TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Ian Erik Varatalu
179699 IABB

GRAPHQL TEHNOLOOGIALE ÜLEMINEK OLEMASOLEVA ANDMEBAASIGA LAHENDUSE NÄITEL

Bakalaureusetöö

Juhendaja: Tarvo Treier
MSc

Tallinn 2020

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: Ian Erik Varatalu

07.03.2020

Annotatsioon

Antud lõputöö eesmärgiks oli viia üks suur aegunud arhitektuuriga rakendus koos andmebaasiga üle GraphQL tehnoloogiale ning leida ja näidata võtteid, millega lahendada koodis üleminekuga kaasnevaid probleeme.

Lõputöö realisatsiooni käigus sai keskseks teemaks töö suure mahukuse tõttu automatiseerimine ning koodi taaskasutus.

Realisatsiooni tulemusel tekkis GraphQL serveriprogramm, mis täidab samu operatsioone mis olemasolev rakendus, aga veebikeskkonnas.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 58 leheküljel, 7 peatükki, 6 joonist, 68 pilti.

Abstract

Moving to GraphQL technology on the example of a pre-existing database

The aim of this thesis was to move an architecturally outdated application with its existing database to GraphQL technology and to find and show the techniques used to solve the rising problems in code that were found during the process.

During the realization of this thesis, automation and code reusage became the most important parts, because of the size of the project.

The end product was a GraphQL application programming interface, that performs the same operations as the existing application in a web environment.

The thesis is in estonian and contains 58 pages of text, 7 chapters, 6 figures, 68 images.

Lühendite ja mõistete sõnastik

API	*Application Program Interface*
REST	*REpresentational State Transfer,* veebiressurside liides
GraphQL	Päringukeel
CRUD	*Create, Read, Update, Delete,* andmebaasioperatsioonid
HTTP	*Hypertext Transfer Protocol,* veebiliikluse edastusprotokoll
URL	*Uniform Resource Locator,* internetiaadress
Field	GraphQL Type küljes olev väli
Query	GraphQL päringuoperatsioon
Mutation	GraphQL *Create*, *Update*, *Delete* operatsioonitüüp
Type	GraphQL võrgulüli, objektitüüp
Argument	GraphQL operatsiooni täpsustuseks mõeldud konkreetne parameeter
Directive	GraphQL operatsiooni täpsustuseks mõeldud üldine parameeter
Dictionary	C# andmetüüp
Resolve	GraphQL Field-i väärtuse allikas või vastav delegaat
Repository	Koht, kus hoitakse tarkvaras keerulist päringuloogikat taaskasutuseesmärkidel
SQL	*Structured Query Language*
Pistikprogramm	*Plugin*
Postgres, PostgreSQL	Andmebaasisüsteem

Sisukord

1 Sissejuhatus ... 13
2 Olemasolev lahendus ... 15
2.1 .NET Framework raamistikult üleminek .NET Core raamistikule ... 16
2.2 Päringute loogika säilitamine ... 16
2.3 Päringute jõudlus üle veebi ... 16
2.4 Suur andmebaasi skeem ning palju koodi ... 17
2.5 Olemasoleva rakendusega uuendustega kaasas püsimine ... 17
3 GraphQL ... 18
3.1 GraphQL-i arhitektuur ... 18
3.1.1 Operatsiooni põhi ... 20
3.1.2 Query Operatsioon ... 21
3.2 GraphQL .NET raamistiku lühiülevaade ... 22
3.2.1 Type ... 23
3.2.2 Query ... 24
4 GraphQL API realiseerimisel leitud probleemide analüüsimine ning lahendamine 25
4.1 API Arhitektuuri planeerimine ... 25
4.2 Olemasolevalt rakenduselt migreerimise probleemid ja lahendused ... 27
4.2.1 Klasside loomine andmebaasi põhjal ... 27
4.2.2 Muutuva andmebaasimudeli peale koodi ehitamine ... 29
4.2.3 SQL Vaated Entity Frameworkis ... 31
4.3 Üldisemad GraphQL projekti struktuuri ning valitud raamistiku probleemid 32
4.3.1 GraphQL põhja ülesehitus ja tükeldamine ... 33
4.3.2 *Dependency Injection* arhitektuurimuster ning selle skaleeritavus ... 34
4.3.3 GraphQL tüübi loomine ... 35
4.3.4 GraphQL väljade taaskasutus koodis ... 37
4.3.5 Päringu täpsustamine ... 40
4.3.6 Päringu vaikimisi filtrid ning *directive* tüüp ... 42
4.4 CRUD lahendamine ... 44

4.4.1 *Input Object Type* loomine ... 45
4.4.2 Mitme CRUD operatsiooni ühe päringuga tegemine ... 45
4.4.3 Mitme CRUD operatsiooni optimeerimine ... 47
4.4.4 Olemasolevate objektidega seoste loomine CRUD operatsioonides ... 48
4.4.5 Uuendusoperatsioonid Mutation operatsiooni keskel ... 49
4.4.6 Automaatselt seotud objektide ja listide genereerimine Input Object Type jaoks 49
4.4.7 Vaikimisi CRUD operatsioonid GraphQL väljadena ... 52
4.4.8 GraphQL väljas sisalduvate objektidega vaikimisi CRUD ... 53
4.4.9 CRUD täielikult automatiseerimine ... 55
4.5 Jõudluse probleemid ja lahendused ... 56
4.5.1 Partii korraga laadimine ... 57
4.5.2 SQL optimisatsioonid ... 58
4.5.3 *Unit of Work* eksemplaride taaskasutus ... 58
4.5.4 Asünkroonsed väljad / Asünkroonsed andmebaasioperatsioonid ... 59
5 Tulemused ja valideerimine ... 61
5.1 PostGraphile lahendusega võrdlus ... 62
5.1.1 Päringute arvu vähendamine ning N+1 probleem ... 63
5.1.2 GraphQL *Schema* laiendamine ... 64
5.1.3 Andmebaasi põhjal GraphQL *Schema* automaatne loomine ... 64
5.1.4 CRUD ... 65
5.1.5 Filtrite kasutamine, *Argument-id* ... 65
6 Millele edasi mõelda ... 66
6.1 Potentsiaalselt raamistiku muutmine ... 66
6.2 Päringute valiku piiramine ning päringu enda suuruse vähendamine veebiliikluse jaoks ... 67
6.3 Pahatahtlike päringute piiramine / DDOS kaitse ... 68
6.4 Testimine ... 68
7 Kokkuvõte ... 71
Kasutatud kirjandus ... 72
Lisa 1 – SQL Script C# klasside loomiseks ... 74
Lisa 2 – Commitide logi, koodi meetrika ... 75

Jooniste loetelu

Joonis 1 GraphQL Schema näide 19
Joonis 2 API lai arhitektuur 26
Joonis 3 Olemasolevalt rakenduselt migreerimise probleemid ja lahendused 27
Joonis 4 GraphQL kihi ning projekti ülesehitusega seotud probleemid ning nendest mõjutatud arhitektuuriosad 33
Joonis 5 CRUD operatsioonidega seotud probleemid ning nende ülesehitus alates väiksemate probleemide lahendamisest. 45
Joonis 6 Jõudlusega seotud probleemid ning vastavad arhitektuurikihid 56

Ekraanipiltide loetelu

Ekraanipilt 1 PAKK töölauarakenduse kasutajaliides 15
Ekraanipilt 2 Näide GraphQL päringust GraphQL-Playground keskkonnas 20
Ekraanipilt 3 PersonQuery 21
Ekraanipilt 4 GraphQL Playground Argument-ide lisamise näidis 22
Ekraanipilt 5 Object Type sees väljade valiku näide 22
Ekraanipilt 6 GraphQL .NET Field näide 23
Ekraanipilt 7 CarType Manufacturer edasine täpsustamine 23
Ekraanipilt 8 PersonQuery ning people juurpäring 24
Ekraanipilt 9 Näide lihtsamast SSMS genereeritud diagrammist 28
Ekraanipilt 10 SQL scripti tulemus, et luua tabelitele vastavad klassid 29
Ekraanipilt 11 Genereeritud klassi originaalne fail 30
Ekraanipilt 12 Genereeritud klassi laiendused läbi *partial* võtmesõna 30
Ekraanipilt 13 Genereeritud koodist päritud ApplicationDbContext 30
Ekraanipilt 14 View kaardistatud C# klassilt andmebaasitabeli väljadele läbi EntityFrameworki 31
Ekraanipilt 15 CardTransactioni jaoks Viewi laiendusklass 31
Ekraanipilt 16 Viewi navigatsiooniväljade kasutamine EntityFrameworkCore raamistikus filterdamiseks 32
Ekraanipilt 17 GraphQL Schema tükeldatav lahendus 34
Ekraanipilt 18 3.1 punktis toodud skeemi Type Dependency Injection 35
Ekraanipilt 19 Dependency Injection automatiseeritud Namespace põhjal 35
Ekraanipilt 20 Minimaalsete nõuetega GraphQL .NET raamistiku Field 36
Ekraanipilt 21 GraphQL .NET FieldBuilder klassi võimalused 36
Ekraanipilt 22 AutoObjectGraphType kood 37
Ekraanipilt 23 Field seose näide teise Typega 38
Ekraanipilt 24 Company Field ning selle Argumendid ja Resolve delegaat 38
Ekraanipilt 25 Refaktooritud Company Fieldi meetod 39
Ekraanipilt 26 Expressioniga meetod Company Fieldist 39
Ekraanipilt 27 Argumentide ehk filtrite näide GraphQL kihis 40

Ekraanipilt 28 Argumentide rakendamiseks loodud loopi näide 40
Ekraanipilt 29 GraphQL filtri klass 41
Ekraanipilt 30 Company päringu refaktooritud filtrid 41
Ekraanipilt 31 Refaktooritud filtrite rakendusmeetodid 42
Ekraanipilt 32 Listi ning üksiku vaste jaoks eraldi päringud GraphQLis 42
Ekraanipilt 33 Vaikimisi filtritega täpsustatud päringu näide 43
Ekraanipilt 34 Päring ilma Directive-ta 44
Ekraanipilt 35 Päring koos toUpper Directive-ga 44
Ekraanipilt 36 ToUpper Middleware näidis graphql-dotnet raamistikus 44
Ekraanipilt 37 graphql-dotnet lihtne Mutation operatsiooni näide 46
Ekraanipilt 38 graphql-dotnet InputObjectGraphType näide 46
Ekraanipilt 39 Mitme Create Mutation operatsiooni näide 47
Ekraanipilt 40 Mitme Create Mutation operatsioon Repository kihis 47
Ekraanipilt 41 Person võrgustiku lisamine läbi friends seose 48
Ekraanipilt 42 Uue ning olemasoleva seose lisamine samas Mutationis 48
Ekraanipilt 43 Uue ning olemasoleva seose kood Repository kihis 48
Ekraanipilt 44 Sisemise objektiseosega Update näidis 49
Ekraanipilt 45 Create ja Update samas GraphQL operatsioonis 49
Ekraanipilt 46 Seotud väljadega automatiseeritud klass 50
Ekraanipilt 47 Mudeliklasside vaheliste seoste läbikäimiseks loodud meetodid 50
Ekraanipilt 48 CRUD automatiseerimisele erandite lisamine ning nameof() meetod, et võimaldada väljaspool koodimuutusi 51
Ekraanipilt 49 Create ja Update jaoks loodud listi ja üksikobjekti meetodid 51
Ekraanipilt 50 Mõned automaatselt lisatud Company listiseosed 52
Ekraanipilt 51 Dünaamilise Create meetodi näide 53
Ekraanipilt 52 Ilma tüübita graphql-dotnet Create Field-i sisu 53
Ekraanipilt 53 GraphQL väljas sisalduvate objektidega vaikimisi CRUD 54
Ekraanipilt 54 Vaikimisi listide Create ja Update meetod 54
Ekraanipilt 55 Tavaliste üksikobjektide Create ja Update meetod 55
Ekraanipilt 56 Namespace põhjal automatiseeritud CRUD 56
Ekraanipilt 57 graphql-dotnet DataLoader implementatsiooni näide 57
Ekraanipilt 58 ILookup Repository kihis 57
Ekraanipilt 59 Dünaamiline Select lause koostamine vajalikest väljadest 58
Ekraanipilt 60 DbContext pooling näidis 59

Ekraanipilt 61 Asünkroonse Repository meetodi näide 60
Ekraanipilt 62 ResolveAsync näide Fieldil 60
Ekraanipilt 63 Realisatsiooni tulemuse kasutajaliides... 61
Ekraanipilt 64 PostGraphile filterdamise näide... 66
Ekraanipilt 65 Terve GraphQL Schema autoriseerimise nõude näidis 68
Ekraanipilt 66 Middleware loomise näide.. 69
Ekraanipilt 67 HttpContexti kaudu päringu simuleerimine.. 69
Ekraanipilt 68 Postman keskkonnas päringu näide 70

1 Sissejuhatus

Praegu on uute tarkvaralahenduste puhul levinud pilvepõhised veebirakendused, sest nende kasutamisel on rida eeliseid nii kliendi kui ka arendaja poolel. Ometi on olemas palju kasutuselolevaid rakendusi, mis ei ole veel veebi viidud. Sellel on mitmeid põhjuseid, aga üks suuremaid neist on see, et rakendus on liiga mahukaks kasvanud ja üleviimine on kulukas ja aeganõudev. Ühe sellise lahenduse üleviimisel veebi puutus töö autor kokku tööalaselt ja antud lõputöös vaatamegi lähemalt neid võtteid, mille abil õnnestus selline rakendus moderniseerida.

Lõputöö raames moderniseeritav rakendus on monoliitse arhitektuuriga töölauarakendus, mis sisaldab nii kliendisuhete haldust (CRM, Customer Relationship Management), ettevõtte ressursside planeerimist (ERP, Enterprise Resource Planning) kui ka palju muud. Lõputöö kirjutamise ajal oli see rakendus kasutusel kolmes riigis ja neljas erinevas ettevõttes. Rakendus on enda vajalikkust aastatega tõestanud, kuid ajaga kaasas käimiseks on vaja rakenduse arhitektuuri muuta ja arendada juurde rakendusele programmiliides (API, Application Program Interface) ja veebikeskkonda kasutajaliides, mis oleks ühine nii praegustele kui ka tulevastele ettevõtetele, kes seda kasutada tahavad. Kuna rakenduse funktsionaalsus on suur ja seda kasutavate ettevõtete jaoks tuleb tagada rakenduse toimimine kõikidel tööpäevadel, siis on mõeldav ainult järk-järguline üleminek nii, et veebiliidesele üle viidavad osad jäävad veel vähemalt mõneks ajaks paralleelselt toimima töölauarakenduses.

Uue projekti arendusmeeskonna põhiliikmeid oli kaks: käesoleva töö autor ja tema kursusekaaslane. Põhiliikmetel oli võimalus olemasoleva rakenduse kohta lisaselgitusi saada töölauaversiooni meeskonnalt, kuid uus rakendus on täielikult loodud uue arendusmeeskonna poolt kahekesi.

Kasutajaliidese jaoks otsustasime kasutada *React-i*, millega tegeles autori kursusekaaslane ning sellele serveripoole jaoks oli valikus GraphQL ning *REST API*, millest kaldus valik GraphQL-i poole põhiliselt jõudluse poolest, sest GraphQL-is saab

väga täpselt valida milliseid konkreetseid tabelivälju on vaja näha või uuendada ning pärida ühe korraga mitu ressurssi. [1]

GraphQL-i peale ehitamist alustades leidsime, et ühene lahendus kuidas API realiseerida puudub ning ka avalikest allikatest ei ole võimalik kõigile küsimustele vastuseid leida. API otsustasime realiseerida C#-keeles eelneva kogemuse pärast ning sellepärast, et olemasolev rakendus oli samas keeles. GraphQL-i C#-keeles realiseerides valis autor sel hetkel kõige populaarsema raamistiku *GraphQL .NET*.

Sellise API loomise põhiline raskus tuleb sellest, et olemasoleva andmebaasiga lähenemisel ei ole mõistlik asju ühekaupa ning väikselt teha. Arhitektuuri arendamise seisukohalt oli see suurepärane olukord, sest autor oli sunnitud palju mõtlema rakenduse skaleeritavuse peale, jõudes lahendusteni, mis peaksid olema kasulikud ka kordi väiksemal rakendusel.

Käesoleva töö autor on koos kursusekaaslasega kahekesi tegelenud antud töö rakenduse moderniseerimisega alates 2019 juunist kuni praeguseni, kulutades projekti peale üle tuhande töötunni.

Antud lõputöö eesmärk on viia üks suur monoliitne töölauarakendus koos andmebaasiga üle GraphQL tehnoloogiale ning leida ja näidata võtteid, millega lahendada koodis üleminekuga kaasnevaid probleeme.

Eesmärgini jõudmiseks on vaja esmalt uurida olemasolevat rakendust, millest on juttu 2. peatükis. Kolmandas peatükis tutvustatakse GraphQL tehnoloogiat ning valitud raamistikku, et hiljem oleks lihtsam mõista töös lahendatavaid probleeme. Neljandas punktis on toodud uue rakenduse arhitektuur ning realiseerimise käigus tekkinud probleemid ning lahendused. Viiendas punktis on hinnatud loodud lahenduse tulemust ning kuuendas punktis on projekti käigus tekkinud lahendamata tulevikumõtted.

2 Olemasolev lahendus

See peatükk räägib lõputöö raames moderniseeritav rakendusest nimega PAKK ning sellega lähemal tutvumisel leitud suurematest probleemidest, mis pidi uue rakenduse loomisel läbi mõtlema.

PAKK on C#-keeles kirjutatud *.NET Framework* rakendus ning kasutab *Postgres* andmebaasisüsteemi. PAKK-i kaks laiemat kasutust on kliendisuhete haldamine ning ressursside planeerimine, kuid sellele on juurde lisatud veel palju teisi funktsioone, näiteks arvete skanneerimine, maksude arvutamine ning arvete faktooring.

Selleks et kõige paremini mõista kui suure rakendusega on tegemist, alustame koodi meetrikast – PAKK töölauarakenduses on üle 2000 klassifaili ning üle 350 000 rea koodi, millest üle 40 000 on puhas SQL.

PAKK-i kasutajaliides on loodud *Windows Forms* raamistikus ning näeb rakendusest parema ettekujutluse jaoks välja järgnev:

Ekraanipilt 1 PAKK töölauarakenduse kasutajaliides

2.1 .NET Framework raamistikult üleminek .NET Core raamistikule

Esimene mure olemasolevat rakendust lähemalt uurides tekkis oli see, et isegi samas keeles kirjutatud rakendusel ei ole võimalik mudeleid kergesti ümber tõsta. Kuna mõlemad potentsiaalsed soovitud veebirakenduse tehnoloogiatest olid *.NET Core* põhjal ning olemasolev rakendus *.NET Framework-i* põhjal, pidi klasside üle tõstmiseks kaotama kõik *.NET Frameworki* ning sellele eksklusiivsete laienduste põimumised, mis tegi suure lisatöö mahu tõttu domeenimudeli ümbertõstmise ebamõistlikuks.

2.2 Päringute loogika säilitamine

Teiseks tutvus töö autor lihtsamate päringuoperatsioonidega. Vaadates kuidas olemasolevas töölauarakenduses need päringud olid realiseeritud, oli selgelt näha, et rakendus on tänaseks aegunud: Kõik päringud olid *string-idest* kokku pandud puhtas SQL-is ning saadud andmebaasipäringu tulemus oli ümber kaardistatud käsitsi, mis vajas tulevase hallatavuse huvides kindlasti muutmist. Lisaks SQL-ile tegi seal nende päringute ümbertõstmise väga keeruliseks tihe meetodite klassidevaheline põimumine *Windows Forms* ning *.NET Framework* rakendusele unikaalsete klassidega.

Olgugi, et suur osa päringutest oli mõistlik algusest uuesti luua, leidus ka palju mitmesaja koodirea pikkuseid SQL päringuid, mille sees oli kõvasti äriloogikat. Selleks, et mitte tervet SQL päringut uuesti läbi lugeda, pidi töö autor mõtlema mõne päringu puhtalt ületõstmisele.

2.3 Päringute jõudlus üle veebi

Töölauarakenduses on andmebaas ning rakendus ühises võrgus üksteise lähedal, kui liigutada selline rakendus veebi kasutades ühtainust ligipääsupunkti, tuleb arvesse võtta, et igal päringul on lühike viide. PAKK töölauarakendus on hetkel ehitatud üles niiviisi, et iga andmetabeli rea klikil tuleb uus päring ning kõik muudatused salvestatakse samuti eraldi päringuna – see on veebiserveri jaoks väga suur koormus kui kasutajate arv kasvab, samuti on kasutajaliidese jaoks efektiivsem laadida võimalikult palju korraga, et vältida üksikuid viiteaegu.

