Porovnání a implementace GraphQL vs. REST API v prostředí PHP

Bc. Filip Růžička

Diplomová práce
2022

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2021/2022

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Filip Růžička**
Osobní číslo:	**A20606**
Studijní program:	**N0613A140022 Informační technologie**
Specializace:	**Softwarové inženýrství**
Forma studia:	**Kombinovaná**
Téma práce:	**Porovnání a implementace GraphQL vs. REST API v prostředí PHP**
Téma práce anglicky:	**Comparison and Implementation of GraphQL vs. REST API in PHP Environment**

Zásady pro vypracování

1. Analyzujte problematiku a vypracujte literární rešerši na dané téma.
2. Popište technologie REST API a GraphQL.
3. Navrhněte sérii ukázkových aplikací.
4. Implementujte navržené aplikace s praktickými ukázkami.
5. Vyhodnoťte výhody a nevýhody obou prezentovaných nástrojů.

Seznam doporučené literatury:

1. PORCELLO, Eve a Alex BANKS. *Learning GraphQL: declarative data fletching for modern web apps*. Sebastopol,CA: O´Reilly, 2018. ISBN 978-1-492-03071-3.
2. BRITO, Gleison; VALENTE, Marco Tulio. Rest vs graphql: A controlled experiment. In: *2020 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2020. p. 81-91. ISBN 978-1-7281-4659-1.
3. DOGLIO, Fernando. *REST API Development with Node.js: Manage and Understand the Full Capabilities of Successful REST Development*. 2nd ed. 2018. Imprint: Apress, 2018. ISBN 978-1-4842-3715-1.
4. MASSE, Mark. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. „ O'Reilly Media, Inc.", 2011. ISBN 978-1-449-31050-9.
5. BIEHL, Matthias. *GraphQL API Design: API-University Series #5*. API-University Press, 2019. ISBN 1230003164303.
6. BIEHL, Matthias. *API Architecture: The Big Picture for Building APIs: API-University Series #2*. CreateSpace Independent Publishing Platform, 2015. ISBN 150867664X.
7. WIERUCH, Robin. *The Road to GraphQL: Your journey to master GraphQL in JavaScript*. Independently published, 2018. ISBN 1730853935.
8. BUNA, Samer. *GraphQL in Action*. Manning Publications, 2021. ISBN 1730853935.
9. GraphQL [online]. [cit. 2021-11-14]. Dostupné z: https://graphql.org/.

Vedoucí diplomové práce: **doc. Ing. Zdenka Prokopová, CSc.**
Ústav počítačových a komunikačních systémů

Datum zadání diplomové práce: **3. prosince 2021**
Termín odevzdání diplomové práce: **23. května 2022**

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
L.S.

doc. Mgr. Milan Adámek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 24. ledna 2022

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne

Filip Růžička, v. r.
podpis studenta

ABSTRAKT

Práce se zabývá problematikou práce s daty a jejím přenosem s využitím nástrojů REST API a GraphQL. Obě technologie jsou v teoretické části podrobně popsány a vysvětleny, včetně jejich výhod, nevýhod a nedostatků. Praktická část práce zahrnuje samotnou implementaci technologií v praxi a jejich porovnání. Funkcionalitu těchto nástrojů nejlépe demonstrují vytvořené aplikace, které se opírají o nabyté znalosti a poznatky z teoretické části.

Klíčová slova:

REST API, GraphQL, PHP, HTTP, databáze

ABSTRACT

The work deals with the issue of working with data and its transfer using the REST API and GraphQL tools. Both technologies are in detail described and explained in the theoretical part of this thesis, including their advantages, disadvantages and shortcomings. The practical part of the thesis includes the implementation of both technologies in practice and their comparison. The functionality of these technologies is best demonstrated by the created applications, which are based on the acquired knowledge and insights from the theoretical part.

Keywords:

REST API, GraphQL, PHP, HTTP, database

Na úvod práce bych rád poděkoval paní doc. Ing. Zdenka Prokopová, CSc. za přínosné podněty, rady a celkové vedení diplomové práce.

Prohlašuji, že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD

Předkládaná diplomová práce vznikla jako pokus o optimalizaci webové služby, která využívá pro přenos dat technologii REST. Avšak díky své velikosti a opakovanému zjištění, kdy se u technologie REST API začaly dostavovat její technologické nedostatky a nevýhody. Ačkoliv tato technologie stále disponuje enormním zastoupením v komerčním i nekomerčním použití. Přesto se jedná o starší technologii a v současnosti existují i jiné, především novější, které přistupují k práci s daty a jejich přenosu rozdílně. Z toho důvodu se nabízí částečné, popřípadě úplné nahrazení REST technologie za jinou.

Zároveň je zapotřebí zmínit pokrok ve vývoji restových aplikací za poslední dekádu. To znamená například pozvolný přechod z webu 2.0 na web 3.0, kdy se aplikace a webové stránky stávají čím dál více interaktivní a obsahují větší množství obsahu. S tím je spojena potřeba načítat více dat. Jako jeden z dalších důvodů se nabízí vznik průmyslu 4.0 a celková digitalizace korporátních, státních a soukromých infrastruktur. Všechny chytré továrny v podstatě neustále načítají a pracují s daty. Posledním důvodem lze definitivně označit vývoj v oblasti mobilních aplikací, především však v posledních letech, kdy mobilní aplikace jsou v podstatě hybridní webové aplikace využívající API pro přenos dat. I obyčejný člověk tedy cítí, že ve vzdáleném přenosu dat nastává velký posun a lidé jsou s touto technologií v kontaktu každý den. Zároveň ale i v komerčním prostředí je cítit velký tlak a očekávání, přenos dat tedy musí být co nejrychlejší, nejbezpečnější a nejjednodušší. Proto technologie, která byla dostačující před 15 lety, nemusí dnes stačit.

Jednou z možných náhrad REST se jeví nástroj GraphQL. Jedná se o poměrně novou technologii, která se snaží vymezit nedostatků jeho starší alternativy. Přičemž aby mohla být nahrazena, proto jsou v práci obě technologie podrobně porovnány. Současně existují aplikace s identickou funkčností, které jsou mezi sebou porovnány a otestovány.

I. TEORETICKÁ ČÁST

1 HISTORIE A EVOLUCE SLUŽEB

Pod pojmem služba si lze představit principy reálného světa, kdy kupříkladu benzínovou pumpu můžeme označit jako službu, která se skládá z jednotlivých akcí/kolekcí (čerpání paliva, platba, mytí vozu). Službu lze definovat jako kontejner, jenž obsahuje společné možné události a akce. Její součástí by měl být i veřejný výčet všech těchto událostí, které jsou k dispozici.

V IT služba může obecně nabízet API, které poskytuje kolekci dalších možností. Tyto kolekce jsou seskupeny k sobě, protože mají společnou vlastnost, díky které jsou relevantní, tj. například společná funkcionalita, obor, autor. Na druhé straně služby existuje konzument dané služby, ten při vyvolání akce aktivuje požadavek na službu. Služba posléze začne s obsluhou přidělené logiky k dané akci. Konzumentem (klientem) lze definovat koncového uživatele, softwarový program nebo jiný vstup. [1]

Logika služeb se dá rozdělit do dvou základních kategorií, tj. agnostické a neagnostické. Pod agnostickou logiku spadá služba tehdy, pokud jí můžeme opakovaně použít v situacích s různým kontextem. To znamená, že používáním stejné služby vzniknou aplikace a principy s odlišnými účely. Naopak neagnostická logika prezentuje službu, která byla vytvořena za jedním účelem. Takovou službu je možno opakovaně používat, avšak pouze v případech, které si jsou koncepčně podobné (dva různé softwarové programy, avšak spadající do stejného odvětví). [1]

1.1 Webové služby

V průběhu vývoje softwaru se brzy zjistilo, že stav, kdy jeden uživatel pracuje na jednom zařízení s jednou aplikací je nedostačující. Z toho důvodu se začaly vyvíjet multiplatformní a vzdálené aplikace, které se tento problém snažily vyřešit. Možnost pracovat s daty odkudkoliv však přináší několik problémů, které jsou spojené především se vzájemnou kompatibilitou dat a bezpečností přenosu. V posledních dvou dekádách se začal využívat model založený na ontologické integraci dat. Jedná se model architektury, který obsahují definovanou doménu, jenž pracuje s daty z různých zdrojů. Tento model bývá definován podle popisujícího jazyka jako například resource description language (RDL) nebo web ontology language (OWL). Díky tomu jsou definice jasně a unikátně vyobrazeny a zároveň mohou být jednoduše rozšířeny. V doménách se poté implementují webové služby. [2]

Webová služba označuje službu, která využívá internetový přenos pro vzájemnou komunikaci mezi zařízeními, bez ohledu na zvoleném operačním systému nebo programovacím jazyku. Obecně by webová služba měla obsahovat následujících 5 vlastností:

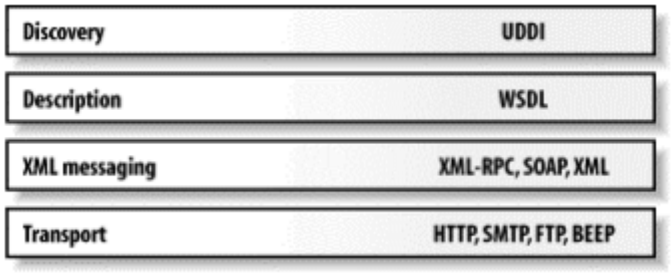
1. Být vzdáleně dostupná
2. Využívat standardizovaný systém pro komunikaci
3. Být nezávislý na OS či vyvíjeném jazyce
4. Být snadno vyhledatelná
5. Být sebe popisující [3]

Ve webových službách a celkově softwarovém inženýrství se často využívá princip separation of concerns (SOC). Smysl principu spočívá v rozdělení problémů na podproblémy, které obsahují rozdílné typy funkcionalit. Rozdělené části obsahují přizpůsobitelné, robustní a znovupoužitelné akce. Funkce jednotlivých akcí by se neměly překrývat. V objektovém programování za části (podproblémy) lze považovat objekty/třídy, a za celý podproblém use case diagram nebo databázové schéma. [4]

1.1.1 Rozdělení

Architektura webové služby se rozděluje následovně (Obrázek 1):

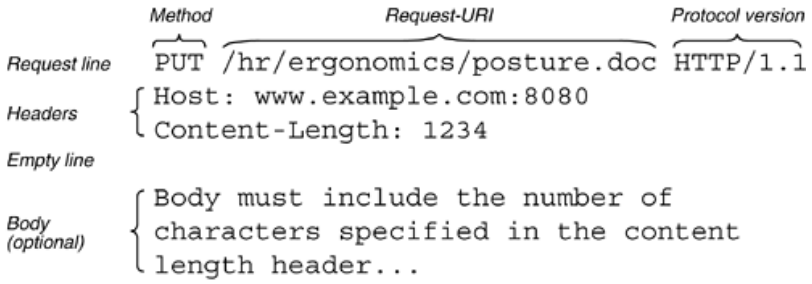
- Role
    - Poskytovatel služby
    - Žadatel služby
    - Registr služby [3]
- Zásobník protokolů
    - Přenos služby
    - XML messaging
    - Popis služby
    - Objevení služby [3]



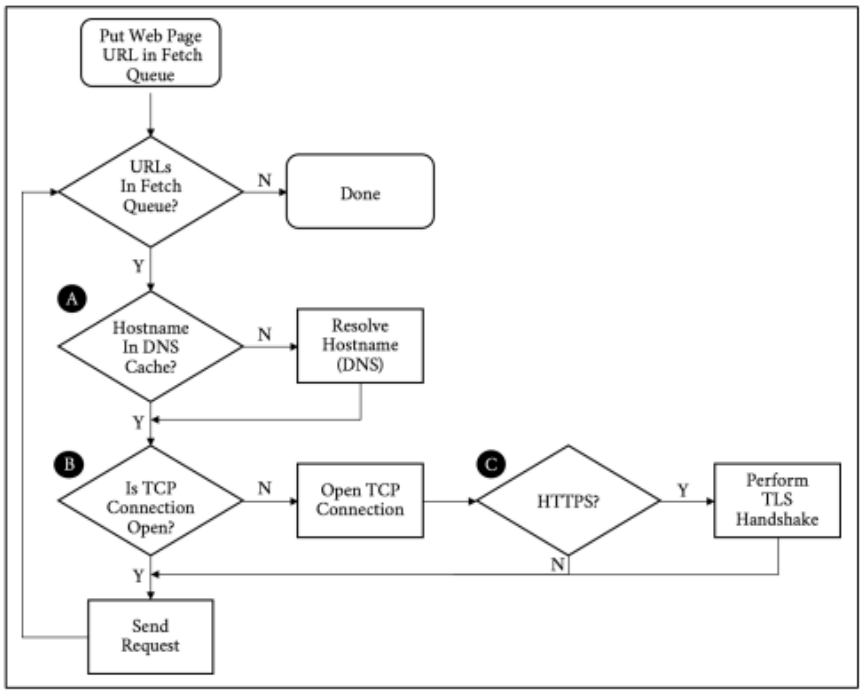
Obrázek 1. Zásobník protokolů [3]

1.2 HTTP

Hypertext Transfer Protocol (HTTP) zaštiťuje komunikaci a přenos dat mezi webovým serverem a webovými službami nebo aplikacemi. HTTP využívá protokoly pro přenos dat, které zajišťují bezpečný přenos, bez ztráty jejich integrity. Princip spočívá v odeslání dat přes zmíněný protokol ze serveru, pokud si o ně klient zažádá. Klient po serveru požaduje různou formu obsah, kterou se server snaží doručit. Pokud klient zašle požadavek na server, jeho součástí jsou popisné atributy umístěné v jeho hlavičce (Obrázek 2). Tyto atributy určují vlastnosti samotného dotazu. Existují 2 typy HTTP dotazu, tj. požadavek (dotaz směřující od klienta k serveru) a odpověď (směřující od serveru ke klientovi). Oba typy dotazu obsahují 3 kategorie: úvod, hlavičku a tělo. HTTP využívá pro přenos TCP/IP protokol, který zajišťuje včasné doručení bez chyb. Jakmile se otevře TCP spojení mezi klientem a serverem, nikdy nedojde ke ztrátě či poškození dat (Obrázek 3). [5]



Obrázek 2. Struktura HTTP požadavku [6]



Obrázek 3. Vývojový diagram HTTP požadavku [7]

Typy HTTP protokolů:

- GET – Načítání a zobrazení zdrojů
- POST – Ukládání zdrojů
- PUT – Aktualizace celého zdroje
- PATCH – Aktualizace části zdroje
- DELETE – Odstranění zdrojů
- HEAD – Odešle pouze HTTP hlavičku požadavku [5]

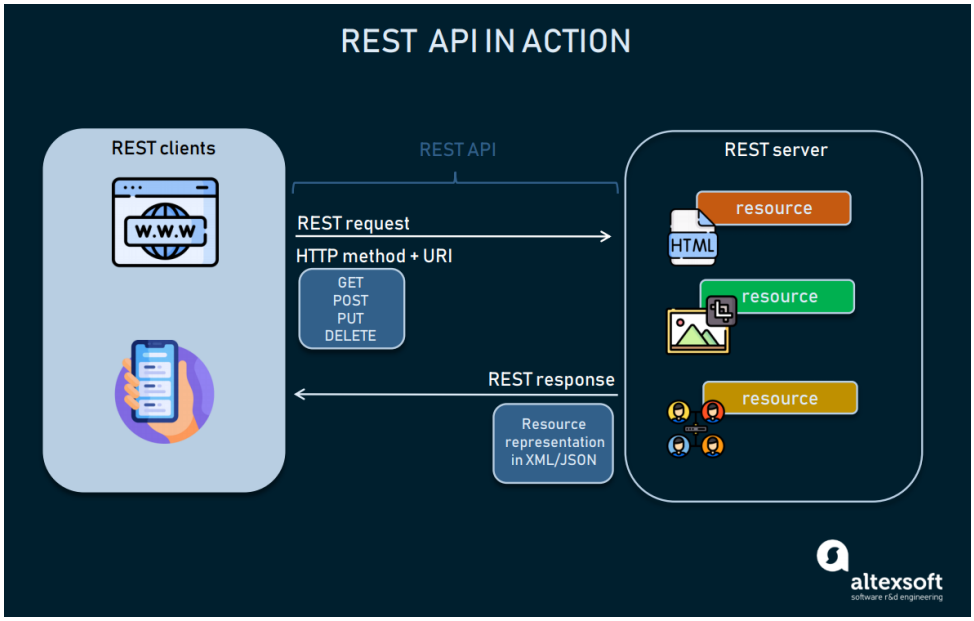
Nejpoužívanější HTTP statusy:

- 200 – Požadavek proběhl v pořádku
- 404 – Zdroj nenalezen
- 500 – Chyba při zpracovávání požadavku (Neošetřená chyba, chyba syntaxe atd.)

