TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Reiko Roopärg 194118IAIB

Andmebaasis kirjutamise skeemi ja JSON dokumentide kasutamise mõju PostgreSQL ja MongoDB andmebaasirakenduste loomisele

Bakalaureusetöö

Juhendaja: Erki Eessaar
PhD

Tallinn 2022

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: Reiko Roopärg

30.05.2022

Annotatsioon

Andmebaasis, kus andmebaasi tasemel on defineeritud ja jõustatud andmete struktuur ja piirangud andmetele, on kirjutamise skeem. Aknapõhine andmebaasirakendus pakub andmete lugemiseks ja muutmiseks graafilise kasutajaliidese. Käesoleva töö eesmärgiks on uurida andmebaasis kirjutamise skeemi kasutamise (andmebaasi struktuuri kirjeldamise, kitsenduste jõustamise) ning andmete JSON dokumentidena salvestamise mõju aknapõhiste andmebaasirakenduste loomisele. Eesmärgi täitmiseks viiakse läbi eksperiment. Selleks kavandatakse erinevate disainidega andmebaasid ning realiseeritakse igale disainile vastav veebipõhine andmebaasirakendus. Nõuded andmetele (milliseid andmeid on vaja ja millised on nendele kehtivad piirangud), rakenduse funktsionaalsed ja mittefunktsionaalsed nõuded ning rakenduse kasutajaliides on kõikidel juhtudel ühesugused, kuid kasutatakse erinevaid andmebaasisüsteeme ja andmebaasi füüsilise disaini lahendusi. Võimalikult palju andmete kontrolle realiseeritakse deklaratiivsel viisil andmebaasi tasemel. Kui see pole võimalik, siis realiseeritakse kontroll rakenduse tasemel. Andmebaasid luuakse ühes SQL-andmebaasisüsteemis (PostgreSQL ver 14) ja ühes dokumendipõhises NoSQL süsteemis (MongoDB ver 5).

Töö tulemusena on realiseeritud erinevate disainidega andmebaasid ning rakendused, mille põhjal luuakse võrdlus, kus hinnatakse vastavaid andmebaase ja rakendusi erinevatest aspektidest. Lahendusi võrreldakse andmebaasi ja andmebaasirakenduste loomise lihtsuse, andmetele kehtivate piirangute jõustamise lihtsuse, veateadete headuse ja andmebaasi struktuuris tehtavate muudatuste sisseviimise lihtsuse alusel. Töö tulemusena valminud andmebaasi ja rakenduse lähtekood on avaldatud avalikul GitHubi lehel [1].

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 69 leheküljel, 7 peatükki, 40 joonist, 7 tabelit.

Abstract

The Influence of Using Schema-on-write and JSON Documents in the Database to the Creation of PostgreSQL and MongoDB Database Applications

Databases, where the data structure and constraints are defined and enforced at the database level, have schema-on-write. In window-on-data database applications, one uses graphical user interface to read and manage (create, update or delete) the data in the database. The goal of the thesis is to investigate how the use of schema-on-write and recording data as JSON documents influence the creation of window-on-data database applications. To fulfil the goal, an experiment is conducted. It means that databases with different designs and a web-based database application for each design are implemented. Requirements for the data (what data is needed and what rules the data must satisfy), functional and non-functional requirements of the application, and the user interface of the application are the same in all the cases. However, different database management systems (DBMSs) and physical database design solutions are used. Databases are created in one SQL DBMS (PostgreSQL ver 14) and in one document-based NoSQL DBMS (MongoDB ver 5).

The main differences in the designs are the usage of embedded (nested; hierarchical) and referenced (non-hierarchical) JSON documents, which also affects the data validation in the database. The database designs that are implemented in the databases are the following.

- "Traditional" PostgreSQL database without columns that have the JSONB type.
- PostgreSQL database where data is in columns with the JSONB type and:
    - the documents are hierarchical, i.e., follow the nested document model,
    - the documents are not hierarchical.
- MongoDB database where documents are checked against the schema and:

    - the documents are hierarchical, i.e., follow the embedded document model,
    - the documents are not hierarchical.

The database applications are meant for managing data about workers. The databases and database applications are not created for a specific organization. All the database applications have the same user interface. However, they have different back-ends due to the database that they must use. As many as possible data validations (checks) are implemented at the database level with the help of declarative constraints. Necessary checks that are impossible to implement in the database declaratively, are implemented in the application back-end. All the applications must return an error message to the user interface, when the data inserted by the user violates a validation rule. All the applications are implemented by using Java with Spring Boot framework and TypeScript with React library.

As a result of the work, databases with different design and database applications have been developed. Based on these a comparison has been made, where the respective databases and applications are evaluated from multiple viewpoints. The results of the subjective evaluations are justified by the results of further analysis, which are based on the development of the databases and applications. The source code of both the databases and the application, which was created as the result of the work, is published on a public GitHub page [1].

The main conclusions of the work are the following.

- Creating a database application was the simplest in case of the design without JSON documents, because all the data checks were implemented in the database.
- Schema changes are easier to implement in MongoDB databases.
- PostgreSQL has the capability to construct more complex validation rules and offer better error messages (based on user-friendliness).
- The usage of hierarchical and non-hierarchical JSON documents to record data changes the complexity of applications in different aspects.

The thesis is in Estonian and contains 69 pages of text, 7 chapters, 40 figures, 7 tables.

Lühendite ja mõistete sõnastik

BSON/JSONB	*Binary JSON.* Kahendkoodis talletatud JSON objekt.
CASE	*Computer Aided System Engineering.* Tarkvara, mis abistab tarkvara kui süsteemi loomist, võimaldades seda modelleerida ning mudelitest teiste mudelite, dokumentatsiooni, lähtekoodi ja teste genereerida ja samuti koodist mudelite genereerimist.
ETL	*Extract, transform, load*. Andmete "lahtilõhkumise", teisendamise ja andmebaasi laadimise protsess.
JSON	*JavaScript Object Notation.* Andmevahetus- ja andmeesitusvorming.
Kirjutamise skeem	*Schema-on-write.* Andmete struktuuri ja piiranguid määrav skeem jõustatakse andmebaasi tasemel.
Lugemise skeem	*Schema-on-read.* Rakendus peab teadma andmete struktuuri ja piiranguid, kuid andmebaasi tasemel seda ei jõustata.
Lõpp-punkt	*Endpoint*. Tagarakenduses defineeritud veebiaadress, läbi mille on võimalik kasutada tagarakenduse funktsionaalsust.
NoSQL	*No-to-SQL/Not Only SQL*. Üldnimi uuema põlvkonna (alates 2010. aastast) andmebaasisüsteemidele, milles on kasutusel mõni muu andmemudel/andmebaasikeel kui SQL aluseks olev andmemudel/SQL.
SQL	*Structurized Query Language.* Struktureeritud lauseid kasutada võimaldav andmebaasikeel, mida saab kasutada SQL-andmebaasisüsteemides andmete lugemiseks, muutmiseks, tehingute juhtimiseks, õiguste jagamiseks, andmebaasiobjektide haldamiseks ja muudeks andmebaasiga seotud toiminguteks.
UML	*Unified Modeling Language.* Üldotstarbeline modelleerimiskeel, mis võimaldab visualiseerida loodava/loodud süsteemi struktuuri ja käitumist. Selles keeles saab luua visuaalseid mudeleid e diagramme e skeeme.
URL	*Uniform Resource Locator*. Veebiaadress.
XML	*Extensible Markup Language.* Andmevahetus- ja andmeesitusvorming.

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

2010ndate alguses turule tulnud NoSQL süsteemide üks müügiargument SQL andmebaasisüsteemide ees oli see, et NoSQL süsteemid võimaldavad luua "skeemituid" andmekogusid [2]. Väidetavalt tagab see suurema paindlikkuse, aitab vältida üleliigseid (skeemi defineerimise) tseremooniaid ja rituaale ning lihtsustab rakenduste loomist. Andmetel peab olema struktuur, sest muidu neid kasutada ei saaks. Küsimus on, kes seda struktuuri teab ja jõustab. NoSQL "skeemitute" andmekogude puhul on tegemist lugemise skeemiga (*schema-on-read*), mis tähendab, et skeemi ei jõustata andmebaasi tasemel, kuid skeem on teada rakendusele ja mingil viisil selle lähtekoodi kaudu kirjeldatud. See on vastandiks SQL-andmebaasides loodavale kirjutamise skeemile (*schema-on-write*), mille korral on skeem andmebaasi tasemel kirjeldatud ja andmebaasisüsteem kontrollib iga andmemuudatuse tulemusena lisanduvate andmete vastavust sellele skeemile. Muuhulgas võimaldab see kohe andmete registreerimisel kontrollida andmete reeglitele vastavust, vältides vajadust teha seda rakenduse tasemel. Rakenduse tasemel tehtud kontrollid võivad olla ebausaldusväärsed (andmete mitme kasutaja poolt samaaegse kasutamise olukorras ei takista need ebasobivate andmete andmebaasi jõudmist) [3]. Rakendusest mööda minnes kontrollid ei rakendu. Kontrolle tuleb dubleerida erinevates rakendustes ja garantiide puudumine registreeritud andmete reeglitele vastamise kohta ei võimalda andmebaasisüsteemil päringute täitmist optimeerida. Näiteks välisvõtme kitsenduste olemasolust tingitud viidete terviklikkuse reeglite kehtimine SQL-andmebaasi andmetes võimaldab andmebaasisüsteemidel teha tabeli elimineerimise teisendust [4], mille tulemusel lihtsustab andmebaasisüsteem käivitatavat lauset nii, et lauses viidatud, kuid tulemuse jaoks mittevajaliku tabeli andmeid ei loeta.

2022. aasta veebruari seisuga on kõige populaarsem NoSQL andmebaasisüsteem MongoDB, mis on andmebaasisüsteemide populaarsuse indeksis viiendal kohal [5]. Tegemist on dokumendipõhise andmebaasisüsteemiga, kus andmebaasi põhiline ehitusplokk on JSON dokumentide kollektsioon. Samal ajal neli kõige populaarsemat andmebaasisüsteemi on SQL-andmebaasisüsteemid. SQL on täienenud võimalustega hoiustada ja töödelda JSON formaadis andmeid ja selline võimekus on näiteks väga hästi olemas populaarsuselt neljandas andmebaasisüsteemis – PostgreSQL [6], [7]. Teisalt on

NoSQL süsteemid, sh MongoDB, täienenud võimalustega defineerida kirjutamise skeemi [8]. Seega piirid nende süsteemide vahel on hägustunud ning NoSQL süsteemidel nagu MongoDB on olnud aega küpseda.

1.1 Töö eesmärk

Käesoleva töö eesmärk *ei ole* valida sobivaimat andmebaasisüsteemi konkreetse tarkvara jaoks ega võrrelda teatud SQL ja NoSQL andmebaasisüsteeme kõikvõimalikes erinevates aspektides. Käesolev töö lähtub eeldusest, et kirjutamise skeemi defineerimine on kasulik. Töö eesmärgiks on uurida kirjutamise skeemi defineerimise mõju andmebaasirakenduse loomisele erinevate PostgreSQL ja MongoDB andmebaasi disainide korral.

Mõju hinnatakse selle kaudu, kui lihtne on andmebaasi tasemel kontrollida andmete reeglitele vastavust ning kui lihtne on teha rakenduse funktsionaalsete nõuete muutmisest tulenevaid skeemimuudatusi, mille tulemusena tuleb omakorda muuta ka rakendust. Operatsioonide töökiiruse küsimusi antud töös ei puudutata, kuna paljud neid süsteeme võrdlevad tööd [9]–[11] just sellele keskenduvadki, pühendades muudele küsimustele vähem tähelepanu. Töö autor kasutas ka Google Scholar otsingusüsteemis otsingustringi “mongodb and postgresql", mille tulemusena oli 02.05.2022 seisuga 9490 vastust. Enamus artiklitest keskendus operatsioonide töökiiruste võrdlemisele, vaid üksikud tööd kirjeldasid vähesel määral rakenduste loomist ning andmete reeglitele vastavuse võrdlusi sisaldavaid artikleid töö autor ei leidnud. NoSQL süsteemide väidetavalt parem töökiirus suurte andmehulkadega ning paremad andmete salvestamise võimalused serverite kobarale on veel üks nende müügiargument, kuid see ei saa olla ainus argument, mille alusel andmebaasisüsteemi valikut teha.

1.2 Töö struktuur

Eesmärgi saavutamiseks luuakse samade analüüsi mudelite (nõuete) alusel järgmised andmebaasid. Iga sellise andmebaasi kohta esitatakse ka andmebaasi disaini kirjeldav füüsilise disaini mudel.

- "Traditsiooniline" PostgreSQL andmebaas, kus JSONB tüüpi veerge ei kasutata.
- PostgreSQL andmebaas, kus andmed on JSONB tüüpi veergudes ja seal olevad dokumendid on hierarhilised (*nested documents*).
- PostgreSQL andmebaas, kus andmed on JSONB tüüpi veergudes ja seal olevad dokumendid ei ole hierarhilised.
- MongoDB andmebaas, kus dokumendid on hierarhilised (*embedded documents*). Dokumente kontrollitakse skeemi suhtes.
- MongoDB andmebaas, kus dokumendid on seotud viidete kaudu. Dokumente kontrollitakse skeemi suhtes.

Kõigi disainide puhul üritatakse võimalikult palju nõuetest tulenevaid andmete kontrolle teha deklaratiivsel viisil andmebaasi tasemel. Deklaratiivne kontroll tähendab, et kontrolli läbiviivale süsteemile (antud juhul andmebaasisüsteemile) öeldakse, mida on vaja kontrollida, kuid ei öelda ette kuidas seda teha (ei anta ette kontrolli protseduuri). Kui andmebaasis pole see võimalik, siis tehakse kontroll rakenduses. Kõikidele andmebaasidele luuakse ühesuguse funktsionaalsusega andmete haldamiseks mõeldud andmebaasirakendus. Rakenduse loomiseks kasutatakse Java-t koos Spring Boot raamistikuga ja TypeScript-i koos React teegiga. Valitakse niisugused töövahendid, kuna need on levinud tarkvaraarendusega tegelevates ettevõtetes, Spring Boot-is on võimalik üpris hõlpsalt luua ühendus mõlema andmebaasisüsteemiga [12], [13] ning autoril on nimetatud kombinatsioonidega programmeerimises eelnevaid kogemusi.

Dokumendis tutvustatakse kõigepealt metoodikat ja esitatakse teoreetiline taust. Seejärel kirjeldatakse eksperimenti, sh nõudeid loodavale andmebaasile ja rakendusele ning rakenduse disaini ja erinevaid andmebaasi füüsilisi disaine. Eksperimendi sisuks on sellise rakenduse ja andmebaaside kavandamine ning realiseerimine. Ühtlasi tuuakse selles peatükis välja, millistes aspektides erinevaid andmebaasi disaine omavahel

võrreldakse. Järgnevas peatükis esitatakse eksperimendi tulemused, sh põhjendatud hinnangud andmebaasi disainidele erinevates aspektides. Seejärel analüüsitakse töö tulemusi. Lõpuks võetakse töö kokku ja nimetatakse tulevasi tegevusi, mida käesoleva töö edasiviimiseks võiks teha.

2 Metoodika

Selles peatükis antakse ülevaade töö tegemise metoodikast.

2.1 Tööprotsessi kirjeldus

Eesmärkide saavutamiseks viiakse läbi eksperiment. Eksperimendi sisuks on andmebaasi ja andmebaasirakenduse kavandamine ja realiseerimine ning seejärel tulemuste analüüsimine.

Töö käigus luuakse töötajate andmete haldamise rakendused, mille peamised eesmärgid on kasutajal (rollis “Juhataja”) hallata ettevõtte heaks töötavate töötajate andmeid ning töötajate ametites olemisi. Tegemist on näiterakendusega, mis ei tulene ühegi konkreetse ettevõtte vajadusest. Samas on funktsionaalsus ja salvestatavad andmed valitud piisavalt keerukad, et erinevate disainilahenduste tugevad ja nõrgad küljed saaksid välja tulla.

Rakenduse loomiseks tehakse esmalt kindlaks süsteemi funktsionaalsust esitavad kasutusmallid, mida rakendus peab kasutajale võimaldama. Kasutusmallide põhjal kirjeldatakse kontseptuaalne andmemudel (olemitüübid, atribuudid ja seosetüübid), millele vastavad andmed on püstitatud kasutusmallide täitmiseks vajalikud. Kui kontseptuaalse andmemudeli loomisel leitakse uusi kasutusmalle, siis lisatakse ka need lõplike kasutusmallide hulka. Peale kontseptuaalse andmemudeli valmimist luuakse andmebaasi füüsilise disaini täpsusega tehnilised kavandid erinevate andmebaasisüsteemide (PostgreSQL ning MongoDB) jaoks ja erinevaid disainilähenemisi järgides. Seejärel luuakse uuritavates andmebaasisüsteemides andmebaasid – iga disaini kohta üks. Iga andmebaasi põhjal luuakse andmebaasirakendus. Kõik andmebaasirakendused on täpselt ühesuguse funktsionaalsusega ja kasutajaliidesega.

Peale esialgset rakenduste loomist tehakse täiendusi süsteemi funktsionaalsuses (uued andmekontrollid, uute andmete registreerimise vajaduse lisamine), mille tulemusel tuleb teha muudatusi nii olemasolevates andmebaasides kui ka andmebaasirakendustes. Pärast muudatuste läbiviimist esitatakse hinnangud ning võrdlused iga loodud andmebaasi ja rakenduse kohta, kus kirjeldatakse erinevate disainide ühiseid ning erinevaid aspekte.

2.2 Tööriistade kirjeldus

Kontseptuaalse andmemudeli osaks olevad olemi-suhte skeemid ning andmebaasi füüsilise disaini mudelid PostgreSQL disainidele loodi kasutades CASE vahendit Rational Rose [14]. Rational Rose-is genereeriti ka PostgreSQL andmebaaside loomise skriptid, mis vajasid peale genereerimist vähesel määral parandamist. Rational Rose-is saab kasutada SQL-andmebaaside modelleerimiseks mõeldud UML-i mudeliprofiili. Nii olemi-suhte skeem kui ka andmebaasi füüsilise disaini mudel on loodud UML klassiskeemi baasil.

MongoDB andmebaaside füüsilise disaini mudelid ja andmebaaside loomise skriptid koos kirjutamise skeemiga tehti esmalt kasutades modelleerimisvahendit Moon Modeler [15], millel on MongoDB andmebaaside disainimiseks võimekam ning mugavam tugi kui Rational Rose-il. Moon Modeler-i puhul on kasutajal esimesed 14 päeva võimalik tasuta kasutada programmi täisversiooni, peale mida vähendatakse programmi funktsionaalsust, kui kasutaja ei soovi osta täisversiooni. Käesoleva töö raames polnud autoril vajadust täisversiooni soetada, sest minimaalse ja tasuta kasutatava programmi funktsionaalsus oli piisav. Lisaks loodi prooviks MongoDB andmebaasi disaini mudelid ka UML klassiskeemidena Rational Rose vahendit kasutades. Töö esitab seega näite ja juhised, kuidas sellist mudelit UML-is luua.

Andmebaasirakenduste loomiseks kasutati IntelliJ IDEA [16] programmi, mis peamiselt on mõeldud Java rakenduste arendamiseks, kuid mis toetab ka teiste programmeerimiskeelte kasutamist. IntelliJ IDEA on võimekas, paindlik ning kasutajasõbralik arendusvahend, mis on kasutajatele saadaval nii tasuta (*Community*) kui ka tasulises (*Ultimate*) versioonis. Tasuline versioon on suurema funktsionaalsusega kui tasuta kättesaadav versioon. Samas tasulise versiooni aastane litsents maksab eraisikule alates ligikaudu 180-st eurost, mis ei pruugi olla sobilik pakkumine ning sunnib leidma alternatiive. Tulenevalt asjaolust, et üliõpilastele on tasuline versioon kättesaadav tasuta, ei pidanud autor alternatiivseid arendusvahendeid otsima ning kasutas *Ultimate* versiooni, mille kasutamisega on autoril ka palju eelnevat kogemust.

PostgreSQL andmebaaside haldamiseks kasutas autor avatud lähtekoodiga ning tasuta kättesaadavat programmi pgAdmin [17]. MongoDB andmebaaside haldamiseks kasutas autor avatud lähtekoodiga ning tasuta kättesaadavat programmi MongoDB Compass [18].

Rakenduse loomiseks kasutati Java-t koos Spring Boot raamistikuga ja TypeScript-i koos React teegiga. Spring Boot [19] on populaarne, avatud lähtekoodiga Java raamistik, mis võimaldab lihtsalt ning kiirelt luua Java rakendusi tänu automaatsele konfiguratsioonide loomisele ning sõltuvuste haldamisele, mis vähendab rakenduse arendamisele kuluvat aega ning loomise keerukust.

TypeScript [20] on JavaScript-il põhinev programmeerimiskeel, mille peamine tugevus on määrata muutujatele tüüpe. Tüüpide määramine aitab luua töökindlamat rakendust, sest rakenduse kompileerimisel viiakse automaatselt läbi muutujate tüübikontroll, mis valideerib muutujate korrektset kasutust kogu rakenduses. Tüübikontroll võimaldab ka turvalisemat ning hallatavamat koodi muutmist ja refaktoreerimist.

React [21] on Facebook-i poolt loodud avatud lähtekoodiga teek, mis on mõeldud JavaScript-i abil kasutajaliideste loomiseks. Seda on võimalik kasutada ka TypeScript-iga. React teek võimaldab luua kasutajaliidest kiiresti ning lihtsalt, sest loodud komponente on võimalik hõlpsalt taaskasutada, kuvatavaid andmeid on võimalik kasutada mitmes komponendis korraga ning andmeid on võimalik uuendada ilma veebilehe uuesti laadimiseta.

Andmebaasisüsteemide valikul eelistati avatud lähtekoodiga ja populaarseid andmebaasisüsteeme. Dokumendipõhiseks andmebaasisüsteemiks valiti MongoDB, kuna see oli 2022. aasta veebruari seisuga kõige populaarsem NoSQL andmebaasisüsteem – andmebaasisüsteemide populaarsuse pingereas oli see siis viiendal kohal [5]. SQL-andmebaasisüsteemiks valiti PostgreSQL, mis 2022. aasta veebruari seisuga oli populaarsuselt neljas andmebaasisüsteem. SQL-andmebaasisüsteemi valikul eelistati PostgreSQL-i MySQL-ile (mõlemad on populaarsed ja avatud lähtekoodiga), sest PostgreSQL toetab rohkem SQL standardit [22], pakub rohkelt andmetüüpe [23] ning töö autoril on kogemusi PostgreSQL andmebaasisüsteemiga töötamisel.

MySQL (ver 8) toetab JSON andmetüüpi [24]. Sellesse kuuluvad väärtused salvestatakse kahendformaadis. PostgreSQL (ver 14) toetab eraldi JSON ja JSONB andmetüüpe, kusjuures JSON tüübi puhul salvestatakse JSON väärtus tekstina, kuid JSONB tüübi puhul kahendformaadis [6]. Andmete salvestamine kahendformaadis muudab küll salvestamise veidi aeglasemaks, kuid kiirendab hilisemat andmete lugemist. Nii MySQL-is [24] kui PostgreSQL-is [7] saab muuta JSON dokumendi võti-väärtus paare

osaliselt, st ilma kontseptuaalsel tasemel dokumenti täies mahus uue dokumendiga asendamata. MySQL-is on nimetatud osaline andmete muutmine piiratud, sest uus väärtus ei tohi olla andmemahult suurem kui asendatav väärtus ning osaline uuendamine ei võimalda uute võti-väärtus paaride salvestamist. Alates MySQL ver 8 on töökiiruse huvides nii [25], et sellisel juhul tehakse muudatusi olemasoleva dokumendi alamosas (ehk tehakse „tõeline“ osaline muutmine), mitte ei tehta koopiat olemasolevast dokumendist, milles tehakse andmemuudatused ning mis salvestatakse vana dokumendi asemele (ehk tehakse „näiline“ osaline muutmine). PostgreSQL kasutab multiversioon konkurentsjuhtimist ja seal tekitab rea andmete muutmine igal juhul süsteemis sisemiselt uue rea versiooni, seega tegemist on „näilise“ osalise muutmisega [26]. Samas ühe 2021. aasta võrdlustesti kohaselt [27] on PostgreSQL-is lugemis- ja kirjutamisoperatsioonid JSON andmetega ikkagi kiiremad kui MySQL-is.

Töös kasutati PostgreSQL versiooni 14.2 ja MongoDB versiooni 5.0.6.

3 Teoreetiline taust

Järgnevates alapeatükkides antakse ülevaade teoreetilisest materjalist, mille põhjal luuakse erinevates andmebaasisüsteemides erinevate disainidega andmebaasid.

3.1 Kirjutamise skeem

Kirjutamise skeemi (*schema-on-write*) korral tuleb enne andmebaasi andmete sisestamist defineerida skeem, mille suhtes kontrollitakse kõikide lisatavate andmete sobivust. Kirjutamise skeemi loomise näiteks on baastabelite e tabelite defineerimine SQL-andmebaasis. Andmed peavad täielikult vastama eelseadistatud skeemile – kui andmed pole sobiva struktuuriga, väärtused pole sobivat tüüpi või ei vasta need muudele seatud piirangutele e kitsendustele, siis muudetud andmeid ei salvestata. Selleks, et hakata salvestama andmeid, mida pole skeemi kirjeldamisel ette nähtud, peab enne muutma skeemi. Kirjutamise skeemiga andmebaasi kasutatakse jagatud andmebaasi (*integration database*) korral [28], kus erinevad rakendused või rakenduse osad suhtlevad omavahel (vahetavad infot) ühise andmebaasi kasutamise kaudu. Andmebaasis defineeritud kirjutamise skeem peab ühendama endas kõikvõimalike erinevate andmebaasi kasutajate soovid ja vajadused. Range andmete kontroll andmete salvestamisel tagab andmetele kehtestatud nõuete kehtivuse igas olukorras, kuid sellisest kontrollist tulenevalt on kirjutamise skeemil ka negatiivseid külgi [29]–[32].

Kirjutamise skeemi korral ei pruugi olla võimalik salvestada otse mingisugusest andmeallikast pärinevaid andmeid, ilma neid eelnevalt töötlemata. Kõigist andmeallikatest pärit andmed tuleb salvestamiseks viia skeemiga vastavusse. Kirjutamise skeemile vastavate andmete salvestamine on ressursikulukas, sest andmed on vaja enne salvestamist viia skeemile sobivale kujule.

See ei pruugi sobida suurandmete (*big data*) puhul, mille üheks tunnuseks on andmete struktuuri muutlikus (*variety*) [33]. See tähendab, et saabuvad andmed võivad sisaldada uusi andmevälju või mitte sisaldada skeemis nõutud andmeväärtuseid. Mida rohkem on andmebaasil erinevat tüüpi kasutajaid, seda keerulisem ja aeganõudvam on kõiki rahuldava skeemi väljatöötamine.

Paljud kirjutamise skeemis jõustatud piirangud tulevad süsteemi vajavale ärile kehtivatest reeglitest/piirangutest (ärireeglitest) ja nende kontroll andmebaasi tasemel realiseerib tegelikult ka olulise osa nende andmete haldamiseks mõeldud tarkvaralt oodatavast funktsionaalsusest.

3.2 Lugemise skeem

Kirjutamise skeemi nõrkuste vältimiseks on alternatiiviks lugemise skeemiga (*schema-on-read*) andmebaasi kasutamine. Lugemise skeemi korral ei jõustata andmebaasi skeemi andmebaasi tasemel andmebaasisüsteemi poolt. Küll aga peab andmete kasutaja (rakendus) teadma andmete struktuuri e skeemi, selleks, et osata andmebaasist loetud andmeid kasutada. Selline olukord on tüüpiline rakenduse andmebaasi korral (*application database*) [34], kus andmebaasist andmete lugemiseks ning muutmiseks on loodud vaid üks rakendus, mis võib lisaks ka võimaldada andmete kasutamist teistele rakendustele. Kui andmebaasi tasemel pole skeemi kirjeldatud, siis andmetest ja nende struktuurist saavad kõige paremini aru andmebaasi ja selle andmeid kasutava rakenduse loojad. Seega teised rakendused, mis ka neid andmeid vajavad, ei pöördu otse andmebaasi poole, vaid küsivad andmeid selle andmebaasi rakenduse vahendusel.

Lugemise skeemi korral saab andmeallikatest pärinevaid andmeid salvestada andmebaasi koheselt, ilma andmebaasi tasemel skeemi kirjeldamiseta ning andmete skeemile vastavaks teisendamiseta. Lugemise skeem võimaldab salvestada igal kujul andmeid, mis ei pruugi olla ühesuguse struktuuriga. Erinevate olemite e objektide puhul võib salvestatav informatsioon hõlmata erinevatele atribuutidele, seosetüüpidele või teistele olemitüüpidele vastavaid andmeid. Niisugune vabadus on tarvilik suurandmete salvestamiseks, kus esmatähtsad on andmete salvestamise paindlikkus ning kiirus, sest lugemise skeemi korral puudub andmete salvestamisel andmete "lahtilõhkumise", teisendamise ja laadimise protsessi (ETL – *extract, transform, load*) läbiviimine, mis on vajalik kirjutamise skeemiga andmebaasi korral.

Lugemise skeemi suurim tugevus (vabadus) on ka samas selle suurim nõrkus, sest see võimaldab salvestada mistahes vormis andmeid. Olemite ja seoste kohta säilitatavad andmed võivad olla puudulikud või ebakorrektsed, mis tähendab, et andmete kasutamine rakenduses vajab rohkem tähelepanu kui kirjutamise skeemi korral. Lugemise skeemi korral tuleks iga olemit käsitleda eraldi ning teha vajalikke operatsioone olemipõhiselt,

ehk ei saa eeldada, et kõikide olemite põhjal on võimalik toimida ühesuguselt, ilma olemi andmete valideerimiseta.

3.3 Dokumendipõhised NoSQL süsteemid

2010nda aasta paiku hakkasid populaarsust koguma NoSQL andmebaasisüsteemid. Tegemist on üldnimega suurele hulgale uue põlvkonna andmebaasisüsteemidele, mida ühendab see, et nendes ei kasutata SQL keelt ja SQL-i aluseks olevat andmemudelit. Selliste süsteemide tekkimisel oli peamiselt kaks põhjust. Esiteks vähenes andmete salvestamise maksumus, tänu millele ei pidanud enam nii palju mõtlema kulude vähendamiseks andmete dubleerimise vältimisele. Teiseks muutus tähtsamaks tarkvaraarendajate tööviljakus, mida NoSQL-i pooldajate arvates pärssis kirjutamise skeemi loomine ning uue informatsiooni talletamise vajaduse tekkides selle skeemi muutmine. Need põhjused olid eeldused NoSQL andmebaasisüsteemide tekkeks, mis võimaldavad uute andmete salvestamist ning kasutamist paindlikumalt, kui see on võimalik SQL-andmebaasisüsteemides [35]–[37].

