ELEKTROTEHNIŠKI VESTNIK 86(1-2): 43–46, 2019
IZVIRNI ZNANSTVENI ČLANEK

Spletne storitve z GraphQL

Domen Kajdič, Matjaž B. Jurič

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko, Večna pot 113, 1000 Ljubljana, Slovenija
E-pošta: domen@kajdic.net, matjaz.juric@fri.uni-lj.si

Povzetek. Spletne storitve so ključnega pomena za razvoj sodobnih aplikacij. Omogočajo komunikacijo čelnega in zalednega dela aplikacij. Čeprav se nove tehnologije pojavljajo dokaj pogosto, se razvijalci aplikacij pogosto oklepajo starejših tehnologij zaradi njihove dozorelosti in široke podpore. Najpogosteje uporabljeni tehnologiji za komunikacijo sta REST in SOAP, ki pa se nista spremenili že več let. V članku bomo predstavili tehnologijo GraphQL, ki je bila razvita kot odgovor na pomanjkljivosti trenutnih načinov komunikacije. Tehnologijo bomo primerjali s storitvami REST kot glavno alternativo in pokazali, na kakšen način odpravlja njene pomanjkljivosti. Na koncu se bomo dotaknili arhitekture mikrostoritev, ki predstavlja sodoben način razvoja aplikacij, ter predstavili uporabo GraphQL in arhitekture mikrostoritev v tandemu.

Ključne besede: tehnologija GraphQL, spletne storitve, arhitektura REST, razvoj sobodnih aplikacij

Web services with GraphQL

Web services are of key importance when it comes to developing modern applications. They allow communication of the front- and back-end. Though new technologies are often developed, developers often choose to stick with the old ones for being more mature and providing more support and documentation. The most widely-used communication technologies today are REST and SOAP which have not changed in the past couple of years. In this paper, the GraphQL technology is presented as an answer to shortcomings of the current technologies. It is compared to REST as its main alternative. At the end, microservices are introduced as a modern way of developing applications. The concept of microservices is used as an efficient way of showing how the GraphQL technology can be used in a modern environment.

Keywords: GraphQL technology, web services, REST architecture, modern applications

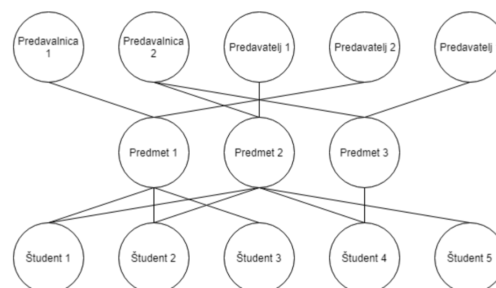
1 UVOD IN DEFINICIJE

GraphQL je poizvedovalni jezik, ki omogoča preprosto pridobivanje podatkov iz zalednega dela aplikacije. Razvilo ga je podjetje Facebook, in sicer predvsem za interno uporabo. Leta 2015 so ga prvič predstavili javnosti skupaj z objavo specifikacije [11] in referenčne implementacije v programskem jeziku JavaScript [5]. Zdaj je na voljo v večini popularnejših programskih jezikov [6], vendar je šele v zadnjem letu začel pridobivati na popularnosti. Kot primer uporabe lahko omenimo podjetje GitHub, ki je novo verzijo svojega apija izdalo s pomočjo GraphQL [8].

Ime GraphQL izvira iz dveh delov: *Graph* označuje graf; kratica QL pomeni *query language* oz. po slovensko poizvedovalni jezik. Pri GraphQLu si torej pridobivanje podatkov lahko predstavljamo kot premikanje po grafu. Klasičen graf definiramo kot množico vozlišč in povezav:

$$G = (V, E) \tag{1}$$

pri čemer je V množica vozšlišč (*vertices*) in E množica povezav (*edges*) [9] [10]. Za lažje razumevanje bomo predstavljeno ilustrirali na grafu na sliki 1. Na ta primer se bomo sklicevali skozi ves članek.



Slika 1: Primer grafa

Na grafu na sliki 1 opazimo štiri entitete: študente, predmete, predavalnice in predavatelje. To so vozlišča grafa, ki jih označimo s krogi. Vsaka izmed entitet ima neke lastnosti (npr. študent ima definirano ime), ki pa jih zaradi berljivosti nismo dodali na sliko. Študent obiskuje predmete, vsak predmet je predavan v neki predavalnici in ima nekega predavatelja. To so povezave grafa, ki jih označimo s puščicami. Povezave so lahko enosmerne ali dvosmerne. Dvosmerno povezavo med vozliščema A in B definiramo kot povezavo od A do B in hkrati povezavo od B do A. Kot smo prej omenili, pridobivanje podatkov poteka s premikanjem po grafu. Poizvedbo dobimo tako, da si izberemo neko točko in se začnemo premikati po njenih izhodnih povezavah.

