, měli bychom si představit další plány na rozvoj aplikace a nedostatky, kterých si je autor vědom.

- Rozšíření o podporu přepravce PPL.
- Lepší práce s XML.
  - Pro přehlednější a „čistší“ tvorbu těla požadavku by bylo vhodné použít knihovnu `xml`, pomocí které by se XML „vybudovalo“. Avšak na obhajobu aktuálního přístupu, pro lepší přehlednost toho, co do externích API odesíláme, byla volba parametrizovaného textu na práci přívětivější. Autorovi se tak s XML pracovalo rychleji.

  - Pro zefektivnění procesů v aplikaci by se nabízelo změnit logiku, kdy XML převádíme na Python `OrderedDict` prostřednictvím doinstalované knihovny na způsob klasického DOM parsování.

- Rozšíření o podporu „zpětné“ zásilky prostřednictvím Zásilkovny.

  - Zásilkovna nabízí službu, prostřednictvím které je možné, aby zákazník jednoduše zásilku vrátil. Nabízelo by se rozšíření aplikace i o tuto funkcionalitu, kdy by mohla obsluha pro každou zásilku, kterou v programu má nahranou, vygenerovat i požadavek na „zpětnou zásilku“.

Jedna z myšlenek byla, aby se řešení projektu zobecnilo natolik, že by bylo využitelné v libovolné firmě. Aplikace by stále mohla pracovat v „cloudu“ a zákazníci by k ní přistupovali jako uživatelé. Formát nahrávaní dat by se dal řešit prostřednictvím tvorby vlastního API naší aplikace či čerpáním dat z nějakého dostupného feedu. Ač se jedná o pouhý nápad a hru budoucnosti, autor věří, že by aplikace našla své uplatnění ve více firmách. Je totiž velmi pravděpodobné, že problém, který jsme vyřešili na příkladu *Firmy*, se vyskytuje i jinde.

Nicméně autor práce shledává projekt jako úspěšný. Zaměstnancům reálně pomáhá na denní bázi a zároveň to byla skvělá zkušenost vytvořit software od A do Z. Tedy od návrhu až po finální nasazení a postupné vylepšování.

8. Závěr

Cílem této práce bylo zefektivnit zavedené procesy a umožnit *Firmě* výrazně jednodušší práci s jejími daty. Bylo navrženo, vytvořeno a nasazeno moderní REST API pro jednoduchou komunikaci se SAP Business One, které je na denní bázi využíváno a pomáhá nejen zaměstnancům ale i odběratelům. Zaměstnanci využívají API v naší aplikaci zajišťující komunikaci mezi přepravními společnostmi a *Firmou* . Ta denně bez problému nahraje několik desítek zásilek a téměř každou noc jich více než tisíc zaktualizuje.

Projekt, i dle zpětné vazby uživatelů jednotlivých aplikací, je úspěšný a svou funkci plní. Architektura a zvolené technologie aplikací se prokázaly jako vhodné a jak bylo předpokládáno, v ničem nás neomezovaly.

Protože jsou uživatelé s aplikacemi spokojeni, je samozřejmé, že jejich rozvoj bude nadále podporován. API v nadcházejících měsících bude využito k napojení nově vznikajícího internetového obchodu *Firmy*. Jeho funkcionalita by například měla být rozšířena o možnosti získávat informace ke konkrétním fakturám. Umožní to například možnost předvyplnit formulář pro vrácení zboží a usnadní to práci jak zákazníkům tak i zaměstnancům *Firmy*, kteří se často setkávali se špatně vyplněnými produktovými kódy v reklamačních formulářích.

Co se týče aplikace pro komunikaci s přepravními společnostmi, je naplánováno rozšíření o napojení na přepravní společnost PPL. Další z plánovaných funkcionalit je možnost přidání reklamačního balíčku od Zásilkovna. To bude *Firmě* umožňovat svým zákazníkům rovnou zaslat adresní štítky na email, pomocí kterých budou moci zboží vrátit.

Autora práce těší nově nabyté zkušenosti a spolupráce s *Firmou*. Nejpřínosnější na celém projektu pro něho doposud bylo, že aplikace, které vytvořil, slouží svému účelu a zefektivňují práci druhých.

Seznam použité literatury

[1] Wikipedia MVC. https://cs.wikipedia.org/wiki/Model-view-controller.

[2] Karel Wolf. E-commerce v roce 2020: Dvou- i trojciferné růsty na vlnách pandemie, 2020.

[3] SAP Business One Wikipedia. https://en.wikipedia.org/wiki/SAP_Business_One.

[4] ODBC Microsoft Docs. https://docs.microsoft.com/cs-CZ/sql/odbc/admin/odbc-data-source-administrator?view=sql-server-ver15.

[5] . https://docs.microsoft.com/cs-cz/dotnet/csharp/.

[6] C++. https://www.cplusplus.com.

[7] PHP. https://www.php.net.

[8] Python. https://www.python.org.

[9] Perl. https://www.perl.org.

[10] SAP Business One Java Connector. https://support.sap.com/en/product/connectors/jco.html.

[11] phpDIServer. https://github.com/manuparra/phpDIServer.

[12] SAP B1 RFC Connector C++. https://github.com/rfcconnector/example-cpp.

[13] Flask SAPB1 DI API wrapper Github. https://github.com/ideabosque/Flask-SAPB1.

[14] SOAP W3 Docs. https://www.w3.org/TR/soap12/.

[15] Python Requests knihovna. https://docs.python-requests.org/en/master/.

[16] C++ libcurl knihovna. https://curl.se/libcurl/.

[17] PHP Guzzle knihovna. https://docs.guzzlephp.org/en/stable/.

[18] REST W3 Docs. https://www.w3.org/2001/sw/wiki/REST.

[19] W3 XML. https://www.w3.org/XML/.

[20] MDN JSON. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON.

[21] YAML oficiální stránky. https://yaml.org.

[22] Dokumentace frameworku Flask. https://flask.palletsprojects.com/en/1.1.x/foreword/.

[23] Github frameworku Pistache. https://github.com/pistacheio/pistache.

[24] Tornado framework Dokumentace. https://www.tornadoweb.org/en/stable/.

[25] FastAPI framework. https://fastapi.tiangolo.com.

[26] Swagger domovská stránka. https://www.google.com/search?client=safari&rls=en&q=swagger&ie=UTF-8&oe=UTF-8.

[27] Flask-RESTX dokumentace. https://flask-restx.readthedocs.io/en/latest/.

[28] About Apache. https://httpd.apache.org/ABOUT_APACHE.html.

[29] Nginx oficiální stránky. https://www.nginx.com.

[30] NoSQL. https://azure.microsoft.com/cs-cz/overview/nosql-database/.

[31] MySQL oficiální stránky. https://www.mysql.com.

[32] SQLite oficiální stránky. https://www.sqlite.org/index.html.

[33] PostgreSQL oficiální stránky. https://www.postgresql.org.

[34] Porovnání SQLite, MySQL a PostgreSQL. https://www.digitalocean.com/community/tutorials/sqlite-vs-mysql-vs-postgresql-a-comparison-of-relational-database-manageme

[35] Ubuntu Server 18.04. https://help.ubuntu.com/18.04/serverguide/index.html.

[36] Docker Compose. https://docs.microsoft.com/cs-cz/visualstudio/docker/tutorials/use-docker-compose.

[37] Docker getting started. https://docs.docker.com/get-started/.

[38] Kubernetes oficiální stránky. https://kubernetes.io.

[39] HTML. https://developer.mozilla.org/en-US/docs/Web/HTML.

[40] CSS. https://developer.mozilla.org/en-US/docs/Web/CSS.

[41] JavaScript. https://www.javascript.com.