Üks enimlevinud NoSQL andmebaasisüsteemide alamtüüp on dokumendipõhised andmebaasisüsteemid. Dokumendis talletatakse iga olemi andmeid võti-väärtus paaridena, kus väärtus võib näiteks olla sõne, arv, kuupäev, massiiv või teine objekt. Dokumente salvestatakse enamasti JSON (*JavaScript Object Notation*) formaadis tekstina või kahendformaadis BSON (*Binary JSON*). Turul on ka andmebaasisüsteeme, mis võimaldavad hallata XML (*Extensible Markup Language*) formaadis dokumente. Ka sellised andmebaasisüsteemid ei kasuta SQL-i, kuid kuna need olid turul juba enne 2010. aastat, siis neid NoSQL süsteemide hulka tavaliselt ei loeta. Dokumendis saab andmevälju lisada, muuta või kustutada ilma igasuguse eelneva skeemi defineerimiseta või muutmiseta. Taoline tegutsemisvabadus toobki esile põhilise NoSQL tugevuse, milleks on paindlikkus.

3.3.1 MongoDB

MongoDB on 2007. aastal loodud avatud lähtekoodiga dokumendipõhine NoSQL andmebaasisüsteem [38]–[40]. Andmeid talletatakse MongoDB-s BSON formaadis ja andmed esitatakse kasutajatele JSON objektidena [41]. Kahendformaat on kasutusel, et kiirendada andmete lugemist ning töötlust.

Dokumente hoitakse kollektsioonides (*collections*), mida võib võrrelda SQL andmebaasi tabeliga, kus hoitakse võrreldavaid või sarnase kasutuseesmärgiga dokumente. Erinevalt aga SQL-tabelist, võivad ühes kollektsioonis olevad dokumendid sisaldada erinevaid andmevälju, ehk välju võib dokumendis eemaldada ja lisada vastavalt igale eraldiseisvale olemile või seosele, mida üks dokument kirjeldab.

MongoDB üks suurem erinevus SQL-andmebaasisüsteemidest seisneb skaleerimise võimalustes. SQL-andmebaasisüsteemid võimaldavad tavaliselt vertikaalset skaleerimist, ehk parema jõudluse tagamiseks saab tõsta olemasoleva arvuti arvutusvõimsust, mida saab teha näiteks kiirema protsessori (*Central Processing Unit/CPU*), kiirema ning rohkema muutmälu (*Random Access Memory/RAM*) või kiirema ning rohkema püsimälu (*Hard Disk Drive/HDD, Solid State Drive/SSD*) paigaldamisega olemasolevasse masinasse. MongoDB toetab aga ka horisontaalset skaleerimist, ehk jõudlust saab parandada taristusse uute masinate lisamisega, mis võib olla odavam ja pakkuda paremat töökindlust kui vertikaalne skaleerimine [42].

3.3.2 Kirjutamise skeemi defineerimine MongoDB-s

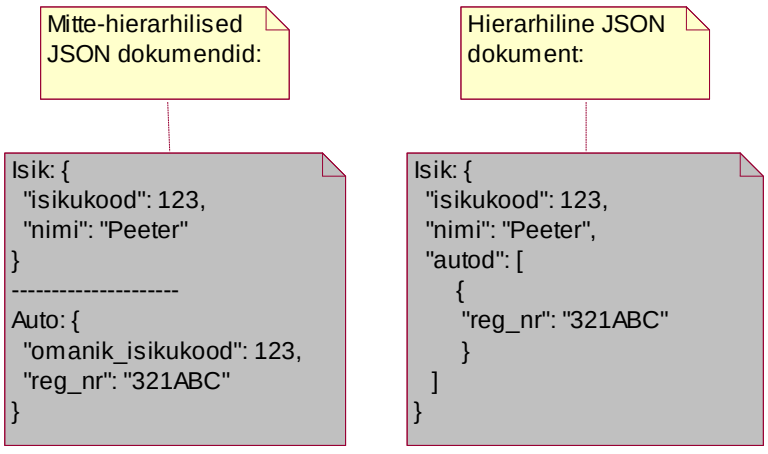
Alates MongoDB versioonist 3.6 [8] on võimalik defineerida dokumentidele kirjutamise skeemi. Kirjutamise skeemi defineerimisel lähtutakse *JSON Schema* standardist [43]. Skeemi abil saab määrata, millised on konkreetse süsteemi jaoks korrektseks peetavad JSON objektid. Skeemis saab näiteks kontrollida välja olemasolu, väärtuse kuulumist andmetüüpi või väärtuse vastamist mingile täiendavale reeglile. Kirjutamise skeemi jõustamisel või muutmisel ei muudeta olemasolevaid dokumente, mis tähendab, et skeemi võib muuta teadmisega, et uue skeemi järgi mittevalideeruvad andmed ei lähe kaotsi. Küll aga peavad kõik andmemuudatused peale skeemi jõustamist olema skeemiga kooskõlas. Ebaõnnestunud validatsiooni korral tagastatakse veateade selle kohta, mis ebaõnnestus andmeid skeemi järgi kontrollides. Veateates ei kirjeldata mitte ainult esimest ebaõnnestumisega lõppenud kontrolli, vaid kõiki ebaõnnestunud kontrolle. See erineb näiteks PostgreSQL andmebaasis kitsenduste kontrollimisest, kus andmete kontrollimine lõpetatakse ning tagastatakse veateade niipea, kui tuvastatakse andmete vastuolu mõne andmebaasis defineeritud kitsendusega.

3.4 JSON-i tugi SQL standardis

SQL:2016 standardisse [44] lisati tugi JSON andmetele (SQL/JSON). Standard ei näe ette uue SQL andmetüübi (JSON) kasutuselevõttu. JSON dokumente esitatakse mõnda olemasolevat andmetüüpi (VARCHAR – *Variable length strings*, CLOB – *Character Large Object* või BLOB – kahendformaadis *CLOB*) väärtustena. Puudub JSON tüüpi andmete põhjal päringute tegemiseks mõeldud standardiseeritud keel, küll aga on olemas võimalused JSON-is navigeerimiseks ja päringute tegemiseks JSON dokumendi andmete põhjal. On tehtud ka ettepanek JSON andmetüübi kirjeldamiseks SQL standardis [45]. Selles ettepanekus pakuti JSON andmete säilitamiseks uue andmetüübi kasutuselevõttu efektiivsema operatsioonide läbiviimise ja uute operaatorite kasutamise võimaldamiseks. See tähendaks, et JSON tüüpi andmete esitamiseks ei kasutataks enam sõne andmetüüpe. Tehti ka ettepanek JSON-i võti-väärtus paaride otsese muutmise võimaldamiseks *UPDATE* lausega, et vältida kogu dokumendi ülekirjutamise vajadust juhul, kui tegelikult on vaja dokumendis muuta näiteks vaid ühe võti-väärtus paari väärtust.

3.5 JSON dokumentide ülesehitamise viisid

Käesolevas töös on kasutusel kaks JSON dokumentide struktuuri: hierarhilised ning mitte-hierarhilised dokumendid. Mitte-hierarhiliste dokumentide korral on võti-väärtus paaril väärtuseks mõni JSON-i toetatud andmetüüpi väärtus (nt sõne või arv). Hierarhiliste dokumentide korral võib võti-väärtus paari korral olla väärtuseks põhimõtteliselt eraldiseisev JSON dokument, mis omakorda koosneb võti-väärtus paaridest. Joonisel 1 on visuaalselt kujutatud nende struktuuride erinevust.



Joonis 1.Erinevate JSON dokumentide struktuuride näited.

3.6 PostgreSQL võimalused JSON dokumentide haldamiseks

Alates PostgreSQL versioonist 9.2 on võimalik kasutada tabelites JSON andmetüübiga veerge ja seega salvestada andmebaasis JSON formaadis andmeid [6]. JSON tüüpi väärtuse salvestamisel salvestab süsteem täpse sisendteksti, mida on tarvis iga operatsiooni korral läbi töödelda ning mis võib sisaldada semantiliselt ebaolulisi tühimärke ja korduvaid võti-väärtus paare. JSON tüüpi väärtuste puhul ei muudeta salvestamisel võtmete järjekorda. Andmete salvestamisel kontrollitakse, et salvestatav sisu oleks korrektses JSON formaadis.

PostgreSQL versiooniga 9.4 lisandus ka võimalus kasutada JSONB andmetüüpi, mis on PostgreSQL JSON andmetüübi edasiarendus. JSONB tüübi korral talletakse informatsioon destrukturiseeritud kahendformaadis, mis teeb küll andmete salvestamise eelnimetatud formaati teisendamise pärast aeglasemaks, kuid edasised operatsioonid väärtustega on kiiremad, sest andmeid ei tule uuesti sõeluda. JSONB tüüpi väärtuse korral ei säilitata tühimärke, võtmete järjekorda ega korduvaid võti-väärtus paare. Kui andmete sisestamisel leidub mitu sama võtmega võti-väärtus paari, siis salvestatakse vaid viimane paar. JSONB andmetüüp toetab ka indekseerimist.

Üldine soovitus on kasutada JSONB andmetüüpi, kuna see on efektiivsem kui JSON andmetüüp [6]. JSON-i eelistamine JSONB-le võib näiteks olla õigustatud juhul, kui on rangelt tarvis säilitada objektis salvestatud võtmete järjekorda, mida JSONB kasutamine võib omavoliliselt muuta.

3.6.1 JSON dokumentide kontrollimine skeemi suhtes PostgreSQL-is

PostgreSQL-i pole sisseehitatud JSON (ega ka JSONB) andmetüüpi väärtuste (objektide, dokumentide) skeemi suhtes kontrollimise mehhanismi, kuid JSON dokumente on siiski võimalik skeemi suhtes kontrollida. Üks variant selleks on kasutada PostgreSQL-i andmebaasi salvestatud JSON dokumendist eraldatud väärtuste teisendamist (*cast*) PostgreSQL-ile sobivat tüüpi väärtuseks ning rakendada saadud väärtusele *CHECK* kitsendust (vt Joonis 2).

```
CONSTRAINT Person_e_mail_contains_atleast_one_at_sign CHECK
((CAST (data ->> 'e_mail' AS VARCHAR)) LIKE '%@%')
```

Joonis 2. PostgreSQL-is JSON väärtuse teisendamise ning kontrollimise näide.

Niisuguse valideerimise puhul võib valukohaks olla väärtuste teisendamine, ehk väärtus, mida soovitakse teisendada, peab olema dokumendis õige süntaksiga esitatud. Näiteks saadab PostgreSQL JSON dokumentidega töötavad tagarakendused andmebaasi isiku registreerimise aja kujul *[2022,4,2,21,44,58,142743800]*, mida andmebaasisüsteem ei suuda teisendada *Timestamp* andmetüüpi väärtuseks registreerimise aja kontrolli sooritamiseks.

Teine võimalus JSON-i kontrollimiseks oleks kasutada avatud lähtekoodiga, vabalt saadavaid PostgreSQL laiendusi [46], [47], mis järgivad samamoodi *JSON Schema* standardit nagu MongoDB-s jõustatav kirjutamise skeem. Antud töös ei kasutatud laiendusi, kuna erinevate disainide õiglaseks võrdlemiseks otsustati mõlema andmebaasisüsteemi puhul kasutada vaid sisseehitatud võimekusi. Autor otsustas taustateadmiste saamiseks siiski proovida ka nende laienduste kasutamist, kuid ei saanud neid Windows-i platvormil töötades tööle, sest mõlemad laiendused on mõeldud installeerimiseks Linux operatsioonisüsteemil ning autoril ei õnnestunud neid installeerida kasutades Linux-i terminali Windows-ile. Samuti ei õnnestunud saada kontakti laienduste autoritega juhiste küsimiseks.

4 Eksperimendi kirjeldus

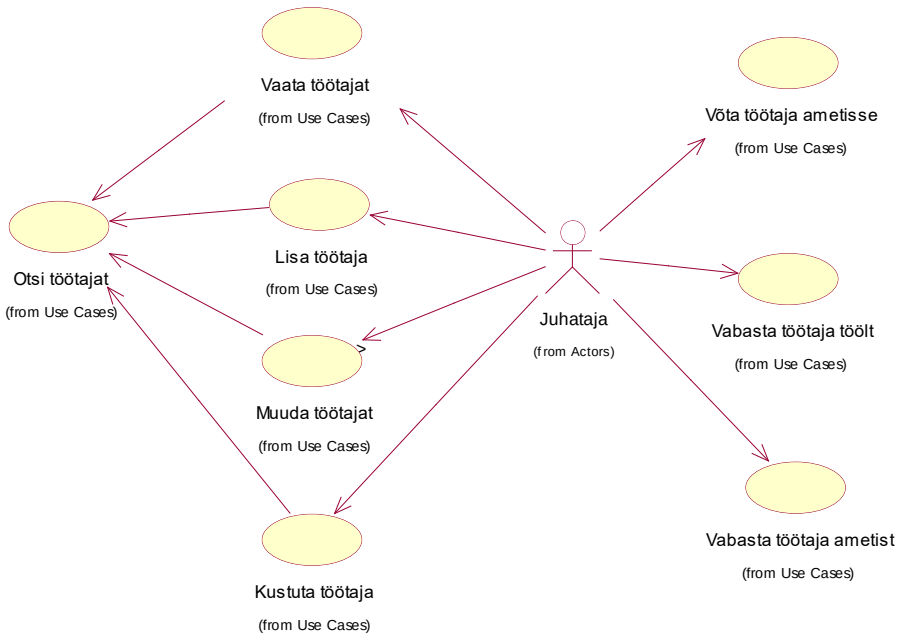
Eksperimendi sisu ning läbiviimist on kirjeldatud jaotises 2.1.

4.1 Funktsionaalsed nõuded

Järgnevates jaotistes esitatakse funktsionaalsed nõuded, mida loodavad andmebaasid ning andmete haldamise rakendused peavad realiseerima. Loodav tarkvara ei ole mõeldud rahuldama ühegi konkreetse ettevõtte vajadusi. Samas on kasutusmallid ja nõuded andmetele valitud piisavalt mitmekesised, et oleks võimalik konkreetse rakenduse ning andmebaasi loomise abil teha kindlaks kirjutamise skeemi ja JSON dokumentide kasutamise mõju rakenduse loomisele.

4.1.1 Töötajate ja töötamiste haldus

Joonisel 3 esitatakse kasutusmallid, mis on seotud töötajate ning töötamiste andmete haldamisega. Tabelis 1 esitatakse nende kasutusmallide sõnalised kirjeldused lühiformaadis.



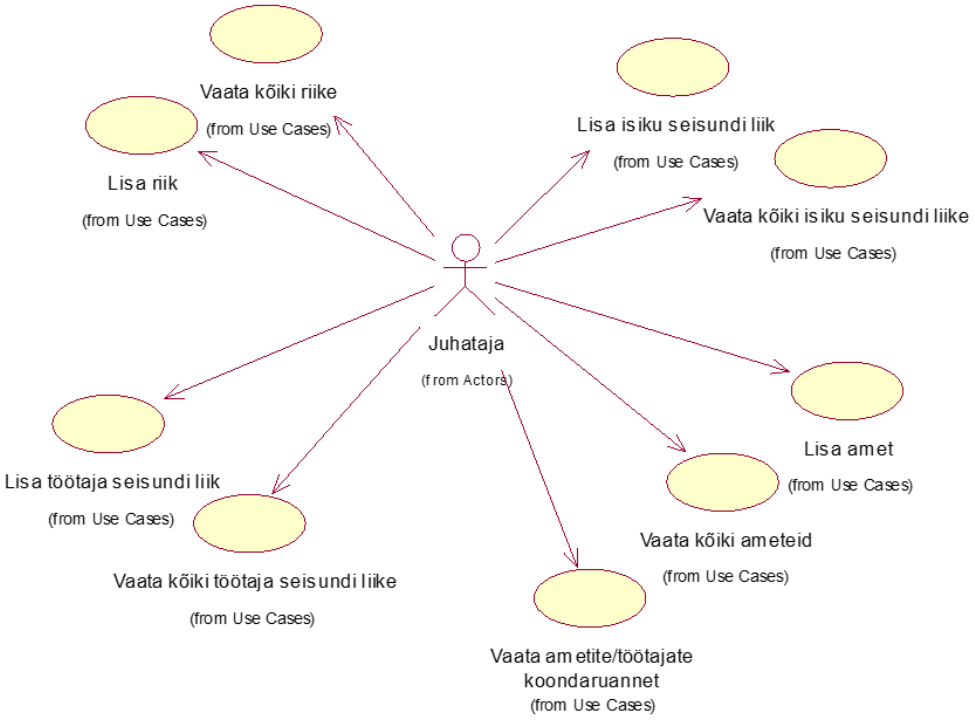
Joonis 3. Töötajate ning töötamiste andmete haldamise kasutusmallid.

Tabel 1. Töötajate ning töötamiste haldamise kasutusmallide kirjeldused.

Kasutusmall	**Kirjeldus**
Otsi töötajat	Juhataja saab vaadata töötajate nimekirja. Juhataja saab nimekirja filtreerida. Samuti saab ta iga töötaja korral vaadata tema kõiki detailseid andmeid, sh hetkeseisund ja registreerimise aeg.
Vaata töötajat	Juhataja vaatab töötajate nimekirja, valib sealt töötaja ning näeb töötaja detailseid andmeid, mille hulgas on ka ametites töötamised ning isikuandmed.
Lisa töötaja	Juhataja registreerib uue töötaja.
Muuda töötajat	Juhataja vaatab töötajate nimekirja, valib sealt töötaja ja muudab tema andmeid. Ei ole võimalik muuta töötaja registreerimise aega, kuid saab lisada ja lõpetada ametites töötamisi.
Kustuta töötaja	Juhataja vaatab töötajate nimekirja, valib sealt töötaja ja kustutab selle andmebaasist. Juhataja saab nimekirja filtreerida. Kustutatav töötaja ei tohi kustutamise hetkel töötada üheski ametis, muidu tema andmeid kustutada ei saa. Töötaja andmete kustutamisel kustutatakse ka kõik tema töötamiste andmed, samas kui isikuandmed säilitakse süsteemis.
Võta töötaja ametisse	Juhataja registreerib töötaja uue ametis töötamise. Uut ametis töötamist ei saa registreerida, kui töötajal leidub samas ametis töötamine, millel pole määratud lõpu aega. Samuti ei saa registreerida töötajale ametis töötamist, kui tal juba leidub samas ametis olemine sama alguse ajaga.
Vabasta töötaja ametist	Juhataja soovib vabastada töötajat ametist, lisades töötaja ametis töötamisele lõpu aja. Lõpu aeg peab olema suurem kui ametis töötamise alguse aeg.
Vabasta töötaja töölt	Juhataja soovib registreerida töötaja sellisesse seisundisse, kus töötajal pole enam võimalik töökohustusi täita. Sellisel juhul vabastatakse töötaja ka kõikidest ametitest, milles töötaja töökohustusi täidab, märkides töötaja ametis töötamisele lõpu aja.

4.1.2 Klassifikaatorite haldus

Joonisel 4 esitatakse kasutusmallid, mis on seotud klassifikaatorite andmete haldamisega. Tabelis 2 esitatakse nende kasutusmallide sõnalised kirjeldused lühiformaadis.



Joonis 4. Klassifikaatorite haldamise kasutusmallid.

Tabel 2. Klassifikaatorite haldamise kasutusmallide kirjeldused.

Kasutusmall	**Kirjeldus**
Lisa riik	Juhataja registreerib uue riigi.
Vaata kõiki riike	Juhataja näeb kõiki süsteemis registreeritud riike.
Lisa isiku seisundi liik	Juhataja registreerib uue isiku seisundi liigi.
Vaata kõiki isiku seisundi liike	Juhataja näeb kõiki süsteemis registreeritud isiku seisundi liike.
Lisa amet	Juhataja registreerib uue ameti.
Vaata kõiki ameteid	Juhataja näeb kõiki süsteemis registreeritud ameteid.
Vaata ametite/töötajate koondaruannet	Juhataja näeb iga ameti kohta selle koodi, nimetust ja seda ametit hetkel pidavate töötajate arvu. Kui ametiga pole hetkel seotud ühtegi töötajat, siis on see arv 0.
Lisa töötaja seisundi liik	Juhataja registreerib uue töötaja seisundi liigi.

Kasutusmall	Kirjeldus
Vaata kõiki töötaja seisundi liike	Juhataja näeb kõiki süsteemis registreeritud töötaja seisundi liike.

4.2 Mittefunktsionaalsed nõuded

Tabelis 3 esitatakse mittefunktsionaalsed nõuded tüüpide kaupa selgitava kirjeldusena.

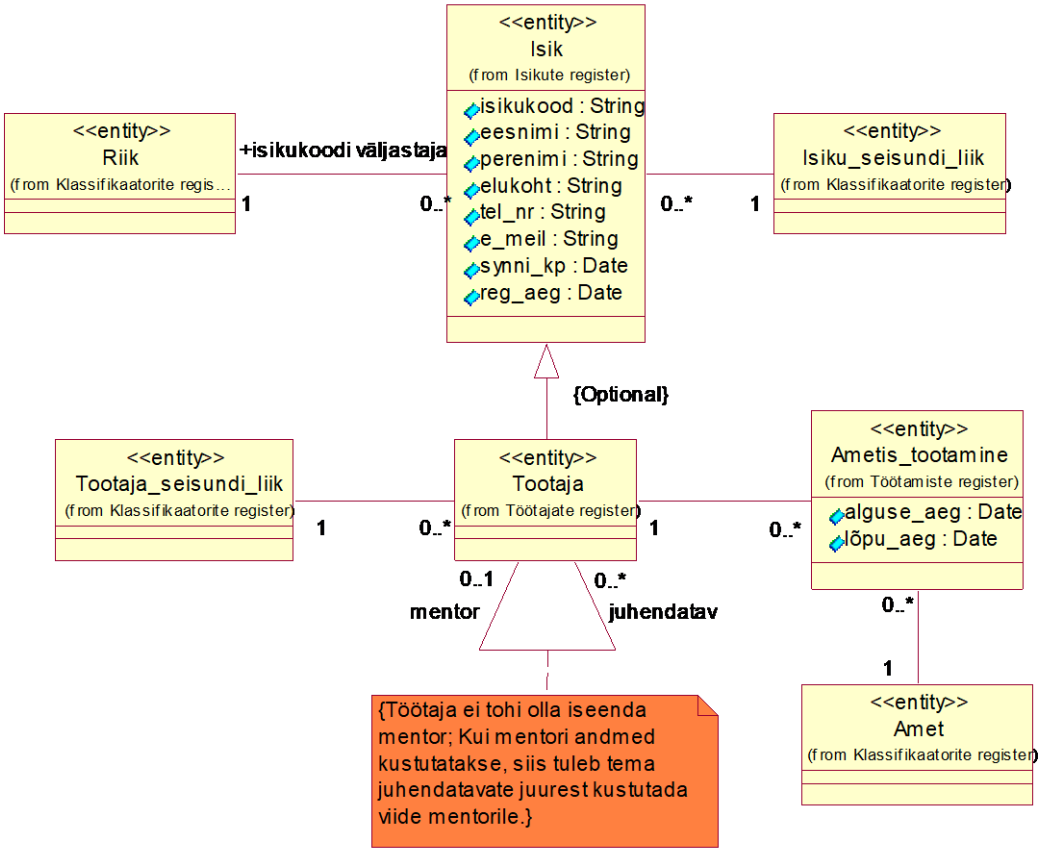
Tabel 3. Mittefunktsionaalsed nõuded

Tüüp	Nõude kirjeldus
Andmebaasi-süsteem	Süsteem peab andmete hoidmiseks kasutama SQL või NoSQL andmebaasisüsteemi abil loodud andmebaasi. Andmebaasisüsteemidena on soovitav kasutada PostgreSQL-i või MongoDB-d, kuna need on avatud lähtekoodiga, neid pakutakse tasuta, need pakuvad häid võimalusi andmebaasi programmeerijale ning nendele on suur kasutajate kogukond. Teiste sõnadega on veebis saadaval palju lisamaterjale probleemide ja nende lahenduste kohta.
Arendus-vahendid	Arendusvahenditena tuleks kasutada CASE tarkvara Rational Rose ning modelleerimisvahendit Moon Modeler. Andmebaasirakendus on kolmekihiline, kus kasutajal on läbi veebibrauseri ligipääs kasutajaliidesele, mis suhtleb tagarakendusega, millel on ühendus serveris paikneva andmebaasiga. Käesolevas töös võib kasutada lokaalset serverit.
Keel	Süsteemi kasutajaliides, lähtekoodis kasutatavad nimed ja andmebaasi andmestruktuurides kasutatavad nimed peavad olema inglise keeles.
Kasutajaliides	Nõuded kasutajaliidese ülesehitusele. • Rakenduses peab olema tööriistariba, kust saab kasutusmallidega määratud tegevuste juurde liikuda. • Välisvõtme väärtuste registreerimiseks tuleb kasutada ripploendit e liitboksi. • Kohustuslikud sisestusväljad tuleb tähistada (nt lisades lipikule *). • Kuupäevad tuleb esitada formaadis DD/MM/YYYY • Ajatemplid tuleb esitada formaadis YYYY-MM-DD HH24:MI:SS:MS • Tegevused, mida süsteem saab ise teha (nt kindlaks tegema, millal andmed registreeriti), peab tegema süsteem ilma kasutajalt tagasiside küsimisega tülitamata. • Kõikides nimekirjades, kus tuuakse välja olemite andmed, tuleb kasutajale näidata sellist hulka andmeid, et nende alusel saaks olemeid üksteisest üheselt eristada. Samas peavad esitatavad andmed olema kasutaja jaoks arusaadavad ja sisukad. • Andmete sisestamiseks ja vaatamiseks mõeldud väljade juures peab olema võimalikult arusaadavalt ja täielikult välja toodud nende andmete tähendus.

Tüüp	Nõude kirjeldus
	• Kasutajale esitatavad andmed peavad olema filtreeritavad viisil, mis võimaldab tal vajalikud andmed lihtsalt üles leida. • Vigade ilmnemisel kuvatakse kasutajale viga/vigu selgitav veateade

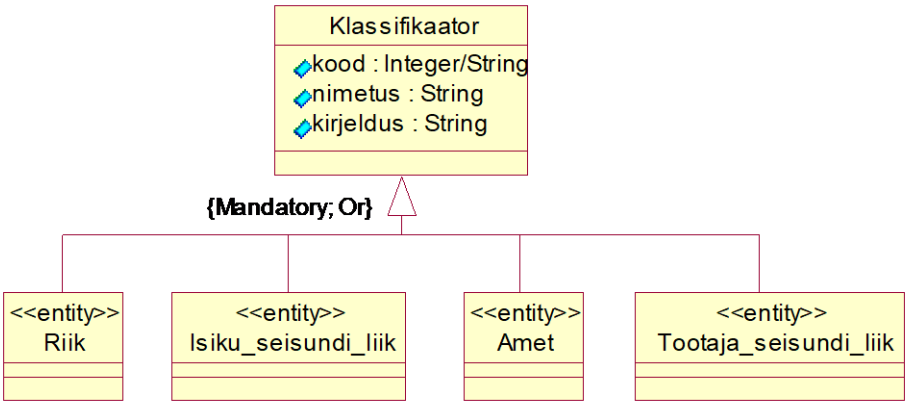
4.3 Nõuded andmebaasile

Joonistel 5 ja 6 kujutatakse andmebaasi olemi-suhte skeeme. Need kirjeldavad olemitüüpe, seosetüüpe ja atribuute, millele vastavaid andmeid peab andmebaasis säilitama. Koos järgnevates jaotistes esitatud olemitüüpide ja atribuutide sõnaliste kirjeldustega moodustab see kontseptuaalse andmemudeli. Kuna realisatsioonis kasutatakse ingliskeelseid nimesid, siis esitatakse olemitüüpide ja atribuutide nimed nii eesti kui ka inglise keeles.



Joonis 5. Olemi-suhte skeem (isikud, töötajad ja töötamised).

Joonisel 6 kujutatakse klassifikaatoreid.



Joonis 6. Olemi-suhte skeem (klassifikaatorid).

4.3.1 Olemitüüpide definitsioonid

Tabelis 4 esitatakse olemitüüpide definitsioonid ning kuuluvused registritesse.

Tabel 4. Kohustuslike olemitüüpide definitsioonid.

Olemitüübi nimi (*nimi inglise keeles*)	**Kuuluvus registrisse**	**Definitsioon**
Amet (*Occupation*)	Klassifikaatorite register	Amet on töökohustuste kogumi üldnimetus. Ametid on klassifikaatorid. Võimalike väärtuste näited on juhataja ja koristaja.
Ametis töötamine (*Employment*)	Ametis töötamise register	Töötaja töötab ametis mingi ajaperioodi jooksul. See on sätestatud töölepingus. Võimaldab säilitada töötaja ametis töötamiste ajaloo.
Isik (*Person*)	Isikute register	Mistahes organisatsiooniga seotud füüsiline isik (eraisik). Isik võib olla seotud organisatsiooniga näiteks kui klient või töötaja.
Isiku seisundi liik (*Person status type*)	Klassifikaatorite register	Võimaldab kirjeldada isiku seisundit, mis on oluline infosüsteemi jaoks. Võimalike väärtuste näited on elus ja surnud.
Klassifikaator	Klassifikaatorite register	Klassifikaatorid on „mistahes andmed, mida kasutatakse andmebaasis teiste andmete liigitamiseks või andmebaasis olevate andmete seostamiseks väljaspool organisatsiooni vastutusala oleva informatsiooniga“ [48].
Riik (*Country*)	Klassifikaatorite register	Riik on „kindlal territooriumil suveräänset võimu teostav poliitiline organisatsioon” [49]. Riigi andmeid

Olemitüübi nimi (*nimi inglise keeles*)	**Kuuluvus registrisse**	**Definitsioon**
		kasutatakse selleks, et identifitseerida isiku isikukoodi väljastajat.
Töötaja (*Employee*)	Töötajate register	Isik, kes on palgatud töökohale organisatsiooni poolt ning rakendab oma aega ja oskusi organisatsiooni toimise tarbeks. Töötaja võib samaaegselt täita mitme ameti töökohustusi.
Töötaja seisundi liik (*Employee status type*)	Klassifikaatorite register	Võimaldab kirjeldada töötaja töökohustuste täitmise võimalikkust. Võimalike väärtuste näited on katseajal ja töösuhe lõpetatud.

4.3.2 Atribuutide definitsioonid

Tabelis 5 kirjeldatakse olemitüüpide atribuute.

Tabel 5. Kohustuslike olemitüüpide atribuutide definitsioonid.

