Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Domen Kajdič

Spletne storitve z GraphQL

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: prof. dr. Branko Matjaž Jurič

Ljubljana, 2018

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Podrobno proučite tehnologijo GraphQL, analizirajte delovanje in jo umestite napram spletnim storitvam REST. Opišite ključne koncepte arhitekture mikrostoritev v povezavi z računalniškim oblakom in umestite tehnologijo GraphQL v arhitekturo mikrostoritev, pri čemer izpostavite prednosti in slabosti. Izdelajte lastno knjižnico, ki bo omogočala uporabo GraphQL v javanskih mikrostoritvah. Izdelajte praktični primer uporabe GraphQL in evalvirajte uporabno vrednost.

Zahvaljujem se prof. dr. Matjažu Branku Juriču za mentorstvo in usmerjanje pri pisanju diplomske naloge. Zahvaljujem se tudi Janu Meznariču in Melaniji Vezočnik za vse podane nasvete.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
API	Application Programming Interface	aplikacijski programski vmesnik
REST	Representational State Transfer	arhitekturni način prenašanja stanj
SOAP	Simple Object Access Protocol	protokol za preprost dostop do objektov
HTTP	Hypertext Transfer Protocol	protokol za prenos hiperteksta
SDL	Schema Definition Language	jezik za definiranje sheme
JSON	JavaScript Object Notation	objektna notacija v JavaScriptu
URL	Uniform Resource Locator	enolični krajevnik vira
JAX-RS	Java API for RESTful Web Services	Java API za storitve REST
JAX-WS	Java API for XML Web Services	Java API za storitve SOAP
CDI	Contexts and Dependency Injection	dodajanje konteksta in vključevanje odvisnosti platforme Java
CRUD	Create, Read, Update, Delete	Ustvari, Preberi, Posodobi, Izbriši
JPA	Java Persistence API	Java API za trajno shranjevanje objektov

Povzetek

Naslov: Spletne storitve z GraphQL

Avtor: Domen Kajdič

Vsak razvijalec, ki sledi trendom, je že slišal za mikrostoritve in tehnologijo GraphQL. Čeprav se o novih tehnologijah veliko govori, se le mali delež razvijalcev odloči za njihovo uporabo. Glaven razlog pripisujejo nezrelosti tehnologij in netrivialni migraciji, ki je lahko včasih zamudnejša kot ponovni začetek razvoja. V okviru diplomske naloge smo tehnologijo GraphQL podrobneje raziskali in jo uvrstili v svet mikrostoritev. Primerjali smo jo s storitvami REST kot glavno alternativo in prikazali, v katerih primerih se nam jo splača uporabiti. Primerna je predvsem za aplikacije, ki vsebujejo poizvedovanje po kompleksnih podatkih z veliko relacijami, medtem ko so storitve REST primerne za preproste aplikacije. Kljub temu tehnologija GraphQL še vedno ni primerna za uporabo v mikrostoritah zaradi pomanjkanja programskih orodij. Zato smo za olajšanje razvoja mikrostoritev GraphQL razvili programsko rešitev KumuluzEE GraphQL. Uporabo rešitve smo prikazali na realnem primeru upravljanja fakultete in s tem prikazali njeno uporabnost v praksi.

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

Abstract

Title: Web services with GraphQL

Author: Domen Kajdič

Every trend-following developer has heard of microservices and GraphQL. Even though new technologies are often discussed, only a minor share of developers decide to use them. The main reason is often found in immaturity of technologies and in untrivial migration, which is often more time-consuming than starting from scratch. In the context of the thesis the GraphQL technology was explored in detail and was placed in the world of microservices. It was compared to REST as its main alternative and the potential use cases were displayed. GraphQL was found to be the most suitable for complex applications with highly related data, while REST was found to be the most suitable for simple applications. Nevertheless, GraphQL is still not ready to be widely used in microservices due to the lack of software support. Therefore, we developed a tool to simplify the development of GraphQL microservices and presented its usage on an example. With that, we were able to show the usefulness of our tool on a real-life project.

Keywords: GraphQL, web services, REST, microservices.

Poglavje 1

Uvod

Dandanes so spletne storitve, ki omogočajo komunikacijo čelnega in zalednega dela aplikacij, bistvene za razvoj sodobnih aplikacij. Najpogosteje uporabljeni tehnologiji za komunikacijo sta REST in SOAP, ki se nista spremenili že več let. Velika podjetja so pri razvoju spletnih aplikacij ugotovila, da lahko omenjeni tehnologiji nadomestijo z novimi in boljšimi. Tako je pod okriljem podjetja Facebook nastala tehnologija GraphQL, ki je bila prvič javno objavljena leta 2015.

Tehnologija GraphQL je bila zasnovana, da odpravlja probleme, s katerimi se danes spopada veliko aplikacij. Najbolj pereč je pošiljanje velikega števila zahtev na strežnik v primeru pridobivanja kompleksnih in strukturiranih podatkov, kar povzroči še vrsto drugih problemov: od počasnejšega nalaganja uporabniškega vmesnika, nepotrebnega vzpostavljanja povezav kot tudi do prenašanja podatkov, ki jih aplikacija sploh ne potrebuje. Ti problemi so še posebej očitni v nerazvitih predelih sveta, kjer še nimajo dostopa do sodobnih tehnologij in hitrih internetnih povezav.

Z razvojem tehnologije GraphQL so razvijalci rešili tudi problem nepotrebne kompleksnosti in težkega verzioniranja spletnih storitev. Ob izidu novih različic aplikacij je pogosto potrebno vzdrževati nekaj starejših različic, ker vsi uporabniki ne morejo takoj izvesti posodobitev in raje ostajajo na starih različicah. Do tega pride v aplikacijah, ki morajo biti stalno dose-

gljive in zahtevajo stabilno delovanje (pred posodobitvijo na nove različice je potrebno biti prepričan, da nova različica ne vsebuje varnostnih lukenj). Posledica je veliko število vstopnih točk, ki jih je potrebno vzdrževati in paziti, da ohranjajo kompatibilnost. Povečujejo se stroški strojne opreme, ker aplikacije zahtevajo večje računske moči zaradi nepotrebnega procesiranja.

Ko se razvijalec odloča za uporabo neke tehnologije, same prednosti tehnologije ponavadi niso dovolj. Pomembna sta tudi programska podpora in število orodij, ki so na voljo pri razvoju. Ker je GraphQL nova tehnologija, na tem področju za alternativnimi tehnologijami zaostaja. Posledično so slabo definirane tudi dobre uporabniške prakse. Ob raziskovanju mikrostoritev smo prišli do ugotovitve, da imata tehnologiji potencial za skupno delovanje. Tako smo se odločili, da ju bomo podrobneje preučili. Kot glavna cilja diplomske naloge smo si zadali analizo pomanjkljivosti skupnega delovanja GraphQL in mikrostoritev ter razvoj programske rešitve, ki identificirane pomanjkljivosti rešuje. Ker sam razvoj rešitve ni bil dovolj za učinkovito razvijanje mikrostoritev GraphQL, smo si kot dodaten cilj zadali še izpolnjevanje konceptov arhitekture cloud-native.

Skozi diplomsko nalogo smo GraphQL podrobneje opisali in ga primerjali s storitvami REST, ki so trenutno najbolj aktualne v razvoju aplikacij. Prej omenjeni problemi so najbolj prisotni ravno v storitvah REST, kar naredi primerjavo smiselno. V drugem delu diplomske naloge smo se dotaknili arhitekture mikrostoritev, ki je ena izmed novejših smernic razvoja aplikacij. Obe tehnologiji smo povezali in pokazali, kakšne so trenutne možnosti za uporabo obeh tehnologij v tandemu. Pri tem smo identificirali morebitne pomanjkljivosti in predlagali potencialne izboljšave. Na koncu smo predstavili programsko rešitev, to je razširitev odprtokodnega ogrodja KumuluzEE, ki smo jo razvili za poenostavljen razvoj mikrostoritev GraphQL. Njeno delovanje smo prikazali na aplikaciji za upravljanje fakultete.

Poglavje 2

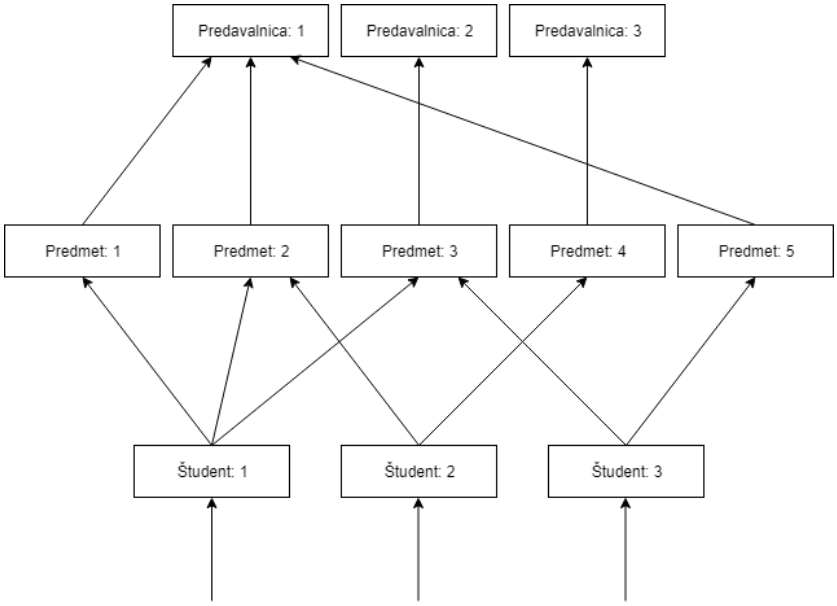
Umestitev in delovanje GraphQL

V tem poglavju bomo tehnologijo GraphQL predstavili in podrobneje opisali njeno delovanje. Glavne koncepte bomo razdelili na tri sklope: shema, poizvedovanje in izvajalno okolje. Poglavje bomo zaključili s primerjavo s storitvami REST.

2.1 Predstavitev GraphQL

GraphQL je poizvedovalni jezik za lažje pridobivanje podatkov iz aplikacijskih strežnikov. Njegova glavna prednost je v strukturiranosti; vse potrebne podatke pridobimo na enem mestu s samo eno zahtevo na strežnik. To omogoča shema, ki vsebuje definicije podatkovnih tipov, relacije med različnimi entitetami in operacije, ki so dostopne končnemu uporabniku. Z definicijo sheme dobimo graf, pri čemer entitete postanejo vozlišča, relacije povezave in operacije robna vozlišča z zunanjo povezavo. Iz te ugotovitve izvira ime tehnologije GraphQL – poizvedovalni jezik za grafe (*graph query language*). GraphQL lahko uporabljamo preko poljubnega transportnega protokola, saj je od njega neodvisen [36]. Izbira je prepuščena razvijalcu (v veliki večini je uporabljen protokol HTTP).

Za lažje razumevanje kot primer vzemimo poenostavljen model fakultete, ki vsebuje tri entitete: *Študent* (enolični identifikator, ime, priimek in polje predmetov), *Predmet* (enolični identifikator, ime predmeta, predavalnica) in *Predavalnica* (enolični identifikator, lokacija predavalnice). V oklepajih so navedene lastnosti, ki jih hranimo o vsaki entiteti.



Slika 2.1: Primer strukturiranih podatkov

Na sliki 2.1 imamo tri študente. Vsak izmed študentov obiskuje enega ali več predmetov. Vsak predmet se predava v določeni predavalnici. Identificirane povezave pretvorimo v GraphQL notacijo: imamo eno operacijo, kjer lahko pridobimo podatke o točno določenemu študentu (študent 1, 2 ali 3). Fakulteta o študentih poleg osebnih podatkov (polja brez relacij smo na sliki izpustili) hrani tudi njegove predmete. Predmeta nismo označili kot operacijo, zato do točno določenega predmeta ne moramo dostopati. Vemo pa, da je vsak predmet predavan v točno eni predavalnici, kar smo nakazali s povezavo predmeta s predavalnico. Za zgoraj omenjene podatke bi lahko naredili sledeče poizvedbe (zaenkrat bomo poizvedbe opisali z naravnim jezikom in ne GraphQL notacijo):

- Vrni ime in priimek študenta z enoličnim identifikatorjem 1.

- Vrni vse predmete, ki jih obiskuje študent z enoličnim identifikatorjem 2.

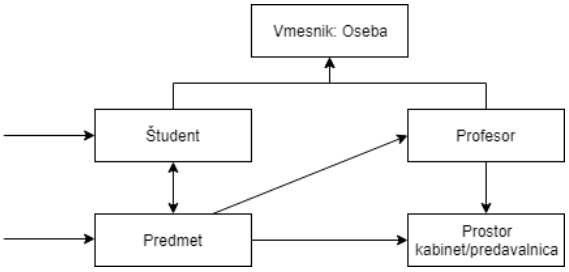
- Vrni ime, priimek in vse predmete, ki jih obiskuje študent z enoličnim identifikatorjem 3; vrni mi tudi lokacijo predavalnice vsakega posameznega predmeta.

Iz zgornjih poizvedb vidimo, da je poizvedovanje zelo preprosto. Z eno poizvedbo smo pridobili vse podatke o študentu. Opazimo lahko tudi izbirnost podatkov: v prvi poizvedbi smo potrebovali samo ime in priimek, medtem ko smo v tretji poizvedbi pridobili njegove predmete kot tudi lokacije predavalnic predmetov. Poizvedbo GraphQL lahko pojasnimo kot sprehod po grafu [34]. Premikamo se lahko samo v smeri puščic.

Možne so tudi dvosmerne povezave, čeprav jih v našem primeru nismo pokazali. V tem primeru bi na grafu ustvarili cikel. Primer takšne povezave je, da bi pri vsakem predmetu zraven hranili tudi študente, ki ga obiskujejo. Takšne relacije so lahko včasih nepredvidljive; več o tem bomo pojasnili v kasnejšem sklopu.

