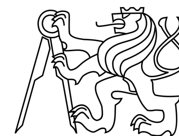
ESKÉ VYSOKÉ U ENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMA NÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název: GraphQL server
Student: Bc. Milan Blažek
Vedoucí: Ing. Martin Horský
Studijní program: Informatika
Studijní obor: Webové a softwarové inženýrství
Katedra: Katedra softwarového inženýrství
Platnost zadání: Do konce zimního semestru 2017/18

Pokyny pro vypracování

Výstupem práce bude funk ní GraphQL server podporující základní dotazy. Server bude mít webové rozhraní pro testování dotaz . Dále bude nabízet možnost m nit schéma dat.

1. Prozkoumejte a popište technologii GraphQL.
2. Porovnejte GraphQL s tradi ním p ístupem REST.
3. Navrhn te a implementujte GraphQL server s využitím vhodné technologie.
4. Pro server vytvo te jednoduché webové rozhraní pro dotazování.
5. Uve te p íklady využití serveru (Relay, React).

Seznam odborné literatury

Dodá vedoucí práce.

L.S.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
d kan

V Praze dne 24. února 2016

České vysoké učení technické v Praze

Fakulta informačních technologií

Katedra softwarového inženýrství

Diplomová práce

GraphQL server

Bc. Milan Blažek

Vedoucí práce: Ing. Martin Horský

10. května 2016

Poděkování

Touto cestou bych rád poděkoval vedoucímu práce Ing. Martinu Horskému za jeho odborné vedení, ochotu a trpělivost.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 10. května 2016 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2016 Milan Blažek. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Blažek, Milan. *GraphQL server.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstrakt

Práce popisuje technologii GraphQL od společnosti Facebook a vystavuje ji porovnání s tradičním přístupem REST. Implementační část práce využívá právě technologii GraphQL spolu s dalšími souvisejícími knihovnami.

Klíčová slova GraphQL, API, REST, schéma, JSON, komunikace.

Abstract

The thesis describes the GraphQL technology by Facebook. This technology is compared with traditional approach REST. Implementation of this work uses GraphQL technology along with other related libraries to demonstrate use in practice.

Keywords GraphQL, API, REST, schema, JSON, communication.

Obsah

Úvod **1**

1 Technologie GraphQL **3**

1.1 Základní specifikace jazyka 3
1.2 Ekosystém 7
1.3 Motivace ke vzniku 8
1.4 GraphQL knihovny a nástroje 8

2 Porovnání GraphQL s tradičním přístupem REST **11**

2.1 Počet dotazů 13
2.2 Tvar a velikost výstupu 14
2.3 Verzování 15
2.4 Modifikace dat 16
2.5 Mockování API 17
2.6 Kešování odpovědí 17
2.7 Dokumentace 18
2.8 Shrnutí 18

3 Využití introspekce **21**

3.1 Dokumentace 21
3.2 Validace 22
3.3 Vizualizace typového systému 22

4 Příklady použití GraphQL **25**

4.1 GraphQL nad REST API 25
4.2 Mockování API 26
4.3 Relay 27

5 Implementace **31**

5.1 Popis aplikace . . . 31
5.2 Architektura . . . 32
5.3 Server + jeho rozhraní . . . 33
5.4 GraphQL . . . 33
5.5 GraphiQL . . . 37
5.6 Webový server . . . 38
5.7 Klientská aplikace . . . 38
5.8 Vývojové nástroje . . . 40

6 Možné využití implementace 43

Závěr 45

Literatura 47

A Seznam použitých zkratek 49

B Instalace a návod k použití 51
B.1 Zapnutí vývojového režimu . . . 51
B.2 Zapnutí produkčního režimu . . . 52

C Typový systém implementované aplikace 53

D Obsah přiloženého CD 57

Seznam obrázků

1.1 Ekosystém GraphQL 8

2.1 Přístup k REST a GraphQL API 12

3.1 Validování pomocí introspekce 22

4.1 GraphQL jako prostředník pro REST API 26

5.1 Architektura aplikace 32
5.2 GraphiQL IDE 37
5.3 Vykreslená klientská aplikace 39

C.1 Vizualizace typového systému 56

Seznam tabulek

5.1 Popis typového systému 34

Úvod

Na počátku webových stránek bylo statické HTML. S příchodem Javascriptu začali stránky postupně ožívat a objevovala se první interaktivita. XMLHttpRequest přidal možnost získávat další data ze serveru a postupem času se objevili první tzv. single–page aplikace. Dnešní webové aplikace pracují ve webových prohlížečích jak mobilních tak desktopových zařízeních. S vývojem webových stránek se ale také vyvíjela komunikace se serverovou částí. Bylo nutné přijít s metodou jak propojit serverové komponenty s klientskými. Typicky se začali používat aplikační rozhraní typu REST, popsané Royem Fieldingem. Klient u takového rozhraní ale nemůže ovlivnit, která data dostane, to určuje server. Dále nemá žádnou kontrolu nad počtem dotazů, data, která klientská aplikace potřebuje můžou být dostupná přes několik komunikací se serverem. Tyto a další vlastnosti se stali ve webových aplikacích velice nepraktické.

Společnost Facebook v roce 2015 popsala technologii GraphQL, která by měla komunikaci klientských a serverových aplikací povznést na další úroveň a vyřešit problémy, které obnáší tradiční přístup. Tato technologie dovoluje klientským aplikacím specifikovat své požadavky a serverová část tyto požadavky zpracuje a vrátí klientovi. Technologie také zavádí typový systém do komunikace mezi klientem a serverem.

V této práci porovnáme tradiční přístup komunikace klienta se serverem právě s touto novou technologií GraphQL. Popíšeme si existující nástroje pro práci s touto technologií. Dále pomocí této technologie implementujeme ukázkovou aplikaci a zaměříme se na další možná využití technologie GraphQL.

Kapitola **1**

Technologie GraphQL

GraphQL je dotazovací jazyk na úrovni aplikační vrstvy. Server interpretuje řetězec s dotazem a v požadovaném formátu vrátí odpověď s daty. Za vznikem GraphQL stojí společnost Facebook. Vývoj technologie začal s přechodem mobilních aplikací společnosti Facebook řízených HTML5 na nativní aplikace v roce 2012. V roce 2015 byl GraphQL uvolněn jako open source. Od té doby vzniklo mnoho implementací v různých jazycích jako je javascript, python, c#, php a mnoha dalších. GraphQL není spojen s žádnou konkrétní technologií pro uložení dat. Při implementaci serverové části GraphQL pouze mapujeme typový systém (viz 1.1.2) GraphQL na námi zvolený úložný systém.

V následujících částech si popíšeme základní principy GraphQL a implementaci v jazyce javascript od společnosti Facebook. Kompletní specifikace GraphQL je popsána v github repozitáři viz [1].

1.1 Základní specifikace jazyka

GraphQL je jazyk, ovšem ne programovací, který umožňuje dotazovat se na data aplikačního serveru, který splňuje specifikaci GraphQL. Není spojen s žádným konkrétní programovacím jazykem nebo úložným systém aplikačního serveru.

1.1.1 Vlastnosti GraphQL

Hierarchický Dotaz v GraphQL je strukturován jako hierarchická množina polí. Odpověď s daty potom kopíruje podobu dotazu. To přináší klientům specifikovat podobu dat, se kterými potom bude pracovat.

Produktově orientovaný jazyk Požadavky na data určuje klient svým pohledem na data. Odpověď na dotaz je ve tvaru podle dotazu, to znamená

že podobu dat si určuje klient a není specifikována globálně serverem. Tedy každý klient bude mít přizpůsobené odpovědi pro své požadavky a zároveň všechny odpovědi dokáže vystavovat jedno obecné API.

Silně typový GraphQL server určuje typový systém specifický pro aplikaci. Dotazy jsou v tomto systému validovány před vykonáním. Lze tedy ušetřit jak výpočetní výkon, tak lze se znalostí typového systému provádět validaci ještě před odesláním dotazu na aplikační server.

Klientem specifikované dotazy Server poskytuje možnosti prostřednictvím typového systému. Odpovědností klienta je využití tohoto systému. Klient dostává přesně to o co požádá a nic navíc.

Introspekce Tedy typový systém musí být dotazovatelný pomocí GraphQL dotazů. Tuto vlastnost lze využít při prozkoumávání schopností API nebo pro stavbu dokumentace, či pro automatické generování dotazů a podobně.

GraphQL je složen z typového systému, dotazovacího jazyka, vykonávací sémantiky, statické validace a introspekce. Jednotlivé části si stručně rozebereme, detailní specifikace celého jazyka je uvedená v github repozitáři viz [1].

1.1.2 Typový systém

Typový systém popisuje možnosti aplikačního serveru a určuje zdali je dotaz validní. Možnosti systému jsou nazývány schématem. Schéma je definováno podporovanými typy a direktivy. Schéma obsahuje kořenové typy pro dotazy a mutace. Zatímco dotazy jsou **read-only**, mutace jsou určená pro **modifikaci dat**.

Všechny typy ve schématu musí mít unikátní jméno, totéž platí i pro direktivy. Ovšem direktiva a typ může mít společné jméno, protože neexistuje zaměnitelnost mezi nimi.

Základním typem systému je **Scalar**, ten je reprezentovaný řetězcem nebo celým číslem. Spolu s typem **Enum**, tedy výčtem hodnot, tvoří skalární hodnota listy ve stromové struktuře odpovědi.

Další úroveň typů je **Object** a **Enum**. Tedy objekt je množina polí, zatímco rozhraní definuje množinu polí, kterou objekt implementuje. Pomocí typu **InputObject** je možné určit, jaká data klient používá v dotazu.

Definice možných typů je určena za pomocí typu **Union**.

Pro účely této práce je nejdůležitější typ **Object**, pomocí kterého popíšeme typy serverové části aplikace. Jako příklad si můžeme ukázat definici typu článku, viz kód 1.1.

Kód 1.1: Typ Článek

```
type Article {
	id: String
	name: String
	url: Url
	content: String
	isPublic: Boolean
}
```

Článek je tedy definován svými parametry. V uvedeném případě obsahuje tři znakové řetězce, jedenu *boolean* hodnotu a jeden vlastní typ **Url**, u kterého server zajišťuje validitu URL adresy.

1.1.3 Dotazovací jazyk

Dotazy deklarativně[1] popisují jaká data si tazatel přeje získat z aplikačního serveru. Opět pro názornost uvedeme dotaz na článek v kódu 1.2.