Olemitüübi nimi	**Atribuudi nimi (*nimi inglise keeles*)**	**Atribuudi definitsioon**	**Näiteväärtus**
Ametis_tootamine	alguse_aeg (*start_time*)	Töötaja ametisse asumise aeg kohaliku aja järgi – kuupäev ja kellaaeg sekundi täpsusega, ilma ajavööndi ning sekundi murdosadeta. Selle võib süsteem ise automaatselt määrata. **{Registreerimine on kohustuslik. Väärtus peab olema vahemikus 01. jaanuar 2010 00:00:00 ja 31. detsember 2100 kell 23:59:59 (otspunktid kaasa arvatud).}**	12.08.2014 17:01:05
Ametis_tootamine	lopu_aeg (*end_time*)	Töötaja ametist vabastamise aeg kohaliku aja järgi – kuupäev ja kellaaeg sekundi täpsusega, ilma ajavööndi ning sekundi murdosadeta. Kui lõpu aeg puudub, siis järelikult kestab töötamine	13.09.2015 12:05:05

Olemitüübi nimi	Atribuudi nimi (*nimi inglise keeles*)	Atribuudi definitsioon	Näiteväärtus
		piiramatu aja, st lõppaega pole paika pandud. **{Lõpu aja võimalikud väärtused on vahemikus 01. jaanuar 2010 ja 31. detsember 2100 (otspunktid kaasa arvatud). Ametis töötamise lõpu aeg ei tohi olla väiksem ametis töötamise alguse ajast.}**	
Isik	isikukood (*nat_id_code*)	Riigi poolt väljastatud isiku identifikaator, mis on unikaalne selle väljastanud riigi piires. **{Registreerimine on kohustuslik. Koos riigi identifikaatoriga on isiku unikaalne identifikaator.** **Isikukoodis on lubatud tähed (lubatud on ka muud tähed kui ASCII tähed a-zA-Z), numbrid, tühikud, sidekriipsud, plussmärgid, võrdusmärgid ja kaldkriipsud.** **Isikukood ei tohi olla tühi string ja ainult tühimärkidest koosnev string. Isikukood võib olla kuni 50 märki pikk.}**	39204010231
Isik	synni_kp (*birth_date*)	Isiku sünni kuupäev sünnikoha kohaliku aja järgi. **{Registreerimine on kohustuslik. Sünni kuupäeva võimalikud väärtused on vahemikus 01. jaanuar 1900 ja 31. detsember 2100 (otspunktid kaasa arvatud). Sünni kuupäev ei tohi olla suurem isiku registreerimise ajast.}**	12.08.1993

Olemitüübi nimi	Atribuudi nimi (*nimi inglise keeles*)	Atribuudi definitsioon	Näiteväärtus
Isik	reg_aeg (*reg_time*)	Isiku registreerimise kuupäev ja kellaaeg kohaliku aja järgi sekundi täpsusega, ilma ajavööndi ning sekundi murdosadeta. Selle võib süsteem ise automaatselt määrata. **{Registreerimine on kohustuslik. Väärtus peab olema vahemikus 01. jaanuar 2010 00:00:00 ja 31. detsember 2100 kell 23:59:59 (otspunktid kaasa arvatud).}**	12.08.2014 17:01:05
Isik	eesnimi (*given_name*)	Täielik eesnimi, mis võib olla isikut tõendavasse dokumenti kantud eesnimest pikem. **{ Vähemalt üks kahest – eesnimi või perenimi peab olema registreeritud. Eesnimi ei tohi olla tühi string ja ainult tühimärkidest koosnev string. Eesnimi võib olla kuni 1000 märki pikk.}**	Mart
Isik	perenimi (*surname*)	Täielik perenimi, mis võib olla isikut tõendavasse dokumenti kantud perenimest pikem. **{Vähemalt üks kahest – eesnimi või perenimi peab olema registreeritud. Perenimi ei tohi olla tühi string ja ainult tühimärkidest koosnev string. Perenimi võib olla kuni 1000 märki pikk.}**	Mets
Isik	elukoht (*address*)	Isiku alalise elukoha aadress. **{Elukoht ei tohi olla tühi string, ainult tühimärkidest koosnev string ja ainult numbritest koosnev string.**	Tallinn, Pikk tn. 12

Olemitüübi nimi	Atribuudi nimi (*nimi inglise keeles*)	Atribuudi definitsioon	Näiteväärtus
		Elukoht võib olla kuni 1000 märki pikk.}	
Isik	e_meil (*e_mail*)	Aadress, millele saab üle võrgu (ühest arvutist või tööjaamast teise) saata isikule mõeldud kirjalikke sõnumeid. Võimaldab isiku identifitseerimist süsteemi sisenemisel. **{Registreerimine on kohustuslik. Isiku tõstutundetu unikaalne identifikaator. Teiste sõnadega, kui süsteemis on näiteks registreeritud isik meiliaadressiga MatiAndres@metstamm.ee, siis teisele isikule meiliaadressi matiandres@metstamm.ee lisada ei saa.** **E_meil peab sisaldama vähemalt ühte "@" märki. Meiliaadress võib olla kuni 254 märki pikk.}**	matimets@meil.com
Isik	tel_nr (*tel_nr*)	Telefoninumber, mida kasutades on võimalik ühendust võtta helistades või sõnumeid saates. **{Tel_nr võib sisaldada numbreid tühimärke, "+" ja "-" märke. Telefoninumber võib olla 7–20 märki pikk (otspunktid kaasa arvatud). Telefoninumber ei tohi olla tühi string ja ainult tühimärkidest koosnev string.}**	+372 5123 9871
Klassifikaator	kood (*code*)	Klassifikaatori väärtust esitav märkide jada. Seda kasutatakse klassifikaatori	1; EST

Olemitüübi nimi	Atribuudi nimi (*nimi inglise keeles*)	Atribuudi definitsioon	Näiteväärtus
		väärtusele lühidalt viitamiseks. Kood võib olla tekstiline või arvuline väärtus (nõude täpsustused on kirjas atribuudi kitsenduste juures). Kood peaks olema võimalikult hästi meeldejääv, mis tähendab, et kui kasutaja näeb koodi, siis seostub see talle võimalikult kergesti koodi poolt iseloomustatava klassifikaatori väärtusega. **{Klassifikaatori unikaalne identifikaator. See on klassifikaatori tüübi piires unikaalne. Registreerimine on kohustuslik.** **Riikide koodid koosnevad vastavalt ISO 3166 standardile täpselt kolmest suurtähest A-Z.** **Tekstiline kood ei tohi olla tühi string ja ainult tühimärkidest koosnev string.}**	
Klassifikaator	nimetus (*name*)	Ametlik klassifikaatori väärtuse nimetus. Riikide nimetused leitakse Eesti Statistika kodulehelt alajaotusest Riikide ja territooriumide klassifikaator 2021 [50]. **{Klassifikaatori unikaalne identifikaator, mis on unikaalne klassifikaatori tüübi piires.** **Registreerimine on kohustuslik. Nimetus ei tohi olla tühi string ja ainult tühimärkidest koosnev string. Nimetus võib olla kuni 50 märki pikk.}**	Aktiivne

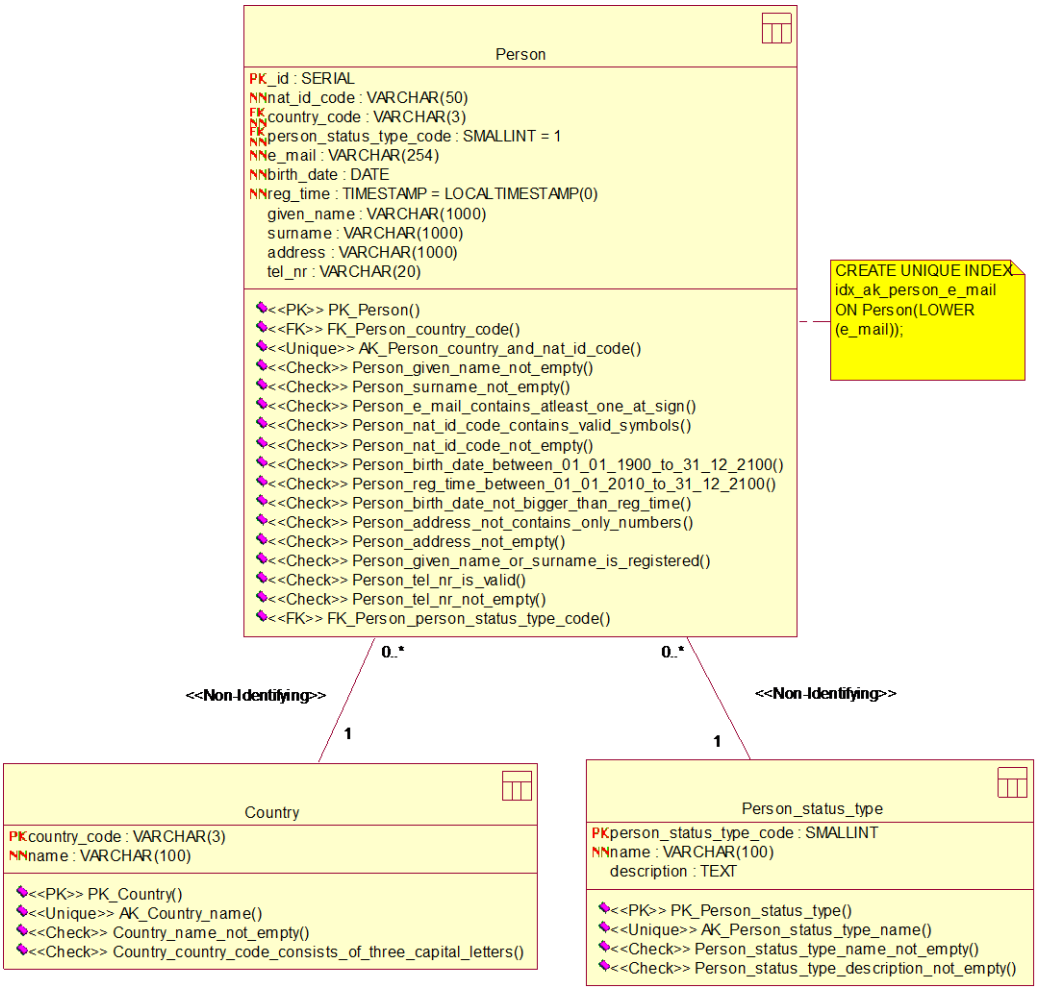
Olemitüübi nimi	Atribuudi nimi (*nimi inglise keeles*)	Atribuudi definitsioon	Näiteväärtus
Klassifikaator	kirjeldus (*description*)	Klassifikaatori väärtuse vabatekstiline kirjeldus. **{Kirjeldus ei tohi olla tühi string ja ainult tühimärkidest koosnev string. Kasutada tuleb andmetüüpi, mis võimaldab suurimat võimalikku stringi pikkust.}**	Juhib organisatsiooni igapäevast tööd ning langetab strateegilisi otsuseid

4.4 Andmebaasi füüsilise disaini mudelid ja realisatsioon

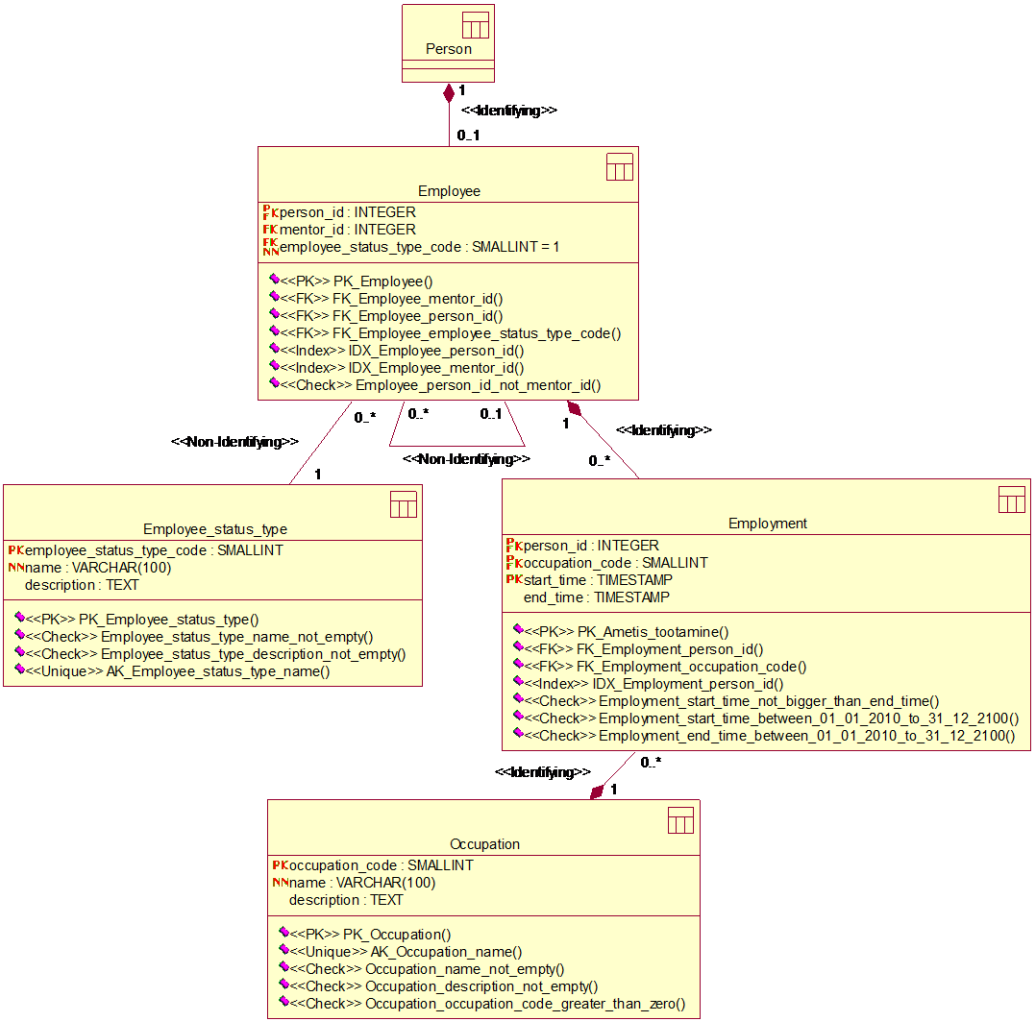
Järgnevates jaotistes esitatakse kõikide erinevate disainidega andmebaaside füüsilise disaini mudelid. PostgreSQL andmebaasi disaini mudelid on loodud kasutades Rational Rose-i ja selle SQL-andmebaaside modelleerimiseks mõeldud UML-i mudeliprofiili. MongoDB andmebaasi modelleerimiseks kasutatakse Moon Modeler modelleerimisvahendit. Kõik andmebaasi disainilahendused lähtuvad kontseptuaalses andmemudelis esitatud nõuetest andmetele. Kuna kitsenduste nimed ilmuvad andmebaasirakenduse veateadetesse, siis on oluline, et kitsendustel oleksid ka lõppkasutajatele arusaadavad nimed. Kitsenduste nimetamisel prooviti sõnaliselt võimalikult täpselt kirjeldada kitsenduse sisu.

4.4.1 „Traditsiooniline“ PostgreSQL andmebaas

Joonised 7 ja 8 esitavad „traditsioonilise“ PostgreSQL andmebaasi füüsilise disaini mudeli. Vastava andmebaasi skeemi defineerimise kood on esitatud Lisas 2. Selles andmebaasis ei kasutata JSON ja JSONB tüüpi veerge.



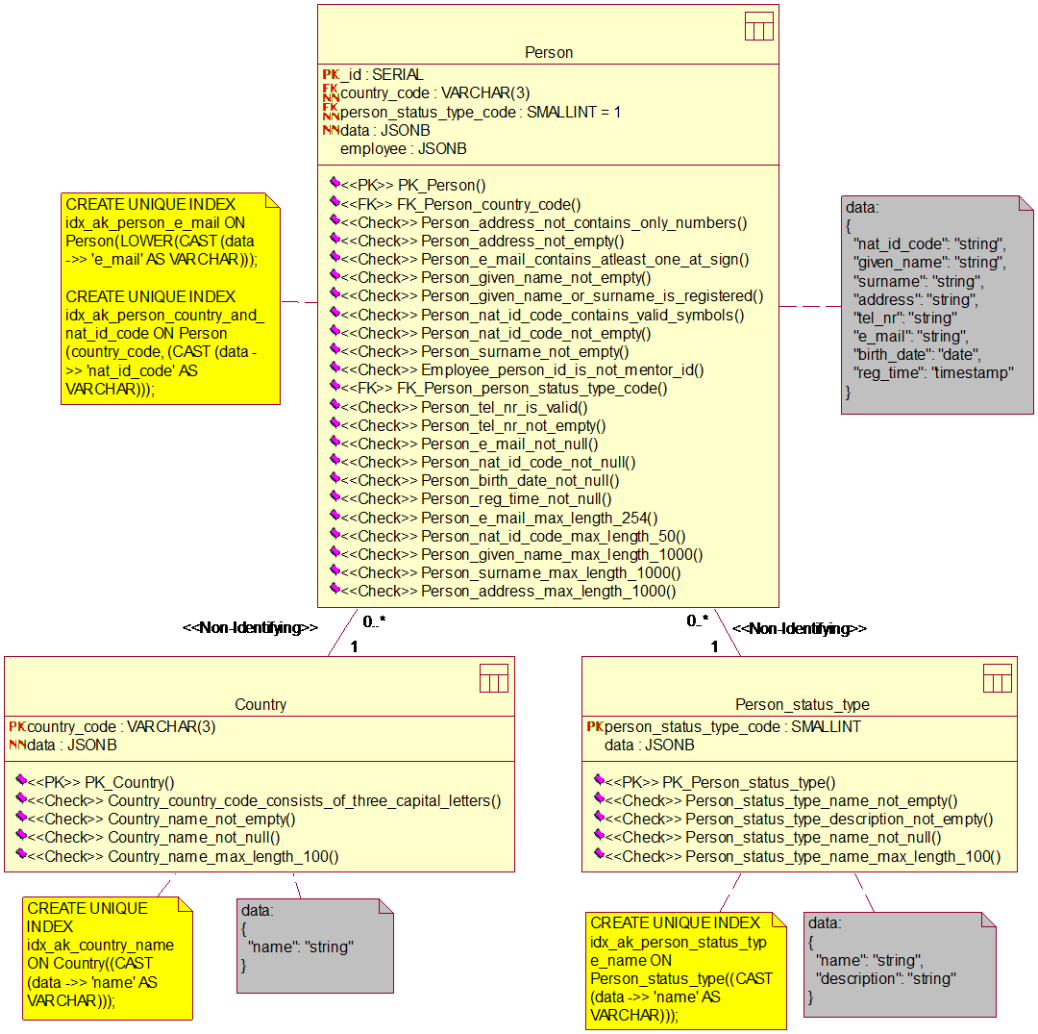
Joonis 7. „Traditsioonilise“ PostgreSQL andmebaasi füüsilise disaini mudel (Isikuga seonduvad tabelid).



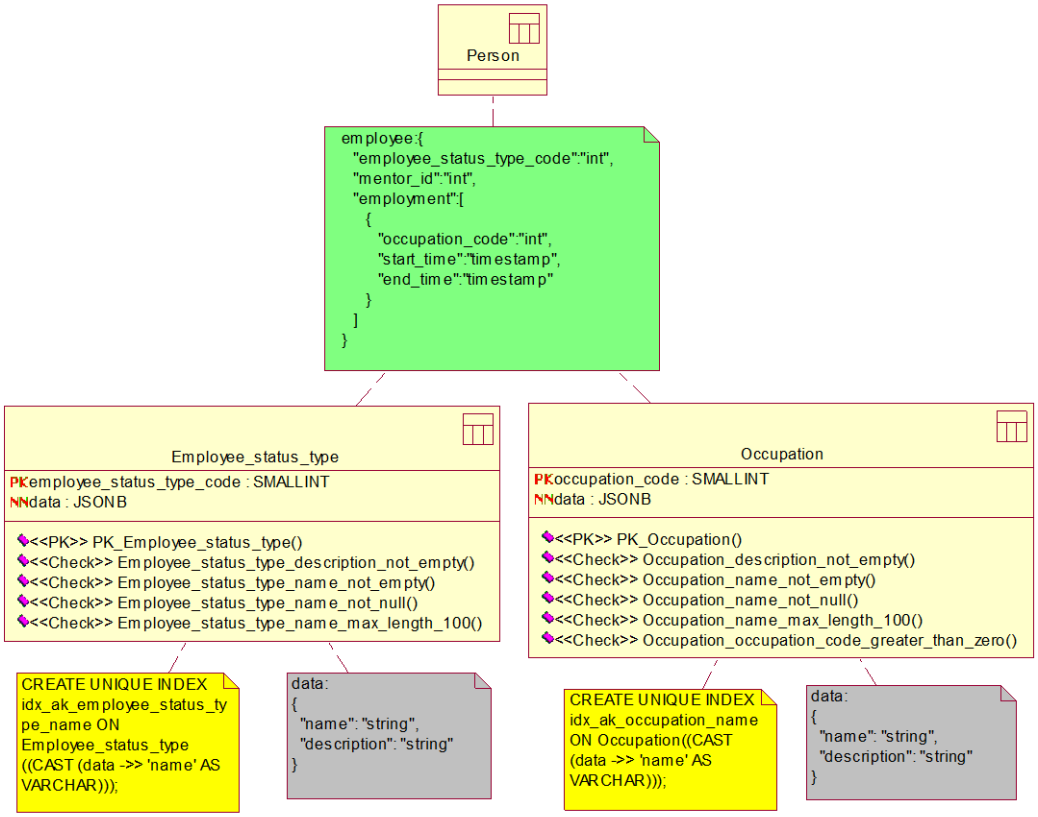
Joonis 8. „Traditsioonilise“ PostgreSQL andmebaasi füüsilise disaini mudel (Töötajaga seonduvad tabelid).

4.4.2 PostgreSQL andmebaas, kus on hierarhilised JSON dokumendid

Joonised 9 ja 10 esitavad hierarhiliste JSON dokumentidega PostgreSQL andmebaasi füüsilise disaini mudeli. Vastava andmebaasi defineerimise skeemi kood on esitatud Lisas 3.



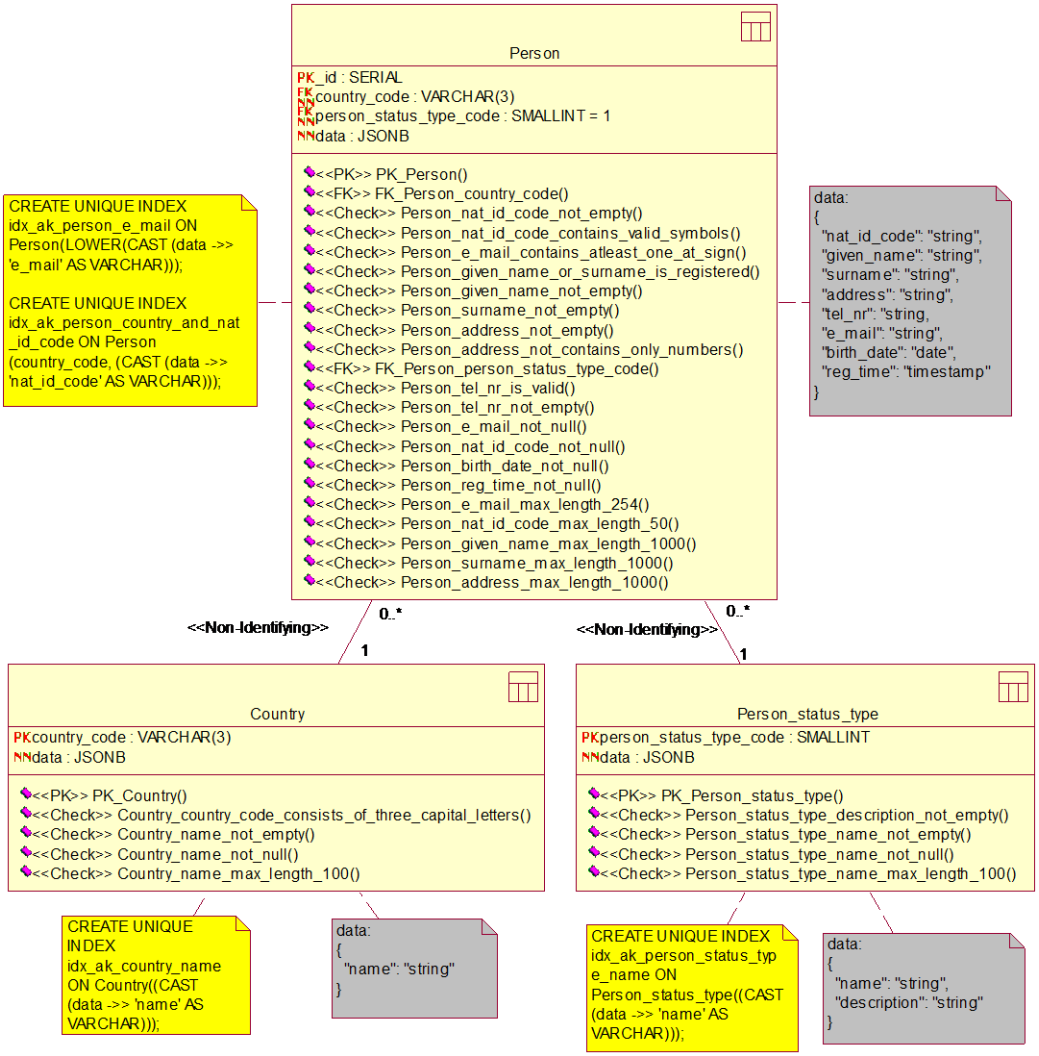
Joonis 9. Hierarhiliste JSON dokumentidega PostgreSQL andmebaasi füüsilise disaini mudel (Isikuga seonduvad tabelid).



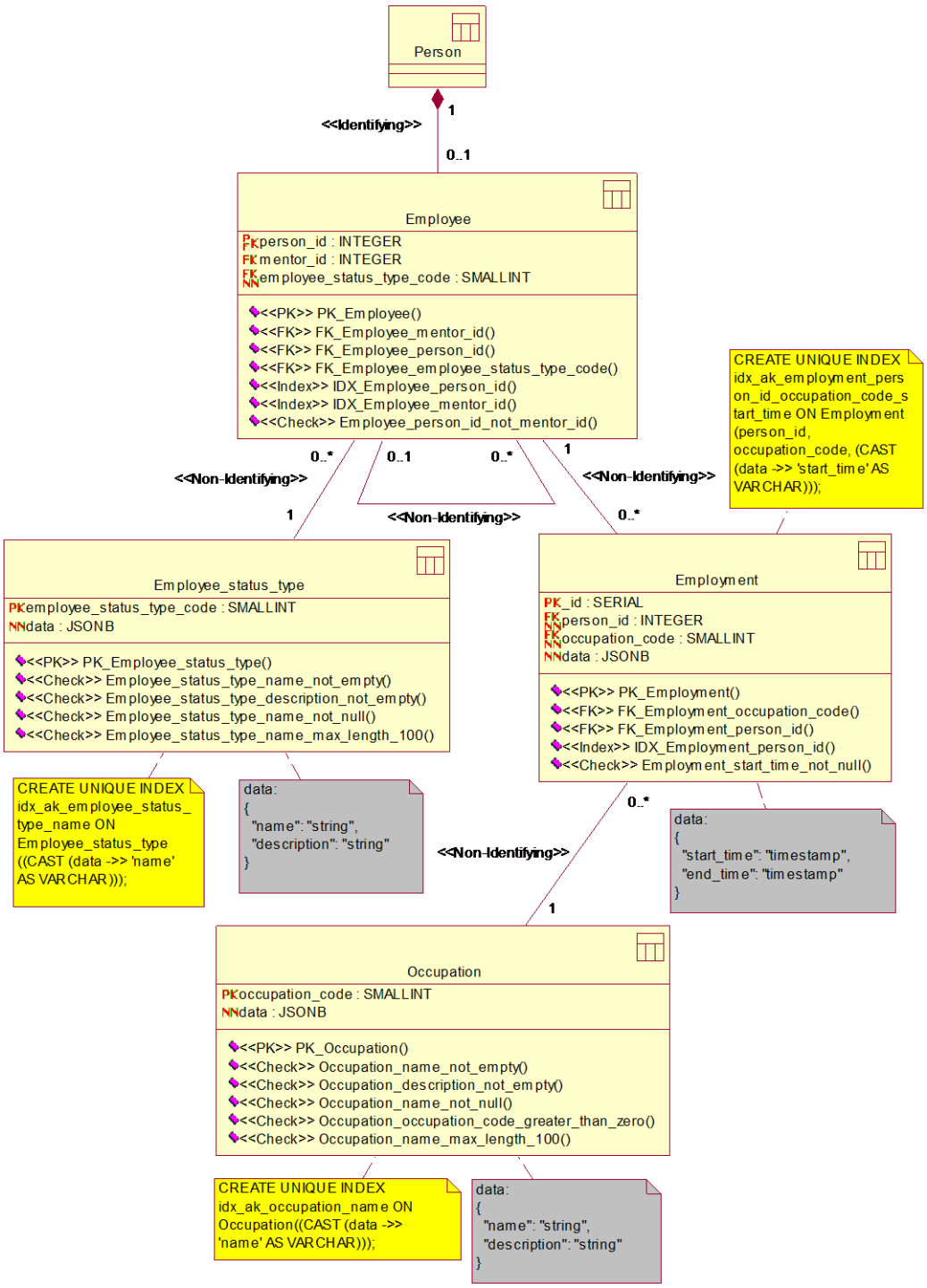
Joonis 10. Hierarhiliste JSON dokumentidega PostgreSQL andmebaasi füüsilise disaini mudel (Töötajaga seonduvad tabelid).

4.4.3 PostgreSQL andmebaas, kus on mitte-hierarhilised JSON dokumendid

Joonised 11 ja 12 esitavad mitte-hierarhiliste JSON dokumentidega PostgreSQL andmebaasi füüsilise disaini mudeli. Vastava andmebaasi skeemi defineerimise kood on esitatud Lisas 4.



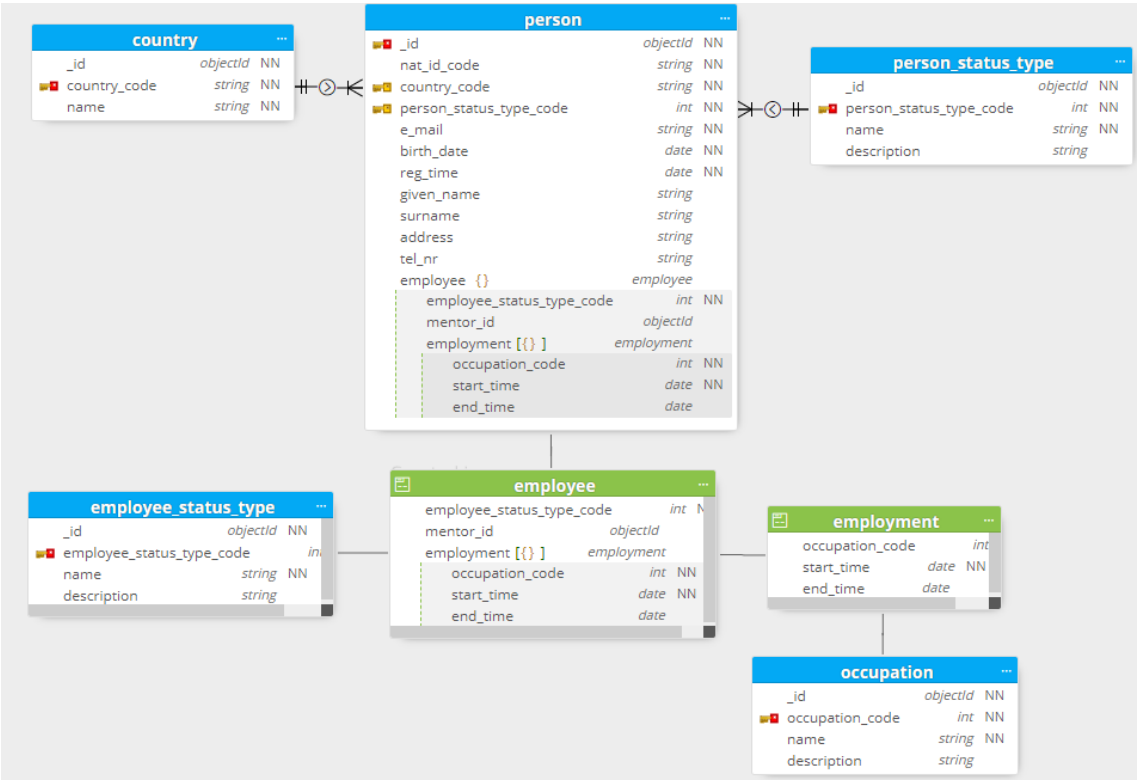
Joonis 11. Mitte-hierarhiliste JSON dokumentidega PostgreSQL andmebaasi füüsilise disaini mudel (Isikuga seotud tabelid).



