TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Indrek Nurja 175023IDDR

Liidestuse ja töökohatarkvara pistikprogrammi arendus geograafiliste alade haldamiseks

Diplomitöö

Juhendaja: Meelis Antoi
MSc

Tallinn 2021

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: Indrek Nurja

06.01.2021

Annotatsioon

Käesoleva diplomitöö eesmärgiks oli luua lahendus geograafiliste alade haldamiseks ettevõttele X. Lahendus peab võimaldama kasutajatele regioonide haldamist õigustepõhiselt. Seejuures peab ruumiandmete haldus olema kasutajasõbralik. Kuigi lahendus oli arendatud kindlale ettevõttele, on võimalik saadud tulemit ja põhimõtteid rakendada ka kolmandatel osapooltel.

Diplomitöös tutvustatakse veidi geoinfosüsteemiga seotud mõisteid, standardeid, ruumiandmete salvestamise viise. Kirjeldatakse veidi ka probleemi olemust. Arendus jagunes kaheks: tagarakendus ja esirakendus. Diplomitöös analüüsiti erinevaid tehnoloogiaid, mis sobituksid probleemi lahendamiseks. Tagarakenduse puhul osutus sobivaimaks lahenduses Java ja GeoToolsil põhinev liidestus, mis ei olnud küll kõige kiirem, kuid sobitus rakenduse arhitektuuriga. Esirakenduse puhul sai valitud QGIS ja sinna pistikprogrammi arendamine, mis suhtleb diplomitöö käigus loodud liidestusega. Tulemusena valmis lahendus, mis täidab ettevõtte nõudeid. Töö autor tõi kokkuvõttes välja ka edasiarendused, mida võiks tulevikus teha.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 21 leheküljel, 7 peatükki, 10 joonist, 2 tabelit.

Abstract

Development of an interface and a GIS application plugin for managing geospatial data

The goal of this thesis was to create a solution for managing geospatial data for company X. The end product has to provide an authorized interface for users to manage the data. The handling of geospatial data should be made as user-friendly as possible. Although the finished product was developed for a certain company, the code and the principles of this solution could be applied by anyone.

First, the author of this thesis explains some terms, standards and datatypes that are used in geographical information systems. The approach of a separate backend interface and a GIS application is also explained from the companies point of view. The solution can be split into two parts: the backend and the frontend. For the backend, the author of this thesis analyzed a few different technologies, which are used for geospatial data. One of the deciding factors was the speed of handling geospatial data. The architecture of the underlying application was also taken into account. The best fitting solution for the interface ended up being a mix of Java Grails and GeoTools. For frontend, QGIS was chosen mainly because of the companies wish and its employees were already familiar with its user interface. A plugin was developed for QGIS, which interfaces with the backend.

The result of this thesis is a QGIS frontend with a custom plugin, which interfaces with the backend. The backend provides an authorized interface, making it possible to limit access to geospatial data, which a user could see or manage. QGIS with its built in tools makes managing geospatial data user-friendly.

The thesis is in Estonian and contains 21 pages of text, 7 chapters, 10 figures, 2 tables.

Lühendite ja mõistete sõnastik

ASCII	*American Standard Code for Information Interexchange*, keelemärkide tabel, mis sisaldab endas tähti, numbreid ja muid märke [1]
CSV	*Comma Separated Values*, tekstikujul olev failiformaat andmete salvestamiseks [2]
Feature	Ruumiobjekt koos lisa andmeväljadega [3]
FeatureCollection	Kogum ruumiobjektides, lisa andmeväljadega [3]
GeoJSON	Formaat ruumiandmete edastamiseks [3]
GET	HTTP meetod, mille abil on võimalik allikast andmeid pärida [4]
GIS	Geograafiline infosüsteem
GML	*Geography Markup Language*, formaat ruumiandmete edastamiseks [5]
HTTP	*Hypertext Transfer Protocol*, kliendi ja serveri vahel suhtlust võimaldav protokoll [4]
JSON	*JavaScript Object Notation*, andmete salvestamise ja edastamise formaat [6]
KEMIT	Keskkonnaministeeriumi Infotehnoloogiakeskus
KML	*Keyhole Markup Language*, formaat ruumiandmete visualiseerimiseks, edastamiseks [7]
MapServer	Avatud lähtekoodiga platvorm, mis serveerib ruumiandmeid interaktiivse kaardina veebi [8]
OGC	*Open Geospatial Consortium*, ühendus asutustest, kes aitavad kaasa GIS arengule [9]
POST	HTTP meetod, mille abil on võimalik sihtkohta andmeid saata, et neid salvestada või uuendada [4]
QGIS	Vabavaraline lauatarkvara ruumiandmete andmete visualiseerimiseks ja töötluseks [10]
REST	*Representational State Transfer*, arhitektuuri stiil, mis standardiseerib arvutisüsteemide suhtluse veebis [11]
XML	*Extensible Markup Language*, andmete salvestamise ja edastamise formaat [12]
WKT	*Well-Known Text*, ASCII kujul olevad ruumiandmed [13]