2 REST API

Pojem REST API označuje styl architektury pro komunikaci mezi službami. Tato architektura zaznamenala velké využití především díky své jednoduchosti a škálovatelnosti. Jedná se o klient-server architekturu, která pro přenos dat využívá HTTP protokol. REST API využívají především distribuované služby (aplikace), jenž mezi sebou interagují. Nejedná se však o komplexní technologii nebo standard, ale spíše metodu pro definování stylu architektury. [8]

REST API rozděluje aplikaci/služby do menších části nebo zdrojů, které jsou rozděleny podle použitelnosti a funkčnosti. Každá část může být zavolána pomocí jedné nebo více unikátních URL adres využívající HTTP protokol. Výhoda při použití HTTP protokolu pro přenos dat spočívá v přesunutí směrování, formátování a ukládání do mezi paměti na daný protokol, nikoliv samotné REST API. Komunikace mezi serverem a klientem probíhá na principu potřesení rukou, kdy klient odešle požadavek na server, který klientovi zašle odpověď (Obrázek 4). [8]



Obrázek 4. Komunikace pomocí REST API [9]

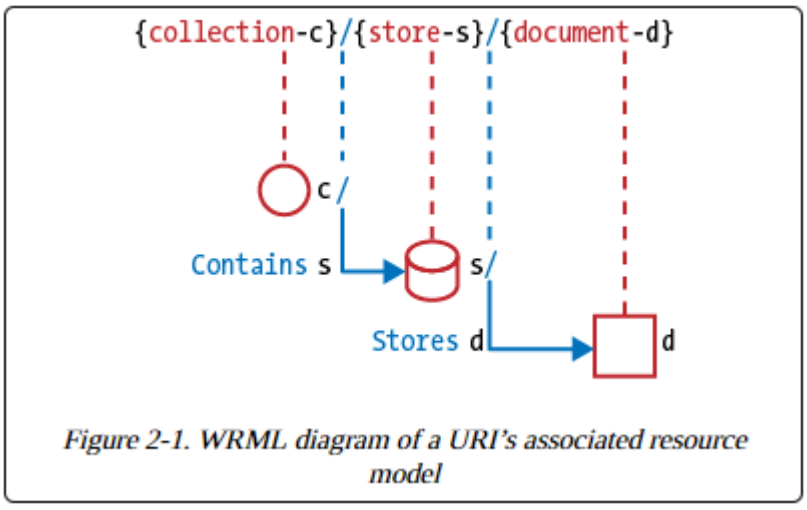
Data, jež se mají dostat k serveru, mohou nabývat různých forem (HTML, XML, JSON atd.). Na straně serveru dojde ke zpracování a vykonání definované obsluhy. Kromě odpovědi zašle server klientovi i HTTP status, který informuje o výsledku a úspěšnosti. Každý požadavek směřující na server obsahuje následující atributy:

- URL adresu (unikátní adresa pro vykonání příslušné akce serverem)

- ID odesílatele (klienta)
- Parametry (data pro předání zdroji, hlavička požadavku atd.)
- Typ HTTP požadavku (GET, POST atd.) [8]

2.1 Struktura

Adresa REST API je definována pomocí adresy URI, kde každá samostatná část je oddělena lomítkem. Model fungování REST API se podobá stromové struktuře, kdy při každé další části se dostáváme hlouběji do API infastruktury. Obecně platí, že aktuální část vychází, dědí nebo je jiným způsobem propojená s částí předchozí (Obrázek 5).



Obrázek 5. Zanoření zdrojů podle URI adresy [10]

2.2 Controller

Controller neboli ovladač se nachází na straně serveru, pomocí kterého dochází k řízení zdroje a tvoří tak procedurální koncept. Jedná se v podstatě o metodu, jenž může obsahovat řadu funkcí, parametrů, vstupů a výstupů. Jedná se o rozšíření klasických CRUD metod (vytvoření, načtení, aktualizace a smazání). [10]

Controller se v praxi také používá jako první vstupní bod na straně serveru po přijmutí dotazu. V němž se většinou kontroluje celistvost a úplnost dat příchozích dat, autorizace atd. Controller poté provolává další zanořenější třídy, které mají na starost samotné provedení požadavku. Velmi uživatelsky přehledné a jednoduché řešení nabízí PHP framework Slim, ve kterém se definují jednotlivé controllery v jasných metodách. Po zavolání REST API s danou adresou dojde k provolání příslušného controlleru a vykonání obsluhy.

2.3 Cíle

Správně navrhnuté REST API by mělo nabývat promyšlené a logické struktury. Základ každého správně sestaveného REST API spočívá v dobře zvoleném návrhu architektury. Z této architektury potom vychází návrh serverový, klientský a funkční. Při sestavování REST API by se mělo vycházet z několika následujících vzorů a typů.

2.3.1 Definování požadavků

Nejdříve je zapotřebí určit účel používání daného REST API. REST API neimplementuje žádnou logiku, tento úkol má na starosti serverová část. Pro jednotlivé dotazy se musí definovat, z jakého zdroje se data budou zpracovávat. Popřípadě upřesnit typ backendu, jelikož v praxi může nastat situace, kdy různá data mají být zpracovávána různými backendy. Struktura výstupních musí být naformátovaná tak, aby byla čitelná, srozumitelná a strana klienta s nimi mohla dále pracovat. Pokud například zašleme v těle dotazu data pro vytvoření záznamu ve formátu XML, namísto JSON, který server očekává, dojde v případě neošetření k chybě. Zároveň vstupní data, která přijdou na server, mají povinnost být v takovém formátu, ve kterém mohou být zpracována. API by mělo být tvořeno podle představ klienta, aby plnilo všechny potřeby a usnadňovalo práci s daty. Mezi hlavní rysy se řadí jednoduchost, jednoznačnost a přístupnost. Dále by REST API mělo být intuitivní, aby noví uživatelé mohli okamžitě začít s jeho používáním bez podrobnější či předchozí znalosti. [11]

2.3.2 Bezpečnost

Bezpečnost lze rozdělit do 3 kategorii, přičemž každá z nich by měla být brána v potaz:

1. Zabezpečení přenosu – Používat zabezpečenou variantu HTTP protokolu, tj. HTTPS.
2. Autorizace klienta – Každý klientův požadavek na server by měl být autorizován (např. pomocí tzv. bearer tokenu, který se často využívá při přihlašování)
3. Zabezpečení samotného serveru, respektive backendu – Ošetření vstupů, které by mohly způsobit například injekce kódu.

2.4 Vlastnosti

2.4.1 Spolehlivost

Díky využití a komunikaci pouze pomocí HTTP protokolu, disponuje REST API velkou spolehlivostí. Nicméně při vytvoření požadavku od klienta nesmí dojít k žádnému vedlejšímu účinku na straně serveru. V úvahu je potřeba brát i spolehlivost samotných typů HTTP dotazů, které by mohly zničit integritu nebo správnost dat. Například u typu GET dostaneme stejný výsledek bez ohledu na počtu zavolání, na rozdíl od POST, kdy při každém samostatném zavolání dojde na straně serveru k unikátní akci (vytvoření záznamu v databázi, vytvoření souboru atd.). Požadavek se tudíž stává idempotentní (Obrázek 6). [12]

HTTP method	Safe	Idempotent
GET	Yes	Yes
POST	No	No
PUT	No	Yes
DELETE	No	Yes

Obrázek 6. Idempotence HTTP metod [12]

2.4.2 Škálovatelnost

REST API by mělo být lehce škálovatelné, nové funkce a adresy by neměly používání nikterak zpomalovat nebo ovlivňovat již hotové části. Většina REST API vychází z jádra, pomocí kterého lze funkce dále rozšiřovat a vytvářet tak modulární strukturu. [12]

2.4.3 Výkon

Výkon (performance) značí, jak dlouho trvá serveru, než vyřídí daný klientův požadavek. Na performance je kladen velký důraz, protože moderní API musí být co nejrychlejší a nejresponzívnější, bez ohledu na jeho rozsáhlost, vytížení nebo náročnost dotazu. [12]

2.4.4 Nezávislost formátu

Data, která klient zašle serveru, mohou být typově nezávislá. Klient pouze musí definovat typ dat v HTTP hlavičce dotazu, server poté přijatá data příslušným způsobem zpracuje. [12]

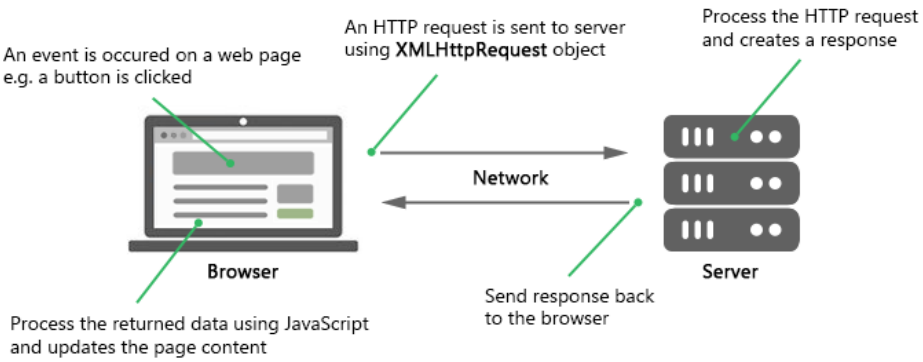
2.4.5 Nezávislost platformy

Mezi velké výhody patří skutečnost, REST API může být navrhnuto, sestaveno či používáno bez určení operačního systému, typu aplikace/služby nebo programovacího jazyka. Smyslem REST API je poskytnout data, které lze bez jakýkoliv dalších potřebných úprav využít. [12]

2.5 AJAX

Pojem AJAX (Asynchronous JavaScript and XML) označuje nástroj vycházející z JavaScriptu. Pomocí tohoto nástroje může klient posílat serveru požadavku v pozadí bez vědomí uživatele (Obrázek 7). Mezi největší výhody spadá možnost dynamicky měnit obsah webových stránek či služeb bez nutnosti jejího obnovení. Díky tomu se práce a plynulost webové služby zvýší a zároveň se tím redukuje počet zaslaných dotazů na server. [13] Nástroje pro vývoj webu lze rozdělit na 2 kategorie: na straně klienta (JavaScript, Angular, React) a na straně serveru (PHP, Java, C#). [13] Výhody nástroje AJAX:

- Zlepšení responzivity
- Asynchronní zpracování dotazů
- Snížení vytížení sítě
- Platformě neutrální [8]

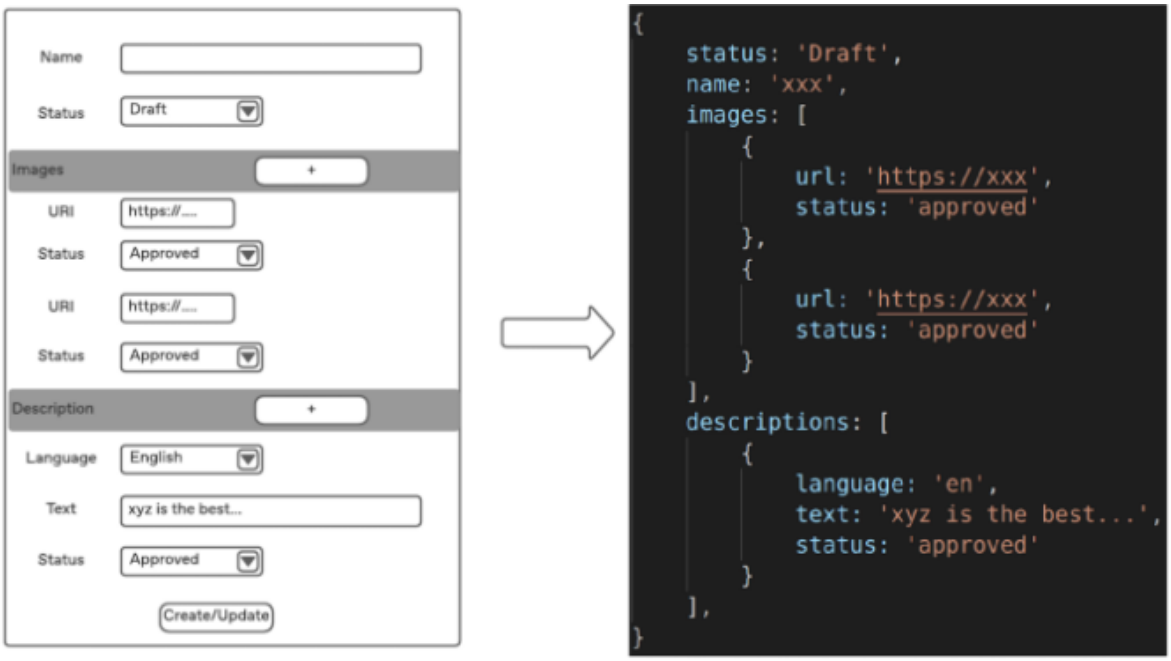


Obrázek 7. Schéma AJAX dotazu [14]

3 SOUČASNÉ NÁROKY NA PŘENOS DAT

V reálném světě často dochází k nevhodnému sestavení REST API dotazů, které vychází ze špatného návrhu nebo nevhodného udržování celého API. Zejména při postupném přibývání nových zdrojů, jež na sebe navazují, dochází k nevhodné strukturalizaci dotazů. Jednotlivé zdroje nejsou správně rozděleny, ba naopak vše navenek vystupuje jako jedna velká entita. Zazdí se tak fakt, kdy právě samotná hlavní entita, se kterou se nejčastěji pracuje, bývá zahlcena jejími potomky, u kterých k jejich manipulaci nedochází příliš často (Obrázek 8). [15]

V tomto momentě může docházet k zásadnímu problému, tj. zpomalení dotazu a jeho následné zpracování v důsledku většího množství nepotřebných dat. Anglicky se jevu přezdívá overfetching dat. Overfetching dat ale může nastat i na opačné straně, tzn. klient dostane v odpovědi dotazu data, které nepotřebuje.



Obrázek 8. Struktura těla REST API dotazu [15]

3.1 Koncepční limitace

Již kvůli svému konceptu a architektuře, ze které vychází, se REST API nehodí na určitý typ služeb. Mezi jednu ze zmíněných, na kterou se díky svému fungování nehodí, patří posílání zpráv. I přesto že tato technologie pro danou službu použít lze, docházelo by k neustálému odesílání a přijímání REST dotazů neboli pollingu. Z toho důvodu je v takovém případě lepší zvolit řešení vycházející z jiné architektury. [16]

3.1.1 Polling a polling madness problém

Jedná se o termíny, které označují vzorec chování REST API, kdy pro každý požadavek na nová data se odešle REST API dotaz. Při neustálém pollingu dochází k plýtvání zdrojů (hardwarové prostředky na straně serveru, využití sítě, zaneprázdnění klienta kvůli zpracování příchozích dat). Podle analýzy API providera Zapier z roku 2013 až 98,5 % všech dotazů jsou zbytečné z pohledu vytížení zdrojů, přenosu a dosaženého výsledku. Polling si lze představit jako pravidelné stisknutí tlačítka „Obnovit“. Hlavní příčinou, proč se tento jev stal běžným byl fakt, že až do nedávna neexistovaly způsoby, které by tento problém mohly vyřešit. Například pomocí resthook – Speciálně vytvořená REST API adresy, jenž kontroluje změnu dat již na straně serveru a následně informuje klienta o proběhlé změně. Toto řešení zahrnuje hned několik výhod, tzn. klient není povinen neustále odesílat dotazy, čímž dochází k menší zátěži sítě a serveru, navíc se zlepší i plynulost aplikace či služby. [17]

3.1.2 Monolitní služby a aplikace

Je zapotřebí si uvědomit, pokud rozdělíme monolitní službu či aplikaci na mikro služby, které pro vzájemnou komunikaci využívají REST API. Náš výsledek bude stále stejný, tzn. ve finále se služba, nebo aplikace bude chovat monolitně. Pod pojmem monolitní aplikace si lze představit souhrn služeb, které mezi sebou komunikují, avšak navenek vystupují jako jeden jednotný celek. Při volbě typu komunikace se právě REST API jeví jako nejjednodušší možná varianta, ale je nutno si uvědomit, že tato technologie je synchronní. V případě synchronní aplikace se jednotlivé dotazy vykonávají postupně. Mikro služby, které mezi sebou komunikují synchronně, mohou na sebe navazovat a tím spoléhat jedna na druhou, proto hrozí hned několik problémů. Mezi jeden z nich se řadí náchylnost aplikace na výpadek či zpomalení jednotlivých služeb, čímž trpí celá monolitní aplikace. Jako alternativa se nabízí použití asynchronního způsobu komunikace, kdy mikro služba nečeká na uvolnění prostředků, navíc řetězec jednotlivých aplikací zůstane zachován. Jeden ze způsobů vychází z principu, kdy sama cílová mikro služba je spouštěčem pro odeslání dotazu (v případě jeho aktivování) a nečeká až je oslovena. Avšak i zde nalezneme určité limity, tj. horší konzistence dat anebo jistota, kdy cílové službě jsou doručena ty nejnovější data. [18]

3.2 Internet of things

Smyslem IoT se rozumí sjednocování reálného a digitálního světa za využití informačních technologií. Obzvláště v poslední dekádě lze zpozorovat nárůst digitalizace fyzického světa.

