Tehničko veleučilište u Zagrebu  
Stručni studij Računarstva

Marin Jurišić

0246082071

**Analiza GraphQL radnog okvira u kontekstu razvoja ASP.NET web aplikacija**

Završni rad br. 1383

Zagreb, lipanj, 2022.

# **Sažetak**

GraphQL je novi jezik za pisanje upita razvijen od strane tvrtke Meta platforms. Najčešće se uspoređuje s REST API-em, te se često predstavlja kao njegova zamjena. U ovom radu analizira se korištenje GraphQL-a u .NET 6 razvojnom okviru koristeći Hot Chocolate GraphQL server i Entity Framework. Cilj rada je analizirati i usporediti implementaciju GraphQL radnog okvira sa standardnom REST API implementacijom na primjeru E-Library aplikacije, te istaknuti prednosti i mane korištenja svakog od ova dva pristupa. Posebna pažnja posvećena je sigurnosti GraphQL-a, te su istaknute najvažnije metode zaštite GraphQL servera. Iz navedenog je vidljivo da iako korištenje GraphQL-a predstavlja neke jedinstvene sigurnosne izazove, postoje alati koji čine GraphQL dovoljno sigurnim za produkcijsku upotrebu. Provedeno je i testiranje i usporedba performansi GraphQL-a i REST-a, iz kojeg se zaključuje da usprkos tome što je REST još uvijek nešto efikasniji, postoje opravdani razlozi za korištenje GraphQL-a u određenim razvojnim ekosustavima.

***Ključne riječi*** : GraphQL, REST, API, .NET, web aplikacija

**Sadržaj**:

**[Sažetak](#_Toc106723595)** [1](#_Toc106723595)

[Popis kratica 4](#_Toc106723596)

[Popis slika 5](#_Toc106723597)

**[1.](#_Toc106723598)****[Uvod](#_Toc106723598)** [5](#_Toc106723598)

**[2.](#_Toc106723599)****[GraphQL](#_Toc106723599)** [6](#_Toc106723599)

[2.1. Koncept razvoja aplikacije 7](#_Toc106723600)

[2.2. Hipoteza 7](#_Toc106723601)

**[3.](#_Toc106723602)****[Arhitektura aplikacije](#_Toc106723602)** [8](#_Toc106723602)

[3.1. Baza podataka 9](#_Toc106723603)

[3.2. Prezentacijski sloj 9](#_Toc106723604)

[3.2.1. Struktura prezentacijskog sloja aplikacije 12](#_Toc106723605)

[3.3. Sloj poslovne logike 12](#_Toc106723606)

[3.3.1. Struktura sloja poslovne logike 13](#_Toc106723607)

[3.4. Implementacija REST API-a 13](#_Toc106723608)

[3.4.1. Podatkovni sloj 14](#_Toc106723609)

**[4.](#_Toc106723610)****[Implementacija GraphQL-a](#_Toc106723610)** [16](#_Toc106723610)

[4.1. Postavljanje GraphQL servera 16](#_Toc106723611)

[4.2. Klase potrebne za izvršavanje upita 18](#_Toc106723612)

[4.3. Dodatne mogućnosti GraphQL upita 18](#_Toc106723613)

[4.4. GraphQL mutacije 20](#_Toc106723614)

**[5.](#_Toc106723615)****[Sigurnost GraphQL-a](#_Toc106723615)** [21](#_Toc106723615)

[5.1.](#_Toc106723616) *[Execution timeout](#_Toc106723616)* [i](#_Toc106723616) *[Pagination amount](#_Toc106723616)* [21](#_Toc106723616)

*[5.2.](#_Toc106723617)**[Max query depth](#_Toc106723617)* [i](#_Toc106723617) *[Max query complexity](#_Toc106723617)* [22](#_Toc106723617)

[5.3. Autentikacija i autorizacija 24](#_Toc106723618)

**[6.](#_Toc106723619)****[Usporedba performansi GraphQL-a i REST API-a](#_Toc106723619)** [31](#_Toc106723619)

[6.1. Rezultati mjerenja 31](#_Toc106723620)

**[7.](#_Toc106723621)****[Zaključak](#_Toc106723621)** [36](#_Toc106723621)

# Popis kratica

* API – *Application Programming Interface* / Aplikacijsko programsko sučelje
* CORS - *Cross-origin resource sharing* / Preuzimanje resursa sa različitih izvora
* CSS - *Cascading Style Sheets*
* DAL – *Data Access Layer*
* DBMS – *Database Management System* / Sustav upravljanja bazom podataka
* DOM - *Document Object Model*
* HTML - *HyperText Markup Language*
* HTTP - *Hypertext Transfer Protocol*
* JSON - *JavaScript Object Notation*
* JWT – *JSON Web Token*
* ms – milisekunde
* MS SQL – *Microsoft SQL Server*
* MVC - *Model-View-Controller*
* REST - *Representational state transfer*
* SOAP - *Simple Object Access Protocol*
* SQL - *Structured Query Language* / strukturirani jezik za upite
* URI - *Uniform Resource Identifier*

# Popis slika

[Slika 1. ER dijagram baze podataka [Autorski rad] 4](#_Toc106268009)

[Slika 2. NuGet paketi korišteni u projektu. [Autorski rad] 12](#_Toc106268010)

[Slika 3. Lijevo: primjer kompleksnog upita. Desno: poruka o pogrešci. [Autorski rad] 20](#_Toc106268011)

[Slika 4. Pokušaj pristupa podacima kojima korisnik nema pristup. [Autorski rad] 23](#_Toc106268012)

[Slika 5. Podatci od dohvaćanju veće količine podataka, mjereno pomoću Performance API-a [Autorski rad] 27](#_Toc106268013)

[Slika 6. Podatci od dohvaćanju veće količine podataka, mjereno pomoću Chrome DevTools. [Autorski rad] 27](#_Toc106268014)

[Slika 7. Podatci od dohvaćanju manje količine podataka, mjereno pomoću Performance API-a. [Autorski rad] 28](#_Toc106268015)

[Slika 8. Podatci od dohvaćanju manje količine podataka, mjereno pomoću Chrome DevTools. [Autorski rad] 28](#_Toc106268016)

[Slika 9. Podatci od slanju manje količine podataka na server, mjereno pomoću Performance API-a. [Autorski rad] 29](#_Toc106268017)

[Slika 10. Podatci od slanju manje količine podataka na server, mjereno pomoću Chrome DevTools. [Autorski rad] 29](#_Toc106268018)

# **Uvod**

GraphQL je jezik za pisanje i strukturiranje upita putem HTTP(S) protokola. GraphQL pruža potpun i razumljiv opis podataka pomoću GraphQL sheme, te daje klijentu mogućnost da dohvaća točno one podatke koji mu trebaju i ništa više[1]. Razvijan je od strane Facebook-a do 2018., kada je osnovana GraphQL zaklada.

U ovom radu razmatra se upotreba GraphQL-a u izradi web aplikacija u .NET 6 programskom okviru, te se uspoređuje s REST API-em u području sigurnosti, performansi i načina upotrebe. Koristi se Hot Chocolate server verzije 12 za .NET programski okvir. Hot Chocolate prati najnoviju GraphQL specifikaciju i pojednostavljuje postavljanje GraphQL servera i pisanje GraphQL shema[2].

Usprkos rastućoj popularnosti GraphQL-a u izradi web aplikacija , još uvijek postoje određene nedoumice među razvojnim inženjerima pri odlučivanju o tome koja bi se tehnologija trebala koristiti: GraphQL ili REST? Mnogi se još uvijek odlučuju za REST, s obzirom da je to svima poznata i sveprisutna tehnologija koja se koristi već više od dvadeset godina. Neki od razloga koji se za to navode su familijarnost s REST-om, naizgledna kompleksnost GraphQL upita, te nedoumice oko njegove sigurnosti i performansi. Ovaj rad je nastao s ciljem da se daju odgovori na ta pitanja na praktičnom primjeru aplikacije napravljene korištenjem i jedne i druge tehnologije, te da se jasno navedu prednosti i mane svakog od ova pristupa.

U prvom poglavlju objašnjen je koncept rada aplikacije te teorija na kojoj se ona bazira. Opisane su osnove arhitekture aplikacije, način na koji ona funkcionira i što je cilj njene izrade. U sljedećem poglavlju arhitektura aplikacije je razložena na slojeve, te je za svaki detaljno opisan postupak njegove izrade i njegova namjena. Posebna pažnja posvećena je implementaciji i upotrebi GraphQL-a. Nakon toga slijedi poglavlje koje govori o sigurnosti GraphQL aplikacija te ističe posebne ranjivosti GraphQL servera i načine na koji se te ranjivosti mogu ispraviti. Posljednje poglavlje bavi se performansama GraphQL-a u usporedbi s REST API-em. U njemu se prikazuju rezultati testiranja odrađenog na E-Library aplikaciji, na temelju kojih se donosi zaključak o relativnim performansama GraphQL-a.

