Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tim Štromajer

Visokoperformančne poizvedbe GraphQL z uporabo mikrostoritev

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: prof. dr. Branko Matjaž Jurič

Ljubljana, 2020

Copyright. Rezultati diplomske naloge so intelektualna lastnina avtorja in Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavo in koriščenje rezultatov diplomske naloge je potrebno pisno privoljenje avtorja, Fakultete za računalništvo in informatiko ter mentorja.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Podrobno proučite tehnologijo GraphQL, analizirajte delovanje ter jo primerjajte s tehnologijo REST. Izberite in primerjajte najbolj relevantne strežniške platforme, ki omogočajo postavitev GraphQLa. Poiščite in predstavite dobre prakse uporabe izbranih platform. Predstavite in postavite GraphQL API kot brezstrežniško aplikacijo. Predstavite učinkovite poizvedbe z uporabo avtorizacije in ovrednotite dobljene rezultate.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
API	Application Programming Interface	aplikacijski programski vmesnik
REST	Representational State Transfer	arhitekturni način prenašanja stanj
HTTP	Hypertext Transfer Protocol	protokol za prenos hiperbesedila
JSON	JavaScript Object Notation	objektna notacija v JavaScriptu
URL	Uniform Resource Locator	enolični označevalec vira
SDL	Schema Definition Language	jezik za definiranje sheme

Povzetek

Naslov: Visokoperformančne poizvedbe GraphQL z uporabo mikrostoritev

Avtor: Tim Štromajer

Večina mikrostoritev danes med sabo komunicira preko REST storitev. Te dobro služijo svojemu namenu, dokler mikrostoritve ne zahtevajo pogosto spremenljivih in kompleksnejših struktur podatkov. Nekatera večja podjetja so se že odločila za izbiro poizvedovalnega jezika GraphQL, kjer odjemalec pridobi le tiste podatke, ki jih predhodno v poizvedbi deklarativno zahteva. Zaradi tega se med strežnikom in odjemalcem ne pošilja nobenih odvečnih poizvedb ali podatkov.

V okviru diplomske naloge smo raziskali načine visokoperformančnega izvajanja poizvedb GraphQL. Med sabo smo primerjali že obstoječe platforme GraphQL Apollo Server, Graphene Python in GraphQL Java. Pokazali smo uporabo predpomnenja in trajnih poizvedb. Izdelali smo primer brezstrežniškega GraphQL APIja, pri čemer smo uporabili tehnologije GraphQL Apollo in Netlify. S pomočjo žetona JWT smo prikazali učinkovite poizvedbe z uporabo avtorizacije.

Ključne besede: GraphQL, spletne storitve, REST, mikrostoritve.

Abstract

Title: High-performance GraphQL queries using microservices

Author: Tim Štromajer

Most microservices today communicate with each other through REST services. These serve their purpose well, as long as microservices do not require often variable and more complex data structures. Some larger companies have already decided to use the GraphQL query language, where the client obtains only the data that is previously declaratively requested in the query. As a result, no redundant queries or data are sent between the server and the client.

As part of the diploma thesis, we investigated the methods of high-performance implementation of GraphQL queries. We compared the existing GraphQL platforms Apollo Server, Graphene Python and GraphQL Java. We demonstrated the use of caching and persistent queries. We created an example of a serverless GraphQL API, using GraphQL Apollo and Netlify technologies. We used JWT token to show efficient queries with the use of authorization.

Keywords: GraphQL, web services, REST, microservices.

Poglavje 1

Uvod

Zaradi vse večje uporabe mobilnih naprav in večanju kompleksnosti spletnih aplikacij, je težnja po učinkoviti komunikaciji med spletnimi storitvami čedalje večja. Trenutna najbolj uporabljena komunikacijska tehnologija REST predstavlja kar nekaj omejitev, zaradi česar so leta 2015 pri podjetju Facebook razvili poizvedovalni jezik imenovan GraphQL. GraphqQL za razliko od RESTa vedno pridobi le tiste podatke, ki jih odjemalec zahteva, kar reši problem pridobivanaja preveč ali premalo podatkov. V diplomski nalogi se bomo poleg primerjanja GraphQLa in RESTa, osredotočili predvsem na učinkovite načine poizvedovanja GraphQL in uporabe jezika GraphQL.

Med sabo bomo primerjali različne, že obstoječe platforme za izdelavo GraphQL APIjev. Pri tem bomo primerjali način zapisa sheme, definiranje skalarjev po meri, ravnanje z napakami, vpeljavo nalagalca podatkov in vmesne opreme.

Veliko podatkov, ki jih pridobimo iz API strežnikov, se večino časa v bazah ne spremeni. Takšne podatke želimo začasno shraniti, da ob naslednji, enaki poizvedbi ne dostopamo zopet do baze. Primer takšnega predpomnenja si bomo pogledali v četrtem poglavju, ki ga bomo nato nadgradili še s trajnimi poizvedbami. Z njimi rešujemo problem dolgih poizvedb, ki zasedejo veliko pasovne širine, in namesto njih pošljemo le zgoščeno vrednost poizvedbe.

Objava mikrostoritev preko oblačnih storitev lahko predstavlja zelo velik

strošek. Velikokrat se zgodi, da kupimo paket storitev, ki so večino časa v neuporabi. Brezstrežniška arhitektura omogoča skaliranje po potrebi, zanjo pa plačamo le toliko, kot jo uporabljamo. Ker je skaliranje mikrostoritev avtomatično, se razvijalci veliko manj ukvarjajo z zalednim delom. V okviru diplomske naloge bomo pokazali primer izdelave in postavitve GraphQL APIja na platformo AWS Lambda.

Podatki, s katerimi med sabo komunicirajo mikrostoritve, so lahko javni ali zasebni. Izredno pomembno je, da dostop do privatnih podatkov omejimo, ter pri tem ohranjamo dobro uporabniško izkušnjo. V diplomski nalogi bomo raziskali dva učinkovita načina uporabe avtorizacije v poizvedbah GraphQL.

Poglavje 2

Pregled GraphQL in primerjava z REST

V tem poglavju bomo predstavili tehnologijo GraphQL, njene osnovne gradnike in delovanje, ter jo primerjali s tehnologijo REST.

2.1 Uvod v GraphQL

Podatke v sodobnih aplikacijah velikokrat lahko prikažemo kot graf povezav in vozlišč, kjer vozlišča označujejo objekte, povezave pa relacije med njimi. GraphQL je deklarativni poizvedovalni jezik, ki se uporablja za poizvedovanje in manipulacijo podatkov aplikacijskih strežnikov [15].

Od ostalih spletnih storitev se loči predvsem v strukturi, saj uporabnik sam določi nabor podatkov, ki jih želi prejeti. Zahtev je zaradi tega manj, saj uporabnik lahko že z eno poizvedbo pridobi vse želene podatke, poleg tega pa so tudi hitrejše, saj se ne pošiljajo odvečni podatki, ki jih uporabnik ne nujno potrebuje. Namesto več končnih točk, kjer vsaka vrne fiksno določene podatke, GraphQL uporablja le eno končno točko, kjer je možno z različnimi poizvedbami dostopati do vseh razpoložljivih podatkov.

Za opis strukture in medsebojnih relacij podatkov GraphQL uporablja shemo. V njej poleg objektov definiramo tudi poizvedbe, mutacije in naročnine.

2.2 Shema

Kot smo povedali že v prejšnjem poglavju, shema natančno definira strukturo in medsebojne relacije med podatki. Za zapis sheme se uporablja jezik SDL (schema definition language). Definirana shema omogoča odkrivanje napak, še preden se zahteva sploh pošlje, poleg tega pa razvijalcem ni treba ročno pisati dokumentacije, saj se ta sama generira na podlagi sheme.

V shemi lahko definiramo dve vrsti tipov. Prvi tip so skalarji, ki so našteti v tabeli 2.1.

skalar	slovenski izraz
String	niz znakov
Int	celo število
Float	decimalno število
Boolean	logični izraz
ID	enolični identifikator

Tabela 2.1: Tipi skalarjev in njihovi slovenski izrazi

GraphQL podpira tudi definiranje tipa *enum* (izsek 2.1). *Enum* je posebna vrsta skalarja, ki ima omejeno zalogo vrednosti.

```
enum Zvrst {
    ROMAN
    PRAVLJICA
    ESEJ
}
```

Izsek 2.1: Inicializacija tipa *enum*

Drugi tip podatkov pa so objekti, ki so lahko sestavljeni iz več polj skalarjev ali objektov.

```
type Avtor {
    id: ID!
    ime: String!
    letnicaRojstva: Int!
    dela: [Knjiga!]!
```

```
}
type Knjiga {
    id: ID!
    naslov: String!
    zvrst: Zvrst!
    avtorji: [Avtor!]
}
```

Izsek 2.2: Inicializacija objektnega tipa *Avtor* in *Knjiga*

Na izseku, ki 2.2 prikazuje inicializacijo objektnega tipa, smo tip polja *naslov*, objekta *Knjiga*, označili s klicajem (String!). To pomeni, da polje ne sme biti prazno oziroma da bodo poizvedbe tega polja vedno vračale neko vrednost. Tip polja *avtorji* objekta *Knjiga*, smo označili z oglatimi oklepaji in klicajem ([Avtor!]). To pomeni, da polje vrača listo avtorjev, v kateri ne sme biti praznih vrednosti. Knjiga je lahko brez avtorjev (npr.: avtor je neznan), lahko pa ima več avtorjev, od katerih nihče ne sme imeti vrednosti *null*.

Naslednja funkcionalnost, ki jo omogoča GraphQL, so vmesniki (angl. *interface*). Z njimi lahko določimo, katera polja morajo posamezni objekti vsebovati. Uporabni so predvsem, kadar vračamo več objektov, ki imajo lahko različne tipe.

```
interface Vozlisce {
    id: ID!
}
type Knjiga implements Vozlisce{
    id: ID!
    naslov: String!
    zvrst: Zvrst!
    avtorji: [Avtor!]
}
```

Izsek 2.3: Primer uporabe vmesnika

Izsek 2.3 prikazuje primer uporabe vmesnika. Ker objekt tipa *Knjiga* implementira objekt *Vozlisce*, mora vsebovati vsa polja, ki jih vsebuje tudi *Vozlisce*. V tem primeru mora vsebovati polje *id*.

Tip objekta v shemi GraphQL lahko definiramo tudi v obliki unije.

```
type Knjiga {
    id: ID!
    naslov: String!
    zvrst: Zvrst!
    avtorji: [Avtor!]
}
type Revija {
    id: ID!
    naslov: String!
    stevilka: Int!
}
union Literatura = Knjiga | Revija
```

Izsek 2.4: Primer uporabe unije tipov

Izsek 2.4 prikazuje primer uporabe unije tipov. Objekt tipa *Literatura* je lahko tipa *Knjiga* ali *Revija*.