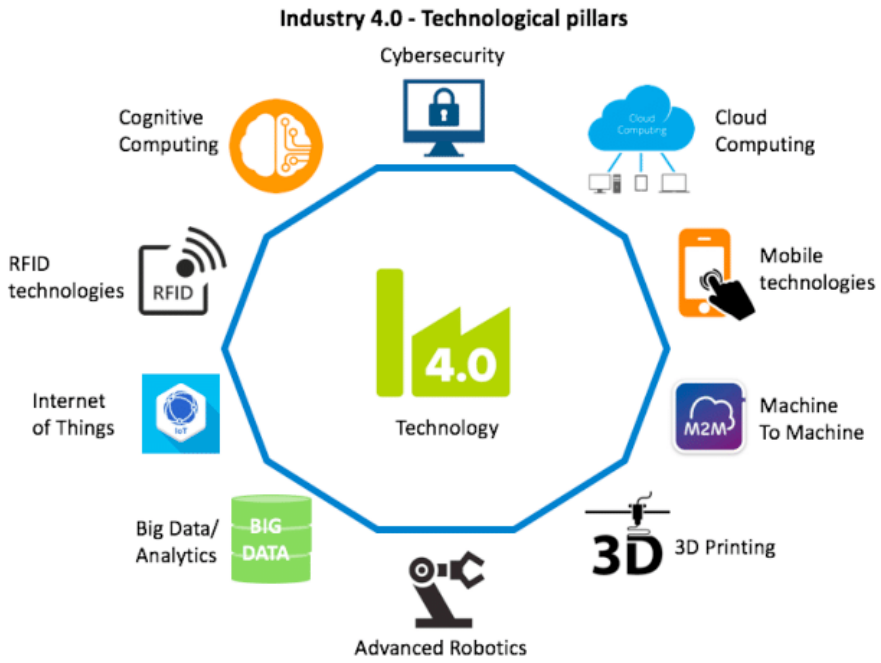
Od chytré domácnosti propojenou a ovladatelnou přes mobilní aplikaci, přes evoluci v komerční sféře a průmyslu, až po složité integrované systémy měst či států. Internet of things pomalu mění pohled, jak se na své okolí díváme a jakým způsobem jsme schopni jej sledovat. [19]

Díky kontrole objektů z fyzického světa pomocí IT technologií v reálném čase, jsme navíc schopni velmi jednoduše a efektivně optimalizovat již zavedené systémy a procesy. To může představovat například lepší rozhodování a sledování v businessu, bezpečnostní systémy, samořídící auta nebo pokrok v medicíně. Přesto se jedná pouze o zlomek, čeho je IoT schopno. Technologické firmy vytváří strategie a plány pro zaplnění díry mezi sběrem potřebných dat z reálného světa a možností tyto data včas zachytit a zanalyzovat. [19]

3.3 Průmysl 4.0

Pojem průmysl 4.0 pochází z myšlenky nové industriální revoluce, jejímž cílem je zahrnutí posledních technologických inovací za účelem rychlé a přizpůsobené produkce. Jedním z hlavních záměrů průmyslu 4.0 se stala automatizace výrobního procesu a zvyšování produktivity skrze „chytré továrny“. Dala by se i označit jako vzájemná implementace výrobního procesu a internetu. Jedná se o pokročilý výrobní model, který v sobě obsahuje technologie, jež jsou integrované v jedné velké službě a komunikují mezi sebou. Cílem chytrých továren a průmyslu 4.0 je nabízet službu, která nabízí digitální, virtuální a technologickou alternativu již k zavedeným principům. Mezi výhody se řadí následná lehká udržitelnost výrobního procesu, zvýšení jeho efektivity, lepší přehled a manipulace. [20]

S digitalizací souvisí i velké množství dat, které se odesílají na servery. Následně zpět na klientské zařízení umístěné v chytré továrně, tj. od počítačů přes tablety, tiskárny, výrobní stroje až po mobilní zařízení. Pro komunikaci se velmi často využívá právě REST API, jelikož je dobře škálovatelné, čemuž nahrává fakt, kdy vývoj softwaru pro chytré továrny se provádí agilní metodikou vývoje. To znamená, že jak funkcionalita chytré továrny, tak struktura REST API se neustále během vývoje rozšiřuje. Zároveň výměna dat musí být rychlá a plynulá, bez ovlivňování již stávajících funkcí.



Obrázek 9. Vizualizace struktury průmyslu 4.0 [21]

3.4 Mobilní aplikace

Za poslední dekádu mobilní a celkově malé elektronické zařízení zažily velký růst. Na počátku drtivá většina mobilních aplikací byla čistě nativní. Jedná se o aplikaci vytvořenou speciálně pro daný systém s využitím jejich prostředků, hardwaru a knihoven. Mezi výhody patří rychlost, využití většiny možných prostředků zařízení nebo možnost běžet asynchronně. K nevýhodám se naopak řadí úzká spjatost na daný systém a vydávání aktualizací. V průběhu času se ale začaly objevovat alternativy jako mobilní webové aplikace. V tomto případě není nutné aplikaci lokálně stahovat, vše běží v prohlížeči zařízení. Oproti nativním aplikacím není vyžadován oddělený vývoj pro jednotlivé operační systémy. Vývojářům stačí udržovat pouze jednu verzi aplikace pro všechny systémy a typy zařízení. [22]

Zajímavou alternativou, která se objevila zhruba před 7 lety, je tzv. progresivní webová aplikace. Dalo by se říci, že se jedná o hybrid dvou předchozích typů mobilních aplikací. V prohlížeči se chová jako webová, avšak jedním gestem si lze danou aplikaci stáhnout do zařízení a pracovat s ní, jako kdyby byla nativní. Velkou výhodou je možnost využití online režimu a komunikace pomocí API, zároveň však při nativním režimu lze pracovat offline. Na rozdíl od nástrojů jako lokální databáze či mezi paměťové API, progresivní aplikace nabízí opravdovou kontrolu nad zdrojem. [23]

Právě klasické mobilní i progresivní webové aplikace využívají v drtivém případě pro svoji komunikaci REST API. Na REST API se začíná projevovat jeho stáří a pro nové technologie nemusí být vyhovující. Zároveň se může začít objevovat i jeho koncepční nedostatky.

3.5 Code on demand

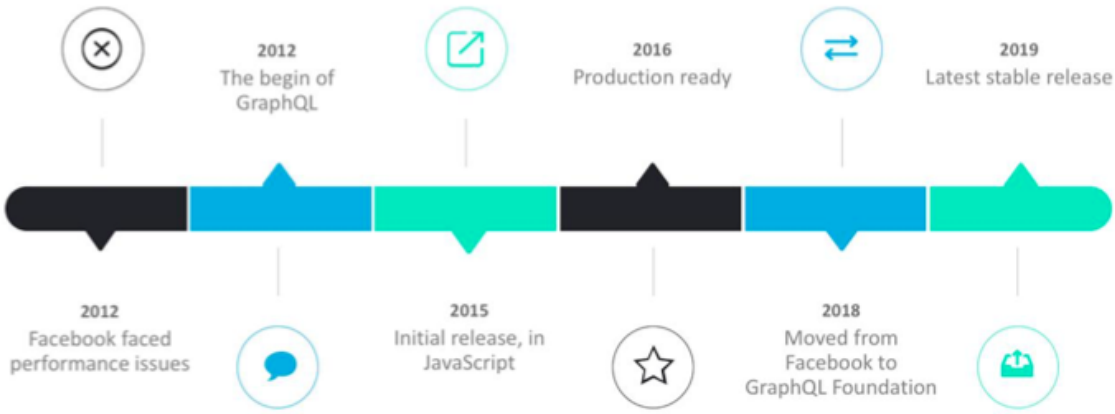
Označuje rozšíření REST API nebo i jiných typů způsobu komunikace mezi serverem a klientem. Server kromě klasické odpovědi zašle i kód nebo hotovou funkcionalitu, která může být klientem spuštěna. Ve webové komunikaci to může znamenat například zaslání javascriptového kódu přímo samotným serverem. Klientovi potom stačí pouze odpověď bez jakékoliv úpravy spustit. Zmíněná funkcionalita se týká třeba webového oznámení nebo animace, která je formou HTML sestavena, a do ní je následně vložena javascriptová část. Danou část označujeme jako CoD. [24]

4 GRAPHQL

Jelikož se aplikace, využívající webové služby, stávají z různých důvodů více a více rozšířené. Je kladen důraz na zvolení co nejlépe postavené architektury pro přenos a správu dat. Architektura by měla být centralizovaná, multiplatformní a zároveň zvládat pracovat s více systémy současně. Zmíněným podmínkám vyhovovalo dlouho dobu REST API, které se stále velmi hojně využívá. Avšak v případě velkých aplikací, které pracují se značným množstvím dat, se mohou začít objevovat nedostatky. Při řešení tohoto problému projevila první iniciativu společnost Facebook, kdy v roce 2012 začala využívat vlastní technologii GraphQL, kterou o 3 roky později poskytla jako open source (Obrázek 10). [25]

Mezi jeho velké výhody se řadí:

- Eliminování potřeby neustále vytvářet nové URL adresy – v GraphQL zastává komunikaci 1 URL adresa.
- Eliminace přenášení zbytečných dat – jak na straně serveru, tzn. zatěžování zdroje (databáze, soubor, služby), tak na straně klienta (server nezasílá v odpovědi dotazu zbytečná data, která stěžují jejich zpracování klientem a vytěžují síťový přenos).

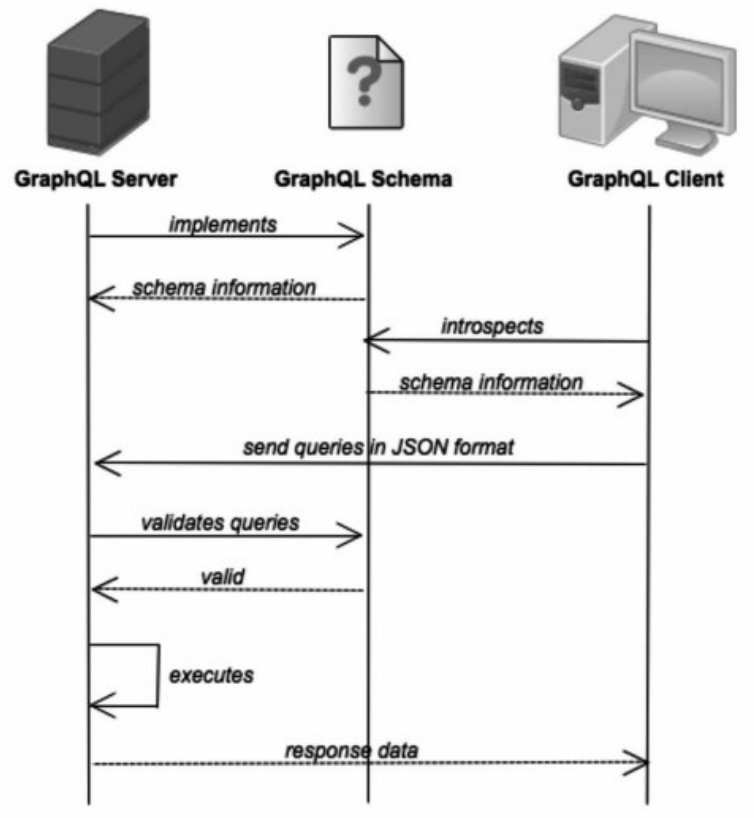


Obrázek 10. Časová osa průběhu vývoje GraphQL [25]

4.1 Princip fungování

GraphQL se označuje jako jazyk pro definování struktury API dat a jejich typů. Jedná se poměrně o novou technologii. GraphQL nezajišťuje práci se samotnými zdroji na serverové části služby či aplikace. Slouží pouze pro komunikaci mezi klientem a server, kteří tuto komunikaci podporují. Jedná se o nástroj, jenž slouží jako spouštěcí vrstva, která očekává vstupy, ať už ze strany klienta nebo serveru. Tato vrstva může reagovat na zaslané dotazy klienta pouze v jazyce GraphQL (nazývaném GraphQL Query Language). Klient může zaslat na server dotazy ve zmíněném jazyce, ale pokud na serveru není naimplementována potřebná infrastruktura, která příchozímu dotazu rozumí, nelze očekávat od serveru validní odpověď. [26]

Princip komunikace spočívá v GraphQL schématu, jež je uloženo na serveru. Toto schéma slouží jako stěžejní bod, který se používá k validaci dat. GraphQL je striktně typové, všechny data musí odpovídat typům uvedeným v schématu. Dotaz klienta na server obsahuje výčet požadovaných dat, server si následně zkontroluje, zda výčet má správnou strukturu a odpovídá schématu (Obrázek 11). Pokud validace projde, server pomocí příslušných resolverů zajistí dodání dat, které odešle zpět klientovi.



Obrázek 11. Princip fungování GraphQL [25]

4.2 GraphQL schéma

Jak již bylo řečeno, celé GraphQL vychází z definovaného schématu, ze kterého vznikne graf podobný stromové struktuře. Jednotlivé listy si lze například představit jako atributy entity či sloupce databázové tabulky, které jsou napojeny do entity (uzlu). Jednotlivé entity lze do sebe zanořovat a strom tímto způsobem dále větvit. Celý strom značí maximální množství dat, jenž jsme schopní jedním provoláním získat. Schéma se píše v jazyce nazvaném Schema Definition Language (SDL). Existují 2 nejčastěji používané způsoby pro definování:

- Dynamické vytvoření objektů
- Pomocí souboru

4.2.1 Query Language

Pro vytvoření a odeslání dotazu v GraphQL jazyce je zapotřebí v těle dotazu určit jaké akce a data jsou požadovány. Jelikož na rozdíl od REST API, kde stačí pouze zavolat příslušnou URL adresu a server ví, které data klientovi vrátit. V GraphQL se tato informace musí specifikovat v těle samotného dotazu. Proto se v dotaze odesílá text, jemuž se říká Query Language (QL). Dále následují objekty a atributy podle SDL schématu, veškeré zanoření, struktury a syntaxe musí respektovat jeho definici.

Hned na začátku dotazu se definuje typ dotazu:

- Query: Typ pro čtení dat, jedná se o alternativu HTTP GET dotazu v REST API.
- Mutation: Typ pro uložení, úpravu či mazání dat, alternativou v REST API to jsou typy POST, PUT a DELETE.

4.2.2 GraphQL odpověd

V GraphQL odpovědi na dotaze lze každý atribut označit je pole, které může obsahovat další pole. GraphQL zpracovává přijatý a úspěšně validovaný dotaz od nejméně zanořeného objektu po nejvíce zanořené. U každého objektu dojde k inicializaci a vykonání příslušného resolveru a naplnění výslednými daty.

