GraphQL

GraphQL Foundation

Václav Sobotka, 3. ročník, 5. semestr

2. prosince 2019

1 Charakteristika

GraphQL je technologie založená na práci s webovými API s pomocí k tomu určeného dotazovacího jazyka. Specifikace GraphQL popisuje, co musí splňovat a poskytovat strana serveru a jak následně mohou její služby využívat (právě prostřednictvím již zmiňovaného dotazovacího jazyka) její klienti. Právě na ty je celá technologie orientovaná především, GraphQL vznikl z potřeby efektivního a škálovatelného přístupu k API ze strany klientů.

2 Vývoj technologie

GraphQL byl vyvinut v roce 2012 pro interní potřeby Facebooku. V roce 2015 uvolnil Facebook GraphQL veřejnosti, technologie je jako OpenSource aktivně rozvíjena. Poslední release verze specifikace byla vydána 10. června 2018.[Fou18] GraphQL se od svého zveřejnění těší rostoucí popularitě.

Zajímavý je samotný původ názvu odkazující na grafy. Navzdory dojmu, který může název vyvolávat, není GraphQL v žádném zásadnějším ohledu vázaný na grafy (jako třeba grafové databáze). Podle autorů má zvolený název reflektovat jejich chápání a přístup k objektům a jejich vzájemným vztahům.

3 Cíle

Hlavním cílem GraphQL je efektivní a škálovatelný standard webového API. Lee Byron, jeden z autorů technologie, vytyčil za cíl rozšířit GraphQL jako „všudypřítomný“ API standard napříč Internetem. [Ant18]

4 Popis

Jak už bylo zmíněno, GraphQL se zaměřuje na služby poskytované webovým API a na to, co musí poskytovat strana serveru, aby mohly klientské aplikace její API využívat s pomocí GraphQL dotazovacího jazyka. Z čeho ale pramení potřeba vytvářet API dotazovatelné ve stylu GraphQL?

Hlavní motivací pro vznik GraphQL byl vývoj nativních mobilních aplikací Facebooku, respektive problémy pramenící z různorodosti dat, se kterými tyto aplikace pracují.[Byr15] Rozhodnutí vyvinout nové řešení namísto využití standardních postupů ale muselo mít rozumný důvod – tímto důvodem byla neefektivita a neškálovatelnost tradičních přístupů.

Aby bylo jasné, v čem spočívá užitečnost přístupu GraphQL, pokusím se popsat problémy, na které by při využití standardních postupů muselo dojít. Pro ilustraci použiji API typu REST.

4.1 Nedostatky RESTu

Přístup ke zdrojům u RESTového API je řešen skrze přístup na URL adresu příslušnou danému zdroji. [res19] Tímto zdrojem může být např. objekt popisující knihu uložený v databázi serverové aplikace. Pokud potřebuje klientská aplikace získat tento objekt, dotáže se serverového API na dané URL a obdrží ho jako odpověď.

Co jsou tedy ty nejvýraznější důvody, proč může být problém výše popisovaným způsobem efektivně pracovat s takto zpřístupněnými zdroji?

4.1.1 Přebytečnost informací ve zdroji

Uvažme následující situaci – klientská aplikace potřebuje informace obsažené v takto zpřístupněném zdroji, ale vystačí si např. jen s několika málo atributy výsledného objektu.

První přístup k tomuto problému je ignorovat jej – klientská aplikace bude pracovat s celým objektem, i když z něj potřebuje jen část. Je zjevné, že takovýto přístup je z podstaty neefektivní – ať už jen proto, že bude muset klientská aplikace zbytečně v paměti držet nepotřebné údaje, nebo z toho důvodu, že se bude zbytečně přenášet (po síti) informace, která nebude mít využití. Co se škálovatelnosti týká, situace nebude o nic lepší.

Alternativně jde přistoupit k věci tak, že rozšíříme API o možnost vyžádat si pouze relevantní podčást požadovaného objektu. Tímto krokem vyřešíme dva největší problémy předchozího přístupu – nebudeme zatíženi tím, co nepotřebujeme. Zároveň ale vznikne problém jiný. Pokud pomineme to, že se tímto odkloníme od čistého REST přístupu (což není nijak neobvyklé), dojdeme velice brzo k nemožnosti tímto způsobem škálovat. Pokud bychom totiž pro každý podobný případ rozšiřovali API, dostaneme se velice brzo do dlouhodobě neudržitelného stavu.

4.1.2 Několikanásobné dotazy

Nezřídkakdy může chtít klientská aplikace agregovat několik zdrojů do jednoho výsledku. Za tímto účelem potom musí provést několik separátních dotazů a výsledek si následně sloučit sama.[Fou19a] Vzhledem k tomu, že provedení takového dotazu není z principu levná operace, bylo by záhodno umět dostat agregovanou odpověď v jednom jediném dotazu na API.

