FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Integrační platforma pro LMS
Student:	Bc. Martin Suchan
Vedoucí:	Ing. Jaroslav Kuchař, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Webové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2024/2025

Pokyny pro vypracování

Cílem práce je vytvořit integrační platformu pro existující LMS (Learning Management System) umožňující především přenos dat ze vzdálených služeb (FTPS, SFTP, REST API, SOAP API) a jejich následné parsování do/z LMS, které komunikuje pomocí REST API služby. Integrační platforma bude schopna zpracovávat formáty CSV, XML a JSON. Jednotlivé integrace musí být snadno spravovatelné pomocí konfiguračního souboru konkrétní integrace.

- Seznamte se s existujícími řešeními a podobnými nástroji/platformami.
- Analyzujte požadavky na integrační platformu.
- Navrhněte architekturu a jednotlivé části zahrnující minimálně:
  - způsob zpracování dat s možností spojení více zdrojových dat do jednoho,
  - způsob transpozice zdrojových dat,
  - formát konfiguračního souboru integrace,
  - doménový model integrační platformy.
- Řešení implementujte a zaměřte se na:
  - jádro platformy a plánovač,
  - parser dat,
  - komunikaci přes protokoly FTPS a SFTP,
  - komunikaci přes webové služby (REST API, SOAP API) z pozice klienta,
  - komunikaci přes webové služby (REST API, SOAP API) z pozice serveru.
- Výsledné řešení řádně otestujte a dokumentujte.

Elektronicky schválil/a Ing. Jaroslav Kuchař, Ph.D. dne 7. září 2023 v Praze.

Diplomová práce

INTEGRAČNÍ PLATFORMA PRO LMS

Bc. Martin Suchan

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Jaroslav Kuchař Ph.D.
11. ledna 2024

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Chtěl bych poděkovat vedoucímu diplomové práce Ing. Kuchařovi, Ph.D. za odborné rady, ochotu a trpělivost, které mi poskytl během vytváření této diplomové práci.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

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

V Praze dne 11. ledna 2024 .....

Abstrakt

Cílem diplomové práce je vytvoření integrační platformy pro sjednocení různých typů existujících i budoucích integrací na již existující vzdělávací systém tak, aby správu integrací dokázal obsluhovat i administrátor bez znalosti programování. Místo programování integrací bude stačit napsat metadatový soubor ve formátu JSON, ve kterém je integrace popsána včetně způsobu získávání dat od klienta, ať už se jedná o práci se soubory na lokálním či vzdáleném serveru (SFTP, FTPS), o komunikaci se vzdálenou webovou službou nebo nabídnutí serveru s vlastní webovou službou. Práce obsahuje návrh a implementaci integrační platformy a několik příkladů zápisu integrací do metadatového souboru.

Klíčová slova integrace, integrační platforma, middleware, LMS, transformace dat, webové služby

Abstract

Objective of this thesis is to create integration platform for the unification of various types of integrations on an already existing learning management system. Integration can manage any administrator without knowledge of any programming language. Integrations are written into JSON file called metadata, where data format and method of obtaining data of the integration is described. Integration platform can obtain data via local server, remote server (SFTP, FTPS), remote server with web service or can provide own web service server. The thesis contains the design and implementation of the integration platform, as well as several examples of desribing integration into a metadata file.

Keywords integration, integration platform, middleware, LMS, data transformation, web services

Shrnutí

Analýza požadavků

První kapitola obsahuje analýzu všech dopředu známých požadavků na integrační platformu. Popisuje, jaké typy připojení bude muset integrační platforma podporovat a s jakými formáty dat bude muset pracovat. Zároveň je zde uvedeno několik možných existujících řešení a důvody, proč je nelze využít.

Ukázky existujících integrací

Druhá kapitola obsahuje popis několika existujících integrací mezi vzdělávacím systémem a systémy klientů.

Návrh zápisu integrace

Ve třetí kapitole je navržen kompletní formát metadat integrace, pomocí kterých je integrace popsána. Již v této kapitole je kladen důraz na návrh některých funkcí integrační platformy sloužící k transformaci přijatých dat tak, aby z nich mohla vzniknout data očekávaná na výstupu.

Návrh implementace

Čtvrtá kapitola obsahuje návrh implementace integrační platformy včetně doménového modelu s popisem jednotlivých tříd. Zároveň je v této kapitole seznam využitých knihoven a způsob práce s nimi.

Implementace a testování

V páté kapitole jsou popsána některá vylepšení, která se objevila v průběhu implementace integrační platformy a během základního testování zaměřeného na jednotlivé funkce. Také popisuje využití několika unit testů pro dlouhodobé testování integrační platformy v případě úpravy zdrojových kódů. Dále se zaměřuje na komplexní testování pomocí čtyř use casů uvedených v druhé kapitole. Stručně je zde popsán i budoucí plán testování všech existujících integrací, které by mělo zamezit vzniku chyb při úpravě integrační platformy nebo vzdělávacího systému.

Závěr

Šestá kapitola shrnuje závěry z průběhu implementace a testování. Zároveň jsou v ní uvedeny možnosti rozšíření integrační platformy.

Seznam zkratek

LMS	Learning Management System
HTTP	Hypertext Transfer Protocol
API	Application Programming Interface
REST	Representational State Transfer
SOAP	Simple Object Access Protocol
JSON	JavaScript Object Notation
CSV	Comma-Separated Values
XML	Extensible Markup Language
FTPS	File Transfer Protocol Secured
SSH	Secure Shell
SFTP	Secure File Transfer Protocol
ORM	Object–relational mapping
RFC	Request for Comments

Úvod

V dnešní době využívají společnosti mnoho různých softwarů, ve kterých spravují svojí organizaci. Mezi těmito programy se často nachází i vzdělávací nástroje, které jsou nutné jak pro splnění zákonné povinnosti v podobě kurzů bezpečnosti práce, požární ochrany, řidičů referentů atd., tak i pro vzdělávání zaměstnanců v rámci vlastní organizace. Aby práce se všemi těmito programy v rámci jedné společnosti byla uživatelsky co nejsnazší, je zapotřebí, aby tyto programy mezi sebou uměly komunikovat. Například personální systém by měl ostatním programům umožnit získání seznamu zaměstnanců, aby personalista nemusel při nástupu nového pracovníka otevřít všechny programy, které bude zaměstnanec používat a ručně v nich vytvářet nové konto. Stejně tak ostatní programy by měly své výsledky vracet personálnímu systému, aby vedoucí měl všechny informace o zaměstnanci na jednom místě a nemusel se kvůli kontrole vzdělávání přihlašovat do LMS, ke kontrole docházky se přihlašovat do docházkového systému apod.

Dobře provedené automatizace mezi jednotlivými softwary dokážou ve firmách uspořit nejen mnoho času, ale i minimalizovat případné lidské chyby. Jako příklad lze uvést situaci, kdy personalista při potřebě provedení velkého počtu úkonů při každé sebemenší změně ve společnosti může něco opomenout, což vyvolá dominový efekt. Pokud by například u nového zaměstnance dobře uvedl požadavky na pracovní pozici, ale při vytváření konta v LMS by zapomněl přidělit jeden z deseti požadovaných kurzů, mohla by společnost v lepším případě dostat pokutu za neproškoleného zaměstnance, v horším případě by se pak mohl stát úraz na pracovišti z důvodu špatného proškolení. Pokud by zde existovala automatizace mezi systémy, mohl by personalista vytvořit konto v personálním systému, přidělit uživateli jeho pracovní pozici a automatizace by se postarala o vše ostatní. Přidělila by tedy deset kurzů podle pracovní pozice a po jejich absolvování by platnosti školeních načetla do personálního systému.

Na vytvoření integrace mezi různými systémy od odlišných dodavatelů neexistuje žádný univerzální standard, který by určoval, co, kdy a jak si mají informační systémy předávat. Z pohledu výše zmíněného vzdělávacího nástroje je tedy potřeba umět komunikovat s různými systémy, které ale mohou mít některé společnosti vytvořené na míru. Nelze ani jednotně určit, že komunikace má probíhat pouze s personálními systémy, protože mohou nastat situace, kdy je potřeba informaci o proškolení zaměstnance předat do systému, který se stará o povolení vstupů zaměstnance do oblastí společnosti, kam je bez potřebného proškolení vstup zakázán. Vytvořit tedy pouze několik standardních integrací pro populární systémy, které používá většina společností, není dostačující řešení.

Jako další varianta pro vytvoření integrace pro vzdělávací systém by mohla být implementace vlastního univerzálního REST API rozhraní, které by umožňovalo ostatním informačním systémům komunikovat s LMS a potřebná data si samovolně stahovat či vkládat. Ani tato varianta není dostatečná, protože pokud malá společnost používá mezinárodní informační systém, tak by si sama musela implementovat převodník dat, který by propojil LMS se systémem, který

potřebují. V praxi ale něco takového nefunguje, protože by takové společnosti musely zaměstnávat vývojáře, který by tyto převodníky implementoval. Ve velkých společnostech může být tento typ přenosu dat zablokován kvůli často nesmyslným vnitřním předpisům.

To znamená, že pokud dodavatel nějakého informačního systému chce vyjít svým zákazníkům vstříc, musí být maximálně flexibilní v možnosti připojení jeho systému s ostatními systémy společnosti. Z tohoto důvodu je efektivní využívat integrační platformu, která umožní dodavateli s minimálními náklady propojení informačního systému pomocí mnoha různých způsobů. Mít vlastní integrační platformu přímo na úrovni dodavatele je výhodné pro malé i velké firmy. Malé firmy se nemusí zatěžovat s vlastní integrační platformou, kterou by pravděpodobně dostatečně nevyužily a velké firmy často zajímá primárně rozpočet, který by byl vyšší v případě využití dalších nástrojů třetí strany.

Cíle práce

Cílem této práce je navrhnout takovou integrační platformu, která bude splňovat požadavky uvedené výše. Bude sloužit dodavateli vzdělávacího systému, aby mohl snadno svůj systém napojit většinou běžných způsobů na jiné informační systémy. Vzdělávací nástroj již obsahuje desítky integrací do jiných systémů, které musí být možné přepracovat do nové integrační platformy, aby se všechny nové i existující integrace daly spravovat z jedné centrální administrace. V administraci také bude docházet kromě správy integrací i k logování úspěchů a neúspěchů jednotlivých běhů. Požadavek na umožnění přepisu starých integrací do integrační platformy pomáhá generovat minimální požadavky na cíl této práce.

Aktuální integrace jsou napsané v programovacím jazyce PHP a spravuje je několik vývojářů s různými znalostmi programování bez jasně dané struktury, jak by měl zápis integrace vypadat. Většinou se jedná o jeden PHP skript, který provede všechny potřebné operace v řadě za sebou, tedy získání dat, jejich transformace a následné uložení. Během celého tohoto procesu jsou často špatně využívány prostředky mnoha nadbytečnými dotazy přímo do databáze LMS nebo opakované zpracování dat stejným způsobem. Navíc jsou aktuální integrace závislé na datové struktuře LMS, proto i menší zásah do databáze může vést k neočekávanému chování integrace. Aby navrhovaná integrační platforma zamezila plýtvání prostředků, zavedla jednotnou strukturu zápisu a nebyla závislá na struktuře databáze LMS, bude obsahovat popis každé integrace metadatově a s LMS komunikovat pouze pomocí REST API rozhraní, které LMS poskytuje a nebude sama přímo zasahovat do dat LMS. Tím se bude jednat o samostatně fungující middleware.

Metadatový zápis by měl být v přehledném formátu, aby integraci dokázal s pomocí jednoduché dokumentace spravovat i administrátor. To bude mít další vedlejší pozitivní účinek v podobě menšího vytížení vývojového týmu, který se musí momentálně zaobírat každou menší změnou v integracích. Zároveň musí být metadatový popis snadno čitelný i strojově, aby bylo vždy naprosto jasné, co má integrace provést za činnost. V budoucnu se navíc zvažuje vytvoření formulářových prvků do centrální administrace, aby administrátoři nemuseli vůbec přijít do styku s metadatovým popisem a jakoukoliv integraci by si byl schopný administrátor doslova naklikat. Tyto formulářové prvky nejsou součástí této práce, ale je s nimi potřeba počítat při návrhu formátu popisu integrace.

Aby integrační platforma splnila momentální minimální požadavky, bude muset umět pracovat s daty ve formátu CSV, JSON a XML. Pro komunikaci bude muset umět využívat REST i SOAP API z pozice klienta i serveru. Dalším způsobem přenosu dat s klientem by mělo být SFTP, FTPS, e-mail nebo přímo s MSSQL databází.

Celá integrační platforma by měla být navržena tak, aby jakékoliv nové požadavky (například nový typ připojení nebo nový typ formátu dat) byly maximálně modulární. Pokud by nový klient měl požadavek, kterému nedokáže integrační platforma vyhovět, neměl by být problém poměrně rychle novou funkcionalitu dopracovat.

Kapitola 1
Analýza požadavků

V kapitole jsou probírané minimální požadavky na integrační platformu. Jedná se o způsoby správy integrací, podporované způsoby přenosu dat mezi vzdělávacím nástrojem a ostatními informačními systémy, způsoby autorizace, možnosti kódování a šifrování dat a formáty dat, se kterými musí integrační platforma pracovat. Zároveň je zde uvedena existující integrační platforma a důvod, proč podobné integrační platformy nelze využít.

1.1 Propojení s LMS

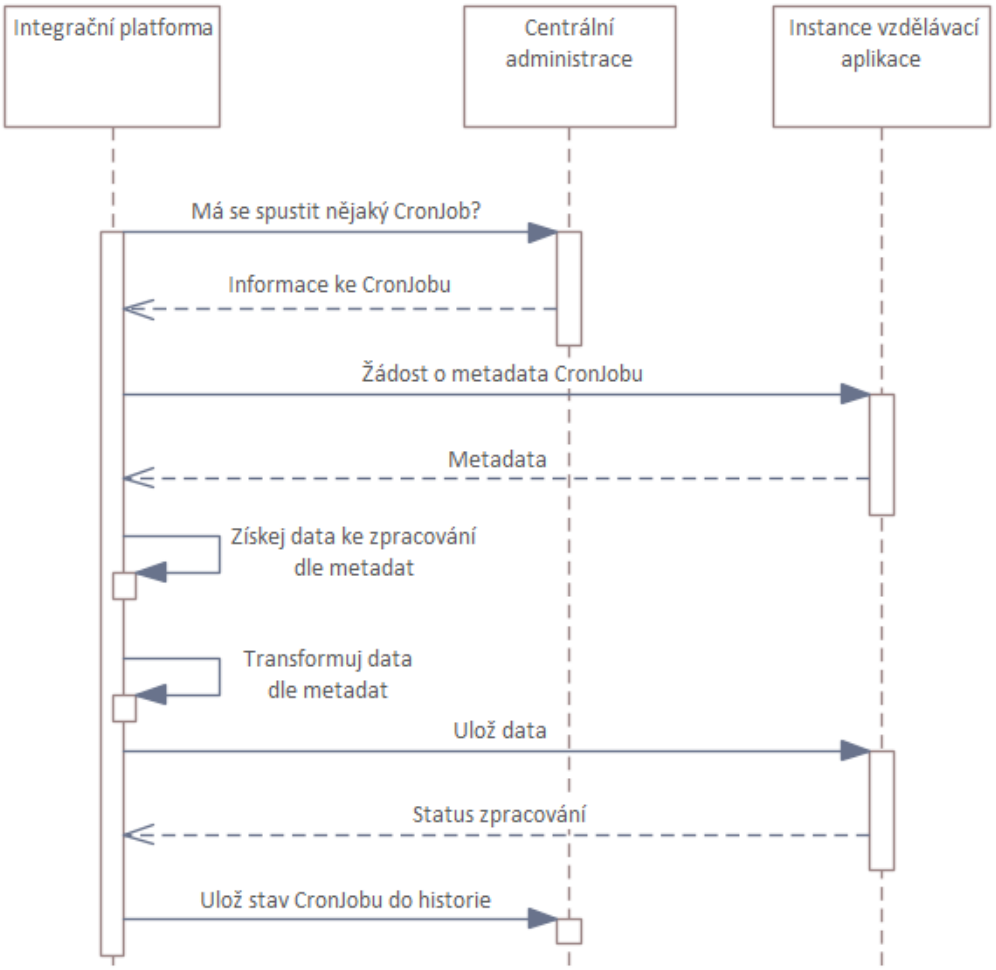
Každý klient využívající vzdělávací systém má vlastní instanci LMS, která se chová jako samostatný celek. Klienti navzájem nemají žádné informace o ostatních klientech na stejném serveru, každý má vlastní databázi a souborový prostor. Integrační platforma bude na stejném serveru jako LMS a s každou klientskou instancí bude pracovat samostatně. Seznam klientských instancí LMS a požadavky na spuštění integrace bude integrační platforma získávat z centrální administrace, kterou má již dodavatel LMS v provozu. Komunikace s konkrétní instancí LMS bude probíhat pomocí REST API rozhraní, které má LMS implementované. Princip komunikace mezi systémy během importování dat do vzdělávacího systému přes integrační platformu je ukázán na obrázku se sekvenčním diagramem 1.1.

1.1.1 Centrální administrace

Systém centrální administrace obsahuje informace a nastavení všech klientských instancí na jednom místě. V tomto systému se budou vytvářet záznamy k jednotlivým integracím pro každého klienta, které musí obsahovat minimálně následující položky:

- Název integrace.
- Klientská instance, s níž je integrace svázána.
- Je-li integrace aktivní (může být z nějakého důvodu deaktivována).
- Konkrétní čas, v jakém se má integrace automaticky spouštět.
- Následující integrace – pokud se okamžitě po této integraci má spustit nějaká další.

V centrální administraci pak musí administrátor vidět zalogované všechny běhy integrací a pokud během provádění integrace dojde k nějaké chybě, tak se chybová zpráva uloží k tomuto záznamu, aby administrátor věděl, co se stalo a jak může problém vyřešit. Struktura tabulky v databázi pro logování integrací bude navržena v této práci.

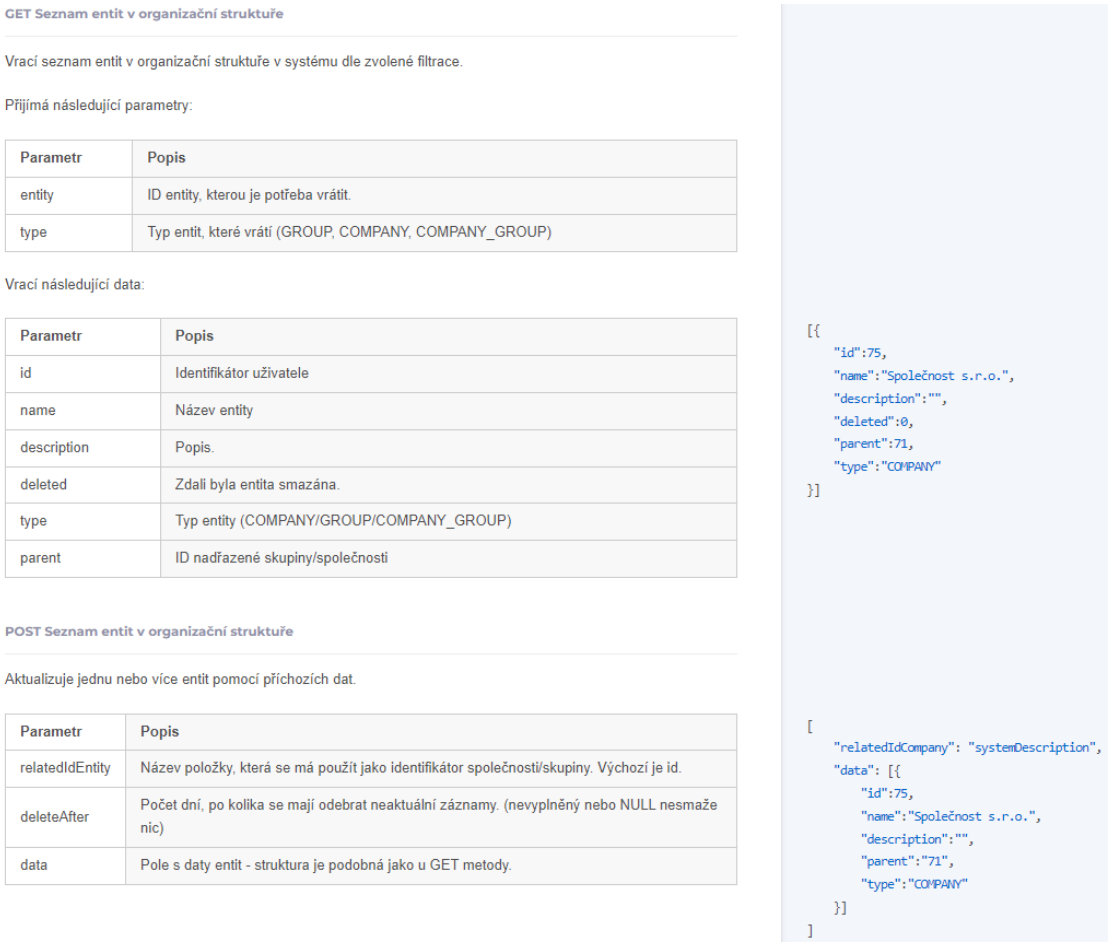


Obrázek 1.1 Komunikace mezi systémy při importu přes integrační platformu.

Z výše uvedeného vyplývá, že integrační platforma musí s centrální administrací komunikovat pro získání informace o spuštění integrace a zalogování výsledku běhu integrace. V tento moment administrace ještě neposkytuje žádné rozhraní, přes které by tato komunikace mohla fungovat a z kapacitních důvodů se momentálně o implementaci takového rozhraní ani neuvažuje. Proto bude muset integrační platforma zatím pracovat přímo s databází centrální administrace sama. Jedná se o dočasné řešení a bude zapotřebí při vytváření integrační platformy počítat s tím, že komunikace se časem změní z přímého napojení do databáze na podobné rozhraní, jaké již nyní poskytuje LMS.

1.1.2 LMS REST api

LMS, ke kterému vzniká integrační platforma, již obsahuje REST rozhraní, které mělo původně sloužit ke komunikaci klientů s LMS. Až při jeho tvorbě bylo rozhodnuto, že se raději využije na propojení s integrační platformou, která komunikaci s klienty zaštítí sama. Díky tomu, že komunikaci bude zpracovávat integrační platfoma, bude moci každý klient mít svůj REST server přizpůsobený svým potřebám a data se během procesu integrace upraví tak, aby je bylo možné přijmout na LMS REST rozhraní. To znamená, že integrační platforma sice získá například seznam uživatelů v daném formátu od LMS, ale bude ho moci dále poskytnout ze svého rozhraní



GET Seznam entit v organizační struktuře

Vrací seznam entit v organizační struktuře v systému dle zvolené filtrace.

Přijímá následující parametry:

Parametr	Popis
entity	ID entity, kterou je potřeba vrátit.
type	Typ entit, které vrátí (GROUP, COMPANY, COMPANY_GROUP)

Vrací následující data:

Parametr	Popis
id	Identifikátor uživatele
name	Název entity
description	Popis.
deleted	Zdali byla entita smazána.
type	Typ entity (COMPANY/GROUP/COMPANY_GROUP)
parent	ID nadřazené skupiny/společnosti

POST Seznam entit v organizační struktuře

Aktualizuje jednu nebo více entit pomocí příchozích dat.

Parametr	Popis
relatedIdEntity	Název položky, která se má použít jako identifikátor společnosti/skupiny. Výchozí je id.
deleteAfter	Počet dní, po kolika se mají odebrat neaktuální záznamy. (nevyplněný nebo NULL nesmaže nic)
data	Pole s daty entit - struktura je podobná jako u GET metody.


```
[{
    "id":75,
    "name":"Společnost s.r.o.",
    "description":"",
    "deleted":0,
    "parent":71,
    "type":"COMPANY"
}]
```

```
[
    "relatedIdCompany": "systemDescription",
    "data": [{
        "id":75,
        "name":"Společnost s.r.o.",
        "description":"",
        "parent":"71",
        "type":"COMPANY"
    }]
]
```


Obrázek 1.2 Ukázka z dokumentace API LMS pro zdroj OS.

v úplně jiném formátu a s jinými parametry.

REST rozhraní u LMS obsahuje dokumentaci pro každý zdroj. Jako příklad je uvedena dokumentace pro zdroj **/os** na obrázku 1.2. Rozhraní komunikuje ve formátu JSON a autorizuje se pomocí protokolu OAuth2.

1.1.3 OAuth 2.0

Jedná se o autorizační protokol, který byl navržený primárně pro ověřování oprávnění ke zdrojům, které jsou poskytnuty například přes REST API. OAuth 2.0 využívá access token, který slouží právě k ověřování oprávnění přístupu ke zdroji a o jehož generování se stará autorizační server. [1]

V tomto případě je REST API poskytované vzdělávacím systémem jak resource, tak i authorization server zároveň. Jako klient zde vystupuje integrační platforma, která má uložené hodnoty client_id a client_secret, pomocí kterých se autentizuje k získání access tokenu. S tímto tokenem již komunikuje REST API vzdělávacího systému a může získávat a spravovat data v LMS.

1.2 Technologie

Jelikož má být integrační platforma spuštěna na stejném serveru jako LMS a má být dlouhodobě spravována stejným vývojovým týmem, měla by být vytvořena ve stejném programovacím jazyce a ideálně využívat i podobné nebo stejné technologie a knihovny. I když by se tedy pro takovýto systém mohlo hodit využití jazyka Python nebo Javascriptu spouštěného přes NodeJS, bude stejně jako u LMS zvolen jazyk PHP s připojením do MariaDB.

1.2.1 PHP 7.4

Jde o skriptovací jazyk, který nachází využití primárně pro webové aplikace. Tento jazyk se i přes svojí téměř třicetiletou existenci stále aktivně vyvíjí a díky tomu vychází každý rok nová verze. Na serveru, kde bude integrační platforma spuštěna, běží momentálně kvůli jiné aplikaci verze 7.4, které na konci roku 2022 skončila podpora. [2] Proto je zřejmé, že příští rok dojde k aktualizaci na PHP 8.3 a tak jako v jiných bodech i s tímto bude vhodné během implementace integrační platformy počítat, aby kvůli případné aktualizaci serveru nebylo potřeba žádných velkých zásahů do kódu.

Pro práci s PHP knihovnami bude využit nástroj Composer, kterému stačí zadat seznam požadovaných knihoven a o jejich případné stažení a aktualizace se stará tento nástroj jednoduchými příkazy v konzoli.

1.2.2 MariaDB

Jde o relační databázi, se kterou bude muset integrační platforma pracovat, aby získala potřebná data o integracích z centrální administrace, jak bylo uvedeno dříve. Zároveň do ní bude zapisovat výsledky běhů integrací. Práce s databází by měla fungovat přes metody, které jsou již implementované v PHP. Jelikož se bude jednat o jednoduché dotazy nad databází a zároveň jde pouze o dočasné řešení, není potřeba implementovat žádné složitější prvky jako ORM.

1.3 Spouštění integrací

Integrace se budou spouštět třemi různými způsoby:

- Plánovačem úloh – CRON.
- Manuálně uživatelem.
- Pomocí zavolání endpointu API.

Jednotlivé způsoby jsou popsány níže.

1.3.1 CRON

Každých 5 minut se na serveru spustí skript, který zkontroluje, zda neměla proběhnout nějaká integrace během posledních 5 minut (integrace spouštěná tímto způsobem bude dále označována jako CronJob). Tento časový interval je volen jako minimální časový interval, který umožňuje server, na kterém bude integrační platforma spuštěna. Skript bude muset také hlídat, zda předchozí skripty spuštěné stejným způsobem proběhly správně, aby nemohlo dojít k situaci, že nějakou chybou serveru nedojde ke spuštění v jednom z intervalů a díky tomu nebudou některé CronJoby provedeny. Tím by totiž nevznikl ani záznam o neúspěšném CronJobu v logu a administrátor by se tak nedozvěděl, že je na serveru chyba.

1.3.2 Manuálně

Tento způsob bude využit primárně pro testovací účely, kdy si administrátor, který vytvoří či změní integraci, bude chtít otestovat její funkčnost na testovacím prostředí. Testovat může být potřeba opakovaně, a proto není vhodné, aby k tomu byly využity CronJoby, které by musel administrátor opakovaně pozměňovat a pak čekat až 5 minut na spuštění. Jediný potřebný parametr pro manuální spuštění je identifikátor integrace. Pokud by chtěl klient přenášet data ručně, bude mu nabídnut endpoint v API (viz. níže) a tento manuální způsob nebude využit.

Druhý účel, pro který je tento způsob vhodný jsou jednorázové importy a exporty dat. Jde například o nového klienta, který nemůže využít pravidelného spuštění integrace pomocí CronJobu a přeje si alespoň jednorázový import svých dat před spuštěním vzdělávacího systému. Tento import může být opět z personálního či podobného systému, ale také může jít o import z konkurenčního vzdělávacího systému, se kterým klient ukončil smlouvu. V takovém případě nejspíš nevyjde dodavatel předchozího vzdělávacího systému klientovi vstříc, aby jakkoliv exportovaná data z LMS přizpůsobil pro import do budoucího LMS. Pro všechny tyto jednorázové importy bude možné využít integrační platformu, protože i zde bude moci celý import provést administrátor bez nutného zásahu vývojářů. Jediný rozdíl při implementaci takovéhoto importu oproti CronJobu bude pouze v tom, že se spustí jednorázově.