Joonis 12. Mitte-hierarhiliste JSON dokumentidega PostgreSQL andmebaasi füüsilise disaini mudel (Töötajaga seotud tabelid).

4.4.4 MongoDB andmebaas, kus on hierarhilised JSON dokumendid

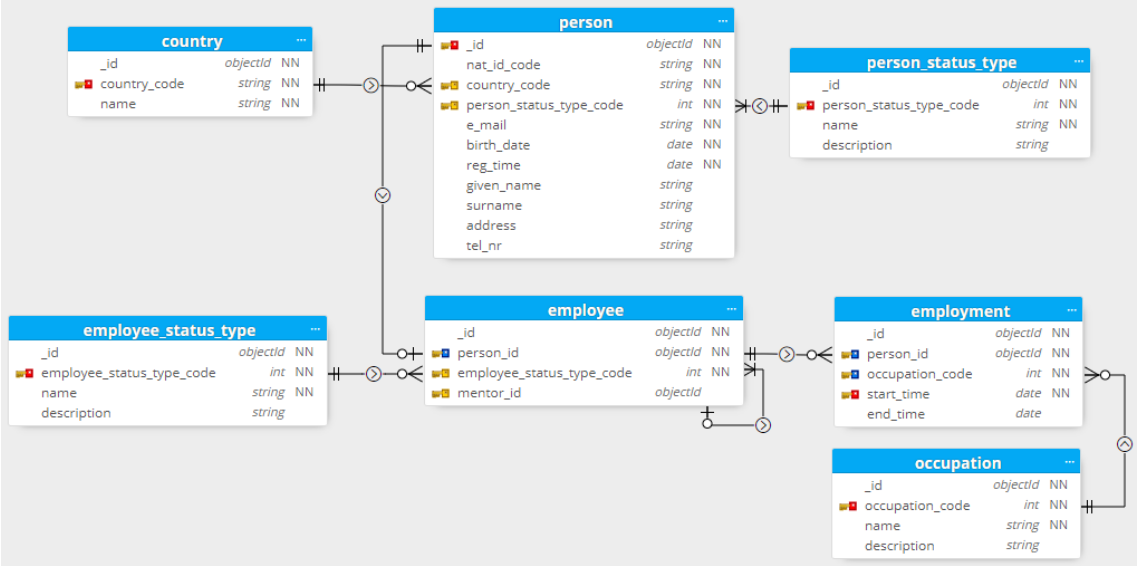
Joonis 13 esitab hierarhiliste JSON dokumentidega MongoDB andmebaasi füüsilise disaini mudeli. Vastava andmebaasi skeemi defineerimise kood on esitatud Lisas 5.



Joonis 13. Hierarhiliste JSON dokumentidega MongoDB andmebaasi füüsilise disaini mudel.

4.4.5 MongoDB andmebaas, kus on mitte-hierarhilised JSON dokumendid

Joonis 14 esitab hierarhiliste JSON dokumentidega MongoDB andmebaasi füüsilise disaini mudeli. Vastava andmebaasi skeemi defineerimise kood on esitatud Lisas 6.



Joonis 14. Mitte-hierarhiliste JSON dokumentidega MongoDB andmebaasi füüsilise disaini mudel.

4.5 Mida lahenduste põhjal võrreldakse?

Andmebaasi disaine võrreldakse järgmistes aspektides. Subjektiivseid hinnanguid põhjendatakse täiendava analüüsiga. Subjektiivsed hinnangud antakse ainuisikuliselt töö autori poolt.

- Andmebaasi skeemi füüsiliste koodiridade arv. Skeemi loomise kood genereeritakse loodud mudelite põhjal automaatselt Rational Rose-is või Moon Modeler-is ning selle struktuuri käsitsi ei muudeta. Koodiridade arv leiti programmi IntelliJ ridade numeratsiooni kasutades.
    - Rational Rose-i skeemi koodis esitatakse veerud, kitsendused jms eraldi ridadel. Moon Modeler-is genereeritud skeemi kood järgib JSON-i vormingut, mistõttu on sagedasest tabulatsiooni kasutamisest tulenevalt koodis märgatavalt rohkem ridu.
- Rakenduse füüsiliste koodiridade arv (ligikaudne – võib sisaldada kommentaare, tühimärke jms). Koodiridade arv leiti iga tagarakenduse versioonis kasutatud Java klassifailide koodiridade liitmisel, kus koodiridade arv võeti IntelliJ-sse sisseehitatud koodiridade numeratsioonist.
- Kontrollide arv, mida õnnestus realiseerida deklaratiivselt andmebaasi tasemel. Kõiki kontrolle üritatakse esmalt jõustada andmebaasi tasemel deklaratiivselt ja rakenduses tehakse seda vaid siis, kui see pole andmebaasis võimalik. Deklaratiivse kontrolliga öeldakse süsteemile, mida on vaja kontrollida, mitte ei esitata kontrollimise protseduuri e sammude jada.
- Kontrollide arv, mis tuli realiseerida rakenduse tasemel, sest andmebaasi tasemel ei saanud seda deklaratiivselt teha.
- Kompenseerivate tegevuste arv, mille sai realiseerida andmebaasi tasemel.
- Subjektiivne hinnang andmekontrollide andmebaasi tasemel jõustamise lihtsusele (skaalal 1 (halvim) … 10 (parim)). Parima hinnangu saamiseks peaks andmebaasis andmekontrollide jõustamine muuhulgas täitma järgmiseid tingimusi.

    - Olema sisult võimalikult lihtne, ehk andmekontrolli toimimiseks poleks vaja läbi viia tegevusi, mis tegelikult ei kuulu valideerimisreegli sisusse (nt teisendamisi).
    - Võimaldama ühes andmekontrollis andmete kontrollimist üle mitme andmevälja (nt juhul, kui mõne välja väärtuse sobivus või kohustuslikkus sõltub teise välja väärtusest või olemasolust).
    - Võimaldama mitme, eraldiseisva andmekontrolli seadmist ühele väljale.
    - Võimaldama võimalikult paljude väärtuste (näiteks ajatempli või kuupäeva) andmetüüpi kuulumise kontrollimist.
    - Võimaldama kontrollida viidete terviklikkust.
- Subjektiivne hinnang kasutajale väljastatud veateadete arusaadavusele (skaalal 1 (halvim) … 10 (parim)). Parima hinnangu saamiseks peaks veateade muuhulgas täitma järgmiseid tingimusi.
    - Sisaldama võimalikult lihtsat ja arusaadavat tekstilist seletust, miks sisestatud andmed polnud sobivad.
    - Sisaldama infot vaid ühe vea kohta, et lihtsustada kasutajal vea leidmist ning parandamist.
    - Sisaldama kasutaja poolt sisestatud andmeid, mis polnud andmebaasis seatud andmekontrollidele sobivad, et lihtsustada vea parandamist.
- Subjektiivne hinnang andmebaasi skeemi ja rakenduse muutmise lihtsusele (skaalal 1 (halvim) … 10 (parim)). Parima hinnangu saamiseks peaks muuhulgas olema võimalik jõustada skeemimuudatus ilma eelnevalt salvestatud andmete muutmiseta.

Eksperimendi käigus viiakse kõikide erinevate andmebaasi disainide korral läbi kolm alameksperimenti. Tehtud skeemimuudatused ilmestavad tüüpilisi muudatusi, mida võidakse teha andmebaasi tavapärase elutsükli käigus. Läbiviidavad muudatused on järgmised.

1. Valideerimisreegli „kood on suurem nullist“ lisamine olemitüübile *Amet*.
2. Uue atribuudi *tel_nr* lisamine olemitüübile *Isik* koos valideerimisreegliga.
3. Uue klassifikaatori *Isiku seisundi liik* loomine koos valideerimisreeglitega ning sellele olemasolevast olemitüübist *Isik* viitamine.

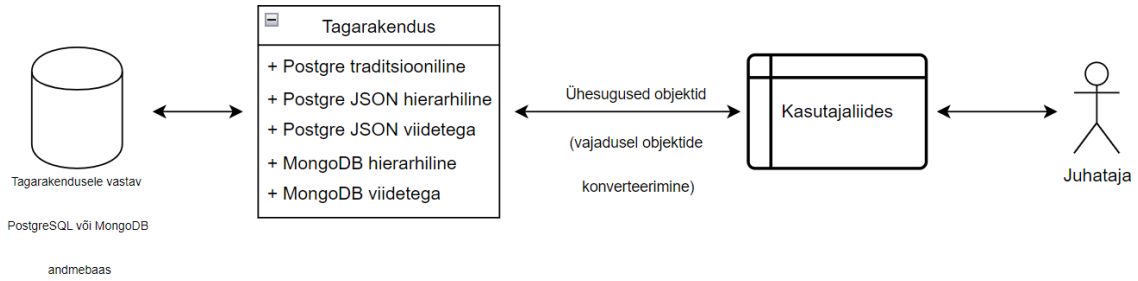
Nii töös esitatud kontseptuaalne andmemudel kui ka andmebaasi disaini mudelid kirjeldavad andmebaasi nende muudatuste järel. Tegelikult loodi andmebaas algselt ilma nende täiendusteta ja eksperimendi käigus tehti ettenähtud täiendused.

- Subjektiivne hinnang üldisele andmebaasi ja rakenduse loomise lihtsusele (skaalal 1 (halvim) … 10 (parim)). Parima hinnangu saamiseks peaksid muuhulgas olema täidetud järgmised tingimused.
    - Võimalikult palju andmekontrolle peab olema võimalik jõustada andmebaasis deklaratiivsel viisil.
    - Andmebaasisüsteem peab tagama võimalikult suurel määral viidete terviklikkuse reegli täidetuse.
    - Olemi andmete muutmine peaks võimalikult vähe mõjutama teiste olemite andmeid, et väheneks ebakorrektsete andmete salvestamise tõenäosus ja oleks lihtsam luua rakendust.

Kui eesmärgiks oleks parima andmebaasi disaini väljavalimine konkreetse olukorra jaoks, siis oleks lähtuvalt valiku kontekstist (projekti oludest, ettevõtte huvidest) nendel aspektidel erinev suhteline tähtsus. Kuna käesoleva töö eesmärk ei ole valida välja konkreetset disaini konkreetse olukorra jaoks, siis käsitletakse neid aspekte võrdselt tähtsana.

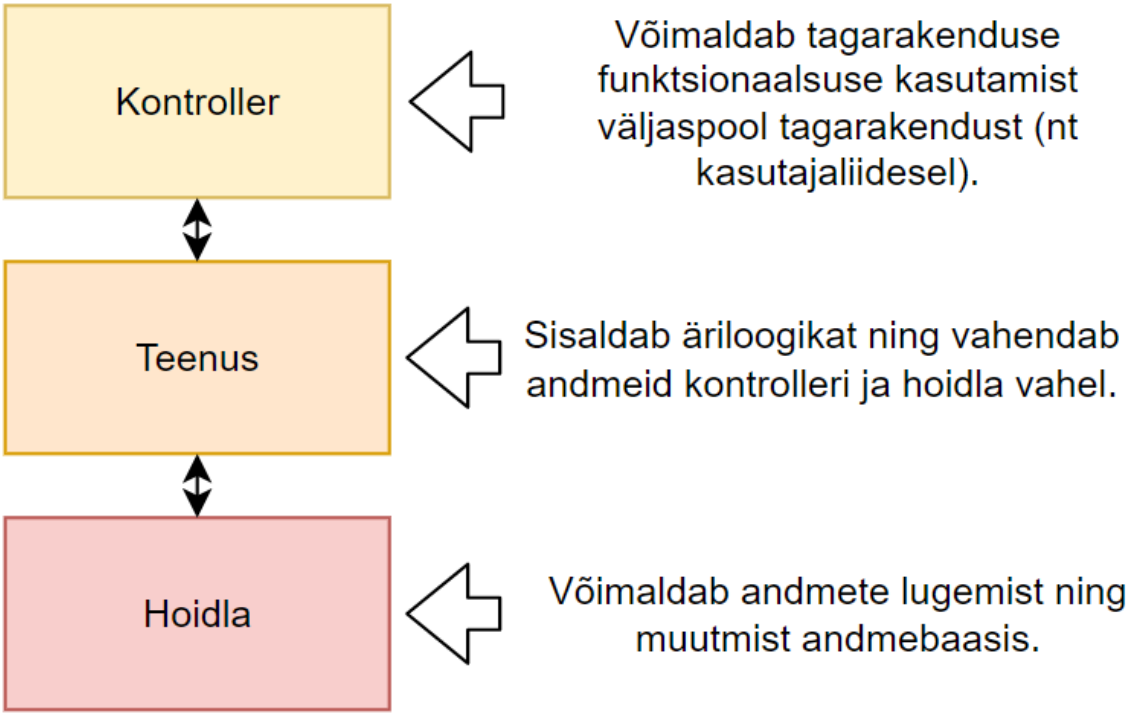
4.6 Rakenduste disain

Ülesehituselt on erinevate disainidega andmebaaside kasutamiseks loodud rakendused ühesugused (vt Joonis 15). Kasutajaliides (vt Joonised 17–21) on kõikidel rakendustel samasugune. Selleks, et kasutada kõikide erinevate tagarakenduste korral sama kasutajaliidest, on osadel rakendustel vaja ümber teisendada mõningad kasutajaliidesest tulevad objektid tagarakendusele sobivale kujule ning sama tuleb teha ka teistpidi suheldes. Taoline objektide teisendamine tuleneb asjaolust, et kasutajaliides on loodud käsitlema andmete struktuuri, milles pole hierarhilisi andmeid, nagu on kahel andmebaasi disainil. Kasutajaliidese loomisel tegi autor sellise arhitektuurilise otsuse, sest see võimaldas kõige hõlpsamalt luua ühtse kasutajaliidese kõikidele tagarakenduste versioonidele. Samuti on kõikidel tagarakenduste versioonidel samade URL-idega lõpp-punktid (*endpoints*), et lihtsustada rakenduste loomist ning vältida üksteist kordavate rakenduste toimimisloogikate loomist. Lisa 7 sisaldab nimekirja kasutusel olevatest lõpp-punktidest.



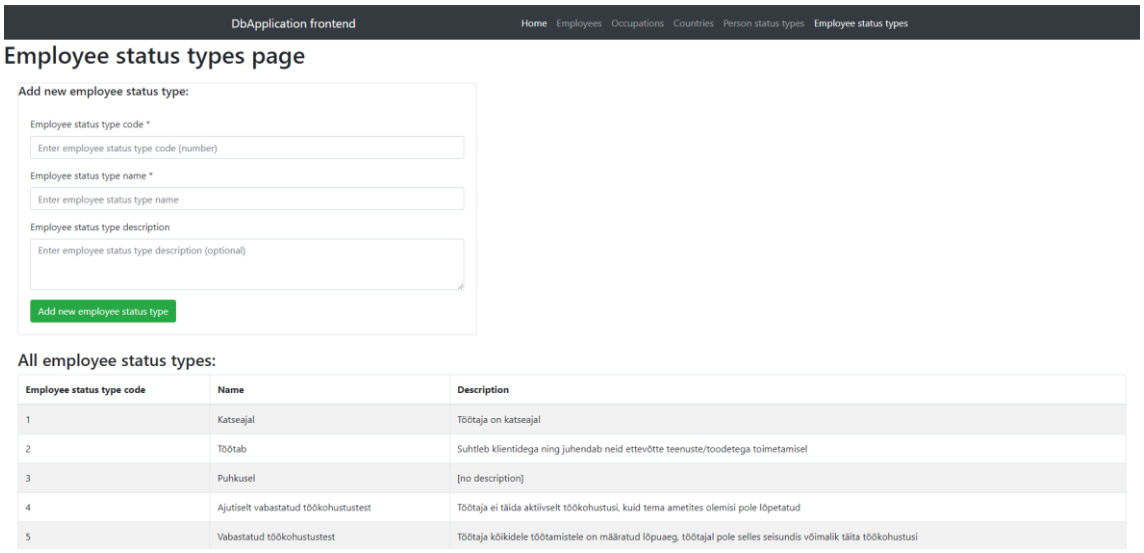
Joonis 15. Rakenduste disain.

Tagarakendused järgivad Spring Boot-iga tehtud tagarakendustes levinud kontroller-teenus-hoidla (*controller-service-repository*) mustrit (vt Joonis 16), ehk rakendus on jaotatud kolmeks kihiks otstarbe lahususe põhimõttel, mis lihtsustab rakenduse arhitektuuri mõistmist ning üksustestide kirjutamist.

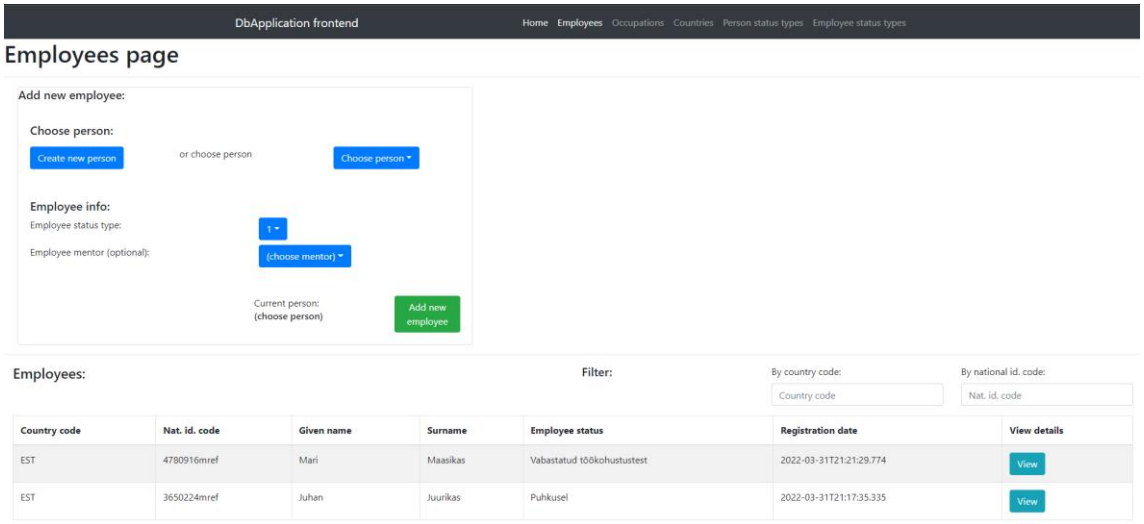


Joonis 16. Tagarakenduste disain.

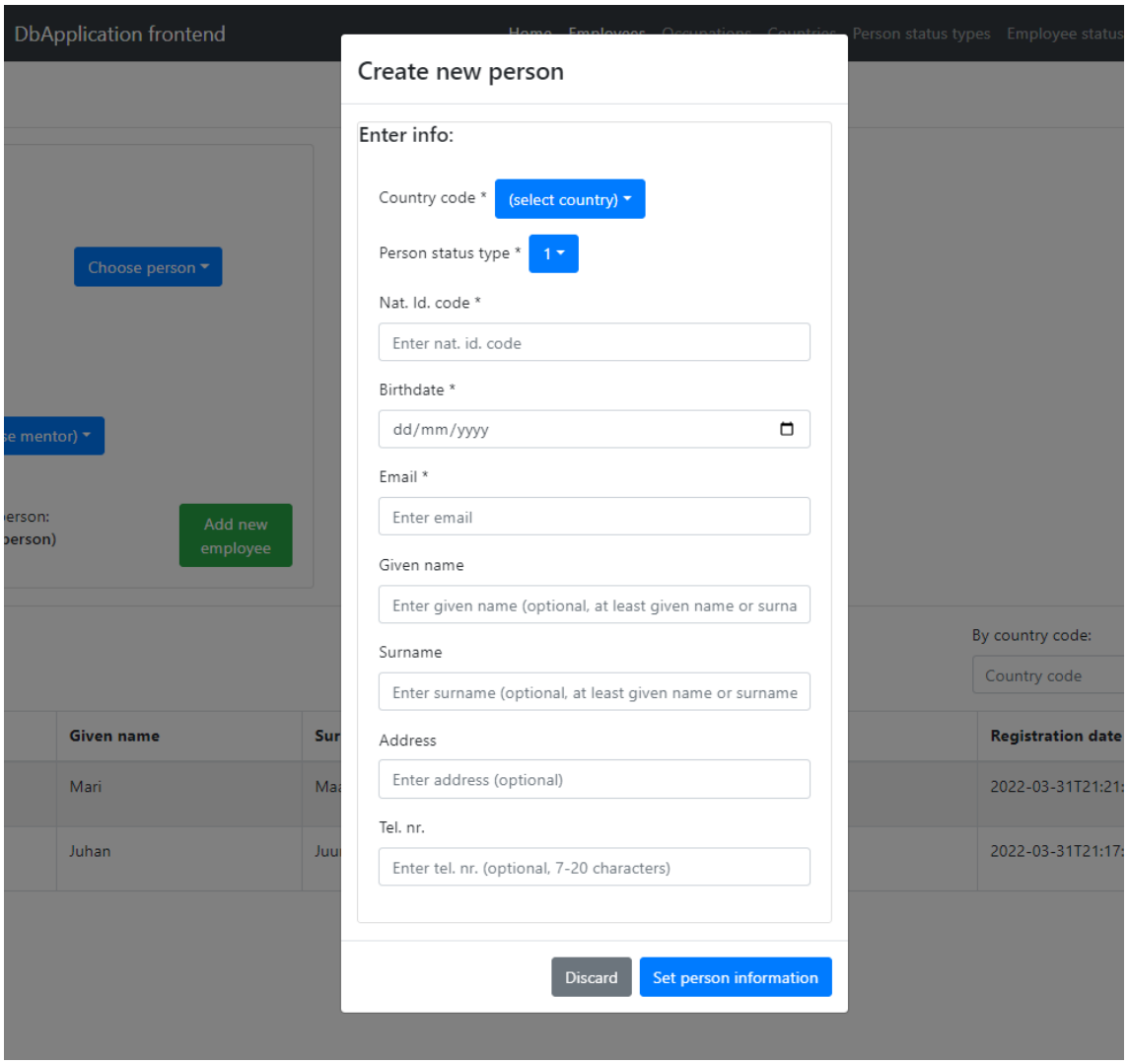
Kontrolleri eesmärgiks on võimaldada tagarakenduse funktsionaalsuse kasutamist kasutajaliideses või teistes tagarakendustes. Teenusekiht sisaldab tagarakenduse äriloogikat, kus näiteks sooritatakse rakendusepoolseid andmete valideerimiskontrolle ning vajadusel objektide teisendamist teenusele või hoidlale sobivale kujule. Hoidla ülesanne on suhelda andmebaasiga, teostades andmete lugemise või muutmise päringuid.



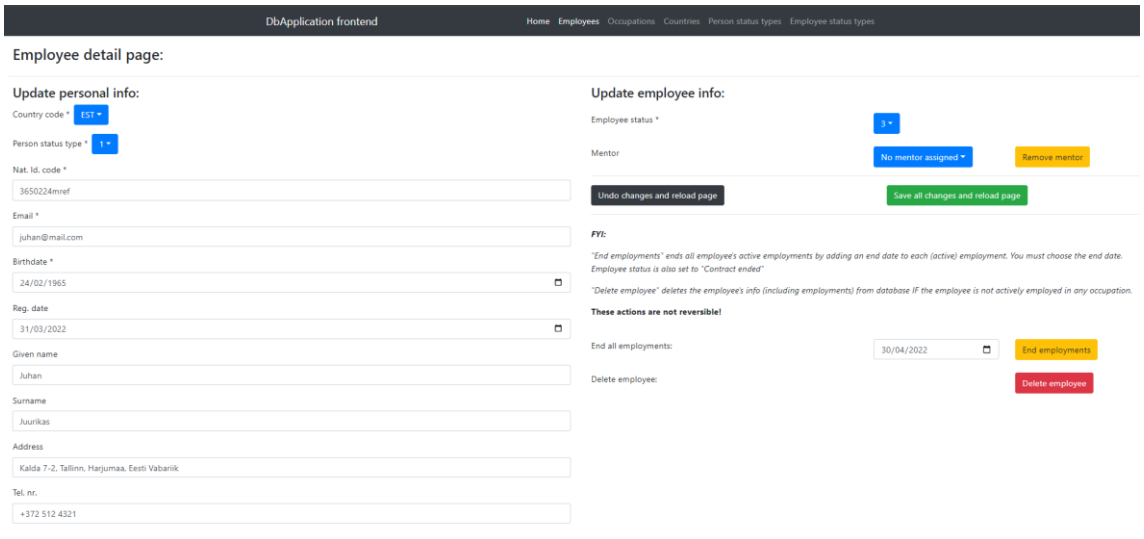
Joonis 17. Kasutajaliidese näide - töötaja seisundi liikide leht.



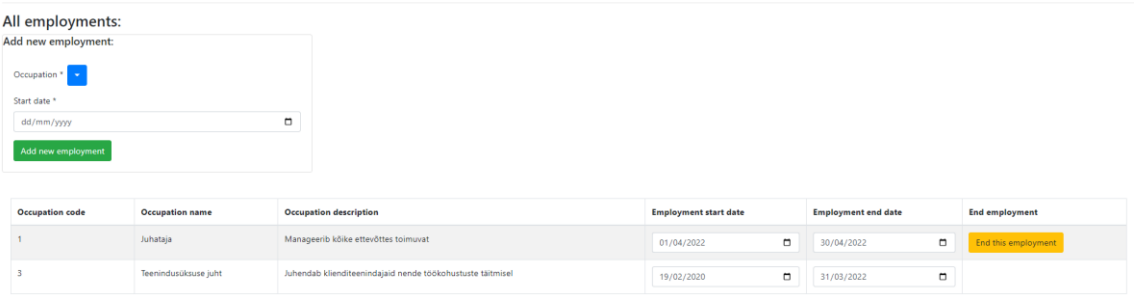
Joonis 18. Kasutajaliidese näide - töötajate leht.



Joonis 19. Kasutajaliidese näide - uue isiku loomine.



Joonis 20. Kasutajaliidese näide - töötaja detailvaate leht.



Joonis 21. Kasutajaliidese näide - töötaja ametis olemised.

4.7 Rakenduse testimine

Rakendust ja andmebaase testiti kolmes etapis. Esmalt prooviti andmeid salvestada andmebaasis kasutades vastava andmebaasisüsteemi haldusrakendust (pgAdmin või MongoDB Compass). Teiseks prooviti salvestada andmeid läbi tagarakenduse ning viimaks testiti andmete salvestamist läbi kasutajaliidese. Läbi tagarakenduse ning kasutajaliidese andmete salvestamist kontrolliti eraldi, et veenduda korrektses andmete vahetamises rakenduse eri kihtide vahel. Eraldi testides oli ka paremini märgata ning parandada rakenduste loomisel tehtud vigu.

Kõikide etappide korral prooviti esmalt sisestada korrektseid andmeid, veendumaks, et valideerimisreeglid lubavad nõuete järgi sobivate andmete salvestamist. Peale seda prooviti sisestada skeemi suhtes mittesobivaid andmeid, et kontrollida selliste andmete salvestamise ebaõnnestumist.

5 Eksperimendi tulemused

Järgnevates jaotistes esitatakse hinnangud igale andmebaasi disainile aspektides, mida kirjeldati jaotises 4.5. Hinnangute lühikokkuvõte on tabelina esitatud jaotises 5.5. Lisas 8 on olemas andmekontrollide nimekirjad, milles on kirjeldatud, milliseid kontrolle oli võimalik teha andmebaasi tasemel. Lisas 9 on kirjeldatud, milliseid viidete terviklikkuse kompenseerivaid tegevusi sai andmebaasides realiseerida. Andmebaasis jõustatud kontrollide ja realiseeritud kompenseerivate tegevuste arvu kasutatakse iga andmebaasi hindamisel. Andmebaasis jõustatud kontrollide kokkuvõte on esitatud jaotises 5.2.7 Tabelis 6. Töö tulemusena valminud andmebaasi ja rakenduse lähtekood on avaldatud avalikul GitHubi lehel [1].

5.1 Andmebaasi skeemimuudatused

Jaotistes 5.1.1–5.1.3 kirjeldatakse andmebaasi skeemimuudatusi. Skeemimuudatusi tehti kasutades vastava andmebaasisüsteemi haldusrakendust.

Rakendustes tehtud uuendused ei olnud suuremahulised ning oli võimalik mõningate muudatustega kasutada olemasolevat koodi. Rakenduste kasutajaliideses tuli skeemi uue atribuudi lisamisel luua uued andmesisestusväljad ning olemitüübi lisamisel haldamise leht, mis suhtlevad tagarakendustega. Tagarakendustes oli vaja lisada uusi lõpp-punkte, teenuseid ning klasse, et võimaldada uute andmete salvestamist.

5.1.1 Valideerimisreegli „kood on suurem nullist“ lisamine olemitüübile Amet

Muudatuse jõustamine oli erinevates andmebaasides ning disainides sarnase sisu ning raskusastmega. Antud valideerimisreegli jõustamiseks tuli kõikides PostgreSQL andmebaasides käivitada lause, mis on esitatud Joonisel 22. See lisas vastava kitsenduse.

```
ALTER TABLE Occupation ADD CONSTRAINT
Occupation_occupation_code_greater_than_zero
CHECK (occupation_code > 0)
```

Joonis 22. Valideerimisreegli "kood on suurem nullist" jõustamine PostgreSQL andmebaasides.

MongoDB andmebaasides sai aga skeemi muudatuse teha hõlpsamalt läbi MongoDB Compass-i, lisades vastava kollektsiooni skeemile uue valideerimisreegli „*minimum: 0“*.