# **GraphQL**

GraphQL je jezik za pisanje upita koji se izvršava na serveru i koji daje prioritet klijentima dajući im točno one podatke koje traže i ništa više. Dizajniran je da API-je učini bržim, fleksibilnijim i prilagođenim programerima. Kao alternativa REST-u, GraphQL dopušta programerima da konstruiraju zahtjeve koji povlače podatke iz više izvora podataka u jednom API pozivu. Osim toga, GraphQL pružateljima API-ja pruža fleksibilnost dodavanja ili odbijanja polja bez utjecaja na postojeće upite. Programeri mogu izgraditi API-je s bilo kojim metodama koje žele, a GraphQL specifikacija će osigurati da funkcioniraju na predvidljiv način za klijente. [3]

Za razliku od mnogih drugih jezika za pisanje upita, kao što je na primjer SQL, GraphQL nije vezan za određeni tip spremišta podataka, već je potpuno agnostičan prema izvoru podataka. To znači da se GraphQL može koristiti za dohvaćanje podataka iz bilo kojeg izvora, uključujući i REST API. [4]

GraphQL se izvršava na serveru, dok se GraphQL upiti najčešće šalju putem HTTP zahtjeva. GraphQL se bazira na dvjema vrstama zahtjeva: upitima i mutacijama. Upiti služe isključivo za dohvaćanje podataka, dok se mutacije mogu koristiti za stvaranje, ažuriranje i brisanje podataka. Glavna prednost GraphQL-a je to što sprječava dohvaćanje previše ili premalo podataka sa servera. GraphQL je, zahvaljujući GraphQL shemi, potpuno neovisan o vlastitioj implementaciji. Naime, neovisno o programskom jeziku ili okviru koji je korišten na serveru, svi GraphQL upiti će biti isti. U ovom radu fokus je na GraphQL aplikaciji, te njenoj sigurnosti i performansama, implementiranoj u .NET 6 razvojnom okruženju koristeći Hot Chocolate server.

* 1. Koncept razvoja aplikacije

E-Library aplikacija je koncipirana kao jednostavan praktični primjer aplikacije koja implementira i GraphQL i REST na način da se zapravo sastoji od dvije identične aplikacije koje se razlikuju samo po korištenju GraphQL-a ili REST-a. Ovakav pristup dozvoljava direktnu usporedbu GraphQL-a i REST-a, kako u području performansi, tako i u potrebnom pristupu razvoju aplikacije i količini koda potrebnoj za ostvarivanje potpune funkcionalnosti svake od ovih tehnologija. S druge strane, ovakav pristup ima i svojih mana. Naime, skup podataka je relativno ograničen, čime se riskira to da se neće uspješno primijetiti sve prednosti jednog od ovih pristupa.

S obzirom da je GraphQL novija, manje poznata tehnologija, fokus ovoga rada je ipak na njegovoj implementaciji i mogućnostima. Usprkos tome što je GraphQL relativno nova tehnologija, SmartBear u svom godišnjem izvještaju iz 2020. godine iznosi da GraphQL koristi čak 19% razvojnih inženjera[3]. Imajući to na umu, ovaj rad se fokusira na implementaciju GraphQL-a u .NET razvojnom okruženju, a posebna pažnja posvećena je njegovoj sigurnosti i performansama.

* 1. Hipoteza

U sklopu ovoga rada obavljena su dva testiranja kako bi se odgovorilo na određena pitanja i uklonile određene sumnje o korištenju GraphQL-a.

Prvo testiranje se tiče sigurnosti i sigurnosnih propusta GraphQL aplikacija. Cilj ovoga testiranja bio je pokazati kako se GraphQL server može zaštiti od napada kojima je cilj usporiti ili srušiti server, kao i od malicioznih upita kojima je cilj doći do podataka za koje nisu autorizirani. Pretpostavka je da, s obzirom na veliku raširenost i popularnost GraphQL-a, postoje mehanizmi i tehnike koje uspješno štite GraphQL server od svih vrsta napada.

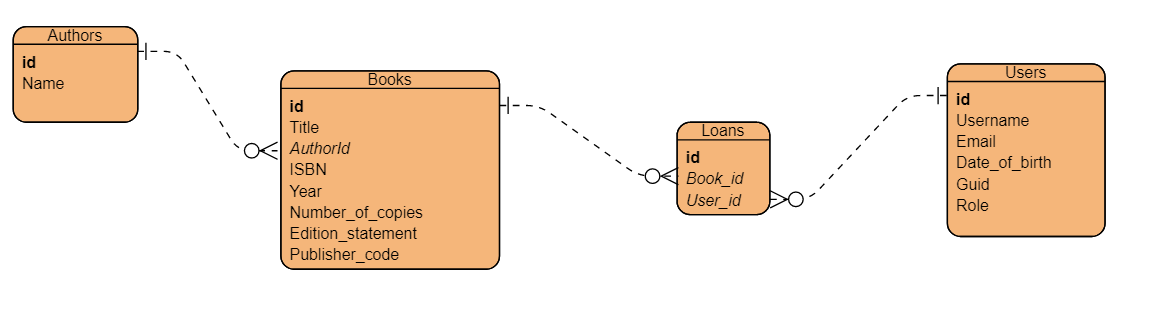
Drugo testiranje bavi se performansama GraphQL-a. Ovo testiranje razmatra samo brzinu prijenosa podataka putem GraphQL-a i REST-a, ali se uzima u obzir da to nije jedino mjerilo performansi vrijedno spomena. S obzirom da je GraphQL novija tehnologija, bilo bi lako pretpostaviti da je brži i efikasniji. Međutim, uzevši u obzir da REST direktno rukuje podatcima, dok GraphQL mora razriješiti svaki upit prije njegovog izvršavanja, pretpostavka je da će REST ipak biti nešto brži, dok prednosti GraphQL-a leže u drugim područjima efikasnosti.

# **Arhitektura aplikacije**

U ovom poglavlju detaljno je opisana arhitektura aplikacije, te tehnologije korištene pri njenom razvoju, počevši s bazom podataka. Aplikacija je rađena u .NET 6 programskom okviru, s obzirom da je to najnovije izdanje .NET-a. Za GraphQL server odabran je Hot Chocolate verzije 12, kao najnoviji popularni GraphQL server za .NET. Hot Chocolate je implementiran koristeći Entity Framework kao konekciju na bazu podataka. Osim toga, na *frontend*-u su korišteni Bootstrap i JQuery. Podatci se dohvaćaju iz MS SQL Server baze podataka. MS SQL baza je odabrana primarno radi jednostavnosti uporabe i integracije s .NET i Entity Framework programskim okvirima. Entity Framework (EF) je ORM programski okvir koji obavlja operacije nad bazom podataka i mapira podatke iz baze podataka na objekte u objektno orijentiranom programiranju.

* 1. Baza podataka

Baza podataka na kojoj se aplikacija zasniva je Microsoft SQL Server. S obzirom da je proizvedena od strane Microsofta, ova baza se jednostavno i brzo integrira s .NET programskim okvirom i Entity Framework alatom koji su također razvijani i podržani od strane Microsofta. U bazi se nalaze 4 tablice nad kojima se vrše upiti i koje predstavljaju online katalog knjižnice. Tablica *Books* je najbitnija ovdje, s obzirom da sadrži najviše podataka te će se pri dohvaćanju podataka iz nje najjasnije pokazati eventualne razlike u performansama između GraphQL-a i REST-a. Tablica *Books* povezana je s tablicom *Authors*, dok tablica *Loans* spaja tablice *Users* i *Books*, te predstavlja mehanizam posuđivanja knjiga.



Slika . ER dijagram baze podataka [Autorski rad]

* 1. Prezentacijski sloj

Prezentacijski sloj ili *frontend* je dio aplikacije koji korisnik vidi i s kojim može vršiti interakcije. Pri izradi frontenda korištene su HTML 5, CSS 3 i JavaScript tehnologije. Radi lakoće izgradnje grafičkih sučelja i ubrzanja razvoja aplikacije korišten je Bootstrap 4. Također je korišten JQuery programski okvir za JavaScript koji omogućuje olakšano dohvaćanje i manipulaciju HTML DOM elementima. Uz to, u aplikaciji je napravljeno nekoliko pomoćnih funkcija kako bi se ubrzao i pojednostavio razvoj i smanjila pojava greški. Većina ovih pomoćnih funkcija se odnosi na pojednostavljivanje korištenja JavaScript Fetch API-a. Fetch API služi za slanje zahtjeva preko mreže koristeći HTTP protokole, te omogućuje slanje i primanje podataka, kao i definiranje HTTP zaglavlja i CORS-a [4]. Ovo programsko sučelje koristi se za interakcije i sa REST API-em i GraphQL API-em.

**let** id **=** js**.**page**.**get\_id**();**

const query **=** `

query **{**

book **(where:** **{**author**:** **{**id**:** **{**eq**:** $**{**id**}}}},** first**:** 100**)** **{**

nodes **{**

id**,**

title**,**

year**,**

isbn**,**

author **{**

id**,**

name

**}**

**}**

**}**

**}**

`**;**

**let** json **=** **await** js**.**api**.**graphql**(**query**);**

Isječak koda 1. Dohvaćanje popisa svih knjiga određenog autora koristeći GraphQL. [Autorski rad]

Pozivi na REST API vrše se kao GET zahtjevi pri dohvaćanju podataka sa servera, dok se za slanje podataka na server koriste POST zahtjevi. S druge strane, svi pozivi na GraphQL *endpoint* (krajnju točku) vrše se kao POST zahtjevi kojima se u tijelu zahtjeva šalje GraphQL upit koji se treba izvršiti. S obzirom da se svi GraphQL upiti šalju na istu krajnju točku i vraćaju odgovor u JSON formatu, napravljena je posebna pomoćna funkcija koja prima samo GraphQL upit koji treba poslati na server u tijelu zahtjeva i koja vraća formatirani odgovor servera, dok je za korištenje REST API-a napravljeno više pomoćnih funkcija. Funkcije za GET zahtjev primaju samo URI na koji se šalju, te vraćaju ili HTTP odgovor ili formatirani JSON, dok funkcije za slanje POST zahtjeva primaju i URI na koji se šalju i podatke koji će se poslati u tijelu zahtjeva. Slijede primjeri nekih funkcija koje se koriste.

