TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Aleksandr Trofimov 213359IADB

GraphQL ja REST API-de juurutamise ja hooldamise võrdlus pangasüsteemi näitel

Bakalaureusetöö

Juhendaja: Siim Rebane
Bakalaureusekraad

Tallinn 2025

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Aleksandr Trofimov

30.11.2024

Annotatsioon

Käesolevas lõputöös analüüsitakse ja võrreldakse GraphQL-i ja REST API-de juurutamise ning hooldamise praktilisi aspekte pangasüsteemi näitel. Töö keskendub kaasaegsete API arhitektuuride võimekusele ja nende mõjule süsteemide andmeedastuse tõhususele, jõudlusele ning arenduse paindlikkusele. Pangasüsteemi kontekstis, kus kõik olemasolevad API-d on ehitatud REST-standardi järgi, hinnatakse GraphQL-i potentsiaali pakkuda tõhusamat ja paindlikumat lahendust suurte andmemahtude ja keerukate päringustruktuuridega töötamisel.

Lõputöö käigus viidi läbi prototüübi loomine ja jõudlustestid staging-keskkonnas, kus võrreldi nelja erineva andmemahu korral GraphQL-i ja REST-i efektiivsust. Tulemused näitasid, et GraphQL võimaldab andmebaasipäringute arvu ja võrgu koormust märkimisväärselt vähendada, pakkudes kasutajatele suuremat paindlikkust vajalike andmete määratlemisel. Samas toodi esile ka GraphQL-i juurutamise spetsiifilised väljakutsed, sealhulgas skeemide loomise ja turvalisuse integreerimise keerukus.

Lõputöö pakub praktilisi juhiseid ja soovitusi GraphQL-i rakendamiseks suurtes ja andmemahukates süsteemides, sealhulgas panga API infrastruktuuri optimeerimiseks. Töö tulemusi saab rakendada laiemas tarkvaraarenduse kontekstis, pakkudes väärtuslikku sisendit kaasaegsete API arhitektuuride valikul ning võimaldades organisatsioonidel luua kvaliteetsemaid ja efektiivsemaid tarkvaralahendusi.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 35 leheküljel, 5 peatükki, 18 joonist, 4 tabelit.

Abstract

Comparison of Implementation and Maintenance of GraphQL and REST APIs in Banking System

This thesis analyzes and compares the practical aspects of implementing and maintaining GraphQL and REST APIs in a banking system. The research focuses on the capabilities of modern API architectures and their impact on data exchange efficiency, performance, and development flexibility. In the context of banking system, where all existing APIs are based on the REST standard, the potential of GraphQL to provide a more efficient and flexible solution for handling large datasets and complex query structures is evaluated.

A prototype was developed and performance tests were conducted in a staging environment, comparing GraphQL and REST APIs across four distinct data volumes. The results demonstrated that GraphQL significantly reduces the number of database queries and network overhead while providing users with greater flexibility in defining the data they need. However, the study also highlighted specific challenges in adopting GraphQL, such as schema creation and the integration of security measures.

This thesis provides practical guidelines and recommendations for implementing GraphQL in large-scale, data-intensive systems, including optimizing API infrastructure of a banking system. It identifies areas for improvement, such as enhancing the efficiency of QueryCriteria and BatchMapping mechanisms to reduce redundant database queries and minimize API requests to other services. The findings underscore the benefits of GraphQL in modern software projects, particularly in scenarios where selective data fetching and minimal network load are critical.

The results of this thesis are applicable beyond the context of the banking system and offer valuable insights for organizations considering the adoption of GraphQL. By presenting practical solutions and strategic recommendations, this study supports

informed decision-making in API architecture selection and contributes to the development of more efficient and scalable software systems.

The thesis is in Estonian and contains 35 pages of text, 5 chapters, 18 figures, 4 tables.

Lühendite ja mõistete sõnastik

API	*Application Programming Interface*, Mehhanism, mis võimaldab kahel tarkvarakomponendil definitsioonide ja protokollide komplekti kasutades omavahel suhelda [1]
APM	*Application Performance Monitoring* Tööriist, mida kasutatakse jõudluse jälgimiseks ja analüüsimiseks, probleemide tuvastamiseks ning kasutajakogemuse parandamiseks. [1]
backend	Tarkvara osa, mis tegeleb andmetöötluse, loogika ja andmebaasiga, jäädes kasutajale nähtamatuks.
DAO	*Data Access Object* Disainimuster, mis pakub abstraktse liidese andmebaasidele ja muudele püsivusmehhanismidele juurdepääsuks, võimaldades rakendusel hallata andmeid sõltumatult andmebaasi rakenduse detailidest. [2]
DTO	*Data Transfer Object* Objekt, mida kasutatakse andmete edastamiseks rakenduse eri osade vahel, vähendades meetodikutsede arvu. [3]
GraphiQL	Brauseripõhine graafiline tööriist, mis võimaldab GraphQL päringuid interaktiivselt testida ja uurida. [4]
GraphQL	*Graph Query Language* Päringukeel ja töökeskkond, mis keskendub sellele, kuidas andmeid API-st täpselt ja tõhusalt küsida. [5]
HTTP	*Hypertext Transfer Protocol* Rakenduskihi protokoll, mis on loodud teabe edastamiseks võrguga ühendatud seadmete vahel ja töötab võrgu protokollipaki teiste kihtide peal. [6]
IBAN	*International Bank Account Number* Rahvusvaheline standardiseeritud kontonumber, mis võimaldab maksete kiiret ja täpset töötlemist ning kontonumbrite kontrollimist. [7]
Intellij IDEA	Integreeritud arenduskeskkond (IDE), mis toetab mitmeid programmeerimiskeeli ja pakub arendajatele tööriistu koodi kirjutamiseks, testimiseks ja haldamiseks.

JFrog Artifactory	Universaalne DevOps-lahendus, mis võimaldab binaarfailide ja artefaktide majutamist, haldamist ja levitamist. [8]
JSON	*JavaScript Object Notation* Avatud andmevahetusvorming, mis on loetav nii inimestele kui masinatele, olles programmeerimiskeelest sõltumatu ja levinud API väljund paljudes rakendustes.
JWT	*JSON Web Token* Avatud turvastandard, mis määratleb kompaktse ja iseseisva viisi teabe turvaliseks edastamiseks osapoolte vahel JSON-objektina. [9]
Järjestusdiagramm	*Sequence diagramm (inglise keeles)* Visuaalne mudel, mis kujutab süsteemi loogika voogu, keskendudes komponentidevahelisele suhtlusele ja käitumisele. [10]
Kibana	Tööriist andmete visualiseerimiseks ja analüüsimiseks, mida kasutatakse logide, ajaseeria andmete ning rakenduste jälgimiseks [1]
Mikroteenus (teenus)	*Microservice (inglise keeles)* Tarkvaraarenduse arhitektuuriline lähenemine, kus rakendus koosneb väikestest iseseisvatest teenustest, mis suhtlevad selgelt määratletud API-de kaudu, võimaldades kiiremat arendust ja paremat skaleeritavust.
REST	*Representational State Transfer* Arhitektuurne stiil standardite pakkumiseks arvutisüsteemide vahel veebis, hõlbustades süsteemide omavahelist suhtlemist. [1]
SQL	*Structured Query Language* Programeerimiskeel teabe salvestamiseks ja töötlemiseks Relatsiooniandmebaasis [1]
Swagger	Tööriist REST API-de dokumenteerimiseks, testimiseks ja visualiseerimiseks, mis aitab arendajatel mõista ja kasutada API-de funktsionaalsust.
Three-Tier Architecture	Kolmekihiline arhitektuur, disainimuster, mis jagab rakenduse kolmeks kihiks – esitluskiht, äriloogikakiht ja andmekiht, et parandada süsteemi hallatavust, skaleeritavust ja paindlikku [11]
URL	*Uniform Resource Locator* Unikaalse ressursi aadress internetis, mida kasutavad erinevad kliendid, nagu veebilehitsejad ja rakendused, ressursside, näiteks HTML-lehtede, piltide ja muude failide hankimiseks [12]

Sisukord

1 Sissejuhatus 12
2 REST ja GraphQL teoreetiline kirjeldus ja võrdlus 13
2.1 REST: kontseptsioon, arhitektuur ja põhimõtted 13
2.1.1 REST: Kontseptioon 13
2.1.2 REST: Arhitektuur 13
2.1.3 REST: Põhimõtted 14
2.2 GraphQL: kontseptsioon, arhitektuur ja põhimõtted 16
2.2.1 GraphQL: Kontseptioon 16
2.2.2 GraphQL: Arhitektuur 17
2.2.3 GraphQL: Põhimõtted 17
2.3 GraphQL ja REST API päringute võrdlus 19
2.4 Peamised tegurid API arhitektuuri valimisel 20
3 Praktiline lahendus pangasüsteemi näitel: REST ja GraphQL 21
3.1 Pangasüsteemi olemasolev API arhitektuur 21
3.1.1 Arendusmeeskonnade ülesanded ja API arhitektuuri jaotus 21
3.1.2 Praeguse REST API arhitektuuri ülesehitus 22
3.1.3 Tehnoloogiad 25
3.2 GraphQL API implementeerimine 26
3.2.1 Väljakutsed ja arhitektuurilised otsused 26
3.2.2 Skeemi loomine 28
3.2.3 GraphQL Controller loomine 29
3.2.4 Selektiivne andmete laadimine 31
3.2.5 Batch-päringud 34
3.2.6 GraphiQL 35
3.2.7 Turvalisus 37
3.2.8 Järjestusdiagramm 40
4 Tulemused ja edasised arendussuunad 41
4.1 Võrdlus ja järeldus: andmevahetuse efektiivsus ja jõudlus 41
4.1.1 Üksikpäringute tulemused 41

4.1.2 Graafiline võrdlus ja järeldus 44
4.2 Edasised arendussuunad ja tehnilised võimalused 45
5 Kokkuvõte 47
Kasutatud kirjandus 48
Lisa 1 – Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks 50
Lisa 2 – REST API vastuse JSON näide 51
Lisa 3 – GraphQL skeemid 63
Lisa 4 – GraphQL API päringu skeem 66

Jooniste loetelu

Joonis 1. REST API näite diagrammis: päring ja vastus kliendi-serveri suhtluses, kasutades näitena GET-meetodit 16
Joonis 2. GraphQL näite diagrammis: päring konkreetse andmeväljadega ja vastus kliendi-serveri suhtluses 18
Joonis 3. Illustreeriv ülevaade DCP, BBL ja ABC teenuste suhtlusest REST API kaudu 22
Joonis 4. Järjestusdiagramm: DCP, BBL ja ABC teenuste omavaheline suhtlus REST API kaudu 23
Joonis 5. Illustreeriv ülevaade DCP, BBL ja ABC teenuste suhtlusest koos integreeritud GraphQL API-ga 28
Joonis 6. Kohandatud annotatsioon GraphqlController 29
Joonis 7. GraphQL skeemi päringu seos Spring GraphQL kontrolleriga 30
Joonis 8. Selektiivne andmete laadimine meetodis findContractsWithDetails 32
Joonis 9. Class ContractDetailsQueryCriteria ja selle funktsioonid 33
Joonis 10. GraphQL skeemi ja BatchMapping annotatsiooni vahelise seose näide 34
Joonis 11. Kogutud kaaslaenuvõtjate ID-de laadimine ühe optimeeritud API päringuga 35
Joonis 12. GraphiQL ülevaade: Skeemide dokumentatsioon 36
Joonis 13. GraphiQL ülevaade: Explorer - päringuehitaja töörist 37
Joonis 14. GraphiQL ülevaada: JWT tokeniga päringu tulemused 38
Joonis 15. GraphiQL ülevaade: Puuduva Authorization headeri veateade 39
Joonis 16. GraphiQL ülevaade: Skeema inntrospektsioonipäringu avatus 39
Joonis 17. Järjestusdiagramm: Selektiivne andmete laadimine ja teenustevaheline suhtlus GraphQL API kaudu 40
Joonis 18. Graafik: REST ja GraphQL API vastuse aegade võrdlus erinevate andmemahtude korral 44
Joonis 19. JSON vastuse struktuur 52

Tabelite loetelu

Tabel 1. REST ja GraphQL API-de põhivõrdlus 22
Tabel 2. REST API päringute jõudluse tulemused 49
Tabel 3. GraphQL API päringute jõudluse tulemused 49
Tabel 4. GraphQL ja REST API jõudluse suhe erinevate andmemahtude korral 50

1 Sissejuhatus

Tänapäeva tarkvaraarenduses on API-d (*Application Programming Interface*) olulised tõhusa andmevahetuse võimaldajad. REST (*Representational State Transfer*) API-d on traditsiooniliselt olnud standardiks, pakkudes lihtsat andmevahetust. Kuid sageli tagastavad päringud rohkem andmeid, kui vaja, halvendades süsteemi jõudlust ja suurendades võrgukoormust.

Pangasüsteem, mida käesolevas lõputöös käsitletakse, kasutab REST API-sid, mis võivad tagastada liigseid andmevälju, mida klientrakendused ei vaja. Näiteks võib üks päring sisaldada üle 63 andmevälja, millest vaid väike osa on vajalikud. Selline probleem – üleliigsete andmete edastamine, suurendab süsteemi ja võrgu koormust ning mõjutab negatiivselt kasutajakogemust.

Lõputöö eesmärk on uurida, kas GraphQL (*Graph Query Language*) võiks olla tõhusam lahendus. GraphQL võimaldab klientidel pärida täpselt vajalikke andmeid, vältides liigset andmeedastust. Töö raames luuakse GraphQL API prototüüp, mis töötab koos olemasoleva REST API-ga, ning viiakse läbi jõudlustestid nende lahenduste tõhususe hindamiseks.

Lõputöö tulemused toetavad arendajaid ja arhitekte API arhitektuuri valikul ning aitavad parandada tarkvaralahenduste kvaliteeti. GraphQL-i paindlikkus avab uusi võimalusi ka keerulistes ja suuremahulistes süsteemides.

2 REST ja GraphQL teoreetiline kirjeldus ja võrdlus

API-d ehk rakenduste programmeerimisliidesed on kaasaegse tarkvaraarenduse lahutamatu osa, võimaldades süsteemidel standardiseeritult andmeid vahetada. Arhitektuuri valik, olgu selleks REST või GraphQL, mõjutab otseselt süsteemi efektiivsust, hooldatavust ja skaleeritavust. Mõlemal lähenemisel on oma tugevused ja piirangud, sõltuvalt rakenduse vajadustest.

Selles peatükis antakse ülevaade REST-i ja GraphQL-i teoreetilistest alustest ja uuritakse nende rakendatavust erinevates olukordades, valmistades ette praktilisi võrdlusi pangasüsteemi näitel.

2.1 REST: kontseptsioon, arhitektuur ja põhimõtted

2.1.1 REST: Kontseptioon

REST on arhitektuuristiil, mille lõi Roy Fielding oma doktoritöös 2000. aastal, et määratleda efektiivne viis hajusate süsteemide, nagu veebiteenuste, andmevahetuseks. REST põhineb HTTP-protokolli (*Hypertext Transfer Protocol*) kasutamisel ja on mõeldud tagama, et andmete edastamine serveri ja kliendi vahel toimuks võimalikult lihtsalt ja skaleeritavalt. REST API-d kasutavad staatusteta lähenemist, mis tähendab, et iga päring serverile on iseseisev ja sisaldab kogu vajalikku informatsiooni vastuse andmiseks, võimaldades seeläbi kiiret ja töökindlat andmevahetust.

REST-i populaarsus tuleneb selle lihtsusest ja laialdasest ühilduvusest erinevate platvormide ja tehnoloogiatega. Kuna REST API-d on üles ehitatud HTTP-le, saab neid kasutada igas süsteemis, mis toetab HTTP-protokolli. See teeb REST-ist paindliku lahenduse, mida kasutatakse laialdaselt erinevates veebirakendustes, mobiilirakendustes ja mikroteenuste arhitektuurides. [13]

2.1.2 REST: Arhitektuur

REST arhitektuur võimaldab kliendil hallata serveri ressursse standardsete päringumeetodite abil, toetades mitmeid vastusevorminguid nagu XML, YAML ja JSON (*JavaScript Object Notation*), kus JSON on tänapäeval üks eelistatumaid vorminguid tänu

oma lihtsusele ja loetavusele. Kliendi ja serveri suhtlus toimub kindlate komponentide kaudu:

- **Lõpp-punkt URL** (*Endpoint*)**:** Lõpp-punkti URL (*Uniform Resource Locator*) on unikaalne identifikaator, mis osutab kindlale ressursile võrgus. Iga ressursi jaoks on defineeritud kindel URL, (näiteks */api/v1/resource/{id}*).
- **HTTP meetod:** REST kasutab viit põhilist meetodit:
    - **GET:** Andmete pärimine.
    - **POST**: Ressursside loomine.
    - **DELETE:** Ressursi kustutamine.
    - **PUT:** Ressursi asendamine või loomine..
    - **PATCH**: Osaline uuendamine.
- **Päis** (*Header*)**:** Edastab metaandmed, sealhulgas autentimisinfo, andmetüüpide määratlused. Näiteks: "Authorization", "Content-Type", "Cache-Control" jne.
- **Sisu** (*Body*)**:** Edastab andmeid, mida kasutatakse ressursside loomiseks või muutmiseks, tavaliselt JSON-vormingus.

