Contents

[GraphQL 2](#_Toc138080092)

[Type objekti 2](#_Toc138080093)

[Query i Mutation objekti 4](#_Toc138080094)

[Spring for GraphQL 8](#_Toc138080095)

[N+1 problem 10](#_Toc138080096)

[Kako bi koristili GraphQL u našem sistemu? 13](#_Toc138080097)

[OpenFeign 13](#_Toc138080098)

[MapStruct 14](#_Toc138080099)

[Zašto koristiti MapStruct? 14](#_Toc138080100)

[Upotreba 15](#_Toc138080101)

[UnitTests with MapStruct 20](#_Toc138080102)

[Izvori: 21](#_Toc138080103)

Modulith

….

API Gateway

GraphQL

[This Photo](http://www.pngall.com/laptop-png/download/5426) by Unknown Author is licensed under [CC BY-NC](https://creativecommons.org/licenses/by-nc/3.0/)

Modulith

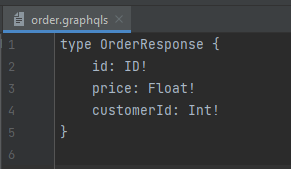
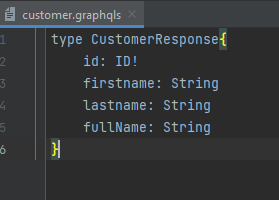
# GraphQL

Sa klasičnim REST API-jem u situaciji kada imamo potrebu da prikažemo više različitih resursa na jednoj stranici, najčešće moramo da šaljemo više HTTP zahtjeva ka serveru. Takođe, u određenim situacijama dolazi do tzv. overfetchinga (npr. endpoint vraća listu kompletnih objekata, a nama su za prikaz potrebna samo određena polja). GraphQL nam rješava ove probleme. Na backend-u definišemo koje podatke je moguće dobiti određenim zahtjevom, a frontend dio, tj. API consumer, u zahtjevu definiše koja polja želi.

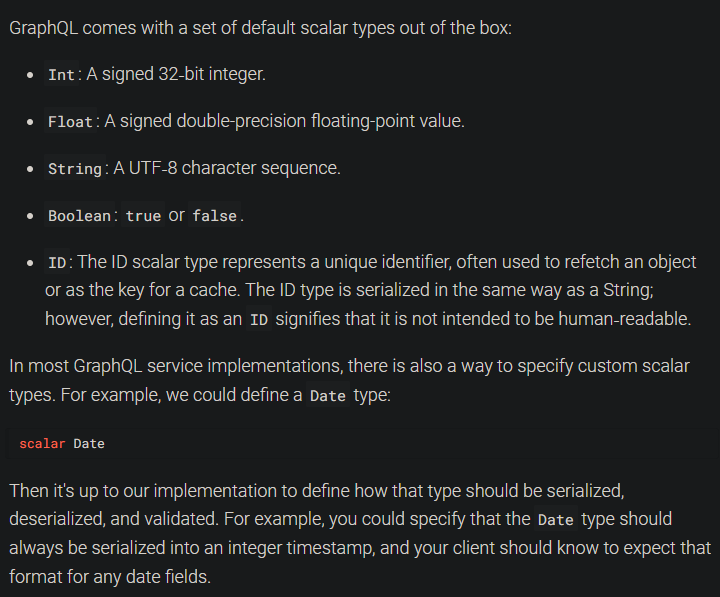
Na backendu definišemo GraphQL šemu, u fajlovima sa ekstenzijom .graphqls ili .gqls na putanji *src/main/resources/graphql/\*\**.

## Type objekti

Jedna od osnovnih komponenti GraphQL šeme su objektni tipovi, koji predstavljaju objekte koje možemo dohvatiti sa servera (dakle tipove koji su povratne vrijednosti našeg API-ja. U našem slučaju bi to bili response objekti. Primjeri response objekta koji se koristi u demo aplikaciji (moguće je i sve tipove definisati u istom fajlu, ali ja sam odvojila zbog preglednosti):

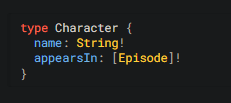


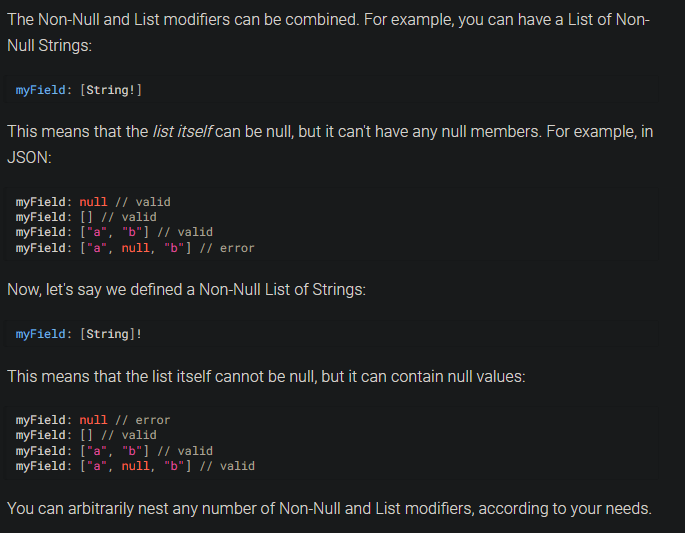
Svaki tip ima određena polja, koja mogu biti skalarne vrijednosti ili kompleksni objekti (koje ćemo takođe samostalno definisati). GraphQL ima predefinisane sledeće skalarne vrijednosti:



! pored tipa označava da je polje non nullable, odnosno da se očekuje da server uvijek za dato polje vraća non-null vrijednost. Ukoliko vrati null, desiće se GraphQL execution error, pa treba obratiti pažnju na korištenje !.