2.4 Suur andmebaasi skeem ning palju koodi

Rakenduse *Postgres* andmebaasis on üle 150 tabeli ning mitu tuhat tabelitevahelist seost ehk *foreign key*-d. Sellest tuleb üks lahenduse suurimaid probleeme, sest ühte ja sama lahendust üle 150 korra käsitsi teha on ebamõistlik, uue rakenduse arhitektuuris peab olema mõeldud skaleeritavusele otsekohe.

2.5 Olemasoleva rakendusega uuendustega kaasas püsimine

Töölauarakendus saab pidevaid uuendusi ning andmebaasimuudatusi, millega peab uues rakenduses sünkroonis olema, selletõttu on tähtis mõelda, kuidas tehtud muudatused kaasa viia.

3 GraphQL

Käesolev peatükk on lühikokkuvõte GraphQL-i arhitektuuri tähtsaimatest osadest ning *GraphQL .NET* raamistikust lugejale, kes pole GraphQL-iga varem tutvunud. Töö valmimise hetkeks on tekkinud ka teisi GraphQL raamistikke, millest töö alustushetkel *GraphQL .NET* populaarseim.

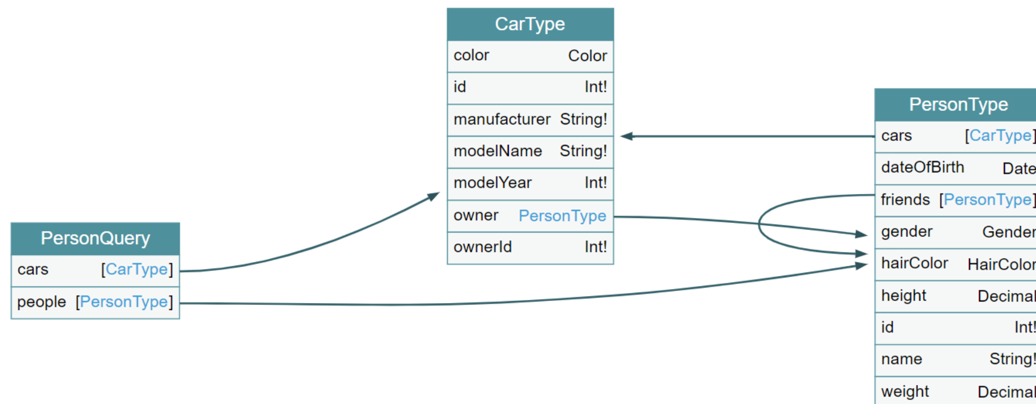
GraphQL on päringukeel API-de jaoks mis võimaldab küsida vaid nõutud andmed ilma ühegi lisata, küsida mitut ressurssi korraga ning koostada dünaamilisi päringuid, säästes võrreldes *REST API-ga* veebiliiklust ja HTTP päringuid, seeläbi pakkudes eeliseid nii kiiruses kui ka muudetavuses, mis on tarkvaraarenduse seisukohalt väga tugev argument tehnoloogia kasutuselevõtuks. [2]

Järgnevates peatükkides on läbivalt kasutatud peale olemasoleva lahenduse näidiste lisaks töö autori poolt loodud väiksemat näidisprojekti, et suure rakenduse arhitektuur oleks paremini hoomatavam. Lisaks tekstile on kasutatud veel pilte erinevatest koodiosadest, et neid selgemini näidata.

3.1 GraphQL-i arhitektuur

GraphQL-i API kõige alumine kiht koosneb tüüpidest (*Type*). *Type* on GraphQL võrgu sees üks objekt, millel võib olla lõputult seoseid teiste *Type-dega*, moodustades niimoodi suuremaid päringuid. Igal *Type-l* peab olema minimaalselt nimi, tüüp ning *resolver*, ehk väljale väärtuse määramiseks mõeldud delegaat.

Järgneval pildil on üks väike näidis *Type-dest* ning nende omavaheliste seoste plaanist ehk *GraphQL Schema-st*.



Joonis 1 GraphQL Schema näide

Joonisel 1 on *GraphQL Schema* sees kaks *Object Type*-i ning üks *Query Type*. *Query Type-ks* kutsutakse GraphQL-is juurt mille kaudu päring algab. *Object Type* on tavaline domeenimudeli klass kajastatud GraphQL notatsioonis.

Samuti on kõik objektiväljad eraldi *Type-d*, näiteks *Int* ja *String*. Neid kutsutakse *Scalar Type-deks*.

Kasutades neid *Type-e* on võimalik luua lõpmatu arv väga konkreetseid päringuid, et saada täpselt soovitud väljad ja mitte midagi enamat, mis pakub kasutajaliidese poolele väga palju vabadust ning väldib väga hästi koodi duplikatsiooni serveri poolel, kuna iga *Type* peab vaid ühe korra looma, ning pakub suuri eeliseid jõudluses veebiliikluse mahu vähendamisega.

Siin on näiteks üks päring, mis küsib ühe konkreetse id-ga inimese ning temaga seotud andmed, alustades eelmiselt pildilt toodud *PersonQuery Type-lt* ning liikudes mööda võrku järgmiste seosteni.

```
query {
  people(id:1){
    name
    height
    weight

    cars(color:BLACK){
      manufacturer
      modelYear
    }

    friends(gender:MALE){
      name
      friends{
        name
      }
    }
  }
}
```

```
{
  "data": {
    "people": [
      {
        "name": "Luke",
        "height": 1.8,
        "weight": 95.6,
        "cars": [
          {
            "manufacturer": "Audi",
            "modelYear": 2005
          }
        ],
        "friends": [
          {
            "name": "Thomas",
            "friends": [
              {
                "name": "Luke"
              },
              {
                "name": "Mary"
              },
              {
                "name": "Sarah"
              }
            ]
          },
          {
            "name": "Jeff",
            "friends": [
              {
                "name": "Luke"
              }
            ]
```

Ekraanipilt 2 Näide GraphQL päringust GraphQL-Playground keskkonnas

3.1.1 Operatsiooni põhi

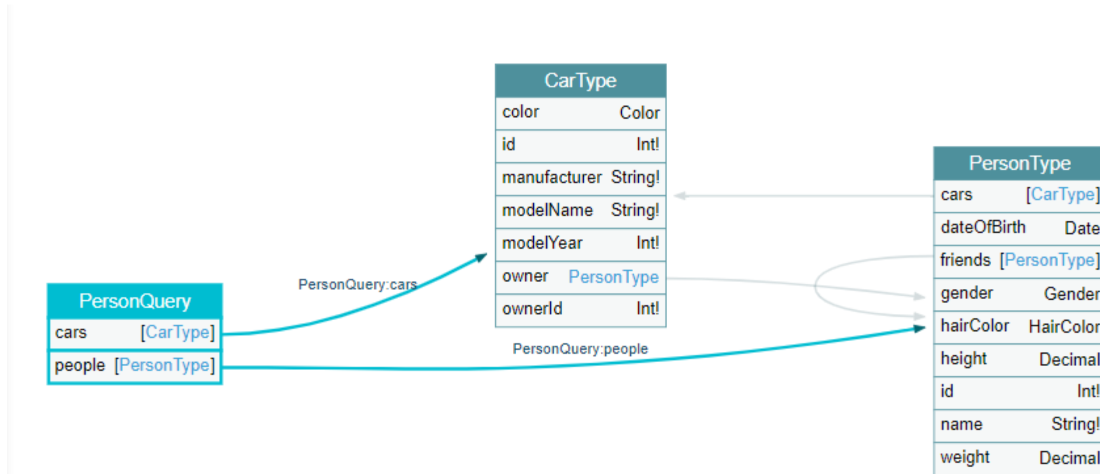
Üks GraphQL päring algab operatsioonitüübi defineerimisega, millele võib lisada nime või kaasaantud väärtusi, aga minimaalselt piisab vaid operatsioonitüübist.

Kaks kõige tähtsamat operatsiooni tüüpi on *Query* ning *Mutation*, millest siin lühikokkuvõttes on täpsustatud vaid *Queryt*, kuna nad ei oma suuri erinevusi ülesehituses.

- *Query* on kõige lihtsamas mõistes päring, see annab edasi ligipääsu juurpäringutele (*Query Type*)
- *Mutationiks* kutsutakse *GraphQLis Create*, *Update* ning *Delete* operatsioone, see liigub edasi muutmisoperatsioonide juurele (*Mutation Type*)

3.1.2 Query Operatsioon

Query operatsioon algab pihta juurpäringust, milles on defineeritud esimesed väljad millest päringut saab alustada, nagu järgneval pildil *cars* ning *people*.



Ekraanipilt 3 PersonQuery

GraphiQL ning *GraphQL-Playground* abirakendused näitavad *Introspection Query* põhjal kõik lubatud väljad ette ära, mis teeb päringute loomise lihtsamaks.

Peale juurpäringu valimist on kõik järgnevad objektid *GraphQL Object Type-id*, mille edasiseks täpsustamiseks saab:

- Lisada *Argument-e* milleks kutsutakse *GraphQL-is* päringu filtreid, mis asuvad sulgude sees peale *Type* nime defineerimist, nagu toodud järgmisel pildil.

Ekraanipilt 4 GraphQL Playground Argument-ide lisamise näidis

- Lisada konkreetsed väljad mida pärida, mis lisatakse loogeliste sulgude sisse peale *Type* ning sellele vastavate Argument-*ide* defineerimist, nagu toodud järgneval pildil.

Ekraanipilt 5 Object Type sees väljade valiku näide

Nagu ülaltoodud näites, võib olla igal valitud väljal ka omaette *Argument-id* (*color: BLUE*) ning oma alamvalik *Type-dest* (*cars* --> *modelYear*, *manufacturer*, *color*).

3.2 GraphQL .NET raamistiku lühiülevaade

Selles peatükis on minimaalne *GraphQL .NET* [3] raamistiku *Type* ja *Query* ülevaade, mida on kasutatud töö realisatsioonis, et paremini mõista järgnevates peatükkides loodud lahenduste ülesehitust.

3.2.1 Type

GraphQL .NET (kutsutud ka *graphql-dotnet*) raamistikus C# keeles näeb üks *GraphQL Type* välja järgnev, kasutades joonis 1 peal toodud *CarType* mudelit.

```
public class CarType : ObjectGraphType<CarDbRecord>
{
    0 references
    public CarType(DataContext data)
    {
        Field(expression: f => f.Id);
        Field(expression: f => f.OwnerId);
        Field(expression: f => f.Manufacturer);
        Field(expression: f => f.ModelName);
        Field(expression: f => f.ModelYear);
        Field<ColorEnumType, KnownColor?>().Name("color");


        Field<PersonType, PersonDbRecord>()
            .Name("owner")
            .Resolve(ctx =>
            {
                return data.GetPeople()
                    .SingleOrDefault(person => ctx.Source.OwnerId == person.Id);
            });


    }
```

Ekraanipilt 6 GraphQL .NET Field näide

Iga väli antud objektil on ka omaette *Type*, seega sellele on võimalik lisada kõiki *GraphQL* omadusi kasutades raamistiku *FieldBuilder* klassi meetodeid.

Kõige tähtsam on neist omadustest täita *Resolve* – sealt saab *Type* oma väärtuse päringus. Näiteks eelneval pildil *CarType Manufacturer* välja on võimalik eraldi laiendada argumendiga, mis muudab kõik tähed suureks, samuti saab nii vajadusel muuta välja nime ja kirjeldust, nagu toodud järgneval pildil:

```
//Field(f => f.Manufacturer);

Field<StringGraphType, string>()
    .Name("manufacturer")
    .Description("Auto tootja mark")
    .Argument<BooleanGraphType>(name: "uppercase", description: "return manufacturer name in uppercase")
    .Resolve(ctx =>
    {
        bool upper = ctx.GetArgument<bool>(name: "uppercase");
        if (upper)
            return ctx.Source.Manufacturer.ToUpper();

        return ctx.Source.Manufacturer;
    });
```

Ekraanipilt 7 CarType Manufacturer edasine täpsustamine

Üleval toodud näites välja kommenteeritud variant on lühim tavaliste valikutega kirjapilt, mis loob *CarDbRecord* klassist päritud objektilt välja ilma ühegi lisamuudatuseta. Kaks kaasaantud tüübiparameetrit (*StringGraphType,string*) on välja *GraphQL* tüüp ning *Resolve* meetodi tagastustüüp.

3.2.2 Query

Nagu varem mainitud on *Query* lihtsalt üks *GraphQL Type*, mis asub päringu alguses, *Query Resolve* meetodis asub kõige esimene andmebaasipäring, siin on sama joonisel 1 toodud näite *PersonQuery* näide koos ühe esimese juurpäringuga, *people*.

```
3 references
public class TestQuery : ObjectGraphType<object>
{
    0 references
    public TestQuery(DataContext data) //or Repository
    {
        Name = "PersonQuery";

        Field<ListGraphType<PersonType>, IEnumerable<PersonDbRecord>>()// Field<GraphQL Type , Resolve Return Type>>
            .Name("people")

            //päringu filtrid
            .Argument<IntGraphType>(name: "id", description: "")
            .Argument<HairColorEnumType>(name: "hairColor", description: "[Enum]")
            .Argument<GenderEnumType>(name: "gender", description: "[Enum]")
            .Argument<StringGraphType>(name: "name", description: "")

            .ResolveAsync(async ctx =>
            {
                //ctx sees on kõik küsitud väljad, parameetrid ning muu päringu info
                var loader = data.GetPeople(ctx.Arguments); //DataContext.cs
                return loader.ToList();
            });
```

Ekraanipilt 8 PersonQuery ning people juurpäring

4 GraphQL API realiseerimisel leitud probleemide analüüsimine ning lahendamine

See peatükk räägib uue GraphQL API realiseerimisest ning selle käigus tekkivatest tüüpprobleemidest ning kuidas need lahendada.

- 1. alampeatükk räägib realiseeritava API arhitektuurist üleüldiselt
- 2. alampeatükk on seotud tihedalt selle konkreetse projekti iseärasustega ning monoliitselt rakenduselt üleminekuga läbi andmebaasi.
- 3. alampeatükis toodud lahendused on seotud hallatava arhitektuuri loomisega ning oleks kasulikud ka kordi väiksema projekti loomisel.
- 4. alampeatükk on seotud CRUD operatsioonidega ning nende jõudluse, hallatavuse ja automatiseerimisega.
- 5. alampeatükis lahendatakse ainult jõudluse probleeme erinevates arhitektuurikihtides.
- 6. alampeatükk arutab rakendusega edasiseks toimetulekuks lahendatavatest probleemidest.

4.1 API Arhitektuuri planeerimine

Nagu 2. punktis toodud, olime töölauarakenduse suure mahu tõttu sunnitud eelkõige mõtlema arhitektuuri skaleeritavuse peale, seega loodud lahenduste keskseks teemaks sai koodi automatiseerimine ning koodi ühisosa võimalikult suureks tegemine.

Selle jaoks, et suurendada koodi ühisosa on olemas kaks põhilist koodivõtet. Esimeseks on *inheritance* [4] ehk koodi päritavus (kutsutakse ka polümorfismiks [5]), millest on kirjutatud alates 1969 aastast. Teiseks on *generic programming* [6], ehk ilma tüüpparameetriteta programmeerimine, millest esimesed kirjed on aastast 1973.

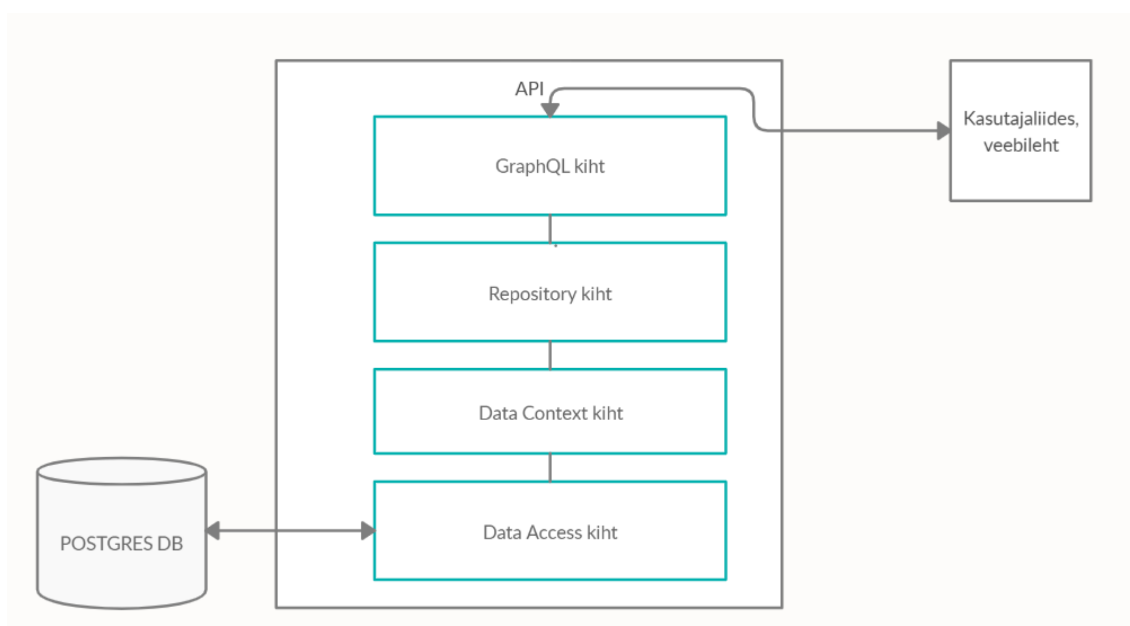
Toodud lahendustes üritab töö autor nende kahe põhivõtte abil luua rakenduse, mis saavutaks minimaalse koodi kirjutamisega võimalikult palju, tuues sisse veel palju teisi tuntud tarkvaramustreid ning valmislahendusi, et tööd kiirendada.

Enne lahendusteni minekut oleks hea mõista plaanitud rakenduse laiemat arhitektuuriskeemi, millesse lähemalt süvenedes jõuame kohtadesse, kus on skaleeritavusega probleeme.

Antud lahendus on vaid API realiseerimine, millele on vaja kasutajaliides eraldi juurde luua, mida antud töö raames ei käsitletud.

API arhitektuur jagunes põhiliselt neljaks suuremaks kihiks: GraphQL, Repository, Data Context ning Data Access. Kõik nendest kihtidest sisaldavad endis alamprobleeme, millele lahendused asuvad järgmistes alampeatükkides.

Järgneval joonisel on üldistatud skeem realiseeritava API arhitektuurist:



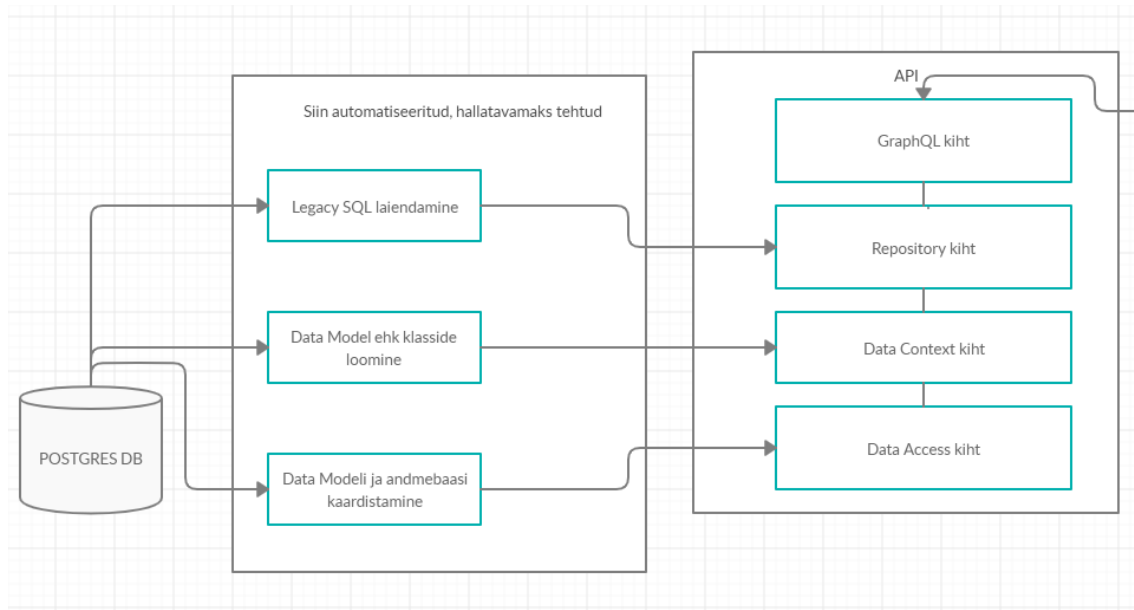
Joonis 2 API lai arhitektuur

Joonisel 2 on näha API üldistatud arhitektuuri ning kihtidevahelist ning kasutajaliidesega ja andmebaasiga suhtlemise plaani. Minimaalselt on vaja neist kihtidest vaid alumist ning ülemist (*Data-Access* andmebaasi jaoks ning GraphQL kiht kasutajaliidese jaoks), kuid kaks keskmist kihti on vajalikud rakenduse hallatavuse ning muudetavuse huvides.