Do sedaj smo v shemi definirali tipe objektov in njihove relacije, sedaj pa bomo dodali še vstopne točke, ki jih delimo na poizvedbe (angl. *query*), mutacije (angl. *mutation*) in naročnine (angl. *subscription*). Vse tri bi lahko definirali v istem tipu, vendar jih zaradi lažje preglednosti ločimo. Poizvedovalne operacije so tiste, ki podatke iz baze le berejo, mutacije podatke v bazi spreminjajo, naročnine pa obvestijo naročnike o spremenjenih podatkih v bazi. Vrstice, ki se začnejo s številskim znakom (#), so komentarji.

```
type Query {
    # vrne seznam vseh knjig v bazi
    vrniKnjige: [Knjiga!]
    # vrne knjgo glede na podan id
    vrniKnjigo(id: ID!): Knjiga
}
type Mutation {
    # v bazo doda knjigo glede na podane argumente
    dodajKnjigo(naslov: String!, zvrst: Zvrst!, avtorId: [ID
    !]): Knjiga
    # izbriše knjigo glede na podan id
    izbrisiKnjigo(id: ID!): Boolean
}
```

```
type Subscription {
    # obvesti naročnika, ko se v bazo doda nova knjiga
    dodanaKnjiga: Knjiga
}
```

Izsek 2.5: Primer poizvedbe, mutacije in naročnine

Izsek 2.5 prikazuje primer poizvedbe, mutacije in naročnine. Tako kot polja objektov so tudi operacije sestavljene iz imena (npr.: *vrniKnjigo*), argumentov (npr.: *id: ID!*) in tipa, ki ga vračajo (npr.: *Knjiga*). Vsako polje ima lahko poljubno število argumentov, ki jim moramo obvezno podati ime in tip. Poleg skalarjev pa so argumenti lahko tudi kompleksnejši objekti, ki jih je priporočljivo uporabljati pri mutacijah z veliko argumentov. Tako bi lahko mutacijo *dodajKnjigo* preoblikovali (izsek 2.6).

```
input KnjigaVhod {
    naslov: String!
    zvrst: Zvrst!
    avtorId: [ID!]
}
type Mutation {
    dodajKnjigo(knjiga: KnjigaVhod): Knjiga
    izbrisiKnjigo(id: ID!): Boolean
}
```

Izsek 2.6: Primer uporabe kompleksnejšega argumenta

2.3 Poizvedovanje

Poizvedbe, ki jih v GraphQLu lahko izvedemo, so določene na podlagi sheme. Poizvedbe vedno vračajo takšno strukturo podatkov, kot jo zahtevamo v poizvedbi.

```
query {
    vrniKnjigo(id: "k1") {
        naslov
        zvrst
    }
```

```
}
```

Izsek 2.7: Primer poizvedbe *vrniKnjige*

Z zgornjo poizvedbo (izsek 2.7) pokličemo operacijo *vrniKnjige*, ki je tipa poizvedba, ter zahtevamo, da nam vrne polji *naslov* in *zvrst*. Operaciji podamo tudi argument tipa *id* ("k1"), ki je pri tej operaciji obvezen.

```
{
    "data" {
        "vrniKnjige": {
            "naslov": "Naslov knjige"
            "zvrst": "ROMAN"
        }
    }
}
```

Izsek 2.8: Primer vrnjenih podatkov poizvedbe *vrniKnjige*

Izsek 2.8 prikazuje primer vrnjenih podatkov ob poslani zahtevi z izseka 2.7.

GraphQL omogoča tudi kompleksnejše poizvedbe.

```
query {
    vrniKnjigo(id: "k1") {
        naslov
        zvrst
        avtorji {
            ime
        }
    }
}
```

Izsek 2.9: Primer kompleksnejše poizvedbe

```
{
    "data" {
        "vrniKnjige": {
            "naslov": "Naslov knjige"
            "zvrst": "ROMAN"
            "avtorji": [
```

```
                {
                    "ime": "ime avtorja"
                }
            ]
        }
    }
}
```

Izsek 2.10: Primer vrnjenih podatkov kompleksnejše poizvedbe

Izsek 2.10 prikazuje primer vrnjenih podatkov kompleksnejše poizvedbe z izseka 2.9. Polja bi lahko dodajali do poljubne globine, če to shema omogoča, vendar pa je dobra praksa maksimalno globino omejiti.

Poizvedbam, ki jih pošljemo na strežnik, lahko dodamo tudi direktive. Direktive nam omogočajo dodatno filtriranje po podatkih. Direktivi *include* in *skip* sta že vgrajeni v GraphQL jezik, razvijalci pa lahko dodajo tudi svoje direktive.

```
query vrniKnjige($zAvtorji: Boolean!) {
    vrniKnjige {
        naslov
        avtorji @include(if: $zAvtorji) {
            ime
        }
    }
}
```

Izsek 2.11: Primer uporabe direktiv

Poizvedba z izseka 2.11 bo glede na podan argument *zAvtorji* vrnila tudi podatke avtorjev.

GraphQL omogoča tudi pošiljanje več poizvedb v eni sami zahtevi. Vsaki poizvedbi moramo dodati neki vzdevek (angl. *alias*), da GraphQL lahko loči med njimi.

```
query {
    #poizvedba ima vzdevek prva
    prva: vrniKnjige {
        naslov
```

```
    }
    #poizvedba ima vzdevek druga
    druga: vrniKnjige {
        naslov
        zvrst
    }
}
```

Izsek 2.12: Primer uporabe vzdevkov

```
{
  "data": {
    "prva": [
      {
        "naslov": "naslov knjige"
      }
    ],
    "druga": [
      {
        "naslov": "naslov knjige"
        "zvrst": "ROMAN"
      }
    ]
  }
}
```

Izsek 2.13: Primer vrnjenih podatkov z uporabo vzdevkov

Z zgornjo zahtevo (izsek 2.12) smo dvakrat zahtevali naslove vseh knjig, poleg tega pa smo v drugi poizvedbi zahtevali tudi zvrst. Prvi poizvedbi smo dodelili vzdevek *prva*, drugi pa vzdevek *druga*. Rezultati poizvedb prikazuje izsek 2.13.

Če se v poizvedbah želimo izogniti ponavljanju večjega števila polj, lahko uporabimo fragmente. Fragmenti predstavljajo podmnožico polj določenega tipa.

```
fragment nas on Knjiga {
  naslov
  zvrst
```

```
}
```

Izsek 2.14: Primer fragmenta

```
# poizvedba z uporabo fragmenta
query {
  vrniKnjige{
    ... nas
  }
}

# poizvedba brez uporabe fragmenta
query {
  vrniKnjige{
    naslov
    zvrst
  }
}
```

Izsek 2.15: Primer uporabe fragmenta in njegova enačica

Na izseku 2.14 smo ustvarili fragment in ga poimenovali *nas*, na izseku 2.15 pa smo ga uporabili. Ob uporabi fragmenta poizvedba vrne tiste podatke, ki jih vsebuje fragment.

2.4 Mutacije

Mutacije so zelo podobne poizvedbam razen, da se začnejo z besedo *mutation*. Z mutacijami podatke spreminjamo in jih hkrati lahko tudi beremo.

```
mutation {
  dodajKnjigo(naslov: "naslov knjige", zvrst: ROMAN, avtorId:
    "a1") {
    naslov
  }
}
```

Izsek 2.16: Primer uporabe mutacije

Z mutacijo na izseku 2.16 smo v bazo dodali novo knjigo. Zraven smo dodali tri argumente: *naslov*, *zvrst* in *avtorId*. Na koncu smo zahtevali, da nam mutacija izpiše naslov dodane knjige. Vrnjeni podatki poizvedbe so vidni na izseku 2.17.

```
{
  "data": {
    "dodajKnjigo": {
      "naslov": "naslov knjige"
    }
  }
}
```

Izsek 2.17: Primer vrnjenih podatkov mutacije

2.5 Naročnine

Določene aplikacije zahtevajo takojšnjo zaznavo spremenjenih podatkov, zato potrebujejo povezavo v realnem času. GraphQL omogoča, da se odjemalec prijavi na določen dogodek in s tem obdrži povezavo s strežnikom. Ko se dogodek zgodi, strežnik odjemalcu posreduje spremembo.

```
subscription {
  dodanaKnjiga {
    id
    naslov
  }
}
```

Izsek 2.18: Primer naročnine

V primeru na izseku 2.18 se je odjemalec naročil na dogodek *dodanaKnjiga* in odprla se je povezava med strežnikom in odjemalcem. Ko naslednjič pride do mutacije in se podatki, na katere je odjemalec naročen, spremenijo, se to sporoči odjemalcu.

2.6 Delovanje GraphQL – algoritem

Ko pride poizvedba na strežnik se izvedejo trije koraki [21]:

1. Razčlenjevanje poizvedbe – Strežnik kodo razčleni in jo spremeni v abstraktno sintaktično drevo. Če je v poizvedbi sintaktična napaka, bo strežnik prenehal razčlenjevanje in vrnil napako odjemalcu.

2. Validacija – Po ugotovljeni sintaktični pravilnosti strežnik ugotovi, ali je poizvedba smiselna. Preveri, ali obstaja podana poizvedba, če so argumenti pravilni, ali imajo tipi podana polja.

3. Izvedba – Vsaka poizvedba ima obliko drevesa, izvedba pa se začne v korenu.

```
query {
  vrniKnjigo(id: "k1") {
    naslov
    avtorji {
        ime
    }
  }
}
```

Izsek 2.19: Poizvedba *vrniKnjigo*

Vsako polje v poizvedbi ima v shemi definiran svoj tip. Ko se določijo tipi, se za vsako polje po vrsti pokliče razreševalna funkcija (angl. *resolver*). V primeru na izseku 2.19 bi se najprej poklicala razreševalna funkcija za *Query.vrniKnjigo*, nato bi te rezultate posredovali razreševalni funkciji *Knjiga.naslov* in *Knjiga.avtorji*, ki pa vrača seznam. Pri vrnjenem seznamu bi nato za vsak element klicali razreševalno funkcijo *Avtorji.ime*. Ko polje proizvede skalar, se izvajanje zaključi. Na koncu bi se vse združilo obliko ključ-vrednost in vrnilo. Celoten algoritem je prikazan tudi na izseku 2.20.

```
Query.vrniKnjigo(root, { id: "k1" }, context) -> knjiga
Knjiga.naslov(knjiga, null, context) -> naslov
Knjiga.avtorji(knjiga, null, context) -> avtorji
```

```
for each avtor in avtorji
  Avtor.ime(avtor, null, context) -> ime
```

Izsek 2.20: Algoritem poizvedbe

2.7 Introspekcija

Poznavanje sheme je za uporabnika zelo pomembno, saj lahko hitro in z malo poizvedb pridobi vse želene podatke. GraphQL omogoča poizvedbo po shemi in temu pravimo introspekcija. To lahko dosežemo s poizvedbo tipa *schema* in *type*.

```
query {
  __schema {
    types {
      name
    }
  }
}
```

Izsek 2.21: Introspekcija tipa *schema*

```
{
  "data": {
    "__schema": {
      "types": [
        # še ostali tipi
        ...
        {
          "name": "Knjiga"
        },
        {
          "name": "Zvrst"
        },
        {
          "name": "Avtor"
        },
        ...
```

```
        # še ostali tipi
      ]
    }
  }
}
```

Izsek 2.22: Rezultat introspekcije tipa *schema*

Na Izseku 2.21 smo zahtevali imena vseh tipov sheme, rezultat poizvedbe pa je prikazan na izseku 2.22.