2.2 Delovanje GraphQL

Da bi lahko razumeli delovanje GraphQL, moramo najprej razložiti osnovne koncepte in sintakso jezika. Grobo lahko GraphQL razdelimo na tri sklope: definiranje sheme, poizvedovanje in izvajanje izvajalnega okolja. Za razlago bomo uporabili dopolnjeni model iz slike 2.1. Entiteti *Študent* in *Profesor* bosta razširili vmesnik *Oseba*. Entiteti *Predmet* bomo dodali dvosmerno povezavo (entiteta *Predmet* bo vsebovala vse študente, ki ta predmet obiskujejo) in ga uporabili kot drugo operacijo. Celotna dopolnjena shema se nahaja na sliki 2.2.



Slika 2.2: Dopolnjeni model

2.2.1 Definiranje sheme

GraphQL za definiranje sheme uporablja svoj jezik, ki se imenuje GraphQL SDL (*schema definition language* oz. jezik za definiranje sheme). Uporaba jezika je najpreprostejši način za definicijo sheme; večinoma implementacij podpira tudi programsko definiranje sheme, vendar tak način definiranja sheme zahteva poglobitev v točno določeno implementacijo. Naš cilj je splošna razlaga, zato bomo ostali pri definiranju s pomočjo jezika.

Pred začetkom razlage sintakse jezika je smiselno, da predstavimo vgrajene podatkovne tipe. GraphQL podpira pet osnovnih skalarnih tipov: celo število (*Int*), decimalno število (*Float*), niz znakov (*String*), logični izraz (*Boolean*) in enolični identifikator (*ID*, obravnavan kot *String*). Poljubno lahko dodamo tudi lastne skalarne tipe, kar pa je odvisno od implementacije. Tipičen primer takšnega tipa je datum. GraphQL podpira še sezname (označimo z [ ]), vmesnike (*Interface*), unije (*Union*) in naštevni tip (*Enum*; tip, ki ima omejeno zalogo vrednosti) [35].

Pomembno je tudi, da omenimo dve posebnosti, na kateri moramo biti pozorni. Prej omenjene skalarne tipe lahko označimo kot neprazne. S tem izvajalnemu okolju povemo, da mora poizvedovano polje vedno vsebovati vrednost (ali bo izvajalno okolje vrnilo napako). To naredimo s klicajem poleg tipa (primer: *String!*). Brez te notacije je lahko vrnjena vrednost prazna.

Druga posebnost pa je ločitev bralnih in pisalnih operacij. Do zdaj smo

govorili samo o branju podatkov, vendar je pomembno tudi spreminjanje podatkov. Primer takšne operacije bi bilo dodajanje novega študenta. Čeprav so pisalne operacije manj uporabljene kot bralne, imajo enako velik pomen, saj bi bile brez njih aplikacije statične. Pisalne operacije se imenujejo mutacije (*mutation*). Za uporabo mutacij je potrebno definirati vhodne tipe (*input type*). Poleg bralnih in pisalnih operacij obstajajo še naročnine (*subscription*), vendar se naročninam zaradi redke uporabe v diplomski nalogi ne bomo posvetili.

Za demonstracijo zgoraj opisanega bomo naš dopolnjen primer fakultete pretvorili v shemo GraphQL (izsek kode 2.1).

```
enum TipProstora {
    KABINET
    PREDAVALNICA
}

type Prostor {
    id: ID!
    lokacija: String
    tip: TipProstora!
}

interface Oseba {
    id: ID!
    ime: String!
    priimek: String!
}

type Student implements Oseba {
    id: ID!
    ime: String!
    priimek: String!
```

```
    predmeti: [Predmet]
}

type Profesor implements Oseba {
    id: ID!
    ime: String!
    priimek: String!
    kabinet: Prostor
}

type Predmet {
    studenti: [Student]
    naziv: String!
    predavalnica: Prostor
    predavatelj: Profesor
}
```

Izsek 2.1: Definicija sheme GraphQL

Da bi lahko dokončali shemo, bomo dodali še en vhodni tip, ki nam bo omogočil dodajanje študentov (izsek kode 2.2). Vhodne tipe označimo z besedo *input*.

```
input StudentInput {
    ime: String!
    priimek: String!
}
```

Izsek 2.2: Vhodni tip

Za dokončanje sheme manjka le še definicija bralnih in pisalnih operacij. Tudi te definiramo kot tipe, vendar moramo uporabiti rezervirani imeni *Query* in *Mutation* (izsek kode 2.3). Na tem mestu uporabimo prej definirani vhodni tip, ki ga zapišemo v oklepaje operacije. Za vhodne tipe lahko

uporabimo tudi osnovne skalarne tipe, ki so po definiciji vhodni in izhodni (za pridobivanje študenta moramo podati njegov enolični identifikator).

```
type Query {
    vrniStudenta(id: ID!): Student
    vrniPredmet(id: ID!): Predmet
}

type Mutation {
    dodajStudenta(student: StudentInput): Student
    izbrisiStudenta(id: ID!): boolean
}
```

Izsek 2.3: Definicija vstopnih točk

2.2.2 Poizvedovanje

Ključen del jezika GraphQL je poizvedovanje. Osnove poizvedovanja smo nakazali že v samem opisu jezika, v tem delu pa bomo predstavili sintakso in možnosti, ki so nam pri poizvedovanju na voljo.

Sintaksa

Sintaksa poizvedovanja, izpeljana iz formata JSON [15], ima naslednjo obliko: tip poizvedbe, ime poizvedbe, zaviti oklepaji, ime operacije, parametri operacije, zaviti oklepaji in seznam polj, ki jih želimo pridobiti. Tip poizvedbe mora biti ena izmed treh vrednosti: *query* (bralna poizvedba), *mutation* (pisalna poizvedba) ali *subscription* (naročnina). Pri bralnih poizvedbah je tip opcijski, vendar le v primeru pošiljanja ene poizvedbe. V primeru pošiljanja več poizvedb moramo vse poizvedbe označiti in poimenovati. Tudi ime poizvedbe je v veliki večini primerov opcijsko (v primerih, ko je poimenovanje obvezno, bomo to posebej poudarili). Če operacija vsebuje parametre, te podamo v oklepaju za imenom operacije. Paziti moramo tudi na zavite okle-

paje, ki jih moramo dodati pri vsaki zahtevi po neskalarnem podatkovnem tipu.

Bralne operacije

Pri bralnih operacijah lahko uporabljamo le operacije, ki so v shemi definirane na tipu *Query*. Primer poizvedbe, ki od študenta z enoličnim identifikatorjem 123 pridobi ime in priimek, je prikazan na izseku kode 2.4.

```
query {
    vrniStudenta(id: "123") {
        ime
        priimek
    }
}
```

Izsek 2.4: Primer bralne operacije

Ena izmed prednosti je, da nam GraphQL podatke vrne v enaki obliki kot je strukturirana zahteva. Primer odgovora na zgornjo zahtevo (izsek kode 2.4) prikazuje izsek kode 2.5.

```
{
    "data": {
        "vrniStudenta": {
            "ime": "Janez",
            "priimek": "Novak"
        }
    }
}
```

Izsek 2.5: Primer rezultata bralne operacije

V eni sami poizvedbi lahko definiramo več operacij. V primeru večkratne uporabe določene operacije je potrebno vsaj eno operacijo preimenovati [5], da zagotovimo enoličnost v vrnjenem objektu (izsek kode 2.6).

```
query {
    student123: vrniStudenta(id: "123") {
        ime
    }
    vrniStudenta(id: "124") {
        ime
    }
}

{
    "data": {
        "student123": {
            "ime": "Janez"
        },
        "vrniStudenta": {
            "ime": "Marko"
        }
    }
}
```

Izsek 2.6: Primer poizvedbe z več operacijami

Zaenkrat smo pokazali poizvedbe, ki vsebujejo le eno entiteto. Pravo moč poizvedovanja GraphQL opazimo šele pri poizvedbah z relacijami. Kot smo že omenili, vsak premik na grafu označimo z zavitimi oklepaji. Primer poizvedbe, ki od študenta z enoličnim identifikatorjem 123 pridobi vse predmete, predavalnice, kjer so predmeti predavani, in ime predavatelja predmeta, je prikazan na izseku kode 2.7.

```
query {
    vrniStudenta(id: "123") {
        predmeti {
            naziv
            predvalnica {
                lokacija
            }
            predavatelj {
                ime
            }
        }
    }
}
```

Izsek 2.7: Primer poizvedbe z relacijskimi podatki

Pri poizvedbah moramo biti pozorni, da poizvedujemo le po poljih, ki so skalarnega tipa. Če pogledamo zgornji primer (izsek kode 2.7): polje *predmeti* ni skalarnega tipa, zato moramo vprašati po dodatnem polju (npr. *naziv*). Če bi poslali poizvedbo, ki zahteva le polje *predmeti* (izsek kode 2.8), bi dobili napako.

```
# poizvedba ni veljavna
# polje predmeti ni skalarnega tipa
{
    vrniStudenta(id: "123") {
        ime
        predmeti
    }
}
```

Izsek 2.8: Primer poizvedbe po neskalarnem tipu

Pisalne operacije

Pisalne operacije imajo skoraj enako sintakso kot bralne. Edina razlika je v uporabi rezervirane besede *mutation*. Paziti moramo, da uporabljamo operacije, ki so v shemi definirane na tipu *Mutation*. Primer poizvedbe s pisalno operacijo je prikazan na izseku kode 2.9.

```
mutation {
    dodajStudenta(student: {
        ime: "Janez",
        priimek: "Novak"
    }) {
        ime
    }
}
```

Izsek 2.9: Poizvedba s pisalno operacijo

Zgoraj omenjeno pravilo (poizvedovanje po skalarnih tipih) se nanaša tudi na pisalne operacije. Operacija *dodajStudenta* vrača tip *Student*. Posledično moramo tudi v tem primeru zahtevati neko polje (npr. ime). V primeru, da nimamo za vrniti nobenega kompleksnega tipa, lahko vrnemo tudi skalarni tip. To smo demonstrirali z operacijo *izbrisiStudenta* (izsek kode 2.10), ki vrne *true* ob uspešnem izbrisu ali *false* ob neuspešnem.

```
mutation {
    izbrisiStudenta(id: "123")
}
{
    "data": {
        "izbrisiStudenta": true
    }
}
```

Izsek 2.10: Poizvedba in rezultat operacije *izbrisiStudenta*

V nekaterih alternativah spletnih storitev ob izbrisu podatkov ne vračamo ničesar. Tega v GraphQL ni mogoče narediti, saj je vračanje tipa obvezno za vse operacije.

Spremenljivke

V poizvedbe, ki jih kličemo večkrat z različnimi vhodnimi podatki, lahko vpeljemo koncept spremenljivk. Spremenljivke klicatelju poenostavijo spreminjanje vhodnih podatkov in optimizirajo delovanje izvajalnega okolja (več o tem v naslednjem sklopu). Za uporabo spremenljivk moramo poizvedbo poimenovati. Poleg imena moramo definirati vse spremenljivke, ki jih bomo uporabili in za vsako navesti tip. Spremenljivke definiramo z dolarskim znakom ($). Vrednost spremenljivk izvajalnemu okolju pošljemo ločeno od poizvedbe in je v formatu JSON. S tem dosežemo poljubno spreminjanje vhodnih parametrov ob ohranjanju enake poizvedbe (izsek kode 2.11).

```
mutation dodaj($st: StudentInput) {
    dodajStudenta(student: $st) {
        id
    }
}

# spremenljivke v prvi in drugi poizvedbi
"st": {
    "ime": "Janez",
    "priimek": "Novak"
}
"st": {
    "ime": "Marko",
    "priimek": "Novak"
}
```

Izsek 2.11: Uporaba spremenljivk

Dokumentno poizvedovanje

GraphQL podpira dokumentno poizvedovanje. To pomeni, da izvajalnemu okolju ne pošljemo le ene poizvedbe, ampak več ločenih. Če uporabljamo ta pristop, moramo vse napisane poizvedbe poimenovati in posebej poudariti, katero poizvedbo želimo izvesti. S tem si olajšamo delo na strani odjemalca, vendar poleg tega ne pridobimo veliko, ampak kvečjemu še izgubimo (nepotrebno pošiljanje odvečnih podatkov preko omrežja). Eden izmed razvijalcev GraphQL je povedal, da so se za to odločili zaradi možnih optimizacij v prihodnosti [1], med drugim:

- Shranjevanje celotnih dokumentov na strežniku; v zahtevi pošljemo le enolični identifikator dokumenta, ime poizvedbe in parametre.
- Verižno poizvedovanje oz. uporaba izhoda neke operacije kot vhod neke druge.

