Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Diplomová práce

Srovnání přístupů GraphQL a REST při vývoji aplikačního rozhraní

Matěj Povolný

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Matěj Povolný

Informatika

Název práce

Srovnání přístupů GraphQL a REST při vývoji aplikačního rozhraní

Název anglicky

Comparison of GraphQL and REST approaches for API development

---

Cíle práce

Cílem práce je srovnání odlišných a v současnosti nejrozšířenějších architektonických přístupů REST a GraphQL při tvorbě aplikačního programového rozhraní (API) z pohledu různorodých požadavků jako je rychlost, objem přenesených dat a bezpečnost.

Metodika

Práce je zpracována ve dvou částech – teoretické a praktické.

Metodika pro zpracování teoretické části práce vychází ze studia odborných informačních zdrojů, na jejichž základě budou stanovena teoretická východiska a konceptuální základ pro praktickou část práce.

Praktická část práce spočívá v přípravě běhového prostředí Node.js, vývoji komponent s využitím frameworku Express.js a databáze MongoDB a dále uskutečnění srovnávacích pokusů. Výstupem praktické části je zhodnocení odlišných přístupů při tvorbě API v otázce rychlosti přenosu, objemu přenesených dat a bezpečnosti.

Doporučený rozsah práce

60-80 stran

Klíčová slova

api, graphql, rest, webová služba

Doporučené zdroje informací

Brown, Ethan. Web Development with Node and Express: Leveraging the JavaScript Stack. O'Reilly Media, Incorporated, 2019.

Doglio, Fernando. REST API Development with Node.js: Manage and Understand the Full Capabilities of Successful REST Development. Apress, 2018.

Jin, Brenda, et al. Designing Web APIs: Building APIs that Developers Love. O'Reilly Media, 2018.

Wieruch, Robin. The Road to GraphQL. Robin Wieruch, 2018.

Předběžný termín obhajoby

2023/24 LS – PEF

Vedoucí práce

Ing. Martin Pelikán, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 5. 9. 2023

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 3. 11. 2023

doc. Ing. Tomáš Šubrt, Ph.D.

Děkan

V Praze dne 19. 03. 2024

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Srovnání přístupů GraphQL a REST při vývoji aplikačního rozhraní" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 31.3.2024 _____

Poděkování

Rád bych touto cestou poděkoval panu doktoru Pelikánovi za cenné rady v rámci vedení práce.

Srovnání přístupů GraphQL a REST při vývoji aplikačního rozhraní

Abstrakt

Diplomová práce se zabývá problematikou tvorby aplikačního programového rozhraní v prostředí webových aplikací z hlediska srovnání dvou v současné době nejvyužívanějších přístupů, jimiž jsou REST a GraphQL.

Srovnání přístupů bylo učiněno nejprve z hlediska kvantitativního srovnání výkonu pomocí měření rychlosti vyřízení požadavků pro čtení, zápis, aktualizaci a výmaz dat realizovaný každým standardem a také náročnosti těchto operací na datový přenos. Na měření výkonu a měření datové zátěže bylo navázáno vyhodnocením uživatelské přívětivosti z hlediska přípravy prostředí, tj. webové aplikace využívající jak REST, tak GraphQL, které bylo vytvořeno za účelem realizace srovnávacího experimentu. Uživatelská přívětivost byla vyhodnocena přes skóre uživatelské přívětivosti definované v teoretické části práce.

Výsledky srovnávacího experimentu ukázaly, že ve většině případů užití a jejich variant z hlediska počtu opakování požadavků je REST přístupem vykazujícím lepší výkon, pokud jde o rychlost vyřízení požadavků a také přístupem vykazujícím menší nároky na datový přenos z hlediska velikosti odpovědi ze strany serveru. Naopak vyhodnocení uživatelské přívětivosti ukázalo, že GraphQL je přístupem, který je uživatelky přívětivější. Kromě předložených výsledků a jejich interpretace závěr práce dále uvádí, že vhodnost využití každého z přístupů je silně závislá na prostředí a podmínkách, do kterého jsou zaváděny.

Klíčová slova: webová služba, webová aplikace, API, REST, GraphQL, vývoj, MongoDB, uživatelská přívětivost

Comparison of GraphQL and REST approaches to application interface development

Abstract

The diploma thesis deals with the topic of creating application programming interfaces in the environment of web applications from the point of view of comparison of two currently most used approaches, which are REST and GraphQL.

The comparison of the approaches was first made in terms of a quantitative performance comparison by measuring the speed of processing requests for reading, writing, updating and deleting data implemented by each standard and the data transfer requirements of these operations. The performance measurements and data load measurements were followed by evaluating the user-friendliness in terms of preparing the environment, i.e., a web application using both REST and GraphQL, which was created to carry out the benchmarking experiment. The user-friendliness was evaluated through the user-friendliness score defined in the theoretical part of the thesis.

The results of the comparative experiment showed that in most of the use cases and their variations in terms of number of requests retries, the REST approach exhibited better performance in terms of request handling speed and also the approach exhibited better data transfer requirements in terms of response size from the server side. On the other hand, the user-friendliness evaluation showed that GraphQL is the approach that is more user-friendly. In addition to the results presented and their interpretation, the conclusion of the paper further states that the applicability of each approach is strongly dependent on the environment and conditions in which they are introduced.

Keywords: web service, web application, API, REST, GraphQL, development, MongoDB, user friendliness

Obsah

1 Úvod

S pokročilou a neustále se rozvíjející digitalizací a s tím spojeným rozvojem informačních technologií dochází nejen v posledních letech ke zvýšené potřebě interakcí za účelem výměny dat mezi systémy, které jsou součástí informačního ekosystému. Za dobu existence internetu a s ním spojených technologií bylo v průběhu let využíváno velkého množství přístupů pro výměnu dat mezi jednotlivými systémy.

V současné době patří mezi v praxi nejvyužívanější přístupy pro tvorbu aplikačních rozhraní v prostředí webových aplikací REST architektura a GraphQL. Oba z těchto přístupů skýtají mnoho výhod a oproti architekturám a přístupům, které jim předcházely, přinesly významný pokrok a do značné míry zásadně a trvale ovlivnily vývoj software produktů.

Diplomová práce se soustředí na srovnání těchto dvou přístupů, pokud jde o tvorbu aplikačního programového rozhraní, a to především z hlediska optimalizace pro rychlost přenosu dat a celkovému množství přenesených dat. Dále je vyhodnocena uživatelská přívětivost, pokud jde o práci s jednotlivými technologiemi, a to z hlediska časové náročnosti programování, pracnosti definované počtem napsaných znaků a srozumitelnosti syntaxe kódu.

Práce ke srovnání přistupuje ve dvou samostatných, avšak neoddělitelných částech. V první, teoretické části, se práce věnuje vydefinování základních konceptů stojících za výměnou dat v prostředí internetu tak, jak ji dnes známe. Dále se zaměřuje na popis obou ze zkoumaných přístupů pro tvorbu aplikačního rozhraní a jejich principů, které pokládají základy a kladou předpoklady pro vydefinování metodiky pro uskutečnění srovnávacích pokusů v praktické části práce. Závěr teoretické části definuje samotnou metodiku pro vykonání srovnávacích experimentů v podobě definice jednotného procesu experimentu, který je možné opakovat jak pro přístup REST tak pro GraphQL. Na vydefinování metodiky pro kvantitativní srovnání je navázáno popisem metodiky pro vyhodnocení uživatelské přívětivosti obou z přístupů.

Druhá, praktická část práce, se věnuje přípravě prostředí pro realizaci srovnávacích experimentů v podobě webové aplikace umožňující provádět operace s daty vůči databázi. Po přípravě prostředí dojde k realizaci samotných srovnávacích pokusů z hlediska rychlosti a velikosti datového přenosu a následného vyhodnocení uživatelské přívětivosti obou z přístupů, kterou hodnotící zaznamenal v rámci přípravy prostředí. Po provedení

srovnávacích pokusů dochází k zaznamenání výsledků, jejich analýze a interpretaci. Nad výsledky je učiněna diskuze a závěr diplomové práce.

2 Cíl práce a metodika

2.1 Cíl práce

Cílem práce je provést srovnání mezi architektonickými přístupy využívanými při tvorbě aplikačních programových rozhraní (API) v prostředí webových aplikací především z hlediska rychlosti přenosu dat a požadavků na objem přenesených dat. Kvantitativní závěry srovnání budou doprovozeny srovnáním týkajícím se pracnosti a celkové uživatelské přívětivosti spojené s přípravou REST nebo GraphQL rozhraní, která byla vyžadována pro přípravu prostředí pro provedení srovnávacích pokusů.

Výstupem práce je získání výsledků srovnávací analýzy týkající se rychlosti a minimálního potřebného datového přenosu každého z přístupů společně s výsledky testů uživatelské přívětivosti. Tyto výsledky jsou interpretovány autorem a porovnány v rámci diskuze, na což je v závěru práce navázáno doporučením autora pro aplikaci každého z přístupů.

2.2 Metodika

První, teoretická, část diplomové práce se zabývá základními termíny a principy vývoje webových služeb a s tím spojených technologií a architektur. Kromě vydefinování základní problematiky za účelem interpretace samotných výsledků srovnávacího experimentu tato část umožní čtenáři pochopit obecné principy webových služeb, které jsou potřebné pro vyhotovení aplikace, s pomocí které budou samotné pokusy provedeny. Bližší specifikace provedení srovnávacího experimentu je blíže popsána v teoretické části práce.

Praktická část se věnuje experimentu samotnému, kterému předchází příprava a vývoj prostředí, v rámci kterého bude experiment proveden. Na experiment samotný je je navázáno analýzou nashromážděných výsledků. S využitím znalostí z teoretické části práce dojde k interpretaci výsledků.

3 Teoretická východiska

3.1 Webová služba

W3C konsorcium definuje webovou službu jako softwarový systém, který podporuje interoperabilní interakci mezi stroji prostřednictvím sítě. Má rozhraní popsané ve strojově zpracovatelném formátu (konkrétně v jazyce pro definici webových služeb WSDL). (1)

V dnešním slova smyslu lze webovou službu popsat jednodušeji jako cokoliv, co přestavuje aplikační rozhraní mezi dvěma aplikacemi a komunikuje přes internetový protokol, nejčastěji HTTP. (2)

Webové služby tedy představují základní komunikační nástroj, který je využíván ke komunikaci mezi dvěma (a/nebo více) aktéry v prostředí internetu. (3) Bez toho, aniž by si to uvědomoval, využívá běžný uživatel na denní bázi množství internetových služeb a produktů, které závisí právě na komunikaci pomocí webových služeb. Webové služby jsou stavěné tak, aby plnily konkrétní úkol nebo sadu úkolů. Pod těmito úkoly si lze přestavit například odeslání požadavku na vyhledání zboží v online katalogu nebo odeslání e-mailu. (4)

Webové služby jsou definovány jako samostatné, modulární, distribuované, dynamické aplikace, které lze popsat, publikovat, umístit nebo vyvolat prostřednictvím sítě. Mohou být lokální, distribuované nebo webové. (5)

Jsou postaveny na otevřených standardech, jako jsou TCP/IP, HTTP, Java, HTML a XML. (5) To zapříčiňuje, že jsou webové služby ve svém jádru platformně nezávislé v tom smyslu, že mohou být využívány bez ohledu na hardware a software prostředí, na kterém jsou implementovány a s kterým interagují. Podobně jako u nezávislosti na platformě se nezávislost webové služby váže i na programovací jazyk, ve kterém je napsána. (4)

3.1.1 API

Webová služba je nicméně příliš široký pojem, než který nám stačí pro správné vydefinování problematiky pro nadcházející kapitoly. Specifickou podskupinou webových služeb je tzv. aplikační programové rozhraní (API, z anglického Application programming interface).

Webové API rozhraní je specifický typ webové služby, kterým aplikace jiným aplikacím v rámci internetu umožňuje komunikaci a výměnu dat za předem známých formalizovaných pravidel pro oba aktéry. Jinými slovy lze tuto interakci popsat jako mechanismus, který umožňuje aplikaci nebo službě přistupovat ke zdroji v rámci jiné aplikace nebo služby. Aplikace nebo služba, která žádá a provádí přístup, se nazývá klient a aplikace nebo služba obsahující zdroj se nazývá server. (6) (7)

Dále, přestože jsou rozhraní API navržena tak, aby spolupracovala s jinými programy, jsou taktéž určena k tomu, aby jim porozuměli a používali lidé, kteří tyto jiné programy vytváří. (6)

3.2 SOAP

Než budou vydefinovány bližší podrobnosti týkající se současně používaných přístupů při tvorbě webových API rozhraní, kterými jsou REST a GraphQL, je zapotřebí představit jejich předchůdce, který položil základní stavební kameny týkající se komunikace pomocí webových služeb. (6)

SOAP je protokol, který umožňuje aplikacím komunikovat pomocí výměny XML zpráv. Webové služby používají SOAP protokol jako základní technologii pro komunikaci. Společně se SOAP existují také další standardy, jako je WSDL a UDDI, které dále rozšiřují a usnadňují použití SOAPu v rámci webového prostředí. SOAP funguje na principu peer-to-peer, což znamená, že dvě aplikace mohou vzájemně komunikovat posíláním XML zpráv napřímo mezi sebou. Tato komunikace je vždy jednosměrná. (3)

3.3 REST API

3.3.1 Historie REST API

Ačkoliv protokol SOAP poskytl standard pro přenos dat v rámci webových služeb, jde o protokol, který je poměrně komplexní a kvůli tomu ne zcela přívětivý vůči vývojářům moderních webových API. Toto vše se změnilo v roce 2000 s příchodem REST API (nebo také RESTful API). Tento významný krok na poli vývoje software v podobě uvedení pojmu REST učinil výzkumník z Kalifornské univerzity Roy Thomas Fielding. Ten ve své disertační práci, která byla zveřejněna právě koncem roku 2000 poprvé definuje pojem

REST API. Podle Fieldinga jde o „*koordinovaný soubor architektonických omezení, který se snaží minimalizovat latenci a síťovou komunikaci a zároveň maximalizovat nezávislost a škálovatelnost implementací komponent*.“ (8)

Jelikož je REST architektura stále tou jednoznačně nejrozšířenější mezi ostatními přístupy při tvorbě webových API, kdy i po 23 letech od vzniku tento přístup využívá 89 % všech webových aplikací ve svém API, lze moment jejího vzniku označit za malou revoluci a změnu paradigma v oblasti vývoje aplikací a konkrétně aplikací poskytující webové API. (9)

3.3.2 Komunikace přes REST API

Komunikace přes REST API rozhraní probíhá mezi klientem, který vysílá požadavky (requesty) a serverem, který požadavky zpracovává a zasílá odpovědi (response). Tato komunikace probíhá přes HTTP protokol. V návrhu REST iniciuje interakci klient – běžným příkladem je přístup k webové stránce prostřednictvím adresy URL realizovaný HTTP požadavkem. (10)

Klient je program nebo osoba, která má přístup k API a vysílá skrz něj požadavky. Tyto požadavky mají podobu HTTP metod. (10) Nejpoužívanějšími metodami jsou:

- GET: Pro získání zdroje a dat o něm
- POST: Pro vytvoření nového zdroje
- PUT: Pro aktualizace dat již existujícího zdroje
- DELETE: Pro výmaz zdroje

Zdroj označuje konkrétní data, které rozhraní API zpřístupňuje klientovi. Pod zdrojem si lze představit cokoliv od profilu uživatele, přes komentáře, které uživatel zveřejnil na sociální síti až například po inzerát na online tržišti. Společnou vlastností všech zdrojů je, že mají jedinečný název nazývaný identifikátor zdroje a že jsou všechny zdroje uloženy na serveru. (6) (10)

Ukázkový HTTP GET požadavek může vypadat například následovně:

https://api.domain.com/authors/7 (11)

Když klient zadá požadavek pomocí rozhraní REST API, server přenese standardizovanou reprezentaci stavu zdroje do systému klienta. To znamená, že server

neposílá klientovi skutečný zdroj, ale pouze reprezentaci jeho stavu v určitém časovém okamžiku. (6) (10)

Odpovědi, stejně jako požadavky, jsou realizovávány prostřednictvím různých datových formátů souborů, kdy mezi v současnosti nejpoužívanější patří JSON (JavaScript Object Notation). JSON je snadno čitelný pro lidi i programy, díky čemuž je práce s tímto formátem jednodušší. Je také kompatibilní s mnoha dalšími programovacími jazyky, což usnadňuje interpretaci. Dalšími datovými formáty, v současnosti již ne tak hojně využívanými při práci s REST API, jsou XML, YAML, CSV, HTML a prostý text. (10)

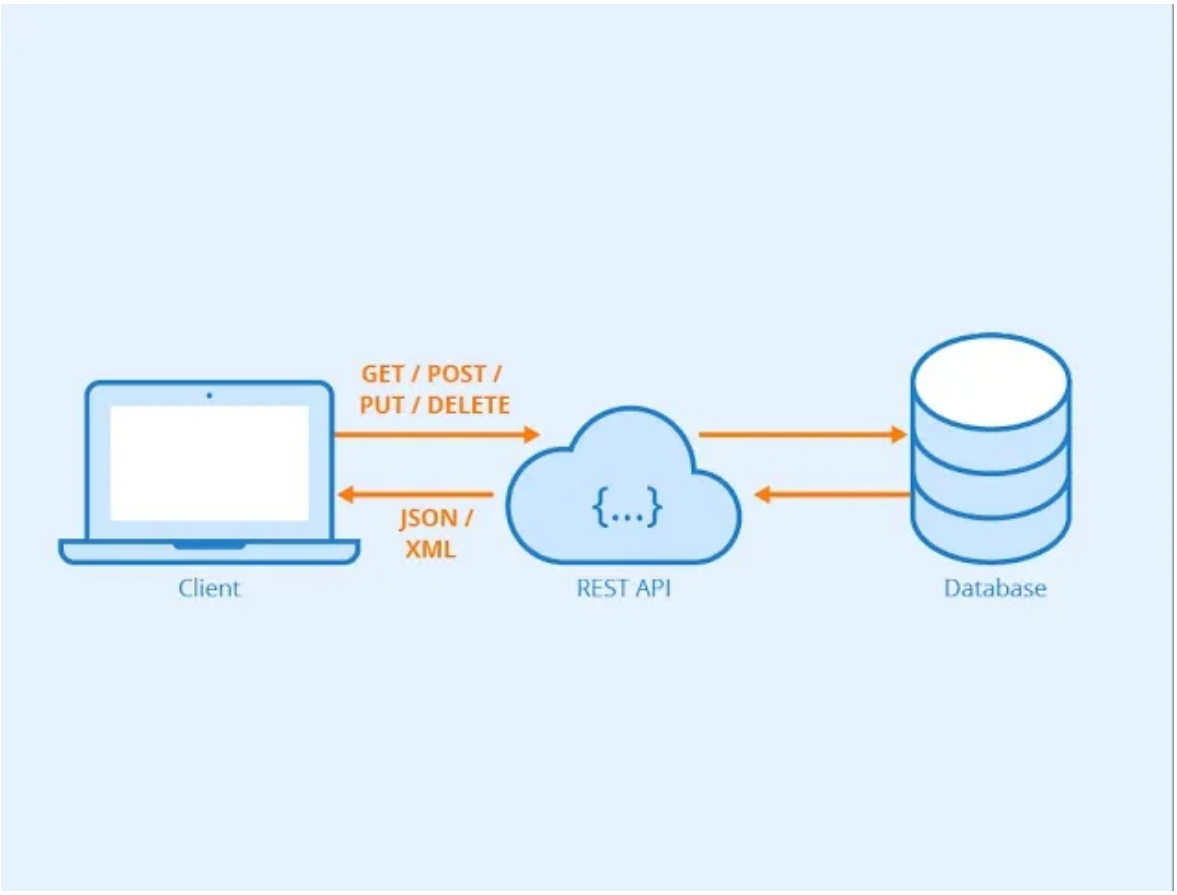
Příklad odpovědi ze serveru v podobě JSON souboru na HTTP GET požadavek uvedený výše je uveden na ukázce kódu číslo 1.

```
{
  "id": "7",
  "name": "Robin Wieruch",
  "avatarUrl": "https://domain.com/authors/7",
  "firstName": "Robin",
  "lastName": "Wieruch"
}
```

Ukázka kódu 1 Příklad odpovědi ze serveru v JSON formátu

Zdroj: (11)

Ukázka komunikace přes REST API je znázorněna na obrázku číslo 1.



Obrázek 1 Schéma architektury REST API

Zdroj: (12)

3.3.3 Základní principy REST API

Fielding ve své disertační práci definoval 6 principů (v původním anglickém znění constraints), které musí služba splňovat, aby mohla být označena za RESTful službu. (8)

Architektura klient-server

Princip pro oddělení strany, na které klient vysílá požadavky, a strany na které server požadavky přijímá a reaguje na ně, je stěžejním prvkem REST architektury. Díky tomuto principu je docíleno nezávislosti mezi klientem a serverem, což umožňuje nezávislý rozvoj obou komponent (např. škálování) bez potřeby měnit komponentu druhou. (8) (10) (13)

Jednotné rozhraní

Princip jednotného rozhraní definuje, že požadavky na přístup k jednomu zdroji na serveru by vždy měly vypadat jednotně, bez rozdílu, od jakého klienta požadavek přichází.

Zároveň zdroj, umístěný na serveru, by měl být vždy reprezentován unikátním identifikátorem URI (Uniform Resource Identifier) a měl by obsahovat všechny informace o zdroji, které by klient mohl vyžadovat. (8) (10) (13)

Bezstavová komunikace

Princip bezstavové komunikace stanovuje, že by se na straně serveru neměla ukládat žádná data potřebná ke zpracování konkrétního požadavku (např. v podobě sessions). Naopak požadavek od klienta by měl obsahovat všechny informace potřebné ke zprocesování požadavku na straně serveru. (8) (10) (13)

Mezipamět (cache)

Za účelem optimalizace výkonu a minimalizace přenášených dat REST prosazuje principy cachování dat ať už na straně klienta nebo na straně serveru. Server by měl definovat, zda pro daný zdroj je cachování povoleno, či nikoliv. (8) (10) (13)

Rozdělení do vrstev

Díky principu rozdělení do vrstev je potřeba počítat s tím, že v komunikaci mezi klientem a serverem může figurovat prostředník (middleware) a tedy že klient a server nemusí nutně komunikovat napřímo. Tento princip aplikacím umožňuje lépe řídit vyvažování zátěže (tzv. load balancing), umožňuje implementovat bezpečnostní prvky přes oddělené vrstvy aplikace a umožňuje snazší škálovatelnost, která je ve světě moderních internetových služeb tak často potřeba. (8) (10) (13)

Code-on-demand

Jediný nepovinný princip, který lze využít při stavbě REST API rozhraní se nazývá code-on-demand. Jeho vynechání nezpůsobí, že by API dále nešlo označit za RESTful. S tímto principem může klient stahovat a spouštět kód poskytnutý serverem (například applety Java, skripty JavaScript atd.). (8) (10) (13)

3.3.4 Výhody REST API

Ačkoliv od zveřejnění návrhu pro REST API uběhlo více než 20 let, stále tento architektonický přístup nabízí škálu benefitů, díky kterým je jedničkou, pokud jde o jeho využívání při budování nových API.