Kód 1.2: Dotaz na GraphQL

```
query ArticleQuery {
	homepage {
		name
		content
	}
}
```

První řádek dotazu uvedeném v kódu 1.2 definuje operaci a její jméno, v našem případě je název **ArticleQuery**. Potom vybíráme kořenový prvek ze schématu, v našem případě je to hlavní stránka, tedy **homepage**. A následuje výčet parametrů článku, které si přejeme zobrazit. Příklad takové odpovědi je znázorněn v kódu 1.3.

Kód 1.3: GraphQL odpověď

```
{
	"homepage": {
		"name": "Homepage",
		"content": "<h1>Homepage title</h1>"
	}
}
```

[1] Definujeme pouze co potřebujeme, ale nijak nespecifikujeme jak má k získání dat dojít.

Z příkladů dotazu 1.2 a příslušné odpovědi 1.3 si můžeme všimnout podobnosti dotazovacího jazyka se zápisem JSON, ve kterém je odpověď. Výhodou takového zápisu je, že odpověď na dotaz je v podstatě jeho kopie s hodnotami jednotlivých parametrů.

1.1.4 Validace

Díky definovaným typům schématu je možné validovat dotazy. Toto vlastnost umožňuje validovat dotaz ještě před vykonáním a lze tedy informovat o chybě dotazu ještě než je odeslán. Validní dotaz musí mít v listech dotazu pouze skalární hodnoty. Naopak skalární hodnoty nesmí obsahovat množinu prvků pod nimi.

Validní dotaz jsme ukázali například v části popisujícím dotazovací jazyk, viz 1.1.3. Pro názornost si ukážeme ještě nevalidní dotaz v kódu 1.4.

Kód 1.4: Nevalidní GraphQL dotaz

```
query ArticleQuery {
        homepage {
                name {
                        id
                }
                author {
                }
        }
}
```

Dotaz z kódu 1.4 je nevalidní právě proto, že pole *name* je typu skalár, takže nemůžeme přistupovat k jeho polím. Dalším problémem v dotazu je, že pole *author* nežádá o žádné pole pro jeho typ.

1.1.5 Introspekce

Vlastnost introspekce GraphQL umožňuje vývojářům prozkoumávat možnosti aplikačního serveru, aniž by musel číst dokumentaci. Dokonce samotný typový systém je definován s popisem typů, který lze pomocí introspekce zobrazit a získat tak aktuální dokumentaci jako odpověď na dotaz. Specifikace GraphQL určuje, že každý typ má popis, který používá syntaxi Markdown (viz [2]. Výhoda tohoto přístupu je, že dokumentace je součástí kódu a není oddělená.

Introspekce je tedy API pro dotazování se typů poskytnutým ve schématu. Pro příklad s článkem, bychom se mohli dotázat na všechna pole článku a jejich typy dotazem v kódu 1.5.

Kód 1.5: Introspekce

```
{
	__type(name: "Article") {
		name
		description
		fields {
			name
			type {
				name
			}
		}
	}
}
```

Pomocí dotazu 1.5 získáme zpět odpověď, která bude obsahovat název a popis typu článku a jeho všechny položky. Tímto způsobem můžeme vygenerovat například dokumentaci, viz další použití introspekce 3.

Z dotazu si také můžeme všimnout, že při k introspekci používáme dotazy nad typy, jejichž jméno je prefixováno dvěma podtržítky. To je konvence zavedená kvůli prevenci kolizi názvů s uživatelskými typy.

Introspekce obsahuje také metadata, jimiž je popis a označení zastaralosti typu spolu s důvodem. Výchozím chováním GraphQL serveru je, že pole označená jako zastaralé nezobrazuje v introspekci. Tedy uživatel, který vyvíjí aplikaci má menší pravděpodobnost že na ně narazí.

1.2 Ekosystém

V krátkosti si popíšeme ekosystém GraphQL. Tedy jednotlivé části a jejich role.

GraphQL specifikace je samotná specifikace jazyka GraphQL.

Aplikační server tedy GraphQL endpoint, využívá **jádro GraphQL**, což je implementace pro specifický jazyk serveru. A využívá **knihoven** pro mapování typů z jazyka na výsledná data. Může to být například knihovna mapující GraphQL do SQL úložiště či jiné datové úložiště nebo jiná služba třetí strany.

Klientská aplikace využívá **vývojové nástroje** pracující s GraphQL endpointem, tedy například Relay (viz [3]) a mnoho dalších technologií. Další vrstvou jsou společné **nástroje**, mezi které patří například GraphiQL.

Obrázek 1.1: Ekosystém GraphQL

1.3 Motivace ke vzniku

Hlavní motivací ke vzniku GraphQL byla snaha lépe strukturalizovat API pro mobilní aplikace a klientské aplikace obecně. Oproti klasickému REST přístupu byla snaha snížit počet dotazů na API a nepotřebnost některých vystavených dat od zdroje. Zdroje dostává klient od serveru tak, jak je definuje server a ne uživatel. V ideálním případě by klient měl dostat pouze ta data, která potřebuje ke svému aktuálnímu pohledu a nejlépe v jedné cestě k API.

Možností vyřešení tohoto problému by bylo nepoužívat REST API, ale vytvářet pro klientské aplikace vlastní endpoint. To ovšem znamená, že mezi jednotlivými částmi může docházet k duplikaci kódu a zbytečně se nějaké funkcionality můžou implementovat dvakrát. Více implementací jedné věci také v rámci celé aplikace může přinášet nekonzistence.

GraphQL je tedy stavěn na myšlence, že uživatel dotazuje strukturu, kterou očekává a server poskytuje svou schopnost vystavovat data či funkčnost. Změna klientské části v tom případě nemusí znamenat úpravu serveru, můžeme pouze využívat jinou část ze serveru vystavené množiny schopností. Pokud dochází k rozšíření funkcionality aplikačního serveru, pouze rozšíří množinu vystavovaných dat, což usnadňuje vývoj nových verzí jak aplikačního serveru tak klientských aplikací.

1.4 GraphQL knihovny a nástroje

V této části si uvedeme seznam knihoven implementující GraphQL nebo nástroje postavené pro GraphQL, vydané společností Facebook.

GraphQL.js Knihovna poskytující serverovou implementaci GraphQL, viz [4]. Knihovna poskytuje obal nad komplexními datovými modely. Při implementaci mapujeme schéma na právě na tyto datové modely a operace s nimi.

GraphiQL Je interaktivní IDE pro prohlížeče. Využívá introspekce 1.1.5 a generuje dokumentaci z kódu. Také kontroluje validaci a nabízí možnost doplňování dotazů, více v kapitole o využití introspekce 3.

Relay Framework využívající knihovny **Relay**, který se stará o komunikaci s GraphQL endpointem. Klient při implementaci pouze deklarativně určuje požadavky na data a výslednou komunikaci poté přebírá samotný framework.

Kapitola **2**

Porovnání GraphQL s tradičním přístupem REST

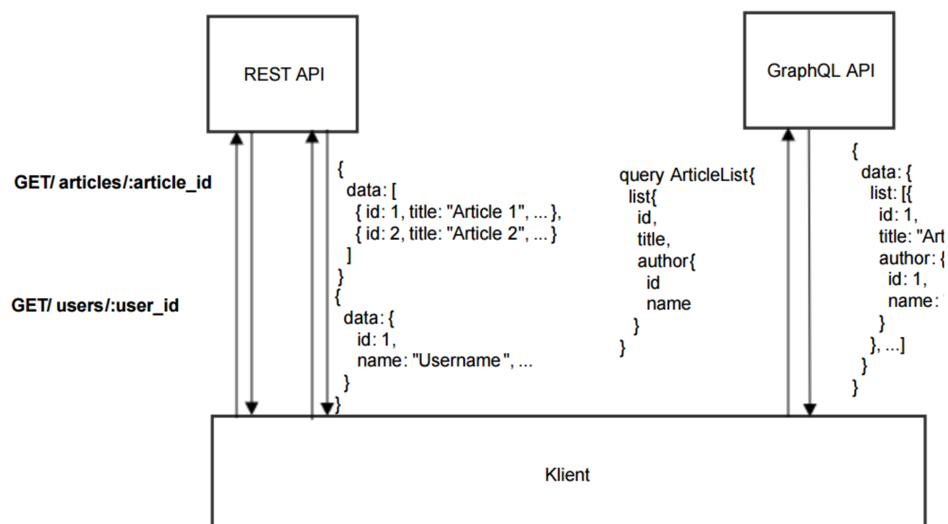
V následujících částech si popíšeme rozdíl mezi přístupem nabízející GraphQL a REST endpoint. Tedy získávání a manipulace s daty z aplikačního serveru. V krátkosti si přiblížíme, jak by mělo REST API vypadat. Popíšeme si rozdíly v počtu dotazů potřebných k získání všech dat pro aplikaci, také rozdílnost jejich podoby a jak se liší možné výstupy. Podíváme se na problém verzování API, který je u REST důležitý pro zachování zpětné kompatibility. Navrhneme způsob řešení verzování u GraphQL, kde lze využít vlastností dotazovacího jazyka a mít všechny dostupné verze přístupné bez nutnosti určení specifické verze. Rozebereme způsoby pro vytváření, aktualizaci a mazání dat. Také se podíváme, jak jednotlivé požadavky na API mockovat a jak odpovědi kešovat.

2.0.1 Popis rozhraní REST

Autorem návrhu a popisu REST rozhraní je Roy Fielding. Celý popis je uveden v práci [5]. **Representational State Transfer**, dále REST je princip návrhu architektury pro distribuované systémy. Systémy, které dodržují tyto principy se nazývají **RESTful**. V krátkosti si uvedeme základní principy REST, jimiž jsou:

Klient—Server Oddělení odpovědnosti klientského uživatelského rozhraní od datového úložiště. Tím zlepšujeme přenosnost uživatelského rozhraní do jiných systémů a škálovatelnost serverových komponent.

Beze stavovost Veškerá komunikace je beze stavová. Odpovědnost za stav je pouze na klientovi. Proto při komunikaci musí klient poskytnout



Obrázek 2.1: Přístup k REST a GraphQL API

veškeré potřebné informace, aby server byl schopen zpracovat požadavek.

Cache Od všech odpovědí na klientův dotaz se očekává, že budou označeny jako kešovatelné nebo nekešovatelné. Pokud jsou data kešovatelné, jsou data znovupoužitelná pro pozdější požadavek.