1.3.3 API

Posledním možným způsobem, jak spustit integraci je pomocí zavolání API endpointu na klientské instanci LMS. To znamená, že klient zavolá daný endpoint na vzdělávací systém, a ten předá informaci integrační platformě specifickým způsobem. Tento způsob se od předchozího manuálního způsobu odlišuje primárně přenosem dat, kdy se u manuálního způsobu nepočítá s tím, že s požadavkem na server přijdou i data. Naopak u API se počítá, že data přijdou právě s požadavkem na server společně s potřebnými daty pro autorizaci požadavku.

1.4 Podporované formáty dat

Jelikož různé systémy komunikující s LMS vyžadují různé formáty dat, je zapotřebí, aby integrační platforma uměla pracovat se všemi formáty, se kterými již vzdělávací systém komunikuje. Samozřejmostí musí být návrh aplikace takový, který umožňuje modulárně přidávat další potřebné formáty i v budoucnu. Momentálně mnoho klientů preferuje na přenos dat obyčejné soubory typu CSV. Přes REST API se hodí využívat formáty jako JSON či XML. SOAP API může přijímat jeden string, který obsahuje některý z uvedených formátů, ale v běžném rozhraní se data poskytují rovnou jako pole objektů.

U všech získaných dat je kromě formátu potřeba zohlednit i znakovou sadu, kdy některá data mohou být standardně v UTF-8 kódování, ale jiná například ve znakové sadě CP-1250 nebo jiných.

```
id,firstname,lastname,courses
1,Tomáš,Matějka,1;2;3
2,Lucie,Modrá,4;5;6
```

Výpis kódu 1.1 Ukázka CSV formátu.

1.4.1 CSV/TSV

Jedná se o běžný textový formát, ve kterém jsou záznamy rozděleny na jednotlivé řádky a každý záznam má mezi svými daty oddělovače. Jako oddělovač může být využit libovolný znak, ale standardně se využívá čárka, středník nebo tabulátor. Popis formátu se nachází v RFC4180. [3]

Důležité je dodržet počet sloupců a jejich řazení pro každý záznam (řádku). Soubor může mít první řádek jako hlavičku, která pojmenovává jednotlivé sloupce. Příklad CSV formátu je na výpisu kódu 1.1.

1.4.2 JSON

JSON je standardní textový formát využívaný v integrační platformě hlavně pro REST rozhraní. Je založený na objektovém zápisu v jazyce JavaScript, ale využít ho lze kdekoliv. Na rozdíl od CSV pracuje se základními datovými typy včetně array a object, takže sice je trochu méně přehledný, ale pro změnu nabízí zanoření v jednotlivých záznamech. [4] To znamená, že jeden záznam může obsahovat hodnotu, která obsahuje pole nebo další objekt. Není v něm tedy zapotřebí hodnoty obsahující více dat dále parsovat. Ve výpisu kódu 1.2 je jednoduchý příklad JSONu včetně zanoření.

```
[
        {
                "id": 1,
                "firstname": "Tomáš",
                "lastname": "Matějka",
                "courses": [1,2,3]
        },
        {
                "id": 2,
                "firstname": "Lucie",
                "lastname": "Modrá",
                "courses": [4,5,6]
        }
]
```

Výpis kódu 1.2 Ukázka JSON formátu.

1.4.3 XML

XML je podobně jako JSON textový formát, který se také v rámci integrační platformy využívá hlavně pro REST a SOAP rozhraní. Strukturu očekávaného XML je možné definovat pomocí schématu, které určí, jak mají být výsledná data formátována. Jde například o datový typ položky nebo počet jejích výskytů. [5] Jeho hlavní nevýhodou pro použití na přenos dat oproti předchozím formátům je jeho robustnost, kdy v případě velkého množství potřebných záznamů se pracuje s výrazně většími daty. Příklad je ve výpisu 1.3.

```
<usersExport>
        <users>
                <user>
                        <id>1</id>
                        <firstname>Tomáš</firstname>
                        <lastname>Matějka</lastname>
                        <courses>
                                <course>1</course>
                                <course>2</course>
                                <course>3</course>
                        </courses>
                </user>
                <user>
                        <id>2</id>
                        <firstname>Lucie</firstname>
```

```
                        <lastname>Modrá</lastname>
                        <courses>
                                <course>4</course>
                                <course>5</course>
                                <course>6</course>
                        </courses>
                </user>
        </users>
</usersExport>
```

Výpis kódu 1.3 Ukázka XML formátu.

1.4.4 Array

SOAP API rozhraní sice komunikuje ve formátu XML, ale při použití předpřipravené knihovny v programovacím jazyce se tomuto XML formátu integrační platforma vyhne a bude pracovat až s rozbalenými daty, ve kterých určuje formát vzdálená služba pomocí WSDL. Většinou se nejedná o žádný výše uvedený formát, ale o objekt či pole objektů. S takovými daty se rovnou pracuje jako s polem objektů, kdy daný objekt obsahuje vlastnosti různých datových typů. Integrační platforma by měla pracovat se všemi daty jako s polem, proto tento formát dat již nebude vyžadovat žádný zásah.

1.5 Způsoby přenosu dat

Různí klienti využívají odlišné softwarové nástroje na správu svých dat, které chtějí mít synchronizované s LMS. To jsou například seznamy zaměstnanců, jejich povinností včetně platností absolvovaných kurzů, organizační struktura a zařazení zaměstnanců do struktury. Každý takový nástroj většinou podporuje jen několik způsobů exportu/importu dat. Takže mimo různé formáty dat musí integrační platforma podporovat i více způsobů přenosu dat oběma směry.

1.5.1 Lokální soubor

Základní způsob zpracování probíhá, pokud se pracuje se souborem uloženým na serveru ve složce klientské instance. V této složce může se soubory pracovat buď ručně administrátor s dostatečným oprávněním nebo skript, který běží na straně klienta a automaticky nahrává/stahuje data přes FTPS, SFTP nebo jiným podobným způsobem. Integrační platformě nezáleží na způsobu, jakým se soubor na server dostane, důležité je, aby tam byl v době spuštění CronJobu. Tímto způsobem je možné dělat jak importy, tak exporty přes integrační platformu a autorizace se řeší přímo na straně serveru.

1.5.2 FTPS

Tento způsob podporuje také import i export dat přes integrační platformu. Na rozdíl od předchozího způsobu se soubory nepracuje na lokálním serveru, ale na vzdáleném serveru klienta, který musí poskytnout jméno a heslo potřebné k autorizaci. Je potřeba dbát na využití FTPS a nikoliv FTP, protože FTP je považován za dnes již nebezpečný protokol, který je snadno napadnutelný, jelikož se data po síti pohybují nezašifrovaná. Zatímco FTPS využívá SSL/TLS vrstvu, která data šifruje a ta jsou tak mnohem komplikovaněji napadnutelná. [6]

1.5.3 SFTP

SSH File Transfer Protocol (SFTP) je bezpečný způsob přenosu dat. Tento protokol funguje na komunikačním protokolu SSH, na který je možné se autorizovat pomocí jména a hesla nebo pomocí jména a soukromého (private) certifikátu. V takovém případě je potřeba, aby veřejný (public) certifikát byl nahraný a povolený na straně serveru. K připojení je pak již nutné zadat jen IP adresu, port a uživatelské jméno.

1.5.4 Napojení na SQL

Další možností je napojení na SQL server, který je veřejně přístupný (ideálně však omezený na připojení z dané IP adresy) a generuje pohledy v databázi s daty, která je potřeba stáhnout do LMS. Ačkoliv se jedná o velmi neobvyklý způsob, tak i taková integrace již existuje ve vzdělávacím systému a je tedy potřeba, aby s ní uměla pracovat i integrační platforma. Tato varianta bude zatím implementována pouze ve směru stahování dat od klienta bez možnosti ukládání exportovaných dat zpět do databáze.

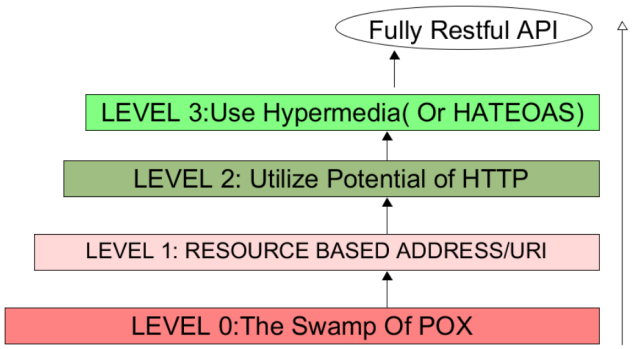
1.5.5 E-mail

Tento způsob je opět trochu specifický, protože někteří klienti požadují odesílání exportovaných dat pomocí e-mailové komunikace. Integrační platforma bude muset vyexportovat data (například v CSV formátu), přiložit je jako přílohu k e-mailové zprávě a odeslat na danou adresu. Uvedený způsob najde využití i k tomu, aby dokázal vygenerovat nejen běžné studijní exporty, ale například všechny toho dne založené uživatelské účty ve vzdělávacím systému, což je také požadavek některých klientů. Tento způsob přenosu dat bude možné využívat pouze k exportování dat, protože pokud by bylo umožněno i importování, musela by integrační platforma přijímat e-maily. Varianta pro importování dat skrze e-mail není plánována ani do budoucna, protože se jedná o atypické řešení.

1.5.6 REST API

Jedná se o komunikační rozhraní mezi serverem a klientem, které je zaměřené na data a nikoliv na volání procedur (jako třeba RPC). Samotný REST navrhl Roy Fielding ve své disertační práci, ve které mimo jiné definoval základní principy RESTu, ve kterých je uvedeno, aby každý zdroj měl vlastní unikátní identifikátor [7]. Přesto se v praxi nacházejí systémy, které nabízejí API službu označovanou jako REST a přitom nejsou splněny ani některé základní požadavky jako například uvedený unikátní identifikátor pro každý zdroj. Příklad takové služby je uveden v sekci 3.3.12. Tyto rozdíly mezi způsobem použití REST architektury daly vzniknout označení výrazu RESTful, kterým je označena REST API služba splňující všechny principy RESTu dle Roye Fieldinga. Jelikož některé REST API služby poskytují pouze některé principy, rozdělil je Leonard Richardson do 4 úrovní, podle kterých se lze rozhodnout, co je potřeba ještě doimplementovat, aby daná služba byla RESTful. Úrovně dle Richardsona jsou zobrazeny na obrázku 1.3. Tento způsob rozdělení je mezi vývojáři poměrně známý a díky němu se již dá rozhodnout, co všechno chybí danému rozhraní do označení RESTful. [8]

Dle tohoto modelu musí integrační platforma podporovat komunikaci z pohledu klienta i serveru už od úrovně 0, protože mezi existujícími integracemi jsou takové, které vyšší úroveň nepodporují. Jako příklad lze uvést integraci, kdy klient měl implementovat REST server na své straně, ale ve chvíli, kdy přišla dokumentace, tak bylo zjištěno, že vývojáři na straně klienta nevědí, co REST architektura znamená. Takový server obsahuje jeden endpoint, který vrací data oproti POST požadavku, který ve své requestBody části musí obsahovat informaci, jaká data chce získat.



Obrázek 1.3 Rozdělení RESTu do 4 úrovní podle Leonarda Richardsona. Zdroj: Rahul 2023

Dále pak pro klientskou část není zapotřebí, aby integrační platforma podporovala úroveň 3, protože s hypermédii zde není potřeba pracovat. Jakýkoliv postup integrace musí být zadaný v metadatech integrace, takže případná stažená hypermédia by byla maximálně zalogována, ale nijak nevyužita. Pro serverovou část sice také není nutně zapotřebí implementace úrovně 3, protože klient tohoto serveru bude skript. Ten se bude chovat naprogramovaně, ale rozhodně by mělo jít tuto úroveň nastavit a zbytečně ji nezablokovávat.

Autorizace skrze REST může probíhat několika způsoby, ať už standardním jménem a heslem (HTTP authentication), pomocí API klíče nebo pomocí OAuth2.0 autorizačního standardu, který je popsán výše. OAuth 2.0 je zapotřebí implementovat pouze na straně klienta, ale stejně jako u ostatních uvedených prvků by bylo dobré počítat i s implementací na straně serveru.

1.5.7 SOAP API

Podobně jako u REST API, tak i u SOAP API je komunikace rozdělena na dvě strany – server a klient. Hlavním rozdílem oproti RESTu je, že nejde o API orientované na data, ale volají se zde procedury definované na serveru. SOAP sám o sobě funguje trochu jako REST na úrovni 0 dle Richardsona. Obsahuje pouze jeden endpoint, na který probíhá volání a název operace k zavolání na serveru je součástí požadavku. Všechny zprávy mezi klientem a serverem jsou v XML formátu definované ve schématu http://schemas.xmlsoap.org/soap/envelope/.

Formální popis všech dostupných procedur, přijímaných dat a dat, která vracejí, by měy být popsán v jazyce WSDL. Jde o jazyk sloužící k popisu webové služby ve formátu XML, na základě kterého lze automaticky generovat požadavek na SOAP server. [9]

1.6 Kódování a šifrování dat

Některá data mohou být přijímána i odesílána v archivu, zakódovaná či zašifrovaná a se všemi těmito variantami si musí integrační platforma poradit. Potřeby jsou detailněji uvedené u každého typu zpracování dat.

1.6.1 Archivace - ZIP

Někteří klienti odesílají do vzdělávacího systému více stejných souborů obsahujících data různých společností, která je potřeba naimportovat. Tato data často posílají v jednom archivu, který navíc chrání heslem, aby se zvýšila bezpečnost přenosu citlivých dat. V tuto chvíli jediný potřebný typ archivu, který integrační platforma musí umět číst/vytvářet, je formát ZIP, který splňuje vše, co klienti potřebují.

1.6.2 Kódování - BASE64

Pokud naopak někteří klienti komunikují skrze webové API a posílají různá data skrze GET požadavky, mohou chtít využívat kódování, aby nedošlo k narušení URL adresy. Jako standardní typ kódování je využit BASE64, který převede text do binární podoby reprezentující se pomocí ASCII. Takové kódování je sice zhruba o třetinu delší než původní text, ale je to spolehlivý způsob, jak přenést data v URL adrese. [10] To znamená, že integrační platforma musí poskytovat možnost vstupní data dekódovat z BASE64 formátu nebo naopak výstupní data zakódovat do BASE64 formátu.

1.6.3 Šifrování - PGP

Z důvodu interních předpisů některých klientů, se nesmějí předávat data žádným způsobem ostatním systémům, pokud nejsou řádně zašifrována. Na to se v rámci aktuálních integrací ve vzdělávacím systémů využívá PGP šifrování. To je založeno na RSA algoritmu pro asymetrickou kryptografii, kdy šifrovací klíč je veřejný, takže může kdokoliv zašifrovat zprávu, ale dešifrovací klíč je soukromý, takže pouze adresát vlastnící soukromý klíč může zprávu dešifrovat. [11] Implementace šifrování stačí pouze na dešifrování, ale i zde je dobré počítat s možným budoucím rozšířením o implementaci šifrování exportovaných dat.

1.7 Logování běhu integrací

Jak již bylo uvedeno, každá integrace musí ukládat informace o svém stavu, a to jak začátek běhu integrace, tak i její konec. Pomocí takového záznamu bude možné zjistit, jak která integrace zatěžuje server a jestli se její doba běhu neblíží k maximální povolené době běhu skriptu nastavené na serveru. Tím se může předejít problémům a velké integrace rozdělit na více menších ještě dříve, než začnou padat do chybového stavu.

Během každé integrace může nastat jeden ze tří typů chyb, které je potřeba rozlišit:

- Chyba v LMS – například data nemají správný formát, chybí povinné položky nebo vznikají kolize v unikátních datech.
- Chyba v integraci – může se jednat o chybějící soubor s daty, špatné údaje pro autorizaci, chybu metadat a podobně.
- Chyba v integrační platformě – i když se bude muset každé nasazení nové verze na produkční prostředí důkladně otestovat, chyba může vzniknout kdykoliv a je potřeba ji odchytit.

Všechny výše uvedené chyby je tedy potřeba logovat s místem a časem jejich výskytu, aby bylo možné zpětně zjistit, co se během integrace stalo.

1.8 Existující řešení

Před zahájením práce na návrhu a implementaci integrační platformy je potřeba prozkoumat existující řešení, která by šla využít místo vývoje vlastní aplikace. Dostupných integračních platforem existuje poměrně velké množství, ale většina z nich nesplňuje ani základní požadavky dodavatele LMS. Jako příklad budou uvedeny tři nejvhodnější existující integrační platformy a jedno možné řešení pro usnadnění implementace vlastní integrační platformy.

1.8.1 Integrační platforma Zapier

Aplikace Zapier splňuje na první pohled téměř všechny nutné požadavky kromě jediného. V Zapieru je možné vytvořit pomocí jednoduchého rozhraní kompletně celou automatizaci, kdy se pouze vybere, co automatizaci spustí a jaké všechny akce se mají provést, přičemž akce lze vybrat z více jak 6000 dostupných. [12] V placeném programu je funkce webhooks, která umožňuje zachytávat volání na konkrétní URL adresy zaregistrované v Zapieru a stejně tak dokáže provolávat jiné URL adresy dostupné na internetu. Při správném nastavení tedy může fungovat jako REST/SOAP API server i klient. Stejně tak je zde podpora MSSQL, FTPS a SFTP připojení. Další výhodou je možnost transformace přijatých dat před předáním další službě.

Jediný, ale zásadní problém je, že služba je dostupná jako cloudové řešení a není možné on-premise řešení na vlastní server. V takovém případě by všechna data přes integrace byla předána serveru třetí strany, u kterého není možné zkontrolovat nastavení zabezpečení. I když se téměř s jistotou dá spolehnout na to, že zabezpečení aplikace Zapier bude maximálně možné, tak někteří korporátní klienti využívající integrace do vzdělávacího systému mají mnoho vlastních požadavků. Tyto požadavky musí dodavatel LMS splnit a pod hrozbou nemalých pokut i důsledně dbát na jejich dodržování. Mezi požadavky může být i striktní zákaz předávání dat třetí straně, a zatímco v některých společnostech je tento důvod pouhou byrokracií, tak v jiných jde o naprosto oprávněný požadavek, protože se přenášejí například i rodná čísla zaměstnanců.

Z tohoto důvodu nelze žádnou cloudovou službu využít a je skutečně nutné, aby integrační platforma byla spuštěna na serveru dodavatele vzdělávacího nástroje, který tak dokáže s jistotou zajistit všechny požadavky svých klientů.

Další aplikací, která vypadá jako skvělé řešení, a dokonce je postavená na PHP frameworku Symfony, je Alumio. Bohužel, stejně jako Zapier, je nabízena pouze jako cloudové řešení a nelze ji tedy využít.

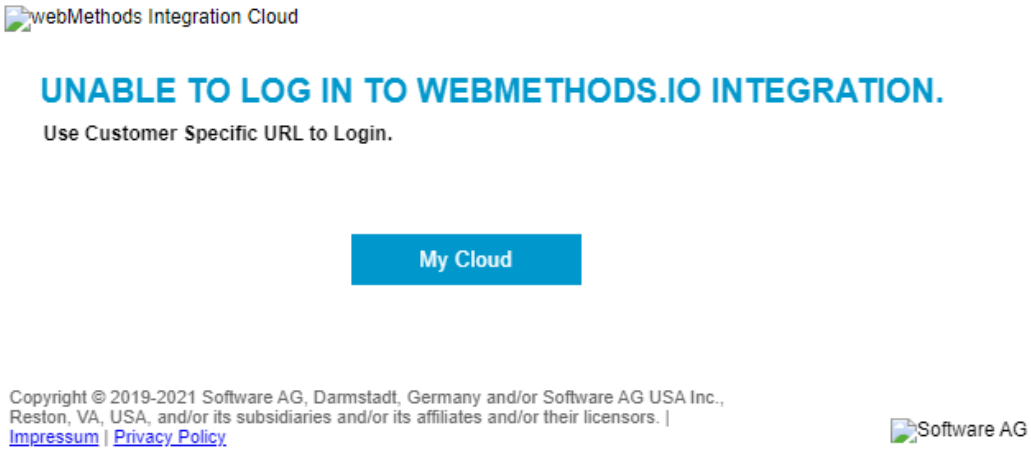
1.8.2 Integrační platforma webMethods.io

Jedná se o integrační platformu, kterou lze získat i jako on-premise řešení bez nutnosti využívat cloud. Tento systém je primárně určen pro společnosti, které chtějí pomocí integrací propojit všechny služby využívané v cloudu s aplikacemi na svých serverech. I když jde o jiný způsob využití, tak by tuto aplikaci mělo být možné využít pro potřeby vzdělávacího systému. Jediný problém by mohla být licenční smlouva, kde by tento způsob využití byl zakázán. Aplikace nabízí testovací přístup zdarma, který v době psaní této práce nefungoval a vracel chybovou hlášku viz. obrázek 1.4. Cena aplikace není poskytovatelem zveřejněna a je nutné, aby si zájemce vyžádal od dodavatele cenu na míru. Na internetu lze ovšem nalézt neověřenou informaci, že cena základní licence začíná na $1.000 za měsíc.[13]

Vzhledem k tomu, že se jedná o velké korporátní řešení, lze očekávat, že cena bude výrazně vyšší než u konkurenčních služeb. Zároveň tato robustnost a cílení na jiný způsob používání je důvodem, proč tuto aplikaci dodavatel LMS nechce využít. Chyba při pokusu o přístup ke free verzi v rozhodování také napomohla a utvrdila potřebu, aby zdrojový kód integrační platformy byl přístupný a mohl jej upravit některý z vývojářů dodavatele LMS.

1.8.3 Integrační platforma n8n

Tato aplikace splňuje to, co ostatní ne. Jedná se o opensource aplikaci, kterou je možné využívat jak v cloudu, tak na svém serveru. Lze ji využívat v docker kontejneru a pro svůj běh potřebuje nainstalovaný NodeJS. Jelikož jde o opensource, tak je možné aplikaci na svém prostředí libovolně upravovat a doplňovat. Je však napsaná v jazyce javascript a tím nesplňuje dodavatelův požadavek na shodný programovací jazyk, jako je použit pro vzdělávací systém, což je PHP. Tato aplikace v základní konfiguraci nepodporuje připojení na SFTP a FTPS. Dále by zde byl problém s implementací OAuth2.0 autorizační metody a SOAP API řešení, které by se muselo



Obrázek 1.4 Chyba při pokusu o vyzkoušení aplikace webMethods

řešit přes webhooks připojení, ve kterém by administrátoři museli pracovat nad celými XML zprávy zasílanými mezi SOAP klientem a serverem. Tento fakt tedy vyžaduje od obsluhujícího administrátora hlubší znalost architektury SOAP API a komunikace v ní.

Z těchto důvodů byla i platforma n8n shledána jako nevyhovující.

1.8.4 Laravel framework

Na rozdíl od předchozích se nejedná o IPaaS, ale o kompletní PHP framework, který je možné stáhnout a nainstalovat na vlastní server. Frameworkem se běžně označuje předpřipravený projekt obsahující vybrané knihovny, které společně vytváří základní kámen pro vytvoření vlastní aplikace. Laravel je konkrétně MVC framework, to znamená, že se skládá z kontrolérů, modelů a pohledů. [14] Laravel sám o sobě neobsahuje možnost využití SOAP API klientské části, SFTP připojení a podobně. Lze ovšem rozšířit o běžné PHP knihovny i knihovny, které jsou přímo určené pro tento framework. Během implementace by bylo potřeba využívat mnoho dalších knihoven, které nejsou součástí frameworku, zatímco například knihovny pro práci s pohledy budou v projektu zcela zbytečné, protože integrační platforma nemusí s výjimkou API serveru vracet uživateli na obrazovku žádná data.

Je tedy možné usoudit, že využití takového frameworku nepřináší nijak připravené řešení a bude potřeba vyvinout minimálně stejné úsilí jako v případě založení projektu v podobě běžné PHP aplikace s využitím Composeru a potřebných knihoven. Využití jakéhokoliv robustního frameworku je tedy pro integrační platformu naprosto nevhodné. Existují ovšem i menší frameworky, které se specializují pouze na dílčí potřeby integrační platformy, jako je například práce s REST API server i klient částí. V takovém případě by bylo potřeba složit aplikaci z několika různých frameworků, což by nejspíš mělo za následek minimálně značně nepřehledný zdrojový kód.

Kapitola 2
Ukázky existujících integrací

Tato kapitola obsahuje čtyři příklady reálných integrací, které musí být možné provádět přes integrační platformu. Slouží primárně pro lepší představu, co bude integrační platforma zpracovávat.

2.1 Klient 1 - import

Jako první příklad je uveden import zaměstnanců, na základě kterého se vygeneruje organizační struktura a nastaví oprávnění přístupů manažerů k organizační struktuře.

Způsob přenosu dat: Klient bude v pravidelnou dobu v nočních hodinách nahrávat CSV soubor přes FTPS nebo SFTP na server s LMS. Pro další zabezpečení souboru využije PGP šifrování.

Formát souboru: CSV bude jako oddělovač používat čárku a hodnoty budou uloženy v dvojitých uvozovkách. Formát souboru bude UTF8 bez BOM. Soubor bude každou noc integrační platformou zkopírován do zálohy a smazán ze složky, kam má klient přístup.

Formát dat: První řádek v CSV obsahuje hlavičku, každý další pak data uživatele. Jednotlivé sloupce:

- Příjmení
  - Obsahuje příjmení uživatele.
- Jméno
  - Obsahuje jméno uživatele.
- Titul před
  - Titul uživatele před jménem.
- Titul za
  - Titul uživatele za jménem.
- Email
  - Obsahuje e-mail, který bude zároveň použit jako username.
- Sekundární email
  - Uloží se do položky „Sekundární e-mail“ u uživatele v LMS.

- OSČ
  - Obsahuje osobní číslo, které bude použito jako unikátní identifikátor uživatelů a zároveň bude uloženo ve vlastní položce 1 v systému.
- Organizační jednotka - kód
- Organizační jednotka - název
  - Na základě této a předchozí hodnoty se vygenerují skupiny v organizační struktuře pod skupinou „Organizační jednotky“.
  - Název bude ve formátu „[kód] – [název]“.
  - Identifikátor skupiny bude „orgj-[kód]“.
  - Uživatel se zařadí do skupiny organizační jednotky na jeho řádku.
- Manažer
  - Tento sloupec může nabývat hodnot 0 a 1.
    - Pokud je 0, uživatel se zařadí do skupiny „Zaměstnanci“ a přidělí se mu zákonné kurzy pro zaměstnance.
    - Pokud je 1, uživatel se zařadí do skupiny „Vedoucí“ a přidělí se mu zákonné kurzy pro vedoucí zaměstnance.
- Profese
  - Hodnota se uloží do vlastní položky ID 3 u uživatele v LMS.
- OSČ nadřízeného
  - Osobní číslo uživatele, který je nadřízeným uživatele, kterého se týká konkrétní řádek.
  - Podle této položky se vygeneruje v organizační struktuře v LMS strom manažerů.

2.2 Klient 2 - import

Druhým příkladem je opět import zaměstnanců, generování organizační struktury, nastavení oprávnění manažerů a navíc přidělení kurzů dle pracovní pozice. Tentokrát je přenos odlišný od prvního příkladu.

Způsob přenosu dat: Integrační platforma si bude v pravidelnou dobu v noční hodinu stahovat ze SOAP API serveru data pomocí předem dané metody. Zabezpečení je řešené na serveru pomocí omezení přístupu z IP adresy a basic HTTP autorizací.

Formát dat: Data budou uložena jako pole s objekty všech uživatelů. Uživatelé, kteří budou v LMS a nebudou v importních datech budou přesunuti v LMS do „koše“. Objekty obsahují následující atributy:

- Surname
  - Obsahuje příjmení uživatele.
- Givenname
  - Obsahuje jméno uživatele.
- Email
  - Obsahuje e-mail, který bude zároveň použit jako username.

- KID
  - Obsahuje osobní číslo, které bude použito jako unikátní identifikátor uživatelů a zároveň bude uloženo ve vlastní položce 1 v systému.
- ManagerKID
  - Osobní číslo nadřazeného uživatele.
  - Slouží k vygenerování stromu organizační struktury a k nastavení manažerských oprávnění vedoucím pracovníkům.
- Location
  - Generuje skupiny do ploché organizační struktury, do které zařadí uživatele.
- WorkPosition
  - Generuje skupiny do ploché organizační struktury, do které zařadí uživatele.
  - Podle pozice přiděluje kurzy. Existuje pomocný CSV soubor, který podle pracovní pozice a toho, zda je uživatel vedoucí (je v managerKID jiného uživatele) určí, zda se má kurz přidělit všem (x) a nebo pouze nově vytvořeným uživatelům (n). Příklad je uveden v tabulce 2.1.

workPosition	manager	bpzam	bpvz	pozam	povz	pp	81
Administrativní pracovník	A		x		x	x	n
Finanční controller	A		x		x	x	n
Finanční controller	N	x		x		x	n

Tabulka 2.1 Příklad pomocného CSV souboru k přidělení kurzů dle pracovní pozice.

2.3 Klient 3 - export

Prvním exportním příkladem je získávání dat o absolvování kurzů za předchozí den, všech právě přidělených neabsolvovaných kurzech a všech nahraných kurzech v LMS.