Polja unutar tipa mogu biti i liste drugih elemenata. Da bi tip polja definisali kao listu koristimo uglaste zagrade.





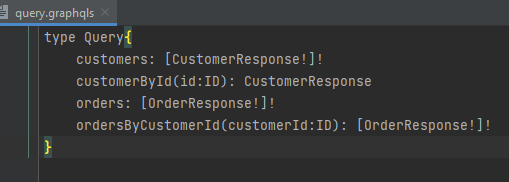
Svaki tip koji definišemo u GraphQL šemi, mora imati odgovarajuću java klasu. Koja klasa odgovara kojem tipu se zaključuje na osnovu Query tipa, što je objašnjeno dalje u tekstu. Važno je da se ista klasa mora uvijek koristiti za isti tip (dakle ne možemo koristiti više klasa za isti tip).

## Query i Mutation objekti

Kada radimo sa GraphQL-om naša aplikacija (tj. naš gateway) ima samo jedan endpoint, u našem slučaju: <http://localhost:8080/graphql> na koji se uvijek šalju POST zahtjevi. Ono što čini da se ti zahtjevi razlikuju nalazi se u tijelu tog POST zahtjeva.

Iako su većina tipova koje definišemo objektni tipovi, postoje dva glavna specijalna tipa: Query i Mutation. Svaka GraphQL šema mora imati tačno 1 Query root objekat i najviše 1 Mutation objekat. Query tip nam služi da u njemu definišemo sve GET zahtjeve koje naša aplikacija resolvuje, dok unutar Mutation tipa definišemo akcije koje vrše izmjenu resursa, tj. pandan su POST, PUT i DELETE zahtjevima.

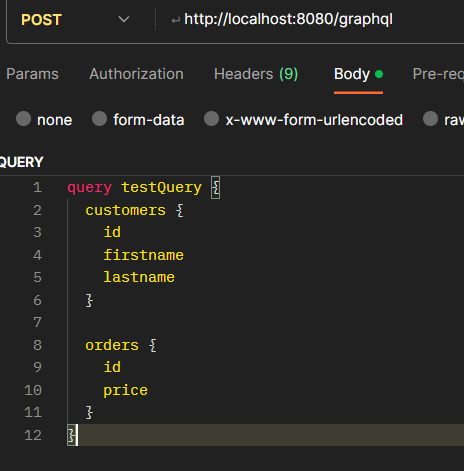
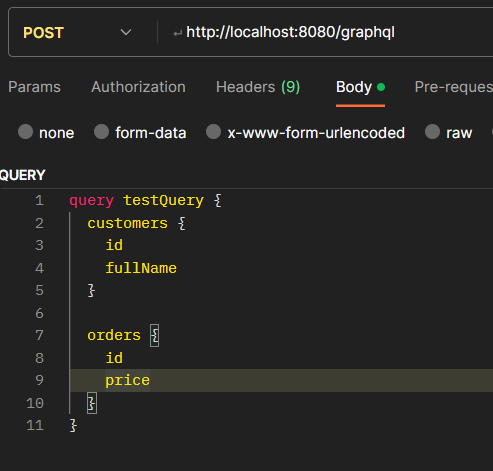
Na naredne dvije slike je dat primjer Query objekta i jednog zahtjeva koji je moguće uputiti.



U šemi je dozvoljeno da postoji samo 1 root query objekat, dakle ne možemo ga razdvajati u više fajlova. Vidimo da u našem Query objektu mi definišemo da će naša aplikacija biti sposobna da odgovori na sljedeće zahtjeve:

* da vrati listu svih customer-a u formi CustomerResponse objekata, koji su ranije prikazani,
* da vrati jednog customer-a na osnovu ID-ja,
* da vrati listu svih order-a u formi OrderResponse objekata,
* da vrati sve order-e za određenog customer-a.

Zahtjeve upućujemo na sljedeći način:

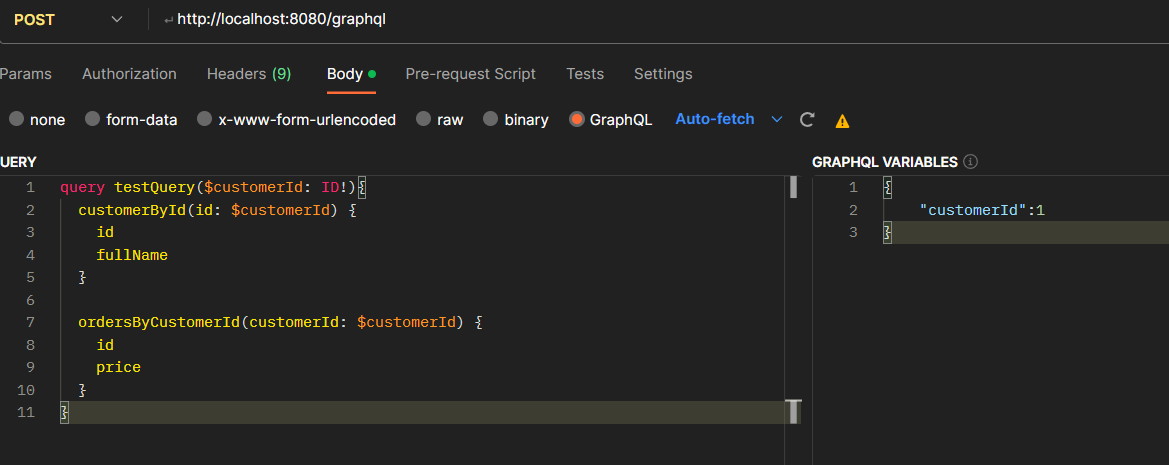


Na osnovu ove dvije slike vidimo sljedeće:

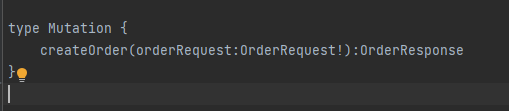
* Mi upućujemo samo 1 zahtjev sa našeg klijenta, iako on zahtjeva komunikaciju sa 2 modula. Način na koji je ovo riješeno će biti objašnjen dalje u tekstu.
* Koristimo isto polje Query tipa, ali zahtjevamo različita polja, u prvom slučaju fullName, a u drugom firstname i lastname. Kako nam u aplikacijama često nisu uvijek potrebna sva polja povratne vrijednosti određenog zahtjeva u različitim slučajevima, korištenjem ove mogućnosti GraphQL-a, da navedemo samo polja od interesa, izbjegavamo overfetching. Ukoliko je polje tipa kompleksnog objekta, i za taj objekat definišemo polja od interesa:



Vidimo u definiciji Query objekta da određena polja imaju definisane neke argumente. Broj argumenata je proizvoljan. Upotreba argumenata prilikom slanja zahtjeva izgleda ovako (napomena: u primjeru koristim Varijable, to je zapravo način da se omogući dinamička izmjena argumenata zahtjeva. Zamislimo da imamo u frontend app definisan query koji će biti tijelo POST zahtjeva, u koji onda po potrebi interpoliramo određene vrijednosti. Detaljnije o ovome: <https://graphql.org/learn/queries/#variables>):

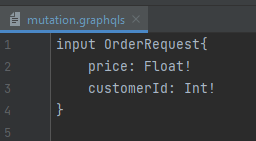


Mutation objekti izgledaju isto kao Query objekti:

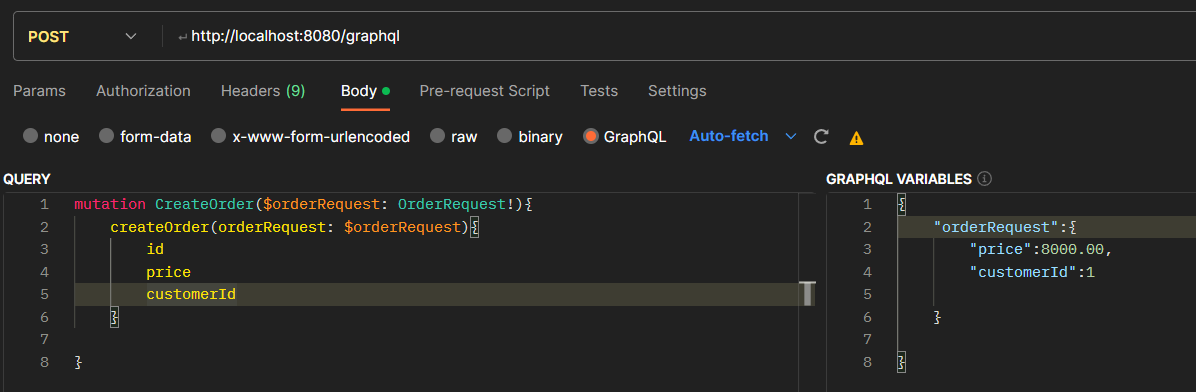


Ovdje vidimo primjer createOrder polja, koje prima argument tipa OrderRequest i vraća OrderResponse koji smo već koristili. Ovdje se prvi put susrećemo i sa kompleksnim tipom kao argumentom određenog polja. Za korištenje kompleksnih objekata kao argumenata, potrebno je definisati input tipove objekata. Input tipovi izgledaju identično kao regularni objektni tipovi, samo što se umjesto type koristi input ključna riječ. Input tipovi bi u našem slučaju logički odgovarali Request objektima.

Primjer input OrderRequest tipa:



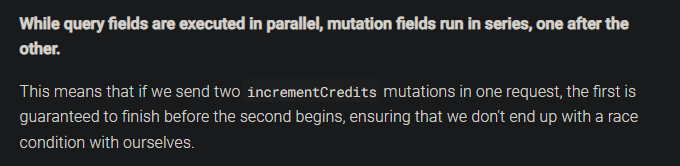
Slanje zahtjeva koji koristi *Mutation* objekat (vidimo da ovdje takođe koristimo varijable):



Vidimo da i u slučaju slanja ovakvih zahtjeva možemo da definišemo koja polja želimo iz odgovora, u ovom slučaju sva.