Introspekcijo pa lahko izvedemo tudi na posameznem tipu.

```
{
  __type(name: "Knjiga") {
    name
    fields {
      name
      type {
        name
        kind
      }
    }
  }
}
```

Izsek 2.23: Introspekcija tipa *type*

```
{
  "data": {
    "__type": {
      "name": "Knjiga",
      "fields": [
        {
          "name": "id",
          "type": {
            "name": null,
            "kind": "NON_NULL"
          }
        },
        ...
```

```
        # še ostala polja
      ]
    }
  }
}
```

Izsek 2.24: Rezultat introspekcije tipa *type*

Na izseku 2.23 smo zahtevali ime in polja tipa *Knjiga*. Rezultat je prikazan na izseku 2.24 in prikazuje, da tip *Knjiga* vsebuje polje z imenom *id*.

2.8 Primerjava GraphQL z REST

REST je danes zelo pogost način pridobivanja podatkov s strežnika. Tako REST kot tudi GraphQL se lahko uporabljata preko kateregakoli zalednega in čelnega dela, si ne zapomnita stanja, ter si ponavadi izmenjujeta JSON podatke. Vendar pa zaradi različnih potreb danes REST postaja velikokrat neučinkovit [8].

Vedno več je uporabnikov mobilnih naprav [31], ki niso tako zmogljive kot računalniki in imajo slabšo spletno povezavo. Takšne naprave želijo pridobiti le tiste podatke, ki jih zares potrebujejo.

Drugi razlog je zmeraj hitrejši razvoj in spreminjanje aplikacij, pri čemer REST ne omogoča tako lahkega prilagajanja spremembam.

Razlika med REST in GraphQL je tudi v verzioniranju [9]. Pri REST APIjih spremembe zahtevajo nove verzije, saj se spremenijo ali dodajo nove končne točke dostopa APIja. Velikokrat je treba sprejeti kompromis med tem, ali bomo večkrat izdali novosti, ali bomo ohranjali razumljivost in vzdrževanost. Pri GraphQL pa imamo le eno končno točko, preko katere dobimo le tiste podatke, ki smo jih zahtevali. Vse novosti lahko dodamo v obliki novega tipa ali novih polj že obstoječih tipov.

Problem tehnologije REST je tudi v tem, da nima sheme, ki bi odjemalcu povedala, katere podatke API vrača. Pri odkrivanju podatkov mora

odjemalec iz baze pridobiti dejanske podatke, da lahko določi njihov tip. Problem se sicer lahko reši z uporabo dodatnih knjižnic, kot je Swagger oziroma OpenAPI 3. Shema v GraphQL pa je obvezna, ter uporabniku omogoča odkrivanje podatkov [12], ki jih strežnik ponuja. Odkrivanje podatkov preko GraphQL je neodvisno od dejanskih podatkov v bazi. Shema se lahko uporablja tudi kot dokumentacija APIja.

REST nudi svoje podatke preko več končnih točk [1], kjer vsaka vrne fiksno določene podatke. Pri tem lahko pride do pridobivanja odvečnih podatkov, čemur s tujko pravimo *overfetching* [13]. Za tak primer bi lahko vzeli nalaganje profila, na katerem je lista prijateljev. Za prikaz profila bi potrebovali le imena ter slike prijateljev, klasični REST API pa nam zraven lahko vrne še veliko dodatnih podatkov, ki jih ne potrebujemo (npr. naslov, starost). Ravno nasproten problem pa je pridobivanje premalo podatkov, ki mu s tujko pravimo *underfetching*. Za primer si lahko zamislimo situacijo, ko bi želeli pridobiti po enega prijatelja vseh naših prijateljev. S tehnologijo REST bi morali za vsakega našega prijatelja klicati svojo poizvedbo, kjer bi zahtevali enega njegovega prijatelja. Poslati bi morali veliko število zahtev, kar bi lahko pripeljalo do neodzivnosti.

Zaradi natančnega opisa zahtevanih podatkov GraphQL razvijalcem omogoča razpoznati, kateri podatki so najbolj in kateri najmanj zahtevani. To jim omogoči boljše poznavanje odjemalcev ter lažje prilagajanje potrebam. Pri REST APIjih pa se lahko razvidi, le do katerih končnih točk je bilo največ zahtev.

V tabeli 2.2 so prikazane ključne razlike med GraphQL in REST.

	GraphQL	**REST**
struktura vrnjenih podatkov	deklarativno opisana z vsako poizvedbo	vnaprej določena za vsako končno točko
prilagoditev na spremembo strukture podatkov	prilagodimo shemo in razreševalne funkcije	potrebno izdati novo verzijo APIja, saj je prišlo do nove ali spremenjene končne točke
razpoložljivost informacij o podatkih, ki jih API vrača	informacije o tipih in strukturi podatkov so zapisane v shemi, po kateri lahko odjemalec poizveduje	uporaba dodatnih knjižnic kot je Swagger oziroma OpenAPI 3
pridobivanje odvečnih (overfetching) ali premalo (underfetching) podatkov	v poizvedbi točno določimo katere podatke želimo pridobiti, zato do tega ne more priti	končne točke vračajo fiksno strukturo podatkov, zaradi česar lahko pride do tega problema
monitoriranje zaželejnosti podatkov	za vsak podatek točno vemo kolikokrat je bil zahtevan	merimo lahko le kolikokrat so bili zahtevani podatki posamezne končne točke

Tabela 2.2: Ključne razlike med GraphQL in REST

Poglavje 3

Primerjava različnih platform GraphQL

Danes obstaja zelo veliko platform za postavitev strežnika GraphQL. V tem poglavju bomo naredili primerjavo med platformami GraphQL, ki so napisane za različne programske jezike. Za vsak jezik bomo izbrali eno izmed trenutno najbolj uporabljenih platform glede na seznam na uradni GraphQL strani [11]. Za jezik Python bomo uporabili Graphene Python, za JavaScript Apollo Server in za Javo GraphQL Java.

Primerjali bomo način zapisa sheme, unije tipov in vmesnika. Pogledali bomo, kako različne platforme podpirajo definiranje skalarjev po meri, ravnanje z napakami, uporabo nalagalca podatkov (angl. *data loader*), ter uporabo vmesne opreme (angl. *middleware*) za potrebe logiranja aktivnosti strežnika ali avtentikacije uporabnikov. Funkcija nalagalca podatkov je zbiranje zahtev do baze, pri čemer si duplikate zahtev shrani le enkrat. Vse podatke zahtev nato pridobi z le enim dostopom do baze, kar lahko pospeši delovanje aplikacij.

3.1 Graphene Python

Graphene Python [14] je python knjižnica za izgradnjo GraphQL APIjev. Graphene je mogoče integrirati v različna spletna ogrodja, kot so Google App Engine, SQLAlchemy in Django.

3.1.1 Definiranje sheme

```
import graphene

class Query(graphene.ObjectType):
    # definiranje polja pozdrav z argumentom ime
    pozdrav = graphene.String(
        ime=graphene.String(default_value="gost"))

schema = graphene.Schema(query=Query)
```

Izsek 3.1: Primer sheme

Izsek 3.1 prikazuje definiranje sheme, ki jo izvedemo programsko. Graphene podpira skalarje tipa *String*, *Int*, *Float*, *Boolean*, *ID*, *JSON String*, *Date* in *Enum*. Skalarji sprejmejo tudi opcijske argumente: *name* tipa *String*, ki nadomesti prvotno ime polja, *description* tipa *String*, kamor zapišemo opis polja, *required* tipa *Boolean*, ki pove, če je polje zahtevano, *deprecation_reason* tipa *String*, ki hrani razlog, zakaj je polje zastarelo, in *default_value*, ki pove, kakšna je privzeta vrednost. Mogoča je tudi izdelava lastnih skalarjev.

3.1.2 Skalarji po meri

```
import graphene
from datetime import datetime
from graphql.language import ast

class Datum(graphene.types.Scalar):
    '''opis tipa'''

    @staticmethod
```

```
    def serialize(dt):
        return str(dt.day) + "." +
            str(dt.month) + "." + str(dt.year)

    @staticmethod
    def parse_literal(node):
        if isinstance(node, ast.StringValue):
            return datetime.strptime(node.value, "%d.%m.%Y")

    @staticmethod
    def parse_value(value):
        return datetime.strptime(value, "%d.%m.%Y")
```

Izsek 3.2: Kreiranje skalarja po meri

Izsek 3.2 prikazuje način kreiranja lastnega skalarja *Datum*. Razredu *Datum* smo podali opis, ter definirali metode *serialize*, *parse_literal* in *parse_value* s katerimi definiramo pravila vhodnih in izhodnih podatkov. V našem primeru sprejmemo in vračamo datum tipa *String*, ki je v obliki dneva, meseca in leta (npr.: 22.8.2020).

3.1.3 Uporaba vmesnika

```
class Node(graphene.Interface):
    id = graphene.ID(required=True)

class Knjiga(graphene.ObjectType):
    class Meta:
        interfaces = (Node, )

    naslov = graphene.String()
```

Izsek 3.3: Primer uporabe vmesnika

Izsek 3.3 prikazuje primer uporabe vmesnika, ko tip *Knjiga* implementira tip *Node*.

3.1.4 Unija tipov

```
class Knjiga(graphene.ObjectType):
```

```
    naslov = graphene.String()
    zvrst = graphene.String()

class Revija(graphene.ObjectType):
    naslov = graphene.String()
    stevilka = graphene.Int()

class Literatura(graphene.Union):
    class Meta:
        types = (Knjiga, Revija)
```

Izsek 3.4: Primer uporabe unije tipov

Izsek 3.4 prikazuje primer uporabe unije tipov.

3.1.5 Ravnanje z napakami

```
from graphql import GraphQLError

class DodajKnjigo(graphene.Mutation):
  class Arguments:
    naslov = graphene.String()
    zvrst = graphene.String()

  Output = Knjiga

  def mutate(root, info, naslov, zvrst):
    if not prijavljen(info):
        raise GraphQLError(
            'Za dodajanje morate biti prijavljeni')

    return Knjiga(naslov=naslov, zvrst=zvrst)
```

Izsek 3.5: Ravnanje z napakami

Izsek 3.5 prikazuje primer ravnanja z napakami. Napaka se bo sprožila v primeru, ko odjemalec, ki želi dodati knjigo, v sistem ni prijavljen.