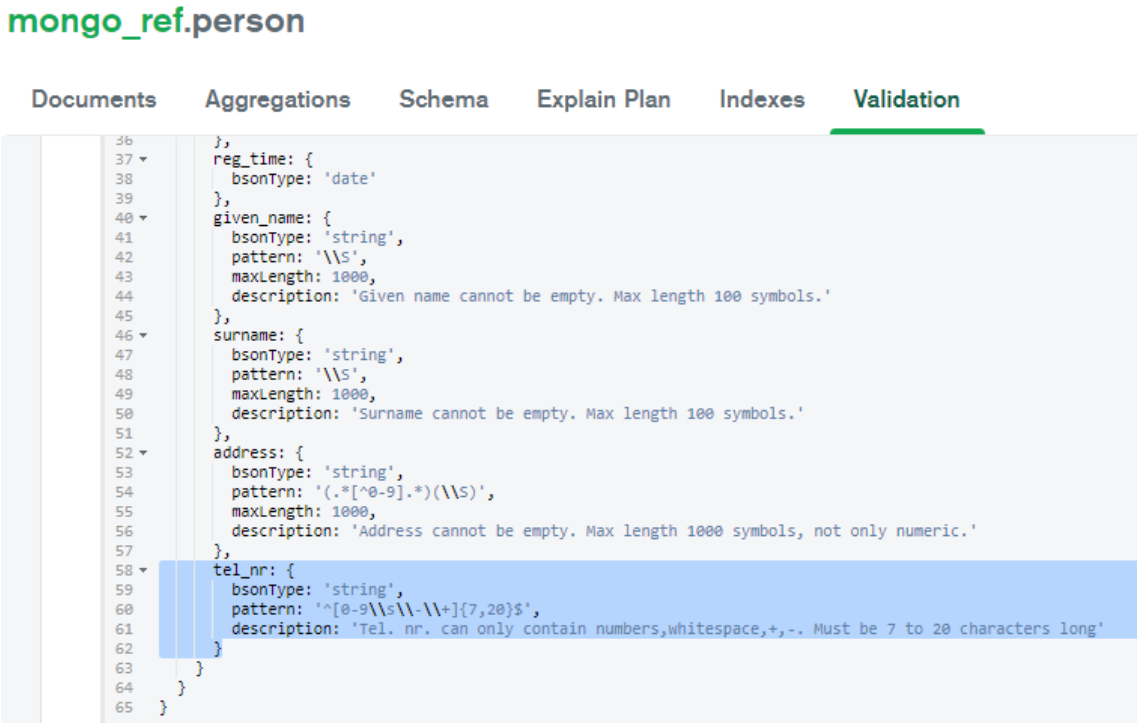
5.1.2 Uue atribuudi *tel_nr* lisamine olemitüübile *Isik* koos valideerimisreegliga

Atribuudi *tel_nr* lisamine olemitüübile *Isik* koos valideerimisreegliga oli MongoDB andmebaasides mõnevõrra lihtsam võrreldes PostgreSQL andmebaasidega. MongoDB andmebaasides sai skeemi taaskord kasutajasõbralikult muuta, kasutades MongoDB Compass-i. PostgreSQL korral tuli traditsioonilise disaini puhul esmalt lisada *Person* tabelisse uus veerg, pärast mida sai lisada ka valideerimist sooritava kitsenduse. PostgreSQL JSON disainide korral ei pidanud lisama tabelisse uut veergu, vaid sai kohe lisada valideerimiseks mõeldud kitsenduse. Läbiviidud skeemimuudatused on nähtavad ka Joonistel 23 ja 24.

```
ALTER TABLE Person
ADD COLUMN tel_nr VARCHAR (20);

ALTER TABLE Person
ADD CONSTRAINT Person_tel_nr_is_valid CHECK (tel_nr ~* '^[0-9\s\-\+]{7,20}$'),
ADD CONSTRAINT Person_tel_nr_not_empty CHECK (tel_nr !~ '^[[:space:]]+$' AND tel_nr <> '');
```

Joonis 23. Näide *tel_nr* lisamisest PostgreSQL andmebaasides.



Joonis 24. Näide *tel_nr* lisamisest MongoDB andmebaasides.

5.1.3 Uue klassifikaatori *Isiku seisundi liik* loomine koos valideerimisreeglitega ning sellele olemasolevast olemitüübist *Isik* viitamine

Uue klassifikaatori – *Isiku seisundi liik* – loomisel ei olnud erinevate andmebaaside puhul märkimisväärset vahet. Küll aga oli märgatav erinevus uue loodud klassifikaatori sidumisel isikuandmetega. Nimelt oli MongoDB andmebaasides võimalik lisada skeemi nõue, et isikul on kohustuslik isiku seisundi liigi olemasolu (ehk teisisõnu *NOT NULL* kitsendus), lisades skeemi *required* massiivi välja nime *person_status_type_code*. Uue skeemi nõude jõustamiseks ei pidanud muutma eelnevalt salvestatud objekte, kus ei olnud määratud seisundi liiki. PostgreSQL andmebaaside korral oli enne tarvis lisada *Person* tabelisse uus seisundit kirjeldav veerg, igal real määrata seisundi väärtus ning alles siis oli võimalik seada seisundi veerg kohustuslikuks. Alternatiivina oleks saanud *NOT NULL* kitsenduse realiseerida *CHECK* kitsenduse abil, mis lisatakse tabelisse *ALTER TABLE* lausega ja mis on seisundis *NOT VALID* [51]. Sellisel juhul ei kontrollita olemasolevate andmete kitsendusele vastavust, kuid seda tehakse kõigi uute andmete korral.

Joonistel 25 ja 26 on toodud näited andmebaasides tehtud skeemimuudatustest.

```
CREATE TABLE Person_status_type (
    person_status_type_code SMALLINT NOT NULL,
    name VARCHAR(100) NOT NULL,
    description TEXT,
    CONSTRAINT PK_Person_status_type PRIMARY KEY (person_status_type_code),
    CONSTRAINT AK_Person_status_type_name UNIQUE (name),
    CONSTRAINT Person_status_type_description_not_empty CHECK (description !~ '^[[:space:]]+$' AND description <> ''),
    CONSTRAINT Person_status_type_name_not_empty CHECK (name !~ '^[[:space:]]+$' AND name <> '')
);
INSERT INTO Person_status_type(person_status_type_code, name) VALUES (1, "Elus");

ALTER TABLE Person ADD COLUMN person_status_type_code SMALLINT;
UPDATE TABLE Person SET person_status_type_code=1 WHERE person_status_type_code = NULL;
ALTER TABLE Person ADD CONSTRAINT FK_Person_person_status_type_code FOREIGN KEY (person_status_type_code)
    REFERENCES Person_status_type (person_status_type_code) ON DELETE NO ACTION ON UPDATE CASCADE;
```

Joonis 25. Näide *Isiku seisundi liigi* loomisest ja viitamisest PostgreSQL andmebaasides.

```
{
  $jsonSchema: {
    bsonType: 'object',
    title: 'person_status_type',
    required: [
      'person_status_type_code',
      'name'
    ],
    properties: {
      person_status_type_code: {
        bsonType: 'int'
      },
      name: {
        bsonType: 'string',
        pattern: '\\S',
        maxLength: 100,
        description: 'Person status type name cannot be empty. Max length 100.'
      },
      description: {
        bsonType: 'string',
        pattern: '\\S',
        description: 'Person status type description cannot be empty.'
      }
    }
  }
}
```

Joonis 26. Näide *Isiku seisundi liigi* loomisest MongoDB andmebaasides.

5.2 Kitsenduste jõustamine

Selles jaotises kirjutatakse kitsenduste jõustamisest andmebaasis. Kitsendusteks andmebaasis olevatele andmetele on:

- atribuudi tüüp (atribuudi väärtus tohib kuuluda ainult tüübiga väärtuste hulka),
- atribuudi väärtuse maksimaalne lubatud pikkus (tekstilise väärtuse puhul) või täpsus ja skaala (püsikomatüüpi väärtuste puhul),
- kohustuslikkus (atribuudi väärtus peab olema registreeritud või seos peab olema olemas),
- unikaalsus (atribuudi väärtus või atribuutide väärtuste kombinatsioon peab kõigi sama tüüpi olemite puhul olema erinev),
- viidete terviklikkus (andmebaasis ei tohi olla katkiseid seoseid),
- lisapiirangud (täiendavad piirangud atribuutide väärtustele või seostele).

5.2.1 Andmetüübid

Andmetüüpide kaudu määratud kitsendused sai peaaegu täielikult jõustada kõikides andmebaasi disainides. Ainukesed, mida ei saanud jõustada, oli kuupäevade ning ajatemplite andmetüüpide kontroll PostgreSQL JSON dokumentidega andmebaasides, sest PostgreSQL ei suuda teisendada näiteks JSON välja *"“reg_time“: [2022,4,19,23,32,39,906454000]“* väärtust ajatempliks või *"“birth_date“: 1648598400000“* väärtust kuupäevaks. Niisugused väljade formaadid tulenevad tagarakendustest andmete serialiseerimise ning andmebaasis deserialiseerimise tagajärjel, mille parendamiseks autor ei leidnud võimalust.

5.2.2 Väljapikkused

Väljapikkuseid oli võimalik kõikides andmebaaside disainides kontrollida. Traditsioonilise PostgreSQL andmebaasi disaini ja MongoDB andmebaasides oli väljapikkuseid lihtne kontrollida, kasutades näiteks traditsioonilises PostgreSQL andmebaasis deklaratsiooni „*VARCHAR (100)“* või MongoDB andmebaasides *„maxLength: 100“*. Väljapikkuste kontrollimine oli tülikam PostgreSQL JSON dokumentidega andmebaasides, kus oli esmalt tarvis teisendada JSON-i dokumendi väljas olev väärtus andmebaasisüsteemile sobivale kujule, peale mida sai alles kontrollida selle pikkust. Näiteks oli riigi nime väljapikkuse kontrollimiseks vaja defineerida kitsendus, mis on esitatud Joonisel 27.

```
CONSTRAINT Country_name_max_length_100
CHECK (char_length((CAST (data ->> 'name' AS VARCHAR))) <= 100)
```

Joonis 27. Riigi nime pikkuse kontrolli defineerimine PostgreSQL JSON dokumentidega andmebaasides.

5.2.3 Kohustuslikkus

Andmete kohustuslikkuse kitsendused olid täielikult jõustatavad kõikides andmebaasi disainides, välja arvatud PostgreSQL hierarhiliste JSON dokumentidega andmebaasis. JSON dokumendi väljade kohustuslikkuse jõustamiseks kasutati kitsendusi nagu näiteks Joonisel 28.

```
CONSTRAINT Occupation_name_not_null
CHECK ((CAST (data ->> 'name' AS VARCHAR)) IS NOT NULL)
```

Joonis 28. Kohustuslikkuse tagamise näide PostgreSQL JSON dokumentidega andmebaasides.

Niisugust kontrolli ei saanud jõustada töötaja seisundi liigi koodile, ametis töötamise ameti koodile ja ametis töötamise alguse ajale, sest neid andmeid pole uue isiku registreerimisel olemas (ehk on *NULL*-id), mille tagajärjel poleks salvestatavad andmed korrektsed ning andmebaasi poleks võimalik lisada uusi isikuid.

Mitte-JSON andmetüüpi olevate väljade kohustuslikkuse tagamiseks kasutati PostgreSQL andmebaasides *NOT NULL* kitsendust. MongoDB andmebaasides oli võimalik kohustuslikkust jõustada skeemis *required* massiivi defineerimisega, nagu näiteks *„required: ['occupation_code','name']“*.

5.2.4 Unikaalsus

Kõik unikaalsuse kontrollid oli võimalik jõustada andmebaasides, mis ei sisaldanud hierarhilisi JSON dokumente. Hierarhilisi JSON dokumente sisaldavates andmebaasides polnud võimalik jõustada ametis töötamise puhul isik_id, ameti koodi ning ameti asumise alguskuupäeva kombinatsiooni unikaalsust, sest vastavates disainides pole hierarhia kasutamise pärast võimalik luua igale ametis olemisele indeksit. Nende disainide korral tuleb seda unikaalsust kontrollida tagarakenduses.

Teistel juhtudel on võimalik luua unikaalsuse kontrolle PostgreSQL andmebaasides näiteks deklaratsiooniga, mida on näha Joonisel 29 või unikaalsuse indeksi loomisega PostgreSQL JSON dokumentidega andmebaasides, nagu näitab Joonis 30. MongoDB andmebaasides saab samuti unikaalsust jõustada unikaalse indeksi loomisega, mis on esitatud Joonisel 31.

```
CONSTRAINT AK_Employee_status_type_name UNIQUE (name)
```

Joonis 29. Unikaalsuse kontrolli jõustamine kitsendusega PostgreSQL andmebaasides.

```
CREATE UNIQUE INDEX idx_ak_employee_status_type_name ON
Employee_status_type((CAST (data ->> 'name' AS VARCHAR)))
```

Joonis 30. Unikaalsuse kontrolli jõustamine indeksiga PostgreSQL andmebaasides.

```
db.employee_status_type.createIndex({
  "name": 1
}, {
  name: "ak_employee_status_type_name",
  unique: true
})
```

Joonis 31. MongoDB andmebaasis unikaalsuse jõustamise näide.

5.2.5 Viidete terviklikkus

Viidete terviklikkust on võimalik jõustada PostgreSQL andmebaasides juhtudel, kui välisvõtme väli pole JSON dokumendis. Sellistel juhtudel on võimalik viiteid seada näiteks lausega, mida on kujutatud Joonisel 32.

```
ALTER TABLE Person ADD CONSTRAINT FK_Person_country_code FOREIGN KEY
(country_code) REFERENCES Country (country_code)  ON DELETE NO ACTION
ON UPDATE CASCADE
```

Joonis 32. Viite seadmise näide PostgreSQL-is.

Kui viide sisaldub PostgreSQL-is JSON dokumendis, siis viidete terviklikkust pole võimalik deklaratiivselt jõustada, sest JSON dokumendi välja teisendamisel *cast* käsuga ja siis sellele välisvõtme kitsenduse deklareerimisel (nagu on Joonisel 32 esitatud lauses *FOREIGN KEY*) annab andmebaasisüsteem veateate. Seega pole PostgreSQL-i loodud niisugusel viisil viidete jõustamise võimekust.

MongoDB andmebaasides pole võimalik viidete terviklikkust skeemi tasemel jõustada. MongoDB JSON dokumentides on võimalik viidata teistele dokumentidele näiteks kasutades dokumentidele andmebaasisüsteemi poolt automaatselt genereeritud *_id* välja, mis kasutab *ObjectId* [52] andmetüüpi (nt *„_id: ObjectId(' 6245eefe565e2c80a16bd872')“*). See ei taga korrektsete viidete olemasolu, sest *ObjectId* väärtuse saab seada omavoliliselt, ehk dokumendis võib olla viide teisele dokumendile, mida pole tegelikult andmebaasis registreeritud. Viidete terviklikkuse puudumise pärast puuduvad ka välisvõtmete kompenseerivad tegevused, mis muudab andmebaasirakendust keerukamaks, sest kompenseerivad tegevused peab realiseerima rakenduses (vt Lisa 9). Näiteks kasutusmalli „Kustuta töötaja“ realisatsioonis MongoDB mitte-hierarhiliste JSON dokumentidega andmebaasi põhjal peab tagarakenduses kirjutama loogika, mis kustutaks kõik kustutatava töötajaga seotud ametis töötamised. Teistes andmebaasi disainides niiviisi tegema ei pea.

5.2.6 Lisapiirangud

Lisapiiranguid oli rohkem võimalik jõustada PostgreSQL andmebaasides. JSON dokumentidega PostgreSQL andmebaasides polnud võimalik jõustada kuupäevade/ajatemplite valideerimisi (vt jaotis 5.2.1) ning kitsenduste süntaks oli keerulisem kui traditsioonilise PostgreSQL andmebaasi disaini korral. JSON dokumendi

välja valideerimisel peab korrektselt navigeerima JSON dokumendis valideeritava väljani ja teisendama selles oleva väärtuse PostgreSQL-ile sobivat tüüpi väärtuseks, peale mida saab alles kontrollida selle reeglile vastavust. Joonisel 33 on kirjeldatud kitsenduse *„Isikukoodis on lubatud tähed (lubatud on ka muud tähed kui ASCII tähed a-zA-Z), numbrid, tühikud, sidekriipsud, plussmärgid, võrdusmärgid ja kaldkriipsud.“* jõustamist erinevates PostgreSQL andmebaasi disainide korral.

```
---Traditsioonilises PostgreSQL andmebaasis---
CONSTRAINT Person_nat_id_code_contains_valid_symbols
      CHECK (nat_id_code ~* '^[[:alnum:][:space:]+=\/-]+$')

---JSON dokumentidega PostgreSQL andmebaasides---
CONSTRAINT Person_nat_id_code_contains_valid_symbols
      CHECK ((CAST (data ->> 'nat_id_code' AS VARCHAR)) ~*
'^[[:alnum:][:space:]+=\/-]+$')
```

Joonis 33. PostgreSQL andmebaasides isikukoodi kitsenduse jõustamise näide.

Joonisel 34 on esitatud MongoDB andmebaasides sama isikukoodi kitsenduse jõustamine.

```
nat_id_code: {
        bsonType: 'string',
        pattern: '^[[:alnum:][:space:]+=/-]+$',
        maxLength: 50,
        description: 'National identification code cannot be
empty. Max length 50 symbols, consists of valid symbols.'
      }
```

Joonis 34. MongoDB andmebaasides isikukoodi kitsenduse jõustamise näide.

MongoDB andmebaasides oli võimalik jõustada vähem kitsendusi. See tuleneb valideerimisel kasutatava *JSON Schema* (vt jaotis 3.3.2) piirangutest. Esiteks pole võimalik JSON skeemis valideerida kuupäevi/ajatempleid, sest selleks pole loodud niisugust vajalikku võimekust. Teiseks pole skeemis võimalik võrrelda omavahel kahes JSON väljas olevat väärtust, mis on vajalik näiteks kitsenduse *„Eesnimi või perenimi on registreeritud.“* jõustamiseks. Kolmandaks pole võimalik JSON väljale seada kahte eraldiseisvat väljakontrolli (nt *“Isikukoodis on lubatud tähed (lubatud on ka muud tähed kui ASCII tähed a-zA-Z), numbrid, tühikud, sidekriipsud, plussmärgid, võrdusmärgid ja kaldkriipsud.“* ja *„Isikukood pole tühi string ega tühimärkidest koosnev string.“*). Kahe

väljakontrolli jõustamiseks tuleb need kontrollid panna kokku üheks regulaaravaldiseks, mis sõltuvalt kokkupandavatest kontrollidest pole mõningatel juhtudel võimalik või mille tulemusena luuakse keerukas ja raskesti arusaadav/muudetav regulaaravaldis. Mõistlikum oli sellisel juhul teha andmebaasis keerukama kitsenduse täidetuse kontrollimine (nt isikukoodi korrektsetest märkidest koosnemine) ning lihtsam kontroll teha rakenduses (nt isikukood ei tohi olla tühi string ja ainult tühimärkidest koosnev string).

5.2.7 Kokkuvõte

Tabelis 6 on kokkuvõte Lisas 8 esitatud andmebaasides jõustatud kitsendustest.

Tabel 6. Andmebaasides jõustatud kitsenduste kokkuvõte.

Kitsenduse tüüp / Andmebaasi disain	**Trad. Postgre-SQL**	**Hier. PostgreSQL JSON-iga**	**Mitte-hier. PostgreSQL JSON-iga**	**Hier. MongoDB**	**Mitte-hier. MongoDB**
Andmetüüp	22/22	18/22	18/22	22/22	22/22
Väljapikkus	12/12	12/12	12/12	12/12	12/12
Kohustuslikkus	19/19	14/17	19/19	17/17	19/19
Unikaalsus	12/12	11/12	12/12	11/12	12/12
Viited	7/7	2/5	7/7	0/5	0/7
Lisapiirangud	27/27	21/27	21/27	17/27	17/27
Andmebaasis jõustatud kontrollide arv	**99/99**	**78/95**	**89/99**	**79/95**	**82/99**
Rakenduses realiseeritud kontrollide arv	**0/99**	**17/95**	**10/99**	**16/95**	**17/99**

Hierarhiliste JSON dokumentidega PostgreSQL ja MongoDB disainide korral on kitsendusi neli tükki vähem kui ülejäänud disainide korral, sest hierarhia korral on töötaja isikuks olemine alati süsteemis registreeritud ning samuti on ametis olemisel alati olemas töötaja identifikaator töötaja dokumendis sisaldumise tõttu. Seega nendele väljadele puudub kohustuslikkuse määramise ning viidete terviklikkuse reegli jõustamise nõue (vt Joonis 10 ja 13).

5.3 Kompenseerivad tegevused

Andmebaasis kitsendusega seostuv kompenseeriv tegevus tagab, et kui andmebaasis tehakse andmemuudatus, mis pole kitsendusega kooskõlas, siis muudatuse tagasilükkamise asemel teeb andmebaasisüsteem täiendava andmemuudatuse, et lõpptulemus oleks kitsendustega kooskõlas ja muudatus saaks õnnestuda. SQL-andmebaasides on võimalik kompenseerivaid tegevusi (nt *ON DELETE CASCADE, ON UPDATE CASCADE, ON DELETE SET NULL*) defineerida välisvõtme kitsenduste juures. Lisas 9 kirjeldatakse kontseptuaalsest andmemudelist tulenevad kompenseerivad tegevused ning nende jõustamise võimalikkuse igas andmebaasi disainis.

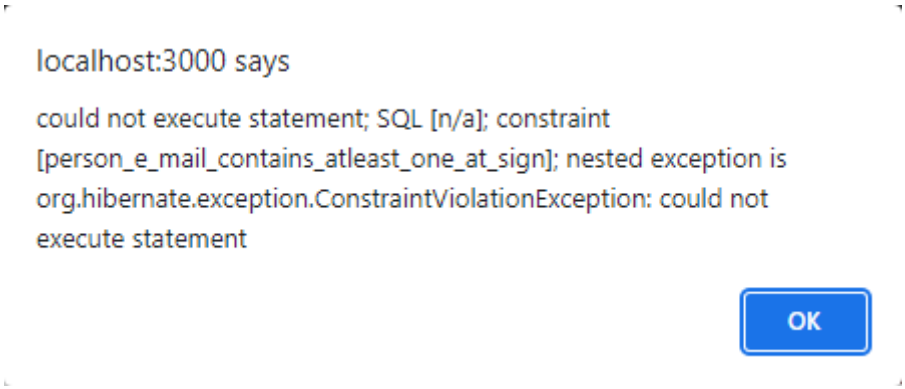
5.4 Disainide hindamine

Selles jaotises esitatakse hinnangud erinevatele uuritud disainidele.

5.4.1 PostgreSQL traditsioonilise disainiga andmebaas

- Andmebaasi skeemi füüsiliste koodiridade arv – **93**
- Rakenduse füüsiliste koodiridade arv – **861**
- Kontrollide arv, mida õnnestus realiseerida deklaratiivselt andmebaasi tasemel – **99/99**
- Kontrollide arv, mis tuli realiseerida rakenduse tasemel – **0/99**
- Andmebaasis realiseeritud kompenseerivate tegevuste arv – **7/7**
- Subjektiivne hinnang andmekontrollide andmebaasi tasemel jõustamise lihtsusele – **10**
    - Andmekontrolle oli lihtne ning selge jõustada.
    - Oli võimalik jõustada kontrolle, mis hõlmasid mitut erinevat välja.
- Subjektiivne hinnang kasutajale väljastatud veateadete kvaliteedile – **8**

- Väljastatud veateates pole kirjas, mis on valideerimisreegli sisu, mille vastu eksiti. Veast aru saamiseks tuleb võimalikult täpselt kirjeldada reegli sisu reegli nimes (vt Joonis 35).



Joonis 35. PostgreSQL veateate näidis.

- Vigade olemasolul tagastatakse vaid üks veateade, mis lihtsustab vea leidmist ning parandamist.
- Veateates pole näha, mis olid kasutaja poolt sisestatud mittesobivad andmed.

- Subjektiivne hinnang andmebaasi skeemi ja rakenduse muutmise lihtsusele – **8**
  - Skeemi muutmise jõustamine võib vajada lisatööd eelnevalt salvestatud andmetes.
- Subjektiivne hinnang üldisele andmebaasi ja rakenduse loomise lihtsusele – **9**
  - Kõiki andmekontrolle oli võimalik jõustada andmebaasis, mis vähendas rakenduse koodiridade arvu ning lihtsustas rakenduse loomist.

5.4.2 PostgreSQL hierarhilisi JSON dokumente sisaldav andmebaas

- Andmebaasi skeemi füüsiliste koodiridade arv – **73**
- Rakenduse füüsiliste koodiridade arv – **1141**
- Kontrollide arv, mida õnnestus realiseerida deklaratiivselt andmebaasi tasemel – **78/95**

- Kontrollide arv, mis tuli realiseerida rakenduse tasemel – **17/95**
- Andmebaasis realiseeritud kompenseerivate tegevuste arv – **4/7**
- Subjektiivne hinnang andmekontrollide andmebaasi tasemel jõustamise lihtsusele – **7**
    - Andmekontrollid on võimalik teha sarnaselt nagu PostgreSQL traditsioonilise disainiga andmebaasi korral, kuid JSONB väljade kontrollimiseks tuleb vastavas väljas olev väärtus enne kontrollimist ümber teisendada (*cast*) PostgreSQL-ile sobivat tüüpi väärtuseks. Teisendatava väärtuse esitus dokumendis peab süntaktiliselt olema teisendamiseks sobilik, vastasel juhul hakkab andmebaasisüsteem tagastama selle kohta veateateid ning andmeid pole võimalik salvestada (vt jaotis 3.6.1).
        - Sellest tulenevalt pole võimalik kontrollida kuupäeva/ajatempli andmetüüpi ega väärtuse reeglitele vastavust.
    - JSON dokumendis olevaid viiteid pole viidete terviklikkuse tagamiseks võimalik skeemi tasemel defineerida.
- Subjektiivne hinnang kasutajale väljastatud veateadete kvaliteedile – **8**
    - Väljastatud veateates pole kirjas, mis on valideerimisreegli sisu, mille vastu eksiti. Veast aru saamiseks tuleb võimalikult täpselt kirjeldada reegli sisu reegli nimes (vt Joonis 35).
    - Vigade olemasolul tagastatakse vaid üks veateade, mis lihtsustab vea leidmist ning parandamist (vt Joonis 35).
    - Veateates pole näha, mis olid kasutaja poolt sisestatud mittesobivad andmed.
- Subjektiivne hinnang andmebaasi skeemi ja rakenduse muutmise lihtsusele – **7**
    - Skeemi muutmise jõustamine võib vajada lisatööd eelnevalt salvestatud andmetes.

    - Kitsenduste jõustamine vajab JSON tüüpi väärtuses olevate väljade puhul teisendamist, kus peavad süntaktiliselt sobivad olema nii teisendatavas väljas olev väärtuse esitus kui ka teisendamise käsk (vt jaotis 3.6.1).

- Subjektiivne hinnang üldisele andmebaasi ja rakenduse loomise lihtsusele – **7**

    - Enamik andmete lisapiirangute kontrollidest oli võimalik jõustada andmebaasis, mis vähendas rakenduse koodiridade arvu ning lihtsustas rakenduse loomist.

    - JSON dokumentidele kitsenduste loomine on keerukama süntaksiga ja subjektiivsel hinnangul keerulisem/veaohtlikum, kui traditsioonilise disainiga tabelitele kitsenduste loomine (vt jaotis 3.6.1).

    - Kui on tarvis muuta *Töötaja* või *Ametis_töötamise* andmeid, siis tegelikult muudetakse *Isiku* andmeid, kuna JSON objektis on isiku alla salvestatud tema töötajaks oleku ja ametis töötamise andmed. Seega ametis töötamise andmete muutmisel ei tohi rakendus muuta isiku ja töötaja andmeid. Teiste sõnadega tähendab see, et andmete muutmiseks tuleb rakendus luua nii, et uute andmete salvestamisega säiliks eelnevalt salvestatud, muutmist mitte vajavate olemite andmed (vt Joonis 10).

5.4.3 PostgreSQL mitte-hierarhilisi JSON dokumente sisaldava andmebaas

- Andmebaasi skeemi füüsiliste koodiridade arv – **95**
- Rakenduse füüsiliste koodiridade arv – **1104**
- Kontrollide arv, mida õnnestus realiseerida andmebaasi tasemel – **89/99**
- Kontrollide arv, mis tuli realiseerida rakenduse tasemel – **10/99**
- Andmebaasis realiseeritud kompenseerivate tegevuste arv – **7/7**
- Subjektiivne hinnang andmekontrollide andmebaasi tasemel jõustamise lihtsusele – **8**

    - Andmekontrollid on võimalik teha sarnaselt nagu PostgreSQL traditsioonilise disainiga andmebaasi korral, kuid JSONB väljade

kontrollimiseks tuleb vastavas väljas olev väärtus enne kontrollimist ümber teisendada (*cast*) PostgreSQL-ile sobivat tüüpi väärtuseks. Teisendatava väärtuse esitus dokumendis peab süntaktiliselt olema teisendamiseks sobilik, vastasel juhul hakkab andmebaasisüsteem tagastama selle kohta veateateid ning andmeid pole võimalik salvestada (vt jaotis 3.6.1).

        - Sellest tulenevalt pole võimalik kontrollida kuupäeva/ajatempli andmetüüpi ega väärtuste reeglitele vastavust.

- Subjektiivne hinnang kasutajale väljastatud veateadete kvaliteedile – **8**

    - Väljastatud veateates pole kirjas, mis on valideerimisreegli sisu, mille vastu eksiti. Veast aru saamiseks tuleb võimalikult täpselt kirjeldada reegli sisu reegli nimes (vt Joonis 35).

    - Vigade olemasolul tagastatakse vaid üks veateade, mis lihtsustab vea leidmist ning parandamist (vt Joonis 35).

    - Veateates pole näha, mis olid kasutaja poolt sisestatud mittesobivad andmed.

- Subjektiivne hinnang andmebaasi skeemi ja rakenduse muutmise lihtsusele – **7**

    - Skeemi muutmise jõustamine võib vajada lisatööd eelnevalt salvestatud andmetes.

    - Kitsenduste jõustamine vajab JSON tüüpi väärtuses olevate väljade puhul teisendamist, kus peavad süntaktiliselt sobivad olema nii teisendatavas väljas olev väärtuse esitus kui ka teisendamise käsk (vt jaotist 3.6.1).

- Subjektiivne hinnang üldisele andmebaasi ja rakenduse loomise lihtsusele – **8**

    - Enamik andmekontrolle oli võimalik jõustada andmebaasis, mis vähendas rakenduse koodiridade arvu ning lihtsustas rakenduse loomist.

- JSON dokumentidele kitsenduste loomine on keerukama süntaksiga ja subjektiivsel hinnangul keerulisem/veaohtlikum, kui traditsioonilise disainiga tabelitele kitsenduste loomine (vt jaotis 3.6.1).