Primer dokumentnega poizvedovanja je prikazan na izseku kode 2.12.

```
query q1 {
    vrniStudenta(id: "123") {
        ime
        priimek
    }
}
query q2 {
    vrniPredmet(id: "1") {
        naziv
    }
}

# izbira poizvedbe
operationName: q1
```

Izsek 2.12: Dokumentno poizvedovanje

Uporaba fragmentov

Če je naša aplikacija kompleksna in vsebuje veliko operacij, ki vračajo iste tipe, si lahko poizvedovanje olajšamo z uporabo fragmentov. Fragmenti so podobni definicijam tipov v shemi z razliko, da definirajo zahtevane podatke v poizvedbi in ne dejanskega tipa. Vsak fragment pripada določenemu tipu, ki ga opredelimo pri definiciji. Fragment uporabimo z operatorjem treh pik. Primer uporabe je poizvedba dveh različnih študentov ob zahtevanju enakih polj za oba (izsek kode 2.13).

```
fragment StudentPolja on Student {
    ime
    priimek
}

query q1 {
    student123: vrniStudenta(id: "123") {
        ...StudentPolja
    }
    student124: vrniStudenta(id: "124") {
        ...StudentPolja
    }
}
```

Izsek 2.13: Uporaba fragmentov

Fragmente lahko definiramo tudi direktno ob uporabi. To je smiselno le pri poizvedovanju vmesnikov ali unij, ker takrat ne vemo, kakšne tipe bomo dobili v rezultatu. Če bi našim poizvedbam dodali poizvedbo, ki bi vrnila vse ljudi na fakulteti (izsek kode 2.14), bi kot rezultat dobili seznam vseh študentov in profesorjev. Obe entiteti imata poleg skupnih polj definirana tudi druga polja, do katerih brez fragmentov sploh ne bi morali dostopati. Fragmenti nam omogočajo pogojno vključitev teh polj.

```
{
    vrniFakultetnoOsebje {
        ime
        ...on Student {
            predmeti {
                naziv
            }
        }
        ...on Profesor {
            kabinet {
                lokacija
            }
        }
    }
}
```

Izsek 2.14: Uporaba fragmentov na primeru vmesnika

Direktive

Direktive primarno omogočajo spreminjanje rezultata poizvedbe brez spremembe poizvedbe. Da je implementacija GraphQL skladna s specifikacijo [18], mora vsebovati direktive *skip*, *include* in *deprecated*. Direktivi *skip* in *include* vsebujeta pogoj, ki je logičnega tipa. Polje, ki je označeno z direktivo *skip* in ima izpolnjen logični pogoj, bo iz rezultata izpuščeno. Delovanje direktive *include* je komplementarno delovanju direktive *skip*. Polje, ki je označeno z direktivo *include* in ima izpoljnjen logični pogoj, bo v rezultatu prikazano. Z direktivo *deprecated* označujemo polja, ki bodo v prihodnosti izbrisana. Edini pogoj je razlog za izbris, ki ga podamo kot niz znakov. Uporaba direktiv je logična le s podanimi spremenljivkami, ker fiksno podane vrednosti ne bi imele nobenega pomena. Direktive označimo z afno (@). Primer uporabe direktiv je prikazan na izseku kode 2.15.

```
query q1($vrniPriimek: Boolean!,
         $preskociIme: Boolean!){
    vrniStudenta(id: "123") {
        ime @skip(if: $preskociIme)
        priimek @include(if: $vrniPriimek)
    }
}
```

Izsek 2.15: Uporaba direktiv

V primeru, da potrebujemo kakšno dodatno direktivo, jo lahko implementiramo sami. Direktivo definiramo v shemi tako, da ji podamo ime, parametre ter pojavna mesta (najpogostejše je FIELD_DEFINITION, ki označuje polje). Napisati moramo tudi implementacijo funkcionalnosti, ki bo izvajalnemu okolju povedala, kako obravnava našo direktivo. Primer uporabniško definirane direktive je direktiva iz dokumentacije javanske implementacije GraphQL [30], ki skriva določena polja glede na uporabnikove pravice (izsek kode 2.16).

```
directive @auth(role : String!) on FIELD_DEFINITION

type Employee
    id : ID
    name : String!
    startDate : String!
    salary : Float @auth(role : "manager")
}
```

Izsek 2.16: Implementacija direktive v shemi

Opazimo, da je polju *salary* dodana direktiva *auth*, ki zgolj menedžerjem omogoča vpogled v plačo zaposlenega.

2.2.3 Izvajalno okolje

Predpogoj za uporabo GraphQL je izvajanje izvajalnega okolja. Izvajalno okolje podano shemo pretvori v izvršljivo shemo in izvaja poizvedbe. Samo okolje ne določa transportnega protokola, po katerem se pošiljajo poizvedbe; programer za to poskrbi sam. Večinoma se uporablja protokol HTTP, zaradi česar tudi večina implementacij vsebuje dodatne knjižnice za poenostavljeno vzpostavitev strežnika HTTP.

Na začetku je bil GraphQL na voljo le v programskem jeziku JavaScript (podjetje Facebook je skupaj s specifikacijo izdalo tudi referenčno implementacijo [19]), medtem ko je danes na voljo že v večini aktualnejših programskih jezikov (Java, Javascript, Go, Ruby, C#, PHP ...). Celoten seznam je na voljo na uradni spletni strani [16].

Pošiljanje poizvedb

V prejšnjem podpoglavju smo govorili o poizvedovanju, vendar nismo nikoli omenili, kako se poizvedba pošlje izvajalnemu okolju. Omenili smo tri različne parametre, ki jih lahko pošljemo: ime poizvedbe (*operationName*), telo poizvedbe (*query*) in spremenljivke (*variables*). Ime poizvedbe lahko izpustimo v primeru, da pošiljamo samo eno poizvedbo. Spremenljivke lahko izpustimo, če jih ne uporabljamo. Edino obvezno polje je telo poizvedbe. V primeru uporabe protokola HTTP sta strukturiranost podatkov in način pošiljanja poizvedbe preprosta: na strežnik GraphQL pošljemo eno izmed naslednjih zahtev:

- Zahteva GET: parametre pošljemo v naslovu URL. Primer prikazuje izsek kode 2.17.

```
Spletni naslov: http://localhost:3000/graphql?
operationName=q1&query=
query q1{vrniStudenta(id: "123"){ime}}
```

Izsek 2.17: Primer zahteve GET

- Zahteva POST: parametre pošljemo kot objekt JSON v telesu poizvedbe. V zaglavje dodamo polje: „Content-Type: application/json“. Primer prikazuje izsek kode 2.18.

```
Spletni naslov: http://localhost:3000/graphql
Zaglavje: Content-Type: application/json
Telo:
{
    "query": "query q1{vrniStudenta(id:'123'){ime}}",
    "operationName": "q1",
    "variables": {}
}
```

Izsek 2.18: Primer zahteve POST

- Zahteva POST: v telesu pošljemo samo poizvedbo (brez imena operacije in spremenljivk; to ni vedno mogoče). V zaglavje dodamo polje: „Content-Type: application/graphql“. Primer prikazuje izsek kode 2.19.

```
Spletni naslov: http://localhost:3000/graphql
Zaglavje: Content-Type: application/graphql
Telo:
query q1 {
    vrniStudenta(id: "123") {
        ime
    }
}
```

Izsek 2.19: Primer zahteve POST

Zgoraj omenjeni načini so načini ročnega pošiljanja poizvedb. Poizvedb ponavadi ne pošiljamo ročno, ampak uporabimo enega izmed odjemalcev

GraphQL na čelnem delu. Odjemalci velik del logike pošiljanja zahtev abstrahirajo. Omogočajo tudi dodatne funkcionalnosti, med drugim predpomnjenje rezultatov poizvedb.

Obdelava poizvedbe

Kot programer je potrebno poleg sheme napisati tudi logiko razreševanja poizvedb. GraphQL deluje na podlagi t. i. razreševalskih funkcij (*resolvers*) – funkcij, ki pridobivajo podatke glede na poslane poizvedbe in zahtevana polja. Vsaka operacija vsebuje svojo razreševalsko funkcijo. Kot primer vzemimo operacijo *vrniStudenta*. Za takšno operacijo moramo implementirati razreševalsko funkcijo, ki kot parameter sprejme študentov enolični identifikator in študenta vrne. Razreševalske funkcije morajo imeti tudi vsa polja (kot so ime, priimek), vendar teh ponavadi ni potrebno implementirati.

Pomemben aspekt je tudi vrstni red izvajanja razreševalskih funkcij, ki določa hitrost obdelava poizvedbe. Za poglobljeno razumevanje bomo ilustrirali glavne korake, ki jih izvajalno okolje naredi, ko prejme poizvedbo. Uporabili bomo poizvedbo iz izseka kode 2.20.

```
{
    vrniStudenta(id: "123") {
        ime
        priimek
        predmeti {
            naziv
        }
    }
}
```

Izsek 2.20: Primer poizvedbe

Poizvedbo najprej pošljemo izvajalnemu okolju (npr. preko protokola HTTP). Ko izvajalno okolje poizvedbo prejme, naredi tri stvari:

- Preveri sintaktično in semantično pravilnosti poizvedbe.

- Pokliče razreševalske funkcije, ki pridobijo zahtevane podatke.
- Podatke sestavi v izhodni format in jih posreduje naprej.

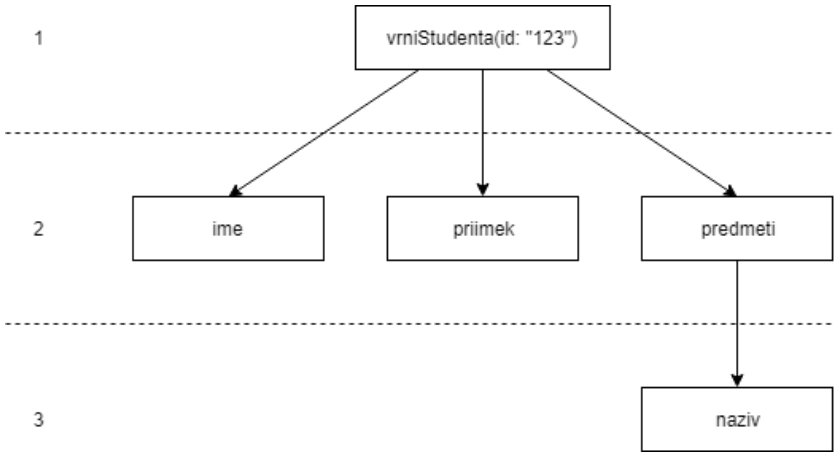
Preverjanje se zgodi pred izvajanjem poizvedbe. Tako ugotovimo morebitne napake še preden začnemo izvajati poizvedbo. GraphQL v primeru napake ne vrne objekta *data* ampak polje *errors*, ki vsebuje opise vseh napak, do katerih je prišlo. Pri vsaki napaki je zraven tudi mesto, na katerem je bila napaka zaznana. Primer napake ob napačnem imenu funkcije je viden na izseku kode 2.21.

```
# napaka je v imenu funkcije vrniStudenta
{
    "errors": [
        {
            "message": "Cannot query field
            \"vrniStudentaa\" on type \"Query\".",
            "locations": [
                {
                  "line": 2,
                  "column": 3
                }
            ]
        }
    ]
}
```

Izsek 2.21: Prikaz napake ob neuspešni poizvedbi

Po preverjanju je na vrsti pridobivanje podatkov s pomočjo razreševalskih funkcij. Te so ponavadi asinhrone, saj so podatki najpogosteje shranjeni v podatkovni bazi ali na kakšnem drugem oddaljenem viru. Vrstni red pridobivanja podatkov je takšen, kot če bi se ročno premikali po grafu (slika 2.2). V prvi iteraciji je poklicana razreševalska funkcija za operacijo *vrniStudenta*. Funkcija vrne študenta, vendar smo mi zahtevali le polja *ime*, *priimek*

in *predmeti*. Zato se v naslednji iteraciji paralelno pokličejo razreševalske funkcije za vsa tri polja. Izvajalno okolje nato počaka, da se vse tri funkcije izvedejo do konca. V zadnji iteraciji izvajalno okolje paralelno pokliče razreševalsko funkcijo za polje *naziv* za vsak predmet, ki je bil vrnjen v predhodnem koraku. Če bi iz zgornjih operacij narisali graf, bi dobili drevo [3], ki ga prikazuje slika 2.3. Opazimo, kako preprosto je pridobivanje podatkov iz različnih virov. Študente imamo lahko shranjene v eni bazi, medtem ko imamo predmete lahko shranjene nekje drugje. Tega končen uporabnik sploh ne bi opazil.



Slika 2.3: Drevesni prikaz poizvedbe

Ko so vse poizvedbe izvedene, je potrebno le še skonstruirati rezultat in ga vrniti uporabniku. Če v kakšni razreševalski funkciji pride do napake, izvajalno okolje vrne delni rezultat poizvedbe. Če bi se to zgodilo v razreševalski funkciji za predmete, bi kot rezultat poizvedbe dobili objekt *data*, ki bi vseboval ime in priimek študenta. Zraven bi dobili še polje *errors*, ki bi vsebovalo vrnjeno napako izvajalnega okolja pri izvajanju funkcije.

Ciklične poizvedbe

Ciklične poizvedbe so poizvedbe, pri katerih lahko z zahtevami po gnezdenem tipu pridemo nazaj do prvotnega tipa [5]. Hitro se lahko izrodijo v neskončne poizvedbe in s tem obremenijo izvajalno okolje. Primer ciklične poizvedbe prikazuje izsek kode 2.22.

```
{
    vrniStudenta(id: "123") {
        predmeti {
            studenti {
                predmeti {
                  # lahko nadaljujemo v neskoncnost
                }
            }
        }
    }
}
```

Izsek 2.22: Primer ciklične poizvedbe

Do tega v realnih situacijah ne bo prišlo, saj takšna gnezdenost v aplikacijah ponavadi nima smisla. Lahko pa to izkoristijo napadalci. Poizvedba z zelo veliko globino (število gnezdenj v poizvedbi) bi lahko sesula strežnik. Temu se izognemo z izbrisom dvosmerne povezave ali z omejitvijo globine poizvedbe.

Optimizacija spremenljivk

V primeru pošiljanja celotnih dokumentov GraphQL lahko izvajalno okolje nastavimo, da vse poizvedbe validira in jih shrani v pomnilnik [29]. Pri tem pristopu moramo obvezno uporabiti spremenljivke, saj le tako dobimo enake poizvedbe pri spreminjanju vhodnih parametrov. Ob naslednjem prejetju poizvedbe bo izvajalno okolje preverilo, če je poizvedba na seznamu validiranih. V tem primeru se preskoči celotno preverjanje (prvi korak izvajanja poizvedbe) in se takoj začne z izvajanjem razreševalskih funkcij.

2.3 Primerjava z arhitekturo REST

V uvodu smo se dotaknili problemov, ki jih GraphQL rešuje v primerjavi z alternativami. Rešitve teh problemov so ravno glavne prednosti GraphQL:

- Z uporabo GraphQL zmanjšamo število zahtev iz potencialno velikega števila zahtev na eno samo. V arhitekturi REST najdemo vsako entiteto na svojem naslovu URL. Posledično moramo za dostop do visoko strukturiranih podatkov poslati zahtevo za vsako entiteto posebej. Pri tem se vzpostavlja veliko število povezav, kar upočasni odzivnost aplikacije.