**let** id **=** js**.**page**.**get\_id**();**

**let** url **=** "api/books/get\_by\_author?id=" **+** id**;**

**let** data **=** **await** js**.**api**.**get\_json**(**url**);**

Isječak koda 2. Dohvaćanje popisa svih knjiga određenog autora koristeći REST API. [Autorski rad]

**this.**save\_book **=** **async** **()** **=>** **{**

**let** id **=** js**.**page**.**get\_id**();**

**let** url **=**"/api/books/save"**;**

**let** data **=** **{};**

data**.**id **=** id**;**

data**.**isbn **=** book\_isbn**.**value**;**

data**.**year **=** book\_year**.**value**;**

data**.**number\_of\_copies **=** book\_copies**.**value**;**

data**.**price **=** book\_price**.**value**;**

data**.**item\_call\_number **=** book\_call\_number**.**value**;**

data**.**publisher\_code **=** book\_publisher**.**value**;**

data**.**edition\_statement **=** book\_edition**.**value**;**

**let** res **=** **await** js**.**api**.**post**(**url**,**data**);**

**}**

Isječak koda 3. Funkcija za spremanje/ažuriranje podataka o knjizi koristeći REST API. [Autorski rad]

**this.**save\_book **=** **async** **()** **=>** **{**

**let** id **=** js**.**page**.**get\_id**();**

**let** number\_of\_copies **=** book\_copies**.**value**;**

**let** price **=** book\_price**.**value**;**

**let** item\_call\_number **=** book\_call\_number**.**value**;**

**let** publisher\_code **=** book\_publisher**.**value**;**

**let** edition\_statement **=** book\_edition**.**value**;**

**let** query **=** `

mutation {

updateBook(input: {id: ${id}, edition\_statement: "${edition\_statement}", price: "${price}",

item\_call\_number: "${item\_call\_number}", publisher\_code: "${publisher\_code}", number\_of\_copies: "${number\_of\_copies}"}) {

book {

id,

title,

year,

authorId

}

}

}

`**;**

**await** js**.**api**.**graphql**(**query**);**

**}**

Isječak koda 4. Funkcija za spremanje/ažuriranje podataka o knjizi koristeći GraphQL. [Autorski rad]

Na prethodnim isječcima koda vide se primjeri nekih REST i GraphQL poziva. Bitno je primijetiti da usprkos tome što se GraphQL pozivi čine puno verbozniji, to nije nužno tako. Naime, dok se u GraphQL polja koja se žele dohvatiti definiraju u GraphQL upitu, u REST-u se ta ista definiraju u kontrolerima i podatkovnom sloju aplikacije. Što to zapravo znači, je da je u slučaju kada *frontend* programer nema pristup *backend-*u, GraphQL zapravo u prednosti, jer programer sam može odlučiti koja polja iz baze želi dohvatiti, dok korisnik REST-a mora koristiti dani API takav kakav je, što može dovesti prekomjernog dohvaćanja podataka ili s druge strane, do toga da nije moguće dohvatiti sve potrebne podatke u jednom pozivu, te da treba vršiti više poziva na server. Upravo ovo je jedna od glavnih prednosti i marketinških točaka GraphQL-a.

* + 1. Struktura prezentacijskog sloja aplikacije

Prezentacijski sloj aplikacije sastoji se od 3 grane do kojih se može doći koristeći navigacijski meni. Dvije grane imaju identično grafičko sučelje, te se razlikuju samo po načinu slanja i primanja podataka. Naime, jedna koristi GraphQL, a druga REST API. Svaka od ovih grana sastoji se od 4 stranice. Prva je popis svih knjiga (tj. prvih 1000 knjiga). Pritiskom na red u tablici otvara se stranica knjige, na kojoj se mogu mijenjati podatci knjige, ili se ista posuditi. Pritiskom na ime autora se otvara popis djela toga autora. S ove stranice se opet može otvoriti stranica s detaljima knjige, ili se može otvoriti stranica za dodavanje nove knjige odabranog autora.

Treća grana je „*User*“ stranica, traži od korisnika da se ulogira ili registrira kao novi korisnik, ovisno o tome ima li postojeći račun. Nakon što je korisnik ulogiran, stranica prikazuje sve korisnikove posudbe iz E-knjižnice. Ovdje korisnik može odabrati hoće li se podatci učitavati putem GraphQL-a ili REST-a. Korisnik također ima mogućnost obrisati sve podatke iz tablice prije osvježavanje tablice kako bi se bolje uočilo potrebno vrijeme za dohvaćanje podataka odabranog API-a.

# Sloj poslovne logike

Sloj poslovne logike je dio programa koji upravlja podatcima i služi kao posrednik između baze podataka i prezentacijskog sloja. Poslovna logika pruža prezentacijskom sloju potrebne podatke, ali je više od samo posrednika između baze i prezentacijskog sloja, te također izvršava dodatnu poslovnu logiku neovisno o prezentacijskom sloju [5]. U ovdje prikazanoj aplikaciji sloj poslovne logike podijeljen je na 3 dijela, koji zrcale podjelu na prezentacijskom sloju. Prvi dio služi za implementaciju REST API-a, drugi za implementaciju GraphQL-a, a treći je odgovoran za prijavljivanje i registraciju korisnika.

* + 1. Struktura sloja poslovne logike

Za upravljanje strukturom aplikacije koriste se .NET 6 MVC kontroleri. Oni primarno služe za navigaciju između stranica. Home kontroler dohvaća naslovnu stranicu i stranice za prijavui registraciju korisnika, te stranicu sa svim posudbama prijavljenog korisnika. Rest i Graph kontroleri upravljaju granama aplikacije koje implementiraju GraphQL i REST API.

* 1. Implementacija REST API-a

REST API je tip web servera koji omogućava klijentu da pristupi resursima koji modeliraju podatke i funkcionalnosti sustava.[6] Za implementaciju REST-a u ovoj aplikaciju korišteni su .NET Web API kontroleri. Kontroleri su mapirani na rutu „*api/[naziv\_kontrolera]/[akcija]*“.

Rute su mapirane korištenjem anotacija. Kontroleri su također označeni kao API kontroleri upotrebom [ApiController] anotacije. Ova anotacija primijenjena na klasu kontrolera omogućuje API specifična ponašanja poput automatskih HTTP 400 odgovora i obaveznog navođenja rute[7].

**using** Microsoft**.**AspNetCore**.**Mvc**;**

**namespace** E\_Library**.**ControllersApi **{**

**[**Route**(**"api/[controller]/[action]"**)]**

**[**ApiController**]**

**public** class AuthorsController **:** ControllerBase **{**

Isječak koda 5. Primjer definicije API kontroler klase. [Autorski rad]

API kontroleri prisutni u aplikaciji su *BooksController*, *AuthorsController*, *LoansController* i *UsersController*. Svaki od njih sadrži metode za dohvaćanje ili slanje tj. izmjenu podataka. API kontroleri nemaju izravan doticaj s bazom podataka, ali u njima se koristi Entity Framework koji se ponaša kao DAL (*Data Access Layer* / podatkovni sloj) i koji vrši direktne upite na bazu. Ovo dozvoljava metodama API kontrolera da budu iznimno kratke (tri do četiri linije koda) i čitljive.

**public** **object** list\_all**()** **{**

var authors **=** \_context**.**Authors**.**OrderBy**(**x **=>** x**.**Id**).**Take**(**100**);**

string json **=** Newtonsoft**.**Json**.**JsonConvert**.**SerializeObject**(**authors**);**

**return** json**;**

**}**

Isječak koda 6. Primjer metode za dohvaćanje prvih sto autora iz baze podataka. [Autorski rad]

Dakle, REST API je strukturiran tako da na određeni kontroler dolazi zahtjev s klijenta, ako je taj zahtjev ispravan izvršava se određena akcija, koja poziva Entity Framework funkciju, koja vraća podatke u model. Ti podatci se onda serijaliziraju u JSON format te se u tom formatu šalju kao odgovor na klijent.

* + 1. Podatkovni sloj

Podatkovni sloj vrši upite direktno na bazu podataka. Sastoji se od Entity Framework poziva koji se izvršavaju nad bazom direktno iz kontrolera.

var books **=** \_context**.**Books**.**Join**(**\_context**.**Authors**,**

b **=>** b**.**AuthorId**,**

a **=>** a**.**Id**,**

**(**b**,** a**)** **=>** **new** **{** Id **=** b**.**Id**,** Isbn **=** b**.**Isbn**,** Title **=** b**.**Title**,**  Year **=** b**.**Year**,** AuthorId **=** b**.**AuthorId**,**

Author\_name **=** a**.**Name **})**

**.**OrderBy**(**x **=>** x**.**Id**)**

**.**Take**(**1000**).**ToList**();**

Isječak koda 7. Primjer metode za dohvaćanje prvih 1000 knjiga u podatkovnom sloju. [Autorski rad]

Iz slike 8 može se zaključiti da je dohvaćanje podataka iz baze vrlo jednostavno. Metoda za dohvaćanje podataka iz baze sastoji se od LINQ upita izvršenog na injektiranom LibraryContext objektu. S druge strane, zapisivanje podataka u bazu je nešto kompleksnije, s obzirom da je potrebno iščitati podatke iz tijela HTTP zahtjeva, dohvatiti podatak koji se izmjenjuje iz baze, učitati podatke u model i onda pozvati Entity Framework metodu za spremanje promjena nad objektom. Pri korištenju Entity Frameworka nije se potrebno brinuti oko validacije, budući da on štiti od SQL injekcije i ne dopušta upisivanje krivog tipa podataka u bazu.