5.4.4 MongoDB hierarhilisi dokumente sisaldava andmebaasi hinnang

- Andmebaasi skeemi füüsiliste koodiridade arv – **259**
- Rakenduse füüsiliste koodiridade arv – **1347**
- Kontrollide arv, mida õnnestus realiseerida deklaratiivselt andmebaasi tasemel – **79/95**
- Kontrollide arv, mis tuli realiseerida andmebaasi tasemel – **16/95**
- Andmebaasis realiseeritud kompenseerivate tegevuste arv – **1/7**
- Subjektiivne hinnang andmekontrollide andmebaasi tasemel jõustamise lihtsusele **– 6**
    - Andmekontrollide sisu on rohkem piiratud kui PostgreSQL-is tehtavatel kontrollidel, mille pärast on rohkem kontrolle vaja luua rakenduses.
    - Lisapiirangute seadmisi andmetele andmebaasis saab luua märgatavalt vähem võrreldes PostgreSQL andmebaasidega.
    - Skeemi abil pole võimalik jõustada viidete terviklikkust.
- Subjektiivne hinnang kasutajale väljastatud veateate kvaliteedile – **7**
    - Väljastatud veateates on kirjas, mis on valideerimisreegli sisu, mille vastu eksiti, mille tõttu on kergem aru saada, miks andmed pole korrektsed. Samas sisaldab veateade ka palju muud informatsiooni, mille pärast võib kasutajale olla väljastatud veateade raskesti mõistetav, kui kasutaja ei tea, mida ta peaks veateatest vea parandamiseks otsima (vt Joonis 36).

Joonis 36. MongoDB veateate näidis.

- Vigade olemasolul tagastatakse veateated kõikide vigade kohta, mis esitatud andmetest esinevad. Kogenud rakenduse kasutajale võib niisugune mitmest veateatest koosnev üks veateade kiirendada andmete parandamist, kuid see eeldab, et kasutaja on veateadete sisu mõistmises piisavalt kogenud (vt Joonis 36).
- Veateates on näha, mis olid kasutaja poolt sisestatud mittesobivad andmed.

- Subjektiivne hinnang andmebaasi skeemi ja rakenduse muutmise lihtsusele – **10**
  - Skeemi saab muuta nii, et olemasolevad andmed ei vaja skeemimuudatuste jõustamiseks muudatusi. Küll aga peavad peale skeemimuudatuste rakendamist lisanduvad uued andmed olema skeemi suhtes korrektsed.

- Subjektiivne hinnang üldisele andmebaasi ja rakenduse loomise lihtsusele – **6**
  - Andmebaasis andmemuudatuste tegemiseks tuleb rakenduse poolel koodis määrata muudatuste läbiviimine *set*/*unset* käskudega (vt Joonis 37), mis muudab rakenduse loomist keerukamaks. *Set* käsuga määratakse olemi andmeid esitava JSON objekti väljale uus väärtus ning *unset* käsuga kustutatakse väärtus objektist.

```
//set firstname to null if not specified
if (embeddedPerson.getGiven_name() != null &&
embeddedPerson.getGiven_name().length() != 0) {
    updatableInfo.set("given_name", embeddedPerson.getGiven_name());
} else {
    updatableInfo.unset("given_name");
}
```

Joonis 37. Näide *set/unset* kasutamisest.

- Võrreldes PostgreSQL andmebaaside disainidega vajab käesolev andmebaasi disain märgatavalt rohkem andmekontrolle rakenduses, eriti lisapiirangute korral.

- Kui on tarvis muuta *Töötaja* või *Ametis_töötamise* andmeid, siis tegelikult muudetakse *Isiku* andmeid, kuna JSON objektis on isiku alla salvestatud tema töötajaks oleku ja ametis töötamise andmed. Seega ametis töötamise andmete muutmisel ei tohi rakendus muuta isiku ja töötaja andmeid. Teiste sõnadega tähendab see, et andmete muutmiseks tuleb rakendus luua nii, et uute andmete salvestamisega ei muudeta ning säiliks eelnevalt salvestatud, muutmist mitte vajavate olemite andmed (vt Joonis 13).

5.4.5 MongoDB mitte-hierarhilisi dokumente sisaldava andmebaasi hinnang

- Andmebaasi skeemi füüsiliste koodiridade arv – **287**
- Rakenduse füüsiliste koodiridade arv – **1310**
- Kontrollide arv, mida õnnestus realiseerida andmebaasi tasemel – **82/99**
- Kontrollide arv, mis tuli realiseerida andmebaasi tasemel – **17/99**
- Andmebaasis realiseeritud kompenseerivate tegevuste arv – **0/7**
- Subjektiivne hinnang andmekontrollide andmebaasi tasemel jõustamise lihtsusele – **6**

    - Andmekontrollide sisu on rohkem piiratud kui PostgreSQL-is tehtavatel kontrollidel, mille pärast on rohkem kontrolle vaja luua rakenduses.

- Lisapiirangute seadmisi andmetele andmebaasis saab luua märgatavalt vähem võrreldes PostgreSQL andmebaasidega.
- Skeemi abil pole võimalik jõustada viidete terviklikkust.

- Subjektiivne hinnang kasutajale väljastatud veateate kvaliteedile – **7**
  - Väljastatud veateates on kirjas, mis on valideerimisreegli sisu, mille vastu eksiti, mille tõttu on kergem aru saada, miks andmed pole korrektsed. Samas sisaldab veateade ka palju muud informatsiooni, mille pärast võib kasutajale olla väljastatud veateade raskesti mõistetav, kui kasutaja ei tea, mida ta peaks veateatest vea parandamiseks otsima (vt Joonis 36).
  - Vigade olemasolul tagastatakse veateated kõikide vigade kohta, mis esitatud andmetest esinevad. Kogenud rakenduse kasutajale võib niisugune mitmest veateatest koosnev üks veateade kiirendada andmete parandamist, kuid see eeldab, et kasutaja on piisavalt kogenud veateadete sisu mõistmises (vt Joonis 36).
  - Veateates on näha, mis olid kasutaja poolt sisestatud mittesobivad andmed.

- Subjektiivne hinnang andmebaasi skeemi ja rakenduse muutmise lihtsusele – **10**
  - Skeemi saab muuta nii, et olemasolevad andmed ei vaja skeemimuudatuste jõustamiseks muudatusi. Küll aga peavad uued andmed peale skeemimuudatuste rakendamist olema skeemi suhtes korrektsed.

- Subjektiivne hinnang üldisele andmebaasi ja rakenduse loomise lihtsusele – **7**
  - Andmebaasis andmemuudatuste tegemiseks tuleb rakenduse poolel koodis määrata muudatuste läbiviimine *set*/*unset* käskudega (vt Joonis 37), mis muudab rakenduse loomist keerukamaks. *Set* käsuga määratakse olemi andmeid esitava JSON objekti väljale uus väärtus ning *unset* käsuga kustutatakse väärtus objektist.

- Võrreldes PostgreSQL andmebaaside disainidega vajab käesolev andmebaasi disain märgatavalt rohkem andmekontrolle rakenduses, eriti lisapiirangute korral.

5.5 Erinevate disainide võrdlused koondtabelitena

Tabelis 7 on esitatud jaotiste 5.4.1 kuni 5.4.5 hinnangute arvulise väärtused. Iga aspekti kohta tuuakse **rasvase tumeda** fondiga välja parimad tulemused ja **rasvase ning punase** fondiga halvimad tulemused.

Tabel 7. Erinevate disainide hinnangute koondtabel.

Andmebaasi disaini nimetus	Trad. PostgreSQL	Hier. PostgreSQL JSON-iga	Mitte-hier. PostgreSQL JSON-iga	Hier. MongoDB	Mitte-hier. MongoDB
Hinnang andmebaasi ja rakenduse loomise lihtsusele	**9**	7	8	**6**	7
Hinnang andmebaasi ja rakenduse muutmise lihtsusele	8	**7**	**7**	**10**	**10**
Hinnang andmebaasi tasemel väljastatud veateadete kvaliteedile	**8**	**8**	**8**	**7**	**7**
Hinnang andmebaasis andmete valideerimise lihtsusele	**10**	7	8	**6**	**6**
Andmebaasis jõustatud kompenseerivate tegevuste arv	**7/7**	4/7	**7/7**	1/7	**0/7**

Andmebaasi disaini nimetus	Trad. PostgreSQL	Hier. PostgreSQL JSON-iga	Mitte-hier. PostgreSQL JSON-iga	Hier. MongoDB	Mitte-hier. MongoDB
Kontrollide arv rakenduse tasemel	**0/99**	**17/95**	10/99	16/95	17/99
Kontrollide arv andmebaasi tasemel	**99/99**	**78/95**	89/99	79/95	82/99
Rakenduse koodi-ridade arv	**861**	1141	1104	**1347**	1310
Andmebaasi skeemi koodiridade arv	93	**73**	95	259	**287**

6 Eksperimendi tulemuste analüüs

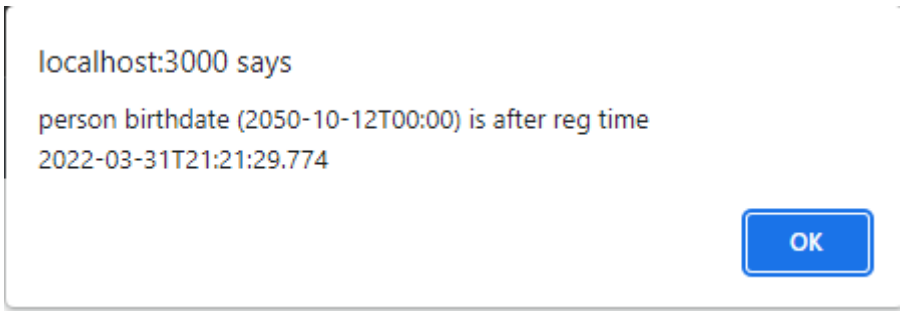
Eksperimendis läbiviidud rakenduste loomise põhjal saab väita, et rakenduse loomise lihtsus sõltub andmebaasis andmekontrollide jõustamise võimalustest. Mida rohkem kontrolle on võimalik teha andmebaasis, seda lihtsam on andmebaasirakendust luua ning hallata. PostgreSQL andmebaasides on võimalik luua märgatavalt rohkem validatsioonireegleid kui MongoDB andmebaasides. Kasutajaliidese kasutajasõbralikkust hinnates on PostgreSQL-i poolt väljastatud veateated paremad kui MongoDB poolt väljastatud. Põhjuseks on, et PostgreSQL veateates tagastatakse andmemuudatusega kooskõlas mitte oleva kitsenduse nimi, mis on enamasti paremini mõistetav, kui MongoDB veateates märgitud valideerimisreegli sisu koos muu taustainformatsiooniga. PostgreSQL-i paremini mõistetav veateade eeldab, et kitsenduse nimi kirjeldab sõnaliselt võimalikult täpselt kitsenduse sisu, mis aitaks kasutajal aru saada, miks tema poolt sisestatud andmed pole sobivad ning millise sisuga andmeid on temalt oodatud.

Andmebaasi skeemimuudatusi on pigem kergem teha MongoDB andmebaasides, sest sõltuvalt tehtavatest muudatustest võivad PostgreSQL-is olevad andmed vajada eelnevat lisatööd uuendatud skeemi kasutusele võtmiseks. MongoDB-s saab uue skeemi võtta kasutusele koheselt, olenemata sellest, et juba salvestatud andmed ei pruugi uuele skeemile vastata. Eelnevalt salvestatud andmed ei vaja enne uue skeemi rakendamist muudatusi, mis tähendab, et skeemimuudatusi on võimalik teha kergemini ning kiiremini. Andmed peavad vastama uutele reeglitele alles siis, kui neid soovitakse andmebaasis muuta. Seega on skeemi muutmiseks tarvis teha vähem eelnevaid andmemuudatusi. PostgreSQL-is saab luua trigerite abil protseduurseid andmekontrolle. Trigeri loomise järel hakatakse kontrollima uusi andmeid, kuid trigeri loomisel ei kontrollita olemasolevaid andmeid trigeriga jõustatud piirangu suhtes. Samuti saab *ALTER TABLE* lausega lisada (PostgreSQL 14 korral) tabelile *FOREIGN KEY* või *CHECK* kitsenduse, mis on seisundis *NOT VALID*. Sellise kitsenduse korral ei kontrollita samuti olemasolevaid andmeid kitsenduse suhtes, vaid ainult uusi.

Hierarhiliste või mitte-hierarhiliste dokumentide kasutusest sõltus rakenduse loomise lihtsus. Mitte-hierarhiliste dokumentide kasutusel oli lihtsam ning kiirem muuta üksikute olemite andmeid, sest selle muutmiseks ei pidanud eelnevalt töötlema seda sisaldava

olemi andmeid. Hierarhiliste dokumentide korral tuli kohati uuesti salvestada kõik muudetud olemi andmeid sisaldanud vanemolemi andmed, mille tõttu vajas andmete muutmine rakenduses rohkem tähelepanu. Küll aga oli hierarhiliste dokumentide disainides märgatavalt lihtsam olemi andmete kustutamine, mille näiteks on kasutusmall "Kustuta töötaja". Mitte-hierarhiliste JSON dokumentidega MongoDB andmebaasi disaini korral tuli töötaja kustutamisel hakata otsima ning kustutama temaga seotud töötamisi teistest kollektsioonidest, samas kui hierarhiliste disainide (nii MongoDB kui PostgreSQL andmebaaside) korral oli vaid vaja kustutada olemi *Isik* atribuudi *töötaja* väärtus, milles sisaldusid kõik temaga seotud töötamised. Teiste sõnadega, taolise andmete kustutamise vajaduse korral on hierarhiliste dokumentidega andmebaasi disain kasulikum kui mitte-hierarhiliste dokumentidega disain nii andmebaasi ressursikasutuselt kui ka rakenduse vaatest, sest andmebaasis ei pea tegema lisaoperatsioone ning rakenduses pole tarvis luua mingisugust lisaloogikat. Samuti on rakenduse disain sellisel puhul lihtne traditsioonilise disainiga tabelite puhul, sest tänu välisvõtmes deklareeritud *ON DELETE CASCADE* kompenseerivale tegevusele kustutatakse töötaja kustutamisel tema töötamised automaatselt.

Andmebaasist jõustatud kitsenduste vastu eksimisest tingitud veateated on kasutajakogemuse seisukohast mõlema andmebaasisüsteemi korral märgatavalt kehvemad kui rakenduses loodud kontrollide korral. Andmebaasisüsteemi veateate sisu on kogenematule kasutajale raskesti mõistetav ning reeglitele mittevastavate andmete parandamiseks vajab see rohkem analüüsimist kui rakenduse poolt tagastatud veateadete korral. Rakenduse veateated (vt Joonis 38) on võimalik luua dünaamiliselt, ehk veateate sisu on võimalik luua sõltuvalt kasutaja sisestatud andmetest, tänu millele on need veateated kõikidele kasutajatele kergelt mõistetavad.



Joonis 38. Rakenduse veateate näide.

Kuna kasutaja näeb andmebaasisüsteemi veateateid, siis veateate täpsuse huvides on see, et andmebaasis jõustatakse iga kitsendus eraldi, mitte ei looda ühte kitsendust, mis kontrollib korraga mitme reegli täidetust ja tagastab selle vastu eksimisel siis liiga üldise kitsenduse nimega veateate.

Andmebaasisüsteemi veateadete parandamiseks on teoorias võimalik luua veateate sisu parser, mis filtreeriks kasutajale väljastatavat sisu, kuid autori hinnangul pole niisugune lahendus jätkusuutlik, ei tõstaks märkimisväärselt veateate sisu kvaliteeti ning pole parseri loomise ja haldamise kulusid arvestades seda väärt. Kui esmatähtis on kasutajakogemus, siis tuleks eelistada rakenduses andmete valideerimist ja veateadete tagastamist.

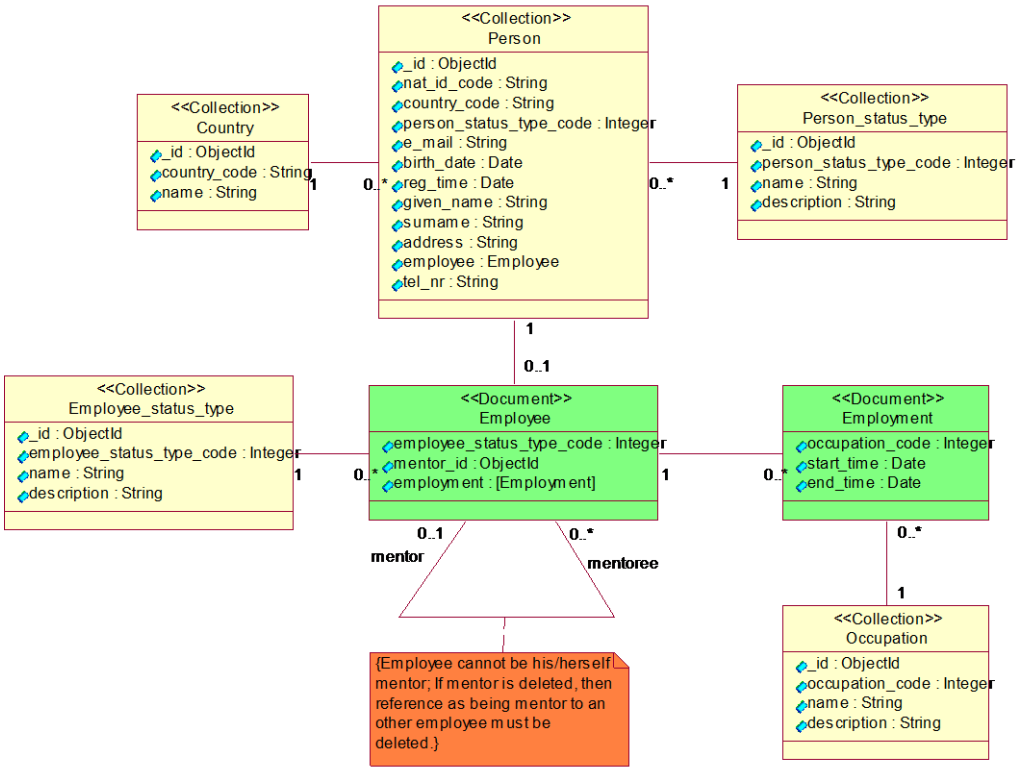
Käesolevas töös jõustati andmebaasis kitsendused teadlikult deklaratiivselt, defineerides erinevat tüüpi kitsendusi. Alternatiiviks oleks olnud luua andmebaasi trigereid, mis on andmemuudatuse sündmustele reageerivad protseduurid, mille abil saab samuti kontrollida andmete reeglitele vastavust ja mis saaksid väljastada ilusamaid veateateid. Ilusamad veateated kõrvale jättes on trigerite kood keerukam kui deklaratiivsetel kitsendustel, seda koodi on raske siluda ja testida ning lihtne on teha vigu, mis annavad andmete kontrollimise osas võlts-turvatunde, sest teatud andmete muutmise stsenaariumid võimaldavad siiski registreerida reeglitele mittevastavaid andmeid. Trigerid võivad ka sisaldada automaagilisi, kasutajale teadmatuid tegevusi, mis võivad aeglustada andmemuudatuste tegemist [53]–[55].

6.1 Soovitused JSON skeemi modelleerimiseks UML klassiskeemina

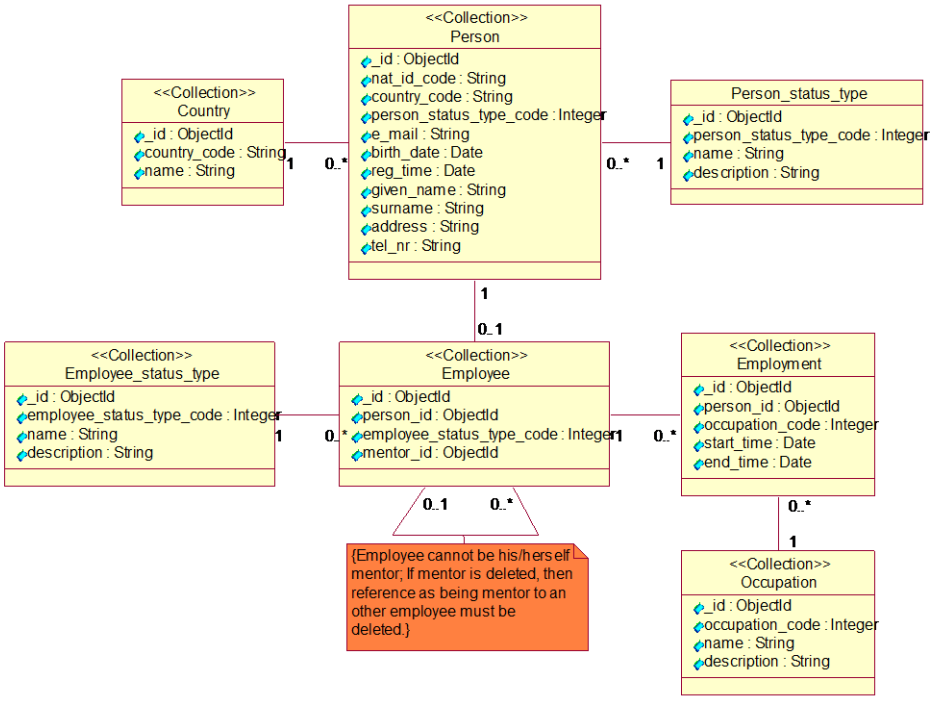
UML on üldotstarbeline modelleerimiskeel, mis tähendab, et seda saab kasutada paljudes erinevates valdkondades. Seal pole spetsiaalset skeemi tüüpi JSON dokumentide skeemi modelleerimiseks. Kuid nagu SQL-andmebaasi tabeleid, saab neid modelleerida klassiskeemide baasil. Nagu on näha Joonisel 10, siis on võimalik kirjeldada oodatavat JSON dokumendi struktuuri füüsilise disaini mudelites kasutades klassiskeemil märkuseid (*note*). Märkuse sisu võiks selguse ning loetavuse nimel olla võimalikult sarnane JSON struktuurile. Erinevat tüüpi JSON dokumentide eristamiseks võiks kasutada märkustel erinevaid taustavärve. Näiteks käesolevas töös on mitte-hierarhilised JSON dokumendid halli taustaga märkustel, hierarhilised JSON dokumendid rohelise taustaga märkustel.

Samuti võiks kõik, mida modelleerimisvahend ei võimalda automaatselt genereerida, panna skeemile kirja märkustena. Näiteks on PostgreSQL-is võimalik luua JSON tüüpi veerule indeks ning vastavad käsud võiks esitada skeemil samuti märkustega, millel on teistest märkustest erinev taustavärv. Näiteks on käesolevas töös unikaalseid indekseid loovad käsud esitatud erkkollase taustavärviga.

Käesolevas töös modelleeriti MongoDB andmebaasi struktuuri Moon Modeler-i abil. Autor koostas need mudelid ka Rational Rose-is UML-i klassiskeemide põhjal ning need on esitatud joonistel 39 ja 40. Võrreldes Moon Modeler-i mudelitega, on Rational Rose-is UML-i klassiskeemidelt raskem aru saada hierarhiliste dokumentide kasutamisest ja välisvõtmetest. Samuti pole võimalik eristada kohustuslikke atribuute (*NOT NULL*) mittekohustuslikest.



Joonis 39. Hierarhiliste dokumentidega MongoDB andmebaasi struktuur UML klassiskeemina.



Joonis 40. Mitte-hierarhiliste dokumentidega MongoDB andmebaasi struktuur UML klassiskeemina.

7 Kokkuvõte

Töö eesmärgiks oli uurida kirjutamise skeemi defineerimise ja JSON dokumentide kasutamise mõju andmebaasirakenduse loomisele erinevate PostgreSQL (ver 14) ja MongoDB (ver 5) andmebaasi disainide korral, kus mõju hinnati selle kaudu, kui lihtne on andmebaasi tasemel kontrollida andmete reeglitele vastavust, kuidas reeglite vastu eksimisest kasutajatele teada antakse ning kui lihtne on teha rakenduse funktsionaalsete nõuete muutmisest tulenevaid skeemimuudatusi. Töö tulemusena valminud andmebaasi ja rakenduse lähtekood on avaldatud avalikul GitHubi lehel [1].

Lõputöö raames sooritatud eksperimendi tulemusena jõuti järgnevatele järeldustele.

- Andmebaasirakendust oli lihtsaim luua ilma JSON dokumentideta disainis, sest kõik andmekontrollid sai teha andmebaasis.
- Skeemimuudatusi on lihtsam teha MongoDB andmebaasides.
- PostgreSQL võimaldab keerukamate valideerimisreeglite loomist kui MongoDB.
- PostgreSQL veateated on kasutajasõbralikkuse vaatest paremad kui MongoDB poolt väljastatud veateated.
- Hierarhiliste ning mitte-hierarhiliste dokumentide kasutamine mõjutab rakenduste lihtsust erinevates aspektidest. Hierarhiliste või mitte-hierarhiliste dokumentide kasuks otsustamisel tuleb lähtuda rakenduse funktsionaalsust kirjeldavatest kasutusmallidest.

Lõputöös läbiviidud eksperimendi käigus loodud andmebaaside ning rakenduste edasiarenduseks pakub autor välja PostgreSQL-ile loodud JSON tüüpi väärtuste valideerimislaienduse kasutamist, tagarakendustes üksustestide kirjutamist, andmebaasides realiseeritavate vaadete loomist ning rohkemate kasutusmallide kirjeldamist ja realiseerimist, mille tulemusena tuleks rakendustes realiseerida uut äriloogikat. Samuti oleks huvitav testida andmebaasisüsteemi poolt väljastatavate veateadete kasutatavust reaalsete rakenduse lõppkasutajate peal. Veel ühe töösuunana võiks suurendada andmebaasides salvestatavate andmete hulka, et saaks hinnata operatsioonide töökiirusi eri andmebaaside disainide puhul. Käesoleva töö käigus loodud

andmebaasides on testandmete hulk liiga väike selleks, et tekiks märgatavaid töökiiruste erinevusi.

Kasutatud kirjandus

[1] Töö tulemusena valminud andmebaaside ja rakenduse lähtekood (*the source code for the databases and application*): https://github.com/reroop/postgresql-and-mongodb-db-application

[2] „Schemaless Data Structures," [Online]. Accessed on: 16.02.2022. Available: https://martinfowler.com/articles/schemaless/

[3] Bailis, P., Fekete, A., Franklin, M. J., Ghodsi, A., Hellerstein, J. M., Stoica, I., 2015. Feral concurrency control: An empirical investigation of modern application integrity. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. ACM. PP. 1327-1342 [WWW] http://www.bailis.org/papers/feral-sigmod2015.pdf

[4] „What is table elimination?," [Online]. Accessed on: 16.02.2022. Available: https://mariadb.com/kb/en/what-is-table-elimination/

[5] „DB-Engines ranking," [Online]. Accessed on: 15.02.2022. Available: https://db-engines.com/en/ranking

[6] „PostgreSQL: Documentation: 14: 8.14. JSON Types," [Online]. Accessed on 14.04.2022. Available: https://www.postgresql.org/docs/14/datatype-json.html

[7] „JSON Functions and Operators," [Online]. Accessed on: 15.02.2022. Available: https://www.postgresql.org/docs/current/functions-json.html

[8] „Schema Validation," [Online]. Accessed on 16.02.2022. Available: https://docs.mongodb.com/manual/core/schema-validation/

[9] Maksimov, D, 2015. Performance Comparsion on MongoDB and PostgreSQL with JSON types. PP. 43-54 [WWW] https://digikogu.taltech.ee/et/Item/6adcc6d9-d27c-47d6-be5cec9e696ba0e3

[10] Saar, G, 2021. Relatsioonilise ja mitterelatsioonilise andmebaasi jõudluse ristanalüüs relatsioonilise ja mitterelatsioonilise andmemudeliga PostgreSQL ja MongoDB näitel. PP. 28- 31, 33-34 [WWW] https://digikogu.taltech.ee/et/Item/6adcc6d9-d27c-47d6-be5cec9e696ba0e3

[11] Shpychka, V, 2017. Analysis on performance and complexity of building a web application based on Couchbase and PostgreSQL with JSONB type. PP. 60-64 [WWW] https://digikogu.taltech.ee/et/Item/1dc34253-95e8-40b4-8505-981517c0efea

[12] „Spring Boot Connect to PostgreSQL Database Examples," [Online]. Accessed on 17.02.2022. Available: https://www.codejava.net/frameworks/spring-boot/connect-topostgresql-database-examples

[13] „Build a CRUD App with Spring Boot and MongoDB," [Online]. Accessed on 17.02.2022. Available: https://www.split.io/blog/crud-spring-boot-mongodb

[14] „Rational Rose | DataONE,“ [Online]. Accessed on 30.04.2022. Available: https://old.dataone.org/software-tools/rational-rose-0

[15] „Moon Modeler – Draw ER Diagrams for your Data Models,“ [Online]. Accessed on 29.04.2022. Available: https://www.datensen.com/data-modeling/moon-modeler-for-databases.html

[16] „IntelliJ IDEA: The Capable & Ergonomic Java IDE by JetBrains,“ [Online]. Accessed on 29.04.2022. Available: https://www.jetbrains.com/idea/

[17] „pgAdmin – PostgreSQL Tools,“ [Online]. Accessed on 29.04.2022. Available: https://www.pgadmin.org/

[18] „MongoDB Compass | MongoDB,“ [Online]. Accessed on 29.04.2022. Available: https://www.mongodb.com/products/compass

[19] „Spring Boot,“ [Online]. Accessed on 29.04.2022. Available: https://spring.io/projects/spring-boot

[20] „TypeScript: JavaScript With Syntax for Types,“ [Online]. Accessed on 29.04.2022. Available: https://www.typescriptlang.org/

[21] „React – A JavaScript library for building user interfaces,“ [Online]. Accessed on 29.04.2022. Available: https://reactjs.org/

[22] „PostgreSQL vs MySQL: Everything You Need to Know in 2022,“ [Online]. Accessed on 02.05.2022. Available: https://hackr.io/blog/postgresql-vs-mysql

[23] Chen, B. „PostgreSQL vs. MySQL: what you need to know,“ [Online]. Accessed on 02.05.2022. Available: https://www.fivetran.com/blog/postgresql-vs-mysql

[24] „11.5 The JSON Data Type,“ [Online]. Accessed on 25.05.2022. Available: https://dev.mysql.com/doc/refman/8.0/en/json.html#json-paths

[25] „Partial update of JSON values,“ [Online]. Accessed on 29.05.2022. Available: https://dev.mysql.com/blog-archive/partial-update-of-json-values/

[26] „PostgreSQL jsonb_set performance,“ [Online]. Accessed on 29.05.2022. Available: https://dba.stackexchange.com/questions/284904/postgresql-jsonb-set-performance

[27] „Performance differences between Postgres and MySQL,“ [Online]. Accessed on 29.05.2022. Available: https://arctype.com/blog/performance-difference-between-postgresql-and-mysql/

[28] „“IntegrationDatabase,“ [Online]. Accessed on 28.05.2022. Available: https://martinfowler.com/bliki/IntegrationDatabase.html

[29] „Schema on Read,“ [Online]. Accessed on 12.04.2022. Available: https://www.techopedia.com/definition/30153/schema-on-read

[30] „Schema-on-Read vs Schema-on-Write,“ [Online]. Accessed on 12.04.2022. Available: https://www.marklogic.com/blog/schema-on-read-vs-schema-on-write/

[31] „Data Management: Schema-on-Write Vs. Schema-on-Read,“ [Online]. Accessed on 12.04.2022. Available: https://www.upsolver.com/blog/manage-your-data-schema-on-read-vs-schema-on-write

[32] „Schema-on-Read vs Schema-on-Write,“ [Online]. Accessed on 12.04.2022. Available: https://luminousmen.com/post/schema-on-read-vs-schema-on-write

[33] „UNDERSTANDING THE 3 VS OF BIG DATA – VOLUME, VELOCITY AND VARIETY,“ [Online]. Accessed on 03.05.2022. Available: https://www.coforge.com/salesforce/blog/data-analytics/understanding-the-3-vs-of-big-data-volume-velocity-and-variety/

[34] „ApplicationDatabase,“ [Online]. Accessed on 23.05.2022. Available: https://martinfowler.com/bliki/ApplicationDatabase.html

[35] „Document Database – NoSQL | MongoDB,“ [Online]. Accessed on 13.04.2022. Available: https://www.mongodb.com/document-databases

[36] „What Is NoSQL? NoSQL Databases Explained | MongoDB,“ [Online]. Accessed on 13.04.2022. Available: https://www.mongodb.com/nosql-explained

[37] „NoSQL database types explained: Document-based databases,“ [Online]. Accessed on 13.04.2022. Available: https://www.techtarget.com/searchdatamanagement/tip/NoSQL-database-types-explained-Document-based-databases

[38] „About Us – Our Story | MongoDB | MongoDB,“ [Online]. Accesssed on 13.04.2022. Available: https://www.mongodb.com/company

[39] „Overview of MongoDB,“ [Online]. Accessed on 13.04.2022. Available: https://www.w3schools.in/mongodb/overview

[40] „MongoDB vs SQL Databases: 4 Comprehensive Aspects,“ [Online]. Accesssed on 15.04.2022. Available: https://hevodata.com/learn/mongodb-vs-sql/

[41] „JSON And BSON | MongoDB,“ [Online]. Accessed on 13.04.2022. Available: https://www.mongodb.com/json-and-bson

[42] „Horizontal vs Vertical Scaling: What you need to know,“ [Online]. Accessed on 15.04.2022. Available: https://touchstonesecurity.com/horizontal-vs-vertical-scaling-what-you-need-to-know/

[43] „JSON Schema: core definitions and terminology draft-zyp-json-schema-04,“ [Online]. Accessed on 13.04.2022. Available: https://datatracker.ietf.org/doc/html/draft-zyp-json-schema-04

[44] Jan Michels, Keith Hare, Krishna Kulkarni, Calisto Zuzarte, Zhen Hua Liu, Beda Hammerschmidt, and Fred Zemke. 2018. The New and Improved SQL: 2016 Standard. SIGMOD Rec. 47, 2 (June 2018), 51–60. https://doi.org/10.1145/3299887.3299897

[45] Petković, D. SQL/JSON Standard: Properties and Deficiencies. Datenbank Spektrum 17, 277–287 (2017). https://doi.org/10.1007/s13222-017-0267-4

[46] „gavinwahl/postgres-json-schema,“ [Online]. Accessed on 16.02.2022. Available: https://github.com/gavinwahl/postgres-json-schema

[47] „furstenheim/is_jsonb_valid,“ [Online]. Accessed on 16.02.2022. Available: https://github.com/furstenheim/is_jsonb_valid

[48] Chisholm, M. (2000). Managing Reference Data in Enterprise Databases: Binding Corporate Data to the Wider World. Morgan Kaufmann.