Unifikované rozhraní Od rozhraní se očekává, že každý zdroj má unikátní identifikátor (**URI**). Jednotlivé zdroje jsou oddělené od reprezentace, která je poskytnuta klientovi. Tedy server například vrací reprezentaci zdroje v nějakém z formátů HTML, XML nebo JSON. Klient poté manipuluje s daty právě skrze tyto reprezentace a využívá **CRUD** operace. Veškeré zprávy musí být **sebe–popisující**, tedy mít dostatek informací ke zpracování zprávy. Stav aplikace je určen pomocí hypermédia (**URL adresy**), k dalším stavům se klient dostává pomocí odkazů, které jsou v odpovědi od serveru.

Rozvrstvení architektury Klient nemůže rozpoznat, jestli komunikuje přímo se serverem nebo s prostředníkem. Prostředník může zvýšit škálovatelnost systému pomocí load balancingu a poskytnutím sdílené cache paměti. Také může vyžadovat bezpečnostní politiku.

Code-On-Demand (nepovinné) REST dovoluje využít funkcionalitu stažením a vykonáním kódu zaslaným ze serveru. Příkladem tohoto je napří-

klad vykonání Javascriptu.

2.1 Počet dotazů

Prvním rozdílem přístupu k rozhraní implementující GraphQL a REST je počet položených dotazů na aplikační server (viz 2.1). Mějme například aplikaci, která zobrazuje seznam článků. Pomocí REST API přistoupí ke zdroji s URI typu:

http://example.com/articles/.

A tím získá odpověď obsahující seznam článků, pro úplnost uvedeme příklad v kódu 2.1.

Kód 2.1: REST odpověď

```
1  {
2          "data": [
3                  {
4                          "id": 1,
5                          "title": "Article 1",
6                          "author": "http://www.example.com/
                              users/1",
7                          ...
8                  },
9                  {
10                         "id": 2,
11                         "title": "Article 2",
12                         "author": "http://www.example.com/
                              users/5",
13                         ...
14                 }
15                 ...
16         ]
17 }
```

Z ukázkové odpovědi 2.1 si můžeme všimnout, že v seznamu článků je pod autorem uvedený odkaz na zdroj, viz popis RESTu 2.0.1. V případě, že chceme v naší aplikaci vypsat například jméno autora, musíme ke zdroji přistoupit zvlášť a požádat tedy aplikační server o data uživatele. To vede k tomu, že se nám zvyšuje počet dotazů, podle potřeby zobrazovat data.

Oproti tomu přístup ke GraphQL aplikačnímu serveru nám dovoluje specifikovat data, která očekáváme. Místo několika dotazů, bychom předchozí případ řešili podobným dotazem získaným z kódu 2.2.

Kód 2.2: GraphQL dotaz

```
query ArticleQuery {
	list {
		id
		title
		author {
			id
			name
		}
	}
}
```

Aplikační server nám tedy na jeden dotaz vrátí veškeré potřebné informace. Toto chování oceníme nejvíce v mobilních aplikacích, kde více požadavků na server může výrazně zpomalit běh aplikace. Navíc počet dotazů na REST API nemusí být předem známý a rychlost aplikace tedy bude dost kolísat v této závislosti. Dotazy na REST API nemusíme být ani schopni paralelizovat, protože jejich URI můžeme zjistit až s položením dotazu. Tedy například musíme první požádat o seznam článků a z odpovědi požádáme o příslušné zdroje související k článku (autor, komentáře a pod.)

2.2 Tvar a velikost výstupu

Základní vlastnost GraphQL je produktová orientace, viz 1.1.1. Tedy klient pomocí svého dotazu určuje formát dat. Pokud některá data nepotřebuje, jednoduše o ně nepožádá a odpověď od serveru je o to kratší. Klient tedy žádá pouze o to, co potřebuje pro aktuální pohled nad daty. Server nic o tomto pohledu nemusí vědět. Tuto vlastnost můžeme využít pro lepší optimalizaci serverové aplikace. Když máme nějaké výpočetně náročné operace, které ale klient nemusí v odpovědi požadovat, můžeme jejich výpočet na serveru přeskočit. Klient může také v dotazu využít tzv. *aliasy*, tedy vlastní pojmenování polí v odpovědi a tím tvar odpovědi ještě lépe specifikovat pro vlastní účely.

V případě REST API klient dostává vždy všechna data, ať už je potřebuje nebo ne. Server zároveň určuje i formát odpovědi a při změně se musí přizpůsobit klient, viz 2.3. Pokud tedy aplikační server rozšíří svou funkcionalitu, například přidá nějaká pole pro určitý zdroj, klient tyto data dostává aniž by je nějak zpracovával. Odpovědi od aplikačního serveru nebudou minimální pro specifický pohled klientské aplikace, pokud pro tyto pohledy nevytvoříme vlastní endpoint, což ale vytvoří další problémy, například horší udržitelnost kódu a jeho složitá znovupoužitelnost.

Rozdíl v přístupu je tedy ten, že u REST API server definuje zdroje, zatímco u GraphQL server pouze určuje možnosti jednotlivých typů. Velikost odpovědí je u REST API daná velikostí zdroje a nelze ji nijak minimalizovat, u GraphQL je velikost závislá na dotázaných částech a nejsou vystavena žádná jiná data.

2.2.1 Parametrizace výstupu

Zvláštní případ nastává, pokud chceme výstup nějakým způsobem parametrizovat. Například v uvedeném seznamu článku (viz 2.1), bychom chtěli použít nějaký stránkovač. V REST API by aplikační server mohl poskytovat tuto možnost nějakými parametry v URI. Seznam článků bychom získali ze zdroje:

http://example.com/articles/?limit=15&skip=30.

Tedy žádáme o 15 článků a prvních 30 přeskakujeme. Respektive náš stránkovač zobrazuje třetí stranu článků.

Zatímco chceme–li implementovat tuto funkčnost pomocí GraphQL, měníme parametry dotazu a ty posíláme spolu s dotazem na aplikační server. Tedy dotaz uvedený zde 2.1, by v případě stránkování mohl vypadat jako je popsaný v kódu 2.3.

Kód 2.3: Stránkování v GraphQL

```
query ArticleQuery($limit: Int, $offset: Int) {
	list(limit: $limit, offset: $offset) {
		...
	}
}
```

V obou případech ovšem musí funkčnost implementovat aplikační server a klient by měl najít v dokumentaci, která v případě GraphQL je součástí kódu.

2.3 Verzování

U REST API, v případě, že chceme zachovat funkčnost aktuálních klientů při změně struktury API, musíme řešit verzování. V praxi se to řeší například prefixem v URI zdroje, tedy URI článků by vypadalo následovně:

http://example.com/api/v1/articles.

Není to samozřejmě jediné řešení, dalším řešením je uvedením verze do HTTP hlaviček či změny hlavičky **content-type**. Způsob verzování není předmětem této práce, proto další informace lze nalézt například v článku [6]. Ve všech případech musíme technicky řešit, aby pro různé verze vracelo

API požadovaný výsledek. A to není zrovna jednoduchý problém kdy je potřeba zachovávat staré implementace a tím je server značně zkomplikován. Pokud bychom se rozhodli neverzovat API, klienti by byli nuceni přizpůsobovat neustále své aplikace s každou změnou.

Samozřejmě v GraphQL bychom mohli narazit na stejný problém, tedy změnou odpovědi od serveru nemusíme zachovat zpětnou kompatibilitu. To by se mohlo stát, třeba když bychom změnili nějakou skalární hodnotu na objekt, abychom mohli popsat složitější strukturu. Toto se dá obejít, využijeme—li přístup "add—only". Tedy když chceme pozměnit nějakou funkcionalitu, přidáme ji jako novou. Tímto přístupem zajistíme zpětnou kompatibilitu bez vedlejších efektů. Zatímco v REST přístupu bychom jako vedlejší efekt měli neustále rostoucí velikost reprezentace zdroje, v GraphQL specifikuje klient dotazem, jak má odpověď vypadat. Tento přístup je navíc podpořen vlastností (viz 1.1.2), že v deklaraci typu můžeme uvést, že je daný typ zastaralý a vysvětlující důvod. Pomocí sebe reflexe (viz 1.1.5) může potom klient jednoduše zjistit, zdali jeho aplikace využívá některou ze zastaralých částí API.

2.4 Modifikace dat

Jak je uvedeno v popisu REST API 2.0.1, ke komunikaci s aplikačním serverem používá klient **CRUD** operace. Konkrétně pro modifikaci nebo vytvoření nových dat či smazání operace **create**, **read** nebo **delete**. Zvolená operace se vybírá podle použité HTTP metody. Tedy chce–li například uživatel smazat zdroj *Article* s ID 5, pošle na adresu zdroje HTTP požadavek s metodou **DELETE**.

U GraphQL ale nepracujeme se zdroji. Pro dotazování nad daty máme nějaký **RootQuery**, tedy kořenový typ, který nám vystavuje své dotazovací možnosti. Ovšem pro změny používáme tzv. **mutace**. GraphQL endpoint může obsahovat kořenový typ s existujícími mutacemi, je ovšem nepovinný a žádné mutace nemusí být dostupné. Dotaz s mutací poté vypadá podobně jako klasický dotaz na data, akorát vkládáme vstupní parametry. Pro úplnost opět uvedeme ukázku jednoduché mutace znázorněné kódem 2.4.

Kód 2.4: Mutace v GraphQL

```
1 mutation {
2         deleteArticle(articleID: 5489) {
3                 deletedArticle {
4                         id
5                         title
6                 }
```

```
7         }
8 }
```

V ukázkové mutaci 2.4 si můžeme všimnout, že celý dotaz je označen jako typ *mutation*, na rozdíl od klasického dotazu je toto označení povinné. I když je v příkladu operace na smazání, tedy *DELETE*, neznamená to, že máme dostupné **CRUD** operace jako je tomu v REST API. Jednotlivé mutace jsou opět definovány v typovém systému v implementaci aplikačního serveru. Jejich pojmenování je libovolné.

V dotazu si také můžeme všimnout, že v odpovědi získáme ID spolu s titulkem článku. Tedy lze se mutací dotázat na smazená data, v případě REST přístupu bychom dostali zpět opět nějakou pevnou strukturu nebo jen signalizaci o úspěšnosti operace pomocí stavového kódu HTTP.

2.5 Mockování API

Mockování[2] využijeme, pokud chceme například začít s klientskou částí aplikace dříve, než je hotový aplikační server. Další možnost využití je pro testovaní. V obou případech to znamená, že musíme udržovat skutečnou podobu endpointu s jeho mockem. Tyto dvě podoby se zneplatní s každou změnou serverové části aplikace, pokud nejsou udržované zároveň. Mockování také vyžaduje nějakou vrstvu v aplikaci, která se stará o jeho zapínání popřípadě vypínání. Také je potřeba popsat podobu mockovaných dat.