Prejet 30. oktober, 2018
Odobren 17. december, 2018

Primeri poizvedb:

- Vrni ime in priimek vseh študentov, ki obiskujejo predmet 1.
- Vrni vpisne številke študentov, ki obiskujejo predmete pod mentorstvom predavatelja 1.
- Vrni vsa imena predmetov, ki jih predavata predavatelja 1 in 2.

2 Kako deluje GraphQL

V tem odseku bomo zgoraj predstavljeno logiko in poizvedbe pretvorili v notacijo GraphQL. Predstavili bomo tudi sintakso jezika in primer poizvedbe.

GraphQL lahko grobo razdelimo na tri dele: poizvedovanje, definiranje sheme in izvajalno okolje. Vsakega izmed treh bomo na kratko predstavili.

2.1 Poizvedovanje

Poizvedovanje je glavni razlog, zakaj se razvijalec odloči za uporabo GraphQL. Pred opisom poizvedovanja bomo pokazali, kako izgleda prava poizvedba na strežnik GraphQL.

```
// poizvedba
query {
  vrniPredmet(id: "1") {
    studenti {
      ime
      priimek
    }
  }
}

// rezultat poizvedbe
{
  "vrniPredmet": {
    "studenti": [
      {
        "ime": "Janez",
        "priimek": "Novak"
      }
    ]
  }
}
```

Če pogledamo zgornjo poizvedbo, opazimo naslednje lastnosti:

- Sintaksa poizvedovanja je izpeljana iz formata JSON [4].
- Izbirnost podatkov: uporabnik si lahko izbira, katera polja naj mu strežnik vrne.
- Število poizvedb: ena poizvedba je ponavadi dovolj za pridobitev vseh potrebnih podatkov.
- Rezultat poizvedbe: rezultat je enake oblike kot zahteva; uporabnik točno ve, v kakšni obliki bo strežnik vrnil podatke.

2.2 Definiranje sheme

Da bi bilo poizvedovanje mogoče, je treba definirati shemo. V shemi je treba definirati vse podatkovne tipe, ki jih bomo uporabljali pri poizvedovanju (v našem primeru so to prej omenjene štiri entitete), kot tudi vse operacije, ki jih uporabnik lahko uporablja v poizvedbah. Če pogledamo na prejšnji graf, operacije so točke na grafu, na katerih lahko vstopimo v graf (podmnožica točk). GraphQL podpira naslednje skalarne tipe: celo število (*Int*), decimalno število (*float*), niz znakov (*String*), logični izraz (*Boolean*) in enolični identifikator (ID; obravnavan kot niz znakov). Poleg tega so podprti še seznami (označimo z []), vmesniki (*Interface*), unije (*Union*) in naštevni tipi (*Enum*). Vsi uporabniško definirani tipi morajo biti sestavljeni iz osnovnih skalarnih tipov ali kakšnih drugih uporabniško definiranih tipov. Definirati je treba tudi operacije. GraphQL podpira tri vrste operacij: bralne operacije (*query*), pisalne operacije (*mutation*) in naročnine (*subscription*) [3]. Bralna operacija je namenjena standardnim poizvedbam za pridobivanje podatkov, pisalne operacije so namenjene spreminjanju podatkov in naročnine so namenjene konstantnemu spremljanju neke vrednosti (npr. cene delnic). Če pogledamo zgornjo poizvedbo, imamo podatkovni tip študent, ki vsebuje vsaj polji ime in priimek; podatkovni tip predmet, ki vsebuje enolični identifikator predmeta (id), in seznam študentov ter bralno operacijo vrniPredmet, ki zahtevani predmet vrne. Shema bi v notaciji GraphQL izgledala tako:

```
type Student {
    ime: String
    priimek: String
    // ostala polja
}

type Predmet {
    id: ID!
    studenti: [Student]
    // ostala polja
}

type Query {
    vrniPredmet(id: ID!): Predmet
    // ostale operacije
}

// ostali tipi in operacije
```