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

Käesoleva diplomitöö eesmärgiks oli luua lahendus geograafiliste alade haldamiseks. Kuigi lahendus oli arendatud kindlale ettevõttele, on võimalik saadud tulemit ja põhimõtteid rakendada ka kolmandatel osapooltel.

Ettevõttel on andmestik, mida võib nimetada regioonideks ja millel on oluline ruumiline mõõde. Regioonideks võivad olla näiteks jahi- või kiirabiteeninduspiirkonnad. Ruumiline mõõde tähendab, et neid alasid on võimalik kaardil kujutada erinevate geomeetriliste kujunditena. Lisaks võib alaga kaasas käia lisaandmed nagu vastutav isik ja tema kontaktandmed.

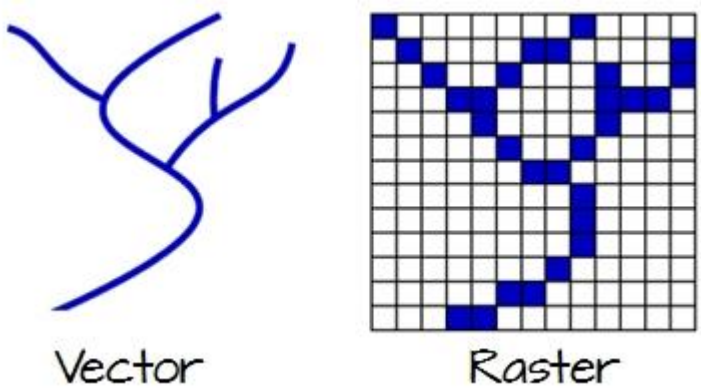
Regiooni ruumiandmed võivad tulevikus muutuda – muutub vastutav isik, kontaktandmed või muutuvad regiooni piirjooned. Samuti võib tulevikus lisanduda juurde uusi regioone. Seejuures peab toimuma andmetele ligipääs asutuse kasutajatel õigustepõhiselt. Diplomitöö eesmärgiks on leida sobivad tehnoloogiad probleemi lahendamiseks ja arendada valmis prototüüp.

Diplomitöö käigus analüüsitakse liidestuse arendamises erinevaid tehnoloogiad. Selgitatakse ära erinevate tehnoloogiate loogikad, iseärasused ja andmete käsitlemise kiirused ning valitakse sobivaim lahendus. Samuti kirjeldatakse töökohatarkvara ning sellele pistikprogrammi arendust, mis teeb kasutajale regioonide haldamise lihtsamaks.

Diplomitöö tulemusena on ettevõtte töötajatel võimalik läbi pistikprogrammi end autentida, pärida ainult neile nähtavaid regioone ja ruumiandmeid, teostada andmetes muudatusi ja neid salvestada kesksesse andmebaasi. Andmete töötlus on tänu liidestusele ja töökohatarkvarale kasutajasõbralik ja turvaline. Ei teki ebakõlasid andmete ajakohalisuses, andmed on nähtavad kindlatele isikutele ning samuti salvestatakse ka auditlogi.