4.2 Olemasolevalt rakenduselt migreerimise probleemid ja lahendused

Siin punktis toodud lahendused käsitlevad automatiseerimist ning taaskasutust domeenimudeli loomisel, muutuva andmebaasi kasutamist automatiseerimisel ning eksisteeriva SQL sisese äriloogika (ehk *legacy* koodi) kasutamisest uues rakenduses.

Järgneval skeemil on siin alampeatükis toodud lahendused ning igale lahendusele vastav arhitektuurikiht, mille jaoks see mõeldud on.



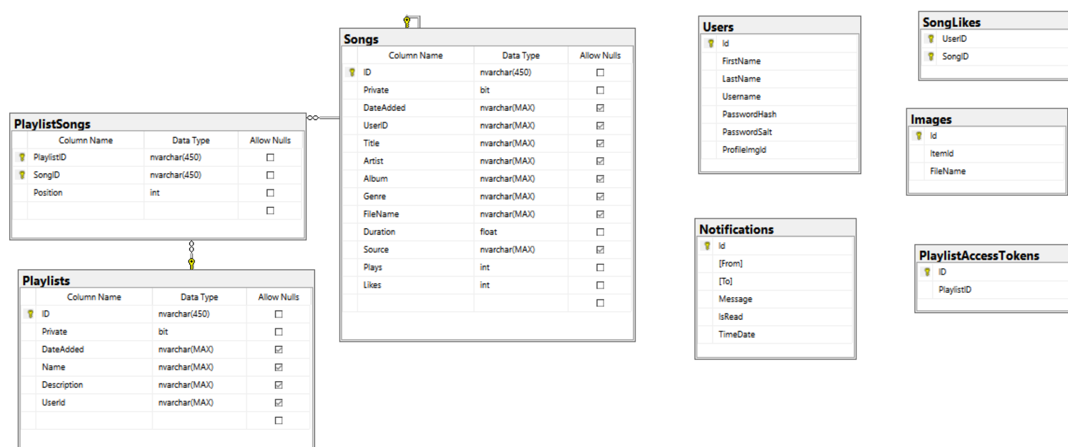
Joonis 3 Olemasolevalt rakenduselt migreerimise probleemid ja lahendused

Joonisel 3 on näha selles punktis toodud andmebaasi kasutades lahendatud probleemid ning neile vastavad arhitektuurikihid.

4.2.1 Klasside loomine andmebaasi põhjal

Selleks et luua üks andmebaasile vastav rakendus, on ilmselgelt vaja luua andmebaasile vastavad tabeliklassid. Väikese rakenduse puhul oleks näiteks üks hea lihtne lahendus avada *Microsoft SQL Server Management Studio* [7] ning genereerida sealt diagramm – see annab lihtsa ülevaate andmebaasimudelist ning selle eeskujul on üpris kerge klasse luua.

Järgneval pildil on üks väiksema andmebaasi tabelite põhjal genereeritud diagrammi näide *Microsoft SQL Server Management Studio* rakenduses.



Ekraanipilt 9 Näide lihtsamast SSMS genereeritud diagrammist

Eelneval ekraanipildil on näha näidet andmebaasitabelite põhjal genereeritud diagrammist ning nende väljasid koos välja andmetüübiga.

Probleem tekib eelneva näitega selles, et see on hea lahendus vaid siis, kui tabeleid on hoomatav kogus. Kui andmebaasimudelis on üle 150 tabeli nagu PAKK rakenduses, siis niiviisi diagrammi vaadata ja ühekaupa klasse ja seoseid kaardistada ei ole mõistlik.

Teades, et kõik mudeliklassid on vaja tabeli põhjal uuesti luua (2.1) ning kõik klassid peavad sünkroonis olema töölauarakendusega (2.5) oli vaja luua dünaamiline lahendus.

Esimeseks lahenduseks sai loodud SQL *script*, mis kirjutas iga tabeli põhjal C# süntaksiga klassi tulemuseks, mis oli parem viis seda teha. (vt. Lisa 1)

Scripti tulemuseks oli selline tekstifail, millele pidi vaid .cs lõpu panema, et saada kõik klassid koos väljadega otse andmebaasimudelist.

```
public class card_transaction {
public Int32 id { get; set;}

public Int32 card_id { get; set;}

public Int32 product_id { get; set;}

public Int32 partner_sales_point_id { get; set;}

[MaxLength(50)]
public String transaction_no { get; set;}

public DateTime transaction_time { get; set;}

public Decimal quantity { get; set;}

public Decimal station_price { get; set;}

public Decimal own_price { get; set;}
```

Ekraanipilt 10 SQL scripti tulemus, et luua tabelitele vastavad klassid

Peale edasist otsimist leidus veelgi parem lahendus. Järgmisena leidis töö autor Entity Framework *dbcontext scaffold* funktsiooni mis võimaldas andmebaasi põhjal terve andmebaasi ligipääsukihi koos klassidega ja *Unit of Work*-iga [8] valmis teha.

Selleks on kasutatud *.NET EF Core tool* [9] rakendust ning andmebaasi ligipääsutööriista (*database provider*), milleks sai PAKK rakenduse *Postgres* andmebaasi pärast *Npgsql* [10].

dbcontext scaffoldi tulemus on kõik andmebaasitabelid koos võtmete ning seostega eraldi failides ning *DbContext* fail, kus on kõik klassid kaardistatud vastavatele andmebaasiväljadele ning kõik muu vajalik andmebaasi info.

4.2.2 Muutuva andmebaasimudeli peale koodi ehitamine

Kuna me ehitasime rakendust teise pidevalt muutuva rakenduse kõrval, siis see tähendas, et meil oli vaja sünkroonis olla töölauarakendusega ning kõikide andmebaasimuudatustega. See tekitas järgneva probleemi, kuna erineva koodibaasiga samade muudatuste pidevalt tegemine on suur lisatöö.

Tänu eelmises punktis loodud *dbcontext scaffold* tulemusele, on võimalik *scaffoldi* tulemuse failide muutmise asemel kasutada tervet genereeritud koodibaasi nagu liikuvat põhja. Iga muudatuse peale on siis võimalik lihtsalt failid uuesti genereerida, ilma lisatööd tekitamata.

Lisaks sellele genereeris *scaffold* kõik klassid *partial* võtmesõnaga, mis võimaldab ühe klassi mitmest failist kokku panna, võimaldades kõik laiendused hoida eraldi kaustas ning pidevalt vahetada *Model* kausta sisu välja.

```
17 references
public partial class CustomerInvoiceStatus
{
    1 reference
    public int Id { get; set; }
    4 references
    public int CustomerInvoiceId { get; set; }
    3 references
    public string StatusCl { get; set; }
    3 references
    public int CreatedByUserId { get; set; }
    1 reference
    public DateTime CreatedOn { get; set; }

    1 reference
    public virtual UserAccount CreatedByUser { get; set; }
    1 reference
    public virtual CustomerInvoice CustomerInvoice { get; set; }
    1 reference
    public virtual Classification StatusClNavigation { get; set;
}
```

Ekraanipilt 11 Genereeritud klassi originaalne fail

```
17 references
public partial class CustomerInvoiceStatus
{
    0 references
    public string Extension1 { get; set; }

    0 references
    public string Extension2 { get; set; }


    0 references
    public string Extension3 { get; set; }
}
```

Ekraanipilt 12 Genereeritud klassi laiendused läbi *partial* võtmesõna

Selleks, et teha muudatusi, mis pole vaid laiendused, on hea idee pärida genereeritud klass uute klassi, mis sai antud lahenduses tehtud *DbContext-iga*, et genereeritud klassis antud meetodeid ülekirjutada. Nii saab *GeneratedContext-i* meetodeid vastavalt vajadusele ka piirata, säilitades kõik ise tehtud muudatused.

Selle jaoks on vaja lihtsalt pärida enda *DbContext* genereeritud *DbContext* klassist nagu toodud järgnevas näites.

```
namespace EntityFrameworkData.Data
{
    67 references
    public class ApplicationDbContext : GeneratedContext
    {

        public ApplicationDbContext(DbContextOptions options) : base(options)
        {
        }
```

Ekraanipilt 13 Genereeritud koodist päritud ApplicationDbContext

4.2.3 SQL Vaated Entity Frameworkis

Kuna töölauarakenduses oli palju suuri SQL päringuid, mis olid täis äriloogikat (2.2) oli vaja kuidagi need uute rakendusse üle viia. Sellised päringud said kasutusele võetud vaadetena (*View*). *View* on üks eeldefineeritud SQL *script*, mida hoitakse andmebaasis.

Reeglina *View-i* ennast pärida ei tohiks mingisugune probleem olla, kuna see ei erine kõvasti tavalisest SQL-ist. *View-i* põhiline probleem tekib aga sellest, et seda SQL päringut oleks vaja filterdada ning vajadusel läbi navigatsiooniväljade laiendada.

Eelmise punkti lahenduse tulemuseks genereeriti igale *View-ile* vastav klass ning kaardistati andmebaasi, kuid näiteks järgnevas *CardTransactionEuroRate View-is* on *card_transaction_id*, mille kaudu ei ole võimalik otse sellele konkreetsele *CardTransaction* objektile ligi pääseda. Järgneval pildil on toodud genereeritud *View-i* kaart andmebaasipäringu tulemuse jaoks.

```
modelBuilder.Entity<ViewCardTransactionEuroRate>(entity =>
{
    entity.HasNoKey();

    entity.ToTable("view_card_transaction_euro_rate");

    entity.Property(e => e.CardTransactionId).HasColumnName("card_transaction_id");

    entity.Property(e => e.MarkupPercentage)
        .HasColumnName("markup_percentage")
        .HasColumnType("numeric(19,4)");

    entity.Property(e => e.Rate)
        .HasColumnName("rate")
        .HasColumnType("numeric(17,5)");
});
```

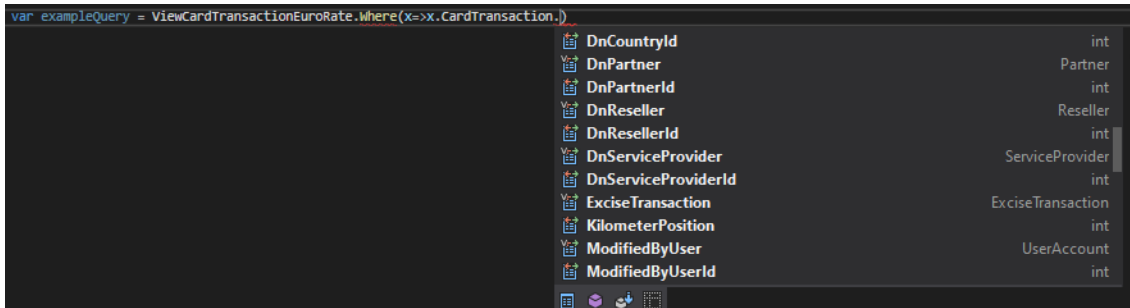
Ekraanipilt 14 View kaardistatud C# klassilt andmebaasitabeli väljadele läbi EntityFrameworki

Selleks, et antud *View-i* navigatsioonivõimalus anda, on mõistlik kasutada *partial* võtmesõna omadust laiendada klassi eraldi failis, kus lisame sellele soovitud välja, nagu ekraanipilt 12 peal. Niiviisi sai lisatud klassile eraldi *CardTransaction* väli, nagu toodud järgneval pildil.

```
4 references
public partial class ViewCardTransactionEuroRate
{
    2 references
    public CardTransaction CardTransaction { get; set; }
}
```

Ekraanipilt 15 CardTransactioni jaoks Viewi laiendusklass

Peale seda on vaja uuele lisatud väljale väärtus anda, mille kaardistame eelmises punktis loodud *ApplicationDbContext* failis ekraanipilt 14 eeskujul. Loodud lisavälja abil on võimalik nüüd päringut nii laiendada kui ka piirata otse SQL-is ilma ridagi SQL-i kirjutamata, kasutades navigatsioonivälju, mis *CardTransaction-i* kaudu tulevad, nagu järgneval pildil.

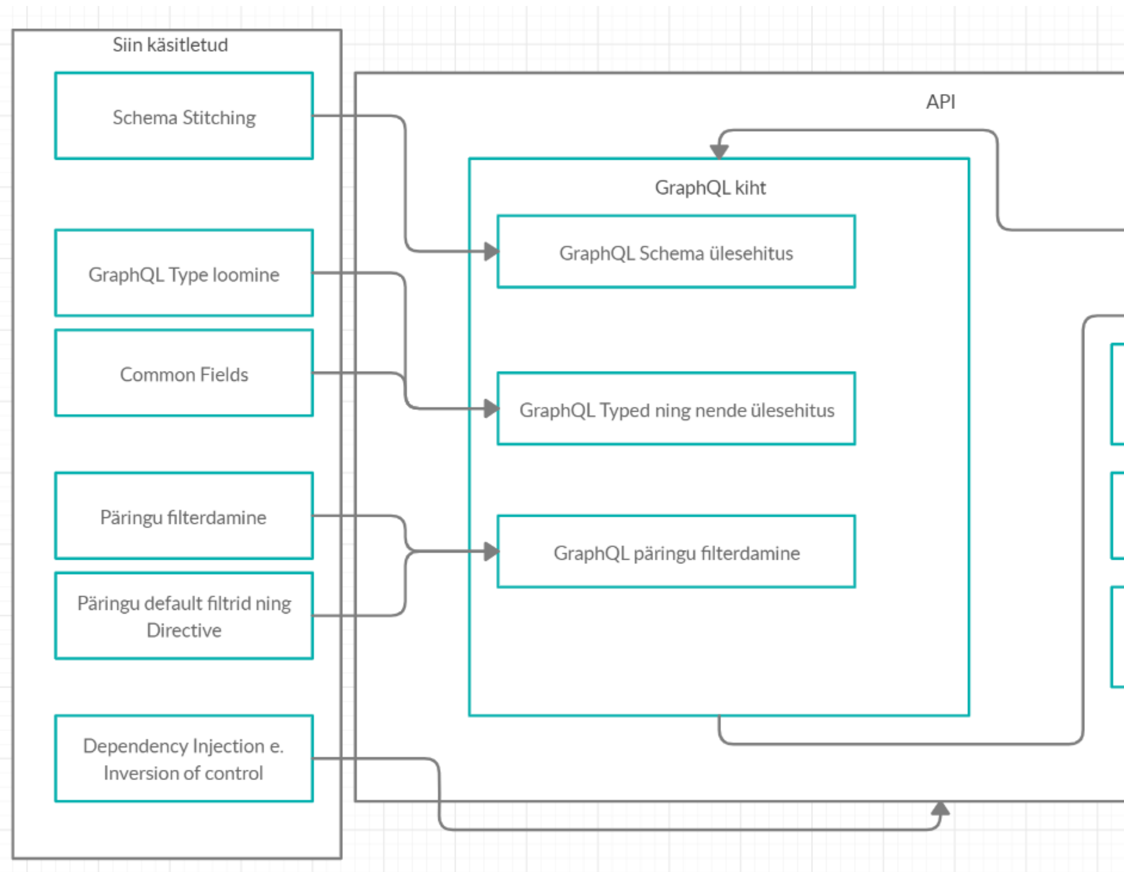


Ekraanipilt 16 Viewi navigatsiooniväljade kasutamine EntityFrameworkCore raamistikus filterdamiseks

4.3 Üldisemad GraphQL projekti struktuuri ning valitud raamistiku probleemid

Siin punktis toodud lahendused käsitlevad automatiseerimist ning taaskasutust projektisiseste sõltuvuste ning kõigega mis on seotud otseselt GraphQL päringute arhitektuuriga ning selle kihi ja valitud raamistikuga.

Järgneval joonisel on toodud selle punkti alamteemad ning nendele vastavad rakenduse arhitektuuriosad.



Joonis 4 GraphQL kihi ning projekti ülesehitusega seotud probleemid ning nendest mõjutatud arhitektuuriosad

4.3.1 GraphQL põhja ülesehitus ja tükeldamine

Kuna GraphQL arhitektuuris on *Schema* objekti sees vaid üks *Query* ning üks *Mutation* klass, siis see tekitab projekti suurenedes ühe väga suure klassi, mis ei ole kergesti muudetav või jaotatav.

Selle lahendamiseks on meetod mida kutsutakse GraphQL skeemi kokkuõmblemiseks (*Schema Stitching)* [11]. *Schema Stitching* on mitme *GraphQL API* kokku panemine üheks suureks. Niiviisi saab näiteks mikroteenustega täiesti erinevaid rakendusi hallatavalt ühe ligipääsupunkti juurde panna.

Selle lõputöö projektis on *Schema Stitching-u* eesmärk see, et meil on märkimisväärne kogus erinevaid päringuid ning objekte, mis tekitaks ühe liiga suure klassifaili, mida oleks vaja hallatavuse eesmärgil väiksematest klassidest kokku panna, et neid saaks vajadusel ühekaupa juurde lisada või eemaldada. Kui rakendust pole vaja struktureerida töötavate

osadena ning eesmärk on vaid väiksem fail saada, on sama probleem lahendatav ka *partial* võtmesõnaga, mis tükeldaks ühe klassifaili mitmeks.

Kuna *GraphQL .NET* raamistikus ei ole *Schema Stitching* lahendust kaasaantud, sai see lahendatud nii, et kõik *Query* objektid said registreeritud *Dependency Injection-i* alla ühise liidesena (*IQueryComponent*) ning rakenduse käivitusel pannakse kokku üks suur *Query* objekt kõikide *Query-de* väljadest, nagu järgneval pildil. Täpselt samamoodi tasub ka teha *Mutation-ite* ehk muutmisoperatsioonidega.

```
public sealed class RootQuery : ObjectGraphType<object>
{
    0 references
    public RootQuery(IEnumerable<IQueryComponent> graphQueryMarkers)
    {
        Name = "RootQuery";

        foreach(var marker in graphQueryMarkers)
        {
            var q = marker as ObjectGraphType<object>;
            foreach(var f in q.Fields)
            {
                AddField(f);
            }
        }
```

Ekraanipilt 17 GraphQL Schema tükeldatav lahendus

4.3.2 *Dependency Injection* arhitektuurimuster ning selle skaleeritavus

Absoluutselt kõik *GraphQL Type-d* registreeritakse antud raamistikus *Dependency Injection* mustri järgi. Selle tulemusel saab neid registreeritud *Type-sid* kasutada kõikjal rakenduses, seehulgas teiste *Type-de* sees.

Dependency Injection-i [12] põhilised eesmärgid on koodi taaskasutatavuse tõstmine ning klassi sõltuvuste muutmine ilma sellest sõltuva klassi muutmiseta.

Probleem tekib selles, et nende *Type-de* kogus võib minna väga suureks, minimaalselt CRUD operatsioonide jaoks on vaja 2 *Type-i* iga objekti kohta, *Object Type* ning *Input Object Type*. Ilusam oleks veel kõik ühe objekti operatsioonid grupeerida ühe Mutation *Type* alla ning teha erinevatest Input-idest variatsioonid, mis läheks üle 150 objektiga juba täiesti hoomamatuks. All on näide sellest, milline oli 3.1 punktis toodud näite *Type-de* registreerimine.

```
//Kõik GraphQL typed peab registreerima siia
services.AddSingleton<GenderEnumType>();
services.AddSingleton<ColorEnumType>();
services.AddSingleton<HairColorEnumType>();

services.AddSingleton<PersonType>();
services.AddSingleton<PersonMutationType>();
services.AddSingleton<PersonInputType>();

services.AddSingleton<CarType>();
services.AddSingleton<CarMutationType>();
services.AddSingleton<CarInputType>();
services.AddSingleton<CarInputTypeWithoutIDs>();
```

Ekraanipilt 18 3.1 punktis toodud skeemi Type Dependency Injection

Selleks, et vältida iga klassi eraldi registreerimist, on mõistlik neid registreerida kõik korraga. Selle lahenduse jaoks sai autor inspiratsiooni *Unit Test-idest*, kus oli kasutatud *Namespace* baasil koodi, mis koostas iga klassi jaoks kausta sees vastavad testid, kust tekkis järgmine küsimus, et milleks kirjutada koodi, mis kontrollib kas nõutud asi on tehtud, kui on võimalik selle sama koodijupiga see lihtsalt automaatselt ära teha.

Kuna *Clean Code* [13] põhimõtetel peaks olema kõik sama tüüpi klassid samas namespaces eraldi failidena, siis on väga hea võimalus kasutada seda ära ning võtta otse kaustast kõik *Type* klassid, mida sai projekti käigus mitusada ning pidi juba edasi alamkaustadeks sorteerima.