Způsob přenosu dat: Integrační platforma bude nabízet na REST API službě možnost stáhnout si data všech absolvování v LMS v XML formátu společně se seznamem všech kurzů v LMS nad jedním zdrojem. Jedná se přesně o variantu REST API, která splňuje pouze úroveň 0 podle Richardsona, jak bylo uvedeno v kapitole 1.

Formát dat: XML schéma, které slouží k vygenerování exportu dat při požadavku na zdroj je vidět v ukázce kódu 2.1.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" id="NewDataSet">
    <xs:element name="NewDataSet">
        <xs:complexType>
            <xs:choice minOccurs="0">
                <xs:element name="Kurz">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="KurzID" type="xs:string"
                                minOccurs="0"/>
                            <xs:element name="Nazev" type="xs:string"
                                minOccurs="0"/>
```

```
                            <xs:element name="Perioda" type="xs:int"
                                minOccurs="0"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <xs:element name="Prirazeni">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="OsobaID" type="xs:int"
                                minOccurs="0"/>
                            <xs:element name="KurzID" type="xs:string"
                                minOccurs="0"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <xs:element name="Absolvovani">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="OsobaID" type="xs:int"
                                minOccurs="0"/>
                            <xs:element name="KurzID" type="xs:string"
                                minOccurs="0"/>
                            <xs:element name="Datum" type="xs:date"
                                minOccurs="0"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
            </xs:choice>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Výpis kódu 2.1 XML schéma pro generování exportu.

2.4 Klient 4 - export

Další exportní služba. Tentokrát se jedná o export absolvovaných kurzů do dvou souborů v danou hodinu.

Způsob přenosu dat: Integrační platforma bude v danou hodinu generovat dvě CSV s exportem všech absolvovaných kurzů. Jedno bude za předchozí den a druhé bude obsahovat všechna absolvování v LMS.

Formát souboru: CSV bude jako oddělovač používat středník. Formát souboru bude UTF8 s BOM.

Formát dat: Požadovaný formát CSV je demonstrován v tabulce 2.2 a jednotlivé sloupce popsány zde:

- osc
  - Osobní číslo zaměstnance (identifikátor).
- id_kurz
  - Identifikátor kurzu.
- nazev_kurzu

  - Název absolvovaného kurzu.
- perioda
  - Jak dlouhá perioda absolvování je u kurzu nastavena. Uvádí se v měsících.
- platnost_od
  - Datum absolvování kurzu.
- platnost_do
  - Datum platnosti konkrétního absolvování.

osc	id_kurz	nazev_kurz	perioda	platnost_od	platnost_do
123	1	Bezpečnost práce pro zaměstnance	12	20230607	20240607
123	2	Školení řidičů	12	20230607	20240607
456	2	Školení řidičů	12	20230608	20240608

Tabulka 2.2 Vzorová data exportovaná do formátu CSV.

Kapitola 3
Návrh zápisu integrace

V této kapitole je podrobně rozebrán formát a struktura zápisu integrací. Jsou zde uvedeny jednotlivé bloky se způsoby připojení a pro transformaci dat včetně popisu jednotlivých parametrů.

Každá jednotlivá integrace se pro snadnou správu bude zapisovat do jednoho souboru. Formát tohoto souboru musí být uživatelsky čitelný, aby jej mohli administrátoři spravovat v různých externích editorech a zároveň se musí jednat o často používaný formát, aby takové editory existovaly. Dále musí být dobře čitelný i strojově, aby integrační platforma věděla, jakými kroky má v průběhu integrace postupovat. Takové formáty jsou například JSON, YAML nebo XML a výběr mezi nimi je spíše na dohodě administrátorů v rámci společnosti, pro kterou je integrační určena. Ti vybrali formát JSON, který je popsán v sekci 1.4.2

Tento formát lze snadno editovat i v nástrojích, které jsou dostupné zdarma, jako je například program Microsoft VS Code. Jednotlivé bloky kódu lze snadno zavírat, což zvyšuje přehlednost, stejně tak jako různé podbarvení názvů parametrů a jejich hodnot. Hlavní výhodou tohoto formátu v popisu integrace je ovšem zmiňované zanoření, kdy lze jednotlivé kroky integrace zapsat do hloubky. Jako příklad lze uvést blok, ve kterém se bude řešit připojení na vzdálený server a v něm bude potřeba vyplnit autorizační údaje. Tyto údaje budou ve vlastním bloku, který bude zanořený v bloku s připojením a po jejich vyplnění v běžném editoru půjde blok zavřít a zvýšit tak přehlednost zbylé části bloku s připojením, které je ještě potřeba doplnit. Zanoření bude více ukázáno v následující sekci. Soubor ve formátu JSON, který popisuje minimálně jednu celou integraci se bude dále nazývat jako **metadata integrace**.

3.1 Struktura metadat integrace

Pro správné pochopení zápisu integrací je potřeba definovat výrazy **kontejner** a **blok**. Blok obsahuje jednu operaci, která má standardně data na vstupu, na výstupu nebo v obou směrech. Blok tato data buď získává z externího zdroje, nahrává na externí službu nebo transformuje – vždy provádí maximálně jednu z těchto operací. Kontejner obsahuje jeden nebo více bloků, které jsou stejného typu. Funkční integrace pak musí obsahovat minimálně dva kontejnery, ale standardně budou používány alespoň tři, kdy jeden získá data, druhý je transformuje a třetí uloží.

Bloky byly pojmenovány podle směru přenosu komunikace (read/write) a zdroje, se kterým pracují (data/EDUcation system) a dělí se na následující typy:

- **readDataDistributor** – načítá data z externího zdroje.
- **writeDataDistributor** – ukládá data požadovaným způsobem.

```
{
  "description": "Stručný popis integrace",
  "failAlert": {
    "addresses": "e-mailové adresy, kam se zašle varování o chybě během integrace"
  },
  "ownFunctions": [
    { "description": "vlastní funkce pro transformaci" }
  ],
  "containers": [
    {
      "type": "readDataDistributor",
      "blocks": [
        { "description": "parametry bloku..." }
      ]
    },
    {
      "type": "dataMixer",
      "blocks": [
        { "description": "parametry bloku..." }
      ]
    },
    {
      "type": "writeDataDistributor",
      "blocks": [
        { "description": "parametry bloku..." }
      ]
    }
  ]
}
```

Výpis kódu 3.1 Ukázka formátu metadata.

- **readEduBuddy** - načítá data ze vzdělávacího systému – jedná se o pouhý alias typu readDataDistributor, který je v integrační platformě přednastavený jako REST API klient s OAuth2.0 autorizací.
- **writeEduBuddy** – podobné jako výše, ale jde o alias writeDataDistributor a data zapisuje.
- **dataMixer** – slouží k filtraci a transformaci dat.

Ukázka struktury zápisu metadat je ve výpisu kódu 3.1. Jak je z ukázky patrné, jsou zde mimo kontejnery ještě další tři parametry. Parametr *ownFunctions* obsahuje vlastní metody pro transformaci dat a je podrobněji popsán v sekci 3.6.

Parametr *description* bude standardně využíván kdekoliv v metadatech a samotnou integrační platformou bude ignorován. Jedná se pouze o jakousi formu komentáře, kterou může využít administrátor, aby si popsal složitější konstrukty, které během integrace využije. Zároveň by se tento parametr mohl využít v případě, že vznikne jednoduchá formulářová aplikace, ve které by integrace bylo možné vytvořit bez nutnosti psaní JSON kódu.

Posledním parametrem je *failAlert*, který obsahuje pouze e-mailové adresy. Na tyto adresy se pošle upozornění v případě, že integrace neproběhne v pořádku. Standardně se o odesílání e-mailů o chybách během integrací bude starat centrální administrace, která je bude rozesílat v dávce 1x za 24 hodin na administrátora integrační platformy. Existují ale situace, kdy někteří klienti vyžadují informaci o chybě, protože by mohla být zapříčiněna jejich vinou – například nekonzistentními daty – a proto je nutné tuto variantu upozornění povolit.

3.2 Placeholdery

Jelikož bude potřeba pracovat v metadatech integrace s některými hodnotami dynamicky, budou zavedeny takzvané placeholdery. Ty bude integrační platforma nahrazovat různými typy dat. Placeholdery se budou zapisovat ve formátu `[__prefix:index__]`. Dále se budou lišit podle nastaveného prefixu:

- Žádný prefix – jedná se o označení vstupních dat zpracovávaných v DataMixeru. Například pro rozlišení v položkách, kde může být použit i běžný statický řetězec.
- Prefix **g** – globální placeholder je nahrazen před zpracováním metadat přímo z hodnot, které budou předávány integrační platformě z centrální administrace s informacemi o CronJobu. Díky tomuto placeholderu je možné využít totožná metadata pro více integrací, které se odlišují jen v několika málo parametrech. V praxi se může jednat o dvě společnosti s naprosto shodným typem importu, ale jinak pojmenovaným vstupním souborem ke zpracování nebo jiným zdrojem na REST API serveru.
- Prefix **d** – vkládá aktuální datum a čas a je možné ho použít ve většině parametrech v metadatech. Jako index používá masku požadovaného výstupu datumu, kdy formát této masky je totožný pro PHP funkci `date()`. Hodnotu aktuálního data lze navíc modifikovat přidáním či odebráním dnů, měsíců a roků pomocí jednoduchého řetězce na konci indexu. Například řetězec pro přidání jednoho dne má formát **+1d**, kde místo **d** lze analogicky použít **m** a **y** pro práci s měsíci a roky. Výsledný placeholder, který získá datum předchozího dne ve formátu **dd.mm.rrrr** bude zapsán `[__d:d.m.Y - 1d__]`.
- Prefix **t** – na rozdíl od prefixu *d*, vrací prefix *t* pouze časovou hodnotu v sekundách ve formátu unix timestampu. Timestamp má vlastní typ placeholderu a není obsažen v placeholderu s prefixem *d*, protože ten by pak nemohl používat čistě jen masku pro funkci `date()` a musel by pomocí podmínky určit, zda se nejedná o placeholder vracející timestamp. Proto byl z důvodu snazšího používání vytvořen další prefix pro placeholder vracející pouze timestamp hodnotu. I ten lze modifikovat přidáním či odebráním dnů, měsíců a roků stejně jako placeholder s datem. Jeho formát však změnit nelze, vždy se bude jednat o unix timestamp.
- Prefix **e** – využije se k určení místa, kam se mají vložit exportovaná data. To bude zatím potřeba pouze u e-mailové komunikace.

3.3 DataDistributor

Bloky typu DataDistributor se budou dále dělit na ty, které data zapisují a které je získávají, jak bylo uvedeno výše. Příklad zápisu obecného DataDistributoru v kódu 3.2. Popis jednotlivých parametrů je uveden zde:

```
{
  "main": false,
  "identifier": "dd_users",
  "dataSource": "dm_users",
  "settings": {
    "typeConnection": "local",
    "format": "csv",
    "formatSettings": { "description": "formát zpracovávaných dat" },
    "connectionSettings": { "description": "nastavení připojení" },
    "wrapperRemove": [],
    "wrapperAdd": [],
    "coding": null,
    "compression": { "description": "nastavení komprese dat" },
```

```
14      "encryption": { "description": "nastavení šifrování dat" }
15    }
16  }
```

Výpis kódu 3.2 Vzor DataDistributoru.

- **main** – bool hodnota, která označuje, zda se jedná o hlavní DataDistributor v integraci. Informace o hlavním DataDistributoru se budou hodit například v momentě, kdy integrace bude poskytovat API server s autorizací klientů vůči integrační platformě. Integrační platforma bude popis nastavené autorizace pro API server hledat právě a pouze u hlavního DataDistributoru.
- **identifier** – unikátní identifikátor DataDistributoru. Pokud jde o readDataDistributor, tak pomocí identifikátoru pracují ostatní bloky s daty získanými pomocí tohoto DataDistributoru.
- **dataSource** – v případě, že se jedná o DataDistributor typu write, použijí se pro zápis do cílové lokace data, která jsou uložena pod tímto identifikátorem. DataSource může obsahovat více identifikátorů uložených v poli, které před zpracováním DataDistributor spojí.
- **settings** – blok kódu s podrobným nastavením DataDistributoru.
- **typeConnection** - typ připojení pro získání či uložení dat. Povolené hodnoty tohoto parametru jsou *local, apiServer, SOAP, REST, SFTP, FTPS a e-mail.*
- **connectionSettings** – podrobné nastavení připojení. Upřesněno individuálně pro každý typ připojení v následujících sekcích.
- **format** – formát dat, která se zpracovávají. Jedná se o hodnoty *csv, xml, json* nebo *array.*
- **formatSettings** – podrobné nastavení formátu dat - upřesněno v sekci 3.3.1.
- **wrapperRemove** a **wrapperAdd** - správa zanoření dat - upřesněno v sekci 3.3.12.
- **compression**, **encryption** a **coding** - upřesněno později v sekci 3.3.2.

3.3.1 Formát dat

Formát dat je možné dále specifikovat tak, jak je uvedeno v kódu 3.3. V tuto chvíli konfigurace nebude dovolovat žádná velká nastavení, ale pouze vlastnosti, které je potřeba konfigurovat kvůli několika existujícím integracím. Primárně jde o znakovou sadu dat, se kterými se pracuje, protože některé integrace využívají *UTF-8*, jiné *CP-1250* a integrační platforma si tak musí umět poradit se všemi běžnými typy znakových sad.

```
1  {
2    "csvDelimiterColumn": ",",
3    "csvHeaders": true,
4    "encoding": "UTF-8"
5  }
```

Výpis kódu 3.3 Nastavení formátu dat.

Další dva parametry v nastavení jsou již pouze pro formát typu *CSV*. V první řadě lze zvolit oddělovač mezi položkami, protože existují integrace, kde je jako oddělovač využita běžná čárka a jiné, kde se používá středník. Dokonce i aplikace jako Microsoft Excel exportuje *CSV* formát s výchozím nastavením v České republice se středníkem jako oddělovačem, což je vzhledem k celému názvu formátu (comma-separated values) trochu zvláštní chování.

Druhou volitelnou položkou je *csvHeaders*, která dává integrační platformě informaci, zda *CSV* data obsahují v prvním řádku názvy sloupců nebo ne. Pokud jsou názvy sloupců v datech obsažené, tak v jakékoliv další práci s daty jsou využívány jako identifikátory položek v každém záznamu zdrojových dat. V opačném případě jsou data indexována od nuly.

3.3.2 Kódování, komprese a šifrování

S daty je možné provádět několik operací dle zadání. Základní z nich je kódování, kdy je potřeba některá data nepředávat ve zvoleném datovém formátu. Jako příklad lze uvést potřebu předávat data na REST API server pomocí GET metody v URL adrese, kdy je nutné se vyhnout některým speciálním znakům, a to ideálně pomocí zakódování dat. Jediná povolená hodnota do položky *coding* je v tuto chvíli *base64*, protože integrační platforma nebude mít zatím implementované žádné další kódování.

Další operací s daty je práce s archivy. Možný zápis v metadatech je zobrazen v kódu 3.4, který je zapsán pro formát typu *ZIP*, což je v tuto chvíli jediný podporovaný typ archivu. Parametr *filepath* definuje umístění hledaného souboru uvnitř archivu a pokud je vyplněn i parametr *password*, tak je využit při práci s archivem k získání/vložení dat.

```
{
  "type": "zip",
  "filepath": "data/users.csv",
  "password": "pass123"
}
```

Výpis kódu 3.4 Nastavení zip archivu.

```
{
  "type" : "pgp",
  "keyPath" : "../settings_data/private.asc",
  "passphrase" : "pass123"
}
```

Výpis kódu 3.5 Nastavení šifrování.

Poslední operací pro přípravu dat v DataDistributoru je šifrování, jehož zápis je vyobrazen na kódu 3.5. Jediné podporované šifrování je typu PGP, a i přes to, že se nejspíš bude jednat o jediný požadovaný typ šifrování, raději i zde bude integrační platforma počítat s možností budoucího rozšíření. Proto mezi požadovanými parametry bude i *type*, který bude momentálně nabývat pouze hodnoty *pgp*. Dalším parametrem je cesta k veřejnému či privátnímu klíči, podle kterého dojde k zašifrování či dešifrování dat. Posledním volitelným parametrem je heslo ke klíči v předchozím parametru.

3.3.3 Lokální soubory

První z možných typů přenosu dat, které se vyplňují do *connectionSettings* je varianta *local*. Nejdříve je nutné upozornit, že tento typ lze využít pouze v případě, že integrační platforma je spuštěna na stejném serveru jako vzdělávací systém, což je v tomto zadání splněno. Jedná se o způsob práce se soubory, které jsou umístěny na lokálním serveru v předem dané složce *integrations/data* na doméně klienta, kterého se daný CronJob týká. Mezi parametry patří cesta k souboru, která může být zanořena do dalších složek a počet dní, po kterých se má daný soubor ze serveru odstranit. Počet dní může být roven hodnotě *null*, což znamená, že se soubor nemá smazat nikdy. Hodnota *null* zároveň slouží jako výchozí hodnota. Ukázka zápisu získávání lokálního souboru je ve výpisu kódu 3.6.

```
{
  "filepath": "company/users.zip",
  "deleteAfter": 0
}
```

Výpis kódu 3.6 Nastavení lokálního připojení.

3.3.4 FTPS

Připojení na vzdálený server pomocí protokolu FTPS je definováno stejnými parametry jako získávání souboru z lokálního serveru. Navíc jsou zde nutné parametry pro připojení, jako je IP adresa či doména, port a autorizační parametry – přihlašovací jméno a heslo. Jednoduchá ukázka je na výpisu 3.7.

```
{
  "filepath": "company/users.csv",
  "deleteAfter": null,
  "host": "123.5.2.132",
  "port": 990,
  "username": "john.doe",
  "password": "pass123"
}
```

Výpis kódu 3.7 Nastavení FTPS připojení.

3.3.5 SFTP

Připojení pomocí SFTP má téměř shodný zápis s připojením přes FTPS. Rozdíly jsou zde pouze dva. Prvním je parametr *keyFilepath*, který obsahuje cestu ke klíči, pokud se SFTP připojení autorizuje pomocí přihlašovacího jména v kombinaci s privátním klíčem. Druhou změnou je, že hodnota *password* v případě autorizace pomocí klíče neobsahuje běžné heslo k přihlášení, ale heslo k privátnímu klíči, pokud ho klíč vyžaduje. Příklad zápisu SFTP připojení je v kódu 3.8.

```
{
  "filepath": "company/users.csv",
  "deleteAfter": null,
  "host": "123.5.2.132",
  "port": 22,
  "username": "john.doe",
  "keyFilepath": "settings_data/private",
  "password": "pasphrase123"
}
```

Výpis kódu 3.8 Nastavení SFTP připojení.

3.3.6 E-mail

Možnost odeslání dat pomocí e-mailové korespondence by měla nabídnout možnost odesílat data jak v příloze, tak jako součást těla zprávy. Název souboru, který se vloží do přílohy, je možné zvolit v položce *filepath*, zatímco do těla zprávy se dostane pomocí placeholderu `[__e:data__]`.

```
{
  "recipients": [
```

```
    {"john.doe@mail.cz": "John Doe"},
    {"jane.doe@mail.cz": "Jane Doe"}
  ],
  "subject": "Absolvované kurzy [__d:d.m.Y - 1d__]",
  "body": "V příloze naleznete včera absolvované kurzy.",
  "filepath": "history.csv",
  "sendEmptyAttachment": false
}
```

Výpis kódu 3.9 Nastavení e-mailové komunikace.

Položka *sendEmptyAttachment* určuje, zda se má e-mail odeslat i v případě, že nejsou žádná data k dispozici. Například pro export absolvovaných kurzů za minulý den není potřeba, aby přišel e-mail s přiloženým souborem, který je prázdný, protože předchozí den nikdo neabsolvoval žádný kurz. Položky *subject* a *body* slouží k vyplnění předmětu a těla zprávy. Do poslední položky *recipients* se vkládají e-mailové adresy, kam bude zpráva odeslána. Je možné vyplnit pouze pole adres nebo pole objektů, které mimo adresy obsahují i jméno a příjmení příjemce, tak jako je na ukázce kódu 3.9.

3.3.7 PDO

Napojení do databáze klienta se bude zapisovat v podobě, jako je uvedeno v kódu 3.10. S položkou *dsnString* pracuje přímo knihovna PDO v PHP a je o ní více napsáno v návrhové části implementace v sekci 4.4.1. V tomto momentě stačí vědět, že *dsnString* obsahuje informace pro připojení k databázi, jako její typ, název, adresu, port a znakovou sadu. Dále je pro připojení potřeba zadat *username* a *password*, což je přihlašovací jméno a heslo do databáze. Poslední dvě položky nastavují databázový dotaz, pomocí kterého získá integrační platforma požadovaná data pro další kroky integrace. V první položce *query* je samotný dotaz a pokud je potřeba do něj vložit data obsahující speciální znaky, tak jsou tyto parametry nastaveny přes položku *queryParams*.

```
{
  "dsnString": "dblib:host=8.8.8.8:63000;dbname=elearning;charset=UTF-8",
  "username": "lms",
  "password": "heslo123",
  "query": "SELECT * FROM v.uzivatele WHERE substr(osc, 1, 1) = :osc",
  "queryParams": {
    "osc": 5
  }
}
```

Výpis kódu 3.10 Nastavení připojení do vzdálené databáze.

3.3.8 REST API klient

Připojení z pozice REST API klienta musí být dobře parametrizované, protože bude fungovat jako připojení k téměř libovolné HTTP API službě. Příklad složitější konfigurace je v ukázce kódu 3.11. V první řadě je nutné umožnit nastavení HTTP metody, což půjde zvolit v parametru *crud* bez ohledu na to, zda je potřeba data získávat či vkládat. Další volitelný parametr je *customRequestBody*, který se v praxi bude vyplňovat pouze v případě, že je potřeba získat data z daného zdroje pomocí parametrů, které jsou odeslány v těle požadavku. Pokud by se tento parametr použil i v případě pokusu o vkládání dat do vzdálené služby, tak by tato položka odesílaná data přepsala. Nastavení autorizace probíhá v parametru *auth* a je popsáné v sekci 3.3.10.

Důležitou volitelnou položkou je *customHeaders*. Ta umožňuje administrátorovi doplnit libovolné hlavičky, které budou odeslány společně s požadavkem na vzdálený server. Tato funkce může být pro některé vzdálené služby nutná pro jejich správnou konfiguraci, kdy by bez správných hlaviček nevrátily žádná nebo špatně formátovaná data.

Pro komunikaci mezi integrační platformou a API serverem bude využit nástroj cURL, jak je později uvedeno v sekci 4.4.4.2 a proto poslední položkou je *additionalCurlOptions*, pomocí které lze dodefinovat mnoho různých nastavení cURL volání. Jako příklad, který lze pomocí této položky nastavit je *CONNECTTIMEOUT*, po jehož uplynutí se přestane integrační platforma pokoušet připojit na vzdálenou službu. Další užitečná nastavení jsou různé konfigurace SSL připojení a jako příklad lze uvést *SSL_VERIFYHOST*. [15] Pomocí tohoto parametru lze vypnout ověření certifikátu a i s neověřeným certifikátem provádět šifrovanou komunikaci. Toto nastavení rozhodně není doporučené na produkčním serveru, ale může nastat situace, kdy klient nemá na testovacím serveru platný certifikát a pokud se nepřenáší žádná citlivá data, lze tímto způsobem snadno vypnout ověření platnosti certifikátu na serveru. Parametr *additionalCurlOptions* by měl používat pouze zkušený administrátor, protože by špatným použitím mohl usnadnit únik citlivých dat přes integrační platformu.

```
{
  "crud": "POST",
  "endpoint": "https://nejaka.api/v1/users",
  "customHeaders": {
    "x-hr-datasource": "company"
  },
  "customRequestBody": "{\"method\": \"getUsers\"}",
  "additionalCurlOptions": {
    "CURLOPT_CONNECTTIMEOUT": 10
  },
  "auth": {
    "description": "Typ autorizace."
  }
}
```

Výpis kódu 3.11 Nastavení HTTP klienta.

3.3.9 SOAP API klient

Druhý typ připojení z pozice API klienta využívá SOAP API standard a je popsán na kódu 3.12. Podobně jako u RESTu, i zde je *endpoint* parametr, který by měl směrovat na WSDL cílové služby. Pomocí dalších parametrů *method* a *methodParams* se nastavuje název metody a parametry, které se do ní předají. Indexy v poli s parametry slouží pouze administrátorovi pro přehlednost zápisu, protože do metody se vkládají postupně a bez svého indexu. Další parametry *soapVersion*, *keepAlive* a *ssl* upřesňují verzi SOAP standardu, nastavení hlavičky *Connection* a ověřování TLS certifikátu podobně jako bylo vysvětleno u REST API klienta. Poslední parametr *auth* pak funguje k autorizaci a je popsán v sekci 3.3.10.

```
{
  "endpoint": "https://soapsluzba.cz/?wsdl",
  "method": "getUsers",
  "methodParams": {
    "company": "alfa a.s."
  },
  "soapVersion": "SOAP_1_2",
  "keepAlive": false,
  "ssl": {
    "active": true,
```

```
    "verifyPeer": true,
    "verifyPeerName": true,
    "allowSelfSigned": false
  },
  "auth": {
    "description": "Typ autorizace."
  }
}
```

Výpis kódu 3.12 Nastavení SOAP API klienta.

3.3.10 Autorizace API klienta

Autorizace u všech typů API klientů se nijak neodlišuje. Zatím lze vybrat ze tří typů autorizace a sice *httpauth*, *apikey* a *oauth2*. V bloku s autorizací musí být vždy definován právě jeden blok s nastavením autorizace. Pokud je tedy použit typ *httpauth*, už není možné implementovat *apikey*. Ukázka zápisu všech bloků je v kódu 3.13 a jejich jednotlivý popis zde:

- **httpauth** – základní typ přihlášení ke službě, kdy probíhá autorizace pomocí jména a hesla. Je možné zvolit typ ověření jako například *basic* nebo *digest* a pomocí *paramName* parametru v nadřazeném bloku určit, v jaké hlavičce se údaje odešlou. Může se využít *Authorization* (výchozí) a *Proxy-Authorization* v případě, že jde o autorizaci na proxy server. [16]
- **apikey** – autorizuje se oproti řetězci označovaného jako API KEY. Tento řetězec je v parametru *key* a odesílá se pomocí HTTP hlaviček podobně jako předchozí autorizace. Často lze zapsat tyto dvě autorizace opačným způsobem, kdy například API KEY se posílá jako uživatelské jméno a znak `x` se posílá jako heslo během běžné HTTP autorizace. Toto ovšem není pravidlo, a proto se API KEY musí implementovat mimo běžnou HTTP autorizaci. Druhým důvodem je přehlednost, kdy je ze zápisu integrace okamžitě jasné, jestli se používá HTTP autorizace nebo API KEY. To se bude případně hodit i na budoucí formulářové prvky pro vytvoření zápisu integrace. Název odesílané hlavičky je ve výchozím stavu nastaven jako `x-api-key`, ale pomocí parametru *paramName* v nadřazeném bloku lze název této hlavičky vyměnit. Méně známým, ale stále obecně podporovaným způsobem přenosu API klíče je i pomocí parametrů dotazu. S tímto způsobem se v tuto chvíli v žádné existující integraci nepracuje a proto zatím nebude tato možnost povolena. Pokud by bylo potřeba tento způsob implementovat, přibyl by jeden parametr označující místo, kde má být API klíč hledán nebo by se mohla integrační platforma pokusit nalézt API klíč automaticky.
- **oauth2** – OAuth2.0 protokol byl popsán v první kapitole 1.1.3. Do parametrů *urlAuthorize* a *urlResource* se nastavují URL adresy, kde dochází k autorizaci a k práci se zdroji. Položka *grantType* určuje použitý grant type na autorizačním serveru, který může zatím být využit pouze jako *clientCredentials* s autorizačními údaji *clientId* a *clientSecret*. V dohledné době bude ovšem potřeba doimplementovat grant type *SAML 2.0 Bearer Assertion Flow*, který využívají systémy od společnosti SAP. Rozdíl mezi *clientCredentials* a *SAML 2.0 Bearer Assertion Flow* je primárně v tom, že získávání access tokenu neprobíhá s kombinací údajů *clientId* a *clientSecret*, ale pomocí *clientId* a *SAML assertion*, který integrační platforma získá od SAML 2.0 identity provider serveru. [17]

```
{
  "paramName": "Authorization",
  "httpauth": {
    "login": "user",
    "password": "password123",
    "type": "basic"
```

```
  },
  "apikey": {
    "key": "AJS8D72JDHAJ2"
  },
  "oauth2": {
    "urlAuthorize": "https://url.domena.cz/oauth2/authorize",
    "urlResource": "https://url.domena.cz/oauth2/resource",
    "grantType": "clientCredentials",
    "clientCredentials": {
      "clientId": "123456",
      "clientSecret": "789A"
    }
  }
}
```

Výpis kódu 3.13 Nastavení autorizace pro API klienta.