**public** **async** Task**<object>** saveAsync**()** **{**

var input **=** **new** StreamReader**(**Request**.**Body**).**ReadToEnd**();**

Newtonsoft**.**Json**.**Linq**.**JObject i **=** **(**Newtonsoft**.**Json**.**Linq**.**JObject**)** JsonConvert**.**DeserializeObject**(**input**);**

var isbn **=** **(**string**)** i**[**"isbn"**];**

var year **=** **(**string**)** i**[**"year"**];**

var price **=** **(**string**)** i**[**"price"**];**

var edition\_statement **=** **(**string**)** i**[**"edition\_statement"**];**

var item\_call\_number **=** **(**string**)** i**[**"item\_call\_number"**];**

var publisher\_code **=** **(**string**)** i**[**"publisher\_code"**];**

var number\_of\_copies **=** **(**string**)** i**[**"number\_of\_copies"**];**

var id **=** **(**int**)** i**[**"id"**];**

var book **=** \_context**.**Books**.**Single**(**b **=>** b**.**Id **==** id**);**

book**.**Item\_call\_number **=** item\_call\_number**;**

book**.**Number\_of\_copies **=** number\_of\_copies**;**

book**.**Price **=** price**;**

book**.**Publisher\_code **=** publisher\_code**;**

book**.**Edition\_statement **=** edition\_statement**;**

**await** \_context**.**SaveChangesAsync**();**

**return** Ok**();**

**}**

Isječak koda 8. Primjer kontrolera za spremanje/ažuriranje podataka koristeći Entity Framework. [Autorski rad]

1. **Implementacija GraphQL-a**

Za implementaciju GraphQL-a u .NET programskom okviru prvo je potrebno odabrati koji GraphQL server će se koristiti. Trenutno najpopularnije opcije su GraphQL .NET i Hot Chocolate. Za ovaj projekt izabran je Hot Chocolate programski okvir koji je noviji i ima kraći razvojni ciklus (nova verzija svakih 17 dana, dok je razvojni ciklus GraphQL .NET-a 67 dana). Uz Hot Chocolate koristiti će se i Entity Framework. Hot Chocolate se dobro integrira s Entity Framework-om, što smanjuje količinu koda i omogućava brži razvoj.

* 1. Postavljanje GraphQL servera

Kako bismo pripremili Hot Chocolate GraphQL server, prvo je potrebno preuzeti određene NuGet pakete. Osim paketa vezanih uz Hot Chocolate, također je potrebno preuzeti i pakete koji omogućavaju korištenje Entity Framework-a.

Slika na kojoj se prikazuje tekst

Opis je automatski generiran

Slika . NuGet paketi korišteni u projektu. [Autorski rad]

Nakon preuzimanja i instalacije svih potrebnih paketa, GraphQL server treba i registrirati u početnoj datoteci projekta. U .NET 6 je ukinuto postojanje Startup.cs datoteke, te je njena funkcionalnost prebačena u Program.cs datoteku. Osim registracije korištenih GraphQL usluga, u Program.cs datoteci također je potrebno i mapirati GraphQL *endpoint*, tj. krajnju točku. Za razliku od REST-a, GraphQL koristi samo jednu krajnju točku na koju se šalju upiti. Toj točki se može pristupiti preko „*/graphql*“ URI-a. To je potrebno napraviti nakon pozivanja naredbe var app = builder.build();,ali prije pozivanja app.run();.

app**.**useEndpoints**(**endpoints **=>**

**{**

endpoints**.**MapGraphQL**();**

**});**

Isječak koda 9. Registracija GraphQL endpoint-a u Program.cs datoteci. [Autorski rad]

Također je potrebno registrirati korištenu DbContext klasu kao servis. Poželjno je proslijediti DbContextKind.Pooled kao argument pri registraciji usluge. Ovo omogućuje paralelno izvršavanje više upita nad bazom podataka. U ovom trenutku u razvoju, odsječak koda za registriranje usluga bi trebao izgledati kao na slici 13.

builder**.**Services

**.**AddGraphQLServer**()**

**.**RegisterDbContext**<**LibraryContext**>(**DbContextKind**.**Pooled**);**

Isječak koda 10. Registrirane GraphQL usluge pri postavljanju Hot Chocolate GraphQL server. [Autorski rad]

Iako je u teoriji ovo sve što je potrebno za postavljanje GraphQL servera, ovako postavljen server neće zapravo moći dohvaćati niti slati podatke. Za dohvaćanje podataka iz baze, prvo je potrebno postaviti Entity Framework i DbContext klasu (u ovom slučaju LibraryContext klasu), te implementirati klasu koja predstavlja GraphQL upite. Jedna od prednosti Hot Chocolate programskog okvira je to što podržava čak tri različita pristupa pisanju koda. Prvi je *annotation-based* pristup, koji koristi sposobnost .NET programskog okvira za pisanje anotacija u kodu. Ovakav način pisanja koda bi trebao biti poznat i prirodan svima koji su radili s Entity Framework-om. Drugi je tzv. *code-first* pristup. S ovim pristupom razvojni inženjer ima detaljan pregled koda koji se izvršava. Ovaj pristup koristi nasljeđivanje klasa i Fluent API. Treći pristup zove se *schema-first.* Ovaj pristup je iznimno sličan *annotation-based* pristupu, ali je više orijentiran pisanju koda prema već osmišljenoj GraphQL shemi. Svi primjeri navedeni u ovome radu pisani su koristeći *annotation-based* pristup.

* 1. Klase potrebne za izvršavanje upita

Kako bi GraphQL upit mogao izvršiti upit nad bazom, prvo je potrebno definirati tipove koje GraphQL shema sadržava [8]. Ovi tipovi su obične C# klase koje odražavaju tablice iz baze koje želimo otkriti klijentu. U njima se mogu koristiti Entity Framework podatkovne anotacije kako bi se olakšao rad s bazom. Nakon definiranja modela podataka, potrebno je isti koristeći Entity Framework povezati s bazom podataka koristeći DbSet klasu. Kada je to učinjeno, može se prijeći na definiranje Query klase. Query klasa sadržava metode za dohvaćanje podataka koristeći Entity Framework. Ove metode zovu se „*GraphQL resolvers*“. Bitno je napomenuti da je moguće napisati vlastiti *resolver* bez upotrebe Entity Framework-a, ali da je taj postupak znatno ubrzan i pojednostavljen korištenjem Entity Framework-a. Nakon što je Query klasa dovršena i registrirana kao usluga u Program.cs datoteci, moguće je izvršiti jednostavan GraphQL upit.

* 1. Dodatne mogućnosti GraphQL upita

Kako bi se omogućilo pisanje kompleksnijih GraphQL upita, Hot Chocolate nudi još nekoliko mogućnosti koje korisnik, ovisno o pretpostavljenim potrebama aplikacije, može ili ne mora koristiti. Te mogućnosti su paginacija, projekcija, filtriranje, sortiranje.

Paginacija u ovom kontekstu označava podjelu podataka na „stranice“, kako bi se ubrzalo dohvaćanje podataka i spriječio pad performansi u slučaju dohvata velike količine podataka. Hot Chocolate programski okvir čini dodavanje i korištenje paginacije u GraphQL-u iznimno jednostavnim. Naime, sve što je potrebno kako bi se mogla koristiti paginacija je dodati [UsePaging] anotaciju iznad metode Query klase na kojoj želimo vršiti paginaciju. Nakon što je omogućena paginacija na upitu, upit više neće direktno vraćati listu zapisa, nego će vratiti tzv. *Connection.* Tip *Connection* sadrži tipove *edges* i *pageInfo.* Tip *edges* sadrži listu čvorova (eng. *nodes*) koji sadržavaju tražene podatke i *cursor*. *PageInfo* sadrži informacije koje omogućavaju korisniku da vidi postoji li još stranica, te sadrži kursore za njihovo dohvaćanje [9]. Paginacija u Hot Chocolate-u također sadrži neke dodatne mogućnosti. Moguće je podesiti maksimalnu količinu podataka koja se može dohvatiti u jednom pozivu koristeći MaxPageSize svojstvo. Također se može odrediti zadana količina podataka koja će se dohvatiti ako se u upitu ne navede eksplicitno količina traženih podataka sa DefaultPageSize svojstvom. Paginacijom je također moguće dohvatiti cjelokupan broj podataka koje je moguće dohvatiti. Sve ove postavke mogu se postaviti na globalnoj razini ili samo za određeno polje.

