FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Otevřená data Pražské integrované dopravy s historií
Student:	Alexander Žibrita
Vedoucí:	Ing. Tomáš Pecka
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Pražská integrovaná doprava je dopravní systém hromadné dopravy osob v Praze a Středočeském kraji.
Informace o jizdních řádech a aktuální poloze vozidel jsou publikovány jako open data, avšak bez možnosti jít do historie.
Cílem práce je navrhnout, implementovat a otestovat řešení, které bude umět udržovat historii těchto dat po jistý časový úsek zpětně a umožňovat nad těmito daty provádět jednoduché statistické a přehledové dotazy.

Seznamte se s otevřenými daty poskytovanými pro Pražskou integrovanou dopravu (PID) a datovou platformou Golemio [1].
Analyzujte poskytovaná data a navrhněte řešení, které bude získávat a udržovat data o historii spojů vypravených v rámci PID ve vhodných datových strukturách.
Pro jednotlivé spoje evidujte minimálně údaje o plnění jízdního řádu (zpoždění) a informace o vozidlu a dopravci.

Navrhněte REST API, které bude umožňovat provádět dotazy nad uloženými daty a vytvořte v něm několik ukázkových endpointů (např. seznam spojů na které bylo nasazeno vozidlo s konkrétním evidenčním číslem, průměrná zpoždění daného spoje v jednotlivých zastávkách, apod.).
Navrhněte jednoduchou webovou aplikaci, která bude zobrazovat data poskytovaná tímto API serverem.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 20. února 2023 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Proveďte průzkum vhodných technologií a zvolte vhodné techonologie pro implementace jednotlivých částí.
Kód vhodně otestujte automatickými testy a zdokumentujte jednotlivé komponenty.
Výsledné řešení nasaďte do prostředí poskytnutém vedoucím práce.

[1] https://pid.cz/o-systemu/opendata/

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 20. února 2023 v Praze.

Bakalářská práce

OTEVŘENÁ DATA PRAŽSKÉ INTEGROVANÉ DOPRAVY S HISTORIÍ

Alexander Žibrita

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Tomáš Pecka
10. května 2023

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Na tomto místě bych rád poděkoval vedoucímu práce Ing. Tomáši Peckovi za velmi důkladné vedení celým procesem tvorby závěrečné práce, ochotu při řešení problémů a neustálou dostupnost ke konzultacím. Dále bych rád poděkoval vývojářům z datové platformy Golemio za vstřícnost při komunikaci ohledně využití jejich služeb. V neposlední řadě bych rád poděkoval své rodině za neutuchající obrovskou podporu v rámci celého bakalářského studia.

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, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 10. května 2023 Alexander Žibrita

Abstrakt

Tato bakalářská práce se zabývá vývojem softwaru sloužícího k zpracovávání a uchovávání otevřených dat Pražské integrované dopravy a následnému poskytování statistických přehledů nad těmito daty. Analýza řeší způsoby poskytování dat hromadné dopravy a některé aplikace postavené nad nimi. Praktická část je zaměřena na návrh a implementaci samotné aplikace s použitím vybraných technologií a je založena na mikroservisní architektuře. Hlavním výstupem práce je webová stránka umožňující přehledové dotazy. V závěru práce jsou nastíněny možnosti dalšího rozšíření.

Klíčová slova otevřená data, hromadná doprava, Pražská integrovaná doprava, uchovávání dat, statistické přehledy, webová aplikace, Java, Spring framework, PostgreSQL, TypeScript

Abstract

This bachelor thesis deals with a development of software for processing and storing open data of Prague Integrated Transport and provision of data statistical overviews. The analysis addresses the methods of providing public transport data and some applications built on top of them. The practical section focuses on the design and implementation of the application using appropriate technologies and is based on a microservice architecture. The key output is a web page providing basic reports. The thesis concludes by naming the possibilities for further extension.

Keywords open data, public transport, Prague Integrated Transport, data storage, statistical reports, web application, Java, Spring Framework, PostgreSQL, TypeScript

Seznam zkratek

API	Application Programming Interface
CI/CD	Continuous Integration and Continuous Delivery
CSS	Cascading Style Sheets
DTO	Data Transfer Object
ER	Entity-Relationship
ETL	Extract-Transform-Load
GTFS	General/Google Transit Feed Specification
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IoC	Inversion of Control
JDBC	Java Database Connectivity
JPA	Jakarta Persistence API
JPQL	Java Persistence Query Language
JSON	JavaScript Object Notation
MVC	Model-View-Controller
ORM	Object Relational Mapping, Objektově relační mapování
PB	Protocol Buffer
PID	Pražská integrovaná doprava
REST	Representational State Transfer
SQL	Structured Query Language
UC	Use Case
UI	User Interface
VPN	Virtual Private Network
XML	Extensible Markup Language

Úvod

Služby Pražské integrované dopravy (dále jen PID) každodenně využívá značné množství lidí k přepravě v rámci samotné Prahy nebo ze stále se rozrůstající sítě pokrytí okolních měst. Zvláště v dnešní době představuje hromadná doprava jednu z nejoblíbenějších forem přepravy, ať už z důvodu ekonomického, environmentálního nebo spolehlivosti.

PID v současné chvíli poskytuje svá data veřejnosti v několika podobách – denně aktualizovaná data o jízdních řádech a zastávkách, roční statistiky nebo data o aktuálně vypravených spojích v reálném čase. Právě poslední zmíněná data jsou předmětem této bakalářské práce. Nad daty spojů v reálném čase je postaveno několik aplikací (například Mapa PID nebo PID Lítačka), nicméně žádná z nich neumožňuje uživatelům zpětné nahlédnutí do historie pro statistické účely.

Právě absence možnosti zpětně dohledat data o vypravených spojích byla mou motivací pro zvolení tématu práce a její následné vypracování. Výstupem práce bude sada několika komponent, které dohromady zajistí sběr, uchování dat a jejich následné zpracování a poskytnutí uživatelům v rámci základních statistických přehledů – průměrné zpoždění linek v rámci zvoleného časového úseku, procentuální pokrytí linky vozidly s bezbariérovým přístupem a další.

Výsledek práce bude užitečným ukazatelem nejen pro samotné cestující, jelikož může poukázat na některé opakující se vzory v každodenním fungování hromadné dopravy. V některých případech může být efektivním pomocníkem pro plánování tras. V neposlední řadě přispěje do nemalé komunity nadšenců do hromadné dopravy a poslouží jako užitečný nástroj pro získávání zajímavých statistických informací, které nejsou v současné chvíli dostupné.

Kapitola 1

Cíle práce

Hlavním cílem bakalářské práce je návrh a implementace softwaru uchovávajícího data o pražské hromadné dopravě s možností náhledu do jejich historie pro statistické účely.

Cílem teoretické části práce je analyzovat konvenční způsoby poskytování dat o hromadné dopravě a srovnat je s daty poskytovanými společností PID. Zmíněny budou také současné aplikace postavené nad těmito daty a jejich využitelnost v praxi. Dalším z cílů je analýza funkčních požadavků na jednotlivé komponenty a vytvoření vhodných modelů softwarového inženýrství.

Cílem praktické části je návrh jednotlivých komponent na základě výše zmíněných funkčních požadavků. Dále budou popsány zvolené technologie pro implementaci včetně argumentace jejich finálního výběru. V neposlední řadě bude následovat samotná implementace včetně testování, nasazení a závěrečné diskuse nad dalším možným rozvojem.

Kapitola 2
Analýza

Tato kapitola pojednává o možnostech poskytování dat veřejné dopravy. Věnuje se především popisu zavedených konvenčních způsobů uchovávání a poskytování těchto dat, ze kterých vychází i data poskytovaná PID. Následně jsou v kapitole vyzdviženy některé odchylky od tohoto konvenčního formátu. V neposlední řadě jsou zmíněny některé aplikace postavené nad otevřenými daty hromadné dopravy.

2.1 Obecný formát dat veřejné dopravy

2.1.1 GTFS formát

General Transit Feed Specification[1], někdy také Google Transit Feed Specitifation, (dále jen GTFS) je zavedený konvenční formát pro data o veřejné dopravě. Formát byl poprvé představen v roce 2006 po spolupráci společností TriMet a Google. Specifikace formátu se dělí na dvě základní části – GTFS Static a GTFS Realtime.

GTFS Static[2] slouží pro uchovávání tzv. statických dat, tedy dat, která jsou relevantní a neměnná delší časové období. V praxi se jedná o zastávky, jízdní řády nebo jednotlivé trasy včetně souvisejících geografických údajů. Tyto údaje jsou poskytovány v tzv. GTFS datasetu, což je sada předem definovaných souborů. Tento dataset je popsán v tabulce 2.1.

GTFS Realtime[3] naopak stanovuje formát pro data relevantní v reálném čase. V současné chvíli jsou definovány tři typy informací:

Trip updates: predikce zpoždění spojů, zrušení výjezdu a změny na trasách,

Service alerts: přesunutí zastávky, různé nepředvídatelné události ovlivňující stanici, trasu nebo celou síť,

Vehicle positions: informace o vozidlech včetně jejich současné lokace.

2.1.2 Protocol Buffer

GTFS Realtime data jsou poskytována ve speciálním formátu založeném na Protocol bufferu.

Název	Povinnost	Význam
`agency.txt`	ano	dopravní společnosti poskytující služby v tomto datasetu
`stops.txt`	ano	zastávky včetně stanic a vchodů do stanic
`routes.txt`	ano	linky, linka je tvořena několika výjezdy (tripy) v rámci dne
`trips.txt`	ano	výjezdy v rámci linky, jeden výjezd je tvořen sekvencí dvou a více zastávek v určitém časovém intervalu
`stop_times.txt`	ano	čas příjezdu a odjezdu na jednotlivé zastávky v rámci výjezdu
`calendar.txt`	podmíněně	určuje, které dny v týdnu daná služba operuje, dáno v určitém období (datum od – datum do), může být vynechán, pokud všechna data obsahuje `calendar_dates.txt`
`calendar_dates.txt`	podmíněně	určen pro dočasné výjimky oproti stálému stavu z `calendar.txt`, pokud je `calendar.txt` vynechán, musí být obsažen a poskytovat všechna data
`fare_attributes.txt`	ne	informace o jízdném na linkách
`fare_rules.txt`	ne	pravidla uplatňování jízdného
`shapes.txt`	ne	sada souřadnic mapující cestu vozidla v rámci linky
`frequencies.txt`	ne	Specifikace jízdních řádů pomocí fixních časových intervalů mezi jednotlivými výjezdy.
`transfers.txt`	ne	specifikace přestupů na přestupních bodech mezi linkami
`pathways.txt`	ne	specifikace cest spojující určitá místa (vstup, nástupiště...) v rámci stanic
`levels.txt`	ne	definice pater v rámci stanice
`feed_info.txt`	podmíněně	metadata samotného datasetu, povinná v případě zahrnutí souboru `translations.txt`
`translations.txt`	ne	překlady dat v datasetu
`attributions.txt`	ne	atribuce dat v datasetu

Tabulka 2.1 Soubory GTFS Datasetu

Protocol buffer[4] (dále také PB) je mechanismus vyvinutý společností Google sloužící k serializaci dat, podobně jako JSON nebo XML. Protocol buffer se od těchto formátů liší ve způsobu uchování dat, jelikož je uchovává v binární podobě, tudíž jsou pro člověka nečitelná. Výhodou takového přístupu je menší objemnost a rychlejší zpracování těchto dat.

Protocol buffer je platformově a jazykově nezávislý. Struktura poskytovaných dat se definuje ve speciálních konfiguračních souborech s koncovkou `.proto`. V nich se uvedou názvy jednotlivých datových struktur a definují jejich atributy včetně povinnosti a datových typů. Z těchto konfiguračních souborů vygeneruje poskytnutý proto kompilátor zdrojový kód ve zvoleném programovacím jazyce. Kromě oficiálně podporovaných jazyků (Java, C++, C#, Python...) existují open-source řešení pro nemalou část nepodporovaných jazyků (Swift, Scala, JavaScript a další). Takto vygenerovaný kód se využije k manipulaci s daty v PB formátu. Kód obsahuje základní přístupové metody (gettery, settery) pro jednotlivé atributy a dále metody pro obousměrný převod dat z definované struktury do bytů.

Google poskytuje oficiální konfigurační soubor `gtfs-realtime.proto`[1], který obsahuje definice struktur zmiňovaných v kapitole 2.1.1 (Trip updates, Service alerts, Vehicle positions). Kromě předpřipravených atributů jsou zde vyhrazené tzv. extensions namespaces, které umožňují vývojářům či soukromým organizacím třetích stran libovolně přidávat vlastní specifikace.

Pro práci s GTFS realtime daty se tedy využije tento dodaný konfigurační soubor k vygenerování potřebného zdrojového kódu ve vybraném programovacím jazyce.

2.2 Otevřená data PID

2.2.1 Datové sady

Na oficiálních webových stránkách PID[5] lze v sekci Otevřená data PID přímo získat několik datových sad. Konkrétně se jedná o jízdní řády ve formátu GTFS Static, dále seznam zastávek a prodejních míst ve formátech JSON a XML.

Některé další datasety lze získat z oficiální stránky Otevřených dat hlavního města Prahy[6], konkrétně například geografická data v různých formátech (GeoJSON, Shapefile a jiné) nebo roční statistiku PID. Poslední roční statistika byla vydána v roce 2019[2] a obsahuje celkové počty dopravců, linek, zastávek nebo průměrné počty různých typů spojů dle jednotlivých dní v týdnu.

Pro účely této práce je nicméně nejrelevantnější způsob získávání dat prostřednictvím API poskytovaným pražskou datovou platformou Golemio[7].

API (Application Programming Interface) je řešení zprostředkovávající komunikaci dvou či více aplikací pomocí definovaných pravidel a protokolů[8]. Webové API typicky slouží k výměně informací pomocí HTTP požadavků zaslaných na vystavené endpointy (koncové body). Každý koncový bod je jednoznačně identifikován svojí cestou a použitou HTTP metodou (GET, POST, DELETE...). Pro účely získávání dat jsou typicky využívány pouze koncové body s metodou GET.

2.2.2 Golemio API

„*Golemio je soubor technických nástrojů pro integraci, ukládání, vizualizaci a poskytování dat.*“[7] Datová platforma Golemio se věnuje datům z oblasti městských systémů,

[1] Dostupný na: `https://github.com/google/transit/tree/master/gtfs-realtime`

[2] Dostupná na: `https://opendata.praha.eu/datasets/https%3A%2F%2Fapi.opendata.praha.eu%2Flod%2Fcatalog%2F8bd15eda-6e06-4f50-a826-4e58af8f476b`

především z hlavního města Prahy.

V rámci spolupráce datové platformy Golemio a PID bylo vyvinuto rozhraní[9], které poskytuje veřejnosti real-time data ze všech vozidel hromadné dopravy. Toto zmíněné rozhraní je provozováno pod názvem Golemio API a v současné chvíli poskytuje data ve 4 sekcích:

GTFS: GTFS Static data ve formátu JSON,

RealTime Vehicle Positions: data o spojích v reálném čase poskytována ve formátu JSON,

GTFS RealTime Vehicle Positions: data o spojích v reálném čase poskytována prostřednictvím Protocol Bufferů a JSON[3],

PID Departure Boards: odjezdové tabule pro jednotlivé evidované zastávky.

2.2.2.1 GTFS endpointy

Sekce GTFS obsahuje několik endpointů poskytujících zmiňovaná statická GTFS data ve formátu JSON. Tato statická data jsou aktualizována denně, zpravidla v ranních hodinách mezi 5:15 a 5:45. Ve výjimečných situacích je potřeba data opravit, děje se tak v odpoledních hodinách mezi 13:00 a 13:30.

Data se v některých případech odchylují od původní definice formátu GTFS Static rozebrané v kapitole 2.1.1. Jednotlivé endpointy a rozdíly dat oproti původní specifikaci popisuje tabulka 2.2. Endpointy pokrývají všechny povinné soubory GTFS Static datasetu s výjimkou jednotlivých přepravců (`agency.txt`), kteří jsou poskytováni v real-time datech.

Pro tuto práci jsou kromě real-time pozic vozidel důležitá i tato statická data, aby uživatel měl k dispozici větší kontext, tedy například bylo možné z identifikátoru (zastávky/linky/výjezdu) dohledat konkrétní detailní data (název zastávky v čitelné podobě...) nebo porovnat zpoždění jednotlivých spojů s pravidelnými jízdními řády.

2.2.2.2 Feature Point struktura

Jak již bylo zmíněno v předchozí kapitole u GTFS Stops, v datech je využívána obalovací struktura zvaná `Feature Point`[9]. Ta obsahuje tři atributy:

`geometry`: geografické údaje (souřadnice),

`properties`: samotná data, která tato struktura obaluje (např. GTFS Stop),

`type`: typ (zpravidla hodnota „Feature“).