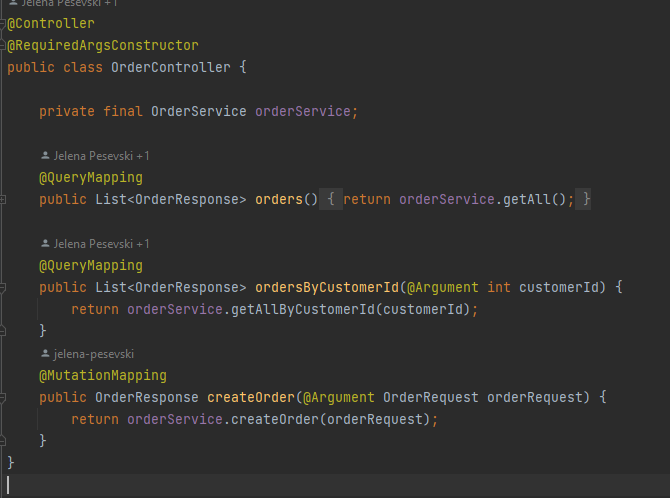
Vidimo da u zahtjevima definišemo *operation type* (koji može biti *query*, *mutation* ili *subscription*) i *operation name* – eksplicitno ime za operaciju. Naziv operacije nije neophodan, ali je poželjan jer olakšava debagovanje i prolaske kroz logove (lakše je identifikovati query na osnovu naziva nego na osnovu sadržaja).

VAŽNO:



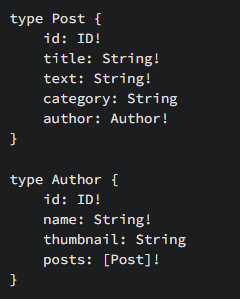
## Spring for GraphQL

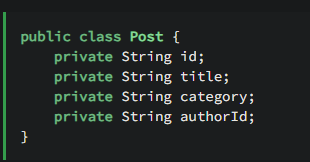
Polja definisana unutar Query i Mutation objekata se mapiraju u metode kontrolera, korištenjem anotacije @QueryMapping. Ukoliko polje ima navedene argumente, oni se navode kao argumenti metode, sa anotacijom @Argument za svaki argument. I metode i argumenti moraju imati iste nazive navedene u .graphqls fajlu. Metode mogu imati i dodatne parametre poput GraphQLContext.

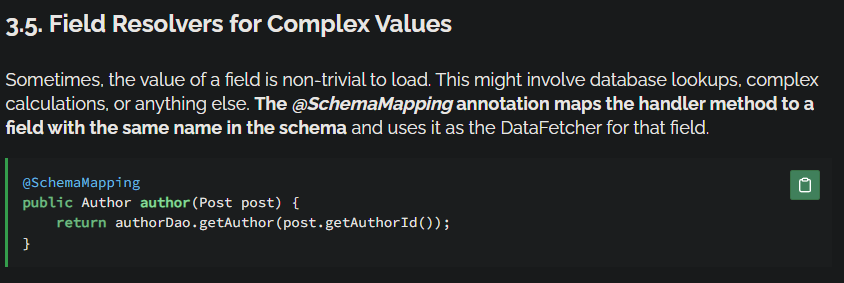


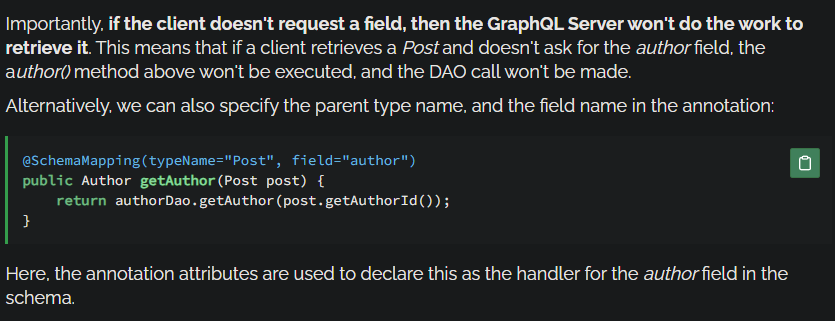
Kao što je već pomenuto, svaki kompleksni tip u GraphQL-u odgovara nekom Java bean-u. Polja unutar Java bean-a se direktno preko naziva mapiraju na polja unutar tipa iz šeme. Polja koja nemaju korespodnentna polja unutar šeme se ignorišu i ne uzrokuju greške, a mogu se koristiti na određeni način da učitaju druga polja.