[42] Wikipedia PHP. https://en.wikipedia.org/wiki/PHP.

[43] W3 statistika PHP. https://w3techs.com/technologies/details/pl-php.

[44] NodeJS. https://nodejs.org/en/.

[45] W3 statistika NodeJS. https://w3techs.com/technologies/details/ws-nodejs.

[46] W3 statistika Python. https://w3techs.com/technologies/details/pl-python.

[47] Wikipedia Framework. https://en.wikipedia.org/wiki/Software_framework.

[48] Flask. https://flask.palletsprojects.com/en/1.1.x/.

[49] Flask templating. https://flask.palletsprojects.com/en/1.1.x/templating/.

[50] Django. https://www.djangoproject.com.

[51] Pyramid. https://help.ubuntu.com/18.04/serverguide/index.html.

[52] Github Djongo. https://github.com/nesdis/djongo.

[53] Flask-SAPB1 PyPI.

[54] pymssql GitHub. https://github.com/pymssql/pymssql.

[55] Flask SQLAlchemy Dokumentace. https://flask-sqlalchemy.palletsprojects.com/en/2.x/.

[56] Flask Migrate Dokumentace. https://flask-migrate.readthedocs.io/en/latest/.

[57] Flask RESTX Dokumentace. https://flask-restx.readthedocs.io/en/latest/.

[58] CherryPy. https://cherrypy.org.

[59] PyWin32 GitHub. https://github.com/mhammond/pywin32.

[60] Automating Windows Applications Using COM. https://pbpython.com/windows-com.html.

[61] Component Object Model (COM). https://docs.microsoft.com/cs-cz/windows/win32/com/component-object-model--com--portal?redirectedfrom=MSDN.

[62] Python setattr. https://docs.python.org/3/library/functions.html#setattr.

[63] pymssql execute(). https://pymssql.readthedocs.io/en/stable/ref/pymssql.html?highlight=execute#pymssql.Cursor.execute.

[64] pymssql execute(). https://pymssql.readthedocs.io/en/stable/ref/pymssql.html?highlight=fetchall#pymssql.Cursor.fetchall.

[65] Python dekorátory dokumentace. https://www.python.org/dev/peps/pep-0318/.

[66] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994.

[67] Django dokumentace. https://docs.djangoproject.com/en/3.2/faq/general/.

[68] Django dokumentace modely. https://docs.djangoproject.com/en/3.2/topics/db/models/.

[69] Vztah Django a MVC. https://docs.djangoproject.com/en/3.2/faq/general/#django-appears-to-be-a-mvc-framework-but-you-call-the-controller-

[70] Django úvodní návod. https://docs.djangoproject.com/en/3.2/intro/tutorial01/.

[71] Django dokumentace views. https://docs.djangoproject.com/en/3.2/topics/http/urls/#views-extra-options.

[72] Web API tisk. https://developer.mozilla.org/en-US/docs/Web/API/Window/print.

[73] Packeta API dokumentace createPacket(). https://docs.packetery.com/03-creating-packets/06-packetery-api-reference.html#toc-createpacket.

[74] Packeta API dokumentace PacketAttributes. https://docs.packetery.com/03-creating-packets/06-packetery-api-reference.html#toc-packetattributes.

[75] Packeta API dokumentace PacketDetail. https://docs.packetery.com/03-creating-packets/06-packetery-api-reference.html#toc-packetiddetail.

[76] Packeta API dokumentace PacketAttributesFault. https://docs.packetery.com/03-creating-packets/06-packetery-api-reference.html#toc-packetattributesfault.

[77] Packeta API dokumentace packetStatus(). https://docs.packetery.com/03-creating-packets/06-packetery-api-reference.html#toc-packetstatus.

[78] Packeta API dokumentace packetLabelsPdf(). https://docs.packetery.com/03-creating-packets/06-packetery-api-reference.html#toc-packetslabelspdf.

[79] Github knihovna requests_pkcs12. https://github.com/m-click/requests_pkcs12.

[80] Python knihovna requests TransportAdapters. https://2.python-requests.org/en/master/user/advanced/#transport-adapters.

[81] Github knihovna xmltodict. https://github.com/martinblech/xmltodict.

Přílohy

V této části jsou popsány přílohy odevzdávané s prací.

- **api/** - adresář obsahující Master a Uživatelskou aplikaci API
  - **SAPB1_master_app/** - adresář Master aplikace API - více rozebráno v 5.3.1
  - **SAPB1_user_app/** - adresář Uživatelské aplikace API
    - **docker-compose** - Docker Compose aplikace, db a Nginx
    - **Dockerfile** - kontejner pro Flask aplikaci
    - **proxy/** - webový server
      - **default.conf** - Nginx konfigurace
      - **Dockerfile** - Dockerfile pro webový server
      - **uwsgi_params** - parametry WSGI aplikace
      - **scripts/** - adresář se skripty pro Dockerfile Flask aplikace
        - **entrypoint.sh** - spouštěcí skript Flask aplikace
    - **user_root/** - adresář s Flask aplikací Uživatelskou app
      - **src/** - adresář uživatelské aplikace - více rozebráno v 5.4.1
- **app_zasilky** - adresář s aplikací pro komunikaci s přepravními společnostmi
  - **docker-compose.yml** - Docker Compose aplikace, db a Nginx
  - **Dockerfile** - kontejner pro Django aplikaci
  - **proxy/** - webový server
    - **default.conf** - Nginx konfigurace
    - **Dockerfile** - Dockerfile pro webový server
    - **uwsgi_params** - parametry WSGI aplikace
  - **scripts/** - adresář se skripty pro Dockerfile Django aplikace
    - **entrypoint.sh** - spouštěcí skript Django aplikace
  - **zasilky/** - adresář obsahující Django aplikaci
- **tex** - adresář se zdrojovými soubory LaTeX
- **BP_text.pdf** - text práce

A. Uživatelská dokumentace - API

API je veřejně přístupné na adrese `http://88.146.158.222:2224/`. Jednotlivým funkcím poté předchází ještě cesta `<prostredi>/v1/`. Parametr `<prostredi>` může nabývat hodnot buď `prod` či `dev`. Těmito parametry se odlišuje testovací a produkční instance SAP *Firmy*. Pokud nebude řečeno jinak, budeme každou zmíněnou adresu funkce uvažovat s prefixem `http://88.146.158.222:2224/<prostredi>/v1`.

A.1 Zřízení účtu

Pro zřízení účtu k API kontaktujte podporu na adrese `it@`*`Firma`*`.cz` a specifikujte, prosím, konkretní požadavky na API.

A.1.1 Funkce pro zřízení účtu

Uživatel s rolí `admin` může využít funkce `http://88.146.158.222:2224/v1/user/create`, které stačí v body zaslat JSON ve formátu:

```
{
        "username": "jmeno-uzivatele",
        "password": "heslo",
        "email": "uzivatel@email.cz",
        "cardcode": "Z0001"
}
```

A.2 Uživatelské role

Každému uživateli jsou přiřazené role dle funkcí, které požaduje. Pokud by mělo být API využíváno pouze pro „získávání“ dat, bude přiřazená role `select-user`, v opačném případě lze přiřadit role `insert-user`.