Projekcija u kontekstu GraphQL-a se odnosi na spajanje tj. dohvaćanje podataka iz dvije ili više tablica baze podataka istovremeno. Projekcije su dio HotChocolate.Data NuGet paketa, te se on mora preuzeti kako bi se mogle koristiti. Potrebno je i registrirati korištenje projekcija u konfiguraciji sheme koristeći direktivu .AddProjections().Nakon toga omogućeno je korištenje [UseProjection] anotacije. Također je moguće koristiti [UseFirstOrDefault] anotaciju u slučaju potrebe da se rezultat upita ograniči na samo jedan rezultat. Projekciju je moguće koristiti s paginacijom, filtriranje i sortiranje, ali bitno je naglasiti da se anotacije koje naznačuju korištenje ovih funkcionalnosti moraju postaviti točno određenim redoslijedom. [10]

**[**UsePaging**(**MaxPageSize **=** 1001**)]**

**[**UseProjection**]**

**[**UseFiltering**]**

**[**UseSorting**]**

**public** IQueryable**<**Book**>** GetBook**([**ScopedService**]** LibraryContext context**)** **{**

**return** context**.**Books**;**

**}**

Isječak koda 11. Nužan redoslijed anotacija na metodi koja koristi paginaciju, projekciju, filtriranje i sortiranje. [Autorski rad]

Kako bi se omogućilo filtriranje u Hot Chocolate-u, potrebno je prvo preuzeti *HotChocolate.Data* NuGet paket. Nakon toga potrebno je dodati direktivu .AddFiltering() u konfiguraciju sheme. Tada je moguće koristiti [UseFiltering] anotaciju nad Query metodama koje se trebaju filtrirati. Kako bi filtriranje podataka ispravno radilo, metoda mora vraćati IQueryable<T> ili IEnumerable<T> sučelje. Filtriranje je također moguće prilagoditi vlastitim željama nasljeđivanjem FilterInputType klase i korištenjem Fluent API-a.[11]

Sortiranje je također dio *HotChocolate.Data* paketa. Kao i filtriranje i paginaciju, i njega je potrebno registrirati u konfiguraciji sheme koristeći .AddSorting()direktivu, te se koristi pomoću [UseSorting] anotacije. Sortiranje u Hot Chocolate programskom okviru prema zadanim postavkama radi s IQueryable sučeljem, ali ga se može konfigurirati da radi s drugim sučeljima. Hot Chocolate će sam automatski pregledati modele koji se koriste i iz njih zaključiti moguće operacije filtriranja koje su na raspolaganju. Također je moguće napraviti vlastiti prilagođeni način na koji način filtriranje funkcionira koristeći Fluent API i tzv. *sorting conventions.*[12]

builder**.**Services

**.**AddGraphQLServer**()**

**.**RegisterDbContext**<**LibraryContext**>(**DbContextKind**.**Pooled**)**

**.**AddQueryType**<**Query**>()**

**.**AddProjections**()**

**.**AddFiltering**()**

**.**AddSorting**();**

Isječak koda 12. Konfiguracija sheme kao servisa nakon dodavanja Query.cs klase, te projekcije, sortiranja i filtriranja [Autorski rad]

* 1. GraphQL mutacije

Upiti u GraphQL služe samo kao mehanizam za dohvaćanje podataka sa servera. Kada serveru treba poslati podatke, koriste se tzv. mutacije. Mutacije omogućavaju dodavanje i izmjenu zapisa u bazi. Kako bi se mutacije koristile u Hot Chocolate-u, nužno je stvoriti klasu koja će predstavljati mutaciju, u ovom slučaju Mutation.cs, i registrirati je kao servis pozivajući direktivu .AddMutationType<Mutation>() na *builder.Services*. Metode unutar klase Mutation su slične onima unutar Query.cs klase, ali im se osim konteksta baze podataka prosljeđuju podatci koje se spremaju u bazu. U GraphQL-u postoje neke konvencije pisanja mutacija. Ove konvencije su podržane od strane Hot Chocolate-a.[13] Tako se kao povratna vrijednost vraća izmijenjeni objekt, koji se zove *Payload.* Također se za unošenje podataka koriste posebni objekti umjesto da se svaka varijabla šalje zasebno. Ovo omogućava skalabilnost i lakše dodavanje ili uklanjanje varijable iz GraphQL sheme.

**[**UseDbContext**(typeof(**LibraryContext**))]**

**public** **async** Task**<**AddBookPayload**>** AddBookAsync**(**AddBookInput input**,** **[**ScopedService**]** LibraryContext context**)** **{**

var book **=** **new** Book **{**

Title **=** input**.**Title**,**

Year **=** input**.**Year**,**

AuthorId **=** input**.**AuthorId**,**

Isbn **=** input**.**Isbn**,**

Price **=** input**.**Price**,**

Edition\_statement **=** input**.**Edition\_statement**,**

Item\_call\_number **=** input**.**Item\_call\_number**,**

Publisher\_code **=** input**.**Publisher\_code**,**

Number\_of\_copies **=** input**.**Number\_of\_copies

**};**

context**.**Books**.**Add**(**book**);**

**await** context**.**SaveChangesAsync**();**

**return** **new** AddBookPayload**(**book**);**

**}**

Isječak koda 13. Primjer mutacije koja koristi Entity Framework za spremanje u bazu. [Autorski rad]

1. **Sigurnost GraphQL-a**

Glavna razlika između GraphQL-a i REST-a je to što su API pozivi u REST-u strogo definirani, dakle točno se zna koji se podatci šalju i dohvaćaju kojim pozivom, te tko im može pristupiti. Velika prednost GraphQL-a je to što frontend razvojni inženjeri mogu samostalno dohvatiti podatke koji im trebaju, ali ovo je također i najveća sigurnosna prijetnja serveru koji koristi GraphQL. Mogućnost pisanja vlastitih upita od strane razvojnih inženjera otvara put zloupotrebi raznim vrstama napada. Najosnovniji način za napasti GraphQL server je pisanjem kompleksnih upita koji zagušuju i usporavaju server. Također je moguće navigacijom po grafu doći do podataka kojima korisnik ne bi trebao imati pristup. Zbog toga su se u GraphQL-u morali razviti posebni oblici zaštite od napada specifičnih za GraphQL. Najjednostavniji od njih su *Execution timeout* i *Pagination amount,* dok su neki kompleksniji *Max query depth*, *Max query complexity*, te autentikacija i autorizacija.

* 1. *Execution timeout* i *Pagination amount*

*Execution timeout* se odnosi na vrijeme nakon kojega se upit prestaje izvršavati. Zadani *execution timeout* u Hot Chocolate serveru je 30 sekundi, ali moguće je postaviti vlastito vrijeme nakon kojeg se upit smatra prekompleksnim. Mana ovoga pristupa je što se upit pokušava izvršiti sve dok ne istekne određeno vrijeme. Zbog toga ipak može doći do usporavanja rada servera i barem djelomičnog uspjeha napada.

*Pagination amount* se zapravo odnosi na opciju u GraphQL-u pomoću koje se može odrediti maksimalan broj zapisa koji se mogu dohvatiti u jednom pozivu. Ako je omogućena paginacija npr. na upitu za dohvaćanje svih knjiga, te potom postavljen *pagination amount* na 100, maksimalna količina knjiga koja se može dohvatiti u jednom pozivu je 100 knjiga. Ova tehnika štiti server na način da smanjuje maksimalnu količinu podataka koja se može dohvatiti i stoga otežava pisanje malicioznih upita.[14]

* 1. *Max query depth* i *Max query complexity*

Maksimalna dubina upita i maksimalna kompleksnost upita su dva vrlo slična načina obrane od malicioznih upita. Maksimalna dubina upita je relativno jednostavan obrambeni mehanizam. Pri registriranju GraphQL servisa poziva se direktiva .AddMaxExecutionDepthRule() kojoj se kao parametri proslijede integer koji određuje maksimalnu dozvoljenu dubinu i boolean *true.* Ako je poznato da se u GraphQL shemi svi podatci mogu dohvatiti kroz upit čija dubina ne prelazi 4, onda je lako zaključiti da su svi upiti koji prelaze ovu dubinu maliciozni.

Maksimalna kompleksnost upita je konceptualno jako slična maksimalnoj dubini upita, ali u praksi se implementira dosta drugačije. Kako bi se omogućilo korištenje maksimalne kompleksnosti upita, potrebno je pri registraciji servisa napisati kod kao na isječku koda 15. Kao što je vidljivo, ovaj mehanizam ima nešto više opcija od maksimalne dubine upita. Moguće je odrediti maksimalnu kompleksnost upita. Također je moguće promijeniti zadane postavke kompleksnosti. Naime, prema zadanim postavkama svaka operacija ima kompleksnost 1, dok svaki upit koji poziva *resolver* ima kompleksnost 5.

builder**.**Services

**.**AddGraphQLServer**()**

**.**ModifyRequestOptions**(**o **=>**

**{**

o**.**Complexity**.**Enable **=** **true;**

o**.**Complexity**.**MaximumAllowed **=** 300000**;**

o**.**Complexity**.**ApplyDefaults **=** **true;**

o**.**Complexity**.**DefaultComplexity **=** 2**;**

o**.**Complexity**.**DefaultResolverComplexity **=** 3**;**

**})**

Isječak koda 14. Primjer koda koji omogućava korištenje maksimalne kompleksnosti upita. [Autorski rad]

Za svaki upit se mogu uvesti i multiplikatori koji povećavaju kompleksnost upita ovisno o količini traženih podataka. Dok se ovaj koncept može činiti prilično jednostavan na prvu, lako je vidjeti kako izračunavanje odgovarajuće maksimalne složenosti upita može postati iznimno složeno što više upita imamo u GraphQL shemi. Bitno je napomenuti da ove dvije tehnike uzimaju u obzir kako upite, tako i mutacije. Jedna od najvećih prednosti maksimalne dubine i složenosti upita u odnosu na *pagination amount* i *execution timeout* je u tome što se upiti analiziraju prije izvođenja. Dakle, ako je upit prekompleksan ili predubok, on se neće uopće početi izvršavati. U tom slučaju server vraća odgovor s odgovarajućom porukom o grešci.[15]



Slika . Lijevo: primjer kompleksnog upita. Desno: poruka o pogrešci. [Autorski rad]

* 1. Autentikacija i autorizacija

Autentikacija ili provjera autentičnosti korisnika nam dozvoljava da odredimo korisnikov identitet i preduvjet je za autorizaciju. Također je korisna ako želimo u shemi dodati polje koje dozvoljava provjerenom korisniku da dohvati podatke o sebi. Hot Chocolate u potpunosti prihvaća mogućnosti provjere autentičnosti ASP.NET Core-a, što olakšava ponovnu upotrebu postojeće konfiguracije provjere autentičnosti i integrira različite pružatelje provjere autentičnosti [16]. U E-Library aplikaciji je implementirana autorizacija pomoću *JSON Web Token*-a.