Tato struktura je používána pro údaje obsahující nějaký geografický údaj, který je dodán prostřednictvím zmíněného atributu geometry. Zpravidla je žádoucí dodat více dat najednou, z toho důvodu jsou jednotlivé `Feature Point` dodávány v instanci `FeatureCollection`, která obsahuje seznam těchto `Feature Point` a dodatečný atribut `type` obsahující textovou hodnotu „FeatureCollection“.

[3] Není garantována dostupnost[10].

Cesta	Odpovídající GTFS Static	Rozdíly
`/gtfs/services`	`calendar.txt`	Kromě změny v názvu soubor obsahuje dodatečný atribut `last_modify`, který udává poslední změnu v dané službě.
`/gtfs/routes`	`routes.txt`	Soubor obsahuje dodatečné atributy `is_night` (indikátor nočního spoje), `is_regional` (indikátor regionálního spoje), `is_substitute_transport` (indikátor náhradního spoje) a znovu `last_modify`.
`/gtfs/trips`	`trips.txt`	Soubor obsahuje dodatečné atributy `exceptional` (indikátor výjimečného spoje) a `last_modify`.
`/gtfs/stops`	`stops.txt`	Jediným rozdílem je obalení samotné GTFS Stop struktury do tzv. `Feature Point`. Tato struktura je popsána v kapitole 2.2.2.2.
`/gtfs/stoptimes/{id}`	`stop_times.txt`	Soubor obsahuje dodatečné atributy `arrival_time_seconds`, `departure_time_seconds`, které udávají čas příjezdu/odjezdu v sekundách od 00:00:00, a `last_modify`.

Tabulka 2.2 Golemio API GTFS endpointy

2.2.2.3 RealTime Vehicle Positions endpointy

Tato sekce obsahuje endpoint `/vehiclepositions`, který vrací výše popsanou kolekci `FeatureCollection` a nabízí nemalé množství parametrů, prostřednictvím kterých lze filtrovat vrácená data nebo je naopak obohatit o dodatečné informace. Kromě tohoto endpointu obsahuje jeho variantu `/vehiclepositions/{gtfsTripId}`, který na základě zadaného identifikátoru výjezdu (tripu) vrátí pouze jediný dotazovaný `Feature Point`.

`Feature Point` v tomto případě obaluje strukturu `Vehicleposition Compound`, která obsahuje následující atributy:

`trip:` struktura `Vehicleposition Trip` obsahující údaje o konkrétním výjezdu (identifikátor výjezdu, obsluhující dopravce, obsluhující vozidlo a informace o něm a další podobné informace),

`last_position:` struktura `Vehicleposition Position` obsahující údaje o posledním nahlášeném stavu z vozidla (poslední projetá zastávka, následující zastávka, počet ujetých kilometrů a především aktuální zpoždění a zpoždění na příjezdu a odjezdu z poslední zastávce),

`all_positions:` všechny dosud nahlášené stavy vozidla od začátku výjezdu, vráceno jako kolekce (`Feature Collection`) `Vehicleposition Position` obalených v objektu `Feature Point`.

Pomocí parametrů endpointu lze filtrovat konkrétní linky (routes), případně omezit počet vracených výsledků. Data je dále možné příznakem `includeNotTracking` obohatit o vozidla, která nejsou aktuálně na žádném výjezdu (jsou před ním nebo již proběhl), a příznakem `includePositions` obohatit o výše popsanou sekci `all_positions`, která v opačném případě vrací prázdný seznam. Dle vypozorovaných dat a komunikace s vývojáři[10] příznak `includeNotTracking` eviduje spoje přibližně hodinu před výjezdem a 10 minut po ukončení výjezdu.

Při zaslání HTTP požadavku s příznakem `includePositions` bez limitování počtu spojů API vrací odpověď 500 (Internal Server Error), tudíž se ukázalo jako klíčové příznak používat v kombinaci s parametry na omezení počtu vrácených dat (`limit`) a odsazením (`offset`). Jinými slovy, je potřeba daty stránkovat a zpracovávat po částech, aby je API zvládlo v pořádku zpracovat a poskytnout.

2.2.2.4 GTFS RealTime Vehicle Positions endpointy

Tato sekce obsahuje právě 3 endpointy, které vracejí data ve výše zmíněném formátu Protocol Buffer, tedy s příponou `.pb`:

- `/vehiclepositions/gtfsrt/trip_updates.pb`
- `/vehiclepositions/gtfsrt/vehicle_positions.pb`
- `/vehiclepositions/gtfsrt/alerts.pb`

Endpoint vracející Trip updates obsahuje informace o současných výjezdech, nicméně pro účely této práce se tato data ukázala jako nerelevantní, jelikož Trip updates slouží především jako predikce zpoždění. Práce nicméně potřebuje pracovat s reálnými daty současnými, případně minulými. Definice Trip updates dovoluje vracet informace o zpožděních v minulých zastávkách, nicméně ne povinně. Endpoint v současné chvíli vrací data k již projetým zastávkám, nicméně k nim pouze nakopíruje aktuální zpoždění, tudíž se historická data mění dle aktuálního zpoždění, což je účely práce nevhodné.

Prostřednictvím e-mailové komunikace s vývojáři datové platformy Golemio[10] jsem si ověřil, že v současné chvíli je vzato poslední známé zpoždění a to nakopírováno ke všem projetým zastávkám. Vývojáři nicméně plánují revizi tohoto GTFS Realtime feedu[11], nicméně pro kontext této práce bude nutné použít endpoint popsaný v předchozí kapitole, jelikož ten obsahuje pravdivá zpoždění v projetých zastávkách.

2.3 Licencování dat

Data dostupná z webové stránky PID jsou opatřena licencí CC BY[12], tedy volně dostupná k dalšímu šíření pod podmínkou uvedení autora.

Golemio API nestanovuje žádná omezení[13] na další použití poskytovaných datových sad (nekomerční i komerční využití).

2.4 Současné projekty nad otevřenými daty hromadné dopravy

Nad otevřenými daty hromadné dopravy v současné chvíli existuje několik aplikací, některé z nich budou krátce zmíněny v následujících podkapitolách. Nicméně jak již bylo zmíněno v úvodu této práce, žádná z nich se nezabývá uchováváním veřejně dostupných provozních dat PID pro statistické účely.

První dvě zmíněné aplikace využívají data PID, následující dvě se zabývají daty o vlakové dopravě.

2.4.1 Mapa PID

Mapa PID[14] je webová aplikace využívající ke svému fungování právě data zpracovávaná datovou platformou Golemio. Mapa PID zobrazuje aktuální polohy všech spojů vykreslené do mapy, umožňuje si jednotlivé spoje detailněji zobrazit a zjistit informace o vozidle, zpoždění v zastávkách a současné trase. Dále je možné spoje na mapě filtrovat a zobrazit pouze některé z nich. Pokud vozidlo ukončí svůj výjezd a již neodesílá svá data, není na mapě dále zobrazováno, Mapa PID tedy slouží jako ukazatel aktuálního stavu, ne historického. Výjimkou jsou vozidla před výjezdem, ovšem za podmínky že právě neobsluhují jiný spoj.

2.4.2 PID Lítačka

PID Lítačka[15] je mobilní aplikace umožňující nákup jízdenek, hledání možných tras mezi zastávkami (včetně přestupů, doby trvání atd.) a s tím související zobrazení jízdních řádů. Pokud je vyhledán právě aktivní spoj, aplikace zobrazuje i jeho aktuální zpoždění v řádu celých minut.

2.4.3 Babitron

Babitron[16] je webová aplikace zobrazující současné polohy vlaků Českých drah na mapě. Pro tuto práci je nicméně zajímavější zobrazení historie zpoždění jednotlivých vlakových spojů ve všech projetých zastávkách. Na stránce je možné vyplnit číslo vlaku a následně zvolit rozsah zpětné evaluace ve dnech. Stránka následně zobrazí jednotlivé zastávky daného spoje včetně minimálního, maximálního a průměrného zpoždění a četnost zpoždění v různých intervalech (například do 5 minut, 5 – 10 minut...).

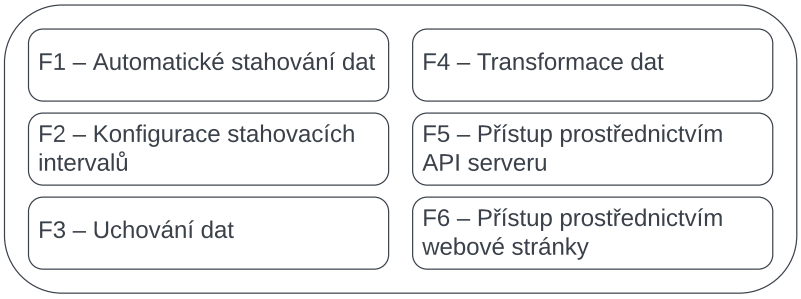
Tento web zmiňuji především jako inspiraci pro vlastní práci, jak by se ve výsledku mohly zobrazovat historické údaje spojů.

2.4.4 RtView

RtView[17] je webová aplikace poskytující statistické přehledy o vlakové hromadné dopravě. Na stránce je možné zvolit konkrétní spoj a časový rozsah pro agregaci jeho statistik. Po zvolení se zobrazí všechny absolvované výjezdy daného spoje ve zvoleném intervalu včetně průměrného zpoždění v jednotlivých stanicích a vizualizace prostřednictvím grafu. Tento web je taktéž zmíněn pro inspiraci výsledných statistických přehledů.

2.5 Funkční požadavky

Funkční požadavky představují jednotlivé žádané funkcionality, které má vyvíjený systém obsahovat. Výčet jednotlivých funkčních požadavků znázorňuje diagram 2.1.



Obrázek 2.1 Diagram funkčních požadavků

F1 – Automatické stahování dat: Aplikace bude automaticky získávat data z Golemio API serveru. Jednou denně (ve stanovenou hodinu) proběhne synchronizace GTFS Static údajů, GTFS Realtime údaje budou synchronizovány neustále ve stanovených časových intervalech.

F2 – Konfigurace stahovacích intervalů: Administrátor aplikace bude moci v konfiguraci definovat čas stažení GTFS Static údajů a také časový interval mezi synchronizacemi GTFS Realtime dat.

F3 – Uchování dat: Všechna získaná data budou uchovávána pro zpětnou trasovatelnost.

F4 – Transformace dat: Pro účely následného poskytování uživatelům budou surová uložená data transformována do čitelnějších struktur.

F5 – Přístup prostřednictvím API serveru: K datům bude možné přistoupit prostřednictvím HTTP požadavků na API server.

F6 – Přístup prostřednictvím webové stránky: K datům bude možné přistoupit prostřednictvím interaktivní webové stránky.

2.6 Nefunkční požadavky

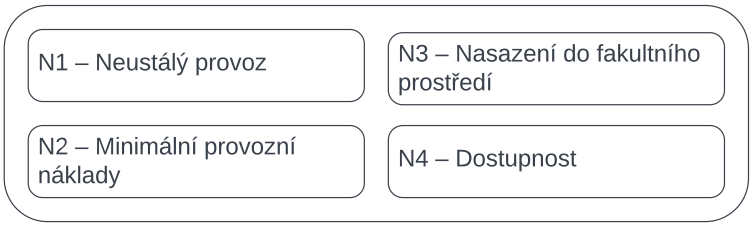
Nefunkční požadavky popisují ostatní nároky na systém nepokryté v rámci funkčních požadavků. Typicky se jedná o nároky výkonnostní, prostředkové nebo na dostupnost. Zachycené nefunkční požadavky znázorňuje diagram 2.2.

N1 – Neustálý provoz: Aplikace poběží v režimu 24/7 (tedy 24 hodin 7 dní v týdnu).

N2 – Minimální provozní náklady: Implementační technologie budou zvoleny s cílem minimalizovat provozní náklady, například použitím open source projektů.

N3 – Nasazení do fakultního prostředí: Výsledná aplikace bude nasazena do prostředí poskytnutého fakultou.

N4 – Dostupnost: Aplikace bude veřejně dostupná.



Obrázek 2.2 Diagram nefunkčních požadavků

2.7 Případy užití

Případy užití popisují jednotlivé možné scénáře používání softwaru určitými uživateli. Funkční požadavek se typicky skládá z několika částí[18]:

účastníci: Pro každý případ užití je nutné definovat alespoň jednoho účastníka, který bude daný případ vykonávat. Pod účastníkem se rozumí libovolná entita interagující se systémem, může se tedy jednat o lidského uživatele, externí systém nebo zařízení.

předpoklady: Předpoklady mohou stanovovat počáteční podmínky stavu systému, bez jejichž splnění není možné daný případ užití provést. Typicky se může jednat například o předchozí přihlášení uživatele.

průběh událostí: Chronologický popis jednotlivých kroků, které nastávají při samotné interakci účastníků se systémem.

koncové podmínky: Koncové podmínky mohou stanovovat stav systému, ve kterém se má nacházet po úspěšném vykonání průběhu všech událostí.

2.7.1 Stanovené případy užití

Stanovené případy užití jsou znázorněny pomocí diagramu případů užití 2.3.

UC1: Automatická synchronizace GTFS Static

účastník: čas

předpoklady: Momentální čas odpovídá stanovenému času synchronizace (v konfiguraci) a v současný den synchronizace těchto dat ještě neproběhla.

průběh: Systém se automaticky připojí ke Golemio API a pomocí HTTP požadavků postupně získá GTFS Static data. Takto získaná data systém zpracuje do připravených struktur a uloží do databáze.

koncové podmínky: Data jsou trvale uložena v databázi a neprobíhají žádné další HTTP požadavky na GTFS Static endpointy Golemio API.

UC2: Automatická synchronizace GTFS Realtime

účastník: čas

předpoklady: Od poslední synchronizace GTFS Realtime uběhl stanovený časový interval a současně neprobíhá předchozí synchronizace.

průběh: Systém se automaticky připojí ke Golemio API a pomocí HTTP požadavků získává GTFS Realtime data. Získaná data přetransformuje do úložných struktur a zapíše do databáze.

koncové podmínky: Data jsou trvale uložena v databázi a neprobíhají žádné další HTTP požadavky na GTFS Realtime endpointy Golemio API.

UC3: Parametrizace synchronizace

účastník: administrátor

průběh: Administrátor v konfiguračním souboru stanoví časové intervaly pro datové synchronizace a uloží. Systém se přenasadí.

koncové podmínky: Systém má aktualizované hodnoty časových intervalů, podle kterých řídí následující synchronizace.

UC4: Přístup k datům prostřednictvím API

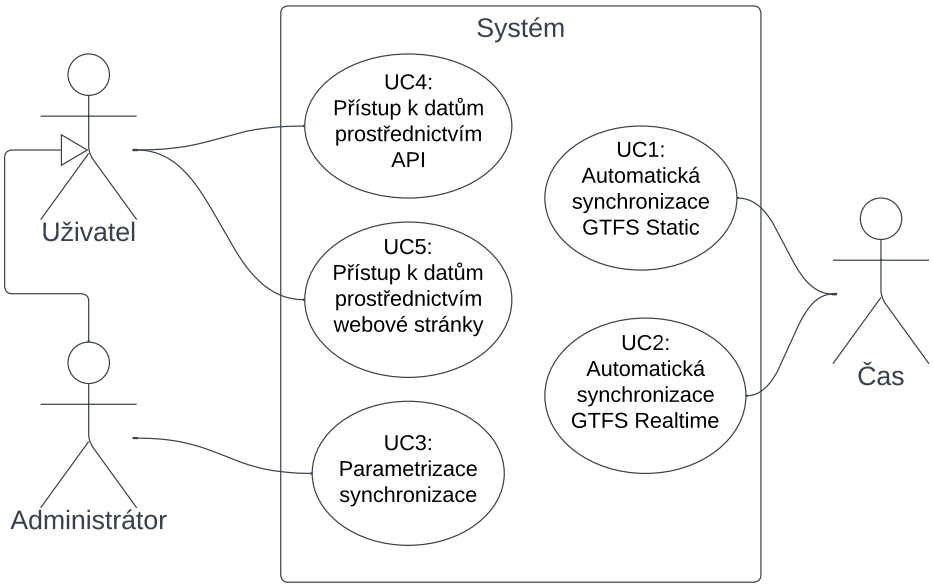
účastník: uživatel

průběh: Uživatel pošle na API HTTP požadavek. API zaslaný požadavek vyhodnotí a v případě chybného vyplnění uživatele upozorní. V případě úspěchu API poskytne uživateli žádaná data z databáze. Uživatel může zaslat libovolný další HTTP požadavek.

UC5: Přístup k datům prostřednictvím webové stránky

účastník: uživatel

průběh: Uživatel přistoupí na interaktivní webovou stránku a pomocí navigace vybere požadovaný typ dat. Webová stránka získá data komunikací s API, v případě neúspěchu je uživatel upozorněn. Získaná data jsou poskytnuta a vizualizována na stránce. Uživatel může dále libovolně pokračovat v navigaci na stránce a zobrazování dalších dat.