A.2.1 Výpis všech rolí

Všechny role může uživatel s přiřazenou rolí `admin` získat prostřednictvím funkce:
`http://88.146.158.222:2224/v1/user/list/roles`.
Příklad odpovědi:

```
[
  {
    "id": 1,
    "name": "admin",
    "description": "Full permissions"
```

```
  },
  {
    "id": 2,
    "name": "select-user",
    "description": "Allows only for usage of prod. GET functions."
  },
  {
    "id": 3,
    "name": "insert-user",
    "description": "Allows only for usage of POST/UPDATE/INSERT
                    functions over products, bp, drafts and orders."
  },
  {
    "id": 4,
    "name": "order",
    "description": "Limits usage only for /orders/ functions."
  },
  {
    "id": 5,
    "name": "order-insert",
    "description": "Not yet supproted role."
  }
]
```

A.2.2 Výpis rolí přiřazených uživateli

V případě vypsání rolí konkrétního uživatele lze využít funkci: `http://88.146.158.222:2224/v1/user/<username>/roles`.

Příklad odpovědi:

```
{
  "username": "username",
  "roles": [
    {
      "name": "order"
    }
  ]
}
```

A.2.3 Přidání role uživateli

Přidání uživateli jednu z výše specifikovaných rolí lze provést funkcí: `http://88.146.158.222:2224/v1/user/<username>/add/<role-name>` s využitím účtu s rolí `admin`.

A.2.4 Odebrání role uživateli

Odebrání uživateli jedné z výše specifikovaných rolí lze provést funkcí: `http://88.146.158.222:2224/v1/user/<username>/remove/<role-name>`. Pro spuštění této funkce je nutný přístup s `admin` rolí.

A.3 Autentizace

K autentizaci uživatele je prováděna při každém volání funkce API prostřednictvím Basic Access Authentication. Stačí tak v hlavičce každého HTTP požadavku uvést přihlašovací jméno a heslo přidělené *Firmou*.

A.4 Swagger dokumentace

API má automaticky vygenerovanou dokumentaci `Swagger`, dostupnou na adrese:
`http://88.146.158.222:2224/api/swagger`. `<env>` v každé adrese specifikuje prostředí (dev, prod).

A.4.1 <env>/v1/products

Skupina funkcí pro dotazování se informací o produktech *Firmy*. Dává nám přístup k funkcím jako:

- `GET /availability` Vrátí dostupnost specifikovaných produktů. Dostupnost může být záporná, vzhledem k možnosti vytvářet rezervace na nedostupné produkty.
- `GET /list` Vrátí všechny prefix (skupiny artiklů) dostupné v databázi.
- `GET /<prefix>/list` Vrátí seznam artiklů vztažených k danému prefixu (skupině artiklů).
- `GET /price/<pricelist-id>` Vrátí cenu specifikovaných artiklů v daném ceníku, který je specifikovaný parametrem `pricelist-id`.
- `GET /<itemcode>/detail` Vrátí detail artiklu.
- `GET /<prefix>/detail` Vrátí detail prefixu.

A.4.2 <env>/v1/draft

Skupina funkcí pro práci s předběžnými doklady.

- `POST /insert` Umožní zapsat předběžný doklad objednávky do SAP. Umožňuje zápis do všech polí dostupných pro předběžné doklady v SAP společně s uživatelskými poli. Pro více informací k příkladu zápisu (fyzická/právnická osoba, apod.) a specifikaci polí, kontaktujte, prosím, podporu na adrese `it@`*`Firma`*`.cz`.

A.4.3 <env>/v1/bp

Skupina funkcí pro práci s obchodními partnery (zákazníky).

- `POST /insert` Umožní zapsat objekt reprezentující nového obchodního partnera do SAP. Umožňuje zápis do všech polí dostupných pro OP v SAP, společně s uživatelskými poli. Pro více informací k příkladu zápisu a specifikaci polí, kontaktujte, prosím, podporu na adrese `it@`*`Firma`*`.cz`.

- `GET /gdpr/get` Slouží k získání informace o tom, zda-li má specifikovaný email souhlas ke zpracování os. údajů za účely marketingu.
- `PUT /gdpr/put` Slouží ke změně informace o souhlasu ke zpracování os. údajů za účely marketingu ke specifikovanému emailu.

A.4.4 <env>/v1/orders

Skupina funkcí pro práci s objednávkami.

- `GET /ceskaposta/new` Umožní získat všechny doposud neodeslané zásilky ČP.
- `GET /ceskaposta/old/<days>` Umožní získat všechny doposud odeslané zásilky ČP ve stáří nejvýše `days`.
- `PUT /ceskaposta/update` Umožní upravit informace jako sledovací číslo, odkaz pro sledování zásilky, status o odeslání a stav zásilky na kartě objednávky ČP.
- `GET /zasilkovna/new` Umožní získat všechny doposud neodeslané zásilky Zásilkovna.
- `GET /zasilkovna/old/<days>` Umožní získat všechny doposud odeslané zásilky Zásilkovna ve stáří nejvýše `days`.
- `PUT /zasilkovna/update` Umožní upravit informace jako sledovací číslo, odkaz pro sledování zásilky, status o odeslání a stav zásilky na kartě objednávky Zásilkovna.
- `GET /ulozenka/new` Umožní získat všechny doposud neodeslané zásilky Uloženka.
- `GET /ulozenka/old/<days>` Umožní získat všechny doposud odeslané zásilky Uloženka ve stáří nejvýše `days`.
- `PUT /ulozenka/update` Umožní upravit informace jako sledovací číslo, odkaz pro sledování zásilky, status o odeslání a stav zásilky na kartě objednávky Uloženka.

A.5 Příklad volání

Nyní si ukážeme příklad využití API v několika programovacích jazycích. Využití napříč jednotlivými funkcemi se prakticky neliší (pouze jen adresa funkce + různý formát dat v těle požadavku), proto uvedeme jako příklad funkci pro získání dostupnosti zboží. na adrese `<env>/v1/products/availability` s body obsahujícím:

```
[
  {
    "ItemCode": "07014-LBH-LXL-UPE"
```

```
  },
  {
    "ItemCode": "07014-LBH-ML-UPE"
  }
]
```

A.5.1 PHP

V PHP využijeme knihovnu `cURL` neboli `Client URL Library`.

```
$itemCodes = ["ItemCode" => "07014-LBH-LXL-UPE",
              "ItemCode" => "07014-LBH-ML-UPE"];
$url = "http://88.146.158.222:2224/prod/v1/products/availability"
$username="username;
$password="$password;
$ch = curl_init();
curl_setopt_array($ch, [
        CURLOPT_PORT => "2224",
        CURLOPT_URL => $url,
        CURLOPT_RETURNTRANSFER => true,
        CURLOPT_ENCODING => "",
        CURLOPT_MAXREDIRS => 10,
        CURLOPT_TIMEOUT => 30,
        CURLOPT_HTTP_VERSION => CURL_HTTP_VERSION_1_1,
        CURLOPT_CUSTOMREQUEST => "GET",
        CURLOPT_POSTFIELDS => json_encode(),
        CURLOPT_HTTPHEADER => [
            "content-type: application/json"
        ],
        CURLOPT_USERPWD => sprintf("%s:%s", $username, $password)
    ]);
$return = curl_exec($ch);
curl_close($ch);

$availability = json_decode($return);
```

A.5.2 C#

Pro C#využijeme knihovnu `System.Net.Http`.

```
using System;
using System.Net.Http;

var username = "username";
var password = "passwd";
var baseAddress = new Uri("http://88.146.158.222:2224/dev/v1/");

using (var httpClient = new HttpClient{ BaseAddress = baseAddress })
{
    var authToken = Encoding.ASCII.GetBytes(\$"{userName}:{passwd}");
```

```
    httpClient.DefaultRequestHeaders.Authorization =
        new AuthenticationHeaderValue(
            "Basic",
            Convert.ToBase64String(authToken)
        );
    using (var content = new StringContent(
        "[{\"ItemCode\": \"07014-LBH-LXL-UPE\"},
        {\"ItemCode\": \"07014-LBH-ML-UPE\"}]",
        System.Text.Encoding.Default, "application/json"))
    {
      using(var response = await httpClient.GetAsync(
        "products/availability"
        ))
      {

            string responseData =
                await response.Content.ReadAsStringAsync();
      }
    }
}
```