Na obrázku níže (Obrázek 12) lze vidět GraphQL dotaz a výsledná data, které server pro tento dotaz vrátil. Můžeme vidět, že se jedná o typ query, ve kterém se žádají data pro objekt charakter s id 1. Na pravé straně obrázku poté vidíme odpověď, jež obsahuje daný objekt s příslušnými skalárními atributy (id, name, status a created). Uvnitř se nachází zanořený

objekt location. Každý atribut v dotaze vystupuje jako pole. Jednotlivé pole lze považovat za potomky rodičovského objektu, pod který patří. Nejméně zanořený objekt nese název Root Query Object, jedná se o vstupní bod či rozcestník, odkud lze procházet celou strukturu GraphQL schématu.

Obrázek 12. Ukázka GraphQL dotazu [27]

4.2.3 Typy

GraphQL je silně typové, tzn. všechny data, které dostane GraphQL ke zpracování, se validují vůči definovanému SDL schématu. Ať už to jsou data klienta poslané v dotaze, nebo výsledná data na serveru. Pokud GraphQL dostane nesprávná data, dotaz neprojde a dochází k validační chybě. Každý atribut může nabývat následujících typů:

- Skalární (integer, float, string)
- Objekt nebo kolekce objektů
- Vlastní datový typ (datum, url adresa, email)

4.2.4 Resolver

Každý objekt definovaný v SDL schématu musí mimo její struktury obsahovat i resolver, který zajišťuje práci se samotnými daty a jejich dodání. Stejně jako schéma lze i resolvery definovat v souboru, odkud se načte a propojí se svým objektem, nebo dynamicky za běhu programu.

4.3 Proměnné

Stejně jako REST API i GraphQL podporuje parametry v dotaze. Parametry lze posílat:

- Přímo v dotaze – parametry jsou vázané k objektu, u kterého jsou definovány.

- Jako GraphQL proměnné – v tomto případě dochází k validaci a následnému vzájemnému propojení s dotazem. Jelikož GraphQL dotaz a proměnné přicházejí na server samostatně, je zapotřebí proměnné jednotlivým zanořeným objektům přiřadit).

4.4 Výhody

- Pouze 1 URL adresa a HTTP typ pro všechny dotazy
- Možnost zaslat v 1 dotaze více poddotazů
- Velmi vhodný pro vytváření jedné aplikace určenou pro různé platformy
- Práce s daty, která jsou skutečně zapotřebí (vyřešen problém under/over fetchingu)
- I přes chybu ve zpracování dotazu se snaží data dodat

4.5 Nevýhody

GraphQL nabízí klientovi volnost poskládat si dotaz podle svých představ a potřeb. Což přináší velkou řadu výhod ale i nevýhod a problémů. V takovém případě, kdy se kontrola nad sestavením dotazu přesouvá zcela na stranu klienta, může docházet k zasílání příliš komplexních a složitých dotazů na server. Server či databáze se následně může přetěžovat. Ve skutečnosti existuje pravidlo, které říká, že s lineárním nárůstem velikosti dotazu, se exponenciálně zvyšuje množství výsledných dat. Při používání REST API lze například snížit vytížení snížením počtem možných dotazů na server za časový interval. Avšak v GraphQL pouze jeden dotaz je schopen celý systém shodit. Jako částečné řešení se nabízí dynamické počítání složitosti daného dotazu, kdy po jeho překročení GraphQL umí zpracovávání dotazu přerušit a vrátit částečný výsledek. Toto řešení ale není ideální, protože výsledná data mohou být nekompletní nebo nepoužitelná. [28]

II. PRAKTICKÁ ČÁST

5 IMPLEMENTACE REST A GRAPHQL TECHNOLOGIÍ

Cílem praktické částí práce je implementování obou technologií za využití stejných nástrojů a prostředků. Oba zmíněné způsoby vyzkoušet v praxi a otestovat je pomocí testovacích scénářů. Součástí praktické části je i výčet technologických omezení, nevýhod a rizik, které z dané technologie vyvstávají.

5.1 Popis funkcionality

Výstupem práce jsou backendové aplikace, které lze považovat za jednotný celek, jež v sobě zahrnují technologie GraphQL i REST. Přidaná hodnota spočívá v návrhu aplikace, kdy se řeší stejné problémy a funkčnost oběma zmíněnými způsoby. Tím pádem lze výstup aplikace zpracovávat a porovnávat. Aplikace v obou formách zvládají všechny potřebné operace s daty, jež jsou uloženy v databázi. To znamená:

- Načítání (GET)
- Vytváření (POST)
- Mazání (DELETE)
- Aktualizace (PUT/PATCH)

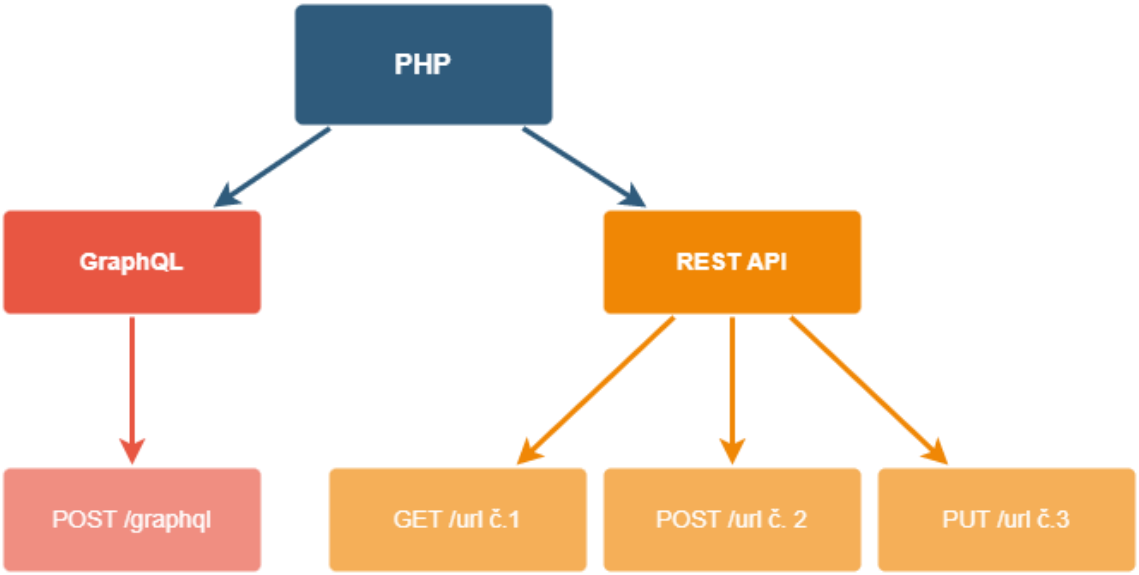
U všech zmíněných bodů lze danou operaci provádět nad jedním záznamem či celou množinou záznamů. U načítání lze tento výstup z dotazu upravovat podle aktuálních potřeb, tj:

- Filtrování (užší výběr z nabízených dat)
- Stránkování (velikost a pořadí stránky)
- Řazení (atribut pro řazení, směr)

5.2 Rozdělení aplikace

Data jsou uložena v Oracle databázi, pro běh aplikace byla zvolena kombinace Apache serveru a PHP. Aplikace je naprogramována v jazyce PHP ve verzi 7.0. Mezi výhody patří skutečnost, že aplikace vychází z jednoho společného jádra. To znamená, že REST i GraphQL využívají stejné metody a funkce. REST API lze provolat definovanými URL adresami společně s příslušným HTTP typem požadavku, zatímco GraphQL naslouchá pouze na jedné URL adrese (Obrázek 13. Diverzifikace GraphQL a REST v aplikaci). Oba způsoby zahrnují několik společných vlastností jako například využití HTTP protokolu či příchozí odpověď dotazů ve formátu JSON. Avšak existují i rozdíly, zejména pak ve struktuře příchozího dotazu na server. Zatímco v REST API části aplikace je každé funkčnosti přiřazena

unikátní URL adresa, v GraphQL si tuto strukturu určuje sám klient. Na rozdíl od REST API se neprovolávají URL adresy, ale jednotlivá klíčová slova v samotném dotaze.



Obrázek 13. Diverzifikace GraphQL a REST v aplikaci

6 TESTOVACÍ DATA

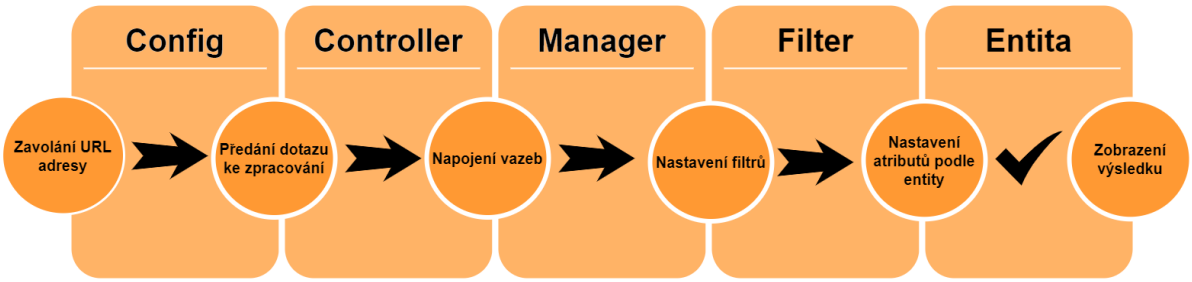
Pro testování bylo zvoleno XXX entit, které představují strukturu testovacích dat. Všechny entity jsou reprezentovány příslušnou relační databázovou tabulkou. Data obsahují všechny nejpoužívanějších datové typy a funkce (indexy, constrainty, triggery). Entity obsahují potřebné základní nastavení pro jejich fungování, zároveň jsou mezi sebou propojeny pomocí všech typů vazeb, tj: 1:1, 1:N, M:N. Seznam testovacích entit:

- [OutboundDelivery]
- [OutboundDeliveryMaterial]
- [InboundDelivery]
- [Status]
- [StatusType]
- [Material]
- [MaterialWorkspaceRel]
- [Workspace]
- [WarehouseMaterial]

Obrázek 14. Ukázka ER digramu entity OutboundDelivery

7 REST API ČÁST APLIKACE

Celá aplikace je rozdělena do jednotlivých entit, kde každá entita představuje entitu databázovou (tabulka či pohled). Tyto entity jsou dále sdruženy do balíků podle podobných vlastností, tj. například společná funkcionalita nebo začlenění entit. Struktura balíku (Obrázek 15) představuje třídy, jež pracují s danou konkrétní entitou. Například entita OutboundDelivery (OD) je umístěna v balíku Delivery, entita OD vlastní svoje třídy (controller, managery, filtry atd.), které mají za úkol obstarávat specifickou část funkcí.



Obrázek 15. Struktura balíku REST API

7.1 Config

Config soubor obsahuje formulované URL adresy RESTového API (Obrázek 16). Mimo jiné jsou zde nastaveny atributy jako název balíku, pod který daná adresa spadá. Dále zde lze najít typ HTTP požadavku nebo název controlleru, jehož obsluha se volá.

Při zavolání URL adresy se daná adresa s nastaveným HTTP typem dotazu hledá ve všech nadefinovaných konfiguračních souborech. V případě nalezení odpovídající varianty se zavolá zmíněný controller.

Bundles > Delivery > Configs > {} outboundDeliveries.routes.json

```
[
    {
        "name": "rOutboundDeliveries",
        "method": "GET",
        "bundle": "Delivery",
        "controller": "OutboundDeliveries",
        "patterns": [
            "/outbound-deliveries",
            "/delivery/outbounds"
        ]
    },
```

Obrázek 16. Ukázka definování REST adresy

7.2 Controller

Controller představuje první vstupní bod při obsluze zavolané adresy. Podle typu HTTP požadavku se zavolá příslušný typ obsluhy. Ke každému rozdílnému typu požadavku se přistupuje odděleně, jelikož každý takový dotaz vyžaduje odlišnou obsluhu a výstup. V případě GET dotazu následuje zavolání svého managera, od kterého jsou vyžádána data. Zatímco v případě POST dotazu dochází k předání dat z frontendu managerovi a uložení.

7.3 Manager

Manager lze označit za jádro každé entity, ve které se provádí hned několik nastavení. V containeru getJoinMap se definují SQL joiny pro jednotlivé entity (Obrázek 17), jež se mají napojit. V metodě getEntitiesMap jsou určeny namespacy entit pro jednoznačnou identifikaci. A v neposlední řadě metoda getFiltersMap vymezuje použité filtry. Každý manager v koncovém balíku extenduje abstraktní manager. Zde se potom například v případě GET požadavku plní QueryBuilder právě podle nastavení v manageru, atributů ve třídě entity a nastaveném filtru. Nastavený QueryBuilder se posléze spustí vůči zvolené databázi a výsledná data jsou sestavena do kolekce a odeslána na výstup REST API.

```
public function joinOutboundStatus(QueryBuilder $qb){
    $on = EOutboundDelivery::getColumnPrefix().EOutboundDelivery::getBPN("idStatus")." = ".
    eStatus::getColumnPrefix().eStatus::getBPN("id");
    $qb->leftJoin(EOutboundDelivery::DB_TABLE_ALIAS,
    eStatus::DB_TABLE_NAME, eStatus::DB_TABLE_ALIAS, $on);
}
```

Obrázek 17. Příklad metody v manageru pro navázání entity

7.4 Entita

Třída entity reprezentuje konkrétní relační tabulky v databázi. K jejímu jednoznačnému určení je zde formulován její databázový název, alias a sekvence. Dále se zde definují její jednotlivé sloupce ve formě atributů. U každého atributu lze nastavit celou řadu parametrů (Obrázek 18). Tj. například název sloupce v databázi, popis, datový typ v aplikaci a databází výchozí hodnota atributu nebo zahrnutí atributu v POST a PUT operacích.

```php
/**
 * Outbound status
 * @var varchar
 */
public static $idStatus = [
    'name'        => "id_status",
    'related'     => "Status\Entities\Status",
    'description' => 'Status',
    'type'        => "integer",
    'insert'      => true,
    'update'      => true,
    'default'     => null,
    'database'    => [
        'null'        => true,
        'default'     => null,
    ],
    'sqlsrv'      => [
        'type'        => "nvarchar(50)"
    ],
    'oracle'      => [
        'type'        => "varchar2(50)"
    ],
    'outputType'  => [
                        EntityOutputType::BASIC,
                        EntityOutputType::DEFAULT,
                        EntityOutputType::EXTENDED
                    ]
];
```

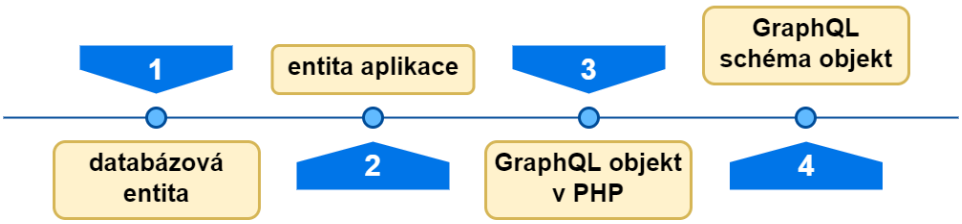
Obrázek 18. Ukázka definice atributu entity

8 GRAPHQL ČÁST APLIKACE

Pro implementaci GraphQL v PHP byl zvolen open source nástroj Webonyx. Jedná se o PHP knihovnu, která v sobě zahrnuje veškeré možnosti klasického GraphQL. Mezi jeho výhody patří podpora všech nejrozšířenějších PHP frameworků a podpora obou typů implementace (code-first a schema-first). Samotná implementace GraphQL v aplikaci využívá pro práci s daty stejných metod a funkci jako restové řešení. Jelikož mezi nevýhody GraphQL patří oproti REST API větší potřebná režie, bylo cílem vytvořit alternativu restového API s využitím dynamické obsluhy GraphQL, a zároveň s důrazem na co největší samostatnost.

8.1 Implementace

Proto, aby byla práce s GraphQL co nejsamostatnější, musela být zvolena varianta code-first. Mezi základy při implementaci GraphQL patří vytvoření schéma. Jak již bylo vysvětleno v restové části implementace, databázová entita/tabulka je reprezentována třídou v aplikaci, ze které se při vytváření schématu pro GraphQL vychází, tzn. entita z aplikace se transformuje do GrapQL objektu (Obrázek 19). Stěžejní bod, kterým je vytvoření schéma, spočívá v procházení jednotlivých entit, jelikož každá entita v sobě obsahuje soupis dalších zanořených entit. K vytvoření schéma tedy dochází za využití rekurze. V případě dynamického vytváření schématu lze jednotlivé objekty ze schématu přirovnat k objektům entit z aplikace.