Obrázek 2.3 Diagram případů užití

2.7.2 Průnik s funkčními požadavky

Případy užití úzce souvisí se stanovenými funkčními požadavky, jelikož se také zaměřují na funkcionality systému. V tabulce 2.3 je zachyceno pokrytí jednotlivých případů užití předem stanovenými funkčními požadavky. Obecně platí, že každý případ užití by měl být pokryt alespoň jedním funkčním požadavkem (pokud systém danou funkcionalitu nemá, nebylo by možné modelovat případ užití s jejím využitím) a naopak (pro ověření, zda každý funkční požadavek bude mít své využití).

	F1	F2	F3	F4	F5	F6
UC1	✓		✓			
UC2	✓		✓			
UC3		✓				
UC4				✓	✓	
UC5				✓		✓

Tabulka 2.3 Průnik případů užití s funkčními požadavky

Kapitola 3
Návrh

Tato kapitola pojednává o návrhu softwaru na základě funkčních požadavků definovaných v předchozí kapitole. Popisuje zvolenou architekturu softwaru a pro každou jeho část konkrétně jmenuje použité postupy a technologie včetně zdůvodnění jejich finálního výběru.

3.1 Architektura

Vzhledem k požadované povaze výsledného softwaru jsem se rozhodl pro jeho rozdělení do několika částí dle jejich hlavní funkcionality:

Databáze sloužící k uchovávání získaných dat a jejich následnému zpracování a předpřipravení pro odběr.

Synchronizační komponenta (nebo loader) sloužící k získávání dat od třetích stran, jejich transformaci do navržených struktur a následnému uložení do databáze.

API (rozhraní) navržené poskytující uložená data a statistické přehledy uživatelům prostřednictvím několika endpointů.

Interaktivní webová stránka komunikující s API komponentou a vizualizující data od ní získaná.

Detailnějšímu popisu a návrhu jednotlivých zmíněných komponent včetně jejich vzájemné komunikace se věnují následující kapitoly.

Při návrhu byly zvažovány dvě klíčové architektury softwaru: monolitická a mikroservisní[19].

3.1.1 Monolitická architektura

Monolitická architektura je způsob navržení aplikace spočívající v umístění veškerých funkcionalit softwaru a jeho zdrojového kódu do jednoho projektu.

Hlavní výhodou monolitického návrhu je minimalizace závislostí na ostatních aplikacích a snazší správa nasazení, jelikož je potřeba nasadit právě jednu aplikaci. Tento

přístup je výhodný především v raných fázích vývoje, protože není nutné členit zdrojový kód do několika komponent a je možné celou aplikaci nasadit najednou.

Nicméně v pozdějších fázích vývoje se tento způsob ukáže jako nevýhodný, jelikož při zanesení libovolné změny je potřeba přenasadit kompletně celou aplikaci, což při větších projektech vede k vyšší časové a prostředkové zátěži. Jelikož jednotlivé funkcionality nejsou odděleny, může selhání jedné z nich ovlivnit chod celé aplikace. Monolitické aplikace se také obtížněji rozšiřují právě kvůli vzájemné provázanosti kódu.

3.1.2 Mikroservisní architektura

Mikroservisní architektura je založena na rozdělení jednotlivých funkcionalit do několika vzájemně komunikujících komponent, jejichž vzájemná provázanost by měla být minimální. Právě kvůli nízké provázanosti je software snáze rozšiřitelný a škálovatelný. Oproti monolitickému návrhu je mikroservisní architektura méně náchylná na celkové selhání při chybě v některé z komponent. Jednotlivé komponenty se dají vyvíjet, testovat a nasazovat zvlášť bez zásahu a ovlivnění ostatních fungujících částí, nicméně za cenu obtížnější správy nasazení, jelikož je potřeba jednotlivé komponenty orchestrovat a zajistit jejich vzájemnou komunikaci.

Hlavní nevýhodou rozdělení do několika komponent může být také obtížnější orientace v jejich zodpovědnostech, což může vést ke zpomalení vývoje. Každá komponenta také k fungování zpravidla potřebuje své vlastní oddělené prostředí, tudíž jejich celkový provoz může být náročnější na prostředky.

Pro tuto práci jsem se rozhodl použít právě mikroservisní architekturu. Jednotlivé logické celky popsané na začátku kapitoly budou tedy představovat samostatně fungující a vyvíjené komponenty. Právě snadná rozšiřitelnost nebo případné úplné nahrazení některé z částí jsou hlavními důvody pro zvolení této architektury.

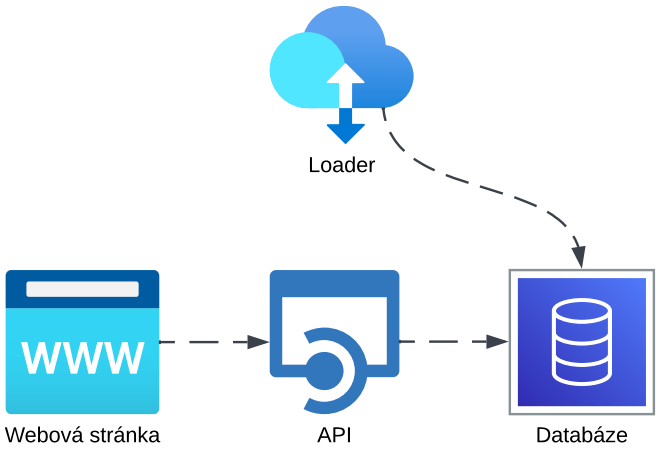
Jednotlivé mikroslužby a znázornění jejich komunikace znázorňuje diagram[1] architektury aplikace 3.1.

3.2 Databáze

Pro průběžné uchovávání a transformaci dat je potřeba zvolit vhodný typ úložiště. Pro tyto účely jsem se rozhodl omezit výběr na relační databáze obsahující nadstavby pro možnost psaní databázových procedur a funkcí.

Databázová procedura je objekt obsahující vstupní, výstupní a vstupně-výstupní parametry a tělo skládající se z příkazů vykonávaných nad databází. Neobsahuje tedy žádná data, nicméně pracuje nad daty již existujícími. Databázové procedury jsou využívány právě pro transformace, přesouvání, mazání dat nebo k uchování často používaných SQL příkazů, které je na rozdíl od pohledů (views) možné parametrizovat vstupním parametrem. V procedurách lze také využívat obecně známé řídící struktury jako jsou podmínky nebo cykly. Procedury nemohou vracet výsledek, pouze prostřednictvím výstupních nebo vstupně-výstupních parametrů. Procedura může řídit průběh databázové transakce a vytvářet nové transakce.

[1] Ikony dostupné z: `https://arch-center.azureedge.net/icons/Azure_Public_Service_Icons_V14.zip`



Obrázek 3.1 Diagram architektury mikroslužeb

Databázová funkce je objekt podobný databázové proceduře. Také neobsahuje data, ale pouze operuje nad již existujícími dle definovaných příkazů v těle. Hlavním rozdílem oproti procedurám je možnost vrácení výsledku. Funkce nejsou schopné řídit průběh transakcí ani vytvářet transakce nové.

Z možnosti relačních databází obsahujících tuto nadstavbu jsem se rozhodoval mezi Oracle Database a PostgreSQL (dále také Postgres). Oracle Database je databázový systém vyvíjený společností Oracle, na rozdíl od Postgresu, který je koncipován jako open source řešení. Právě kvůli open source řešení databáze Postgres jsem se rozhodl pro PostgreSQL s procedurovou nadstavbou PL/pgSQL. Společnost Oracle sice také poskytuje neplacené řešení Oracle Database Express, nicméně s určitými limitacemi prostředků (maximálně 12 GB uživatelských dat, 2 GB RAM, 2 CPU vlákna)[20]. Především limitace na maximální velikost dat je pro účely práce stěžejní.

3.2.1 Entity a vztahy

Návrh jednotlivých databázových entit a vztahů mezi nimi se odvíjel především od definovaných struktur GTFS a struktur poskytovaných rozhraním Golemio API. Zatímco tedy některé databázové entity naprosto identicky replikují GTFS Static formát (trips, stops...), jinde bylo potřeba navrhnout zcela nový model.

Při návrhu některých struktur bylo potřeba vzít v potaz některé skutečnosti:

vozidla: Dopravci mohou svá vozidla v čase měnit. Může se jednat o změnu v bezbariérovosti, instalaci klimatizace, změny evidenčního čísla nebo dokonce typu vozidla (autobus, tramvaj...). Pokud by databáze neřešila tuto skutečnost, mohla by změna definice vozidla v současnosti způsobit nepravdivost dat v historii, tudíž ztracena možnost dohledat informace o vozidle v určitém historickém momentu. Proto bylo pro uchovávání vozidel navrženo užívání uměle generovaného primárního klíče pro

vozidla. Aby se nám stejná vozidla neopakovala v databázi, byla navržena podmínka na unikátnost evidenčního čísla v kombinaci s typem vozidla, bezbariérovostí a klimatizace.

Dalším z důvodů neužití evidenčního čísla vozidla jako primárního klíče bylo zjištění kolizí v evidenci. Konkrétním příkladem je dvojice autobus a tramvaj se stejným evidenčním číslem 9301[2].

synchronizace GTFS Static: Aby bylo předejito opakované synchronizaci GTFS Static dat v případě restartu aplikace, byla navržena entita `gtfs_synced`, která uchovává informace o poslední synchronizaci a její úspěšnosti. Může sloužit externím komponentám při rozhodování, zda je nutné synchronizaci provést.

dopravci: Golemio API vrací identifikátor dopravce `agency_id` provozujícího danou linku při dotazu na `/routes`. Nicméně je vždy uvedena hodnota 99 (PID), proto byla navrhnuta entita `agencies`, která bude samostatně uchovávat získané dopravce z GTFS Realtime dat. Z tohoto důvodu není zmíněné `agency_id` v `routes` cizím klíčem do `agencies`, jedná se o různé interní identifikátory. Synchronizace `agency_id` je zachována pro případnou budoucí změnu ze strany PID.

Výsledný návrh všech databázových entit a jejich vzájemného propojení je znázorněn na ER (entity-relationship) diagramu 3.2.

3.2.2 Databázové pohledy

Pro následné poskytování dat je vhodné uchovat často používané SQL dotazy pro opakované použití. Právě k tomuto účelu slouží databázové pohledy. Jejich prostřednictvím je možné nahlížet na vybranou část surových dat, propojovat je a formátovat dle vlastní potřeby, aniž by bylo nutné měnit samotná uložená data v entitách. Speciálním případem jsou tzv. materializované pohledy, které si získaná data uloží do paměti pro následný rychlejší přístup[21].

Pro statistické pohledy zmíněné v zadání práce bylo navrženo několik základních databázových pohledů. Jejich přehled je zachycen v tabulce 3.1.

3.2.3 Databázové funkce

Pro některé statistické operace nestačí používat pouze databázové pohledy. Konkrétním příkladem mohou být agregace dat o zpoždění (výpočet průměru, minima, maxima...) omezené časovým intervalem, kdy je potřeba mít tyto parametry k dispozici již v době výpočtu, aby použitá agregační funkce počítala s námi omezenými daty.

Přehled navržených databázových funkcí je zachycen v tabulce 3.2. Každá funkce má několik svých variant dle vstupních parametrů, nicméně formát vracených dat je identický.

[2]autobus: `https://seznam-autobusu.cz/vuz/27747`, tramvaj: `https://seznam-autobusu.cz/vuz/58592`

Obrázek 3.2 Databázový ER diagram

3.2.4 Migrace

Po prvotním nasazení aplikace je žádoucí provádět případné další úpravy databáze přímo za běhu. Pokud bychom při každé sebemenší změně na úrovni databáze byli nuceni celou databázi znovu přenasadit od základů, bylo by pokaždé na značnou dobu odstaveno produkční prostředí, tudíž by hrozila ztráta potenciálních synchronizovaných dat a hrozilo by poškození dat již uložených. Právě pro tyto případy je vhodné použít databázové migrace, konkrétně migrace schématu.

Název	Význam
V_TRIPS_SCHEDULES_READABLE	U GTFS Static jízdních řádů mohou být uvedeny hodinové hodnoty větší než 24. Děje se tak v případech, kdy spoj jede přes půlnoc[22]. Tento pohled zajistí sjednocení všech časů do standardní podoby.
V_ACTUAL_TRIPS_DELAYS	Tento pohled zajistí přičtení (případně odečtení) získaného zpoždění daného výjezdu k času v jízdním řádu. Výsledkem je tedy reálný čas příjezdu (případně odjezdu) ze zastávky.
V_WHEELCHAIR_ACCESSIBLE_ROUTES	Výpočet počtu bezbariérových vozidel z celkového počtu vypravených na jednotlivé linky.
V_TRIP_INFO	Pohled přidávající relevantní data k jednotlivým GTFS Static výjezdům, jako jsou například kterou linku daný výjezd pokrývá a počáteční (případně koncovou) zastávku dané linky včetně času odjezdu (případně příjezdu) dle jízdního řádu.
V_ACTUAL_TRIPS_ROUTES	Obdoba V_TRIP_INFO pro skutečné výjezdy. Obsahuje navíc informace o použitém vozidle. Z pohledu lze vyčíst informace o jednotlivých oběžích.

Tabulka 3.1 Databázové pohledy

3.3 Loader

Hlavní úlohou synchronizační komponenty (loaderu) je periodické stahování dat z Golemio API a jejich následná transformace a uložení do databázových struktur. Tato sekvence událostí je označována jako ETL proces[23].

ETL je zkratka pro Extract-Transform-Load proces skládající se ze tří na sebe navazujících základních kroků[24]:

extract: získání dat typicky od třetích stran,

transform: očištění a následná transformace získaných dat do žádoucích struktur,

load: nahrání transformovaných dat do datového úložiště.

Mezi nejpopulárnější programovací jazyky pro implementaci ETL softwaru patří Python a Java[25].

Hlavním rozdílem mezi těmito programovacími jazyky je v typování. Python je dynamicky typovaný, tedy typy proměnných nejsou předem definovány ve zdrojovém kódu, nicméně jsou jazykem určeny za běhu. To umožňuje větší flexibilitu a rychlejší vývoj oproti statickému typování. Java oproti tomu využívá zmíněné statické typování, tedy je

Název	Parametry	Význam
`AGGREGATE_DELAYS`	Časové rozmezí agregace, identifikátor tripu, nebo zastávky.	Agregace zpoždění v jednotlivých zastávkách za zvolené časové období. Spočítáno je minimální, průměrné a maximální zpoždění v každé zastávce.
`AGGREGATE_DELAYS_TIMES`	Časové rozmezí agregace, identifikátor tripu, nebo zastávky.	Obdoba `AGGREGATE_DELAYS`, nicméně místo agregovaných hodnot zpoždění vrací konkrétní časy v porovnání s jízdním řádem. Tedy nejdřívější, průměrný a nejpozdější odjezd ze zastávky.
`WHEELCHAIR_ACCESSIBLE_ROUTES`	Časové rozmezí.	Obdoba pohledu `V_WHEELCHAIR_ACCESSIBLE_ROUTES` s omezením výpočtu nad zadaným časovým obdobím.

Tabulka 3.2 Databázové funkce

nutné striktně definovat typ proměnné při jejím použití. To zajišťuje rychlejší běh (není nutné odvozovat typy proměnných) a vyšší bezpečnost, protože se různá data nemohou nedopatřením promíchat. Právě z důvodu rychlosti a bezpečnosti statického typování, velkého množství podporovaných frameworků a vlastní zkušenosti jsem se rozhodl zvolit Javu pro vývoj loaderu.

3.3.1 Gradle

Pro zjednodušení vývojového procesu jsem se rozhodl použít Gradle[26]. Gradle je open source nástroj pro automatizovanou správu závislostí, sestavení a následné zabalení nebo testování projektu. Funguje na principu různých připravených spustitelných úloh (tasků), které mají na starosti jednotlivé automatizované činnosti.

3.3.2 Spring framework

Spring framework je v současné chvíli jeden z nejpoužívanějších frameworků pro tvorbu Java aplikací a skládá se z několika modulů[27]. Mezi moduly vhodné pro tvorbu loaderu patří:

Core: Klíčovou částí core modulu je tzv. Inversion of Control (IoC) kontejner. Inversion of control (konkrétně dependency injection) je proces při kterém jsou závislosti daného objektu dodány prostřednictvím jeho konstruktoru nebo setterů. Spring Core tyto závislosti automaticky vkládá při vytváření objektů, které jsou označovány jako

beans. Spring tyto vytvořené beans spravuje a v požadovanou chvíli je poskytne při vytváření dalších.