A.5.3 JavaScript

Pro implementaci v `JavaScript` využijeme `XMLHttpRequest`.

```
var request = new XMLHttpRequest();

request.open(
              'GET',
              'http://88.146.158.222:2224/dev/v1/products/availability',
              true,
              username,
              password
            );
request.withCredentials = true;

request.setRequestHeader('Content-Type', 'application/json/');

request.onreadystatechange = function () {
  if (this.readyState === 4) {
    console.log('Status:', this.status);
    console.log('Headers:', this.getAllResponseHeaders());
    console.log('Body:', this.responseText);
  }
};

var body = [
  {
    "ItemCode": "07014-LBH-LXL-UPE"
  },
  {
```

```
    "ItemCode": "07014-LBH-ML-UPE"
  }
];

request.send(JSON.stringify(body));
```

A.5.4 Python

Pro implementaci v Python využijeme standardní knihovnu `requests`.

```
import requests
url = 'http://88.146.158.222:2224/dev/v1/products/availability')
arr = [
        {"ItemCode": "07014-LBH-LXL-UPE"},
        {"ItemCode": "07014-LBH-ML-UPE"}
      ]
resp = requests.get(url=url, json=arr,auth=(username, password))
if resp.status_code == 200:
    data = resp.json()
```

B. Uživatelská dokumentace aplikace pro odesílání zásilek

Aplikace je v rámci firemní sítě přístupná na adrese `http://192.168.1.117`.

B.1 Uživatelské prostředí aplikace

B.2 Přihlášení

Nepřihlášeným uživatelům se po prvním načtení aplikace zobrazí okno `http://192.168.1.117/login/` s polem pro uživatelské jméno a heslo. To vyplňte a stiskněte tlačítko „Přihlásit se“. Po zadání špatných či neexistujících přihlašovacích údajů budete upozorněni zprávou zobrazenou v okně. Pokud jste přihlašovací údaje zapomněli, kontaktujte, prosím, podporu na emailové adrese `it@Firma.cz`.

B.3 Odhlášení

Pokud jste již přihlášeni a chcete se odhlásit, v každém okně (mimo přihlašovacího), je v pravém horním rohu dostupné tlačítko „Odhlásit se“.

B.4 Žádost o zřízení nového účtu

V případě, že si přejete zřídit nový uživatelský účet, kontaktujte, prosím, podporu na emailové adrese `it@Firma.cz`.

B.5 Okna aplikace

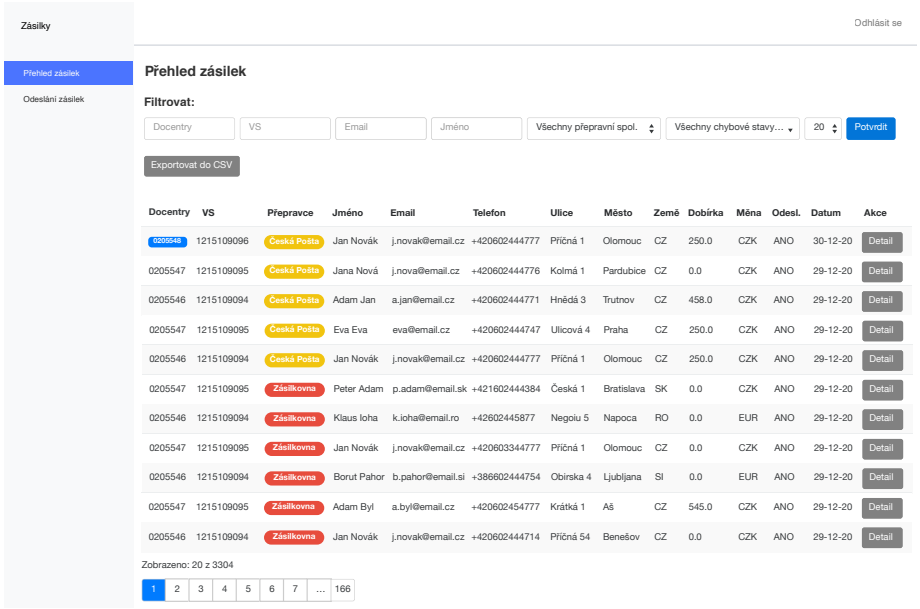
V této sekci budou popsána jednotlivá okna naší aplikace.

B.5.1 Přehled zásilek

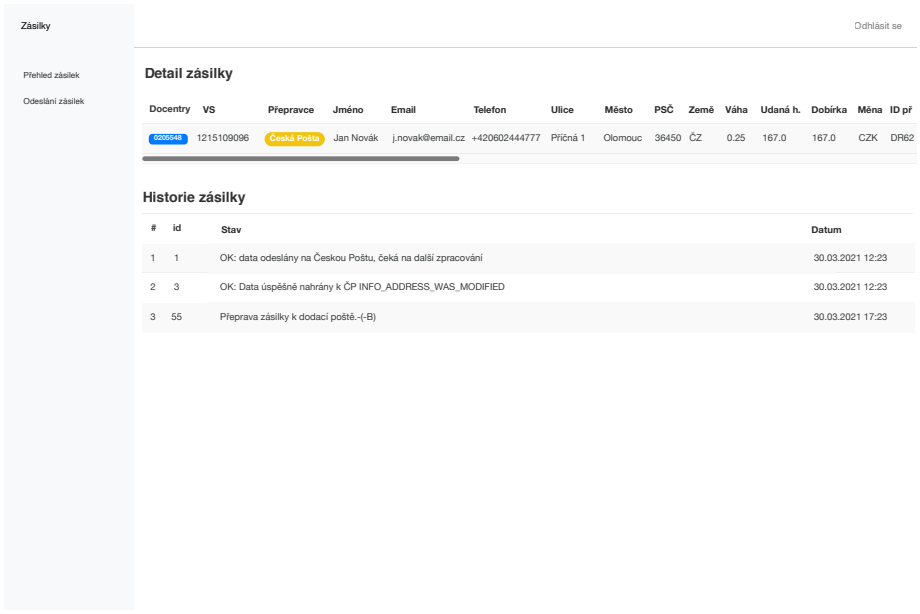
Hlavní stránka zobrazená po přihlášení na adrese `http://192.168.1.117/prehled/`. Obsahuje pole pro filtrování, hlavní tabulku zásilek a levé navigační menu. Uživatelské prostředí aplikace s hlavní tabulkou je k nahlédnutí na obrázku B.1.