Obrázek 19. Transformace entity

Na začátku rekurze se tedy vychází z hlavní entity, nad kterou byl provolán dotaz. Z této entity dojde k vytvoření PHP GraphQL objektu, ve kterém se definují 4 povinné atributy (Obrázek 20):

1) name – název objektu, který odpovídá finálnímu objektu ve schéma
2) fields – množina atributů, které objekt obsahuje
3) args – vstupy dotazu z frontendové části
4) resolve – odkazuje na resolver, tj. funkce, jež zajišťují plnění objektu daty

```php
$obj = new ObjectType([
    'name' => $mainEntityPrefix . $relation['entityName'],
    'fields' => $this->getEntityFields(new $relation['entity'](), $mainEntityPrefix, $summary, $usedEntities),
    'description' => 'Entity description - to do',
]);

$res = [
    'type' => $this->getObjectType($relation, $obj),
    'args' => Filter::createRawInnerInputSchema(),
    'resolve' => function($rootValue, $args) use ($relation, $entity) {
        $buffer = $relation['buffer'];
        $buffer::{"getInstance"}()->add($entity->getColumnAliasPrefix(), $rootValue[$relation['parentAttribute']]);
        return new \GraphQL\Deferred(function () use ($rootValue, $args, $buffer, $relation, $entity) {
            $buffer::{"getInstance"}()->loadBuffered($relation, null, false, $args, $entity->getColumnAliasPrefix());
            return $buffer::{"getInstance"}()->get($entity->getColumnAliasPrefix(), $rootValue[$relation['parentAttribute']]);
        });
    }
];
```

Obrázek 20. Definování GraphQL PHP objektu

8.1.1 Schéma

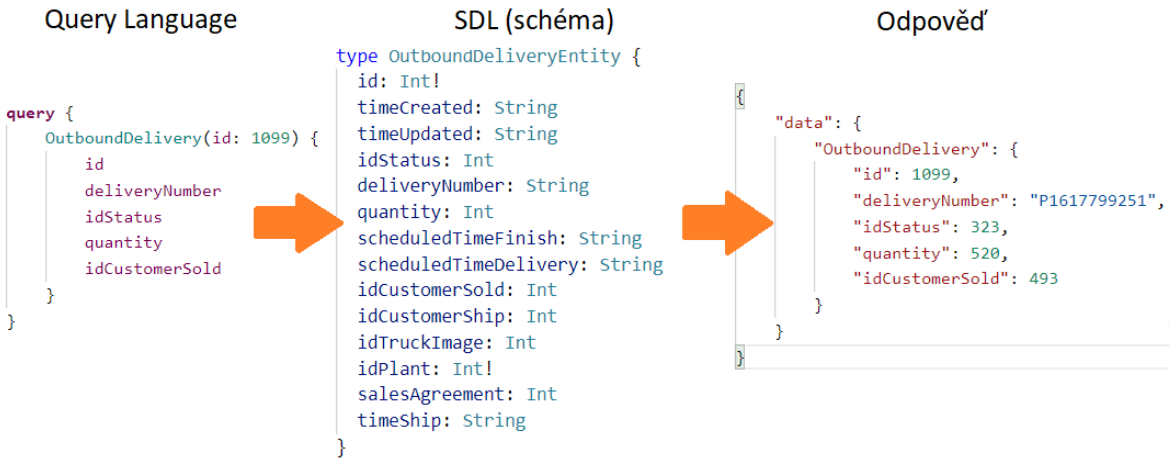
Jednotlivé objekty lze do sebe zanořovat pomocí atributu „fields“. Právě v tomto bodě dochází k rekurzivnímu volání funkce a plnění. Vytvořené objekty se tedy do sebe zanořují a vytvářejí tak stromovou strukturu. Pomocí metody getPropertyPreferedType() dochází k získání datového typu atributu v aplikaci, který je následně převeden na jeden ze skalárních GraphQL typů (Obrázek 21).

```php
$type = $entity->getPropertyPreferedType($property);
switch($type) {
    case 'integer':
        $graphQLType = Type::int();
        break;
    case 'float':
        $graphQLType = Type::float();
        break;
    case 'string':
        $graphQLType = Type::string();
        break;
    case 'date':
    case 'datetime':
        $graphQLType = Type::string();
        break;
    case 'boolean':
    case 'bool':
        $graphQLType = Type::boolean();
        break;
}

if (!$nullType) {
    $graphQLType = Type::nonNull($graphQLType);
}

$fields += [
    $property => [
        'type' => $graphQLType,
        'description' => $entity->getPropertyDescription($property) ?? '---'
    ]
];
```

Obrázek 21. Převod na GraphQL skalární datové typy

Výsledkem rekurze je tedy GraphQL schéma, které odpovídá entitám v aplikaci, jejich vzájemným vazbám a zanoření. V případě získání dat pomocí GraphQL tedy stačí provolat URL adresu /graphql, ve které se pomocí query language dotázat na konkrétní data. GraphQL v odpovědi vrátí pouze data, o které si strana klienta zažádala. Po odeslání dotazů dochází k následujícím operacím:

1) rekurzivní vytvoření GraphQL schéma
2) zaslaný dotaz se validuje vůči poskládanému schématu
3) pro klíčová slova v dotaze se zavolají resolvery, které se postarají o načtení dat
4) načtená data se opět validují oproti schématu
5) klientovi je podle úspěšnosti validace doručena odpověď (Obrázek 22)



Obrázek 22. Postupná obsluha GraphQL dotazu

8.1.2 Buffery

Jelikož se při volání GraphQL dotazů vychází z koncepce, která spočívá v co nejmenším počtu dotazů na server, přičemž v ideálním případě je tento dotaz pouze jeden. Tím lze data ukládat do sdílené vyrovnávací paměti PHP (bufferů). Mezi velké výhody těchto bufferů patří možnost snížení potřebných dotazů vůči databázi. V aplikaci existuje pro každou entitu vlastní buffer, do kterého se data příslušné entity ukládají. Buffer si lze představit jako jednoduché pole, kde jeho index je vyjádřen unikátním identifikátorem, v samotném poli jsou potom uloženy strukturovaná data.

Buffery se využívají především při načítání navázaných entit, aby se zamezilo jejich horizontálnímu načítání. Jelikož GraphQL je defaultně nastaveno tak, aby pro každý navázaný objekt v každém jednotlivém záznamu daný navázaný objekt načetlo (dochází k provolání

příslušného resolveru). Dochází k nadbytečnému zatěžování zdroje a časové prodlevě. Tomuto jevu se říká N+1 problém, přičemž N značí počet navázaných objektů k danému 1 záznamu. Buffer funguje tím způsobem, že při zavolání resolveru daného navázaného objektu, si zapamatuje jeho identifikátor a resolveru zanechá slib (promise). Po načtení všech záznamů a získání všech identifikátorů z navázaných objektů, dojde pouze k 1 provolání zdroje, který vrátí všechny potřebné navázané objekty současně. GraphQL následně jen dodá objekty resolverům, kterým zanechal slib. Tento proces se opakuje pro každou úroveň zanoření. Způsob načítání zanořených objektů se tím změní z horizontálního na vertikální.

Samotné využití bufferu spočívá například při zavolání dvou identických dotazů, jenž se liší pouze rozdílným požadovaným identifikátorem na vstupu dotazu. Popis na konkrétním příkladu (Obrázek 23) - žádost klienta o data entity „OutboundDelivery“ s id: 1 (dotaz č. 1) a „OutboundDelivery“ s id: 2 (dotaz č. 2). Zároveň v obou dotazech je vyžádána navázaná entita status. Pokud je navázaný objekt (v tomto případě status) totožný, dotaz do databáze se, při načítání objektu status v druhém dotazu, nekoná a objekt se načte z bufferu. V každém bufferu lze zvolit, zda se mají při jeho prvním plněním načíst všechny data. Tj. vhodné pro navázané objekty, kterých:

- není mnoho
- často se používají
- k jejich změnám dochází zřídka

```
query {
    query1: OutboundDelivery(id: 1) {
        id
        deliveryNumber
        status {
            id
            title
        }
    }
    query2: OutboundDelivery(id: 2) {
        id
        deliveryNumber
        status {
            id
            title
        }
    }
}
```

Obrázek 23. Využití bufferu při znovupoužití navázaného objektu

9 POROVNÁNÍ

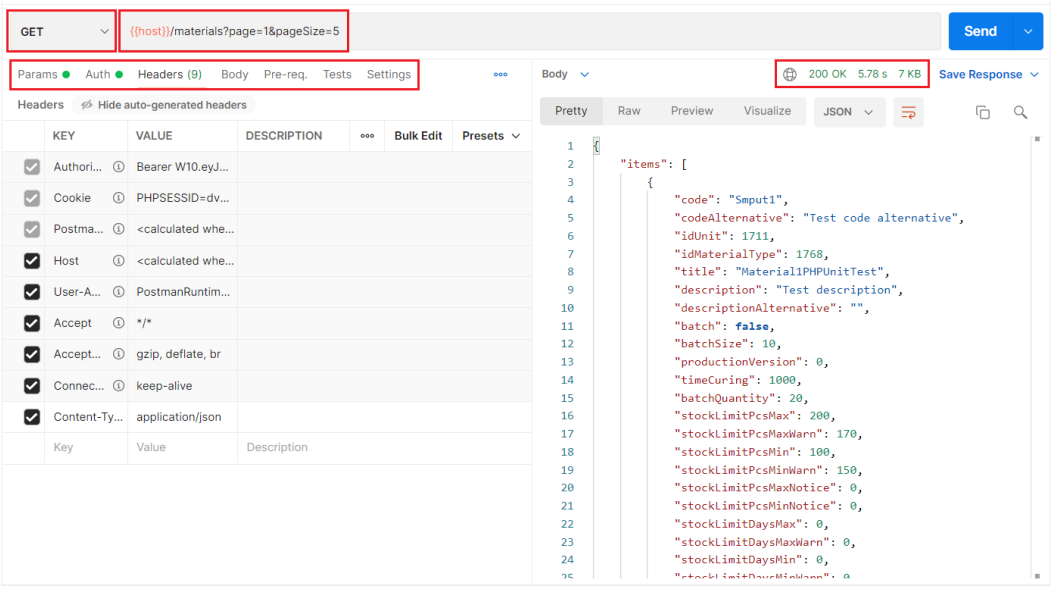
Jak již bylo zmíněno v teoretické části, mezi největší rozdíly mezi REST a GraphQL technologiemi patří odlišnosti, které vycházejí z jejich konceptu. Tato kapitola je věnována popisu zmíněných koncepčních rozdílů za využití testovacích dat.

9.1 Postman

Pro simulaci zasílání API požadavků byl využíván program Postman, který podporuje použití REST i GraphQL. Program dovoluje veškeré nastavení API dotazu podle vlastních potřeb. Jednotlivé dotazy se řadí do záložek, přičemž v každé záložce se nastavuje typ HTTP metody a cílová URL adresa pro odeslání dotazu (Obrázek 24). V prostřední části obrazovky lze nastavit:

- Parametry dotazu
- Typ autorizace
- Nastavení hlavičky
- Data dotazu
- Postman testy a scénáře
- Nastavení dotazu

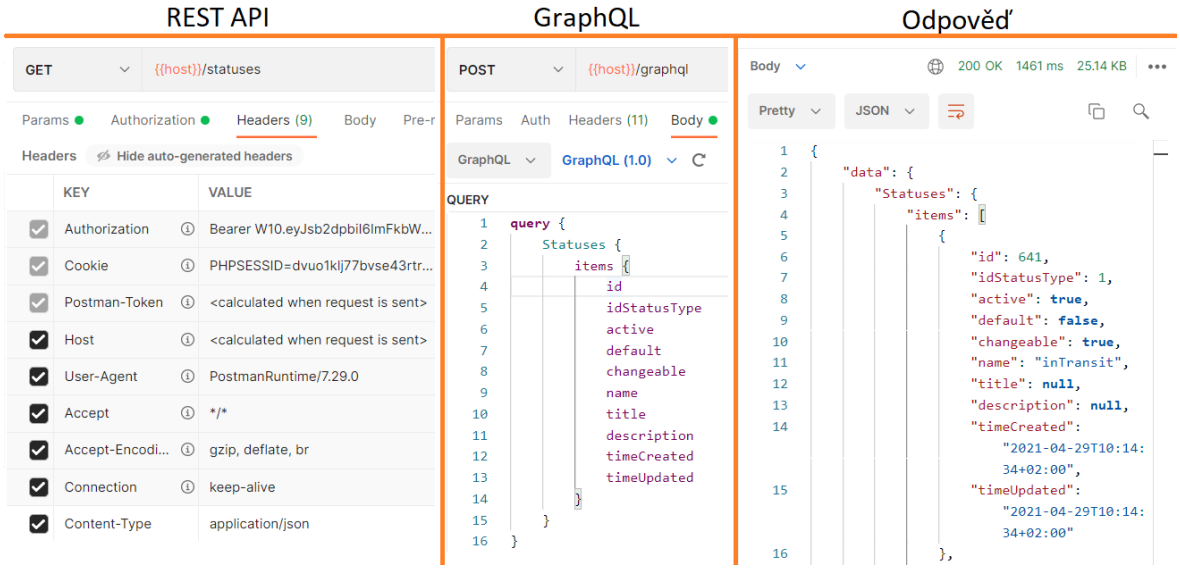
Po odeslání a zpracování dotazu se mimo odpovědi zobrazí HTTP status, strávený čas a velikost dotazu. Provedené dotazy se ukládají do historie, zároveň však lze pro lepší přehled jednotlivé dotazy ukládat do složek a kolekcí.



Obrázek 24. Ukázka API dotazu v programu Postman

9.2 Načítání dat

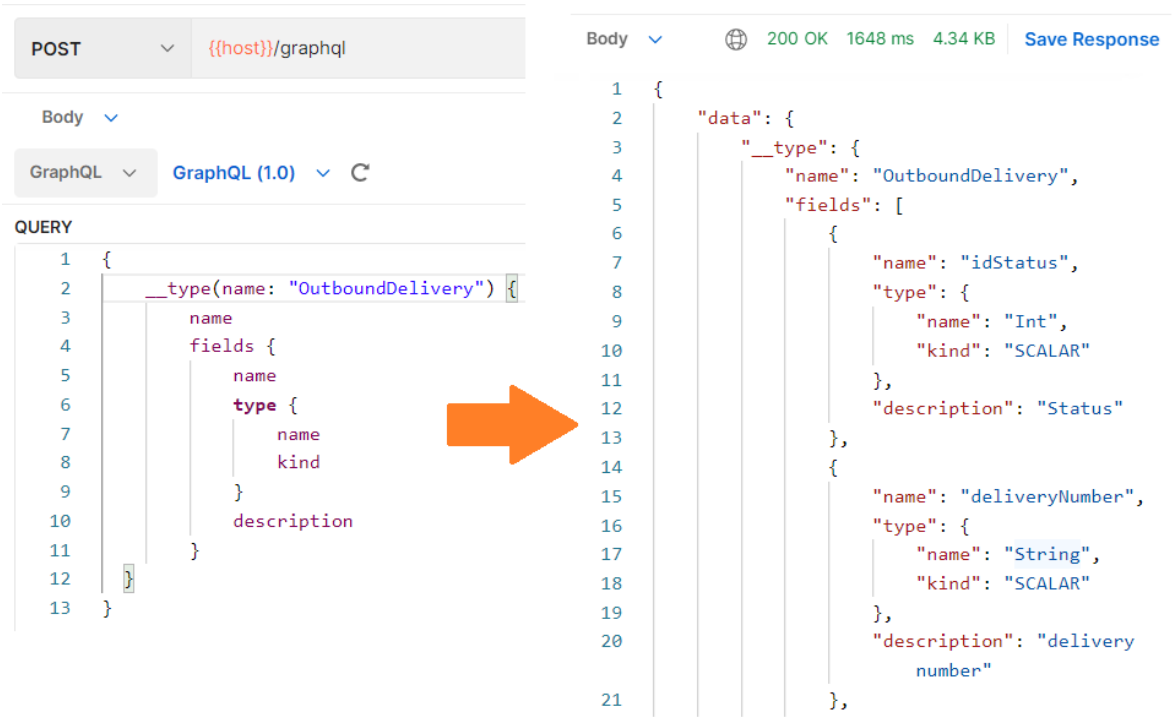
Nejpoužívanější operace pro prací s daty je jednoznačně jejich načítání. Právě v načítání se nachází značný počet rozdílů. Jako ukázkový příklad pro načtení dat byla zvolena entita Status z testovacích dat. Hned prvním rozdílem je identifikace samotné entity, tj. v případě REST aplikace je entita označena přímo v URL adrese dotazu - „{{host}}/statuses“, zároveň je zvolena HTTP metoda GET. Obecně v REST aplikaci platí, že typ HTTP metody určuje konkrétní CRUD operaci, zatímco URL adresa specifikuje entitu. Kdežto v GraphQL aplikaci všechny dotazy směřují na jednu URL adresu – „{{host}}/graphql“, přičemž všechny dotazy musí bez výjimky odkazovat na HTTP metodu POST. Entita je potom identifikována klíčovým slovem „Statuses“ přímo v těle dotazu. Oba způsoby načítání dat, bez ohledu na zvolené technologii, vrací odpověď ve formátu JSON ve stejném formátu (Obrázek 25).