První z výhod tohoto přístupu je interoperabilita a flexibilita, které se vyznačují tím, že přístup REST API je zcela nezávislý na programovacím jazyku, platformě nebo formátu soubor pro přenos dat. To z REST API dělá univerzální nástroj pro komunikaci mezi

aplikacemi/systémy, které mohou být napsány s využitím zcela odlišných technologií nebo programovacích jazyků. (14)

Mezi další výhody se řadí možnost snadné škálovatelnosti, a to díky rozdělení komunikace mezi server a klienta, která umožňuje více týmům pracovat na různých částech aplikace. (14) (15)

Podstatnou částí REST API je možnost klientského přístupu k serverem cachovaným datům. To je umožněno tím, že REST pracuje na principu zabudované mezipaměti, díky které není potřeba realizovat časté požadavky na server opakovaně. V momentě, kdy klient zašle požadavek na server, dojde prvně ke kontrole dat v mezipaměti. Pokud vyžadovaná data v mezipaměti existují, není potřeba dokončovat zaslání požadavku až k samotnému serveru pro vykonání operace a navrácení odpovědi. Ukládání dat v paměti cache tedy funguje na principu mezivrstvy, kdy jsou data k častým dotazům ukládána za účelem jejich opakovaného využití bez potřeby čerpat výpočetní výkon na straně serveru. (14) (15)

3.3.5 Nevýhody REST API

Mezi nejvýznamnější nedostatky REST přístupu se řadí tzv. overfetching, či v některých případech také underfetching dat. Overfetching nastává, jakmile klient v odpovědi od serveru dostává více dat, než kolik potřebuje. Server, kvůli tomu, že má předdefinovanou množinu dat pro odpověď v každém endpointu, vrací stálou množinu parametrů, kdy část z těchto dat nemusí být pro klienta relevantní. Naopak underfetching představuje situaci, kdy data poskytnuta v rámci jednoho endpointu nejsou pro klienta dostatečná a klient je tedy nucen namísto jednoho endopointu dotázat další endpoint (případně více endpointů naráz). Toto může vyústit v obecně pomalejší odpovědi ze strany serveru. (15)

Jako další nevýhodu REST přístupu lze označit verzování API. Kvůli zpětné kompatibilitě je možné spravovat více verzí stejného API současně. Tato praktika se uplatňuje hlavně za účelem mitigace problémů při komunikaci mezi serverem a klientem, kdy API projde takovými změnami, které nejsou zpětně kompatibilní a ohrožují tedy celkovou funkčnost. S více verzemi spravovanými naráz nicméně dochází k problémům s udržitelností a správou. (15)

Závěrem lze jako poslední nevýhodu API přístup uvést absenci stavů, kvůli které jsou kladeny větší nároky na klienta, pokud jde o aplikační logiku. Jako příklad lze uvést

nákup v internetovém obchodě, kdy v tomto konkrétním případě je na straně klienta, aby znal a ukládal počet položek v košíku předtím, než dojde ke zpracování nákupu na straně serveru. (14)

3.4 GraphQL

3.4.1 Historie GraphQL

GraphQL bylo interně vyvinuto společností Facebook v roce 2012 jako pokus o optimalizaci funkčnosti mobilní aplikace, která v této době byla spíše jen adaptací jejich webových stránek. Při tvorbě nového API rozhraní pro news feed Facebooku se začalo poprvé formovat rozhraní připomínající dnešní GraphQL. V roce 2015 byla poprvé zveřejněna specifikace GraphQL v podobě open source spolu s referenční implementací v jazyce JavaScript. (16)

Právě po přechodu na otevřený zdrojový kód byl projekt GraphQL v roce 2018 přesunut ze společnosti Facebook do nově založené GraphQL Foundation, kterou spravuje nadace Linux Foundation. (16) (17)

3.4.2 Komunikace přes GraphQL

GraphQL je dotazovací jazyk, který na rozdíl od většiny alternativních přístupů pro tvorbu API umožňuje zadávat požadavky na konkrétní data, čímž klientům poskytuje větší kontrolu nad specifičností a množstvím dat, které od serveru vyžádají a které jsou jim také vráceny ve formě odpovědi. (17)

Díky tomu je GraphQL na rozdíl od odlišných API přístupů imunní vůči problému nazývanému overfetching. Ten nastává v případech, kdy server je entita, definující, jaká množina dat je k dispozici pro každý zdroj dostupný na konkrétní URL. Klient tedy vždy vyžaduje všechny informace ke zdroji, které jsou vydefinováné a dostane je v odpovědi od serveru, a to i v případech, že některá data, která server ke zdroji poskytuje, jsou pro klienta nepotřebná.

Komunikace na bázi GraphQL není vázána na konkrétní protokol, tudíž vedle standardního HTTP lze využít i jeho alternativ jako je komunikace přímo přes TCP nebo web sockety. Stejně tak je GraphQL nezávislé od návrhového vzoru databáze, tudíž

GraphQL server dokáže pracovat jak s databází na bázi SQL, tak alternativami v podobě NoSQL databází. (17) (18)

GraphQL na straně serveru definuje GraphQL schéma, v rámci kterého je specifikováno, jaká data a v jaké struktuře jsou klientovi zpřístupněna. Klient je v tomto případě entita, která za pomoci dotazů přistupuje pouze k těm datům, která skutečně potřebuje. (17)

Podobně jako REST definuje HTTP metody, GraphQL rozlišuje několik typů dotazů, které zajišťují operaci s daty:

- Query: Pro čtení
- Mutation: Pro zápis a úpravy existujících dat
- Subscription: Pro průběžné čtení

Každá z těchto operací je realizována jako řetězec znaků. (17)

Příklad dotazu ze strany klienta je k dispozici na ukázce kódu číslo 2.

```
getBooks {
        books {
            id
            name
            pages
            author
          }
    }
```

Ukázka kódu 2 Příklad GraphQL dotazu ze strany klienta

Zdroj: (11)

Jakmile je GraphQL dotaz doručen na stranu serveru, ten dotaz interpretuje vůči svému GraphQL schématu. Tato činnost se nazývá (vy)řešení dotazu (z anglického resolving), realizuje ji tzv. resolver definovaný v GraphQL schéma. Po vyřešení dotazu vrací server klientovi data, která prostřednictvím dotazu zažádal, případně změnil. (17) (18)

Příklad odpovědi ze strany serveru na dotaz uvedený výše je k dispozici k náhledu na ukázce kódu číslo 3.

```
{
  "data": {
  "author": {
  "id": "7",
  "name": "Robin Wieruch",
  "avatarUrl": "https://domain.com/authors/7",
  "articles": [
```

```
{
"name": "The Road to learn React",
"urlSlug": "the-road-to-learn-react"
},
{
"name": "React Testing Tutorial",
"urlSlug": "react-testing-tutorial"
}
] }
} }
```

Ukázka kódu 3 Příklad odpovědi ze strany serveru na GraphQL dotaz v podobě JSON formátu

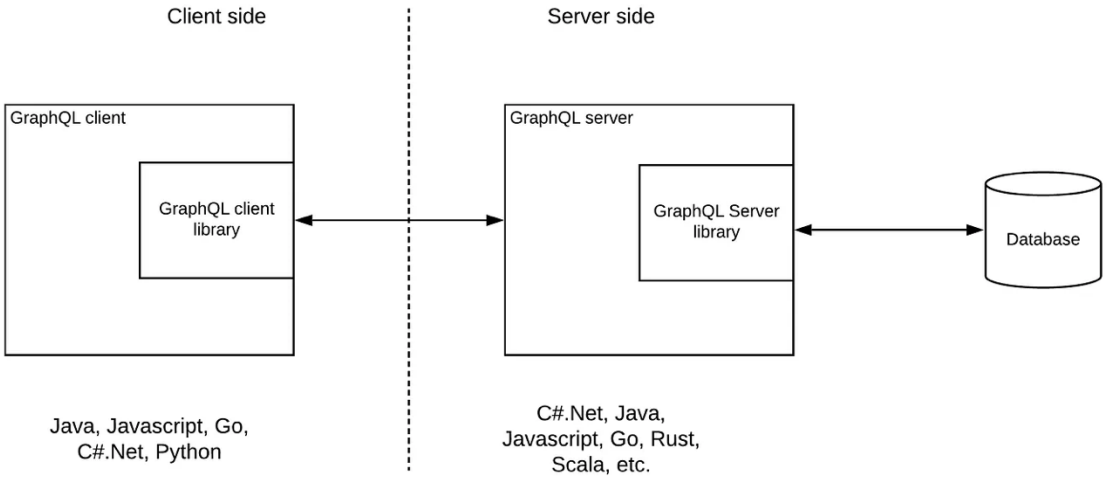
Zdroj: (11)

GraphQL rozlišuje několik druhů architektur podle toho, jakou roli v systému zaujímá GraphQL server. Ačkoliv je díky neustálému pokroku k dispozici stále rostoucí množství návrhových vzorů, základní členění lze rozdělit do 4 návrhů. (19)

3.4.2.1 GraphQL server s napojenou databází

Nejjednodušším návrhovým vzorem typickým pro nově vznikající projekty je GraphQL server existující se svou vlastní databází, kdy server nekomunikuje s dalšími službami. Výhodou tohoto přístupu je nízká komplexita, pokud jde o získávání dat. Taktéž je zdroj dat blíže k aplikaci, která pokrývá aplikační logiku, čímž je zapříčiněna minimalizace zpoždění při přenosu dat. Nevýhodou tohoto přístupu jsou jeho neexistující možnosti integrace na další služby, čímž je jeho implementace limitována pouze na jednodušší funkční celky. V případě implementace GraphQL do systému využívající více souběžně běžících služeb, je potřeba zvolit alternativní návrhový vzor. (18) (19) (20)

Schéma GraphQL serveru s přímo napojenou databází je znázorněno na obrázku číslo 2.



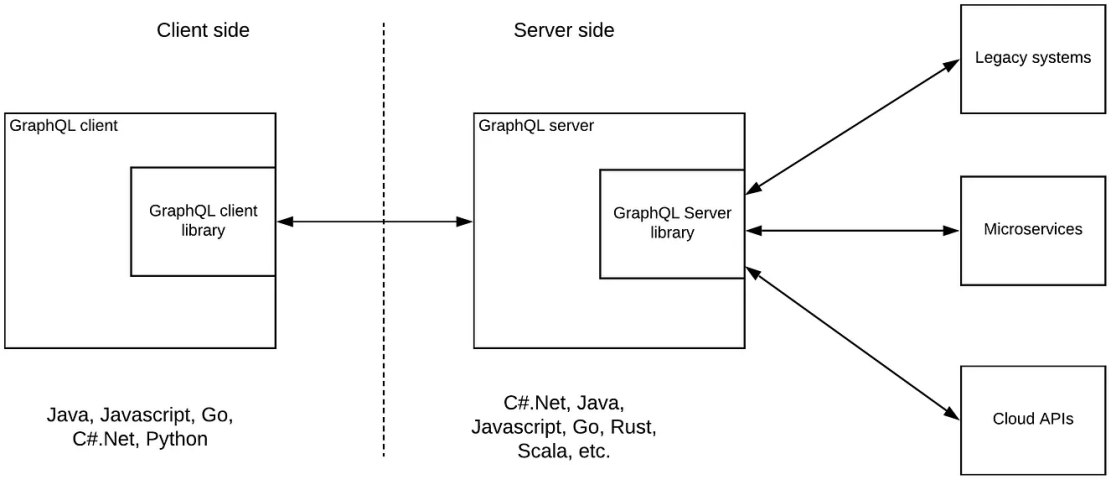
Obrázek 2 GraphQL server s napojenou databází

Zdroj: (19)

3.4.2.2 GraphQL server jako vrstva integrující služby třetích stran

Alternativou k původnímu řešení obsahující pouze samotný GraphQL server a žádnou další integraci je návrhový vzor definující GraphQL jako samostatnou vrstvu realizovanou v podobě serveru, který nedisponuje vlastní databází, nýbrž data výlučně získává v rámci komunikace se službami třetích stran (aplikací a programů). Tento přístup, kdy server přebírá roli middleware systému, umožňuje ve větší míře těžit z výhod GraphQL, jelikož klient, namísto toho, aby pro získání dat dotazoval separátně například 3 různé služby, zašle pouze jeden GraphQL dotaz serveru, který v sobě shromažďuje integrace na koncové backend systémy a tudíž poskytuje unifikované rozhraní pro klienta. (18) (19)

Schéma GraphQL serveru jako vrstvy integrující služby třetích stran je k dispozici obrázku číslo 3.



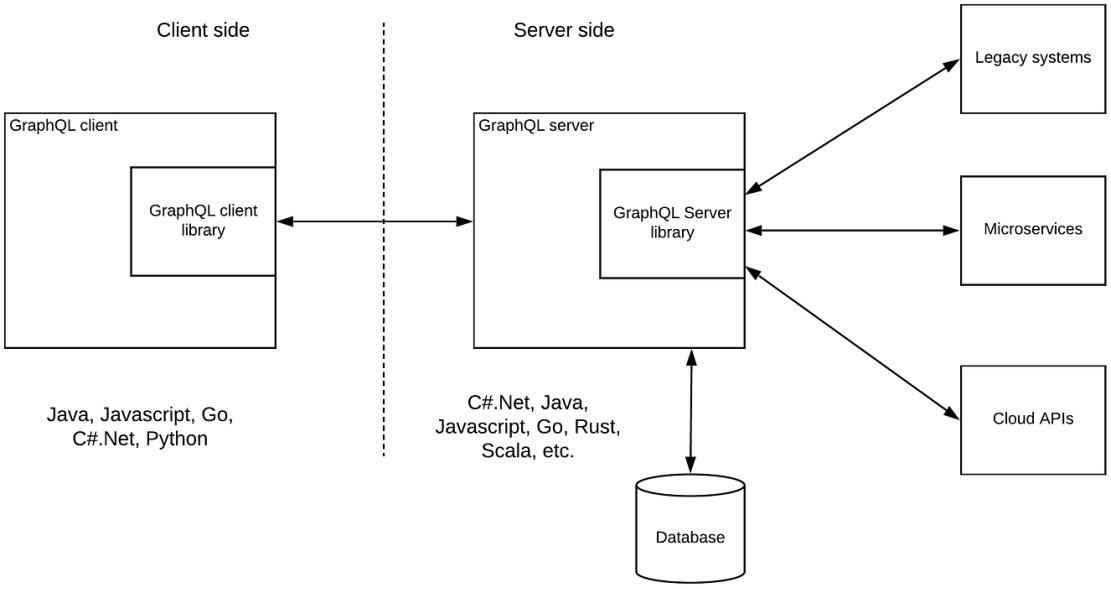
Obrázek 3 GraphQL server jako vrstva integrující služby třetích stran

Zdroj: (19)

3.4.2.3 Hybridní integrace

Třetí variantou návrhového vzoru je hybridní přístup kombinující architektonické řešení dvou předešlých ukázek, čímž lze čerpat výhod obou z nich. Jakmile server obdrží dotaz od klienta, k jeho vyřešení využije buď data ve své připojené databázi, nebo se doptá integrovaných služeb. (18) (19)

Schéma hybridní integrace GraphQL je k náhledu na obrázku číslo 4.



Obrázek 4 Hybridní integrace

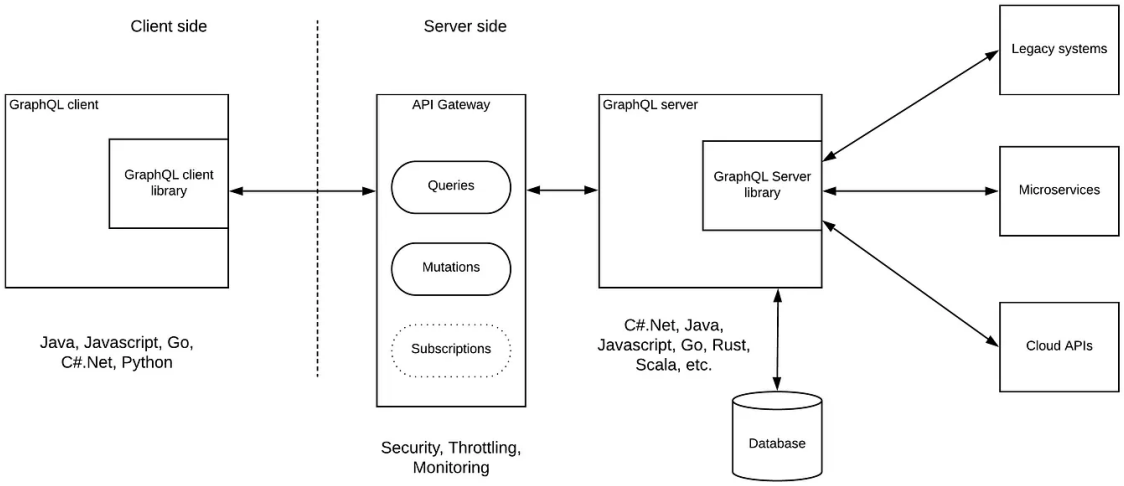
Zdroj: (19)

3.4.2.4 Hybridní integrace využívající API gateway

Vystavování samotných služeb (vč. GraphQL serveru) napřímo pro konzumaci v posledních letech nahrazuje přístup využívající tzv. API gateway. Jinými slovy API brána má za cíl poskytnout způsob zajištění správy API hlavně co se týče omezení množství požadavků putujících na server (throttling), bezpečnosti a monitoringu. (19)

V případě GraphQL implementace je API gateway umístěna mezi klienta a server a stejně jako v případě přímé komunikace mezi klientem a serverem se řídí dle serverem definovaných pravidel pro získávání dat ze schématu. (19)

Schéma hybridní integrace využívající API gateway je k dispozici na obrázku číslo 5.



Obrázek 5 Hybridní integrace využívající API gateway

Zdroj: (19)

3.4.3 Základní principy GraphQL

Hierarchie

Aby mohlo být GraphQL využíváno širokou škálou klientů, je zapotřebí, aby byl splněn požadavek na hierarchické členění struktury dat, na jehož základě většina dnešních klientů pracuje. GraphQL tohoto dociluje hierarchickým strukturováním dotazů, kdy dotaz má stejnou strukturu jako data, která vrací. (21)

Orientace na produkt

GraphQL by jakožto specifikace dotazovacího jazyka měla splňovat požadavky na poskytnutí dat a na jejich zobrazení front-end aplikacím a s tímto cílem by měl být graf dat realizován. (21)

Silně typované

Každá jednotlivá GraphQL služba definuje systém možných typů pro danou aplikaci. Požadavky jsou následně prováděny v kontextu tohoto typového systému. To znamená, že při obdržení dotazu může GraphQL aplikace za pomocí dalších nástrojů určit, zda je dotaz syntakticky správný a platný v rámci definovaného typového souboru a může poskytnout určité záruky ohledně tvaru a povahy odpovědi. (21)

Klientem specifikované dotazy

GraphQL server definuje možnosti pro získání dat, které mohou klienti využívat. Dále za přesnou specifikaci způsobu, jakým budou tyto zveřejněné možnosti využívány

(v podobě specifikace konkrétních dotazů), je zodpovědný klient. Tyto dotazy jsou specifikovány s granularitou na úrovni polí. Ve většině aplikací typu klient-server nevyužívajících GraphQL (např. REST) je server ten, který napevno určuje podobu a strukturu vrácených dat, které jsou definovány v popisu koncových bodů (tzv. endpointů). Dotaz GraphQL naproti tomu vrací přesně to, o co klient požádá, a nic víc. (21)

Introspektiva

GraphQL je introspektivní, což znamená, že systém typů GraphQL serveru musí být dotazovatelný samotným jazykem GraphQL. Využití introspekce jazyka GraphQL umožňuje vytváření společných nástrojů a knihoven. (21)

3.4.4 Výhody GraphQL

GraphQL vzniklo jako alternativa k REST rozhraní, aby v určitých ohledech napravila jeho nedostatky. To je docíleno díky výhodám tohoto přístupu.