V REST API je toto složitý úkol, protože neexistuje žádný standardní formát popisující všechny informace potřebné k tomu, abychom mohli endpoint používat jak v serverové tak klientské části. Samozřejmě na trhu existují produkty, které tento problém nějakým způsobem řeší. Například služba *apiary*[3], ovšem pořád je potřeba tyto nástroje udržovat aktuální.

Díky typovému systému v GraphQL je mockování podstatně jednodušší problém. Typy můžeme sdílet mezi serverovou a klientskou aplikací. Pomocí definice typů získáme veškeré potřebné informace k namockování. Detailněji se na tento problém podíváme v kapitole 4 v sekci 4.2.

2.6 Kešování odpovědí

Pokud máme RESTful endpoint, můžeme využít kešování dat na úrovni HTTP. K určení pravidel kešování můžeme využít HTTP hlavičky a to:

[2] Nahrazení nějaké funkčnosti její imitací

[3] Apiary je služba, ve které je stanoven formát pro popis API endpointu a tento popis lze využít pro mockování v klientské části nebo testování serverové části.

- **Expires** udává klientovi, kdy je zdroj neplatný
- **Cache-Control** určuje politiku a strategii ukládání do mezipaměti
- **Etag** je unikátní identifikátor odpovědi
- **Last-Modified** udává datum poslední změni zdroje

Protože v GraphQL dostáváme data vždy z jednoho endpointu, ztrácíme idempotenci požadavku a nemůžeme tak odpověď na požadavek ze serveru ke klientovi kešovat. Veškeré kešování dat je tedy na klientské aplikaci. V praxi to řeší například framework *Relay* (viz [3]), více v kapitole o klientských aplikacích 5.7.

2.7 Dokumentace

Aby byl klient schopný s jakýmkoliv API pracovat, je nutné aby znal strukturu tohoto rozhraní. Pokud potom potřebuje získat některá data, musí vědět v jaké části rozhraní tyto data najde. K orientaci slouží klientovi dokumentace.

Hlavním rozdílem mezi GraphQL a REST dokumentací je ten, že v případě GraphQL je dokumentace součástí typového systému a lze jí dotazovat pomocí dotazovacího jazyka, tedy konkrétně využití introspekce systému. Lze tedy vytvořit dotaz, který klientovi zobrazí dostupnou dokumentaci nezávisle na tom, na který GraphQL endpoint tento dotaz položí. Zároveň to, že dokumentace je součástí kódu, zvyšuje pravděpodobnost, že jí vývojář aktualizuje, mění–li nějakou část v aplikaci.

V případě REST dokumentace, není žádný požadavek kde má být umístěna. Nelze ji ani získat obecně z API rozhraní a proto pro specifický endpoint je klient nucen zjistit si tyto informace sám z příslušných zdrojů. Navíc pro použitelnost dokumentace je nutná také její aktuálnost.

2.8 Shrnutí

Na příkladech jsme ukázali, že přístup GraphQL a REST je velice rozdílný. Zatímco u RESTu definuje zdroje a jejich strukturu pro klienty server, v GraphQL server difinuje pouze své možnosti, strukturu i velikost získaných dat určuje klient. U GraphQL je zaručeno, že dokážeme získat potřebná data, a to libovolně zanořené stromové struktury, pomocí jediné cesty k API. U REST přístupu musíme použít pro zmenšení počtu dotazů například techniky, kdy určíme pomocí parametrů v požadavku co má server vystavit a ani to v některých případech nemusí zaručit pouze jedinou cestu k API. Pokud chceme toto chování obejít, skončíme s implementací

API endpointu určeného přímo pro specifické pohledy klientské aplikace, což znamená více kódu na straně aplikačního serveru a složitější údržbu. O to víc v případě, že chceme API endpoint používat z více klientských aplikacích.

Ukázali jsme si složitost mockování API v RESTu a uvedli téma o jednoduchém mockování GraphQL. Toto můžeme využít zejména při oddělené implementaci klientské a serverové části aplikace, kde nám stačí mít společné schéma typového systému GraphQL.

Nevýhoda GraphQL v některých případech může být nemožnost cachování službami třetích stran jako je proxy server nebo reverzní proxy server a podobně. Tato odpovědnost v případě GraphQL padá na stranu klienta. Nemožnost cachovat klientské dotazy službami třetích stran je způsobena tím, že neexistuje žádná technika k identifikaci dotazů. V případě API, které jsou zdrojově orientované, jako je například i REST, můžeme využívat pro tuto identifikaci HTTP hlavičky.

Kapitola **3**

Využití introspekce

Vlastnost introspekce nám přináší možnosti, jak vytvářet nástroje pro práci s GraphQL. V této kapitole si popíšeme některé tyto možnosti, jako je práce s dokumentací a klientskou validaci dotazu. Introspekce je dotazování se nad typovým systémem. Každý typ sebou nese například název, popis a seznam svých polí, které mají opět název, popis a mimo to i zdali jsou zastaralé a podobně.

Introspekce je využita také v implementaci, konkrétně v rozhraní pro dotazování popsaném v sekci 5.5. Použité IDE využívá introspekci právě pro stavbu dokumentace a také k napovídání ke stavbě dotazů podle aktuálního zanoření.

3.1 Dokumentace

Jak jsme uvedli v popisu introspekce (viz 1.1.5), součástí typového systému je i jeho dokumentace. Můžeme se tedy dotázat na samotné schéma, které obsahuje kořenový objekt s možnostmi pro dotazování, kořenový objekt s mutacemi, ale také všechny typy schématu a direktivy. Tyto hodnoty můžeme využít pro generování dokumentace. Každý typ obsahuje hodnotu *description*, kde je uveden popis typu v Markdown formátu (viz [2]). Pro každý nástroj je tedy doporučená podpora toho formátu.

Každý typ má také uvedeno zda je zastaralý, popřípadě důvod. Na této hodnotě můžeme postavit nástroje, které nás upozorní na používání částí typového systému, kterému bychom se měli vyhnout.

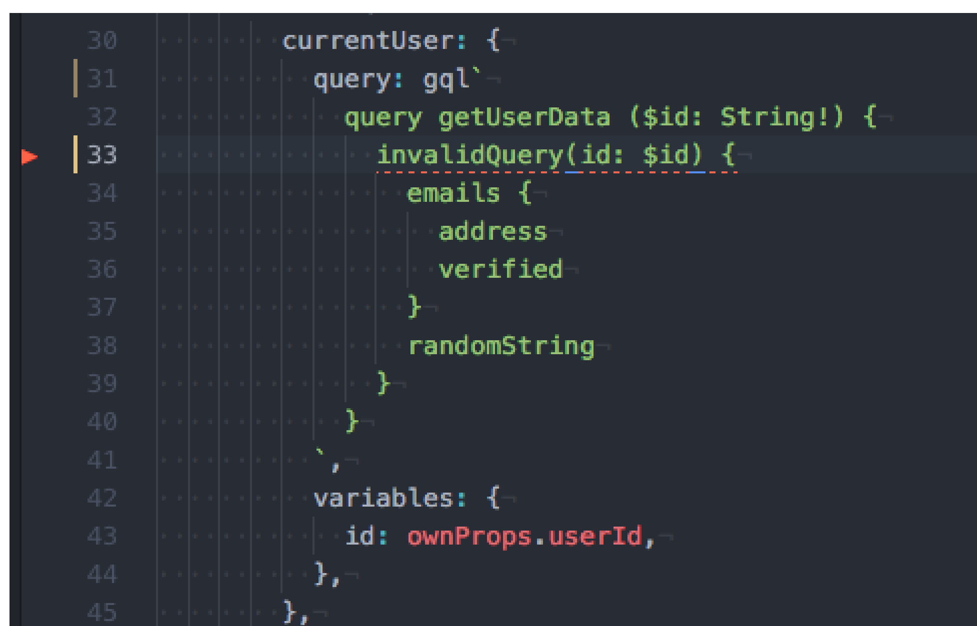
Generování dokumentace podle introspekce používá například knihovna GraphiQL, viz 1.4. Každý GraphQL endpoint lze dotazovat pomocí polí *__schema*, *__type* a *__typename*.

3.2 Validace

Validovat dotaz je možné za podmínek, že máme známé aktuální schéma. Klient tedy nemusí svůj dotaz posílat na server, ale validitu dotazu si může ověřit ještě předtím. Schéma GraphQL endpointu zjistí pomocí introspekce, k vygenerování použije klient pole *__schema*, které obsahuje kořenové typy pro dotazování, mutace a všechny direktivy.

Toto můžeme opět využít pro stavbu nástrojů, které nám zvýrazní nevalidní části dotazu. Na určité úrovni dotazu jsme také schopni říct, jaké máme možnosti pro dotazování v aktuální hierarchii. Lze tedy postavit nástroj, který nad dodaným schématem kontroluje validitu kódu a ukazuje aktuální možnosti dotazování.

Ukázku validace i doplňování kódu podle zanořené úrovně dotazu opět můžeme pozorovat v knihovně GraphiQL, viz 1.4. Další ukázku můžeme pozorovat například v knihovně *eslint-plugin-graphql*, ta dokáže s přiloženým schématem přímo v textovém editoru upozorňovat na nevalidní dotazy, ukázka je znázorněna na obrázku 3.1.



Obrázek 3.1: Validování pomocí introspekce

3.3 Vizualizace typového systému

Existují nástroje, které pomocí jednotného introspekčního dotazu dokáží vizualizovat celý typový systém jako graf. viz [7]. Tento nástroj byl použit

i na výsledný typový systém implementované aplikace a je zobrazen na obrázku v příloze C.1.

Kapitola **4**

Příklady použití GraphQL

Mimo klasický případ použití GraphQL, kterým je mapování dotazovacího jazyka na datové úložiště, si v této kapitole uvedeme několik dalších možných případů použití.

Abychom mohli začít používat GraphQL na existujících aplikacích, které například poskytují RESTful API, nemusíme stavět API od začátku. Ukážeme si, že lze využít již existující API pro nasazení GraphQL.