2.1.3 REST: Põhimõtted

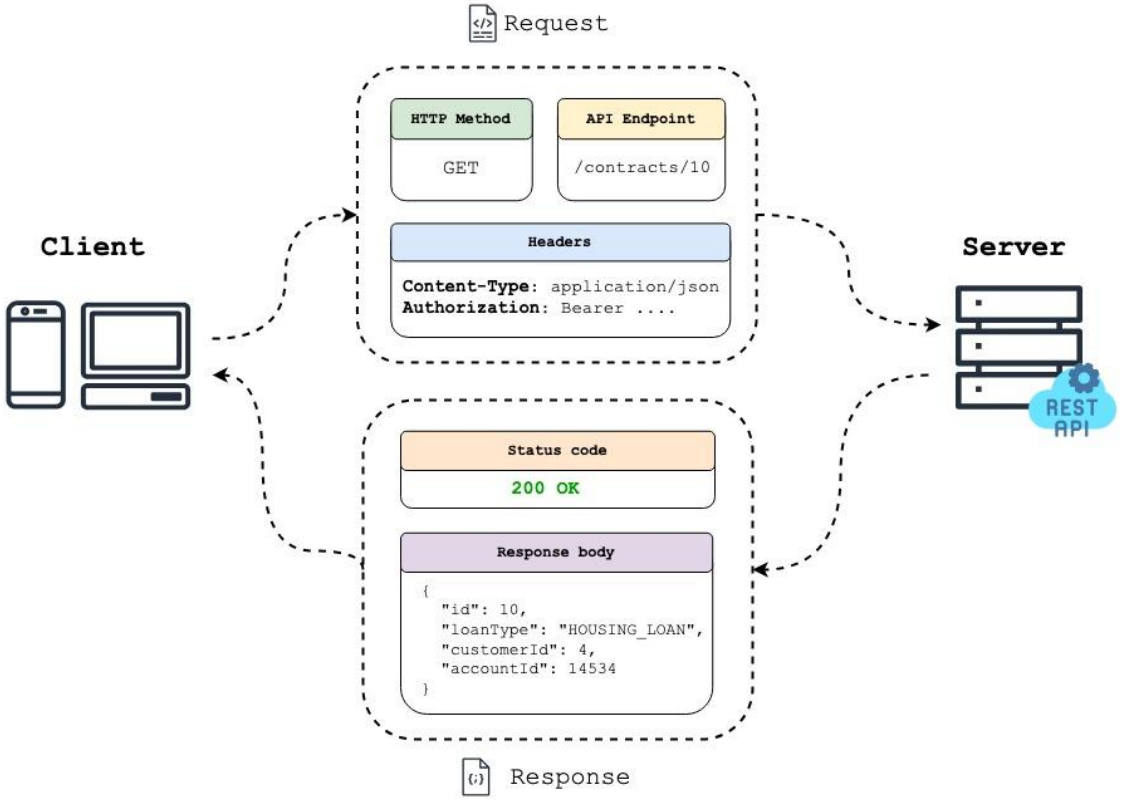
REST arhitektuur põhineb kuuel põhimõttel, mis tagavad API-de töökindluse, lihtsuse ja skaleeritavuse:

1. **Kliendi-serveri arhitektuur:** Klient (nt veebirakendus) ja server (nt API-teenus) on eraldatud, võimaldades neid iseseisvalt arendada ja hallata. Klient vastutab kasutajaliidese eest, server aga andmete töötlemise ja säilitamise eest.
2. **Staatusteta suhtlus** (*Statelessness*)**:** Iga päring on iseseisev ja sisaldab kogu vajaliku informatsiooni vastamiseks. See lihtsustab API skaleerimist ja suurendab töökindlust, kuna server ei pea kliendi seisundit meeles pidama.
3. **Vahemälu kasutamine** (*Cacheability*)**:** REST API-d võimaldavad klientidel salvestada korduvkasutatavaid andmeid vahemällu, mis vähendab serveri

koormust ja parandab andmevahetuse kiirust. Server saab määrata, millised vastused on vahemällu salvestatavad ja kui kaua need kehtivad.

4. **Ühtne liides** (*Uniform Interface*)**:** REST API-d järgivad ühtset liidest, mis hõlmab järgmiseid reegleid:
    - **Ressursside identifitseerimine:** Igal ressursil on unikaalne URL, näiteks */api/v1/resource/{id}.*
    - **Standardsete HTTP meetodite kasutamine**: GET, POST, PUT, DELETE ja PATCH võimaldavad andmete pärimist, loomist, uuendamist ja kustutamist.
    - **Andmete esitamine**: Serveri vastused on organiseeritud ja masintöödeldavad, tavaliselt JSON- või XML-vormingus.
5. **Kihiline süsteem:** REST API arhitektuur toetab kihilist ülesehitust, kus päringud võivad läbida erinevaid vahekihte, näiteks puhverserverid ja vahemälud. Iga kiht täidab kindlat funktsiooni, suurendades turvalisust ja jõudlust.
6. **Kood nõudmisel** (*Code on Demand*) (valikuline): Kuigi seda kasutatakse harva, võib server vajadusel edastada kliendile koodi (nt JavaScript), mis täidab kliendi poolel teatud töötlusülesandeid.

Allpool on toodud näide REST-i päringu kasutamise kohta (Joonis 1), mis illustreerib, kuidas klient esitab serverile päringu koos vajalike päiste, lõpp-punkti URL-i ja päringu sisuga.



Joonis 1. REST API näite diagrammis: päring ja vastus kliendi-serveri suhtluses, kasutades näitena GET-meetodit

2.2 GraphQL: kontseptsioon, arhitektuur ja põhimõtted

2.2.1 GraphQL: Kontseptioon

GraphQL on 2012. aastal Facebooki poolt loodud ja 2015. aastal avalikuks tehtud andmepäringute keel ja API standard, mille eesmärk on pakkuda paindlikumat ja tõhusamat viisi andmete pärimiseks ja manipuleerimiseks. GraphQL võimaldab klientidel täpsustada, milliseid andmeid nad vajavad, ning server tagastab ainult küsitud andmed. See aitab vältida liigseid ja ebapiisavaid andmeid, vähendades võrguliiklust ja optimeerides andmeedastust.

Erinevalt REST-ist, kus iga ressursi jaoks on eraldi lõpp-punktid, kasutab GraphQL ainult ühte universaalset lõpp-punkti. Kliendid saavad päringus määrata, milliseid andmevälju nad vajavad, mis teeb selle eriti tõhusaks keeruliste andmestruktuuride korral. Lisaks

võimaldab GraphQL päringuid, mis koondavad andmeid mitmest allikast, vähendades vajadust teha mitu eraldiseisvat päringut ja parandades seeläbi jõudlust. [5]

GraphQL-i võime pakkuda paindlikkust ja tõhusust muudab selle atraktiivseks valikuks rakenduste jaoks, kus on keerulised ja mitmetasandilised andmestruktuurid ning vajadus minimaalsete andmeedastusmahuga.

2.2.2 GraphQL: Arhitektuur

GraphQL arhitektuur on üles ehitatud paindlikkusele ja efektiivsusele, võimaldades päringuid, mille tulemus on kohandatud vastavalt kliendi vajadustele. GraphQL API-l on vaid üks lõpp-punkt (näiteks */graphql*), mille kaudu saavad kliendid esitada erinevaid päringuid ja määratleda täpselt, milliseid andmevälju ja struktuure nad vajavad.

GraphQL-i arhitektuuris on kolm peamist päringutüüpi:

1. **Query** – andmete pärimiseks, kus klient määrab vajalikud väljad..
2. **Mutation** – andmete muutmiseks, näiteks loomine, uuendamine või kustutamine.
3. **Subscription** – reaalajas teavituste saamiseks andmemuudatuste korral.

GraphQL arhitektuur tagab efektiivse andmevahetuse, võimaldades kliendil esitada kohandatud päringuid, mis parandavad API jõudlust ja vähendavad võrguressursside raiskamist.

2.2.3 GraphQL: Põhimõtted

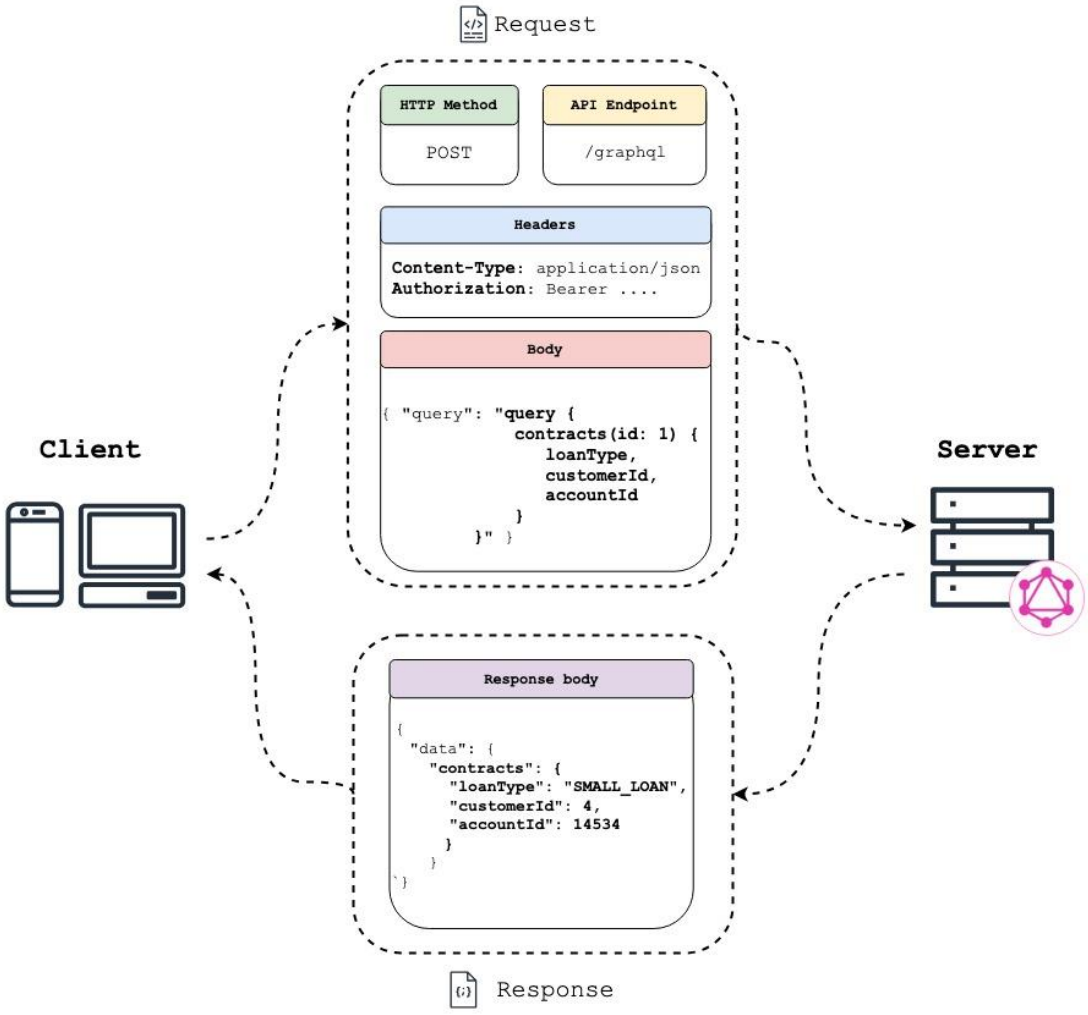
GraphQL-i tööpõhimõtted on suunatud paindlikkusele, täpsusele ja efektiivsusele, et pakkuda dünaamilist ja skaleeritavat andmepäringusüsteemi.

1. **Andmeefektiivsus** (*Data Efficiency*): Klient määrab täpselt, milliseid andmeid on vaja, välistades liigse või puuduliku andmevahetuse REST API-dega võrreldes..
2. **Hierarhiline struktuur**: Päringu struktuur vastab hierarhiliselt saadava andmeobjekti struktuurile, muutes andmed intuitiivsemaks ja selgemaks.
3. **Üks lõpp-punkt** (*Single Endpoint*): GraphQL kasutab ühte paindlikku lõpp-punkti, mis suudab hallata kõiki päringuid ja andmemuudatusi.

4. **Tugevalt tüübistatud süsteem** (*Type-safe*): Serveri skeem määratleb selgelt kõik võimalikud andmeväljad ja nende tüübid, võimaldades päringute valideerimist ja vähendades vigasid.

5. **Reaalajas uuendused** (*Subscriptions*): Subscription-ide abil saab klient reaalajas teavitusi serveripoolsetest muudatustest, mis on oluline dünaamiliste rakenduste jaoks.

Joonis 2 kujutab näidet GraphQL päringust, kus klient esitab serverile andmepäringu koos vajalike päiste ja konkreetsete andmeväljadega, mida klient soovib vastuseks saada.



Joonis 2. GraphQL näite diagrammis: päring konkreetse andmeväljadega ja vastus kliendi-serveri suhtluses

2.3 GraphQL ja REST API päringute võrdlus

REST ja GraphQL on populaarsed andmevahetustehnoloogiad kliendi ja serveri vahel, pakkudes erinevaid lähenemisi päringute tegemisele ja andmete tagastamisele. Mõlema tehnoloogia tugevused ja piirangud määravad nende sobivuse erinevates olukordades.

Tabel 1 annab selge ülevaate REST-i ja GraphQL-i päringute võrdlusest, esitledes nende peamisi erinevusi ja sarnasusi.

Tabel 1. REST ja GraphQL API-de põhivõrdlus

Võrdluspunkt	**REST API**	**GraphQL API**
Protokoll	HTTP	HTTP
Päringustruktuur	Eraldi lõpp-punkt iga ressursi jaoks	Üks lõpp-punkt kõige ressursside jaoks
Andmete hulga juhtimine	Tagastab fikseeritud andmestruktuuri sõltuvalt lõpp-punktist	Klient määrab täpselt vajalike andmeväljade hulga
Mitme ressursi päringud	Nõuab mitut päringut eri ressursside jaoks	Üks päring saab küsida andmeid mitmest ressursist
Päringu formaat	Kasutab GET, POST, PUNT, DELETE HTTP-meetodeid ja saadab JSON või XML vastuse	Enamasti saadetakse kõik päringud POST meetodiga. Päringu struktuur JSON-vormingus kehas
Versioonihaldus	Vajalik tagamaks ühilduvust (nt */v1/resource* ja */v2/resource*)	Skeemi uuendamine ilma versioonihalduse vajaduseta
Tüübikindlus	Pole loomulikult tüübistatud ja dokumentatsioon ei pruugi olla täielikult usaldusväärne	Tugevalt tüübistatud skeem
Status koodid	Kasutab laia valikult staatuskoode (200, 404 jne)	Tagastab alati 200, veateated *errors* väljas

2.4 Peamised tegurid API arhitektuuri valimisel

API arhitektuuri valik sõltub suuresti projekti eesmärkidest ja vajadustest. REST ja GraphQL pakuvad erinevaid eeliseid sõltuvalt andmete keerukusest, koormusest ja hooldusvajadustest. REST API on sobiv väiksemate rakenduste jaoks, kus töötatakse lihtsamate ja standardsete andmete ja päringutega. Seevastu GraphQL-i eelistatakse, kui andmevahetuskoormus on suur ja on oluline vähendada andmeedastust või kombineerida erinevaid andmeallikaid ühes päringus.

Allpool on välja toodud peamised tegurid, mida arendajad ja arhitektid peaksid arvesse võtma, valides REST ja GraphQL API-de vahel:

1. **Andmete täpsus ja optimeerimine**: Kui on oluline vähendada liigset andmeedastust ja määratleda täpsed andmeväljad, on GraphQL paindlikum valik. REST sobib paremini standardsete ja fikseeritud andmestruktuuride jaoks.

2. **Mitme allika päringud**: Kui andmeid on vaja pärida mitmest allikast korraga, siis GraphQL võimaldab seda teha ühe päringuga, samas kui REST nõuab sageli mitut päringut.

3. **Versioonihaldus ja hooldatavus**: REST API-s on versioonihaldus vajalik ühilduvuse tagamiseks, samas kui GraphQL-i skeemi saab uuendada ilma uut versiooni lisamata.

4. **Jõudlus ja vahemällu salvestamine**: REST võimaldab lihtsamat vahemällu salvestamist, mis on kasulik korduvate päringute puhul. GraphQL päringud on dünaamilised, mistõttu vahemälu haldamine on keerulisem.

5. **Arenduse keerukus**: REST on lihtsam ja laialdaselt tuntud, samas kui GraphQL-i rakendamine nõuab rohkem skeemihaldust ja spetsiaalseid tööriistu.

6. **Turvalisus ja ligipääsupiirangud**: REST-is saab turvameetmeid hõlpsasti rakendada lõpp-punkti tasemel. GraphQL nõuab lisakontrolle, et hallata ligipääsu erinevatele andmeallikatele.