GraphQL se snaží řešit palčivý problém overfetchingu a underfetchingu, který s sebou už ze základních principů fungování přináší REST. Na rozdíl o REST rozhraní GraphQL zasílá na server samostatný dotaz, který přesně odpovídá požadavkům na data, které je potřeba v odpovědi obdržet. Server tedy odpovídá přesně tím množstvím dat, která byla vyžádána ze strany klienta a neposílá ani víc, ani míň. Díky tomuto benefitu je docíleno nižších nároků na síťový přenos a paměť na straně klienta. (15) Této výhody může být docíleno díky využití principu tzv. deklarativního získávání dat. (17)

Ačkoliv reference od Facebooku byla napsána v Javascriptu, který umožňuje možnost využití v současné době velmi populárních knihoven jako je Angular, Vue nebo Express, dokáže GraphQL pracovat stejně jako REST zcela nezávisle na platformě a programovacím jazyku, ať už jde o stranu klienta nebo serveru. (17)

Dalším principem představující benefit ve využívání GraphQL je tzv. spojování schémat. Tento princip umožňuje spojit více samostatných GraphQL schémat a vytvořit tak jedno schéma, které je vystaveno pro komunikaci s klientem. Tento přístup lze aplikovat v prostředí s architekturou spočívající na mikroslužbách, kdy každá mikroslužba realizuje omezenou část logiky aplikace (tzv. svou doménu). Každá mikroslužba je reprezentována jedním GraphQL schématem a může mít svůj vlastní GraphQL endpoint. Následně je možné samostatná GraphQL schémata spojit a klientovi všechny endpointy zpřístupnit v rámci

GraphQL API gateway. Tímto přístupem je zajištěna flexibilita řešení mikroslužeb a s tím spojena snazší škálovatelnost celého řešení. (17)

Poslední výhodou, nicméně neméně důležitou, je způsob verzování GraphQL oproti REST. Zatímco REST využívá verzování svého API (např. api.domain.com/v1/, api.domain.com/v2/), aby předcházel problémům s kompatibilitou v případě změny struktury dat u zdroje, GraphQL princip podobného verzování vůbec nepotřebuje. Namísto toho GraphQL poskytuje možnost učinit některé z GraphQL polí zastaralými. Jakmile klient dostane odpověď, obdrží zároveň s tím informaci o tom, že se jedná o zastaralé pole. Zastarání může být v požadavcích vraceno po nezbytně dlouhou dobu, dokud většina klientů konzumující dané schéma nepřejde na aktuální zdroj dat. Poté může dojít k vyřazení a tím je docíleno možnosti postrádat verzování. (17)

3.4.5 Nevýhody GraphQL

Ačkoliv GraphQL přináší velké množství výhod oproti předchozím přístupům pro vývoj API, přináší i nevýhody či výzvy, s kterými musí vývojáři a společnosti akceptující tento přístup počítat.

Mezi nejvýznamnější překážku při implementaci a provozu řešení využívajícího GraphQL lze zařadit komplexitu dotazů, které klienti zasílají směrem k serveru a ten kvůli zvýšené komplexitě provádí výpočetně náročné operace nad vlastní databází. Pokud například požadavek od klienta žádá data o více samostatných entitách, a navíc se jedná o silně větvený požadavek získávající data v rámci několika úrovní, jedná se o náročnou operaci. Jako obranu na přetěžování serveru lze implementovat např. maximální hloubku GraphQL dotazu co se do úrovně větvení týče, vážení složitosti dotazu, zabránění rekurzi anebo využití trvalých dotazů (tzv. persited queries) za účelem zastavení neefektivních požadavků přicházejících na server. (17)

V případě využívání GraphQL je složitější oproti REST alternativně definovat maximální využití API. Takováto omezení se běžně aplikují v případě vystavení veřejných API a REST přístup to snadněji docílí např. přes vydefinování maximálního počtu požadavku za den. Tento přístup má u REST rozhraní své opodstatnění, protože každý požadavek na konkrétní endpoint vrátí vždy stejnou odpověď. To ale není případ GraphQL, kdy každý dotaz a odpověď na dotaz představuje jinou strukturu a množství dat v závislosti

na povaze GraphQL dotazu, který na server dorazí. Jako mitigaci tohoto problému hodně společností vystavující veřejné GraphQL API volí cestu právě opatření typu vážené složitosti dotazu. (17)

Podobně je tomu i při definování mezipaměti a dat, které budou klientům z cache přístupná. REST, díky práci se zdroji definovanými jednoznačnými unikátními URI, může cachovat veškerá data o zdroji k danému okamžiku. U GraphQL tento zjednodušený přístup nelze použít, protože na rozdíl o REST zde každý dotaz vypadá jinak, i když operuje nad jednou a tatáž entitou. To vyžaduje více sofistikovaný přístup ke cachování dat až na úroveň polí, který je nicméně náročnější na implementaci ve srovnání se zmíněnou REST alternativou. (17)

3.5 Technologie pro srovnávací experiment

Pro uskutečnění srovnávacích pokusů je zapotřebí připravit prostředí, v rámci něhož budou pokusy uskutečněny. V této sekci bude blíže představeno, jakých nástrojů a technologií bude využito k jeho tvorbě a jakému slouží účelu.

Prostředí pro uskutečnění srovnání bude představovat Node.js aplikace, využívající Express.js jako framework procesující požadavky a odpovědi a správu routování, Apollo Serveru a Apollo Clienta pro správu GraphQL operací a MongoDB databáze pro ukládání dat. Frontend část aplikace bude vyhotovena s využitím knihovny React.

3.5.1 Node.js

Node.js je open-source Javascript běhové prostředí fungující na straně serveru a knihovna pro běh aplikací mimo prohlížeč. Díky tomu je Node.js široce využíván k vytváření webových aplikací na straně serveru a je vhodným řešením pro vývoj datově náročných aplikací, protože používá asynchronní a událostmi řízený model operace s daty. (13) (22)

Kromě asynchronního programování Node.js nabízí další výhody, za jednu z nich lze označit jednoduchost. Díky spojení s Javascriptem je Node.js jako framework velice vhodný pro tvorbu aplikací i pro ne tolik zkušené programátory, avšak i tak je velice vhodným frameworkem pro budování komplexních řešení. Dalšími, avšak neméně důležitými výhodami, je využití modulárnosti při vývoji řešení s využitím node package manager (npm), který vývojářům umožňuje přepoužívat již existující moduly (např. knihovny), tudíž

vývojáři šetří čas a nemusí vyvíjet všechny komponenty aplikace od samého začátku. S tím souvisí i široká komunita využívající Node.js, která se neustále rozvíjí. (13)

3.5.2 Express.js

Express.js je webový Javascript framework využívaný k vývoji aplikací v rámci Node.js prostředí. Je využívaný pro snadnou práci a vývoj API a webových aplikací. (23) Hlavní funkce, které Express.js v rámci Node.js projektů zastává, jsou (23):

- Middleware – obhospodařuje požadavky a přistupuje k aplikačnímu request-response cyklu
- Routing – definuje, jak endpointy aplikace reagují na požadavky klientů na základě URL
- Šablony - poskytuje šablony pro vytváření dynamického obsahu webových stránek pomocí šablon HTML na serveru
- Debugging – pomáhá s identifikací chyb v kódu tím, že vývojáři přesně hlásí místo vzniku chyby

3.5.3 MongoDB

MongoDB je open-source NonSQL dokumentová databáze. Je postavena na horizontální architektuře, která využívá flexibilního schéma pro ukládání dat. Tento princip znamená, že databáze neukládá data v tabulkách o řádcích, reprezentující jednotlivé záznamy, a sloupcích, reprezentujících vlastnosti daných záznamů, nýbrž v dokumentu popsaným v tzv. BSON, tedy binary JSON. Aplikace z tohoto dokumentu získávají data ve formě JSON formátu. (24)

Dokumentové databáze jsou velice flexibilním nástrojem, především díky variabilitě ve struktuře dokumentů a ukládání dokumentů, které jsou nekompletní. V rámci MongoDB databáze je také možné vkládat dokumenty do sebe. (24)

Na rozdíl od tradičních SQL databází MongoDB vlastnosti dat neukládá do sloupců, ale polí v dokumentu. (24)

Níže je ukázka kódu číslo 4 popisující výskyt entity žák v MongoDB dokumentu:

```
{
  "_id": 1,
  "name": {
```

```
        "first": "Josef",
        "last": "Novák"
      },
      "fakulta": "PEF",
      "predmety": ["matematika", "programování"]
    }
```

Ukázka kódu 4 Ukázka zápisu výskytu třídy žák v MongoDB

Zdroj: autor

Právě kvůli odlišnému přístupu k ukládání dat a přístupu k nim mohou uživatele MongDB čerpat následujích výhod (24):

- BSON, resp. JSON formát představuje přirozený způsob ukládání dat, který je snadno čitelný i pro člověka.
- Jeden a ten samý dokument umožňuje ukládání jak strukturovaných, tak nestrukturovaných dat.
- JSON umožňuje snadné vnořování dat kvůli komplexním objektům.

3.5.4 MongoDB Atlas

MongoDB Atlas patří do skupiny online služeb nazývaných jako databáze jako služba (DBaaS - Database as a service). Jedná se o službu, která jejím uživatelům umožňuje nastavit, nasadit a dále škálovat databázové řešení bez nutnosti se starat o místní fyzický hardware, aktualizace souvisejícího softwaru a další nezbytné konfigurace pro zajištění chodu a dostatečného výkonu databáze. Zmíněná nastavení a konfiguraci databáze totiž zajišťuje samotný poskytovatel služby. (25)

3.5.5 MongoDB Compass

MongoDB Compass je výkonné grafické uživatelské rozhraní (GUI) pro dotazování, agregaci a analýzu dat uložených v MongoDB databázi. Svým uživatelům poskytuje prostředí, přes které mohou v jednoduchosti komunikovat s MongoDB databázi provozované např. v MongoDB Atlas. (26)

3.5.6 Apollo Server

Apollo Server je open-source server vyhovující specifikacím GraphQL, který je kompatibilní se všemi GraphQL klienty. Lze ho využít jako samostatný GraphQL server,

middleware systém nad běhovým prostředím Node.js nebo bránu pro pro přístup k schématům v rámci hlavního GraphQL grafu. (27)

3.5.7 Apollo Client

Apollo Client je Javascript knihovna pro řízení stavů, která umožňuje řízení jak lokálních, tak vzdálených dat pomocí GraphQL. Využívá se jak k získávání dat, jejich cachování v mezipaměti tak modifikaci (pomocí mutací). (28)

3.5.8 React.js

React.js je Javascript knihovna využívaná při vývoji uživatelského rozhraní (GUI - Graphical User Interface) webových aplikací. Každá webová aplikace využívající React obsahuje znovupoužitelné komponenty, z kterých se výsledné GUI skládá. Komponentami mohou být samostatné prvky, které v aplikaci tvoří menší celky jako například navigační panel nebo patičku stránky. (29)

Využití těchto opakovaně použitelných komponent usnadňuje a tím zrychluje vývoj grafického rozhraní, jelikož není potřeba přepisovat stále stejný kód na všech místech aplikace, kde se má komponenta vyskytovat. Namísto toho je vytvořena logika pro zobrazení a komponentu importujeme do libovolné části kódu, kde je potřeba. (29)

React je vhodný pro tvorbu tzv. jednostránkových aplikací (SPA – Single Page Application). Princip jednostránkových aplikací si lze představit tak, že namísto toho aby aplikace při každém vykreslení stránky posílala nový požadavek na serveru, načítá obsah stránky přímo z React komponent, což vede k rychlejšímu vykreslování obsahu bez nutnosti opětovného načítání stránky. Není tedy docíleno pouze rychlejšího načítání zlepšující uživatelskou zkušenost, ale také k úspoře množství přenášených dat mezi serverem a klientem. (29)

Syntaxi pro vytváření React aplikací React nazývá JSX (JavaScript XML). Jedná se syntaktické rozšíření jazyka JavaScript, které umožňuje jedinečným způsobem kombinovat Javascript logiku a logiku vykreslování uživatelského rozhraní. Taktéž s pomocí JSX odstraňujeme potřebu přímé interakce s DOM pomocí metod, jako je document.getElementById, querySelector a další. (29)

Příklad JSX je uveden na ukázce kódu číslo 5.

```
function App() {
```

```
        return (
            <h1>Hello World</h1>
        )
    }

export default App;
```

Ukázka kódu 5 Ukázka JSX

Zdroj: (30)

3.5.9 Visual studio code

Visual Studio Code (VS Code) je bezplatný, open-source integrovaný vývojový prostředek (IDE) vyvinutý společností Microsoft. Jeho flexibilita a široká podpora programovacích jazyků z něj činí oblíbený nástroj mezi vývojáři. VS Code je známý pro svou lehkost, rychlost a rozšiřitelnost, což umožňuje uživatelům snadno psát, ladit a spravovat kód s intuitivním uživatelským rozhraním.

V rámci praktické části bude využito VS Code pro vyhotovení prostředí v podobě webové aplikace schopné odbavit odesílání a přijímání požadavků mezi klientem a serverem.

3.6 Srovnávací experiment

3.6.1 Prostředí pro experiment

Za účelem realizace srovnávacího experimentu je zapotřebí připravit prostředí, v rámci kterého jednotlivé pokusy proběhnou. Tímto prostředím bude prostředí webové aplikace, které bude fungovat s oběma technologickými přístupy pro tvorbu webového API najednou. V rámci kontextu diplomové práce budou serverová i klientská část aplikace spuštěny lokálně. Tím bude zaručena minimalizace rizika ovlivnění výsledků testů z důvodu latence internetové sítě. Jelikož jsou server a klient v rámci počítače spouštěny jako dva samostatné procesy, nesdílejí stejný paměťový prostor, i když jsou spuštěny na stejném počítači. (14)

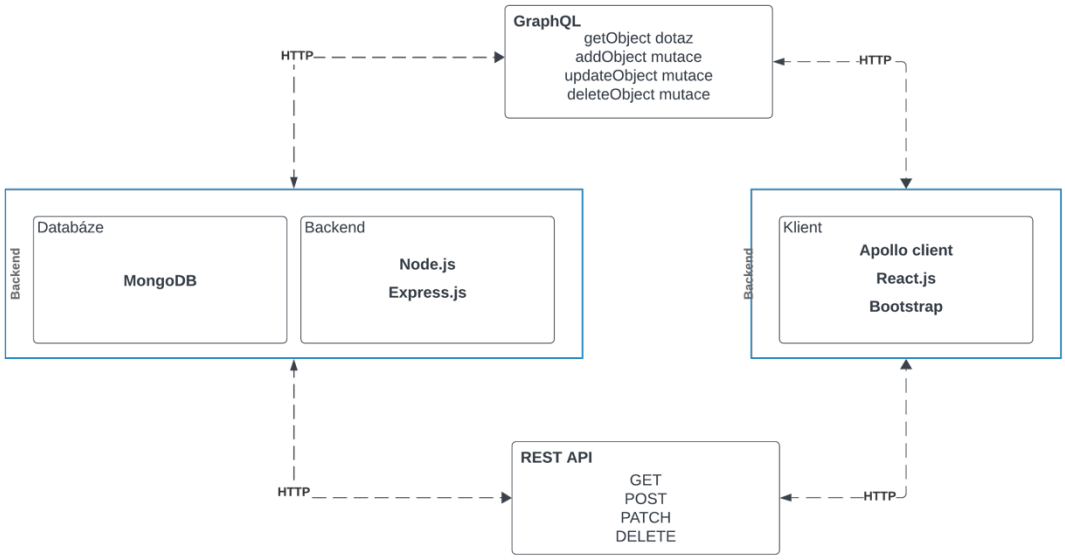
Všechny pokusy experimentu budou provedeny na počítači s následujícími parametry:

- Procesor: Apple M3 (8jádrový procesor se 4 výkonnými a 4 úspornými jádry)
- Operační paměť: 16 GB

- Operační systém: macOS Sonoma v14.1

Řešení webové aplikace bude využívat technologií popsaných v předchozích kapitolách. Serverová strana bude realizována v prostředí Node.js společně s NoSQL databází MongoDB. Serverová část aplikace, starající se o definování modelů, ukládání a operace s daty, bude komunikovat s klientskou částí přes API rozhraní realizované prostřednictvím operací (metod) v REST a také prostřednictvím dotazů a mutací v GraphQL. Pro běh REST API a GraphQL v prostředí Node.js bude využito webového frameworku Express.js. Klientská část aplikace bude využívat kombinaci React, HTML a CSS. Pro snazší vykreslování uživatelského rozhraní bude využito Bootstrap frameworku.

Návrh architektury webové aplikace je k dispozici na obrázku číslo 6.



Obrázek 6 Návrh architektury webové aplikace

Zdroj: autor

3.6.2 Sledované ukazatele

Před implementací samotné aplikace a vykonáním srovnávacích pokusů je potřeba určit základní ukazatele, podle kterých budou oba přístupy mezi sebou porovnávány.

Mezi vůbec nejsledovanější parametry pro určení kvality přenosu dat, je jeho rychlost. (31) Ta bude měřena jako celková doba vyřízení požadavků a bude vyjádřena rozdílem mezi časem zahájení požadavku (odesláním) a obdržením odpovědi. Doba vyřízení

požadavků bude vyjádřena s přesností milisekund. Milisekundy jsou standardní jednotkou pro vyjádření uplynulého času v prostředí implementace webových služeb a taktéž v rámci experimentu například již 100 ms může představovat znatelný rozdíl v celkovém času vyřízení všech požadavků. (32)

Druhou metrikou je velikost odpovědi. Sečtením velikosti bajtů každé odpovědi můžeme určit, který standard API je z hlediska přenosu dat tzv. nejlehčí (obsahuje nejmenší množství dat). Z důvodu omezení prohlížeče nemá kód přístup k hodnotě velikosti dotazu a jeho hlavičce, to znamená, že nástroj je schopen vypočítat pouze velikost těla. (33)

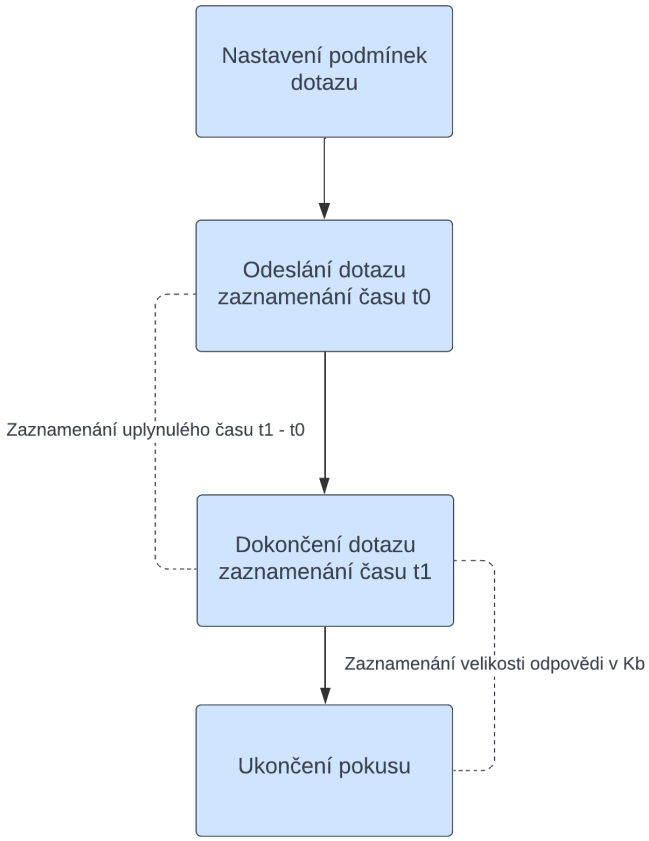
Základní sledované metriky tedy lze shrnout následovně:

1. **Doba provedení** označuje dobu, za kterou standard provede všechny požadavky od zahájení prvního do ukončení posledního. Doba provedení operace bude měřena s využitím Javascript metody performance.now(), která vrací časové razítko na úrovni milisekund. Tato metoda bude volána bezprostředně před voláním metody, která ve svém těle obsahuje kód pro realizaci operace (načtení, uložení, aktualizaci nebo výmaz dat). Navrácená hodnota bude uložena do lokální proměnné *t0*. Metoda performance.now() bude následně volána bezprostředně po opuštění metody realizující operaci s daty a navrácená hodnota bude uložena do lokální proměnné *t1*. Rozdíl hodnot *t1* a *t0* představuje dobu provedení požadavku nebo požadavků, pokud se jedná o pokus realizovanými s více opakováními. (34)

2. **Velikost těla odpovědi** označuje velikost odpovědi ze strany serveru vyjádřenou v kilobajtech. Měření bude probíhat s využitím hlavičky Content-Length v hlavičce HTTP požadavku realizované operace.

3.6.3 Proces porovnání

Za účelem stanovení přesné metodiky měření sledovaných ukazatelů, tj. jak doby provedení, tak velikosti těla požadavku, je zapotřebí stanovit přesný algoritmus, podle kterého vyhotovení měření proběhne. Za tímto účelem byl vyhotoven diagram popisující sled kroků, kterým budou podrobeny všechny dílčí pokusy. Algoritmus pokusu je znázorněn na obrázku číslo 7. Stejný způsob měření bude použit na všechny operace REST API realizovaných přes HTTP metody (GET, POST, PATCH, DELETE), stejně jako na GraphQL dotazy a mutace realizující ekvivalentní akce pro získání, tvorbu, aktualizaci a smazání dat na straně MongoDB serveru.



Obrázek 7 Návrh algoritmu pro experiment

Zdroj: autor

3.6.4 Porovnávané operace

Za účelem poskytnutí uceleného výstupu týkajícího se přenosu rychlosti a velikosti odpovědi bude analyzováno více typů operací. V případě REST API půjde o následující HTTP metody:

- GET – získání dat o zdrojích
- POST – vytvoření nových dat
- PATCH – aktualizace stávajících dat, pouze v atributech podléhajících změně
- DELETE – smazání dat

Alternativně v případě GraphQL půjde o dotazy a mutace, jejichž názvy budou definovány následovně a kde výraz Object zastupuje výskyt třídy definovaný v praktické části práce:

- getObject
- addObject
- updateObject
- deleteObject

Způsob měření doby vyřízení požadavku realizovaný přes HTTP metody a GraphQL operace výše definovaný v přechozí kapitole bude realizován umístěním bodu *t0* bezprostředně před volání metody/operace a současným umístěním bodu *t1* po obdržení potvrzení o dokončení požadavku. Rozdíl mezi *t1* a *t0* je výslednou hodnotou doby vyřízení požadavku, např. tedy získání dat pomocí metody GET a mutace getObjetct ze serveru.

V bodě *t1* bude také zaznamenána velikost odpovědi ze strany serveru.

3.6.5 Metodika vyhodnocení uživatelské přívětivosti

Součástí srovnávací analýzy je kromě výkonnostních parametrů (rychlost vyřízení požadavku a velikost odpovědi) také vyhodnocení uživatelské přívětivosti práce s každým z technologických přístupů pro tvorbu rozhraní. Jelikož je princip uživatelské přívětivosti do značné míry definován subjektivními dojmy z práce s jednotlivými způsoby tvorby aplikačního rozhraní, bude definována metodika, která vyhodnocení přívětivosti dodá objektivní ukazatele, čímž pomůže nezávisle vyhodnotit přívětivost jednotlivých postupů.