- Pri uporabi arhitekture REST moramo včasih hraniti več vzporednih različic API-jev zaradi dodajanja novih funkcionalnosti in ohranjanja podpore za stare različice. Zaradi tega se nam povečajo stroški vzdrževanja strežnikov kot tudi kompleksnost aplikacije. GraphQL ta problem rešuje z definicijo sheme. Tudi če se logika za pridobivanje polj spreminja, tega končen uporabnik ne opazi. Tudi dodajanje polj je preprosto in ne uniči kompatibilnosti, ker uporabnik z izbiro polj sam določi, kakšen bo njegov vrnjen objekt. Če odstranjujemo polja, jih lahko označimo z direktivo *deprecated* in kasneje odstranimo, ko so uporabniki prilagodili svoje poizvedbe.

- S tem, ko si uporabnik sam izbira podatke, ki jih potrebuje v svoji aplikaciji, se močno zmanjša količina prenesenih podatkov. Končne točke REST vračajo fiksen format podatkov oz. ponavadi kar vse podatke o določeni entiteti. Tako se v omrežju pretakajo nepotrebni podatki, ker ponavadi potrebujemo le nekaj določenih polj in ne celotne entitete.

- Ko uporabljamo arhitekturo REST, moramo sami poskrbeti za dokumentacijo. GraphQL že v osnovi podpira introspekcijo [20]. To pomeni, da lahko s poizvedovanjem pridobimo polja, ki so na strežniku na voljo za poizvedovanje.

Do zdaj smo omenjali samo prednosti, vendar kot vsaka tehnologija ima tudi GraphQL slabosti:

- GraphQL je relativno nova tehnologija. Čeprav je bila prvič javno izdana leta 2015, se v začetku ni prijela med uporabniki. Šele v zadnjem letu je začela pridobivati na priljubljenosti. Zaradi tega je na voljo veliko manj orodij kot pri delu z arhitekturo REST. Posledično tehnologijo bolj uporabljajo t. i. „early adopterji“ kot podjetja. Ta trend se je sicer v zadnjem času začel spreminjati (velika podjetja kot je npr. GitHub so začela s ponujanjem API-jev GraphQL [12]).

- REST omogoča enostavnejše predpomnjenje zaradi predoločenega formata podatkov. Pri uporabi GraphQL pa se izhod spreminja glede na uporabnikove poizvedbe, kar predpomnjenje zelo oteži.

Čeprav REST in GraphQL izgledata na zunaj zelo različni tehnologiji, obstajajo tudi določene podobnosti [4]:

- Prva podobnost je uporaba transportnega protokola HTTP in izhodnega formata JSON. Pri obeh uporabljamo zahteve GET in POST, medtem ko pri arhitekturi REST uporabljamo še nekaj dodatnih (PUT, PATCH, DELETE ...). Čeprav je možna uporaba drugačnih pristopov, je kombinacija protokola HTTP in formata JSON daleč najpopularnejša.

- Definiranje operacij v GraphQL je podobno definiranju končnih točk v arhitekturi REST. Oboje končnemu uporabniku pove, kako in kje naj pridobi podatke.

- Razreševalske funkcije v GraphQL lahko primerjamo z preusmerjanjem zahtev v arhitekturi REST. Oboje vodi v izvajanje neke funkcije, ki uporabniku vrne podatke.

Skozi primerjavo smo ugotovili, da arhitekture REST ni mogoče popolnoma zamenjati z GraphQL. Obe tehnologiji imata določene prednosti, vendar moramo paziti na pravilno uporabo. Če pogledamo glavne prednosti GraphQL, opazimo, da je uporaba najbolj smiselna pri projektih, ki vsebujejo visoko strukturirane podatke. Pri takšnih projektih bomo vse potrebne podatke pridobili z eno samo poizvedbo, kar bo posledično poenostavilo delo na čelnem delu aplikacije. GraphQL je primerno uporabiti tudi na projektih, ki uporabljajo odjemalce na različnih platformah (npr. spletna stran in android aplikacija). V primeru več platform ponavadi vsaka platforma potrebuje drugačne podatke. Dodatna prednost uporabe GraphQL na mobilnih platformah je manj prenesenih podatkov. Ker so telefoni že dovolj hitri za osnovno procesiranje, je najšibkejša plat mobilnih aplikacij ravno prenos podatkov preko slabih internetnih povezav.

Če našega projekta nismo našli na prejšnjem seznamu, moramo premisliti, ali bomo z vpeljavo GraphQL več pridobili kot izgubili. Uporaba GraphQL bo v nekaterih primerih prinesla celo poslabšanje delovanja. Glavni primer tega je uporaba v napravah interneta stvari. Tam se prenaša velika količina podatkov, ki imajo vedno enako strukturo (npr. senzorski podatki). Pri poizvedovanju ne bi pridobili veliko, saj nimamo relacijskih podatkov.

Drugi primer, kjer uporaba GraphQL ni zaželena, je uporaba v okoljih, kjer imamo omejene procesorske moči in je v omrežju malo poizvedb. Z uvedbo GraphQL pride do dodatnega procesiranja in povečanja režijskih stroškov. Tipična zahteva REST dostopa do baze in vrne podatke uporabniku. Tipična zahteva GraphQL dostopa do baze, obdela podatke (za vsako polje kliče razreševalske funkcije, pretvori pridobljene podatke v željen format) in jih vrne uporabniku. V aplikacijah, kjer procesorske moči ni v obilici in je pomembna vsaka milisekunda, lahko uvedbo GraphQL smatramo kot dodatno delo na strežniku.

Poglavje 3

Arhitektura mikrostoritev

V tem poglavju bomo predstavili mikrostoritve in izzive, s katerimi se lahko soočamo pri uporabi le teh. Dotaknili se bomo arhitekture cloud-native in navedli njene prednosti pred tradicionalno arhitekturo.

3.1 Opis mikrostoritev

S pojmom mikrostoritve označujemo sodoben način razvoja aplikacij, pri čemer aplikacijo strukturiramo kot skupek več neodvisnih komponent – mikrostoritev [38]. Posamezne mikrostoritve ločimo glede na poslovna področja in funkcije, ki jih opravljajo. Zaželeno je, da so mikrostoritve čim bolj preproste. Odvisnosti med njimi morajo biti minimizirane. Posledično lahko vsaki mikrostoritvi pripišemo svojo razvijalsko ekipo, svoj razvojni cikel in jo namestimo na svoj strežnik. Možna je tudi ponovna uporaba mikrostoritev na drugih projektih (analogija gradnje aplikacije kot sestavljanje lego kock). Ločenost mikrostoritev pripomore k zmanjšanju nepotrebnega skaliranja. Skaliramo lahko le tiste mikrostoritve, ki so najbolj uporabljene oz. ki so ozko grlo sistema.

Pomemben del pri razvoju mikrostoritev je določanje načina komunikacije med mikrostoritvami. Mikrostoritve med seboj komunicirajo izključno preko jasno definiranih vmesnikov. Obstaja več načinov komunikacije (REST,

čakalne vrste, ad-hoc pristopi ...), vendar ponavadi uporabimo kar storitve REST. Ker so mikrostoritve povezane samo preko vmesnikov, je lahko vsaka mikrostoritev implementirana v svojem programskem jeziku. Pri izdajanju novih različic aplikacije moramo paziti, da prej definiranih vmesnikov komunikacije ne uničimo. Za posodobitve, ki vplivajo na več mikrostoritev, moramo sočasno izdati različice vseh prizadetih mikrostoritev.

Nasprotje mikrostoritev je t. i. monolitni razvoj, pri čemer aplikacija vsebuje celotno logiko v enem samem paketu [38]. Glavna slabost takšnega razvoja je velika odvisnost znotraj aplikacije. Najpreprostejša sprememba ima lahko velik vpliv na aplikacijo kot celoto in lahko povzroči veliko dodatnega dela za odpravljanje vseh nepravilnosti ob takšni spremembi. Otežen je tudi prehod na nove tehnologije, saj moramo biti pozorni na celotno aplikacijo.

V preostanku razdelka bomo predstavili še izzive mikrostoritev ter njihovo delovanje na aplikaciji za upravljanje fakultete.

3.1.1 Izzivi mikrostoritev

Mikrostoritve so zelo privlačen način razvoja aplikacij za nove produkte. Kot pri vsaki novi tehnologiji moramo tudi tukaj paziti, da smo seznanjeni tudi z izzivi in nevarnostmi [2].

Preden se odločimo za mikrostoritve, se moramo vprašati, ali je njihova uporaba sploh smiselna za aplikacijo, ki jo razvijamo, in za okoliščine, v katerih se nahajamo. Za pravilno implementacijo mikrostoritev potrebujemo visoko izobražen kader, ki lahko sledi novim tehnologijam. Uporaba mikrostoritev namreč privede do dodatnih kompleksnosti, kot sta poznavanje oblačnih storitev (distribuiranost) in poznavanje novih tehnologij (Docker, Kubernetes ...). Z večanjem števila mikrostoritev se veča število dinamičnih delov aplikacije, na katere moramo paziti pri implementaciji novih funkcionalnosti ali spremembi obstoječih.

Ko se odločimo za uporabo mikrostoritev, moramo aplikacijo pravilno strukturirati in razdeliti na manjše komponente, ki bodo kasneje predsta-

vljale mikrostoritve. Izogniti se moramo nepotrebnemu podvajanju kode in preveliki kompleksnosti. Posamezna mikrostoritev mora biti čimbolj neodvisna od drugih. Če neodvisnosti ni možno doseči, je boljša možnost, da mikrostoritve združimo.

Dodaten izziv je odkrivanje napak in pravilno vzdrževanje. Pri mikrostoritvah imamo večje število izvajajočih komponent, kar pomeni večje število mest, na katerih lahko sistem odpove. Ker morajo mikrostoritve komunicirati med seboj, lahko pride do napak tudi med prenosom oz. v omrežju. Pri monolitnih aplikacijah omrežna komunikacija ni potrebna, razen pri klicu zunanjih API-jev, medtem ko je pri mikrostoritvah omrežni promet potreben že za osnovno komunikacijo. Celovito gledano je vzdrževanje kompleksnejše kot pri monolitnih aplikacijah.

Pri uporabi mikrostoritev je pomembna tudi izbira tehnologije podatkovne baze. Podatkovne baze delimo na relacijske in NoSQL. Ko načrtujemo mikrostoritve, moramo pretehtati, kakšno kombinacijo tehnologij bomo uporabili. Obstaja več možnosti, ki jih izberemo na podlagi zahtev aplikacije [6]:

- Ena podatkovna baza, vsaka mikrostoritev se poveže na isto bazo (ne dosežemo neodvisnosti mikrostoritev). Ta pristop se v praksi ponavadi ne uporablja. Uporaba ene podatkovne baze je smiselna le v primeru, če vsaka mikrostoritev uporablja ločeno shemo.

- Vsaka mikrostoritev ima svojo bazo (možna uporaba različnih tehnologij podatkovnih baz).

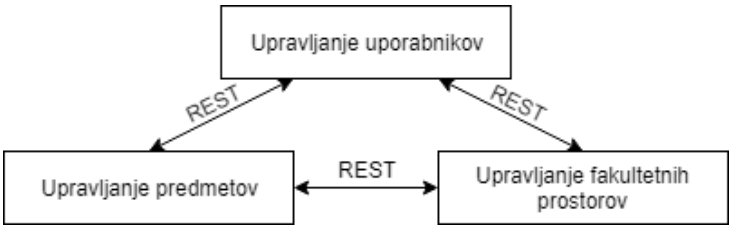
- Uporaba distribuiranih podatkovnih baz NoSQL (npr. MongoDB [28]), ki jih lahko izvajamo v več različicah.

- Kombinacija zgornjih pristopov.

3.1.2 Predstavitev mikrostoritev na aplikaciji za upravljanje fakultete

Za boljše razumevanje konceptov mikrostoritev bomo model poenostavljene fakultete (slika 2.2) uporabili v aplikaciji za upravljanje fakultete. Aplikacijo bomo uporabili za predstavitev logike mikrostoritev. Vsebovala bo entitete *Študent*, *Predmet*, *Profesor* in *Prostor*. Vsaka izmed entitet mora podpirati osnovne operacije CRUD. Pri klasičnem razvoju bi uporabili nek programski jezik za zaledni del aplikacije (npr. Java) in neko podatkovno bazo (v našem primeru najverjetneje relacijsko podatkovno bazo, saj imamo podatke z veliko relacijami).

Pri mikrostoritvah bi postopek potekal drugače. Najprej bi razdelili obravnavano aplikacijo na smiselne mikrostoritve. Tipična delitev bi bila vsaka entiteta kot svoja mikrostoritev, vendar nam v tem primeru ta delitev ne ustreza zaradi tesne povezanosti določenih entitet. Zato bi entiteti *Študent* in *Profesor* združili v eno mikrostoritev, ki bi skrbela za upravljanje vseh uporabnikov. Ker je upravljanje uporabnikov ena izmed pogostejših funkcionalnosti aplikacij, bi lahko zaradi modularnosti mikrostoritev uporabili upravljanje uporabnikov iz kakšnega drugega projekta z mikrostoritvami. Podobno velja za entiteto *Prostor*, saj so na fakulteti že prisotni obstoječi sistemi za upravljanje prostorov. Preostane nam le še entiteta *Predmet*, ki bi jo implementirali ločeno. Končna delitev je prikazana na sliki 3.1.



Slika 3.1: Prikaz mikrostoritev

3.2 Arhitektura cloud-native

Mikrostoritve so zaživele šele z razvojem vsebnikov in širokim pojavom oblačnih storitev. Kot smo prej omenili, je mikrostoritev posamezna komponenta aplikacije, ki se izvaja ločeno in komunicira z drugimi mikrostoritvami preko predoločenih vmesnikov. Pred pojavom vsebnikov je bilo takšno izvajanje težko zaradi problemov pri ročnem zaganjanju, vzdrževanju in nameščanju mikrostoritev. Ta problem je rešil Docker [10], ki omogoča enostavno pakiranje mikrostoritev v vsebnike. Vsebnik je mogoče pognati na vsaki platformi, ki podpira Docker, pri čemer njegovo izvajalno okolje ostane enako [39]. Imamo zagotovilo, da se vedno izvaja enaka koda v enakem okolju. Vsebniki so zelo učinkoviti, saj si delijo operacijski sistem in so med seboj izolirani.