Sellise refaktooringu tulemusel on võimalus muuta tüütu mitmesaja registreeringu haldamine vaid kaheks koodireaks, mis registreerib kõik dünaamiliselt, nagu järgneval pildil.

```
1 reference
public static void Register(IServiceCollection services)
{
    var allTypes = NamespaceTypeHelper.GetTypesFromNamespace("GraphQLData.ObjectTypes");
    foreach (var graphtype in allTypes) services.AddSingleton(graphtype);
}
```

Ekraanipilt 19 Dependency Injection automatiseeritud Namespace põhjal

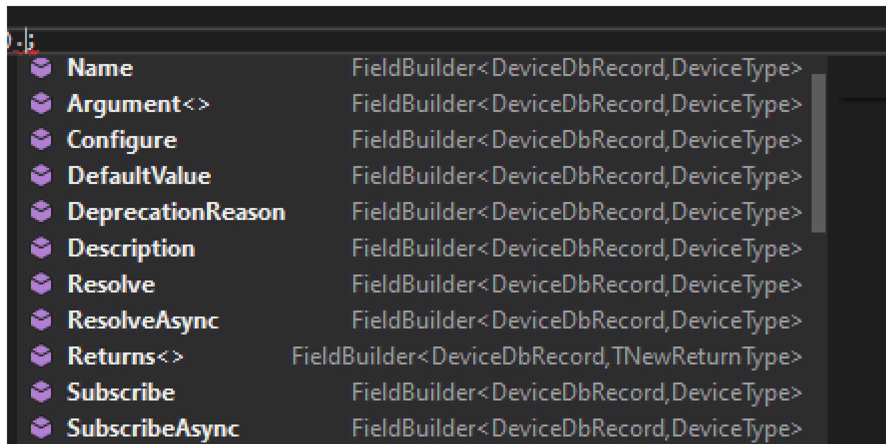
4.3.3 GraphQL tüübi loomine

Iga GraphQL *Type* koosneb *Field-idest*, mis on minimaalselt registreeritud nime, tagastustüübi ning *resolve* delegaadiga – väga sarnane ühele tavalisele objekt-orienteeritud keele meetodile. Järgneval pildil on üks minimaalsete nõuetega *Field graphql-dotnet* raamistikus.

```
Field<DeviceTypeGraphType,DeviceType>()
    .Name("type")
    .Resolve(x=>x.Source.DeviceType);
```

Ekraanipilt 20 Minimaalsete nõuetega GraphQL .NET raamistiku Field

Lisaks sellele on võimalik kasutada raamistiku FieldBuilder klassi, et *Field*-i edasi kohandada, milles on järgneval pildil toodud valikud.



Ekraanipilt 21 GraphQL .NET FieldBuilder klassi võimalused

kui *Type*-l on defineeritud tüübiparameeter, siis on võimalik lisada *Field* ka ühe reaga, siis võetakse lihtsalt välja nimi ning selle tüübile vastav vaikimisi GraphQL tüüp:

```
Field(expression: x => x.Id);
Field(expression: x => x.Name);
Field(expression: x => x.XLocation);
Field(expression: x => x.YLocation);
Field(expression: x => x.State, nullable: true);
Field(expression: x => x.GroupAddress);
```

Väga mitme erineva klassiga läheb seegi liiga pikaks, PAKK projektis oleks see tähendanud, et Field(x=>x) meetodit on vaja üle 1500 korra kirjutada.

Selle jaoks on veel *GraphQL .NET* raamistikus *AutoRegisteringObjectGraphType* [14] klass, mis registreerib niiviisi kõik klassi väljad, aga sellega tekib kohe probleeme. Esiteks tavaline Field(x=>x) meetod ei registreeri ühtegi *Enum* tüüpi. Peale selle ei suuda see klass tulla toime ka näiteks *Int16* ja *BigInteger* tüüpidega. Nullable tüüpe ei saa muuta, ning ühe välja muutmiseks oleks vaja terve mitmekümne väljaga klass käsitsi teha.

Selle lahenduseks on tehtud *AutoRegisteringObjectGraphType* eeskujul oma implementatsioon, mis suudab tegeleda ka *Int16* ning *BigInteger* tüüpidega, muudab

CRUD põhjustel kõik väljad nullitavaks ning võimaldab ühekaupa väljasid käsitsi mappida. Järgneval pildil on realisatsioonis loodud automaatselt väljade registreerimiseks mõeldud klassi kood, et tegeleda puuduvate skalaartüüpide toetusega, *BigInteger-i* puhul tasub kindlasti arvestada, et üle Integer maksimaalse väärtuse see toodud lahendus ei tööta ning vajab eraldi *Scalar Type* loomist.

```
public class AutoObjectGraphType<TSourceType> : ObjectGraphType<TSourceType>
{
    6 references
    public AutoObjectGraphType(params Expression<Func<TSourceType, object>>[] excludedProperties)
    {
        foreach (var propertyInfo in GetRegisteredProperties())
        {
            if (excludedProperties?.Any(p => GetPropertyName(p) == propertyInfo.Name) == true)
                continue;

            if (propertyInfo.PropertyType.Name == "Int16")
            {
                Field(
                        typeof(IntGraphType),
                        propertyInfo.Name
                    ).DefaultValue =
                    (propertyInfo.GetCustomAttributes(typeof(DefaultValueAttribute), inherit: false).FirstOrDefault() as
                        DefaultValueAttribute)?.Value;
                continue;
            }

            if (propertyInfo.PropertyType.Name == "BigInteger")
            {
                Field(
                        typeof(IntGraphType),
                        propertyInfo.Name
                    ).DefaultValue =
                    (propertyInfo.GetCustomAttributes(typeof(DefaultValueAttribute), inherit: false).FirstOrDefault() as
                        DefaultValueAttribute)?.Value;
                continue;
            }


            Field(
                    propertyInfo.PropertyType.GetGraphTypeFromType(
                        ValueTypeHelper.IsNullableProperty(propertyInfo)),
                    propertyInfo.Name
                ).DefaultValue =
                (propertyInfo.GetCustomAttributes(typeof(DefaultValueAttribute), inherit: false).FirstOrDefault() as
                    DefaultValueAttribute)?.Value;
        }
    }
```

Ekraanipilt 22 AutoObjectGraphType kood

4.3.4 GraphQL väljade taaskasutus koodis

Iga *Type* seos teise *Type*-ga on vaja defineerida eraldi *Field*-ina, mis päritakse läbi *resolve* delegaadi. Järgneval pildil on üks lihtne näide *CompanyType* seosest *CustomerBankAccountType* tüübiga.

```
Field<ListGraphType<CustomerBankAccountType>, IEnumerable<CustomerBankAccount>>()
    .Name("CustomerBankAccounts")
    .AddDefaultFilters()
    .ResolveAsync(async ctx =>
    {
        var loader = accessor.Context.GetOrAddCollectionBatchLoader<int?, CustomerBankAccount>
            ("GetBankAccountsByCompanyId",  fetchFunc: (x) =>  repo.GetBankAccountsByCompanyId(x, ctx.SubFields, ctx.Arguments));
        var l = await loader.LoadAsync(ctx.Source.Id);
        return l;
    });
```

Ekraanipilt 23 Field seose näide teise Typega

Lisaks meetodile, mis ühe välja kaudu teise pärib on vaja ka tihti lisada nendele seostele võimalused filtrite jaoks ning pakkuda teisi *FieldBuilder-i* meetodeid – näiteks API versiooni uuendusel ja seose kaotamisel oleks vaja *DeprecationReason* panna.

Probleem on selles, sellise *Field-i* sisu on väga mitmes kohas täpselt sama. Näiteks PAKK rakenduses on väga palju klasse millel on defineeritud *CompanyId* väli, millele tuleks *GraphQLis* lisada seos *Company-ga*.

Lisaks sellele oleks vaja võimalust nende seoste filterdamiseks, näiteks küsida ainult aktiivselt tegutsevaid *Company* objekte. Niiviisi tuleb päris mitmesse kohta rakendusse üks ja sama suur *Field*, mille muutmine läbivalt on ebavajalik monotoonne lisatöö. Järgneval pildil on näidis sellest kui palju erinevaid *Argument-e* võib üks *Field* rakenduse suuruse kasvades nõuda.

```
Field<ListGraphType<CompanyType>, IEnumerable<EntityFrameworkData.Model.Company>>()
    .Name("Company")
    .AddDefaultFilters()
    .Argument<CompanyEnumType>(name: "type", description: "Company type [Enum]")
    .Argument<StringGraphType>(name: "name", description: "desc")
    .Argument<StringGraphType>(name: "reg_no", description: "desc")
    .Argument<StringGraphType>(name: "vat_no", description: "desc")
    .Argument<StringGraphType>(name: "general_phone", description: "desc")
    .Argument<StringGraphType>(name: "email", description: "desc")
    .Argument<StringGraphType>(name: "contact_person", description: "desc")
    .Argument<IntGraphType>(name: "sales_agent", description: "desc")
    .Argument<IntGraphType>(name: "customer_manager", description: "desc")
    .Argument<BooleanGraphType>(name: "without_customer_manager", description: "desc")
    .Argument<BooleanGraphType>(name: "is_active", description: "desc")
    .Argument<BooleanGraphType>(name: "has_port1_contracts", description: "desc")
    .Argument<BooleanGraphType>(name: "has_other_contracts", description: "desc")
    .Argument<DateGraphType>(name: "valid_from", description: "desc")
    .ResolveAsync(async ctx =>
    {
        var loader = accessor.Context.GetOrAddCollectionBatchLoader<int?,
            EntityFrameworkData.Model.Company>
        ("GetCompaniesById", fetchFunc: (x) => companyRepository.GetCompaniesById(x, ctx.SubFields, ctx.Arguments));
        return await loader.LoadAsync(ctx.Source.CustomerId);
    });
```

Ekraanipilt 24 Company Field ning selle Argumendid ja Resolve delegaat

Üks variant on refaktoreerida *resolve* delegaat välja, mille tulemusel saab andmebaasipoolset külge ühiselt muuta ning kasutada filtrite lisamiseks ühist meetodit nagu *AddDefaultFilters* üleval toodud pildil. Peale pikemat katsetamist leidus, et veel

parem idee oleks kogu *Field* koos võtmega välja refaktoreerida, sest see võimaldab absoluutselt kõik ühe *Type* seoste päringud liigutada ühte kohta, vajadusel muuta selle tagastustüüpi, filtreid, nime ning vähendada veelgi koodi kordamist. Selle lahenduseks on tehtud meetod, mis annab parameetrina kaasa võtme, mille järgi pärida *Company* objekt, nagu toodud järgneval pildil.

```
this.AddCompanyFieldFor(exp: x=>x.Source.CustomerId, accessor, companyRepository, fieldName: "customer");
```

Ekraanipilt 25 Refaktooritud Company Fieldi meetod

Kuna selle võtme väärtus on kompileeritud alles rakenduse töötamisel, siis on vaja teha sellise näite jaoks meetod otse *Expression-ist* nagu järgnevas koodinäites (*exp* parameeter).

```
14 references
public static void AddCompanyFieldFor<T>(this ComplexGraphType<T> source, Expression<Func<ResolveFieldContext<T>,object>> exp,
    IDataLoaderContextAccessor accessor, ICompanyRepository companyRepository, string fieldName = null)
{
    if (fieldName == null)
    {
        fieldName = GetFieldNameFromExpression(exp);
    }

    source.Field<ListGraphType<CompanyType>, IEnumerable<Company>>()
        .Name(fieldName)
        .AddDefaultFilters()
        .Argument<CompanyEnumType>(name: "type", description: "Company type [Enum]")
        .Argument<StringGraphType>(name: "name", description: "desc")
        .Argument<StringGraphType>(name: "reg_no", description: "desc")
        .Argument<StringGraphType>(name: "vat_no", description: "desc")
        .Argument<StringGraphType>(name: "general_phone", description: "desc")
        .Argument<StringGraphType>(name: "email", description: "desc")
        .Argument<StringGraphType>(name: "contact_person", description: "desc")
        .Argument<IntGraphType>(name: "sales_agent", description: "desc")
        .Argument<IntGraphType>(name: "customer_manager", description: "desc")
        .Argument<BooleanGraphType>(name: "without_customer_manager", description: "desc")
        .Argument<BooleanGraphType>(name: "is_active", description: "desc")
        .Argument<BooleanGraphType>(name: "has_port1_contracts", description: "desc")
        .Argument<BooleanGraphType>(name: "has_other_contracts", description: "desc")
        .Argument<DateGraphType>(name: "valid_from", description: "desc")
        .ResolveAsync(ctx =>
        {
            var key = (int?)exp.Compile().Invoke(ctx);
            if (key == null)
                return null;

            var loader = accessor.Context.GetOrAddCollectionBatchLoader<int?, Company>
                ("GetCompaniesById", fetchFunc: (x) => companyRepository.GetCompaniesById(x, ctx.SubFields, ctx.Arguments));
            return loader.LoadAsync(key);
        });
}
```

Ekraanipilt 26 Expressioniga meetod Company Fieldist

Niiviisi on sarnaste väljade muudatused hallatavad, Ideaalis võiks kõik teisele *Type-le* suunatud seosed niiviisi defineeritud olla, see välistaks ebavajaliku koodi kordamise täielikult.

4.3.5 Päringu täpsustamine

GraphQL-is on võimalik päringuid filterdada ning täpsustada kahte viisi: Esiteks võib anda igale *Type-le* kaasa sulgudes *Argument-e*, mida võib kirjeldada kui tavaliste päringu parameetritena. Teiseks on võimalik kasutada *Directive*, mis on ühised tervele andmebaasimudelile ja võib kõikjal kasutada eeldefineeritud *Directive Location* lubatud väärtuste põhjal.

Selleks et ühel *Type-l Argument-e* kasutada, on vaja kõigepealt defineerida see *Field-i* peal. *Argument-il* peab olema *Graph Type*, nimi ning kirjeldus. Näiteks siin all näites on *Field-ile* nimega people defineeritud 4 *Argument-i* : *id*, *hairColor*, *gender* ning *name*.

```
Field<ListGraphType<PersonType>, IEnumerable<PersonDbRecord>>()
    .Name("people")
    .Argument<IntGraphType>(name: "id", description: "")
    .Argument<HairColorEnumType>(name: "hairColor", description: "[Enum]")
    .Argument<GenderEnumType>(name: "gender", description: "[Enum]")
    .Argument<StringGraphType>(name: "name", description: "")
    .ResolveAsync(async ctx =>
    {
        var loader = data.GetPeople(ctx.Arguments);
        return loader.ToList();
    });
```

Ekraanipilt 27 Argumentide ehk filtrite näide GraphQL kihis

Lisaks sellele peab olema *resolve* käigus seda *Argument-i* ka päringu loogikas rakendatud, näiteks selle projekti raamistikus saab teha *Argument-ide Dictionary* parameetri jaoks sellise *foreach loopi*.

```
foreach (var arg in arguments.Reverse())
{
    query = arg.Key switch
        {
        "id" => query.Where(x => x.Id == (int) arg.Value),
        "name" =>query.Where(x => x.Name == (string) arg.Value),
        "gender" => query.Where(x => x.Gender == (Gender) arg.Value),
        "hairColor" => query.Where(x => x.HairColor == (HairColor) arg.Value),
        _ => query //default
        };
    arguments.Remove(arg.Key);
}
```

Ekraanipilt 28 Argumentide rakendamiseks loodud loopi näide

Projekti suurenedes tekib probleem esiteks sellega, et andmebaasipäringud on vaja taaskasutuse huvides *Repository-de* alla jaotada, kuna ühe ja sama päringu jaoks iga kord *resolve* sisse meetod kirjutada on ebaefektiivne.

Niiviisi tekib aga olukord, kus on vaja igale *Field-ile* eraldi *Argument* lisada ning *Repository* meetodis selle *Argument-i* nimele vastav filter rakendada – seal on suur oht, et mõned nimed ei ole vastavuses, mis tähendaks, et filter ei tee midagi.

Selle lahenduseks on päringu filtrite refaktoorimine eraldi klassiks, kus on *Argument* ning kõik sellega seonduv samas kohas, milleks on loodud *GraphQLFilter* klass, nagu järgneval pildil:

```
42 references
public class GraphQLFilter<T>
{
    35 references
    public Func<FieldBuilder<object,IEnumerable<T>>,string,FieldBuilder<object,IEnumerable<T>>> Argument { get; set; }

    35 references
    public Func<IQueryable<T>, KeyValuePair<string,object>, IQueryable<T>> ResolveMethod { get; set; }

}
```

Ekraanipilt 29 GraphQL filtri klass

Lisaks on sellele filtrile vaja ka kuskil nimi defineerida, mille otsustas töö autor panna lihtsalt *Dictionary* võtmeks, mis neid filtreid hoiab. Selle põhjuseks on see, et *Dictionary-st* on võimalik ükskõik mis suuruse korral konstantse ajaga nimele vastav filter tagastada, aga *List-ist* tagastades on vaja kõik eelnevad filtrid ühekaupa läbi itereerida. All toodud pildil on näide niiviisi tehtud osadest *Company* päringu filtritest.

```
public static class CompanyGraphQLFilters
{
    public static Dictionary<string,GraphQLFilter<Company>> Filters = new Dictionary<string,GraphQLFilter<Company>>()
    {
        {
            "name",
            new GraphQLFilter<Company>()
            {
                Argument = (field,fieldName) => field.Argument<StringGraphType>(fieldName, description: "desc"),
                ResolveMethod = (query,arg) => query.GetSQLLikeFilter( exp: x=>x.Name,arg),
            }
        },
        {
            "reg_no",
            new GraphQLFilter<Company>()
            {
                Argument = (field,fieldName) => field.Argument<StringGraphType>(fieldName, description: "desc"),
                ResolveMethod = (query,arg) => query.GetSQLLikeFilter( exp: x=>x.RegNo,arg),
            }
        },
        {
            "vat_no",
            new GraphQLFilter<Company>()
            {
                Argument = (field,fieldName) => field.Argument<StringGraphType>(fieldName, description: "desc"),
                ResolveMethod = (query,arg) => query.Where(x =>
                    x.CustomerVatPayer.Select(f=>f.Number).Any(g=>EF.Functions.Like( matchExpression: g, pattern: "__"+arg.Value.ToString()+'%'))
                    || x.CustomerVatPayer.Select(f=>f.Number).Any(g=>EF.Functions.Like( matchExpression: g, pattern: arg.Value.ToString()+'%'))),
            }
        },
```

Ekraanipilt 30 Company päringu refaktooritud filtrid

Selleks et neid filtreid rakendada on tehtud 2 abimeetodit, millest esimene lisab *Field*-ile argumendid, ning teine rakendab need meetodid *IQueryable* peal, nagu järgneval pildil.

```
3 references
public static FieldBuilder<object, IEnumerable<T>> AddFilters<T>(this FieldBuilder<object, IEnumerable<T>> fieldBuilder, Dictionary<string,GraphQLFilter<T>> filters)
{
    foreach (var filter in filters)
    {
        filter.Value.Argument(fieldBuilder,filter.Key);
    }
    return fieldBuilder;
}

3 references
public static IQueryable<T> ResolveFilters<T>(this IQueryable<T> query,Dictionary<string,object> arguments, Dictionary<string,GraphQLFilter<T>> filters)
{
    foreach (var arg in arguments.Reverse())
    {
        if (filters.ContainsKey(arg.Key))
        {
            var filter = filters[arg.Key];
            query = filter.ResolveMethod(query,arg);
        }

        if (!ArgumentHelper.IsDefaultArgument(arg))
            arguments.Remove(arg.Key);
    }
    return query;
}
```

Ekraanipilt 31 Refaktooritud filtrite rakendusmeetodid

4.3.6 Päringu vaikimisi filtrid ning *directive* tüüp

Paljudel väljadel tekib API realiseerimise käigus ühiseid filtreid – näiteks peaaegu kõigil klassidel PAKK rakenduses on oma *Id* või *Name*, seega on enesestmõistetav, et selle jaoks peaks filter olema.

Tüüpiliselt on nii REST API kui ka GraphQL API realisatsioonis toodud ühe objektitüübi näitamiseks kaks eraldi meetodit, millest üks tagastab listi ning teine tagastab vastava id-ga objekti. Alumisel pildil on näide niiviisi loodud *people* ning *person* päringutest.

```
Field<ListGraphType<PersonType>, IEnumerable<PersonDbRecord>>()
    .Name("people")
    .Argument<HairColorEnumType>(name: "hairColor", description: "[Enum]")
    .Argument<GenderEnumType>(name: "gender", description: "[Enum]")
    .Argument<StringGraphType>(name: "name", description: "")
    .ResolveAsync(async ctx =>
    {
        var loader = data.GetPeople(ctx.Arguments);
        return loader.ToList();
    });

Field<PersonType, PersonDbRecord>()
    .Name("person")
    .Argument<NonNullGraphType<IntGraphType>>(name: "id", description: "")
    .Resolve( ctx =>
    {
        var person = data.GetPerson(id: ctx.GetArgument<int>(name: "id"));
        return person;
    });
```

Ekraanipilt 32 Listi ning üksiku vaste jaoks eraldi päringud GraphQLis

Esimene suur probleem mis sellest tekib on see, et see eeldab iga objektitüübi jaoks eraldi *Repository* meetodit. Siis on vaja ka selle meetodi jaoks vastav *Field* teha, mis üle saja erineva klassi puhul tähendab tühja tööd, kui alternatiiviks on see *Id* parameeter lihtsalt sama listi päringu juurde viia.