3.1.6 Nalagalec podatkov

```
from promise import Promise
```

```
from promise.dataloader import DataLoader

class UporabnikLoader(DataLoader):
    def batch_load_fn(self, keys):
        return Promise.resolve(
            [pridobiUporabnika(id=key) for key in keys])

class Uporabnik(graphene.ObjectType):
    ime = graphene.String()
    naj_prijatelj = graphene.Field(lambda: Uporabnik)
    prijatelji = graphene.List(lambda: Uporabnik)

    def resolve_naj_prijatelj(root, info):
        return UporabnikLoader.load(root.naj_prijatelj_id)

    def resolve_prijatelji(root, info):
        return UporabnikLoader.load_many(root.prijatelji_ids)
```

Izsek 3.6: Nalagalec podatkov

Izsek 3.6 prikazuje primer nalaganja podatkov. Funkcija *batch_load_fn* sprejme seznam ključev (angl. *keys*) in vrne obljubo (angl. *Promise*), ki bo razrešila seznam vrednosti. V našem primeru bo vrnila seznam uporabnikov glede na podane ključe.

3.1.7 Vmesna oprema

Pred izvedbo poizvedbe lahko pokličemo vmesno opremo, s pomočjo katere lahko prestrežemo določene podatke, ter jih uporabimo za logiranje ali avtentikacijo.

```
def vmesnaOprema(next, root, info, **args):
    if !avtenticiraj(info.context):
        raise GraphQLError(
            'Avtentikacijski podatki niso pravilni!')
    return next(root, info, **args)

result = schema.execute(poizvedba, middleware=[vmesnaOprema])
```

Izsek 3.7: Primer vmesne opreme

Izsek 3.7 prikazuje dodajanje vmesne opreme poizvedbi. Poizvedba se bo izvedla le v primeru uspešne izvedbe vmesne opreme. V našem primeru smo v vmesno opremo dodali preprosto avtentikacijo, služi pa lahko tudi za logiranje.

3.2 Apollo Server

Apollo Server [4] je odprtokodni strežnik GraphQL v jeziku JavaScript.

3.2.1 Definiranje sheme

```
const typeDefs = gql`
    type Knjiga {
        id: ID!
        naslov: String!
    }

    type Query {
        vrniKnjigo(id: ID!): Knjiga
    }

    type Mutation {
        dodajKnjigo(naslov: String!): Knjiga
    }
`;
```

Izsek 3.8: Definiranje sheme

Izsek 3.8 prikazuje definiranje sheme s pomočjo jezika SDL. Definirali smo objekt *Knjiga*, ki vsebuje polji *id* in *naslov*, ter poizvedbo *vrniKnjigo*, ki sprejme parameter *id* in vrne objekt *Knjiga* in mutacijo za dodajanje nove knjige, ki sprejme parameter *naslov* in vrne objekt *Knjiga*.

Apollo Server podpira skalarje tipa *Int*, *Float*, *String*, *Boolean*, *ID*, *Enum*, lahko pa dodamo tudi svoje.

3.2.2 Skalarji po meri

```
const typeDefs = gql‘
    scalar Datum

    type Knjiga {
        id: ID!
        naslov: String!
        datum: Datum
    }
‘;
```

Izsek 3.9: Dodajanje novega skalarja v shemo

```
const { GraphQLScalarType } = require('graphql');
const { Kind } = require('graphql/language');

module.exports = {
    Query: ...,
    Mutation: ...,
    Datum: new GraphQLScalarType({
        name: 'Datum',
        description: 'opis skalarja',
        parseValue(value) {
            return new Date(value);
        },
        serialize(value) {
            return value;
        },
        parseLiteral(ast) {
            if (ast.kind === Kind.STRING) {
                return new Date(ast.value);
            }
            return null;
        },
    }),
};
```

Izsek 3.10: Kreiranje skalarja po meri poleg razreševalnih funkcij

Pri kreaciji skalarja po meri s pomočjo strežnika Apollo moramo najprej v shemi definirati nov skalar (izsek 3.9). Pravila vhodnih in izhodnih podatkov

skalarja definiramo poleg razreševalnih funkcij (izsek 3.10) s pomočjo metod *parseValue*, *serialize* in *parseLiteral*.

3.2.3 Uporaba vmesnika

```
const typeDefs = gql`
    type Node {
        id: ID!
    }

    type Knjiga implements Node {
        id: ID!
        naslov: String!
    }
`;
```

Izsek 3.11: Primer uporabe vmesnika

Izsek 3.11 prikazuje primer uporabe vmesnika, ko tip *Knjiga* implementira tip *Node*.

3.2.4 Unija tipov

```
const typeDefs = gql`
    type Knjiga {
        id: ID!
        naslov: String!
        zvrst: String!
    }

    type Revija {
        id: ID!
        naslov: String!
        stevilka: String!
    }

    union literatura = Knjiga | Revija
`;
```

Izsek 3.12: Primer unije tipov

Izsek 3.12 prikazuje primer uporabe unije tipov.

3.2.5 Ravnanje z napakami

Apollo Server podpira predefinirane napake *AuthenticationError*, *ForbiddenError*, *UserInputError* in *ApolloError*. Vsaka napaka znotraj *Errors* seznama vsebuje objekt *extensions*, ki vsebuje dodatne informacije napake.

```
const { AuthenticationError } = require('apollo-server');

const resolvers = {
  Query: {
    vrniNapako: (parent, args, context) => {
      throw new AuthenticationError('Napaka pri avtentikaciji
   ');
    },
  },
};
```

Izsek 3.13: Primer napake

Izsek 3.13 prikazuje poizvedbo *vrniNapako*, ki bo vedno vrnila avtentikacijsko napako.

3.2.6 Nalagalec podatkov

```
const DataLoader = require('dataloader');

var knjigaLoader = new DataLoader(
    keys => {
        const promises = keys.map(async (key) => {
            return await pridobiKnjigo(key);
        })
        return Promise.all(promises)
      }
);

const server = new ApolloServer({
    // naložimo preko context.loader.KnjigaLoader.load(id);
```

```
    context: ({req ,res}) => {
        return {
            loader: {
                KnjigaLoader: knjigaLoader
            }
        }
    },
    ...
});
```

Izsek 3.14: Nalagalec podatkov

Izsek 3.14 prikazuje inicializacijo in dodajanje v kontekst nalagalca podatkov. Ob vsaki zahtevi knjige naložimo id knjige v nalagalca in ta bo poskrbel za samo en dostop do baze podatkov.

3.2.7 Vmesna oprema

```
const server = new ApolloServer({
    context: ({req ,res}) => {
        const token = req.headers.authorization || '';
        const uporabnik = pridobiUporabnika(token);
        if (!uporabnik) throw new AuthenticationError('
            avtentikacijski podatki niso pravilni');
        return {
            Uporabnik: uporabnik
        }
    },
    ...
});
```

Izsek 3.15: Primer vmesne opreme

Izsek 3.15 prikazuje primer vmesne opreme, v kateri preverimo avtentikacijo odjemalca.

3.2.8 Monitoriranje in logiranje

Apollo Server omogoča integracijo z Apollovim Studiem, s katerim lahko monitoriramo in logiramo poizvedbe GraphQL. Za logiranje lahko uporabimo

tudi vmesno opremo ali napišemo svoj vtičnik (angl. *plugin*) (izsek 3.16).

```
const mojVticnik = {
    // Sproži, ko je prejeta zahteva
    requestDidStart(requestContext) {
      console.log('Prejeta zahteva. Query:\n' +
   requestContext.request.query);
      return {
        // Sproži ob razčlenjevanju zahteve
        parsingDidStart(requestContext) {
          console.log('Razčlenjevanje');
        },
        // Sproži ob validaciji zahteve
        validationDidStart(requestContext) {
          console.log('Validacija');
        }
      }
    },
};

const server = new ApolloServer({
    plugins: [
        mojVticnik
    ],
    ...
});
```

Izsek 3.16: Primer vtičnika

3.3 GraphQL Java

GraphQL Java [18] je strežniška implementacija GraphQLa v Javi.

3.3.1 Definiranje sheme

Shemo lahko definiramo programsko v obliki kode (izsek 3.18) ali v jeziku SDL (izsek 3.17). V našem primeru smo definirali tip *Knjiga* s poljema *id* in

naslov.

```
type Knjiga {
    id: ID!
    naslov: String!
}
```

Izsek 3.17: Shema v jeziku SDL

```
GraphQLObjectType Knjiga = newObject()
    .name("Knjiga")
    .field(newFieldDefinition()
            .name("id")
            .type(GraphQLID))
    .field(newFieldDefinition()
            .name("naslov")
            .type(GraphQLString))
    .build();
```

Izsek 3.18: Programsko definirana shema

V nadaljevanju bomo uporabljali shemo v jeziku SDL.

GraphQL Java podpira skalarje tipa *String*, *Boolean*, *Int*, *Float*, *ID*, *Long*, *Short*, *Byte*, *BigDecimal*, *BigInteger*, lahko pa dodamo tudi svoje.

3.3.2 Skalarji po meri

```
scalar Datum

type Book {
    id: ID!
    naslov: String!
    datum: Datum
}
```

Izsek 3.19: Dodajanje novega skalarja v shemo

```
import java.time.*;

@Component
```

```
public class DatumScalar extends GraphQLScalarType {

    public DatumScalar() {
        super("Datum", "desc", new Coercing<LocalDate,
         String>() {
            LocalDate pretvoriVDatum(Object input) {
                ...
            }
            @Override
            public String serialize(Object dataFetcherResult)
            {
                return dataFetcherResult.toString();
            }

            @Override
            public LocalDate parseValue(Object input) {
                return pretvoriVDatum(input);
            }

            @Override
            public LocalDate parseLiteral(Object input) {
                if ((input instanceof StringValue)) {
                    String value = ((StringValue) input)
                        .getValue();
                    LocalDate datum = pretvoriVDatum(value);
                    return datum;
                }
                return null;
            }
        });
    }
}
```

Izsek 3.20: Kreiranje skalarja po meri

Pri kreaciji skalarja po meri s pomočjo GraphQL Java [20, 19] moramo najprej v shemi definirati nov skalar (izsek 3.19). Pravila vhodnih in izhodnih podatkov skalarja definiramo v razredu, ki razširja razred *GraphQLScalar-*

Type (izsek 3.20). S pomočjo metod *serialize*, *parseValue* in *parseLiteral* definiramo pravila vhodnih in izhodnih podatkov.