2.3 Izvajalno okolje

Zadnja komponenta GraphQL je izvajalno okolje. To je zaledje GraphQL oz. drugače povedano program, ki sprejema, procesira in vrača poizvedbe (ki so bile prej definirane v shemi). GraphQL je neodvisen od transportnega protokola (transportno agnostičen), kar pomeni, da ga lahko uporabljamo prek poljubnega transportnega protokola (v večini primerov je uporabljen protokol

HTTP) [1]. Celotna logika pridobivanja podatkov temelji na t. i. razreševalskih funkcijah (*resolver functions*). Vsaka operacija in vsako polje v podatkovni shemi vsebuje svojo razreševalsko funkcijo, ki jih izvajalno okolje ob prejetju poizvedbe pokliče v določenem vrstnem redu. Razreševalske funkcije mora napisati razvijalec. Da bi res razumeli, kaj se dogaja ob prejetju poizvedbe, bomo ilustrirali celotno dogajanje ob prejetju prej omenjene poizvedbe z operacijo *vrniPredmet* [13].

1) Uporabnik pošlje poizvedbo na strežnik GraphQL prek izbranega transportnega protokola.
2) Izvajalno okolje sprejme poizvedbo, jo najprej sintaktično preveri (pogleda format poizvedbe in preveri, ali v shemi obstaja operacija *vrniPredmet*). Če obstaja, pokliče njeno razreševalsko funkcijo, ki vrne določeni predmet. Če preverba ne uspe, vrne uporabniku napako.
3) Izvajalno okolje pogleda v shemo, kakšna polja vsebuje entiteta *predmet* in kakšne polja je zahteval uporabnik. V našem primeru je zahtevano le eno polje, in sicer polje *studenti* (ki je tipa seznam študentov).
4) Izvajalno okolje pokliče razreševalsko funkcijo polja *studenti*, ki vrne vse študente.
5) Izvajalno okolje pogleda v shemo, kakšna polja vsebuje entiteta *študent* in kakšna polja je zahteval uporabnik. V našem primeru sta zahtevani polji *ime* in *priimek*.
6) Izvajalno okolje za vsakega študenta pokliče razreševalski funkciji za pridobivanje imena in priimka.
7) Pridobljeni podatki se sestavijo v končni rezultat in vrnejo uporabniku (ponavadi v formatu JSON).

3 PRIMERJAVA GRAPHQL Z ARHITEKTURO REST

Večina sodobnih aplikacij za komunikacijo med čelnim in zalednim delom uporablja arhitekturo REST. Kljub razširjeni uporabi ima arhitektura REST številne pomanjkljivosti [14]. GraphQL je bil zasnovan kot odgovor na te pomanjkljivosti:

- Pri arhitekturi REST se entitete nahajajo na svojem naslovu. Če to preslikamo na naš primer, bi imeli štiri dostopne točke: /student, /profesor, /predmet in /predavalnica. Uporabnik, ki bi hotel delati zahtevnejše poizvedbe, bi moral v najslabšem primeru narediti štiri poizvedbe. Pri kakšnih kompleksnejših aplikacijah se ta številka lahko zelo poveča. Ne glede na število entitet se z uporabo GraphQL število zahtev zmanjša na eno samo.
- Pri arhitekturi REST se lahko zgodi, da mora neka aplikacija hraniti več vzporednih dostopnih točk, saj nekatere aplikacije še niso bile posodobljene na nove verzije. To pomeni, da ima lahko aplikacija za neko entiteto več dostopnih točk: starejša verzija /v1/student in verzija /v2/student z novimi funkcinalnostmi. Pri GraphQLu se lahko temu izognemo, saj je dodajanje in spreminjanje polj v podatkovnih tipih preprosto in neodvisno od njihove implementacije (polje dodamo v shemo in dodamo razreševalsko funkcijo). Dodana polja ne morejo uničiti nobene poizvedbe. Razreševalske funkcije lahko v celoti spremenimo in končni uporabnik tega ne bo opazil, saj bo delal enake poizvedbe, kot jih je delal prej.
- S tem ko uporabnik pri poizvedovanju sam izbira, katere podatke hoče pridobiti, se močno zmanjša količina prenesenih podatkov. Pri arhitekturi REST dostopne točke ponavadi vračajo celotne entitete z vsemi polji, ki pa jih aplikacija ponavadi ne potrebuje. To je bila ena izmed motivacij podjetja Facebook pri razvoju GraphQL, saj je prenos podatkov v nerazvitih predelih sveta velik problem.
- GraphQL omogoča t. i. introspekcijo [7]. Vsak strežnik omogoča poizvedovanje po operacijah in podatkovnih tipih, ki so na voljo. Lahko sestavimo poizvedbo, ki vrne mogoče operacije za naslednje poizvedbe. Pri arhitekturi REST mora za to poskrbeti razvijalec z uporabo različnih orodij (najpopularnejšo je orodje Swagger) ali kakšnih drugih pristopov.