[49] „Esterm. Eesti Keele Instituudi mitmekeelne terminibaas,“ [Online]. Accessed on 04.05.2022. Available: http://termin.eki.ee/esterm/

[50] „Riikide ja territooriumite klassifikaator 2021,“ [Online]. Accessed on 29.05.2022. Available: https://klassifikaatorid.stat.ee/Item/stat.ee/235d4628-954e-47c6-8642-3daedafd9d53/16

[51] „Use Not Valid To Immediately Enforce A Constraint,“ [Online]. Accessed on 28.04.2022. Available: https://til.hashrocket.com/posts/86467be0b3-use-not-valid-to-immediately-enforce-a-constraint

[52] „ObjectId,“ [Online]. Accessed on 06.05.2022. Available: https://www.mongodb.com/docs/manual/reference/method/ObjectId/

[53] Pollack. E, „SQL Server triggers: The good and the scary,“ [Online]. Accessed on 03.05.2022. Available: https://www.red-gate.com/simple-talk/databases/sql-server/database-administration-sql-server/sql-server-triggers-good-scary/

[54] Põlluste, A, 2021. Veebi- ja andmebaasipõhise metamodelleerimise vahendi üleviimine andmebaasi trigeritel põhinevaks süsteemiks. PP. 28- 30 [WWW] https://digikogu.taltech.ee/et/Item/1b4dbaa7-8d76-4565-b8d1-a9e760e9ec88

[55] „Triggers Considered Harmful, Considered Harmful,“ [Online]. Accessed on 03.05.2022. Available: http://harmfultriggers.blogspot.com/

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

Mina, Reiko Roopärg

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose „Andmebaasis kirjutamise skeemi ja JSON dokumentide kasutamise mõju PostgreSQL ja MongoDB andmebaasirakenduste loomisele“, mille juhendaja on Erki Eessaar
   1.1. reprodutseerimiseks lõputöö säilitamise ja elektroonse avaldamise eesmärgil, sh Tallinna Tehnikaülikooli raamatukogu digikogusse lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
   1.2. üldsusele kättesaadavaks tegemiseks Tallinna Tehnikaülikooli veebikeskkonna kaudu, sealhulgas Tallinna Tehnikaülikooli raamatukogu digikogu kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.
2. Olen teadlik, et käesoleva lihtlitsentsi punktis 1 nimetatud õigused jäävad alles ka autorile.
3. Kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest ning muudest õigusaktidest tulenevaid õigusi.

30.05.2022

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

Lisa 2 – “Traditsioonilise” PostgreSQL andmebaasi skeemi kood

```
CREATE TABLE Employment (
	person_id INTEGER NOT NULL,
	occupation_code SMALLINT NOT NULL,
	start_time TIMESTAMP NOT NULL,
	end_time TIMESTAMP,
	CONSTRAINT PK_Employment PRIMARY KEY (person_id, start_time,
occupation_code),
	CONSTRAINT Employment_start_time_between_01_01_2010_to_31_12_2100
CHECK (start_time >= '2010-01-01' AND start_time < '2101-01-01'),
	CONSTRAINT Employment_start_time_not_bigger_than_end_time CHECK
(start_time <= end_time),
	CONSTRAINT Employment_end_time_between_01_01_2010_to_31_12_2100 CHECK
(end_time >= '2010-01-01' AND end_time < '2101-01-01')
	);
CREATE INDEX IDX_Employment_person_id ON Employment (person_id );
CREATE TABLE Employee_status_type (
	employee_status_type_code SMALLINT NOT NULL,
	name VARCHAR ( 100 ) NOT NULL,
	description TEXT,
	CONSTRAINT PK_Employee_status_type PRIMARY KEY
(employee_status_type_code),
	CONSTRAINT AK_Employee_status_type_name UNIQUE (name),
	CONSTRAINT Employee_status_type_description_not_empty CHECK
(description !~ '^[[:space:]]+$' AND description <> ''),
	CONSTRAINT Employee_status_type_name_not_empty CHECK (name !~
'^[[:space:]]+$' AND name <> '')
	);
CREATE TABLE Person_status_type (
	person_status_type_code SMALLINT NOT NULL,
	name VARCHAR(100) NOT NULL,
	description TEXT,
	CONSTRAINT PK_Person_status_type PRIMARY KEY
(person_status_type_code),
	CONSTRAINT AK_Person_status_type_name UNIQUE (name),
	CONSTRAINT Person_status_type_description_not_empty CHECK (description
!~ '^[[:space:]]+$' AND description <> ''),
	CONSTRAINT Person_status_type_name_not_empty CHECK (name !~
'^[[:space:]]+$' AND name <> '')
);
CREATE TABLE Employee (
	person_id INTEGER NOT NULL,
	mentor_id INTEGER,
	employee_status_type_code SMALLINT NOT NULL,
	CONSTRAINT PK_Employee PRIMARY KEY (person_id),
	CONSTRAINT Employee_person_id_not_mentor_id CHECK (person_id !=
mentor_id)
	);
```

```
CREATE INDEX IDX_Employee_person_id ON Employee (person_id );
CREATE INDEX IDX_Employee_mentor_id ON Employee (mentor_id );
CREATE TABLE Person (
	_id SERIAL NOT NULL,
	nat_id_code VARCHAR ( 50 ) NOT NULL,
	country_code VARCHAR ( 3 ) NOT NULL,
	person_status_type_code SMALLINT NOT NULL,
	e_mail VARCHAR ( 254 ) NOT NULL,
	birth_date DATE NOT NULL,
	reg_time TIMESTAMP DEFAULT LOCALTIMESTAMP(0) NOT NULL,
	given_name VARCHAR ( 1000 ),
	surname VARCHAR ( 1000 ),
	address VARCHAR ( 1000 ),
	tel_nr VARCHAR ( 20 ),
	CONSTRAINT PK_Person PRIMARY KEY (_id),
	CONSTRAINT AK_Person_country_and_nat_id_code UNIQUE (nat_id_code,
country_code),
	CONSTRAINT Person_e_mail_contains_atleast_one_at_sign CHECK (e_mail
LIKE '%@%' ),
	CONSTRAINT Person_birth_date_between_01_01_1900_to_31_12_2100 CHECK
(birth_date >= '1900-01-01' AND birth_date < '2101-01-01'),
	CONSTRAINT Person_nat_id_code_contains_valid_symbols CHECK
(nat_id_code ~* '^[[:alnum:][:space:]+=\/-]+$'),
	CONSTRAINT Person_address_not_contains_only_numbers CHECK (address !~
'^\d+?$'),
	CONSTRAINT Person_address_not_empty CHECK (address !~ '^[[:space:]]+$'
AND address <> ''),
	CONSTRAINT Person_given_name_not_empty CHECK (given_name !~
'^[[:space:]]+$' AND given_name <> ''),
	CONSTRAINT Person_given_name_or_surname_is_registered CHECK
((given_name IS NOT NULL) OR (surname IS NOT NULL)),
	CONSTRAINT Person_birth_date_not_bigger_than_reg_time CHECK
(birth_date <= reg_time),
	CONSTRAINT Person_reg_time_between_01_01_2010_to_31_12_2100 CHECK
(reg_time >= '2010-01-01' AND reg_time < '2101-01-01'),
	CONSTRAINT Person_surname_not_empty CHECK (surname !~ '^[[:space:]]+$'
AND surname <> ''),
	CONSTRAINT Person_nat_id_code_not_empty CHECK (nat_id_code !~
'^[[:space:]]+$' AND nat_id_code <> ''),
	CONSTRAINT Person_tel_nr_is_valid CHECK (tel_nr ~* '^[0-9\s\-
\+]{7,20}$'),
	CONSTRAINT Person_tel_nr_not_empty CHECK (tel_nr !~ '^[[:space:]]+$'
AND tel_nr <> '')
	);
CREATE TABLE Country (
	country_code VARCHAR ( 3 ) NOT NULL,
	name VARCHAR ( 100 ) NOT NULL,
	CONSTRAINT PK_Country PRIMARY KEY (country_code),
	CONSTRAINT AK_Country_name UNIQUE (name),
	CONSTRAINT Country_name_not_empty CHECK (name !~ '^[[:space:]]+$' AND
name <> ''),
	CONSTRAINT Country_country_code_consists_of_three_capital_letters
CHECK (country_code ~ '^[A-Z]{3}$')
```

```
	);
CREATE TABLE Occupation (
	occupation_code SMALLINT NOT NULL,
	name VARCHAR ( 100 ) NOT NULL,
	description TEXT,
	CONSTRAINT AK_Occupation_name UNIQUE (name),
	CONSTRAINT PK_Occupation PRIMARY KEY (occupation_code),
	CONSTRAINT Occupation_name_not_empty CHECK (name !~ '^[[:space:]]+$'
AND name <> ''),
	CONSTRAINT Occupation_description_not_empty CHECK (description !~
'^[[:space:]]+$' AND description <> ''),
	CONSTRAINT Occupation_occupation_code_greater_than_zero CHECK
(occupation_code > 0)
	);
ALTER TABLE Person ADD CONSTRAINT FK_Person_country_code FOREIGN KEY
(country_code) REFERENCES Country (country_code)  ON DELETE NO ACTION ON
UPDATE CASCADE;
ALTER TABLE Person ADD CONSTRAINT FK_Person_person_status_type_code FOREIGN
KEY (person_status_type_code) REFERENCES Person_status_type
(person_status_type_code) ON DELETE NO ACTION ON UPDATE CASCADE;
ALTER TABLE Employee ADD CONSTRAINT FK_Employee_person_id FOREIGN KEY
(person_id) REFERENCES Person (_id)  ON DELETE CASCADE ON UPDATE NO ACTION;
ALTER TABLE Employee ADD CONSTRAINT FK_Employee_mentor_id FOREIGN KEY
(mentor_id) REFERENCES Employee (person_id)  ON DELETE SET NULL ON UPDATE NO
ACTION;
ALTER TABLE Employee ADD CONSTRAINT FK_Employee_employee_status_type_code
FOREIGN KEY (employee_status_type_code) REFERENCES Employee_status_type
(employee_status_type_code)  ON DELETE NO ACTION ON UPDATE CASCADE;
ALTER TABLE User_account ADD CONSTRAINT FK_User_account_person_id FOREIGN KEY
(person_id) REFERENCES Person (_id)  ON DELETE CASCADE ON UPDATE NO ACTION;
ALTER TABLE Employment ADD CONSTRAINT FK_Employment_person_id FOREIGN KEY
(person_id) REFERENCES Employee (person_id)  ON DELETE CASCADE ON UPDATE NO
ACTION;
ALTER TABLE Employment ADD CONSTRAINT FK_Employment_occupation_code FOREIGN
KEY (occupation_code) REFERENCES Occupation (occupation_code)  ON DELETE NO
ACTION ON UPDATE CASCADE;
CREATE UNIQUE INDEX idx_ak_person_e_mail ON Person(LOWER(e_mail));
```

Lisa 3 – Hierarhiliste JSON dokumentidega PostgreSQL andmebaasi skeemi kood

```
CREATE TABLE Country (
	country_code VARCHAR ( 3 ) NOT NULL,
	data JSONB NOT NULL,
	CONSTRAINT PK_Country PRIMARY KEY (country_code),
	CONSTRAINT Country_name_not_empty CHECK ((CAST (data ->> 'name' AS
VARCHAR)) !~ '^[[:space:]]+$' AND (CAST (data ->> 'name' AS VARCHAR)) <> ''),
	CONSTRAINT Country_country_code_consists_of_three_capital_letters
CHECK (country_code ~ '^[A-Z]{3}$'),
	CONSTRAINT Country_name_not_null CHECK ((CAST (data ->> 'name' AS
VARCHAR)) IS NOT NULL),
	CONSTRAINT Country_name_max_length_100 CHECK (char_length((CAST (data
->> 'name' AS VARCHAR))) <= 100)
	);
CREATE TABLE Employee_status_type (
	employee_status_type_code SMALLINT NOT NULL,
	data JSONB NOT NULL,
	CONSTRAINT PK_Employee_status_type PRIMARY KEY
(employee_status_type_code),
	CONSTRAINT Employee_status_type_description_not_empty CHECK ((CAST
(data ->> 'description' AS VARCHAR)) !~ '^[[:space:]]+$' AND (CAST (data ->>
'description' AS VARCHAR)) <> ''),
	CONSTRAINT Employee_status_type_name_not_empty CHECK ((CAST (data ->>
'name' AS VARCHAR)) !~ '^[[:space:]]+$' AND (CAST (data ->> 'name' AS
VARCHAR)) <> ''),
	CONSTRAINT Employee_status_type_name_not_null CHECK ((CAST (data ->>
'name' AS VARCHAR)) IS NOT NULL),
	CONSTRAINT Employee_status_type_name_max_length_100 CHECK
(char_length((CAST (data ->> 'name' AS VARCHAR))) <= 100)
	);
CREATE TABLE Person_status_type (
	person_status_type_code SMALLINT NOT NULL,
	data JSONB NOT NULL,
	CONSTRAINT PK_Person_status_type PRIMARY KEY
(person_status_type_code),
	CONSTRAINT Person_status_type_description_not_empty CHECK ((CAST (data
->> 'description' AS VARCHAR)) !~ '^[[:space:]]+$' AND (CAST (data ->>
'description' AS VARCHAR)) <> ''),
	CONSTRAINT Person_status_type_name_not_empty CHECK ((CAST (data ->>
'name' AS VARCHAR)) !~ '^[[:space:]]+$' AND (CAST (data ->> 'name' AS
VARCHAR)) <> ''),
	CONSTRAINT Person_status_type_name_not_null CHECK ((CAST (data ->>
'name' AS VARCHAR)) IS NOT NULL),
	CONSTRAINT Person_status_type_name_max_length_100 CHECK
(char_length((CAST (data ->> 'name' AS VARCHAR))) <= 100)
	);
CREATE TABLE Person (
	_id SERIAL NOT NULL,
	country_code VARCHAR ( 3 ) NOT NULL,
```

```
	person_status_type_code SMALLINT NOT NULL,
	data JSONB NOT NULL,
	employee JSONB,
	CONSTRAINT PK_Person PRIMARY KEY (_id),
	CONSTRAINT Person_e_mail_contains_atleast_one_at_sign CHECK ((CAST
(data ->> 'e_mail' AS VARCHAR)) LIKE '%@%' ),
	CONSTRAINT Person_nat_id_code_contains_valid_symbols CHECK ((CAST
(data ->> 'nat_id_code' AS VARCHAR)) ~* '^[[:alnum:][:space:]+=\/-]+$'),
	CONSTRAINT Person_address_not_contains_only_numbers CHECK (CAST(data
->> 'address' AS VARCHAR) !~ '^\d+?$'),
	CONSTRAINT Person_address_not_empty CHECK (CAST(data  ->> 'address' AS
VARCHAR) !~ '^[[:space:]]+$' AND CAST(data  ->> 'address' AS VARCHAR) <> ''),
	CONSTRAINT Person_given_name_not_empty CHECK ((CAST (data ->>
'given_name' AS VARCHAR)) !~ '^[[:space:]]+$' AND (CAST (data ->>
'given_name' AS VARCHAR)) <> ''),
	CONSTRAINT Employee_person_id_is_not_mentor_id CHECK (_id <> (CAST (
employee ->> 'mentor_id' AS INTEGER))),
	CONSTRAINT Person_given_name_or_surname_is_registered CHECK ((CAST
(data ->> 'given_name' AS VARCHAR)) IS NOT NULL OR (CAST (data ->> 'surname'
AS VARCHAR)) IS NOT NULL),
	CONSTRAINT Person_surname_not_empty CHECK ((CAST (data ->> 'surname'
AS VARCHAR)) !~ '^[[:space:]]+$' AND (CAST (data ->> 'surname' AS VARCHAR))
<> ''),
	CONSTRAINT Person_nat_id_code_not_empty CHECK ((CAST (data ->>
'nat_id_code' AS VARCHAR)) !~ '^[[:space:]]+$' AND (CAST (data ->>
'nat_id_code' AS VARCHAR)) <> ''),
	CONSTRAINT Person_tel_nr_is_valid CHECK ((CAST (data ->> 'tel_nr' AS
VARCHAR)) ~* '^[0-9\s\-\+]{7,20}$'),
	CONSTRAINT Person_tel_nr_not_empty CHECK ((CAST (data ->> 'tel_nr' AS
VARCHAR)) !~ '^[[:space:]]+$' AND (CAST (data ->> 'tel_nr' AS VARCHAR)) <>
''),
	CONSTRAINT Person_e_mail_not_null CHECK (((data ->> 'e_mail' AS
VARCHAR)) IS NOT NULL),
	CONSTRAINT Person_nat_id_code_not_null CHECK (((data ->> 'nat_id_code'
AS VARCHAR)) IS NOT NULL),
	CONSTRAINT Person_birth_date_code_not_null CHECK (((data ->>
'birth_date' AS VARCHAR)) IS NOT NULL),
	CONSTRAINT Person_reg_time_not_null CHECK (((data ->> 'reg_time' AS
VARCHAR)) IS NOT NULL),
	CONSTRAINT Person_e_mail_max_length_254 CHECK (char_length((CAST (data
->> 'e_mail' AS VARCHAR))) <= 254),
	CONSTRAINT Person_nat_id_code_max_length_50 CHECK (char_length((CAST
(data ->> 'nat_id_code' AS VARCHAR))) <= 50),
	CONSTRAINT Person_given_name_max_length_1000 CHECK (char_length((CAST
(data ->> 'given_name' AS VARCHAR))) <= 1000),
	CONSTRAINT Person_surname_max_length_1000 CHECK (char_length((CAST
(data ->> 'given_name' AS VARCHAR))) <= 1000),
	CONSTRAINT Person_address_max_length_1000 CHECK (char_length((CAST
(data ->> 'address' AS VARCHAR))) <= 1000)
	);
CREATE TABLE Occupation (
	occupation_code SMALLINT NOT NULL,
	data JSONB NOT NULL,
	CONSTRAINT PK_Occupation PRIMARY KEY (occupation_code),
```

```
      CONSTRAINT Occupation_name_not_empty CHECK ((CAST (data ->> 'name' AS
VARCHAR)) !~ '^[[:space:]]+$' AND (CAST (data ->> 'name' AS VARCHAR)) <> ''),
      CONSTRAINT Occupation_description_not_empty CHECK ((CAST (data ->>
'description' AS VARCHAR)) !~ '^[[:space:]]+$' AND (CAST (data ->>
'description' AS VARCHAR)) <> ''),
      CONSTRAINT Occupation_name_not_null CHECK ((CAST (data ->> 'name' AS
VARCHAR)) IS NOT NULL),
      CONSTRAINT Occupation_occupation_code_greater_than_zero CHECK
(occupation_code > 0),
      CONSTRAINT Occupation_name_max_length_100 CHECK (char_length((CAST
(data ->> 'name' AS VARCHAR))) <= 100)
      );
ALTER TABLE Person ADD CONSTRAINT FK_Person_country_code FOREIGN KEY
(country_code) REFERENCES Country (country_code)  ON DELETE NO ACTION ON
UPDATE CASCADE;
ALTER TABLE Person ADD CONSTRAINT FK_Person_person_status_type_code FOREIGN
KEY (person_status_type_code) REFERENCES Person_status_type
(person_status_type_code)  ON DELETE NO ACTION ON UPDATE CASCADE;
CREATE UNIQUE INDEX idx_ak_country_name ON Country((CAST (data ->> 'name' AS
VARCHAR)));
CREATE UNIQUE INDEX idx_ak_employee_status_type_name ON
Employee_status_type((CAST (data ->> 'name' AS VARCHAR)));
CREATE UNIQUE INDEX idx_ak_person_status_type_name ON
Person_status_type((CAST (data ->> 'name' AS VARCHAR)));
CREATE UNIQUE INDEX idx_ak_occupation_name ON Occupation((CAST (data ->>
'name' AS VARCHAR)));
CREATE UNIQUE INDEX idx_ak_person_country_and_nat_id_code ON
Person(country_code, (CAST (data ->> 'nat_id_code' AS VARCHAR)));
CREATE UNIQUE INDEX idx_ak_person_e_mail ON Person(LOWER(CAST (data ->>
'e_mail' AS VARCHAR)));
```

Lisa 4 – Mitte-hierarhiliste JSON dokumentidega PostgreSQL andmebaasi skeemi kood

```
CREATE TABLE Country (
	country_code VARCHAR ( 3 ) NOT NULL,
	data JSONB NOT NULL,
	CONSTRAINT PK_Country PRIMARY KEY (country_code),
	CONSTRAINT Country_name_not_empty CHECK ((CAST (data ->> 'name' AS
VARCHAR)) !~ '^[[:space:]]+$' AND (CAST (data ->> 'name' AS VARCHAR)) <> ''),
	CONSTRAINT Country_country_code_consists_of_three_capital_letters
CHECK (country_code ~ '^[A-Z]{3}$'),
	CONSTRAINT Country_name_not_null CHECK ((CAST (data ->> 'name' AS
VARCHAR)) IS NOT NULL),
	CONSTRAINT Country_name_max_length_100 CHECK (char_length((CAST (data
->> 'name' AS VARCHAR))) <= 100)
	);
CREATE TABLE Occupation (
	occupation_code SMALLINT NOT NULL,
	data JSONB NOT NULL,
	CONSTRAINT PK_Occupation PRIMARY KEY (occupation_code),
	CONSTRAINT Occupation_name_not_empty CHECK ((CAST (data ->> 'name' AS
VARCHAR)) !~ '^[[:space:]]+$' AND (CAST (data ->> 'name' AS VARCHAR)) <> ''),
	CONSTRAINT Occupation_description_not_empty CHECK ((CAST (data ->>
'description' AS VARCHAR)) !~ '^[[:space:]]+$' AND (CAST (data ->>
'description' AS VARCHAR)) <> ''),
	CONSTRAINT Occupation_name_not_null CHECK ((CAST (data ->> 'name' AS
VARCHAR)) IS NOT NULL),
	CONSTRAINT Occupation_occupation_code_greater_than_zero CHECK
(occupation_code > 0),
	CONSTRAINT Occupation_name_max_length_100 CHECK (char_length((CAST
(data ->> 'name' AS VARCHAR))) <= 100)
	);
CREATE TABLE Employee_status_type (
	employee_status_type_code SMALLINT NOT NULL,
	data JSONB NOT NULL,
	CONSTRAINT PK_Employee_status_type PRIMARY KEY
(employee_status_type_code),
	CONSTRAINT Employee_status_type_description_not_empty CHECK ((CAST
(data ->> 'description' AS VARCHAR)) !~ '^[[:space:]]+$' AND (CAST (data ->>
'description' AS VARCHAR)) <> ''),
	CONSTRAINT Employee_status_type_name_not_empty CHECK ((CAST (data ->>
'name' AS VARCHAR)) !~ '^[[:space:]]+$' AND (CAST (data ->> 'name' AS
VARCHAR)) <> ''),
	CONSTRAINT Employee_status_type_name_not_null CHECK ((CAST (data ->>
'name' AS VARCHAR)) IS NOT NULL),
	CONSTRAINT Employee_status_type_name_max_length_100 CHECK
(char_length((CAST (data ->> 'name' AS VARCHAR))) <= 100)
	);
CREATE TABLE Person_status_type (
	person_status_type_code SMALLINT NOT NULL,
	data JSONB NOT NULL,
```

```
	CONSTRAINT PK_Person_status_type PRIMARY KEY
(person_status_type_code),
	CONSTRAINT Person_status_type_description_not_empty CHECK ((CAST (data
->> 'description' AS VARCHAR)) !~ '^[[:space:]]+$' AND (CAST (data ->>
'description' AS VARCHAR)) <> ''),
	CONSTRAINT Person_status_type_name_not_empty CHECK ((CAST (data ->>
'name' AS VARCHAR)) !~ '^[[:space:]]+$' AND (CAST (data ->> 'name' AS
VARCHAR)) <> ''),
	CONSTRAINT Person_status_type_name_not_null CHECK ((CAST (data ->>
'name' AS VARCHAR)) IS NOT NULL),
	CONSTRAINT Person_status_type_name_max_length_100 CHECK
(char_length((CAST (data ->> 'name' AS VARCHAR))) <= 100)
	);
CREATE TABLE Person (
	_id SERIAL NOT NULL,
	country_code VARCHAR ( 3 ) NOT NULL,
	person_status_type_code SMALLINT NOT NULL,
	data JSONB NOT NULL,
	CONSTRAINT PK_Person PRIMARY KEY (_id),
	CONSTRAINT Person_e_mail_contains_atleast_one_at_sign CHECK ((CAST
(data ->> 'e_mail' AS VARCHAR)) LIKE '%@%' ),
	CONSTRAINT Person_nat_id_code_contains_valid_symbols CHECK ((CAST
(data ->> 'nat_id_code' AS VARCHAR)) ~* '^[[:alnum:][:space:]+=\/-]+$'),
	CONSTRAINT Person_address_not_contains_only_numbers CHECK ((CAST (data
->> 'address' AS VARCHAR)) !~ '^\d+?$'),
	CONSTRAINT Person_address_not_empty CHECK ((CAST (data ->> 'address'
AS VARCHAR)) !~ '^[[:space:]]+$' AND (CAST (data ->> 'address' AS VARCHAR))
<> ''),
	CONSTRAINT Person_given_name_or_surname_is_registered CHECK ((CAST
(data ->> 'given_name' AS VARCHAR)) IS NOT NULL OR (CAST (data ->> 'surname'
AS VARCHAR)) IS NOT NULL),
	CONSTRAINT Person_surname_not_empty CHECK ((CAST (data ->> 'surname'
AS VARCHAR)) !~ '^[[:space:]]+$' AND (CAST (data ->> 'surname' AS VARCHAR))
<> ''),
	CONSTRAINT Person_given_name_not_empty CHECK ((CAST (data ->>
'given_name' AS VARCHAR)) !~ '^[[:space:]]+$' AND (CAST (data ->>
'given_name' AS VARCHAR)) <> ''),
	CONSTRAINT Person_nat_id_code_not_empty CHECK ((CAST (data ->>
'nat_id_code' AS VARCHAR)) !~ '^[[:space:]]+$' AND (CAST (data ->>
'nat_id_code' AS VARCHAR)) <> ''),
	CONSTRAINT Person_tel_nr_is_valid CHECK ((CAST (data ->> 'tel_nr' AS
VARCHAR)) ~* '^[0-9\s\-\+]{7,20}$'),
	CONSTRAINT Person_tel_nr_not_empty CHECK ((CAST (data ->> 'tel_nr' AS
VARCHAR)) !~ '^[[:space:]]+$' AND (CAST (data ->> 'tel_nr' AS VARCHAR)) <>
''),
	CONSTRAINT Person_e_mail_not_null CHECK (((data ->> 'e_mail' AS
VARCHAR)) IS NOT NULL),
	CONSTRAINT Person_nat_id_code_not_null CHECK (((data ->> 'nat_id_code'
AS VARCHAR)) IS NOT NULL),
	CONSTRAINT Person_birth_date_code_not_null CHECK (((data ->>
'birth_date' AS VARCHAR)) IS NOT NULL),
	CONSTRAINT Person_reg_time_not_null CHECK (((data ->> 'reg_time' AS
VARCHAR)) IS NOT NULL),
	CONSTRAINT Person_e_mail_max_length_254 CHECK (char_length((CAST (data
->> 'e_mail' AS VARCHAR))) <= 254),
```

```
	CONSTRAINT Person_nat_id_code_max_length_50 CHECK (char_length((CAST
(data ->> 'nat_id_code' AS VARCHAR))) <= 50),
	CONSTRAINT Person_given_name_max_length_1000 CHECK (char_length((CAST
(data ->> 'given_name' AS VARCHAR))) <= 1000),
	CONSTRAINT Person_surname_max_length_1000 CHECK (char_length((CAST
(data ->> 'given_name' AS VARCHAR))) <= 1000),
	CONSTRAINT Person_address_max_length_1000 CHECK (char_length((CAST
(data ->> 'address' AS VARCHAR))) <= 1000)
	);
CREATE TABLE Employee (
	person_id INTEGER NOT NULL,
	mentor_id INTEGER,
	employee_status_type_code SMALLINT NOT NULL,
	CONSTRAINT PK_Employee PRIMARY KEY (person_id),
	CONSTRAINT Employee_person_id_not_mentor_id CHECK (person_id !=
mentor_id)
	);
CREATE INDEX IDX_Employee_person_id ON Employee (person_id );
CREATE INDEX IDX_Employee_mentor_id ON Employee (mentor_id );
CREATE TABLE Employment (
	_id SERIAL NOT NULL,
	person_id INTEGER NOT NULL,
	occupation_code SMALLINT NOT NULL,
	data JSONB NOT NULL,
	CONSTRAINT PK_Employment PRIMARY KEY (_id),
	CONSTRAINT Employment_start_time_not_null CHECK (((data ->>
'start_time' AS VARCHAR)) IS NOT NULL)
	);
CREATE INDEX IDX_Employment_person_id ON Employment (person_id );
ALTER TABLE Employment ADD CONSTRAINT FK_Employment_person_id FOREIGN KEY
(person_id) REFERENCES Employee (person_id)  ON DELETE CASCADE ON UPDATE NO
ACTION;
ALTER TABLE Employment ADD CONSTRAINT FK_Employment_occupation_code FOREIGN
KEY (occupation_code) REFERENCES Occupation (occupation_code)  ON DELETE NO
ACTION ON UPDATE CASCADE;
ALTER TABLE Person ADD CONSTRAINT FK_Person_country_code FOREIGN KEY
(country_code) REFERENCES Country (country_code)  ON DELETE NO ACTION ON
UPDATE CASCADE;
ALTER TABLE Person ADD CONSTRAINT FK_Person_person_status_type_code FOREIGN
KEY (person_status_type_code) REFERENCES Person_status_type
(person_status_type_code) ON DELETE NO ACTION ON UPDATE CASCADE;
ALTER TABLE Employee ADD CONSTRAINT FK_Employee_person_id FOREIGN KEY
(person_id) REFERENCES Person (_id)  ON DELETE CASCADE ON UPDATE NO ACTION;
ALTER TABLE Employee ADD CONSTRAINT FK_Employee_mentor_id FOREIGN KEY
(mentor_id) REFERENCES Employee (person_id)  ON DELETE SET NULL ON UPDATE NO
ACTION;
ALTER TABLE Employee ADD CONSTRAINT FK_Employee_employee_status_type_code
FOREIGN KEY (employee_status_type_code) REFERENCES Employee_status_type
(employee_status_type_code)  ON DELETE NO ACTION ON UPDATE CASCADE;
CREATE UNIQUE INDEX idx_ak_country_name ON Country((CAST (data ->> 'name' AS
VARCHAR)));
CREATE UNIQUE INDEX idx_ak_employee_status_type_name ON
Employee_status_type((CAST (data ->> 'name' AS VARCHAR)));
```

```
CREATE UNIQUE INDEX idx_ak_occupation_name ON Occupation((CAST (data ->>
'name' AS VARCHAR)));
CREATE UNIQUE INDEX idx_ak_person_country_and_nat_id_code ON
Person(country_code, (CAST (data ->> 'nat_id_code' AS VARCHAR)));
CREATE UNIQUE INDEX idx_ak_person_e_mail ON Person(LOWER(CAST (data ->>
'e_mail' AS VARCHAR)));
CREATE UNIQUE INDEX idx_ak_employment_person_id_occupation_code_start_time ON
Employment(person_id, occupation_code, (CAST (data ->> 'start_time' AS
VARCHAR)));
CREATE UNIQUE INDEX idx_ak_person_status_type_name ON
Person_status_type((CAST (data ->> 'name' AS VARCHAR)));
```