*JSON Web Token* (JWT) je kompaktan i samostalan način sigurnog prijenosa informacija. Namijenjen je za korištenje pri autentikaciji i autorizaciji korisnika prilikom prijave na razne servise. Glavne karakteristike su mu jednostavnost korištenja i sigurnost.[17] JWT se sastoji od zaglavlja, sadržaja i potpisa. Ova 3 dijela se pretvaraju u string koji predstavlja kodirani JWT koji se može dekodirati kako bi se provjerila autentičnost korisnika. [18] U sadržaju JWT-a je moguće poslati zaštićene podatke i upravo ti podatci se koriste za autorizaciju GraphQL upita u E-Library web aplikaciji. Primjer metode za generiranje JWT-a korišten u E-Library aplikaciji može se vidjeti na isječku koda 16. Nakon što je JWT generiran, potrebno ga je poslati u „*Authorization*“ zaglavlju API poziva.

**public** static string GetJWTAuthKey**(**User user**)** **{**

var securityKey **=** **new** SymmetricSecurityKey**(**Encoding**.**UTF8**.**GetBytes**(**"MySuperSecretKey"**));**

var credentials **=** **new** SigningCredentials**(**securityKey**,** SecurityAlgorithms**.**HmacSha256**);**

var claims **=** **new** List**<**Claim**>();**

claims**.**Add**(new** Claim**(**"role"**,** user**.**Role**));**

var jwtSecurityToken **=** **new** JwtSecurityToken**(**

issuer**:** "https://localhost:5000"**,**

audience**:** "https://localhost:5000"**,**

expires**:** DateTime**.**Now**.**AddMinutes**(**30**),**

signingCredentials**:** credentials**,**

claims**:** claims

**);**

**return** **new** JwtSecurityTokenHandler**().**WriteToken**(**jwtSecurityToken**);**

**}**

Isječak koda 15. Primjer metode za generiranje JSON Web Tokena koji zapisuje uloge i ističe nakon 30 minuta. [Autorski rad]

Kako bi bilo moguće koristiti JWT-e potrebno je preuzeti *Microsoft.AspNetCore.Authentication.JwtBearer* NuGet paket. Ovo je obični ASP NET Core paket za autentikaciju. Nakon toga potrebno je postaviti autentikacijsku shemu kao što je prikazano na isječku koda 17.

builder**.**Services**.**AddAuthentication**(**JwtBearerDefaults**.**AuthenticationScheme**)**

**.**AddJwtBearer**(**options **=>**

**{**

SecurityKey signingKey **=** **new** SymmetricSecurityKey**(**

Encoding**.**UTF8**.**GetBytes**(**"MySuperSecretKey"**));**

options**.**TokenValidationParameters **=**

**new** TokenValidationParameters **{**

ValidIssuer **=** "https://localhost:5000"**,**

ValidAudience **=** "https://localhost:5000"**,**

ValidateIssuerSigningKey **=** **true,**

IssuerSigningKey **=** signingKey

**};**

**});**

builder**.**Services**.**AddAuthorization**();**

Isječak koda 16. Primjer registracije JWT autentikacijske sheme. [Autorski rad]

Također je potrebno pozvati app.UseAuthentication();. Time je omogućeno korištenje JWT tokena u .NET aplikaciji. Nakon toga je potrebno dodati autorizaciju u Hot Chocolate. Prvo je potrebno preuzeti *HotChocolate.AspNetCore.Authorization* paket. Nakon toga treba pozvati direktivu .AddAuthorization() pri registraciji servisa. Ovo omogućuje pristup identitetu provjerenog korisnika*.*

Autorizacija omogućava provjeru korisnikovih dopuštenja u sustavu. Koristeći autorizaciju moguće je ograničiti pristup određenim resursima ili dozvoliti samo određenim korisnicima da izvršavaju određene mutacije.[19] Kako bismo postavili autorizaciju u Hot Chocolate-u, moramo preuzeti *HotChocolate.AspNetCore.Authorization* NuGet paket. Također je potrebno dva puta pozvati services.AddAuthorization(); metodu: prvi put kako bi se omogućila autorizacija u .NET-u, drugi put kako bi se omogućila u Hot Chocolate-u. Potrebno je i pozvati app.UseAuthorization();. Nakon toga moguće je koristiti [Authorize] anotaciju kako bi se ograničio pristup određenim resursima. Anotaciju [Authorize] može se koristiti iznad *resolvera* kojem se želi ograničiti pristup, ali i u klasi modela. Korištenjem autentikacije u suradnji sa sesijom moguće je riješiti jedan od sigurnosnih problema specifičnih za GraphQL. U aplikaciji je trenutno moguće dohvatiti podatke o svim posudbama pomoću upita za posudbe. U tom pozivu potrebno je napraviti filter koji će dohvaćati samo podatke trenutno ulogiranog korisnika. To se može implementirati korištenjem sesije. Kako bismo koristili sesiju u .NET-u, potrebno je u Program.cs klasi injektirati sesije na sljedeći način:

builder**.**Services**.**AddSingleton**<**IHttpContextAccessor**,** HttpContextAccessor**>();**

builder**.**Services**.**AddScoped**<**UsersController**>();**

builder**.**Services**.**AddScoped**<**Query**>();**

builder**.**Services**.**AddSession**(**options **=>**

**{**

option**.**IdleTimeout **=** TimeSpan**.**FromSeconds**(**1000**);**

options**.**Cookie**.**HttpOnly **=** **true;**

options**.**Cookie**.**IsEssential **=** **true;**

**});**

Isječak koda 17. Primjer koda potrebnog za upotrebu sesije. [Autorski rad]

U Program.cs klasi je također potrebno pozvati app.UseSession();. Tada je moguće pri prijavi korisnika u sesiju postaviti njegov id ili guid. Pri tome je potrebno u svakoj klasi u kojoj se sesija koristiti napraviti injekciju sesije kao što je napravljeno na isječku koda 18. Nakon toga moguće je pristupiti id-u prijavljenog korisnika i prema njemu filtrirati sve podatke koji se dohvaćaju iz baze kao što je demonstrirano na isječku koda 20. Sesiju je dobro koristiti s GraphQL autorizacijom, kako bi samo ulogirani korisnik imao pristup upitu.

**private** **readonly** IHttpContextAccessor \_httpContextAccessor**;**

**private** **readonly** ISession \_session**;**

**public** Query**(**IHttpContextAccessor httpContextAccessor**)** **{**

\_httpContextAccessor **=** httpContextAccessor**;**

\_session **=** \_httpContextAccessor**.**HttpContext**.**Session**;**

**}**

*Isječak koda 18. Primjer metode za dohvaćanje podataka o svim korisnicima koja je autorizirana samo za korisnika s ulogom administratora. [Autorski rad]*

Međutim, to neće pomoći u situaciji gdje se posudbe dohvaćaju kao element knjiga, već samo pri direktom pozivu za dohvaćanje posudbi. To je još jedan od načina na koji se u GraphQL-u mogu izložiti ranjivi podatci. Navigiranjem po čvorovima grafa korisnik može ostvariti pristup podatcima koji mu ne pripadaju. Na primjer, korisnik može ostvariti pristup podatcima drugih korisnika koristeći upit za dohvaćanje svih knjiga. To se ostvaruje pisanjem upita koji poziva *Books* > *Loans* > *User* . S tim pozivom korisnik vidi podatke svih korisnika koji su izvršili posudbu. Nažalost, u Hot Chocolate implementaciji GraphQL-a ne postoji „*out of the box*“ rješenje kojim se može zaštititi od ovakvog upita te ovaj problem zasad ostaje otvoren. No, to ne znači da je GraphQL inherentno nesiguran, već da je došlo do greške pri dizajniranju GraphQL sheme. Naime, mogućnost dohvaćanja posudbi preko knjige ne dodaje nikakvu novu funkcionalnost aplikaciji, dapače, redundantna je. Obični korisnik već ima funkcionalnost pregleda svih svojih posudbi pomoću *Loans* upita, a nema razloga da obični korisnik može vidjeti tuđe posudbe. Ako je potrebno omogućiti nekom korisniku, na primjer administratoru, da vidi sve posudbe može se implementirati zaseban *resolver* koji koristi autorizaciju za ulogu administratora ili se može dozvoliti administratoru da vidi posudbe kroz *Books* *resolver*. U svakom slučaju, ovo je vrijedan primjer kako treba pažljivo pristupiti oblikovanju GraphQL sheme, te uzeti u obzir sve funkcionalnosti koje je potrebno implementirati i koji je najbolji način njihove implementacije. U suprotnom, dolazi do otkrivanja potencijalno ranjivih podataka prema korisnicima koji te podatke mogu zloupotrijebiti. Što se ostalih vrsta sigurnosnih rizika tiče, GraphQL nudi mnoge alate i mehanizme za zaštitu podataka i samoga servera, a najmoćniji među njima su autentikacija i autorizacija. Ako korisnik pokuša neautorizirano pristupiti resursu koji je zaštićen autorizacijom, dobiva odgovor kao na slici 4. Potrebno je naglasiti da u kodu, kako u modelima tako i u Query klasi, treba koristiti Hot Chocolate [Authorize] anotaciju, s obzirom da ista postoji i u .NET-u. Prilikom pokušaja neautoriziranog korisnika da dohvati podatke koji zahtijevaju autorizaciju, server vraća poruku pogreške.