Tehnologija vsebnikov sama po sebi ni bila dovolj za učinkovito uporabo in skaliranje mikrostoritev. Zato je nastal Kubernetes [22], ki upravlja (ustavlja, ponovno zaganja) vsebnike na podlagi uporabniških zahtev, omrežnega prometa in diagnostike [39]. V primeru visokega prometa na strani lahko Kubernetes avtomatično zažene več različic nekega vsebnika, kar omogoča večje število konkurenčnih uporabnikov na strežniku [37].

Da bi bilo takšno uporabljanje vsebnikov mogoče, morajo mikrostoritve nekako pridobiti omrežni naslov vseh različic nekega vsebnika, ki se trenutno izvajajo. Za to poskrbi koncept odkrivanja storitev. Trenutno obstaja na trgu več ponudnikov (Consul, Etcd, Apache Zookeeper ...). Vsi ponudniki imajo skupno decentralizirano shrambo, ki hrani vse trenutno izvajajoče različice vsebnika.

Do zdaj smo omenjali le določene koncepte, vendar še vedno nismo definirali arhitekture cloud-native. Arhitektura cloud-native je način razvoja, ki omogoča, da lahko aplikacije učinkovito izvajamo na strežnikih ponudnikov oblačnih storitev (AWS, Azure, GCP, Bluemix ...) oz. v „oblaku“ [7]. Izvajanje aplikacij v oblaku je porazdeljeno (strežniki so razpršeni na več lokacijah po svetu) [37]. Porazdeljenost omogoča, da uporabnik vedno dostopa do storitve po najkrajši možni poti (uporabnik iz Evrope bo dostopal

do strežnikov v Evropi; uporabnik iz Amerike bo dostopal do strežnikov v Ameriki). Vendar nam sam premik aplikacije v oblak ne bo prinesel teh koristi, če bomo še vedno uporabljali monolitno zgrajene aplikacije. Monolitne aplikacije lahko skaliramo le vertikalno (povečanje procesorske moči, dodatni RAM ...), medtem ko mikrostoritve skaliramo horizontalno (več paralelno izvajajočih različic iste mikrostoritve s samodejnim preverjanem zdravja in z nadzorom delovanja).

V naslednjem sklopu bomo navedli prednosti arhitekture cloud-native pred tradicionalno arhitekturo in s tem še dodatno ilustrirali njen doprinos v modernih aplikacijah.

3.2.1 Prednosti arhitekture cloud-native pred tradicionalno arhitekturo

Arhitektura cloud-native ima pred tradicionalno arhitekturo kar nekaj prednosti [9] [38]:

- **Odpornost na napake**: aplikacije cloud-native so zasnovane za maksimalno odpornost proti izpadom. K temu pripomore zapakiranost v vsebnike in avtomatizacija procesov.
- **Abstrakcija operacijskega sistema**: razvijalci se lahko posvetijo razvoju in ne konfiguraciji okolja, saj vsebniško okolje omogoča replikacijo okolja na večini operacijskih sistemov. Tako je objava aplikacij na ponudnike oblačnih storitev preprosta (kot tudi menjava ponudnika).
- **Optimizirana poraba sistemskih virov**: cloud-native platforme omogočajo dinamično dodeljevanje in odvzemanje virov. Posledično se izognemo nepotrebnim stroškom zaradi odvečnih sistemskih virov. K temu pripomore tudi možnost horizontalnega skaliranja.
- **Izboljšani DevOps procesi**: omogočajo tesnejše sodelovanje med razvijalci in nadzorniki operacij. Rezultat je hitrejša objava nove različice v produkcijo.

- **Hitrejši razvoj**: zaradi krajšega časa od kode do objave v produkcijo lahko ekipe objavljajo nove različice takoj, ko so pripravljene. Kupci aplikacij lahko testirajo hitreje. Zmanjša se krog: nova različica – povratne informacije – popravki.

- **Neodvisnost**: pri arhitekturi cloud-native najpogosteje uporabljamo mikrostoritve. Rezultat je aplikacija, ki je sestavljena iz več mikrostoritev. Za vsako mikrostoritev lahko skrbi druga razvijalska ekipa, kar poenostavi razvojni cikel, omogoča horizontalno skaliranje in skrajša čas ponovnega zagona v primeru odpovedi.

- **Avtomatizirano skaliranje**: različna orodja (npr. Kubernetes) omogočajo avtomatizirano skaliranje, kar zmanjša možnost nedosegljivosti aplikacije zaradi človeških napak. Cloud-native skaliranje temelji na principu skaliranja sistemov in ne strežnikov (abstrakcija).

- **Hitra obnova ob sesutju**: avtomatičen nadzor vsebnikov pripomore k hitrejši ugotovitvi napak in njihovemu ponovnemu zagonu. Zaradi razdeljenosti aplikacij na mikrostoritve tudi hitreje opazimo, kje je prišlo do napak in napake tudi hitreje odpravimo.

Poglavje 4

GraphQL v arhitekturi mikrostoritev

Do sedaj smo o tehnologiji GraphQL in o mikrostoritvah govorili le na splošno. To poglavje bomo posvetili njunemu skupnemu delovanju. Osredotočili se bomo na mikrostoritve v programskem jeziku Java.

4.1 Pomanjkljivosti GraphQL v današnjih tehnologijah mikrostoritev

Čeprav sta GraphQL in arhitektura mikrostoritev zaželeni tehnologiji pri razvoju programske opreme, se razvijalci pogosto odločijo za uporabo alternativnih tehnologij. Da bi odgovorili na vprašanje *zakaj*, smo analizirali pomanjkljivosti tehnologije GraphQL v svetu mikrostoritev.

Prva večja pomanjkljivost, ki smo jo identificirali, je slaba podpora za razvoj mikrostoritev GraphQL v javanski ogrodjih. V Javi je na voljo veliko število ogrodij za razvoj mikrostoritev (Spring Boot [14], WildFly Swarm [33], KumuluzEE [23] ...), vendar le eno podpira GraphQL (GraphQL Spring Boot [32]). Pri ostalih se moramo zanesti na lastno vpeljavo GraphQL knjižnic. Lastna uporaba knjižnic pogosto privede do slabega razumevanja in podaljšanja časa razvoja. Druga stran podpore leži tudi v javanski API-

jih. Platforma Java EE ima vgrajene module za obravnavo storitev REST (JAX-RS) in SOAP (JAX-WS). To dejstvo dokazuje, da sta REST in SOAP dovolj zreli tehnologiji za standardizacijo, medtem ko je GraphQL šele na začetku poti.

Kot smo že omenili, je GraphQL relativno nova tehnologija. Pri vsaki razvijajoči tehnologiji lahko pride do velikih sprememb in nekompatibilnosti s starejšimi različicami ob izidu novih. Tem spremembah se pri razvoju resnejših aplikacij poskušamo izogniti, saj lahko vodijo do podaljšanja razvoja zaradi uvajanja prilagoditev.

Pomanjkanje standardizacije vodi tudi v fragmentacijo. Trenutno obstaja več načinov za postavitev strežnika GraphQL (Apollo GraphQL [8], GraphQL Prisma [17], ročna postavitev v različnih programskih jezikih ...). Pri vsakem načinu so razvijalci dodali nekaj svojega. Zaradi tega težje identificiramo razvojne dobre prakse, ker je na voljo preveč različnih načinov za doseganje enakega cilja.

Naslednja pomanjkljivost je nepoznavanje pravilne uporabe GraphQL. V arhitekturi mikrostoritev sta možni dve glavni uporabi: uporaba GraphQL za omogočanje komunikacije med mikrostoritvami in uporaba GraphQL kot združitev več mikrostoritev v eno skupno dostopno točko za končnega uporabnika. Prva možnost lahko izgleda privlačno, saj ima GraphQL veliko poizvedovalno moč. Še preden se odločimo za ta način, se moramo vprašati, če naše mikrostoritve res potrebujejo GraphQL. Ker so mikrostoritve funkcijsko ločene (vsebujejo manjše količine podatkov), je ponavadi dovolj, da za komunikacijo uporabimo REST ali sporočilni sistem (izbira je odvisna od vrste aplikacije in je lahko od primera do primera drugačna). Uvedba GraphQL znotraj mikrostoritev nas lahko stane procesorske moči in hitrosti:

- Druge mikrostoritve morajo vedeti klicati dostopno točko GraphQL: klicanje dostopnih točk GraphQL je težje in počasneje kot klicanje dostopnih točk REST.

- Za medsebojno komunikacijo bi morala imeti vsaka mikrostoritev definirano svojo shemo in postavljeno svojo dostopno točko GraphQL;

postavitev dostopnih točk GraphQL je dražje od postavitve dostopnih točk REST.

- GraphQL je namenjen zmanjšanju števila zahtev in lažjemu poizvedovanju po relacijskih podatkih; mikrostoritve so omejene s podatki, zato uvedba GraphQL ne bi prinesla večjih prednosti.

Primernejši način uporabe GraphQL v arhitekturi mikrostoritev je uporaba kot za združitev več mikrostoritev (agregacija). Končnemu uporabniku sploh ni pomembno, če so v ozadju mikrostoritve. Uporabnik vidi le eno dostopno točko GraphQL, na kateri lahko pridobi vse zahtevane podatke. Pri tem pristopu moramo definirati le eno skupno shemo. Razreševalske funkcije uporabimo kot posrednike, ki pošiljajo REST zahteve na zahtevane mikrostoritve.

4.2 Implementacija knjižnice GraphQL za javanske mikrostoritve

Da bi odpravili čim več zgornjih pomanjkljivosti, smo razvili GraphQL razširitev za odprtokodno mikrostoritveno ogrodje KumuluzEE [23]. Ogrodje KumuluzEE smo izbrali, ker ima v primerjavi z drugimi ogrodji najkrajši čas zagona in najmanjšo porabo pomnilnika [40], poleg tega pa je enostavno za uporabo zaradi zasnove ogrodja in bogate dokumentacije. Razširitev se imenuje KumuluzEE GraphQL. Podpira naslednje funkcionalnosti:

- Hitra postavitev strežnika.
- Generacija GraphQL sheme s pomočjo anotiranja javanskih tipov in funkcij.
- Poenostavljeno pisanje razreševalskih funkcij.
- Testiranje poizvedb na grafičnem orodju GraphiQL [13].

- Integracija z razširitvijo KumuluzEE za implementacijo varnosti (KumuluzEE Security [27]).
- Integracija z razširitvijo KumuluzEE za optimizacijo poizvedb JPA (KumuluzEE REST [26]).
- Integracija z razširitvijo KumuluzEE za odkrivanje storitev (KumuluzEE Discovery [24]).
- Vgrajeno razvrščanje, filtriranje in paginacija po vzoru REST praks.

Primarno je razširitev namenjena razvoju aplikacij po principu *najprej koda* (*code-first*), ki označuje razvoj aplikacij, pri katerem začnemo s pisanjem kode pred načrtovanjem sheme. Uporaba alternativnega pristopa *najprej shema* (*schema-first*) je otežena, saj bi morali sami poskrbeti za generacijo javanskih razredov. Razširitev je možno uporabiti kot agregator za več mikrostoritev ali za postavitev svojega strežnika GraphQL na vsaki mikrostoritvi posebej. Celotna razširitev je na javno objavljena na GitHubu [25].

4.3 Predstavitev razširitve KumuluzEE GraphQL

V tem podpoglavju predstavili osnovne funkcionalnosti implementirane razširitve in integracijo z drugimi razširitvami KumuluzEE (KumuluzEE Discovery [24], KumuluzEE REST [26] in KumuluzEE Security [27]).

4.3.1 Vključitev v projekt

Vključitev v projekt KumuluzEE je preprosta. Vse kar moramo narediti je dodati razširitev v seznam knjižnic. Pod sklop *dependencies* v datoteki *pom.xml* dodamo referenco na KumuluzEE GraphQL (izsek kode 4.1).

```
<dependency>
    <groupId>com.kumuluz.ee.graphql</groupId>
    <artifactId>kumuluzee-graphql</artifactId>
    <version>trenutna_razlicica</version>
</dependency>
```

Izsek 4.1: Vključitev razširitve v projekt

Če hočemo uporabiti orodje GraphiQL [13] za testiranje poizvedb, ga lahko dodatno vključimo (izsek kode 4.2).

```
<dependency>
    <groupId>com.kumuluz.ee.graphql</groupId>
    <artifactId>kumuluzee-graphql-ui</artifactId>
    <version>trenutna_razlicica</version>
</dependency>
```

Izsek 4.2: Vključitev GraphiQL v projekt

4.3.2 Osnovna uporaba

Za osnovno uporabo razširitve je potrebno dodati le nekaj anotacij. Bralno operacijo prikazuje izsek kode 4.3.

```
@GraphQLClass
public class Test {
    @GraphQLQuery
    public String testnaBralnaOperacija() {
        return "GraphQL!";
    }
}
```

Izsek 4.3: Primer bralne operacije

Anotacija *GraphQLClass* pove razširitvi, da pogleda datoteko. Anotacija *GraphQLQuery* pa razširitvi sporoči, da naj funkcijo preslika v razreševalsko funkcijo in jo doda v shemo. Zgornji primer (izsek kode 4.3) bi v shemo dodal bralno operacijo *testnaBralnaOperacija*, ki bi ob klicu vrnila besedo „GraphQL!“.

Na podoben način poteka dodajanje pisalnih operacij, edina razlika je v uporabi anotacije.

```
    @GraphQLClass
    public class Test2 {
        @GraphQLMutation
        public String testnaPisalnaOperacija(
        @GraphQLArgument(name="test") String test) {
            // procesiranje
            return test;
        }
    }
```

Izsek 4.4: Primer pisalne operacije

Zgornja pisalna operacija (izsek kode 4.4) sprejme argument *test*, ki ga kasneje vrne (vsaka operacija mora nekaj vrniti).