Přívětivost práce s každým z technologických postupů bude definována pomocí následujících ukazatelů:

1. **Doba vyhotovení operace** (REST API požadavku a GraphQL dotazu/mutace). Doba vyhotovení operace označuje čistý čas, který byl vykázán na psaní metody pro získání dat s využitím REST API endpointu či GraphQL dotazu či mutace. Naopak se do doby vyhotovení operace nezapočítává čas vynaložený na úkony spojené s tvorbou uživatelského rozhraní s využitím JSX, ze kterého je samotná metoda pro operaci s daty volána. Taktéž doba vyhotovení nepokrývá čas potřebný pro zanesení nutných mechanismů pro vyhodnocení kvantitativního srovnání, tj. měření rychlosti vyřízení požadavku s využitím metody performance.now() a velikosti odpovědi ze strany serveru.
2. **Počet napsaných znaků** pro vyhotovení operace (REST API požadavku a GraphQL dotazu/mutace). Jedná se o počet znaků, který je nezbytně nutný pro odbavení metody realizující operaci nad daty. Nepokrývá znaky pro měření výkonnosti realizace požadavku přes performance.now() metodu.
3. Srozumitelnost syntaxe kódu obou z přístupů – autorovo vyhodnocení pochopitelnosti a logické návaznosti kódu pro oba zmíněné přístupy a jejich metody, respektive dotazy a mutace.

Uživatelská přívětivost bude vyhodnocena přes skóre uživatelské přívětivosti. Body budou přiřazovány podle tabulky č. 1.

V rámci každého sledovaného ukazatele (průměrná doba vyhotovení operace, počet napsaných znaků pro vyhotovení operace, srozumitelnost syntaxe kódu) bude určena výsledná hodnota a následně každé hodnotě přiřazen bodový výsledek. Skóre přívětivosti je definováno jako průměr součtu přiřazených bodů každé kategorie děleno počtem kategorií. kdy čím více bodů v průměru přístup nasbírá, tím je hodnocen jako uživatelsky přívětivější.

HTTP metoda / GraphQL operace	Doba vyhotovení		Počet napsaných znaků		Srozumitelnost syntaxe kódu	
	čas	body	počet	body	srozumitelnost	body
GET / createObject	0-5 min.	10	Méně než 100	10	Vysoká	10
	5-15 min.	8	100 až 300	8	Střední vyšší	8
	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4
POST / addObject	0-5 min.	10	Méně než 100	10	Vysoká	10
	5-15 min.	8	100 až 300	8	Střední vyšší	8

	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4
PATCH / updateObject	0-5 min.	10	Méně než 100	10	Vysoká	10
	5-15 min.	8	100 až 300	8	Střední vyšší	8
	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4
DELETE / deleteObject	0-5 min.	10	Méně než 100	10	Vysoká	10
	5-15 min.	8	100 až 300	8	Střední vyšší	8
	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4

Tabulka 1 Bodování skóre uživatelské přívětivosti

Zdroj: autor

Tabulka číslo 1 popisuje metodiku přiřazení bodu způsobem, kdy je každému z měřených ukazatelů přiřazena stejná váha a výsledné skóre přívětivosti je tedy definováno jako aritmetický průměr. Tento přístup lze využít pouze pro případy zkoumání, které nijak nezohledňují preference hodnotícího.

Za účelem eliminace subjektivity bude dále zavedena tabulka rozšiřující výpočet skóre o element vah, kdy každému ukazateli bude na základě preferencí hodnotícího přiřazena váha a následně výsledné skóre přívětivosti bude definováno jako vážený průměr všech ukazatelů a jejich přiřazených vah. Váha *w* může nabývat hodnot od 0 do 1 a označuje váhu, kterou bude bodová hodnota každého z měřených ukazatelů násobena před vydělením součtu vah pro získání váženého průměru. Součet vah všech 3 ukazatelů je vždy roven 1. Hodnoty vah *w* jsou za účelem srovnání definované následující škálou:

- $w = 0$ – ukazatel je pro výzkum zcela nedůležitý
- $w = 0{,}5$ – ukazatel je pro výzkum důležitý
- $w = 1$ – ukazatel je pro výzkum kritický

V tabulce číslo 2 je metodika výpočtu skóre přívětivosti obohacena o váhu *w*. Maximální bodový zisk pro každý z ukazatelů (zisk s vahou 1) je 40 bodů, minimální bodový zisk pro každý ukazatel (zisk s vahou 0) je 0 bodů. Na základě vlastních preferencí uživatel určuje míru relevance jednotlivých ukazatelů, které se následně promítnou do celkového bodového skóre. Tímto způsobem bude docíleno větší objektivity s možností využití metodiky pro prostředí s různými okolnostmi, kdy například začínající programátor

bude preferovat ukazatel srozumitelnosti kódu, zatímco doba vyhotovení pro něj nebude tak podstatným faktorem.

HTTP metoda / GraphQL operace	Doba vyhotovení		Počet napsaných znaků		Srozumitelnost syntaxe kódu	
Váha w	0 - 1		0 - 1		0 - 1	
	čas	body	počet	body	srozumitelnost	body
GET / createObject	0-5 min.	10	Méně než 100	10	Vysoká	10
	5-15 min.	8	100 až 300	8	Střední vyšší	8
	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4
POST / addObject	0-5 min.	10	Méně než 100	10	Vysoká	10
	5-15 min.	8	100 až 300	8	Střední vyšší	8
	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4
PATCH / updateObject	0-5 min.	10	Méně než 100	10	Vysoká	10
	5-15 min.	8	100 až 300	8	Střední vyšší	8
	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4
DELETE / deleteObject	0-5 min.	10	Méně než 100	10	Vysoká	10
	5-15 min.	8	100 až 300	8	Střední vyšší	8
	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4

Tabulka 2 Bodování uživatelské přívětivost přes vážený průměr

Zdroj: autor

Za účelem zpřístupnění metodiky dalším uživatelům je soubor pro výpočet uživatelského skóre přívětivosti včetně zakomponované složky vah součástí přílohy číslo 1.

3.7 Přehled soudobého stavu

Jelikož jsou REST i GraphQL zavedenými technologickými přístupy, v rámci akademické půdy i mimo ni již proběhly výzkumné pokusy o jejich srovnání, a to z různých pohledů. Tato kapitola se zaměřuje na již dostupné publikace a závěry, které ve svém zkoumání přinesly.

Ve své práci *How fast GraphQL is compared to REST APIs* se autoři zaměřili na srovnání GraphQL a REST z pohledu rychlosti a velikosti datového přenosu. To jsou taktéž hlavní metriky zkoumané v rámci této diplomové práce. V rámci svého srovnávacího experimentu autoři došli k závěru, který favorizuje GraphQL jak v otázce rychlosti, tak v otázce nezbytného datového přenosu nutného k vyřízení požadavku. Na základě srovnávacích analýz autoři dále zjistili, že v rámci dvou oddělených srovnávacích pokusů bylo GraphQL výrazně rychlejší a lehčí než REST. V rámci rychlosti byl GraphQL přístup o 35 % až 46 % rychlejší a o 21 % až 71 % lehčí co do velikosti požadavků v kilobajtech. Autoři zároveň zdůrazňují, že závěry pokusů jsou platné pouze s ohledem na zvolenou metodologii a konkrétní zkoumané situace. (35)

Další prací, která se zabývala srovnáním výkonu mezi REST a GraphQL je práce *A Comparative study between Graph-QL&Restful services in API management of stateless architectures*. V rámci své práce autoři provedli celkem 3 srovnávací experimenty, pokaždé s jinými nároky na množství přenesených dat. V prvním případě autoři využili 1 REST API endpoint oproti 1 GraphQL endpointu. Zaslání požadavku o 1 000 záznamech neukázalo na významnější rozdíly. V druhém případě autoři využili 2 REST API endpointy a 1 GrahpQL endpoint a pokus realizovali požadavkem o 10 000 záznamech. V tomto případě GraphQL vyřídilo požadavek s časem o 35 % rychlejším, než v tomu bylo případě REST API. V třetím případě bylo využito 3 REST API endpointů a 1 GraphQL endpointu a velikost požadavků tentokrát byla 100 000 záznamů. V tomto případě vykázalo GraphQL přibližně o 40 % rychlejší vyřízení požadavku, než v tomu bylo v případě REST služeb. (36)

Naopak autoři práce *Evaluating GraphQL and REST API Services Performance in a Massive and Intenstive Accessible Information System* došly k odlišným výsledkům. Jejich práce se orientovala na zodpovězení otázek rychlosti a dále také výpočetní zátěží na procesor (CPU) a operační paměť (RAM). Experiment byl narozdíl od izolovaného prostředí v předchozí uvedené práci proveden v rámci komplexního informačního systému. Výsledky ukazují, že je REST o 51 % rychlejší, co se týče doby odezvy a o 37 % efektivnější, co se týče propustnosti, tedy množství přenesených dat za jednotku času. Pokud jde o využití výpočetních zdrojů, výsledky analýzy ukázaly, že je GraphQL o 37 % úspornější v požadavcích na CPU a o 40 % úspornější v požadavcích na RAM. V rámci závěru práce autoři jako jednu z možných příčin uvádí to, že GraphQL k vyřízení operací (dotazů, ale i mutací) komunikuje pouze s jednotlivým koncovým bodem, zatímco REST k získávání dat

týkající se více entit využívá zpravidla více koncových bodů. Stejně jako v předchozí práci autoři závěrem uvádí, že rozhodnutí o využití REST nebo GraphQL závisí na konkrétních požadavcích systému a konkrétní situaci, ve kterých by měly být technologie použity. (37)

Autoři práce *REST API vs GraphQL – A literature and experimental study* ve svém zkoumání došli k podobnému závěru, a to sice, že REST API je oproti GraphQL rychlejší, pokud jde o rychlost přenosu dat. Tento závěr platí pro všechny pokusy provedené v rámci jejich srovnávacího experimentu, kdy došlo k oddělenému srovnání REST a GraphQL. Autoři dále doporučují vhodnost použití jednotlivých přístupů na konkrétních situacích. GraphQL je na základě jejich zkoumání vhodné pro využití v prostředích, která jsou limitována na šíři pásma jako jsou například mobilní aplikace, aplikace typu Internet of things nebo aplikace pro tzv. wearables zařízení (např. chytré hodinky). Svým principem fungování je GraphQL také vhodné pro využití v aplikacích, které budou těžit z jeho schopnosti větvení dotazů a dat (např. pro dohledání knihy a autora knihy jediným dotazem na jediný GraphQL endopoint). Autoři naopak doporučují využití REST architektury v prostředí, kde je využíváno ukládání většího množství dat do mezipaměti a jsou kladeny náročnější požadavky na autentizaci. (14)

Poslední ze zkoumaných prací v otázce srovnání výkonu REST API oproti GraphQL nese název *Comparative Analysis of Rest and GraphQL Technology on Nodejs-Based Api Development*. Její autoři zkoumali rychlost vyřízení požadavku pro 4 různé REST API metody a jejich ekvivalenty v podobě dotazů a mutací na straně GraphQL. Při vykonání pokusu na požadavku o 1000 záznamech autoři došli k závěru, že rychlost vyřízení požadavku s využitím REST API je rychlejší ve všech 4 operacích (GET, POST, PUT, DELETE) než jejich GraphQL alternativy. (38)

Jelikož je jako jeden z cílů diplomové práce definováno zhodnocení technické náročnosti a uživatelské přívětivosti týkající se přípravy a následné práce s údržbou přístupu REST a GraphQL, jsou součástí zkoumání také práce, které se věnují právě těmto aspektům. V práci *Comparison of REST and GraphQL Interfaces for OPC UA* autoři přišli se závěry, ve kterých GraphQL uvádí jako uživatelsky vhodnější a přívětivější z hlediska implementace, protože GraphQL samotné poskytuje grafické uživatelské rozhraní s názvem GraphiQL, které umožňuje přímou kontrolu a zkoušení funkčnosti GraphQL dotazů a mutací přímo v prostředí prohlížeče. (39)

V práci *REST vs GraphQL: A Controlled Experiment* se autoři zaměřili na otázku náročnosti implementace, kdy podnikli experiment s 22 subjekty s aspoň jednoroční praxí v oblasti programování. Všichni účastníci výzkumu měli za úkol implementovat REST API operaci a GraphQL dotazy pro získání dat z GitHub serveru. Účastníkům byl po provedení implementace položen soubor otázek, v rámci kterých měli účastníci zodpovědět, kolik času jim zabrala implementace v jedné a v druhé technologii a následně jak subjektivně hodnotí svou zkušenost, co se týče náročnosti práce s oběma přístupy. Podobně jako v předchozí práci autoři došli k závěru, že je GraphQL programátory považováno za jednodušší z hlediska časové náročnosti na vývoj. Zároveň účastníci výzkumu uvedli, že GraphQL implementace vyžaduje méně snahy, a to i pro vývojáře, kteří nemají předchozí zkušenost s touto technologií. Většina účastníků opět uvedla jako jednu z hlavních výhod GraphQL jeho doprovodný nástroj GraphiQL. Jeho funkce našeptávače ulehčující programátorům práci při definování správného formátu dotazů. Posledním zmíněným benefitem GraphQL je snazší syntaxe pro lidské pochopení zdrojového kódu a méně práce s definováním parametrů. (40)

V rámci analýzy soudobého stavu byly prozkoumány dostupné zdroje rozebírající problematiku srovnání REST API a GraphQL. Stejně jako diplomová práce dostupná díla porovnávala tyto dva přístupy především na základě výkonu (tj. rychlost vyřízení požadavku, případně velikost přenášených dat a z toho vyplývající náročnost na síť) a následně také z hlediska uživatelské přívětivosti pro programátory, kteří stojí na implementační straně webových aplikací a jejich API. V rámci srovnání výkonu dostupné zdroje neposkytly jednoznačný závěr, jelikož z hlediska rychlosti vyřízení požadavku došly zkoumané práce k odlišným výsledkům. Tyto odlišnosti mohou být do určité míry způsobeny odlišnými podmínkami realizace jednotlivých experimentů. Práce provádějící experimenty spíše v izolovaném prostředí obsahující pouze nezbytné komponenty pro realizaci experimentu favorizují spíše přístup využívající GraphQL (*How fast GraphQL is compared to REST APIs, Comparative study between Graph-QL&Restful services in API management of stateless architectures*). Naopak práce, které do svých testovacích prostředí vnesly vyšší úroveň komplexity (např. komplexní IT systém, který neslouží výhradně pro realizaci experimentu nebo implementace mezipaměti jak na straně REST, tak GraphQL) favorizují REST API (*Evaluating GraphQL and REST API Services Performance in a Massive and Intenstive Accessible Information Systém, REST API vs GraphQL – A literature*

and experimental, Comparative Analysis of Rest and GraphQL Technology on Nodejs-Based Api Development).

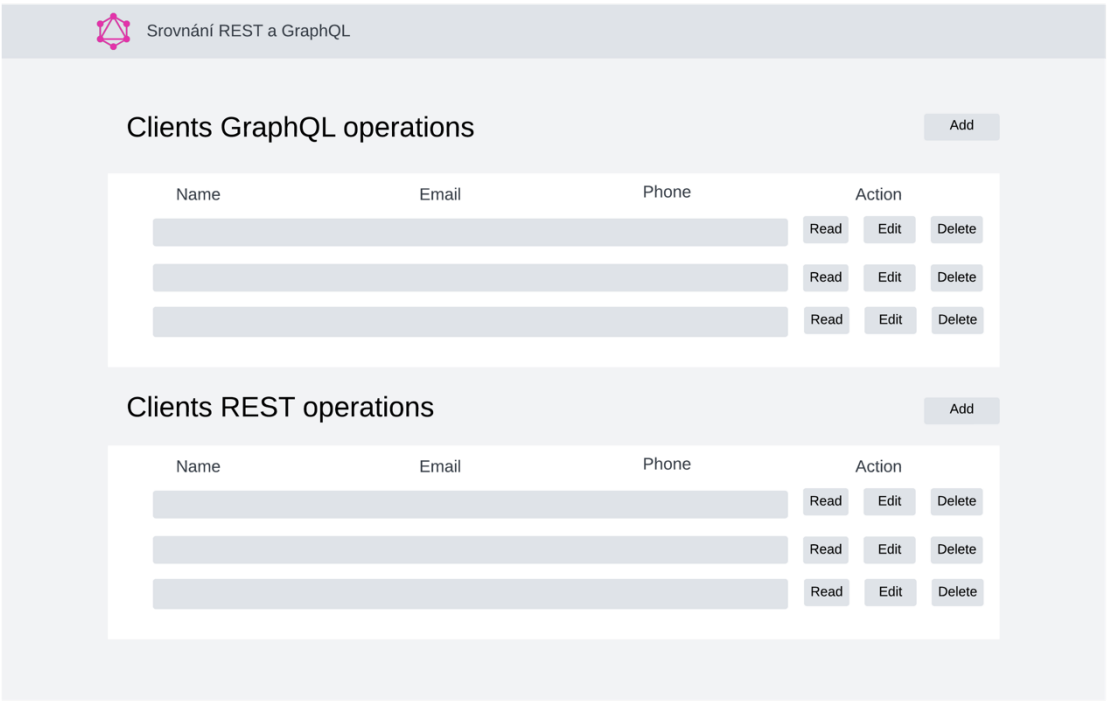
Analýza soudobého stavu se zaměřila i na otázku přívětivosti práce s každým zkoumaným přístupem z pohledu programátora. Zkoumané zdroje potvrdily, že GraphQL je z hlediska uživatelské přívětivosti preferovanou variantou pro většinu programátorů. Je tomu tak z hlediska potřebného času na vývoj tohoto rozhraní, kdy REST vyžaduje více času a je taktéž hůř uživatelsky čitelný než GraphQL syntaxe. Jednoduchost práce s GraphQL dále podporuje nástroj GraphiQL, který poskytuje uživatelské rozhraní pro zkoušení GraphQL dotazů a mutací přímo do databáze z prostředí prohlížeče. GraphiQL navíc svým uživatelům díky našeptávání pomáhá s definicí dotazů a mutací.

4 Vlastní práce

4.1 Návrh uživatelského rozhraní

4.1.1 Domovská obrazovka

V rámci standardů ve vývoji webových aplikací, které disponují svým vlastním uživatelským rozhraním, byly připraveny návrhy v podobě drátových modelů. Tyto návrhy zachytí základní rozložení prvků webové aplikace v prohlížeči, a tak nastíní výčet funkcí jednotlivých stránek aplikace.



Obrázek 8 Drátový návrh domovské obrazovky aplikace

Zdroj: autor

Na obrázku číslo 8 je k dispozici návrh drátového modelu domovské obrazovky aplikace. Domovská obrazovka umožní uživateli základní operace nad daty, kterými je získání informací o spravovaných entitách – klientech.

Klient je za účelem experimentu smyšlená entita definovaná třídou Client s atributy definovanými v tabulce číslo 3.

Název atributu	Datový typ
Name	String
E-mail	String
Phone	String

Tabulka 3 Třída Client

Zdroj: autor

Pro každý přístup (tj. REST a GraphQL) existuje samostatná tabulka výskyty klientů v aplikaci. Zatímco data jsou zobrazena uživateli v oddělených tabulkách, jejich zdroj na straně databáze je identický, tudíž úprava v jedné tabulce se zákonitě projeví i v druhé tabulce. Toto rozdělení umožní jednoznačné rozlišení REST operací (metod) a GraphQL dotazů/mutací. Domovská obrazovka tedy uživateli nabídne 2 oddělené rozhraní. Aplikace bude funkčně obstarávat následující operace:

- Získání dat o všech klientech (načtení domovské obrazovky)
- Získání dat o jednom klientovi
- Úprava existujícího klienta
- Smazání výskytu klienta

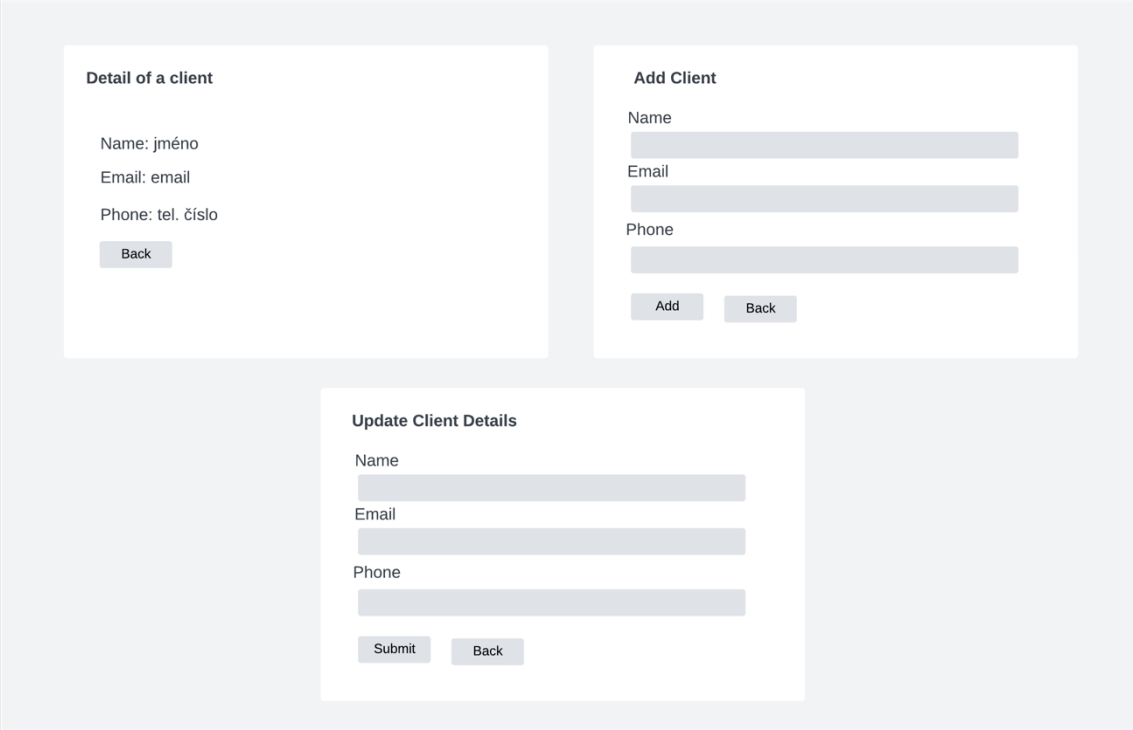
4.1.2 Výsledek operace

Po vykonané REST operaci nebo GraphQL dotazu či mutaci budou výsledky operace dostupné v konzoli webového prohlížeče.