3.3.11 API server

Typ API serveru se nebude muset rozeznávat na straně integrační platformy, ale ke správnému zpracování komunikace dojde na straně LMS, kam budou směřovat endpointy poskytnuté klientům. Toto je více popsáno v kapitole s návrhem implementace v sekci 4.4.3.1. Pro integrační platformu bude nutné znát hlavně formát dat. V bloku s konfigurací API serveru, který může vypadat například jako v ukázce 3.14, je možné uvést statickou odpověď ze serveru, pokud například jde o import dat a klient očekává zprávu s potvrzením. Dále se již nastavuje pouze autorizace, která má podobné možnosti jako v případě API klient připojení. Na rozdíl od API klienta zde ovšem není podpora OAuth2.0 ověření, ale pro změnu je zde navíc ověření dle IP adresy klienta, který se připojuje k serveru. Lze tedy odpovědi serveru omezit pouze na uvedený výčet IP adres, ze kterých bude klient k serveru přistupovat.

```
{
  "overwriteResponse": "{\"status\": \"OK\"}",
  "auth": {
    "restrictedToIpAddresses": ["123.123.123.123", "123.123.123.124"],
    "httpauth": {
      "username": "user",
      "password": "pass123",
      "type": "basic"
    }
  }
}
```

Výpis kódu 3.14 Nastavení autorizace pro API server.

3.3.12 Wrappery

Parametry *wrapperAdd* a *wrapperRemove* jsou k zabalení či rozbalení dat z nějakého obalu. V XML datech se může jednat o nadřazené tagy, v JSON datech o nadřazené objekty a v datech uložených v poli o nadřazená pole. Příklad pro *wrapperRemove* lze uvést z existující integrace, která se bude muset pomocí integrační platformy připojovat na HTTP API. To vrací data v JSON formátu, kde jsou data potřebná pro další zpracování zanořena v parametru `users` (viz. ukázka kódu 3.15).

Aby se s takovou integrací mohlo pracovat, je potřeba integrační platformě ukázat, kde jsou v přijatých datech důležité informace pro další zpracování a co je pouze okolní obal k zahození.

Obecný zápis práce s wrappery v metadatech integrace bude vypadat jako na kódu 3.16. Tento konkrétní kód slouží zároveň i pro zpracování výše uvedeného příkladu.

```
{
  "method": "importUsers",
  "users": [
    "Zde jsou data zaměstnanců."
  ]
}
```

Výpis kódu 3.15 Příklad přijatých zanořených dat z API serveru klienta.

Pomocí položky *dataIndex* dojde k zanoření v datech o úroveň dále podle jména tohoto indexu. Atribut *storedParsesHere* určuje, zda jde již o finální zanoření, kde se data mají hledat. To je potřeba určit, protože pomocí položky *wrapper* se lze rekurzivně zanořovat hlouběji. Další položka *defaultData* se hodí pouze pro přidávání obalu přes *wrapperAdd*. V této položce lze zvolit, zda se do wrapperu mají uložit nějaká výchozí statická data.

```
[{
  "dataIndex": "users",
  "storedParsesHere": true,
  "defaultData": null,
  "isAttribute": false,
  "wrapper": []
}]
```

Výpis kódu 3.16 Příklad zápisu nastavení, které odebere wrappery.

Pro lepší představu lze použít výše zmíněný příklad 3.15, který tentokrát nebude integrační platforma rozbalovat, ale naopak vytvářet při exportu dat ze vzdělávacího systému. Rozdílem bude název metody `exportCourses` a využití *defaultData* parametru. Zápis přidání takového obalu kolem dat je uveden v kódu 3.17.

```
[{
  "outgoingIndex": "method",
  "defaultData": "exportCourses"
},
{
  "outgoingIndex": "courses",
  "storedParsesHere": true
}]
```

Výpis kódu 3.17 Příklad přípravy zanořených dat pro export.

Poslední položka, která v žádném příkladu nebyla využita je *isAttribute*. Tato položka se hodí během přidávání obalu při exportování do XML formátu, kdy se může takto zanořená položka nechovat jako další vnořený tag, ale jako atribut nadřazeného tagu.

Přidávání či odebírání obalů lze využít i v metadatech v bloku *DataMixer*. To je umožněno hlavně proto, že stejně jako v DataDistributoru, tak i u DataMixeru se příkazy na *wrapperAdd* a *wrapperRemove* volají jako poslední operace při zpracování dat. Může se tedy stát, že data připravená pro export je potřeba ještě mezi zpracováním v DataMixeru a odesláním přes DataDistributor obalit, což kvůli pořadí provádění operací nelze v DataDistributoru udělat, protože by se data obalila až po odeslání. Teoreticky by šlo pořadí zpracování obalů nastavit podle typu DataDistributoru (read/write). Tím by zase mohl vzniknout problém, že pokud klientova služba vrací odpověď, na kterou je potřeba použít odebrání obalů, nastala by podobná situace, jaká je teď mezi DataMixerem a DataDistributorem. Bude tedy snazší i přehlednější umožnit administrátorovi nastavovat obaly v obou těchto typech kontejnerů.

3.4 EduBuddy

Podobně jako DataDistributor, tak i EduBuddy kontejner je typu *write* i *read*. Hlavní důvod podobnosti je ten, že se ve skutečnosti jedná pouze o alias k DataDistributoru nastaveného na typ přenosu REST API klient s OAuth 2.0 autorizací. EduBuddy kontejnery slouží k připojení do vzdělávacího systému, do kterého jsou známy autorizační údaje a díky získaným datům z centrální administrace i doména, na které jsou poskytnuty zdroje. Z toho důvodu je zbytečné, aby administrátor musel u každého požadavku na serveru ve všech metadatech zadávat DataDistributory odlišné pouze v autorizačních parametrech dle klienta a domény. Tento alias umožňuje pouze nutná nastavení REST API klienta, která nelze unifikovat, jako jsou *identifier*, *crud*, *zdroj* a *dataSource*. Tato základní nastavení se dále rozšiřují o několik dalších tak, jak je znázorněno v kódu 3.18. Mezi hlavní funkční parametr patří *canContinueAfterFail*, který určuje, zda se může pokračovat v integraci, pokud vzdělávací systém vrátí na tomto zdroji chybový stav. Výchozí hodnota je nastavena na `false`, ale existují případy, kdy není potřeba kvůli chybě při generování dat v jedné části vzdělávacího systému neumožnit zpracování následujících dat, která by se odeslala až v následujícím *EduBuddy* bloku.

```
{
  "crud": "GET",
  "identifier": "eb_usersCourses",
  "dataSource": "dm_users",
  "endpoint": "/UsersCourses",
  "canContinueAfterFail": false,
  "settings":
  {
    "params": {
       "relatedIdUser"   : "username",
    }
    "filtration": {
      "filterBy": {
        "completedDateTo": "[__d:Ymd__]"
      },
      "orderBy": {
        "completedDateTo": "DESC"
      },
      "limit": {
        "from": 50,
        "num": 50
      }
    }
  }
}
```

Výpis kódu 3.18 Zápis EduBuddy bloku.

3.4.1 Dynamické parametry

V objektu *settings* se nachází několik parametrů, které se odesílají do vzdělávacího systému. Data se dělí do několika objektů:

- **params** – v tomto objektu jsou atributy nastavující způsob zpracování dat na straně LMS. Tyto parametry jsou dostupné pro každý zdroj zvlášť v dokumentaci LMS API a je možné je nalézt na ukázce z dokumentace na obrázku 1.2. Výše v příkladu je uvedena velmi užitečná položka *relatedIdUser*, která se stará o to, že integrační platforma nemusí znát UID uživatelů

v LMS. Pro jejich identifikaci během integrace je použita hodnota uložená v LMS pod hodnotou položky *relatedIdUser*.

- **filtration** – v tomto objektu jsou uloženy parametry, pomocí kterých LMS filtruje a řadí data z požadovaného zdroje. Ve filtraci musí fungovat datumové placeholdery, aby bylo možné vyfiltrovat například všechny záznamy, které byly pozměněny v posledních několika dnech. Filtrování má sice připravené i řazení a stránkování, ale LMS zatím tyto dva parametry nemá implementované, takže zde se bude jednat pouze o přípravu do budoucna, kdy už nebude potřeba zasahovat do kódu integrační platformy.

Oba výše zmíněné objekty se do LMS API posílají shodným způsobem, takže by se i filtrace mohla v metadatech zapisovat do objektu *params*, ale v rámci lepší orientace v metadatech integrace jsou tyto dvě položky od sebe odděleny a integrační platforma je před odesláním spojí.

3.5 DataMixer

Jedná se o poslední typ kontejneru, který slouží pro filtraci a transformaci přijatých dat předtím, než budou předána dále. Vstup i výstup DataMixeru je možné pro lepší představu přirovnat k tabulkám v databázi, kdy každá tabulka na výstupu musí mít svůj unikátní identifikátor (*identifier*), aby se s ní mohlo pracovat i dále. Data, se kterými by se již nikde nepracovalo a nepotřebovala by identifikátor jsou zbytečná. Identifikátor vstupních dat se nastavuje do parametru *dataSource*, kde ovšem nemusí být pouze jeden identifikátor. Pokud vstupní data obsahují více identifikátorů, provede se nad všemi tabulkami s těmito identifikátory prosté sjednocení do jedné tabulky ještě předtím, než dojde v DataMixeru k jakékoliv další operaci. V takovém případě budou muset všechny nastavené identifikátory obsahovat data se stejným počtem a názvem sloupců.

Na vstupu může být i několik tabulek, které se nebudou pouze sjednocovat, ale budou se nad nimi provádět operace *join*, filtrovat vstupní data pomocí podmínek v objektu *where*, sjednocovat a řadit data pomocí *groupBy* a *orderBy* a vybírat data do výstupu pomocí pole objektů *parse*. Dále je zde možnost data transponovat ze sloupců na řádky. Zápis obecného DataMixeru je na ukázce kódu 3.19 a jednotlivé objekty budou popsány v následujících sekcích. Operace budou muset mít určené pořadí, v jakém se budou provádět, a to bude následovné:

1. join
2. where
3. groupBy
4. orderBy
5. transpose
6. parse
7. wrapperRemove
8. wrapperAdd

```
{
  "identifier": "dm_users",
  "dataSource" : "dd_users",
  "settings": {
    "join": [ "Pole s nastavením propojení více vstupních dat." ],
    "where": "Objekt s nastavením filtrace",
    "groupBy": {
```

```
      "columns": "username",
      "groupArray": ["osc", "lastname"]
    },
    "orderBy": {
      "username": "DESC",
      "osc": "ASC"
    },
    "transpose": [ "Transpozice sloupců na řádky." ],
    "parse": [ "Zpracování jednotlivých položek v datech." ],
    "wrapperRemove": [],
    "wrapperAdd": []
  }
}
```

Výpis kódu 3.19 Zápis DataMixer kontejneru.

3.5.1 Join

Vstupní tabulky lze mezi sebou spojovat pomocí několika jednoduchých podmínek. Počet spojení není omezen, lze tedy na jednu vstupní tabulku spojit libovolné množství dalších tabulek. Všechny parametry, které lze využít včetně způsobu zápisu jsou ve výpisu kódu 3.20.

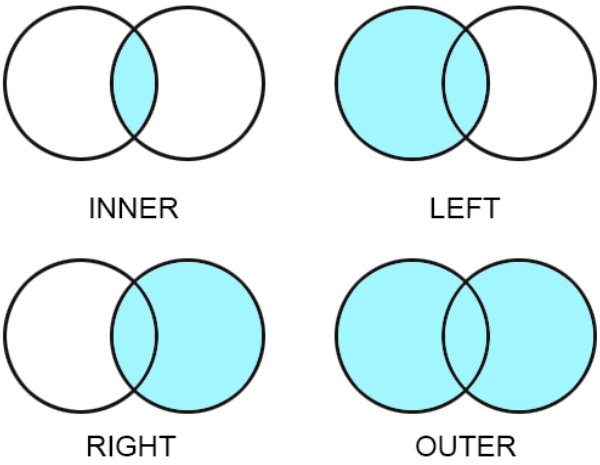
Parametry *leftTable* a *rightTable* budou určovat zdrojové tabulky, mezi kterými dojde ke spojení. Pomocí parametrů *prefixLeftTable* a *prefixRightTable* lze určit prefixy, které se doplní do vstupních tabulek před názvy všech sloupců. Prefixy lze využít pro zpřehlednění práce s velkými tabulkami, které vzniknou z několika menších. Dále existují případy, kdy zde musí být z důvodu, že dvě tabulky mohou mít shodné názvy některých sloupců a během spojení by pak nešlo určit, který sloupec se má zachovat. Jelikož prefixy nejsou povinný parametr a nemusí být tedy nastaveny, může i tak dojít ke kolizi dvou sloupců. V takovém případě se zpracují dle typu spojení a ve výsledné tabulce zůstanou data z poslední zpracovávané tabulky.

```
{
  "leftTable":   "dd_users",
  "rightTable":  "dd_companies",
  "prefixLeftTable": "users",
  "prefixRightTable": "companies",
  "joinType": "LEFT",
  "compareWithWildcards": false,
  "conditionsOperator": "AND",
  "conditions": [
    {
      "leftValue": "[__users.company__]",
      "operator": "==",
      "rightValue": "[__companies.id__]"
    }
  ],
  "groupBy": "username"
}
```

Výpis kódu 3.20 Objekt join v DataMixeru.

Typ spojení se bude určovat pomocí parametru *joinType* a lze zadat čtyři různé typy spojení. Jejich princip je zobrazen na obrázku 3.1.

- **INNER** – provede se spojení mezi levou a pravou tabulkou a na výstupu budou pouze záznamy, které mají nějakou vazbu se záznamem z opačné tabulky. Ostatní záznamy z levé



Obrázek 3.1 Zvýrazněné části zobrazují části množin, které budou ve výsledku spojení.

i pravé tabulky budou zahozeny. Položky, které se povedlo spojit se záznamy z druhé tabulky vícekrát, jsou ve výsledku se všemi svými vazbami.

- **LEFT** – dojde k pokusu napojit záznamy z *rightTable* na zdrojovou tabulku v parametru *leftTable*. Všechny záznamy z *rightTable*, ke kterým se nepodaří najít žádná vazba, budou zahozeny. Záznamy z *leftTable* bez vazby budou ve výsledku zachovány. I zde záznamy z *leftTable*, které budou mít více vazeb na záznamy z *rightTable* budou ve výsledku právě tolikrát, kolik vazeb budou mít.

- **RIGHT** – logikou stejné, jako typ *LEFT*, akorát zdrojová tabulka je ta, která je nastavena v parametru *rightTable*.

- **OUTER** – jedná se o rozšíření *INNER* joinu. To znamená, že ve výsledku jsou i záznamy z levé a pravé tabulky, které nemají žádnou vazbu.

Další parametr v nastavení spojení je *compareWithWildcards*. Tento parametr vznikl kvůli pomocné tabulce pro přidělování e-learningových kurzů, jejíž ukázka je v tabulce 3.1. Pokud v jedné tabulce bude seznam uživatelů, který obsahuje název společnosti a zároveň označení, zda se jedná o vedoucího zaměstnance či nikoliv, může dojít ke spojení s pomocnou tabulkou. V *join* podmínce budou právě dvě podmínky na to, jestli je shodná společnost a označení vedoucího. Pokud by nebylo možné použít parametr *compareWithWildcards* a na hodnotě v parametru `manager` by nezáleželo, tak by musela být informace o přidělení kurzu uvedena pro výčet všech možností v daném sloupci tak, jako je to na ukázkové tabulce u společnosti **Beta**. Pokud by nešlo o jednoduchý parametr obsahující nulu nebo jedničku, ale například o vlastnost obsahující jazyk uživatele, mohl by být výčet poměrně dlouhý. Proto je navrženo využít místo výčtu všech položek parametr *compareWithWildcards*. Pokud bude tento parametr nastaven na `true`, tak chování pro společnosti **Beta** a **Gama** bude totožné, ale zápis společnosti **Gama** je úspornější.

Předposlední operací, kterou je během spojení nutné udělat, je určení podmínek pro spojení. Podmínek může být několik a lze použít operátor `AND` nebo `OR`. Prozatím půjde pro jedno spojení s více podmínkami zvolit pouze jeden typ operátoru v parametru *conditionsOperator*. Ze zkušeností získaných z implementací integrací bez integrační platformy by nemělo být zapotřebí používat oba parametry najednou, protože ve všech zatím známých integracích do vzdělávacího systému jeden operátor stačí. Možnost využití obou typů operátorů zůstane zatím jako námět ke zlepšení.

company	manager	bpzam	bpvz	pp
Alfa	0	x		x
Alfa	1		x	
Beta	0			n
Beta	1			n
Gama	*			n

Tabulka 3.1 Příklad pomocného CSV souboru k přidělení kurzů dle společnosti a příznaku manager.

Samotné podmínky jsou pak jako pole objektů v parametru *conditions*, kde se zvolí hodnota z levé a pravé tabulky zvlášť a zapíše se pomocí placeholderu. Ten je zde použit proto, aby bylo možné zapsat do podmínky i statickou hodnotu, kdyby bylo potřeba vybrat všechny záznamy z jedné tabulky, kde je hodnota v nějakém sloupci rovna názvu konkrétní společnosti. K porovnání hodnot dojde pomocí zadaného operátoru, který může nabývat jedné z následujících hodnot:

- >, < – porovná hodnotu, jestli je ostře větší/menší než druhá hodnota.
- >=, <= – porovná hodnotu, jestli je větší/menší nebo rovna druhé hodnotě.
- ==, **!=** – ověří, že jsou obě hodnoty stejné/rozdílné. Nejedná se o striktní porovnání včetně datových typů, takže hodnota 1 číselného typu bude shodná s hodnotou "1" v řetězci.
- ===, **!==** – podobné jako předchozí porovnání, ale dochází ke striktnímu porovnání. To znamená, že hodnota čísla 1 číselného typu bude jen a pouze shodná s jinou číselnou hodnotou 1.
- **IN**, **NOT IN** – pokud je na jedné straně porovnání hodnota typu pole, dojde k porovnání hodnoty z opačné strany, zda je/není obsažena v tomto poli.
- **LIKE** – funguje obdobně jako v MySQL. V databázi *LIKE* funguje tak, že řetězec, oproti kterému dochází k porovnání hodnoty obsahuje masku, ve které jsou kromě statické části využity dva speciální znaky. První znak je **%** a místo něj může být v řetězci žádný, jeden nebo neomezený počet libovolných znaků. Druhý znak je **_** a místo něj je v řetězci právě jeden jakýkoliv znak. [18]

Posledním nastavitelným parametrem konfigurace spojení je *groupBy*. Tento parametr funguje shodně jako ten, který je přímo v objektu *settings* v DataMixeru a je tedy více popsán v sekci 3.5.3. V *join* objektu je možnost sjednocovat data proto, že to může být potřeba mezi jednotlivými spojeními, pokud jich je více.

3.5.2 Where

Parametr *where* v DataMixeru slouží k filtraci dat podle zadaných podmínek. Podmínky fungují na podobném principu jako podmínky v *join* bloku. Primárně jde o recyklaci shodného zdrojového kódu, který bude mít pouze jiný způsob získávání dat k porovnání. Zároveň i zde platí, že by v tuto chvíli nemělo být potřeba dělat složité podmínky pro filtraci. Pokud by se časem ukázala potřeba složitost podmínek rozšířit, půjde dočasně využít možnost vytvořit více DataMixerů a podmínky do nich rozdělit. Vstupní data se zadávají do podmínek pomocí parametrů *value* a *compareWith*. Lze využít opět jak placeholdery, tak statickou hodnotu, a dokonce i spojení placeholderu a statické hodnoty. Zápis bloku *where* je ukázán na kódu 3.21.

```
1 {
2   "conditionsOperator": "AND",
3   "conditions": [
```

```
    {
      "value": "[__users.company__]",
      "operator": "IN",
      "compareWith": ["Alfa", "Beta"]
    }
  ]
}
```

Výpis kódu 3.21 Vyhledávací objekt where.

3.5.3 GroupBy a OrderBy

Pomocí parametru *groupBy* lze sjednotit prvky se stejnou hodnotou ve vybraných sloupcích. Zápis lze provést třemi různými způsoby:

1. Jeden textový řetězec s názvem sloupce, podle kterého se data sjednotí.
2. Pole řetězců, které obsahují názvy sloupců, podle kterých se data sjednotí. Všechny záznamy, které se sjednotí do jednoho, musí obsahovat stejné hodnoty ve všech sloupcích.
3. Objekt s parametry `columns` a `groupArray`. Do `columns` lze zadat stejné hodnoty jako v bodech 1. a 2. výše. Parametr `groupArray` funguje na podobném principu jako funkce `GROUP_CONCAT` v MySQL. Tato funkce vytvoří ke každému sjednocenému záznamu sloupec, ve kterém jsou hodnoty z určitého sloupce každého jednoho záznamu před sjednocením oddělené čárkou a zřetězené do jediné hodnoty. [19] V integrační platformě nedojde ke zřetězení a hodnoty jsou ve sloupci vráceny jako pole.

Objekt *orderBy* nabídne možnost seřadit data před zpracováním na výstup, na kterém se již pořadí nemění. Řazení lze zapsat třemi možnými způsoby:

1. Pomocí jednoho textového řetězce s názvem sloupce, podle kterého se data seřadí vzestupně.
2. Jako pole řetězců, které obsahují názvy sloupců, podle kterých se data postupně vzestupně seřadí.
3. Pole objektů, které obsahuje jeden nebo více parametrů, jejichž název je zároveň název sloupce, podle kterého se mají data seřadit a hodnota je rovna `ASC` nebo `DESC`, což určuje směr řazení.

3.5.4 Transpose

Tento objekt slouží k transpozici vstupních dat ze sloupečků na řádky. Hlavní využití je opět v pomocných tabulkách, které slouží k přidělování vzdělávacích akcí, zařazení do organizační skupiny a podobně. Použití ale nalezne i v případě, kdy je část přijímaných dat se stejnou funkcionalitou popsána v jednotlivých sloupcích. Zápis transpozice je na ukázce kódu 3.22.

Položka *byColumns* určuje, které sloupce se mají transponovat do řádků. Pokud nastane opačný problém, kdy jsou známy sloupce, které je potřeba uchovat a podle všech ostatních transponovat, nastaví se parametr *byColumnsReverseLogic* na `true`. Dále se nastavují názvy dvou nových sloupců, kdy do jednoho (*transposedColumnName*) se vloží původní název transponovaného sloupce a do druhého (*compareWithColumnName*) se vloží původní hodnota, kterou měl záznam uloženou v transponovaném sloupci. Aby se daly vyfiltrovat jen záznamy, které ve sloupci mají uloženou požadovanou hodnotu, je možné použít parametr *condition* a v něm určit podmínky Ty se zapisují stejně jako u objektu *where*, kromě hodnoty *value*, protože ta je zde již určena. Tato filtrace pomůže ve velkých tabulkách, kde záznamy, se kterými se nebude dále pracovat, budou rovnou zahozeny.

```
{
  "byColumns": ["company", "manager"],
  "byColumnsReverseLogic": true,
  "transposedColumnName": "course",
  "compareWithColumnName": "flag",
  "condition": {
    "operator": "IN",
    "compareWith": ["x", "n", "r"]
  }
}
```

Výpis kódu 3.22 Příklad nastavení transpozice.

Pro lepší představu je zde uvedeno, jak by tabulka 3.1 měla vypadat po použití transpozice v kódu 3.22. Výsledek je v tabulce 3.2. Pokud by s takovou tabulkou byla spojena data uživatelů dle sloupců *company* a *manager*, tak by bylo možné pracovat se záznamy, které rovnou obsahují ID kurzu a příznak určující požadovanou operaci. Na takové záznamy lze pomocí dalšího DataMixeru provést *groupByArray* a získat tak jeden záznam, který v sobě bude obsahovat pole všech kurzů, které mají být přidělené.

company	manager	course	flag
Alfa	0	bpzam	x
Alfa	0	pp	x
Alfa	1	bpvz	x
Beta	0	pp	n
Beta	1	pp	n
Gama	*	pp	n

Tabulka 3.2 Výsledná tabulka po transpozici pomocné tabulky na přidělení kurzu.

3.5.5 Parse

Pokud by se stále DataMixer přirovnával k tabulce s daty, tak pomocí parametru *parse* lze nastavit jednotlivé sloupce, které v tabulce budou. To znamená, že parametr *parse* funguje jako projekce a jeho chování je podobné klíčovému slovo *SELECT* v databázovém jazyce. Pokud hodnota *parse* není uvedena nebo je prázdná, tak na výstupu z DataMixeru budou všechny sloupce, které byly na vstupu a případně byly přidány dalšími operacemi jako třeba spojením s jinou tabulkou. Každý objekt v poli označuje právě jeden sloupec, který lze nastavit různými parametry uvedenými v ukázce kódu 3.23. Každý objekt v *parse* může rekurzivně obsahovat další pole s *parse* objekty. To bude sloužit k možnosti zanoření dat v XML a JSON formátech.

Vstupní data pro každý sloupec půjde nastavit různě v několika parametrech:

- **incomingIndex** – zde se bude zadávat název sloupce ze zdrojových tabulek.
- **defaultData** – statická vstupní data. To se bude hodit v případě, kdy sloupec bude obsahovat pro všechny záznamy shodná statická data.
- **functionsOnData** – tento parametr přijímá jak statická data, tak data ze zdrojových tabulek. Podrobnější popis tohoto parametru bude uveden později.

```
{
  "*": null,
  "skipIfEmpty": true,
```

```
4   "incomingIndex": "osobni-cislo",
5   "outgoingIndex": "username",
6   "transposeArrayValues": false,
7   "defaultData": null,
8   "isAttribute": false,
9   "exchangeValue": [
10    {
11      "operator": "==",
12      "compareWith": 12345,
13      "exchangeTo": "0012345"
14    }
15  ],
16  "functionsOnData": [
17    {
18      "function": "substr",
19      "parameters": [1,5]
20    }
21  ],
22  "parse": []
23 }
```

Výpis kódu 3.23 Zápis jednoho parse objektu.

Pokud má sloupec identifikátor *outgoingIndex*, pak se zapisuje do odchozích dat a jako název nového sloupce je právě využita hodnota v *outgoingIndex*. Tento parametr není povinný, protože se může jednat jen o pomocný sloupec, díky kterému dojde k filtraci dat pomocí parametru *skipIfEmpty*. Jak lze usoudit z názvu, tak tento parametr, přeskočí celý záznam, pokud by hodnota v tomto sloupci měla být prázdná.

Jestliže by byl nastavený přepínač * na `true`, pak v daném objektu by již nemělo být vyplněno nic jiného, a to proto, že všechny sloupečky, které jsou na vstupu se zkopírují i na výstup. Rozdíl oproti tomu, kdy není vůbec *parse* vyplněný je ten, že zde lze za všechny zkopírované sloupce přidávat ještě další pomocí dalších objektů v poli *parse*.

Položka *isAttribute* bude sloužit pouze pro data generovaná do XML souboru, kde by výstup těchto dat měl být zapsaný v rodičovském tagu a nikoliv jako samostatný tag.

Pokud ve vstupních datech bude pole, pro které by bylo potřeba vygenerovat pro každou hodnotu v poli vlastní záznam ve výstupních datech, pak lze využít položku *transposeArrayValues*, která transponuje pole na vstupu do řádků. Funkčnost je podobná jako v případě transponování sloupců vstupních dat do řádků, akorát místo sloupců jsou zvoleny právě položky v poli, které mohou být pro každý záznam rozdílné.

Data sloupce pak půjde transformovat pomocí následujících dvou parametrů:

- **exchangeValue** – obsahuje jednoduchou podmínku, která při splnění nahradí aktuální hodnotu sloupce za hodnotu definovanou v parametru *exchangeTo*. Ukázka této podmínky je na pseudokódu 3.24. Podmínky fungují pomocí *operator* a *compareWith* parametrů stejně, jako je tomu u *where* objektu popsaného v sekci 3.5.2.

- **functionsOnData** – tento parametr bude sloužit k možnosti zavolání funkcí definovaných přímo v jazyce PHP a případně funkcí, které si bude moci administrátor sám v PHP vytvořit a uložit do metadat, což je popsané v sekci 3.6. Do položky *parameters* bude možné zadat hodnoty ze vstupních dat pomocí placeholderů, statická data nebo jejich kombinaci. Může nastat situace, kdy bude konkrétní *parse* obsahovat již nějaká data, která jsou vložena pomocí *incomingIndex*. V tomto případě záleží na tom, jestli je u funkce definována položka *parameters*. Pokud ano, lze takto nastavená data předat do funkce pomocí placeholderu `[__this__]`. Pokud položka *parameters* definována není, tak se aktuální data sloupce pošlou do funkce na pozici prvního parametru. To se může hodit například pro volání funkce jako `strtoupper`,

která nastaví všechny znaky v řetězci na velká písmena a jako jediný parametr přijímá právě řetězec určený pro úpravu.

```
IF input <operator> <compareWith> THEN
        OUTPUT <exchangeTo>
END IF
```

Výpis kódu 3.24 Pseudokód exchangeValue podmínky.

Pořadí provádění operací je stejně jako u jiných parametrů v metadatech dané. Nejdříve se provede *exchangeValue* a až následně *functionsOnData*, což může vyvolat problém ve chvíli, kdy je potřeba data první zpracovat nějakou funkcí a až podle výsledku této funkce provést operaci výměny dat. Proto zde bude možnost znovupoužitelnosti již vytvořených sloupců v jednom DataMixer objektu pomocí prefixu `this.nazev_sloupce`. Takto půjde sloupec s daty využít v několika po sobě následujících *parse* objektech a jedinou nevýhodou bude, že u každého musí být zadaný parametr *outgoingIndex*. To by mohlo zapříčinit předání dočasných sloupců do výstupních dat. Tomu půjde předcházet dvěma způsoby. Buď se bude používat stejný název sloupce ve všech techto *parse* objektech anebo se vytvoří jeden DataMixer objekt, ve kterém se vyfiltrují pouze sloupce určené k uložení v následujícím DataDistributoru.