B.5.2 Detail zásilky

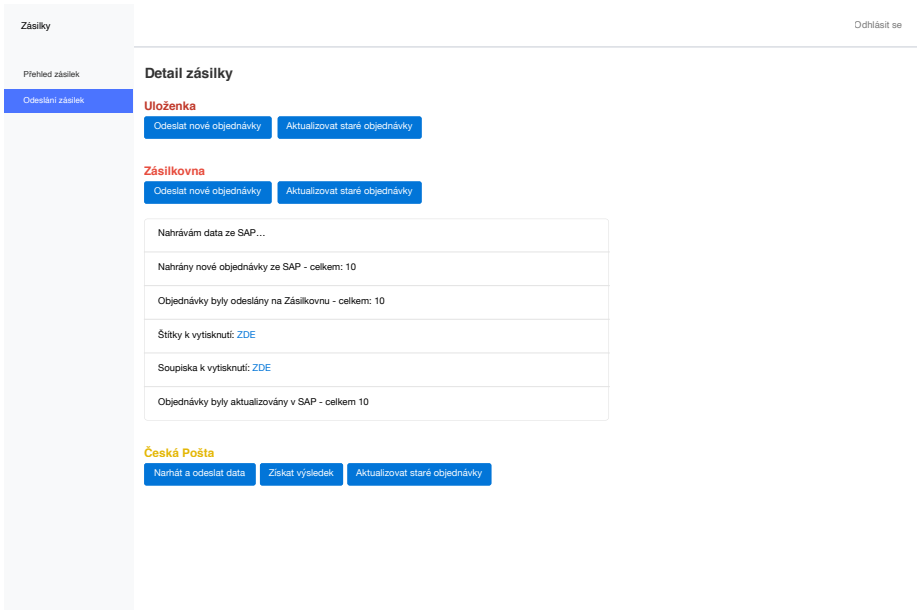
Detail každé zásilky je dostupný na adrese `http://192.168.1.117/detail/<int:id>`, kde `id` je pořadové číslo zásilky v databázi. O to se uživatel nemusí starat. Na této stránce máme dostupné téměř všechny informace o zásilce v tabulce, se kterou lze pohybovat do šířky. Uživatelské prostředí detailu zásilky je k vidění na obrázku B.2.



Obrázek B.1: Uživatelské prostředí hlavní tabulky zásilek



Obrázek B.2: Uživatelské prostředí detailu zásilky



Obrázek B.3: Uživatelské prostředí hlavní tabulky zásilek

Sledovací číslo zásilky

Sledovací číslo zásilky je dostupné ve sloupci tabulky `ID přepravce`. Odkaz pro sledování zásilky by měl být dostupný ve sloupci `Sledovací odkaz`.

Příznaky zásilky

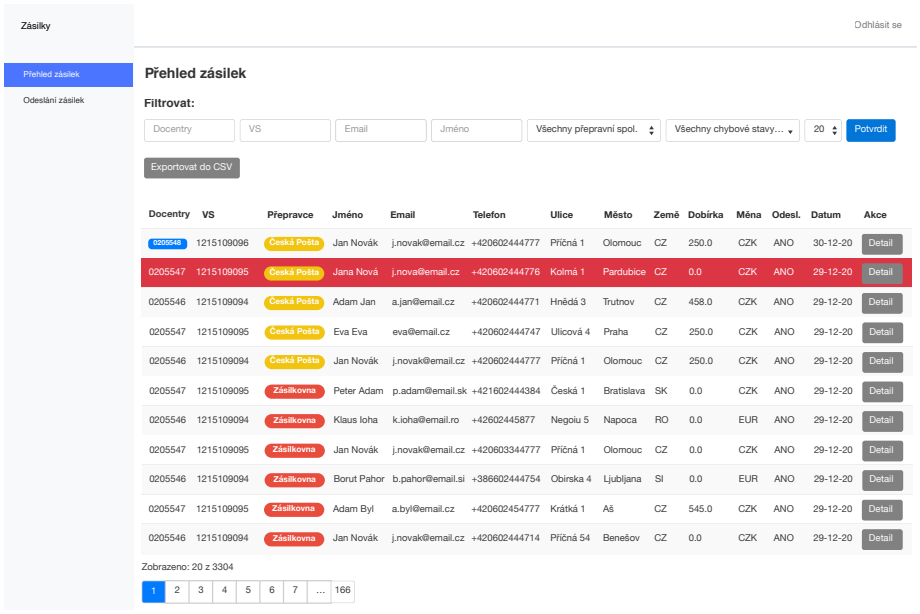
Pro lepší orientaci jsou v tabulce zobrazeny i příznaky určující, zda-li zásilka prošla některými důležitými stavy. Sledujeme informace jako *Odesláno k přepravci*, *Odesláno do SAP*, *Na štítku* (určuje, zda-li byla zásilka alespoň jednou exportována na adresní štítek) a *Na soupisce* (určující, zda-li byla zásilka alespoň jednou exportována na soupisku).

B.5.3 Odeslání zásilek

Pro pohodlné ovládání odesílání dat (směrem k přepravci či do SAP), existuje okno `http://192.168.1.117/odeslani/` obsahující ovládací prvky pro spouštění jednotlivých akcí. Více bude popsáno v sekci B.7 Uživatelské prostředí pro odeslání zásilek je k vidění na obrázku B.3 včetně náhledu tabulky, jenž informuje o průběhu zpracování.

Průběžné stavy zásilky

Průběžné stavy zásilky jsou dostupné v tabulce `Historie zásilky`. Do této tabulky se zapisují přepravní i chybové stavy, kterých zásilka nabyla. Např. tam může být uvedeno `Není zadána hodnota zásilky pro účely pojištění. Prosíme zadejte ji`. Jedná se o chybové hlášení od přepravce Zásilkovna. Chybové stavy České Pošty nejsou obvykle tak srozumitelné. U velké spousty zásilek ČP se objevuje stav `INFO_ADDRESS_WAS_MODIFIED`, který pouze informuje o tom, že na straně ČP došlo k mírné úpravě adresy. Obecně jsou stavy ČP začínající jako



Obrázek B.4: Uživatelské prostředí hlavní tabulky zásilek s chybou nahrání zásilky k přepravci

`INFO_` pro uživatele nedůležité. Občas se stane, že Česká Pošta vrátí informaci `INVALID_ADDRESS`. Univerzální postup na opravu této chyby neexistuje, nejlepší je zkontrolovat adresu dané zásilky, např. přes `mapy.cz` či `maps.google.com` a pokusit se o opětovné nahrání zásilky. Pokud se u zásilky ČP objeví stav `UNFINISHED_PROCESS`, data teprve prochází zpracováním a ČP prozatím nevrátila jednoznačnou chybu či sledovací číslo.

B.6 Otázky a odpovědi

V této sekci si projdeme pár základní otázek a odpovědí, které Vás mohou při práci s aplikací napadnout.

B.6.1 Jak zobrazit detail zásilky?

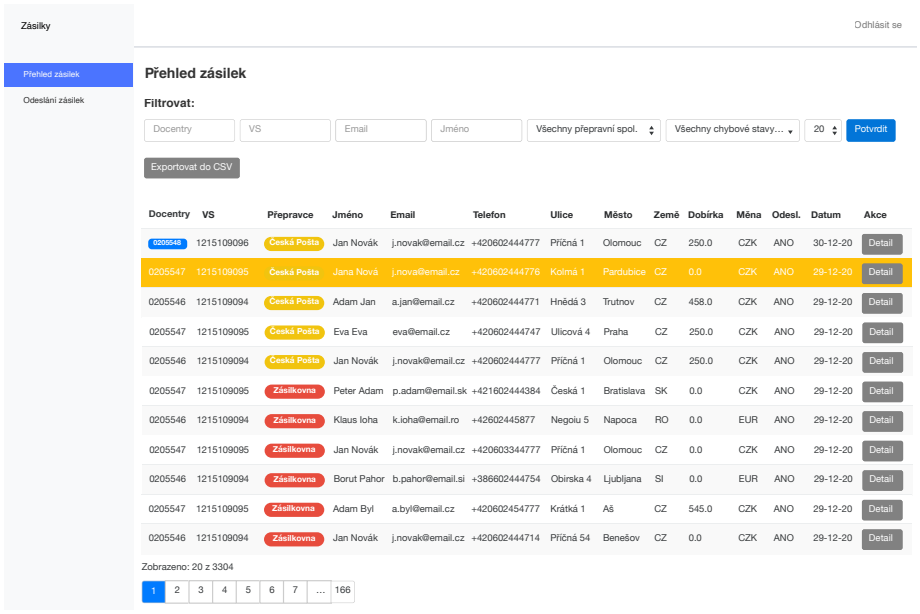
Nejjednodušší způsob je v posledním sloupci tabulky na stránce `http://192.168.1.117/prehled/` kliknout na tlačítko *Detail*.