- Doba vyřízení požadavku – vypsání výsledku formou Javacript metody console.log()
- Velikosti odpovědi ze serveru v Kb – vypsání výsledku v rámci hlavičky Content-Length v detailu požadavku

4.1.3 Operace nad entitou klient

Pro případy operací nad jednotlivými výskyty třídy Client je dále zapotřebí připravit návrh drátového modelu pro zobrazení, přidání a úpravu záznamu. Operace mazání bude realizována prostým potvrzením smazání. Tyto návrhy jsou k dispozici na obrázku číslo 9.



Obrázek 9 Návrh drátového modelu operace třídy Klient

Zdroj: autor

4.2 Příprava serverové části aplikace

4.2.1 Instalace balíčků a dependencí

K vytvoření projektu a následnému vývoji bylo využito nástroje Visual Studio Code. Po vytvoření projektové složky je zapotřebí nainstalovat všechny balíčky a dependence, které budou v rámci projektu využívány. Součástí rozsahu práce není instalace Node na počítač, je tudíž pouze potřeba provést instalaci navazujících balíčků s využitím Node package manager. Terminálové příkazy sloužící k instalaci balíčků jsou k náhledu na obrázku číslo 10.

```
matejpovolny@Matejs-MacBook-Pro-2 PROJECT-MGMT-APP copy 2 % npm init -y npm i express express-graphlql graphql mongoose cors colors -
D nodemon dotenv
```

Obrázek 10 Instalace npm balíčků a dependencí pro serverovou část aplikace

Zdroj: autor

4.2.2 Vytvoření serveru a napojení MongoDB databáze

V rámci projektu je zapotřebí vytvořit serverovou složku, která bude obsahovat všechen zdrojový kód týkající se serverové části aplikace. Tento serverový kód je rozdělen do několika samostatných souborů, kde mezi hlavní z nich patří soubor index.js, soubor db.js soubor .env, soubor schema.js a soubor package.json.

V rámci souboru index.js je potřeba do aplikace importovat veškeré dependence, s kterými bude serverová část aplikace pracovat a následně je inicializovat v kódu. Jedná se především o import express frameworku, definování portu, přes který bude realizováno napojení na databázi, inicializaci databáze, inicializaci CORS a inicializaci GraphQL. Ukázka zdrojového kódu souboru index.js je k dispozici na ukázce kódu číslo 6.

```
const express = require('express');
const colors = require('colors');
const cors = require('cors');
require('dotenv').config();
const { graphqlHTTP } = require('express-graphql');
const schema = require('./schema/schema');
const connectDB = require('./config/db');
const port = process.env.PORT || 4000;
const mongoose = require('mongoose')
const Client = require('../server/models/Client')

const app = express();

// Connect to database
connectDB();
app.use(cors());
app.use(express.json());

app.get('/', async (req, res) => {
  try {
      const clients = await Client.find()
      res.json(clients)
  } catch (err) {
      res.status(500).json({message: err.message })
  }
})

app.use(
  '/graphql',
  graphqlHTTP({
    schema,
```

```
    graphiql: process.env.NODE_ENV === 'development',
  })
);

app.listen(port, console.log(`Server running on port ${port}`));
```

Ukázka kódu 6 Obsah souboru index.js na straně serveru

Zdroj: autor

Jak je vidět z ukázky kódu číslo 6, tak v rámci souboru index.js dochází ke spouštění databáze přes funkci connectDB(). Parametry funkce connectDB() jsou definovány v souboru db.js. Obsah kódu obsaženého v tomto souboru je k dispozici na ukázce kódu číslo 7.

```
const mongoose = require('mongoose');

const connectDB = async () => {
    const conn = await mongoose.connect(process.env.MONGO_URI);

    console.log(`MongoDB Connected:
${conn.connection.host}`.cyan.underline.bold);
};

module.exports = connectDB;
```

Ukázka kódu 7 Obsah souboru db.js definující funkci connectDB()

Zdroj: autor

Funkce connectDB() asynchroně zpracovává funkci mongoose.connect(), s jejímž využitím dojde k napojení na MongoDB databázi. Funkce mongoose.connect() přijímá parametr *MONGO_URI*, který je definován v rámci .env souboru a obsahuje connection string vygenerovaný MongoDB po vytvoření databáze a získaný z uživatelského rozhraní MongoDB Atlas. Obsah zdrojového kódu v souboru .env je k dispozici na ukázce kódu číslo 8. Za účelem bezpečnosti bylo v rámci connection string v proměnné *MONGO_URI* nahrazeno reálné heslo potřebné k přístupu k databázi za výraz „password“.

```
NODE_ENV = 'development'
PORT = 4000
MONGO_URI =
'mongodb+srv://matejpovolny:<password>cluster0.qsltoaf.mongodb.net/mgmt_db?retryW
rites=true&w=majority'
```

Ukázka kódu 8 Obsah souboru .env

Zdroj: autor

4.2.3 Definice datových modelů

Jak již bylo zmíněno v předcházejících kapitolách, projekt bude pracovat s třídou Client, nad kterou budou moci uživatelé vykonávat operace ať už s využitím REST API endpointů nebo GraphQL dotazů a mutací.

Model Client odpovídá definici třídy Client a disponuje následujícími atributy:

- Jméno – textový řetězec
- Email – textový řetězec
- Phone – textový řetězec

Ukázka kódu číslo 9 uvádí definici modelu Project v souboru Client.js.

```
const mongoose = require('mongoose');

const ClientSchema = new mongoose.Schema( {
    name: {
        type: String,
        required: true
    },
    email: {
        type: String,
        required: true
    },
    phone: {
        type: String,
        required: true
    },


})

module.exports = mongoose.model('Client', ClientSchema);
```

Ukázka kódu 9 Definice modelu Client v souboru Client.js

Zdroj: autor

4.2.4 Vytvoření GraphQL schéma

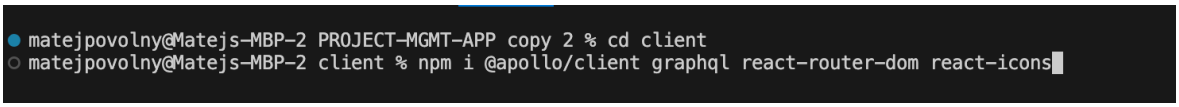
Aby bylo možné realizovat operace s GraphQL na straně serveru, je zapotřebí vytvořit GraphQL schéma v rámci souboru schema.js. Soubor obsahuje definici typů objektů, s kterými bude GraphQL pracovat (které odpovídají předefinovaným modelům, které jsou do souboru schema.js importovány), definici GraphQL dotazů a v neposlední řadě

také definici GraphQL mutací. Ukázka celého souboru schema.js je uvedena v příloze číslo 2.

4.3 Příprava klientské části aplikace

4.3.1 Instalace balíčků a dependencí

Stejně jako u serverové části je zapotřebí i na straně klienta nainstalovat potřebné balíčky a dependence. V nově vytvořené složce client byl nainstalován Apollo client, GraphQL, react-router-dom, react-icons. Terminálové příkazy jsou k náhledu na obrázku číslo 11.



Obrázek 11 Instalace npm balíčků a dependencí pro klientskou část aplikace

Zdroj: autor

4.3.2 Příprava App.js souboru

V rámci souboru App.js je iniciován Apollo Client za účelem zprovoznění GraphQL na straně klienta. Jako parametr pro spuštění GraphQL na serveru byla použita URI adresa s portem, na kterém je dostupná databáze. V případě aplikace jde o 'http://localhost:4000/graphql'. V rámci souboru App.js jsou také definovány routy webové aplikace označující URL cesty směřující na konkrétní stránky, včetně domovské obrazovky (root adresy) definované lomítkem („/“) a dalších obrazovek, například „/createclient“ za účelem nasměrování do obrazovky pro tvorbu nového výskytu třídy Client. Obsah souboru App.js je k dispozici v příloze číslo 3.

4.3.3 GraphQL dotazy a mutace

V klientské části aplikace je rovněž zapotřebí nadefinovat obsah dotazů a mutací použitých pro získání a úpravu dat za pomoci GraphQL. Na základě povahy operací byly operace zaneseny do dvou soborů:

- clientQueries.js
- clientMutations.js

V rámci souborů jsou za pomocí jazyka Graph Query Language (GQL) definovány operace nad grafy. Soubory definují logiku pro získávání a úpravu dat s pomocí GraphQL na straně klienta. Obsah souboru clientQueries.js je k dispozici na ukázce kódu číslo 10 a ukazuje GQL dotaz na získání dat jednoho klienta uloženého v databázi. Data jsou uložena do proměnné GET_CLIENT, která je následně využita pro zobrazení dat v rámci komponent či stránek aplikace, uložených v souborech JSX.

```
import {gql} from '@apollo/client';

const GET_CLIENT = gql`
  query getClient($id: ID!) {
    client(id: $id) {
      id
      name
      email
      phone
    }
  }
`;

export { GET_CLIENT };
```

Ukázka kódu 10 Obsah souboru clientQueries.js

Zdroj: autor

4.3.4 JSX soubory pro vykreslení uživatelského rozhraní

V rámci projektu je využito souborů ve formátu JSX k vykreslení uživatelského rozhraní včetně logiky navigace v rámci aplikace (linkování) a zobrazení dat uživatelům včetně jejich možné úpravy. Zkratka JSX znamená JavaScript XML a soubory tohoto typu vývojářům, kteří je využívají, umožňují vkládat HTML (veškerý syntax) do React komponent. (29)

V rámci projektu byly vytvořeny JSX soubory pro stránky aplikace a také pro komponenty, které jsou přepoužívány na více stránkách aplikace. Obsah souboru CreateClient.jsx, který k dispozici na routu vedoucí na adresu /createclient je k dispozici v příloze číslo 4.

Zdrojový kód aplikace je k dispozici v příloze číslo 5.

4.4 Realizace srovnávacího experimentu

Jak již bylo navrženo v teoretické části práce, srovnávací experiment bude spočívat v provádění REST API operací pomocí endpointů realizovaných přes standardní HTTP metody (GET, POST, PATCH, DELETE) a srovnávat je s jejich ekvivalenty v podobě GraphQL dotazů a mutací. Všechny akce budou realizovány nad stejnými daty, tj. výskyty třídy Client uložených v rámci databáze MongoDB.

Měření srovnávacích pokusů bylo realizováno podle tabulky číslo 2 pro metody REST API, každá metoda je realizována ve 3 provedeních.

Jednotlivá provedení se od sebe navzájem liší počtem opakování požadavku. Každý z požadavků je v jednotlivých provedeních opakován jedenkrát, stokrát, nebo tisíckrát. Opakování požadavku přispěje k vyšší diverzitě a významnějšímu rozsahu zkoumaných scénářů.

Zároveň pro každé z provedeních bude provedeno 10 pokusů, kdy výsledná hodnota provedení bude určena aritmetickým průměrem všech těchto pokusů.

Tabulka číslo 4 znázorňuje rozvržení provedení pokusů pro REST API.

HTTP Metoda	Endpoint	Počet opakování	Počet pokusů	Popis
GET	/clients/:id	1	10	Získání dat 1 klienta
GET	/clients/:id	100	10	Získání dat 100 klientů
GET	/clients/:id	1000	10	Získání dat 1000 klientů
POST	/clients	1	10	Vytvoření 1 klienta
POST	/clients	100	10	Vytvoření 100 klientů
POST	/clients	1000	10	Vytvoření 1000 klientů
PATCH	/clients/:id	1	10	Aktualizace 1 klienta
PATCH	/clients/:id	100	10	Aktualizace 100 klientů
PATCH	/clients/:id	1000	10	Aktualizace 1000 klientů
DELETE	/clients/:id	1	10	Smazání 1 klienta
DELETE	/clients/:id	100	10	Smazání 100 klientů
DELETE	/clients/:id	1000	10	Smazání 1000 klientů

Tabulka 4 Realizace REST API požadavků za účelem měření výkonu

Zdroj: autor

Stejným způsobem jsou realizovány operace GraphQL, jak je popsáno v tabulce číslo 5.

GraphQL operace	Query	Počet záznamů	Počet pokusů	Popis
Query	getClient	1	10	Získání dat 1 klienta
Query	getClient	100	10	Získání dat 100 klientů
Query	getClient	1000	10	Získání dat 1000 klientů
Mutace	addClient	1	10	Vytvoření 1 klienta
Mutace	addClient	100	10	Vytvoření 100 klientů
Mutace	addClient	1000	10	Vytvoření 1000 klientů
Mutace	updateClient	1	10	Aktualizace 1 klienta
Mutace	updateClient	100	10	Aktualizace 100 klientů
Mutace	updateClient	1000	10	Aktualizace 1000 klientů
Mutace	deleteClient	1	10	Smazání 1 klienta
Mutace	deleteClient	100	10	Smazání 100 klientů
Mutace	deleteClient	1000	10	Smazání 1000 klientů

Tabulka 5 Realizace GraphQL operací za účelem měření výkonu

Zdroj: autor

4.5 Vyhodnocení výsledků měření

4.5.1 Získání dat

4.5.1.1 Rychlost vyřízení požadavků

Jako první z pokusů bylo realizováno srovnání na operaci získání dat ze serveru. V rámci REST se jedná o požadavek využívající HTTP metodu GET a v rámci GraphQL jde o dotaz getClient. Souhrnné výstupy uvedené v tabulce číslo 6 ukazují aritmetický průměr vypočítaný z 10 prováděných pokusů pro každou variantu provedení. Detailní výstup neobsahující agregované průměrné hodnoty, nýbrž výstupy jednotlivých pokusů, je k dispozici v příloze číslo 6.

Jak je patrné z tabulky č. 6, aritmetický průměr poukazuje na lepší výkon v rychlosti vyřízení požadavku u GrahpQL, nicméně pouze tehdy, pokud se jedná o ojedinělý záznam. V tomto případě vykazuje GraphQL o 75 % rychlejší výkon. Pakliže je dotazování na server opakováno v rámci více požadavků (100 a 1000), celková doba vyřízení požadavků je u

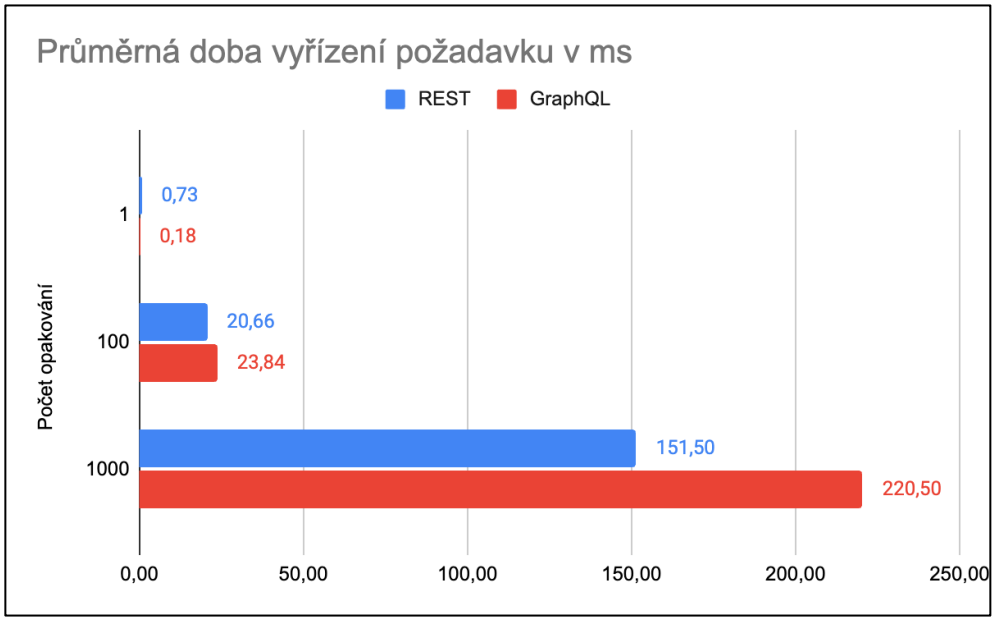
GraphQL naopak delší než u tradičního REST přístupu, a to konkrétně o 13,3 % v případě 100 požadavků a o 31,3 % v případě 1000 požadavků. Kromě aritmetického průměru je v tabulce č. 6 k dispozici hodnota směrodatné odchylky poukazující na (ne)konzistenci naměřených výsledků.

HTTP Metoda / GraphQL operace	Endpoint / Query	Počet opakování	Aritmetický průměr	Směrodatná odchylka
GET	/client/:id	1	0,73	0,39
GET	/client/:id	100	20,66	2,03
GET	/client/:id	1000	151,50	29,54
Query	getClient	1	0,18	0,16
Query	getClient	100	23,84	7,74
Query	getClient	1000	220,50	26,20

Tabulka 6 Srovnání rychlosti požadavku získání dat v milisekundách

Zdroj: autor

Popsaný trend, kdy REST vykazuje lepší výsledky, než GraphQL hlavně pokud jde o práci s větším množstvím dat, je vizualizován na grafu číslo 1.

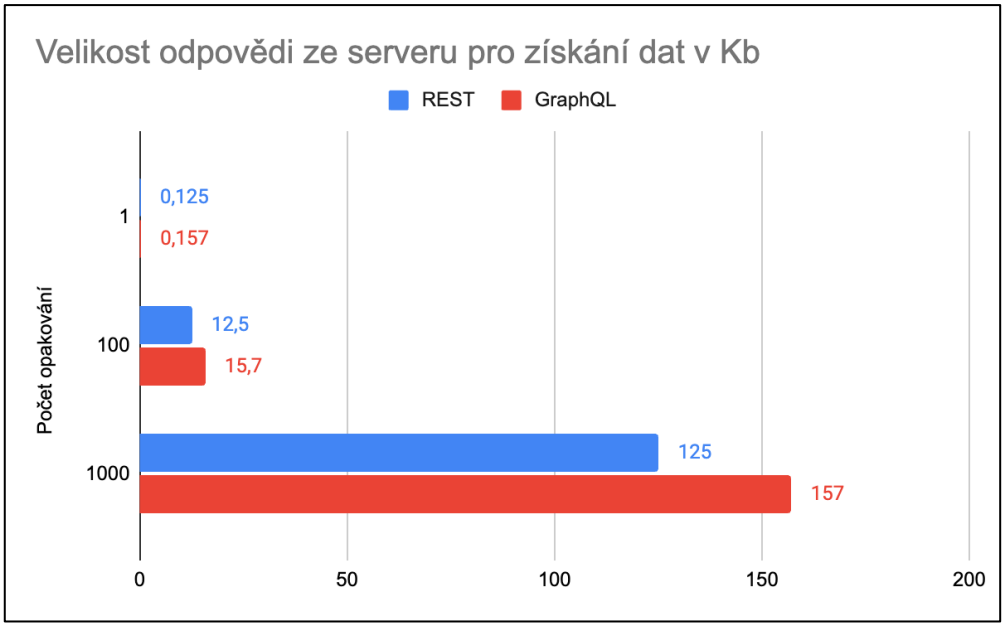


Graf 1 Průměrná doba vyřízení požadavku získání dat v ms

Zdroj: autor

4.5.1.2 Velikost odpovědi ze serveru

Kromě měření rychlosti vyřízení všech požadavků byla taktéž zaznamenána velikost odpovědi ze serveru v hlavičce Content-Length, kterou pro navrácení dat klientovi musel server poskytnout. Naměřené výsledky nevykazují významnější odchylky ani jednoho z přístupů, nicméně v rámci přenosu úspory dat vykazuje mírně lepší výsledky (tj. menší zátěž co se týče velikosti odpovědi) přístup REST. Souhrnné výsledky srovnání pokusu získání dat ze serveru ve spojitosti s velikostí odpovědi ze serveru jsou k dispozici na grafu číslo 2.



Graf 2 Velikost odpovědi ze serveru pro získání dat v Kb

Zdroj: autor

4.5.2 Vytvoření dat

4.5.2.1 Rychlost vyřízení požadavků

Na pokusy k získávání dat ze serveru bylo navázáno pokusy s opačnou operací, kterou je vytváření dat na serveru. Tato operace je realizována HTTP metodou POST v případě REST API přístupu a mutací s názvem addClient vytvořenou za těmito účely v GraphQL.

Souhrnné výsledky obsahující aritmetický průměr všech pokusů a jejich směrodatnou odchylku jsou k dipozici v tabulce číslo 7. Již z tabulky je patrné, že REST API vykazuje kratší dobu vyřízení požadavku na serveru ve všech provedeních. V případě jednoho opakování je o 50 % rychlejší než vyřízení u GraphQL, v případě 100 opakování potom o 23 % a v případě 1000 požadavků je rychlost kratší o 42 %. Společně s kratší dobou REST API taktéž vykazuje menší výkyvy ve výkonu, kdy je směrodatná odchylka doby vytvoření 1000 nových záznamů v databázi dokonce o 84 % menší než v případě GraphQL.

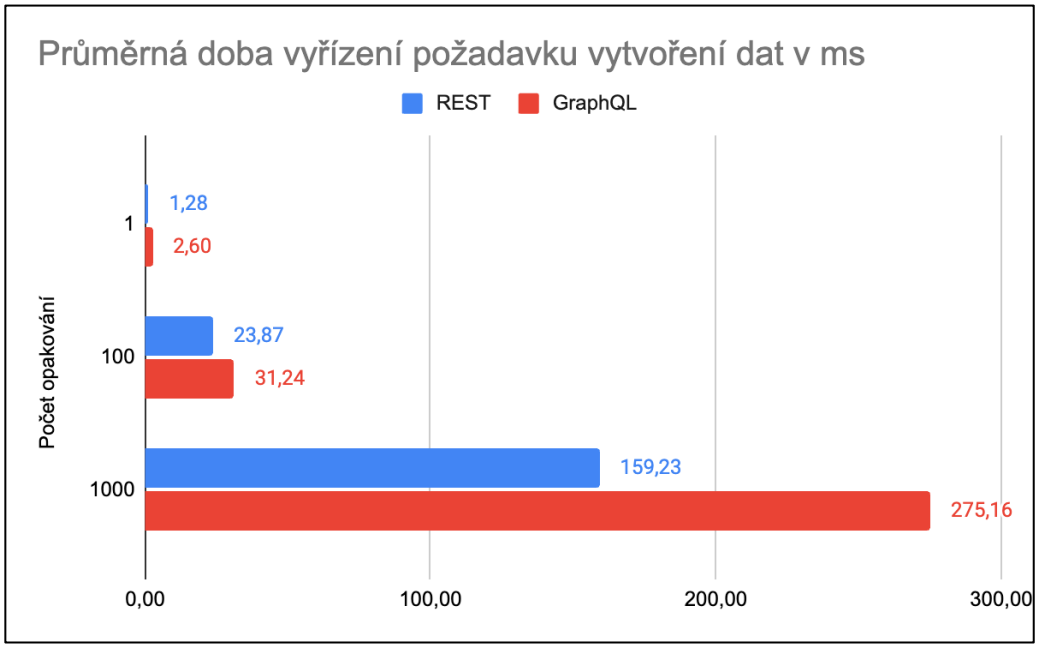
Detailní výstup neobsahující agregované průměrné hodnoty, nýbrž výstupy jednotlivých pokusů, je k dispozici v příloze číslo 7.