3 Praktiline lahendus pangasüsteemi näitel: REST ja GraphQL

3.1 Pangasüsteemi olemasolev API arhitektuur

Pangasüsteemi API arhitektuur põhineb mikroteenuste põhimõtetel, kus erinevad meeskonnad vastutavad oma teenuste ja andmete eest. Selline arhitektuur võimaldab paindlikkust ja iseseisvust arenduses, kuid toob kaasa väljakutseid, kuna andmed ja süsteemid on jaotatud mitme meeskonna vahel. API-de peamine eesmärk on andmevahetus erinevate rakenduste vahel ning juurdepääsu pakkumine finantsteenuste ja kliendihalduse jaoks vajalikele süsteemidele.

3.1.1 Arendusmeeskonnade ülesanded ja API arhitektuuri jaotus

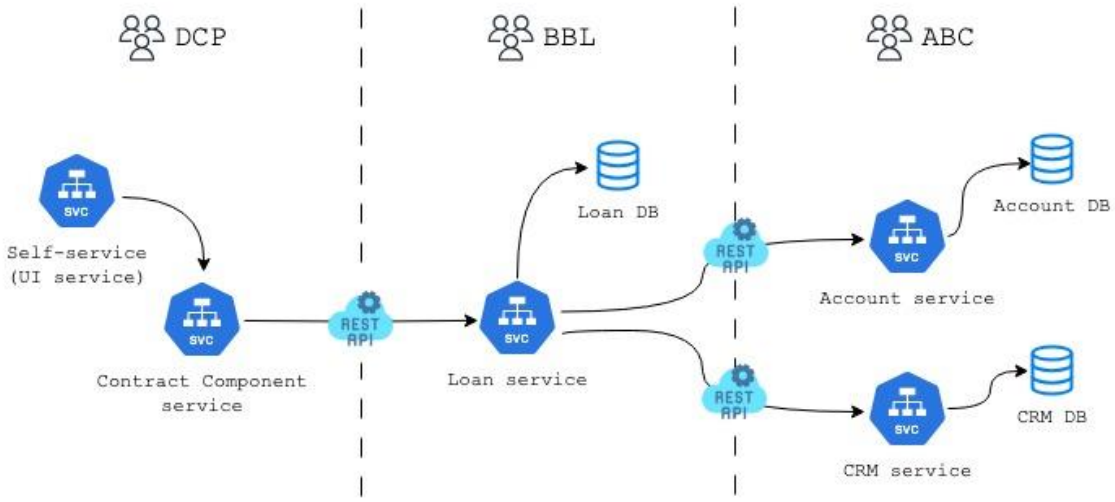
Pangas tegutsevad erinevad arendusmeeskonnad, kes haldavad spetsiifilisi teenuseid ja nendega seotud API-sid:

BBL meeskond vastutab laenu- ja liisinguteenuste eest, sealhulgas maksegraafikute, päevavahetusprotsesside ja intresside arvutamise haldamise eest. Nende töö hõlmab kogu lepingu elutsükliga seotud loogikat, alustades lepingu loomisest kuni lõpetamiseni. BBL meeskond keskendub backend-arendusele ja pakub erinevaid REST API lõpp-punkte, mida teised meeskonnad, näiteks DCP, saavad kasutada laenu ja liisingu andmete tarbimiseks.

ABC meeskond haldab deposiidi- ja kontohaldusteenuseid, sealhulgas kliendikontosid ja hoiuseid. ABC vastutab konkreetse andmekogumi eest, mis on seotud kontode ja deposiitide haldamisega. Nende hallatavad API lõpp-punktid võimaldavad ligipääsu ainult nende spetsiifilistele andmetele ja on suunatud teenustele, mis vajavad kontohaldust.

DCP meeskond keskendub iseteeninduse projekti arendusele, pakkudes klientidele juurdepääsu laenu- ja kontoinfole. Kuna iseteenindus vajab andmeid nii BBL-i kui ka ABC meeskonna süsteemidest, peab DCP kasutama nende REST API-sid, et koguda vajalikku kliendiinfot.

Joonis 3 annab illustreeriva ülevaate, kuidas DCP meeskonna teenus suhtleb BBL-i ja ABC meeskondade teenustega, et koguda vajalikku kliendiinfot.



Joonis 3. Illustreeriv ülevaade DCP, BBL ja ABC teenuste suhtlusest REST API kaudu

3.1.2 Praeguse REST API arhitektuuri ülesehitus

Pangasüsteemii REST API arhitektuuris on arendustiimidel spetsiifilised lõpp-punktid, mis teenindavad kindlaid ärivaldkondi. Näiteks BBL meeskonna lõpp-punkt */api/v2/contract/header/{headerId}/details* võimaldab DCP meeskonnal saada lepinguandmeid.

DCP meeskonna töövoog koosneb kahest olulisest sammust:

1. **Kõigi kliendilepingute ID-de päring**: DCP meeskond kasutab lõpp-punkti /api*/v1/contractSearch/search*, et hankida kliendi lepingute ID-d. Siiski tagastatakse siin rohkem andmeid, kui vaja, kuna DCP vajab ainult ID-sid.

2. **Iga lepingu kohta eraldi päring**: Pärast ID-de saamist tehakse iga lepingu kohta eraldi päring lõpp-punktile */api/v2/contract/header/{headerId}/details*, kusjuures iga päring tagastab rohkem andmeid, kui DCP tegelikult vajab. Näiteks 100 lepingu korral tehakse 100 individuaalset päringut.

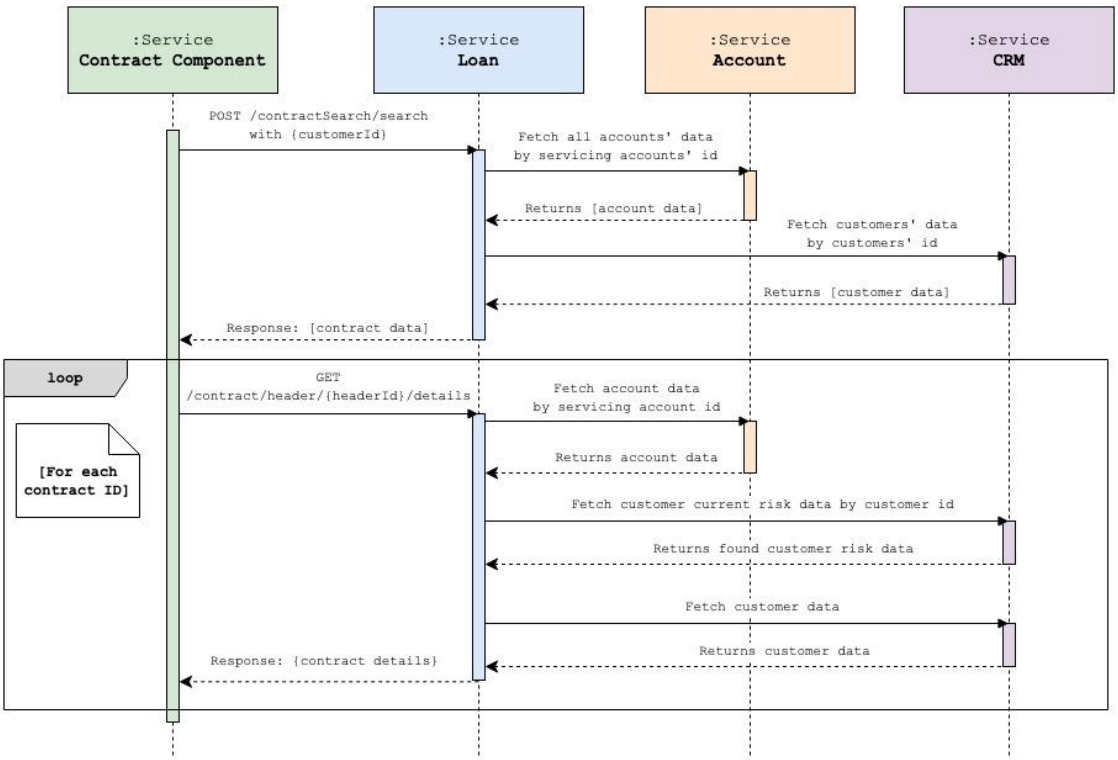
Sellise töövoo tagajärjel tekib mitmeid probleeme, mis vähendavad tootlikkust:

- **Liigne andmeedastus**: Lõpp-punktid tagastavad rohkem andmeid, kui DCP meeskond tegelikult vajab. Näiteks, kui vaja on ainult laenusummat või intressimäära, kaasatakse vastuses kogu lepingu detailid.

- **Mittevajalikud lisapäringud**: Lepingu põhiinfo hankimine toob kaasa lisapäringud "account" ja "crm" teenustesse, isegi kui neid andmeid igakordselt ei vajata.

- **Suure koormusega töövoog**: Mitme lepingu korral suureneb päringute arv märkimisväärselt – näiteks 100 lepingu puhul tehakse 100 individuaalset päringut, mis suurendab serverikoormust ja pikendab vastamise aega.

Järjestusdiagramm (Joonis 4) illustreerib DCP töövoogu, näidates päringuid lepingute ID-de ja detailide hankimiseks, ning BBL teenuse suhtlust teiste teenustega, näiteks konto ja "crm" teenustega kaaslaenuvõtjate andmete kogumiseks. Lõplik JSON vastus on **Lisa 2**.



Joonis 4. Järjestusdiagramm: DCP, BBL ja ABC teenuste omavaheline suhtlus REST API kaudu

DCP meeskond kasutab lõpp-punkti erinevates iseteeninduse rakenduse vaadetes, kus kuvatakse lepingu andmeid, näiteks laenusummat, maksegraafikut või intressimäära. Lõpp-punkt tagastab siiski vaikimisi kõik lepingu detailid, mis sisaldavad sageli üle 60

andmevälja, sh andmeid, mida iga kasutusjuht ei vaja, põhjustades liigset andmeedastust ja päringute vajadust teistesse teenustesse.

Lisaks lepingu põhiandmetele on BBL teenusel vaja andmeid kahest ABC meeskonna teenusest:

- **"account" teenus**: Igal lepingul on seotud IBAN (*International Bank Account Number*) konto, mille ID (*servicingAccountId*) on salvestatud BBL-i andmebaasis. Selle konto detailse info saamiseks, sealhulgas saldo ja konto staatus, pöördub BBL meeskond ABC "account" teenuse poole, kasutades selle API lõpp-punkte.

- **"crm" teenus**: Kaaslaenuvõtjate (*coBorrowers*) info saadakse ABC "crm" teenusest vastavalt kliendi ID-dele (*customerId*).

Kui DCP meeskond pärib andmeid lõpp-punktidest */api/v1/contractSearch/search* ja */api/v2/contract/header/{headerId}/details*, kaasnevad sellega sageli lisapäringud ABC "account" ja "crm" teenustesse. See suurendab kogu töövoo keerukust ja võrguressursside tarbimist.

Kuigi teoreetiliselt oleks võimalik luua iga kasutusjuhu jaoks eraldi REST lõpp-punktid, tekitaks see uusi probleeme:

- Iga kord, kui DCP meeskond vajab lisavälju, tuleb BBL meeskonnal olemasolevat lõpp-punkti täiendada või luua uus.

- Uued lõpp-punktid võivad aja jooksul muutuda keerukaks ja ebaefektiivseks.

- Aja jooksul tekib taas liigse andmeedastuse probleem, kui lõpp-punktidesse lisanduvad väljad, mida kõik kliendid ei pruugi vajada.

Praegune REST API arhitektuur põhjustab liigset koormust ja keerukust. Pidev vajadus liigse andmeedastuse ja mitme lisapäringu järele suurendab serverikoormust ja raskendab tõhusat andmevahetust. GraphQL-i rakendamine võiks pakkuda paindlikumat lahendust, võimaldades klientidel pärida täpselt vajalikke andmevälju ja vähendades seeläbi liigset andmeedastust ning optimeerides serveri ressursikasutust.

3.1.3 Tehnoloogiad

Olemasolev REST API arhitektuur kasutab kaasaegseid tehnoloogiaid ja disainimustreid, mis toetavad süsteemi skaleeritavust, hooldatavust ja turvalisust:

- Tehnoloogiad:
    - **Java ja Spring Boot:** Mikroteenused on Java keeles ja Spring Boot raamistikus, mis võimaldab modulaarset ja lihtsat arendust. Annotatsioonid nagu *@RestController* ja *@RequestMapping* defineerivad REST API lõpp-punktid.
    - **MyBatis**: Kasutatakse andmepäringute ja andmebaasihalduse jaoks, lihtsustades DAO-de (*Data Access Object*) ja SQL (*Structured Query Language*) *Mapper*-ite haldust.
- Disainimuster on *Three-Tier Architecture*:
    - **Esitluskiht** (*Presentation*): REST API lõpp-punktid (*Controller*).
    - **Äriloogika kiht** (*Business Logic*): Domeeniloogika ja andmetöötlus (*Service*).
    - **Andmejuurdepääsu kiht** (*Data Access*): Andmebaasi ühendus DAO objekti kaudu (MyBatis).
- Turvalisus: Kõik turvalisusega seotud loogikad on rakendatud eraldi „Commons“ teegis, mis on saadaval ettevõtte JFrog Artifactory-s. See teek pakub:
    - Spring Security OAuth2 Resource Server: Autentimise ja autoriseerimise jaoks.
    - Ühiskasutatavaid turvalahendusi, mida kasutavad mitmed mikroteenused, tagades standardiseeritud lähenemise API turvalisusele.

Selline arhitektuuriline lähenemine toetab suurt arendustiimide hulka, võimaldades erinevatel meeskondadel iseseisvalt oma mikroteenuseid arendada ja hallata. Samas loob see sõltuvusi jagatud turvateekide ja andmebaasilahenduste osas, mis võivad mõjutada arhitektuuriliste muudatuste tegemist, nagu näiteks GraphQL-i integreerimine.

3.2 GraphQL API implementeerimine

Pangasüsteemis rakendati GraphQL API eesmärgiga luua paindlik ja tõhus andmepäringumehhanism, mis võimaldab klientidel küsida ainult vajalikke andmeid, vältides liigset andmeedastust. Lahendus tugines kahel põhieesmärgil:

1. **Funktsionaalsuse võrreldavus REST API-ga**: GraphQL API pidi pakkuma samu andmevälju ja teenuseid, mida olemasolev REST API toetab, et säilitada süsteemide ühilduvus.

2. **Päringute arvu ja võrgukoormuse vähendamine**: GraphQL võimaldab dünaamilist ja selektiivset andmehaldust, mis optimeerib andmevahetust ja ressursikasutust.

GraphQL API rakendamise protsess hõlmas mitmeid arhitektuurilisi ja tehnoloogilisi väljakutseid, kuna süsteem pidi säilitama olemasolevate REST API-dega kooskõla, integreeruma olemasolevate mikroteenuste ja turvalahendustega ning pakkuma samas paindlikumat ja efektiivsemat andmehalduse mudelit. Järgmistes jaotistes kirjeldatakse rakendamise peamisi väljakutseid, tehtud arhitektuurilisi otsuseid ning praktilisi lahendusi.

3.2.1 Väljakutsed ja arhitektuurilised otsused

Alljärgnevalt on välja toodud peamised väljakutsed ja nende ületamiseks tehtud arhitektuurilised otsused.

1. **Lähenemisviisi valik: skeemipõhine või koodipõhine arendus**
   GraphQL API arenduses tuli valida skeemipõhise (*schema-first*) ja koodipõhise (*code-first*) lähenemisviisi vahel. Kuigi algselt kaaluti koodipõhist lähenemist, viitasid järgmised tegurid skeemipõhise mudeli paremale sobivusele:
   - **Suurkorporatsioonide praktikad:** Ettevõtted nagu Netflix ja Meta kasutavad skeemipõhist lähenemist, mis on end tõestanud küpse ja usaldusväärse mudelina.
   - **Tööriistade ja raamistikude tugi:** Spring GraphQL ja Netflix-i DGS (*Domain Graph Service*) raamistikud toetavad skeemipõhist arendust,

pakkudes pikaajalist tuge ja uuendusi. Koodipõhised teegid, vastupidi, ei paku samal tasemel tuge ega arenduskiirust.

Otsus: Need kaks võtmetegurit viisid otsuseni valida skeemipõhine arendus, mis tagas tugeva aluse pikaajaliseks ja skaleeritavaks lahenduseks.

2. **Turvalisuse integreerimine olemasoleva süsteemiga**

   GraphQL API turvalisus pidi järgima pangasüsteemi olemasolevaid REST API turvastandardeid. Selleks kasutati ettevõttes jagatud „Commons“ teeki, mis haldab autentimist ja autoriseerimist Spring Security abil. Kõik päringud peavad läbima filtreerimisahela, kus erinevad filtrid täidavad spetsiifilisi ülesandeid, nagu juurdepääsu valideerimine ja autoriseerimine.

   - **Skeemi introspektsioon:** Introspektsioonipäringud tehti avalikult kättesaadavaks, et meeskonnad saaksid kasutada GraphiQL- skeemi uurimiseks.
   - **Päringute valideerimine:** Kõik muud GraphQL-i päringud pidid järgima olemasolevat REST API turvamudelit, sealhulgas autentimist ja autoriseerimist.