B.6.2 Jak zjistit, že se aplikace nahrála k přepravci?

Pokud by se zásilka nenahrála, pak je celý její řádek v tabulce `http://192.168.1.117/prehled/` označený červeně jako je k nahlédnutí na obrázku B.4.

B.6.3 Jak zjistit, že se aplikace nahrála do SAP?

Pokud by se zásilka nenahrála, pak je celý její řádek v tabulce `http://192.168.1.117/prehled/` označený žlutě jako je k nahlédnutí na obrázku B.5.



Obrázek B.5: Uživatelské prostředí hlavní tabulky zásilek s chybou nahrání zásilky do SAP

B.6.4 Jak vyexportovat adresní štítky od vybraných zásilek?

Aby bylo možné exportovat adresní štítky, je nutné nejdříve vybrat konkrétního přepravce. To lze provést jednoduchým filtrováním, např. Česká Pošta v rozbalovacím menu nad tabulkou. Poté, co jsou zásilky vyfiltrované, zobrazí se na každém řádku v prvním sloupci zaškrtávací pole. Zde je možné vybrat požadované zásilky (lze i několik najednou pomocí klasického výběru za stisku `SHIFT`). Poté je již možné dole pod tabulkou v rozbalováním menu „Spustit akci na vybraných“ zvolit „Vygenerovat štítek“.

B.6.5 Jak filtrovat zásilky?

Zásilky je možné filtrovat pomocí docentry, variabilního symbolu, emailu, jména, přepravní společnosti či dokonce stavu v aplikaci. Mimo to lze i zvolit požadovaný počet řádků tabulky na stránku prostřednictvím rozbalovacího menu. Zásilky je poté možné filtrovat tlačítkem „Potvrdit“.

B.6.6 Jak vyexportovat zásilky do CSV?

Stačí aplikovat požadované filtry a stisknout tlačítko „Exportovat do CSV“ nad tabulkou.

B.6.7 Jak vyexportovat soupisku od vybraných zásilek?

Aby bylo možné exportovat soupisku zásilek, je nutné nejdříve vybrat konkrétního přepravce. To lze provést jednoduchým filtrováním, např. Česká Pošta v rozbalovacím menu nad tabulkou. Poté, co jsou zásilky vyfiltrované, zobrazí se na každém řádku v prvním sloupci zaškrtávací pole. Zde je možné vybrat požadované

zásilky (lze i několik najednou pomocí klasického výběru za stisku `SHIFT`). Poté je již možné dole pod tabulkou v rozbalováním menu „Spustit akci na vybraných“ zvolit „Vygenerovat soupisku“.

B.6.8 Jak poznám, že byla zásilka nahrána dnes?

Docentry na řádku se zásilkou v hlavní tabulce bude označeno modře, jako je k vidění na prvním řádku tabulky na obrázku B.1.

B.6.9 Mohu zásilku opakovaně nahrát?

Ano, za podmínky, že se zásilku nepodařilo nahrát k přepravci (její řádek je označený červeně). Stačí upravit požadovaná data v SAP a spustit opětovné nahrání.

B.7 Ovládání

V této části si popíšeme velice jednoduché ovládání naší aplikace. Jelikož jsou prakticky jediné ovládací prvky na stránce `http://192.168.1.117/odeslani/`, budeme se věnovat právě té.

B.7.1 Odeslání zásilek Zásilkovnou

Pro spuštění akce stačí stisknout tlačítko „Odeslat nové objednávky“ pod nápisem „Zásilkovna“. Odeslání zásilek doprovází několik dalších akcí. První z nich je nahrání či aktualizace dat ze SAP (pouze těch, které jsou přiřazeny Zásilkovně). Poté se program pokusí odeslat data k přepravci. O celé své práci informuje v tabulce zobrazené pod tlačítkem. Po úspěšném zpracování dat se spustí odeslání nových dat do SAP a uživateli jsou nabídnuty odkazy pro vygenerování soupisky a štítků.

B.7.2 Odeslání zásilek Českou Poštou

Pro spuštění akce stačí stisknout tlačítko „Nahrát a odeslat data“ pod nápisem „Česká Pošta“. Odeslání zásilek doprovází několik dalších akcí. První z nich je nahrání či aktualizace dat ze SAP (pouze těch, které jsou přiřazeny České Poště). Poté se program pokusí odeslat data k přepravci. O celé své práci informuje v tabulce zobrazené pod tlačítkem. Vzhledem ke způsobu, kterým jsou data odesílány k České Poště, je nutné, aby uživatel vyčkal zhruba 30 vteřin a poté pokračoval stiskem tlačítka „Získat výsledek“. Tím se z České Pošty program pokusí o nahrání sledovacích čísel a označí zásilky jako „odeslané“. Po úspěšném zpracování dat se spustí odeslání nových dat do SAP a uživateli jsou nabídnuty odkazy pro vygenerování soupisky a štítků.

B.7.3 Aktualizace stavů zásilek Zásilkovny

Aktualizace stavů dříve odeslaných zásilek (aktuálně je nastaveno 20 dní zpět), je prováděna jednoduše stiskem tlačítka „Aktualizovat staré objednávky“. Pro-

střednictvím toho jsou nahrány nové stavy od přepravce a ty jsou aktualizovány v SAP.

B.7.4 Aktualizace stavů zásilek České Pošty

Aktualizace stavů dříve odeslaných zásilek (aktuálně je nastaveno 20 dní zpět), je prováděna jednoduše stiskem tlačítka „Aktualizovat staré objednávky“. Prostřednictvím toho jsou nahrány nové stavy od přepravce a ty jsou aktualizovány v SAP.

B.8 Django administrace

Aby bylo možné jednoduše a rychle upravovat data aplikace, má přihlášený uživatel přístupnou jednoduchou administraci pro správu dat dostupnou na adrese `http://192.168.1.117/admin/`.

B.8.1 Zavedení nového uživatele

Tvorba nového uživatele je prováděna v administraci na adrese `http://192.168.1.117/admin/`. Tam, v sekci „AUTHENTICATION AND AUTHORIZATION“ po stisku řádku „Users“, bude zobrazený výpis všech uživatelů. Nového uživatele přidáme pomocí tlačítka vpravo nahoře „ADD USER“.

B.8.2 Úprava dat o zásilkách

Úprava dat o zásilkách (v případě, kdy by charakter problému z nějakého důvodu neumožňoval přímou úpravu v SAP), je prováděna v administraci `/admin` v sekci „APP“. Po stisknutí odkazu na řádku „Zásilky“, bude zobrazený výpis všech zásilek, které byly do aplikace nahrány. V detailu každé z nich lze přímo upravovat jednotlivé informace o zásilkách. Tučně označené řádky jsou povinné. Uložení proběhne stiskem tlačítka „Save“ dole na stránce.

B.8.3 Úprava stávající přepravní společnosti

Úpravu stávající přepravní společnosti lze provést výběrem „APP“. Po stisknutí odkazu na řádku „Přepravní společnosti“ v zobrazené tabulce, je nutné vybrat konkretní přepravní společnost. V případě změny hodnoty v poli „Name parameter“ může přestat aplikace správně fungovat, neboť na toto pole ja navázána logika programu. Pole RGB a Name je možné libovolně upravovat.

C. Administrátorská příručka - API

V této příručce si popíšeme, jak je API (resp. obě jeho části *Master* a *Uživatelská*) *Firmy* nasazené do produkčního prostředí a jak ho obecně spravovat.