3.6 Vlastní funkce

Pokud administrátor bude potřebovat provést nad daty operaci, kterou nelze vykonat pouze pomocí *exchangeValue* a funkcí definovaných v PHP, bude mu umožněno napsat si vlastní funkci v jazyce PHP a vložit ji do speciálního bloku v metadatech pojmenovaného *ownFunctions*. Funkce se bude do metadat vkládat zakódovaná v *BASE64* kódování v parametru *data*, jak je zobrazeno v ukázce kódu 3.25. V tomto kódu je funkce zpracovávající příchozí osobní číslo uživatele, které může být doplněno o prefix s identifikátorem společnosti a postfix s identifikátorem smlouvy. Pokud poskytnutý údaj obsahuje oddělovače pro prefix a postfix, tak je tato funkce odstraní a vrátí pouze čisté osobní číslo bez okolních dat. Pokud žádný z oddělovačů údaj neobsahuje, tak funkce vrátí přijaté osobní číslo bez jakékoliv změny.

```
{
      "name": "getOSC",
      "parameters": ["id"],
      "data": "JHRtcCA9IGV4cGxvZGUoIl8iLCAkaWQpOwppZihjb3VudCgkdG1wKSA+IDEpIHsKICA
               gICR0bXAgPSBleHBsb2RlKCIuIiwgJHRtcFsxXSk7CiAgICByZXR1cm4gJHRtcFswXT
               sKfSBlbHNlIHsKICAgIHJldHVybiAkaWQ7Cn0="
}
```

Výpis kódu 3.25 Zápis vlastní funkce pro ověření osobního čísla.

Pomocí vlastních funkcí půjde vytvořit naprosto libovolnou transformaci. Nevýhodou by mohlo být, že tato možnost by mohla zkušenější administrátory se znalostí programování svádět k tomu, aby veškeré transformace napsali ve vlastních funkcích a nevyužívali funkce integrační platformy k tomu určené. Zde by mělo platit nepsané pravidlo, že použití *ownFunctions* bude doporučeno využít pouze v případě, kdy daná operace nepůjde zapsat v metadatech žádným jiným (byť zdlouhavějším) způsobem. Důvody jsou tři.

1. V první řadě jde o přehlednost metadat integrace, kdy v případě nadměrného využívání *ownFunctions* by mnoho operací bylo zapsáno v kódované podobě a na první pohled by nebylo vůbec jasné, co se v metadatech děje. Vše by muselo být popsáno v komentářích.

2. Dalším důvodem je správa takových integrací. V případě sebemenší změny by bylo potřeba dekódovat řetězec s funkcí, tu pozměnit a znovu zakódovat. V případě složitých PHP funkcí

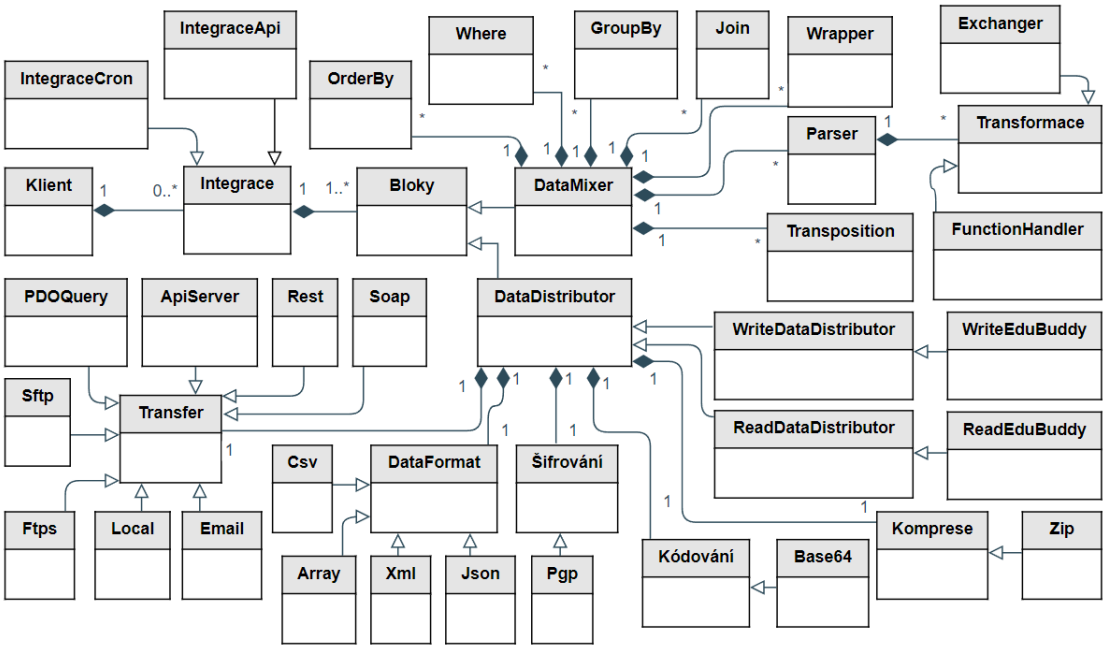
by pak docházelo ještě k tomu, že by musel být kód nejprve důkladně prostudován, aby byl správně pochopen a až pak upraven.

3. Posledním důvodem je optimalizace zpracování. Zatímco integrační platforma si bude moci některé operace předpřipravit a se zdroji pracovat tak, aby zbytečně nezatěžovala server, tak do vlastních funkcí nebude nijak zasahovat

Kapitola 4
Návrh implementace

V této kapitole je popsán návrh struktury aplikace pomocí návrhového modelu tříd, který je pro lepší přehlednost rozdělen do několika menších logických celků. Společně s tímto modelem je pak u každé důležité funkce integrační platformy popis způsobu její implementace a případně i seznam využitých externích knihoven.

V rámci vytvoření návrhu integrační platformy vznikl konceptuální model tříd zobrazený na obrázku 4.1. Z tohoto doménového modelu byl následně vytvořen složitější implementační model tříd, který obsahuje 61 tříd a z důvodu jeho velikosti je zahrnut mezi přílohami na přiloženém médiu s označením class-diagram.png. Tento návrh již slouží jako hlavní podklad k implementaci integrační platformy. Jednotlivé bloky s třídami, které vždy tvoří logický celek, jsou popsány v následujících sekcích společně s výřezy z diagramu tříd.



Obrázek 4.1 Doménový model tříd.

4.1 Samostatné třídy

Jako první jsou uvedeny třídy, které se neinstancují, protože obsahují pouze statické metody a neuchovávají si žádné vlastnosti. Primárně jde o třídy, které slouží jako pomocné prvky pro často se opakující akce v integrační platformě, ke kterým dochází napříč celým systémem. Lze je tedy využít kdekoliv a není potřeba vytvářet nové instance.

4.1.1 GlobalService

Tato třída bude obsahovat různé metody, které bude potřeba využít na více místech integrační platformy, kdy tyto metody mezi sebou nemají žádnou spojitost. Metody by mohly mít vždy každá vlastní třídu, ale jelikož se nepřepokládá, že jich bude velké množství, bude využita globální třída s několika krátkými metodami než několik různých tříd s jednou metodou. Většina metod uvedených v této třídě bude doplněna až v rámci implementace, ale jsou zde již i metody, o kterých je známo, že bude potřeba jejich implementace.

- jsonEncode a jsonDecode – k využití těchto vlastních funkcí je více důvodů, zde bude uveden pouze hlavní. Výchozí funkce v PHP, určené pro zakódování a dekódování formátu JSON, vracejí při neúspěchu hodnotu false. Zároveň je ale možné nastavit příznak, že místo návratové hodnoty mají vyhodit výjimku. Takové chování pro integrační platformu bude vhodnější, protože bude moci odchytávat tento typ výjimek a v případě jejího odchycení ukončit integraci, protože se bude jednat o zásadní chybu během integrace. Pokud by tyto funkce nebyly implementované, musel by vývojář aplikace na všech místech nastavit potřebný příznak, což by v případě zapomenutí mohlo vést k neodhalení chyby a k neočekávanému chování integrační platformy.
- objectToArray – u SOAP třídy a XML formátu bude potřeba využít rekurzivního převodu hodnoty typu *object* na *array*.
- compareValue – bude sloužit k porovnání dvou hodnot pomocí zadaného operátoru. Centralizace takového porovnání je primárně proto, aby se nestalo, že by se v metadatech dvě stejné hodnoty na dvou různých místech porovnávaly jiným způsobem.

4.1.2 LogService

Pomocí třídy *LogService* bude docházet k vytváření záznamů logů. Půjde jak o logy integrace, které se budou posílat do centrální administrace, tak textových logů. Ty budou sloužit primárně k debuggování integrace, kdy při zapnutí logování admin v souboru na serveru uvidí dosavadní průběh a data integrace, aby snadno odladil případné chyby v integraci.

4.1.3 PlaceholdersService

Využití placeholderů, které byly popsány v sekci 3.2 bude obstarávat tato třída. Bude obsahovat metody pro zjištění, zda je v daném řetězci placeholder a případně rovnou jeho záměnu za požadovaná data. Pomocí toho půjde ze všech míst systému, kde je umožněna práce s placeholdery pouze zavolat metodu ze třídy *PlaceholdersService* a vše ostatní vyřeší třída sama.

4.1.4 CronService

Jelikož CronJoby budou mít uložený údaj o tom, kdy se mají spustit pomocí zápisu ve formátu UNIX cron format, bude tato třída sloužit k tomu, aby rozhodla, zda se má daný CronJob spustit v době běhu skriptu nebo ne. UNIX cron format se zapisuje pomocí pěti hodnot oddělených

mezerou, které určují minuty, hodiny, dny, měsíce a dny v týdnu. Každou z částí lze zapsat pomocí údaje s jednou konkrétní hodnotou, výčtu hodnot, rozmezí hodnot nebo hvězdičkou, která označuje jakoukoliv příchozí hodnotu. Rozmezí lze dále určit pomocí kroků, kdy například bude vykonán každý druhý krok.[20]

Příklad je uveden na výpisu kódu 4.1, který označuje zápis integrace, která proběhne v půl třetí odpoledne každý druhý den v měsíci, pokud jde o všední den.

```
14 30 */2 * 1-5
```

Výpis kódu 4.1 Ukázka zápisu UNIX cron formátu.

4.1.5 OwnFunctionsStorage

Při využívání vlastních funkcí v *parse* bloku DataMixeru budou nastávat situace, kdy některé funkce se budou používat častěji, a proto budou uloženy v této třídě. Následně je pak nebude nutné definovat u každé integrace, která je bude potřebovat využít. Půjde primárně o funkce sloužící jako aliasy k funkcím definovaných v PHP jazyce. Jako příklad, který je dopředu známý, lze uvést funkci *dateFormat*, která bude přijímat dva parametry. První bude hodnota obsahující datum v libovolném formátu a druhá požadovaný formát, do kterého se má datum převést. Ten bude shodný s formátem, který přijímá funkce *date()* v PHP. Takováto operace se zapíše v PHP tak, jak je uvedeno v kódu 4.2. Jde tedy o zavolání dvou funkcí, kdy první převede přijaté datum na timestamp a druhá převede timestamp na požadovaný formát. Pokud by se toto používalo v integrační platformě, bylo by potřeba pro každý převod data do jiného formátu využít dva *functionsOnData* bloky. A jelikož se s datumy bude pracovat v mnoha integracích, bude lepší si tuto část kódu uložit pod alias *dateFormat()* ve třídě *OwnFunctionsStorage*.

```
$timestamp = strtotime("2020-04-03");
return date("d.m.Y", $timestamp);
```

Výpis kódu 4.2 Převod data z jednoho formátu do jiného v PHP.

Tato třída se bude dynamicky rozšiřovat dle potřeb administrátorů při psaní integrací. Jakmile se zjistí, že se opakuje nějaký požadavek napříč více integracemi, bude možné mu vytvořit alias a administrátorům usnadnit zápis nových integrací.

4.2 Třídy pro spuštění integrace

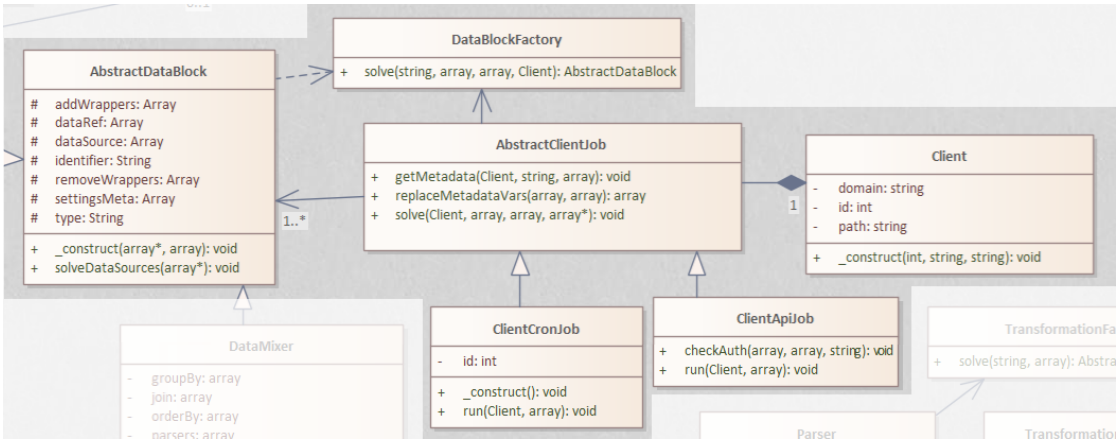
Následující třídy patří mezi nejdůležitější, protože jsou potřeba ke spuštění integrace. Pracují s klienty, CronJoby a vytváří objekty pro jednotlivé bloky v metadatech. Část diagramu tříd, ve kterém jsou zakresleny je k vidění na obrázku 4.2.

4.2.1 Client

Tato třída obsahuje všechny důležité informace o klientovi, kterého se integrace týká. Tím je jeho doména, na které je spuštěna instance vzdělávacího systému a která je součástí všech URL adres zdrojů, které vzdělávací systém nabízí. Druhý důležitý údaj je cesta na serveru ke klientově instanci, která je potřeba v případě využití DataDistributorů s typem přenosu *local* pracujícím se soubory v adresářové struktuře klientské instance.

4.2.2 AbstractClientJob

Jedná se o abstraktní třídu, která se stará o běh integrace. Obsahuje metody, které jsou shodné pro oba její potomky, jako je získání metadat integrace ze vzdělávacího systému, nahrazení



Obrázek 4.2 Část diagramu tříd s třídami pro spuštění integrace.

globálních placeholderů v metadatech a zpracování integrace, což znamená zpracování všech kontejnerů v metadatech a vytvoření potřebných objektů pro jejich zpracování.

Z tohoto abstraktu dále vycházejí třídy *ClientCronJob* a *ClientApiJob*, které se liší ve způsobu spuštění integrace a získání dat o klientovi. Integrace spuštěna automaticky pomocí CRONu načítá informace o CronJobu z databáze centrální administrace a s touto informací získává rovnou data klienta. Taková integrace vytváří instanci třídy *ClientCronJob*. Pokud jde o integraci, která je spuštěna pomocí volání na API server ve vzdělávacím systému, je celá situace mnohem komplikovanější. Taková integrace musí autorizovat požadavek a pomocí přijatých dat získat informace o klientovi a integraci z databáze centrální administrace. Způsob implementace API serveru na straně vzdělávacího systému je detailně popsán v sekci 4.4.3.1 a pro tento typ integrace bude potřeba vytvořit instanci třídy *ClientApiJob*, která se o všechny uvedené náležitosti bude muset postarat.

4.2.3 DataBlockFactory

Jedná se o třídu, která funguje dle návrhového vzoru továrna. Továrna se využívá na vytvoření objektů z dané skupiny tříd, aniž by bylo potřeba znát název a parametry třídy nutné k vytvoření potřebného objektu. V tomto návrhovém vzoru jde o přenesení zodpovědnosti k vytváření tříd pouze na jednu danou třídu (továrnu) [21] a v integrační platformě bude tento návrhový vzor využit opakovaně.

```
for each: container in containers
        for each: block in container[blocks]
                var blockObject = DataBlockFactory::solve(
                        container[type],
                        integrationData,
                        metadataBlock,
                        Client)
                blockObject->solve();
```

Výpis kódu 4.3 Pseudokód vytvoření objektů bloků a jejich zpracování pomocí továrny.

Tato konkrétní továrna vytvoří instanci správné třídy pro každý blok dle typu kontejneru, ve kterém je blok umístěn. To znamená, že pro kontejner typu *readDataDistributor* vytvoří instanci třídy *ReadDataDistributor*. Díky továrně bude možné pomocí dvou zanořených cyklů vytvořit a spustit všechny operace nad všemi bloky v metadatech. Příklad takového spuštění je zobrazen na ukázce kódu 4.3

Z výše uvedené ukázky je patrné, že se do každého nového bloku předává nejen jeho typ, část metadat, které se ho týkají a objekt klienta, ale i položka *integrationData*. Jde o proměnnou, která je typu array a předává se do všech bloků jako reference, což umožňuje všem blokům toto pole nejen číst, ale i zapisovat. Uchovávají se v ní všechna data integrace, kdy jako index v poli je použit identifikátor u daného bloku, který data vygeneroval. To znamená, že pokud blok typu readDataDistributor bude mít v metadatech v položce *identifier* hodnotu **dd_users**, budou data načtená tímto DataDistributorem v poli *integrationData* pod indexem **dd_users**. Pokud bude dále existovat nějaký další blok, například DataMixer, který jako položku *dataSource* bude mít uvedeno **dd_users**, bude přesně vědět, kde má svá zdrojová data hledat.

4.2.4 AbstractDataBlock

Všechny třídy, které budou sloužit k obsluze bloků v metadatech budou dědit z abstraktní třídy *AbstractDataBlock*. Tato abstraktní třída bude deklarovat všechny potřebné vlastnosti, které jsou shodné pro bloky, jako je identifikátor bloku, identifikátor vstupních dat, reference na data integrace, nastavení wrapperů a část metadat, která se týkají daného bloku. Ke všem těmto vlastnostem bude obsahovat navíc všechny potřebné gettery a settery a navíc metodu *solveDataSources*, která bude sloužit pro sjednocení všech zdrojů daného bloku, pokud jich bude více než jeden.

4.3 DataDistributor a EduBuddy

Všechny třídy pro obsluhu bloků DataDistributor a EduBuddy budou dědit ze společné třídy *AbstractDataDistributor*, která obsahuje potřebné vlastnosti bloků a k nim samozřejmě i gettery a settery. O veškerou logiku se pak starají potomci sami, protože navzájem pracují s daty opačným způsobem. Část diagramu týkající se těchto tříd je na obrázku 4.3.

4.3.1 ReadDataDistributor a WriteDataDistributor

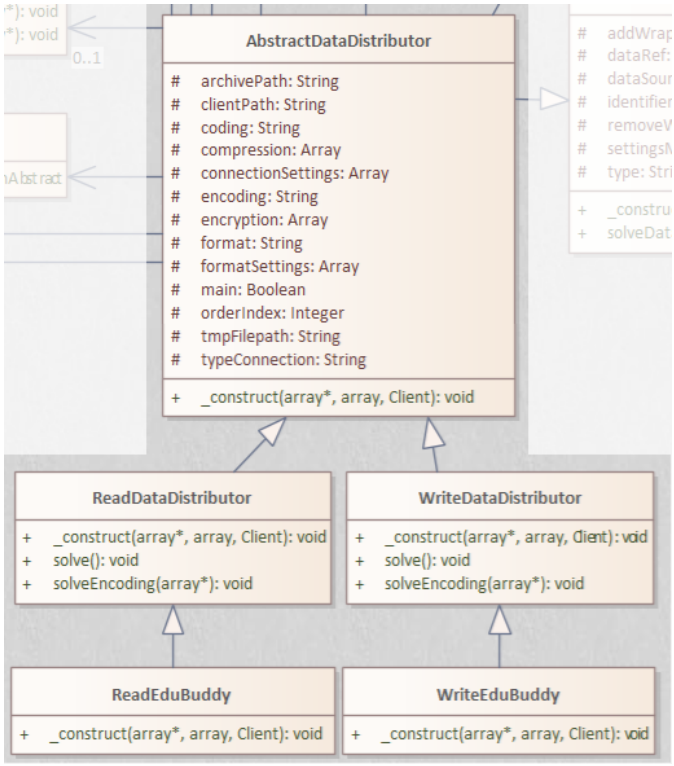
Třídy obsluhující DataDistributor bloky budou primárně delegovat práci pomocí dalších tříd typu továrna, které vytvoří odpovídající objekty například pro stažení dat pomocí zadaného typu přenosu. Seznam všech továren, které budou DataDistributor bloky volat je uveden v sekci 4.3.3. Rozdíl mezi ReadDataDistributor a WriteDataDistributor je v pořadí volání zmíněných továren, kdy DataDistributor sloužíci ke čtení dat první provede přenos dat a až pak následnou práci s daty jako dešifrování, dekódování, rozbalení archivu a získání dat dle typu formátu dat. Naopak DataDistributor pro zápis dat bude první ukládat data do zvoleného formátu, pak je zabalí, zakóduje a zašifruje a až potom přenese.

4.3.2 ReadEduBuddy a WriteEduBuddy

Bloky typu DataDistributor a EduBuddy budou zpracovány shodným způsobem, protože jak již bylo uvedeno, tak bloky typu *readEduBuddy* a *writeEduBuddy* jsou pouhými aliasy pro odpovídající DataDistributor bloky. Proto budou třídy obsluhující EduBuddy bloky dědit ze tříd *ReadDataDistributor* a *WriteDataDistributor*, kdy rozdíl bude v konstruktoru těchto tříd. Tam budou nadefinované hodnoty, jako typ způsobu přenosu (REST API klient), URL ke zdroji a další, které jsou pro EduBuddy bloky známé.

4.3.3 Továrny využité DataDistributory

Jedná se o továrny, které využívají třídy obsluhující DataDistributor bloky v metadatech. Dle nastavení v metadatech vytvoří odpovídající objekty a DataDistributor objekty na ně již volají



Obrázek 4.3 Část diagramu tříd s třídami pro zpracování bloků DataDistributor a EduBuddy.

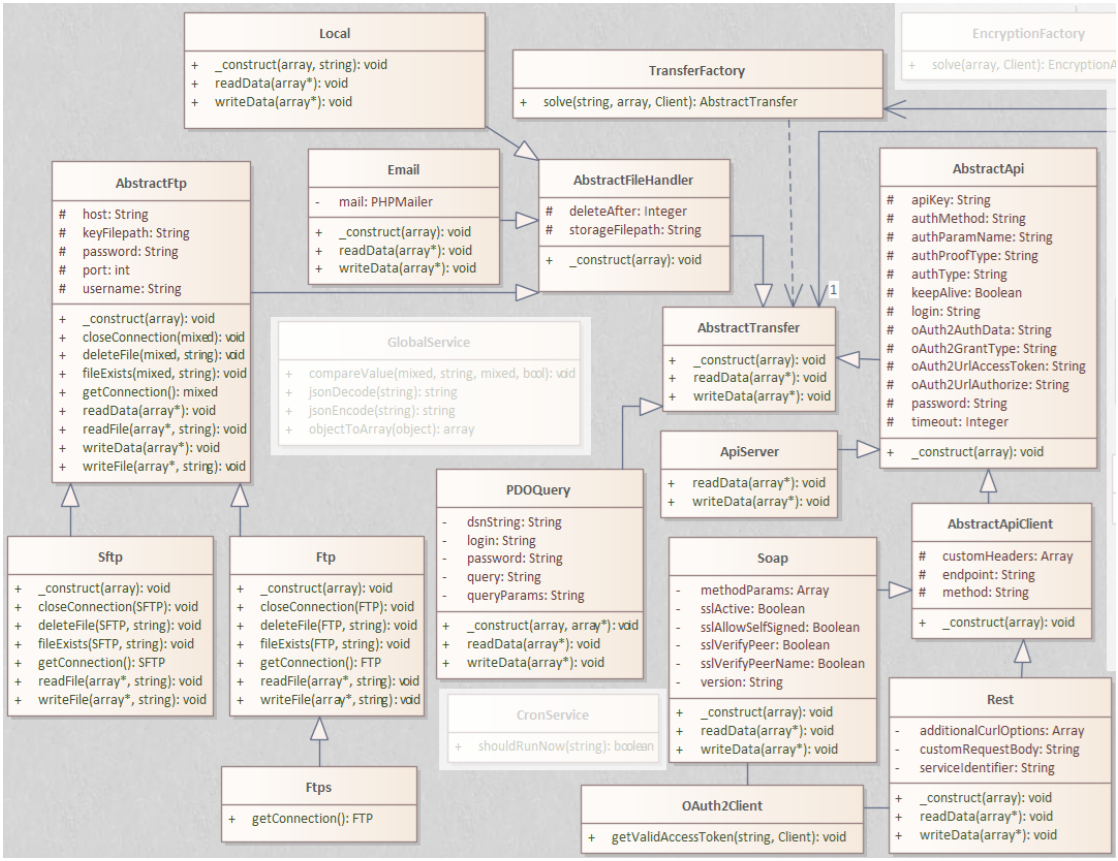
pouze metody, které jsou pro každou skupinu tříd stejné. Jedná se o následující továrny:

- TransferFactory – stará se o vytvoření objektů pro přenos dat. Vytváří instance tříd Local, Email, Ftp, Ftps, Sftp, PDOQuery, Rest, Soap a ApiServer.
- EncryptionFactory – zajišťuje vytvoření objektů pro šifrování a dešifrování dat. Zatím se bude jednat pouze o třídu Pgp, ale díky továrně nebude těžké přidat další třídu s jiným způsobem šifrování.
- CodingFactory – pomocí této továrny bude vytvořen pouze objekt ze třídy Base64 zajišťující dekódování a zakódování dat. Stejně jako u EncryptionFactory, tak i zde bude možné kdykoliv pomocí další třídy a rozšíření továrny přidat další způsob kódování dat.
- DataFormatFactory – bude vytvářet správný objekt pro obsluhu dat dle jejich formátu. To znamená, že bude vytvářet instance tříd Csv, Json, Xml a PhpArray.
- CompressionFactory – k práci s archivem bude sloužit tato továrna, která podobně jako některé předchozí továrny bude zatím pracovat pouze s jednou třídou a bude možné ji snadno v budoucnu rozšířit o další formáty kompresí.

4.4 Třídy pro přenos dat

Následující třídy se starají o přenos dat, kdy každá jedna třída obsluhuje právě jeden způsob přenosu, který integrační platforma nabízí. Všechny tyto třídy dědí z abstraktní třídy *AbstractTransfer*. V této třídě budou definované pouze tři abstraktní metody, které musejí potomci třídy definovat. Bude se jednat o konstruktor a metody *readData* a *writeData*. Tyto metody budou

jediné veřejné metody v potomcích. Všechny třídy, které mají vliv na přenos dat jsou na části diagramu tříd na obrázku 4.4.



Obrázek 4.4 Část diagramu tříd s třídami pro přenos dat.

4.4.1 PDOQuery

Tato třída se bude starat o získávání dat ze vzdálených SQL serverů. To znamená, že v metodě *writeData*, kterou musí podle abstraktu definovat, bude pouze chybová hláška, že tento směr přenosu nelze využít. Čtení dat bude probíhat pomocí třídy PDO, která je součástí PHP Data Object rozšíření instalovaného přímo na server.

```
$conn = new PDO($dsnString, $username, $password);
$conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
$stmt = $conn->prepare($this->getQuery());
$stmt->execute($this->getQueryParams());
$resultData = $stmt->fetchAll(PDO::FETCH_ASSOC);
```

Výpis kódu 4.4 PHP kód na získání dat ze vzdálené databáze pomocí třídy PDO.

PDO třída vyžaduje k připojení *dsnstring*, který obsahuje informace o vzdálené databázi, jako jsou IP adresa, port, typ databáze a jméno databáze. Dále je potřeba třídě poskytnout autorizační údaje, pokud jsou nutné. Po vytvoření instance třídy PDO je nutné provést přípravu dotazu, kdy je nejprve třídě předán SQL dotaz pomocí metody *prepare* a následně volitelné parametry dotazu, které se předají do metody *execute*. Touto metodou se provede volání dotazu

na databázový server. Následně se zavolá metoda *fetchAll*, která vrátí všechny nalezené řádky odpovídající dotazu. Ukázka kódu v PHP je na obrázku 4.4. Jak je z kódu patrné, je nutné nastavit příznak pro vyhazování výjimek místo vracení běžných chyb, aby tyto výjimky mohla snáze integrační platforma odchytit.

Požadavkem na integrační platformu je možnost připojovat se do MSSQL databáze. Jelikož bude administrátorům umožněno měnit celý DSN řetězec v metadatech integrace, tak se bude možné připojit na libovolnou SQL databázi, kterou podporuje třída *PDO*.

4.4.2 AbstractFileHandler

Dalších několik tříd slouží pro přenos dat mezi integrační platformou a cílovou destinací pomocí běžných souborů, a proto budou tyto třídy sdílet společného abstraktního předka. Třída *AbstractFileHandler* bude obsahovat parametry s údaji, po kolika dnech se má zpracovávaný soubor smazat a celou cestu k souboru včetně názvu. To se v každém z potomků využije trochu jinak.