3.3.3 Uporaba vmesnika

```
interface Node {
    id: ID!
}

type Knjiga implements Node {
    id: ID!
    naslov: String!
}
```

Izsek 3.21: Primer uporabe vmesnika

Izsek 3.21 prikazuje primer uporabe vmesnika, ko tip *Knjiga* implementira tip *Node*.

3.3.4 Unija tipov

```
type Knjiga {
    id: ID!
    naslov: String!
    zvrst: String
}

type Revija {
    id: ID!
    naslov: String!
    stevilka: Int
}

union Literatura = Knjiga | Revija
```

Izsek 3.22: Primer unije tipov

Izsek 3.22 prikazuje primer unije tipov.

3.3.5 Ravnanje z napakami

GraphQL knjižnica v Javi ima že predefinirano napako *GraphQLException*. Če želimo implementirati svojo napako, moramo dodati razred, ki razširja razred *RuntimeException* in implementira *GraphQLError* (izsek 3.23). V metodah *getLocations* in *getErrorType* vračamo vrednost *null*, saj ju privzet krmilnik izjem ignorira [29]. V metodi *getExtensions* lahko dodamo poljubne razširitve. Na izseku 3.24 je primer vračanja napake.

```
public class MojaNapaka extends RuntimeException implements
    GraphQLError {

    public MojaNapaka(String message) {
        super(message);
    }

    @Override
    public Map<String, Object> getExtensions() {
        return Collections.singletonMap("extension1", "ext");
    }

    @Override
    public List<SourceLocation> getLocations() {
        return null;
    }

    @Override
    public ErrorType getErrorType() {
        return null;
    }
}
```

Izsek 3.23: Implementacija napake

```
public DataFetcher vrniNapako = new DataFetcher() {
        @Override
        public Object get(DataFetchingEnvironment
```

```
         environment) {
            throw new MojaNapaka("napaka");
        }
    };
```

Izsek 3.24: Vračanje napake

3.3.6 Nalagalec podatkov

```
BatchLoader<Integer, Knjiga> knjigaBatchLoader =
 new BatchLoader<Integer, Knjiga>() {
    @Override
    public CompletionStage<List<Knjiga>> load(List<Integer>
     ids) {
        return CompletableFuture.supplyAsync(() -> {
            return knjigaManager.pridobiKnjige(ids);
        });
    }
};

DataLoader<Integer, Knjiga> knjigaLoader = DataLoader
    .newDataLoader(knjigaBatchLoader);
```

Izsek 3.25: Inicializacija nalagalca podatkov

Izsek 3.25 prikazuje inicializacijo nalagalca podatkov [22].

3.3.7 Vmesna programska oprema

Monitoriranje in logiranje lahko opravimo z instrumentacijo, ki omogoča opazovanje izvajanja poizvedbe, ter spreminjanje obnašanja (izsek 3.26). Na izseku 3.27 je prikazana vpeljava instrumentacije v GraphQL kot vmesna oprema.

```
class CustomInstrumentationState implements
    InstrumentationState {
}

class CustomInstrumentation extends SimpleInstrumentation {
```

```
@Override
public InstrumentationState createState() {
    // izvede na začetku
    return new CustomInstrumentationState();
}

@Override
public InstrumentationContext<ExecutionResult>
 beginExecution(InstrumentationExecutionParameters
 parameters) {
    // izvede za createState, na začetku poizvedbe
    return new SimpleInstrumentationContext
     <ExecutionResult>() {
        @Override
        public void onCompleted(ExecutionResult result,
         Throwable t) {
            // izvede po končani poizvedbi
        }
    };
}

@Override
public DataFetcher<?> instrumentDataFetcher(
 DataFetcher<?> dataFetcher,
 InstrumentationFieldFetchParameters parameters) {
    // izvede za vsako razreševalno funkcijo
    return dataFetcher;
}

@Override
public CompletableFuture<ExecutionResult>
 instrumentExecutionResult(ExecutionResult
 executionResult, InstrumentationExecutionParameters
 parameters) {
    // izvede čisto na koncu
    return CompletableFuture.completedFuture
     (executionResult);
}
```

```
}
```

Izsek 3.26: Instrumentacija po meri

```
CustomInstrumentation instrumentation = new
    CustomInstrumentation();

this.graphQL = GraphQL.newGraphQL(graphQLSchema)
        .instrumentation(instrumentation)
        .build();
```

Izsek 3.27: Dodajanje instrumentacije kot vmesna oprema

3.4 Primerjava predstavljenih platform

Vse tri platforme, ki smo jih primerjali med sabo, podpirajo večino primerjalnih kriterijev iz tabele 3.1.

Graphene Python je edina platforma, ki ne podpira zapisa sheme v obliki SDL. Podprt je le programski zapis sheme, ki ga pa je veliko težje razbrati, ter nadgrajevati.

Izdelava skalarjev po meri razvijalcem omogoča definiranje novih podatkovnih tipov (npr. datum), lahko pa služi tudi kot validator obstoječih tipov. V večini primerov za delovanje APIja ne potrebujemo novega sklarja po meri, zato ta kriterij ni tako pomemben, kljub temu pa ga podpirajo vse tri platforme.

Po poslani zahtevi odjemalec pričakuje podatke. Če med razreševanjem zahteve pride do napake in podatkov ne moremo vrniti, moramo odjemalca o napaki nujno obvestiti. Pomemben je predvsem razlog napake, saj odjemalcu pove, kako naj reagira na napako (npr.: vir trenutno ni na voljo, napačna avtentikacija, manjkajoči podatki). Vse tri platforme podpirajo privzeto napako GraphQL, poleg tega pa ima Apollo Server predefinirane še tri druge, bolj specifične tipe napak: napaka avtentikacije (*AuthenticationError*), prepovedan dostop (*ForbiddenError*) in napačen vnos podatkov (*UserInputError*).

Kot smo že omenili, je funkcija nalagalca podatkov zbiranje zahtev do baze, pri čemer si duplikate zahtev shrani le enkrat. Vse podatke zahtev nato pridobi z le enim dostopom do baze. Nalagalec podatkov lahko pospeši delovanje aplikacij, kjer se v poizvedbah večkrat dostopa do enakega tipa podatkov ali če v poizvedbah večkrat zahtevamo enak podatek. Vse tri platforme podpirajo vpeljavo nalagalca podatkov.

Logiranje in monitoriranje strežniške aktivnosti je lahko zelo pomembno. Razvijalci lahko tako razvidijo, kateri podatki so najbolj zahtevani ter kateri se skoraj ne uporabljajo, kar jim omogoči prilagajanje API strežnika. Vse tri platforme podpirajo vpeljavo vmesne opreme, ki se uporablja za pregled in spremembo poizvedb (npr. logiranje, avtentikacija). Poleg tega pa se lahko Apollo Server poveže tudi z Apollovim Studiem, ki omogoča dodaten nadzor in poročanje meritev.

Platforme lahko primerjamo tudi na podlagi enostavnosti uporabe in razpoložljive dokumentacije na internetu. Največ dokumentacije in različnih primerov smo našli za platformo Apollo Server, poleg tega pa se nam zdi tudi najbolj enostavna za uporabo in konfiguracijo.

	Graphene Python	**Apollo Server**	**GraphQL Java**
SDL zapis sheme	ne	da	da
skalar po meri	da	da	da
ravnanje z napakami	da	da	da
nalagalec podatkov	da	da	da
vmesna oprema	da	da	da
monitoriranje	ne	da	ne

Tabela 3.1: Podpora primerjalnih kriterijev platform Graphene Python, Apollo Server in GraphQL Java

Poglavje 4

Predpomnenje in trajne poizvedbe

V tem poglavju bomo pokazali kako lahko GraphQL strežniku dodamo funkcionalnost predpomnenja podatkov, s čimer se lahko izognemo odvečnim dostopom do baze. Pokazali bomo tudi, kako strežniku učinkovito poslati poizvedbe, sestavljene iz dolgih nizov znakov, z uporabo trajnih poizvedb.

V obeh primerih bomo za prikaz uporabili platformo Apollo Server, saj se nam zdi enostavna za uporabo in konfiguracijo, ter hkrati uporabna.

4.1 Predpomnenje

V večini aplikacij želimo podatke, ki smo jih ravno pridobili iz podatkovne baze, začasno shraniti. S tem lahko omogočimo boljšo uporabniško izkušnjo, saj ob zahtevi enakih podatkov posredujemo shranjene podatke. Izognemo se odvečnim dostopom do baze, hkrati pa odjemalcu hitreje odgovorimo na zahtevo. Pri storitvah REST je predpomnenje zelo enostavno, saj vedno pričakujemo enako strukturo podatkov, pri GraphQL pa ni vedno razvidno, katere podatke bo odjemalec pridobil. GraphQL poizvedba lahko zahteva več različnih polj. Od teh se nekaterih ne da predpomniti, nekatera se da predpomniti samo za določen čas, spet druga pa so za vse uporabnike enaka.

4.1.1 Predpomnenje s tehnologijo Apollo

Apollo Server omogoča predpomnenje polj in tipov [10]. To lahko naredimo statično v shemi z uporabo direktive ali dinamično v razreševalnih funkcijah s pomočjo info parametra.

```
import responseCachePlugin from 'apollo-server-plugin-
    response-cache';
const server = new ApolloServer({
    ...
    cacheControl: {
        defaultMaxAge: 10,
        calculateHttpHeaders: true,
        stripFormattedExtensions: false
    },
    plugins: [responseCachePlugin()],
})
```

Izsek 4.1: Dodatne nastavitve pri kreaciji strežnika

Na izseku 4.1 so prikazane dodatne nastavitve pri kreaciji strežnika, kar nam bo omogočilo predpomnenje. S parametrom *defaultMaxAge* nastavimo koliko sekund naj se hranijo pridobljeni podatki v shrambi, če jim sami tega nismo eksplicitno določili. S parametrom *calculateHttpHeaders* v glavo odgovora dodamo *cache-control*, ki vsebuje nastavljeno maximalno starost podatkov (angl. *maxAge*) in okvir (angl. *scope*). V glavo odgovora se doda tudi polje *age*, ki vsebuje trenutno starost podatkov v predpomnilniku. S parametrom *stripFormattedExtensions* v odgovoru ohranimo polje *extensions*, kjer so zapisani podatki o predpomnenju. S parametrom *plugin* določimo vtičnik, ki ga bomo namestili ob kreaciji strežnika. Z njim bo Apollo hranil podatke v predpomnilniku.