Data Access: Data Access modul slouží k přístupu k datům a jejich mapování do Java struktur a skládá se z několika částí. JDBC (Java Database Connectivity) část zajišťuje spojení s databází a vykonávání SQL dotazů nad ní včetně správy transakcí a chyb. ORM (Object Relational Mapping) část poskytuje rozhraní, které usnadňuje integraci s frameworky zajišťujícími ORM (například Hibernate). Objektově relační mapování je řešení zajišťující oboustrannou konverzi dat z relační databáze do struktur používaného objektově orientovaného programovacího jazyka.

WebFlux: WebFlux je součástí rozsáhlejšího modulu Spring Web a nabízí možnosti implementace asynchronní webové aplikace. Kromě tříd pro implementaci API poskytuje třídy pro zasílání HTTP požadavků a zpracování jejich odpovědí.

Test: Test modul poskytuje řešení pro snazší implementaci různých druhů testů (jednotkové, integrační...). Mezi klíčové funkcionality patří mockování závislostí testované části. Mockování je nahrazení původního funkčního objektu jeho prázdnou alternativou, které předdefinujeme chování. Tím je umožněno otestovat vždy pouze požadovanou část bez nutnosti spoléhat se na funkcionalitu ostatních závislostí.

3.3.3 Architektura

Navržená architektura jednotlivých částí loaderu se odvíjí od toku dat při synchronizaci:

1. Připojení ke Golemio API a poslání HTTP požadavků.
2. Namapování odpovědí HTTP požadavků do objektů.
3. Transformace objektů na databázové entity.
4. Připojení k databázi a uložení vytvořených entit.

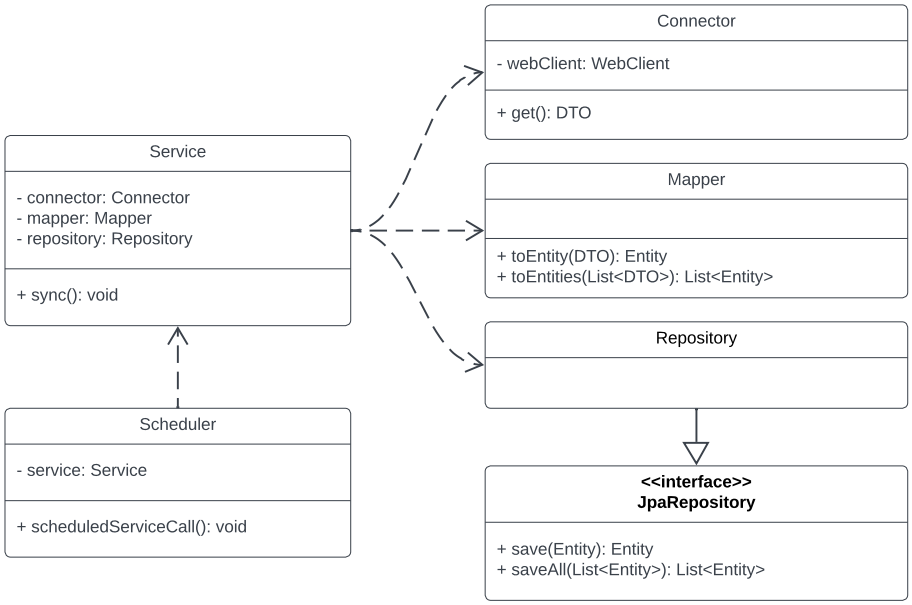
Rozhodl jsem se zvolit návrhový vzor DTO (Data Transfer Object). Ten spočívá v obousměrném mapování DTO objektů na objekty reprezentující databázové entity. DTO objekt je typicky jednoduchá struktura bez aplikační logiky obsahující data a případně metody pro manipulaci s nimi (například serializaci). Objekt databázové entity koresponduje s tabulkou v databázi a je na ní namapován pomocí výše popsané techniky ORM.

Naplnění DTO objektů reálnými daty zajistí komponenta nazvaná `Connector`, která se připojí ke Golemio API, pošle HTTP požadavky a jejich odpovědi vrátí ve formě příslušného DTO objektů. Jejich transformaci na databázové entity zajistí příslušné mapovací třídy, tzv. `Mappery`. Samotné databázové entity budou následně uloženy do databáze prostřednictvím `Repository` objektů poskytujících rozhraní pro komunikaci s databází.

Samotnou orchestraci celého procesu zajistí komponenta `Service`, která bude jako své závislosti obsahovat jednotlivé `Connectory`, `Mappery` a `Repository`. Komponenta typicky zavolá `Connector`, získaná data z něj transformuje pomocí příslušného `Mapperu` a prostřednictvím `Repository` uloží do databáze.

Jelikož je žádoucí provádět synchronizační proces opakovaně v předem stanovených časových intervalech, musí být `Service` komponenta zavolána v příslušný čas. To zajistí komponenta `Scheduler`, která bude hlídat aktuální čas nebo čas uplynutý od minulého zavolání dané `Service` a ve vhodnou chvíli ji zavolá znovu.

Výše popsanou architekturu loaderu obecně zachycuje diagram 3.3.



Obrázek 3.3 Diagram tříd loaderu

3.3.4 Databázové migrace

Jak bylo nastíněno v kapitole 3.2.4, existuje řada frameworků pro automatizaci databázových migrací. Při zvolení jsem se rozhodoval mezi dvěma migračními frameworky pro Javu: Flyway[28] a Liquibase[29]. Obě zmíněná řešení jsou založena na velmi podobném principu, nicméně Liquibase oproti Flyway umožňuje například[3]:

- definovat databázové změny v jiném formátu než SQL (například JSON nebo XML),
- libovolně stanovit pořadí provádění změn (ve Flybase je pořadí automaticky abecedně odvozeno od názvů),
- provádět rollbacky (uvedení do původního stavu) provedených migrací,
- a další.

[3]Porovnávány jsou neplacené verze těchto frameworků.

Výše zmíněné rozdíly mne vedly k finálnímu upřednostnění frameworku Liquibase. Díky přímočaré integraci se Spring aplikací se požadované migrační skripty provedou dle definovaného konfiguračního souboru při spuštění (nasazení) loaderu nad připojenou databází.

3.4 API

Hlavní funkcionalitou API komponenty je poskytovat uložená data prostřednictvím endpointů. Pro tvorbu API jsem se rozhodl pro architekturu REST (Representational State Transfer). REST je specifikace pravidel a doporučení pro tvorbu API, hlavními z nich jsou[30]:

bezstavovost: Server si nepamatuje aktuální stav klienta, musí tedy při každém požadavku obdržet veškeré potřebné informace pro kontext.

cache odpovědí : Možnost ukládat odpovědi v mezipaměti (cache) pro redukci volání API a zrychlení odpovědí.

nezávislost klient-server: Klient a server vzájemně interagují pouze prostřednictvím HTTP požadavků a jsou vzájemně nezávislí.

jednotné rozhraní : Všechny požadavky na stejná data musí být shodné nezávisle na jejich zdroji.

API které splňuje podmínky stanovené ve specifikaci REST se označuje jako RESTful.

Pro tvorbu API jsem se rozhodl rovněž použít Javu se Spring frameworkem v kombinaci s Gradlem, protože Spring obsahuje klíčovou komponentu pro tvorbu RESTful API – Spring Web.

3.4.1 Spring Web

Spring Web[31] je další z modulů Spring frameworku. Základní část Spring Webu označovaná jako Starter obsahuje řešení pro tvorbu webových aplikací založených na MVC (Model-View-Controller) architektuře.

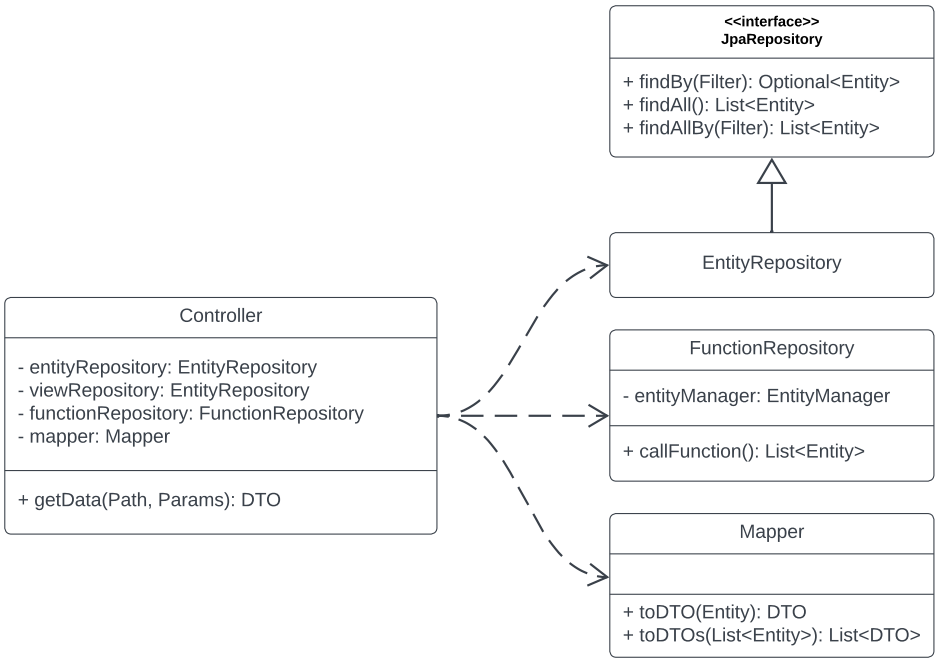
MVC architektura rozděluje aplikaci do tří oddělených vrstev:

model: představuje aplikační (business) logiku a samotná data,

view: poskytuje data zpracovaná modelem,

controller: přijímá uživatelské požadavky, předává je modelu a jejich výsledek zobrazí view.

Kromě části pro implementaci MVC architektury používá framework také vestavěný (embedded) server (v základu Tomcat), díky kterému je možné webovou aplikaci přímo spustit a není potřeba mít v prostředí předinstalovaný server.



Obrázek 3.4 Diagram tříd API

3.4.2 Architektura

Návrh API je inspirován výše popsanou MVC architekturou a je znázorněn na diagramu 3.4.

Obdobně jako u loaderu jsou pomocí techniky ORM namapovány databázové objekty do `Entit`. Kromě samotných databázových tabulek jsou nově mapovány i pohledy (views) a funkce. To umožňuje jejich invokaci přímo z Javy a namapování výsledků do odpovídajících `Entit`. Připojení k databázi a získání požadovaných dat zajišťují `Repository`. Právě `Entity` a `Repository` představují Model v MVC návrhu.

Controller budou představovat třídy `Controller`, které zpracují uživatelské HTTP požadavky, validují je a na jejich základě zavolají příslušné `Repository`.

Část View v tomto případě budou představovat `DTO` objekty. Jejich vytvoření a namapování z příslušných `Entit` zajistí `Mappery`. Vrácené DTO objekty budou následně vizualizovány poslední komponentou – interaktivní webovou stránkou.

3.4.3 Endpointy

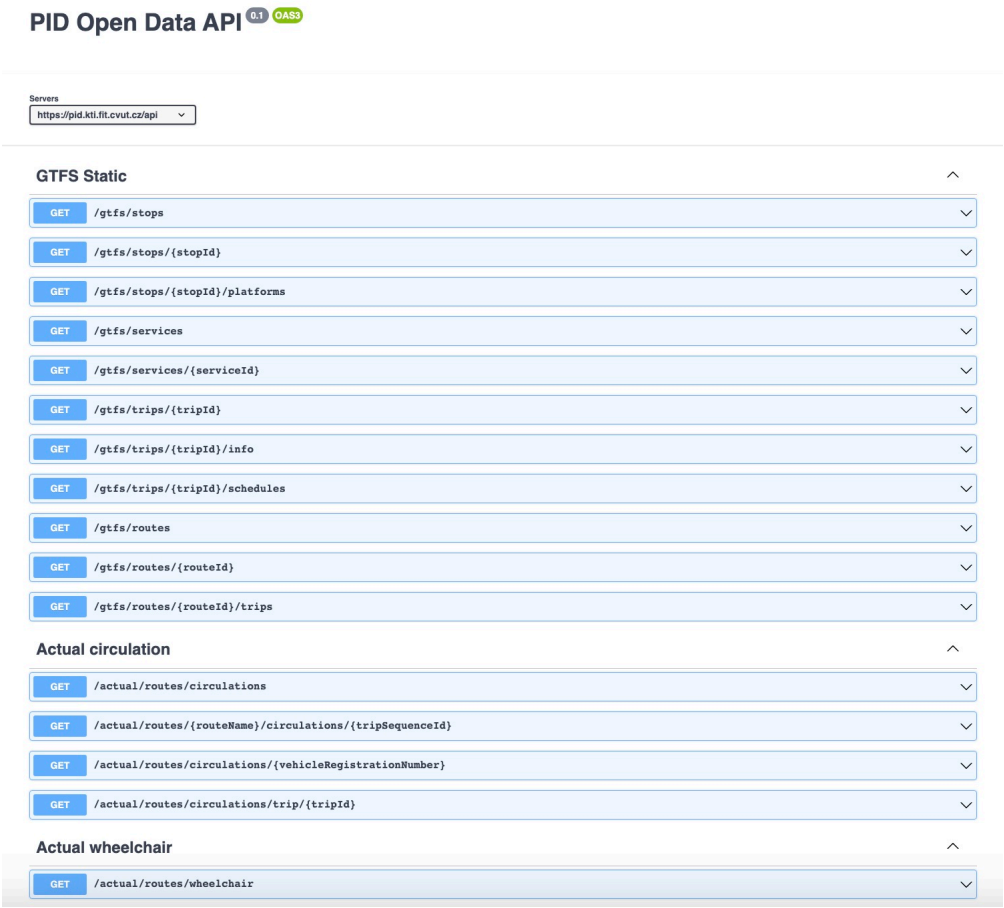
Pro získávání dat z prostřednictvím REST API bylo navrženo několik základních kategorií endpointů. Protože data budou z API pouze stahována a neumožňují úpravu, je jedinou použitou HTTP metodou GET. Základní kategorie a jejich rámcový obsah je vymezen v tabulce 3.3.

Kategorie	Význam endpointů
GTFS Static	Různá GTFS Static data (linky, zastávky, výjezdy...).
Actual circulation	Data o skutečných obězích vozidel.
Actual wheelchair	Data o bezbariérovosti linek.
Actual aggregations	Agregovaná data o zpoždění spojů v jednotlivých zastávkách.
Actual vehicles	Data o všech vypravených vozidlech.

Tabulka 3.3 Základní kategorie endpointů

Detailní specifikace navrženého REST API je zachycena pomocí Swagger dokumentace poskytnuté v příloze práce.

Swagger[32] je open source řešení pro dokumentaci API pomocí definovaných pravidel. Pomocí některých Swagger nástrojů lze generovat dokumentaci přímo ze zdrojového kódu a naopak. Vytvořenou dokumentaci lze pomocí Swagger UI zobrazit v interaktivní podobě a jednotlivé endpointy lze jejím prostřednictvím přímo zavolat například z prohlížeče. Názornou ukázku dokumentace navrženého API zachycuje obrázek 3.5.



Obrázek 3.5 Dokumentace API zobrazená Swagger UI

3.5 Webová stránka

Hlavní funkcionalitou webové stránky je získávání dat od API komponenty a jejich vizualizace pro uživatele. Právě z důvodu získávání dat a jejich proměnlivosti nestačí k tvorbě webové stránky použít pouze HTML a CSS. Nejpoužívanějším programovacím jazykem pro tvorbu dynamických webových stránek je JavaScript[33]. Právě kvůli vysoké používanosti, podpoře a velkého množství dostupných frameworků jsem se rozhodl pro jeho použití, konkrétně s jeho nadstavbou TypeScript.

3.5.1 TypeScript

TypeScript[34] je staticky typovaný programovací jazyk přidávající velké množství funkcionalit a konceptů do JavaScriptu. Nejdůležitějšími z nich jsou silné statické typování a koncepty objektově orientovaného programování (třídy, rozhraní, dědičnost...).

Právě kvůli výše zmíněným nadstavbám jsem se rozhodl použít TypeScript pro tvorbu webové stránky. Stejně jako u loaderu a API, statické typování zajistí vyšší bezpečnost a kontrolu nad daty, koncepty objektově orientovaného programování zajistí vyšší čitelnost a snazší rozšiřitelnost zdrojového kódu a asynchronicita funkcí umožní webové stránce paralelně provádět HTTP požadavky na API nezávisle na sobě, tudíž výrazně urychlí získávání dat.

3.5.2 React

React[35] je JavaScript knihovna pro tvorbu uživatelských rozhraní. Jejím základem jsou takzvané komponenty. Komponenta je objekt obsahující definici vlastního chování a způsobu vykreslení pomocí JavaScript (případně TypeScript) funkcí. Základními komponentami mohou být například tlačítko, zaškrtávací pole a podobné. Tyto menší komponenty jsou postupně kombinovány do složitějších a větších komponent, dokud není vytvořena výsledná podoba aplikace. Díky opětovnému užívání definovaných komponent je zajištěna konzistence v rámci aplikace a značně urychlen proces vývoje.