4.4.2.1 Local

Třída obsluhující načítání a ukládání souborů přímo na stejný server, na kterém běží integrační platforma. Pomocí konstruktoru přijme do proměnné *clientPath* celou cestu na serveru, ve které má se souborem pracovat. Cestu doplní do konstruktoru továrna *TransferFactory*, která obsahuje objekt *Client* s touto informací. Jedná se o implementačně nejjednodušší způsob, protože se zde využívá pouze práce s lokálním souborem.

4.4.2.2 Email

Podobně jako PDO třída, tak i *Email* třída bude mít sice definované obě metody *readData* a *writeData*, ale první uvedená bude pouze vyhazovat výjimku. Ta bude obsahovat informaci, že tento způsob přenosu nelze využít ve zvoleném směru, protože skrze e-mail půjde data pouze odesílat, jak již bylo dříve uvedeno.

Na odesílání e-mailů bude použita knihovna *PHPMailer*, která je dostupná skrze Composer. Pomocí této knihovny se vytvoří objekt *PHPMailer*, kterému se pomocí několika jednoduchých metod doplní potřebné informace včetně možného přiložení přílohy. Příklad využití v PHP je v ukázce kódu 4.5.

```
$mailer = new PHPMailer(true);
$mailer->setFrom("our@email.com");
$mailer->addRecipient("destination@email.com");
$mailer->CharSet = 'utf-8';
$mailer->Subject = "String subject;
$mailer->Body = "<p>HTML body</p>;
$mailer->isHTML();
$mailer->addStringAttachment("Data;to;send", "filename.csv");
$mailer->send();
```

Výpis kódu 4.5 PHP kód s ukázkou práce s PHPMailer třídou.

4.4.2.3 AbstractFtp

Jak protokol SFTP, tak FTPS budou obsahovat společného předka, který určí některé abstraktní metody potřebné definovat v potomcích. Funkce *writeData* a *readData* budou definované pouze v tomto abstraktu a budou využívat právě ty metody, které musí potomci definovat. Postup při přenosu dat bude tedy pro všechny potomky stejný, jen bude každý z nich používat svůj vlastní způsob. Příklad, jak by mělo jít zapsat metodu *readData* je k vidění na PHP kódu 4.6.

```
$connection = $this->getConnection();
$this->readFile($dataRef, $connection, $this->getStorageFilePath());
$this->closeConnection($connection);
```

Výpis kódu 4.6 Příklad zápisu metody readData v AbstractFtp třídě.

Potomci třídy *AbstractFtp* jsou následující:

- Ftp – třída pro připojení na FTP server. Pro komunikaci s FTP serverem budou využity funkce, které jsou již obsažené v PHP jazyce. Pro připojení se použije funkce *ftp_connect* a pro práci se souborem funkce *ftp_fget*. FTP protokol neměl být původně implementován, ale jelikož jeho implementace společně s FTPS vyžaduje minimum práce, bylo rozhodnuto, že i tento protokol se implementuje. Mohl by být využit v budoucnu například v rámci testování.
- Ftps – jde o potomka třídy *Ftp*. Veškeré funkce kromě jedné jsou shodné. Rozdíl je pouze ve funkci *getConnection*, kde se k připojení nepoužije funkce *ftp_connect*, ale funkce *ftp_ssl_connect*. Tím dojde k připojení přes protokol FTPS.
- Sftp – pro připojení přes SFTP protokol bude využita knihovna *SFTP* z balíku *phpseclib3*, kterou lze stáhnout pomocí Composeru. Knihovna poskytne veškerou práci protokolu, takže integrační platforma pouze vytvoří novou instanci třídy *libSFTP*, které v konstruktoru předá cestu k serveru a port. Následně zavolá metodu *login*, do které nastaví autorizační parametry, ať už uživatelské jméno a heslo nebo privátní klíč a passphrase a tím dojde k připojení na server. Soubor se pak získá pomocí metody *get*, které se v parametru pošle název souboru.

4.4.3 AbstractApi

Jde o abstraktní třídu ke všem typům připojení, které využívají API, a to bez ohledu na to, jestli mají fungovat jako API server nebo API klient. Stejný abstrakt slouží pouze k uchování většiny stejných parametrů, jako je například *apiKey*, *authType*, *authParamName*, *authMethod*. Dále pak parametry týkající se OAuth2.0 autorizace, i když ta bude momentálně dostupná pouze pro klientskou stranu API.

4.4.3.1 ApiServer

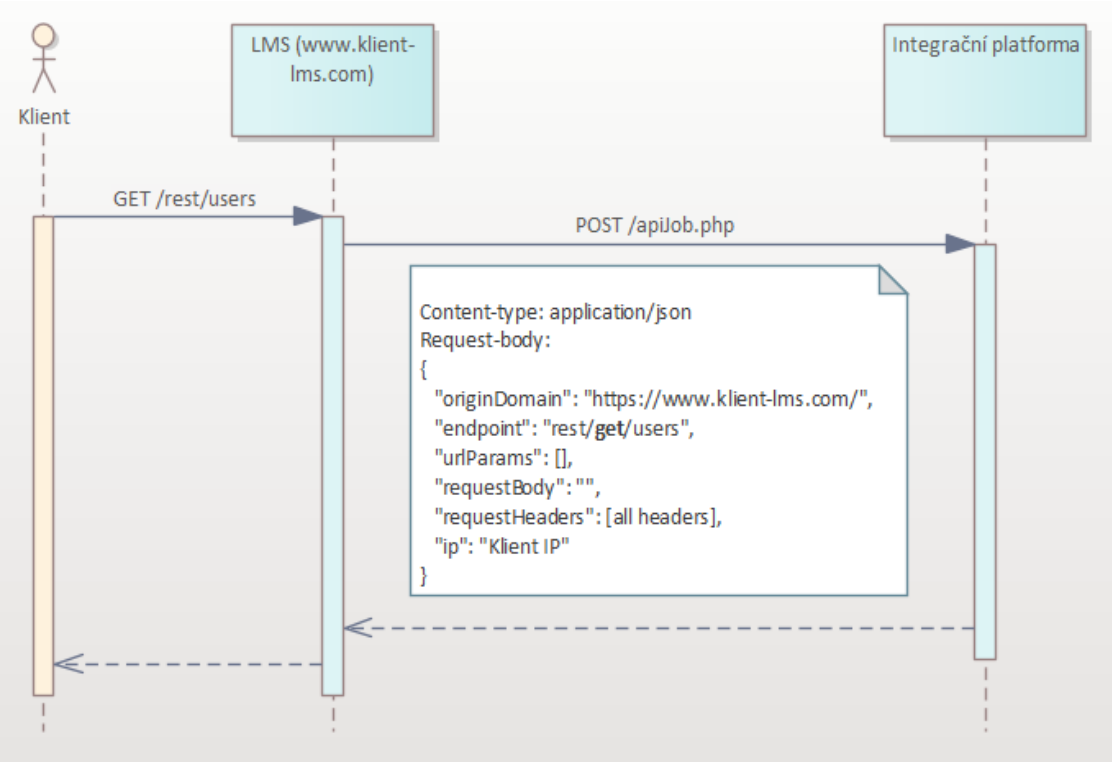
Typ připojení, kdy vzdělávací systém nabídne API server, bude na implementaci o něco složitější. Veškeré endpointy poskytnuté klientům využívajícím API servery, musí směřovat na doménu, kde je nainstalovaný vzdělávací systém klienta. Z tohoto důvodu bude potřeba ve vzdělávací aplikaci nastavit, aby při zavolání některého z těchto endpointů předala data a veškeré nutné informace integrační platformě na speciálně určeném endpointu integrační platformy. Vzdělávací systém bude muset předat do integrační platformy následující data:

- originDomain – URL domény, na kterou byl zavolán požadavek. Pomocí domény bude možné dohledat klienta, jehož integrace má být spuštěna.
- endpoint – část URL adresy za doménou, která označuje, jaká integrace se má spustit. Tento řetězec bude obsahovat dvě nebo tři části oddělené lomítkem v závislosti na typu služby. První část bude označovat typ služby (rest nebo soap) a v případě, že půjde o REST API, bude druhá část označovat HTTP metodu, která byla pro volání použita. Poslední část obsahuje název metadat integrace, podle kterého půjde dohledat, o jakou integraci konkrétního klienta jde. Každý zdroj pro REST API server bude muset mít nadefinovaná vlastní metadata, stejně tak jako každá funkce v SOAP API.
- urlParams – všechny parametry, které přišly s požadavkem v URL adrese. To bude potřeba v případě GET požadavku na REST API server nebo v případě některých typů autorizace.

- requestBody – tělo zprávy, které bude součástí REST API požadavku využívajícího metodu POST.
- requestHeaders – všechny hlavičky, které byly klientem poslány na endpoint API serveru. Také bude sloužit primárně pro ověření autorizace v integrační platformě.
- ip – IP adresa, ze které přišel požadavek. Pro některé integrace omezené na IP adresu dojde k ověření, že požadavek byl vyslán ze seznamu povolených IP adres.

Samotná třída *ApiServer* nebude obsahovat nic, než dvě prázdné metody, protože jejich definice je povinná z nadřazených abstraktních tříd. Veškerá přijatá data pak budou zpracována na výše uvedeném endpointu integrační platformy, který již bude mít data přijatá ze vzdělávacího systému a nebude tak potřebovat žádnou třídu z *TransferFactory*. Tento endpoint bude navíc muset umět autorizovat požadavky a kromě toho bude již shodný s běžným skriptem, který spouští cron v pravidelných intervalech.

Princip fungování běžného HTTP API (včetně REST API) a SOAP API bude z pohledu integrační platformy naprosto shodný a veškerou přípravu dat provede skript na straně vzdělávacího systému. Komunikace přes HTTP API je poměrně přímočará a je znázorněna na diagramu 4.5. V takovém případě budou metadata mít identifikátor `rest/get/users`, který rovnou označuje jejich umístění na serveru.



Obrázek 4.5 Komunikace od klienta k integrační platformě v případě využití typu přenosu ApiServer.

SOAP API server bude fungovat na straně vzdělávacího systému odlišně. Klientům bude potřeba poskytnout URL s WSDL a zařídit veškerou komunikaci mezi SOAP klientem a SOAP serverem. Jak pro poskytnutí WSDL, tak pro komunikaci bude využita knihovna PhpWsdl, která je implementována pro aktuální integrace běžící bez integrační platformy. Aby bylo možné tuto knihovnu použít, je potřeba vygenerovat PHP kód třídy, ve které jsou definované jednotlivé metody poskytnuté SOAP serverem a tento soubor nastavit jako vstupní data v PhpWsdl knihovně.

Knihovna se již postará o vše ostatní, co je potřeba dodržet při komunikaci mezi SOAP klientem a serverem. Pokud klient zavolá metodu, která je definována, tak knihovna tuto metodu spustí tak, jak je definována v dříve poskytnutém PHP souboru.

Pro potřeby integrační platformy bude stačit, aby metoda obsahovala cURL volání do integrační platformy na stejný endpoint, jako výše zmíněná HTTP služba. Parametry přijaté do metody budou odeslány v parametru *requestBody* a název metody, který je zároveň názvem souboru s metadaty, bude odeslán v parametru *endpoint*. Příklad definice takové metody je znázorněn v ukázce kódu 4.7.

```
public function getUsers($filtration) {
   $requestData = [
      'originDomain' => 'https://www.klient-lms.com',
      'endpoint' => 'soap/getUsers',
      'urlParams' => [],
      'requestHeaders' => ".var_export(getallheaders(), true).",
      'ip' => 'Klient IP',
      'requestBody' => $filtration
   ];

   $ch = curl_init();
   curl_setopt($ch, CURLOPT_URL, "https://int-plat.com/apiJob");
   curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
   curl_setopt($ch, CURLOPT_POST, true);
   curl_setopt($ch, CURLOPT_POSTFIELDS, json_encode($requestData));
   $responseJSON = json_decode(curl_exec($ch), true);
   curl_close($ch);

   return $responseJSON['responseBody'];
}
```

Výpis kódu 4.7 PHP kód s ukázkou definice metody pro PhpWsdl knihovnu obsluhující SOAP API server.

4.4.4 AbstractApiClient

Třídy, které dědí z abstraktní třídy *AbstractApiClient*, fungují pro získání dat pomocí připojení k API serveru. Tato třída bude oproti *AbstractApi*, které je potomkem, obsahovat pouze vlastnost *endpoint* a upravený konstruktor.

4.4.4.1 Soap

Pokud integrace získává data ze SOAP serveru, o přenos se postará tato třída. Ta by pro získání dat měla použít třídu *SoapClient*, která je již součástí PHP jazyka a obslouží veškerou nutnou komunikace se serverem. Stačí jí pouze doplnit nutné parametry, jako WSDL adresu serveru a metodu, kterou má na serveru zavolat. Dále nabízí některé volitelné paametry, jako způsob návratu chyb, maximální čas pro získání odpovědi ze serveru nebo autorizační údaje v hlavičce požadavku. Možný způsob získání dat ze SOAP serveru pomocí metody *getUsers* je znázorněn na kódu 4.8.

Implementace připojení na SOAP server by tedy neměla být nijak složitá.

```
// nastavení, aby se chyby vyhazovaly jako výjimky
$client = new SoapClient("https://www.klient.com/", ["exception" => 1]);
$users = $client->__soapCall("getUsers");
```

Výpis kódu 4.8 PHP kód pro získání dat z metody getUsers přes SoapClient třídu.

4.4.4.2 Rest

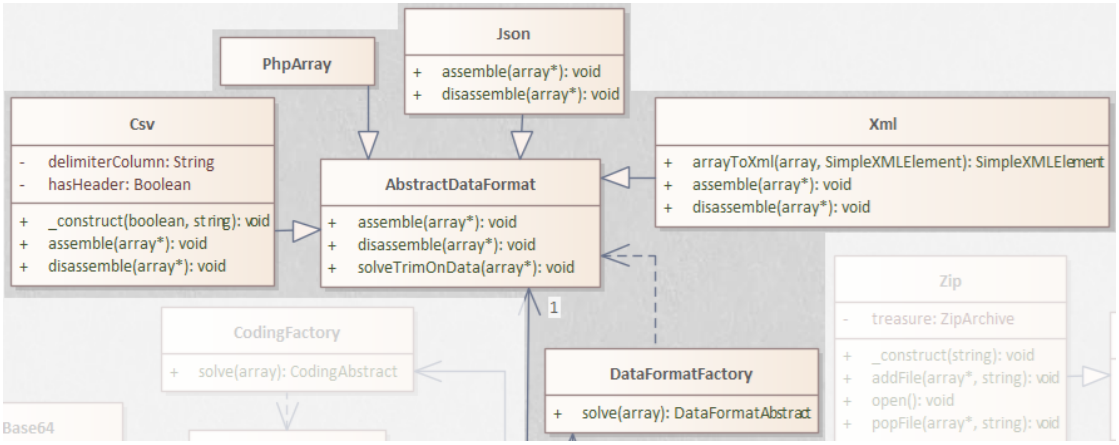
Připojení na REST API server, případně na libovolný HTTP API server, by se mělo řešit třídou *Rest* a na připojení by se měl používat nástroj *cURL*, který je možné využít pomocí výchozích funkcí v PHP. Vlastnost *method* označuje HTTP metodu, pomocí které se posílá request. Položka *customRequestBody* označuje data, která se pošlou jako tělo požadavku, pokud je potřeba definovat a *serviceIdentifier* označuje libovolné ID serveru. Toto ID bude sloužit k tomu, pokud autorizační způsob vygeneruje token, který lze použít opakovaně a v rámci jedné integrace je více požadavků na jednu službu. V takovém případě se nemusí generovat nový token pro každý požadavek. Jak je možné využít nástroj cURL v PHP je zobrazeno na ukázce kódu 4.7.

4.4.4.3 OAuth2Client

Tato třída bude sloužit k získání AccessTokenu, pomocí kterého se budou požadavky na API servery autorizovat. OAuth2.0 bude implementován pomocí knihovny *League/oauth2-client*, kterou je možné stáhnout pomocí Composeru. Tato knihovna nenabízí plně implementovaný OAuth2.0, protože každý provider využívá tento protokol jinak, například využitím jiného způsobu autorizace – GrantType. Knihovnu je potřeba doimplementovat dle potřeb providera, na kterého se klient snaží připojit.

4.5 Zpracování formátu dat

Všechny třídy pracující s formátem vstupních i výstupních dat budou vycházet z abstraktní třídy *AbstractDataFormat* a jsou znázorněné na obrázku 4.6. Abstraktní třída bude obsahovat abstraktní metody *assemble* a *disassemble* sloužící ke konverzi dat mezi polem, ve kterém jsou data uchována v průběhu integrace a zvoleným datovým formátem. Jelikož ze všech vstupních dat bude nutné ještě odebrat bílé znaky na začátku a na konci každého řetězce, bude zde ještě metoda *solveTrimOnData*. Ta se o to postará funkcí *trim* integrované v PHP jazyce a v případě, že položka bude obsahovat pole, tak bude pracovat rekurzivně.



Obrázek 4.6 Část diagramu tříd s třídami pro zpracování formátu dat.

Jednotlivé datové typy se budou zpracovávat následovně:

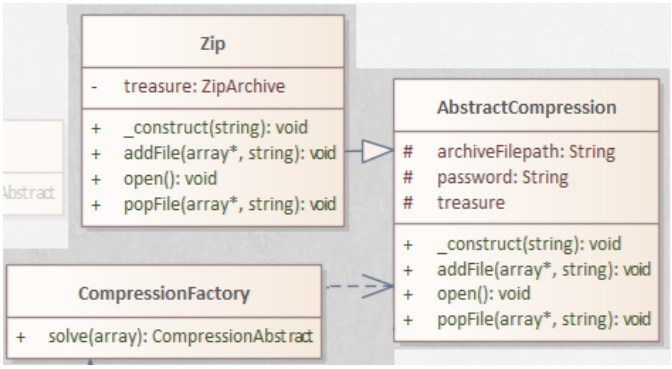
- Csv – v případě čtení i zápisu se využijí existující funkce v PHP, které pracují s CSV daty po řádcích. Jedná se o funkce *fputcsv* a *fgetcsv*, které pracují s filestream ukazatelem. Je tedy nutné vždy pracovat s dočasným souborem, i kdyby se data měla posílat přes API.

- Json – na tento formát budou využity funkce *jsonEncode* a *jsonDecode*, které jsou definované jako statické metody ve třídě *GlobalService*.
- Xml – i na tento formát se využijí prostředky v PHP jazyce k tomu určené. Konkrétně to bude třída *SimpleXMLElement*, ve které je možné XML formát vygenerovat pomocí metody na vytvoření nového potomka, případně zadání atributu. Naopak pro čtení XML a jeho převodu do *SimpleXMLElement* objektu bude použita funkce *simplexml_load_string()*.
- PHPArray – pokud budou data zpracovávána jako pole, například pro SOAP API rozhraní, není potřeba žádné další operace. Stačí je předat tak jak byla přijata.

4.6 Komprese, kódování a šifrování dat

4.6.1 Komprese

Pro potřeby komprese zatím budou existovat dvě třídy a továrna. První je abstraktní třída *AbstractCompression*, která bude uchovávat nutné vlastnosti jako jsou cesta k archivu, heslo, instanci objektu využitého k archivaci a dále abstraktní metody, které je potřeba v rámci potomků definovat. PHP jazyk nabízí několik tříd, které pracují s různými typy archivů a pokud by někdy mělo dojít k rozšíření typů archivů, budou využity další podobné třídy. V tuto chvíli bude využita pouze třída *ZipArchive* obsluhující archiv typu zip. O implementaci práce s touto třídou se postará třída v integrační platformě pojmenovaná *Zip*, která bude dědit z abstraktní třídy *AbstractCompression* definující abstraktní metody v nadřazené třídě. Všechny tyto třídy jsou zobrazené na části diagramu tříd na obrázku 4.7.



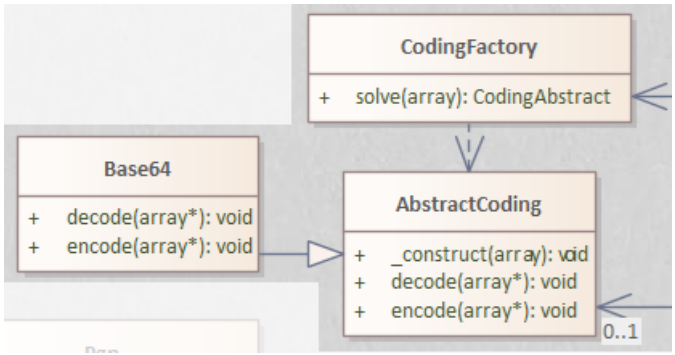
Obrázek 4.7 Část diagramu tříd s třídami pro zpracování komprese souborů.

4.6.2 Kódování

Pro kódování bude sloužit třída *Base64*, která bude dědit z abstraktní třídy *AbstractCoding*. Tyto třídy a továrna pro vytvoření třídy pro kódování je znázorněna na části diagramu tříd na obrázku 4.8. Na zakódování a dekódování do BASE64 se využijí výchozí funkce PHP jazyka, které jsou *base64_encode()* a *base64_decode()*. Díky abstraktní metodě bude kódování připraveno na případné další algoritmy, které lze pro zakódování využít.

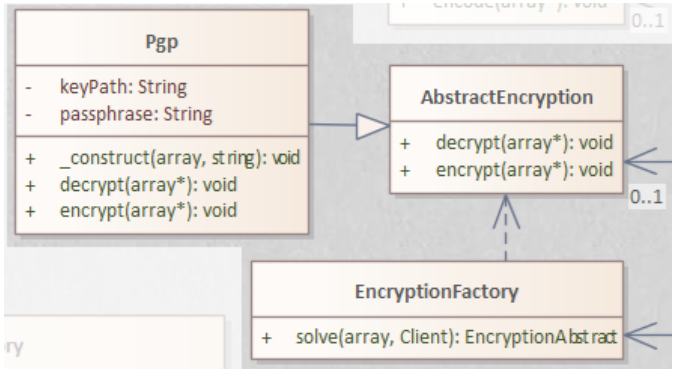
4.6.3 Šifrování

Podobně jako u archivace a kódování, i zde zatím bude pouze jedna abstraktní třída *AbstractEncryption*, třída z ní vycházející *Pgp* a továrna, jak je znázorněno na obrázku 4.9. Pro práci



Obrázek 4.8 Část diagramu tříd s třídami pro zpracování kódovaných dat.

s PGP šifrou bude využita opět třída, která je již součástí PHP jazyka a jedná se o třídu *gnupg*. Do této třídy stačí nahrát klíč k šifrování a jeho heslo, pokud je k dispozici. Poté již stačí jen zvolit cílová data, která se mají zašifrovat či dešifrovat.



Obrázek 4.9 Část diagramu tříd s třídami pro zpracování šifrovaných dat.

4.7 DataMixer

Instance třídy *DataMixer* se bude starat o jeden konkrétní blok stejnojmenného typu a bude obsahovat vlastnosti a metody jaké jsou zobrazeny na části diagramu 4.10. Přijme data ze vstupu a v předem daném pořadí uvedeném v sekci 3.5 provede jednotlivé operace nad daty. Pro každou operaci pak existuje vlastní třída vykonávající potřebné kroky k provedení operace. Instance těchto tříd se vytvoří v konstruktoru DataMixer třdy dle toho, co obsahuje blok v metadatech popisující tento DataMixer. Pro vykonání transformací se zavolá metoda *solve* stejně jako u ostatních bloků vycházejících z *AbstractDataBlock* třídy. Metoda *solve* bude vykonávat větší množství operací, a proto by měly být tyto operace rozděleny do jednotlivých metod, aby byla udržena přehlednost kódu této metody.

4.8 Operace s bloky daty

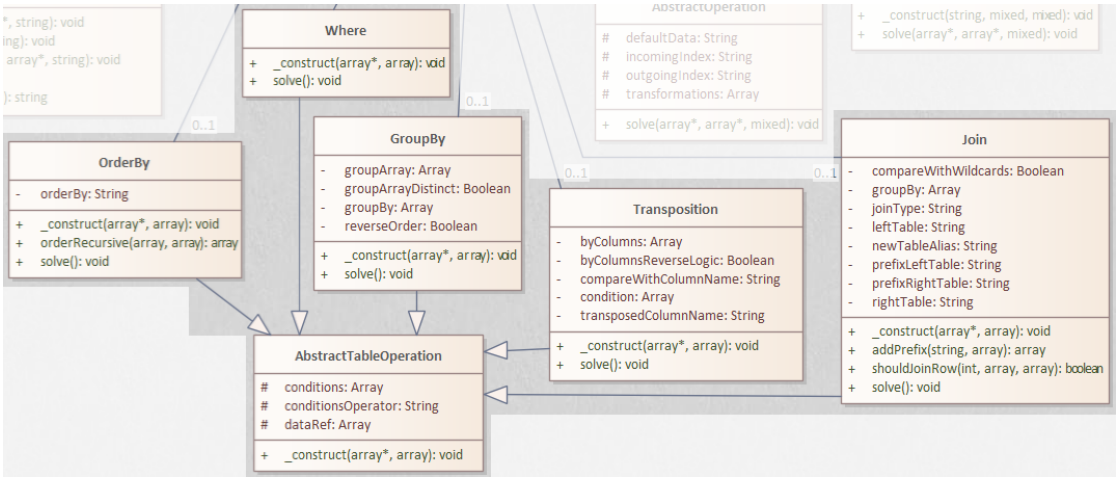
Veškeré operace s celými bloky dat vycházejí z abstraktní třídy *AbstractTableOperation*. Každá z těchto operací obsahuje podmínky, které musí být splněny pro další proces s daty. Tyto podmínky jsou proto uvedeny v abstraktní třídě. Jednotlivé operace jsou následující:

Obrázek 4.10 Část diagramu tříd s DataMixer třídou.

- **Join** – pomocí této třídy bude docházet ke spojení dvou vstupních polí na základě typu a podmínek. Veškeré operace budou probíhat nejspíš pomocí dvou zanořených cyklů a ověřování podmínek dle typu spojení.
- **Where** – tato třída pomocí metody *GlobalService::compareValue* určí pro každý řádek v bloku vstupních dat, zda patří i do výstupních dat nebo ne. Pokud ne, tak jej zahodí a nezařadí do výstupních dat.
- **GroupBy** – na sjednocování záznamů pomocí vybraných sloupců je potřeba zkontrolovat každý řádek v datech zvlášť pomocí operací s polem. Pravděpodobně by bylo vhodné využít formu indexování pole tak, aby bylo u každého kontrolovaného záznamu oproti výstupním datum jasné, jestli takový záznam již je ve výsledku a má být zahozen nebo ponechán. Během takové kontroly by pak neměl být problém sjednocovat konkrétní sloupečky, pokud je to požadované v metadatech.
- **OrderBy** – třída bude provádět řazení pole pomocí funkce *usort* obsažené přímo v PHP. Bude ovšem muset umět řadit data rekurzivně v případě, že se mají řadit pomocí více sloupců.
- **Transposition** – podobně jako některé předchozí operace bude potřeba pomocí cyklů projet všechny řádky a všechny sloupce. Následně data upravit tak, aby byla transponovaná. Žádná výchozí funkce v PHP neexistuje a je proto potřeba napsat celou tuto třídu svépomocí.

4.9 Wrapper

Pro práci s obaly dat bude existovat jedna třída obsahující funkci *solve*, která se bude v případě potřeby volat rekurzivně. Bude záležet na tom, jestli bude potřeba pracovat se zanořenými obaly nebo pouze na jedné úrovni. Jelikož obaly budou zpracovávány v DataDistributoru i v DataMixeru, tak budou v průběhu zpracování v datovém poli a nebudou již v původním datovém formátu, jako je XML nebo JSON. Z tohoto důvodu nelze použít selektory, které některé z těchto



Obrázek 4.11 Část diagramu tříd s třídami provádějícími operace nad bloky dat.

formátů nabízejí, jako je třeba *XPath* nebo *JSONPath* a pro extrakci dat pomocí *wrapperRemove* parametru bude potřeba pole s daty prohledat běžným způsobem pomocí rekurze a cyklu. I přesto by funkce *solve* měla být poměrně snadná a mohla by vypadat podobně jako na ukázce kódu 4.9.