Če želimo hraniti podatke različno za različne uporabnike, moramo ustreznim poljem nastaviti okvir (angl. *scope*) na *PRIVATE*. Poleg tega moramo vtičniku povedati, kako naj loči med uporabniki.

```
directive @cacheControl(
    maxAge: Int,
```

```
    scope: CacheControlScope
) on OBJECT | FIELD_DEFINITION

enum CacheControlScope {
    PUBLIC
    PRIVATE
}

type Query {
    vrniAvtorje: [Avtor!] @cacheControl(maxAge: 30, scope:
   PUBLIC)
}
```

Izsek 4.2: Dodajanje direktive *cacheControl* v shemo

Na izseku 4.2 je prikazano dodajanje direktive *cacheControl* [7] in njena uporaba na poizvedbi *vrniAvtorje*. Z nastavitvijo *maxAge* smo povozili *defaultMaxAge* in jo nastavili na 30 sekund. Z nastavitvijo *scope* smo določili, da za različne uporabnike ni potrebno različno shranjevanje podatkov.

Če sedaj pošljemo več poizvedb z imenom *vrniAvtorje* v roku tridesetih sekund, bo le prva poizvedba dostopala do baze podatkov, drugim pa bodo vrnjeni podatki iz predpomnilnika. Prva poizvedba po tridesetih sekundah pa bo zopet dostopala do baze.

```
{
  "data": {...}
  "extensions": {
    "tracing": {...}
    "cacheControl": {
      "version": 1,
      "hints": [
        {
          "path": [
            "vrniAvtorje"
          ],
          "maxAge": 30,
          "scope": "PUBLIC"
        }
```

```
      ]
    }
  }
}
```

Izsek 4.3: Vrnjeni podatki prve poizvedbe, ki dostopa do baze

Na izseku 4.3 so prikazani vrnjeni podatki prve poizvedbe, ki dostopa do baze.

```
{
  "data": {...}
  "extensions": {
        "cacheControl": {
            "version": 1,
            "hints": []
        }
    }
}
```

Izsek 4.4: Vrnjeni podatki ostalih poizvedb v roku 30 sekund, ki ne dostopajo do baze

Na izseku 4.4 so prikazani vrnjeni podatki ostalih poizvedb v roku 30 sekund, ki ne dostopajo do baze, ampak pridobijo podatke iz predpomnilnika.

4.2 Trajne poizvedbe

Poizvedbe, ki so sestavljene iz dolgih nizov znakov, lahko porabijo zelo veliko pasovne širine. To lahko odpravimo z uporabo trajnih poizvedb (angl. *persisted queries*). Odjemalec strežniku namesto celotne poizvedbe pošlje le njen ID ali zgoščeno vrednost (angl. *hash*). Strežnik na drugi strani hrani slovar poizvedb in njihovih vrednosti. V primeru, da odjemalec pošlje še neznano vrednost strežniku, ta lahko poizvedbo zavrne ali pa odjemalca prosi, naj zraven vrednosti posreduje tudi poizvedbo. Če odjemalec poizvedbo posreduje, si strežnik lahko shrani novo vrednost in poizvedbo.

4.2.1 Trajne poizvedbe s tehnologijo Apollo

Apollo omogoča uporabo trajnih poizvedb, brez dodatnih konfiguracij [5]. Če pošiljamo poizvedbo strežniku prvič, moramo poleg zgoščene vrednosti poslati tudi poizvedbo. To storimo s HTTP metodo *GET*.

```
GET http://localhost:4000/?query=query {vrniAvtorje {
id ime letnicaRojstva}}&extensions={"persistedQuery":
{"version":1,"sha256Hash":"0a20b8b5e63a24ed8ed96001ae
1735597624a02b32caf227bba0d0e93fe83c7f"}}
```

Izsek 4.5: Shranjevanje trajne poizvedbe

Izsek 4.5 prikazuje HTTP zahtevo, s katero shranimo poizvedbo na strani strežnika. URL vsebuje dva parametra: *query*, kamor zapišemo GraphQL poizvedbo, ter *extensions*, kamor zapišemo zgoščeno vrednost poizvedbe.

```
GET http://localhost:4000/?extensions={"persistedQuery":{
"version":1,"sha256Hash":"0a20b8b5e63a24ed8ed96001ae17355
97624a02b32caf227bba0d0e93fe83c7f"}}
```

Izsek 4.6: Zahteva trajne poizvedbe

Izsek 4.6 prikazuje HTTP zahtevo, s katero zahtevamo trajno poizvedbo, shranjeno na strežniku. Če strežnik ima shranjeno poizvedbo z enako zgoščeno vrednostjo, bo poizvedbo izvedel in odjemalcu poslal podatke. V nasprotnem primeru, če strežnik nima shranjene poizvedbe z enako zgoščeno vrednostjo, bo odjemalcu sporočil, da trajne poizvedbe ni našel.

```
{
    "errors": [
        {
            "message": "PersistedQueryNotFound",
            "extensions": {
                "code": "PERSISTED_QUERY_NOT_FOUND",
                "exception": {
                    "stacktrace": [...]
                }
            }
        }
```

```
    ]
}
```

Izsek 4.7: Trajna poizvedba ni najdena

Izsek 4.7 prikazuje odgovor strežnika v primeru, da za določeno zgoščeno vrednost ne najde poizvedbe.

4.3 Evalvacija

S primerom smo pokazali uporabo predpomnenja v GraphQL APIju narejenim s pomočjo Apollo Serverja. Tehnika je uporabna predvsem v primerih, ko se podatki v bazi redko spreminjajo ali pa večkrat zahtevamo enake podatke. Nadzor nad izbiro polj za predpomnenje ter dolžino časa predpomnenja je zelo dober, saj parametre nastavimo v shemi poleg samih polj.

Pokazali smo tudi primer uporabe trajnih poizvedb, ki jih lahko uporabimo namesto pošiljanja dolgih poizvedb oziroma, ko želimo privarčevati na pasovni širini.

Poglavje 5

Učinkovite poizvedbe z uporabo avtorizacije

Avtentikacija in avtorizacija sta danes zelo pogosti zahtevi aplikacij, saj aplikacije vsebujejo različne vrste zaupnih podatkov. Za primer lahko vzamemo podatke o knjigah, kot sta naslov in avtor, ter podatke o uporabnikih, kot sta ime in kraj bivanja. Do podatkov o knjigah lahko dostopa vsak uporabnik, dostop do podatkov uporabnikov pa bi želeli omejiti, da lahko z njimi upravljajo le določene osebe. To lahko naredimo s pomočjo avtorizacije, ki pri vsaki zahtevi do podatkov preveri, ali ima odjemalec pravice do dostopa podatkov, ki jih zahteva. Vsak API, ki ima implementirano avtentikacijo/avtorizacijo, mora odgovoriti na tri vprašanja[16]:

1. Ali so zahtevani podatki zasebni?

2. Ali zahteva vsebuje podatke o avtentikaciji/avtorizaciji?

3. Ali je ta informacija veljavna?

Glede na ta vprašanja želimo narediti sistem avtorizacije, ki se proži le ob zahtevah, ki želijo dostopati do zasebnih podatkov, in ne ob vsaki. Poleg tega mora sistem iz zahteve razbrati podatke o uporabniku, ter obdelati morebitne napake.

V nadaljevanju poglavja bomo opisali dva pristopa k avtorizaciji s pomočjo platforme Apollo Server. Platformo smo izbrali zaradi enostavne uporabe in konfiguracije, ter njene uporabnosti. V obeh primerih bomo uporabili standard OpenID Connect, ki predstavlja plast nad ogrodjem OAuth 2.0. OAuth 2.0 se tipično uporablja, da aplikacijam dovolimo dostop do svojih podatkov drugih aplikacij [25]. OpenID Connect standardnemu OAuth 2.0 toku doda JSON Web Token (JWT), ki se posreduje odjemalcu ob uspešni prijavi [26].

JWT je standard, ki definira varno prenašanje informacij med dvema skupinama v obliki JSON objekta [23]. Vsaki poslani zahtevi, ki vsebuje JWT, lahko preverimo integriteto, saj vsebuje digitalni podpis. Digitalni podpis lahko naredimo z uporabo skrivnosti ali para zasebnega in javnega ključa. JWT je sestavljen iz treh delov, ločenih s piko, in sicer glave (angl. *header*), koristne vsebine (angl. *payload*) in podpisa (angl. *signature*). Glava ponavadi vsebuje tip žetona in algoritem podpisa. Koristna vsebina vsebuje trditve o entiteti (npr. uporabniku) ter dodatne podatke. V podpisu z algoritmom v glavi skupaj podpišemo zakodirano glavo in koristno vsebino.

V večini aplikacij REST se za avtorizacijo uporablja id seje odjemalca. Ob uspešni prijavi strežnik uporabniku posreduje id seje, ki jo mora odjemalec posredovati ob vsaki ponovni zahtevi podatkov s strežnika. Strežnik sejo vsakega odjemalca hrani v svojem pomnilniku, ki ga mu lahko ob velikem številu sej zmanjka. Pri uporabi JWT pa strežnik ob uspešni prijavi odjemalcu posreduje žeton, podpisan s skrivnostjo, ki jo ve le strežnik. Strežniku si ni treba hraniti uporabniške seje, saj lahko z žetonom in skrivnostjo ob vsaki zahtevi preveri uporabnika. Prednost JWT se pokaže tudi pri uporabi več mikrostoritev. Odjemalec se lahko avtorizira pri vseh mikrostoritvah z enakim JWT, če mikrostoritve uporabljajo enako skrivnost.

5.1 Ovijanje razreševalnih funkcij

Pri tem pristopu ovijemo razreševalne funkcije s funkcijo, ki preverja veljavnost žetona JWT ter dovoljenja, ki ga ima odjemalec shranjenega na tem

žetonu. Dobra stran tega pristopa je, da lahko ovijemo le tiste razreševalne funkcije, ki potrebujejo takšno preverjanje.

```
preveriJwtInDovoljenje = (context, pricakovanoDovoljenje) =>
   {
    const token = context.req.headers.authorization;
    try {
      const jwtPayload = jwt.verify(token.replace(
        "Bearer ", ""), secretKey);
      const imaDovoljenje = jwtPayload.dovoljenja.includes(
       pricakovanoDovoljenje);
      if(!pricakovanoDovoljenje.length || imaDovoljenje) {
        return true;
      } else {
        throw new Error();
      }
    } catch(err) {
        throw new AuthenticationError("Napaka pri
            avtorizaciji");
    }
}
```

Izsek 5.1: Primer funkcije, ki preveri veljavnost JWT žetona in dovoljenja zapisana na njem

Na izseku 5.1 imamo primer funkcije, ki preverja avtentikacijo in avtorizacijo. Funkcija sprejeme dva argumenta: *context* in *pricakovanoDovoljenje*. *Context* je objekt, do katerega imajo dostop vse razreševalne funkcije, ter lahko vsebuje različne informacije, med katere spada tudi glava zahteve in podatki avtentikacije. Argument *pricakovanoDovoljenje* pa predstavlja dovoljenje, ki ga mora imeti odjemalec za dostop do podatkov (npr. upravljalec). Funkcija najprej iz argumenta *Context* izlušči žeton JWT, ter ga na podlagi skrivnega ključa preveri. Nato iz žetona JWT pridobi informacijo o dovoljenjih, ter preveri, če vsebujejo pričakovano dovoljenje. Če *pricakovanoDovoljenje* ni podano ali če ima odjemalec zahtevano dovoljenje, funkcija vrne true, v nasprotnem primeru pa vrne avtorizacijsko napako.

```
{"authorization": "Bearer yJhbGciOiJIUzI1Ni..."}
```

Izsek 5.2: Podatek, ki ga pošljemo v glavi zahteve

Izsek 5.2 prikazuje podatek, ki ga moramo skupaj z zahtevo, poslati v glavi.

```
{
  "errors": [
    {
      "message": "Napaka pri avtorizaciji",
      ...
      "path": [
        "dodajKnjigo"
      ],
      ...
    }
  ],
  "data": {
    "dodajKnjigo": null
  }
}
```

Izsek 5.3: Primer avtorizacijske napake

Izsek 5.3 prikazuje primer, ko strežnik naše zahteve ni mogel avtorizirati. V takem primeru strežnik GraphQL vrne dva objekta: *errors* in *data*. Objekt *errors* vsebuje informacije o napaki, medtem ko je objekt *data* prazen (angl. *null*).