3.5.3 Tok obrazovek

Rozvrhnutí jednotlivých stránek a která data budou poskytovat jsem se rozhodl zachytit diagramem toku obrazovek. Ten zachycuje obsah jednotlivých stránek včetně možné navigace z nich na stránky ostatní. Dohromady dává představu, jak uživatel bude moci s výslednou stránkou interagovat. Pro vytvoření diagramu jsem se rozhodl použít nástroj pro tvorbu uživatelských rozhraní Figma[36], který umožňuje si výsledný diagram interaktivně projít z pohledu koncového uživatele. Ukázka návrhu toku obrazovek v nástroji Figma je zachycena na obrázku 3.6.

Obrázek 3.6 Návrh toku obrazovek v nástroji Figma

3.6 Vývoj a nasazení

Jednotlivé zdrojové kódy budou průběžně verzovány na platformě GitLab. Pro zajištění platformní nezávislosti systému a jednodušší implementaci vzájemné komunikace mikroslužeb jsem se rozhodl pro používání Dockeru v kombinaci s GitLab CI/CD pipelines pro nasazení.

3.6.1 Docker

Docker[37] je open source virtualizační platforma sloužící pro tzv. kontejnerizaci aplikací. Aplikace se mohou chovat různě v závislosti na prostředí (například operační systém). Docker tato rizika eliminuje pomocí containers (kontejnerů). Nejdříve je vytvořena takzvaná container image, která obsahuje sestavený zdrojový kód včetně všech závislostí potřebných pro jeho spuštění. Z těchto sestavených container images se při spuštění stanou samostatné kontejnery, uvnitř kterých běží samotné aplikace.

Každá mikroslužba bude tedy představovat jeden samostatný kontejner. Jednotlivé kontejnery spolu budou dle potřeby vzájemně komunikovat.

3.6.2 GitLab, CI/CD pipelines

GitLab[38] je webová platforma pro správu Git repozitářů. Git[39] je open source distribuovaný systém správy verzí. Git repozitář je úložiště daného projektu a může se vyskytovat lokálně nebo centrálně na serveru. Zpravidla se při vývoji softwaru používá kombinace těchto přístupů, kdy vývojáři postupně aktualizují centrální repozitář svými lokálními změnami a naopak.

Kromě verzování samotného zdrojového kódu a jeho správu GitLab nabízí také funkcionality pro automatizaci CI/CD. CI/CD[40] je proces vývoje a dodání softwaru a skládá se ze 4 hlavních částí:

vývoj: psaní zdrojového kódu,

sestavení: sestavení (build) zdrojového kódu,

testování: manuální testování nebo spuštění dostupných automatických testů,

nasazení: nasazení do testovacího (případně produkčního) prostředí

Tento proces se v průběhu vývoje mnohokrát opakuje s dodáním nových verzí softwaru. K automatizaci procesu slouží nástroj GitLab CI/CD. Ten umožňuje vývojáři definovat sérii navazujících kroků, tzv. CI/CD pipeline, která typicky zajišťuje automatické sestavení, testování a následné nasazení nové verze aplikace.

Pro účely práce jsem se rozhodl používat fakultní GitLab[4].

[4]Dostupný na: `https://gitlab.fit.cvut.cz/`

Kapitola 4

Implementace

Tato kapitola popisuje implementaci a výsledné nasazení jednotlivých komponent pomocí technologií zvolených v návrhu. Kromě názorných příkladů zdrojového kódu se věnuje problémům, na které jsem v průběhu vývoje narazil, a postupu jejich řešení.

4.1 Databáze

4.1.1 Inicializace

K prvotní inicializaci databáze byl vytvořen soubor obsahující SQL skripty pro vytvoření základních entit včetně příslušných primárních a cizích klíčů a sekvencí. Tento skript je zamýšlen k jedinému spuštění a to pouze při prvotním nasazení databáze. Jakékoliv další úpravy jsou zajišťovány prostřednictvím migrací, aby nedošlo k přepsání již existujících struktur.

4.1.2 Migrace

K automatizaci databázových migrací byl v návrhu zvolen framework Liquibase. Ten využívá k definici migrací soubory zvané changelogy. Changelog obsahuje sekvenční výčet jednotlivých migrací v pořadí, ve kterém budou provedeny nad databází. Jednotka změny v changelogu se označuje jako changeset a jeho změny mohou být definovány ve formátech SQL, XML, YAML nebo JSON. Hlavní changelog může obsahovat reference na další changelogy pro zvýšení přehlednosti při vývoji.

Pro definici změn v rámci changesetů jsem zvolil SQL. Jednotlivé definice jsou pro přehlednost odděleny do separátních `.sql` souborů obsahující samotný SQL (případně PL/pgSQL) kód. Ukázka changelogu je zachycena ve výpisu 1.

Jednotlivé changesety se nad databází provedou standardně pouze jednou, nicméně lze pomocí parametrů nastavit opakované spouštění nebo exekuci pouze při detekované změně souboru.

```
<databaseChangeLog xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
                   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                   xsi:schemaLocation="http://www.liquibase.org/xml/ns/
                   dbchangelog
                   http://www.liquibase.org/xml/ns/dbchangelog/
                   dbchangelog-latest.xsd">

    <changeSet id="0001_views_delays" author="zibriale">
        <sqlFile path="migrations/0001_v_trips_schedules_readable.sql"/>
        <sqlFile path="migrations/0001_v_actual_trips_delays.sql"/>
    </changeSet>

    <changeSet id="0002_view_wheelchair_accessibility"
    author="zibriale">
        <sqlFile
        path="migrations/0002_v_wheelchair_accessible_routes.sql"/>
    </changeSet>
</databaseChangeLog>
```

Výpis kódu 1 Liquibase changelog

Liquibase si v databázi automaticky vytvoří dvě pomocné entity `databasechangelog` a `databasechangeloglock`:

`databasechangelog:` Obsahuje seznam jednotlivých provedených changesetů, datum jejich provedení a MD5 hash obsahu pro detekci případné změny. Slouží Liquibase jako řídicí tabulka při rozhodování, které změny je potřeba provést.

`databasechangeloglock:` Slouží k „zamknutí“ Liquibase pro případy paralelního spuštění více instancí. Zajistí, že vždy běží pouze jedna a nedojde ke konfliktům při případných paralelních migracích.

4.1.3 Optimalizace

V průběhu vývoje bylo potřeba přistoupit k několika optimalizacím oproti původnímu návrhu z důvodu zrychlení jednotlivých dotazů.

První z optimalizací bylo odstranění používání umělého unikátního identifikátoru v pohledech. Aby jednotlivé řádky pohledů byly snáze jednoznačně identifikovatelné, zvolil jsem zprvu generované číslo z příkazu `row_number() over ()`. Pomocí příkazu `EXPLAIN ANALYZE`, který provede dotaz a zobrazí náročnost jednotlivých operací, se ukázalo, že probíhá očíslování všech dostupných dat a požadovaný filtr v klauzuli `WHERE` je proveden až na samém konci. To mělo za následek vysoký čas exekuce. Proto byl tento klíč z pohledů odstraněn a jako unikátní identifikátory řádků slouží kombinace existujících dat. Po této úpravě je filtr aplikován na začátku, tudíž značně redukuje počet zpracovávaných dat při dotazu.

Další optimalizace proběhla nad pohledem `V_TRIP_INFO`, který byl následně používán v dalších pohledech pro poskytnutí informací o výjezdu. Pomocí příkazu `EXPLAIN ANALYZE` jsem zjistil, že při spojování pohledů neprobíhala optimalizace filtrování dat a probíhal sken veškerých dostupných dat. To vedlo k pomalé exekuci pohledů postavených nad tímto pohledem. Jako řešení jsem zvolil změnu pohledu `V_TRIP_INFO` na materializovaný pohled `MV_TRIP_INFO`, který je následně používán v dalších pohledech. Jelikož se GTFS Statické údaje aktualizují jednou denně, stačí po jejich synchronizaci provést aktualizaci dat uložených v materializovaném pohledu.

Další zrychlení dotazů zajistilo vhodné použití indexů. Databázový index[41] je struktura umožňující rychlejší vyhledávání nad zvolenými sloupci. Typicky je vhodné indexovat sloupce, podle kterých se často filtruje nebo třídí. Jedním z použití bylo indexování sloupce `vehicle_registration_number` v entitě `vehicles`, protože při získání všech oběhů vozidla dle evidenčního čísla je podle něj často filtrováno. Ukázka vytvoření indexu je zachycena na výpisu 2.

```
CREATE INDEX VEHICLES_VEHICLE_REGISTRATION_NUMBER_IDX
ON VEHICLES(vehicle_registration_number)
```

Výpis kódu 2 Vytvoření indexu nad evidenčními čísly vozidel

4.1.4 Docker

Jako instance používané databáze byla zvolena oficiální Postgres docker image[1]. Konfigurace kontejneru je řešena prostřednictvím Docker compose[42]. Konfigurace databáze v souboru `docker-compose.yml` je znázorněna na výpisu 3.

```
db:
    image: postgres:15.1
    restart: always
    env_file:
      - db.env
    ports:
      - '5432:5432'
    volumes:
      - ./db:/docker-entrypoint-initdb.d
      - /mnt/data/pid_volumes/postgres_data:/var/lib/postgresql/data
```

Výpis kódu 3 Konfigurace Postgres Docker kontejneru

Kromě zvolené image je v souboru definována cesta k souboru `db.env` obsahující proměnné prostředí. Ty obsahují citlivé údaje vytvořeného databázového uživatele (jméno, heslo). Dále je definováno mapování portů a volumes. Volumes zajišťují oboustrannému mapování souborů uvnitř kontejneru na soubory na lokálním souborovém systému:

[1] Dostupná z: `https://hub.docker.com/_/postgres`

- První volume slouží k namapování inicializačních skriptů ve složce `db` do speciální složky `docker-entrypoint-initdb.d`. Všechny skripty dostupné v této složce postgres spustí při prvotním spuštění databáze a provede nad databází.

- Druhý volume slouží k mapování samotných dat na lokální souborový systém. Pokud bychom kontejner restartovali bez této volume, ztratili bychom veškerá data.

4.2 Loader

4.2.1 Závislosti

Jak bylo zmíněno v návrhu (viz kapitola 3.3.1), ke správě závislostí jsem se rozhodl použít Gradle. Jednotlivé závislosti se definují v souboru `build.gradle` a jsou několika typů (zmiňuji jen v práci užívané typy):

`implementation` : závislosti potřebné ke zkompilování zdrojového kódu a při následném běhu aplikace,

`compileOnly:` závislosti potřebné pouze v době kompilace,

`runtimeOnly:` závislosti potřebné pouze při běhu aplikace,

`testImplementation:` závislosti potřebné pro zkompilování a běh testů,

`annotationProcessor:` závislost poskytující anotační procesor (popsán u Lombok závislosti).

Kromě závislostí popsaných v návrhu jsem se rozhodl pro zrychlení vývoje použít Lombok jako `compileOnly` závislost. Lombok[43] je knihovna poskytující různé anotace pro generování základních metod tříd (konstruktory, metody pro přístup k atributům, buildery a další) při kompilaci. K zpracování jednotlivých anotací používá Lombok anotační procesor[44], který slouží k vyhodnocení použitých anotací, podle kterých je následně vygenerován zdrojový kód. Použití Lombok anotací je zachyceno na výpisu 4.

```
@NoArgsConstructor
@AllArgsConstructor
@Data
@Builder
public class VehicleEntity {
    //...
}
```

Výpis kódu 4 Použití Lombok anotací

Další použitá závislost `org.postgresql:postgresql`[45] je typu `runtimeOnly` a jedná se o JDBC ovladače potřebné k připojení k Postgres databázi.

4.2.2 Konfigurace

Vzhledem k funkčnímu požadavku UC3 (viz kapitola 2.7.1) je potřeba zařídit snadnou konfiguraci. K tomu slouží soubor `application.properties`, který funguje na principu klíč-hodnota a umožňuje měnit některé parametry aplikace. V souboru jsou kromě konkrétních hodnot použity i reference na proměnné prostředí, typicky pro citlivá data (přístupové tokeny, údaje pro připojení k databázi...).

4.2.3 Komponenty

4.2.3.1 Connector

Komponenta `Connector` je implementována prostřednictvím třídy z frameworku Spring Web `WebClient`. Jsou vytvořeny dvě separátní `WebClient` instance (pro GTFS Static a GTFS Realtime) a zaregistrovány jako Spring beans. Tyto beans jsou základními závislostmi pro `Connector`. Jejich prostřednictvím je voláno Golemio API a zpracováván výsledek volání. Ukázka volání třídy `WebClient` v `Connector` je zachycena na výpisu 5.

```
@Component
public class GtfsConnector {
    //...
    @Autowired
    @Qualifier(value="gtfsWebClient")
    private WebClient gtfsWebClient;

    private ResponseSpec get(String path, int offset){
    return gtfsWebClient.get()
      .uri(path, uriBuilder -> uriBuilder
        .queryParam("offset", offset * limit)
        .queryParam("limit", limit)
        .build())
      .retrieve();
    }
    //...
}
```

Výpis kódu 5 Volání Golemio API endpointů

Aby aplikace po prvotním spuštění zvládla stáhnout veškerá potřebná GTFS Static data co nejrychleji a tedy zkrátila interval neúplnosti dat na nutné minimum, bylo získávání jízdních řádů implementováno asynchronně. Jednotlivá GET volání jsou prováděna paralelně v samostatných vláknech. Nicméně bylo třeba brát v úvahu zatížení Golemio API ze strany loaderu.

Během nasazení prvotní verze aplikace nebylo dostatečně optimalizováno množství těchto zaslaných požadavků a byla vytvořena přílišná zátěž na Golemio API. Po emailové komunikaci s vývojáři Golemio[10] bylo dosaženo vzájemné spokojenosti omezením

zasílaných požadavků z loaderu a stanovením adekvátního intervalu na synchronizaci pozic vozidel (jednou za 5 minut).

4.2.3.2 Entity a Repository

K mapování databázových entit na `Entity` třídy pomocí techniky ORM slouží JPA (Jakarta Persistence API) anotace. Každá entita je označena anotací `@Entity` a musí obsahovat identifikátor (odpovídá primárnímu klíči v databázi). Atribut sloužící jako identifikátor je označen anotací `@Id`. K specifikaci tabulky slouží anotace `@Table`. Jednotlivé atributy obsahují dodatečné anotace potřebné ke korektnímu mapování (jméno sloupce, mapování na databázovou sekvenci nebo zachycení cizího klíče do jiné tabulky). Všechny tyto mapování jsou zachyceny na výpisu 6.

```
@Entity
@Table(name=TABLE_VEHICLES)
public class VehicleEntity {
    @Id
    @SequenceGenerator(name="vehicles_vehicle_row_id_seq",
      sequenceName="vehicles_vehicle_row_id_seq", allocationSize=1)
    @GeneratedValue(strategy=GenerationType.SEQUENCE,
      generator="vehicles_vehicle_row_id_seq")
    @Column(name="vehicle_row_id")
    private Integer vehicleRowId;

    @ManyToOne
    @JoinColumn(name = "vehicle_type_id",
      referencedColumnName = "vehicle_type_id")
    private VehicleTypeEntity vehicleType;

    @Column(name="vehicle_registration_number")
    private Integer vehicleRegistrationNumber;
    //...
}
```

Výpis kódu 6 Technika ORM pomocí JPA

K přístupu k jednotlivým entitám a operacím jsou používány `Repository` rozhraní. Ta jsou postavena nad rozhraním `JpaRepository`, které obsahuje základní metody pro získávání nebo ukládání dat. Pomocí `JpaRepository` je možné definovat další vlastní potřebné metody, které jsou následně přeloženy na příslušné SQL dotazy, případně je možné pomocí anotace `@Query` napsat libovolný dotaz přímo v SQL, případně JPQL (Java Persistence Query Language) – vlastní dotazovací jazyk založený na SQL. Definice `Repository` s vlastní metodou pokrývá výpis 7.

```
    public interface GtfsSyncedRepository
    extends JpaRepository<GtfsSyncedEntity, Integer> {

    List<GtfsSyncedEntity> findAllByStatusAndStartTimestampAfter
    (String status, Timestamp startTimestamp);
}
```

Výpis kódu 7 Definice `Repository` rozhraní

4.2.3.3 DTO a Mapper

K uchování získaných dat před jejich transformací do databázových entit slouží třídy `DTO`. Ty jsou implementovány jako jednoduché třídy obsahující pouze jednotlivé atributy pro data a příslušné přístupové metody. Aby byla zajištěna jednoznačnost atributů v `DTO` při mapování atributů z přijatého JSONu, jsou před atributy použity anotace `@JsonAlias(value = "value")`, kde value v uvozovkách je nahrazeno jménem příslušného atributu v JSONu.