Třída *Wrapper* bude dědit z abstraktní třídy *AbstractOperation*, která bude obsahovat většinu potřebných vlastností a jejich gettery a settery.

```
function solve (array $dataArray, array &$finalData) : void {
   // pokud se má wrapper přidat, vytvoří prázdné pole nebo doplní výchozí data
   if (type == self::TYPE_ADD) {
      $dataArray[$this->getOutgoingIndex()]=$this->getDefaultData ?? [];
   }

   // pokud se mají nějaké wrappery zanořit, rekurzivně proveď
   foreach($this->getNestedWrappers() as $nW) {
      $nW->solve($dataArray, $finalData);
   }

   // pokud se v tomto wrapperu mají uchovat/načíst data, proveď
   if($this->getStoredParsesHere()) {
      if (type == self::TYPE_ADD) {
         $dataArray[$this->getOutgoingIndex()] = $finalData;
      } else {
         $finalData = $dataArray[$this->getOutgoingIndex()];
      }
   }
}
```

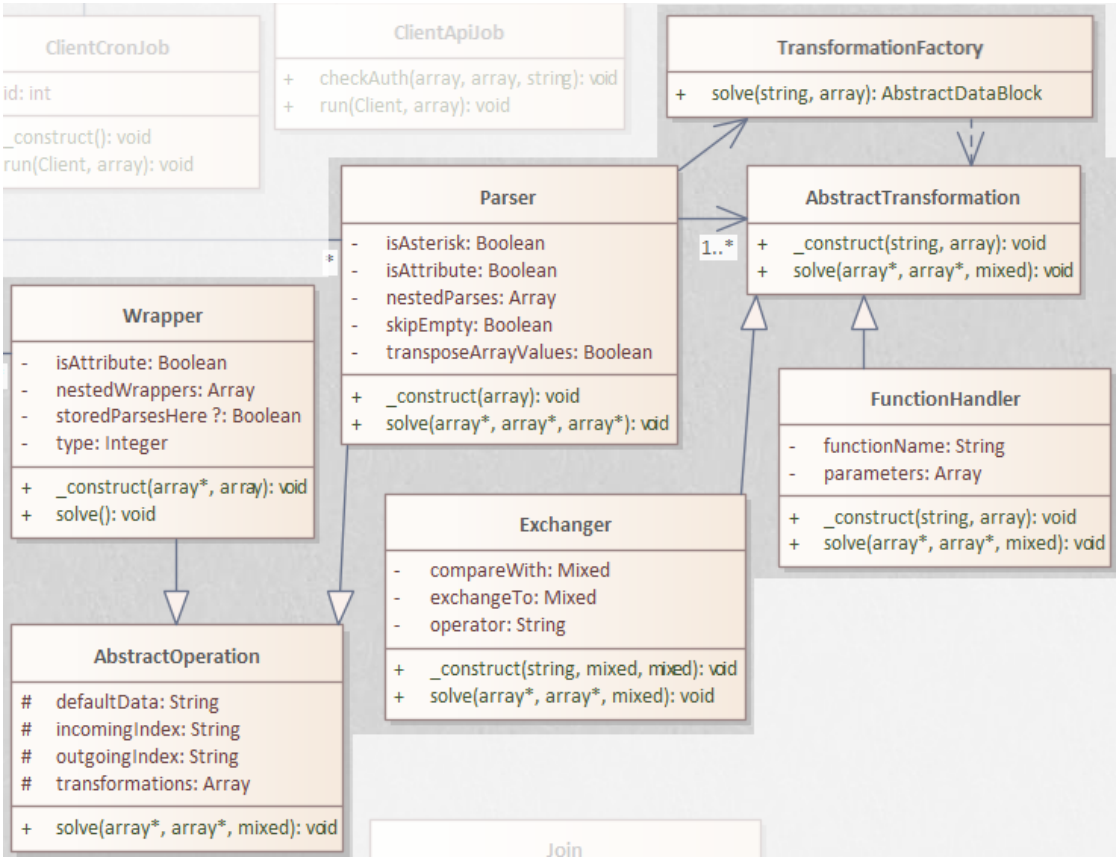
Výpis kódu 4.9 Ukázka kódu metody pro práci s obaly.

4.10 Projekce dat

Jak již bylo uvedeno v předchozí kapitole, tak k projekci dat bude v metadatech pole objektů *parsers*, pro které bude vytvořena třída *Parse*. Každý objekt pak bude zpracováván právě jednou instancí této třídy. Stejně jako třída *Wrapper*, bude i *Parse* dědit z abstraktní třídy *AbstractOperation*, kde jsou definované vlastnosti, settery a gettery, které jsou pro tyto dvě třídy společné.

Tyto třídy a všechny další třídy starající se o transformaci dat jsou zobrazeny na části diagramu 4.12.

Samotná třída *Parse* bude mimo dalších setterů a getterů obsahovat také konstruktor a metodu *solve*. Ta bude vykonávat veškerou logiku, která se má provést nad každým záznamem v datech. To znamená, že bude data jak přesouvat do výstupních dat daného DataMixeru pod správným indexem, tak na tato data volat transformace uvedené v metadatech jako *exchangeValue* a *functionsOnData*. Pro transformace budou vytvořeny instance tříd v konstruktoru třídy *Parse* pomocí továrny *TransformationFactory*.



Obrázek 4.12 Část diagramu tříd s třídami obsluhující obaly, projekci a transformaci záznamů.

4.10.1 Transformace záznamů

Továrna *TransformationFactory* bude vytvářet instance tříd *FunctionHandler* a *Exchanger*, které obě dědí z abstraktní třídy *AbstractTransformation*. Tato abstraktní třída existuje pouze kvůli abstraktnímu konstruktoru a metodě *solve*.

4.10.1.1 Exchanger

Funkce *solve* ve třídě *Exchanger* porovná zadanou hodnotu se vstupní hodnotou s využitím zadaného operátoru pomocí metody *GlobalService::compareValue*, která již byla využita například v podmínkách u tříd *Where* a *Join*. Navíc ověří, jestli není v některé z hodnot placeholder a případně ho nahradí, což provede zavoláním metody *PlaceholdersService::solvePlaceholders*. Žádná další operace v této třídě nebude potřeba.

4.10.1.2 FunctionHandler

Stejně jako předchozí třídy, i třída *FunctionHandler* bude pravděpodobně obsahovat pouze metodu *solve*. Celá třída bude obsluhovat bloky metadat pro zavolání funkce na objekt třídy *Parse*. Bude tedy muset připravit vstupní parametry funkce, které mohou být doplněny o již existující data *Parse* objektu pomocí placeholderu. Následně budě zjišťovat, jestli požadovaná funkce je definována blokem *ownFunctions* nebo je definována ve třídě *OwnFunctionsStorage* a nebo jde o funkci definovanou v jazyce PHP. Pokud funkce nikde definována není, tak bude vyhozena výjimka, které bude odchycena speciální funkcí k tomu určenou.

4.11 Shutdown funkce a výjimky

V integrační platformě mohou nastat tři různé situace, kdy bude ukončen proces integrace a zaznamenán chybový stav včetně zprávy. Pro každý takový typ chyby existuje obslužná funkce definovaná přímo v PHP, ve které je možné si zaregistrovat vlastní nebo anonymní funkci. Tyto funkce je potřeba nastavit na začátku běhu každého skriptu a díky tomu budou všechny chyby a neodchycené výjimky správně zalogovány. Tyto funkce jsou:

- **set_error_handler** – funkce zaregistrovaná tímto způsobem zpracovává chyby, které se vyvolávají pomocí zavolání funkce *trigger_error*. Tu používá většina výchozích funkcí v PHP. Chyby tohoto typu využívá například funkce *fopen* sloužící k otevření souboru, který nemusí existovat nebo může být určen pouze pro čtení.
- **set_exception_handler** – tato funkce registruje zadanou funkci k obsluze neodchycených výjimek. To bude využito v případě, že některé bloky kódu nebo externí knihovny vyhazují výjimky, kvůli kterým musí být ukončeno provádění skriptu. V takovém případě nemá smysl odchytávat a obsluhovat výjimku lokálně v místě jejího výskytu, ale stačí mít zaregistrovanou jednu funkci, která se o všechny tyto výjimky postará.
- **register_shutdown_function** – poslední funkce registruje obslužnou funkci, která se zavolá vždy, když PHP skript dokončí svou práci. Jelikož je potřeba obsluhovat pouze chybové stavy, je možné v této funkci zavolat *error_get_last()*, která vrátí pole s informacemi o chybě nebo prázdnou hodnotu *null*. Mezi chybami, které lze takto odchytit, patří například ukončení běhu skriptu z důvodu dosažení hodnoty *max_time_allowed* nastavené v php.ini souboru nebo překročením maximální dovolené paměti pro běh skriptu.

Všechny výše uvedené funkce pak budou primárně ukládat chyby do databáze. Dále pokud je v metadatech integrace nastavena položka *failAlert* s e-mailovými adresami, kam má být odeslán e-mail s informací, když integrace nedoběhne, tak k tomuto odeslání dojde ve výše definovaných funkcích.

Kapitola 5
Implementace a testování

Tato kapitola popisuje několik problémů a funkcí, které bylo potřeba vyřešit v průběhu implementace. Jedná se primárně o vylepšení, která nebyla objevena během návrhu aplikace. Některé problémy byly odhaleny v rámci lokálního testování při vývoji a na takové byly vytvořeny unit testy. Ty byly vytvořeny i na některé další funkce. Celé unit testování integrační platformy je v této kapitole popsané. Je zde uvedeno testování pomocí integračních metadat k příkladům z druhé kapitoly a návrh komplexního testování pro bezpečný dlouhodobý vývoj vzdělávacího systému i integrační platformy.

5.1 Debuggování integrací

Při prvních pokusech o spuštění integrace se vzdělávacím systémem bylo zjištěno, že debuggování chyb v průběhu integrace je poměrně složitá záležitost. Tyto chyby mohou vznikat jak špatně napsanými metadaty, vadnými vstupními daty, tak chybou přímo v kódu integrační platformy. Pokud se importní integrace o velkém počtu záznamů nechá proběhnout celá, včetně finálního zápisu do vzdělávacího systému, může trvat jednotky až desítky minut, kdy většinu tohoto času využívá LMS na vytváření nových uživatelských účtů a další složitější operace. Z tohoto důvodu bylo rozhodnuto, že by bylo dobré prohlédnout obsah pole s daty v integrační platformě ještě před provedením zápisu dat.

Jelikož se všechny typy integrací spouští pomocí požadavku na konkrétní URL adresu, byla přidána možnost do URL adresy zadat parametr *dryRun*. Pokud integrační platforma při spuštění integrace odhalí, že je tento parametr přítomný, tak před prvním kontejnerem typu *writeEduBuddy* nebo *writeDataDistributor* zastaví provádění integrace a vypíše do těla zprávy pole s daty celé integrace. Příklad, jak takové pole může vypadat, je uveden na ukázce kódu 5.1.

Na uvedeném výpisu kódu je možné si všimnout parametru *dd_id* u záznamu uživatele. Tento parametr byl doplněn také během implementace a vyplňuje se do něj identifikátor readDataDistributoru, který tento záznam načetl. Parametr vznikl z toho důvodu, že některé korporáty mohou předávat jednotlivé informace dceřinných společností v samostatných datech, kdy pro každou takovou společnost je napsán vlastní readDataDistributor. Formát dat je pak totožný a zpracovávají se jedním DataMixerem. Může ale nastat situace, kdy společnosti mohou mít nastavenou nějakou položku odlišně. V takovém případě je možné napsat pro každou společnost vlastní DataMixer, kdy se může stát, že jedna metadata budou obsahovat 5 DataMixeru, které se liší jen v jednom parse objektu a jinak jsou totožné, což by ztížilo jakoukoliv úpravu Datamixeru v metadatech. V případě využití parametru *dd_id* by zůstal zachován jeden DataMixer a ve zmíněném jednom parse objektu se využije *exchangeValue* nebo *functionOnData* pro individuální chování dle společnosti, od které přišla data.

```
{
  "dd_users": [
    {
      "Příjmení": "Doe",
      "Jméno": "John",
      "Titul Před": "Bc.",
      "Titul Za": "",
      "Email": "john.doe@mail.com",
      "Sekundární email": "",
      "OSČ": "1234",
      "Organizační jednotka - kód": "001",
      "Organizační jednotka - název": "Centrála",
      "Manažer": "0",
      "Profese": "Uklízeč",
      "OSČ nadřízeného": "3456",
      "dd_id": "dd_users"
    },
    { // data dalších zaměstnanců }
  ],
  "dd_courses": [ // data ze souboru example1_courses.csv ],
  "dm_users": [ // připravená data uživatelů pro import do LMS ],
  "dm_os_orgj": [ // data organizačních jednotek připravená pro import ],
  "dm_os_managers": [ // strom manažerů ],
  "dm_osUsers_orgj": [ // zařazení uživatelů do organizačních jednotek ],
  "dm_osUsers_zamestnanci": [ // zařazení uživatelů do skupiny zaměstnanci ],
  "dm_osUsers_vedouci": [ // zařazení uživatelů do skupiny vedoucí ],
  "dm_osUsers_manazeri": [ // zařazení uživatelů pod své manažery ],
  "dd_users_min_prepare": [ // pomocný DataMixer ],
  "dm_usersCourses": [ // data s vazbou uživatel - kurz ]
}
```

Výpis kódu 5.1 Ukázka dat vypsaných pomocí parametru dryRun.

Z obrázku je patrné to, co bylo uvedeno v návrhu implementace v sekci 4.2.3 a sice, že indexem pole je vždy identifikátor bloku, pod kterým jsou uložena jeho výstupní data.

5.2 Vylepšení výkonu během spojování dat pomocí Join třídy

Nejpomalejší operace, která se v integrační platformě provádí, je spojování více vstupních polí do jednoho. Asymptotická složitost spojení je rovna `m * n`, kde m je počet záznamů v prvním poli s daty a n je počet záznamů v druhém poli. Navíc se velmi často spojují data uživatelů sama se sebou, aby se ke každému zaměstnanci načetl jeho vedoucí, který je ve stejných datech. U větších firem se může jednat o napojení dat, která mají i více jak 15 tisíc záznamů, což v případě spojení tabulky na sebe je více jak 225 tisíc operací. V případě, že porovnání vyjde pozitivně, dochází ke sjednocení dvou polí a uložení do jiného pole, což je také složitější operace. Z těchto důvodů bylo vypozorováno, že tato složitější spojení mohou trvat i několik desítek minut.

Aby se režie porovnávání urychlila, bylo navrženo využití indexů. To funguje tak, že se pro každý záznam nastaví index. Ten je složený z položek vyskytujících se ve všech podmínkách využitých k provedení spojení, které jsou součástí řetězce s jedním speciálním znakem jako oddělovačem. Jelikož spojení v rámci integrací jsou málokdy unikátní, dojde tím k zásadní úspoře počtu porovnání. Jako příklad lze uvést spojení s třemi podmínkami, které ověřují osobní číslo nadřízeného, stejný název společnosti jako má nadřízený a stejnou adresu výkonu práce. Vzorová data jsou ukázána na tabulce 5.1.

osobni-cislo	username	spolecnost	misto-vykonu-prace	osobni-cislo-vedouci
123	manager	Alfa s.r.o.	Praha	000
456	poddany1	Alfa s.r.o.	Praha	123
789	poddany2	Beta s.r.o.	Praha	123
001	poddany3	Alfa s.r.o.	Praha	123

Tabulka 5.1 Vzorová data pro spojení tabulky dle nadřízeného.

Takováto data budou pomocí výše uvedených podmínek zapsaná v metadatech jako na výpisu kódu 5.2.

```
"dataSource": "dd_users",
"join": [
{
      "rightTable": "dd_users",
      "joinType": "LEFT",
      "prefixRightTable": "manager",
      "conditions": [
         {
            "leftValue": "[__osobni-cislo-vedouci__]",
            "condition": "==",
            "rightValue": "[__osobni-cislo__]"
         },
         {
            "leftValue": "[__spolecnost__]",
            "condition": "==",
            "rightValue": "[__spolecnost__]"
         },
         {
            "leftValue": "[__misto-vykonu-prace__]",
            "condition": "==",
            "rightValue": "[__misto-vykonu-prace__]"
         }
      ]
   }
]
```

Výpis kódu 5.2 Zápis spojení tabulky dle nadřízeného v metadatech integrace v DataMixeru.

V případě, že by zde nebylo žádné indexování, došlo by k porovnání 3 hodnot mezi 4 řádky. Pokud by došlo k vytvoření indexů, tak by indexy vypadaly pro jednotlivé řádky tak, jak je uvedeno v tabulce 5.2 s tím, že každá vstupní tabulka využitá ke spojení má vlastní sadu indexů.

Tabulka	Index	Čísla řádků pro daný index
Levá	000^Alfa s.r.o.^Praha	1
Levá	123^Alfa s.r.o.^Praha	2,4
Levá	123^Beta s.r.o.^Praha	3
Pravá	123^Alfa s.r.o.^Praha	1
Pravá	456^Alfa s.r.o.^Praha	2
Pravá	789^Beta s.r.o.^Praha	3
Pravá	001^Alfa s.r.o.^Praha	4

Tabulka 5.2 Vygenerované indexy pro spojení dvou tabulek.

Porovnání pak funguje tak, že se porovná první řádek z každého indexu z levé tabulky

s prvním řádkem každého indexu z pravé tabulky a pokud porovnání vyjde pozitivní, napojí se dle typu spojení všechny záznamy z levé tabulky pod konkrétním indexem se záznamy z pravé tabulky pod indexem. Ve výše uvedeném případě dojde k porovnání 3 záznamů z levé tabulky se 4 záznamy z pravé tabulky. V porovnání s verzí bez indexů dojde k úspoře 4 operací. Asymptotická složitost prohledávání záznamů ke spojení v případě využití indexů má horní mez `2m + n`, což je rozhodně o dost méně než původní `m * n`.

Pro ověření v praxi došlo k testu na reálných datech jednoho z klientů, který zaměstnává větší počet zaměstnanců. V původní integraci se zdrojová data od klienta napojila sama na sebe podobně, jako je uvedeno v příkladu výše, protože bylo potřeba ke každému uživateli najít záznam jeho nadřízeného zaměstnance. Tato operace se vykonala ještě podruhé, aby u původního záznamu zaměstnance nebyl pouze jeho přímý vedoucí, ale i vedoucí jeho vedoucího. Původní integraci nebylo bez indexů vůbec možné přepsat do metadatové podoby, protože skript s dvěmi spojeními běžel déle jak hodinu, což je maximální povolená doba běhu skriptu na serveru. I kdyby nebylo na serveru takové omezení, není rozumné, aby každou noc zabrala na hodinu jedno jádro procesoru pouze jediná integrace. V přípapadě využití indexů u spojení se doba běhu skriptu zkrátila na pouhých 276 sekund. V rámci testů byla vyzkoušena integrace pouze s jedním spojením, kdy se k zaměstnanci našel jeho přímý vedoucí a už se nehledal vedoucí vedoucího. V takovém případě již verze bez indexování doběhla za 3029 sekund a s index za 250 sekund. To znamená, že verze bez indexů byla v tomto případě víc jak 11x pomalejší. Detailní výsledky testu jsou zobrazeny v tabulce 5.3.

Využití indexů	Počet spojení	Počet zaměstnanců celkem	Čas (s)
Ne	1	10308	3029
Ne	2	10308	Více jak 3600
Ano	1	10308	250
Ano	2	10308	276

Tabulka 5.3 Výkonnostní test spojení s využitím indexů.

5.3 SOAP API klient a časový limit

Když byl vytvořen způsob přenosu dat přes SOAP API rozhraní, kdy je integrační platforma v pozici klienta, došlo zároveň i k drobnému otestování u jednoho z klientů, který tento typ integrace používá starým způsobem bez integrační platformy. Tento klient v danou dobu vyžadoval změnu v aktuální integraci, což byla ideální doba na to, přepsat celou integraci do metadatového souboru. Změna spočívala v přidání další dceřinné společnosti, jejíž data bude přes integraci předávat do vzdělávacího systému. V součtu již integrace obsahovala zhruba pět tisíc zaměstnanců včetně dat organizační struktury. Během testování ovšem integrace občas proběhla v pořádku a jindy bez zjevného důvodu spadla na chybě SOAP rozhraní. Po důkladném prozkoumání chyby a logů bylo zjištěno, že SOAP serveru, který poskytuje data integrační platformě, trvá vygenerování odpovědi na požadavek v rozmezí 53 až 65 sekund. Ve výchozím nastavení čeká PHP na odpověď 60 sekund, a proto docházelo k tomu, že někdy již PHP nečekalo a komunikaci ukončilo. Z tohoto důvodu byla v integrační platformě nastavena výchozí hodnota pro čekání na odpověď na 500 sekund a zároveň bylo umožněno ji nastavit v metadatech jako položku *timeout*. Pomocí zmíněného opatření by se již neměl vyskytnout problém se získáním dat ze SOAP API serveru kvůli krátké době čekání na odpověď.

5.4 Podpora GraphQL

Během implementace bylo zjištěno, že vedlejší výhodou toho, jak je navržený HTTP API klient (třída Rest) je, že se při správném nastavení v metadatech lze připojit i na server využívající GraphQL. Ten specifikuje dotazovací jazyk, který se využívá pro požadavky na API server. Oproti REST architektuře je potřeba v GraphQL specifikovat konkrétní data, která chce klient získat. Tím se minimalizuje objem přenesených dat, protože klient nestahuje i data, o která nemá zájem, jak je tomu u REST architektury. [22] Získání dat pomocí HTTP protokolu a GraphQL dotazovacího jazyka může vypadat jako na ukázce kódu 5.3.

```
curl -X POST
http://hrsys.com/graphql
-H "Content-Type: application/json"
-d '{ "query": "query { user(id: \"1203\") { id name username } }" }'
```

Výpis kódu 5.3 Ukázka HTTP požadavku na server podporující GraphQL.

Takový požadavek by se v metadatech integrace mohl zapsat jako *readDataDistributor* a vypadal by jako na ukázce kódu 5.4.

```
{
   "identifier": "dd_user1203",
   "description": "Data uživatele s ID 1203.",
   "settings": {
      "typeConnection": "REST",
      "format": "json",
      "connectionSettings": {
         "rest": {
            "method": "POST",
            "endpoint": "http://hrsys.com/graphql",
            "customHeaders": {
               "Content-Type": "application/json"
            },
            "customRequestBody": "{ 'query': 'query { user(id: \"1203\")
                { id name username } }' }"
         }
      }
   }
}
```

Výpis kódu 5.4 Metadata po získání dat pomocí GraphQL.

Vzhledem k těmto skutečnostem by bylo vhodné udělat několik změn v třídách integrační platformy. Konkrétně přejmenovat třídu *Rest* na *HttpApi*, ze které by měla nově vytvořená třída *Rest* dědit stejně jako nová třída *GraphQL*. Jelikož žádný z klientů vzdělávacího systému momentálně nevyužívá GraphQL jazyk, nebudou tyto změny zatím realizovány. Stejně jako nebylo otestováno reálné použití GraphQL a pouze se předpokládá, že by vše mělo fungovat a vyžadovat minimální úpravy na straně integrační platformy. Primárně by byla potřeba úprava zápisu metadat, aby typ připojení byl *GraphQL* a nikoliv *REST* a nebylo potřeba dotaz posílat v položce *requestBody* jako řetězec, ale bylo umožněno zadat i přímo dotaz ve formátu JSON.

5.5 Unit testy

Aby bylo možné integrační platformu snadno otestovat při změnách ve zdrojových kódech, bylo k ní napsáno několik unit testů, které během několika málo sekund ověří základní funkčnost některých dílčích celků integrační platformy. Pro unit testování byl zvolen PHPUnit framework,

který je dostupný přes službu Composer. Vytváření testů probíhá tak, že jsou vytvořeny nové třídy, které jsou potomky třídy *TestCase* definované v PHPUnit frameworku. Každá veřejná metoda v této třídě pak funguje jako samostatný test.

U EduBuddy bloků lze pomocí unit testů otestovat pouze přípravu parametrů a dat do CURL požadavku, což není část kódu, která by se měla měnit. CURL volání do vzdělávacího systému nelze pomocí unit testů plnohodnotně otestovat, protože vzdělávací systém může sice vrátit chybu v případě, kdy je požadavek nevalidní, ale pokud by přišla validní špatná data, naimportuje je a vrátí HTTP kód 200. Pro tento případ bude potřeba vymyslet složitější způsob testování popsaný v sekci 5.7. Podobné pak platí pro DataDistributor testy, kdy se unit testy nedá ověřit připojení na FTPS, SFTP, API atd. pokud nebudou na testovacím serveru existovat přístupy na jednotlivé protokoly. Z tohoto důvodu je unit testování zaměřené primárně na DataMixer bloky, které lze plnohodnotně a rychle otestovat v rámci požadavku na serveru s integrační platformou.

V době psaní této práce existovalo celkem 86 unit testů, z nichž 41 byly testy pro DataMixer bloky, které by měly otestovat všechny jejich funkce. Pro testování DataMixer bloků je potřeba nejdříve nadefinovat základní univerzální metadata, nad kterými se unit test provede. Každý konkrétní test pak v metadatech provede změnu, aby spuštění metadat v integrační platformě vyžadovalo spuštění testované funkce. Tato univerzální metadata a vstupní data pro testování jsou uložena na přiloženém médiu jako soubory unitMetadata.json a unitInputData.json. Jako příklad je uveden test pro ověření funkčnosti spojení typu LEFT na výpisu kódu 5.5.

```
public function testLeftJoin(): void {
    // simulovaná příchozí data do integrace
    $dataFromDataDistributor = [ ... ];
    $metaData = static::$metaData;
    // úprava univerzálních metadat o správný typ funkce JOIN
    $metaData['containers'][0]['blocks'][0]['settings']
             ['join'][0]['joinType'] = 'LEFT';

    // očekávaný výsledek DataMixer bloku
    $expectedData["mixer1"] = [ ... ];

    static::resolveResult($metaData,
                          $dataFromDataDistributor,
                          $expectedData);
}
```

Výpis kódu 5.5 PHPUnit test LEFT JOIN funcke.

Z metody provádějící test je patrné vše výše uvedené. Na konci metoda zavolá statickou metodu, kterou volají všechny DataMixer unit testy. Ta provede spuštění *solve* metody nad DataMixer blokem a porovná výsledek z této metody s očekávaným výsledkem. Pokud se shodují, test úspěšně proběhl.

5.6 Komplexní testování dle příkladů

Po vytvoření unit testů a kompletní implementaci integrační platformy bylo potřeba vytvořit několik integrací pro komplexní otestování integrační platformy jako celku. Jako vhodné zadání pro takové integrace byla vybrána vzorová zadání z druhé kapitoly. Ta by měla otestovat většinu běžných use casů, ke kterým v mnoha integracích dochází.

5.6.1 Metadata integrace k příkladu 1

První testovací metadata popisují příklad ze sekce 2.1. Jde o import uživatelů, organizační struktury a přidělení kurzů pomocí lokálního souboru v CSV formátu zašifrovaného PGP šifrováním.

Nově vzniklá metadata integrace jsou k nalezení na přiloženém médiu jako soubor example1.json. Tato integrace obsahovala několik konstruktů, které bylo zapotřebí vymyslet jednou a v mnoha dalších integracích se využijí jen s minimálními změnami.

Během psaní metadat bylo rozhodnuto o implementaci funkce do *OwnFunctionsStorage* třídy, která bude spojovat textové řetězce a bude se jmenovat *joinStrings*. Tato funkce bude aliasem k volání funkce `implode("", $args)` a v této konkrétní integraci se bude využívat na vytvoření názvu a identifikátoru skupin organizačních jednotek a názvu skupin v manažerském stromu, které se budou skládat z příjmení a jména manažera.

Tato integrace obsahuje dva zajímavé konstrukty. Prvním z nich je vytvoření stromu manažerů v organizační struktuře, kde do jednotlivých skupin manažerů jsou zařazeni jejich podřízení. Druhou zajímavostí je přidělování kurzů s využitím pomocné tabulky v CSV.

5.6.1.1 Strom manažerů v organizační struktuře

Pro vytvoření stromu manažerů pomocí sloupce s ID manažera, které je dostupné u každého zaměstnance, bude potřeba provést spojení dat ”samy na sebe”. Aby takto nalezeného vedoucího, kterému se vytvoří skupina ve stromu manažerů, bylo možné zařadit pod skupinu jeho vlastního vedoucího, dojde ještě k opakovanému spojení nově vytvořených dat s původními zdrojovými daty uživatelů. Jelikož identifikátory skupin jsou složeny z osobních čísel manažerů, tato druhá operace spojení by teoreticky nebyla nutná, ale dojde k ní z toho důvodu, aby se všechny skupiny zařadily vždy buď pod data existujícího nadřízeného manažera vyskytujícího se v příchozích datech nebo pod kořenovou skupinu manažerského stromu. Toto druhé spojení je potřeba proto, aby data odesílaná z integrační platformy do vzdělávacího systému byla validní a všechny skupiny, které jsou označené jako nadřazené ve vzdělávacím systému existovaly. Jelikož by data od klientů mohla obsahovat chybu v podobě chybějících záznamů, tak by některé nadřazené skupiny nemusely ve vzdělávacím systému existovat. Ukázka těchto spojení je vidět na výpisu kódu 5.6. Ve zbytku DataMixeru již dojde jen k selekci potřebných dat a nahrazení neexistujících nadřazených skupin za výchozí kořenovou skupinu manažerského stromu.

```
"dataSource": "dd_users",
"join": [{
    "rightTable": "dd_users",
    "joinType": "LEFT",
    "prefixRightTable": "manager",
    "conditions": [{
        "leftValue": "[__OSČ nadřízeného__]",
        "condition": "==",
        "rightValue": "[__OSČ__]"
      }
    ]
  },{
    "rightTable": "dd_users",
    "joinType": "LEFT",
    "prefixRightTable": "top_manager",
    "conditions": [{
        "leftValue": "[__manager.OSČ nadřízeného__]",
        "condition": "==",
        "rightValue": "[__OSČ__]"
      }
    ]
  }
]
```

Výpis kódu 5.6 Spojení v DataMixeru pro vytvoření stromu manažerů v organizační struktuře.