Jako řešení se opět jeví možnost rozšířit API na straně serveru, to ale není z již výše zmiňovaných důvodů dobrým východiskem.

4.1.3 Netransparentnost REST API

Další nevýhodou RESTového API je nemožnost zvenku zjistit, jaké zdroje nabízí. Pokud máme k dispozici jen kořenové URL, bez znalosti struktury API nebo konkrétních URL adres zdrojů z něj příliš mnoho nedostaneme. To sice mnohdy nepředstavuje problém (např. vývojář, který má přístup jak k serverové, tak klientské aplikaci a informace o API si může vyčíst z kódu), pokud bychom ale chtěli např. automaticky strojově pracovat s API, neumíme zvenku o daném API prakticky nic říct.

4.2 Jak GraphQL funguje

Základní myšlenkou GraphQL je poskytnout API, na které se mohou klienti dotazovat podobným způsobem, jako to dělají databázoví klienti vůči databázi pomocí SQL. Dalším důležitým prvkem je možnost jasně specifikovat, jaké dotazy je a není možné po API požadovat. [Fou19d]

Pokud je možné s API pracovat tímto způsobem, okamžitě odpadá např. problém s redundantními atributy ve výsledných objektech – stačí jednoduše zvolit takový dotaz, který

nepotřebné atributy nepožaduje.[Fou19a] Vyhneme se tím pádem všem zmiňovaným problémům – neefektivitě přenosu požadovaných dat, nutnosti držet nevyužívanou informaci v paměti, nebudeme muset ani rozšiřovat API pro každý podobný případ, abychom se redundanci vyhnuli.

Podobně můžeme s výhodou využít dotazovací jazyk k agregaci více zdrojů do jednoho výsledku – a to v jednom jediném dotazu. Opět se tím vyhneme nutnosti manipulovat s výsledky jednotlivých dotazů na straně klienta a především potřebě volat více separátních požadavků na API serveru.

Posledním zmiňovaným bodem je možnost (respektive povinnost) jasné deklarace toho, jak může vypadat dobře utvořený dotaz na API dotazovatelné pomocí GraphQL. Každé GraphQL dotazovatelné API musí mít definované tzv. *schema*[Fou19d] – předpis, ve kterém je deklarováno, jaké dotazy je možné API zasílat, jak jsou tyto dotazy parametrizované, s jakými typy dotazy manipulují, jakou strukturu (atributy) objekty těchto typů mají... Tím se alespoň v nástinu dostáváme k řešení posledního zmiňovaného problému – netransparentnosti služeb poskytovaných API. Jedním z požadavků na GraphQL API je totiž právě zpřístupnění svého schema skrze API.

Po hrubém přehledu principů, na kterých GraphQL dotazovatelná API stojí, se je pokusím přiblížit ve větším detailu – především koncepty dotazů, API schema a to, jak takové GraphQL API vypadá.

4.2.1 GraphQL API

Velkým rozdílem mezi tradičními API a GraphQL API je počet *endpointů*.[Ant18] Zatímco např. REST API poskytuje endpoint pro každý zdroj zvlášť, GraphQL API má endpoint jediný. Skrze ten přijímá požadavky formulované v dotazovacím jazyce a tímto způsobem zpřístupňuje všechny zdroje svým klientům. Požadované zdroje tak nejsou popisovány volanou URL (endpointem, resp. případnou parametrizací), ale k tomu určeným dotazovacím jazykem. To přináší již zmiňovanou flexibilitu ve formátu požadované informace – právě možnost specifikovat, co přesně bude obsahovat odpověď serveru[Fou19a], tvoří stěžejní výhodu oproti fixnímu formátu odpovědi.

4.2.2 Dotazy v GraphQL

Prozatím byl dotazovací jazyk zmiňován pouze jako mechanismus ke zformulování požadavků na API bez uvedení jakékoli podoby. Jako nejlepší ilustrace zmiňovaného dotazovacího jazyka poslouží uvedení jednoduchého dotazu.

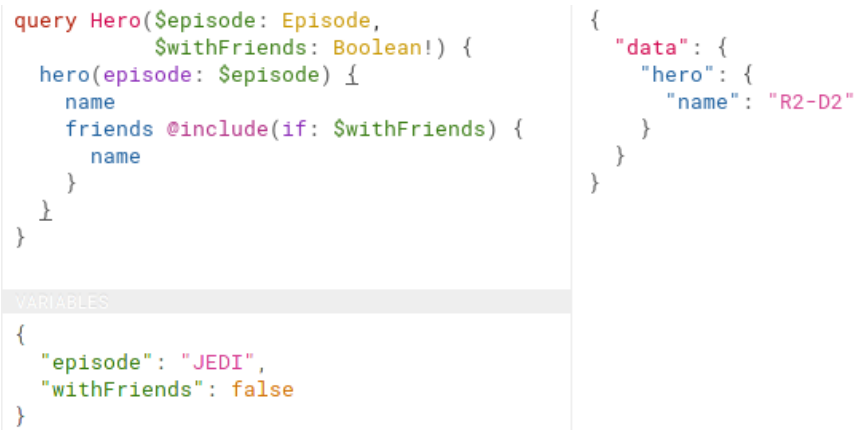
Obrázek 1: Jednoduchý dotaz na GraphQL API (vlevo) a odpověď (vpravo)