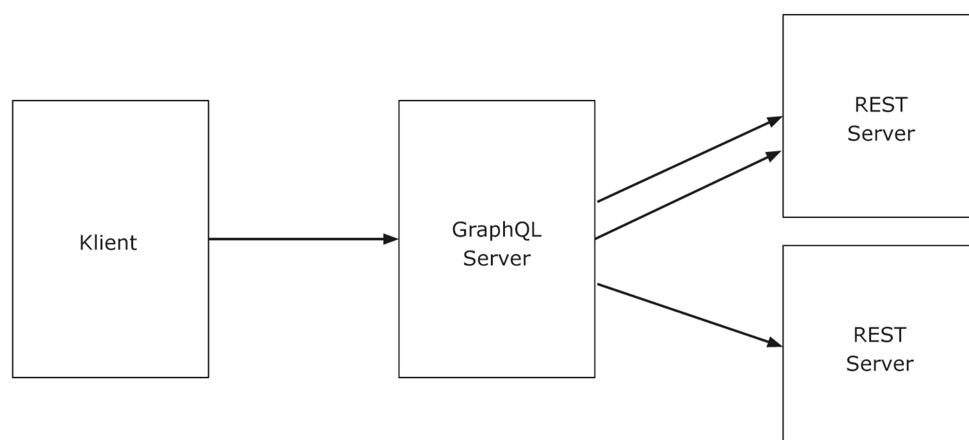
Dále se budeme zabývat mockováním API, které umožnuje vývoj frontendové části aplikace ještě před vývojem samotného API endpointu. Navážeme na sekci o rozdílu v mockování v REST a GraphQL API, viz 2.5.

Podíváme se také na Javascriptový framework Relay, který komunikuje s GraphQL serverem. Tento framework se stará o efektivní dotazování GraphQL serveru a vývojář pouze definuje, která data potřebuje pro své komponenty. Relay také podporuje mutace dat a umí například dělat optimistické změny (viz 4.3).

4.1 GraphQL nad REST API

Pokud nevyvíjíme serverovou část aplikace úplně od začátku, můžeme ztrácet motivaci k přechodu na GraphQL. Na našem endpointu můžou být závislé klientské aplikace nebo na vývoj od začátku nemusíme mít dostatek prostředků. Naopak se můžeme ocitnout v situaci, kdy v klientské aplikaci chceme používat GraphQL ale endpoint, se kterým pracujeme je RESTful.

GraphQL endpoint můžeme využít k mapování dotazovacího jazyka na existující REST API, jak je znázorněno na obrázku 4.1. V tomto případě GraphQL server komunikuje s REST API, ale nevadí nám, že REST vrací dlouhé odpovědi a nebo některá data jsou potřeba získat pomocí několika



Obrázek 4.1: GraphQL jako prostředník pro REST API

HTTP requestů, protože komunikace probíhá na úrovni serverů[4]. Zatímco klient plně využívá výhod GraphQL, tedy odpovědi jsou ve formátu podle dotazu a všechna data je schopen získat přes jeden dotaz a podobně.

4.2 Mockování API

Pro mockování GraphQL API potřebujeme schéma. Z něho dostaneme typový systém. Protože víme, že dotaz API je ve stromové struktuře, v jejíž listech jsou skalární hodnoty, stačí nám pro základní mockování vytvořit data pro ně. Tedy klientská část může dotazovat endpoint a podle tvaru svého dotazu dostane data, kde jejich skutečné hodnoty jsou pouze nějak určené skalární hodnoty (například náhodně nebo z nějaké knihovny). Serverová aplikace má k dispozici schéma a to implementuje.

Protože každý GraphQL server požaduje schéma ke svému fungování, nevytvořila potřeba mockovat data nutnost zvláštního kódu. Ten přibude v případě, že chceme pro mockovaná data použít nějaké nestandardní generování. Ale i v tomto případě se pořád budeme zabývat mapováním polí jednotlivých typů do nějaké sady dat.

K mockování lze využít některou z již existujících knihoven. V jazyce Javascript by to mohla být například knihovna *GraphQL-tools*, konkrétně její část určená k mockování.

[4]Typicky máme větší šířku pásma pro komunikaci než mezi klientem a serverem

4.3 Relay

Relay je *Javascriptový* framework propojující **GraphQL** a aplikace využívající **React**, více o knihovně React zde 4.3.1. Uživatel deklaruje, která data jsou pro jednotlivé komponenty aplikace potřebná pomocí dotazů GraphQL a Relay obstarává komunikaci s GraphQL endpointem. Relay umožňuje mutaci dat na klientu a serveru pomocí GraphQL mutací a obstarává automatickou konzistenci dat, optimistické změny a zpracování chyb.

Stejně jako GraphQL nebo React je i framework Relay od společnosti Facebook.

4.3.1 React

React je opět *Javascriptová* knihovna pro vytváření uživatelských rozhraní. Hierarchicky strukturované komponenty poskytují metodu pro vykreslení a vracejí virtuální DOM. Virtuální DOM je při každé změně porovnávaný se skutečným DOM a pouze minimální množina manipulací je použita pro manipulaci k aktualizaci aktuálního pohledu aplikace. Protože jakákoliv DOM mutace je pomalá[5], má existence virtuálního DOM dobrý smysl. Virtuální DOM lze jednoduše porovnávat s novějšími verzemi sebe sama a lze ho reflektovat na skutečný DOM.

Rozdělení UI do komponent zjednodušuje znovupoužitelnost. Komponenty pouze definují která data potřebují ke svému vykreslení. Protože je využíván virtuální DOM, což je v podstatě mapující funkce na DOM, lze jednotlivé komponenty vykreslit už na serveru, tedy před odesláním HTML klientovi.

4.3.2 GraphQL kompatibilita

Abychom mohli nad určitým GraphQL endpointem využívat framework Relay, jsou na GraphQL kladeny určité podmínky. Nelze tedy říct, že každy GraphQL endpoint je kompatibilní pro Relay. Tři hlavními předpoklady, na které se v následujících částech podrobněji podíváme. Pro vytvoření kompatibilního GraphQL serveru můžeme využít knihovnu *graphql-relay-js* psanou v jazyce *Javascript*.

4.3.2.1 Odkazování

Existuje mechanismus pro odkazovaní na jednotlivé objekty. Kořenový dotazovací typ poskytuje rozhraní pro dotazování libovolného objektu podle

[5] Rychlost operací lze najít například v práci [8]

jeho unikátního identifikátoru. Takové schéma by mohlo vypadata jako je uvedeno v kódu 4.1.

Kód 4.1: Relay ID

```
interface Node {
	id: ID!
}

type Query {
	node(id: ID!): Node
}
```

V uvedeném příkladu 4.1 by všechny objekty, se kterými chceme pracovat používali rozhraní *Node*. Na straně serveru je potom nutný mechanizmus, který podle identifikátoru pozná typ objektu.

4.3.2.2 Stránkování

Existuje popis jak stránkovat nějaké propojení objektů, tedy například seznam článků. Uvedené schéma by mohlo vypadat jako je v kódu 4.2.

Kód 4.2: Relay stránkování

```
type ArticleConnection {
	edges: [ArticleEdge]
	pageInfo: PageInfo!
}

type ArticleEdge {
	cursor: String!
	node: Article
}

type PageInfo {
	hasNextPage: Boolean!
	hasPreviousPage: Boolean!
	startCursor: String
	endCursor: String
}

type Query {
	mainArticles: Articles
}
```

V uvedeném příkladě je vynechán typ *Article*, který by byl už konkrétním typem s daty. Můžeme si všimnout, že seznamem hlavních článků je propojení na jednotlivé články spolu s informacemi o stránkování.

4.3.2.3 Struktura mutací

Pro mutace existují vstupní a výstupní struktury, které umožňují předvídat jejich chování. Od vstupních a výstupních struktur je požadována pouze existence parametru *clientMutationID*, kde vstupní hodnota v jedné mutaci musí být stejná jako výstupní. Tento parametr potom slouží v Relay pro identifikaci mutace, lze totiž v jednom dotazu poslat více mutací najednou a v samotném GraphQL by neexistoval způsob, jak jednotlivé mutace identifikovat. Bez této vlastnosti by nebylo možné zjistit, která data má Relay aktualizovat u klienta.

Kapitola **5**

Implementace

V implementaci GraphQL se zaměříme na všechny části popsané v ekosystému, viz 1.2. Tedy implementujeme aplikaci poskytující GraphQL endpoint a klientskou aplikaci, která s tímto API endpointem komunikuje. Implementace bude využívat knihoven od společnosti Facebook, které jsou psané pro *Javascript* a v tomto jazyce také budou obě části aplikace napsané.

Pro demonstraci možností využití GraphQL bude aplikační server mapovat požadavky na *Flickr API*[6] a část požadavků namapujeme na interní databázi využívající *mongodb*[7].

V této kapitole si podrobně popíšeme co aplikace implementuje, z jakých částí je složená a jak probíhá komunikace mezi nimi. Podíváme se na použité nástroje a knihovny a popíšeme typový systém GraphQL.

5.1 Popis aplikace

Aplikace demonstruje možné využití GraphQL. Aplikační server komunikuje s interní databází a s API služby *Flicker.com*. Z této služby získává informace o nejnovějších nahraných příspěvcích a k jednotlivým příspěvkům dohledává dodatečné informace (např. kdo je vlastníkem, odkazy na zdroje příspěvku a podobně). Dále komunikuje aplikační server s interní databází implementované v *mongodb* a z ní získává informace o uživatelích aplikace. Každý uživatel má také uloženou množinu příspěvků, které označil za oblíbené. Klient přistupující k tomuto GraphQL endpointu je od služeb třetích stran odstíněn a pracuje pomocí dotazů a mutací s příspěvky a uživateli pomocí jediného rozhraní.

---

[6]Flicker.com je aplikace pro správu a sdílení fotografií.