5.2 Ustvarjanje direktiv

Alternativen pristop prejšnjemu je pristop z ustvarjanjem direktiv. Podobno zopet preverimo veljavnost žetona JWT in dovoljenja, ki jih vsebuje. Prednost pred prejšnjim pristopom je predvsem v jasnosti, katere poizvedbe potrebujejo avtorizacijo, hkrati pa nam omogoča lažje spreminjanje in dodajanje novih zahtev po avtorizaciji.

```
directive @auth(potrebnoDovoljenje: Role) on FIELD_DEFINITION

enum Role {
    UPRAVLJALEC
    UPORABNIK
}

type Query {
    vrniKnjige: [Knjiga!] @auth(potrebnoDovoljenje: UPORABNIK
     )
}
```

Izsek 5.4: Sprememba sheme za dodajanje nove direktive

Izsek 5.4 prikazuje spremembo sheme, ki jo moramo narediti, če želimo dodati novo direktivo [17]. Najprej definiramo direktivo in jo poimenujemo *auth*. Direktiva sprejme argument *potrebnoDovoljenje* tipa *Role*, ki je enum in lahko zaseda vrednosti *upravljalec* in *uporabnik*. Na koncu definiramo še, kje se bo direktiva uporabljala. Poizvedbi *vrniKnjige* smo dodali direktivo, ter nastavili potrebnoDovoljenje na *UPORABNIK*. Če bo odjemalec imel pravilno dovoljenje (v tem primeru *UPORABNIK*), bo lahko dostopal do podatkov.

```
class DovoljenjeDirective extends SchemaDirectiveVisitor {
    visitFieldDefinition(field) {
        const { potrebnoDovoljenje } = this.args;
        const { resolve = defaultFieldResolver } = field;
        field.resolve = async function(...args) {
            const context = args[2];
            const token = context.req.headers.authorization;
            try {
                const jwtPayload =
                    jwt.verify(token.replace(
                        "Bearer ", ""), secretKey);
                const imaDovoljenje =
                    jwtPayload.dovoljenja.includes(
                        potrebnoDovoljenje);
                if(!potrebnoDovoljenje.length ||
```

```
                    imaDovoljenje) {
                    return resolve.apply(this, args);
                } else {
                    throw new Error();
                }
            } catch(err) {
                throw new AuthenticationError(
                    "Napaka pri avtorizaciji");
            }
        }
    }
}
```

Izsek 5.5: Razred, ki obravnava novo direktivo

Na izseku 5.5 je implementiran razred *DovoljenjeDirective*, ki razširja razred *SchemaDirectiveVisitor*. Razred je namenjen temu, da obišče določene dele naše sheme. V našem primeru smo definirali, da se bo shema uporabljala na definiciji polj (angl. *field definition*), zato, razredu dodamo metodo *visitFieldDefinition*. Zatem iz argumenta direktive pridobimo *potrebnoDovoljenje* in ga shranimo. Vsakič, ko se pokliče polje, ki je označeno z direktivo, se bo izvedel podoben postopek avtentikacije in avtorizacije kot v prejšnjem primeru. Preveri se veljavnost žetona, ter primerja odjemalčevo dovoljenje s potrebnim dovoljenjem. V primeru nepravilnosti se odjemalcu pošlje napaka, drugače pa se nadaljuje z izvajanjem razreševalne funkcije.

5.3 Evalvacija

S primeroma smo pokazali dva preprosta in učinkovita načina uporabe avtorizacije v GraphQL APIju narejenim s pomočjo Apollo Serverja. V obeh primerih se proces avtorizacije izvede le v poizvedbah, ki zahtevajo zasebne podatke. Avtorizacijo z ustvarjanjem direktiv je po našem mnenju nekoliko težje implementirati, vendar lažje vpeljati spremembe in novosti. Primeroma lahko dodamo poljubno število različnih pravic, pri avtorizaciji pa jih lahko zahtevamo tudi po več naenkrat.

Poglavje 6

Brezstrežniško skaliranje

Če želimo, da so aplikacije uporabnikom dostopne tudi v primerih velikih obremenitev, potrebujemo nekakšen mehanizem skaliranja. Mehanizem skaliranja je dober, če se prilagaja potrebam in niso vsi računski viri vedno v teku. Takšen primer je brezstrežniško računanje (angl. *serverless computing*) [28].

Brezstrežniško računanje je model oblačnega računanja, kjer lastnik oblačne storitve poganja strežnik, ter dinamično prilagaja računske vire potrebam. Koda, ki jo pošljemo ponudniku, je v obliki funkcije, ta pa se običajno izvaja v vsebnikih brez stanja, ki se proži ob različnih dogodkih (npr. HTTP poizvedbe, planirani dogodki, nalaganje datotek) [27]. Zaradi tega lahko predpostavimo, da se bo izvedba vsake funkcije izvedla v drugem vsebniku, ter ne bo imela dostopa do konteksta prejšnje funkcije. Kljub temu da brezstrežniško računanje zakrije osnovno infrastrukturo razvijalcem, strežnik še vedno izvaja funkcije.

Model je stroškovno zelo učinkovit, saj velja, da plačaš, kolikor porabiš in ne nekega vnaprejšnjega, fiksnega paketa. Poenostavi se tudi delo razvijalcev zalednega dela, saj se jim ni treba ukvarjati z ročnim skaliranjem, ampak se to izvede samodejno. V primeru napak te ne ustavijo strežnika.

Kot vsaka arhitektura pa ima tudi brezstrežniško računanje svoje slabosti. Redko uporabljene aplikacije imajo potencialno lahko zelo počasno

odzivnost. Po določenem času neuporabe bo nov zagon aplikacije trajal dlje, saj bo morala oblačna storitev na novo zagnati računske vire. Slaba stran je tudi diagnosticiranje in razreševanje napak, saj težje pridemo do potrebnih podatkov kot pri klasičnih strežnikih. Problem je lahko tudi varnost, saj ima aplikacija več komponent kot klasična arhitektura, vsaka komponenta pa je lahko vstopna točka do brezstrežniške aplikacije.

Obstaja več ponudnikov brezstrežniškega skaliranja, ki jih lahko uporabimo za GraphQL. Med najbolj uporabljene spadajo Amazon Web Service (AWS) Lambda, ki je bil ustvarjen s strani Amazona, Azure Functions, ki je bil ustvarjen s strani Microsofta in je dostopen preko Microsoft Azure, Google Cloud Functions, ki je bil ustvarjen s strani Googla in je dostopen preko Google Cloud Platform, IBM Cloud Functions, ki je bil ustvarjen s strani IBMa. V nadaljevanju poglavja bomo opisali postopek postavitve GraphQL APIja na platformo AWS Lambda [6] s pomočjo funkcij Netlify [24]. Strežnik bomo napisali v jeziku JavaScript s pomočjo knjižnice Apollo Server lambda.

Funkcije Netlify omogočajo postavitev strežniške kode na platformo AWS Lambda. Kodo lahko prenesemo preko GitHuba ali pa kar neposredno. Koda funkcij je skrita javnosti, kljub temu pa z njo lahko komuniciramo, kot z vsakim APIjem.

6.1 GraphQL API s funkcijami Netlify

Za postavitev GraphQL APIja s pomočjo funkcij Netlify, najprej potrebujemo dodati novo datoteko *graphql.js* v mapo *functions* [2]. Vse kar dodamo v mapo *functions*, bo Netlify naredil javno vidno, ter omogočil dostop preko naslova URL: https://ime_nase_strani/.netlify/functions/ .

```
const { ApolloServer, gql } = require('apollo-server-lambda')

const typeDefs = gql`
  type Query {
    pozdrav: String
  }
```

```
‘


const resolvers = {
  Query: {
    hello: (root, args, context) => {
      return ‘Pozdravljeni‘
    }
  }
}

const server = new ApolloServer({
  typeDefs,
  resolvers,
  introspection: true,
  playground: true,
})

exports.handler = server.createHandler()
```

Izsek 6.1: Primer kreiranja preprostega strežnika Apollo Lambda v datoteki *graphql.js*

Na izseku 6.1 je prikazano kreiranje preprostega Apollovega strežnika. Edina razlika med kreiranjem klasičnega Apollovega strežnika je v tem, da tukaj funkcijo za kreiranje strežnika pridobimo iz knjižnice *apollo-server-lambda*. Pomembno je tudi, da na koncu izvozimo krmilnik Lambda funkcije (*server.createHandler()*).

V naslednjem koraku bomo v mapo site dodali datoteko *index.html*, ki bo predstavljala našo osnovno stran.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <title>GraphQL API</title>
</head>

<body>
```

```
    <h1>Dobrodošli na strani</h1>
</body>
```

Izsek 6.2: Primer osnovne strani, ki se nahaja v datoteki *site/index.html*

Na izseku 6.2 je prikazan primer osnovne strani, ki jo bomo poslali Netlifyju.

V naslednjem koraku bomo v glavno mapo projekta dodali mapo *netlify.toml*, kjer Netlifyju sporočimo, kaj naj ob izgradnji naredi [30].