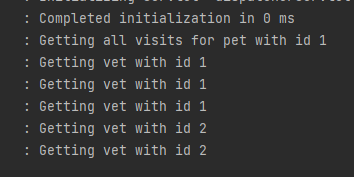
Primjer definisanog tipa i korespodentne klase:



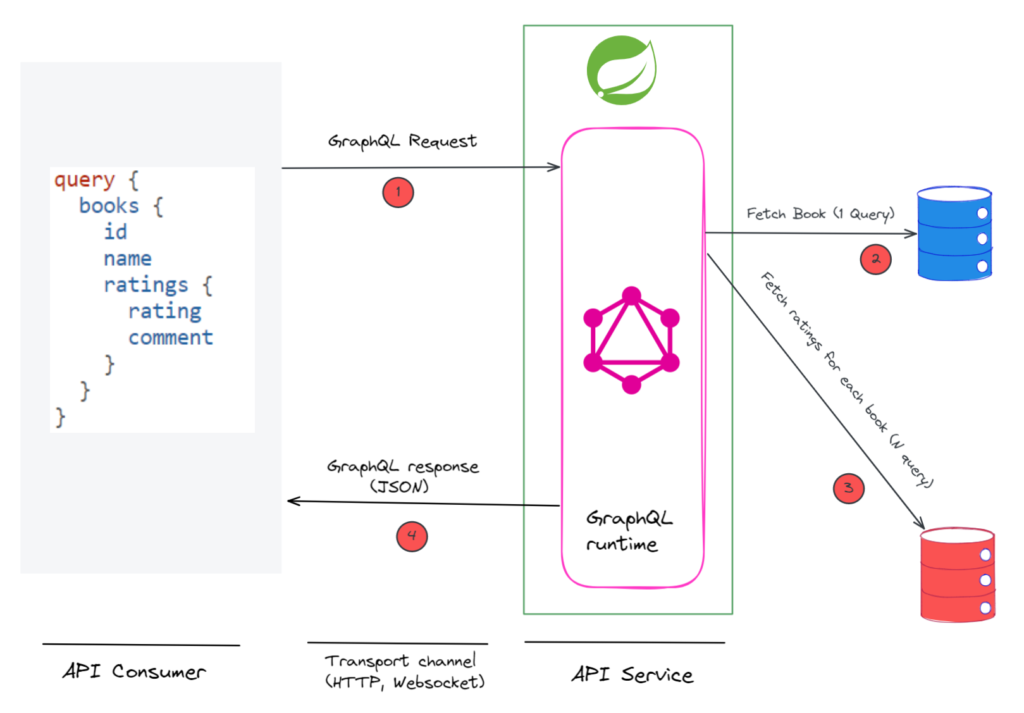






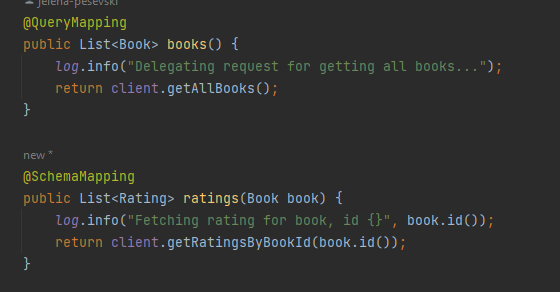


## N+1 problem

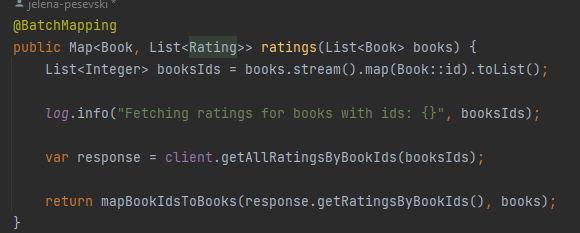


U primjeru iznad vidimo da klijent zahtjeva sve knjige i rejtinge za svaki od njih. To uzrokuje n HTTP poziva ka API-ju rejting modula. Bolje bi bilo da se jednim upitom dohvate svi rejtinzi za sve knjige, odnosno da imamo 1 HTTP zahtjev ka modulu za knjige i zatim 1 zahtjev ka modulu za rejtinge. GraphQL nudi rješenje za tzv. N+1 problem.

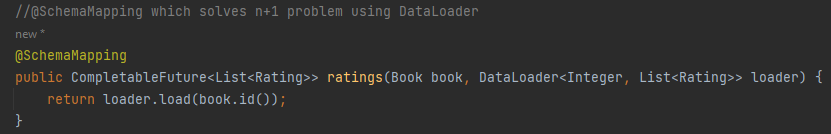
Prvi način je korištenje @BatchMapping anotacije i odgovarajuće resolver metode.

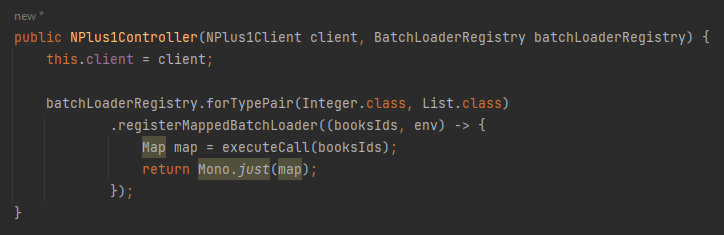


Umjesto slanja više zahtjeva za dohvatanje rejtinga, šalje se 1 zahtjev koji dohvata sve rejtinge za prosljeđene knjige. Koristi se notifikacija @BatchMapping,



Drugi način na koji se može riješiti n+1 problem, koji dodatno omogućava i keširanje na nivou jednog zahtjeva je korištenje DataLoader-a.