Zdroj: [Fou19a]
Odpověď od GraphQL API přesně kopíruje strukturu požadavku z dotazu

Důležitým postřehem je nápadná podobnost mezi formátem dotazu a formátem odpovědi. Klient dostal v odpovědi přesně to, co potřeboval – nic víc, nic míň. Dotaz v příkladu zatím ukazoval jen možnost výběru atributů na objektu, tím samozřejmě možnosti GraphQL ani zdaleka nekončí.

Druhý příklad ukazuje některé další možnosti GraphQL:

Obrázek 2: Komplexnější dotaz na GraphQL API (vlevo) a odpověď (vpravo)



Zdroj: [Fou19a]
Dotazy je možné předdefinovat a parametrizovat - dokonce je možné podmínečně zahrnout určitou podčást dotazu

- Pojmenování dotazu

  Narozdíl od prvního příkladu používá druhý dotaz plnou syntax – ta mj. uvádí, o jaký druh dotazu jde (v tomto případě „query“ – existuje více možností) a umožňuje pojmenování dotazu. Jméno dotazu se následně dá využívat např. při logování pro jednodušší lokalizaci chyb.

- Parametrizace dotazu

  Dotaz využívá dva parametry – „episode“ a „withFriends“. Výsledek volání se tak odvíjí od externě dodaných argumentů (mapa hodnot pod dotazem). GraphQL navíc požaduje specifikaci typu pro parametry dotazu – v našem příkladu jde o typy „Episode“ a „Boolean“ (s modifikátorem „!“ – jedná se o povinný parametr). To, s jakými typy mohou dotazy pracovat, zadává již zmiňované schema.[Fou19d]

- Podmínečné zahrnutí atributů

  GraphQL umožňuje podmínečné zahrnutí podobjektů do požadavku – tato možnost je užitečná např. v případě, kdy chceme na základě pravdivostní proměnné pracovat s kompletním objektem včetně detailů, respektive pouze s několika málo základními atributy. V příkladu jde o podmínečné zahrnutí atributu „friends“ – toho je dosaženo pomocí tzv. *direktivy* @include.

Kromě možností zmiňovaných v příkladech umožňuje GraphQL poskytování defaultních hodnot proměnných, znovuvyužívání podčástí dotazů formou tzv. *fragmentů* a v neposlední řadě tzv. *mutace* – typ dotazu (na stejné úrovni jako „query“) určený k vytváření nebo upravování zdrojů.

4.2.3 API schema

Každé GraphQL API je popsáno schematem. Schema obsahuje popisy typů[Fou19d] – strukturu, jejich atributy (může jít opět o složené typy nebo některé z předdefinovaných primitivních, tzv. skalárních, typů), případné argumenty atributů... GraphQL umožňuje i vytváření abstraktních typů. Prvním druhem abstraktních typů jsou tzv. *rozhraní* – ta definují, jaké atributy musí mít typy, které toto rozhraní implementují. Druhým abstraktním typem je *sjednocení* – to umožňuje zastřešit pod jeden typ několik (i potenciálně

Obrázek 3: Definice složeného typu

```
type Starship {
  id: ID!
  name: String!
  length(unit: LengthUnit = METER): Float
}
```

Zdroj: [Fou19d]
Příklad ukazuje definici složeného typu „Starship“ – typ obsahuje povinné (modifikátor „!“) atributy „id“ a „name“ a jeden nepovinný atribut „length“. Ten nepovinně přijímá argument „unit“ typu „LengthUnit“ s defaultní hodnotou „METER“.

velice rozdílných) typů. Taková možnost může přijít vhod např. pokud chceme provádět již diskutovanou agregaci více zdrojů v jednom dotazu.[Fou19d]

Kromě definic používaných typů musí schema obsahovat definici speciálního typu „Query“.[Fou19d] Tento typ slouží jako vstupní bod pro zpracování požadavků – definuje, co, pod jakými jmény, typovými omezeními a s jakými argumenty smí být na daném GraphQL API dotazováno.