Slika na kojoj se prikazuje tekst

Opis je automatski generiran

Slika . Pokušaj pristupa podacima kojima korisnik nema pristup. [Autorski rad]

Zasad se autorizacija svodila na dodjeljivanje JWT-a provjerenom korisniku, međutim to nije uvijek dovoljno. Na primjer, svaki ulogirani korisnik mora imati pristup popisu vlastitih posudbi knjiga, no svaki korisnik ne smije imati pristup popisu svih posudbi ili podatcima o drugim korisnicima. Stoga u GraphQL-u postoje i uloge, koje su također podržane od strane Hot Chocolate-a. Dopuštanje pristupa određenom resursu od strane korisnika koji posjeduju određene uloge je iznimno jednostavno u Hot Chocolate-u. Jednostavno se u [Authorize] anotaciji proslijede uloge koje joj mogu pristupiti. Kako bi se odredila uloga koju provjereni korisnik ima, potrebno je u tablicu *Users* u bazi podataka dodati polje *Role*. Prilikom prijave korisnika ovo se polje dohvaća i prosljeđuje metodi za generiranje JWT-a, koja čita korisnikovu ulogu i zapisuje je u JWT. Prilikom pokušaja pristupanju ograničenom resursu ta uloga se uspoređuje s popisom uloga kojima je dopušteno pristupiti tom resursu, te se u korisniku dopušta pristup u slučaju da ima odgovarajuću ulogu. Treba napomenuti da, ako je definirano više uloga koje mogu pristupiti resursu, korisnik mora posjedovati samo jednu od njih kako bi pristupio resursu.

**[**Authorize**(**Roles **=** **new[]** **{** "admin" **})]**

**[**UsePaging**(**MaxPageSize **=** 1001**)]**

**[**UseProjection**]**

**[**UseFiltering**]**

**public** IQueryable**<**User**>?** GetUser**([**ScopedService**]** LibraryContext context**)** **{**

**return** context**.**Users**;**

**}**

Isječak koda 19. Primjer metode za dohvaćanje podataka o svim korisnicima koja je autorizirana samo za korisnika s ulogom administratora. [Autorski rad]

U isječku koda 20 vidljiv je *resolver* kojem mogu pristupiti samo autorizirani korisnici, ali koji ne provjerava ulogu korisnika. Ovo je napravljeno kako bi svaki korisnik mogao pristupiti svojim vlastitim podatcima. S obzirom da *resolveri* dopuštaju da se u njima piše vlastita logika, također bi bilo moguće koristeći autentikaciju i sesiju unutar *resolvera* provjeriti jesu li podatci provjerenog korisnika jednaki podatcima koji se pokušavaju dohvatiti, te spriječiti pristup podatcima ako nisu.

**public** IQueryable**<**Loan**>** GetLoan**([**ScopedService**]** LibraryContext context**)** **{**

var user\_id **=** \_session**.**GetInt32**(**"userid"**);**

var data **=** context**.**Loans**.**Where**(**e **=>** e**.**User\_id **==** user\_id**);**

**return** data**;**

**}**

Isječak koda 20. Primjer resolvera dohvaća podatke o korisnikovim posudbama. [Autorski rad]

Osim uloga, u Hot Chocolate omogućava i korištenje tzv. *policies* za autorizaciju korisnika. *Policies* tj. preduvjeti za pristupanje resursu, funkcioniraju na isti način kao i uloge, ali s jednom ključnom razlikom. Potrebno ih je definirati, nakon čega se uspoređuju određeni korisnikovi podatci s preduvjetima potrebnim za pristup resursu. To na primjer može biti država porijekla korisnika, njegova dob, itd. Glavna razlika između uloga i preduvjeta pristupa resursu je to što korisnik mora ispuniti sve preduvjete kako bi mu bio dozvoljen pristup resursu, dok je dovoljno da ima samo jednu od navedenih uloga kako bi pristupio resursu.

Tijekom implementacije sigurnosnih značajki GraphQL-a testirali smo aplikaciju na razne sigurnosne rizike te smo iste otklonili. Iz provedenog testiranja je vidljivo da GraphQL posjeduje sve potrebne alate i funkcionalnosti kako bi se osigurao od napada. Među njima najvrijedniji su autentikacija i autorizacija koje štite podatke aplikacije, dok su za zaštitu od DOS (eng. *Denial Of Service* / uskraćivanje resursa) napada najefikasnije metode zaštite maksimalna dubina upita i maksimalna kompleksnost upita, dok se maksimalno vrijeme izvođenja upita koristi kao posljednja linija obrane.

# **Usporedba performansi GraphQL-a i REST API-a**

U ovom poglavlju usporediti će se brzina dohvaćanja i slanja podataka putem GraphQL-a i REST-a. S obzirom da za svaku stranicu u E-Library aplikaciji postoji identična, s jedinom razlikom u tome koja se metoda dohvaćanja/slanja podataka koristi, testovi su se vršili na E-Library aplikaciji. Uspoređivalo se dohvaćanje istih podataka GraphQL-om i REST-om. U svim pozivima na kojima se izvodilo mjerenje korišten je Entity Framework kao konekcija na bazu. Mjerenje se odvijalo na dva načina: u kodu i u Chrome DevTools alatu.

U kodu se za mjerenje koristi performance.now() metoda iz *Performance* API-a. Pozivanjem ove metode direktno prije i nakon pozivanja fetch() metode u pomoćnim funkcijama dobiti će se vrijeme trajanja poziva u milisekundama. Potrebno je naglasiti kako je preciznost performance.now() funkcije smanjena u nekim web preglednicima kako bi se izbjegle određene vrste napada.[19]

Osim Performance API-a koristi se i kartica s podatcima o mreži u Chrome DevTools alatu. Ova kartica zapisuje svaki poziv napravljen preko mreže, te sadrži mnoštvo podataka o njemu, kao na primjer brzinu izvršavanja poziva i količinu podataka koji su preneseni.

Mjerile su se performanse API poziva za dohvaćanje prvih tisuću knjiga iz baze podataka, poziva za dohvaćanje svih knjiga koje je korisnik posudio, te poziva za spremanje podataka već postojeće knjige.

* 1. Rezultati mjerenja

Prvo provedeno mjerenje se vršilo na API pozivu koji dohvaća podatke o 1000 knjiga iz baze. Rezultati mjerenja Performance API-em i Chrome DevTools alatom mogu se vidjeti na slikama 5 i 6. Iz grafova je jasno vidljivo da REST brže dohvaća podatke od GraphQL-a. Bitno je istaknuti da je prvo dohvaćanje podataka u GraphQL-u uvijek znatno sporije od uzastopnih poziva. Međutim, ovaj problem je moguće zaobići pozivanjem direktive .InitializeOnStartup() pri registraciji GraphQL sheme. Ova direktiva stvara shemu i šalje testni zahtjev na GraphQL server, što pomaže performansama.[21] Unatoč tome, GraphQL u prosjeku i dalje pokazuje lošije rezultate.

Slika . Podatci od dohvaćanju veće količine podataka, mjereno pomoću Performance API-a [Autorski rad]

Slika . Podatci od dohvaćanju veće količine podataka, mjereno pomoću Chrome DevTools. [Autorski rad]

Drugi set mjerenja obavljen je na API pozivu za dohvaćanje podataka o posudbama na korisniku koji je imao 10 posudbi. Iako ni u jednom od testnih slučajeva vrijeme dohvaćanja podataka nije dovoljno veliko da bi bilo osjetno u praksi, može se primijetiti da i ovdje GraphQL pokazuje znatno lošije rezultate od REST-a, s prosječnim vremenima izvođenja od 19,725 milisekundi i 13,9 milisekundi, naspram 17,259 ms i 2,1ms izmjerenima pri korištenju REST-a (slike 7 i 8).

Slika . Podatci od dohvaćanju manje količine podataka, mjereno pomoću Performance API-a. [Autorski rad]

Slika . Podatci od dohvaćanju manje količine podataka, mjereno pomoću Chrome DevTools. [Autorski rad]

Posljednji test se vršio na pozivu za ažuriranje podataka jedne knjige. Prilikom svakog ažuriranja slala se jednaka količina podataka. Usprkos tome sadržaj GraphQL poziva je bio neznatno veći, što je prouzrokovano činjenicom da se u tijelu GraphQL zahtjeva šalje cijeli upit, dok REST zahtjev u tijelu sadrži samo podatke koje treba spremiti u bazu. Rezultati testova vidljivi su na slikama 9 i 10. Još jednom, REST se pokazuje primjetno bržim od GraphQL-a u izvršavanju operacija nad serverom.