K samotné transformaci `DTO` na `Entity` slouží `Mapper` implementovaný jako třída obsahující dvě metody na převod jednoho (případně více) `DTO` objektu do `Entity`. Fungují na principu inicializace `Entity` (případně seznamu `Entity`), jejím naplnění příslušnými atributy z `DTO` objektu (případně seznamu objektů) a navrácení.

4.2.3.4 Service

Třída `Service` slouží jako orchestrační třída pro předchozí implementované třídy. Obsahuje samotnou aplikační logiku a stará se o zpracování případných chyb při běhu a jejich logování do souboru stanoveného v konfiguraci aplikace. Pod aplikační logikou se skrývá cyklické volání `Connector` s vhodnými parametry (stránkování požadavků), následné mapování jejich odpovědí na `Entity` a jejich uložení do databáze prostřednictvím `Repository`. Jedna z implementovaných metod je znázorněna na výpisu 8.

4.2.3.5 Scheduler

K zajištění opakovaného volání `Service` metod ve stanovený čas slouží `Scheduler` třídy. Metody pro zavolání `Service` jsou anotovány pomocí `@Scheduled` s parametrem stanovujícím časový interval mezi jednotlivými invokacemi. Spring s použitím této anotace zajistí opakovaná volání metod třídy v příslušný čas. Hodnoty stanovující tyto intervaly jsou získány z konfiguračního souboru (viz kapitola 4.2.2) při spuštění aplikace.

4.2.4 Testování

Stěžejní třídy loaderu byly pokryty jednotkovými (unit) testy. Jednotkový test slouží k izolovanému testu funkcionality právě jednoho logického celku. Ostatní potřebné závislosti jsou vhodně simulovány (mockovány), aby jejich funkcionalita neměla vliv na průběh testu a bylo dosaženo maximální možné izolace. K implementaci jednotkových testů[46] jsem využil platformu JUnit[47] v kombinaci s frameworkem Mockito[48].

```
    private void getStops(){
        try {
            int offset = 0;
            List<StopEntity> stops;
            do {
                stops = stopMapper.toStopEntities(Objects.requireNonNull(
                  gtfsConnector.getStops(offset++).block())
                );
                stopRepository.saveAll(stops);
            } while(!stops.isEmpty());
        } catch (Exception e) {
            log.error("Error getting stops: " + e.getMessage());
            throw new RuntimeException(e.getMessage());
        }
    }
```

Výpis kódu 8 Metoda `Service` třídy pro synchronizaci zastávek

Události testu typicky tvoří definování vhodných mocků, následné zavolání testované metody a vyhodnocení výsledků volání. Základní ukázku průběhu testu pomocí JUnit a Mockito ilustruje výpis 9.

```
@Test
public void getServicesTest() throws Exception {
    //Konstrukce vracených struktur
    List<ServiceEntity> serviceEntities = List.of(
      ServiceEntity.builder().serviceId("serviceId1").build(),
      ServiceEntity.builder().serviceId("serviceId2").build());
    //Definice chování mock objektů
    Mockito.when(gtfsWebClient.get())
      .thenReturn(requestHeadersUriSpec);
    Mockito.when(requestHeadersUriSpec.uri(Mockito.eq("/services"),
      ArgumentMatchers.any(Function.class)))
        .thenReturn(requestHeadersSpec);
    Mockito.when(requestHeadersSpec.retrieve())
        .thenReturn(responseSpec);
    Mockito.when(responseSpec.bodyToFlux(ServiceEntity.class))
      .thenReturn(Flux.fromIterable(serviceEntities));
    //Zavolání testované metody a ověření výsledku
    assertEquals(gtfsConnector.getServices(0).collectList().block(),
    serviceEntities);
}
```

Výpis kódu 9 Unit testování metody v třídě `Connector`

4.2.5 Docker

Loader je podobně jako DB nasazen jako Docker kontejner. Jeho základem je Alpine Linux[49] (distribuce Linuxu často používaná v kontejnerech kvůli malé velikosti 5 MB) v kombinaci s OpenJDK[50] pro kompilaci a spuštění zdrojového kódu. Opět je použit Docker compose, oproti databázi je doplněn o cestu k `Dockerfile`. `Dockerfile` je soubor definující způsob sestavení vlastní Docker image. Definice `Dockerfile` je znázorněna na výpisu 10. Skládá se ze dvou kroků:

build: Zkompilování zdrojového kódu a vytvoření výsledného spustitelného `.jar` souboru pomocí Gradle.

run: Nakopírování spustitelného souboru z build fáze a jeho spuštění při vytvoření kontejneru.

```
FROM alpine:3.17 AS builder
RUN apk add openjdk17
ADD . /build
WORKDIR /build
RUN ./gradlew assemble

FROM alpine:3.17 AS run
RUN apk add openjdk17-jre-headless tzdata &&
ln -s /usr/share/zoneinfo/Europe/Prague /etc/localtime
WORKDIR /app
COPY --from=builder
/build/build/libs/loader-0.0.1-SNAPSHOT.jar ./loader.jar
ENTRYPOINT ["java", "-jar", "loader.jar"]
```

Výpis kódu 10 `Dockerfile` pro loader komponentu

4.3 API

4.3.1 Závislosti

API komponenta je založena na podobných technologiích jako loader. Obsahuje nově závislosti:

`spring-boot-starter-web` pro tvorbu požadovaných endpointů,

`spring-boot-starter-validation` pro validaci parametrů požadavků,

`springdoc-openapi-starter-webmvc-ui` pro generování Swagger OpenAPI dokumentace ze zdrojového kódu.

Generování dokumentace pomocí poslední zmíněné závislosti slouží především k základnímu jednoduchému přehledu dostupných endpointů. Vygenerovaná dokumentace byla pro snazší čitelnost a větší přehlednost ručně upravena do výsledné podoby.

4.3.2 Komponenty

4.3.2.1 Entity a Repository

Identicky jako v loaderu je použita technika ORM pomocí JPA (viz kapitola 4.2.3.2). Kromě databázových tabulek lze mapovat i pohledy (do anotace `@Table` je vloženo jméno příslušného pohledu) nebo databázové funkce a procedury. K mapování databázových funkcí jsem použil anotaci `@NamedStoredProcedureQuery` v kombinaci s anotací `@SqlResultSetMapping` pro transformaci výsledku SQL dotazu do příslušné `Entity` třídy. Ty jsou implementovány obdobně jako v loaderu. Pro volání funkcí bylo potřeba vytvořit speciální `Repository`, jelikož se nejedná o standardní využití na základě `JpaRepository`. Zdrojový kód ilustrující implementaci vlastní `Repository` včetně metody pro zavolání databázové funkce s parametry zachycuje výpis 11.

```
@Repository
@AllArgsConstructor
public class WheelchairRoutesRepository {
    private final EntityManager entityManager;

    public List<WheelchairRoutes> getWheelchairRoutes(Date dateFrom,
    Date dateTo) {
        StoredProcedureQuery query = entityManager.
          createNamedStoredProcedureQuery("wheelchairAccessibleRoutes");
        query.setParameter(PARAM_DATE_FROM, dateFrom);
        query.setParameter(PARAM_DATE_TO, dateTo);
        return query.getResultList();
    }
}
```

Výpis kódu 11 Implementace `Repository` pro funkci

4.3.2.2 DTO a Mapper

Postup implementace `DTO` objektů je totožný jako v loaderu (viz kapitola 4.2.3.3. Jediným rozdílem je způsob jejich následného použití. V loaderu sloužily `DTO` objekty k deserializaci přijatého JSONu. Zde jsou naopak použity pro konstrukci výsledného JSONu zaslaného jako odpověď na HTTP požadavek. U atributů je použita anotace `@JsonProperty`, díky které je možné specifikovat konkrétní jméno atributu ve výsledném JSONu.

Implementace `Mapper` tříd se liší ve směru transformace objektů. Pro poskytnutí dat na výstup je potřeba transformovat získané databázové objekty (`Entity`) do příslušných `DTO` struktur pro následnou tvorbu JSONu. V loaderu byl tento směr opačný. V `Mapper` třídách jsou opět metody pro transformaci jedné `Entity` na `DTO`, případně jejich seznamů.

4.3.3 Controller

Třídy `Controller` zajišťují zpracování přijatých HTTP požadavků, obstarání příslušných dat a následné vrácení požadované odpovědi. Každá z veřejných metod třídy za-

jišťuje mapování jednoho z endpointů popsaných v návrhu (viz kapitola 3.4.3. Tohoto mapování je zajištěno prostřednictvím anotace `@GetMapping` dostupné ze Spring Web. K definici proměnných požadavku slouží anotace `@PathVariable` a `@RequestVariable` dle jejich typu. K základní validaci formátu přijatých dat (použit formát ISO 8601[51]) byla použita anotace `@DateTimeFormat`. Názornou ukázku implementace jedné z metod `Controller` třídy zobrazuje výpis 12.

```
@GetMapping(value = "/routes/circulations/trip/{tripId}",
            produces = "application/json")
public List<ActualTripsRoutesDTO> getRoutesCirculationByTripId(
    @PathVariable String tripId,
    @RequestParam(name = "date_from")
        @DateTimeFormat(pattern = "yyyy-MM-dd") @Valid Date dateFrom,
    @RequestParam(name = "date_to")
        @DateTimeFormat(pattern = "yyyy-MM-dd") @Valid Date dateTo) {
    return actualTripsRoutesMapper
        .toActualTripsRoutesDTOs(actualTripsRoutesViewRepository
            .findAllByTripIdAndDateBetween(tripId, dateFrom, dateTo));
    }
```

Výpis kódu 12 Implementace metody ve třídě `Controller`

4.3.4 Docker

K vytvoření výsledného Docker kontejneru byl zvolen identický postup a technologie jako v případě loaderu (Alpine Linux, OpenJDK). Samotný `Dockerfile` soubor se od verze v loaderu (viz výpis 10) liší pouze ve jméně vytvořeného výsledného `.jar` souboru.

4.3.5 Testování

Jednotlivé endpointy tříd `Controller` jsou pokryty jednotkovými testy. Jejich implementace je založena na stejných technologiích a principech jako v loaderu (viz kapitola 4.2.4). Pro testy endpointů je použita třída `MockMvc`. Ta slouží k simulaci požadavků oproti mockovému prostředí, tudíž nedochází k reálné HTTP komunikaci. Použití této třídy včetně náznaku validace odpovědi je naznačena na výpisu 13.

```
mockMvc.perform(get("/gtfs/stops"))
    .andExpect(status().isOk())
    .andExpect(jsonPath("$", Matchers.hasSize(2)))
    .andExpect(jsonPath("$[0].stop_id", Matchers.is("stopId1")))
    //... more assertions
```

Výpis kódu 13 Použití `MockMvc` při testování třídy `Controller`

4.4 Webová stránka

4.4.1 Typy

Jak bylo popsáno v návrhu (viz kapitola 3.5.1), TypeScript umožňuje statické typování. Pro data získávaná z API bylo vyvinuto několik odpovídajících rozhraní obsahující jednotlivé atributy včetně jejich datových typů. Tím je zajištěno odpovídající mapování odpovědí do TypeScript struktur pro další následné přístupy při vykreslování.

4.4.2 Komponenty

Jak bylo zmíněno v návrhu (viz kapitola 3.5.2), React je založen na komponentách. Pro webovou stránku bylo implementováno několik základních komponent. Na definici vlastní triviální React komponenty lze nahlédnout na výpisu 14.

```
import React from 'react'
import { IVehicle } from '../types'

export default function VehicleBadge({ vehicle }: { vehicle: IVehicle })
{
  return (
    <span>{vehicle.vehicle_registration_number}</span>
  )
}
```

Výpis kódu 14 Definice vlastní React komponenty

Ukázková komponenta využívá jedno z dříve definovaných rozhraní `IVehicle` a přistupuje k jeho atributu `vehicle_registration_number`. Tato komponenta je základem pro případná další rozšíření, například ikony s typem vozidla a evidenčním číslem (podobně jako u zobrazení linek).

4.4.3 Klient

Klient slouží k zasílání HTTP požadavků na API komponentu a mapování odpovědí do příslušných struktur. Základ vlastní implementace tvoří `Axios`[52] – HTTP klient založený na `Promise`. `Promise` je obalovací struktura pro hodnoty, které nemusí být v době návratu z funkce známy. Využívá se jako návratová hodnota při asynchronních voláních, jelikož není žádoucí čekat na odpověď (popíralo by význam asynchronicity).

`Axios` klient je následně využíván závislostí `react-query`[53] – knihovna pro správu získávání asynchronních dat. Poskytuje řešení pro udržování dat v mezipaměti (cache), jejich aktualizování v požadovaných případech nebo optimalizaci zasílaných HTTP požadavků.

4.4.4 Směrování

K navigaci mezi jednotlivými vizualizacemi byla použita závislost `react-router`[54]. Pomocí této závislosti jsou definovány různé cesty (routes). Při napsání URL do prohlížeče je uživatel v případě existující cesty přesměrován na stránku s požadovanými daty. Tím je zajištěna synchronizace URL se zobrazovaným uživatelským rozhraním.

Ve zdrojových souborech nejsou obsaženy statické HTML soubory pro jednotlivé cesty, obsah je řešen pomocí JavaScriptu. Změna URL je změna stavu aplikace, to znamená načtení potřebných dat a zobrazení příslušného uživatelského pohledu (view).

Ukázka navigace na stránku s dalšími údaji o dané lince je zobrazena na výpisu 15.

```
<td>
    <Link to={`/routes/${r.route_id}`} style={{textDecoration: 'none'}}>
        <RouteBadge route={r} />
    </Link>
</td>
```

Výpis kódu 15 Směrování pomocí `React router`

4.4.5 Docker

`Dockerfile` pro sestavení image se opět skládá ze dvou základních kroků: build a run. V build kroku je znovu použit jako základ Alpine Linux, do kterého je doinstalován správce JavaScript závislostí NPM, který zařídí sestavení webové aplikace. Sestavená webová aplikace je následně v dalším kroku nakopírována do run fáze, jejíž základ tvoří webový server Nginx postavený opět nad Alpine Linuxem. To vede k minimální prostorové náročnosti Docker kontejneru.

4.5 Nasazení

4.5.1 Docker compose

K výsledné orchestraci všech kontejnerů jednotlivých mikroslužeb a jejich vzájemné komunikační dostupnosti byl použit Docker compose. Výsledný vytvořený konfigurační soubor `docker-compose.yml` je dostupný v kořenové složce zdrojových souborů. Docker compose umožňuje definování portů, pod kterými budou jednotlivé mikroslužby dostupné. Pro kontejnery bylo zvoleno následné rozložení portů:

databáze: 5432 (standard pro Postgres databáze),

api: 8081,

webová stránka: 8080.

4.5.2 Směrování portů

Jednotlivé kontejnery jsou nasazeny na virtuální stroj dostupný za fakultním proxy serverem na adrese `pid.kti.fit.cvut.cz`. Na virtuálním stroji je použit Nginx poslouchající na portu 80, což je jediná možná cesta k vnějšímu přístupu k virtuálnímu stroji. Hlavní zodpovědností Nginxu je přesměrování GET požadavků podle cesty požadavku na příslušné interní Docker kontejnery:

/ přesměrování na localhost:8080 (kontejner s webovou stránkou),

/api přesměrování na localhost:8081 (kontejner s API).

4.5.3 CI/CD Pipeline

K automatizaci procesu nasazení aplikace při změně byla implementována GitLab CI/CD pipeline. Ta se skládá ze tří základních kroků (stages):

build: Sestavení jednotlivých Docker images pro mikroslužby.

test: Spuštění automatických jednotkových testů.

deploy: Nasazení aplikace do produkčního prostředí.

Jednotlivé kroky jsou na sobě závislé, tudíž při neúspěchu sestavení nebo testování alespoň jedné z mikroslužeb nelze pokračovat k nasazení. Nasazovat do produkčního prostředí lze pouze z master větve, aby bylo předejito případnému nasazení testovací verze na produkční prostředí.

Build a test fáze jsou spuštěny automaticky při každém novém commitu v libovolné větvi repozitáře. Deploy fáze je nastavena na manuální spuštění po úspěšných build a test fázích. Jedná se o další pojistku proti případnému nasazení špatné úpravy do produkčního prostředí. Kompletní výsledná definice CI/CD pipeline je dostupná v souboru `.gitlab-ci.yml` v kořenovém adresáři zdrojových souborů.

Závěr

Cílem práce bylo navrhnout a implementovat řešení pro průběžné uchovávání, zpracovávání a následné poskytování otevřených dat Pražské integrované dopravy.