Sellest jõuame kohe järgmise probleemini, et neid *Id* ja *Name* filtreid, seehulgas ka veel näiteks *Count* tulemuste piiramiseks või *Order* järjekorra jaoks, on vaja väga mitmes kohas korrata, et saada *Query* sees ka alamobjekte ja objektisiseste listide suurust piirata.

Sellised filtrid on lahendatud ühise *AddDefaultFilters* meetodiga, mis on lisatud igale päringuobjektile ning objektivahelisele seosele ning *HandleDefaultFilters* meetodiga, mis rakendab päringule need filtrid. Selle tulemuseks on võimalik ka päringu alamobjekte ilma suurema vaevata filterdada.

Näiteks siin on toodud päring mis küsib nimejärekorras firmad 11-20, nende 5 esimest lepingut ning nende lepingute staatuse ajaloo 3 hiliseimat kirjet. Lisaks sellele hoiab see päring ka mälu ning veebiliiklust kokku küsides otse andmebaasist vaid need (ja *Id*) väljad. See on täpsus ja jõudlus mida tüüpiline *REST API* pakkuda ei suuda.

```
query{
  companies(order:NAME_ASC,count:10,offset:10){
    name
    contracts(count:5,order:DATE_ASC){
      contractNo
      contractTypeCl
      contractStatusHistory(count:3,order:DATE_DESC){
        statusCl
        statusDate
      }
    }
  }
}
```

Ekraanipilt 33 Vaikimisi filtritega täpsustatud päringu näide

Lisaks vaikimisi filtritele on ka võimalik kasutada *Directive*, millest on *GraphQL .NET* raamistikus algul kaasas @include ja @skip, millega on võimalik anda näiteks tingimusi mis juhul üldse välja pärida ning millal mitte.

Directive saab kahjuks muuta ainult resolve tulemust ning edasist hargnemist, mitte päringut ennast, seega sellepärast on see kasulik pigem näiteks kuupäeva formaadi valimiseks või mõne muu sarnase järeltöötluse operatsiooni jaoks. Siin on näide *Directive-st* mis muudab *string-i* kõik tähed suureks.

```
query{
  people{
    name
  }
}
```

```
{
  "data": {
    "people": [
      {
        "name": "Luke"
      },
      {
        "name": "Mary"
      },
      {
        "name": "Thomas"
      },
      {
        "name": "Sarah"
      },
      {
        "name": "Jeff"
      }
```

Ekraanipilt 34 Päring ilma Directive-ta

```
query{
  people{
    name @toUpper
  }
}
```

```
{
  "data": {
    "people": [
      {
        "name": "LUKE"
      },
      {
        "name": "MARY"
      },
      {
        "name": "THOMAS"
      },
      {
        "name": "SARAH"
      },
      {
        "name": "JEFF"
      }
    ]
  }
```

Ekraanipilt 35 Päring koos toUpper Directive-ga

Directive rakendamiseks tuleb *graphql-dotnet* raamistikus järgneva näite eeskujul registreerida sellele vastav middleware eraldi *GraphQLMiddleware* külge *Chain of Responsibility* [5] mustriga.

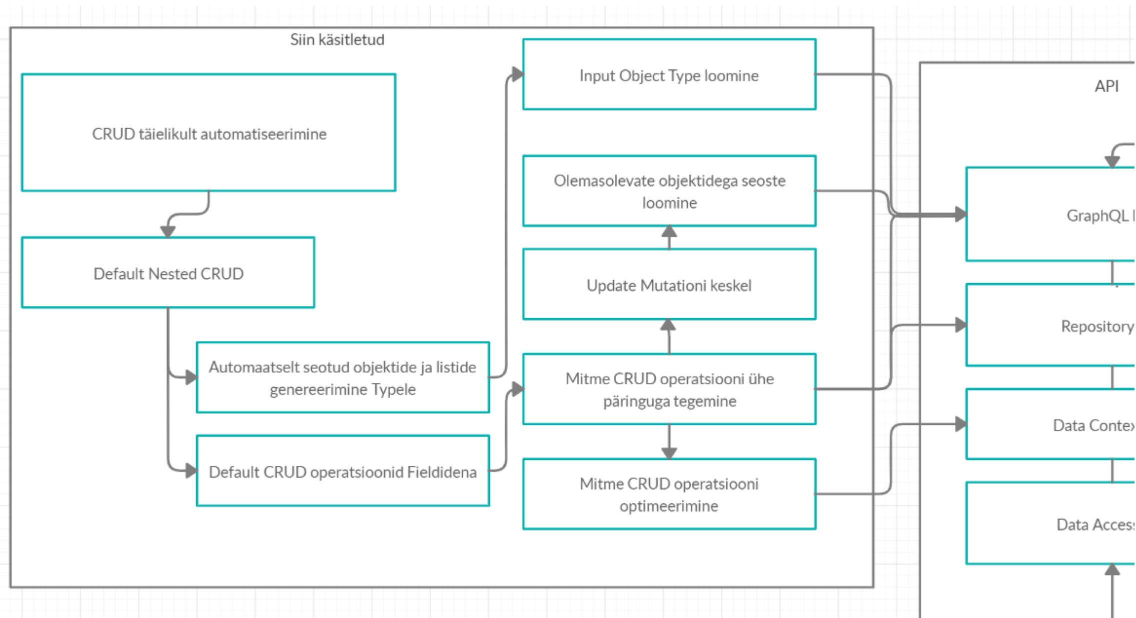
```
public class ToUpperDirectiveFieldMiddleware
{
    0 references
    public async Task<object> Resolve(
        ResolveFieldContext context,
        FieldMiddlewareDelegate next)
    {
        var result = await next(context);
        var toUpperDirective = context.FieldAst.Directives.Find(name: "toUpper");

        var toUpperDirectiveExists = (toUpperDirective != null);

        return toUpperDirectiveExists ? result.ToString().ToUpper() : result;
    }
}
```

Ekraanipilt 36 ToUpper Middleware näidis graphql-dotnet raamistikus

4.4 CRUD lahendamine

Siin punktis toodud lahendused käsitlevad automatiseerimist ning koodi taaskasutust andmebaasis andmete loomise, muutmise ning uuendamisega ning selle jaoks läbitavate kihtidega. Siin peatükis toodud lahendused on ehitatud järjest üksteise otsa, alustades väiksematest tükkidest ning lõpetades nende tükkide koondamisega üheks.

Järgneval joonisel on skeem järgnevatest alampunktidest ning nende punktide poolt kasutatud eelnevad punktid ja arhitektuurikihid, et saada automatiseeritud lahendus.



Joonis 5 CRUD operatsioonidega seotud probleemid ning nende ülesehitus alates väiksemate probleemide lahendamisest.

4.4.1 *Input Object Type* loomine

Input Object Type loomisel tekivad samad probleemid nagu *Object Type* loomisel, (vt 4.2.3) aga selle erinevusega, et *Input Type* jaoks ei eksisteeri üldse olemasolevat väljasid registreerivat *AutoRegisteringObjectGraphType* lahendust, seega seal on vaja ise teha *InputObjectGraphType* implementatsioon, mis registreerib *Field-id*.

Lihtsaim viis seda teha on võtta sama *Object Type* implementatsiooni kood, aga selle erinevusega, et *ObjectGraphType* asemel tuleb pärida loodud klassitüüp *InputObjectGraphType* klassist.

4.4.2 Mitme CRUD operatsiooni ühe päringuga tegemine

Tüüpilises *REST API*-s on igal ressursil eraldi POST ligipääsupunkt, kust neid ühekaupa andmebaasi lisada saab. Suurte objektide puhul võib see tähendada väga mitut HTTP päringut ning sellega kaasnevat eraldi andmebaasioperatsiooni.

Sarnaselt *REST API-le* on vaja teha nende ligipääsupunktide asemel GraphQL-is *Mutation Field*, millele antakse argumendina objekt mida muuta või lisada. Üks lihtsam *Create* Operatsioon võib näha GraphQL-is välja selline:

```
Field<StringGraphType, string>()
    .Name("createPerson")
    .Argument<NonNullGraphType<PersonInputType>>(name: "input", description: "")
    .Resolve(context =>
    {
        var person = context.GetArgument<PersonDbRecord>(name: "input");
        data.AddPerson(person);
        return "added person successfully";
    });
```

Ekraanipilt 37 GraphQL .NET Mutation operatsiooni näide

Niiviisi iga CRUD operatsiooni luues tekib probleem sellega, et PAKK rakenduses oleks näiteks ühe *Company* objekti lisamine või uuendamine koos seostega üle 15 *Create* operatsiooni, potentsiaalselt isegi veel rohkem, kui näiteks lisada firmale palju kommentaare, kontaktisikuid ning aadresse.

Kuna antud töös loodud rakenduse kasutajaliidesel on kasutusel *Redux*, mis hoiab kergesti meeles objektiga tehtud muudatused, on sellega võimalik koostada täpselt muudatustele vastav GraphQL operatsioon, mis säästab niiviisi potentsiaalselt mitukümmend HTTP päringut ning andmebaasipäringut. See võimaldab rakendusel väga hästi skaleeruda, kui selle kasutajate arv suureneb.

Selle lahenduseks on esiteks vaja lisada *Input Object Type* külge *Field*, mis sisaldab listi soovitud kaasobjektidest, nagu järgnevas näites *cars* ning *friends*. Samuti on soovituslik mitte-hädavajalikud väljad teha *Nullable*-ks, et neid ei peaks iga kord tühjalt kaasa saatma, kui neid pole vaja muuta/lisada.

```
public PersonInputType(DataContext data)
{
    Name = "PersonInput";

    Field(expression: x => x.Name);
    Field(expression: f => f.Height, nullable: true);
    Field(expression: f => f.Weight, nullable: true);
    Field(expression: f => f.FriendIdsList, nullable: true);
    Field(expression: f=>f.DateOfBirth, nullable: true);
    Field<GenderEnumType, Gender>().Name("gender");
    Field<HairColorEnumType, HairColor>().Name("hairColor");

    Field<ListGraphType<CarInputType>, string>().Name("cars");
    Field<ListGraphType<PersonInputType>, string>().Name("friends");
}
```

Ekraanipilt 38 GraphQL .NET InputObjectGraphType näide

Selle tulemuseks on võimalik anda *Mutation-i* päringusse kaasa list objekte, mis luua koos põhiobjektiga, võimaldades luua nüüd puu struktuuris päringuid ka *Mutation-i* pool.

Peale seda on vaja *Add* ja *Update* operatsioonid selliseks teha, et need töötaks ka kaasaantud objektidega, Näiteks nii.

```
mutation{
  person{
    createPerson(input:{
      name:"newPerson"
      friends:[
        {name:"friend1"}
        {name:"friend2"}
      ]
      cars:[
        {
          color:BLUE
          manufacturer:"Toyota"
          modelName:"Auris"
          modelYear:2010
```

Ekraanipilt 39 Mitme Create Mutation operatsiooni näide

```
2 references
public PersonDbRecord AddPerson(PersonDbRecord person)
{
    person.Id = people.Count + 1;
    people.Add(person);

    foreach (var car in person.Cars)
    {
        car.Id = cars.Count + 1;
        car.OwnerId = person.Id;
        AddCar(car);
    }
```

Ekraanipilt 40 Mitme Create Mutation operatsioon Repository kihis

Sellisel kujul on võimalik lisada juba mitu objekti ühe operatsiooniga, aga sellel on veel kaks puudust: esiteks selline lahendus teeb iga objekti kohta eraldi andmebaasipäringu, teiseks on sellega hetkel võimalik vaid lisada uusi objekte, kuid oleks vaja ka lisada seoseid olemasolevate objektidega.

4.4.3 Mitme CRUD operatsiooni optimeerimine

Selleks et teha kõik sama tüübi lisamisoperatsioonid ühe andmebaasipäringuga, kasutame me *DbContext SaveChanges()* meetodit alles peale kõikide lisamisoperatsioonide lõpetamist, mis on tarkvaraarhitektuuris võte tuntud *Unit of Work* [8] nime all. Lisaks sellele on rangelt soovituslik kasutada *EntityFramework* raamistikus *Add*() asemel eeltöödeldud listiga *AddRange*() meetodit, et vältida operatsioonieelset ajakulu (*overhead)* iga objekti *DetectChanges* kutsumisel ning optimeerimata SQL *scripti* koostamist läbi üksikute insert lausete. [15] [16]

Peale neid muudatusi on meil meetod, mis kutsub *DetectChanges()* iga listi kohta ühe korra, mis avaldab palju mõju suurtele listidele ning kutsub *SaveChanges* alles täiesti lõpus, mis tähendaks näiteks järgneva seitsmest inimesest koosneva puu lisamiseks vaid ühtainust andmebaasioperatsiooni, võimaldades väga kiiret objektivõrgustike lisamist.

```
mutation{
  person{
    createPerson(input:{
      name:"newPerson"
      friends:[
        {name:"friend1"
            friends:[
              {name:"friend1Offriend1"}
              {name:"friend2Offriend1"}]
        }
         {name:"friend2"
            friends:[
              {name:"friend1Offriend2"}
              {name:"friend2Offriend2"}]
        }
```

Ekraanipilt 41 Person võrgustiku lisamine läbi friends seose

4.4.4 Olemasolevate objektidega seoste loomine CRUD operatsioonides

Kuna eelmises punktis loodud operatsiooni saab kasutada ainult uute objektide esmaseks loomiseks, on vaja ka viisi, kuidas luua objektile olemasolevaid seoseid.

Selle lahenduseks on kasutatud *Input Object Type-l Id* välja – Näiteks igal *Person-il* on *Id*, mille olemasolul ta juba eksisteerib. Kui *Id* on 0 või üldse märkimata, siis luuakse uus *Person*. Niiviisi saab ühe sama *Mutation-iga* lisada *Person-ile* uusi sõpru nii olemasolevatest kirjetest, kui ka täiesti uute kirjetena

```
person{
  createPerson(input:{
    name:"newPerson"
    friends:[
      {id:5}
      {id:0 name:"newFriend"}
    ]
  })
```

Ekraanipilt 42 Uue ning olemasoleva seose lisamine samas Mutationis

```
foreach (var friend in person.Friends)
{
    if (friend.Id == 0) //loob uue
    {
        friend.Id = people.Count + 1;
        AddPerson(friend);
        person.FriendIdsList.Add(friend.Id);
    }
    else
    {
        person.FriendIdsList.Add(friend.Id);
    }
}
```

Ekraanipilt 43 Uue ning olemasoleva seose kood Repository kihis

4.4.5 Uuendusoperatsioonid Mutation operatsiooni keskel

Sarnaselt *Create* operatsioonile, on mõistlik ka kõik seotud *Update* operatsioonid teha ühe sama HTTP päringuga. Selle jaoks kasutame ära eelmises punktis jäänud olemasoleva *Id*-ga objekti.

Kui varem kasutasime *Person-i Id* väärtust vaid selleks, et lisada *friends* listi selle *Id*-ga *Person*, siis nüüd saame laieneda sellelt ka uuendusoperatsioonile, ehk muudame *Mutation-i* operatsioonis koos põhiobjekti enda väljadega ka kõik olemasoleva *Id*-ga objektide väljad, saades kõik samad jõudluse eelised nagu eelmistes punktides. Näiteks siin on muudetud nii *Person-i* kui ka tema sõbra (*id:5*) nime sama *Mutation-i* sees.

```
person{
  updatePerson(input:{
    id:7
    name:"nameChanged"
    friends:[
      {id:5 name:"newName"}
     ]
```

Ekraanipilt 44 Sisemise objektiseosega Update näidis

Lisaks sellele on ka selle sama *updatePerson-i* operatsiooni sees võimalik kasutada täpselt sama (4.3.2) create operatsiooni, kui objekti *Id* oleks 0, vähendades veelgi vajadust teha HTTP päringuid erinevate operatsioonide jaoks, nagu järgneval pildil.

```
  updatePerson(input:{
    id:7
    friends:[
      {id:5 name:"updateName"}
      {id:0 name:"addNewPerson"}
     ]
```

Ekraanipilt 45 Create ja Update samas GraphQL operatsioonis

4.4.6 Automaatselt seotud objektide ja listide genereerimine Input Object Type jaoks

Kuna 4.3.2 punktis toodud lahenduses toodud *Field* listid on vaja koos iga muudatusega kaasa muuta ning PAKK rakenduses on nende muutmine (ja eelkõige lisamine) suur

lisatöö, siis on mõistlik kõik need objektivahelised seosed sinna automaatselt genereerida ning hiljem tegeleda üksikute eranditega.

Selle jaoks laiendame *AutoInputType* klassi uue klassiga, mis lisab konstruktori sees kõik objektiga seotud väljad *Field-ideks*, nagu järgnevas näites.

```
50 references
public class AutoInputTypeWithRelatedFields<TSourceType> : AutoInputType<TSourceType>
{
    49 references
    public AutoInputTypeWithRelatedFields(ICommonRepository commonRepository)
    {
        var sourceType = typeof(TSourceType);
        AddAllInnerMutations(source: this,commonRepository,sourceType);
    }
```

Ekraanipilt 46 Seotud väljadega automatiseeritud klass

Nende *Field-ide* lisamise jaoks kasutame *System.Reflection* meetodeid, mis on tuntud üpris halva jõudluse poolest [17], kuid sellest pole vahet, sest nende väljade registreerimine toimub vaid ühe korra – rakenduse käivitamisel. Selle jaoks on tehtud kaks abimeetodit (järgnev pilt), mis käivad läbi *TSourceType* tüübiparameetris etteantud klassi ning võtavad sealt kõik väljad mis on kas *Model* kausta klassi tüüpi või listid neist.

```
var innerListMutationTypes = MutationBaseType.GetInnerMutationListTypes();
foreach (var innerListMutation in innerListMutationTypes)
{
    source.AddInnerMutationList(innerListMutation.InnerMutationName, innerListMutation.InnerMutationType);
}

var innerMutationTypes = MutationBaseType.GetInnerMutationTypes();
foreach (var innerMutation in innerMutationTypes)
{
    source.AddInnerMutation(innerMutation.InnerMutationName, innerMutation.InnerMutationType);
}
```

Ekraanipilt 47 Mudeliklasside vaheliste seoste läbikäimiseks loodud meetodid

Nende meetodite realiseerimisel on kasutatud C# klassi *FullName* väärtust, et määrata kus *Namespace* sees klass asub. Selleks et mitte panna koodi kokku stringidest ning veenduda et mõne kausta või projekti ümbernimetamine koodi ära ei lõhuks, on ka hea idee kasutada *nameof*() meetodit (järgnev pilt).