C.1 API - master aplikace

Master aplikace našeho API je vytvořena pro Windows a musí být spuštěna u instance SAP B1, ke které se připojuje. V rámci vnitřní firemní sítě je API dostupné na adrese `192.168.1.106:5010`.

C.1.1 Požadavky na server

Jak již bylo zmíněno, musí se jednat o Windows Server, na kterém je nutná přítomnost SAP Business One vč. DI API SDK a Microsoft SQL databáze.

C.1.2 Nasazení aplikace

Protože jsme při nasazení aplikace byli poměrně omezeni faktem, že si *Firma* z logických důvodů nepřála jakékoli „větší“ doinstalace do serveru, a že SAPB1 využívá `Apache` ve své výchozí instalaci, nezbylo moc možností, jak aplikaci „stabilně“ nasadit. Proto jsme se rozhodli ke spuštění aplikace v konzole na pozadí. Časovanou úlohou spouštíme skript, jenž testuje „živost“ aplikace.

Spouštění aplikace

Aplikaci spouštíme prostřednictvím skriptu `server.py`, kde využíváme knihovnu `cherrypy`. Neboť v `Cherrypy` „hostujeme“ cizí WSGI aplikaci (konrkétně `Flask`) prostřednictvím `cherrypy.tree.graft`. Tohoto přístupu bylo i mimo jiné využito, abychom mohli hostovat paralelně dvě aplikace. Jelikož máme jeden Flask objekt (z `manage.py` přistupující do testovací databáze) a druhý (využívající produkční databázi SAP B1), pomocí `cherrypy` jsme schopni obě aplikace sloučit v jednu a odlišit je pouze cestou, kterou k nim je přistupováno.

Aplikace se jednoduše spustí za využití `cherrypy.subscribe()` funkce.

Skript pro testování „živosti“ aplikace

Abychom mohli průběžně kontrolovat, zda-li aplikace pracuje, byl vytvořený velice jednoduchý skript `ping_master.bat` využívající `netstat` ke kontrole, zda-li na specifikovaném portu aplikace „poslouchá“. Pokud ano, nic se neděje. Pokud ne, spustí specifikovanou aplikaci. Skript se spouští jednoduše. Jako první argument je předaný port, na kterém je aplikace očekávaná, a jako druhý předáme skript, jenž se provede, pokud na daném portu skript nic nenajde. Příklad použití je `./ping_master.bat 5010 ./start_master.bat`. Ve skriptu `start_master.bat` máme pouze klasické spuštění představeného Python skriptu `server.py`.

Časovaná úloha

Pro průběžnou kontrolu „živosti“ aplikace jsme vytvořili časovanou úlohu spouštějící skript z předchozí části. Jedná se jednoduchý *úkol* v programu `Windows Task Manager`, jenž každou minutu spouští příkaz představený v minulé části textu.

C.2 API - uživatelská aplikace

Pro Uživatelskou aplikaci API byl vytvořený separovaný virtuální server uvnitř *Firmy* s nainstalovaným Ubuntu 18.04 Server.

C.2.1 Požadavky na server

Protože, jak si popíšeme v další části, je aplikace nasazená v Docker kontejneru, prakticky jediné omezení, které je na nás kladeno u Uživatelské aplikace, jsou požadavky přímo Docker Engine. Ten je podporovaný v Desktop verzi *>= macOS 10.14* (v době psaní příručky pouze pro Intel macOS) a *>= Windows 10* obě s minimem 4GB RAM. Docker Server má podporu pro distribuce Linux *CentOS*, *Debian*, *Fedora*, *Raspbian*, a *Ubuntu*. Docker Server má jediné HW požadavky kladené ve formě architektury daného CPU.

C.2.2 Nasazení aplikace

V této části si popíšeme, jak je aplikace nasazená a jak jsme při práci využili docker-compose.

Webový server

Webový server využívaný pro poskytování API je Nginx. Používáme ho jako proxy poskytující WSGI aplikaci s naším API. Jeho velice jednoduchá konfigurace je uvedena v souboru `proxy/default.conf`.

Pro tento server máme vytvořený separátní kontejner definovaný v `proxy/-Dockerfile` pojmenovaný jako `michal_nginx_1`.

Databáze

Naše PostgreSQL databáze je hostována jako další Docker kontejner. Využíváme `docker image` *postgres* s namapovaným portem 5481 na 5432 uvnitř kontejneru. Kontejner je pojmenoaný jako `michal_db_1`.

Aplikace

Samotná aplikace je hostována v kontejneru `michal_api_1`. Dockerfile pro sestavení kontejneru je specifikovaný v souboru `Dockerfile` v kořenové složce uživatelského API. Kontejner je odvozený od `docker image` *python:3.8-alpine*.

docker-compose

Protože využíváme poměrně hodně kontejnerů k vzájemné spolupráci, využili jsme nástroj `docker-compose`. Ten má uvedenou konfiguraci v souboru `docker-compose.yml` v kořenovém adresáři uživatelského API. Specifikuje tři kontejnery, které na sobě vzájemně závisí a udává jednotnou konfiguraci.

C.2.3 Práce s kontejnery

Protože se občas stane, že je nutné kontejner vypnout/zapnout, restartovat či nahlédnout do jeho logů, tak si v této části obecně popíšeme, jak s kontejnery pracovat. Postup je aplikovatelný téměř na každý Docker kontejner.

Výpis všech aktivní kontejnerů

Vypsání aktuálně pracujících kontejnerů je provedeno příkazem `docker ps`

Výpis všech aktivní i neaktivních kontejnerů

Vypsání aktuálně pracujících i nepracujících kontejnerů je provedeno příkazem `docker ps -a`

Vypnutí kontejneru

Vypnutí konkrétního kontejneru je provedeno příkazem:
`docker stop nazev-kontejneru`

Spuštění kontejneru

Spuštění konkrétního kontejneru je provedeno příkazem:
`docker start nazev-kontejneru`

Restart kontejneru

Restartování konkrétního kontejneru je provedeno příkazem:
`docker restart nazev-kontejneru`

Log kontejneru

Výpis logu konkrétního kontejneru je provedeno příkazem:
`docker logs nazev-kontejneru`

C.2.4 Aktualizace aplikace

V případě vzniku nové verze aplikace je nutné ji nahrát na server. To je aktuálně prováděno postupem, kdy je aplikace prostřednictvím FTP přesunuta na server a Docker kontejner je restartován. Případně se celý kontejner přestaví, to záleží na velikosti změny.

C.2.5 Přesunutí aplikace na jiný server

V případě nutnosti přesunout aplikace na jiný server (v rámci firemní sítě, neboť API musí nutně komunikovat s Master aplikací), je třeba mít na daném serveru nainstalovaný `Docker Engine` vč. `docker-compose`. Poté stačí celou složku přesunout na požadovaný server. Je potřeba upravit cestu v souboru `./wsgi.py` a adresu databáze definovanou v *config* Flask aplikace v souboru `./server/instance.py` v poli `SQLALCHEMY_DATABASE_URI`. V kořeni projektu poté stačí spustit příkaz `docker-compose up -build -d`. Tímto spustíme kompilaci kontejnerů a jejich následné spuštění na pozadí.

D. Administrátorská příručka - aplikace pro komunikaci s přepravci

D.1 Požadavky na server

Jelikož, jak si popíšeme v další části, je aplikace nasazená v Docker kontejneru, tak jediné omezení, které je na nás kladeno u Uživatelské aplikace, jsou požadavky přímo Docker Engine. Ten je podporovaný v Desktop verzi *>= macOS 10.14* (v době psaní příručky pouze pro Intel macOS) a *>= Windows 10* obě s minimem 4GB RAM. Docker Server má podporu pro distribuce Linux *CentOS*, *Debian*, *Fedora*, *Raspbian*, a *Ubuntu*. Docker Server má jediné HW požadavky kladené ve formě architektury daného CPU.