Obrázek 25. Ukázka načítání dat entity Status

V případě požadavků na REST API je obsah jeho odpovědi vždy konstantní, pokud nedojde k úpravě na straně serveru. V GraphQL si sám klient určuje, jaké data mají být součástí odpovědi na daný požadavek. Pro tento účel existuje v GraphQL introspekce, jenž dokáže z GraphQL schéma získat všechny informace, o které si klient může zažádat. Při introspekci (odkaz na obrázek) se provolává GraphQL URL adresa. Z introspekce lze získat všechny atributy dané entity. Díky tomu uživatel, který nezná funkcionalitu daného GraphQL API, dokáže pochopit jeho koncept a funkčnost. Mimo atributů entity je možno zjistit například jejich popisy, klíčová slova a velikost schématu nebo datové typy.

Pokud tedy chceme vytvořit 2 dotazy pro získání totožných dat, v případě REST aplikace stačí zavolat zmíněnou URL adresu „{{host}}/delivery/outbounds“. Avšak v GraphQL aplikaci je zapotřebí si takový dotaz poskládat vložením všech možných atributů. V současnost existují programy jako Postman nebo GraphQL Playground, které dokážou při psaní dotazu napovídat. K tomu využívají buď již zmíněnou introspekci, nebo nabízejí ruční vložení SDL popisující celé schéma přímo do programu. Jako příklad (Obrázek 26) lze uvést introspekci, která pro entitu „OutboundDelivery“ vrátí její strukturu atributů, jejich název, datový typ, druh datového typu a popis atributu.



Obrázek 26. Příklad použití GraphQL introspekce

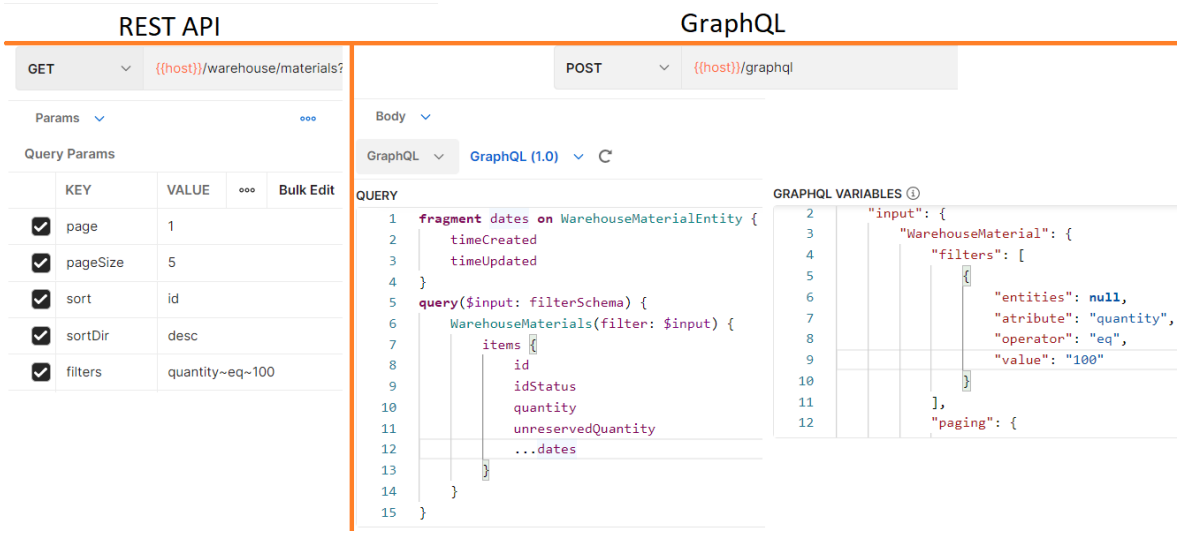
9.2.1 Filtrování

Obě aplikace disponují možností backendová data filtrovat a řadit podle vlastní volby. Načítání je v REST i GraphQL rozděleno na jednotné a hromadné (single a multi) varianty. V případě single varianty dotazu, který má za úkol načíst data konkrétní entity, se očekává kromě URL adresy i číselný identifikátor (id), který reprezentuje záznam v databázi. Při využití hromadné varianty daný identifikátor není zapotřebí uvádět, jelikož se načítají všechny záznamy. Avšak existuje možnost si tyto data filtrovat a řadit podle vlastní potřeby.

Tzn. v každém dotazu, který slouží pro načítání dat, lze nastavit následující parametry:

1. Filtrování, ve kterém se nastavuje:
    a. Atribut, podle kterého se data filtrují
    b. Operátor
    c. Hodnota
2. Řazení
    a. Atribut, podle kterého se data mají seřadit
    b. Směr řazení
3. Omezení počtu zobrazených dat
    a. Počet výsledných záznamů pro zobrazení

Zatímco v REST aplikace se tyto atributy vkládají do hlavičky HTTP dotazu, v GraphQL se umístí za klíčové slovo, které reprezentuje daný dotaz. Na příkladu (Obrázek 27) byl dotaz nastaven tak, aby odpověď dotazu vrátila data pro entitu „WarehouseMaterial“, tyto data jsou vyfiltrována pouze pro materiály s atributem „quantity“ roven hodnotě 100. Současně je v dotaze nastaveno sestupné řazení podle atributu „id“ a počet výsledků omezen na 5. Tento dotaz pro variantu REST i GraphQL vrátí totožný vzorek dat. U GraphQL dotazu je možno si navíc povšimnout využití fragmentu, což je libovolně nastavitelný sub dotaz, který lze vkládat do dotazů. Jejich výhoda spočívá v opětovné použitelnosti. Zároveň lze do sebe fragmenty hierarchicky zanořovat a dotaz tak zůstává přehlednější.



Obrázek 27. Poskládání vyfiltrovaných dotazů v REST a GraphQL variantě

9.3 Operace pro úpravu dat

Pro úpravu dat jsou k dispozici následující operace:

- Vytváření
- Aktualizace
- Mazání

Stejně jako v případě načítání se i u dalších operací v REST části aplikace určuje daná operace, pomocí zvoleného typu HTTP metody a URL adresy. Zároveň všechny zmíněné akce pro úpravu dat existují opět v jednotné i hromadné variantě. V GraphQL znovu všechny dotazy směřují na jednu URL adresu, avšak klíčové slovo nyní zastřešuje označení „mutation“. Pomocí tohoto označení se v GraphQL rozlišují akce pro úpravu a načítání dat. Další rozdíl vyvstává z vkládání dat do dotazu. V REST aplikaci postačí data ve formátu JSON pouze vložit do těla dotazu. Zatímco v GraphQL se data ukládají do připravené struktury GraphQL proměnných, které se musí předat danému sub dotazu v mutaci. Akce pro aktualizaci a vytváření očekávají vstupní data ve formátu JSON. Odlišnosti jednotlivých akcí pro úpravu dat entity „OutboundDelivery“ jsou zaznačeny v tabulce (Tabulka 1).

REST API		***GraphQL***
HTTP metoda	*URL*	*HTTP metoda POST + URL /graphql*

POST	/outbound-delivery
	[$data]
		mutation {
		createOutboundDelivery($data)
		}

POST	/outbound-deliveries
	[$data]
		mutation {
		createOutboundDeliveries($data)
		}

PUT	/outbound-delivery
	[$data]
		mutation {
		updateOutboundDelivery($data)
		}

PUT	/outbound-deliveries
	[$data]
		mutation {
		updateOutboundDeliveries($data)
		}

DELETE	/outbound-delivery/[id]	mutation {
		deleteOutboundDelivery(id: [id])
		}

DELETE	/outbound-deliveries	mutation {
		deleteOutboundDeliveries(ids: [[ids]])
		}

Tabulka 1. Jednotlivé akce pro úpravu dat v REST a GraphQL

9.4 Výhody a nevýhody

Je zapotřebí zdůraznit, že i přes nevýhody dané technologie může být její použití vhodné. Důležitým faktorem je zejména velikost aplikace či API, nad kterým se technologie využívá. V praxi se často využívá kombinace obou zmíněných, jelikož GrahpQL stále v určitých oblastech není schopno REST nahradit.

9.4.1 GraphQL

\+ Všechny dotazy směřují na 1 URL adresu typu POST

1) Rychlost
2) Lepší práce s pamětí
3) Eliminace pollingu
4) Menší zatížení sítě při přenosu
5) Možnost jednoduše ukládat data do bufferů
6) Možnost vytvářet, upravovat a mazat data v jediném dotazu
7) V případě PHP znatelně menší náročnost obsluhy dotazu

+ Strukturu dotazu sestavuje klient
    1) Zamezení načítání nepotřebných dat (overfetching)
    2) Zamezení načítání nedostatečných dat (underfetching)
    3) Klient dostane pouze data, která opravdu potřebuje
    4) Menší vytíženost serverových zdrojů
+ Striktně typové
+ Vychází ze samo popisující konceptu (dokumentace)
+ Jednoduchá práce se zanořenými daty díky vertikálnímu načítání
    1) Počet zanořených entit je téměř neomezený
    2) Možnost samostatně tyto zanořené entity řadit a filtrovat
+ Možnost filtrovat data podle jakéhokoliv atributu v zanořené entitě
+ Vhodný pro multiplatformní použití
+ Lepší ve správě paměti
– Větší potřebná režie při obsluze dotazu
– Větší schéma je zapotřebí optimalizovat
– Není vhodný pro práci se:
    1) Soubory
    2) Daty, které jsou reprezentovány stromovou strukturou (rekurze)

9.4.2 REST API

+ Škálovatelnost
+ Menší potřebná režie oproti GraphQL
+ Jednodušší na vytvoření
+ Vhodný pro práci se soubory
+ Možnost customizace výstupu podle potřeb
– Ve větších aplikací může díky pollingu docházet k pomalejší responzivitě
– Počet úrovní zanořených entit je omezen na 1
– V načítání dat je pomalejší vůči GraphQL
– Pro 1 dotaz existuje pouze 1 handler/resolver
– Slabé/žádné typování dat
– Dokumentace

		REST	GraphQL
Resources	*Identification*	Yes	Yes
	HTTP Usage	Yes	Yes
	JSON Response	Yes	Yes
	Object Identify	Endpoint	Separate from how is fetch
	Determination of Shape and Size	Server	Client
URL routes vs. GraphQL schemas	*List of Operations*	List of Endpoints	List of fields (at Query and Mutation)
	The distinction between Reading and Writing	Yes	Yes
	Multiple Calls to Relate Resources	Yes	No
	First-class Concept	No	Yes
	Modify Reading into Writing/ Writing into Reading	HTTP verbs	Keyword in the query
Route handlers vs. resolvers	*Function Call*	Endpoints	Fields
	Handle Networking Boilerplate	Using frameworks or libraries	Using frameworks or libraries

Tabulka 2. Souhrn rozdílů mezi REST a GraphQL [25]

9.5 Benchmarking

Pro testování obou technologií byla zvolena komponenta Apache HTTP server benchmarking tool (AB). Výhodou tohoto nástroje je velká škála možných nastavení a variant testů. Na rozdíl od jiných alternativ AB nevyužívá testování pomocí smyček, ale dokáže vytvořit kontinuální zatížení serveru nebo aplikace. Díky tomu se získaná data neliší od dat z běžného provozu.

9.6 Testování načítání dat REST vs. GraphQL

Jedná se o test, při kterém se spustí REST a GraphQL akce, jež jsou nastaveny tak, aby vrátily totožný vzorek dat. S každým testem se inkrementuje počet načtených záznamů. (Tabulka 3) Dotaz se skládá z:

- Dat pocházející z entity „Material“
- Žádné navázané entity
- Entita obsahuje 22 atributů
- Celkový počet záznamů v databázi: 13500
- Pro každý test se spustí 5 dotazů, jejichž výsledky jsou zprůměrovány

Při prvním testování se měří následující hodnoty:

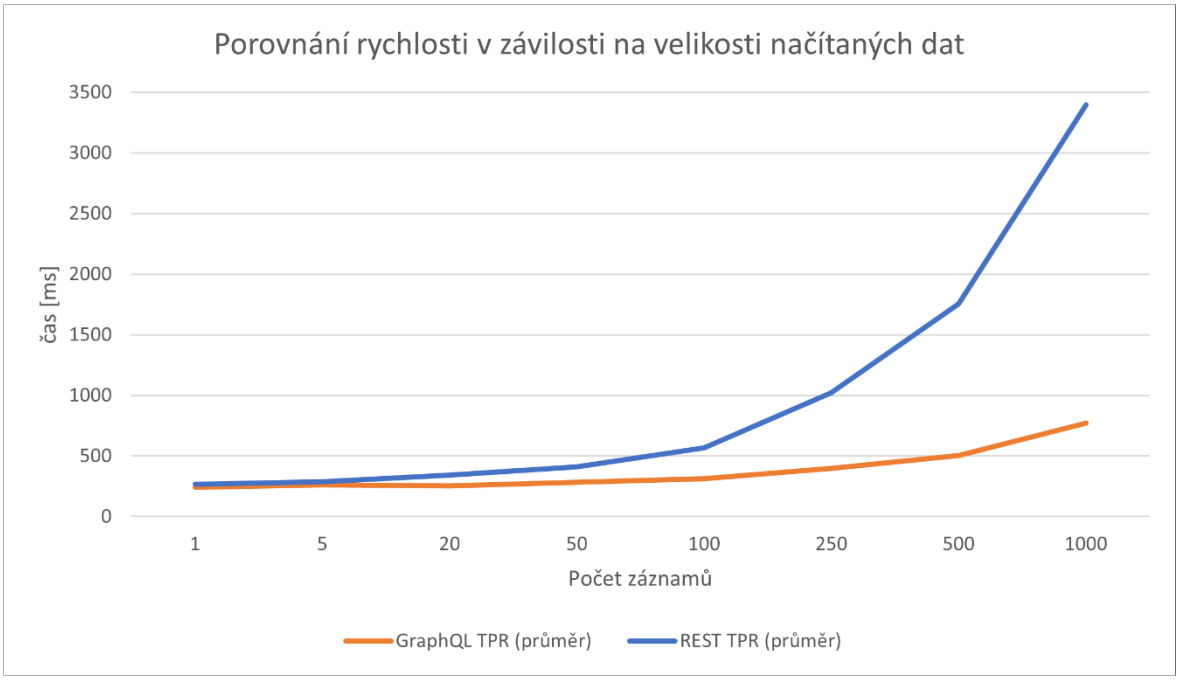
- # - číslo testu
- DC (data count) – počet záznamů v dotaze, které se načítají
- CR (completed requests) – celkový počet úspěšně provedených dotazů/vln
- TPR (time per request) [ms] – průměrný čas pro 1 dotaz
- TTFT (total time taken for test) [s] – doba trvání testu
- TR (transfer rate) [Kbytes/s] – přenosová rychlost
- TT (total transfer) [bytes] – množství odeslaných dat
- CL (concurrency level) – počet souběžně odeslaných dotazů
- TPR (celkový průměr) – průměrná doba 1 dotazu v 1 vlně
- N – celkový počet všech vykonaných dotazů