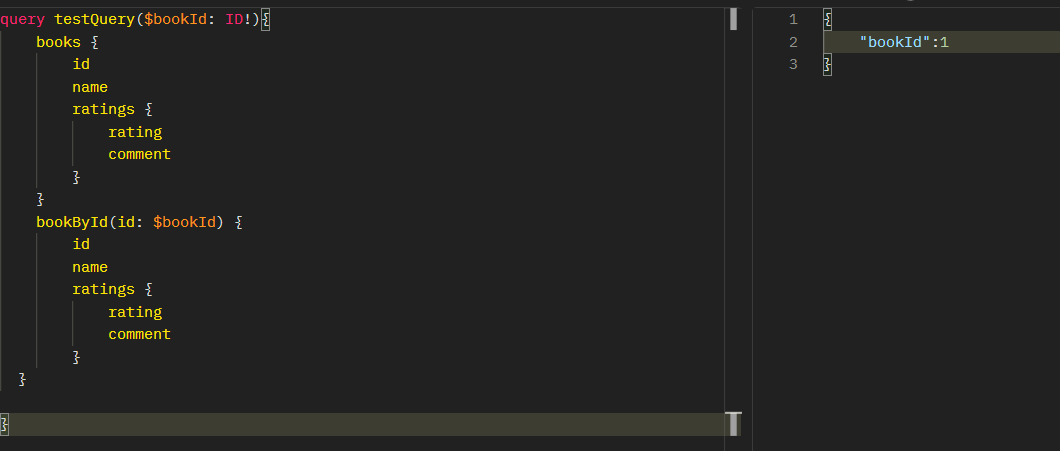




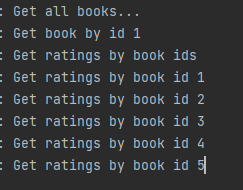
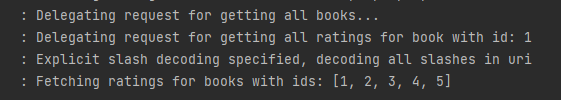


## Caching

DataLoader radi keširanje na nivou Query-ja. Primjer: dohvatamo sve knjige i rejtinge za svaku od njih i dohvatamo knjigu sa id-jem 1 i rejtinge za nju, sve u istom zahtjevu.



Vidimo da se rejtinizi za knjigu sa id-jem 1 izvršavaju dva puta:



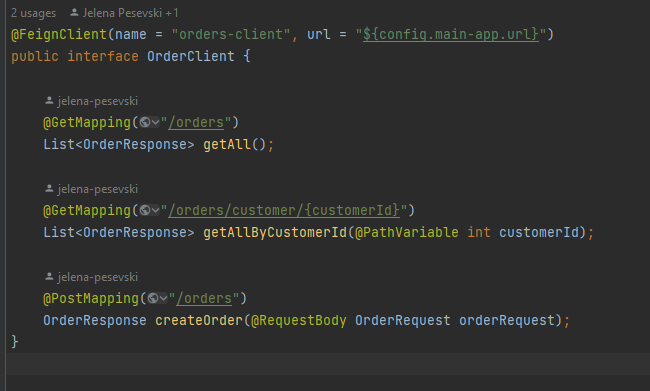
## Kako bi koristili GraphQL u našem sistemu?

Svi zahtjevi koji stižu na API Gateway se prosljeđuju modulith-u, gdje je implementirana poslovna logika. Dakle, mi bi u svakom modulu naše aplikacije imali odgovarajući kontroler, koji bi odgovarao kontroleru unutar API Gateway-a, samo što bi bio klasični RestController sa kojim smo već radili, te je sve isto kao inače. Na ovaj način imamo loosely coupling između gateway-a i modula, jer gateway ne zna gdje se nalaze moduli, niti da pripadaju istoj aplikaciji.

Za komunikaciju sa modulima se koristi OpenFeign biblioteka, koja pripada Spring Cloud-u. Ovo je odlična zamjena za recimo RestTemplate ili druge načine komunikacije sa udaljenim resursima, jer je kod veoma pregledan, a sam FeignClient je veoma lak za korištenje.

# OpenFeign

OpenFeign je deklarativni REST klijent. Dobra stvar je što mi ne moramo da pišemo nikakav kod za pozivanje servisa, osim definicije interfejsa. Potrebno je da main klasu anotiramo sa @EnableFeignClients, što označava da je potrebno skenirati interfejse koji su anotirani sa @FeignClient. Primjer jednog klijenta:



Kao što vidimo, potrebno je da damo logičko ime klijentu, koji se koristi za service discovery i load balancing i da navedemo API base URL. Najbolje je URL učitavati iz application.yml fajla. Nakon toga mi samo deklarišemo metode i anotiramo ih korištenjem klasičnih Spring Web anotacija. Možemo koristiti i @PathVariable, @RequestBody i dr.

Spring Cloud kreira default konfiguraciju za svakog imenovanog klijenta koristeći FeignClientsConfiguration klasu, koja sadrži određene bean-ove, a mi po potrebi možemo napraviti custom konfiguraciju i redefinisati neke od tih bean-ova.