[7]MongoDB [9] je open—source databáze využívající dokumentově orientovaný datový model

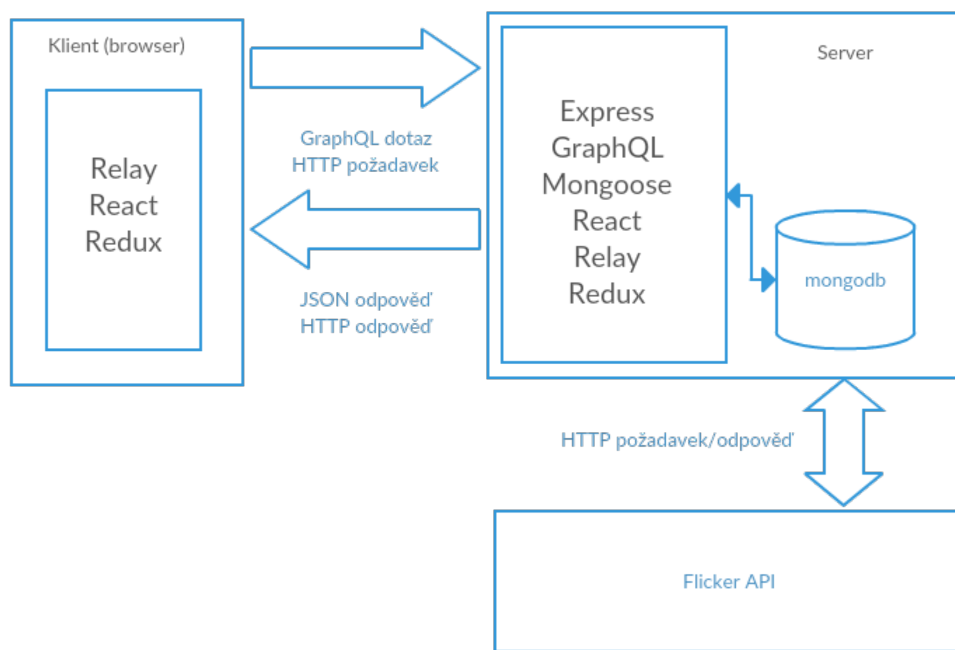
Aplikační server je také schopný generovat klientskou aplikaci, která poskytuje rozhraní pro práci s GraphQL endpointem. V tomto rozhraní je aktuální uživatel zobrazovat příspěvky, které získává právě z API endpointu. S těmito příspěvky potom může pracovat, tedy označovat, které patří z množiny oblíbených a nebo je z této množiny odstraňovat.

Demonstrace také obsahuje interaktivní rozhraní pro testování dotazů nad GraphQL endpointem.

Celá aplikace je implementovaná v jazyce *Javascript* a spouštěná v prostředí *Node.js*. Závislosti aplikace obstarává správce balíčků *npm*.

5.2 Architektura

Na obrázku 5.1 je znázorněna architektura aplikace. Z obrázku si můžeme všimnout, že serverová část komunikuje s API třetí strany (Flicker.com) a k persistenci vlastních dat používá databázi mongodb. Uživatel přistupuje k aplikaci pomocí HTTP požadavků. V klientském prohlížeči potom běží apliace využívající framework Relay a knihovny React + Redux. Framework Relay posílá na server GraphQL dotazy a očekává odpověď ve formátu JSON.



Obrázek 5.1: Architektura aplikace

V následujících částech rozebereme detailněji způsob implementace jed-

notlivých částí aplikace. Tedy konkrétně se podíváme na klientkou část, její před vykreslování na serveru a GraphQL endpoint.

5.3 Server + jeho rozhraní

Serverová aplikace poskytuje jak webové rozhraní, tak přístup ke GraphQL API. K vystavení těchto rozhraní je využit framework **express**.

Při přístupu na GraphQL rozhraní přes webový prohlížeč je klientovy vystaven nástroj na dotazování **GraphiQL**, toto rozhraní a práci s ním popíšeme v sekci 5.5. Obsluhu požadavků nad GraphQL API se stará knihovna **express-graphql**. Ta pomocí definovaného schématu mapuje dotazy na jejich vnitřní implementaci, detailněji se na implementaci schématu aplikace podíváme v následující části 5.4.

Pokud požadavek na server není směrován na GraphQL endpoint, je předpokládáno, že se jedná o požadavek na webový server. Popis implementace nalezneme v sekci 5.6.

5.4 GraphQL

K implementaci GraphQL jsou využity zejména následující knihovny:

- graphql,
- graphql—relay,
- graphiql,
- express—graphql.

GraphQL [graphql-js] Je knihovna která pracuje se schématem aplikace definovaným pomocí objektů. Dotazy potom mapuje na metody těchto objektů a výsledky vystavuje v podobě objektu.

Graphql—relay [graphql-relay] Knihovna s nástroji pro vytváření GraphQL endpointů kompatibilními s frameworkem Relay. Mezi nástroje patří například vytvoření unikátního identifikátoru pro jednotlivé GraphQL typy.

GraphiQL [graphiql] je interaktivní GraphQL IDE fungující v prohlížeči. Více o tomto IDE popíšeme v sekci 5.5.

Tabulka 5.1: Popis typového systému

Název	Mapování	Popis
Application	-	Typ popisující aplikaci, její název a verzi.
Camera	Flicker	Reprezentuje jednotlivé kamery s jejich názvy.
Photo	Flicker	Příspěvek, neboli fotografie. Obsahuje propojení na informace o fotografii a jejího autora.
PhotoInfo	Flicker	Informace o příspěvku.
PhotoOwner	Flicker	Informace o autorovi.
PhotoSize	Flicker	URL Odkaz na příspěvek spolu s informacemi o rozměrech fotografie příspěvku.
User	MongoDB	Interní uživatel aplikace.

Express-graphql [express-graphql] zpracovává HTTP požadavky a extrahuje z nich GraphQL dotazy, které potom překládá na objektové GraphQL schéma. Výslednou odpověď potom převádí do formátu JSON a vrací klientovi.

5.4.1 Typové schéma

Celé typové schéma je možné najít v příloze C. V tabulce 5.1 popíšeme tedy jen funkci jednotlivých typů bez jejich seznamů polí. V tabulce je také popsáno, kam jsou data mapovaná, tedy na službu Flicker nebo vnitřní databázi. Speciálním případem je typ *Application*, jehož data jsou popsaná v konfiguračním souboru. Pro vygenerování typového schématu je implementovaný nástroj, který je popsaný v sekci 5.8. V příloze je také z tohoto schématu vytvořen graf C.1.

5.4.2 Dotazovací metody

Schéma GraphQL obsahuje dva kořenové typy. Jedním je *RootQueryType*, tedy kořenový typ pro dotazování. Tento typ obsahuje následující pole:

- *application*,
- *me*,
- *user*,
- *recentPhotos*,
- *cameraBrands*,
- *node*.

Poslední pole uvedené v seznamu 5.4.2, tedy pole s názvem *node*, je speciální pole přidané pro framework Relay [3]. Pomocí tohoto pole se lze nad kořenovým typem dotazovat na libovolný typ z typového systému pouze pomocí unikátního identifikátoru objektu.

Pole *application* umožňuje dotazovat se na typ *Application*. Tomuto typu předává informace z konfiguračního souboru.

Dále kořenový typ obsahuje dvě pole mapované na interní uživatele, tedy na typ *User*. Těmito poli jsou *me* a *user*. Prvním pole mapuje na uživatele z kontextu komunikace, tedy na přihlášeného uživatele. Druhé pole je mapováno na libovolného uživatele vyhledaného pomocí povinného ID.

Pro práci s příspěvky vystavuje kořenový typ pole *recentPhotos*. Jedná se o propojení na typ *Photo*. Propojení využívá stránkování, tedy tazatel může určit velikost a číslo stránky. Pokud tyto argumenty neuvede, jsou jako výchozí hodnoty nastaveny 15 pro velikost stránky a 1 pro číslo stránky.

Podobně jako se lze dotazovat na příspěvky, neboli fotografie, lze využít pole *cameraBrands* pro dotaz na seznam názvů fotoaparátů. Tedy toto pole mapuje na typ *Camera*.

5.4.3 Mutace

Druhým kořenovým typem schématu je typ se seznamem mutací. V implementaci nalezneme pouze jednu mutaci a to pro přidání/smazání příspěvku z množiny oblíbených příspěvků aktuálního uživatele v kontextu komunikace. Kvůli kompatibilitě s frameworkem Relay je vstup i výstup mutace určen typem, který obsahuje pole *clientMutationId*. Toto pole určuje parametrem klient při vykonávání mutace a výstup z mutace má hodnotu tohoto pole totožnou s jeho vstupní hodnotou. Relay toto vyžaduje, aby dokázal identifikovat mutaci a věděl, které data u sebe v úložišti aktualizovat.

Kořenový typ mutace tedy obsahuje jedno pole s názvem *toggleFollow*. Vstupními parametry pro operaci mutace jsou *clientMutationId* a *photo*. Tedy identifikace mutace a fotografie. Výstupem z mutace je opět klientská identifikace mutace spolu s aktualizovanými daty uživatele.

5.4.4 Způsob mapování na Flicker

Pokud je na GraphQL endpoint položen dotaz, který vystavuje některý z typů mapovaných na službu Flicker (které to jsou je uvedeno v tabulce 5.1), provede server dotaz na API služby Flicker. Toto dotazování není v aplikaci nijak kešované a dochází k němu při každém dotazu. Protože API služby Flicker není konzistentní a například seznam posledních příspěvků

je stránkovaný, zatím co seznam fotoaparátů stránkovaný není, obsahuje implementace možnost simulace stránkování. Tedy pokud se uživatel dotáže na seznam fotoaparátů, server se dotáže externího API na kompletní seznam a data zpracuje. Uživateli potom vrací jen podmnožinu těchto dat, tato podmnožina je určená dotázanou stránkou a velikostí stránky.

Odpovědi z Flicker API jsou tedy mapovány na interní typy typového systému aplikace. Tyto typy můžou obsahovat opět některá pole o které je nutné opět provést další dotaz na externí API. Pro názornost uvedeme dotaz uvedený v kódu 5.1. Tento dotaz vybírá poslední příspěvky na straně 2. Z příspěvků bere pole *flickerId*, *title* a vlastníka, tedy *owner*. Server při takovém dotazu první požádá externí API o seznam posledních příspěvků s příslušným stránkováním. V odpovědi chybí ale informace o vlastníkovi. Server proto pošle další požadavek na externí API a to na uživatele, jehož ID má uvedené v předchozí odpovědi. Oba tyto dotazy jsou mapovány do jedné odpovědi.