HTTP Metoda / GraphQL operace	**Endpoint / mutace**	**Počet opakování**	**Aritmetický průměr**	**Směrodatná odchylka**
POST	/clients	1	1,28	0,49
POST	/clients	100	23,87	2,83
POST	/clients	1000	159,23	6,32
Query	addClient	1	2,60	2,63
Query	addClient	100	31,24	6,61
Query	addClient	1000	275,16	40,71

Tabulka 7 Srovnání rychlosti požadavku získání dat v milisekundách

Zdroj: autor

Výsledky průměrné doby vyřízení požadavků pro vytvoření nových výskytů třídy Client na serveru jsou k dispozici na grafu číslo 3.

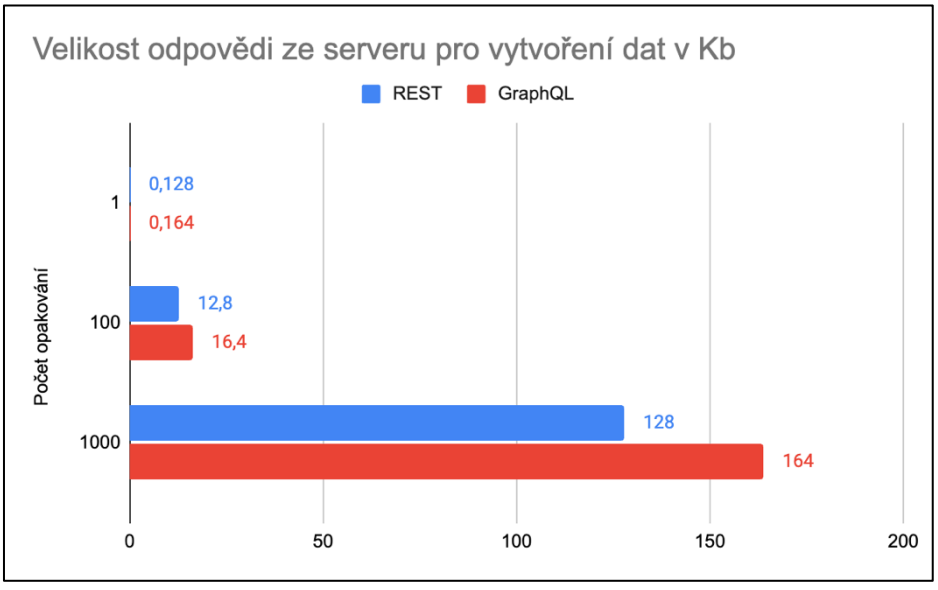


Graf 3 Průměrná doba vyřízení požadavku vytvoření dat v ms

Zdroj: autor

4.5.2.2 Velikost odpovědi ze serveru

Na grafu číslo 4 jsou k dispozici výsledky týkající se velikosti odpovědi zaslané od serveru po vytvoření dat na základě POST požadavku nebo addClient mutace.



Graf 4 Velikost odpovědi ze serveru pro vytvoření dat v Kb

Zdroj: autor

Stejně jako v přechozím pokusu týkajícího se čtení dat ze serveru, i zasílání dat na server hovoří ve prospěch REST API, kdy vykazuje o 22 % menší velikost přenosu dat směrem ze serveru.

4.5.3 Aktualizace dat

4.5.3.1 Rychlost vyřízení požadavků

V rámci REST API lze aktualizaci dat provádět přes dvě metody. První z nich je metoda PUT, v rámci které dochází k aktualizaci celého objektu, tj. všech jeho proměnných bez rozdílu, zda individuální hodnoty proměnných prošly změnou či nikoliv. Oproti tomu metoda PATCH aktualizuje pouze proměnné, ve kterých oproti původním hodnotám došlo ke změně. (6) PATCH je tedy principem fungování blíže aktualizaci dat s využitím GraphQL mutace a proto byl použit za účelem srovnávacího pokusu.

V rámci aktualizace proměnných v objektu se potvrzuje trend rychlejšího vyřízení požadavků s využitím REST API. Při jednom opakování požadavku aktualizace přes metodu PATCH trvala v průměru o 37 % méně. Konkrétně šlo o 0,83 milisekund, oproti tomu GraphQL aktualizace s využitím mutace updateClient při jednom opakování požadavku potřebovala 1,31 milisekund. Trend potvrzují i pokusy s opakováním požadavku stokrát a tisíckrát. REST API v případě 100 požadavků vyřizovalo všechny o 32 % rychleji, kdy potřebovalo 23,34 milisekund oproti 34,02 milisekundám u GraphQL. V případě 1000 požadavků šlo o 36 %, kdy bylo naměřeno 156,58 milisekund při využití REST metody oproti 246,39 milisekundám v případě využití GraphQL mutace updateClient. GraphQL si v rámci pokusů nicméně vedlo lépe v rámci konzistence při vyřízení operace s 1000 opakováními, kdy je směrodatná odchylka o více než 50 % menší. Souhrnná data jsou k dispozici v tabulce číslo 8 a jejich vizualizace v rámci grafu číslo 5.

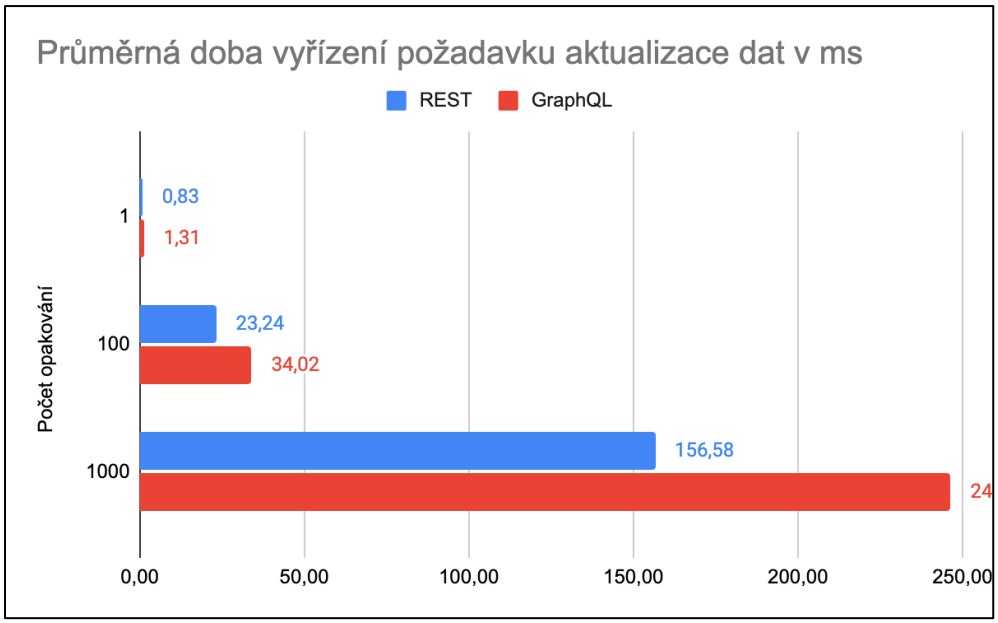
Detailní výstup neobsahující agregované průměrné hodnoty, nýbrž výstupy jednotlivých pokusů, je k dispozici v příloze číslo 8.

HTTP Metoda / GraphQL operace	Endpoint / mutace	Počet opakování	Aritmetický průměr	Směrodatná odchylka
PATCH	/clients/:id	1	0,83	0,16

PATCH	/clients/:id	100	23,24	3,20
PATCH	/clients/:id	1000	156,58	18,26
Mutace	updateClient	1	1,31	0,29
Mutace	updateClient	100	34,02	2,99
Mutace	updateClient	1000	246,39	7,79

Tabulka 8 Srovnání rychlosti požadavku aktualizace dat v milisekundách

Zdroj: autor

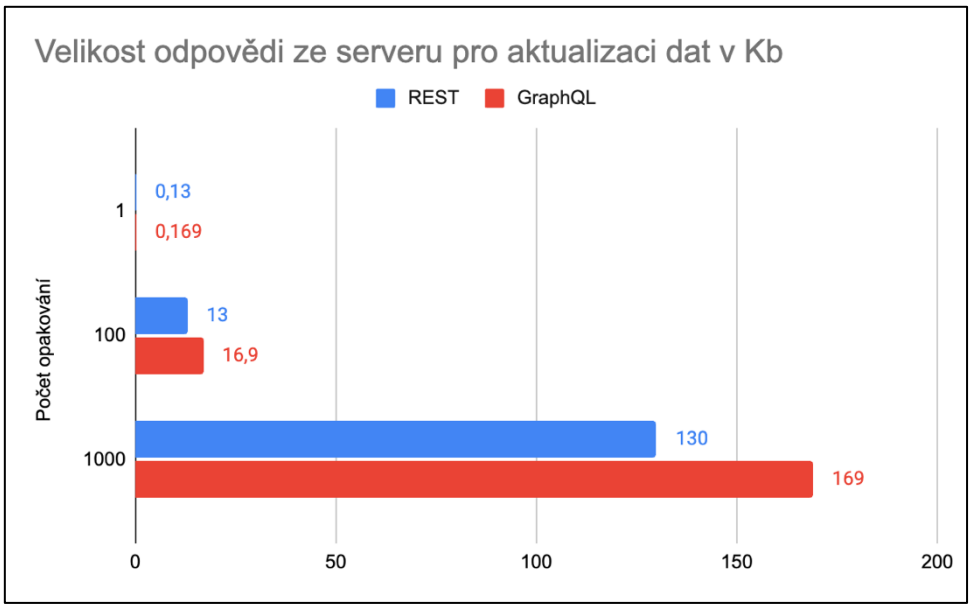


Graf 5 Průměrná doba vyřízení požadavku vytvoření dat v ms

Zdroj: autor

4.5.3.2 Velikost odpovědi ze serveru

V rámci vyhodnocení úspory datového přenosu při odpovědi ze strany serveru srovnání pro aktualizaci dat potvrzuje výsledky předchozích pokusů, kdy i v rámci aktualizace dat REST vykazuje menší hodnoty co do počtu přenesených Kb. Konkrétně jde o 23 % menší velikost oproti odpovědi ze serveru s operací využívající GraphQL. Srovnání velikosti odpovědi ze serveru pro aktualizaci dat s pomocí metody PATCH a mutace updateClient je k dispozici na grafu číslo 6.



Graf 6 Velikost odpovědi ze serveru pro aktualizaci dat v Kb

Zdroj: autor

4.5.4 Smazání dat

4.5.4.1 Rychlost vyřízení požadavků

Poslední z měřených operací je mazání dat ze serveru. REST API tuto operaci realizuje s HTTP metodou DELETE a pro mazání s využitím GraphQL byla vytvořena mutace deleteClient. Oba přístupy pracují s konkrétním objektem, kdy dochází k jeho mazání z databáze na základě ID. Jelikož po smazání objektu z databáze objekt v databázi nadále neexistuje, byla operace na výmaz dat provedena pouze s jedním opakováním požadavku za účelem zjednodušení pokusu.

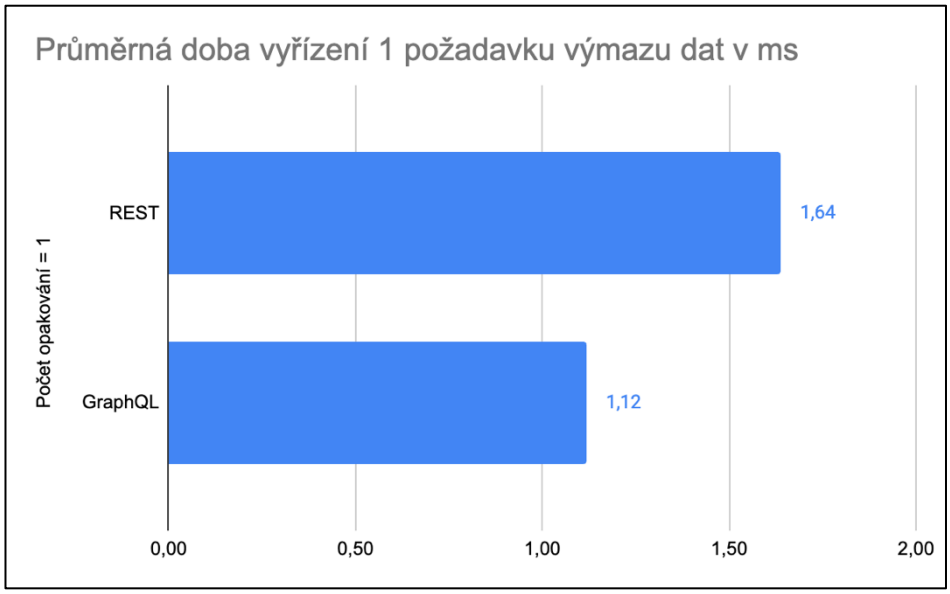
Srovnání operací výmazu dat ukázalo, že byl REST API požadavek vyřízen v průměru o 31 % pomaleji než totožná operace s využitím GraphQL. Konkrétně byla průměrná doba vyřízení požadavku, která byla vypočítána z 10 opakování, 1,64 milisekund v případě REST API a její metody DELETE a 1,12 milisekund v případě GraphQL mutace deleteClient. Konzistence výsledků je mezi oběma přístupy srovnatelná, kdy hodnota směrodatné odchylky byla 0,62 milisekund v případě REST přístupu a 0,71 milisekund v případě GraphQL.

Souhrnné výsledky jsou k dispozici v tabulce číslo 9 a grafu číslo 7. Detailní výstup neobsahující agregované průměrné hodnoty, nýbrž výstupy jednotlivých pokusů, je k dispozici v příloze číslo 9.

HTTP Metoda / GraphQL operace	Endpoint / mutace	Počet opakování	Aritmetický průměr	Směrodatná odchylka
DELETE	/clients/:id	1	1,64	0,62
Mutace	deleteClient	1	1,12	0,71

Tabulka 9 Srovnání rychlosti požadavku výmazu dat v milisekundách

Zdroj: autor

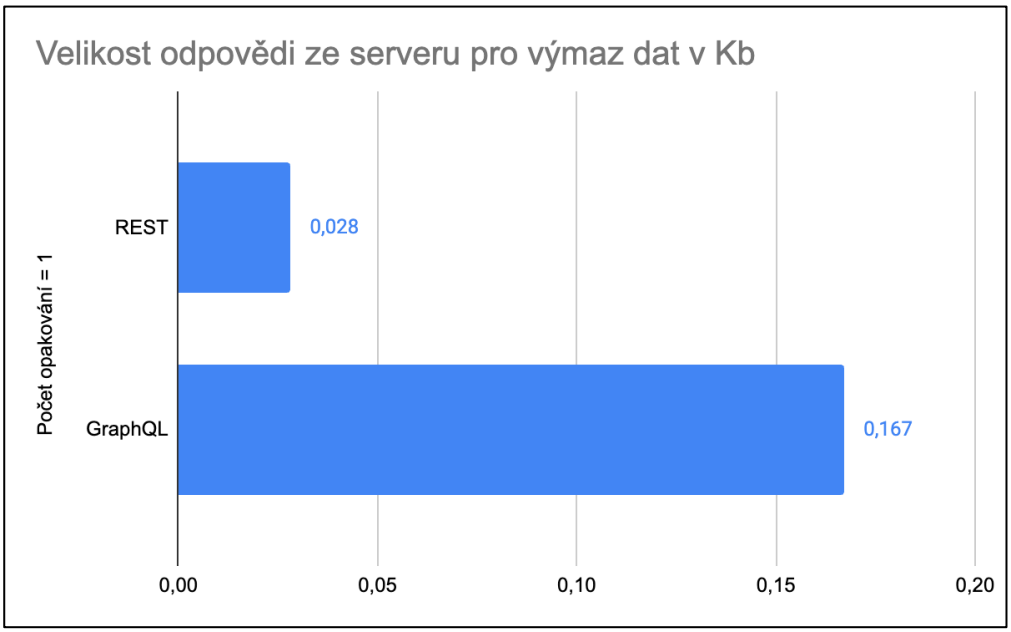


Graf 7 Průměrná doba vyřízení požadavku výmazu dat v ms

Zdroj: autor

4.5.4.2 Velikost odpovědi ze serveru

V případě srovnání datové velikosti odpovědi ze strany serveru REST API vykázalo v průměru o 83 % úspornější výsledek oproti GraphQL, kde průměrná odpověď ze strany serveru ukázala velikost pouhých 0,028 Kb, zatímco odpověď s využitím GraphQL 0,167 Kb. Srovnání výsledků je vizuálně znázorněno na grafu číslo 8.



Graf 8 Velikost odpovědi ze serveru pro výmaz dat v Kb

Zdroj: autor

4.6 Vyhodnocení přívětivosti

Na kvantitativní srovnání rychlosti vyřízení požadavků a velikosti odpovědi ze strany serveru bylo navázáno vyhodnocením přívětivosti práce s jednotlivými přístupy.

Hodnotící byl v tomto případě začátečníkem v obou zmíněných přístupech pro tvorbu aplikačního rozhraní. K vyhodnocení přívětivosti bylo využito skóre přívětivosti počítané přes vážený průměr definované v teoretické části práce.

Váhy byly na základě preference hodnotícího určeny následovně. Době vyhotovení byla připsána váha 0,375, což jej klasifikuje jako spíše středně důležitý faktor v celkovém hodnocení. Počtu napsaných znaků byla přiřazena váha 0,125, která jej řadí mezi faktory spíše nedůležité. Jelikož je hodnotící začátečníkem v tvorbě aplikačního rozhraní pro oba přístupy, je srozumitelnost kódu každého z přístupů nejdůležitějším faktorem z celkových tří, a proto mu byla připsána nejvyšší váha 0,5.

Jak bylo definováno v teoretické části práce, přístup ohodnocený více body ve výsledném váženém průměru je vyhodnocen jako uživatelsky přívětivější.

4.6.1 Skóre uživatelské přívětivosti REST API

Jako první byl vyhodnocen přístup REST API. Jako začínajícímu programátorovi trvalo hodnotícímu vyhotovení GET požadavku 5-15 minut, POST požadavku 15-30 minut, PATCH požadavku 15-30 minut a DELETE požadavku 5-15 minut, připisující celkové vážené skóre 10,5 bodů co se týče doby vyhotovení. Jak již bylo zmíněno v teoretické části práce, dobou vyhotovení se myslí čistý čas na psaní metody pro získání dat s využitím REST API endpointu, naopak není do něj zahrnut čas pro tvorbu uživatelského rozhraní v rámci JSX, ze kterého je metoda volána. Taktéž doba vyhotovení nepokrývá čas strávený zanesením mechanismů pro vyhodnocení kvantitativního srovnání, tj. měření doby vyřízení požadavku a velikost odpovědi ze strany serveru.

Vyhodnocení v oblasti počtu napsaných znaků ukázalo korelaci s dobou vyhotovení, kdy pro GET požadavek bylo potřeba 100 až 300 znaků, POST stejně jako PATCH požadavek s větším objemem vyžadovali srovnatelně 300 až 500 znaků a požadavek DELETE byl vyhotoven s počtem znaků od 100 do 300. Jelikož byla váha pro ukazatel počet napsaných znaků stanovena na 0,125, je celkové vážené skóre pro tento ukazatel rovno 3,5 bodům.

Nejdůležitějším ukazatelem, tedy ukazatelem s nejvyšší přiřazenou vahou z pohledu hodnotícího, byla srozumitelnost syntaxe kódu. Tomuto ukazateli byla přiřazená váha 0,5. Srozumitelnost požadavků GET, POST a PATCH byla vyhodnocena jako střední vyšší, kdežto požadavek DELETE díky své jednoduchosti získal 10 bodů. S váhou 0,5 je srozumitelnost kódu ohodnocena váženým skóre 17 bodů.

Váhy a skóre je k dispozici k náhledu v tabulce číslo 10.

HTTP metoda / GraphQL operace	Doba vyhotovení		Počet napsaných znaků		Srozumitelnost syntaxe kódu	
Váha w	0,375		0,125		0,5	
	čas	body	počet	body	srozumitelnost	body
GET	0-5 min.	10	Méně než 100	10	Vysoká	10
	5-15 min.	8	100 až 300	8	Střední vyšší	8
	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4
POST	0-5 min.	10	Méně než 100	10	Vysoká	10

	5-15 min.	8	100 až 300	8	Střední vyšší	8
	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4
PATCH	0-5 min.	10	Méně než 100	10	Vysoká	10
	5-15 min.	8	100 až 300	8	Střední vyšší	8
	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4
DELETE	0-5 min.	10	Méně než 100	10	Vysoká	10
	5-15 min.	8	100 až 300	8	Střední vyšší	8
	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4

Tabulka 10 Výpočet skóre přivětivosti pro REST API metody

Zdroj: autor

Skóre přívětivosti se rovná výsledku váženého průměru pro všechny 3 ukazatele. v případě REST API operací je vážený průměr roven 31. Skóre přívětivosti práce s REST API z pohledu programátora, který je začátečníkem v práci s tímto technologickým přístupem, je tedy rovno 31 bodům.

4.6.2 Skóre uživatelské přívětivosti GraphQL

Stejný postup vyhodnocení skóre přívětivosti byl proveden na přístupu GraphQL. Doba vyhotovení se v případě realizace s využitím GraphQL pohybovala mezi 5-15 minutami pro dotaz getClient a mutaci addClient a byla tedy ohodnocena 8 body. Mutace updateClient vyžadovala více času na přípravu a to 15-30 minut, naopak mutace deleteClient díky své nenáročnosti zabrala méně než 5 minut na realizaci. Celkové skóre ukazatele doby vyhotovení s vahou 0,375 všech GraphQL operací je 12.