Obrázek 4: Definice speciálního typu Query

```
type Query {
  hero(episode: Episode): Character
  droid(id: ID!): Droid
}
```

Zdroj: [Fou19d]
Typ Query definuje, co všechno je možné na daném GraphQL API dotazovat

4.2.4 Zpracování dotazů na straně serveru

Doposud byl kladen důraz především na to, jak se zvenku pracuje s GraphQL API – tedy jak vypadají dotazy, čeho s nimi lze dosáhnout a co je výhoda takového rozhraní oproti jiným přístupům. Jak ale probíhá zpracování takto „forumlovaných“ dotazů na straně serveru?

Prvním krokem před samotným vyhodnocením dotazu je jeho validace. Ta proběhne na základě obecných syntaktických pravidel (ta jsou popsána v rámci specifikace, obecný formát je podobný formátu JSON), následně je dotaz podroben kontrole vůči schematu daného API. Až poté se přistupuje k samotnému vyhodnocování dotazu.

Klíčová myšlenka vyhodnocování dotazů spočívá v zastoupení atributů (ať už na úrovni kořene dotazu, nebo v jeho nejzazším listu) funkcemi – tzv. *resolvery*.[Fou19c] Vyhodnocování potom probíhá tak, že je v počátku zavolán tzv. *kořenový resolver* (resolver speciálního typu „Query“) a ten volá resolvery všech svých podčástí, které dotaz požadoval. Takto postupují i další resolvery – volají resolvery svých požadovaných podčástí. Rekurze se pak zastaví až na resolverech, které pracují nad primitivními (skalárními) typy.[Fou19c] Resolvery dostávají navíc k dispozici objekt kontextu (v tom si mohou předávat informace typu připojení k databázi atp.) a argumenty předané atributu v rámci dotazu pro zohlednění během resolvování.

Pokud má tedy serverová aplikace poskytovat GraphQL API svým klientům, je v zásadě zapotřebí implementovat dvě stěžejní věci – GraphQL schema a systém resolverů pokrývající potřeby tohoto schematu.

4.3 Možnost integrace do existujícího projektu

Výhodou GraphQL je možnost integrace do již existujících aplikací. Aplikace může bez problému nabízet jak REST, tak GraphQL API. Navíc by nemělo být potřeba provádět zásadní změny na úrovni aplikační logiky – ta by měla fungovat jako zdroj informací pro resolvery. Integrace GraphQL tak spočívá především v definici schematu, systému resolverů a jejich napojení na už existující aplikační logiku. Vzhledem k tomu, že již existují implementace knihoven pro GraphQL v mnoha jazycích[Fou19b] (Java, Python, C, Javascript, Go...), není obvykle zapotřebí začínat „od píky".

5 Hodnocení

Po seznámení s klíčovými koncepty a okolnostmi kolem vzniku GraphQL musím technologii jako takovou hodnotit jednoznačně kladně. Zaujala mě motivace vzniku GraphQL pramenící z problémů, se kterými se (byť pravděpodobně ve zlomkovém měřítku oproti prostředí Facebooku) sám často setkávám – např. již zmiňovaná redundance informací. Za obzvlášť přesvědčivý argument ve prospěch GraphQL považuji fakt, že byl vyvinut a úspěšně využíván jako interní nástroj ve Facebooku – to ukazuje na efektivitu a škálovatelnost GraphQL jako přístupu ke tvorbě API. Na stranu druhou zůstává otázkou, kdy se doopravdy vyplatí GraphQL využít – pro množství aplikací totiž nemusí přinášet vzniklá flexibilita zásadní výhody a potom se nemusí vyplatit investovat čas do implementace.

6 Dublin core metadata

```
<dc:title>GraphQL</dc:title>
<dc:creator>Václav Sobotka</dc:creator>
<dc:description>
   Přehledová esej o GraphQL -- technologii pro tvorbu webových API podporujících
   požadavky formulované dotazovacím jazykem.
</dc:description>
<dc:date>28. října 2019</dc:date>
<dc:type>Text</dc:type>
<dc:format>pdf</dc:format>
<dc:language>CZ</dc:language>
```

Reference

[Ant18] Art Anthony. Is graphql moving toward ubiquity? https://nordicapis.com/is-graphql-moving-toward-ubiquity/, 2018.

[Byr15] Lee Byron. Graphql: A data query language. https://engineering.fb.com/core-data/graphql-a-data-query-language/, 2015.

[Fou18] GraphQL Foundation. GraphQL specifikace. https://graphql.github.io/graphql-spec/June2018/, 2018.

[Fou19a] GraphQL Foundation. GraphQL dotazovací jazyk. https://graphql.org/learn/queries/, 2019.

[Fou19b] GraphQL Foundation. GraphQL knihovny. https://graphql.org/code/, 2019.

[Fou19c] GraphQL Foundation. GraphQL provádění dotazů. https://graphql.org/learn/execution/, 2019.

[Fou19d] GraphQL Foundation. GraphQL schema. https://graphql.org/learn/schema/, 2019.

[res19] Representational state transfer. https://en.wikipedia.org/wiki/Representational_state_transfer, 2019.