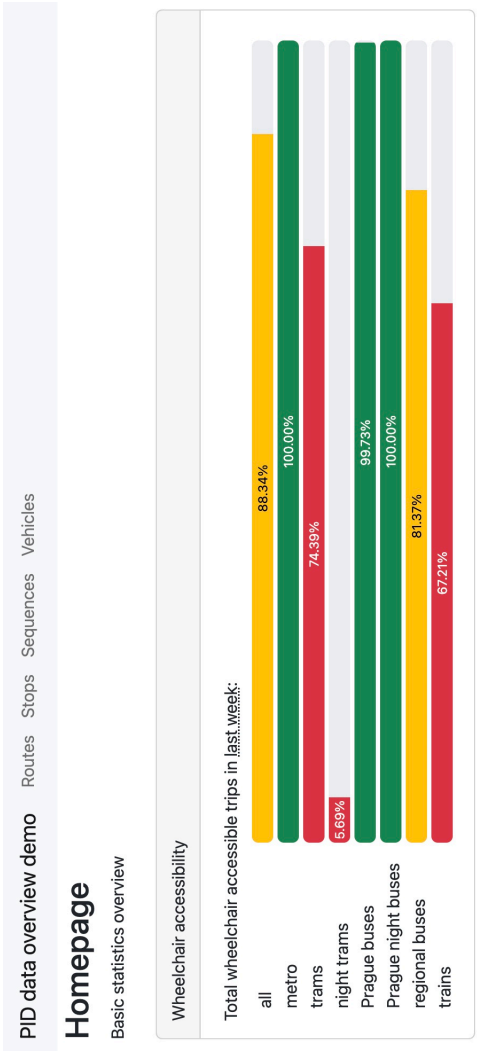
Po analýze standardního způsobu poskytování dat hromadné dopravy a existujících řešení následoval návrh jednotlivých částí aplikace: synchronizační komponenty, databáze, rozhraní (API) poskytující data z databáze a samotné webové stránky s jejich vizualizacemi. Na základě analýzy a návrhu jsem přistoupil k implementaci všech zmíněných komponent, jejich testování a v neposlední řadě nasazení do prostředí poskytnutého fakultou. Nasazená aplikace je veřejně dostupná na adrese `pid.kti.fit.cvut.cz`.

Výsledný vyvinutý software svou funkcionalitou pokrývá všechny vytyčené cíle, proto považuji jejich splnění za úspěšné.

V budoucnu by bylo možné aplikaci rozšířit například o další relevantní statistické operace nad získávanými daty nebo upravit současný způsob získávání dat s využitím popsaných Protocol buffers, které v současné chvíli nejsou poskytovány ve vhodném stavu pro účely práce.

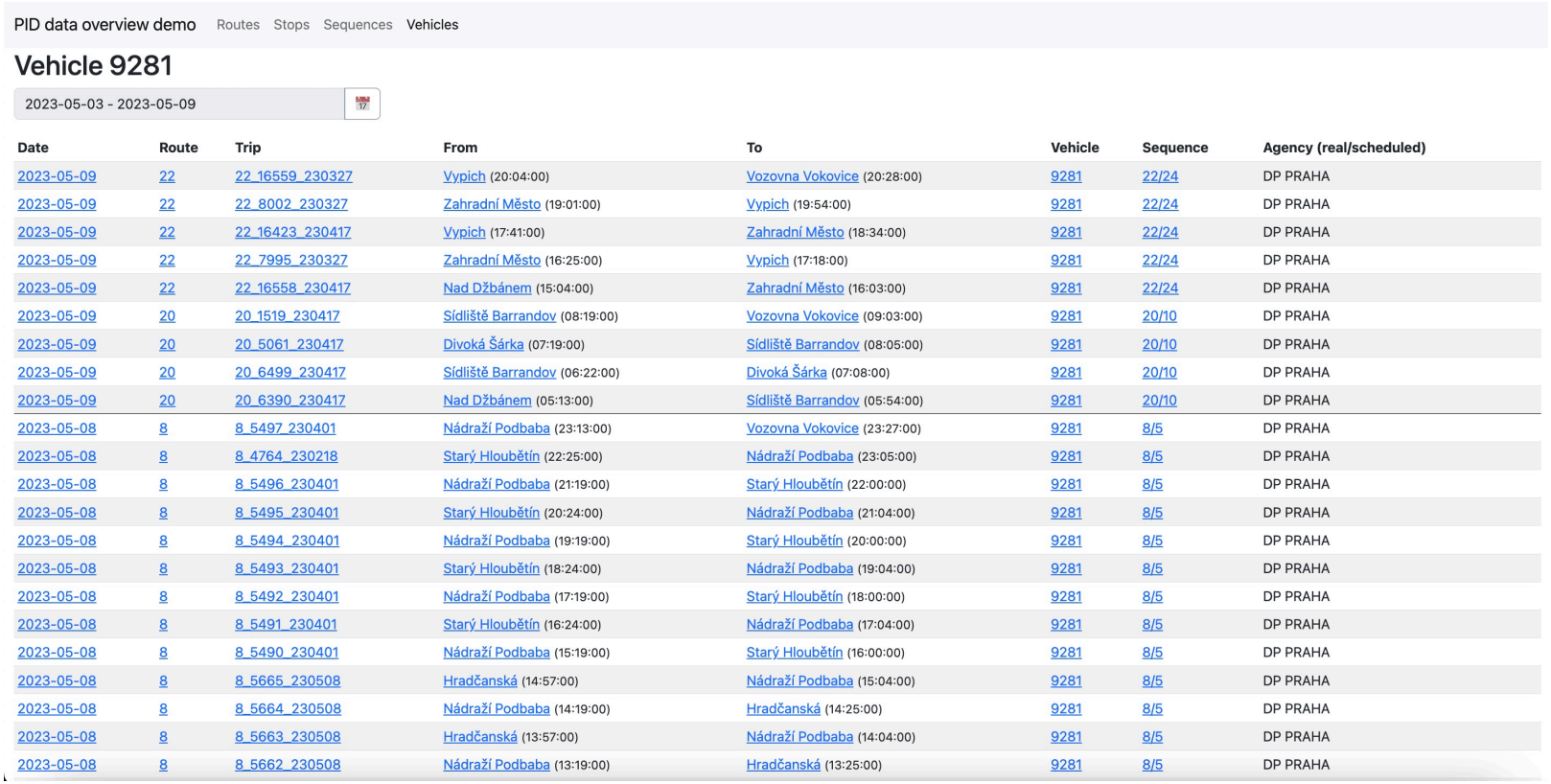
Příloha A

Snímky výsledné webové aplikace



Obrázek A.1 Hlavní stránka

Obrázek A.2 Seznam linek



PID data overview demo Routes Stops Sequences Vehicles

Vehicle 9281

2023-05-03 - 2023-05-09

Date	Route	Trip	From	To	Vehicle	Sequence	Agency (real/scheduled)
2023-05-09	22	22_16559_230327	Vypich (20:04:00)	Vozovna Vokovice (20:28:00)	9281	22/24	DP PRAHA
2023-05-09	22	22_8002_230327	Zahradní Město (19:01:00)	Vypich (19:54:00)	9281	22/24	DP PRAHA
2023-05-09	22	22_16423_230417	Vypich (17:41:00)	Zahradní Město (18:34:00)	9281	22/24	DP PRAHA
2023-05-09	22	22_7995_230327	Zahradní Město (16:25:00)	Vypich (17:18:00)	9281	22/24	DP PRAHA
2023-05-09	22	22_16558_230417	Nad Džbánem (15:04:00)	Zahradní Město (16:03:00)	9281	22/24	DP PRAHA
2023-05-09	20	20_1519_230417	Sídliště Barrandov (08:19:00)	Vozovna Vokovice (09:03:00)	9281	20/10	DP PRAHA
2023-05-09	20	20_5061_230417	Divoká Šárka (07:19:00)	Sídliště Barrandov (08:05:00)	9281	20/10	DP PRAHA
2023-05-09	20	20_6499_230417	Sídliště Barrandov (06:22:00)	Divoká Šárka (07:08:00)	9281	20/10	DP PRAHA
2023-05-09	20	20_6390_230417	Nad Džbánem (05:13:00)	Sídliště Barrandov (05:54:00)	9281	20/10	DP PRAHA
2023-05-08	8	8_5497_230401	Nádraží Podbaba (23:13:00)	Vozovna Vokovice (23:27:00)	9281	8/5	DP PRAHA
2023-05-08	8	8_4764_230218	Starý Hloubětín (22:25:00)	Nádraží Podbaba (23:05:00)	9281	8/5	DP PRAHA
2023-05-08	8	8_5496_230401	Nádraží Podbaba (21:19:00)	Starý Hloubětín (22:00:00)	9281	8/5	DP PRAHA
2023-05-08	8	8_5495_230401	Starý Hloubětín (20:24:00)	Nádraží Podbaba (21:04:00)	9281	8/5	DP PRAHA
2023-05-08	8	8_5494_230401	Nádraží Podbaba (19:19:00)	Starý Hloubětín (20:00:00)	9281	8/5	DP PRAHA
2023-05-08	8	8_5493_230401	Starý Hloubětín (18:24:00)	Nádraží Podbaba (19:04:00)	9281	8/5	DP PRAHA
2023-05-08	8	8_5492_230401	Nádraží Podbaba (17:19:00)	Starý Hloubětín (18:00:00)	9281	8/5	DP PRAHA
2023-05-08	8	8_5491_230401	Starý Hloubětín (16:24:00)	Nádraží Podbaba (17:04:00)	9281	8/5	DP PRAHA
2023-05-08	8	8_5490_230401	Nádraží Podbaba (15:19:00)	Starý Hloubětín (16:00:00)	9281	8/5	DP PRAHA
2023-05-08	8	8_5665_230508	Hradčanská (14:57:00)	Nádraží Podbaba (15:04:00)	9281	8/5	DP PRAHA
2023-05-08	8	8_5664_230508	Nádraží Podbaba (14:19:00)	Hradčanská (14:25:00)	9281	8/5	DP PRAHA
2023-05-08	8	8_5663_230508	Hradčanská (13:57:00)	Nádraží Podbaba (14:04:00)	9281	8/5	DP PRAHA
2023-05-08	8	8_5662_230508	Nádraží Podbaba (13:19:00)	Hradčanská (13:25:00)	9281	8/5	DP PRAHA

Obrázek A.3 Detail vozidla

PID data overview demo Routes Stops Sequences Vehicles

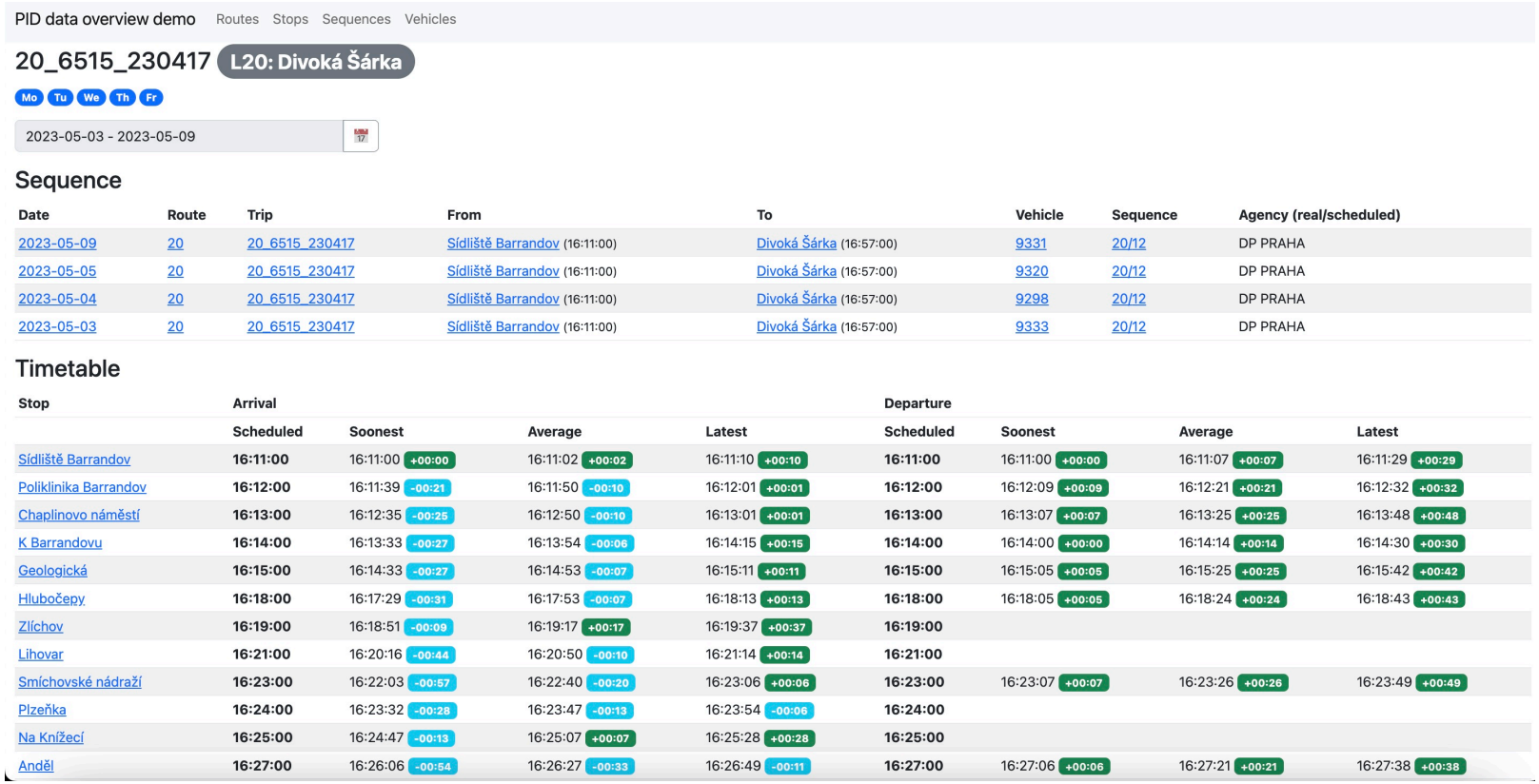
Hradčanská platform B

1 2 A B C D F L Z2

2023-05-03 – 2023-05-09

Trip				Arrival					Departure					
Route	Trip	From	To	Scheduled	Soonest		Average		Latest		Scheduled	Soonest	Average	Latest
25	25_7139_230213	Lehovec (23:32:00)	Bílá Hora (00:19:00)	**00:03:00**	00:02:30	-00:30	00:03:11	+00:11	00:03:45	+00:45	**00:03:00**	00:03:11 +00:11	00:03:48 +00:48	00:04:27 +01:27
2	2_6253_230401	Podolská vodárna (23:41:00)	Sídliště Petřiny (00:16:00)	**00:04:00**	00:03:40	-00:20	00:05:46	+01:46	00:09:08	+05:08	**00:04:00**	00:04:24 +00:24	00:06:20 +02:20	00:09:36 +05:36
26	26_7831_230415	Nádraží Hostivař (23:26:00)	Divoká Šárka (00:24:00)	**00:08:00**	00:07:26	-00:34	00:08:03	+00:03	00:09:15	+01:15	**00:08:00**	00:08:06 +00:06	00:08:42 +00:42	00:09:47 +01:47
18	18_5724_221107	Vozovna Pankrác (23:41:00)	Nádraží Podbaba (00:17:00)	**00:10:00**	00:10:05	+00:05	00:10:23	+00:23	00:10:43	+00:43	**00:10:00**	00:10:37 +00:37	00:10:58 +00:58	00:11:23 +01:23
18	18_5724_230430	Vozovna Pankrác (23:41:00)	Nádraží Podbaba (00:17:00)	**00:10:00**	00:09:48	-00:12	00:10:06	+00:06	00:10:35	+00:35	**00:10:00**	00:10:25 +00:25	00:10:41 +00:41	00:11:10 +01:10
1	1_6705_230121	Sídliště Ďáblice (23:42:00)	Sídliště Petřiny (00:25:00)	**00:13:00**	00:13:18	+00:18	00:14:12	+01:12	00:15:39	+02:39	**00:13:00**	00:13:00 +00:00	00:14:21 +01:21	00:16:16 +03:16
20	20_1191_230415	Sídliště Barrandov (23:50:00)	Vozovna Vokovice (00:32:00)	**00:18:00**	00:17:36	-00:24	00:18:52	+00:52	00:20:30	+02:30	**00:18:00**	00:18:00 +00:00	00:19:10 +01:10	00:21:04 +03:04
8	8_4750_230213	Starý Hloubětín (23:45:00)	Nádraží Podbaba (00:25:00)	**00:18:00**	00:17:46	-00:14	00:19:02	+01:02	00:21:58	+03:58	**00:18:00**	00:18:00 +00:00	00:19:18 +01:18	00:22:31 +04:31
25	25_7141_230213	Lehovec (23:52:00)	Bílá Hora (00:39:00)	**00:23:00**	00:22:37	-00:23	00:23:17	+00:17	00:24:52	+01:52	**00:23:00**	00:23:00 +00:00	00:23:52 +00:52	00:25:27 +02:27
1	1_6706_230121	Sídliště Ďáblice (00:02:00)	Sídliště Petřiny (00:45:00)	**00:33:00**	00:32:22	-00:38	00:33:14	+00:14	00:34:03	+01:03	**00:33:00**	00:33:00 +00:00	00:33:44 +00:44	00:34:38 +01:38
91	91_2_230415	Radošovická (23:45:00)	Divoká Šárka (00:52:00)	**00:35:00**	00:34:47	-00:13	00:35:22	+00:22	00:37:01	+02:01	**00:35:00**	00:35:00 +00:00	00:35:54 +00:54	00:37:55 +02:55
97	97_759_230326	Nádraží Hostivař (23:54:00)	Bílá Hora (01:00:00)	**00:44:00**	00:44:03	+00:03	00:45:02	+01:02	00:46:31	+02:31	**00:44:00**	00:44:54 +00:54	00:45:50 +01:50	00:47:10 +03:10
96	96_681_230201	Spořilov (00:05:00)	Sídliště Petřiny (01:01:00)	**00:49:00**	00:48:18	-00:42	00:49:34	+00:34	00:50:54	+01:54	**00:49:00**	00:49:00 +00:00	00:50:04 +01:04	00:51:43 +02:43
91	91_6_230415	Radošovická (00:15:00)	Divoká Šárka (01:21:00)	**01:05:00**	01:04:28	-00:32	01:05:30	+00:30	01:06:00	+01:00	**01:05:00**	01:05:06 +00:06	01:06:15 +01:15	01:06:49 +01:49
97	97_762_230326	Nádraží Hostivař (00:24:00)	Bílá Hora (01:30:00)	**01:14:00**	01:13:45	-00:15	01:15:00	+01:00	01:17:15	+03:15	**01:14:00**	01:14:00 +00:00	01:15:16 +01:16	01:17:49 +03:49
96	96_683_230201	Spořilov (00:35:00)	Sídliště Petřiny (01:31:00)	**01:19:00**	01:18:18	-00:42	01:19:01	+00:01	01:19:30	+00:30	**01:19:00**	01:19:00 +00:00	01:19:44 +00:44	01:20:23 +01:23
91	91_10_230415	Radošovická (00:45:00)	Divoká Šárka (01:52:00)	**01:35:00**	01:34:21	-00:39	01:35:17	+00:17	01:37:28	+02:28	**01:35:00**	01:35:04 +00:04	01:35:57 +00:57	01:38:12 +03:12
97	97_765_230327	Nádraží Hostivař (00:54:00)	Bílá Hora (02:00:00)	**01:44:00**	01:44:27	+00:27	01:45:03	+01:03	01:45:45	+01:45	**01:44:00**	01:44:59 +00:59	01:45:39 +01:39	01:46:21 +02:21
96	96_676_230201	Spořilov (01:05:00)	Sídliště Petřiny (02:01:00)	**01:49:00**	01:48:30	-00:30	01:48:50	-00:10	01:49:10	+00:10	**01:49:00**	01:49:00 +00:00	01:49:20 +00:20	01:49:45 +00:45