Kljub temu tehnologija GraphQL ne more popolnoma zamenjati alternativ [15]. V nekaterih primerih lahko uporaba GraphQL prinese celo negativne posledice:

- Pri zelo preprostih podatkih oz. podatkih z malo relacijami lahko uvedba GraphQL poizvedovanje upočasni. Tipičen primer so aplikacije interneta stvari, ki zbirajo senzorske podatke z vnaprej določeno obliko. Poizvedba REST bo hitrejša, saj procesiranje in tudi definiranje sheme (ki bi bila trivialna) v tem primeru ne bosta potrebna.
- Pri uporabi GraphQL je lahko zaradi izbirnosti vsak izhod poizvedbe drugačen. Zato je oteženo predpomnjenje rezultatov poizvedb. V arhitekturi REST je izhod vedno iste oblike.
- GraphQL je relativno nova tehnologija, kar se pokaže predvsem v manjšem številu orodij in dokumentacije. Če gre kaj narobe, bomo z uporabo alternativ lažje našli rešitve kot pri uporabi GraphQL.

4 GRAPHQL IN MIKROSTORITVE

Pod pojmom mikrostoritve označujemo sodoben način razvoja aplikacij, pri čemer aplikacije razdelimo na manjše komponente – mikrostoritve [2]. Pri tem poskušamo doseči čim večjo neodvisnost med posameznimi mikrostoritvami. Pri razvoju se nagibamo k čim večji preprostosti posameznih mikrostoritev: vsaka mikrostoritev opravlja neko določeno funkcijo. Komunikacija poteka izključno prek predefiniranih vmesnikov (REST, čakalne vrste, ad - hoc pristopi...). Ta lastnost se imenuje šibka sklopljenost. Posledično lahko za vsako posamezno mikrostoritev skrbi druga razvijalska ekipa

z drugim razvojnim ciklom. Še več: vsaka mikrostoritev je lahko napisana v svojem programskem jeziku. Glavna prednost uporabe mikrostoritev je predvsem učinkovitejše skaliranje aplikacij. Aplikacijo lahko skaliramo glede na najbolj uporabljane funkcionalnosti oz. skaliramo samo tiste mikrostoritve, ki morajo obdelati največ prometa. Primer takšnega skaliranja je skaliranje mikrostoritve *katalog izdelkov* v neki spletni trgovini. Skaliranje, pri katerem skaliramo le posamezni del aplikacije z zagonom več instanc te aplikacije, imenujemo horizontalno skaliranje. Nasprotje horizontalnega skaliranja je vertikalno skaliranje, pri katerem pa skaliramo celo aplikacijo s povečanjem strežniških zmogljivosti (več RAM, hitrejši CPU...).

Aplikacija, ki sledi konceptom mikrostoritev, je sestavljena iz velikega števila mikrostoritev. Če se postavimo na stran razvijalca zalednega dela aplikacije, opazimo problem. Da bo aplikacija pridobila zahtevane podatke, bo morala narediti veliko zahtev na posamezne mikrostoritve. Pride ravno do problema, ki smo ga omenjali prej. Ta problem lahko rešimo ravno z uvedbo GraphQL. Na prvi pogled imamo dva načina uvedbe: uvedba na posameznih mikrostoritvah ali ustvarjanje nove mikrostoritve, ki bo agregirala več mikrostoritev. Drugi način je veliko boljši od prvega, saj s tem združimo celotno aplikacijsko shemo v eno mikrostoritev. Ustvarimo novo dostopno točko, ki omogoča pridobitev poljubnih podatkov iz celotne aplikacije z močjo poizvedovanja GraphQL. Razreševalske funkcije uporabimo kot način pridobivanja podatkov iz preostalih mikrostoritev. Če bi uvajali GraphQL na posameznih mikrostoritvah, bi s tem aplikacijo naredili kompleksnejšo. V arhitekturi mikrostoritev je komunikacija med posameznimi mikrostoritvami ključnega pomena. GraphQL bi to komunikacijo upočasnil, saj je pošiljanje poizvedb GraphQL kompleksnejše od pošiljanja poizvedb REST.