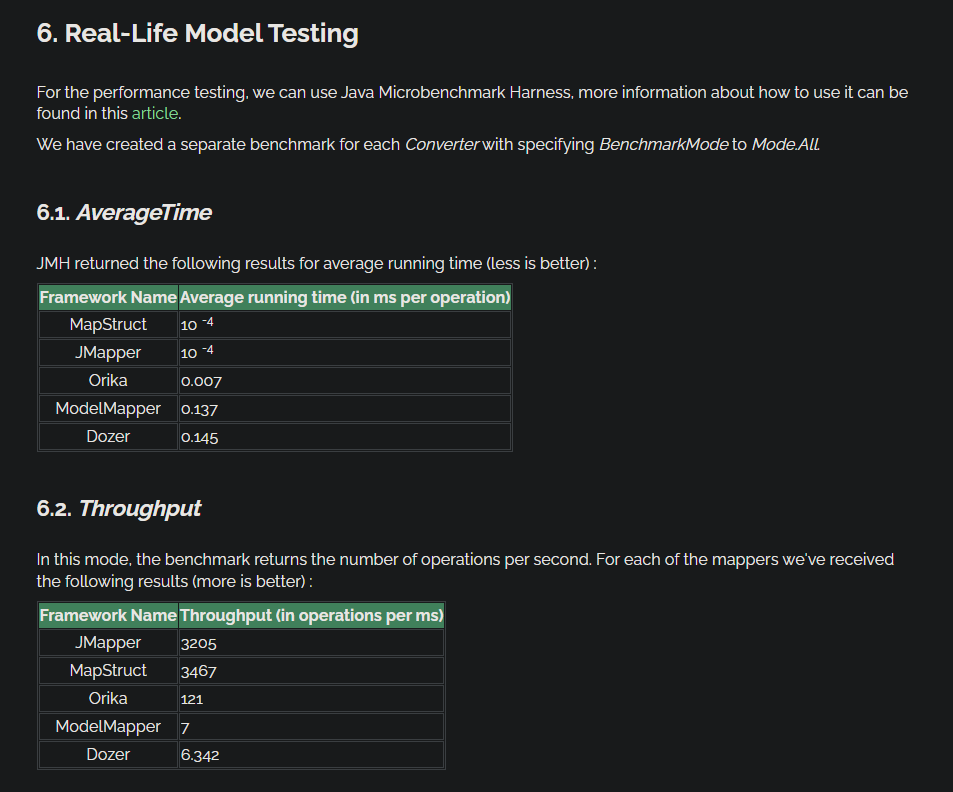
Moguće je dodavati i header-e u zahtjev, npr:



# MapStruct

## Zašto koristiti MapStruct?

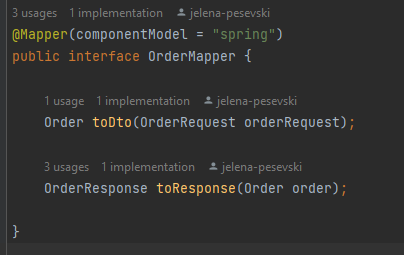
Performanse su daleko bolje od ModelMapper-a i nudi različite dodatne funkcionalnosti. Detaljno poređenje je dato ovdje: <https://www.baeldung.com/java-performance-mapping-frameworks>.



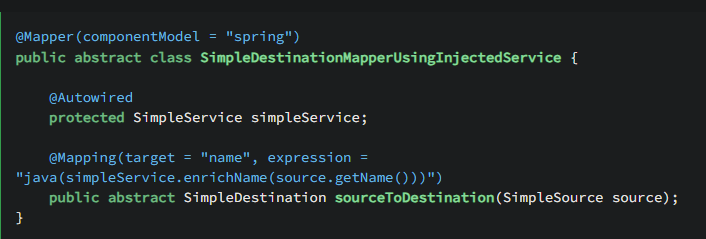
## Upotreba

MapStruct je Java Bean maper. Sa ovim maperom samo je potrebno kreirati interfejse, a sama biblioteka automatski kreira implementaciju tokom kompajliranja. Da bi se to omogućilo, potrebno je pored dependency-ja dodati i odgovarajući plugin koji zapravo generiše implementaciju tokom build-a. (napomena: kada se koristi i lombok i mapstruct plugin ima određene dodatke) Generisane klase se nalaze na putanji */target/generated-sources/annotations*.

Primjer mapera koji je korišten u demo projektu:



MapStruct ima podršku za Spring i CDI. Sa atributem componentModel=“spring“ unutar anotacije kreiramo bean za ovaj maper. Takođe, korištenjem ove anotacije moguće je inject-ovati druge komponente unutar mapera, te ih koristiti za mapiranje određenih polja. Npr:

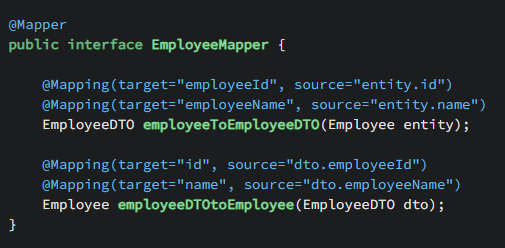


Implementacija koja se generiše na osnovu OrderMapper-a:

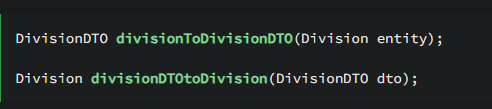


Umjesto interfejsa pri deklarisanju mapera, možemo koristiti i apstraktne klase.