Obrázek A.4 Agregace zpoždění dle zastávky



PID data overview demo Routes Stops Sequences Vehicles

20_6515_230417 L20: Divoká Šárka

Mo Tu We Th Fr

2023-05-03 - 2023-05-09

Sequence

Date	Route	Trip	From	To	Vehicle	Sequence	Agency (real/scheduled)
2023-05-09	20	20_6515_230417	Sídliště Barrandov (16:11:00)	Divoká Šárka (16:57:00)	9331	20/12	DP PRAHA
2023-05-05	20	20_6515_230417	Sídliště Barrandov (16:11:00)	Divoká Šárka (16:57:00)	9320	20/12	DP PRAHA
2023-05-04	20	20_6515_230417	Sídliště Barrandov (16:11:00)	Divoká Šárka (16:57:00)	9298	20/12	DP PRAHA
2023-05-03	20	20_6515_230417	Sídliště Barrandov (16:11:00)	Divoká Šárka (16:57:00)	9333	20/12	DP PRAHA

Timetable

Stop	Arrival				Departure			
	Scheduled	Soonest	Average	Latest	Scheduled	Soonest	Average	Latest
Sídliště Barrandov	**16:11:00**	16:11:00 +00:00	16:11:02 +00:02	16:11:10 +00:10	**16:11:00**	16:11:00 +00:00	16:11:07 +00:07	16:11:29 +00:29
Poliklinika Barrandov	**16:12:00**	16:11:39 -00:21	16:11:50 -00:10	16:12:01 +00:01	**16:12:00**	16:12:09 +00:09	16:12:21 +00:21	16:12:32 +00:32
Chaplinovo náměstí	**16:13:00**	16:12:35 -00:25	16:12:50 -00:10	16:13:01 +00:01	**16:13:00**	16:13:07 +00:07	16:13:25 +00:25	16:13:48 +00:48
K Barrandovu	**16:14:00**	16:13:33 -00:27	16:13:54 -00:06	16:14:15 +00:15	**16:14:00**	16:14:00 +00:00	16:14:14 +00:14	16:14:30 +00:30
Geologická	**16:15:00**	16:14:33 -00:27	16:14:53 -00:07	16:15:11 +00:11	**16:15:00**	16:15:05 +00:05	16:15:25 +00:25	16:15:42 +00:42
Hlubočepy	**16:18:00**	16:17:29 -00:31	16:17:53 -00:07	16:18:13 +00:13	**16:18:00**	16:18:05 +00:05	16:18:24 +00:24	16:18:43 +00:43
Zlíchov	**16:19:00**	16:18:51 -00:09	16:19:17 +00:17	16:19:37 +00:37	**16:19:00**			
Lihovar	**16:21:00**	16:20:16 -00:44	16:20:50 -00:10	16:21:14 +00:14	**16:21:00**			
Smíchovské nádraží	**16:23:00**	16:22:03 -00:57	16:22:40 -00:20	16:23:06 +00:06	**16:23:00**	16:23:07 +00:07	16:23:26 +00:26	16:23:49 +00:49
Plzeňka	**16:24:00**	16:23:32 -00:28	16:23:47 -00:13	16:23:54 -00:06	**16:24:00**			
Na Knížecí	**16:25:00**	16:24:47 -00:13	16:25:07 +00:07	16:25:28 +00:28	**16:25:00**			
Anděl	**16:27:00**	16:26:06 -00:54	16:26:27 -00:33	16:26:49 -00:11	**16:27:00**	16:27:06 +00:06	16:27:21 +00:21	16:27:38 +00:38

Obrázek A.5 Agregace zpoždění dle výjezdu (tripu)

Bibliografie

1. MOBILITYDATA. *General Transit Feed Specification* [online]. [cit. 2023-04-10]. Dostupné z: `https://gtfs.org/`.
2. GOOGLE. *GTFS Static* [online]. Google [cit. 2023-04-14]. Dostupné z: `https://developers.google.com/transit/gtfs`.
3. GOOGLE. *GTFS Realtime* [online]. Google [cit. 2023-04-14]. Dostupné z: `https://developers.google.com/transit/gtfs-realtime`.
4. GOOGLE. *Protocol Buffers* [online]. Google [cit. 2023-04-16]. Dostupné z: `https://protobuf.dev/`.
5. ROPID. *Pražská integrovaná doprava* [online]. 2023. [cit. 2023-04-16]. Dostupné z: `https://pid.cz`.
6. OPERÁTOR ICT A.S.; GOLEMIO; PORTÁL HLAVNÍHO MĚSTA PRAHY. *Open data hlavního města Prahy* [online]. [cit. 2023-04-19]. Dostupné z: `https://opendata.praha.eu`.
7. GOLEMIO. *Golemio: Úvod* [online]. 2021. [cit. 2023-04-03]. Dostupné z: `https://golemio.cz/`.
8. IBM. *What is an application programming interface (API)* [online]. 2023. [cit. 2023-04-17]. Dostupné z: `https://www.ibm.com/topics/api`.
9. GOLEMIO. *Public Transport | Golemio API* [online]. [cit. 2023-04-19]. Dostupné z: `https://api.golemio.cz/v2/pid/docs/openapi`.
10. HÁNA, František; KNOTEK, Daniel. *PID polohy API* [e-mail]. 2023. [cit. 2023-04-07].
11. JIRÁČEK, Zbyněk. *Predikce odjezdů v GTFS-Realtime po jednotlivých zastávkách* [online]. GitLab Inc, 2023-03 [cit. 2023-04-27]. Dostupné z: `https://gitlab.com/operator-ict/golemio/code/modules/pid/-/issues/235`.
12. CREATIVE COMMONS. *Creative Commons – CC BY 4.0* [online]. [cit. 2023-04-27]. Dostupné z: `https://creativecommons.org/licenses/by/4.0/`.
13. GOLEMIO. *API Keys Management | Licencování* [online]. [cit. 2023-04-19]. Dostupné z: `https://api.golemio.cz/api-keys/terms-and-conditions`.

14. PRAŽSKÁ INTEGROVANÁ DOPRAVA. *Mapa PID* [online]. [cit. 2023-04-19]. Dostupné z: https://mapa.pid.cz/.
15. OPERÁTOR ICT A.S. *PID Lítačka* [online]. [cit. 2023-04-19]. Dostupné z: https://app.pidlitacka.cz/.
16. BABILON, Robert. *Babitron | Zpoždění vlaků* [online]. [cit. 2023-04-19]. Dostupné z: https://kam.mff.cuni.cz/~babilon/zpmapa2.html.
17. KOŇAŘÍK, David. *RtView* [online]. [cit. 2023-04-19]. Dostupné z: https://rt.jrutil.konarici.cz/.
18. ARMOUR, F.; MILLER, G. *Advanced Use Case Modeling: Software Systems*. Addison-Wesley, 2001. Addison-Wesley object technology series, č. sv. 1. ISBN 9780201615920. Dostupné také z: https://books.google.cz/books?id=VINuQgAACAAJ.
19. BLINOWSKI, Grzegorz; OJDOWSKA, Anna; PRZYBYŁEK, Adam. Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation. *IEEE Access* [online]. 2022, roč. 10, s. 20357–20374 [cit. 2023-04-14]. Dostupné z DOI: 10.1109/ACCESS.2022.3152803.
20. ORACLE. *Oracle Database Express Edition* [online]. 2023. [cit. 2023-04-13]. Dostupné z: https://www.oracle.com/database/technologies/appdev/xe.html.
21. THE POSTGRESQL GLOBAL DEVELOPMENT GROUP. *PostgreSQL: Documentation: 15: 41.3. Materialized Views* [online]. 2023-02. [cit. 2023-05-01]. Dostupné z: https://www.postgresql.org/docs/current/rules-materializedviews.html.
22. GOOGLE. *GTFS Static | stop_times.txt* [online]. [cit. 2023-05-01]. Dostupné z: https://developers.google.com/transit/gtfs/reference#stop%5C_timestxt.
23. KIMBALL, R.; CASERTA, J. *The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data*. Wiley, 2011. ISBN 9781118079683. Dostupné také z: https://books.google.cz/books?id=TCLfzU2ilVkC.
24. ZODE, Madhu. The Evolution of ETL [online]. [B.r.], roč. 6, č. 06 [cit. 2023-04-10]. Dostupné z: http://hosteddocs.ittoolbox.com/mz071807b.pdf.
25. RODDEWIG, Stephen. *18 Best ETL Tools for 2023* [online]. 2022-08. [cit. 2023-05-01]. Dostupné z: https://blog.hubspot.com/website/etl-tools.
26. GRADLE. *Gradle Build Tool* [online]. 2023. [cit. 2023-04-14]. Dostupné z: https://gradle.org/.
27. VMWARE. *Spring Framework Documentation* [online]. 2023. [cit. 2023-04-13]. Dostupné z: https://docs.spring.io/spring-framework/docs/6.0.x/reference/html.
28. LTD, Red Gate Software. *Homepage - Flyway* [online]. 2020-12. [cit. 2023-04-28]. Dostupné z: https://flywaydb.org/.
29. LIQUIBASE. *The Liquibase Community: The Database DevOps Community* [online]. 2022-10. [cit. 2023-04-28]. Dostupné z: https://www.liquibase.org/.

30. MASSE, M. *REST API Design Rulebook*. O'Reilly Media, 2011. Oreilly and Associate Series. ISBN 9781449310509. Dostupné také z: https://books.google.cz/books?id=4lZcsRwXo6MC.

31. VMWARE. *Spring | Web Applications* [online]. [cit. 2023-05-01]. Dostupné z: https://spring.io/web-applications.

32. SMARTBEAR SOFTWARE. *API Documentation & Design Tools for Teams | Swagger* [online]. 2023. [cit. 2023-05-01]. Dostupné z: https://swagger.io/.

33. PLURALSIGHT. *Technology Index* [online]. [cit. 2023-05-06]. Dostupné z: https://www.pluralsight.com/tech-index.

34. MICROSOFT. *TypeScript: JavaScript With Syntax For Types* [online]. [cit. 2023-05-06]. Dostupné z: https://www.typescriptlang.org/.

35. META OPEN SOURCE. *React* [online]. [cit. 2023-05-06]. Dostupné z: https://www.typescriptlang.org/.

36. FIGMA INC. *Figma: the collaborative interface design tool.* [online]. [cit. 2023-05-08]. Dostupné z: https://www.figma.com/.

37. DOCKER INC. *Accelerated, containerized application development* [online]. 2023-04. [cit. 2023-04-28]. Dostupné z: https://www.docker.com/.

38. GITLAB B.V. *The DevSecOps Platform | GitLab* [online]. 2023. [cit. 2023-04-27]. Dostupné z: https://gitlab.com/.

39. CHACON, Scott; LONG, Jason. *Git* [online]. [cit. 2023-04-29]. Dostupné z: https://git-scm.com/.

40. GITLAB. *CI/CD Concepts* [online]. [cit. 2023-05-02]. Dostupné z: https://docs.gitlab.com/ee/ci/introduction/index.html.

41. THE POSTGRESQL GLOBAL DEVELOPMENT GROUP. *PostgreSQL :Documentation: 15: Chapter 11. Indexes* [online]. 2023-02. [cit. 2023-05-02]. Dostupné z: https://www.postgresql.org/docs/current/indexes.html.

42. DOCKER INC. *Docker Compose overview* [online]. 2023-05. [cit. 2023-05-02]. Dostupné z: https://docs.docker.com/compose/.

43. THE PROJECT LOMBOK AUTHORS. *Project Lombok* [online]. [cit. 2023-05-03]. Dostupné z: https://projectlombok.org/.

44. TRANTINA, Jonáš. *Studie rámce Lombok* [online]. 2014. [cit. 2023-05-03]. Dostupné z: https://is.muni.cz/th/jve2g/.

45. MVNREPOSITORY. *Maven Repository: org.postgresql » postgresql* [online]. [cit. 2023-05-03]. Dostupné z: https://mvnrepository.com/artifact/org.postgresql/postgresql.

46. GULATI, S.; SHARMA, R. *Java Unit Testing with JUnit 5: Test Driven Development with JUnit 5*. Apress, 2017. ISBN 9781484230152. Dostupné také z: https://books.google.cz/books?id=pR0-DwAAQBAJ.

47. THE JUNIT TEAM. *JUnit 5* [online]. [cit. 2023-05-04]. Dostupné z: https://junit.org/junit5/.

48. FABER, Szczepan. *Mockito framework site* [online]. [cit. 2023-05-04]. Dostupné z: https://site.mockito.org/.

49. ALPINE LINUX DEVELOPMENT TEAM. *Alpine Linux* [online]. [cit. 2023-05-04]. Dostupné z: https://www.alpinelinux.org/.

50. ORACLE. *OpenJDK* [online]. [cit. 2023-05-04]. Dostupné z: https://openjdk.org/.

51. INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. *ISO - ISO 8601 – Date and time format* [online]. [cit. 2023-05-05]. Dostupné z: https://www.iso.org/iso-8601-date-and-time-format.html.

52. SARJEANT, J. J.; ZABRISKIE, M. *Axios* [online]. [cit. 2023-05-06]. Dostupné z: https://www.axios-http.com/.

53. LINSLEY, Tanner. *TankStack Query / React Query* [online]. [cit. 2023-05-06]. Dostupné z: https://www.tanstack.com/query/v3/.

54. REMIX SOFTWARE INC. *React Router* [online]. [cit. 2023-05-06]. Dostupné z: https://www.reactrouter.com/.

Obsah přiloženého archivu

- `readme.md` ... popis obsahu archivu
- `thesis` ... soubory k textu práce
  - `src` ... zdrojová forma práce ve formátu LaTeX
  - `zibriale_thesis.pdf` ... text práce
- `src` ... zdrojové kódy implementace
  - `pid`
    - `api` ... zdrojové kódy API komponenty
    - `db` ... zdrojové kódy pro inicializaci databáze
    - `loader` ... zdrojové kódy loader komponenty
    - `www` ... zdrojové kódy webové stránky