Samuti on turvalisuse mõttes mõistlik eemaldada sellest automatiseerimisest kõik tundlikud klassid nagu PAKK näites *UserAccount*, et mõne *CreatedByUser* või *ModifiedByUser* seose kaudu neid puutuda ei saaks. Kui neid rohkem tekib, oleks

mõistlik ka need refaktoreerida eraldi oluliste klasside kogumi alla, aga üks lihtsam näide sellest on toodud järgneval pildil.

```
public static List<InnerMutationStruct> GetInnerMutationTypes(this Type objType)
{
    PropertyInfo[] properties = objType.GetProperties();

    var modelNamespace = string.Format("{0}.{1}",nameof(EntityFrameworkData), nameof(EntityFrameworkData.Model));

    //kõik klassi väljad EntityFrameworkData.Model namespacest
    var props = properties.Where(x => x.PropertyType.FullName.StartsWith(modelNamespace)).ToList();


    props.RemoveAll(match:o => o.PropertyType.Name == nameof(UserAccount));
    props.RemoveAll(match:o => o.PropertyType.Name == nameof(Permission));
    props.RemoveAll(match:o => o.PropertyType.Name == nameof(PermissionRole));
    props.RemoveAll(match:o => o.PropertyType.Name == nameof(PermissionUser));
```

Ekraanipilt 48 CRUD automatiseerimisele erandite lisamine ning nameof() meetod, et võimaldada väljaspool koodimuutusi

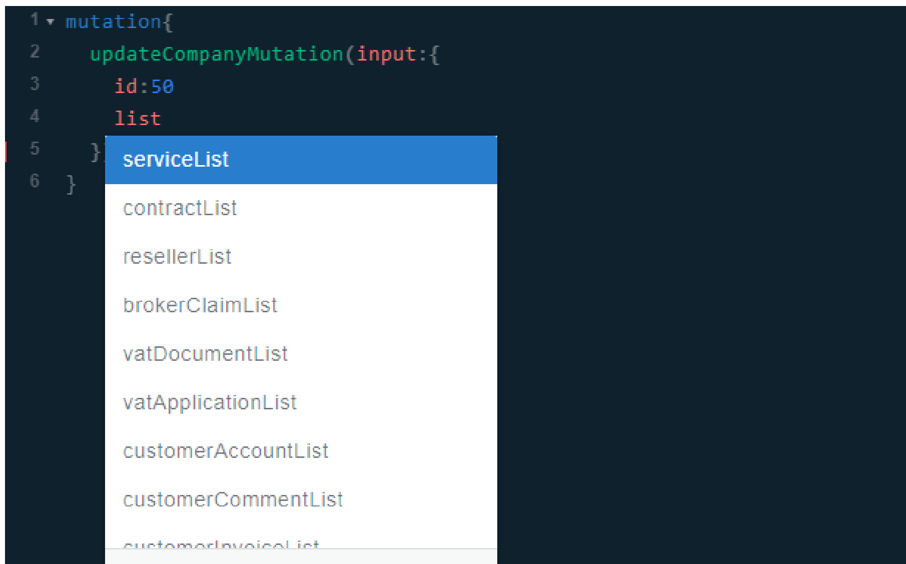
Sarnaselt on ka võimalik saada kõik listiväljad, mille *FullName* erinevuseks on see, et *EntityFrameworkData.Model* nimepõhja väärtuse asemel on näiteks *ICollection-i* implementatsioonil *System.Collections.Generic.ICollection`1[[EntityFrameworkData.Model.*

Peale seda on võetud *InputType-s* alt kõik klassid ning nendest on kokku pandud objektid kogu vajaliku infoga, et luua *Type-le* vastavad *Field-id* selle klassi seostest, millega on *Field-i* lisamise jaoks lihtsad *generic* meetodid tehtud, mis tagastavad *int* väärtuse põhimõttel, et saada uue loodud objekti *Id* väärtuse tagasi edasiseks kasutuseks. All toodud pildil on näide nendest *Create* või *Update* operatsioonide jaoks mõeldud *generic Field-ide* meetoditest.

```
1 reference
public static void AddInnerMutation<T>(this ComplexGraphType<T> source, string name, Type argumentType)
{
    // Lisab ainult välja inputile, peab Create / Update meetodi sees eraldi handlima
    FieldBuilder<ComplexGraphType<T>, int> f = FieldBuilder.Create<ComplexGraphType<T>, int>(argumentType);
    f.Name(name);
    source.AddField((FieldType) f.FieldType);
}

1 reference
public static void AddInnerMutationList<T>(this ComplexGraphType<T> source, string name, Type argumentType)
{
    // Lisab ainult välja inputile, peab Create / Update meetodi sees eraldi handlima
    FieldBuilder<ComplexGraphType<T>, int> f = FieldBuilder.Create<ComplexGraphType<T>, int>(ValueTypeHelper.CreateGenericGraphQLListOfType(argumentType));
    f.Name(name);
    source.AddField((FieldType) f.FieldType);
}
```

Ekraanipilt 49 Create ja Update jaoks loodud listi ja üksikobjekti meetodid

Nüüd selle tulemuseks on võimalik dünaamiliselt anda päringusse juba kõiki objektiseoseid kaasa (näide järgneval pildil), mis kaotab veelgi ühe hallatavuse probleemi API-st.



Ekraanipilt 50 Mõned automaatselt lisatud Company listiseosed

4.4.7 Vaikimisi CRUD operatsioonid GraphQL väljadena

Kuna *Create*, *Update* ja *Delete* operatsioonid on reeglina erinevate objektide vahel väga sarnased, on mõistlik ka neid refaktoreerida *generic* meetoditeks. Selleks on tehtud *Mutation-i* jaoks laiendusmeetod, mis sooritab *resolve* sees soovitud andmebaasioperatsiooni.

Et luua üks dünaamiline *Create* operatsioon, on kõigepealt vaja lahendada see andmebaasi ligipääsul, ehk *Repository-s*. Seal on kasutatud *dynamic* võtmesõna, mis võimaldab C#-is ilma tüüpe kontrollimata koodi kirjutada. Väiksema projektiga alustades tasuks kindlasti *Id-ga* väärtused pärida ühiselt alamklassilt, et seda vältida, aga antud lahenduses oli see parim võimalik variant, sest genereeritud klassides oli *Id* eraldi.

CommonRepository klassi sisse on tehtud meetod mis kasutab ära *DbContexti DbSet-ide* nimetust – need on projektis läbivalt täpselt sama nimega mis klassid, seega neile on võimalik dünaamiliselt ligi pääseda. Järgneval pildil on näide ühest *generic Create* meetodist *Repository* kihis.

```
public async Task<int> CreateObject<T>(T obj, Type objectType) where T : class
{
    //Tagastab "Id" väärtuse või 0 kui dbcontext kukkus läbi
    dynamic typedInput = Convert.ChangeType(obj, objectType); //peab tegema et T oleks instance
    dynamic dbsetTest = _applicationDbContext.GetPropertyValue(objectType.Name);
    dbsetTest.Add(typedInput);
    await _applicationDbContext.SaveChangesAsync();
    var value = Helpers.ValueTypeHelper.GetPropValue(typedInput, propName:"Id");
    if (value != null)
        return value;
    return 0;
}
```

Ekraanipilt 51 Dünaamilise Create meetodi näide

Peale seda andmebaasioperatsiooni tagastatakse värskelt loodud objekti *Id*, et selle *Id* väärtust kasutada ülejäänud objektivaheliste seoste koostamisel. Nüüd saab värskelt loodud meetodit kasutada *Field-i* sees.

Järgneva *generic Create Field-i* tööle panemiseks jaoks on vaja nüüd vaid ühte meetodit, mille abil on võimalik taaskord iga klassi jaoks koodi kokku hoida.

```
var fieldtoadd = source.Field<IntGraphType, int>()
    .Name("create" + name)
    .ResolveAsync(async ctx =>
    {
        Authorization

        //kasutaja info võtta tokenist
        GraphQLUserContext user = ctx.UserContext as GraphQLUserContext;
        if(!int.TryParse(user.User.Claims.FirstOrDefault(c => c?.Type == "id")?.Value,out int userID))
            throw new UnauthorizedAccessException(message:QueryConstants.QueryUnauthorized);

        var input = ctx.GetGenericArgument(baseType, name:"input");
        var currentTime = DateTime.Now;

        //Todo Reflectionist lahti saada
        input.GetType().GetProperty(name:"CreatedByUserId")?.SetValue(obj:input, value:userID, index:null);
        input.GetType().GetProperty(name:"ModifiedByUserId")?.SetValue(obj:input, value:userID, index:null);
        input.GetType().GetProperty(name:"CreatedOn")?.SetValue(obj:input, value:currentTime, index:null);
        input.GetType().GetProperty(name:"ModifiedOn")?.SetValue(obj:input, value:currentTime, index:null);

        //uue objekti id
        int result = await commonRepository.CreateObject(input, baseType);
        return result;
    });

fieldtoadd.FieldType.AddArgument(name:"input", description:"description", argumentType);
```

Ekraanipilt 52 Ilma tüübita graphql-dotnet Create Field-i sisu

4.4.8 GraphQL väljas sisalduvate objektidega vaikimisi CRUD

Et eelnevas punktis lahendus töötaks ka 4.3.2 punktis alamoperatsioonidega, on vaja resolve sees kaasaantud *Field-id* eraldi läbi käia ning rakendada. Selleks on kasutatud samu 4.3.6 punktis toodud laiendusmeetodeid, et saada vajalik info klassitüübi seoste kohta. Järgnevas pildis on selleks loodud *generic Create Field-i* meetodi sisse pandud kood.

```
var fieldtoadd = source.Field<BooleanGraphType, bool>()
    .Name("update" + name)
    .ResolveAsync(async ctx =>
    {
        Authorization
        var input = ctx.GetGenericArgument(baseType, name: "input");
        var changedFields = ctx.GetArgument<Dictionary<string, object>>(name: "input");

        #region Nested CRUD

        var innerListMutationTypes = baseType.GetInnerMutationListTypes();
        var innerMutationListFields = changedFields.GetInnerMutationListFields();

        CreateOrUpdateInnerObjectListFields(innerListMutationTypes,innerMutationListFields,commonRepository,ctx);
        CreateOrUpdateInnerObjectFields(baseType, changedFields, commonRepository,ctx);

        #endregion
```

Ekraanipilt 53 GraphQL väljas sisalduvate objektidega vaikimisi CRUD

Peale seda on loodud kaks meetodit, millest üks tegeleb üks-mitmele seostega, ehk listidega ning teine üks-ühele seostega, mis on otseselt vastava objekti *Id* väljaga seoses.

Listidele vastav meetod töötab järgnevalt:

- Antakse kaasa *Dictionary* kujul kõik objektilistid
- Otsitakse lubatud operatsioonide listist objektilisti nime
- Nime vastavusel otsitakse igalt listis olevalt objektilt *id* väärtust
- Kui id on 0, luuakse listi uus objekt, kui mitte siis uuendatakse objekti väljasid

Ettevaatlikkuse huvides pole meetodit rekursiivseks tehtud.

```
public static async void CreateOrUpdateInnerObjectListFields(List<InnerMutationStruct> allowedInnerMutations,
    Dictionary<string,object> changedMutationFields, ICommonRepository commonRepository,ResolveFieldContext<object> ctx)
{
    //TÖÖTAB 1x NESTED - kaugemale igaksjuhuks ei lähe, erandid teha käsitsi

    foreach (var changedMutationField in changedMutationFields)
    {
        foreach (var allowedInnerMutation in allowedInnerMutations)
        {
            if (allowedInnerMutation.InnerMutationName.Equals(changedMutationField.Key, StringComparison.OrdinalIgnoreCase))
            {
                foreach (var listobject in (List<object>)changedMutationField.Value)
                {
                    Dictionary<string, object> listObjectFields = (Dictionary<string, object>) listobject;

                    if (listObjectFields.TryGetValue("id",out var objectId) && (int)objectId != 0)
                    {
                        Type baseType = ValueTypeHelper.GetMutationBaseType(allowedInnerMutation.InnerMutationType);

                        bool updateSuccessful = await HandleUpdateMutation(baseType, listObjectFields,commonRepository,ctx);
                    }
                    else
                    {
                        Type baseType = ValueTypeHelper.GetMutationBaseType(allowedInnerMutation.InnerMutationType);

                        int newCommentId = await HandleCreateMutation(baseType,listObjectFields,commonRepository,ctx);
                    }
                }
```

Ekraanipilt 54 Vaikimisi listide Create ja Update meetod

Otseste *Id* seostega objektid uuendatakse järgnevalt:

- Küsitakse samamoodi lubatud muudatuste nimekiri
- Võrreldakse muudetud väljade nimesid lubatud muudatustega
- Kui id on 0, luuakse uus objekt, millelt küsitakse objekti *Id*, kui mitte siis uuendatakse objekti väljasid
- Otsitakse vanemalt *Field-ilt* vastava objekti *Id* nimega välja ning pannakse sinna värskelt loodud objekti *Id*

```
public static async void CreateOrUpdateInnerObjectFields(Type baseType, Dictionary<string,object> fields,
    ICommonRepository commonRepository,ResolveFieldContext<object> ctx)
{
    //objekti seosed, mida on lubatud muuta
    var innerObjectMutationTypes = baseType.GetInnerMutationTypes();

    //muudetud fieldid, mis on eelnevalt päritud listi sees
    var innerMutationObjectFields = fields.GetInnerMutationObjectFields(innerObjectMutationTypes);

    foreach (var objectStruct in innerMutationObjectFields)
    {
        if (objectStruct.fields == null)
            continue;

        var MappedObject = objectStruct.fields.ToObject(objectStruct.baseType);
        var idVal = MappedObject.GetType().GetProperty(name: "Id")?.GetValue(MappedObject);
        if (idVal != null && (int)idVal == 0)
        {
            int newId = await HandleCreateMutationInstance(objectStruct.baseType, MappedObject, commonRepository,ctx);
            if (!fields.ContainsKey(objectStruct.fieldName + "Id"))
            {
                fields.Add(objectStruct.fieldName + "Id",newId);
            }
        }
        else
        {
            bool updateSuccessful = await HandleUpdateMutation(objectStruct.baseType, objectStruct.fields, commonRepository,ctx);
        }
    }
}
```

Ekraanipilt 55 Tavaliste üksikobjektide Create ja Update meetod

4.4.9 CRUD täielikult automatiseerimine

Kõik eelnevad CRUD ülesanded on juba mõneks üksikuks reaks pigistatud – milleks peatuda seal? Kasutades kõiki eelnevaid punkte on lahenduses loodud *GraphQLData.Types.InputTypes.Default namespace*, mille sisse *InputObjectGraphType* tüüpi klassi liigutades, loob rakendus automaatselt sellele objektile vastava optimeeritud väljas sisalduvate alamobjektidega CRUD-i operatsioonid – niiviisi on loodud ühe *foreach loop-iga* üle 90% rakenduse muutusoperatsioonidest, nagu järgnevas näites.

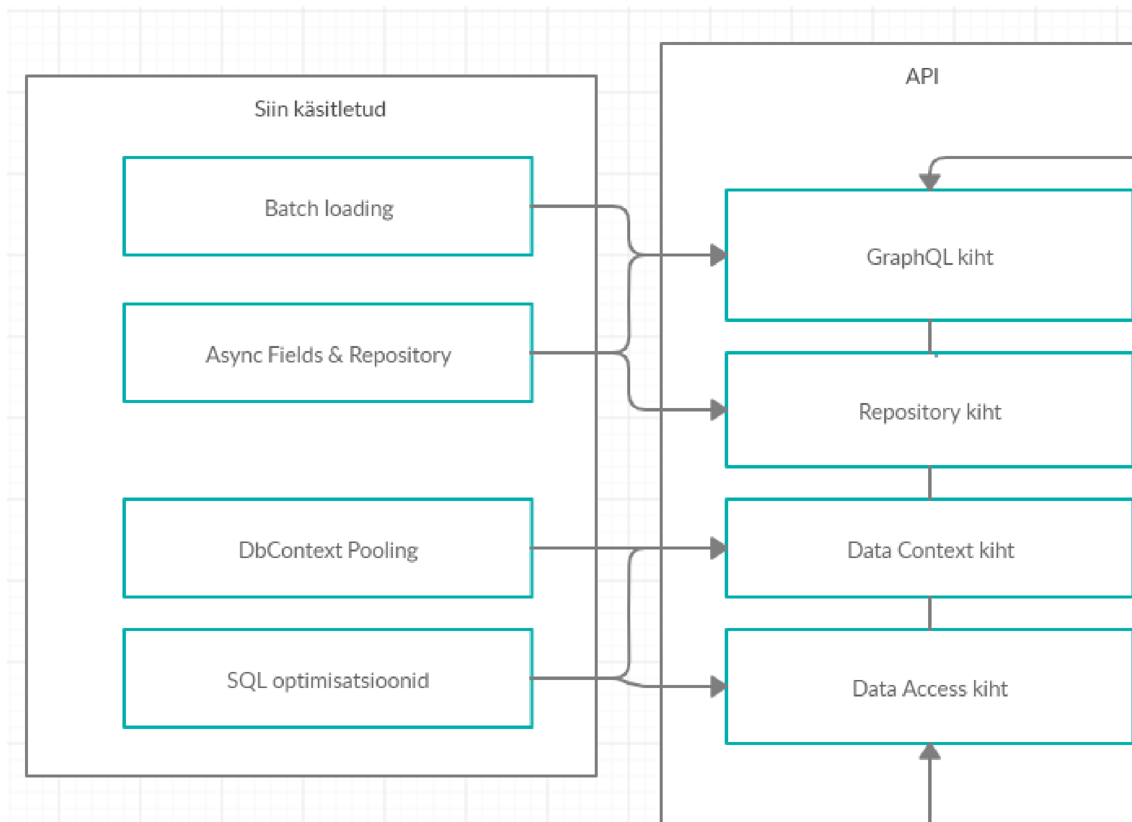
```
1 reference
public class CRUDMutation : ObjectGraphType, IMutationComponent
{
    0 references
    public CRUDMutation(ICommonRepository commonRepository,IUserAccountRepository userAccountRepository)
    {
        var inputTypes = new List<Type>();
        inputTypes.AddRange(collection: NamespaceTypeHelper.GetTypesFromNamespace("GraphQLData.Types.InputTypes.Default"));
        inputTypes.RemoveAll(match: o => o.BaseType!= null && !o.BaseType.IsSubclassOf(c: typeof(InputObjectGraphType)));

        foreach (var inputType in inputTypes)
        {
            var baseType = inputType.BaseType?.GenericTypeArguments[0]; //database class type
            this.AddCreateMutation(inputType.Name, commonRepository, argumentType: inputType, baseType,userAccountRepository);
            this.AddUpdateMutation(inputType.Name, commonRepository, argumentType: inputType, baseType,userAccountRepository);
            this.AddDeleteMutation(inputType.Name, commonRepository, argumentType: inputType, baseType,userAccountRepository);
        }
    }
}
```

Ekraanipilt 56 Namespace põhjal automatiseeritud CRUD

4.5 Jõudluse probleemid ja lahendused

Siin punktis toodud lahendused on seotud veebirakenduse jõudluse parandamisega ning ressursside säästmisega erinevates arhitektuurikihtides.

Järgneval joonisel on nendele lahendustele vastavad arhitektuurikihid.



Joonis 6 Jõudlusega seotud probleemid ning vastavad arhitektuurikihid

4.5.1 Partii korraga laadimine

Kuna andmebaasipäringutel on suur operatsiooniga kaasnev ajakulu (*overhead*), siis võib näiteks ühe firma arvete päring minna väga pikaks, sest iga järgneva alamobjekti jaoks puu struktuuris oleks vaja eraldi andmebaasipäring teha, mida kutsutakse tarkvaraarenduse kommuunis N+1 probleemiks. [18]

Selle lahendamiseks on kõik päringust saadud *Id*-d pandud kokku üheks suureks listiks enne järgmise päringu tegemist, mis kaotab *overhead-i* ja teeb rakenduse seeläbi kiiremaks. Kõigis populaarsemates GraphQL raamistikes on selle jaoks mõni *Facebook DataLoader-i* [19] implementatsioon. Näiteks siin on registreeritud *Company.Id* võti "*GetCustomerCommentByCompany*" nime alla.

```
Field<ListGraphType<CustomerCommentType>, IEnumerable<CustomerComment>>()
    .Name("CustomerComments")
    .AddDefaultFilters()
    .ResolveAsync(async ctx =>
    {
        var loader = accessor.Context.GetOrAddCollectionBatchLoader<int?,
            CustomerComment>("GetCustomerCommentByCompany",
            fetchFunc: (x) => repo.GetCustomerCommentByCompany(x,
                ctx.SubFields,
                ctx.Arguments));
        return await loader.LoadAsync(ctx.Source.Id);
```

Ekraanipilt 57 graphql-dotnet DataLoader implementatsiooni näide

See kogub päringu käigus kõik sama nimega *Id-d* kokku ning teeb alles siis päringu, mis tagastab *ILookup-i* ja sorteerib päringust saadud väljad *Repository* meetodis *Company.CustomerId* põhjal uuesti laiali, nagu all toodud pildi viimasel koodireal.

```
var dbSet = _applicationDbContext.CustomerComment;
var query = ArgumentHelper.HandleDefaultSubQueryArguments(dbSet,arguments,parentTable,
string selection = QueryHelper.GetQuerySelection(fields, typeof(CustomerComment));
var result = query.Select<CustomerComment>(selector: "new(" + selection + ")").ToList();
return result.ToLookup(x => (int?)x.CustomerId);
```

Ekraanipilt 58 ILookup Repository kihis

Selline päring koostab dünaamiliselt SQL lause sisse "*where id in (1, 2, 3 ,4 ,5 ,6...)*" lõpu, mis hakkab avaldama väga suurt mõju kiirusele kui päringu puud suureks lähevad, säästes iga *Id* kohta peale esimest ühe andmebaasipäringu.

4.5.2 SQL optimisatsioonid

Kuna andmebaasilt kõiki tabeliridu pärida lihtsalt selleks et neid hiljem GraphQL kihis eraldada on andmebaasi ressursside ning rakenduse mälu raiskamine, siis on selle jaoks otse SQL-i lause tehtud nii et võtta ainult küsitud tabeliväljad.