5.6.1.2 Přidělování kurzů s pomocnou tabulkou v CSV

Tento konstrukt se skládá ze tří kroků. Prvním je předfiltrace vstupních dat uživatelů v pomocném DataMixeru, který vrátí ke každému uživateli pouze nezbytná data k provedení transpozice, aby nebylo nutné vypisovat všechny nadbytečné položky uživatele do položky *byColumns* během transpozice. Druhým krokem je provedení spojení mezi záznamy uživatelů a daty s kurzy z pomocného CSV, kdy se použijí všechny potřebné podmínky. V tomto konkrétním příkladu jde pouze o podmínku, zda je uživatel manažerem nebo nikoliv. Posledním krokem je provedení transpozice spojených dat, která byla již detailně popsána během návrhu v sekci 3.5.4.

5.6.2 Metadata integrace k příkladu 2

Další testovací metadata popisují příklad ze sekce 2.2 a jsou k nalezení na přiloženém médiu jako soubor example2.json. Podobně jako v předchozích metadatech, i zde byly využity konstrukty pro vytvoření stromu manažerů a přidělování kurzů s využitím pomocné tabulky. Rozdíl ve využití přidělování kurzů z předchozí integrace je v rozšíření, kdy jsou využity dvě podmínky ve spojení a využití operátoru $*$ ve sloupci *workPosition*, aby byl jeden z kurzů přidělen všem manažerům bez ohledu na jejich pracovní pozici. Příznak, zda je uživatel manažerem nebo ne se zjistí pomocí DataMixeru, který se pokusí spojit seznam uživatelů sám se sebou za využití sloupečku *ManagerKID*. Pokud se uživatelovo osobní číslo vyskytuje ve sloupečku *ManagerKID* jiného uživatele, pak se takovému uživateli nastaví příznak *manager* na 1. V opačném případě se nastaví na 0.

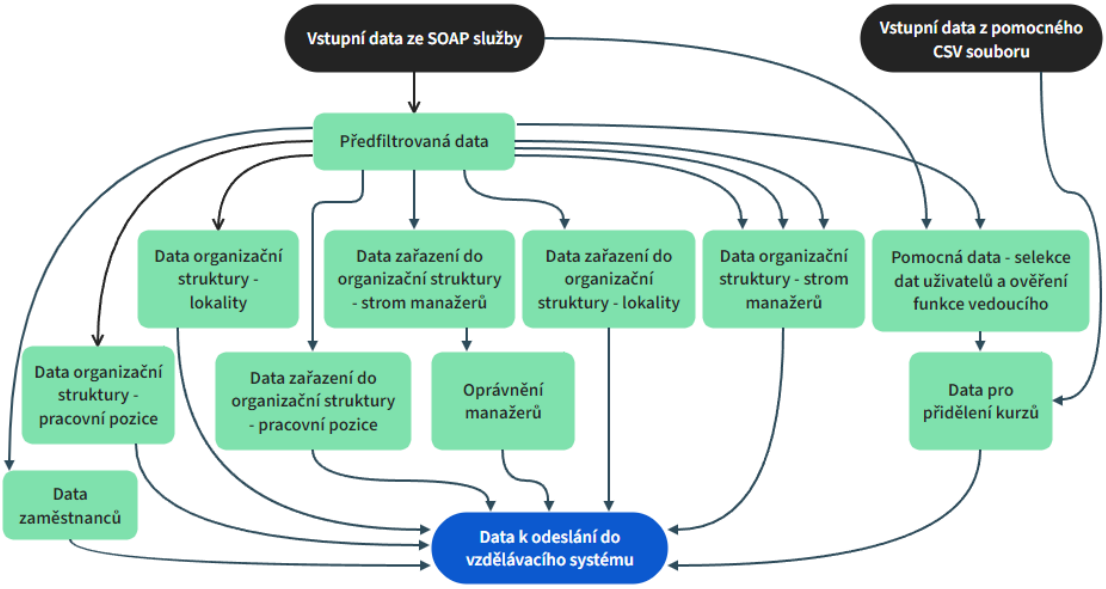
Důležitým rozdílem oproti předchozí integraci je vytvoření pomocného DataMixeru s identifikátorem *dm_users_prepare*, ve kterém dojde k filtraci neúplných záznamů ze vstupních dat a s výstupem tohoto DataMixeru pak pracují všechny následující DataMixery. Tím se hned na začátku běhu integrace zredukují vstupní data o nevalidní vstupy, což již není potřeba kontrolovat nikde znovu. Další funkcí, která byla vyzkoušena pomocí tohoto příkladu na úrovni vzdělávacího systému, je přidělování oprávnění manažerům tak, aby měli oprávnění reportovat uživatele ze své skupiny ve stromě manažerů. K vytvoření dat, která přidělují oprávnění byl jako zdroj využit DataMixer, který generuje strom manažerů, čímž byli uživatelé s oprávněním manažera již předfiltrováni jiným DataMixerem.

Na obrázku 5.1 jsou pomocí zelených bloků zobrazeny všechny DataMixer bloky, které se v této integraci vyskytují. Zároveň obsahují stručný popis, jaká data se v nich připravují. Černé bloky znázorňují vstupní data získaná pomocí DataDistributor kontejneru a modrý blok pak EduBuddy kontejner, který všechna data odesílá do vzdělávacího systému. U každého DataMixeru je pak znázorněno, jaká data přijímá jako vstupní a kde jsou využita data, která generuje.

5.6.3 Metadata integrace k příkladu 3

Jedná se o testovací metadata popisující příklad ze sekce 2.3 a uložená na přiloženém médiu jako soubor example3.json. To znamená, že jako první metadata testují opačný směr integrace a exportují data ze vzdělávacího systému ke klientovi. Na těchto metadatech byl otestován nejen opačný směr, ale také využití API serveru na straně vzdělávacího systému, filtrace dat získaných ze vzdělávacího systému a hlavně funkce pro přidání obalu pomocí *wrapperAdd* parametru. Jelikož je formát výstupních dat zvláštní tím, že v jednom XML mají být všechna požadovaná data různě zanořena, což je vidět na XML schématu 2.1, nebude stačit využití *wrapperAdd* parametru v DataDistributoru. Každý z DataMixerů, který poskytuje data do výstupu, bude muset mít vlastní obal, který je nastaven pomocí zanoření dalšího *parse* bloku. Parse objekt jednoho z těchto DataMixerů, který připravuje data se seznamem všech kurzů, je na ukázce kódu 5.7.

Nejedná se o nijak složitá metadata, protože v případě exportu dat ze vzdělávacího systému bude vždy důležité správně nastavit parametry pro filtraci dat získaných z LMS, čímž se většina



Obrázek 5.1 Diagram práce s daty během integrace.

logiky integrace přesune na stranu vzdělávacího systému a v integrační platformě pak dojde k selekci požadovaných dat a případně k jejich doplnění.

```
"parse": [
  {
    "outgoingIndex": "Kurz",
    "parse": [
      {
        "incomingIndex": "courseId",
        "outgoingIndex": "KurzID"
      },
      {
        "incomingIndex": "name",
        "outgoingIndex": "Nazev"
      },
      {
        "incomingIndex": "schedule",
        "outgoingIndex": "Perioda"
      }
    ]
  }
]
```

Výpis kódu 5.7 DataMixer pro přípravu dat kurzů.

5.6.4 Metadata integrace k příkladu 4

Poslední testovací metadata popisují příklad ze sekce 2.4 a jsou uložena v souborech example4all.json a example4-1d.json na přiloženém médiu. Jedná se o podobný export jako v předchozím příkladu až na dva rozdíly. Prvním je ukládání exportovaných dat do lokálního CSV souboru a druhým je, že výstupem musí být dva soubory, kdy jeden obsahuje data absolvování za poslední den a druhý obsahuje kompletně všechna data o absolvovaných kurzech. Způsoby provedení

integrace a uložení výstupů do dvou souborů s požadovanými daty jsou dva:

1. Pomocí jednoho EduBuddy bloku v metadatech se získají všechna potřebná data ze vzdělávacího systému najednou. Následně jeden DataMixer připraví všechna tato data pro první soubor a druhý DataMixer využije parametru *where* a vyfiltruje pouze záznamy za předchozí den, které budou sloužit jako data do druhého souboru.

2. Vytvoření dvou nezávislých metadat, kdy první získají data z LMS pouze za poslední den pomocí nastavené filtrace přímo při vytváření požadavku na vzdělávací systém a ty uloží do souboru a druhá metadata získají všechna data a uloží je do druhého souboru. Tuto varianta lze zapsat do jednoho metadatového souboru, ale pro lepší přehlednost bude vhodnější využít soubory dva.

Jako lepší řešení se zdá být druhý způsob, kdy dojde k filtraci dat na straně vzdělávacího systému přímo při dotazu do databáze a nikoliv jako u prvního způsobu, kdy by se data musela projít pomocí cyklu a porovnat u každého záznamu dva řetězce obsahující datum. Z tohoto důvodu vznikla dvoje metadata, která jsou spouštěna po sobě.

5.6.5 Shrnutí výsledků během testování celých integrací

Testování integrační platformy pomocí uvedených příkladů neodhalilo žádné zásadní chyby, a to pravděpodobně díky lokálnímu testování během vývoje a vzniklým unit testům. Pokud se nějaká chyba vyskytla, jednalo se o překlep či nedokončenou myšlenku v rámci zdrojových kódů integrační platformy. K čemu se ovšem tento způsob otestování nakonec hodil, bylo k vytvoření několika konstruktů pro snazší vytváření nových integrací administrátory do budoucna. Jedná se hlavně o strom manažerů v organizační struktuře a přidělování kurzů s využitím pomocného CSV souboru. Oba tyto požadavky se v integracích opakují velmi často a při jejich opakovaném použití se projevuje i celková výhoda integrační platformy, kdy pro podobná zadání dvou různých klientů budou využity velmi podobné metadatové soubory s minimálními rozdíly. Například bude použit jiný DataDistributor pro získání dat, v DataMixeru budou použity jiné názvy vstupních položek a v pomocném CSV souboru pro přidělení kurzu budou jiné podmínky a jiné identifikátory kurzů.

Samotná integrační platforma se v tomto momentě jeví jako dostatečně otestovaná a pokud by se vyskytly nějaké další chyby v průběhu vytváření integrací využívajících jiné funkce integrační platformy, budou opraveny vydáním nové verze s těmito opravami. Jelikož by mohlo dojít jednou opravou ke vzniku nové chyby, kterou by se nepodařilo odhalit dosavadním testováním, vznikl návrh pro automatické komplexní testování, které by bylo vhodné pro integrační platformu vytvořit.

5.7 Návrh automatického komplexního testování

Během nasazování integrační platformy a spouštění prvních integrací došlo několikrát k problému, kdy menší změna v chování, ať už na straně integrační platformy nebo vzdělávacího systému, zapříčinila v lepším případě nefunkčnost některých integrací a v horším případě začaly integrace do vzdělávacího systému posílat jinak formátovaná data, která způsobila například špatné sestavení organizační struktury. Využití unit testů nedokáže se 100% účinností eliminovat všechny takové chyby, protože za jejich vznikem často stojí návrh na změnu, který by stejnou chybu mohl propsat do očekávaných výsledků unit testů.

Z tohoto důvodu bylo navrženo řešení, které by mělo eliminovat téměř všechny nežádoucí změny jak v integrační platformě, tak LMS, vedoucí k nechtěnému chování celého procesu integrace. Návrh spočívá v přípravě testovacích dat a v samotném testování, během kterého se budou integrace spouštět v testovacím provozu na vlastním serveru k tomu určeném, protože by

takové testy nadbytečně zatěžovaly produkční prostředí. Porovnání výsledků testu bude provedeno pomocí porovnání databáze instance vzdělávacího systému před a po provedení integrace.

Příprava testovacích dat bude probíhat v následujícím pořadí:

1. Na produkčním prostředí v centrální administraci zvolí administrátor, že chce zařadit do fronty požadavek na vytvoření testovacích dat.
2. Jakmile dojde ke spuštění ostré integrace na produkčním prostředí, dojde ke kontrole, zda se mají vytvořit testovací data, pokud ano, pokračuje následujícími kroky. Pokud ne, pokračuje v běžném zpracování integrace.
3. Vytvoří export aktuálního stavu databáze instance LMS, na které se spouští integrace a tento export si uloží.
4. Zkopíruje si aktuální metadata integrace.
5. Zkopíruje všechny vstupní soubory, se kterými integrace pracuje, pokud nějaké jsou (například CSV soubor od klienta, pomocné CSV soubory pro přidělení kurzů atd.)
6. Pokud se jedná o integraci, která získává data pomocí jiného způsobu než zpracováním souboru (SOAP API a REST API rozhraní, ať už z pozice klienta či serveru, atd.) tak nastaví příznak, pomocí kterého bude integrační platforma vědět, že všechna takto získaná data má uložit v předem určeném formátu (pravděpodobně JSON) k připraveným testovacím datům.
7. Následně proběhne integrace běžným způsobem.
8. Pokud integrace úspěšně doběhne, vytvoří se druhý export celé databáze klienta, který se uloží a zaloguje se úspěch k získání všech dat potřebných k testování integrace.
9. Jestliže integrace nějaká data exportuje, uloží se tato data k testovacím datům. Buď jako soubory tak, jak jsou nakonfigurované v DataDistributorech a pokud nejde o soubory, ale exporty předávané přes API, uloží se podobně jako importy ve formátu JSON.
10. Všechna testovací data se automaticky pomocí SSH a rsync přenesou na testovací server a z produkčního serveru se odstraní.
11. Pokud by integrace v kterémkoliv bodě vykázala chybu, připravená testovací data budou zahozena, požadavek na vygenerování testovacích dat zůstane aktivní a administrátor bude informován o neúspěchu pomocí e-mailové zprávy.

Z toho vyplývá, že nová testovací data lze vygenerovat pouze při ostrém běhu integrace, a to z toho důvodu, aby se získaly exporty databáze a před a po provedení integrace včetně všech vstupních dat, která dané změny v databázi zapříčinila.

Problém v testování nastává ve chvíli, kdy je potřeba otestovat připojení na vzdálenou službu (FTPS, SFTP, API atd.) nebo připojení od klienta na vlastní API server. Pro tyto případy musí být možné se na testovací server připojit všemi způsoby, kterými integrační platforma disponuje. Jako příklad lze uvést připojení na FTPS. Pokud se mají připravit data pro integraci využívající připojení FTPS, musí být pozměněna metadata integrace při přenosu k testovacím datům tak, že bude nahrazena IP adresa klienta a autorizační údaje za IP adresu testovacího serveru a vlastní autorizační údaje. Zároveň se musí vstupní soubory integrace nahrát do adresáře, kam bude FTPS účet odkazovat. Během testování pak nebude docházet k připojení na FTPS ke klientovi, ale na vlastní testovací server, který poskytne data integraci stejným způsobem, jako by se připojovala ke klientovi. Pro každý typ připojení bude způsob trochu odlišný a jeho detailní návrh poměrně rozsáhlý, proto bude dále uveden pouze způsob testování s lokálními soubory.

Testování probíhá na vyžádání administrátora nebo vývojáře přes centrální administraci. Mělo by jít testovat jak jednu konkrétní integraci, tak zadat požadavek, aby se všechny integrace otestovaly po sobě, což může být vyvoláno jednoduše tím, že se u všech integrací nastaví

požadavek na otestování. Takové testování by bylo vhodné vždy před nasazením nové verze vzdělávacího systému nebo integrační platformy, aby došlo k ověření, že mají všechny integrace požadovaný výstup. Na testovacím serveru bude vytvořena jedna instance vzdělávacího systému s produkční nebo novou verzí, kterou je nutné otestovat a zároveň integrační platforma rovněž s produkční nebo novou verzi, která se má otestovat. Každých 5 minut bude na testovacím serveru spuštěn skript provádějící testování s následujícími operacemi:

1. Zkontroluje, jestli právě běží test některé z integrací pomocí záznamu v logu. Pokud test běží, zkontroluje, jestli neběží delší dobu, než je maximální doba běhu PHP skriptu na serveru. Pokud je doba delší, zaloguje k testu neúspěch a pokračuje dále. Jestliže je doba běhu testu kratší než maximum, skript se ukončí.
2. Načte z databáze první integraci, u které je nastaveno, že u ní má proběhnout test. Pokud žádnou nenajde, ukončí se.
3. Do databáze instance vzdělávacího systému naimportuje obsah databáze získaného během přípravy testovacích dat před spuštěním integrace.
4. Nastaví adresářovou strukturu instance LMS a nahraje do ní vstupní data a metadata integrace.
5. Provede integraci na testovací instanci LMS.
6. Porovná očekávaný stav databáze (exportovaný v přípravě metadat) s aktuálním stavem databáze vzdělávacího systému.
7. Pokud jsou výstupem integrace soubory, porovná je s těmi, které jsou v očekávaných datech.
8. Zaloguje konec testu a ukončí se.

Komplexní způsob testování by měl být implementován co nejdříve po nasazení integrační platformy, aby bylo bezpečné vydávat další aktualizace jak pro integrační platformu, tak pro vzdělávací systém. Vzhledem k tomu, že po převodu všech integrací pod integrační platformu se očekává minimálně 100 různých integrací, bude tento způsob testovat pravděpodobně všechny eventuality, které mohou během integrací nastat.

Kapitola 6
Závěr

V této kapitole je zhodnocení celého projektu integrační platformy. Zároveň jsou v ní uvedeny další možnosti rozšíření.

Celý projekt integrační platformy splnil očekávání. Jeho využívání se vzdělávacím systémem usnadnilo práci při vytváření nových integrací a jednoduché integrace jsou administrátoři schopni napsat sami bez pomoci vývojářů, a to pouze po jednoduchém zaškolení. Bloky EduBuddy se administrátoři naučili spíše kopírovat z již existujících integrací, protože vyjma několika málo parametrů jsou neměnné. Jelikož administrátoři ve společnosti ovládají základní operace v MySQL databázích, tak jsou pro ně DataMixer bloky srozumitelné a s využitím menší dokumentace, která popisuje, co může DataMixer obsahovat, jsou schopni jeho používání. Co by stálo za zvážení je přejmenovat *parse* parametr na *select*, aby zápis DataMixer bloku byl ještě více podobný zápisu dotazu v SQL databázích. Jediné, s čím je potřeba administrátorům během vytváření DataMixeru pomoci, je případ, kdy se musí napsat složitější vlastní funkce. S tímto problémem jim díky implementaci PHP funkcí může pomoci kterýkoliv vývojář a jedná se pouze o jednotky případů. Co je ovšem pro většinu administrátorů problém, tak jsou DataDistributor bloky, kde není problém v integrační platformě jako takové, ale v neznalosti administrátorů ohledně některých způsobů přenosu dat. V praxi to pak dopadá podobně jako EduBuddy bloky, kdy se administrátoři snaží si pomoci tím, že najdou existující integraci s podobným typem připojení a například pouze zamění IP adresu a autorizační údaje. V případě, že takový typ připojení nenajdou a neznají ho, poradí se s některým z vývojářů, který integracím rozumí. Tomuto by šlo zabránit pouze důkladným proškolením administrátorů, ke kterému v dohledné době dojde.

Jelikož byl dost důkladně proveden návrh integrační platformy včetně zápisu integrací do metadatového souboru, tak samotná implementace probíhala bez jakýchkoliv větších komplikací. Během implementace a lokálního testování byly objeveny pouze výkonnostní problémy, ale nebyl zde žádný problém co do způsobu provádění integrací. V rámci komplexního testování pak bylo objeveno několik drobných chyb vývojáře, vzniklých během implementace, například špatným použitím proměnné či různými překlepy.

Vytvořená integrační platforma splnila požadované zadání. Umožnila přenést odpovědnost za správu integrací z vývojářů na administrátory. Dále díky integrační platformě již není problém získat a transformovat téměř jakákoliv příchozí data klientů na očekávaná data pro vzdělávací systém. Jelikož v době psaní práce byly již převedeny všechny typy integrací, které jsou napojené na starou verzi vzdělávacího systému, splňuje integrační platforma i hlavní požadavek, aby do ní šly přepsat všechny existující integrace.

6.1 Možnosti rozšíření integrační platformy

Během vývoje bylo objeveno několik různých námětů, kterými by šla integrační platforma zdokonalit, aby práce s ní byla ještě snazší než teď. Některé tyto náměty již byly uvedeny, jako například automatické komplexní testování integrací v předchozí kapitole. Některé další jsou popsané na následujících řádcích.

6.1.1 Formulářový editor integrací

Jak již bylo uvedeno v zadání integrační platformy, tak nejzajímavější možností rozšíření je vytvoření formulářových prvků, skrze které by šla v centrální administraci doslova naklikat integrace. S využitím jednoho souboru k popisu metadat by se tento soubor velmi snadno generoval. Zásadně by to ulehčilo administrátorům práci s DataDistributory, protože by je to mohlo jednoduchými kroky dovést do správného cíle. V případě potřeby vytvoření integrace využívající například připojení na HTTP API server klienta by mohly kroky vypadat následovně:

1. Administrátor by v detailu instance klienta zvolil, že chce vytvořit novou integraci. Tím by vznikl nový záznam o integraci a vygenerovala by se prázdná kostra integračních metadat.

2. Vybral by, že chce načíst data od klienta, což by automaticky vytvořilo nový blok readDataDistributor.

3. V dalším kroku by pak zvolil ze select inputu, jakým způsobem chce data získávat (HTTP API – klient).

4. Po výběru způsobu by se zobrazily jen nutné položky, které musí vyplnit. V tomto případě by to byla URL adresa, HTTP metoda, vlastní hlavičky, vlastní tělo požadavku a opět select input, ve kterém by mohl zvolit typ autorizace.

5. Pokud by vybral typ autorizace, objevily by se další položky k vyplnění, které by se vázaly pouze k vybranému typu.

6. Následně by mohl tlačítkem "Test" vyzkoušet, zda se data tímto způsobem podařilo právě teď načíst.

7. V tento moment by mohl uložit readDataDistributor blok a vytvořit další blok jakéhokoliv typu. V případě, že by prošel test v předchozím bodě, mohla by mu rovnou centrální administrace ukázat položky, které jsou v přijatých datech, aby se administrátorovi lépe zpracovával DataMixer, kde by nemusel položky vypisovat, ale mohl by je vybírat ze select inputu.

Všechny výše uvedené kroky a inputy by mohly obsahovat jednoduché a heslovité nápovědy, aby administrátor přesně věděl, co vyplňuje.

6.1.2 Agregační funkce

Jelikož jsou DataMixer bloky v integrační platformě postaveny na podobné logice jako dotazování v MySQL, bylo by možné implementovat agregační funkce tak, že by se využívaly společně s parametrem *groupBy*. V aktuálně existujících integracích není nic takového potřeba, to však neznamená, že by někteří klienti nevyužili možnosti získávat i reporty, kde by viděli nejen poslední, ale i procentuálně nejlepší vyhodnocení kvalifikačního testu u kurzu. Podobných příkladů i mimo kvalifikační testy by šlo vymyslet více. Před implementací takové funkce by ovšem bylo vhodné oslovit potenciální zájemce z řad aktuálních klientů, zda by o takovou funkci měli zájem, aby nebyla implementována zbytečně.

6.1.3 Rozšíření Buddy bloků

Po uvedení integrační platformy do provozu byl zanedlouho vznesen dotaz, zda by nešlo tuto platformu použít i pro další systémy společnosti nabízející vzdělávací systém. V případě, že budou ostatní systémy podporovat napojení přes REST API stejně jako vzdělávací systém, není problém s nimi komunikovat stejným způsobem a pro usnadnění práce v metadatech pak mohou existovat podobné bloky sloužící jako aliasy k DataDistributor blokům, podobně jako EduBuddy bloky, aby administrátoři ke každému takovému systému nemuseli v každém DataDistributoru vyplňovat všechny parametry. Jeden z těchto systémů je ovšem v tomto směru trochu zastaralý a všechny integrace, které do něj směřují, musí podporovat konkrétní CSV formát. S takovým požadavkem mají někteří klienti problém, protože neumějí ze svých systémů vygenerovat CSV a ještě v naprosto přesném formátu. I v tomto případě však půjde využít middleware v podobě integrační platformy, který by získal data od klienta, přetransformoval je do požadovaného formátu a přes FTPS či SFTP nahrál na umístění, kde je bude systém očekávat. Pro takový případ by šlo opět použít nový Buddy blok, který by sloužil jako alias k FTPS nebo SFTP připojení, protože nikde není uvedeno, že tyto bloky musí pracovat pouze s API komunikací.

Integrační platforma bude tedy využita nejen ke vzdělávacímu systému, ale bez jakéhokoliv nutného zásahu i k ostatním systémům, které společnost nabízí a jediný zásah, ke kterému dojde bude vytvoření aliasů pro komunikaci s těmito systémy. Tím tato platforma zásadně přesahuje požadované zadání.

Literatura

[1] Okta: What is OAuth 2.0? [online]. [vid. 2023-10-05]. Dostupné z: https://auth0.com/intro-to-iam/what-is-oauth-2

[2] Group, T. P.: Supported Versions [online]. [vid. 2023-10-05]. Dostupné z: https://www.php.net/supported-versions.php

[3] Wikipedia: CSV [online]. [vid. 2023-10-05]. Dostupné z: https://cs.wikipedia.org/wiki/CSV

[4] mozilla.org contributors, I.: Working with JSON [online]. [vid. 2023-10-05]. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON

[5] Microsoft: XML pro začátečníky [online]. [vid. 2023-10-05]. Dostupné z: https://support.microsoft.com/cs-cz/office/xml-pro-zaÄDÃąteÄDnÃŋky-a87d234d-4c2e-4409-9cbc-45e4eb857d44

[6] Jakubová, V.: FTP, SFTP, SMB a další protokoly pro přenos souborů: který vybrat? [online]. [vid. 2023-10-05]. Dostupné z: https://www.master.cz/blog/ftp-sftp-smb-protokoly-pro-prenos-souboru-ktery-vybrat/

[7] Pichlík, R.: A REST [online]. [vid. 2023-10-05]. Dostupné z: https://dagblog.cz/a-rest-c5156313d79e

[8] Kumar, R.: Know how RESTful your API is: An Overview of the Richardson Maturity Model [online]. [vid. 2023-10-05]. Dostupné z: https://developers.redhat.com/blog/2017/09/13/know-how-restful-your-api-is-an-overview-of-the-richardson-maturity-model

[9] Kosek, J.: Využití webových služeb a protokolu SOAP při komunikaci [online]. [vid. 2023-10-05]. Dostupné z: https://www.kosek.cz/diplomka/html/websluzby

[10] mozilla.org contributors, I.: Base64 [online]. [vid. 2023-10-05]. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/Base64

[11] Wikipedia: Pretty Good Privacy [online]. [vid. 2023-10-05]. Dostupné z: https://cs.wikipedia.org/wiki/Pretty_Good_Privacy

[12] Zapier: How it works [online]. [vid. 2023-10-05]. Dostupné z: https://zapier.com/how-it-works

[13] WEBMETHODS [online]. [vid. 2023-10-05]. Dostupné z: https://www.predictiveanalyticstoday.com/webmethods/

[14] Lupčík, J.: Lekce 1 - Úvod do Laravel frameworku pro PHP [online]. [vid. 2023-10-05]. Dostupné z: https://www.itnetwork.cz/php/laravel/uvod-do-laravel-frameworku-pro-php

[15] Group, T. P.: curl_setopt [online]. [vid. 2023-11-04]. Dostupné z: https://www.php.net/manual/en/function.curl-setopt.php

[16] mozilla.org contributors, I.: HTTP authentication [online]. [vid. 2023-11-04]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication

[17] SAML 2.0 Bearer Assertion Flow for OAuth 2.0 [online]. [vid. 2023-11-04]. Dostupné z: https://help.sap.com/doc/saphelp_nw73ehp1/7.31.19/en-US/12/41087770d9441682e3e02958997846/content.htm

[18] MySQL LIKE Operator [online]. [vid. 2023-11-04]. Dostupné z: https://www.w3schools.com/mysql/mysql_like.asp

[19] MySQL GROUP_CONCAT() function [online]. [vid. 2023-11-04]. Dostupné z: https://www.w3resource.com/mysql/aggregate-functions-and-grouping/aggregate-functions-and-grouping-group_concat.php

[20] UNIX cron format [online]. [vid. 2023-11-21]. Dostupné z: https://www.ibm.com/docs/en/db2oc?topic=task-unix-cron-format

[21] Factory method for designing pattern [online]. [vid. 2023-11-21]. Dostupné z: https://www.geeksforgeeks.org/factory-method-for-designing-pattern/

[22] Wikipedia: GraphQL [online]. [vid. 2023-11-24]. Dostupné z: https://cs.wikipedia.org/wiki/GraphQL

Obsah přiloženého média

- `readme.txt` — stručný popis obsahu média
- `attachments` — přílohy
  - `class-diagram.png` — návrhový model tříd
  - `example1.json` — integrační metadata k prvnímu příkladu
  - `example1_courses.csv` — pomocný CSV soubor k prvnímu příkladu
  - `example2.json` — integrační metadata k druhému příkladu
  - `example2_courses.csv` — pomocný CSV soubor k druhému příkladu
  - `example3.json` — integrační metadata ke třetímu příkladu
  - `example4-1d.json` — integrační metadata ke třetímu příkladu pro denní export
  - `example4-all.json` — integrační metadata ke třetímu příkladu pro kompletní export
  - `uniInputData.json` — vstupní data pro unit testy
  - `uniMetadata.json` — univerzální metadata pro unit testy
- `src`
  - `thesis` — zdrojová forma práce ve formátu LaTeX
- `text` — text práce
  - `thesis.pdf` — text práce ve formátu PDF