```
[build]
  publish = "site"
  functions = "functions"
```

Izsek 6.3: Primer datoteke *netlify.toml*

Na izseku 6.3 je prikazan primer preproste *netlify.toml* datoteke. V njej povemo, kje se nahaja naša spletna stran (spremenljivka *publish*), ter kje se nahaja *graphql.js* datoteka (spremenljivka *functions*).

Ko imamo vse datoteke pripravljene in ustvarjen račun Netlify, je potrebno funkcijam Netlify sporočiti, da želimo objaviti API strežnik. Za to imamo na razpolago dve možnosti. Prva možnost je, da mapo z našim projektom neposredno primemo in spustimo v brskalnik uporabniškega vmesnika aplikacije Netlify (angl. *drag and drop*). Druga možnost pa je, da funkcijam Netlify posredujemo naš GitHub direktorij. Potrebno je slediti naslednjim ukazom [3]:

1. V terminal v korenski mapi vpišemo: *netlify init*.
2. Izberemo: *Create and Configure a new site*.
3. Določimo ime skupine in ime strani.
4. Stran je sedaj kreirana.
5. Po potrebi dodamo še ukaz za izgradnjo (angl. *build command*) in spremenimo postavitveno mapo (angl. *directory to deploy*).

Netlify bo sedaj postavil našo spletno stran, ki je dostopna na URL naslovu https://imeStrani.netlify.app, in API, ki je dostopen na URL naslovu https://imeStrani.netlify.app/.netlify/functions/graphql. Od sedaj naprej bo kakršnakoli sprememba na git repozitoriju povzročila ponovno postavitev spletne strani in APIja. Netlify bo tudi poskrbel za avtomatično skaliranje GraphQL APija.

6.2 Evalvacija

S preprostim primerom smo pokazali objavo APIja na brezstrežniško platformo AWS Lambda, ki bo namesto nas poskrbela za samodejno skaliranje računskih virov ter varnost APIja. Za objavo APIja pri drugem ponudniku je potrebno kodo le malenkostno spremeniti. Način zelo dobro podpira spremembe oziroma nadgradnjo APIja, saj spremenjeno kodo preprosto posredujemo Gitu, Netlify pa bo to objavil na platformo AWS Lambda.

Poglavje 7

Zaključek

V diplomski nalogi smo predstavili ključne lastnosti tehnologije GraphQL, ter jo primerjali s tehnologijo REST. Spoznali smo, da je ključna razlika med tehnologijama v pošiljanju podatkov. Tehnologija REST vedno vrača fiksno obliko podatkov in je primerna za aplikacije, kjer se oblika zahtevanih podatkov in struktura podatkov v bazi redko spreminja. Tehnologija GraphQL pa odjemalcu vrne le tiste podatke, katere je zahteval, zaradi česar je primernejša za aplikacije, kjer je pomembna količina prenesenih podatkov in kjer se lahko struktura zahtevanih podatkov stalno spreminja.

Primerjava različnih platform GraphQL je pokazala, da lahko z vsako platformo naredimo skoraj vse primerjane funkcionalnosti. Razlika pa se je pokazala predvsem v enostavnosti uporabe in količine razpoložljive dokumentacije in primerov. Največ primerov uporabe in dokumentacije smo našli za platformo Apollo Server, ter jo spoznali tudi za najbolj primerno in enostavno za novega razvijalca.

Ker se podatki v bazah aplikacijskih strežnikov večino časa ne spreminjajo, smo želeli te podatke začasno shraniti. Apollov strežnik omogoča, da vsakemu tipu ali polju posebej določimo, za koliko časa bomo podatke shranili v predpomnilnik, preden jih bomo zopet pridobili iz baze. Tak način se izkaže za učinkovitega, ko odjemalec v kratkem času zahteva enake podatke, ali pa se podatki v bazi redko spreminjajo.

Poizvedbe, ki so sestavljene iz dolgih nizov znakov, lahko zasedejo veliko pasovne širine in zahtevajo dlje časa, da jih strežnik obdela, kar pa lahko negativno vpliva na uporabniško izkušnjo. To smo rešili z uporabo trajnih poizvedb, s pomočjo katerih strežniku posredujemo le zgoščeno vrednost poizvedbe. Celotno poizvedbo mora strežnik pridobiti le enkrat, da si jo lahko zapomni.

Dostop do podatkov želimo velikokrat omejiti, zato smo se v diplomski nalogi ukvarjali tudi z učinkovito avtentikacijo in avtorizacijo ter izpostavili dva načina. Tako kot pri ovijanju razreševalnih funkcij, kot tudi pri ustvarjanju direktiv, se avtentikacija preverja le pri poljih, ki to zahtevajo, in ne pri vsaki poizvedbi. Pristop z uporabo direktiv, se nam zdi primernejši, saj je veliko bolj jasen ter omogoča lažje spreminjanje.

V diplomski nalogi smo se ukvarjali tudi z objavo API strežnika preko oblačne storitve in samodejnega skaliranja. To smo naredili s pomočjo brezstrežniškega računanja, kjer oblačna storitev samodejno prilagaja računske vire potrebam. Model je praktičen predvsem s finančnega vidika, saj kupec plača le toliko storitev, kot jih porabi njegov API. Primer APIja smo uspešno objavili na platformo AWS Lambda s pomočjo funkcij Netlify.

GraphQL je bil prvotno razvit z namenom povečanja učinkovitosti pridobivanja podatkov na mobilnih napravah. Ker tako mobilne kot tudi ostale aplikacije postajajo vse kompleksnejše pri zahtevi podatkov, je zanje GraphQL najbolj primeren, zaradi česar bo v prihodnosti GraphQL verjetno pridobival na pomenu. Poizvedovalni jezik je še relativno nov in v razvoju, zato v nekaterih programskih jezikih še niso podprte vse funkcionalnosti. GraphQL podpira veliko možnosti izboljšav. V tej diplomski nalogi smo se ukvarjali z izboljšavami poizvedovanja na strani strežnika, dodatne izboljšave pa lahko izvedemo tudi na strani odjemalca, kar bi učinkovitost poizvedb le še izboljšalo.

Literatura

[1] Francisco J. García-Peñalvo Andrea Vázquez-Ingelmo, Juan Cruz-Benito. Improving the oeeu's data-driven technological ecosystem's interoperability with graphql. *TEEM 2017: Proceedings of the 5th International Conference on Technological Ecosystems for Enhancing Multiculturality*, 52(89):1–8, 2017.

[2] Deploying with netlify functions. Dosegljivo: `https://www.apollographql.com/docs/apollo-server/deployment/netlify/`. [Dostopano: 23.6.2020].

[3] Netlify functions + graphql. Dosegljivo: `https://github.com/netlify-labs/functions-and-graphql`. [Dostopano: 23.6.2020].

[4] Apollo server. Dosegljivo: `https://www.apollographql.com/docs/apollo-server/`. [Dostopano: 23.6.2020].

[5] Automatic persisted queries. Dosegljivo: `https://www.apollographql.com/docs/apollo-server/performance/apq/`. [Dostopano: 23.6.2020].

[6] Aws lambda. Dosegljivo: `https://en.wikipedia.org/wiki/AWS_Lambda`. [Dostopano: 23.6.2020].

[7] Prabhakar Borah. Apollo server caching : Getting it right ! Dosegljivo: `https://medium.com/@prabhakar.borah/apollo-server-caching-getting-it-right-76e3dcd200c4`. [Dostopano: 23.6.2020].

[8] Nikolas Burk. How to wrap a rest api with graphql - a 3-step tutorial. Dosegljivo: https://www.prisma.io/blog/how-to-wrap-a-rest-api-with-graphql-8bf3fb17547d. [Dostopano: 23.6.2020].

[9] Nikolas Burk. Top 5 reasons to use graphql. Dosegljivo: https://www.prisma.io/blog/top-5-reasons-to-use-graphql-b60cfa683511. [Dostopano: 23.6.2020].

[10] Caching. Dosegljivo: https://www.apollographql.com/docs/apollo-server/performance/caching/. [Dostopano: 23.6.2020].

[11] Code. Dosegljivo: https://graphql.org/code/. [Dostopano: 4.9.2020].

[12] Thomas Eizinger. Api design in distributed systems: A comparison between graphql and rest. Master thesis, FH Technikum Wien, University of Applied Sciences, 2017.

[13] Alex Banks Eve Porcello. *Learning GraphQL: Declarative Data Fetching for Modern Web Apps.* O'Reilly Media, Inc., 2018.

[14] Graphene python. Dosegljivo: https://graphene-python.org/. [Dostopano: 23.6.2020].

[15] Graphql. Dosegljivo: https://graphql.org/. [Dostopano: 23.6.2020].

[16] Apollo GraphQL. Ryan chenkie - handling authentication and authorization in graphql. Dosegljivo: https://www.youtube.com/watch?v=4_Bcw7BULC8, 2017. [Dostopano: 23.6.2020].

[17] Apollo GraphQL. Supercharge your schemas with custom directives (ryan chenkie). Dosegljivo: https://www.youtube.com/watch?v=Gc8bSXX7oyU, 2018. [Dostopano: 23.6.2020].

[18] Graphql java. Dosegljivo: https://www.graphql-java.com/. [Dostopano: 23.6.2020].

[19] graphql-java. Dosegljivo: https://github.com/graphql-java/graphql-java/blob/master/src/main/java/graphql/Scalars.java. [Dostopano: 6.7.2020].

[20] graphql-java-datetime. Dosegljivo: https://github.com/donbeave/graphql-java-datetime. [Dostopano: 6.7.2020].

[21] Jonas Helfer. Graphql explained. Dosegljivo: https://www.apollographql.com/blog/graphql-explained-5844742f195e. [Dostopano: 23.6.2020].

[22] java-dataloader. Dosegljivo: https://github.com/graphql-java/java-dataloader. [Dostopano: 6.7.2020].

[23] Json web token. Dosegljivo: https://jwt.io/. [Dostopano: 14.7.2020].

[24] Netlify. Dosegljivo: https://www.netlify.com/. [Dostopano: 23.6.2020].

[25] OktaDev. An illustrated guide to oauth and openid connect. Dosegljivo: https://www.youtube.com/watch?v=t18YB3xDfXI, 2018. [Dostopano: 23.6.2020].

[26] Openid connect explained. Dosegljivo: https://connect2id.com/learn/openid-connect. [Dostopano: 15.7.2020].

[27] Learn to build full-stack serverless apps. Dosegljivo: https://serverless-stack.com/. [Dostopano: 23.6.2020].

[28] Serverless computing. Dosegljivo: https://en.wikipedia.org/wiki/Serverless_computing. [Dostopano: 23.6.2020].

[29] Philippe Simo. Understanding graphql error handling mechanisms in spring-boot. Dosegljivo: https://medium.com/@philippechampion58/understanding-graphql-error-handling-mechanisms-in-spring-boot-604301c9bedb. [Dostopano: 6.7.2020].

[30] Khalil Stemmler. How to deploy a serverless graphql api on netlify. Dosegljivo: https://khalilstemmler.com/articles/tutorials/deploying-a-serverless-graphql-api-on-netlify/. [Dostopano: 23.6.2020].

[31] The fullstack tutorial for graphql. Dosegljivo: https://www.howtographql.com/. [Dostopano: 23.6.2020].