Počtem napsaných znaků se GraphQL umístilo na 6 bodové hranici pro dotazy, respektive mutace getClient, addClient a updateClient s rozpětím 300-500 napsaných znaků pro každou operaci. Mutace deleteClient vyžadovala na realizaci pouze 100-300 napsaných znaků, tudíž byla ohodnocena 8 body. Celkové skóre s vahou 0,125 pro počet napsaných znaků GraphQL operací je 3,25.

Posledním ukazatelem chybějícím pro vyhodnocení skóre přívětivosti pro GraphQL je srozumitelnost syntaxe kódu. Tento ukazatel byl hodnocen nejvyšším bodovým ohodnocením, tedy vysokou srozumitelností, pro dotaz getClient a mutace addClient a

deleteClient. Mutace updateClient kvůli své mírně vyšší komplexnosti obdržela ohodnocení 8 bodů, tedy střední vyšší srozumitelnost. Celkové skóre ukazatele srozumitelnost syntaxe kódu pro GraphQL operace je 19 bodů.

Váhy a skóre je k dispozici k náhledu v tabulce číslo 11.

GraphQL operace	Doba vyhotovení		Počet napsaných znaků		Srozumitelnost syntaxe kódu	
Váha w	0,375		0,125		0,5	
	čas	body	počet	body	srozumitelnost	body
getClient	0-5 min.	10	Méně než 100	10	Vysoká	10
	5-15 min.	8	100 až 300	8	Střední vyšší	8
	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4
addClient	0-5 min.	10	Méně než 100	10	Vysoká	10
	5-15 min.	8	100 až 300	8	Střední vyšší	8
	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4
updateClient	0-5 min.	10	Méně než 100	10	Vysoká	10
	5-15 min.	8	100 až 300	8	Střední vyšší	8
	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4
deleteClient	0-5 min.	10	Méně než 100	10	Vysoká	10
	5-15 min.	8	100 až 300	8	Střední vyšší	8
	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4

Tabulka 11 Výpočet skóre přívětivosti pro GraphQL dotazy a mutace

Zdroj: autor

Vážený průměr určující celkovou uživatelskou přívětivost pro operace čtení, vytvoření, aktualizaci a smazání dat je v případě hodnotícího, který nemá žádné přechozí zkušenosti s GraphQL přístupem, roven 34,25 bodům.

5 Výsledky a diskuse

5.1 Výsledky srovnávacího experimentu

Srovnávací experiment mezi přístupy pro tvorbu aplikačních rozhraní REST API a GraphQL proběhl ve dvou oblastech. První z nich byla rychlost daného přístupu, která byla definována jako rychlost vyřízení požadavku od momentu odeslání operace ze strany klienta do momentu obdržení odpovědi ze strany serveru. Druhou z měřených oblastí byla datová zátěž vyvíjená na síť, kterou každý z přístupů pro realizaci požadavků vyžaduje.

Metrika doba vyřízení požadavku, stejně jako metrika velikost odpovědi ze serveru, byla měřena na 4 rozdílných operacích nad daty uloženými v MongoDB databázi. Tyto operace byly čtení dat z databáze, vytvoření nových dat v databázi, aktualizace existujících dat v databázi a mazání existujících dat z databáze. Každá z těchto operací byla za účely získání relevantního vzorku pro vyhodnocení experimentu opakována v 10 pokusech. Kromě toho byla každá operace navíc realizována ve 3 provedeních s lišícím se počtem opakování každého z požadavků za účelem odhalení trendů, kdy je každý z přístupů vystaven výzvě čelit zpracování většího množství dat. Prvním provedením bylo odeslání požadavku jedenkrát, druhým provedením bylo odeslání požadavku stokrát a v rámci třetího provedení došlo k opakování požadavku tisíckrát.

5.1.1 Doba vyřízení požadavku

Doba vyřízení požadavku ukázala, že REST API ve většině operací a jejich variantách provedení (1, 100 a 1000 opakování) vykazuje lepší výkonnost než operace psané v alternativě v podobě GraphQL.

Závěr, že REST API vykazuje lepší výsledky než GraphQL, se potvrdil u všech provedení (1, 100 i 1000 opakování) v operacích vytváření nových dat a aktualizaci stávajících dat. Konkrétně byl při vytváření nových dat na serveru REST v průměru o 51 % procent rychlejší při opakování požadavku jedenkrát, o 24 % rychlejší při opakování požadavku stokrát a o 42 % rychlejší při opakování požadavku tisíckrát než tomu bylo u GraphQL.

Podobně jako při vytváření nových dat byl i při aktualizaci stávajících dat REST rychlejší v průměru o 37 % při jednom opakování požadavku, o 32 % rychlejší při opakování požadavku stokrát a o 36 % rychlejší při opakování požadavku tisíckrát.

REST předčil GraphQL i v provedení operací na čtení a výmaz dat a to ve srovnání se 100 a 1000 opakováními požadavku. V rámci srovnání čtení existujících dat ze serveru REST vykázal rychlejší zpracování v průměru o 13 % při opakování sto požadavků a o 31 % rychlejší zpracování při opakování požadavku tisíckrát. Naopak GraphQL předčilo REST při čtení dat ze serveru v rámci jednoho opakování volání požadavku, kdy bylo o 75 % rychlejší než zpracování metodou GET v rámci REST rozhraní.

Ke srovnatelným výsledkům došlo i srovnání v rámci výmazu dat ze serveru. Tato jediná operace byla prováděna pouze s jedním opakováním pro zachování jednoduchosti celé operace bez nutnosti vytváření logiky pro výmaz tolika záznamů ze serveru, kolik je navoleno uživatelem. GraphQL v případě výmazu a jednoho opakování požadavku vykázalo o 32 % rychlejší zpracování, než tomu bylo v případě zpracování přes REST API.

Výsledky srovnání doby vyřízení požadavků tedy lze shrnout ve prospěch REST API, kdy v rámci dvou operací ze čtyř ve všech jejich provedeních vykázal lepší výkon pokud jde o rychlost vyřízení požadavku. Ve zbylých dvou metodách REST API rychleji zpracovávalo data v případě sto a tisíc opakování požadavku, tj. v případě většího množství dat. GraphQL naopak předčilo REST v rychlosti zpracování malých, kompaktních požadavků, kdy docházelo pouze k jednomu opakování.

Výsledky srovnání se shodují se závěry většiny dostupných prací uvedených v kapitole 1.9, kde autoři třech z pěti uvedených prací došli k závěru, který v otázce výkonnosti a rychlosti favorizuje REST oproti GraphQL. Autoři zmíněných prací nicméně dodávají, že i přes závěry doložené v rámci jejich prací je potřeba přihlédnout k individuálním potřebám a konkrétnímu případu užití, na základě kterého je potřeba učinit rozhodnutí o využití obou z přístupů. Ke stejnému závěru v rámci měření výsledků srovnávacího experimentu došel i hodnotící této diplomové práce.

Návrh srovnávacího experimentu na měření rychlosti spočíval v měření rychlosti pro čtení, zápis, aktualizaci a mazání dat na serveru. Kromě vyhodnocení ve prospěch REST API autor ovšem v rámci měření a zhodnocení výsledků došel k závěru, že návrh provedení srovnávacího experimentu mírně favorizoval REST. Autor se domnívá, že příčinou této výhody může být využívání REST API v prostředí, které dostatečně nevystavuje jeho slabé

stránky, kterými jsou overfetching anebo naopak underfetching dat. Za účelem ověření této hypotézy autor připravil dodatečné srovnání.

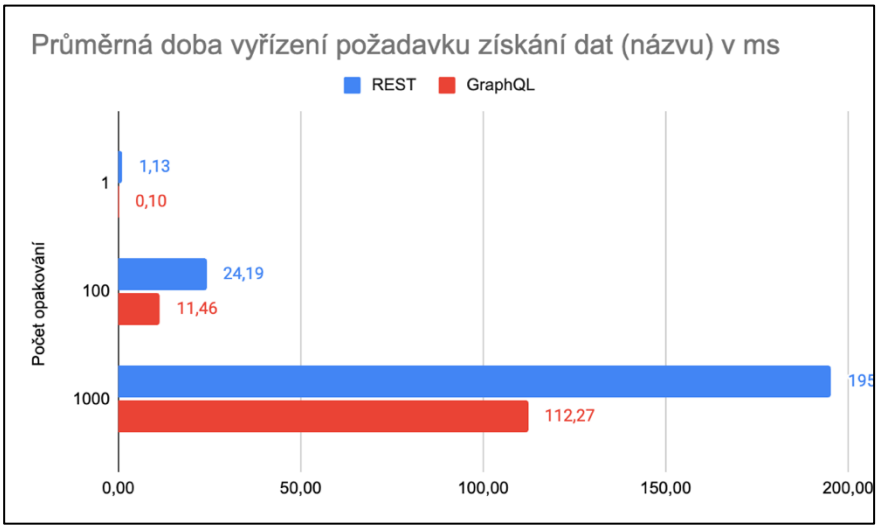
Srovnání se týkalo operace čtení dat, kdy bylo ze strany klienta vyžadováno pouze omezené množství datových proměnných z celého objektu. Objektem byla opět instance třídy Client a dodatečný pokus měl za úkol získat název klienta.

Srovnání výkonnosti z hlediska získání pouze jména klienta s existujícím REST API endpointem clients/:id a s nově vytvořeným GraphQL dotazem getClientName ukazuje, že v případě potřeby získat specifickou podmnožinu atributů jednoho objektu GraphQL vykazuje lepší výsledky oproti REST API. V případě 1 opakování požadavku jde o přibližně desetkrát rychlejší zpracování, v případě 100 opakování je potom GraphQL rychlejší přibližně o 52 % a v případě 1000 opakování potom o 42 %. Souhrnné výsledky jsou k dispozici v tabulce číslo 12 a jejich vizualizace na grafu číslo 9.

HTTP Metoda / GraphQL operace	**Endpoint / mutace**	**Počet opakování**	**Aritmetický průměr**	**Směrodatná odchylka**
GET	/clients/:id	1	1,13	0,58
GET	/clients/:id	100	24,19	1,33
GET	/clients/:id	1000	195,41	11,50
Query	getClientName	1	0,10	0,06
Query	getClientName	100	11,46	2,10
Query	getClientName	1000	112,27	21,52

Tabulka 12 Srovnání rychlosti požadavku získání dat (názvu) v milisekundách

Zdroj: autor



Graf 9 Srovnání rychlosti požadavku získání dat (názvu) v milisekundách

Zdroj: autor

Na základě provedení a vyhodnocení dodatečného srovnání, které více využívá potenciál a přednosti GraphQL v podobě schopnosti doptávat se na specifické proměnné objektu bez nutnosti tvořit nový endpoint, jako je tomu u REST API, lze konstatovat, že doporučení pro využití každého z přístupů je silně závislé od případu použití a prostředí, v rámci kterého je přístup implementován. V systémech, u kterých je potřeba flexibilně přistupovat k získávání dat ze strany serveru, lze více těžit z výhod GraphQL, naopak u systémů s relativně fixními požadavky na operace s daty je favorizován spíš REST API přístup. Pro selekci správného přístupu je doporučeno podstoupení dalšího výzkumu a srovnávacích experimentů.

5.1.2 Velikost odpovědi ze serveru

Pokusy zaměřené na měření velikost odpovědi ze serveru ukázaly, že REST API lze považovat za úspornější přístup, pokud jde o velikost vyřízeného požadavku v Kb zaslaného ze strany serveru klientovi.

V případě čtení dat ze serveru se REST API ukázalo o 20 % úspornější. Srovnatelně lze shrnout i výsledky pro vytvoření nových dat na serveru a aktualizaci existujících dat s menšími serverovými odpověďmi o 22 %, respektive 23 %. V případě výmazu dat šlo dokonce o rozdíl v úspornosti o celých 83 % ve prospěch REST API ve srovnání s GraphQL. Naměřené hodnoty se neztotožňují se závěry již dostupného výzkumu.

Podobně jako v případě srovnání rychlosti i v případě datové velikosti lze konstatovat, že rozhodnutí o zvolení přístupu REST API a GraphQL závisí především na požadavcích implementujícího subjektu.

5.2 Výsledky srovnání uživatelské přívětivosti

Výsledky vyhodnocení přívětivosti obou z přístupů ukázaly, že v rámci váhového ohodnocení ukazatelů, jimiž jsou doba vyhotovení, počet napsaných znaků a srozumitelnost syntaxe kódu, uživatelem, který upřednostňuje srozumitelnost kódu (váha 0,5) oproti době vyhotovení (váha 0,375) a počtu napsaných znaků (váha 0,125), je GraphQL uživatelsky přívětivějším přístupem. Jeho celkové skóre přívětivosti je rovno 34,25 bodům oproti 31 bodům v případě REST API.

Výsledná hodnota skóre přívětivosti, která je vypočítána jako vážený průměr skóre přiřazeného všem 3 ukazatelům, je nicméně závislá na váze, kterou hodnotící uživatel ukazatelům přiřadí.

Pokud by uživatel oproti srozumitelnosti syntaxe kódu preferoval spíše čas potřebný k vyhotovení požadavku, bude potřeba zvýšit skóre tohoto ukazatele, což v důsledku ovlivní skóre přívětivosti obou z přístupů. Příklad matice s vahami, které upřednostňují dobu vyhotovení před počtem znaků jako druhého nejdůležitějšího ukazatele a srozumitelnosti syntaxe kódu jako třetího nejdůležitějšího ukazatele je uveden v tabulce číslo 13 pro REST API a tabulce číslo 14 pro GraphQL. Na základě přiřazených vah pro dobu vyhotovení (váha 0,5), počet napsaných znaků (váha 0,125) a srozumitelnost kódu (0,375) lze konstatovat, že GraphQL s uživatelským skóre 33,5 poskytuje lepší míru uživatelské přívětivosti než REST API se skóre 30,25 bodů.

Váhy a skóre je k dispozici k náhledu v tabulce číslo 13.

HTTP metoda / GraphQL operace	Doba vyhotovení		Počet napsaných znaků		Srozumitelnost syntaxe kódu	
Váha w	0,5		0,125		0,375	
	čas	body	počet	body	srozumitelnost	body
GET	0-5 min.	10	Méně než 100	10	Vysoká	10
	5-15 min.	8	100 až 300	8	Střední vyšší	8
	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4
POST	0-5 min.	10	Méně než 100	10	Vysoká	10
	5-15 min.	8	100 až 300	8	Střední vyšší	8
	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4
PATCH	0-5 min.	10	Méně než 100	10	Vysoká	10
	5-15 min.	8	100 až 300	8	Střední vyšší	8
	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4
DELETE	0-5 min.	10	Méně než 100	10	Vysoká	10
	5-15 min.	8	100 až 300	8	Střední vyšší	8
	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4

Tabulka 13 Výpočet skóre přívětivosti pro REST API metody – doba vyhotovení

Zdroj: autor

GraphQL operace	Doba vyhotovení		Počet napsaných znaků		Srozumitelnost syntaxe kódu	
Váha w	0,5		0,125		0,375	
	čas	body	počet	body	srozumitelnost	body
getClient	0-5 min.	10	Méně než 100	10	Vysoká	10
	5-15 min.	8	100 až 300	8	Střední vyšší	8
	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4
addClient	0-5 min.	10	Méně než 100	10	Vysoká	10
	5-15 min.	8	100 až 300	8	Střední vyšší	8
	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4
updateClient	0-5 min.	10	Méně než 100	10	Vysoká	10
	5-15 min.	8	100 až 300	8	Střední vyšší	8
	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4
deleteClient	0-5 min.	10	Méně než 100	10	Vysoká	10
	5-15 min.	8	100 až 300	8	Střední vyšší	8
	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4

Tabulka 14 Výpočet skóre přívětivosti pro GraphQL dotazy a mutace – doba vyhotovení

Zdroj: autor

Pokud budeme uvažovat uživatele, který preferuje ukazatel počet napsaných znaků, kterému přiřadí váhu 0,5, oproti ukazateli srozumitelnost syntaxe kódu s přiřazeným skóre 0,125 a ukazateli doba vyhotovení s přiřazeným skóre 0,375, lze pozorovat, že skóre přívětivosti v případě GraphQL je 29,75, zatímco v případě REST API 28,75. Hodnoty jsou k náhledu v tabulce číslo 15 pro REST API a tabulce číslo 16 pro GraphQL.

HTTP metoda / GraphQL operace	Doba vyhotovení		Počet napsaných znaků		Srozumitelnost syntaxe kódu	
Váha w	0,375		0,5		0,125	
	čas	body	počet	body	srozumitelnost	body
GET	0-5 min.	10	Méně než 100	10	Vysoká	10
	5-15 min.	8	100 až 300	8	Střední vyšší	8
	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4

POST	0-5 min.	10	Méně než 100	10	Vysoká	10
	5-15 min.	8	100 až 300	8	Střední vyšší	8
	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4
PATCH	0-5 min.	10	Méně než 100	10	Vysoká	10
	5-15 min.	8	100 až 300	8	Střední vyšší	8
	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4
DELETE	0-5 min.	10	Méně než 100	10	Vysoká	10
	5-15 min.	8	100 až 300	8	Střední vyšší	8
	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4

Tabulka 15 Výpočet skóre přívětivosti pro REST API metody - počet znaků

Zdroj: autor

GraphQL operace	Doba vyhotovení		Počet napsaných znaků		Srozumitelnost syntaxe kódu	
Váha w	0,375		0,5		0,125	
	čas	body	počet	body	srozumitelnost	body
getClient	0-5 min.	10	Méně než 100	10	Vysoká	10
	5-15 min.	8	100 až 300	8	Střední vyšší	8
	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4
addClient	0-5 min.	10	Méně než 100	10	Vysoká	10
	5-15 min.	8	100 až 300	8	Střední vyšší	8
	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4
updateClient	0-5 min.	10	Méně než 100	10	Vysoká	10
	5-15 min.	8	100 až 300	8	Střední vyšší	8
	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4
deleteClient	0-5 min.	10	Méně než 100	10	Vysoká	10
	5-15 min.	8	100 až 300	8	Střední vyšší	8
	15-30 min.	6	300 až 500	6	Střední nižší	6
	30 a více min.	4	Více než 500	4	Nízká	4

Tabulka 16 Výpočet skóre přívětivosti pro GraphQL dotazy a mutace - počet znaků

Zdroj: autor

Na základě předložených dat lze konstatovat, že ve srovnání uživatelské přívětivosti realizovaných v rámci diplomové práce GraphQL implementace předčí implementaci využívající REST API pokud uvažujeme ukazatele dobu vyhotovení, počet napsaných znaků a srozumitelnost syntaxe kódu společně s přiřazenými vahami. Výsledky zkoumání uživatelské přívětivosti se tedy shodují s výsledky již dostupných prací uvedených v kapitole 1.9 diplomové práce. Výsledné hodnoty skóre uživatelské přívětivosti nicméně podléhají vlivu vah přiřazeným jednotlivým ukazatelům. Pro zpřístupnění zdrojů pro další výzkum je v rámci přílohy číslo 1 přiložen soubor, který hodnotícím uživatelům usnadní vložení preferovaných vah k jednotlivým ukazatelům a volbu konkrétních naměřených hodnot, na základě kterých je uskutečněn výpočet skóre uživatelské přívětivosti.

6 Závěr

Diplomová práce se věnovala problematice tvorby aplikačních programovacích rozhraní (API) a srovnání soudobých populárních přístupů pro jeho tvorbu v rámci prostředí webových aplikací, jimiž jsou REST API a GraphQL. Přestože REST je technologický přístup představený ji na přelomu tisíciletí, stále je považován za základní stavební kámen v komunikaci mezi klientem a serverem, kdy jeho principy využívá až 89 % v současnosti provozovaných webových služeb. Naproti tomu GraphQL je přístup, který byl zpřístupněn v podobě open source v roce 2015. Od té doby si získal velké množství příznivců, nicméně v rámci podílu na tvorbě webových API zdaleka nepředčil REST API. Diplomová práce zkoumala výkonnostní rozdíly obou přístupů společně se zhodnocením uživatelské přívětivosti při tvorbě webových služeb využívající jak REST, tak GraphQL.

Teoretická část práce se zaměřila na vymezení základních pojmů především v oblasti vývoje a provozu webových služeb. Na představení základních principů bylo navázáno popisem REST API a GraphQL přístupů z hlediska jejich historie a zavedení, přes základní komunikační mechanismy až po principy, které musí REST architektura a GraphQL splňovat. V rámci teoretické části práce došlo taktéž k vymezení technologických pojmů, které byly využity v rámci tvorby aplikačního prostředí za účelem realizace srovnávacího experimentu. Kromě prostředí pro experiment byla taktéž představena metodika samotného experimentu, který se zaměřil na oblast výkonu z hlediska doby vyřízení požadavku od jeho odeslání ze strany klienta, do obdržení odpovědi ze strany serveru. Kromě rychlosti šlo o nezbytnou velikost datového přenosu pro vyřízení požadavku ze strany serveru na stranu klienta. Pro oba typy měření byly definovány konkrétní časové okamžiky sběru dat a formy, kterými budou data sbírána, aby bylo zachována jednotnost jak pro přístup REST API tak GraphQL. Mimo kvantitativní výzkum byly taktéž přístupy podrobeny zkoumání věnující se uživatelské přívětivosti. Za účelem vyhodnocení uživatelské přívětivosti byl v rámci teoretické části práce definován termín skóre uživatelské přívětivosti včetně jeho výpočtu přes vážený průměr.

Praktická část práce se věnovala popisu přípravy prostředí pro realizaci srovnávacích experimentů. Tato příprava spočívala nejprve v definování náležitostí uživatelského rozhraní aplikace. Hlavními komponentami webové aplikace jsou seznamy zobrazující výskyty třídy Client, kde v rámci oddělených seznamů pro REST a GraphQL lze realizovat

operace nad daty využívající danou technologii. Operace, které byly v rámci práce zkoumány, byly čtení dat, vytvoření nových dat, aktualizace dat a mazání dat. Po představení návrhů uživatelského rozhraní byla doložena dokumentace k tvorbě webové aplikace jakožto prostředí, v rámci kterého byl srovnávací experiment proveden. Dokumentace pokryla jak serverovou část aplikace zastřešující nastavení databáze, její propojení s aplikací, nastavení datového modelu Client, tak klientskou část aplikace věnující se především získávání dat ze strany serveru přes REST API enpdointy a GraphQL operace za účelem jejich vykreslení v rámci grafického uživatelského rozhraní.