Kód 5.1: Dotaz využívající externí API

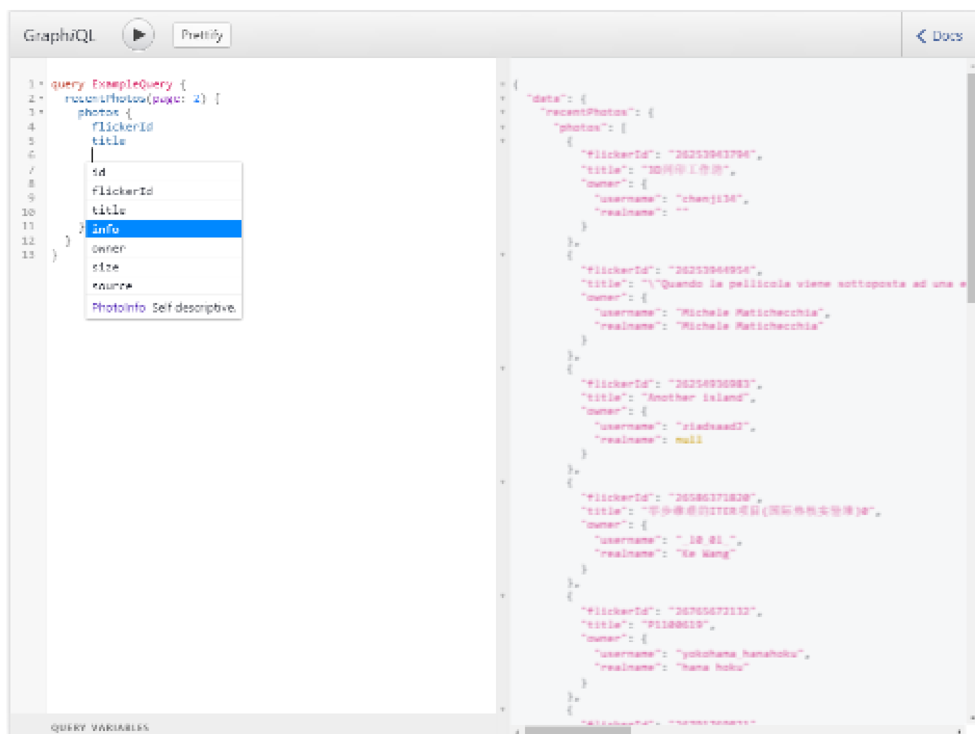
```
query ExampleQuery {
  recentPhotos(page: 2) {
    photos {
      flickerId
      title
      owner {
        username
        realname
      }
    }
  }
}
```

5.4.5 Způsob mapování na interní databázi

Typ *User* uvedený v tabulce 5.1 je jediným typem mapovaným na data v databázi. Implementace umožňuje dotazovat uživatele pouze z aktuálního kontextu. Pokud žádný takový uživatel v databázi není nalezen, je vytvořený nový a jeho identity je vrácena v odpovědi dotazu. V opačném případě dochází pouze k vrácení identity. Na databázi je také mapována jediná mutace popsaná v implementaci. Tato mutace a její funkce je popsaná v sekci 5.4.3.

5.5 GraphiQL

Při přístupu na GraphQL endpoint pomocí prohlížeče je klientovi vystaven nástroj pro práci s GraphQL. Jedná se o IDE vytvořené pomocí knihovny *GraphiQL* a je znázorněné na obrázku 5.2. Toto IDE pracuje s typovým systémem aplikačního serveru, takže dokáže uživateli napovídat i v kontextu tohoto typového systému. Podle úrovně zanoření nabízí možnosti dotazování určené pro aktuální úroveň zanoření. Pomocí IDE lze také zobrazovat dokumentaci příslušného typového systému.



Obrázek 5.2: GraphiQL IDE

Jak lze vidět na obrázku 5.2, levá strana IDE je určena pro zadávání dotazu. Na pravé straně je potom zobrazena odpověď k dotazu.

Toto rozhraní je využitelné jak při implementaci GraphQL endpointu, tak také při vývoji klientské aplikace. Při vývoji GraphQL endpointu dokážeme jednoduše vyvolávat části kódu a ihned vidíme odpověď. V klientské části potom dotaz nebo jeho části můžeme rovnou přidat do implementace.

5.6 Webový server

Webový server poskytuje klientskou aplikaci využívající framework **Relay** spolu s knihovnou **React**. Knihovna React poskytuje vykreslení dat do HTML. Skládá se z komponent reprezentovanými vlastními tagy HTML. React udržuje tyto komponenty jako virtuální DOM který je mapován na DOM. Při vykreslení dochází k porovnání právě virtuálního DOMu s tím skutečným a převedeny do DOMu jsou pouze rozdílnosti. Při prvním vykreslení je tedy do DOMu převedena celá aplikace využívající React. Více o klientské aplikaci v sekci 5.7.

Aby klient dostával už vykreslené kompletní HTML a nemusel provádět první vykreslování, je tato operace přesunuta na server. To využijeme například při indexování HTML roboty vyhledávačů.

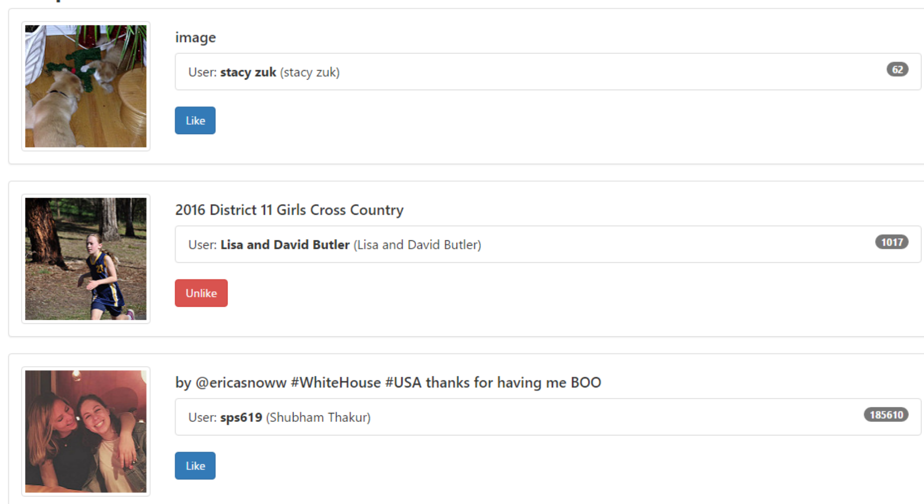
Před vykreslením aplikace na serveru je tedy nutné provést první komunikaci s GraphQL API a získat potřebná data pro aplikaci. K tomuto účelu je využita knihovna *react—relay*. Framework Relay z komponent aplikace vytáhne všechny fragmenty potřebné pro složení GraphQL dotazu. Tyto fragmenty jsou součástí komponent a více jsou popsány v sekci 5.7. S odpovědí získanými od GraphQL endpointu je poté aplikace vykreslená do řetězce se zbytkem HTML, o to se stará knihovna *react—dom*. Celé HTML je poté odeslané v HTTP odpovědi zpět klientovi. Aplikace běžící u klienta provede překreslení znovu, tentokrát ale nemusí pro první dotaz posílat na GraphQL endpoint, protože všechny potřebná data již framework Relay má uložené.

5.7 Klientská aplikace

Jak bylo zmíněno v předchozí sekci o webovém serveru 5.6, využívá klientská aplikace knihovnu **React**. Celá aplikace je tedy rozdělená do komponent a výsledný pohled je z těchto komponent složen a vykreslen do HTML.

V aplikaci je implementován jeden pohled nad daty. Prvním GraphQL dotazem jsou vytaženy informace o aplikaci, uživateli a poslední příspěvky. Příspěvky jsou stránkované a číslo stránky je určené parametrem v URL. O stránkování se tedy nestará framework Relay a nemusí GraphQL server podporovat stránkování, které framework Relay požaduje. I když aplikace implementuje pouze jeden pohled nad daty, skládá se z dvou rout. Indexová, tedy defaultní routa a potom routa, určující číslo stránky, které je předáno jako parametr do komponenty vykreslující pohled nad daty.

Aplikace ve svém pohledu vykreslí jednotlivé příspěvky, u kterých je možnost přidání/odebrání označení oblíbenosti příspěvku. Tato akce vy-



Obrázek 5.3: Vykreslená klientská aplikace

užívá mutace *toggleFolow*, viz sekce 5.4.3. Výsledná podoba vykreslené aplikace je znázorněna na obrázku 5.3.

Komponenty aplikace, které chtějí využívat některá data získané z GraphQL API, musí deklarovat, která data chtějí a to pomocí fragmentů. Příklad takové deklarace může být kód 5.2. Konkrétně kód deklaruje, že komponenta **Page** vyžaduje fragment nad typem *Application* a žádá o pole s jménem *name* a *version*. O zbytek se stará už samotný framework Relay. Tedy poskládá celkový dotaz pro GraphQL a zajistí, že každá aplikace dostane právě ta data o která požádá. Každá routa aplikace má přiřazený seznam možných fragmentů, tedy v příkladě kódu 5.2 bychom v takovém seznamu měli fragment s názvem *application*. Pokud o stejný fragment žádá více komponent v aplikaci, framework Relay zajištuje, že jsou z API získány pouze potřebná data a to jedním dotazem.

Kód 5.2: Deklarace fragmentů

```
1 export default createContainer(Page, {
2   fragments: {
3     application: () => Relay.QL`
4       fragment on Application {
5         name
6         version
7       }
8     `,
9     photos: () => Relay.QL`
```

```
10        fragment on RecentPhotoConnection {
11          photos {
12            ${PhotoPreview.getFragment('photo')}
13          }
14        }
15      `
16    }
17  });
```

Komponenty aplikace, které obsahují zanořené komponenty využívající data z GraphQL endpointu, musí o těchto datech dát vědět i ve své deklaraci fragmentů. Tedy do jednotlivých fragmentů musí připojit fragmenty z komponent, jejichž jsou rodiči. Tento příklad můžeme vidět v kódu 5.2, kde fragment s názvem *photos* obsahuje fragment z komponenty *PhotoPreview*, s názvem *photo*. Pod komponenta *PhotoPreview* si potom potřebná data určuje opět pomocí fragmentů, v tomto případě musí definovat fragment *photo*.

5.8 Vývojové nástroje

Celá aplikace je psaná ve standardu **ECMAScript 2015**. Tento standard není v prohlížečích podporován a je nutné jej kompilovat pomocí nástroje *babel* do Javascriptu, který dokáže prohlížeč zpracovat. Součástí implementace je proto také vytvoření automatizačních úkolů, které se starají o kompilaci celé aplikace a to pomocí nástroje *gulp* a *webpack*. Při kompilaci dochází k vytvoření minifikované aplikace pro prohlížeč.

Aplikace pracuje ve dvou režimech. Jeden je produkční, kdy ke kompilaci klientské aplikace dojde pouze jednou a je spuštěn aplikační server. Druhý je vývojový režim, který při změně souborů vyvolá kompilaci a změny odešle všem připojeným klientům. Klientské aplikace jsou tedy překreslovány bez nutnosti obnovovat ručně aplikaci.

Mezi vývojové nástroje také patří nástroj pro kontrolu a opravu syntaxe ESLint. Využití tohoto nástroje zajišťuje konzistenci kódu napříč aplikací pomocí předem definovaných pravidel. Konkrétní pravidla lze najít v implementaci, v souboru *ėslintrc*.