$$TPR(celkový\ průměr) = \frac{TPR}{CL}$$

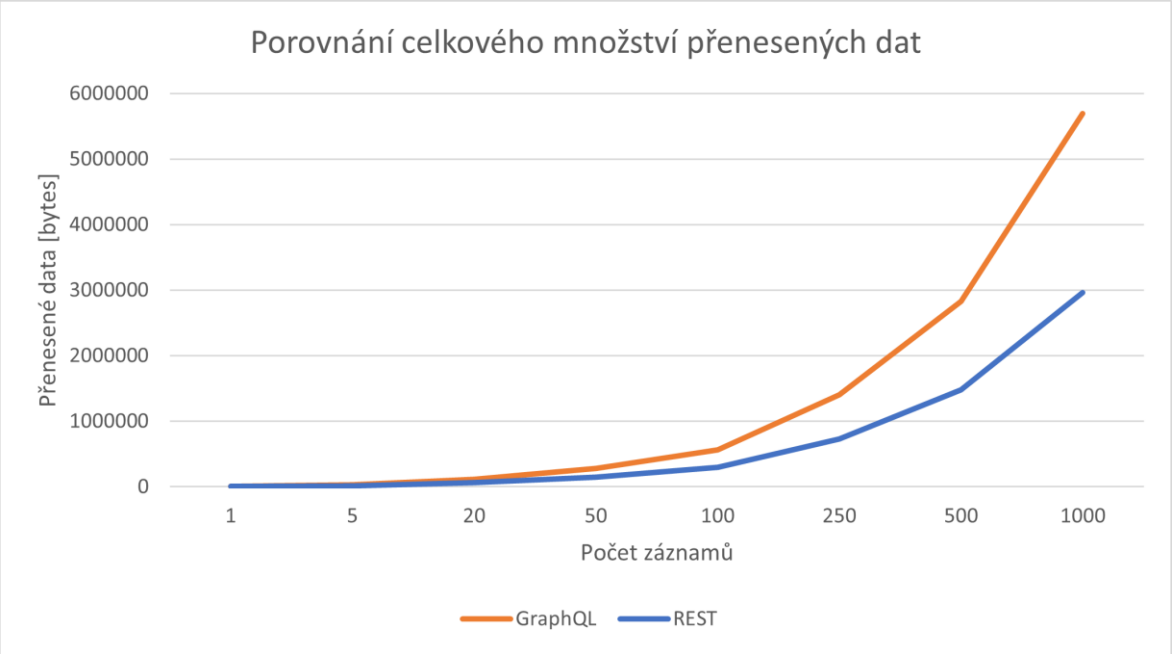
$$N = CR * CL$$

				[ms]	[s]	[Kbytes/sec]	[bytes]
#	Typ	DC	CR	TPR (průměr)	TTFT	TR	TT
1	REST	1	5	266	1,33	4,82	6570
	GraphQL	1	5	240	1,43	9,38	9155
2	REST	5	5	286	1,43	12,57	18415
	GraphQL	5	5	262	1,30	27,37	32095
3	REST	20	5	343	1,71	35,20	61895
	GraphQL	20	5	252	1,26	92,51	114940
4	REST	50	5	411	2,05	71,33	149950
	GraphQL	50	5	282	1,41	199,90	283900
5	REST	100	5	569	2,86	102,29	298075
	GraphQL	100	5	313	1,57	356,95	564770
6	REST	250	5	1021	5,10	140,49	734125
	GraphQL	250	5	398	1,99	692,61	1405245
7	REST	500	5	1756	8,77	164,20	1476115
	GraphQL	500	5	504	2,52	1099,63	2831250
8	REST	1000	5	3395	16,97	170,42	2962315
	GraphQL	1000	5	770	3,94	1411	5695640

Tabulka 3. Soupis naměřených výsledků



Obrázek 28. Porovnání rychlosti GraphQL vs. REST



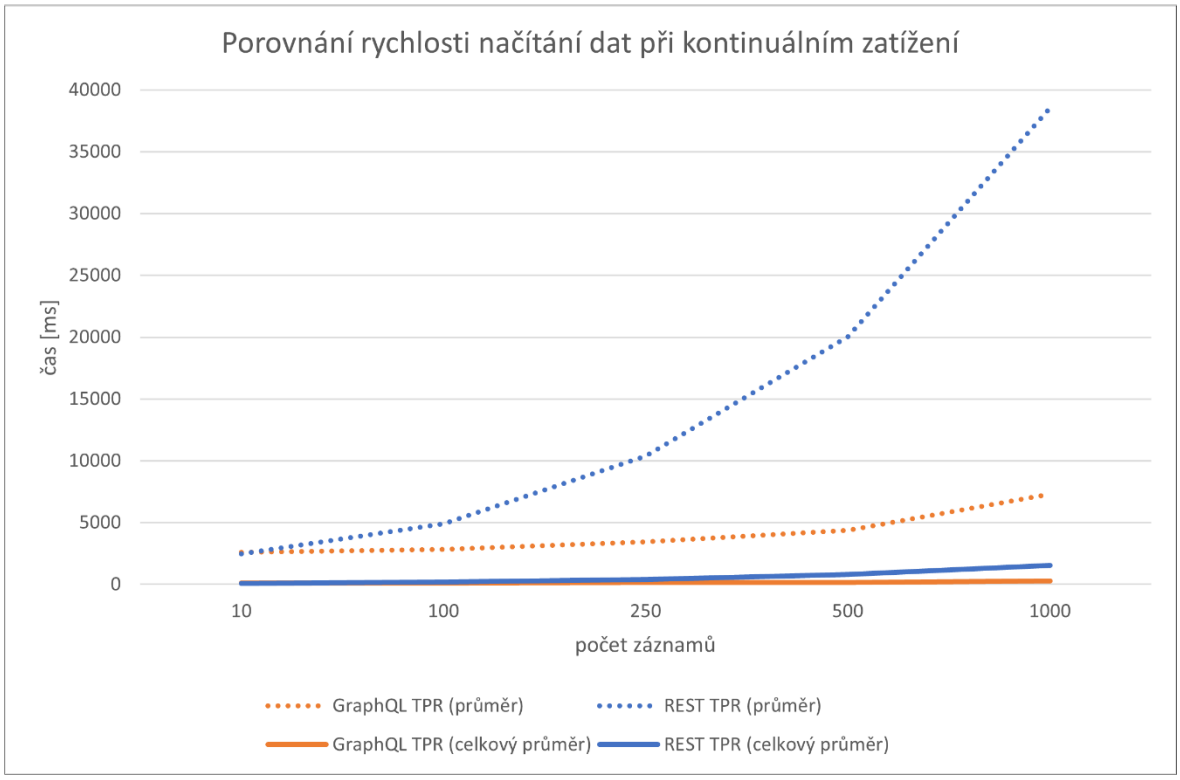
Obrázek 29. Porovnání množství přenesených dat GraphQL vs. REST

9.7 Stress test

Cílem stress testu bylo otestovat chování obou aplikací ve větším provozu a zatížení. Výsledkem je 5 provedených testů, přičemž každý test obsahuje 2 sub testy. Každý sub test odeslal na REST a GraphQL aplikaci 4 vlny dotazů. Každá vlna se skládá z 25 souběžně odeslaných dotazů, celkově se tedy každý sub test skládal ze 100 dotazů.

					[ms]	[ms]	[Kbytes/sec]	[bytes]
#	Typ	DC	CR	CL	TPR (průměr)	TPR (celkový průměr)	TR	TTFT
1	REST	**10**	100	25	**2457**	**98**	65,7	9,8
	GraphQL	**10**	100	25	**2595**	**103**	113,0	10,4
2	REST	**100**	100	25	**4919**	**196**	295,9	19,7
	GraphQL	**100**	100	25	**2845**	**113**	982,5	11,4
3	REST	**250**	100	25	**10365**	**414**	345,8	41,5
	GraphQL	**250**	100	25	**3424**	**136**	2014,0	13,7
4	REST	**500**	100	25	**20059**	**802**	359,3	80,2
	GraphQL	**500**	100	25	**4371**	**174**	3167,0	17,5
5	REST	**1000**	100	25	**38560**	**1542**	375,1	154,2
	GraphQL	**1000**	100	25	**7296**	**291**	3814,0	29,2

Tabulka 4. Soupis naměřených hodnot stress testu



Obrázek 30. Porovnání rychlosti GraphQL vs. REST při zatížení

Obrázek 31. Porovnání přenosové rychlosti GraphQL vs. REST při zatížení

ZÁVĚR

Ze získaných dat lze zjistit, že GraphQL disponuje lepší rychlostí v načítání dat. Na menším vzorku dat v souvislosti jeho větší režií je na tom v porovnání vůči REST velmi podobně. Avšak na velkém množství dat je REST jednoznačně pomalejší (Obrázek 28). Tento problém je způsoben z velké části objektovým způsobem plnění dat v REST aplikaci. Na druhou stranu u REST aplikace bylo zaznamenáno menší množství přenesených dat (Obrázek 29), což je způsobeno strukturou GraphQL dotazů a jejich větší velikostí.

Při implementaci GraphQL nebo REST technologií je zapotřebí brát v úvahu jejich výhody, nevýhody a účel, za jakým bude technologie využívána. Správný výběr je poměrně nejednoznačný, jelikož závisí na celé řadě faktorů. Pokud budeme například vytvářet aplikaci či službu pro práci se soubory, potom implementovat GraphQL nedává smysl. Na druhou stranu je zapotřebí brát v potaz, proč GraphQL vlastně vzniklo a uvědomit si, že řeší značnou část nedostatků, kterými REST disponuje. Kupříkladu jeho slabé typování – tzn. pokud je vyžadován konzistentní a spolehlivý výstup, může být GraphQL ideální volbou. Mezi další důležité faktory lze zařadit současný stav aplikace či služby – samotná implementace se bude lišit v případě vytváření aplikace nebo služby nové nebo v případě již existujícího, které je určeno k optimalizaci či přepisu. V takovém případě se nabízí využití obou technologií současně. REST část využít pro jednoduché a méně náročné dotazy a GraphQL využít v dotazech náročnějších. Přičemž u GraphQL, v jakékoliv jeho formě implementace, se musí rozhodnout o jeho způsobu implementace, zda vytvářet nejdříve schéma či kód. Závěrem se dá vyvodit následující: pokud je cílem vytvořit středně velkou nebo velkou aplikaci, jeví se GraphQL jako bezprecedentní volba, i když je jeho zavedení oproti REST náročnější. Jestliže aplikace či služba nepotřebuje pro svoje fungování načítat velké množství dat, lze bez obav zvolit REST.

SEZNAM POUŽITÉ LITERATURY

[1] THOMAS, Erl. *Service-Oriented Architecture: Analysis and Design for Services and Microservices*. Pearson, 2016. ISBN 9780133858587.

[2] SHETH, Amith. *Semantic Web: Ontology and Knowledge Base Enabled Tools, Services, and Applications*. IGI Global, 2013. ISBN 9781466636101.

[3] CERAMI, Ethan. *Web Services Essentials (O'Reilly XML)*. O'Reilly Media, 2002. ISBN 9780596002244.

[4] AKSIT, Mehmet, Bedir TEKINERDOGAN a Lodewijk BERGMANS. *The Six concerns for Separation of Concerns* [online]. 2001, 4 [cit. 2022-04-02]. Dostupné z: https://www.researchgate.net/publication/216884785_The_Six_concerns_for_Separation_of_Concerns

[5] CLINTON, Wong. *HTTP Pocket Reference: Hypertext Transfer Protocol*. O'Reilly Media, 2000. ISBN 9781565928626.

[6] DUSSEAULT, Lisa. *WebDAV: Next-Generation Collaborative Web Authoring: Next-Generation Collaborative Web Authoring*. Prentice Hall, 2003. ISBN 9780130652089.

[7] STEPHEN, Ludin. *Learning HTTP/2: A Practical Guide for Beginners*. O'Reilly Media, 2017. ISBN 9781491962442.

[8] KANJILAL, Joydip. *ASP.NET Web API: Build RESTful Web Applications and Services on the .NET Framework*. Packt Publishing, 2013. ISBN 9781849689748.

[9] REST API: Key Concepts, Best Practices, and Benefits. In: *Altexsoft* [online]. 2022 [cit. 2022-04-06]. Dostupné z: https://www.altexsoft.com/blog/rest-api-design/

[10] MASSÉ, Mark. *Rest Api Design Rulebook*. O'Reilly Media, 2011. ISBN 9781449317904.

[11] BIEHL, Matthias. *RESTful API Design*. CreateSpace Independent Publishing Platform, 2016. ISBN 9781514735169.

[12] BOJINOV, Valentin. *RESTful Web API Design with Node.js 10: Learn to create robust RESTful web services with Node.js, MongoDB, and Express.js*. Packt Publishing, 2018. ISBN 9781788623322.

[13] CRISTIAN, Darie a Brinzarea BOGDAN. *AJAX and PHP: Building Modern Web Applications*. Packt Publishing, 2010. ISBN 978-1847197726.

[14] JavaScript Ajax: What is Ajax?. In: *TutorialRepublic* [online]. 2022 [cit. 2022-04-04]. Dostupné z: https://www.tutorialrepublic.com/javascript-tutorial/javascript-ajax.php

[15] BITTENCOURT, Mario. When REST is not Enough. In: *Medium* [online]. 2021 [cit. 2022-04-02]. Dostupné z: https://medium.com/ssense-tech/when-rest-is-not-enough-ea4604c64f5e

[16] DOERRFELD, Bill. Is REST Still A Good API Design Style to Use?. In: *Nordic APIs* [online]. 2019 [cit. 2022-04-02]. Dostupné z: https://nordicapis.com/is-rest-still-a-good-api-design-style-to-use/

[17] SANDOVAL, Kristopher. Stop Polling and Consider Using REST Hooks. In: *Nordic APIs* [online]. 2017 [cit. 2022-04-02]. Dostupné z: https://nordicapis.com/stop-polling-and-consider-using-rest-hooks/

[18] JACKSON, Joab. How Synchronous REST Turns Microservices Back into Monoliths. In: *The New Stack* [online]. 2016 [cit. 2022-04-02]. Dostupné z: https://thenewstack.io/synchronous-rest-turns-microservices-back-monoliths

[19] MANYIKA, James, Michael CHUI a Peter BISSON. The Internet of Things: Mapping the Value Beyond The Hype. In: *McKinsey & Company* [online]. 2015 [cit. 2022-04-02]. Dostupné z: https://www.mckinsey.com/~/media/mckinsey/industries/technology%20media%20and%20telecommunications/high%20tech/our%20insights/the%20internet%20of%20things%20the%20value%20of%20digitizing%20the%20physical%20world/the-internet-of-things-mapping-the-value-beyond-the-hype.pdf

[20] ORTIZ, a Jesús HAMILTON. *Industry 4.0: Current Status and Future Trends*. Intechopen, 2020. ISBN 978-1-83880-093-2.

[21] GULER, Busra. Where is the Industry 4.0 in our life. In: *AIE Internship Erasmus* [online]. 2019 [cit. 2022-05-08]. Dostupné z: https://aie-internship.com/where-is-the-industry-4-0-in-our-life-by-busra-guler/

[22] MONTECUOLLO, Michael. Native or Web-Based? Selecting the Right Approach for Your Mobile App. In: *UX Magazine* [online]. 2014 [cit. 2022-04-02]. Dostupné z: https://uxmag.com/articles/native-or-web-based-selecting-the-right-approach-for-your-mobile-app

[23] GUSTAFSON, Aaron. Yes, That Web Project Should Be a PWA. In: *A List Apart* [online]. Application Development, 2017 [cit. 2022-04-02]. Dostupné z: https://alistapart.com/article/yes-that-web-project-should-be-a-pwa/

[24] HAAFIZ WAHEED-UD-DIN, Ahmad. *Building RESTful Web Services with PHP 7: Lumen, Composer, API testing, Microservices, and more*. Packt Publishing, 2017. ISBN 978-1-78712-774-6.

[25] PANG, Chung-Yeung. *Software Engineering for Agile Application Development: Advances in Computer and Electrical Engineering*. IGI Global, 2020. ISBN 9781799825319.