Uporaba poljubnih tipov

Razširitev omogoča preslikavo javanskih razredov v tipe GraphQL.

```
public class PodatkovniTip {
    private String polje1;

    public String getPolje1() {
        return polje1;
    }

    public void setPolje1(String polje1) {
        this.polje1 = polje1;
    }
}
```

Izsek 4.5: Primer podatkovnega tipa

Zgornji javanski tip (izsek kode 4.5) bi se ob uporabi v bralni operaciji preslikal v tip GraphQL in ob uporabi v pisalni operaciji v vhodni tip GraphQL. Če hočemo tipu dodati polje izven javanskega razreda, lahko to storimo z anotacijo *GraphQLContext*.

```
@GraphQLQuery
public String polje2(@GraphQLContext PodatkovniTip pt) {
    return "poljubno besedilo";
}
```

Izsek 4.6: Primer vrinjenega polja

Zgornji primer (izsek kode 4.6) bi podatkovnemu tipu v shemi dodal polje z imenom „polje2“.

Kot zadnjo anotacijo bomo predstavili anotacijo *GraphQLNonNull*. To anotacijo uporabljamo v primeru, ko hočemo narediti neko polje obvezno za vnos oz. neničelno (v shemi smo to označili s klicajem poleg tipa). Primer uporabe je prikazan na izseku kode 4.7. Anotacija se dodaja na funkcije *getter* in *setter* v podatkovnem tipu.

```
public @GraphQLNonNull String getPolje1() {
    // moramo vrniti nenicelni tekst
    return "nek tekst";
}

public void setPolje1(@GraphQLNonNull String polje1) {
    // podan argument polje1 ne sme biti nic
    this.polje1 = polje1;
}
```

Izsek 4.7: Primer neničelnega polja

4.3.3 Paginacija, razvrščanje in filtriranje

Ker imamo velikokrat opravka s seznami rezultatov, smo razširitvi dodali funkcionalnost paginacije, razvrščanja in filtriranja rezultatov. Za to poskrbijo funkcije v razredu *GraphQLUtils*. Poleg vgrajenih funkcij, smo dodali ovojne funkcije za funkcije iz razširitve KumuluzEE REST [26]. Celotno delovanje bomo pojasnili na izseku kode 4.8, ki je primer iz dokumentacije KumuluzEE GraphQL [25].

```
@GraphQLQuery
public List<Student> vrniStudente() {
    return fakultetaZrno.vrniSeznamStudentov();
}

@GraphQLQuery
public PaginationWrapper<Student> vrniStudente(
@GraphQLArgument(name="pagination") Pagination p,
@GraphQLArgument(name="sort") Sort s,
@GraphQLArgument(name="filter") Filter f) {
    return GraphQLUtils.process(
        fakultetaZrno.vrniSeznamStudentov(),
        p, s, f
    );
}
```

Izsek 4.8: Delovanje paginacije, razvrščanja in filtriranja

Izsek kode 4.8 prikazuje primerjavo med razreševalsko funkcijo z in brez dodane logike za paginacijo/razvrščanje/filtriranje. Za lažje razumevanje bomo med njima naredili primerjavo:

- Izhodni tip funkcije je bil spremenjen iz *List* v *PaginationWrapper*. Razred *PaginationWrapper* je ovojni razred, ki pri poizvedovanju doda polja zamik (*offset*), omejitev števila rezultatov (*limit*) in število zapisov (*total*).

- Dodani so bili argumenti *Pagination*, *Sort* in *Filter*. Uporaba teh argumentov omogoči vnos želenih parametrov pri poizvedovanju. Argument *Pagination* doda možnost poizvedovanja po poljih *limit* in *offset*. Z argumentom *Sort* lahko dodamo seznam polj, po katerih bomo podatke razvrščali (za vsako polje je potrebno podati smer razvrščanja: naraščajoče ali padajoče). Argument *Filter* je podoben argumentu *Sort*. Namesto smeri razvrščanja vsebuje operacijo (je enako, ni enako, večje, manjše ...) in vrednost filtriranja.

- Podane argumente je potrebno obdelati na seznamu podatkov. Za to poskrbi razred *GraphQLUtils*, ki vsebuje funkcijo *process*. Funkcija je definirana na 12 različnih načinov (opcijska vključitev argumentov; lahko uporabimo optimizirane poizvedbe JPA ali izpustimo kakšnega izmed argumentov) in vrača tip *PaginationWrapper*. V primeru, da nočemo uporabljati paginacije, so na voljo funkcije *processWithoutPagination*, ki vračajo tip *List*.

Primer poizvedbe, ki pridobi prvih deset študentov z imenom Janez in jih razvrsti po naraščajoči vpisni številki, je prikazan na izseku kode 4.9.

```
{
  allStudents(pagination: {
    offset: 0, limit: 10
  },
  sort: {fields:
    [{field: "studentNumber", order: ASC}]
  },
  filter: {fields:
    [{field: "name", op: EQ, value: "Janez"}]
  }) {
    result {
      studentNumber
      name
      surname
    }
    pagination {
      offset
      limit
      total
    }
  }
}
```

Izsek 4.9: Primer poizvedbe s paginacijo, razvrščanjem in filtriranjem

4.3.4 Konfiguriranje razširitve

Razširitev je možno konfigurirati preko vseh načinov konfiguracije ogrodja KumuluzEE (datoteka yaml, okoljske spremenljivke ...). Podpira naslednje nastavitve:

- Pot do dostopne točke GraphQL (*mapping*).
- Pot do grafičnega orodja GraphiQL (*ui.mapping*) in opcijski vklop/izklop (*ui.enabled*).
- Privzete nastavitve za paginacijo: zamik (*defaults.offset*) in število prikazanih zadetkov (*defaults.limit*).

Konfiguracijski ključi se začnejo s *kumuluzee.graphql*, celoten ključ nastavitve je unija tega ključa in ključa v oklepaju (npr. *kumuluzee.graphql.mapping*).

4.3.5 Dodajanje aplikacijskega razreda GraphQL

Aplikacijski razred je bil dodan po zgledu aplikacijskega razreda v JAX-RS. Razred ima lahko poljubno ime, vendar mora biti anotiran z anotacijo *GraphQLApplicationClass* in razširjati razred *GraphQLApplication*. Namenjen je konfiguraciji različnih nastavitev za izvajalno okolje GraphQL kot tudi za nastavitev varnosti in odkrivanja storitev. Nastavitve spremenimo s prepisom funkcije znotraj razreda. Uporaba razreda je prikazana na izseku kode 4.10.

```
@GraphQLApplicationClass
public class Razred extends GraphQLApplication {}
```

Izsek 4.10: Primer aplikacijskega razreda

Integracija s KumuluzEE Security

Zaradi nekompatibilnosti obstoječe različice razširitve KumuluzEE Security s KumuluzEE GraphQL, smo razširitev dopolnili. Dodali smo ji prepoznavanje mikrostoritev GraphQL s pomočjo anotiranja aplikacijskega razreda z anotacijo *DeclareRoles* (izsek kode 4.11).

Razširitev za shrambo uporabnikov in avtentikacijo uporablja Keycloak [21]. Podprta sta dva načina delovanja: zaščita storitev REST in zaščita zrn CDI. Podporo v KumuluzEE GraphQL smo dosegli preko zaščite zrn CDI. Za vklop zaščite je potrebno anotiranje zrna z anotacijo *Secure* (izsek kode 4.12) in anotiranje posameznih razreševalskih funkcij z javanskimi varnostnimi anotacijami:

- *RolesAllowed* omogoča dostop le določenim uporabniškim skupinam.
- *PermitAll* omogoča dostop vsem uporabnikom.
- *DenyAll* ne omogoča dostopa nobenemu uporabniku.

```
@DeclareRoles({"admin","student"})
@GraphQLApplicationClass
public class Razred extends GraphQLApplication {}
```

Izsek 4.11: Primer aplikacijskega razreda z dodano varnostjo

```
@Secure
@GraphQLClass
public class Test {
    @PermitAll
    @GraphQLQuery
    public String testnaPoizvedba() {
        return "GraphQL!";
    }
}
```

Izsek 4.12: Uporaba varnostnih anotacij

Integracija s KumuluzEE Discovery

Podobno kot pri varnosti, tudi KumuluzEE Discovery ni bil kompatibilen s KumuluzEE GraphQL, saj je bil primarno zasnovan za odkrivanje mikrostoritev REST. Ta problem smo rešili z dodajanjem tipa mikrostoritve vsem funkcionalnostim razširitve. S tem smo razširitev naredili generično in omogočili dodajanje novih tipov v prihodnosti. Primer registracije mikrostoritve GraphQL je prikazan na izseku kode 4.13. Primer odkrivanja mikrostoritev GraphQL je prikazan na izseku kode 4.14.

```
@RegisterService
@GraphQLApplicationClass
public class Razred extends GraphQLApplication {}
```

Izsek 4.13: Primer registracije mikrostoritve GraphQL

```
# avtomatsko
@Inject
@DiscoverService(value="graphql",environment="dev",
version="1.0.0",serviceType=ServiceType.GRAPHQL)
private Optional<URL> url;

# programsko
discoveryUtil.getServiceInstances("graphql", "dev",
"1.0.0", AccessType.DIRECT, ServiceType.GRAPHQL);
```

Izsek 4.14: Primer odkrivanja mikrostoritev GraphQL

Poglavje 5

Praktični primer in evalvacija

V tem poglavju bomo predstavili primer uporabe razširitve KumuluzEE GraphQL. Na koncu bomo s pomočjo implementirane aplikacije povzeli, če smo z razvojem razširitve dosegli zadane cilje.

5.1 Primer implementacije aplikacije za upravljanje fakultete

Pri predstavitvi mikrostoritev smo kot primer vzeli aplikacijo za upravljanje fakultete. To aplikacijo smo tudi implementirali s pomočjo razširitve KumuluzEE GraphQL [25]. Celotna izvorna koda je na voljo na GitHubu [31].

5.1.1 Predstavitev projekta

Projekt je razdeljen na štiri dele:

- Mikrostoritev *prostori* vsebuje logiko za dodajanje, spreminjanje in brisanje prostorov (dva tipa prostorov: kabinet in predavalnica).
- Mikrostoritev *predmeti* vsebuje logiko za dodajanje, spreminjanje in brisanje predmetov (pri dodajanju predmeta preveri v mikrostoritvi *prostori*, če podana predavalnica obstaja).

- Mikrostoritev *uporabniki* vsebuje logiko za dodajanje, spreminjanje in brisanje študentov in profesorjev (pri dodajanju profesorja preveri v mikrostoritvi *prostori*, če podan kabinet obstaja). Omogoča dodajanje in odstranjevanje študentovih predmetov (preveri v mikrostoritvi *predmeti*) ter preverjanje, kateri študenti so izbrali določen predmet.

- Mikrostoritev *agregator* vsebuje celotno podatkovno shemo, ki smo jo definirali pri predstavitvi GraphQL, z izjemo, da vsebuje vse bralne in pisne operacije, ki jih omogočajo naše mikrostoritve.

Vsi štirje deli so zapakirani v slike Docker, pri čemer je le GraphQL del odprt za zunanji promet. Vsaka izmed mikrostoritev vsebuje svojo podatkovno bazo. Za potrebe odkrivanja storitev je dodan etcd [11], ki ga uporablja razširitev KumuluzEE Discovery [24]. Celoten projekt lahko zaženemo z ukazom „docker-compose up -d“.

Zaradi preprostosti mikrostoritev (vsebujejo le operacije CRUD na podatkovni bazi) bomo njihove opise izpustili. V nadaljevanju bomo podrobneje opisali mikrostoritev *agregator*.

5.1.2 Implementacija entitet

Da bi entitete lahko uporabili v shemi, smo jih definirali kot javanske razrede. Skupno smo definirali šest razredov: *Predmet*, *Profesor*, *Prostor*, *Študent* in *Uporabnik* ter naštevni tip *TipProstora*.

TipProstora (izsek kode 5.1) vsebuje možnosti *Kabinet* in *Predavalnica*. Uporabljen je v entiteti *Prostor* (izsek kode 5.2).

```
public enum TipProstora {
    KABINET, PREDAVALNICA
}
```

Izsek 5.1: Naštevni tip TipProstora

```
public class Prostor {
    private Integer id;
    private String lokacija;
    private TipProstora tipProstora;

    @GraphQLIgnore
    public void setId(Integer id) {
        this.id = id;
    }
    // ostale funkcije getter in setter
}
```

Izsek 5.2: Entiteta Prostor

Posebej smo izpostavili funkcijo *setter setId*, ki je anotirana z anotacijo *GraphQLIgnore*. S tem smo polje *id* izbrisali iz vhodnega tipa *ProstorInput*, ki ga generira razširitev avtomatično. Tako smo označili, da polja *id* ne smemo podati ob ustvarjanju novega prostora, ampak bo polje avtomatično generirano.

Naslednja izmed entitet je entiteta *Uporabnik* (izsek kode 5.3). Entitete ne bomo direktno uporabili v operacijah, ampak le kot osnovo, ki jo bosta entiteti *Študent* in *Profesor* razširili. Posledično vsebuje le skupna polja vsakega uporabnika. Podobno kot v entiteti *Prostor* (izsek kode 5.2), smo tudi v entiteti *Uporabnik* funkcijo *setter* za polje *id* anotirali z *GraphQLIgnore*.

```
public class Uporabnik {
    private Integer id;
    private String ime;
    private String priimek;

    @GraphQLIgnore
    public void setId(Integer id) {
        this.id = id;
    }
    // ostale funkcije getter in setter
}
```

Izsek 5.3: Entiteta Uporabnik

Sledi entiteta *Študent* (izsek kode 5.4), ki vsebuje le seznam enoličnih identifikatorjev vseh študentovih predmetov. Tukaj smo z *GraphQLIgnore* anotirali funkcijo *getter* kot tudi funkcijo *setter*. Funkcijo *getter* smo anotirali, da uporabniku onemogočimo direktno poizvedovanje enoličnih identifikatorjev predmetov. Cilj uporabe GraphQL je poizvedovanje po grafu. Poizvedovanje po enoličnih identifikatorjih ni ravno smiselno, ker bi uporabnik za pridobivanje predmetov moral narediti še eno poizvedbo. Zato bomo entiteti *Študent* ročno dodali polje, ki bo vsebovalo seznam vseh študentovih predmetov. Razreševalska funkcija novo ustvarjenega polja bo uporabila enolične identifikatorje predmetov in priskrbela dejanski seznam predmetov.