Neméně důležitým nástrojem pro vývoj je generátor GraphQL schémat z aplikace. Tento nástroj vygeneruje dva druhy schémat. Jedno schéma je ve formátu JSON a obsahuje všechny potřebné informace pro klientské aplikace. Tedy na základě tohoto schématu jsou klientské aplikace schopni provádět validaci dotazů ještě před odesláním. Druhé vygenerované schéma s koncovkou *ġraphql* slouží čistě pro vývojáře jako reference.

Obsahuje pouze definici všech typů bez dokumentace. Takové schéma můžeme vidět v příloze v kódu C.1.

Kapitola **6**

Možné využití implementace

Implementace vytváří základ pro aplikace, které využívají moderní knihovny jako je *React* a *Redux*. Zároveň implementace řeší vykreslování již na serverové části a celou klientskou část aplikace kompiluje do Javascriptu pro prohlížeče. V implementaci je základ pro GraphQL server a pomocí frameworku Relay je vyřešena komunikace s tímto typem API. Pro další vývoj implementace připravuje nástroje, které se starají o kompilaci a kontrolu syntaxe. Implementace jako celek lze využít jako základ pro moderní webovou aplikaci. Při vývoji takové aplikace se vývojář může soustředit pouze na klientskou aplikaci či rozšiřování a mapování typového systému GraphQL na data.

Závěr

Cílem práce bylo popsat technologii GraphQL a podrobit ji porovnání s tradičním způsobem komunikace klient—server, tedy přístupem typu REST. Dalším cílem bylo tuto technologii demonstrovat v praxi, tedy vytvořit aplikaci používající GraphQL.

Ačkoliv si osobně nemyslím, že by technologie GraphQL měla plně nahradit REST, rozebral jsem v práci důkladně jednotlivé rozdíly v obou těchto technologiích. Věřím, že uvedené rozdílnosti můžou přispět k rozhodnutí, kterou technologii použít při implementaci nějaké reálné aplikace. Využití GraphQL technologie vidím na místech, kde klientská aplikace musí fungovat s co nejkratší a nejrychlejší komunikací se serverem. Typicky tedy mobilní aplikace. Naopak využití klasického přístupu, tedy REST, bych volil na komunikaci mezi jednotlivými serverovými aplikacemi, teda tam, kde chceme mít větší kontrolu nad kešováním komunikace a kde nám nevadí více požadavků mezi servery.

Výstupem práce je také implementace aplikace, která vystavuje jak serverovou část, tak část pro prohlížeče a obstarává jejich komunikaci. Tato implementace může být vhodným stavebním kamenem pro složitější aplikace, využívající popsané technologie.

Osobně pro mě měla práce veliký přínos. Podrobně jsem se seznámil s technologií GraphQL a s frameworkem Relay, který právě s GraphQL velice souvisí. Aplikace které jsem do této doby v mém profesním životě vytvářel, využívali přístup REST, se kterým jsem v této práci také GraphQL porovnával. Myslím si, že poznatky z této práce povedou v mém případě k náhradě REST přístupu právě technologií GraphQL ve spoustě mnou napsaných aplikacích.

Literatura

[1] Facebook: GraphQL. 2015. Dostupné z: http://facebook.github.io/graphql/

[2] Gruber, J.: Markdown. 2004. Dostupné z: http://daringfireball.net/projects/markdown/

[3] Facebook: Relay | A JavaScript framework for building data-driven React applications. 2015. Dostupné z: https://facebook.github.io/relay/

[4] Facebook GraphQL: graphql/graphql-js. 2015. Dostupné z: https://github.com/graphql/graphql-js

[5] Fielding, R. T.: *REST: Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000. Dostupné z: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[6] Hunt, T.: 2014. Dostupné z: https://www.troyhunt.com/your-api-versioning-is-wrong-which-is/

[7] Smith, N.: GraphQL Visualizer. https://github.com/NathanRSmith/graphql-visualizer, 2016.

[8] Raento, M.: innerHTML vs appendNode vs DocumentFragment - Optimizing bulk DOM operations for mobile. 2014. Dostupné z: http://blog.mikie.iki.fi/2014/05/innerhtml-vs-appendnode-vs.html

[9] Chodorow, K.; Dirolf, M.: *MongoDB - The Definitive Guide: Powerful and Scalable Data Storage.* O'Reilly, 2010, ISBN 978-1-449-38156-1, I-XVII, 1-193 s.

Příloha **A**

Seznam použitých zkratek

API Application Programming Interface

REST Representational State Transfer

CRUD Create Read Update Delete

DOM Document Object Model

UI User Interface

IDE Integrated Development Environment

Příloha **B**

Instalace a návod k použití

Ke spuštění aplikace jsou vyžadovány následující nástroje:

- **node.js** verze 5 a vyšší,
- **npm** verze 3 a vyšší,
- **gulp** instalace pomocí npm (*npm install -g gulp*).

Zkopírujte adresář s implementací z cd a nainstalujte potřebné knihovny, viz kód B.1.

Kód B.1: Instalace závislostí

```
1 npm install
```

Aplikace vyžaduje běžící **MongoDB**, připojení k němu lze upravit v konfiguračním souboru v adresáři implementace *src/server/config.js*.

B.1 Zapnutí vývojového režimu

Pro zapnutí vývojového režimu použijte příkaze B.2. Aplikace je v základním nastavení po spuštění dostupná na portu **8000**. GraphiQL IDE je dostupné na stejném portu na adrese */graphql*. Vývojový režim hlídá změny souborů a při každé změně znovu překompiluje kód pro Javascript prohlížečů.

Kód B.2: Zapnutí vývojového režimu

```
1 gulp
```

B.2 Zapnutí produkčního režimu

Produkční režim zapnutý příkazem B.3 zkompiluje klientské knihovny jednou při spuštění. Aplikace je opět dostupná v základním nastavení na portu **8000**.

Kód B.3: Zapnutí produkčního režimu

```
1 gulp -p
```

Příloha C

Typový systém implementované aplikace

V kódu C.1 je vypsán v čitelné podobě GraphQL typový systém. Tento systém neobsahuje popis dokumentace, ale pouze jednotlivé typy a jejich pole. V obrázku C.1 je znázorněna potom vizualizace výsledného typového systému implementované aplikace.

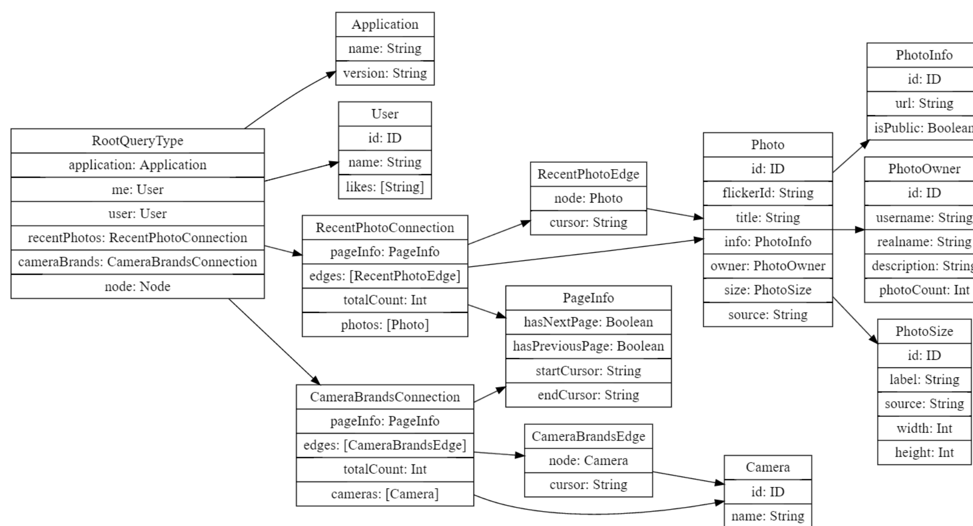
Kód C.1: Typový systém

```
schema {
  query: RootQueryType
  mutation: RootMutation
}

type Application {
  name: String
  version: String
}

type Camera {
  id: ID!
  name: String
}

type CameraBrandsConnection {
  pageInfo: PageInfo!
  edges: [CameraBrandsEdge]
  totalCount: Int
  cameras: [Camera]
```

```
21 }
22
23 type CameraBrandsEdge {
24   node: Camera
25   cursor: String!
26 }
27
28 input LikeInputType {
29   user: String!
30   photo: String!
31   clientMutationId: String
32 }
33
34 type LikeOutputPayload {
35   clientMutationId: String
36   user: User
37 }
38
39 interface Node {
40   id: ID!
41 }
42
43 type PageInfo {
44   hasNextPage: Boolean!
45   hasPreviousPage: Boolean!
46   startCursor: String
47   endCursor: String
48 }
49
50 type Photo {
51   id: ID!
52   flickerId: String
53   title: String
54   info: PhotoInfo
55   owner: PhotoOwner
56   size: PhotoSize
57   source: String
58 }
59
60 type PhotoInfo {
61   id: ID!
```

```
62   url: String
63   isPublic: Boolean
64 }
65
66 type PhotoOwner {
67   id: ID!
68   username: String
69   realname: String
70   description: String
71   photoCount: Int
72 }
73
74 type PhotoSize {
75   id: ID!
76   label: String
77   source: String
78   width: Int
79   height: Int
80 }
81
82 type RecentPhotoConnection {
83   pageInfo: PageInfo!
84   edges: [RecentPhotoEdge]
85   totalCount: Int
86   photos: [Photo]
87 }
88
89 type RecentPhotoEdge {
90   node: Photo
91   cursor: String!
92 }
93
94 type RootMutation {
95   toggleFollow(input: LikeInputType): LikeOutputPayload
96 }
97
98 type RootQueryType {
99   application: Application
100  me: User
101  user(id: String): User
102  recentPhotos(pageSize: Int = 15, page: Int = 1):
```

```
        RecentPhotoConnection
103   cameraBrands(pageSize: Int = 15, page: Int = 1):
        CameraBrandsConnection
104   node(id: ID!): Node
105 }
106
107 type User implements Node {
108   id: ID!
109   name: String
110   likes: [String]
111 }
```



Obrázek C.1: Vizualizace typového systému

Příloha **D**

Obsah přiloženého CD

```
readme.txt ........................ stručný popis obsahu CD
src ............................... zdrojové kódy implementace
text .............................. text práce
    thesis.pdf .................... text práce ve formátu PDF
```