   Otsus: Kohandati “Commons” teeki, lisades GraphQL-i jaoks sobivad turvafiltrid.

3. **Dünaamilise andmete laadimise rakendamine**

   GraphQL-i üks peamisi eeliseid on võimalus laadida ainult päringus nõutud andmed. Selle rakendamine tõi kaasa väljakutse optimeerida andmete laadimist viisil, mis väldib tarbetuid päringuid ja vähendab serverikoormust. Alguses kaaluti Spring Data JPA kasutamist, kuid lõpuks otsustati jääda MyBatis-e juurde, kuna see oli juba integreeritud olemasolevasse süsteemi ja pakkus vajalikku paindlikkust.

   Otsus: Loositi dünaamiline päringukriteeriumide mootor – *ContractDetailsQueryCriteria*. See kasutas *DataFetchingEnvironment*-i, et määrata, milliseid välju päringus nõutakse, ja laadis ainult vajalikud andmed. Näiteks:

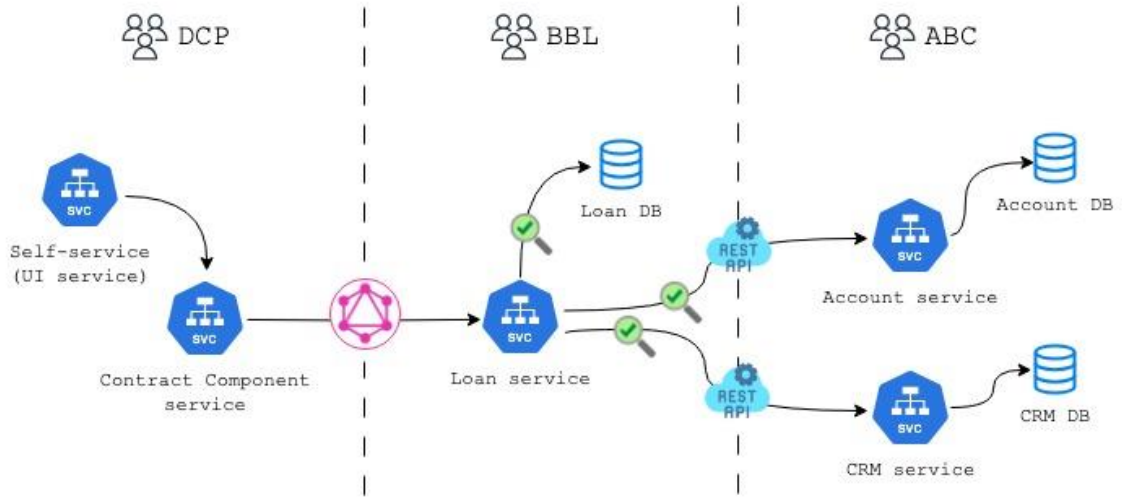
   - Kui kliendi päring nõudis ainult välju *proceedings* ja *coBorrowers*, laaditi ainult vastavad andmed.

- Kui teatud väljad puudusid, jäeti andmebaasi ja kolmandate osapoolte API päringud tegemata.

Selline lähenemine võimaldas serveriressursside efektiivsemat kasutamist ja aitas vältida liigset andmeedastust.

4. **Dokumentatsiooni loomine ja GraphiQL integreerimine**
   REST API-d dokumenteeriti Swagger-i abil, mis pakkus standardiseeritud kirjeldust. GraphQL-i jaoks oli vaja samaväärset lahendust, et arendajad saaksid testida päringuid ja uurida skeemi.
   Otsus: Integreeriti GraphiQL kasutajaliides, mis võimaldab reaalajas päringuid ja skeemi uurimist. Introspektsioonipäringud tehti avalikuks, kuid kõik muud päringud nõuavad autentimist ja autoriseerimist.

Joonis 5 annab illustreeriva ülevaate, kuidas integreerirtud GraphQL API muudab andmevahetuse olemasolevate teenuste vahel tõhusamaks.



Joonis 5. Illustreeriv ülevaade DCP, BBL ja ABC teenuste suhtlusest koos integreeritud GraphQL API-ga

3.2.2 Skeemi loomine

GraphQL skeemi defineerimiseks kasutati nelja eraldi **.graphqls** faili:

1. **operations.graphqls**: Sisaldas kõiki päringuid (*Query*), sealhulgas *contractDetailsV2*, mis võimaldas klientidel küsida lepingu detailseid andmeid.

2. **details.graphqls:** Defineeriti kõik andmetüübid, nagu *ContractDetailsV2*, mis sisaldas lepingu põhiinfot ja seotud välju (nt *coBorrowers*, *components* jne).

3. **enums.graphqls:** Koondas kõik enum-tüübid, nagu lepingu staatuse koodid (*ContractStatus*) ja lepingu tüübid (*LoanType*), mis aitasid skeemi selgemalt struktureerida ja valideerida.

4. **scalars.graphqls:** Lisati kohandatud scalar-tüübid, näiteks kuupäevad (*Date*) ja rahalised väärtused (*BigDecimal*), kasutades *graphql-java-extended-scalars* teeki.

Skeemi defineerimisel tagati, et iga päring oleks paindlik, võimaldades klientidel valida täpselt neid välju, mida nad vajavad, ilma üleliigsete andmeteta. [15] Kõik mainitud skeemid on lisatud **Lisasse 3**.

3.2.3 GraphQL Controller loomine

Pärast GraphQL skeemi ja päringutüübi defineerimist loodi ContractDetailsController klass, mis haldab päringutele vastamist. Kasutati annotatsiooni @GraphqlController, mis lihtsustab Spring Security integreerimist ja standardiseerib kontrollerite loomise.

Kohandatud annotatsioon @GraphqlController

Kohandatud annotatsioon *@GraphqlController* loodi eesmärgiga lihtsustada GraphQL kontrollerite arendust ja integreerida turvamehhanismid sujuvalt GraphQL päringutega. Annotatsioon ühendab mitmeid Spring komponentide annotatsioone, et pakkuda ühtne ja turvaline lähenemine päringute töötlemiseks. Annotatsiooni definitsioon:

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Controller
@PreAuthorize("isAuthenticated()")
public @interface GraphqlController {
}
```

Joonis 6. Kohandatud annotatsioon GraphqlController

Annotatsiooniga *@PreAuthorize("isAuthenticated()")* tagatakse, et kõik päringud peavad olema autenditud, mis on kooskõlas ettevõtte turvapoliitikatega.

Kontrolleri loomine ja @QueryMapping meetodi rakendamine

ContractDetailsController klassis implementeeriti meetod, mis vastab GraphQL skeemis defineeritud päringule *contractDetailsV2*. Selleks kasutati *@QueryMapping* annotatsiooni, mis seob Java meetodi otse GraphQL skeemis defineeritud päringutüübiga. [15]

Põhipunktid @QueryMapping ja skeemi vahelise seose kohta:

1. **Tüüp Query**: GraphQL skeemis peab olema defineeritud tüüp Query, mis sisaldab kõiki saadaval olevad päringuid. Ilma selleta ei seosta Spring GraphQL raamistik päringuid meetoditega.

2. **Meetodi nimi**: Java meetodi nimi peab vastama GraphQL skeemis defineeritud päringunimele (nt *contractDetailsV2*), et raamistik neid seostaks.

3. **Argumentide vastavus**: Meetodi argumendid ja nende tüübid peavad täpselt ühtima skeemis määratletud argumentidega (nt *customerId*).

4. **IDE tugi**: Kui kõik tingimused on täidetud, kuvab Intellij IDEA GraphQL plugin meetodi kõrval ikooni, kinnitades skeemi ja meetodi korrektset seost.

Seose visualiseerimine

Joonis 7 illustreerib, kuidas GraphQL skeemis defineeritud *contractDetailsV2* päring on seotud Java meetodiga kontrolleris. Vasakpoolsed ikoonid Intellij IDEA-s näitavad korrektset seost skeemi ja meetodi vahel. [16]

Joonis 7. GraphQL skeemi päringu seos Spring GraphQL kontrolleriga

Olulised aspektid:

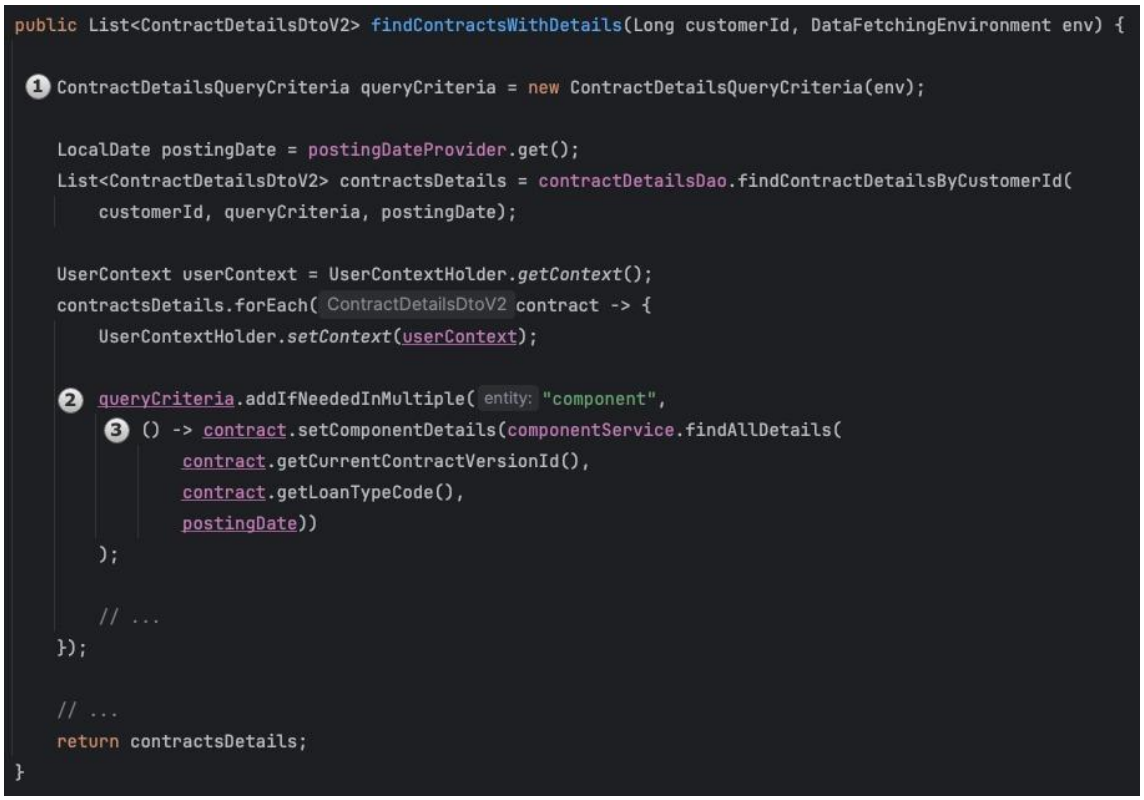
- **@QueryMapping annotatsioon**: Seob meetodi GraphQL skeemis defineeritud päringuga (*Query*), tagades automaatse ühenduse.
- **Meetodi ja päringu nimi**: Java meetodi nimi (*contractDetailsV2*) peab ühtima GraphQL skeemis määratud nimega, et Spring GraphQL neid seostaks.
- **Argumendid**: Annotatsioon *@Argument* (nt *@Argument Long customerId*) tagab kliendi ID korrektse kaardistuse. Argumendi nimi (*customerId*) peab täpselt vastama skeemis määratletule.
- **DataFetchingEnvironment:** Pakub ligipääsu päringus valitud väljadele ja metaandmetele, võimaldades selektiivset andmete laadimist (vt 3.2.4).

3.2.4 Selektiivne andmete laadimine

GraphQL API üheks olulisemaks funktsiooniks on võime laadida ainult kliendi poolt nõutud välju. Selle rakendamiseks loodi klass *ContractDetailsQueryCriteria*, mis analüüsis *DataFetchingEnvironment*-i ja määras, millised väljad päringus sisaldusid. Näiteks:

- Kui klient küsis välju *component* või muu, laaditi ainult need andmed, mis oli küsitatud.
- Kui päringus polnud kindlat välja, jäeti andmete laadimine vahele, vähendades andmebaasi ja võrgu koormust.

Controller (*ContractDetailsController*) kasutab meetodit *findContractsWithDetails*, mis asub klassis *ContractDetailsService*. See meetod rakendab selektiivset laadimist, tuginedes *ContractDetailsQueryCriteria* klassile. Allpool olev joonis näitab näidet selle kohta, kuidas andmete laadimine on selektiivne. Joonises on kasutatud ainult komponentide (*component*) laadimise näidet, kuid tegelikkuses sisaldab lahendus rohkem väljade määratlusi.



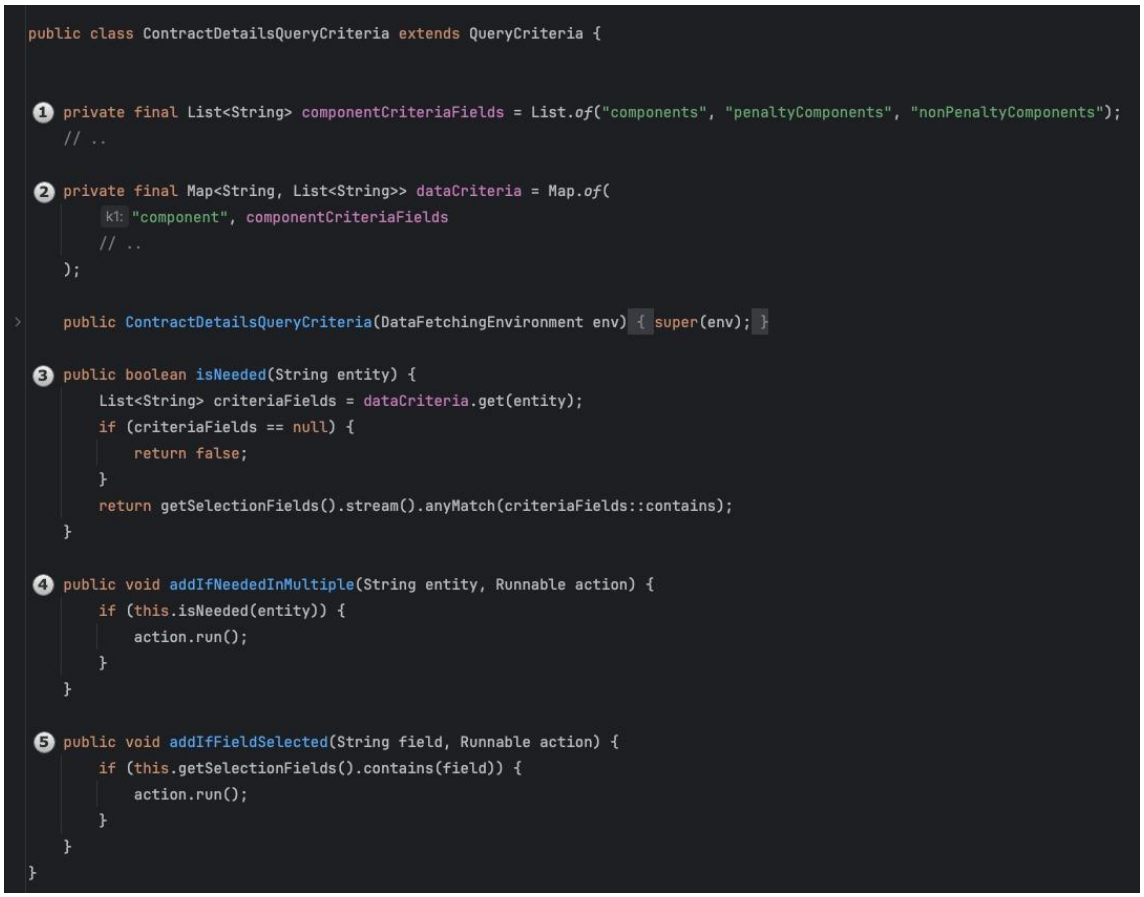
Joonis 8. Selektiivne andmete laadimine meetodis findContractsWithDetails

Joonis 8 selgitab, kuidas meetod *findContractsWithDetails* kasutab mehhanismi *addIfNeededInMultiple*, et laadida vaid kliendi GraphQL päringus nõutud andmeid.

Joonise numbrite selgitused:

1. *ContractDetailsQueryCriteria* klass loodi, et analüüsida *DataFetchingEnvironment*-i ja tuvastada, milliseid välju on kliendi GraphQL päringus küsitud.
2. *addIfNeededInMultiple* kontrollib, kas konkreetne väli, näiteks *component*, on päringus.
3. Kui *component* väli on päringus, laaditakse andmed klassist componentService ja andmed lisatakse vastavalt (*contract.setComponentDetails*). Kui väli ei ole päringus, jääb see tegevus vahele, säästes andmebaasi ja võrguressursse.

Joonise 9 abil selgitatakse *ContractDetailsQueryCriteria* klassi struktuuri ja loogikat. Klass defineerib, millised väljad on saadaval, ja tagab, et ainult kliendi poolt nõutud väljad laaditakse.



Joonis 9. Class ContractDetailsQueryCriteria ja selle funktsioonid

Allpool on toodud joonise 9 numbrite selgitused:

1. **Andmeväljade kriteeriumid**: Klassis defineeritakse väljad (nt *components*, *penaltyComponents*), mida GraphQL päringus saab küsida.
2. **Sõnastik dataCriteria**: Seob väljad (nt *component*) vastavate kriteeriumiväljadega.
3. **Meetod isNeeded**: Kontrollib, kas väli on päringus nõutud, ja lubab vastavalt andmete laadimise.
4. **Meetod addIfNeededInMultiple**: Käivitab tegevuse (nt andmete laadimise), kui väli on päringus küsitud.
5. **Meetod addIfFieldSelected**: Kontrollib konkreetseid välju ja käivitab tegevuse ainult siis, kui väli on valitud.

Need mehhanismid tagavad, et andmeallikaid küsitletakse ainult vastavalt kliendi päringu vajadustele, minimeerides üleliigset koormust ja optimeerides süsteemi jõudlust.

3.2.5 Batch-päringud

GraphQL-i üks suurimaid eeliseid on võimalus optimeerida andmete laadimist batch-mehhanismi abil, mis ühendab mitmed individuaalsed päringud üheks andmebaasipäringuks või API väljakutseks. Lahenduses kasutati Spring GraphQL-i poolt pakutavat annotatsiooni *@BatchMapping*, et vähendada REST API päringute arvu, optimeerides seeläbi serveriressursside kasutust ja parandades süsteemi jõudlust.

Rakenduse näide

Antud lahenduses laaditi kõik lepinguga seotud kaaslaenuvõtjate (*coBorrowers*) andmed ühe API päringuga.

Joonis 10. GraphQL skeemi ja BatchMapping annotatsiooni vahelise seose näide

Joonis 10 näitab, kuidas annotatsioon *@BatchMapping* on seotud GraphQL skeemi *typeName* ja *field* atribuutidega, võimaldades andmete laadimist määratud skeemistruktuuri kaudu. Selle saavutamiseks oli esmalt vaja lisada annotatsioon *@BatchMapping*. Annotatsiooni atribuudid *typeName* ja *field* peavad vastama täpselt GraphQL skeemis määratud väärtustele – antud juhul *typeName = "ContractDetailsV2"* ja *field = "coBorrowers"*. See sidumine on kriitiline, kuna määrab ära, millise tüübi ja välja kohta on *@BatchMapping* rakendatud. Intellij IDEA GraphQL plugin toob selle seose visuaalselt esile, kuvades GraphQL ikooni annotatsiooni kõrval. See lihtsustab arendajate jaoks skeemi ja annotatsiooni omavahelise vastavuse jälgimist ning vähendab võimalike konfiguratsioonivigade riski.

Joonis 11. Kogutud kaaslaenuvõtjate ID-de laadimine ühe optimeeritud API päringuga

Joonis 11 näitab, kuidas *fillDataForCoBorrowers* meetod koondab kõik kaaslaenuvõtjate ID-d üheks unikaalseks kogumiks ning kasutab neid ühe optimeeritud API päringu tegemiseks. See lahendus tagab efektiivse andmete laadimise, vältides korduvaid ja tarbetuid päringuid.

ID-de kogumine: Esimese sammuna koondati kõik kaaslaenuvõtjate ID-d, mis olid seotud *ContractDetailsV2* tüüpi objektidega. Selleks iteratiivselt loodi unikaalne ID-de kogum kõigi lepingute coBorrowers väljadelt, kasutades stream-funktsioone. Selle lähenemise eesmärk oli vähendada päringute arvu ja tagada, et sama ID ei küsitaks rohkem kui üks kord teises API-s.

Kogutud ID-de kasutamine: Saadud ID-dega tehti üksainus API päring *customerService.findCustomersData(coBorrowersIds)* abil. See päring tagastas kõik vajalikud andmed vastavate ID-dega kaaslaenuvõtjate kohta. Seejärel jagati saadud andmed lepingute kaupa ja lisati tagasi vastavatele *ContractDetailsV2* objektidele, et tagada andmete vastavus algse skeemi struktuurile.

3.2.6 GraphiQL

Pangasüsteemi REST API dokumentatsiooni jaoks kasutati Swagger-it (OpenAPI), mis võimaldas arendajatel vaadata kõiki lõpp-punkte ja nende seonduvaid andmemudeleid DTO (*Data Transfer Object*). GraphQL-i rakendamise puhul oli vaja pakkuda sarnast

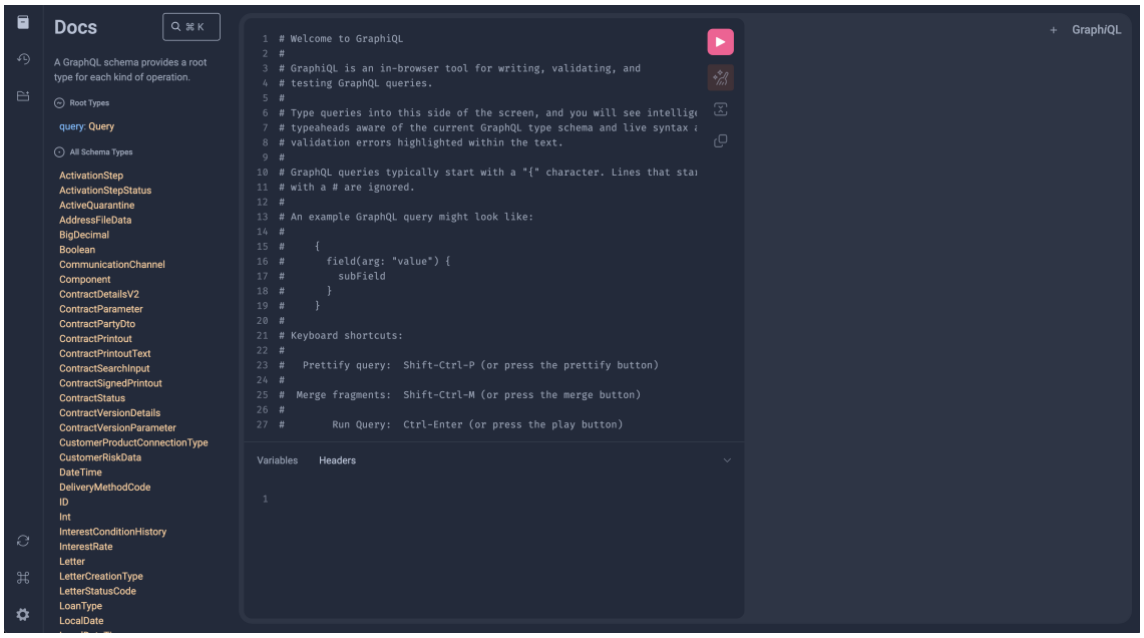
lahendust, mis võimaldaks teistele meeskondadele ligipääsu skeemidele ja nende testimist. Selleks kasutati Spring GraphQL raamistikus saadaval olevat GraphiQL-i. GraphiQL on veebipõhine tööriist, mis võimaldab arendajatel kirjutada, testida ja uurida GraphQL päringuid.

GraphiQL pakub järgmisi võimalusi:

1. **Skeemi ülevaade**: Meeskonnad saavad vaadata kõiki BBL-i GraphQL skeeme, sealhulgas päringuid, tüüpe ja nende struktuure.

2. **Päringute testimine**: GraphiQL võimaldab arendajatel otse veebiliideses päringuid esitada ja tulemusi reaalajas testida. See on eriti kasulik uute funktsioonide arendamisel ja veaotsingul.

Joonis 12 annab visuaalse näite sellest, kuidas erinevate meeskondade liikmed, sealhulgas nii meie kui ka teised meeskonnad, saavad skeemi uurida ja päringuid esitada.



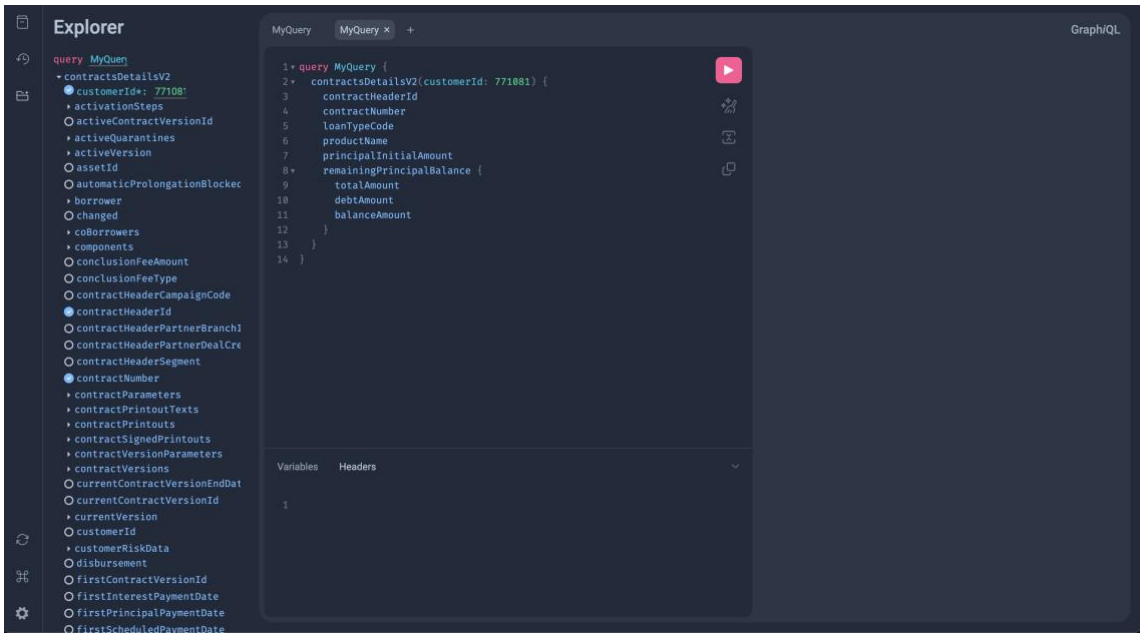
Joonis 12. GraphiQL ülevaade: Skeemide dokumentatsioon

GraphiQL-i üks olulisemaid funktsioone on *Explorer*, mis võimaldab kasutajatel hõlpsasti päringuid üles ehitada. *Explorer* toetab kõiki vajalikke omadusi, sealhulgas:

- Väljade välimine päringutes.
- Sisendparameetrite lisamine ja redigeerimine.

- Päringuehitaja töörist, mis kuvab kohe kehtiva GraphQL päringu koodi

Joonis 13 demonstreerib *Explorer* funktsiooni kasutamist, kus arendajad saavad intuitiivselt koostada keerukaid päringuid ja kohandada väljundeid.



Joonis 13. GraphiQL ülevaade: Explorer - päringuehitaja töörist

GraphiQL toetab kõiki vajalikke tööriistu, et muuta andmevahetus meeskondade vahel tõhusaks ja sujuvaks, täites samal ajal dokumenteerimisvahendi rolli, mis oli varem kaetud Swagger-iga.

3.2.7 Turvalisus

GraphQL API turvalisus integreeriti olemasolevasse Spring Security ja Commons teeki, mis oli algselt loodud REST API jaoks. Kohandused tagasid, et Commons toetaks nüüd ka GraphQL API päringute autentimist ja autoriseerimist, säilitades REST API-le sarnase turvamudeli. Eesmärk oli võimaldada kõigi päringute turvalisust ning introspektsioonipäringute lihtsustatud kasutamist.

Turvameetmed ja kohandused

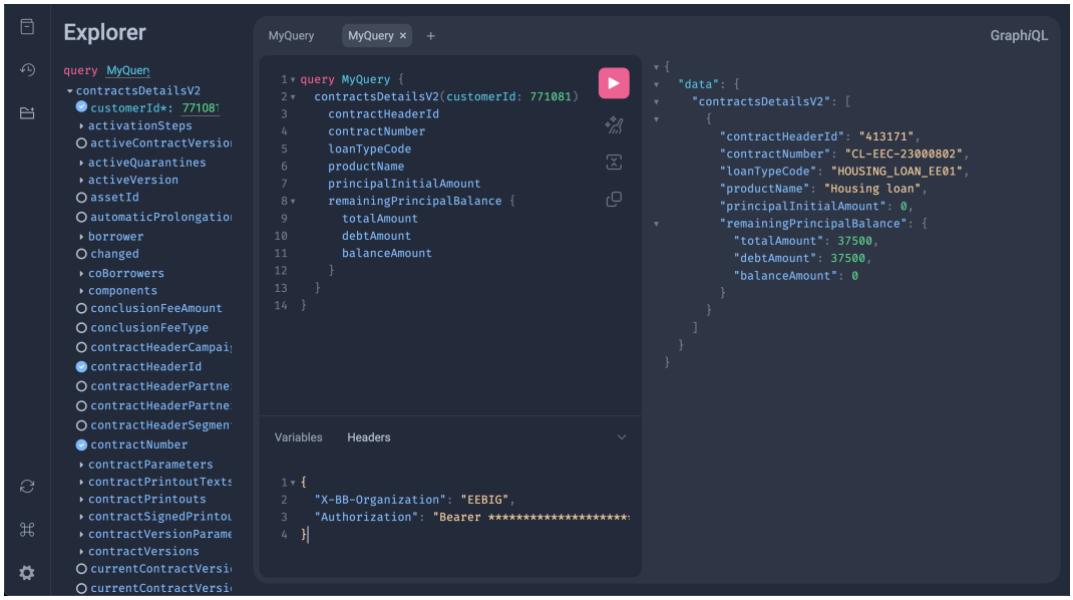
1. **GraphQL eraldi turvakonfiguratsioon:**

- Loodi konfiguratsioon, mis eristas GraphQL ja REST API päringuid. Introspektsioonipäringud (*__schema* ja *__type*) lubati ilma autentimiseta

2. Turvafiltrite kohandamine introspektsioonipäringutele:
    - Kohandati olemasolevaid filtreid, et tuvastada introspektsioonipäringud ja käsitleda neid eraldi.
3. Autentimine ja autoriseerimine teiste päringute jaoks:
    - Kõik muud GraphQL päringud nõudsid JWT (*JSON Web Token*) valideerimist ja kasutaja õiguste kontrolli.
4. Päringu valideerimine ja veakäsitlus:
    - Kui päring ei vastanud GraphQL nõutele, näiteks kui kasutati valet HTTP meetodit (ainult POST on lubatud), tagastati asjakohane veateade.

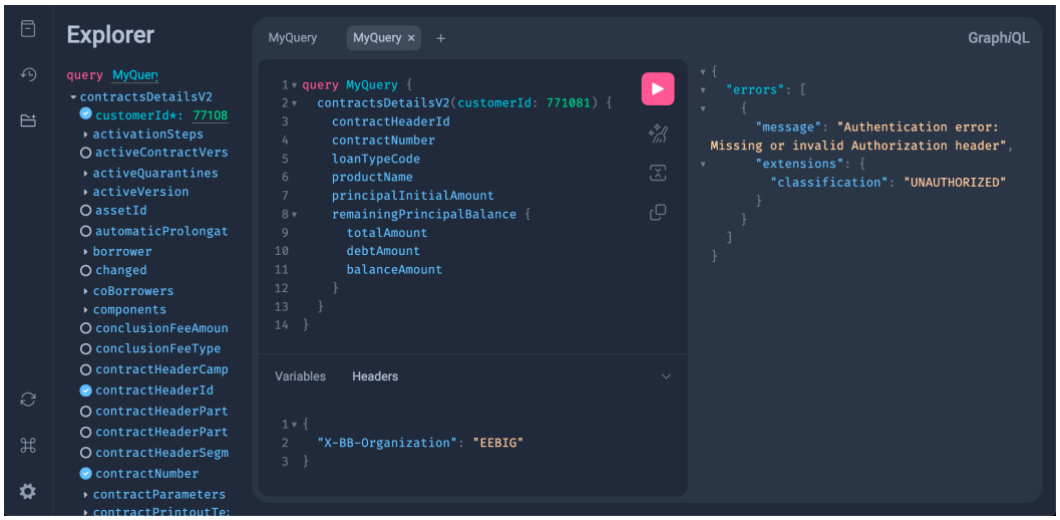
Visuaalsed näited turvalisusest:

Joonis 14 kuvab, kuidas kehtiv *Authorization* päis tagab korrektse vastuse GraphQL API kaudu.



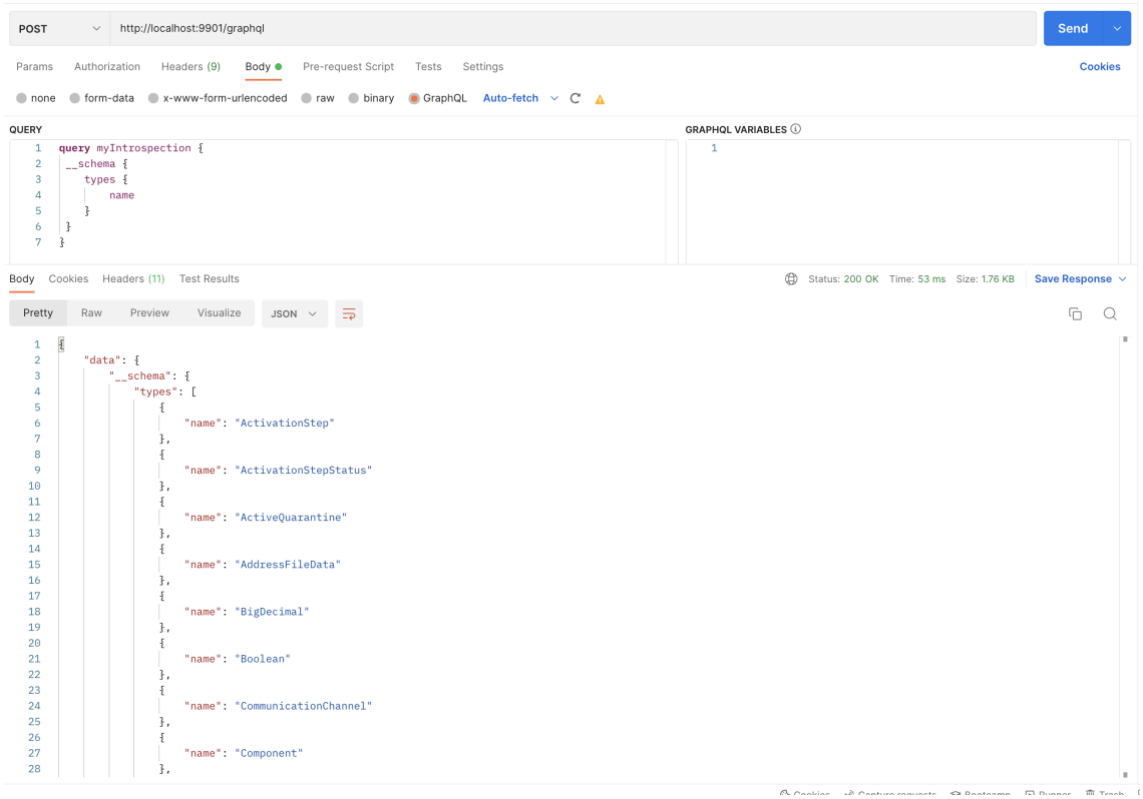
Joonis 14. GraphiQL ülevaada: JWT tokeniga päringu tulemused

Joonis 15 näitab, kuidas süsteem reageerib puudulikule autentimisele, andes selge teate, et päring nõuab kehtivat JWT-it.



Joonis 15. GraphiQL ülevaade: Puuduva Authorization headeri veateade

Joonis 16 esitab skeemi dokumentatsiooni päringu, mis on lubatud ka ilma autentimiseta, võimaldades ka teistele meeskondadele hõlpsat juurdepääsu skeemi detailidele.



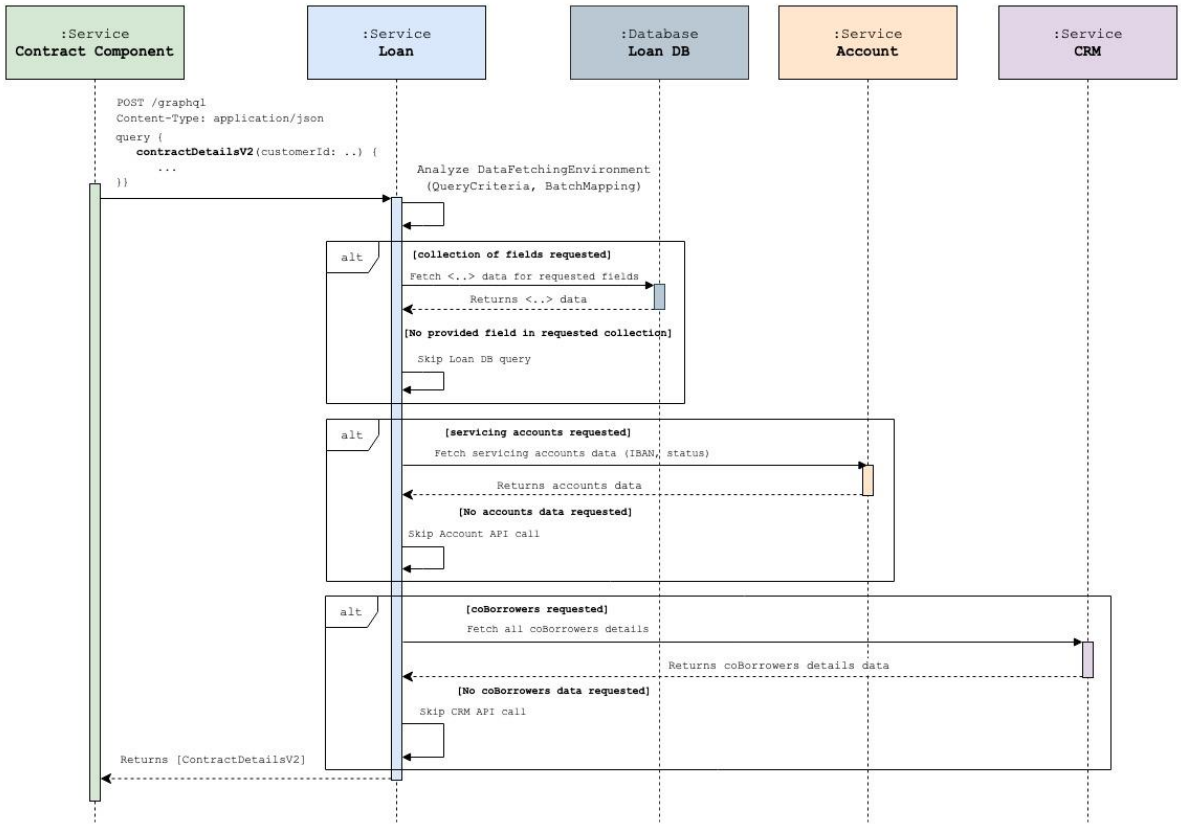
Joonis 16. GraphiQL ülevaade: Skeema inntrospektsioonipäringu avatus

3.2.8 Järjestusdiagramm

GraphQL API rakendamine tõi kaasa olulisi muudatusi andmevahetuses, pakkudes REST API-ga võrreldes tõhusamat ja paindlikumat lahendust.

Peamised erinevused REST ja GraphQL lahenduste vahel:

1. **Andmete koondamine:** REST API nõuab mitut päringut, samas kui GraphQL koondab kõik vajalikud andmed ühe päringu kaudu.
2. **Dünaamiline andmevalik**: GraphQL võimaldab kliendil küsida täpselt vajalikke andmeid, vältides liigset andmeedastust.
3. **Batch-päringud**: GraphQL kombineerib mitu päringut üheks optimeeritud API-kõneks, vähendades süsteemikoormust.



Joonis 17. Järjestusdiagramm: Selektiivne andmete laadimine ja teenustevaheline suhtlus GraphQL API kaudu

Järjestusdiagramm (Joonis 17.) näitab, kuidas GraphQL API vähendab andmevahetuse koormust ja parandab süsteemi jõudlust.

4 Tulemused ja edasised arendussuunad

4.1 Võrdlus ja järeldus: andmevahetuse efektiivsus ja jõudlus

Analüüs keskendus kõige keerukamale ja ressursimahukamale päringule, mida DCP meeskond teeb "loan" teenuse *contractsDetailsV2* lõpp-punktile. See päring nõuab suurt hulka andmeid ja peegeldab reaalset stsenaariumi, kus sama lõpp-punkti kasutatakse mitmesuguste ärivajaduste täitmiseks. Valik langes sellele päringule, kuna see pakub head ülevaadet REST ja GraphQL API-de käitumisest keeruliste ja andmemahukate operatsioonide puhul.

Testid viidi läbi *staging*-keskkonnas nelja erineva kliendi andmetega, kellel oli vastavalt **1, 10, 55 ja 159 lepingut**, simuleerimaks erinevaid andmemahtusid ja hinnates API-de töökindlust ning jõudlust. Üksikpäringute tegemiseks kasutati Postman-i, kus igale kliendile tehti 10 päringut. Vastuse aegade mõõtmine viidi läbi Kibana APM-i (*Application Performance Monitoring*) abil, mis võimaldas jälgida täpseid teenuse tasemel vastuse aegu. Saadud tulemused struktureeriti tabelitesse, mis võimaldavad analüüsida REST ja GraphQL API-de jõudlust ning võrrelda nende käitumist erinevates tingimustes.

4.1.1 Üksikpäringute tulemused

Alljärgnevad tabelid (Tabel 2 ja Tabel 3) illustreerivad REST API ja GraphQL API jõudluse tulemusi erinevate andmemahtude korral. Mõõdeti järgmised parameetrid:

- **Kõige kiirem vastus**: minimaalne vastusaeg (ms).
- **Kõige aeglasem vastus**: maksimaalne vastusaeg (ms).
- **Keskmine vastus**: kõigi päringute keskmine vastusaeg (ms).

Tabel 2. REST API päringute jõudluse tulemused

REST API			
Lepingute arv kliendil	Kõige kiirem vastus (ms)	Kõige aeglasem vastus (ms)	Keskmine vastus (ms)
1 leping	414	585	516
10 lepingut	661	1080	750
55 lepingut	2112	2417	2286
159 lepingut	5461	6953	5890

Tabel 3. GraphQL API päringute jõudluse tulemused

GraphQL API			
Lepingute arv kliendil	Kõige kiirem vastus (ms)	Kõige aeglasem vastus (ms)	Keskmine vastus (ms)
1 leping	269	329	290
10 lepingut	306	382	335
55 lepingut	1173	1454	1280
159 lepingut	2968	4040	3531

Tabel 4 detailselt illustreerib REST ja GraphQL API jõudluse suhtarve erinevate andmemahtude korral. Need suhtarvud on arvutatud, jagades REST API vastuse aja GraphQL API vastuse ajaga.

- **Kõige kiirem vastus (suhtarv)**: REST API minimaalne vastusaeg jagatud GraphQL API minimaalse vastusajaga.
- **Kõige aeglasem vastus (suhtarv)**: REST API maksimaalne vastusaeg jagatud GraphQL API maksimaalse vastusajaga.
- **Keskmine vastus (suhtarv)**: REST API keskmine vastusaeg jagatud GraphQL API keskmise vastusajaga.

Näiteks kliendi puhul, kellel on 1 leping:

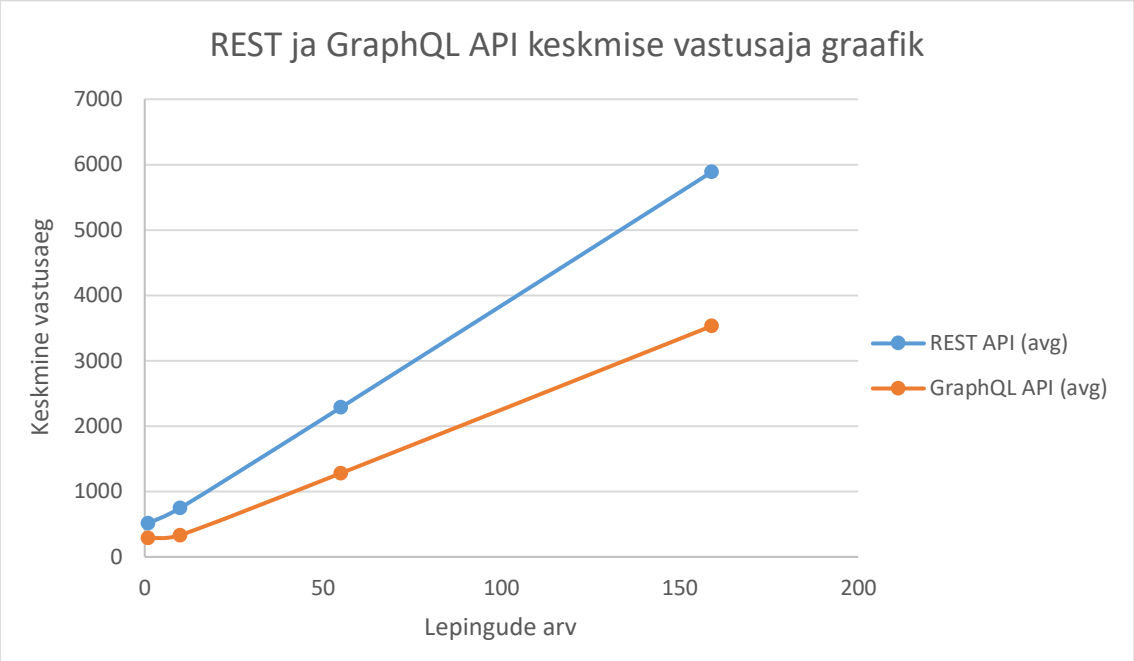
- **Keskmine vastus (suhtarv)**: 516 ms / 290 ms = 1,78 (ümardatud)

Tabel 4. GraphQL ja REST API jõudluse suhe erinevate andmemahtude korral

REST ja GraphQL API vastuse aegade suhtarvud			
Lepingute arv kliendil	Kõige kiirem vastus (suhtarv)	Kõige aeglasem vastus (suhtarv)	Keskmine vastus (suhtarv)
1 leping	1.54	1.78	1.78
10 lepingut	2.16	2.83	2.24
55 lepingut	1.80	1.66	1.79
159 lepingut	1.84	1.72	1.67

Tabel 4 näitajate põhjal arvutati keskmine suhtarv üle kõigi lepingute arvu kategooriate. Arvutuste kohaselt on GraphQL API keskmiselt 1,87 korda kiirem kui REST API, võttes arvesse erinevate andmemahtude keskmisi vastuse aegu.

4.1.2 Graafiline võrdlus ja järeldus



Joonis 18. Graafik: REST ja GraphQL API vastuse aegade võrdlus erinevate andmemahtude korral

Graafik joonises 18 illustreerib REST ja GraphQL API-de keskmiste vastusaegade erinevust nelja kliendi andmemahukuse põhjal. Graafikust on selgelt näha, et GraphQL API on igas andmemahtude kategoorias kiirem kui REST API. Väiksemate andmemahtude (1 ja 10 lepingut) puhul on GraphQL eelis küll märgatav, kuid suuremate andmemahtude korral (55 ja 159 lepingut) muutub erinevus eriti silmatorkavaks, kuna REST API vastuse aeg kasvab eksponentsiaalselt, samas kui GraphQL API puhul on kasv lineaarsem.

GraphQL API eelis tuleneb selle selektiivsest andmelaadimisest – klient saab määrata täpselt need andmed, mida ta vajab, mis vähendab nii päringute töötlemise aega serveris kui ka tagastatava andmemahu suurust. REST API puhul tagastatakse kõik ette määratud andmeväljad sõltumata kliendi tegelikest vajadustest, mis toob kaasa suurema andmemahtude töötlemise ja võrgukoormuse. See erinevus on eriti märgatav suurte andmemahukate päringute puhul, kus REST API vastuse aeg kasvab eksponentsiaalselt, samas kui GraphQL API jääb lineaarseks.

See seletab ka, miks GraphQL jõudlustestides 1.87 korda kiirem oli. Eelis ilmneb olukordades, kus REST API tagastab märkimisväärselt rohkem andmeid, kui kliendil

vaja, ja andmed sisaldavad keerukaid SQL-päringuid suurtest tabelitest. Kui mõlemad API-d küsiksid täpselt sama andmemahtu, oleks jõudluserinevus minimaalne, kuna mõlemad kasutavad HTTP-päringuid ja SQL-i.

Järedlus: GraphQL API eelised tulevad esile spetsiifilistes olukordades, kus klient vajab piiratud hulka andmeid suurtest andmestruktuuridest. Seega ei saa väita, et GraphQL on automaatselt kiirem, kuid selle paindlikkus andmepäringute kohandamisel teeb selle efektiivseks suure andmemahtude ja keerukate päringute korral.

4.2 Edasised arendussuunad ja tehnilised võimalused

Pärast REST ja GraphQL API-de analüüsi ning jõudlustestide tulemusi on ilmnenud mitmeid tehnilisi ja arhitektuurilisi võimalusi, mis võiksid parandada pangasüsteemi API infrastruktuuri tulevikus. Alljärgnevalt on toodud olulised suunad ja soovitused edasisteks arendusteks:

- **GraphQL API laiendamine:** Tuginedes jõudlustestide tulemustele, võiks kaaluda GraphQL API ulatuslikumat kasutuselevõttu, eriti süsteemides, kus on suur andmemaht ja keerulised päringustruktuurid. GraphQL API võimaldab tõhusamat andmevahetust ja paindlikkust klientrakenduste jaoks, vähendades tarbetut võrgu- ja andmebaasikoormust.

- **QueryCriteria optimeerimine:** Praegust mehhanismi saab täiustada, viies andmete laadimise loogika otse MyBatis-i päringutesse. See vähendaks eraldi SQL-päringute arvu ja parandaks süsteemi jõudlust.

- **Jõudlustestide automatiseerimine**: Grafana K6 või sarnaste tööriistade kasutamine võimaldaks automatiseeritud testimist ja reaalajas jõudluse jälgimist, muutes testimise tõhusamaks

- **Mikroteenuste optimeerimine**: GraphQL ja REST API-de kasutamise puhul on oluline optimeerida mikroteenuste taseme päringud. Näiteks "loan" teenuse puhul tuleks tagada, et *@BatchMapping* ja *QueryCriteria* mehhanismid teevad võimalikult vähe eraldi andmebaasi päringuid, kombineerides need vajadusel üheks tõhusaks päringuks.

- **Edasised analüüsid**: Tulevikus võiks keskenduda API-de kuluefektiivsuse ja skaleeritavuse analüüsidele production-keskkonnas. Eriti oluline on hinnata reaalsete kasutajate koormusel API-de toimivust ja vastupidavust, pakkudes lahendusi, mis on pikaajaliselt jätkusuutlikud ning vastavad ettevõtte kasvavatele vajadustele.

- **GraphQL API ökosüsteem**: Erinevate meeskondade GraphQL API-de ja skeemide koondamine ning korraldamine ühtseks ökosüsteemiks, et parandada integreeritavust ja lihtsustada haldamist.

Need sammud aitavad optimeerida pangasüsteemi API infrastruktuuri jõudluse, skaleeritavuse ja kasutajakogemuse parendamiseks.

5 Kokkuvõte

Lõputöö eesmärk oli analüüsida ja võrrelda GraphQL-i ja REST API-de praktilist juurutamist ja hooldamist pangasüsteemi näitel. Uuring keskendus küsimusele, kas GraphQL võiks parandada andmevahetuse efektiivsust, süsteemi hooldatavust ja arenduse paindlikkust võrreldes REST-iga.

Prototüübid ja jõudlustestid tõestasid, et GraphQL suudab keeruliste ja andmemahukate päringute korral pakkuda suuremat efektiivsust. Näiteks vähendas GraphQL API märkimisväärselt andmebaasipäringute arvu ja tarbetut andmevahetust, olles keskmiselt 1.87 korda kiirem kui REST API. Samas selgus, et GraphQL-i eelis võib väheneda olukordades, kus REST API tagastab juba vaikimisi kõik vajaminevad andmed. See näitab, et API arhitektuuri valik sõltub konkreetse kasutusjuhtumi vajadustest ja päringute optimeerimise tasemest.

Töö analüüsis ka tehnilisi väljakutseid GraphQL-i integreerimisel olemasolevasse REST-põhisesse infrastruktuuri. Lahendused hõlmasid turvalisuse rakendamist, skeemide loomist ja andmelaadimise optimeerimist, sealhulgas QueryCriteria ja BatchMapping mehhanismide täiustamist. Samuti toodi esile pangasüsteemi API infrastruktuuri optimeerimisvõimalused tulevikus.

Töö tulemusi saab rakendada ka teistes suurtes ja keerukates süsteemides, pakkudes väärtuslikke juhiseid API arhitektuuri valikul. GraphQL-i selektiivse andmelaadimise võimekus ja paindlikkus muudavad selle eriti sobivaks projektidesse, kus võrgu ja serveri koormuse vähendamine on kriitilise tähtsusega.

Edasised uurimissuunad võiksid keskenduda GraphQL-i ja REST API-de kulutõhususe ning skaleeritavuse hindamisele *production* keskkonnas. Lisaks on oluline analüüsida GraphQL-i mõju mikroteenuste ökosüsteemile suurte ja keeruliste süsteemide puhul. Töö tulemused annavad tugeva aluse organisatsioonidele, kes kaaluvad GraphQL-i kasutuselevõttu, pakkudes praktilisi lahendusi ja strateegilisi juhiseid.

Autor saavutas lõputöö eesmärgid, pakkudes põhjalikku võrdlust REST ja GraphQL API-de vahel ning esitades suunised, mis toetavad pangasüsteemi ja teiste valdkondade API infrastruktuuri arendamist ja optimeerimist.

Kasutatud kirjandus

[1] AWS, „Cloud Computing Concepts Hub: What is,“ [Võrgumaterjal]. Available: https://aws.amazon.com/what-is/. [Kasutatud 25.10.2024].

[2] E. Paraschiv, „The DAO Pattern in Java,“ [Võrgumaterjal]. Available: https://www.baeldung.com/java-dao-pattern. [Kasutatud 02.11.2024].

[3] A. Paraschiv, „The DTO Pattern (Data Transfer Object),“ [Võrgumaterjal]. Available: https://www.baeldung.com/java-dto-pattern. [Kasutatud 12.11.2024].

[4] Gatsby, „Introducing GraphiQL: What is GraphQL?,“ [Võrgumaterjal]. Available: https://www.gatsbyjs.com/docs/how-to/querying-data/running-queries-with-graphiql/. [Kasutatud 27.10.2024].

[5] „Learn GraphQL,“ [Võrgumaterjal]. Available: https://graphql.org/learn/. [Kasutatud 28.10.2024].

[6] Cloudflare, „What is HTTP?,“ [Võrgumaterjal]. Available: https://www.cloudflare.com/learning/ddos/glossary/hypertext-transfer-protocol-http/. [Kasutatud 10.11.2024].

[7] Eesti Pangaliit, „IBAN - Eesti rahvusvahelise kontonumbri standard,“ [Võrgumaterjal]. Available: https://pangaliit.ee/arveldused/iban. [Kasutatud 25.11.2023].

[8] JFrog, „What is Artifactory?,“ [Võrgumaterjal]. Available: https://jfrog.com/blog/what-is-artifactory-jfrog/. [Kasutatud 25.11.2024].

[9] Auth0, „Introduction to JSON Web Tokens,“ [Võrgumaterjal]. Available: https://jwt.io/introduction. [Kasutatud 20.11.2024].

[10] S. Ambler, „Agile Modeling. UML Sequence Diagrams: An Agile Introduction,“ [Võrgumaterjal]. Available: https://agilemodeling.com/artifacts/sequencediagram.htmd. [Kasutatud 21.10.2024].

[11] IBM, „What is three-tier architecture?,“ [Võrgumaterjal]. Available: https://www.ibm.com/topics/three-tier-architecture. [Kasutatud 05.11.2024].

[12] MDN, „Learn web development,“ [Võrgumaterjal]. Available: https://developer.mozilla.org/en-US/docs/Learn. [Kasutatud 12.11.2024].

[13] R. T. Fielding, „Architectural Styles and the Design of Network-based Software Architectures,“ 2000. [Võrgumaterjal]. Available: https://ics.uci.edu/~fielding/pubs/dissertation/top.htm. [Kasutatud 24.10.2024].

[14] AWS, „What’s the Difference Between GraphQL and REST?,“ [Võrgumaterjal]. Available: https://aws.amazon.com/compare/the-difference-between-graphql-and-rest/. [Kasutatud 10.10.2024].

[15] A. Marek ja D. Zhou, GraphQL with Java and Spring, 2023.

[16] G. Cox, „Getting Started with GraphQL and Spring Boot,“ [Võrgumaterjal]. Available: https://www.baeldung.com/spring-graphql. [Kasutatud 15.10.2024].

[17] Spring, „Spring GraphQL,“ [Võrgumaterjal]. Available: https://docs.spring.io/spring-graphql/reference/index.html. [Kasutatud 02.10.2024].

Lisa 1 – Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks[1]

Mina, Aleksandr Trofimov

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose "GraphQL ja REST API-de juurutamise ja hooldamise võrdlus pangasüsteemi näitel", mille juhendaja on Siim Rebane
   1.1. reprodutseerimiseks lõputöö säilitamise ja elektroonse avaldamise eesmärgil, sh Tallinna Tehnikaülikooli raamatukogu digikogusse lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
   1.2. üldsusele kättesaadavaks tegemiseks Tallinna Tehnikaülikooli veebikeskkonna kaudu, sealhulgas Tallinna Tehnikaülikooli raamatukogu digikogu kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.
2. Olen teadlik, et käesoleva lihtlitsentsi punktis 1 nimetatud õigused jäävad alles ka autorile.
3. Kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest ning muudest õigusaktidest tulenevaid õigusi.

30.11.2024

1 Lihtlitsents ei kehti juurdepääsupiirangu kehtivuse ajal vastavalt üliõpilase taotlusele lõputööle juurdepääsupiirangu kehtestamiseks, mis on allkirjastatud teaduskonna dekaani poolt, välja arvatud ülikooli õigus lõputööd reprodutseerida üksnes säilitamise eesmärgil. Kui lõputöö on loonud kaks või enam isikut oma ühise loomingulise tegevusega ning lõputöö kaas- või ühisautor(id) ei ole andnud lõputööd kaitsvale üliõpilasele kindlaksmääratud tähtajaks nõusolekut lõputöö reprodutseerimiseks ja avalikustamiseks vastavalt lihtlitsentsi punktidele 1.1. ja 1.2, siis lihtlitsents nimetatud tähtaja jooksul ei kehti.

Lisa 2 – REST API vastuse JSON näide

Otsustasin eraldada kogu JSON-i iga kihi põhjal. Joonis 5 näitab JSON-i vastuse esimest kihti:

```
{
  "interestRates": [{..}], // 1.
  "contractHeaderId": 0,
  "currentContractVersionId": 0,
  "currentContractVersionEndDate": "2024-11-10",
  "firstContractVersionId": 0,
  "activeContractVersionId": 0,
  "contractNumber": "string",
  "loanTypeCode": "LOANFI01",
  "productName": "string",
  "changed": true,
  "firstScheduledPaymentDate": "2024-11-10",
  "customerId": 0,
  "partnerId": 0,
  "assetId": 0,
  "servicingAccountId": 0,
  "servicingAccountIban": "string",
  "servicingAccountStatus": "string",
  "writeOffStatusCode": "IN_WRITE_OFF_LIST",
  "writeOffReasonCode": "string",
  "writeOffDescription": "string",
  "activeQuarantines": [{..}], // 2.
  "contractHeaderSegment": "string",
  "contractHeaderCampaignCode": "string",
  "contractHeaderPartnerBranchId": 0,
  "contractHeaderPartnerDealCreatorId": 0,
  "conclusionFeeType": "string",
  "conclusionFeeAmount": 0,
  "automaticProlongationBlockedReason": "string",
  "numberOfScheduleRows": 0,
  "signingMethod": "string",
  "isSigningRequired": true,
  "borrower": [{..}], // 3.
  "relationshipManager": "string",
  "loanType": {..}, // 4.
  "remainingPrincipalBalance": {..}, // 5.
  "activationSteps": [{..}], // 6.
  "contractPrintouts": [{..}], // 7.
  "contractSignedPrintouts": [{..}] // 8.
  "paymentInstructions": [{..}] // 9.,
  "scheduleDates": [{..}], // 10.
  "letters": [{..}], // 11.
  "provisionCalculations": [{..}], // 12.
  "customerRiskData": [{..}], // 13.
```