Lisa 5 – Hierarhiliste JSON dokumentidega MongoDB andmebaasi skeemi kood

```
db.createCollection('person', {
  validator: {
    $jsonSchema: {
      bsonType: 'object',
      title: 'person',
      required: ['nat_id_code', 'country_code', 'person_status_type_code',
'e_mail', 'birth_date', 'reg_time'],
      properties: {
        nat_id_code: {
          bsonType: 'string',
          pattern: '^[[:alnum:][:space:]+=\/-]+$',
          maxLength: 50,
          description: 'National identification code cannot be empty. Max
length 50 symbols, consists of valid symbols.'
        },
        country_code: {
          bsonType: 'string',
          pattern: '^[A-Z]{3}$',
          description: 'Country code consists of three capital letters.'
        },
        person_status_type_code: {
          bsonType: 'int'
        },
        e_mail: {
          bsonType: 'string',
          pattern: '^\\S+@\\S+$',
          maxLength: 254,
          description: 'E-mail contains atleast one at sign. Max length 254
symbols.'
        },
        birth_date: {
          bsonType: 'date'
        },
        reg_time: {
          bsonType: 'date'
        },
        given_name: {
          bsonType: 'string',
          pattern: '\\S',
          maxLength: 1000,
          description: 'Given name cannot be empty. Max length 100 symbols.'
        },
        surname: {
          bsonType: 'string',
          pattern: '\\S',
          maxLength: 1000,
```

```
          description: 'Surname cannot be empty. Max length 100 symbols.'
        },
        address: {
          bsonType: 'string',
          pattern: '(.*[^0-9].*)(\\S)',
          maxLength: 1000,
          description: 'Address cannot be empty. Max length 1000 symbols, not
only numeric.'
        },
        tel_nr: {
          bsonType: 'string',
          pattern: '^[0-9\\s\\-\\+]{7,20}$',
          description: 'Tel. nr. can only contain numbers,whitespace,+,-.
Must be 7 to 20 characters long.'
        },
        employee: {
          bsonType: 'object',
          title: 'employee',
          required: ['employee_status_type_code'],
          properties: {
            employee_status_type_code: {
              bsonType: 'int'
            },
            mentor_id: {
              bsonType: 'objectId'
            },
            employment: {
              bsonType: 'array',
              items: {
                title: 'employment',
                required: ['occupation_code', 'start_time'],
                properties: {
                  occupation_code: {
                    bsonType: 'int'
                  },
                  start_time: {
                    bsonType: 'date'
                  },
                  end_time: {
                    bsonType: 'date'
                  }
                }
              }
            }
          }
        }
      }
    }
  }
});
db.person.createIndex({
```

```
  "e_mail": 1
}, {
  name: "ak_person_e_mail",
  unique: true,
  collation: {
    locale: "et",
    strength: 2
  }
})
db.person.createIndex({
  "country_code": 1,
  "nat_id_code": 1
}, {
  name: "ak_person_country_and_nat_id_code",
  unique: true
})
db.createCollection('country', {
  validator: {
    $jsonSchema: {
      bsonType: 'object',
      title: 'country',
      required: ['country_code', 'name'],
      properties: {
        country_code: {
          bsonType: 'string',
          pattern: '^[A-Z]{3}$',
          description: 'Country code consists of three capital letters.'
        },
        name: {
          bsonType: 'string',
          pattern: '\\S',
          maxLength: 100,
          description: 'Country name cannot be empty. Max length 100
symbols.'
        }
      }
    }
  }
});
db.country.createIndex({
  "name": 1
}, {
  name: "ak_country_name",
  unique: true
})

db.country.createIndex({
  "country_code": 1
}, {
  name: "pk_country",
  unique: true
```

```
})
db.createCollection('employee_status_type', {
  validator: {
    $jsonSchema: {
      bsonType: 'object',
      title: 'employee_status_type',
      required: ['employee_status_type_code', 'name'],
      properties: {
        employee_status_type_code: {
          bsonType: 'int'
        },
        name: {
          bsonType: 'string',
          pattern: '\\S',
          maxLength: 100,
          description: 'Employee status type name not empty. Max length 100
symbols.'
        },
        description: {
          bsonType: 'string',
          pattern: '\\S',
          description: 'Employee status type description cannot be empty.'
        }
      }
    }
  }
});
db.employee_status_type.createIndex({
  "name": 1
}, {
  name: "ak_employee_status_type_name",
  unique: true
})

db.employee_status_type.createIndex({
  "employee_status_type_code": 1
}, {
  name: "pk_employee_status_type",
  unique: true
})
db.createCollection('occupation', {
  validator: {
    $jsonSchema: {
      bsonType: 'object',
      title: 'occupation',
      required: ['occupation_code', 'name'],
      properties: {
        occupation_code: {
          bsonType: 'int',
          minimum: 1
        },
```

```
        name: {
          bsonType: 'string',
          pattern: '\\S',
          maxLength: 100,
          description: 'Occupation cannot be empty. Max 100 symbols.'
        },
        description: {
          bsonType: 'string',
          pattern: '\\S',
          description: 'Occupation description cannot be empty.'
        }
      }
    }
  }
});
db.occupation.createIndex({
  "name": 1
}, {
  name: "ak_occupation_name",
  unique: true
})

db.occupation.createIndex({
  "occupation_code": 1
}, {
  name: "pk_occupation",
  unique: true
})
db.createCollection('person_status_type', {
  validator: {
    $jsonSchema: {
      bsonType: 'object',
      title: 'person_status_type',
      required: ['person_status_type_code', 'name'],
      properties: {
        person_status_type_code: {
          bsonType: 'int'
        },
        name: {
          bsonType: 'string',
          pattern: '\\S',
          maxLength: 100,
          description: 'Person status type name cannot be empty. Max length
100.'
        },
        description: {
          bsonType: 'string',
          pattern: '\\S',
          description: 'Person status type description cannot be empty.'
        }
      }
```

```
    }
  }
});
db.person_status_type.createIndex({
  "person_status_type_code": 1
}, {
  name: "pk_person_status_type",
  unique: true
})

db.person_status_type.createIndex({
  "name": 1
}, {
  name: "ak_person_status_type_name",
  unique: true
})
```

Lisa 6 – Mitte-hierarhiliste JSON dokumentidega MongoDB andmebaasi skeemi kood

```
db.createCollection('person', {
  validator: {
    $jsonSchema: {
      bsonType: 'object',
      title: 'person',
      required: ['nat_id_code', 'country_code', 'person_status_type_code',
'e_mail', 'birth_date', 'reg_time'],
      properties: {
        nat_id_code: {
          bsonType: 'string',
          pattern: '^[[:alnum:][:space:]+=\/-]+$',
          maxLength: 50,
          description: 'National identification code cannot be empty. Max
length 50 symbols, consists of valid symbols.'
        },
        country_code: {
          bsonType: 'string',
          pattern: '^[A-Z]{3}$',
          description: 'Country code consists of three capital letters.'
        },
        person_status_type_code: {
          bsonType: 'int'
        },
        e_mail: {
          bsonType: 'string',
          pattern: '^\\S+@\\S+$',
          maxLength: 254,
          description: 'E-mail contains atleast one at sign. Max length 254
symbols.'
        },
        birth_date: {
          bsonType: 'date'
        },
        reg_time: {
          bsonType: 'date'
        },
        given_name: {
          bsonType: 'string',
          pattern: '\\S',
          maxLength: 1000,
          description: 'Given name cannot be empty. Max length 100 symbols.'
        },
        surname: {
          bsonType: 'string',
          pattern: '\\S',
          maxLength: 1000,
```

```
          description: 'Surname cannot be empty. Max length 100 symbols.'
        },
        address: {
          bsonType: 'string',
          pattern: '(.*[^0-9].*)(\\S)',
          maxLength: 1000,
          description: 'Address cannot be empty. Max length 1000 symbols, not
only numeric.'
        },
        tel_nr: {
          bsonType: 'string',
          pattern: '^[0-9\\s\\-\\+]{7,20}$',
          description: 'Tel. nr. can only contain numbers,whitespace,+,-.
Must be 7 to 20 characters long'
        }
      }
    }
  }
});
db.person.createIndex({
  "e_mail": 1
}, {
  name: "ak_person_e_mail",
  unique: true,
  collation: {
    locale: "et",
    strength: 2
  }
})
db.person.createIndex({
  "nat_id_code": 1,
  "country_code": 1
}, {
  name: "ak_person_country_and_nat_id_code",
  unique: true
})
db.createCollection('country', {
  validator: {
    $jsonSchema: {
      bsonType: 'object',
      title: 'country',
      required: ['country_code', 'name'],
      properties: {
        country_code: {
          bsonType: 'string',
          pattern: '^[A-Z]{3}$',
          description: 'Country code consists of three capital letters.'
        },
        name: {
          bsonType: 'string',
          pattern: '\\S',
```

```
          maxLength: 100,
          description: 'Country name cannot be empty. Max length 100
symbols.'
        }
      }
    }
  },
  validationLevel: 'strict',
  validationAction: 'error'
});
db.country.createIndex({
  "name": 1
}, {
  name: "ak_country_name",
  unique: true
})

db.country.createIndex({
  "country_code": 1
}, {
  name: "pk_country",
  unique: true
})
db.createCollection('employee', {
  validator: {
    $jsonSchema: {
      bsonType: 'object',
      title: 'employee',
      required: ['person_id', 'employee_status_type_code'],
      properties: {
        person_id: {
          bsonType: 'objectId'
        },
        employee_status_type_code: {
          bsonType: 'int'
        },
        mentor_id: {
          bsonType: 'objectId'
        }
      }
    }
  }
});
db.employee.createIndex({
  "person_id": 1
}, {
  name: "pk_employee",
  unique: true
})
db.createCollection('employee_status_type', {
  validator: {
```

```
    $jsonSchema: {
      bsonType: 'object',
      title: 'employee_status_type',
      required: ['employee_status_type_code', 'name'],
      properties: {
        employee_status_type_code: {
          bsonType: 'int'
        },
        name: {
          bsonType: 'string',
          pattern: '\\S',
          maxLength: 100,
          description: 'Employee status type name not empty. Max length 100
symbols.'
        },
        description: {
          bsonType: 'string',
          pattern: '\\S',
          description: 'Employee status type description cannot be empty.'
        }
      }
    }
  }
});
db.employee_status_type.createIndex({
  "name": 1
}, {
  name: "ak_employee_status_type_name",
  unique: true
})

db.employee_status_type.createIndex({
  "employee_status_type_code": 1
}, {
  name: "pk_employee_status_type",
  unique: true
})
db.createCollection('employment', {
  validator: {
    $jsonSchema: {
      bsonType: 'object',
      title: 'employment',
      required: ['person_id', 'occupation_code', 'start_time'],
      properties: {
        person_id: {
          bsonType: 'objectId'
        },
        occupation_code: {
          bsonType: 'int'
        },
        start_time: {
```

```
          bsonType: 'date'
        },
        end_time: {
          bsonType: 'date'
        }
      }
    }
  }
});
db.employment.createIndex({
  "person_id": 1,
  "occupation_code": 1,
  "start_time": 1
}, {
  name: "pk_employment",
  unique: true
})
db.createCollection('occupation', {
  validator: {
    $jsonSchema: {
      bsonType: 'object',
      title: 'occupation',
      required: ['occupation_code', 'name'],
      properties: {
        occupation_code: {
          bsonType: 'int',
          minimum: 1
        },
        name: {
          bsonType: 'string',
          pattern: '\\S',
          maxLength: 100,
          description: 'Occupation cannot be empty. Max 100 symbols.'
        },
        description: {
          bsonType: 'string',
          pattern: '\\S',
          description: 'Occupation description cannot be empty.'
        }
      }
    }
  }
});
db.occupation.createIndex({
  "name": 1
}, {
  name: "ak_occupation_name",
  unique: true
})

db.occupation.createIndex({
```

```
  "occupation_code": 1
}, {
  name: "pk_occupation",
  unique: true
})
db.createCollection('person_status_type', {
  validator: {
    $jsonSchema: {
      bsonType: 'object',
      title: 'person_status_type',
      required: ['person_status_type_code', 'name'],
      properties: {
        person_status_type_code: {
          bsonType: 'int'
        },
        name: {
          bsonType: 'string',
          pattern: '\\S',
          maxLength: 100,
          description: 'Person status type name cannot be empty. Max length
100.'
        },
        description: {
          bsonType: 'string',
          pattern: '\\S',
          description: 'Person statys type description cannot be empty.'
        }
      }
    }
  }
});
db.person_status_type.createIndex({
  "person_status_type_code": 1
}, {
  name: "pk_person_status_type",
  unique: true
})

db.person_status_type.createIndex({
  "name": 1
}, {
  name: "ak_person_status_type_name",
  unique: true
})
```

Lisa 7 – Tagarakenduste lõpp-punktid (*endpoints*)

```
Endpoints for all controllers:

GET http://localhost:8080 <-- validate current running version of backend



Countries:

GET http://localhost:8080/countries <-- get all countries
GET http://localhost:8080/countries/{countryCode} <-- get country by code
POST http://localhost:8080/countries <-- add a new country


Occupations:

GET http://localhost:8080/occupations <-- get all occupations
GET http://localhost:8080/occupations/{occupationCode} <-- get occupation by
code
POST http://localhost:8080/occupations <-- add a new occupation


Employee status types:

GET http://localhost:8080/employeeStatusTypes <-- get all employee status
types
GET http://localhost:8080/employeeStatusTypes/{employeeStatusTypeCode} <--
get employee status type by code
POST http://localhost:8080/employeeStatusTypes <-- add a new employee status
type


Person status types:

GET http://localhost:8080/personStatusTypes <-- get all person status types
GET http://localhost:8080/personStatusTypes/{personStatusTypeCode} <-- get
person status type by code
POST http://localhost:8080/personStatusTypes <-- add a new person status type


Persons:

GET http://localhost:8080/persons <-- get all persons
GET http://localhost:8080/persons/{objectID} <-- get person by id
POST http://localhost:8080/persons <-- add a new person
PUT http://localhost:8080/persons <-- update person
```

Employees:

GET http://localhost:8080/employees <-- get all employees
GET http://localhost:8080/employees/{personId} <-- get employee by person_id
POST http://localhost:8080/employees <-- add a new employee
PUT http://localhost:8080/employees <-- update employee
DELETE http://localhost:8080/employees/{personId} <-- delete employee by person_id

Employments:

GET http://localhost:8080/employments/occupationCode={occupationCode} <-- get all employments for occupation_code
GET http://localhost:8080/employments/personId={personId} <-- get all employments for employee by person_id
POST http://localhost:8080/employments <-- add a new employment for employee
PUT http://localhost:8080/employments <-- update employment for employee
PUT http://localhost:8080/employments/endEmployments <-- end all employments for employee

Lisa 8 – Andmebaasides realiseeritud kitsendused

+ kitsenduse saab deklaratiivselt andmebaasis jõustada

- kitsendust ei saa deklaratiivselt andmebaasis jõustada

X – kitsendust pole tarvis jõustada, sest see on juba jõustatud andmestruktuuride ülesehituse kaudu

Andmetüüp

1 – Traditsiooniline PostgreSQL
2 – PostgreSQL + hierarhiline JSON
3 – PostgreSQL + mittehierarhiline JSON
4 – MongoDB + hierarhiline JSON
5 – MongoDB + mitte-hierarhiline JSON

Kitsendus	**1**	**2**	**3**	**4**	**5**
Isik.isikukood on tekstiline väärtus	+	+	+	+	+
Isik.riik_kood on tekstiline väärtus.	+	+	+	+	+
Isik.eesnimi on tekstiline väärtus	+	+	+	+	+
Isik.perenimi on tekstiline väärtus	+	+	+	+	+
Isik.elukoht on tekstiline väärtus	+	+	+	+	+
Isik.e_meil on tekstiline väärtus	+	+	+	+	+
Isik.telefoni_nr on tekstiline väärtus	+	+	+	+	+
Isik.synni_kp on kuupäev	+	-	-	+	+
Isik.reg_aeg on ajatempel	+	-	-	+	+
Riik.kood on tekstiline väärtus.	+	+	+	+	+
Riik.nimi on tekstiline väärtus.	+	+	+	+	+
Töötaja_seisundi_liik.kood on arvuline väärtus.	+	+	+	+	+
Töötaja_seisundi_liik.nimetus on tekstiline väärtus.	+	+	+	+	+
Töötaja_seisundi_liik.kirjeldus on tekstiline väärtus.	+	+	+	+	+
Ametis_töötamine.alguse_aeg on ajatempel.	+	-	-	+	+
Ametis_töötamine.lõpu_aeg on ajatempel.	+	-	-	+	+
Amet.amet_kood on arvuline väärtus.	+	+	+	+	+
Amet.nimetus on tekstiline väärtus.	+	+	+	+	+

Kitsendus	1	2	3	4	5
Amet.kirjeldus on tekstiline väärtus.	+	+	+	+	+
Isiku_seisundi_liik.isiku_seisundi_liik_kood on arvuline väärtus.	+	+	+	+	+
Isiku_seisundi_liik.nimetus on tekstiline väärtus.	+	+	+	+	+
Isiku_seisundi_liik.kirjeldus on tekstiline väärtus.	+	+	+	+	+
Kokku (maksimaalsest 22st)	**22**	**18**	**18**	**22**	**22**

Väljapikkus

1 – Traditsiooniline PostgreSQL
2 – PostgreSQL + hierarhiline JSON
3 – PostgreSQL + mittehierarhiline JSON
4 – MongoDB + hierarhiline JSON
5 – MongoDB + mitte-hierarhiline JSON

Kitsendus	1	2	3	4	5
Isik.isikukood on kuni 50 märki	+	+	+	+	+
Isik.e_mail on kuni 254 märki	+	+	+	+	+
Isik.eesnimi on kuni 1000 märki	+	+	+	+	+
Isik.perenimi on kuni 1000 märki	+	+	+	+	+
Isik.aadress on kuni 1000 märki	+	+	+	+	+
Isik.telefoni_nr on kuni 20 märki	+	+	+	+	+
Isik.riik_kood on kuni 3 märki.	+	+	+	+	+
Riik.riik_kood on kuni 3 märki.	+	+	+	+	+
Riik.nimetus on kuni 100 märki.	+	+	+	+	+
Töötaja_seisundi_liik.nimetus on kuni 100 märki.	+	+	+	+	+
Amet.nimetus on kuni 100 märki.	+	+	+	+	+
Isiku_seisundi_liik.nimetus on kuni 100 märki.	+	+	+	+	+
Kokku (maksimaalsest 12st)	**12**	**12**	**12**	**12**	**12**

Kohustuslikkus

1 – Traditsiooniline PostgreSQL
2 – PostgreSQL + hierarhiline JSON
3 – PostgreSQL + mittehierarhiline JSON
4 – MongoDB + hierarhiline JSON
5 – MongoDB + mitte-hierarhiline JSON

Kitsendus	**1**	**2**	**3**	**4**	**5**
Isik.isikukood on kohustuslik.	+	+	+	+	+
Isiku isikukoodi riik on kohustuslik.	+	+	+	+	+
Isiku seisundi liik on kohustuslik.	+	+	+	+	+
Isik.e_mail on kohustuslik.	+	+	+	+	+
Isik.synni_kp on kohustuslik.	+	+	+	+	+
Isik.reg_aeg on kohustuslik.	+	+	+	+	+
Riik_riik_kood on kohustuslik.	+	+	+	+	+
Riik.nimetus on kohustuslik.	+	+	+	+	+
Töötaja.isik_id on kohustuslik.	+	X	+	X	+
Töötaja.töötaja_seisundi_liik_kood on kohustuslik.	+	-	+	+	+
Töötaja_seisundi_liik.kood on kohustuslik.	+	+	+	+	+
Töötaja_seisundi_liik.nimetus on kohustuslik.	+	+	+	+	+
Ametis_töötamine.isik_id on kohustuslik.	+	X	+	X	+
Ametis_töötamine.amet_kood on kohustuslik.	+	-	+	+	+
Ametis_töötamine.alguse_aeg on kohustuslik.	+	-	+	+	+
Amet.amet_kood on kohustuslik.	+	+	+	+	+
Amet.nimetus on kohustuslik.	+	+	+	+	+
Isiku_seisundi_liik.kood on kohustuslik.	+	+	+	+	+
Isiku_seisundi_liik.nimetus on kohustuslik.	+	+	+	+	+
Kokku	**19/ 19**	**14/ 17**	**19/ 19**	**17/ 17**	**19/ 19**

Unikaalsus

1 – Traditsiooniline PostgreSQL
2 – PostgreSQL + hierarhiline JSON
3 – PostgreSQL + mittehierarhiline JSON
4 – MongoDB + hierarhiline JSON
5 – MongoDB + mitte-hierarhiline JSON

Kitsendus	**1**	**2**	**3**	**4**	**5**
Isiku isikukoodi ja riigi koodi kombinatsioon on unikaalne.	+	+	+	+	+
Isiku meiliaadress on tõstutundetult unikaalne.	+	+	+	+	+
Riigi kood on unikaalne.	+	+	+	+	+
Riigi nimetus on unikaalne.	+	+	+	+	+
Töötaja.isik_id on unikaalne.	+	+	+	+	+
Töötaja seisundi liigi kood on unikaalne.	+	+	+	+	+
Töötaja seisundi liigi nimetus on unikaalne.	+	+	+	+	+
Ametis töötamise isik_id, ameti koodi ning ameti asumise alguskuupäeva kombinatsioon on unikaalne.	+	-	+	-	+
Ameti kood on unikaalne.	+	+	+	+	+
Ameti nimetus on unikaalne.	+	+	+	+	+
Isiku seisundi liigi kood on unikaalne.	+	+	+	+	+
Isiku seisundi liigi nimetus on unikaalne.	+	+	+	+	+
Kokku (maksimaalsest 12st)	**12**	**11**	**12**	**11**	**12**

Viidete terviklikkus

1 – Traditsiooniline PostgreSQL
2 – PostgreSQL + hierarhiline JSON
3 – PostgreSQL + mittehierarhiline JSON
4 – MongoDB + hierarhiline JSON
5 – MongoDB + mitte-hierarhiline JSON

Kitsendus	**1**	**2**	**3**	**4**	**5**
Isiku juures olev riik on riigina registreeritud.	+	+	+	-	-
Isiku juures olev seisundi liik on isiku seisundi liigina registreeritud.	+	+	+	-	-
Töötaja juures olev isik on registreeritud isikuna.	+	X	+	X	-
Töötaja juures olev mentor on registreeritud töötajana.	+	-	+	-	-

Kitsendus	1	2	3	4	5
Töötaja juures olev seisundi liik on registreeritud töötaja seisundi liigina.	+	-	+	-	-
Ametis töötamise juures olev isik on registreeritud töötajana.	+	X	+	X	-
Ametis töötamise juures olev amet on registreeritud ametina.	+	-	+	-	-
Kokku	**7/7**	**2/5**	**7/7**	**0/5**	**0/7**

Lisapiirangud

1 – Traditsiooniline PostgreSQL
2 – PostgreSQL + hierarhiline JSON
3 – PostgreSQL + mittehierarhiline JSON
4 – MongoDB + hierarhiline JSON
5 – MongoDB + mitte-hierarhiline JSON

Kitsendus	1	2	3	4	5
Isikukoodis on lubatud tähed (lubatud on ka muud tähed kui ASCII tähed a-zA-Z), numbrid, tühikud, sidekriipsud, plussmärgid, võrdusmärgid ja kaldkriipsud.	+	+	+	+	+
E-meil sisaldab vähemalt ühte @ märki.	+	+	+	+	+
Sünnikuupäev on vahemikus 01.01.1900-31.12.2100.	+	-	-	-	-
Registreerimise aeg on vahemikus 01.01.2010-31.12.2100.	+	-	-	-	-
Sünnikuupäev pole suurem kui registreerimise aeg.	+	-	-	-	-
Eesnimi või perenimi on registreeritud.	+	+	+	-	-
Aadress ei koosne vaid numbritest.	+	+	+	+	+
Aadress pole tühi string ega tühimärkidest koosnev string.	+	+	+	+	+
Eesnimi pole tühi string ega tühimärkidest koosnev string.	+	+	+	+	+
Perenimi pole tühi string ega tühimärkidest koosnev string.	+	+	+	+	+
Isikukood pole tühi string ega tühimärkidest koosnev string.	+	+	+	-	-
Telefoninumbris on lubatud numbrid, tühikud, plussmärgid ning sidekriipsud.	+	+	+	+	+
Telefoninumber on vähemalt 7 tähemärgi pikkune.	+	+	+	+	+
Telefoninumber pole tühi string ega tühimärkidest koosnev string.	+	+	+	-	-
Riigi nimetus pole tühi string ega tühimärkidest koosnev string.	+	+	+	+	+
Riigi nimetus koosneb kolmest suurest tähest.	+	+	+	+	+
Töötaja pole iseenda mentor.	+	+	+	-	-

Kitsendus	**1**	**2**	**3**	**4**	**5**
Töötaja seisundi liigi nimetus pole tühi string ega tühimärkidest koosnev string.	+	+	+	+	+
Töötaja seisundi liigi kirjeldus pole tühi string ega tühimärkidest koosnev string.	+	+	+	+	+
Ametis olemise alguse aeg on vahemikus 01.01.2010-31.12.2100.	+	-	-	-	-
Ametis olemise lõpu aeg on vahemikus 01.01.2010-31.12.2100.	+	-	-	-	-
Ametis olemise alguse aeg on suurem lõpu ajast.	+	-	-	-	-
Ameti nimetus pole tühi string ega tühimärkidest koosnev string.	+	+	+	+	+
Ameti kirjeldus pole tühi string ega tühimärkidest koosnev string.	+	+	+	+	+
Ameti kood on suurem 0-st.	+	+	+	+	+
Isiku seisundi liigi nimetus pole tühi string ega tühimärkidest koosnev string.	+	+	+	+	+
Isiku seisundi liigi kirjeldus pole tühi string ega tühimärkidest koosnev string.	+	+	+	+	+
Kokku (maksimaalsest 27st)	**27**	**21**	**21**	**17**	**17**

Lisa 9 – Andmebaasides realiseeritud kompenseerivad tegevused

1 – Traditsiooniline PostgreSQL
2 – PostgreSQL + hierarhiline JSON
3 – PostgreSQL + mittehierarhiline JSON
4 – MongoDB + hierarhiline JSON
5 – MongoDB + mitte-hierarhiline JSON

Viidete terviklikkusega seotud kompenseeriv tegevus	**1**	**2**	**3**	**4**	**5**
Riigi identifikaatori muutmisel riigi juures muutub see ka vastavate isikute juures.	+	+	+	-	-
Isiku seisundi liigi identifikaatori muutmisel isiku seisundi liigi juures muutub see ka vastavate isikute juures.	+	+	+	-	-
Isiku kustutamisel kustutatakse ka tema kui töötaja andmed.	+	+	+	-	-
Töötaja kustutamisel kustutatakse ka tema ametis töötamiste andmed.	+	+	+	+	-
Töötaja seisundi identifikaatori muutmisel töötaja seisundi liigi juures muutub see ka vastavate töötajate juures.	+	-	+	-	-
Ameti identifikaatori muutmisel ameti juures muutub see ka vastavate ametis töötamiste juures.	+	-	+	-	-
Kui mentori andmed kustutatakse, siis tuleb tema juhendatavate juurest kustutada viide mentorile.	+	-	+	-	-
Kokku (maksimaalsest 7st)	**7**	**4**	**7**	**1**	**0**