Slika . Podatci od slanju manje količine podataka na server, mjereno pomoću Performance API-a. [Autorski rad]

Slika . Podatci od slanju manje količine podataka na server, mjereno pomoću Chrome DevTools. [Autorski rad]

Ovi rezultati nisu pretjerano iznenađujući ako se uzme u obzir da su slične studije već rađene, te da su pokazale slične rezultate. U svom radu iz 2018., Hartina, D. A., et al, navode da je REST konzistentno stabilniji od GraphQL-a[22]. Seabra, M., et al, u radu iz 2019. zaključuju da REST i GraphQL imaju slične performanse, osim kad je broj zahtjeva u sekundi iznad 3000, kada mu performanse znatno opadaju[23]. Studija iz 2020. godine od rađena od strane Erlandsson, P. i Remes, J. navodi da GraphQL ima gore performanse ne samo od REST-a, već i od SOAP-a [24].

Usprkos tome, rezultate ovih mjerenja treba oprezno tumačiti kako bi se izbjeglo donošenje pogrešnih zaključaka. Naime, testiranje je rađeno na iznimno maloj aplikaciji sa REST krajnjim točkama koje dohvaćaju točno one podatke koji su potrebni na *frontendu*. GraphQL nije stvoren kao odgovor na taj problem, već je nastao kao odgovor na situaciju u kojoj u velikom sustavu ne postoji nijedna krajnja točka koja dohvaća točno određene potrebne podatke, već je potrebno dohvatiti previše ili premalo podataka. Također je bitno obratiti pažnju na ono što se nalazi ispod površine. Pri korištenju višestrukih REST poziva ne utječe se samo na performanse servera, već se vrše i višestruki pozivi na bazu. S druge strane, upit pisan GraphQL-om će izvršiti samo jedan upit na bazu. Zbog tih razloga GraphQL još uvijek ostaje primjenjivo rješenje u razvoju web aplikacija, pogotovo u razvoju velikih sustava, gdje su timovi koji rade na *frontend*-u i *backend*-u potpuno odvojeni, te imaju malo doticaja jedan s drugim.

# **Zaključak**

U ovom radu prikazana je implementacija GraphQL-a u .NET 6 programskom okruženju koristeći Hot Chocolate programski okvir. Detaljno su razrađene funkcionalnosti GraphQL upita i mutacija. Obrađena je tematika sigurnosti GraphQL-a i uspoređene su njegove performanse s REST-om.

Iz rezultata prikazanih u radu vidljivo je zašto je GraphQL sve popularnija tehnologija. Usprkos svojim nedostatcima, GraphQL nalazi svoje mjesto u skupu vještina sve više razvojnih inženjera, ne kao zamjena za REST, nego kao alternativna opcija. Glavna prednost GraphQL-a nad REST-om nije povećanje performansi aplikacije, već omogućavanje bržeg razvoja aplikacija s minimalnim utjecajem na performanse. Stoga ne treba slijepo pratiti trendove i koristiti GraphQL jer je najnovija tehnologija, već treba razmotriti opseg i potrebe aplikacije i odabrati najbolju opciju prema vlastitim potrebama. GraphQL najbolje funkcionira kada je korišten od strane velikih timova gdje *frontend* i *backend* timovi nemaju doticaja jedan s drugim. I dok je početno vrijeme postavljanja GraphQL-a veće od onog potrebnog za postavljanje REST API-a, to se vrijeme, zahvaljujući skalabilnosti GraphQL-a, vrlo uspješno nadoknadi prilikom naknadnog proširivanja API-a. GraphQL je relativno jednostavan jezik, te uz malo truda i istraživanja može postati vrijedan alat svakom razvojnom inženjeru. Kao i sa svakom novom tehnologijom, potrebno je provesti neko vrijeme svladavajući njene izazove, konvencije i zahtjeve. GraphQL predstavlja neke jedinstvene izazove u svom korištenju, ponajviše u području postavljanja GraphQL servera, sigurnosti servera i korisničkih podataka, te zahtijeva odstupanje od klasičnog načina razmišljanja koji povezujemo s razvojem web aplikacija. Najveća prepreka učenju GraphQL-a je to što se on, kao nova tehnologija, još uvijek često mijenja i evoluira. I dok su većina ovih promjena dobrodošli dodatci GraphQL-u, one otežavaju intimno poznavanje jezika. Ovo je također razlog radi kojega je teško pronaći materijale i vodiče koji se bave implementacijom GraphQL-a u nekim specifičnim razvojnim okruženjima, što dodatno otežava upoznavanje s jezikom. Unatoč tome, GraphQL je tehnologija koja postaje sve raširenija te je radi toga apsolutno vrijedna truda potrebnog za njeno svladavanje.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Popis literature:

[1] GraphQL Foundation, „GraphQL“ <https://spec.graphql.org/October2021/> , (Preuzeto: 28.05.2022).

[2] ChilliCream , „Introduction | ChilliCream GraphQL Platform“ <https://chillicream.com/docs/hotchocolate> , (Preuzeto: 28.05.2022).

[3] RedHat, „What is GraphQL?“, <https://www.redhat.com/en/topics/api/what-is-graphql> ,

(Preuzeto: 16.06.2022)

[4] ApolloGraphQL, „What is GraphQL? GraphQL introduction - Apollo GraphQL Blog“, https://www.apollographql.com/blog/graphql/basics/what-is-graphql-introduction/?\_ga=2.202547158.673307776.1655301254-1847066418.1655301254 , (Preuzeto: 16.06.2022)[5] SmartBear, „State of API 2020 Report | SmartBear“, https://smartbear.com/resources/ebooks/the-state-of-api-2020-report/ , (2020)., (Preuzeto: 13.06.2022)[6] MDN, „Fetch API - Web APIs | MDN“ <https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API> , (Preuzeto: 29.05.2022).

[7] HCL Software, „Business logic layer“, <https://help.hcltechsw.com/commerce/7.0.0/com.ibm.commerce.developer.doc/concepts/csdbusinesslogicbase.html>, (Preuzeto: 30.05.2022).

[8] M. Masse, (2011)., *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*, str.5, O'Reilly Media, Inc.

[9] Microsoft, „Create web APIs with ASP.NET Core | Microsoft Docs“, <https://docs.microsoft.com/en-us/aspnet/core/web-api/?view=aspnetcore-6.0#attribute-routing-requirement> , (Preuzeto: 30.05.2022).

[10] ChilliCream , „Get started with Hot Chocolate | ChilliCream GraphQL Platform“,

<https://chillicream.com/docs/hotchocolate/get-started> , (Preuzeto: 31.05.2022).

[11] ChilliCream , „Pagination | ChilliCream GraphQL Platform“, <https://chillicream.com/docs/hotchocolate/fetching-data/pagination> ,(Preuzeto: 01.06.2022).

[12] ChilliCream , „Projections | ChilliCream GraphQL Platform“, <https://chillicream.com/docs/hotchocolate/fetching-data/projections> ,(Preuzeto: 01.06.2022).

[13] ChilliCream , „Filtering | ChilliCream GraphQL Platform“, <https://chillicream.com/docs/hotchocolate/fetching-data/filtering> , (Preuzeto: 01.06.2022).

[14] ChilliCream , „Sorting | ChilliCream GraphQL Platform“, <https://chillicream.com/docs/hotchocolate/fetching-data/sorting> , (Preuzeto: 01.06.2022).

[15] ChilliCream , „Mutations | ChilliCream GraphQL Platform“, <https://chillicream.com/docs/hotchocolate/defining-a-schema/mutations> , (Preuzeto: 05.06.2022).

[16] ChilliCream , „Security | ChilliCream GraphQL Platform“, <https://chillicream.com/docs/hotchocolate/v10/security/security> , (Preuzeto: 05.06.2022).

[17] ChilliCream , „Operation Complexity | ChilliCream GraphQL Platform“, <https://chillicream.com/docs/hotchocolate/security/operation-complexity> , (Preuzeto: 05.06.2022).

[18] ChilliCream , „Authentication | ChilliCream GraphQL Platform“, <https://chillicream.com/docs/hotchocolate/security/authentication> , (Preuzeto: 05.06.2022).

[19] A. Sabol, (2019). „Autentikacija i autorizacija pomoću Json web tokena“, str.1, *University of Pula / Sveučilište Jurja Dobrile u Puli*

[20] A. Sabol, (2019). „Autentikacija i autorizacija pomoću Json web tokena“, str.6, *University of Pula / Sveučilište Jurja Dobrile u Puli*

[21] ChilliCream, „Authorization | ChilliCream GraphQL Platform“, <https://chillicream.com/docs/hotchocolate/security/authorization> , (Preuzeto: 06.06.2022).

[22] MDN, „performance.now() - Web APIs | MDN“ <https://developer.mozilla.org/en-US/docs/Web/API/Performance/now> , (Preuzeto: 06.05.2022).

[23] ChilliCream, „Performance | ChilliCream GraphQL Platform“, <https://chillicream.com/docs/hotchocolate/performance> , (Preuzeto: 07.06.2022).

[24] D.A. Hartina, A. Lawi, B.L.E. Panggabean, (2018)., „Performance Analysis of GraphQL and RESTful in SIM LP2M of the Hasanuddin University“, str. 240, *Proceedings - 2nd East Indonesia Conference on Computer and Information Technology: Internet of Things for Industry, EIConCIT 2018*, DOI: 10.1109/EICONCIT.2018.8878524

[25] M. Seabra, M.F. Nazário, G. Pinto, (2019)., „REST or GraphQL? A performance comparative study“, *SBCARS '19: Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse* <https://doi.org/10.1145/3357141.3357149>

[26] Erlandsson, P. i Remes, J., (2020)., „Performance comparison : Between GraphQL, REST & SOAP, Dissertation“, str. 36, *Diva Portal*, URN: urn:nbn:se:his:diva-18713