[26] BUNA, Samer. *Learning GraphQL and Relay*. Packt Publishing, 2016. ISBN 9781786465757. Dostupné také z: https://search.ebscohost.com/login.aspx?direct=true&db=nlebk&an=1344059&scope=site

[27] FUHRMANN, Axel. The Rick and Morty API GraphQL. In: *The Rick and Morty API* [online]. 2022 [cit. 2022-04-03]. Dostupné z: https://rickandmortyapi.com/graphql

[28] MAVROUDEAS, Georgios, Guillaume BAUDART a Alan CHA. *Learning GraphQL Query Cost*. 36th IEEE/ACM International Conference, 2021, 1146-1150 s. ISBN 9781665403375. Dostupné z: doi:10.1109/ASE51524.2021.9678513

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

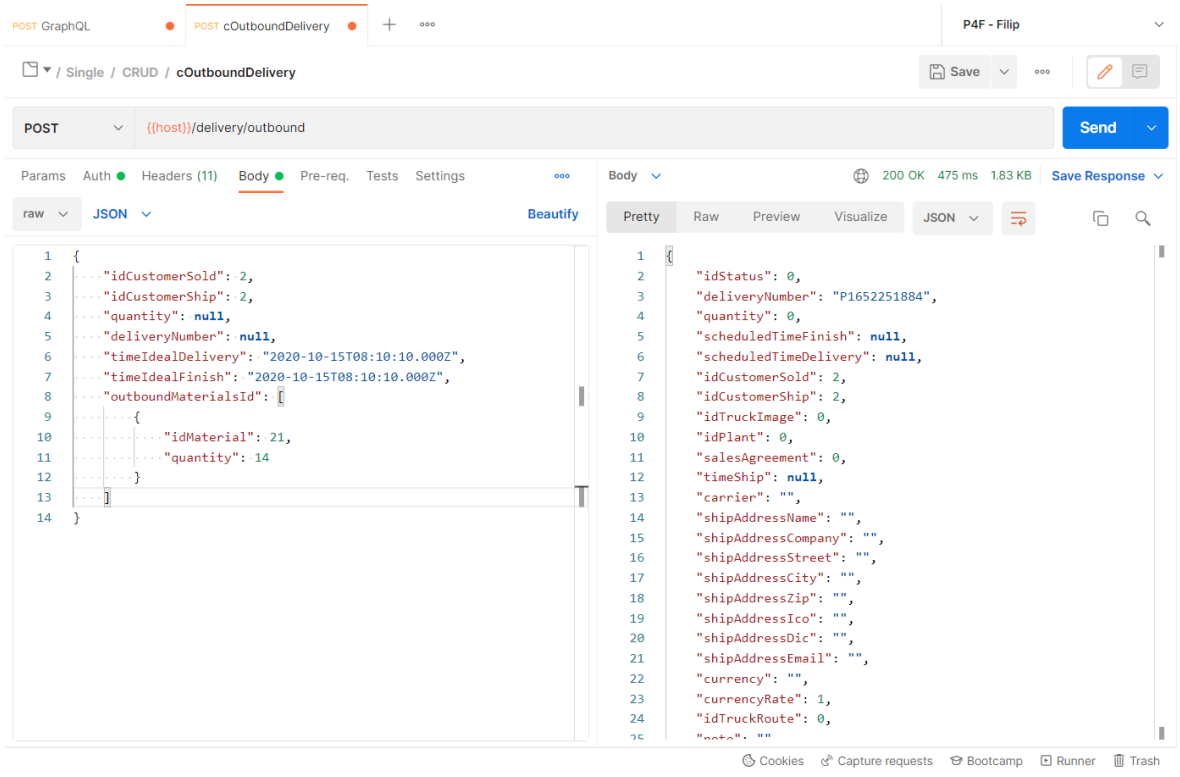
AB	Apache Bench
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
CoD	Code on Demand
CRUD	Create Read Update Delete
HTML	Hypertext Markup Language
HTTP	HyperText Transfer Protocol
IoT	Internet of things
JSON	JavaScript Object Notation
OWL	Web Ontology Language
PHP	Hypertext Preprocessor
QL	Query Languager
RDL	Resource Description Language
REST	Representational State Transfer
RPC	Remote Procedure Call
SDL	Schema Definition Language
SOAP	Simple Object Access Protocol
SoC	Separation of Concerns
TCP/IP	Transmission Control Protocol/Internet Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WSDL	Web Service Description Language
XML	Extensible Markup Language

SEZNAM OBRÁZKŮ

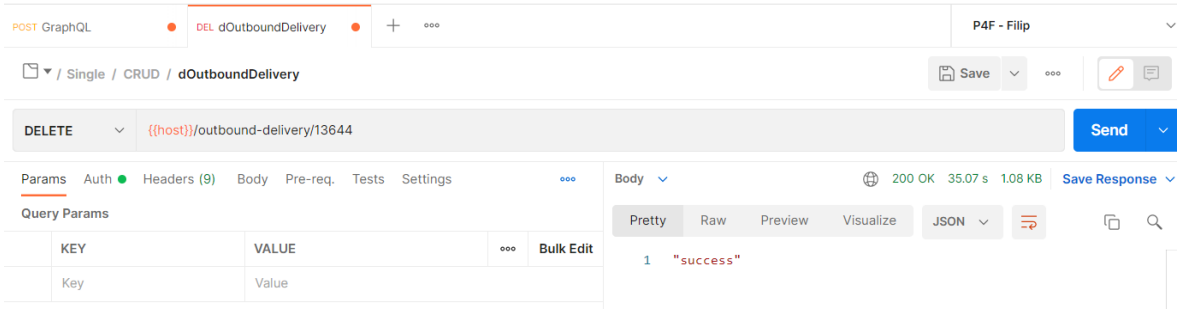
SEZNAM TABULEK

SEZNAM PŘÍLOH

PŘÍLOHA P I: REST

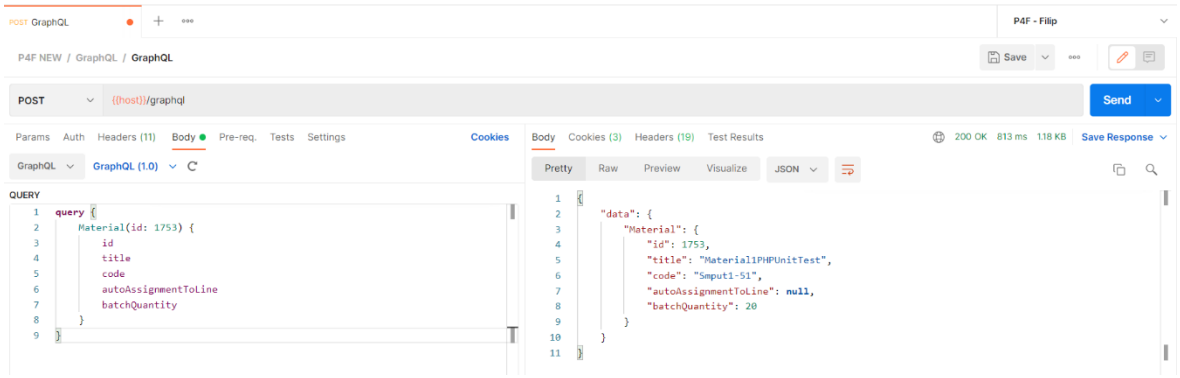


P I. 1 – Operace vytvoření záznamu entity „OutboundDelivery“

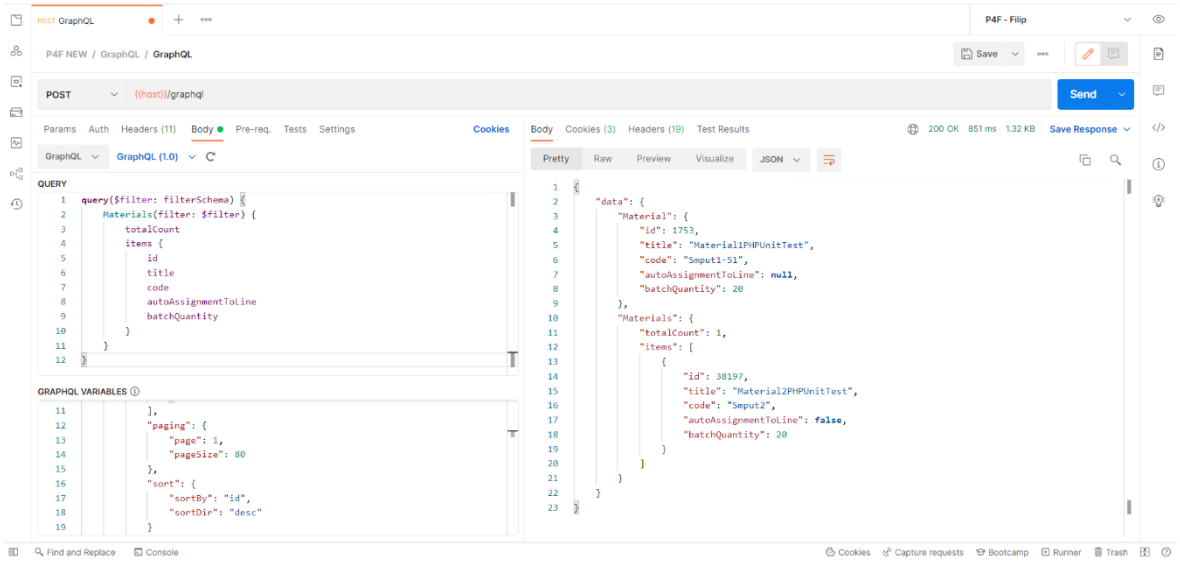


P I. 2. Operace smazání záznamu entity „OutboundDelivery“

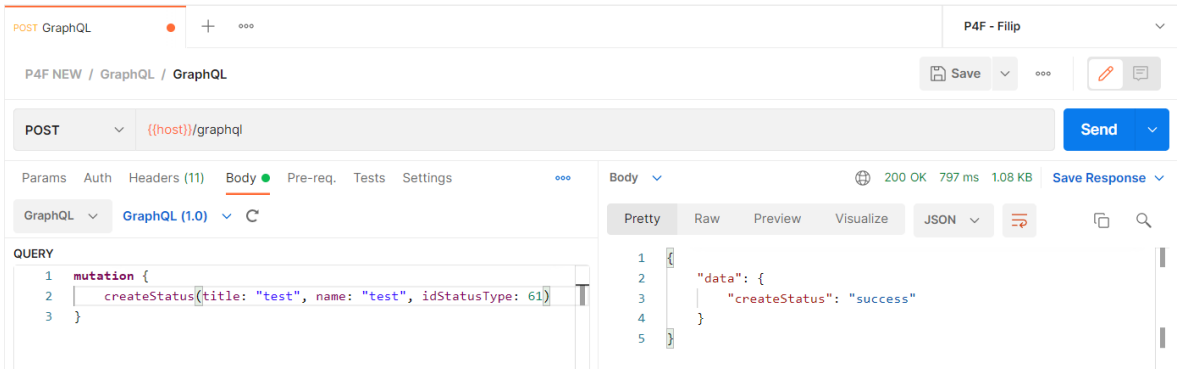
PŘÍLOHA P II. GRAPHQL



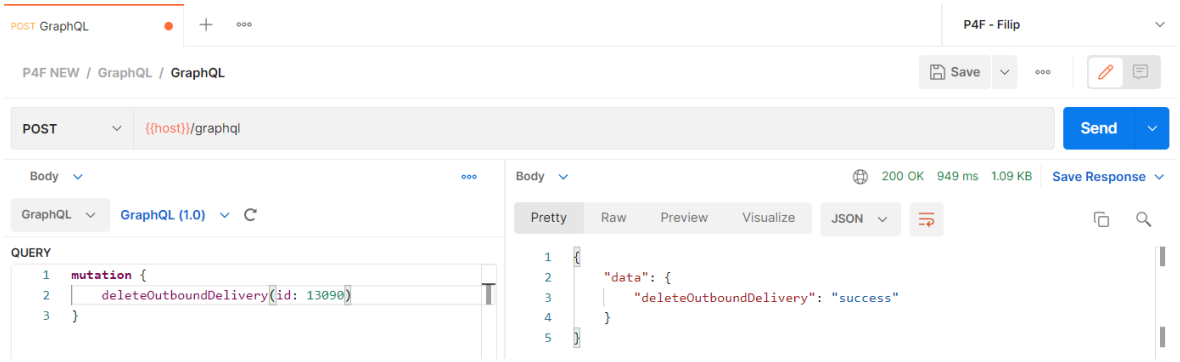
P II. 1 Operace načtení jednoho záznamu entity „Material“



P II. 2. Operace načtení všech záznamů entity „Material“ s využitím filtrů

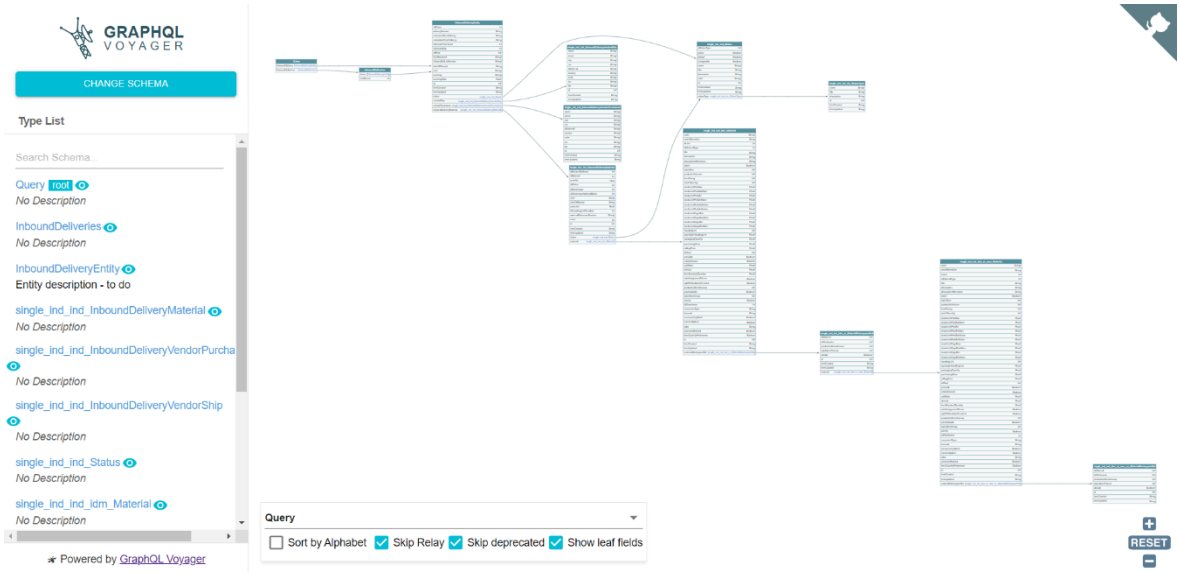


P II. 3 Operace vytvoření záznamu entity „Status“



P II. 4 Operace smazání záznamu entity „Status“

PŘÍLOHA P III. POUŽITÉ NÁSTROJE



P III. 1 Ukázka GraphQL Voyager pro grafické znázornění struktury entity

```
C:\Windows\System32\cmd.exe
c:\_productoo\p4f\program\apache\2.4.39\bin>ab -p graphql/query.txt -T 'application/json' http://p4f-backend/graphql
This is ApacheBench, Version 2.3 <$Revision: 1843412 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking p4f-backend (be patient).....done


Server Software:        Apache/2.4.39
Server Hostname:        p4f-backend
Server Port:            80

Document Path:          /graphql
Document Length:        1138476 bytes

Concurrency Level:      1
Time taken for tests:   1.037 seconds
Complete requests:      1
Failed requests:        0
Total transferred:      1139128 bytes
Total body sent:        922
HTML transferred:       1138476 bytes
Requests per second:    0.96 [#/sec] (mean)
Time per request:       1037.193 [ms] (mean)
Time per request:       1037.193 [ms] (mean, across all concurrent requests)
Transfer rate:          1072.54 [Kbytes/sec] received
                        0.87 kb/s sent
                        1073.41 kb/s total

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.0      0       0
Processing:  1037 1037   0.0   1037    1037
Waiting:     1033 1033   0.0   1033    1033
Total:       1037 1037   0.0   1037    1037
```

P III. 2 Ukázka ApacheBench pro testování

PŘÍLOHA P IV. CD

Součástí fyzické verze diplomové práce je přiložené CD, jehož obsahem je DP ve formátu PDF/A a zdrojové kódy obou aplikací.