```
  "proceedings": [{..}], // 14.
  "contractParameters": [{..}], // 15.
  "contractVersionParameters": [{..}], // 16.
  "contractPrintoutTexts": [{..}], // 17.
  "representatives": [{..}], // 18.
  "interestConditionHistories": [{..}], // 19.
  "statusCodeForDisplay": "string",
  "penaltyComponents": [{..}], // 20.
  "nonPenaltyComponents": [{..}], // 21.
  "principalInitialAmount": 0,
  "interestInitialAmount": 0,
  "disbursement": true,
  "currentVersion": {..}, // 22.
  "activeVersion": {..}, // 23.
  "principalScheduleDefinitionTypeCode": "AUTOMATIC",
  "firstInterestPaymentDate": "2024-11-10",
  "firstPrincipalPaymentDate": "2024-11-10",
  "coBorrowers": [{..}], // 24.
  "contractVersions": [{..}], // 25.
  "components": [{..}] // 26.
}
```

Joonis 19. JSON vastuse struktuur

Alljärgnevalt on esitatud peamised teise taseme objektid koos nende struktuuriga:

1. **Objekt „interestRate“ sisu:**

```
{
  "interestRate": 0,
  "startDate": "2024-11-10",
  "excludedFromAnnuity": true,
  "numberOfPayments": 1,
  "interestType": "string",
  "euriborRate": 0
}
```

2. **Objekti sisu „activeQuarantines“ massiivist:**

```
{
  "id": 0,
  "contractHeaderId": 0,
  "customerId": 0,
  "quarantineStartDate": "2024-11-10",
  "quarantineEndDate": "2024-11-10",
  "quarantineType": "PROCEDURAL",
  "isActive": true,
  "orgCode": "string"
}
```

3. **Objekti sisu “borrower” massiivist:**

```
{
  "customerId": 0,
  "fullName": "string",
  "legalPrivateCode": "string",
  "identificationCountryCode": "string",
  "identificationCode": "string",
  "customerRelationId": 0,
  "communicationChannels": [
    {
      "communicationTypeCode": "string",
      "communicationTypeCodeValue": "string",
      "phoneAreaCode": "string",
      "contactData": "string",
      "contactPersonName": "string",
      "fullContactData": "string"
    }
  ]
}
```

4. **Objekt “loanType” sisu:**

```
{
  "code": "LOANFI01",
  "name": "string",
  "orgCode": "string",
  "description": "string",
  "initiationAllowed": true,
  "groupCode": "BUSINESS_LOAN",
  "productClassCode": "string",
  "productCategoryCode": "BUSINESS_LOAN",
  "migrated": true,
  "productRegistryCode": "string",
  "productTypeCode": "string",
  "legalPrivateCode": "string"
}
```

5. **Objekt “remainingPrincipalBalance” sisu:**

```
{
  "totalAmount": 0,
  "balanceAmount": 0,
  "debtAmount": 0
}
```

6. **Objekti sisu “activationSteps” massiivist:**


```
{
  "id": 0,
  "orgCode": "string",
  "contractVersionId": 0,
  "activationStepId": 0,
  "amount": 0,
  "vatAmount": 0,
  "currencyCode": "string",
  "statusCode": "PREPARED",
  "stepTypeCode": "FEE",
  "versionStatusCode": "string"
}
```


7. **Objekti sisu “contractPrintouts” massiivist:**


```
{
  "id": 0,
  "orgCode": "string",
  "contractVersionId": 0,
  "printoutTypeCode": "CONTRACT",
  "fileId": "string",
  "valid": true,
  "invalidatedDtime": "2024-11-10",
  "templateId": 0,
  "createdDtime": "2024-11-10T18:57:55.537Z",
  "manualGenerationAllowed": true,
  "languageCode": "string"
}
```


8. **Objekti sisu “contractSignedPrintouts” massiivist:**


```
{
  "id": 0,
  "versionId": 0,
  "signedDateTime": "2024-11-10T18:57:55.537Z",
  "signedPlaceRef": 0,
  "signedPlaceSourceName": "string",
  "witnessedByRef": 0,
  "witnessedBySourceName": "string",
  "customerId": 0,
  "fileId": "string",
  "signedBySourceName": "string",
  "signingMethod": "string",
  "archiveNumber": "string",
  "comment": "string",
  "customerRelationId": 0
}
```

9. Objekti sisu “paymentInstructions” massiivist:


```
{
  "id": 0,
  "orgCode": "string",
  "contractVersionId": 0,
  "customerId": 0,
  "sourceRef": "string",
  "sourceName": "string",
  "beneficiaryName": "string",
  "beneficiaryIban": "string",
  "amount": 0,
  "currencyCode": "string",
  "details": "string",
  "referenceNumber": "string",
  "processedDate": "2024-11-10",
  "copiedFromPaymentInstructionId": 0,
  "beneficiaryClearingNumber": "string",
  "beneficiaryAddress": "string",
  "accountOwner": "CUSTOMER_OTHER_BANK_ACCOUNT",
  "statusCode": "INSTANTLY",
  "instantPaymentAllowed": true,
  "plannedPaymentDate": "2024-11-10",
  "sendScheduleToCustomer": true,
  "refinancingContractNumber": "string",
  "channelCode": "string",
  "createdBy": "string",
  "createdDtime": "2024-11-10T18:57:55.537Z",
  "acceptedBy": "string",
  "acceptedDtime": "2024-11-10T18:57:55.537Z",
  "rejectedBy": "string",
  "rejectedDtime": "2024-11-10T18:57:55.537Z",
  "isEditable": true
}
```


10. Objekti sisu “scheduleDates” massiivist:


```
{
  "date": "2024-11-10",
  "payInt": true,
  "payPri": true,
  "principalPaymentAmount": 0,
  "regenerateFixed": true,
  "scheduleTypeCode": "ANNUITY",
  "scheduleRangeIndex": 0
}
```


11. Objekti sisu “letters” massiivist:

```
{
  "id": 0,
  "contractHeaderId": 0,
  "loanLetterTypeCode": "COLLECTION_LETTER",
  "customerId": 0,
  "contractNumber": "string",
  "currencyCode": "string",
  "sentDtime": "2024-11-10T18:57:55.537Z",
  "statusCode": "NEW",
  "contractOwner": true,
  "sendDueDate": "2024-11-10",
  "recipientType": "string",
  "suretyAgreementDate": "2024-11-10",
  "suretyAgreementNumber": "string",
  "externalStatusCode": "string",
  "error": "string",
  "letterNumber": "string",
  "refNumber": "string",
  "iban": "string",
  "paymentDate": "2024-11-10",
  "balanceAmount": 0,
  "principalAmount": 0,
  "interestAmount": 0,
  "administrationAmount": 0,
  "administrationFeeDebtAmount": 0,
  "conclusionAmount": 0,
  "otherAmount": 0,
  "obligationInterestAmount": 0,
  "totalAmount": 0,
  "identificationCode": "string",
  "firstName": "string",
  "lastName": "string",
  "languageCode": "string",
  "streetAddress": "string",
  "street": "string",
  "houseNumber": "string",
  "flatNumber": "string",
  "postcode": "string",
  "city": "string",
  "county": "string",
  "countryCode": "string",
  "sourceContractLetterDataId": 0,
  "fileId": "string",
  "feeCurrencyCode": "string",
  "letterFeeAmount": 0,
  "deliveryMethodCode": "POST",
  "email": "string",
  "penaltyRate": 0,
  "createdDtime": "2024-11-10T18:57:55.537Z",
  "penaltyAmount": 0,
  "recipientIntermediator": "string",
  "recipientAddress": "string",
  "channelCode": "string",
  "scheduleDebtAmount": 0,
  "createdBy": "string",
  "creationType": "AUTOMATIC",
  "messageId": "string",
  "numberOfCoborrowers": 0,
  "ownerDeliveryMethodCode": "POST",
```

```
  "coborrowerDeliveryMethodCode": "POST",
  "customerProductConnectionType": "CREDIT_CARD",
  "addressFileData": [
    {
      "id": 0,
      "contractLetterDataId": 0,
      "relatedContractHeaderId": 0,
      "streetAddress": "string",
      "street": "string",
      "houseNumber": "string",
      "flatNumber": "string",
      "postCode": "string",
      "city": "string",
      "county": "string",
      "countryCode": "string",
      "addressTypeCode": "string",
      "fileId": "string",
      "relatedContractDataJson": "string",
      "relatedContractNumber": "string",
      "relatedContractTerminationDate": "2024-11-10"
    }
  ],
  "letterGroupCode": "INVOICE",
  "loanLetterTypeCommonCode": "COLLECTION_LETTER",
  "description": "string",
  "vatAmount": 0,
  "vatRate": 0
}
```


12. Objekti sisu “provisionCalculations” massiivist:


```
{
  "id": 0,
  "provisionPeriod": "string",
  "contractHeaderId": 0,
  "componentTypeCode": "PRI",
  "componentType": "string",
  "effectiveDate": "2024-11-10",
  "provisionAmount": 0,
  "currencyCode": "string",
  "discountRate": 0,
  "stage": "1.0",
  "stageDelayDayGroup": "DELAY_0",
  "mobGroup": "MOB_1",
  "lgd": 0,
  "error": "string",
  "sentToFin": true,
  "statusCode": "INACTIVE",
  "customerId": 0,
  "auditId": "string",
  "orgCode": "string",
  "thirdStageSubGroup": "string",
  "delayDays": 0,
  "thresholdDelayDays": 0
}
```

13. Objekti sisu “customerRiskData” massiivist:


```
{
  "comment": "string",
  "customerId": 0,
  "id": 0,
  "parameterFormatCode": "NUMBER_WITH_ZERO",
  "parameterTypeCode": "BUCKET",
  "parameterValue": "string",
  "validFrom": "2024-11-10",
  "validTo": "2024-11-10"
}
```


14. Objekti sisu “proceedings” massiivist:


```
{
  "id": 0,
  "proceedingId": 0,
  "contractHeaderId": 0,
  "initialDebtAmount": 0,
  "initialDebtLastRegDate": "2024-11-10",
  "currencyCode": "string",
  "removedDate": "2024-11-10",
  "removeReasonCode": "string",
  "typeCode": "SALE",
  "statusCode": "ACTIVE",
  "startDate": "2024-11-10",
  "endDate": "2024-11-10",
  "proceedingStageCode": "string",
  "coefficient": 0
}
```


15. Objekti sisu “contractParameters” massiivist:


```
{
  "id": 0,
  "contractHeaderId": 0,
  "contractParameterTypeCode": "COLLATERAL_TYPE",
  "validityRange": {
    "startDate": "2024-11-10",
    "endDate": "2024-11-10"
  },
  "valueText": "string",
  "valueNumber": 0,
  "valueDate": "2024-11-10",
  "valueTime": "2024-11-10T18:57:55.537Z",
  "valueBoolean": true,
  "lookupCode": "string",
  "valueType": "string"
}
```

16. Objekti sisu “contractVersionParameters” massiivist:


```
{
  "id": 0,
  "contractVersionId": 0,
  "contractParameterTypeCode": "COLLATERAL_TYPE",
  "validityRange": {
    "startDate": "2024-11-10",
    "endDate": "2024-11-10"
  },
  "valueText": "string",
  "valueNumber": 0,
  "valueDate": "2024-11-10",
  "valueTime": "2024-11-10T18:57:55.537Z",
  "valueBoolean": true,
  "lookupCode": "string"
}
```


17. Objekti sisu “contractPrintoutTexts” massiivist:


```
{
  "id": 0,
  "contractVersionId": 0,
  "fragmentCode": "string",
  "text": "string",
  "paragraphNumber": "string",
  "dueDate": "2024-11-10"
}
```


18. Objekti sisu “representatives” massiivist:

Sama stuktuur nagu objektis “borrower” massiivist.

19. Objekti sisu “interestConditionHistories” massiivist:


```
{
  "id": 0,
  "orgCode": "string",
  "componentTypeCode": "PRI",
  "validityEffectiveDateFrom": "2024-11-10",
  "validityEffectiveDateTo": "2024-11-10",
  "rate": 0,
  "calculationDaysMonth": "string",
  "calculationDaysYear": "string",
  "excludedFromAnnuity": true
}
```


20. Objekti sisu “penaltyComponents” massiivist:

```
{
  "scheduleTypeCode": "ANNUITY",
  "scheduleDefinitionTypeCode": "AUTOMATIC",
  "scheduleStartDate": "2024-11-10",
  "initialAmount": 0,
  "id": 0,
  "orgCode": "string",
  "contractVersionId": 0,
  "componentTypeCode": "PRI",
  "currencyCode": "string",
  "balanceAmount": 0,
  "debtAmount": 0,
  "debtStartDate": "2024-11-10",
  "adjustedDebtStartDate": "2024-11-10",
  "calculationDaysMonth": "string",
  "calculationDaysYear": "string",
  "rateBaseCode": "string",
  "rate": 0,
  "baseRate": 0,
  "marginRate": 0,
  "fixingDate": "2024-11-10",
  "nextFixingDate": "2024-11-10",
  "accruedUntilDate": "2024-11-10",
  "paymentIntervalMonths": 0,
  "paymentDay": 0,
  "regularAmount": 0,
  "baseInvalidFromDate": "2024-11-10",
  "calculationStoppedFromDate": "2024-11-10",
  "modifiedDtime": "2024-11-10T18:57:55.537Z",
  "componentType": "string",
  "scheduleType": "string",
  "componentTypeInterestLike": true,
  "componentTypeIsPenalty": true,
  "productComponentSchedular": true,
  "componentTypeFeeLike": true,
  "componentPrincipalLike": true,
  "pendingTransactionsAmount": 0,
  "debtDays": 0,
  "adjustedDebtDays": 0,
  "scheduleTypeGenerationTypeCode": "string",
  "invoiceable": true,
  "rangeInitialAmount": 0,
  "rangeResidualAmount": 0,
  "rangeExcludeBulletFromSchedule": true
}
```


21. Objekti sisu “nonPenaltyComponents” massiivist:

Sama stuktuur nagu objektis “penaltyComponents” massiivist.

22. Objekti “currentVersion” sisu:


```
{
  "id": 0,
  "orgCode": "string",
  "contractHeaderId": 0,
  "versionNumber": 0,
```

```
"changeTypeCode": "APPENDIX",
"statusCode": "ACTIVE",
"currencyCode": "string",
"limitAmount": 0,
"plannedActivationDate": "2024-11-10",
"factualActivationDate": "2024-11-10",
"plannedSigningDate": "2024-11-10",
"signingDate": "2024-11-10",
"endDate": "2024-11-10",
"gracePeriodMonths": 0,
"gracePeriodType": "PRINCIPAL",
"numberOfPayments": 0,
"nrOfSuperAnnuityPayments": 0,
"factualEndDate": "2024-11-10",
"invalidatedDate": "2024-11-10",
"sourceName": "string",
"sourceRef": "string",
"apr": 0,
"terminationDate": "2024-11-10",
"terminationReasonCode": "string",
"terminationComment": "string",
"channelCode": "string",
"replacedByContractVersionId": 0,
"createdBy": "string",
"feeCollectionType": "string",
"annuityAmount": 0,
"totalPaymentAmount": 0,
"restructuringDate": "2024-11-10",
"appendixNumber": 0,
"inWriteOffList": true,
"writeOffDate": "2024-11-10",
"maturityPercentage": 0,
"disbursementPeriodEndDate": "2024-11-10",
"lgd": 0,
"lgdSetBy": "DEFAULT",
"partnerId": 0,
"partnerBranchId": 0,
"partnerDealCreatorId": 0,
"preserveSchedule": true,
"maturityAmount": 0,
"activationDate": "2024-11-10",
"gracePeriodMonthlyPaymentAmount": 0,
"createdDtime": "2024-11-10T18:57:55.537Z",
"interestRate": 0,
"marginRate": 0,
"euriborRate": 0,
"rateBaseCode": "string",
"interestRateCalculationAlgorithm": "string",
"paymentDay": 0,
"outstandingAmount": 0,
"remainingPeriod": 0,
"debtAmount": 0,
"debtDays": 0,
"adjustedDebtDays": 0,
"nextPaymentDate": "2024-11-10",
"nextPaymentAmount": 0,
"nextEuriborFixingDate": "2024-11-10",
"feeAmount": 0,
"activatedBy": "string",
```

```
  "diminishedObligation": 0,
  "diminishedObligationErrorReason": "string",
  "closed": true,
  "active": true,
  "signed": true,
  "terminated": true,
  "applicationBased": true,
  "restructuring": true,
  "prepared": true,
  "draft": true,
  "annulled": true,
  "replaced": true
}
```


23. Objekti “activeVersion” sisu:

Sama stuktuur nagu “currentVersion” objektis.

24. Objekti sisu “coBorrowers” massiivist:

Sama stuktuur nagu objektis “borrower” massiivist.

25. Objekti sisu “contractVersions” massiivist:

Sama struktuur nagu “currentVersion” objektis.

26. Objekti sisu “components” massiivist:

Sama struktuur nagu “penaltyComponent” objektis.

Lisa 3 – GraphQL skeemid

1. **operations.graphqls** sisu**:**

```
type Query {
    contractDetailsV2(customerId: Long!): [ContractDetailsV2]
}
```

2. **operations.graphqls** sisu**:**

```
type ContractDetailsV2 {
    interestRates: [InterestRate],
    contractHeaderId: ID,
    currentContractVersionId: Int,
    currentContractVersionEndDate: LocalDate,
    firstContractVersionId: ID,
    activeContractVersionId: ID,
    contractNumber: String,
    loanTypeCode: String,
    productName: String,
    changed: Boolean,
    firstScheduledPaymentDate: LocalDate,
    customerId: ID,
    partnerId: ID,
    assetId: ID,
    servicingAccountId: ID,
    servicingAccountIban: String,
    servicingAccountStatus: String,
    writeOffStatusCode: String,
    writeOffReasonCode: String,
    writeOffDescription: String,
    activeQuarantines: [ActiveQuarantine],
    contractHeaderSegment: String,
    contractHeaderCampaignCode: String,
    contractHeaderPartnerBranchId: ID,
    contractHeaderPartnerDealCreatorId: ID,
    conclusionFeeType: String,
    conclusionFeeAmount: BigDecimal,
    automaticProlongationBlockedReason: String,
    numberOfScheduleRows: Int,
    signingMethod: String,
    isSigningRequired: Boolean,
    borrower: ContractPartyDto,
    relationshipManager: String,
    loanType: LoanType,
    remainingPrincipalBalance: RemainingPrincipalBalance,
    activationSteps: [ActivationStep],
    contractPrintouts: [ContractPrintout],
    contractSignedPrintouts: [ContractSignedPrintout],
    paymentInstructions: [PaymentInstruction],
    scheduleDates: [ScheduleDate],
    letters: [Letter],
    provisionCalculations: [ProvisionCalculation],
    customerRiskData: [CustomerRiskData],
    proceedings: [Proceeding],
```

```
    contractParameters: [ContractParameter],
    contractVersionParameters: [ContractVersionParameter],
    contractPrintoutTexts: [ContractPrintoutText],
    representatives: [ContractPartyDto],
    interestConditionHistories: [InterestConditionHistory],
    statusCodeForDisplay: String,
    penaltyComponents: [Component],
    nonPenaltyComponents: [Component],
    principalInitialAmount: BigDecimal,
    interestInitialAmount: BigDecimal,
    disbursement: Boolean,
    currentVersion: ContractVersionDetails,
    activeVersion: ContractVersionDetails,
    principalScheduleDefinitionTypeCode:
ScheduleDefinitionTypeCode,
    firstInterestPaymentDate: LocalDate,
    firstPrincipalPaymentDate: LocalDate,
    coBorrowerDetails: [ContractPartyDto!],
    contractVersions: [ContractVersionDetails],
    components: [Component]
}
```

3. **enums.graphqls** sisu**:**

```
enum ContractStatus {
    ACTIVE, ANNULLED, DRAFT, PREPARED, REPLACED, SIGNED, CLOSED
}

enum ScheduleTypeCode {
    ANNUITY,
    SUPER_ANNUITY,
    SUPER_ANNUITY_ADJUSTED,
    AMORTISATION,
    FOLLOWFIX,
    ACT,
    ANNUITY_ADJUSTED,
    ANNUITY_ADJUSTED_VAT,
    AMORTISATION_ADJUSTED,
    MANUAL,
    FOLLOWVAT,
    NO_SCHEDULE,
    FOLLOWASSET,
    MANUAL_ADJUSTED
}

enum ScheduleDefinitionTypeCode {
    AUTOMATIC,
    FIRST_DATES,
    ALL_DATES,
    ALL_DATES_PRINCIPAL_AMOUNTS
}

enum QuarantineType {
    PROCEDURAL, FORBORNE, NON_PERFORMING, STAGE_3_PROBATION
}

enum LetterStatusCode {
    NEW,
```

```
    PENDING,
    SENT,
    ERROR,
    RESENT,
    ANNULLED,
    CREDITED,
    DRAFT
}

enum DeliveryMethodCode {
    POST,
    EMAIL,
    SMS,
    EINVOICE,
    INVOICE,
    DIRECT_DEBIT,
    EMAIL_OR_POST,
    EMAIL_OR_SMS,
    EMAIL_AND_POST
}

enum LetterCreationType {
    AUTOMATIC, MANUAL
}

enum CustomerProductConnectionType {
    CREDIT_CARD, CREDIT_CARD_SMALL_LOAN,
CREDIT_CARD_HOUSING_COAPPLICANT
}

enum ProvisionStatusCode {
    INACTIVE, ACTIVE, DRAFT, ERROR
}

enum StageDelayDayGroup {
    DELAY_0, DELAY_30D, DELAY_60D, DELAY_90D
}

enum ActivationStepStatus {
    PREPARED, PROCESSED, REVERSED
}
```

4. **scalars.graphqls** sisu**:**

```
scalar DateTime
scalar BigDecimal
scalar Long
scalar LocalDate
scalar Timestamp
```

Lisa 4 – GraphQL API päringu skeem

```
query MyQuery {
  contractsDetailsV2(customerId: 2108693) {
    contractHeaderId,
    contractNumber,
    firstContractVersionId,
    activeContractVersionId,
    currentContractVersionEndDate,
    productName,
    loanTypeCode,
    partnerId,
    changed,
    servicingAccountId,
    servicingAccountIban,
    servicingAccountStatus,
    isSigningRequired,
    conclusionFeeType,
    conclusionFeeAmount,
    numberOfScheduleRows,
    activeVersion {
        remainingPeriod,
        debtAmount,
        debtDays,
        statusCode,
        terminated
    },
    currentVersion {
        remainingPeriod,
        debtAmount,
        debtDays,
        statusCode,
        terminated
    },
    coBorrowerDetails {
        fullName
    },
    proceedings {
        proceedingStageCode
    },
    nonPenaltyComponents {
        balanceAmount
    }
    borrower {
      fullName,
      identificationCode
    }
    loanType {
      code,
```

```
      nmae,
      groupCode,
      productTypeCode
    },
    components {
      componentType,
      balanceAmount,
      debtAmount,
      initialAmount,
      rate,
      rateBaseCode,
      baseRate,
      marginRate,
      nextFixingDate
    },
    paymentInstructions {
        contractVersionId,
        beneficiaryName,
        beneficiaryIban,
        amount,
        accountOwner,
        currencyCode
    }
  }
}
```