Selle jaoks on *Query* kihist *Repository-le* antud kaasa *fields Dictionary*, mis sisaldab kõiki päringus valitud väljasid. Selle põhjal saab *IQueryable Select*() meetodiga küsida otse andmebaasist ainult need väljad mis päringus vajalikud on.

Kuna sellistel olukordadel võib tekkida päring kus on küsitud *Contract*, aga mitte *ContractId*, mille põhjal seda *Contract-i* üldse saaks, siis on järgnevas näites toodud select lauses kohe absoluutselt kõik "*Id*" lõpuga väljad sees, et vältida segadust või kogemata tühjasid päringuid.

```
string selection = QueryHelper.GetQuerySelection(fields, typeof(Contract));
return query.Select<Contract>(selector: "new(" + selection + ")").ToList();
```

Ekraanipilt 59 Dünaamiline Select lause koostamine vajalikest väljadest

Ülal toodud näites *GetQuerySelection*() paneb kokku kõik *Id* lõpuga väljad, lisab kõik väljad *fields* seest ning eemaldab ka *fields* seest väljad, mis tegelikult andmebaasitabelis pole (i.e. kalkuleeritud väljad GraphQL tüübi küljes.).

4.5.3 *Unit of Work* eksemplaride taaskasutus

4.1.2 punkti lõpuks loodud *Unit of Work* kiht töötab niiviisi, et iga API päringu jaoks luuakse uus *DbContext* eksemplar, mis hoiab meeles kõik selle päringu käigus tehtud muudetused, siis kutsutakse päringu lõpuks (või vajadusel manuaalselt varem), *SaveChanges*() meetod, mis koostab vastava andmebaasioperatsiooni, et säästa üleliigseid andmebaasipäringuid ning siis kustutatakse loodud *DbContext* objekt mälust. See on väga hea viis kuidas vähendada rakenduse kihis operatsioone ning seeläbi saavutada turvalisus, et kõik muudetused on kooskõlas ning tehakse samal ajal ning ka parem jõudlus võrreldes eraldi operatsioonidega vähendades edasi-tagasi liikluse kogust andmebaasi ja rakenduse vahel.

Probleem tekib selles, et iga päringu kohta eraldi loodud ja kustutatud *DbContext* objekt on tänu genereeritud mudelite ja kaardistamise üle 12 000 koodirea pikk mis kannab endaga kaasas jõudlusekadu, kui niivõrd suurt objekti on vaja luua ja kustutada iga kliendipoolse päringu jaoks, pole vahet kui väike soovitud päring on.

Selle jaoks on olemasolev lahendus nimega *DbContext Pooling*, mis loob kohe eeldefineeritud koguse *DbContext* eksemplare ning võimaldab päringutel kasutada ühte vabadest eksemplaridest, mis peale tehtud operatsiooni taastatakse kustutamise asemel algseisu.

DbContext Pooling annab kõige märkimisväärsema jõudluse kasvu just väikestel päringuoperatsioonidel, kuna see kaotab ainult ühe konstantse pikkusega operatsiooni päringust – *DbContext-i* loomise. Suurematel operatsioonidel on *DbContext-i* loomise osakaal päringust niivõrd väike, et see ei avalda enam väga suurt mõju. [20]

Entity Framework raamistikus on *DbContext pool-i* loomiseks valmislahendus, selleks kasutame kaasaantud *AddDbContextPool* meetodit, kus saab ka defineerida soovitud *pooli* suuruse. Järgneval pildil on näidis 10 *DbContextist* koostatud poolist.

```
services.AddDbContextPool<ApplicationDbContext>( optionsAction: options =>
    {
        options.UseNpgsql(connectionString);
    }, poolSize:10
);
```

Ekraanipilt 60 DbContext pooling näidis

4.5.4 Asünkroonsed väljad / Asünkroonsed andmebaasioperatsioonid

Andmebaasioperatsioonid on ühe veebirakenduse jaoks tihti kaua aega võtvad operatsioonid – kui rakendus jääb ootama üle võrgu andmebaasis tehtud operatsiooni, läheb selle operatsiooni ning veebiliikluse alla kasulikku aega raisku, mida oleks parem kasutada millegi muu jaoks. Selleks, et vältida rakenduse seiskumist iga andmebaasioperatsiooni ootamisel, on läbi asünkroonsete operatsioonide loodud need analoogselt restorani kelneri tööga – võetakse tellimus, liigutakse edasi kuniks tellimust täidetakse ning antakse soovitud tellimus üle alles siis kui see valmis saab.

Selle jaoks on andmebaasile ligipääsukihis tehtud meetodid *Taskidena* – Need meetodid ei tagasta otse tulemust, vaid tagastavad *Taski*, mis on antud rakendusele järjekorda ootamiseks, et hiljem selle *Taski* tulemusega midagi edasi teha. C#-is on asünkroonsed meetodid märgistatud *async* võtmesõnaga ning teevad kompileerumisel *await* sõnaga märgitud meetodid asünkroonseteks operatsioonideks, millest saab kood edasi liikuda. Ilma *await* sõnata on võimalik kasutada sama meetodit sünkroonselt. Järgneval pildil on üks lihtne näide asünkroonsest *Repository* meetodist, mis ootab *Company* objektide listi andmebaasipäringust.

```
public async Task<IEnumerable<Company>> GetCompanies(IDictionary
{
    var query = _applicationDbContext.Company.AsNoTracking();

    return await query.ToListAsync();
}
```

Ekraanipilt 61 Asünkroonse Repository meetodi näide

Samamoodi nagu tehakse andmebaasioperatsioonid asünkroonseks, on võimalik ka *Field-i resolver-it* teha asünkroonseks – ehk anda *Field-ile* väärtusele andmiseks mingisugune käsk ning kohe liikuda järgmisele *Field-ile* edasi.

Selleks on projekti *graphql-dotnet* raamistikus kaasaantud *ResolveAsync* meetod, mis kasutades kooskõlas 4.5.1 punktis toodud *DataLoader-iga* moodustab kaks kihti asünkroonsust. Esiteks ootab *DataLoader* järgnevas näites *GetVatBankAccountsById* nime alla registreeritud võtmete kogunemist. Teiseks on veel kiht asünkroonsust *Repository* meetodis, mis sooritab vastavalt kogunenud võtmetele andmebaasipäringu.

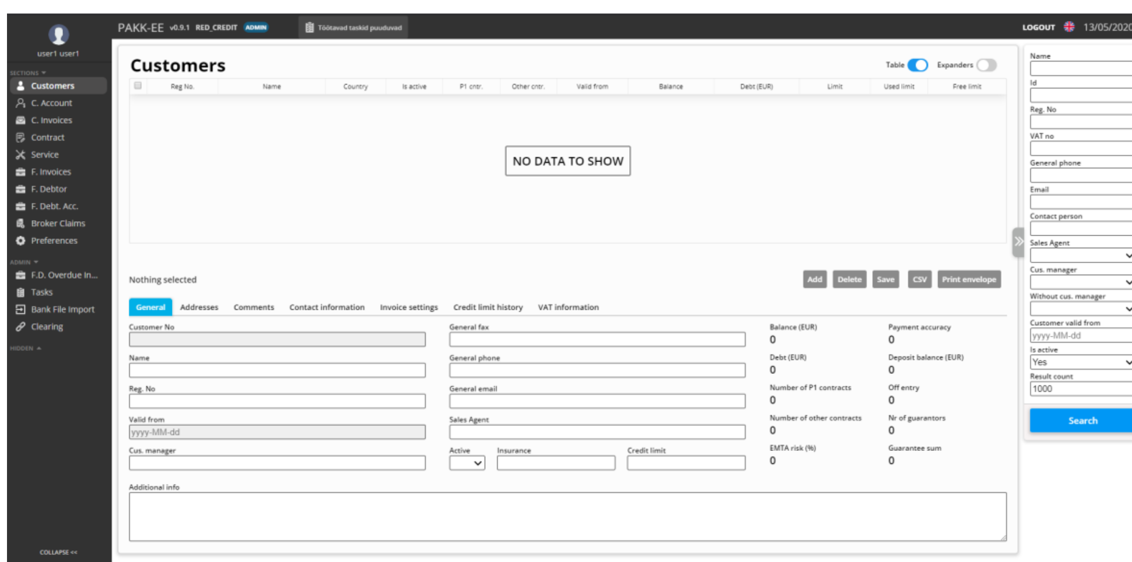
```
Field<ListGraphType<CustomerVatBankAccountType>, IEnumerable<CustomerVatBankAccount>>()
    .Name("CustomerVatBankAccounts")
    .AddDefaultFilters()
    .ResolveAsync(async ctx =>
    {
        var loader = accessor.Context.GetOrAddCollectionBatchLoader<int?, CustomerVatBankAccount>("GetVatBankAccountsById",
            fetchFunc: async (x) => await repo.GetVatBankAccountsById(x, ctx.SubFields, ctx.Arguments));
        return await loader.LoadAsync(ctx.Source.Id);
    });
```

Ekraanipilt 62 ResolveAsync näide Fieldil

5 Tulemused ja valideerimine

Selles peatükis hindab töö autor realisatsiooni tulemust ning võrdleb seda potentsiaalse konkureeriva rakendusega.

Realisatsiooni tulemusel sai loodud töötav ning osaliselt juba päriskasutuses API, mis võimaldab kasutajatel internetikeskkonnas ligi pääseda vajalikele andmetele. Lisaks sellele lõi töö autori kursusekaaslane ka rakendusele kasutajaliidese, millega saavutasime kokkuvõtteks sama baasfunktsionaalsuse nagu originaalsel töölauarakendusel. Järgneval pildil on kursusekaaslase poolt loodud kasutajaliidese sama vaade mis 2. peatükis toodud 1. ekraanipildil.



Ekraanipilt 63 Realisatsiooni tulemuse kasutajaliides

Loodud rakendust on võimalik kasutada paralleelselt olemasoleva töölauarakendusega, kasutades sama kasutajainfot ning andmebaasi. Praegusel kujul suudab rakendus täita sellele määratud ülesanded, võimaldades rahuldava jõudlusega läbi veebikeskkonna sooritada kõiki päringuid, mis ka originaalses rakenduses.

Uue rakenduse valideerimiseks pakkusime seda paralleelselt töölauarakendusega kasutada päriskasutajatel, kellelt saime positiivset tagasisidet. Loodud serveri arhitektuur ei ole kahjuks võrdväärsetel tingimustel võrreldav töölauarakendusega, kuna töölauarakenduses sooritatakse palju väiksemaid päringuid, mis on GraphQL API-s

pandud kokku jõudluse paranduse eesmärkidel suuremateks, et veebiserveri koormust läbi päringute arvu vähendada.

Valmisrakenduses on ~50 000 koodirida, millest ~25 000 on automaatselt genereeritud andmebaasi põhjal. Alates juunist on tehtud ~500 commiti. (Lisa 2)

Tööga alustades ei tulnud autor selle peale, et lahendus võiks olla täielikult automatiseeritud, ning selle pärast ei otsitud võrdluseks sellele midagi sarnast. Realisatsiooni valmimise ajaks otsis töö autor sellele tagantjärele sarnaseid lahendusi, millest kõige lähemale loodule jõuavad Prisma *database toolkit* ning PostGraphile. Prisma loob andmebaasi põhjal valmispäringud ning CRUD meetodid, et neid kõrgemates arhitektuurikihtides kasutada. PostGraphile läheb sealt veel sammu kaugemale ning loob andmebaasimudeli põhjal terve täisfunktsionaalsusega GraphQL API. Kuna PostGraphile on kahest raamistikust loodud realisatsiooni tulemusele sarnasem kasutan seda raamistikku võrdluseks, et hinnata loodud lahendust.

5.1 PostGraphile lahendusega võrdlus

PostGraphile loob otse *PostgreSQL* andmebaasimudeli põhjal automatiseeritud GraphQL API, mida on võimalik ka edasi laiendada ning vastavalt vajadustele kohandada. PostGraphile lahendab paljud samad probleemid, mis on ka töö realisatsiooni lahenduses, kuid miinuseks ei paku soovitud kontrolli ning kindlust andmebaasimuudatuste vastu võrreldes päris äritarkvaralahendusega.

PostGraphile töötab Node.js tehnoloogial ning on baasvajaduste jaoks väga lihtne tööle saada – üks käsurida võimaldab töötava GraphQL API otse andmebaasikasutaja õiguste põhjal luua. Kõik sätted ning lisamuudatused PostGraphile jaoks võetakse otse andmebaasist, läbi lisanduvate JSON kommentaaride väärtuste.

Lisaks käsurea stiilis kasutusele võib ka seda raamistikku kasutada koodipõhjana, mis võimaldab raamistikus defineeritud kohtadele laienduseks või muutusteks pistikprogramme luua. Niiviisi kasutades on see lahendus võrreldav selle töö raames loodud realisatsiooniga.

PostGraphile rakenduses on lahendatud paljud ühised tüüpprobleemid, selles peatükis on võrreldud PostGraphile lahendusi ning nendega kaasnevaid eeliseid ja puudujääke võrreldes loodud realisatsiooniga.

Lahenduste vahel on palju sarnast, mõlema alguspunkt on PostgreSQL andmebaas, mõlema lõpp-punkt on GraphQL API. Mõlemas lahenduses on maksimaalselt süsteemi üritatud automatiseerida, millest mõnes kohas on töö realisatsiooni isegi teadlikult vähem automatiseeritud.

Mõlemat lahendust on võimalik puhta SQL-iga laiendada, lisaväljadega laiendada. Samuti on mõlemas võimalik enamus funktsionaalsust vastavalt vajadusele kohandada. Üks esimesi eeliseid mis küll antud projekti puhul oli realisatsioonil, on osaliselt C# koodi loogika taaskasutamine, sest töölauarakendus oli samas keeles.

5.1.1 Päringute arvu vähendamine ning N+1 probleem

Nii realiseeritud rakenduses kui ka PostGraphile rakenduses on kaotatud kõik N+1 päringu probleemid – see pakub väga suure kasvu jõudluses, kuna ilma partiideks jaotamiseta kasvab andmebaasipäringute arv lineaarselt tulemuste arvuga, mis korrutub iga päringu puu hargnemisel.

PostGraphile lahenduses on see isegi veel paremini lahendatud. Olgugi et mõlemas lahenduses on iga päringu tulemus konstantne kogus andmebaasipäringuid, siis PostGraphile paneb absoluutselt kõik päringu osad ühte select lausesse – muutes näiteks 5 partii päringut loodud realisatsioonis vaid üheks, vähendades veelgi edasi-tagasi liiklust.

PostGraphile-s kutsutakse eeltoodud lahendust Look-Ahead [21] funktsionaalsuseks, mille käigus pannakse otse SQL päringu sisse vajalikud meta-andmed ning objektid päringu puu struktuurist, millest koostatakse otse andmebaasikihis soovitud tulemusega objekt. Selle jaoks on kasutatud päringu sees JSON objekte, mis pannakse järjest suuremateks objektideks kokku ning tagastatakse juba otse andmebaasist kokku pandud GraphQL päringu vastusena.

Niiviisi päringute koostamine võib pakkuda suure kasvu jõudluses ning tasuks kindlasti kasutusele võtta kohtades, kus jõudlus võib probleem olla. Kahjuks on sealt võidetud jõudlusega kaotatud aga keerukam eelfilterdamise võimalus igal GraphQL väljal.

PAKK rakenduses on näiteks mõne filtri rakendamise jaoks vaja eraldi andmebaasipäring ja eeltöötlus teha, mida ei oleks võimalik SQL sees teha. Lisaks sellele on vaja mõnes kohas eraldi välja *resolver-is* sooritada päringuid näiteks välisele API-le, mis oleks PostGraphile tavapärase lahendusega võimatu.

5.1.2 GraphQL *Schema* laiendamine

GraphQL *Schema* laiendamine on PostGraphile suurim nõrkus, kuna see lahendus on mõeldud andmebaasile lihtsaks ja efektiivseks ligipääsuks, mitte keeruliseks äritarkvara ehituseks.

PostGraphile rakenduses on võimalik andmebaasikommentaaridega muuta tabelite ning väljade kajastust GraphQL-is, lisaks sellele on võimalik pistikprogrammi stiilis lisada väljasid ning uusi *Type-sid*, samuti on võimalik ka muuta automaatselt väljade genereerimisprotsessi.

PostGraphile kasutus koodipõhjana ning sinna pistikprogrammide lisamine, võimaldab lahendada mõned probleemid, kuid need probleemid lähevad ülalt-alla muutes siiski niivõrd mahukaks, et lihtsam on süsteem nullist ise ehitada.

PostGraphile kaudu muudatusi tehes tekib üks suuremaid raskusi sellest, et pistikprogrammi stiilis väljadel ei ole tüübikindlust ega mingisugust polümorfismi lahendust. Konkreetse näitena väga paljudel tabelitel PAKK rakenduses on *ModifiedOn* väli, millele oleks mõne tabelirea muutmisel väärtus anda. PostGraphile-s oleks vaja sellele teha kas väga palju erinevaid pistikprogramme või süsteem täielikult lahti võtta.

5.1.3 Andmebaasi põhjal GraphQL *Schema* automaatne loomine

Mõlemas lahenduses on suurem osa kogu GraphQL schemast automatiseeritult loodud. Töö realisatsioonis on teadlikult jäetud sellest automatiseerimisest kaks kihti välja: esimene neist on Query juures asuvad päringud, et otsest ligipääsu piirata ning teine on *Type-de* vahelised seosed päringutes.

PostGraphile-s on iga tabeli võtme järgi loodud seos teise tabeliga kajastatud otse GraphQL *Type-l*, näiteks *userByAuthorId*. Sellised seosed võimaldavad väga lihtsat ligipääsu ühelt väljalt teisele, kuid rakenduse turvalisuse ning kontrolli huvides jättis töö autor realisatsioonis selle võimaluse täielikult kasutamata. Selleks, et jätta näiteks mõni konkreetne seos mitmest kohast välja, on jällegi vaja PostGraphile-s teha mitu pistikprogrammi, et neid muudatusi teostada.

5.1.4 CRUD

Üks asi, mis on realisatsioonis, kuid puudub täielikult PostGraphile-s, on väljas sisalduvate objektide CRUD, ehk võimalus muuta või lisada mitut objekti korraga.

PostGraphile-s on sarnaselt 4.4.9 punktis toodud lahendusele loodud iga tabeli jaoks *create*, *update* ning *delete* operatsioonid, kuid sellest on täielikult jäetud välja 4.4.8 punktis toodud sisemiste objektide muutmise võimalus. PostGraphile-s on CRUD loodud andmebaasikasutaja tasemel, ehk igat tabelit saab muuta vastavalt andmebaasikasutaja õigustele.

Kui realiseeritud rakenduses oleks vaja lisada kirjeid tabelisse, kuhu kasutaja ligipääsu ei soovita, näiteks midagi logi sarnast, on võimalik seda PostGraphile-s pistikuna lahendada.

Üks suurimaid probleeme, mis CRUD käigus algul ilmsiks ei tule on PostGraphile-s kõikide pisikeste automatsioonide muutmine, mis selle äritarkvaraks muutmisel vaja teha on. Näiteks on vaja võtta *ModifiedByUserId* väärtus *JSON web token-ist*, kontrollida enne muudatusi vastavaid õigusi läbi puhvri, sooritada mõni välise API päring – kõiki selliseid probleeme on vaja hakata eraldi lahendama.

5.1.5 Filtrite kasutamine, *Argument-id*

PostGraphile vaikimisi filtrid võimaldavad absoluutselt iga tabelivälja väärtuse järgi filterdada ning ka sorteerida. Samuti on ka olemas võimalus tulemuste arvu piirata ning tulemusi vahele jätta. Konkreetse väärtuse filterdamise võimalused on kõik pandud *condition* nimega *Argument-i* sisse, nagu toodud järgneval pildil *companyContactPeopleByCompanyId* väljal.

```
{
  allCompanies(first: 1) {
    nodes {
      id
      companyContactPeopleByCompanyId(first:3,
        condition:{id:3, languageCl:"LANGUAGE.ET-EE"}) {
        nodes {
          id
          languageCl
          addressByAddressId{
            countryByCountryId{
              code
            }
          }
        }
      }
    }
  }
}
```

Ekraanipilt 64 PostGraphile filterdamise näide

Funktsioonid mis oleks vaja eraldi lisada oleks PAKK rakenduse stiilis otsingu rakendamine, ehk täpse vaste asemel tulemust alguse järgi otsida, mis oleks tõenäoliselt ilma suurtemate raskusteta muudetav, kuna läbivalt genereerimisprotsessi muutmise võimalus on PostGraphile-s olemas.

See mille muutmine tekitaks tõelist peavalu pistikprogrammina, on läbi navigatsiooniväljade seoste ning täiesti eraldi päringute tehtud filtrid. Neid oleks võimalik küll ühekaupa väljadena lisada, kuid neid on niivõrd palju, et sealt jõutaks kohe sama hallatavusprobleemi juurde nagu realisatsioonis.