Pri funkciji *setter* pa smo uporabili anotacijo z razlogom, da onemogočimo podajanje predmetov ob ustvarjanju novega študenta. Tako vhodni tip *StudentInput* v shemi ne bo vseboval polja *subjectIds*.

```
public class Student extends Uporabnik {
    private List<Integer> subjectIds;

    @GraphQLIgnore
    public List<Integer> getSubjectIds() {
        return subjectIds;
    }

    @GraphQLIgnore
    public void setSubjectIds(List<Integer> subjectIds) {
        this.subjectIds = subjectsIds;
    }
}
```

Izsek 5.4: Entiteta Študent

Druga razširjena entiteta iz entitete *Uporabnik* je entiteta *Profesor* (izsek kode 5.5). Vsebuje le eno polje (enolični identifikator kabineta). Z anotacijo *GraphQLIgnore* smo anotirali le funkcijo *getter*. S tem smo onemogočili poizvedovanje po polju *cabinetId*. Polje *cabinet* bomo zraven dodali ročno. Razreševalska funkcija polja *cabinet* bo s pomočjo enoličnega identifikatorja pridobila podatke o kabinetu.

Funkcije *setter* nismo anotirali, ker želimo, da vhodni tip *ProfesorInput* to polje vsebuje. V tem primeru je bolj smiselno, da se ob kreaciji profesorja poda enolični identifikator njegovega kabineta.

```
public class Profesor extends Uporabnik {
    private Integer cabinetId;

    @GraphQLIgnore
    public Integer getCabinetId() {
        return cabinetId;
    }

    public void setCabinetId(Integer cabinetId) {
        this.cabinetId = cabinetId;
    }
}
```

Izsek 5.5: Entiteta Profesor

Preostane le še entiteta *Predmet* (izsek kode 5.6). V entiteti *Predmet* smo z anotacijo *GraphQLIgnore* anotirali pet funkcij. Vse primere uporabe teh anotacij smo opisali že v prejšnjih primerih. Funkcijo *setter* za polje *id* smo anotirali, ker nočemo, da se pojavi v vhodnem tipu (enolični identifikator se generira avtomatično; ne sme ga podati uporabnik). Pri polju *studentIds* smo anotirali funkciji *getter* in *setter*, kar smo obrazložili na entiteti *Študent*. Funkcijo *getter* na poljih *idProstora* in *idProfesorja* smo anotirali, ker nočemo poizvedovanj po enoličnih identifikatorjih, ampak po njihovih dejanskih tipih (kar dosežemo z ročnih dodajanjem polja v shemo; smo obrazložili na entiteti *Študent*).

```
public class Predmet {
    private Integer id;
    // polje, ki omogoca ciklicno poizvedovanje
    private List<Integer> studentIds;
    private String naziv;
    private Integer idProstora;
    private Integer idProfesorja;
```

```
    @GraphQLIgnore
    public void setId(Integer id) {
        this.id = id;
    }

    @GraphQLIgnore
    public List<Integer> getStudentIds() {
        return studentIds;
    }

    @GraphQLIgnore
    public void setStudentIds(List<Integer> studentIds) {
        this.studentIds = studentIds;
    }

    @GraphQLIgnore
    public Integer getIdProstora() {
        return idProstora;
    }

    @GraphQLIgnore
    public Integer getIdProfesorja() {
        return idProfesorja;
    }

    // ostale funkcije getter in setter
}
```

Izsek 5.6: Entiteta Predmet

5.1.3 Implementacija zrn CDI

Za potrebe izvajanja klicev REST na mikrostoritve skrbijo zrna CDI. Aplikacija vsebuje tri zrna:

- Zrno za uporabnike; vsebuje vse operacije CRUD za mikrostoritev *uporabniki*. Vsebuje tudi operaciji za vpis in izpis študenta v določen predmet.
- Zrno za prostore; vsebuje vse operacije CRUD za mikrostoritev *prostori*.
- Zrno za predmete; vsebuje vse operacije CRUD za mikrostoritev *predmeti*.

Vsa tri zrna so zelo podobna; razlikujejo se le po ciljni entiteti. Zato bomo v celoti predstavili le zrno za uporabnike (funkcije za študente).

Preden začnemo z opisom funkcij, ki opravljajo klice REST, bomo razložili elemente zrna in njegovo inicializacijo na primeru implementacije zrna za entiteto *Uporabnik* (izsek kode 5.7).

```
@ApplicationScoped
public class UporabnikiZrno {
    private Client httpClient;
    private String url;

    @Inject
    DiscoveryUtil discoveryUtil;

    @PostConstruct
    private void init() {
        httpClient = ClientBuilder.newClient();
        url = discoveryUtil
            .getServiceInstance("users", "1.0.0", "dev")
            .get()
            .toString() + "/v1";
    }

    // funkcije za opravljanje REST klicev
}
```

Izsek 5.7: Implementacija zrna za entiteto Uporabnik

Najprej opazimo različne anotacije CDI. Prva izmed njih je anotacija *RequestScoped*. S to anotacijo smo povedali, da je naše zrno potrebno inicializirati ob vsaki novi zahtevi zanj. Tako dosežemo, da bo ob vsaki zahtevi za zrno ponovno klicana funkcija *init*. Anotacija *PostConstruct* označuje funkcijo, ki se bo izvedla takoj po inicializaciji zrna. Anotacija *Inject* nam vrne različico nekega drugega zrna.

Znotraj zrna smo definirali dve spremenljivki, ki nam bosta kasneje po-

magali pri izvedbi klicev REST. Prva je razreda *Client*, ki je del specifikacije JAX-RS. Omogoča nam izvajanje klicev REST. Inicializirali smo jo v prvi vrstici *init* funkcije.

Druga spremenljivka vsebuje spletno povezavo do naše mikrostoritve. Ker smo aplikacijo zasnovali s pomočjo odkrivanja storitev, naslov do mikrostoritve pridobimo šele ob izvajanju aplikacije. Uporabimo razred *DiscoveryUtil*, ki je del KumuluzEE Discovery [24]. Funkcija *getServiceInstance* nam vrne eno izmed več možnih različic replicirane mikrostoritve.

Preostali del zrna so funkcije, ki opravljajo klice REST na mikrostoritve. Celotno zrno vsebuje trinajst funkcij (pet operacij CRUD za entiteto *Študent* in dodatnih pet za entiteto *Profesor*, klic za pridobivanje vseh študentov na določenem predmetu in dva klica za dodajanje/brisanje predmeta na določenem študentu). Osnovnih pet operacij CRUD sestavljajo naslednji klici:

- Pridobi določen vnos glede na njegov enolični identifikator (GET).
- Pridobi seznam vseh vnosov (GET).
- Ustvari nov vnos (POST).
- Posodobi celoten vnos (PUT).
- Izbriši vnos (DELETE).

Primer klica storitve REST za pridobivanje študenta s podanim *id* prikazuje izsek kode 5.8.

```
public Student vrniStudenta(Integer id) {
    Response r = httpClient
        .target(url)
        .path("students")
        .path("{id}")
        .resolveTemplate("id", id)
        .request()
        .get();
    if(r.getStatusInfo().getStatusCode() ==
        Response.Status.OK.getStatusCode()) {
        return r.readEntity(Student.class);
    } else {
        Error e = r.readEntity(Error.class);
        throw new GraphQLException(e.getOpis() + " " +
            e.getLokacija());
    }
}
```

Izsek 5.8: Funkcija za pridobivanje študenta s podanim enoličnim identifikatorjem

Prvi korak pošiljanja zahteve REST je gradnja objekta *Response*. Objektu podamo pot, parametre in način klica (v zgornjem primeru GET). Preden nadaljujemo z obdelavo podatkov, preverimo, če je prišlo do morebitne napake. V primeru, da je bila zahteva na mikrostoritev uspešna, lahko vrnjeni JSON pretvorimo v javansko entiteto z ukazom *readEntity*, ki mu podamo entitetni razred. Če zahteva ni bila uspešna, je velika verjetnost da nismo dobili vrnjenega pravilnega objekta JSON, ampak informacije o napaki. Zato smo vrnjeni JSON prebrali v javanski razred *Error* (izsek kode 5.9), ki vsebuje tri informacije o napaki: kodo, opis in lokacijo napake. Da takšen način deluje, mora naša mikrostoritev vračati napake v istem formatu.

```
public class Error {
    private Integer koda;
    private String opis;
    private String lokacija;
    // funkcije getter in setter
}
```

Izsek 5.9: Razred Error

V naslednjih primerih bomo obvladovanje napak izpustili zaradi večje berljivosti kode.

Primer klica storitve REST za pridobivanje seznama študentov prikazuje izsek kode 5.10.

```
public List<Student> vrniStudente() {
    Response r = httpClient
        .target(url)
        .path("students")
        .request()
        .get();
    return r.readEntity(
        new GenericType<List<Student>>{}
    );
}
```

Izsek 5.10: Funkcija za pridobivanje seznama študentov

Primer klica storitve REST za dodajanje študentov prikazuje izsek kode 5.11.

```
public Student dodajStudenta(Student student) {
    Response r = httpClient
        .target(url)
        .path("students")
        .request()
        .post(Entity.json(student));
    return r.readEntity(Student.class);
}
```

Izsek 5.11: Funkcija za dodajanje študentov

Primer klica storitve REST za posodabljanje študentov prikazuje izsek kode 5.12.

```
public Student spremeniStudenta(Integer id, Student
    student) {
    Response r = httpClient
        .target(url)
        .path("students")
        .path("{id}")
        .resolveTemplate("id", id)
        .request()
        .put(Entity.json(student));
    return r.readEntity(Student.class);
}
```

Izsek 5.12: Funkcija za posodabljanje študentov

Primer klica storitve REST za brisanje študentov prikazuje izsek kode 5.13.

```
public boolean izbrisiStudenta(Integer id) {
    Response r = httpClient
        .target(url)
        .path("students")
        .path("{id}")
        .resolveTemplate("id", id)
        .request()
        .delete();
    return true;
}
```

Izsek 5.13: Funkcija za brisanje študentov

Opazimo, da so si funkcije zelo podobne. Razlike opazimo le v tipu poslane zahteve (GET, POST, PUT, DELETE), zato smo opisali zgolj en tip zahteve. V naslednjem delu bomo predvidevali, da so funkcije implementirane in delujejo pravilno.

5.1.4 Implementacija razreševalskih funkcij

Zadnji del aplikacije so razreševalske funkcije. Razdelili smo jih na tri javanske razrede, podobno kot so bila razdeljena zrna CDI. Zaradi uporabe razširitve KumuluzEE GraphQL je pisanje razreševalskih funkcij enakovredno pisanju javanskih funkcij. Potrebno je dodati le anotacije (vrsta operacije, parametri operacije ...). Razreševalske funkcije zaradi berljivosti in lažjega vzdrževanja ne vsebujejo veliko kode, celotna programska logika se nahaja v zrnih CDI. Podobno kot pri zrnih, bomo tudi tukaj predstavili le razreševalske funkcije za študente (da se izognemu nepotrebnemu podvajanju programske kode).

```
@ApplicationScoped
@GraphQLClass
public class UporabnikiRazresevalskeFunkcije {
    @Inject
    private UporabnikiZrno uporabnikiZrno;

    @Inject
    private ProstoriZrno prostoriZrno;

    @Inject
    private PredmetiZrno predmetiZrno;

    // vrinjena polja
    @GraphQLQuery
    public List<Predmet> predmeti(
        @GraphQLContext Student student) {
        List<Predmet> predmeti = new ArrayList<Predmet>();
        for(Integer id: student.getSubjectIds()) {
            predmeti.add(predmetiZrno.vrniPredmet(id));
        }
        return predmeti;
    }

    // bralne operacije
    @GraphQLQuery
    public List<Student> vrniStudente() {
        return uporabnikiZrno.vrniStudente();
    }

    @GraphQLQuery
    public Student vrniStudenta(@GraphQLArgument(name="id")
```

```
        @GraphQLNonNull Integer id) {
        return uporabnikiZrno.vrniStudenta(id);
    }

    //pisalne operacije
    @GraphQLMutation
    public Student dodajStudenta(@GraphQLArgument(name="place"
        @GraphQLNonNull Student student) {
        return uporabnikiZrno.dodajStudenta(student);
    }

    @GraphQLMutation
    public Student urediStudenta(@GraphQLArgument(name="id")
        @GraphQLNonNull Integer id,
        @GraphQLArgument(name="student")
        @GraphQLNonNull Student student) {
        return uporabnikiZrno.spremeniStudenta(id, student);
    }

    @GraphQLMutation
    public boolean izbrisiStudenta(@GraphQLArgument(name="id")
        @GraphQLNonNull Integer id) {
        return uporabnikiZrno.izbrisiStudenta(id);
    }
}
```

Izsek 5.14: Razreševalske funkcije entitete Študent

Izsek kode 5.14 vsebuje vrinjeno polje, dve bralni in tri pisalne operacije. Vrinjeno polje z imenom *predmeti* je avtomatično dodano na entiteto *Študent* (zaradi anotacije *GraphQLContext*), medtem ko je polje *subjectIds* odstranjeno zaradi anotacije *GraphQLIgnore*. Zaradi uporabe razširitve KumuluzEE GraphQL in anotacij se vse funkcije in entitete preslikajo v shemo GraphQL.