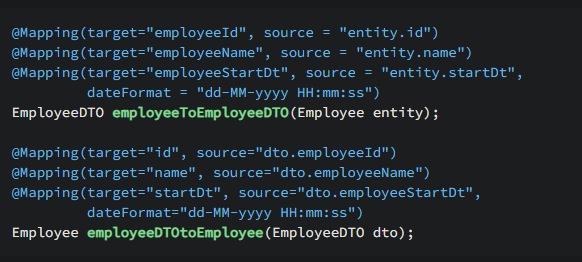
Nije neophodno da su nazivi korespodentnih polja isti i u source i u target objektima. Sa anotacijom možemo naznačiti koja polja se mapiraju u koja.



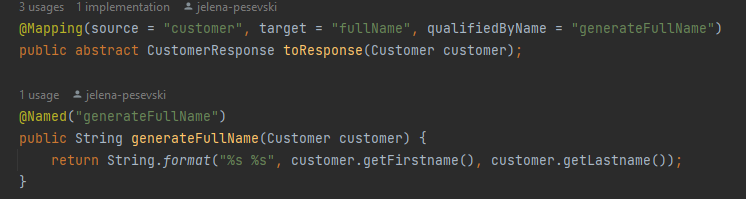
Ukoliko objekat sadrži referencu na drugi objekat, koji nema svoj maper, potrebno je u maperu deklarisati metodu i za taj objekat, samo bez anotacija. Npr. Employee ima referencu na Division i tada dodajemo sljedeće metode:



MapStruct ima podršku za neke implicitne konverzije tipova, npr. String u Date.

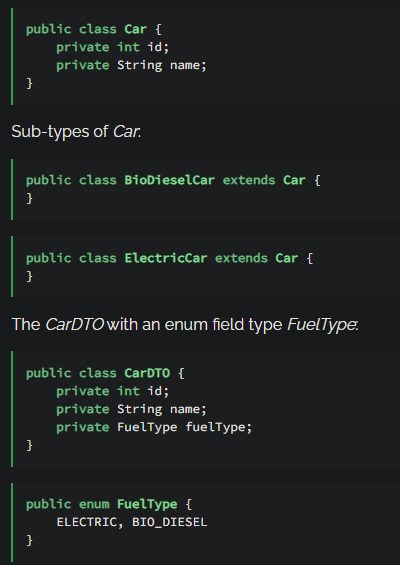


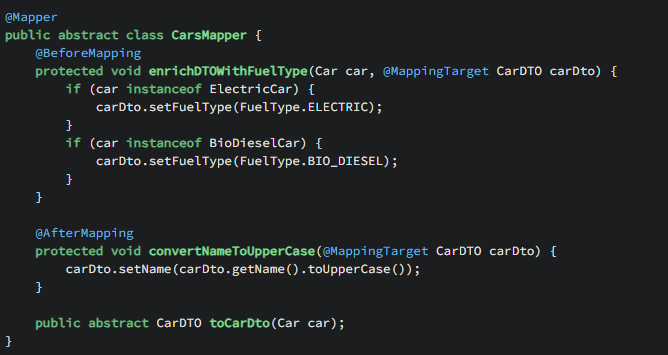
Ponekad je potrebno za mapiranje određenih polja primjeniti neku složeniju logiku. Da bi obezbijedili korištenje te logike, dodajemo sljedeće:



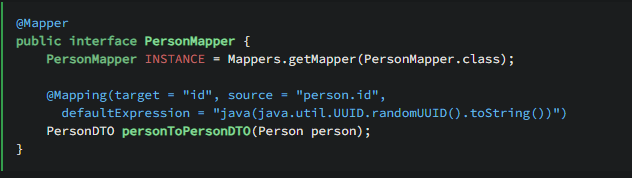
U slučaju da smo koristili apstraktne klase, metode koje se koriste mogu imati bilo koji modifikator pristupa. U slučaju interfejsa koriste se default metode. U nekim slučajevima je moguće da sve default metode imaju iste argumente i iste povratne tipove (postoji više polja istog tipa), te u tom slučaju korištenje @Named anotacije i davanje imena jedini način da ih maper razlikuje.

MapStruct omogućava korištenje @BeforeMapping i @AfterMapping anotacija, koje označavaju metode koje se pozivaju prije i nakon mapiranja. Primjer korištenja:





Postoje još neke opcije poput defaultExpression-a kojim se specifikuje default vrijednost ukoliko je source polje null. Npr:



## UnitTests with MapStruct



Na ovaj način koristi se prava implementacija.

# Izvori:

<https://www.baeldung.com/spring-graphql>

<https://graphql.org/learn/queries/>

<https://graphql.org/learn/schema/>

<https://www.danvega.dev/blog/2023/03/20/graphql-mutations/>

<https://www.baeldung.com/spring-cloud-openfeign>

<https://www.baeldung.com/mapstruct>

<https://www.baeldung.com/java-performance-mapping-frameworks>

<https://www.baeldung.com/mapstruct-custom-mapper>