6 Millele edasi mõelda

Selles peatükis räägib töö autor projekti edasiarenduseks tekkivatest küsimustest ning selle töö raames lahendamata jäetud probleemidest.

6.1 Potentsiaalselt raamistiku muutmine

Projekti alustushetkel oli C# GraphQL raamistikest ainus kasutatav valik *graphql-dotnet*. Samas praeguseks projekti valmimishetkeks on tekkinud mõned teised ahvatlevad raamistikud, millest üks paremaid on *Hot Chocolate* [22]. Hot Chocolate oli projekti

alustusel veel poolik ning selle raamistiku sees on paralleelselt töö autori projekti loomise ajaga lahendatud paljusid sarnaseid probleeme.

Hot Chocolate-s on praeguseks mõned lisafunktsioonid ning lahendused, mis selle töö realisatsioonis pidi (või peab veel) iseseisvalt lahendama ning pakuksid uue projekti jaoks parema alguspunkti kui *graphql-dotnet*, näiteks:

- *Schema Stitching* (lahendatud 4.3.1 punktis)
- Hallatav päringu sorteerimine ja filtreerimine otse raamistikus (sarnane lahendus 4.3.6 punktis)
- *Repository* meetodite GraphQL-i automaatselt ületõstmine (pole antud töös lahendatud, suurim põhjus miks raamistikku vahetada)
- Automaatne *Type-le* väljade lisamine (lahendatud 4.3.3 ning 4.4.1)
- Parem *Common Fields* süsteem (lahendatud 4.3.4 punktis)
- GET päringud veebibrauseri puhverdamise eesmärkidel (lahendamata)
- Päringu salvestamine id järgi, et seda identselt korrata (lahendamata).

Kuna *graphql-dotnet* raamistikus pole ühtegi funktsiooni, mis puudub *Hot Chocolate*-st, tasub hoida eelnevas lõigus toodud lahendamata punktide pealt aega kokku ja vahetada praegune raamistik selle vastu välja. Selleks on vaja muuta kõik *Field* ja *Type* loomiseks mõeldud klassid uue raamistiku süntaksi peale ning vastavalt *Hot Chocolate* dokumentatsioonile võtta kasutusele eelneva lõigu punktides toodud valmislahendused.

6.2 Päringute valiku piiramine ning päringu enda suuruse vähendamine veebiliikluse jaoks

Päringu valiku piiramisel peaks mõtlema lisaks autoriseerimisele. *graphql-dotnet* raamistikus on kaasaantud autoriseerimislahendus integreeritud *resolver-i* sisse.

Rakenduse laia päringuvaliku ning vabaduse tõttu võib tekkida olukord kus kogemata mõne seose kaudu võib mööda võrku jõuda klient millegini, mida ta ei tohiks näha. Selle jaoks oleks üks lahendus pakkuda näiteks mõne rakenduse osa jaoks täiesti eraldi *Schema*. PAKK rakenduses sai tehtud näiteks sisselogimise jaoks täiesti eraldi GraphQL API, tänu millele oli järgnevas näites võimalus panna tervele ülejäänud rakenduse *Schema-le* sisse

logitud oleku nõude, peites kogu API ülesehituse ning potentsiaalsed päringud välise silma eest.

```
public PAKKSchema(IDependencyResolver resolver) : base(resolver)
{
    Query = resolver.Resolve<MainQuery>();
    Query.AuthorizeWith(policy: "User");
```

Ekraanipilt 65 Terve GraphQL Schema autoriseerimise nõude näidis

Samuti tasuks ka mõelda GraphQL *Fragment-ide* funktsionaalsuse kasutamisele ning mõne populaarsema päringu serveripoolele salvestamisega, kui päringud ise suuremaks lähevad – selle projekti raames läks mõni suurem päring üle 150 rea pikaks, sealt on võimalik juba päringu enda poolt veebiliiklust kokku hoida. Eelmises punktis toodud *Hot Chocolate* raamistikus on see probleem juba osaliselt lahendatud.

6.3 Pahatahtlike päringute piiramine / DDOS kaitse

Andes ühele päringule väga palju vabadust, on võimalik seda pahatahtlikult ära kasutada. Kui ühele päringu keerukusele piirangut ei panda, on võimalik, et keegi saadab sinna pidevalt väga suure keerukusega päringuid, mis koormab veebirakendust nii palju, et kliendid ei saa rakendust kasutada.

Selle jaoks on *GraphQL .NET* raamistikus olemas *ComplexityConfiguration* klass, et piirata seda kui süvitsi või millise maksimaalse keerukusesummaga operatsioone klient saab pärida. Kasutades seda klassi on vaja keerukusarvutuse algoritm ise luua, pidades meeles, et see peab ka olema kiire, et mitte liigset lisakoormust päringule endale anda.

6.4 Testimine

Kuna antud projekt sai loodud vaid kahe inimese poolt, mis oli veel eraldi jaotatud kasutajaliideseks ning API-ks, otsustas töö autor teha testid vaid kõige tähtsamatele operatsioonidele ning läbi võimalikult suurte päringute, kuna kitsama sihiga testid oleks lihtsalt pidurdanud ülejäänud arenduse kiirust.

Arendamise käigus leidis töö autor kaks sobilikku viisi, kuidas loodud rakendust testida, millest esimene on läbi *middleware* kihi *GraphQL* päringute testimine.

Selleks et testida ühte *GraphQL* päringut, on üks variant kasutada otse *GraphQL Middleware-i* klassi, Simuleerides seda ise loodud *HttpContext-i* abil. Niiviisi testimiseks on kõigepealt vaja luua *middleware* objekt, näiteks nii:

```
var middleware = new GraphQLMiddleware(
    next: next => null,
    writer: _serviceProvider.GetRequiredService<IDocumentWriter>(),
    executor: _serviceProvider.GetRequiredService<IDocumentExecuter>(),
    new GraphQLSettings());
```

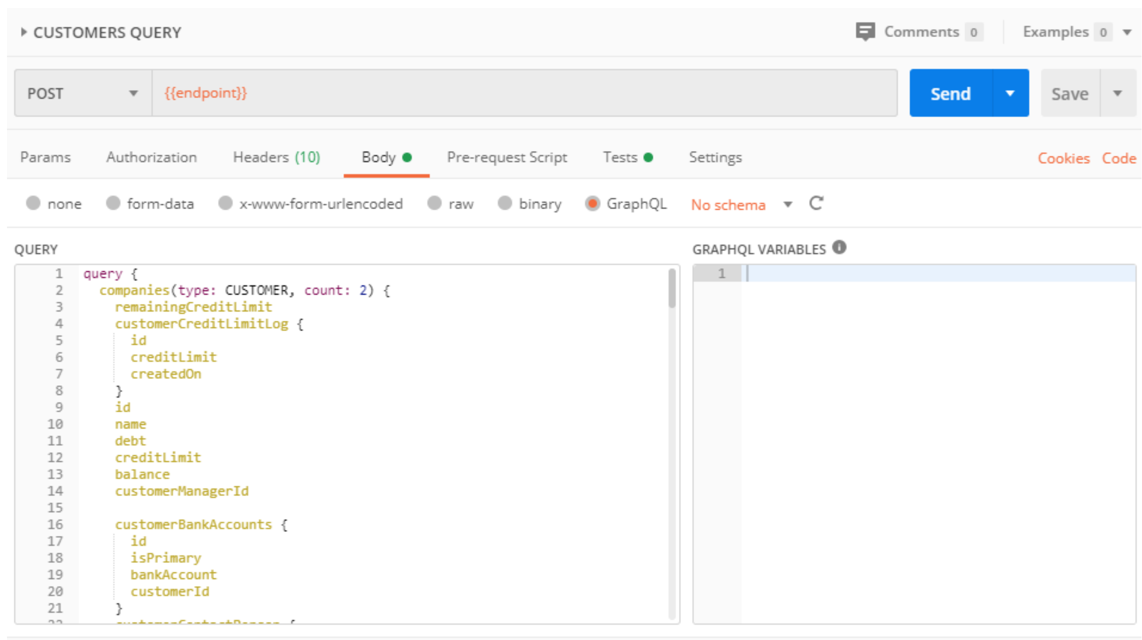
Ekraanipilt 66 Middleware loomise näide

Peale *middleware* loomist on järgnevas näites .NET raamistikust võetud *HttpContext*, mida on kõige lihtsam luua läbi *DefaultHttpContext-i* konstruktori ning siis on vaja loodud *contexti middleware* sees kasutada, saades sealt soovitud tulemust edasi töödelda ning testides kasutada.

```
var context = new DefaultHttpContext();
context.Response.Body = new MemoryStream();

await middleware.Invoke(context, schema: _serviceProvider.GetRequiredService<PAKKSchema>(), _serviceProvider);
context.Response.Body.Seek(offset: 0, SeekOrigin.Begin);

var reader = new StreamReader(context.Response.Body);
var streamText = reader.ReadToEnd();
```

Ekraanipilt 67 HttpContexti kaudu päringu simuleerimine

Teine hea viis testida on kasutades välist tarkvara *Postman*, mis võimaldab läbi päris HTTP päringute kogu rakendust testida. Kuna *Postman-is* saab rakendust nii testkeskkonnas kui ka päris töökeskkonnas vaid URL-i muutmisega testida ning ka vajadusel kombineerida mitmeid API-sid testides, otsustasime testida tähtsamad kliendipoolsed päringud seal. Järgneval pildil on näide ühest sellisest päringust *Postman-is*.



Ekraanipilt 68 Postman keskkonnas päringu näide

Postman-is testimiseks lõi töö autor samas rakenduses testide kollektsiooni, mis jooksutab nupuvajutusel järjest eeldefineeritud HTTP päringud kaasaarvatud rakendusse sisselogimisega ning sooritab neile samas raamistikus kirjutatud testid.

7 Kokkuvõte

Käesoleva bakalaureusetöö eesmärgiks oli moderniseerida aegunud tehnoloogiaga töölauarakendus GraphQL tehnoloogiale ning näidata võtteid, millega lahendada sellega kaasnevaid tüüpprobleeme.

Peamised probleemid seisnesid selles, et olemasolev rakendus oli mahukas ning GraphQL tehnoloogia on veel uus, seega puudus vajalik info kuidas seda skaleeritavalt kasutada.

Bakalaureusetöö käigus sai autor paremad teadmised, kuidas suuri tarkvaraprojekte arhitektuuriliselt üles ehitada ning kuidas disainida ühe GraphQL serverit. Lisaks sellele õppis töö autor ka mitmeid automatiseerimisvõtteid, millega võimalikult palju koodi taaskasutada.

Töö eesmärk sai täidetud. Tulemuseks loodud rakendus sai proovitud päriskasutajate poolt ning vastas ootustele. Kuna projekti alustushetkel oli raamistike valik väga väike ning alles tekkimas, on praeguseks mõnes raamistikus osad siin töös mainitud probleemid juba ette lahendatud. Töö käigus sai ka leitud mitmeid lahendusi, mida ei ole töö autor veel mujal näinud.

Kuna projekti raames kasutatud raamistikus on vaja paljud praegusel hetkel pakutavad valmislahendused ise luua, soovitab töö autor võtta parema alustuspunktina *GraphQL .NET* raamistiku asemel *Hot Chocolate* raamistiku.

Kasutatud kirjandus

[1] T. Eizinger, *API Design in Distributed Systems: A Comparison between GraphQL and REST,* 2017, pp. (39-43).

[2] „GraphQL ametlik leht,“ [Võrgumaterjal]. Available: https://graphql.org/. [Kasutatud 3 5 2020].

[3] „GraphQL .NET Dokumentatsioon,“ [Võrgumaterjal]. Available: https://graphql-dotnet.github.io/. [Kasutatud 3 5 2020].

[4] R. E. Mike Mintz, Hardware Verification with C++: A Practitioner's Handbook. United States of America: Springer. p. 22. ISBN 978-0-387-25543-9., 2006.

[5] E. Gamma, J. Vlissides, R. Helm ja R. Johnson, Design Patterns: Elements of Reusable Object-Oriented Software, 1994.

[6] R. Milner, L. Morris ja M. Newey, "A Logic for Computable Functions with Reflexive and Polymorphic Types". Proceedings of the Conference on Proving and Improving Programs., 1975.

[7] „Microsoft SSMS dokumentatsioon,“ [Võrgumaterjal]. Available: https://docs.microsoft.com/en-us/sql/ssms/sql-server-management-studio-ssms?view=sql-server-ver15. [Kasutatud 3 5 2020].

[8] M. Fowler, „Unit of Work Pattern,“ [Võrgumaterjal]. Available: https://martinfowler.com/eaaCatalog/unitOfWork.html. [Kasutatud 3 5 2020].

[9] „EF Core Reverse Engineering dokumentatsioon,“ [Võrgumaterjal]. Available: https://docs.microsoft.com/en-us/ef/core/managing-schemas/scaffolding. [Kasutatud 3 5 2020].

[10] „Npgsql dokumentatsioon,“ [Võrgumaterjal]. Available: https://www.npgsql.org/. [Kasutatud 3 5 2020].

[11] „Apollo GraphQL Dokumentatsioon,“ [Võrgumaterjal]. Available: https://www.apollographql.com/docs/graphql-tools/schema-stitching/. [Kasutatud 3 5 2020].

[12] M. Fowler, „Inversion of Control Containers and the Dependency Injection pattern,“ [Võrgumaterjal]. Available: https://martinfowler.com/articles/injection.html. [Kasutatud 3 5 2020].

[13] R. C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, 2008.

[14] „GraphQL .NET source kood, AutoRegisteringObjectGraphType klass,“ [Võrgumaterjal]. Available: https://github.com/graphql-dotnet/graphql-dotnet/blob/master/src/GraphQL/Types/Composite/AutoRegisteringObjectGraphType.cs. [Kasutatud 3 5 2020].

[15] „EntityFrameworkCore kommuuni poolt loodud dokumentatsioon, Add operatsiooni jõudlus,“ [Võrgumaterjal]. Available: https://entityframework.net/improve-ef-add-performance. [Kasutatud 3 5 2020].

[16] „Microsoft EntityFramework dokumentatsioon,“ [Võrgumaterjal]. Available: https://docs.microsoft.com/en-us/ef/ef6/fundamentals/performance/perf-whitepaper. [Kasutatud 3 5 2020].

[17] J. Singleton, „ASP.NET Core 2 High Performance,“ 2017, p. 215.

[18] G. Kuizinas, „"Using DataLoader to batch requests", N+1 probleemist,“ [Võrgumaterjal]. Available: https://medium.com/@gajus/using-dataloader-to-batch-requests-c345f4b23433. [Kasutatud 3 5 2020].

[19] „DataLoader,“ [Võrgumaterjal]. Available: https://github.com/graphql/dataloader. [Kasutatud 3 5 2020].

[20] A. Vickers, „DbContext Poolingu kohta täpsustused ühe raamistiku looja poolt,“ [Võrgumaterjal]. Available: https://github.com/dotnet/efcore/issues/10125. [Kasutatud 3 5 2020].

[21] „PostGraphile Look-Ahead,“ [Võrgumaterjal]. Available: https://www.graphile.org/graphile-build/look-ahead/. [Kasutatud 3 5 2020].

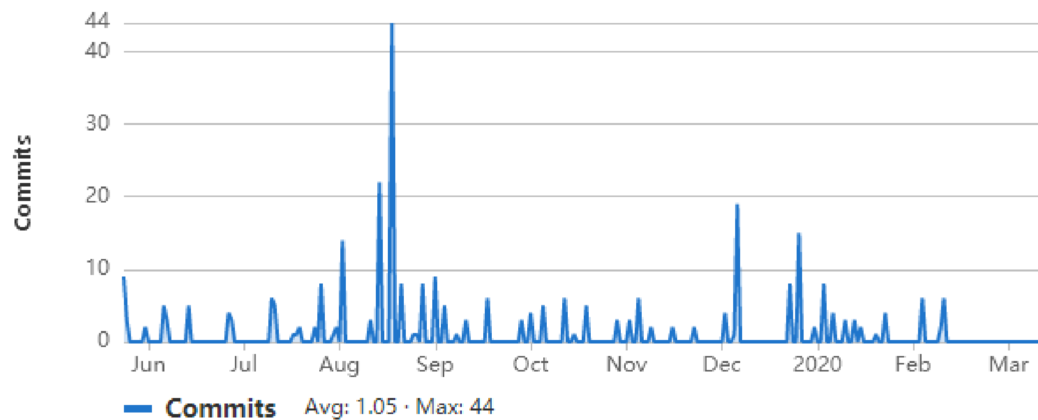
[22] „Hot Chocolate,“ [Võrgumaterjal]. Available: https://hotchocolate.io/. [Kasutatud 3 5 2020].

Lisa 1 – SQL Script C# klasside loomiseks

```
declare @tableName varchar(200)
declare @columnName varchar(200)
declare @nullable varchar(50)
declare @datatype varchar(50)
declare @maxlen int

declare @sType varchar(50)
declare @sProperty varchar(200)

DECLARE table_cursor CURSOR FOR
SELECT TABLE_NAME
FROM [INFORMATION_SCHEMA].[TABLES]

OPEN table_cursor

FETCH NEXT FROM table_cursor
INTO @tableName

WHILE @@FETCH_STATUS = 0
BEGIN

PRINT 'public class ' + Upper(left(@tableName,1)) +
Lower(SUBSTRING(@tableName,2,Len(@tableName))) + ' {'

    DECLARE column_cursor CURSOR FOR
    SELECT COLUMN_NAME, IS_NULLABLE, DATA_TYPE,
isnull(CHARACTER_MAXIMUM_LENGTH,'-1')
  from [INFORMATION_SCHEMA].[COLUMNS]
       WHERE [TABLE_NAME] = @tableName
       order by [ORDINAL_POSITION]

    OPEN column_cursor
    FETCH NEXT FROM column_cursor INTO @columnName, @nullable, @datatype, @maxlen

    WHILE @@FETCH_STATUS = 0
    BEGIN

       -- datatype
       select @sType = case @datatype
       when 'int' then 'int'
       when 'decimal' then 'decimal'
       when 'money' then 'decimal'
       when 'char' then 'string'
       when 'nchar' then 'string'
       when 'varchar' then 'string'
       when 'nvarchar' then 'string'
       when 'uniqueidentifier' then 'Guid'
       when 'datetime' then 'DateTime'
       when 'bit' then 'bool'
       else 'string'
       END
              if (@sType = 'string' and @maxLen <> '-1')
                     Print '[MaxLength(' +  convert(varchar(4),@maxLen) + ')]'
```

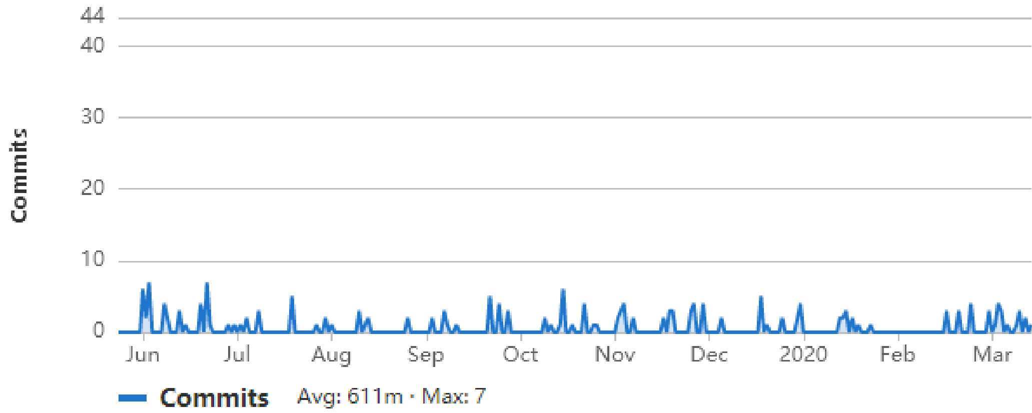
```
            SELECT @sProperty = 'public ' + @sType+
            iif(@nullable = 'YES','?','')
             + ' ' + Lower(@columnName) + ' { get; set;}'
            PRINT @sProperty

            print ''
            FETCH NEXT FROM column_cursor INTO @columnName, @nullable,
@datatype, @maxlen
       END
    CLOSE column_cursor
    DEALLOCATE column_cursor

       print '}'
       print ''
    FETCH NEXT FROM table_cursor
    INTO @tableName
END
CLOSE table_cursor
DEALLOCATE table_cursor
```

Lisa 2 – Commitide logi, koodi meetrika



ian

184 commits (jan.varatalu@gmail.com)



Hierarchy	Lines of Source code	Lines of Executable code
EntityFrameworkData (Debug)	31,431	11,604
GraphQLData (Debug)	13,821	3,861
Helpers (Debug)	4,686	707
PAKKWeb (Debug)	1,170	310
Tests (Debug)	548	195