5.1.5 Varnost

Trenutno so vse naše operacije javno dostopne. V realnih projektih je potrebno operacije zaščititi. Za zaščito bomo uporabili razširitev KumuluzEE Security [27], ki smo jo dopolnili s podporo za razširitev KumuluzEE GraphQL. Za dodajanje varnosti sledimo naslednjim korakom:

- Preden lahko začnemo s konfiguracijo, moramo vključiti razširitev v datoteki *pom.xml* (izsek kode 5.15).

```
<dependency>
   <groupId>com.kumuluz.ee.security</groupId>
   <artifactId>kumuluzee-security-keycloak</artifactId>
   <version>trenutna_razlicica</version>
</dependency>
```

Izsek 5.15: Vključitev razširitve KumuluzEE Security

- Sledi definicija vlog na aplikacijskem razredu (izsek kode 5.16).

```
@GraphQLApplicationClass
@DeclareRoles({"student","profesor","skrbnik"})
public class App extends GraphQLApplication {}
```

Izsek 5.16: Definicija vlog

- V naslednjem koraku moramo anotirati razrede, ki vsebujejo razreševalske funkcije, z anotacijo „Secure“ (izsek kode 5.17).

```
@ApplicationScoped
@GraphQLClass
@Secure
public class UporabnikiRazresevalskeFunkcije { ... }
```

Izsek 5.17: Anotiranje razredov z anotacijo Secure

- Preostane nam le še anotiranje posameznih razreševalskih funkcij z javanskimi varnostnimi anotacijami (izsek kode 5.18).

```
@GraphQLMutation
@RolesAllowed("skrbnik")
public Student dodajStudenta(
    @GraphQLArgument(name="student")
    @GraphQLNonNull Student student) {
    return uporabnikiZrno.dodajStudenta(student);
}

@RolesAllowed("{"profesor","student"})
@GraphQLQuery
public Student vrniStudenta(
    @GraphQLArgument(name="id")
    @GraphQLNonNull Integer id) {
    return uporabnikiZrno.vrniStudenta(id);
}
```

Izsek 5.18: Anotiranje funkcij z varnostnimi anotacijami

5.1.6 Testna poizvedba

Za poglobljeno razumevanje bomo podrobneje opisali izvedbo operacije *vrniStudente*. Uporabili bomo poizvedbo, ki je prikazana na izseku kode 5.19.

```
{
   vrniStudente {
     id
     ime
     priimek
     predmeti {
       id
       naziv
     }
   }
 }
```

Izsek 5.19: Primer poizvedbe

Ko strežnik prejme zgornjo poizvedbo se zgodi naslednje:

- Klic razreševalske funkcije *vrniStudente*.
- Klic funkcije *vrniStudente* v *UporabnikiZrno*.
- Zahteva REST na mikrostoritev *uporabniki*.
- Naslednje akcije za vsakega študenta:
  - Klic funkcije *getter* za pridobivanje enoličnega identifikatorja študenta.
  - Klic funkcije *getter* za pridobivanje imena študenta.
  - Klic funkcije *getter* za pridobivanje priimka študenta.
  - Klic vrinjene razreševalske funkcije za pridobivanje predmetov študenta.
  - Klic funkcije *getter* za pridobivanje enoličnih identifikatorjev predmetov študenta.

  - Naslednje akcije za vsak študentov predmet:
    * Klic funkcije *vrniPredmet* v *PredmetiZrno*.
    * Zahteva REST na mikrostoritev *predmeti*.
    * Klic funkcije *getter* za enoličnega identifikatorja predmeta.
    * Klic funkcije *getter* za naziv predmeta.

- Združitev pridobljenih podatkov v format JSON.
- Vračanje rezultata uporabniku.

Klici funkcij niso nujno v takšnem zaporedju, nekateri so tudi paralelni; izvajanje je odvisno od implementacije GraphQL in konfiguracije strežnika.

5.2 Evalvacija

Skozi praktični primer smo prikazali, da je implementirana razširitev uporabna v praksi. Z implementacijo agregatorja GraphQL smo dosegli združitev vseh podatkov iz treh mikrostoritev v enotni podatkovni model. S tem smo odstranili nepotrebno pošiljanje poizvedb in odvečno prenašanje podatkov, saj lahko z eno poizvedbo pridobimo vse zahtevane podatke. Ker končen uporabnik poizvedbe pošilja le na agregator, ga tehnologija v ozadju ne zanima. Večje spremembe na poizvedovanje nanj ne bodo vplivale, saj ob ohranitvi enake sheme ne bo opazil nobene razlike.

Druga pomembna stvar, ki smo jo dosegli z razvojem razširitve, je preprostost uporabe. Pisanje kode je relativno preprosto, saj so koncepti GraphQL abstrahirani z uporabo anotacij. Posledično se razvijalcem ni potrebno poglabljati v tehnologijo, ampak je za razvoj potrebno le znanje osnovnih konceptov (predvsem poizvedovanja). Čeprav je razširitev namenjena poenostavitvi razvoja mikrostoritev GraphQL, vsebuje tudi napredne funkcionalnosti. S tem smo naprednim uporabnikom omogočili prilagajanje nastavitev izvajalnega okolja, ki v večini primerov ni potrebno.

Ne smemo pozabiti na integracijo z drugimi razširitvami KumuluzEE. V praktičnem delu smo poleg implementirane razširitve uporabili tudi razširitvi KumuluzEE Security za varnost in KumuluzEE Discovery za odkrivanje posameznih mikrostoritev. V produkcijskem okolju je integracija ključnega pomena:

- Brez varnosti bi bile naše aplikacije odprte za nepooblaščen dostop.
- Brez odkrivanja storitev bi bilo horizontalno skaliranje praktično nemogoče, ker bi morali ročno skrbeti za spletne naslove posameznih mikrostoritev.
- Optimizacija poizvedb JPA poskrbi, da se iz baze pridobijo le zahtevani podatki (ki smo jih navedli v poizvedbi) in ne celotne entitete.

Poglavje 6

Zaključek

V okviru diplomske naloge smo tehnologijo GraphQL podrobneje raziskali in jo povezali z mikrostoritvami. Glavna cilja diplomske naloge sta bila analizirati pomanjkljivosti skupnega delovanja GraphQL in mikrostoritev ter razviti programsko rešitev, ki odpravlja te pomanjkljivosti. Ker sam razvoj rešitve ni bil dovolj za učinkovito razvijanje mikrostoritev GraphQL, smo si kot dodaten cilj zadali še izpolnjevanje konceptov arhitekture cloud-native.

Pri analizi smo ugotovili, da v javanskem ekosistemu ni dovolj programske podpore za GraphQL. Manjka standardizacija, ki je zelo pomembna, saj zagotavlja stabilnost tehnologije. Problem smo odkrili tudi pri uporabi javanske implementacije GraphQL. Uporaba ni preprosta, saj je že definiranje sheme in razreševalskih funkcij kompleksno. Da bi identificirane pomanjkljivosti odpravili, smo se odločili za razvoj razširitve KumuluzEE GraphQL. S tem smo uporabnikom ogrodja KumuluzEE omogočili lažjo uporabo GraphQL v mikrostoritvah. Razširitev je odprtokodna in javno objavljena, podpira pa dovolj funkcionalnosti za osnovno postavitev strežnika GraphQL z bralnimi in pisalnimi operacijami. Delovanje razširitve smo predstavili na praktičnem primeru.

Da bi dosegli ustreznost razširitve z arhitekturo cloud-native, smo poleg razvoja razširitve dopolnili tudi druge komponente KumuluzEE. Razširitvi KumuluzEE Security smo dodali podporo za nastavitev varnosti iz mikrosto-

ritve GraphQL. Razširitev KumuluzEE Discovery smo dopolnili s podporo za definiranje tipa mikrostoritve in jo tako naredili generično. Razširitev KumuluzEE REST smo direktno vključili v implementirano razširitev in dodali ovojne funkcije za uporabo njenih funkcionalnosti. Tako smo izpolnili vse cilje, ki smo si jih zadali pred pisanjem diplomske naloge.

V prihodnosti bomo nadaljevali z razvojem razširitve KumuluzEE GraphQL. Najprej bomo dodali naslednje funkcionalnosti: podporo za naročnine (*subscriptions*) in podporo za način razvoja *najprej shema* (*schema-first*), ki omogoča večjo organiziranost pri načrtovanju projektov.

Literatura

[1] What is the response format of a multi-operations query? Dosegljivo: https://github.com/facebook/graphql/issues/29, 2015. [Dostopano: 21. 6. 2018].

[2] 4 Challenges You Need to Address with Microservices Adoption. Dosegljivo: https://blog.appdynamics.com/product/4-challenges-you-need-to-address-with-microservices-adoption/, 2016. [Dostopano: 2. 7. 2018].

[3] GraphQL explained. Dosegljivo: https://dev-blog.apollodata.com/graphql-explained-5844742f195e, 2016. [Dostopano: 26. 6. 2018].

[4] GraphQL vs. REST. Dosegljivo: https://dev-blog.apollodata.com/graphql-vs-rest-5d425123e34b, 2017. [Dostopano: 28. 6. 2018].

[5] Pain Points of GraphQL. Dosegljivo: https://labs.getninjas.com.br/pain-points-of-graphql-7e83ba5ddef7, 2017. [Dostopano: 31. 8. 2018].

[6] Sharing data in a Microservices Architecture using GraphQL. Dosegljivo: https://labs.getninjas.com.br/sharing-data-in-a-microservices-architecture-using-graphql-97db59357602, 2017. [Dostopano: 31. 8. 2018].

[7] What Is Cloud-Native Architecture and Why Is It So Important? Dosegljivo: https://www.contino.io/insights/what-is-cloud-native-

architecture-and-why-is-it-so-important, 2017. [Dostopano: 2. 7. 2018].

[8] Apollo GraphQL. Dosegljivo: https://www.apollographql.com/, 2018. [Dostopano: 17. 7. 2018].

[9] Cloud-Native Applications: Ship Faster, Reduce Risk, Grow Your Business. Dosegljivo: https://pivotal.io/cloud-native, 2018. [Dostopano: 22. 8. 2018].

[10] Docker. Dosegljivo: https://www.docker.com/, 2018. [Dostopano: 21. 8. 2018].

[11] etcd. Dosegljivo: https://github.com/coreos/etcd, 2018. [Dostopano: 17. 7. 2018].

[12] GitHub GraphQL API v4. Dosegljivo: https://developer.github.com/v4/guides/intro-to-graphql/, 2018. [Dostopano: 28. 6. 2018].

[13] GraphiQL. Dosegljivo: https://github.com/graphql/graphiql, 2018. [Dostopano: 17. 7. 2018].

[14] GraphQL and GraphiQL Spring Framework Boot Starters. Dosegljivo: https://github.com/graphql-java/graphql-spring-boot, 2018. [Dostopano: 17. 7. 2018].

[15] GraphQL Best Practices. Dosegljivo: https://graphql.org/learn/best-practices/, 2018. [Dostopano: 31. 8. 2018].

[16] GraphQL libraries. Dosegljivo: https://graphql.github.io/code/, 2018. [Dostopano: 26. 6. 2018].

[17] GraphQL Prisma. Dosegljivo: https://www.prisma.io/, 2018. [Dostopano: 17. 7. 2018].

[18] GraphQL specification. Dosegljivo: https://facebook.github.io/graphql/draft/, 2018. [Dostopano: 21. 6. 2018].

[19] GraphQL.js. Dosegljivo: https://github.com/graphql/graphql-js, 2018. [Dostopano: 26. 6. 2018].

[20] Introspection. Dosegljivo: https://graphql.org/learn/introspection/, 2018. [Dostopano: 28. 6. 2018].

[21] Keycloak. Dosegljivo: https://www.keycloak.org/, 2018. [Dostopano: 21. 8. 2018].

[22] Kubernetes. Dosegljivo: https://kubernetes.io/, 2018. [Dostopano: 21. 8. 2018].

[23] KumuluzEE. Dosegljivo: https://ee.kumuluz.com/, 2018. [Dostopano: 17. 7. 2018].

[24] KumuluzEE Discovery. Dosegljivo: https://github.com/kumuluz/kumuluzee-discovery, 2018. [Dostopano: 17. 7. 2018].

[25] KumuluzEE GraphQL. Dosegljivo: https://github.com/kumuluz/kumuluzee-graphql, 2018. [Dostopano: 17. 7. 2018].

[26] KumuluzEE REST. Dosegljivo: https://github.com/kumuluz/kumuluzee-rest, 2018. [Dostopano: 22. 8. 2018].

[27] KumuluzEE Security. Dosegljivo: https://github.com/kumuluz/kumuluzee-security, 2018. [Dostopano: 21. 8. 2018].

[28] MongoDB. Dosegljivo: https://www.mongodb.com/, 2018. [Dostopano: 31. 8. 2018].

[29] Query caching. Dosegljivo: https://graphql-java.readthedocs.io/en/latest/execution.html#query-caching, 2018. [Dostopano: 27. 6. 2018].

[30] Schema Directives. Dosegljivo: https://graphql-java.readthedocs.io/en/latest/sdldirectives.html, 2018. [Dostopano: 21. 6. 2018].

[31] Simplified faculty model with microservices. Dosegljivo: https://github.com/evader1337/simplified-faculty-microservices, 2018. [Dostopano: 17. 7. 2018].

[32] Spring Boot. Dosegljivo: https://spring.io/projects/spring-boot, 2018. [Dostopano: 17. 7. 2018].

[33] WildFly Swarm. Dosegljivo: http://wildfly-swarm.io/, 2018. [Dostopano: 17. 7. 2018].

[34] Stephen Grider. GraphQL with React: The Complete Developers Guide. Dosegljivo: https://www.udemy.com/graphql-with-react-course/, 2017. [Dostopano: 06.08.2018].

[35] GraphQL: Schemas and Types. Dosegljivo: https://https://graphql.github.io/learn/schema/, 2018. [Dostopano: 20. 6. 2018].

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

[37] Christian Posta. *Microservices for Java Developers*. O'Reilly Media, 2016.

[38] Mark Richards. *Microservices AntiPatterns and Pitfalls*. O'Reilly Media, 2016.

[39] James A. Scott. *A practival guide to microservices and containers*. 2017.

[40] Eva Zupančič. KumuluzEE has the fastest start-up and smallest memory footprint. Dosegljivo: https://blog.kumuluz.com/product/2017/10/29/kumuluzee-fastest-start-up-smallest-memory-footprint, 2017. [Dostopano: 31. 8. 2018].