5 Sklep

V članku smo predstavili GraphQL, ki je ena od novejših tehnologij za komunikacijo čelnega dela z zalednim. Primerjali smo ga z arhitekturo REST ter predstavili njegove prednosti in slabosti. Na koncu smo se dotaknili tudi arhitekture mikrostoritev, ki je eden od sodobnejših načinov razvoja spletnih aplikacij. Pokazali smo, kako se lahko GraphQL učinkovito uporabi v arhitekturi mikrostoritev.

Glavni cilj pisanja tega članka je bila seznanitev bralca s sodobnimi koncepti spletnih storitev in razvoja aplikacij. Posledično smo v članku nanizali veliko novih pojmov in konceptov, ki jih neizobraženi bralec na tem področju ne bo poznal oz. bodo zanj novi. Zainteresirani bralec se lahko podrobneje seznani s tematiko z branjem diplomskega dela z naslovom "Spletne storitve z GraphQL", ki je javno dostopno na spletni strani Fakultete za računalništvo in informatiko [12]. V diplomskem delu se veliko bolj poglobimo v specifike GraphQL in arhitekturo mikrostoritev. Velik del je namenjen tudi programiranju mikrostoritev GraphQL v programskem jeziku Java.

Literatura

[1] E. Porcello, A. Banks, *Learning GraphQL: Declarative Data Fetching for Modern Web Apps*, O'Reilly Media, 2018.

[2] M. Richards, *Microservices AntiPatterns and Pitfalls*, O'Reilly Media, 2016.

[3] *GraphQL: Schemas and Types*, dosegljivo: https://graphql.github.io/learn/schema/, 2018. Dostopano: 22. 10. 2018.

[4] *GraphQL: Best Practices*, dosegljivo: https://graphql.org/learn/best-practices/, 2018. Dostopano: 22. 10. 2018.

[5] *GraphQL.js*, dosegljivo: https://github.com/graphql/graphql-js, 2018. Dostopano: 22. 10. 2018.

[6] *GraphQL libraries*, dosegljivo: https://graphql.github.io/code/, 2018. Dostopano: 22. 10. 2018.

[7] *GraphQL Introspection*, dosegljivo: https://graphql.org/learn/introspection/, 2018. Dostopano: 22. 10. 2018.

[8] *GitHub GraphQL API v4*, dosegljivo: https://developer.github.com/v4/guides/intro-to-graphql/, 2018. Dostopano: 22. 10. 2018.

[9] *Teorija grafov*, dosegljivo: https://sl.wikipedia.org/wiki/Teorija_grafov, 2018. Dostopano: 22. 10. 2018.

[10] *Teorija grafov*, dosegljivo: http://www.educa.fmf.uni-lj.si/izodel/sola/2006/ura/oblak/html/Uvod.html, 2006. Dostopano: 22. 10. 2018.

[11] *GraphQL specification*, dosegljivo: https://facebook.github.io/graphql/draft/, 2018. Dostopano: 22. 10. 2018.

[12] D. Kajdič, *Spletne storitve z GraphQL: diplomsko delo*, dosegljivo: http://eprints.fri.uni-lj.si/4231/ , 2018. Dostopano: 15. 10. 2018.

[13] *GraphQL explained*, dosegljivo: https://dev-blog.apollodata.com/graphql-explained-5844742f195e, 2016. Dostopano: 22. 10. 2018.

[14] *GraphQL vs. REST*, dosegljivo: https://dev-blog.apollodata.com/graphql-vs-rest-5d425123e34b, 2017. Dostopano: 22. 10. 2018.

[15] *Pain Points of GraphQL*, dosegljivo: https://labs.getninjas.com.br/pain-points-of-graphql-7e83ba5ddef7, 2017. Dostopano: 22. 10. 2018.

Domen Kajdič je magistrski študent Fakultete za računalništvo in informatiko. Diplomiral je leta 2018 pod mentorstvom prof. dr. Matjaža Branka Juriča na temo GraphQL.

Prof. dr. Matjaž B. Jurič, redni profesor, avtor 18 knjig, izdanih pri mednarodnih založbah, ter več kot 600 drugih publikacij. Prejel je več mednarodnih nagrad, med drugim nagrado za najboljšo SOA knjigo (New York), nagrado za najboljši SOA projekt v telekomunikacijah (Las Vegas), nagrado Java Duke's Choice Award Winner (San Francisco) za najboljšo inovacijo v Javi, naziv java champion, nagrado za najboljši znanstveni članek s področja storitev, nagrado za najboljšega raziskovalca po mnenju industrije in zlato plaketo za izjemne zasluge pri razvijanju znanstvenega ustvarjanja.