D.2 Nasazení aplikace

V této části si popíšeme, jak je aplikace nasazená a jak jsme při práci využili docker-compose.

D.2.1 Webový server

Webový server využívaný pro poskytování naší aplikace je Nginx. Používáme ho jako proxy poskytující WSGI aplikaci. Jeho velice jednoduchá konfigurace je uvedena v souboru `proxy/default.conf`.

Pro tento server máme vytvořený separátní kontejner definovaný v `proxy/-Dockerfile`, pojmenovaný jako `zasilky_deploy_nginx_1`.

D.2.2 Databáze

Naše PostgreSQL databáze je hostována jako další Docker kontejner. Využíváme `docker image` *postgres* s namapovaným portem 5481 na 5432 uvnitř kontejneru. Kontejner je pojmenoaný jako `zasilky_deploy_db_1`.

D.2.3 Aplikace

Samotná aplikace je hostována v kontejneru `zasilky_deploy_zasilky_1`. Dockerfile pro sestavení kontejneru je specifikovaný v souboru `Dockerfile` v kořenové složce uživatelského API. Kontejner je odvozený od `docker image` *python:3.8-alpine*.

D.2.4 docker-compose

Protože využíváme poměrně hodně kontejnerů, které musejí spolupracovat, využili jsme nástroj `docker-compose`. Ten má uvedenou konfiguraci v souboru

`docker-compose.yml`, v kořenovém adresáři uživatelského API. Specifikuje tři kontejnery, které na sobě vzájemně závisí a udává jednotnou konfiguraci.

U všech aplikací uvedených v konfiguračním souboru je uvedený parametr `volume`, jenž nám umožňuje zrcadlit konkrétní složku kontejneru na lokální složku serveru. Díky tomu stačí při jakékoli změně ve zdrojových souborech jednotlivých aplikací kontejner pouze restartovat bez nutnosti celé „přestavby“.

D.2.5 Práce s kontejnery

Jelikož se občas stane, že je nutné kontejner vypnout/zapnout, restartovat či nahlédnout do jeho logů, tak si v této části obecně popíšeme, jak s kontejnery pracovat. Postup je aplikovatelný téměř na každý Docker kontejner.

Výpis všech aktivní kontejnerů

Vypsání aktuálně pracujících kontejnerů je provedeno příkazem `docker ps`

Výpis všech aktivní i neaktivních kontejnerů

Vypsání aktuálně pracujících i nepracujících kontejnerů je provedeno příkazem `docker ps -a`

Vypnutí kontejneru

Vypnutí konkrétního kontejneru je provedeno příkazem:
`docker stop nazev-kontejneru`

Spuštění kontejneru

Spuštění konkrétního kontejneru je provedeno příkazem:
`docker start nazev-kontejneru`

Restart kontejneru

Restartování konkrétního kontejneru je provedeno příkazem:
`docker restart nazev-kontejneru`

Log kontejneru

Výpis logu konkrétního kontejneru je provedeno příkazem:
`docker logs nazev-kontejneru`

D.3 Aktualizace aplikace

V případě vzniku nové verze aplikace, je nutné ji nahrát na server. To je aktuálně prováděno postupem, kdy je aplikace prostřednictvím FTP přesunuta na server a Docker kontejner je restartován. Případně se celý kontejner přestaví, záleží na velikosti změny.

D.4 Přesunutí aplikace na jiný server

V případě nutnosti přesunout aplikace na jiný server, je nutné mít na daném serveru nainstalovaný `Docker Engine` vč. `docker-compose`. Poté stačí celou složku přesunout na požadovaný server a v jejím kořeni spustit příkaz `docker-compose up -build -d`. Tímto spustíme kompilaci kontejnerů a jejich následné spuštění na pozadí.

D.5 Packeta API

Protože naše aplikace komunikuje s Packeta API kvůli předávání dat přepravci Zásilkovna, je potřeba se k API přihlásit. To je v tomto případě řešeno předáním unikátního API klíče přiřazeného každému zákazníkovi. Klíč zasíláme s každým požadavkem na API.

D.5.1 Zavedení nového API klíče

Pokud se z nějakého důvodu API klíč změní, je možné ho upravit v administraci `http://192.168.1.117/admin/` v záložce `APP > Konfigurace > Defaultní konfigurace` v poli `Zasilkovna api password`.

D.6 Česká Pošta B2B API

B2B API České Pošty využívá k autentizaci Postsignum certifikát vystavený na žádost *Firmy*. Certifikát je uložený ve složce `cert/ceskaposta` a má nastavené velice univerzální heslo `1234` neboť u formátu `.p12` je heslo povinné.

D.6.1 Zavedení nového Postsignum certifikátu

Protože má certifikát nějakou platnost, je nutné ho průběžně prodlužovat. Postup je uvedený na stránce `https://www.postsignum.cz/isignum.htm` vč. exportu do formátu `.p12` (je nutné zachovat identické heslo). Certifikát poté stačí nahradit v uvedené složce a přehrát soubor `ca4_2021.p12`. Po nahrání certifikátu na server stačí kontejner restartovat příkazem `docker restart zasilky_-deploy_zasilky_1`.

D.7 Časované úlohy

Abychom nemuseli nutně spoléhat na uživatele v tom, že bude spouštět průběžné aktualizace zásilek, byly vytvořeny cron úlohy spouštějící `django-admin commands`. Obecně se jedná o způsob, kterým je možné spouštět Django kód z příkazové řádky (podobně jako v `django shell`).

D.7.1 Časově spouštěné úlohy pro nahrání nových zásilek do SAP

V souborech `app/management/commands/update_new_ceskaposta.py` a `app/management/commands/update_new_zasilkovna.py` máme připravené skripty, využívající již vytvořené funkce z `zasilkovna.logic` resp. `ceskaposta.logic` pro nahrání nových zásilek do SAP společně s odesíláním notifikací na `Telegram Messenger`.

Abychom mohli tyto skripty jednoduše spouštět z `crontab`, vytvořili jsme si skripty `/cron_update_new_zasilkovna.sh` resp. `/cron_update_new_ceskaposta.sh`, které jsou spouštěny každý všední den ve 14 a 15 hod. (pro jistotu, kdyby během nahrávání zásilek došlo k chybě). Obsluha totiž zásilky standardně nahrává mezi 12. a 13. hodinou. V 17. hodin je potom spouštěný externí skript, který zákazníkům se zásilkami označenými jako „odeslané“ odesílá emailovou notifikaci se sledovacím číslem zásilky.

D.7.2 Časově spouštěné úlohy pro aktualizaci stavů

V souborech `app/management/commands/update_old_ceskaposta.py` a `app/management/commands/update_old_zasilkovna.py` máme připravené skripty využívající již vytvořené funkce z `zasilkovna.logic`, resp. `ceskaposta.logic` pro aktualizaci stavů dříve odeslaných zásilek do SAP společně s odesíláním notifikací na `Telegram Messenger`. Oba tyto skripty očekávají jako první parametr číslo označující stáří zásilky ve dnech.

Abychom mohli tyto skripty jednoduše spouštět z `crontab`, vytvořili jsme si skripty `/cron_update_old_zasilkovna.sh` resp. `/cron_update_old_ceskaposta.sh`, které jsou spouštěny každý všední den o půlnoci resp. v 1 hodinu ráno. Není totiž nutné stavy zásilek aktualizovat častěji, neboť standardně nikdo stavy natolik rychle nekontroluje v SAP. Tento čas byl zvolen, aby zápis co nejméně vytěžoval SAP i server hostující aplikaci během pracovní doby.