Po dokončení prostředí pro realizaci experimentu došlo k realizaci srovnávacích pokusů jako takových. Jak již bylo zmíněno, všechny operace probíhaly nad výskyty (objekty) třídy Client. Každá z operací pro čtení, vytvoření a aktualizaci byla realizována pro každý z přístupů REST a GraphQL ve 3 provedeních, pokaždé s jiným počtem opakování volání daného požadavku. První provedení obsahovalo pouze jeden požadavek, druhé provedení opakovalo realizaci požadavku stokrát a třetí provedení tisíckrát. Odlišná provedení měla za účel vnést do experimentu větší variabilitu a prostor pro sledování výkonnosti v případě zvětšující se zátěže. Požadavek na výmaz dat byl na rozdíl od předchozích zmíněných operací uskutečněn pouze v rámci jednoho provedení – a to výmazu jednoho záznamu. Výsledky srovnávacího experimentu ukázali, že REST API je z hlediska rychlosti vyřízení požadavků i z hlediska menšího datového přenosu vhodnějším kandidátem pro tvorbu webových API, nicméně je potřeba upozornit na fakt, že výsledky experimentu jsou silně závislé na podmínkách a předpokladech prostředí, v rámci kterého experiment probíhá. V rámci výsledků a diskuze bylo ověřeno, že v případě jiného případu užití, kdy je požadována operace pouze nad podmnožinou proměnných v rámci jednoho objektu, je GraphQL vhodnějším přístupem jak z hlediska rychlosti vyřízení požadavku, tak z hlediska velikosti datového přenosu. Autor závěrem konstatuje, že každý subjekt, který provádí srovnání či implementuje webové API by tedy měl kriticky rozhodnout o hlavním případu užití dané služby a na tomto základě učinit rozhodnutí o zvolení konkrétního přístupu.

Na kvantitativní srovnání bylo navázáno srovnáním uživatelské přívětivosti spojené s tvorbou aplikačního rozhraní s využitím každého z přístupů. Na základě metodiky definované v teoretické části práce bylo s využitím váženého průměru přiřazeno skóre přívětivosti. Z obou přístupů bylo vyhodnoceno jako uživatelky přívětivější GraphQL,

nicméně v rámci diskuse a výsledků bylo zdůrazněno, že se celkový výsledek uživatelské přívětivosti do značné míry odvíjí od individuálních preferencí hodnotícího uživatele (programátora).

Diplomová práce poskytla komplexní souhrn aktuálně nejvyužívanějších technologických přístupů pro tvorbu webových API, kterými jsou REST a GraphQL. V rámci srovnání a připojené diskuse bylo zjištěno, že REST API poskytlo lepší výsledky týkající se výkonu, zatímco GraphQL nabízí lepší úroveň uživatelské přívětivosti. Obě části zkoumání nicméně do značné míry podléhají individuálním potřebám uživatelů a podmínkám prostředí, v rámci kterého jsou realizovány.

7 Seznam použitých zdrojů

1. Booth, David. Web Services Architecture. *W3C.* [Online] 11. 02 2004. [Citace: 19. 11 2023.] https://www.w3.org/TR/ws-arch/#id2260892.

2. Brown, Ethan. *Web Development with Node and Express: Leveraging the JavaScript Stack.* místo neznámé : O'Reilly Media, 2014. 978-1491949306.

3. James Snell, Ken MacLeod, Doug Tidwell, Pavel Kulchenko. *Programming Web Services With Soap.* místo neznámé : Oreilly & Associates Inc, 2001. 978-0596000950.

4. What is a web service? *IBM.com.* [Online] 03. 05 2021. https://www.ibm.com/docs/en/cics-ts/5.1?topic=services-what-is-web-service.

5. Thangenthiran, Rajeenthiran. What are Web Services ? *Medium.* [Online] 22. 08 2021. [Citace: 19. 11 2023.] https://medium.com/@rajee781996/web-services-f0db1eca7c6b.

6. Brenda Jin, Saurabh Sahni, Amir Shevat. *Designing Web APIs: Building APIs That Developers Love.* místo neznámé : O'Reilly Media, 2018. 978-1492026921.

7. What is a REST API? *IBM.com.* [Online] IBM. [Citace: 12. 11 2023.] https://www.ibm.com/topics/rest-apis.

8. Fielding, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures.* UNIVERSITY OF CALIFORNIA, IRVINE : autor neznámý, 2000.

9. Simpson, J. Top Architectural Styles for APIs in 2023. *Nordic APIs.* [Online] Nordic APIs AB, 9. 2 2023. [Citace: 23. 11 2023.] https://nordicapis.com/top-architectural-styles-for-apis-in-2023/.

10. Lau, Grace. Web Service: What Is a REST API and How Does It Work? . *Nordic APIs.* [Online] Nordic APIs AB , 19. 1 2022. [Citace: 26. 11 2023.] https://nordicapis.com/web-service-what-is-a-rest-api-and-how-does-it-work/.

11. Wieruch, Robin. Why GraphQL: Advantages and Disadvantages. *Robinwieruch.* [Online] © Robin Wieruch, 3. 7 2018. [Citace: 25. 11 2023.] https://www.robinwieruch.de/why-graphql-advantages-disadvantages-alternatives/.

12. U, Santhosh Adiga. Evolution of API Architecture. *Medium.* [Online] 11. 3 2023. [Citace: 26. 11 2023.] https://santhosh-adiga-u.medium.com/evolution-of-api-architecture-228624472d79.

13. Doglio, Fernando. *Pro REST API Development with Node.js.* místo neznámé : Apress, 2015.

14. Tobias Andersso, Håkan Reinholdsson. *REST API vs GraphQL - A literature and experimental study.* místo neznámé : Faculty of Natural Science, 2021.

15. Gopalakrishnan, Ashita. GraphQL vs REST API. *Bejamas.* [Online] Bejamas.io, 3. 11 2021. [Citace: 27. 11 2023.] https://bejamas.io/blog/graphql-vs-rest-api/.

16. What is the GraphQL Foundation? *Graphql.org.* [Online] The GraphQL Foundation. [Citace: 27. 11 2023.] https://graphql.org/foundation/.

17. Wieruch, Robin. *The Road to GraphQL: Your journey to master pragmatic GraphQL in JavaScript with React.js and Node.js.* 2018. 978-1730853937.

18. Big Picture (Architecture). *How to GraphQL.* [Online] [Citace: 28. 11 2023.] https://www.howtographql.com/basics/3-big-picture.

19. Chanaka Fernando Chanaka Fernando Chanaka Fernando Book Author Writes about Microservices, APIs, and Integration. Author of “Designing Microservices Platforms with NATS” and "Solution Architecture Patterns for Enterprise" 2.2K Followers Follow. GraphQL based solution architecture patterns. *Medium.* [Online] Medium, 9. 12 2019. [Citace: 11. 27 2023.] https://chanakaudaya.medium.com/graphql-based-solution-architecture-patterns-8905de6ff87e.

20. Dashora, Saurabh. The 3 Types of GraphQL Architectural Patterns. *Progressive Coder.* [Online] 26. 1 2022. [Citace: 2023. 10 27.] https://progressivecoder.com/the-3-types-of-graphql-architectural-patterns/.

21. GraphQL. *GraphQL.org.* [Online] 2018. [Citace: 28. 11 2023.] https://spec.graphql.org/June2018/#sec-Overview.

22. Sufiyan, Taha. What is Node.js: A Comprehensive Guide. *SimpliLearn.* [Online] Simplilearn Solutions, 16. 5 2023. [Citace: 28. 11 2023.] https://www.simplilearn.com/tutorials/nodejs-tutorial/what-is-nodejs#what_is_nodejs.

23. Sharma, Anubhav. Express JS Tutorial. *SimpliLearn.* [Online] SimpliLearn Solutions, 5. 10 2023. [Citace: 28. 11 2023.] https://www.simplilearn.com/tutorials/nodejs-tutorial/what-is-express-js#what_is_express_js.

24. Why Use MongoDB and When to Use It? *MongoDB.* [Online] MongoDB, Inc. [Citace: 28. 11 2023.] https://www.mongodb.com/why-use-mongodb.

25. MongoDB Atlas Tutorial. *MongoDB.* [Online] MongoDB, Inc. [Citace: 28. 11 2023.] https://www.mongodb.com/basics/mongodb-atlas-tutorial.

26. Roberts, Sienna. What is MongoDB Compass? *The Knowledge Academy.* [Online] The Knowledge Academy Ltd, 16. 9 2023. [Citace: 28. 11 2023.] https://www.theknowledgeacademy.com/blog/mongodb-compass/.

27. Kataria, Saransh. Creating a GraphQL API with Apollo Server. *Medium.* [Online] 4. 12 2020. [Citace: 29. 11 2023.] https://www.linkedin.com/pulse/creating-graphql-api-apollo-server-saransh-kataria/.

28. Rostami, Mohammad. Exploring Apollo Client in React: A Comprehensive Guide. *LinkedIn.* [Online] 23. 9 2023. [Citace: 29. 11 2023.] https://www.linkedin.com/pulse/exploring-apollo-client-react-comprehensive-guide-mohammad-rostami/.

29. Alex Banks, Eve Porcello. *Learning React: Modern Patterns for Developing React Apps.* místo neznámé : O'Reilly Media, 2020. 978-1492051725 .

30. Morgan, Joe. How To Create React Elements with JSX. *DigitalOcean.* [Online] DigitalOcean, LLC. [Citace: 26. 11 2023.] https://www.digitalocean.com/community/tutorials/how-to-create-react-elements-with-jsx.

31. What is web performance? *MDN Web Docs.* [Online] [Citace: 28. 11 2023.] https://developer.mozilla.org/en-US/docs/Learn/Performance/What_is_web_performance.

32. Recommended Web Performance Timings: How long is too long? *MDN Web Docs.* [Online] [Citace: 11. 28 2023.] https://developer.mozilla.org/en-US/docs/Web/Performance/How_long_is_too_long.

33. API Response Times: A Quick Guide to Improving Performance. *Prismic.* [Online] 28. 6 2023. [Citace: 29. 11 2023.] https://prismic.io/blog/api-response-times.

34. Performance: now() method. *MDN Web Docs.* [Online] [Citace: 29. 11 2023.] https://developer.mozilla.org/en-US/docs/Web/API/Performance/now.

35. *How fast GraphQL is compared to REST APIs.* Oggier, Camille. 2020.

36. *A COMPARATIVE STUDY BETWEEN GRAPH-QL& RESTFUL SERVICES IN API MANAGEMENT OF STATELESS ARCHITECTURES.* Sayan Guha, Shreyasi Majumder. 2, 2020, Sv. 11.

37. *Evaluating GraphQL and REST API Services Performance in a Massive and Intensive Accessible Information System.* Armin Lawi, Benny L. E. Panggabean, Takaichi Yoshida. 138, 2021, Sv. 10.

38. *Comparative Analysis of Rest and GraphQL Technology on Nodejs-Based Api Development.* Gede Susrama Mas Diyasa, Gideon Setya Budiwitjaksono, Hafidz Amarul Ma'rufi, Ilham Ade Widya Sampurno. 2020. Sv. 2021.
39. *Comparison of REST and GraphQL Interfaces for OPC UA.* Ala-Laurinaho, Riku, a další. 5, 2022, Sv. 11.
40. Gleison Brito, Marco Tulio Valente. *REST vs GraphQL: A Controlled Experiment.* 2020.

8 Seznam obrázků, tabulek a grafů

8.1 Seznam obrázků

8.2 Seznam tabulek

8.3 Seznam grafů

8.4 Seznam ukázek kódu

9 Přílohy

9.1 Příloha č. 1 – soubor Kalkulátor uživatelské přívětivosti

URL: https://github.com/mattpovolny/SkoreUzivatelskePrivetivosti

9.2 Příloha č. 2 – GraphQL schéma

```
//Mongoose models
const Project = require('../models/Project');
const Client = require('../models/Client');


const {
    GraphQLObjectType,
    GraphQLID,
    GraphQLString,
    GraphQLSchema,
    GraphQLList,
    GraphQLNonNull,
    GraphQLEnumType,
} = require('graphql');

// Client Type
const ClientType = new GraphQLObjectType({
    name: 'Client',
    fields: () => ({
        id: { type: GraphQLID },
        name: { type: GraphQLString },
        email: { type: GraphQLString },
        phone: { type: GraphQLString },
    }),
});

const RootQuery = new GraphQLObjectType({
    name: 'RootQueryType',
    fields: {
        clients: {
            type: new GraphQLList(ClientType),
            resolve(parent, args) {
                return Client.find();
            }
        },
        client: {
            type: ClientType,
            args: { id: { type: GraphQLID } },
```

```
            resolve(parent, args) {
                return Client.findById(args.id);
            }
        }
    }
});

// Mutations
const mutation = new GraphQLObjectType({
    name: 'Mutation',
    fields: {
        // Add a client
        addClient: {
            type: ClientType,
            args: {
                name: { type: GraphQLNonNull(GraphQLString) },
                email: { type: GraphQLNonNull(GraphQLString) },
                phone: { type: GraphQLNonNull(GraphQLString) },
            },
            resolve(parent, args) {
                const client = new Client({
                    name: args.name,
                    email: args.email,
                    phone: args.phone,
                });
                return client.save();
            },
        },
        // Delete a client
        deleteClient: {
            type: ClientType,
            args: {
                id: { type: GraphQLNonNull(GraphQLID) },
            },
            resolve(parent, args) {
                Project.deleteOne({clientId: args.id }).exec();
                return Client.findByIdAndDelete(args.id);
            }
        },
        // Update a client
                updateClient: {
                    type: ClientType,
                    args: {
                        id: { type: GraphQLNonNull(GraphQLID) },
                        name: { type: GraphQLString },
                        email: { type: GraphQLString },
                        phone: { type: GraphQLString },
                    },
                    resolve(parent, args) {
```

```
                        return Client.findByIdAndUpdate(
                            args.id,
                            {
                                $set: {
                                    name: args.name,
                                    email: args.email,
                                    phone: args.phone,
                                },
                            },
                            { new: true }
                        );
                    }
                },

});

module.exports = new GraphQLSchema({
    query: RootQuery,
    mutation
});
```

9.3 Příloha č. 3 – obsah App.js souboru

```
import { BrowserRouter as Router, Route, Routes} from 'react-router-dom';
import Header from "./components/Header";
import { ApolloProvider, ApolloClient, InMemoryCache }
from '@apollo/client';
import Home from './pages/Home';
import NotFound from './pages/NotFound';
import Project from './pages/Project';
import CreateClient from './pages/CreateClient';
import UpdateClient from './pages/UpdateClient';
import ReadClient from './pages/ReadClient';



const cache = new InMemoryCache({
  typePolicies: {
    Query: {
      fields: {
        clients: {
          merge(existing, incoming) {
            return incoming;
          }
        },
        projects: {
          merge(existing, incoming) {
```

```
            return incoming;
          }
        }
      }
    }
  }
})


const client = new ApolloClient({
  uri:'http://localhost:4000/graphql',
  cache, //: new InMemoryCache(),
});

function App() {
  return (
    <>
    <ApolloProvider client={client}>
      <Router>
    <Header/>
      <div className="container">
        <Routes>
          <Route path='/' element={<Home/>} />
          <Route path='/projects/:id' element={<Project/>} />
          <Route path='/createclient' element={<CreateClient/>} />
          <Route path='/updateclient/:id' element={<UpdateClient/>} />
          <Route path='/readclient/:id' element={<ReadClient/>} />
          <Route path='*' element={<NotFound/>} />
        </Routes>
      </div>
      </Router>
      </ApolloProvider>
    </>
  );
}

export default App;
```

9.4 Příloha č. 4 – Obsah souboru CreateClient.jsx

```
import React, { useState } from 'react';
import { Link, useNavigate } from 'react-router-dom';
import axios from "axios";
```

```
function CreateClient() {
    var t0
    var t1
    var responseTime
    const [values, setValues] = useState({
        name: "",
        email: "",
        phone: "",
    })
    const navigate = useNavigate();
    const handleSubmit = (event) => {
        t0 = performance.now();
        event.preventDefault();
        axios.post('http://localhost:4000/clients', values)
            .then(res => {
                console.log(res);
                t1 = performance.now();
                responseTime = (t1 - t0);
                console.log("Call to create a client took " + (t1 - t0) + "
milliseconds.")
            })
            .catch(err => console.log(err))
    }

    return (
        <div>
            <div className='d-flex w-100 vh-100 justify-content-center align-
items-center bg-light'>
                <div className='w-50 border bg-white shadow px-5 pt-3 pb-5
rounded'>
                    <h1>Add a User</h1>
                    <form onSubmit={handleSubmit}>
                        <div className='mb-2'>
                            <label htmlFor='name'>Name:</label>
                            <input type='text' name="name" className='form-
control' placeholder="Enter Name"
                                onChange={e => setValues({ ...values, name:
e.target.value })} />
                        </div>
                        <div className='mb-2'>
                            <label htmlFor='email'>Email:</label>
                            <input type='email' name="email" className='form-
control' placeholder="Enter Email"
                                onChange={e => setValues({ ...values, email:
e.target.value })} />
                        </div>
                        <div className='mb-3'>
                            <label htmlFor='name'>Phone:</label>
```

```
                            <input type='text' name="phone" className='form-
control' placeholder="Enter Phone"
                                onChange={e => setValues({ ...values, phone:
e.target.value })} />
                        </div>
                        <button className='btn btn-success'>Submit</button>
                        <Link to="/" className="btn btn-primary ms-3">Back</Link>

                        <input type="text" className='form-control'
id='responseTime' value={responseTime}
                                                />

                    </form>

                </div>
            </div>


        </div>
    )
}

export default CreateClient
```

9.5 Příloha č. 5 – Zdrojový kód aplikace

URL: https://github.com/mattpovolny/Srovna-ni-REST-a-GraphQL_zdrojovy-ko-d

9.6 Příloha č. 6 – Získání dat

HTTP Metoda / GraphQL operace	Endpoint / Query	Počet opakování	pokus č. 1	pokus č. 2	pokus č. 3	pokus č. 4	pokus č. 5	pokus č. 6	pokus č. 7	pokus č. 8	pokus č. 9	pokus č. 10
GET	/client/:id	1	0,70	1,80	0,70	0,80	0,70	0,50	0,40	0,60	0,50	0,60
GET	/client/:id	100	19,00	25,70	19,10	20,20	21,30	19,70	21,90	18,90	20,30	20,50
GET	/client/:id	1000	226,60	165,70	126,60	154,10	148,40	140,00	129,50	131,50	133,90	158,70
Query	getClient	1	0,10	0,10	0,10	0,10	0,10	0,60	0,20	0,10	0,30	0,10
Query	getClient	100	28,50	31,50	7,50	14,26	30,60	24,00	30,00	26,70	21,00	24,30
Query	getClient	1000	270,00	207,00	261,00	225,00	210,00	216,00	195,00	192,00	225,00	204,00

9.7 Příloha č. 7 – Vytvoření dat

HTTP Metoda / GraphQL operace	Endpoint / mutace	Počet opakování	pokus č. 1	pokus č. 2	pokus č. 3	pokus č. 4	pokus č. 5	pokus č. 6	pokus č. 7	pokus č. 8	pokus č. 9	pokus č. 10
POST	/clients	1	1,199999928	0,8999999762	1,400000095	2,5	1,100000024	1,5	0,7999999523	1	1	1,400000095
POST	/clients	100	23,89999998	21,29999995	23	30,89999998	22,39999998	24,70000005	24,30000007	23,10000002	20,5	24,60000002
POST	/clients	1000	172,1	158,6	160,1	150,7	157,5	159	155,9	156,8	167,6999999	153,9000001
Mutace	addClient	1	4,600000024	1,899999976	1,700000048	2,600000024	1,100000024	1	9,399999976	1,200000048	1,600000024	0,8999999762
Mutace	addClient	100	29,20000005	29,80000007	26,69999993	48,29999995	25,60000002	30,30000007	30,5999999	32,4000001	34	25,5
Mutace	addClient	1000	280,8	288,6	337,5	314,5	327,4000001	252,6	248,9	222,1999999	239,3000001	239,8

9.8 Příloha č. 8 – Aktualizace dat

HTTP Metoda / GraphQL operace	Endpoint / mutace	Počet opakování	pokus č. 1	pokus č. 2	pokus č. 3	pokus č. 4	pokus č. 5	pokus č. 6	pokus č. 7	pokus č. 8	pokus č. 9	pokus č. 10
PATCH	/clients/:id	1	1	0,8000000715	0,6999999285	0,6000000238	1	0,6999999285	1	0,7999999523	0,7000000477	1
PATCH	/clients/:id	100	17,10000002	24,89999998	23,29999995	23,5999999	22,29999995	21,70000005	23	22,10000002	30	24,39999998
PATCH	/clients/:id	1000	198,6	144,1999999	149,4	144,9	142,9	152,5	149,8	174,6	166,9	142
Mutace	updateClient	1	2	1	1,299999952	1,100000024	1,200000048	1,5	1,399999976	1,100000024	1,100000024	1,399999976
Mutace	updateClient	100	30,60000002	35,40000001	34,60000002	35,70000005	38	37	35,19999993	33,30000007	32,20000005	28,20000005
Mutace	updateClient	1000	254,5	259,5	250,2	238,4	245,3	252,6	237,4	243,9	236,1999999	245,9000001

9.9 Příloha č. 9 – Výmaz dat

HTTP Metoda / GraphQL	Endpoint / mutace	Počet opakování	pokus č. 1	pokus č. 2	pokus č. 3	pokus č. 4	pokus č. 5	pokus č. 6	pokus č. 7	pokus č. 8	pokus č. 9	pokus č. 10

operace												
DELETE	/clients/:id	1	0,8999999762	1,200000048	1	2	1,600000024	3,100000024	1,800000072	1,600000024	1,699999928	1,5
Mutace	deleteClient	1	3,100000024	1	0,800000715	0,899999762	0,900000954	0,699999285	1	0,899999762	0,800000715	1,100000024