2 Ruumiandmed, GIS, standardid

Ruumiandmed hõlmavad endas andmeid, kus lisaks tavaandmetele on juurde lisatud ka ruumiline mõõde [14]. Ruumiandmed võimaldavad tavaandmeid siduda asukohaga läbi koordinaatide või ka seotus aadressiga, indeksikoodiga või mõne muu omadusega, mille abil on võimalik objekti ruumis paigutada. Ruumiandmeid saab klassikaliselt tasapinnal jagada omakorda kaheks – vektorandmed [15] ja rasterandmed, kujutatud Joonis 1 [16], [17]. Vektorandmestikus talletatakse info koordinaatide koguna, mille tulemusena saab andmeid kuvada erinevate kujunditena – punkt, joon, polügoon. Rasterkujul salvestatud ruumiandmed on kõige levinumalt seotud piksliga, millel on enamasti küljes üks atribuut. Pikslid on paigutatud maatriksisse ning moodustab üldise võrgustiku [18].



Joonis 1. Vektor- ja rasterandmestik visualiseeritud

„Geograafiline infosüsteem (GIS) on omavahel seotud kogum tarkvarast ja andmetest, mida kasutatakse geograafilise info vaatamiseks ja haldamiseks, ruumiliste seoste analüüsimiseks ning ruumiliste protsesside modelleerimiseks. GIS annab raamistiku ruumiandmete ja nendega seotud informatsiooni kogumiseks ja haldamiseks selliselt, et seda saab visualiseerida ja analüüsida.“ [19]

Geograafilise andmete salvestamiseks on tänapäeval erinevaid viise ja standardeid. Neid saab hoiustada nii relatsioonilistes kui ka mitte relatsioonilistes andmebaasides ja failidena. Mõni andmebaas, nagu MongoDB, on võimeline ruumiandmeid salvestama ja

töötlema vaikimisi [20]. PostgreSQL andmebaasile on võimalik paigaldada PostGIS laiendus, mis lisab andmebaasile võimekuse salvestada ja hallata ruumiandmeid [21].

Peamised faili formaadid vektorandmete salvestamiseks on ESRI Shapefile, Geopackage, MapInfo TAB, GeoJSON. Neid kasutatakse laialt nii lauatarkvaras kui ka veebis andmevahetuseks [22], [23].

Geograafiliste andmete edastamiseks veebis või rakendustes on võimalik kasutada veel erinevaid formaate. Näiteks GML ja KML on XML-il põhinevad geograafilise informatsiooni jaoks edastatavad standardid, mis on välja töötatud OpenGIS Consortiumi poolt [5], [7]. OGC on üks peamisi institutsioone, kes annab oma panuse ruumiandmete standardite väljatöötamisel [24], [25]. Formaate on mitmeid, kuid kõige levinumad on ESRI Shapefile ja GeoJSON [26]. Ka on veel geoandmete vahetamisel kasutusel CSV, kus objekti asukoht on defineeritud läbi koordinaatide või WKT-na [13].

Enim levinud rasterandmete salvestamise formaat on GeoTIFF. Rasterandmete puhul on kasutusel väga erinevaid formaate ning tihti kasutavad erinevat tarkvarad just enda jaoks välja töötatud pildiformaate [23].

2.1 Aktuaalsus ja probleem

Tellijal on palju andmestikke, millel on oluline ruumiline mõõde. Näiteks kiirabiteeninduspiirkonnad, jahipiirkonnad, aadressid. Osasid selliseid andmeid saaks hoiustada mõnes tavapärasemas laialt levinumas formaadis nagu Excel, kuid sellel on mitu piirangut. Üheks suuremaks miinuseks on andmete haldus. Excelis on kasutajal lihtne teha sisestamisvigu. Samuti ei saa Excelis sooritada andmetele sellist auditit nagu tellija seda nõuab. Excel on loodud rohkem lokaalseks kasutamiseks ja muutub aeglaseks suuremate andmekogude puhul [27]. Kui hoiustada andmeid keskses andmebaasis, saab kõik eelpool nimetatud Exceli puudujäägid elimineerida. Andmebaasi ja rakendusega on võimalik suunata kasutajat sisestama õigeid andmeid. Samuti saab andmebaasiga ja rakenduse liidestusega kontrollida rollipõhiselt, kes võib kaardikihti näha, kes seda muuta ning salvestada kasutaja tegevustest auditlogi. Tuleviku vaates, on andmebaasi ja rakenduse abil võimalik ruumiandmed saata ka näiteks MapServerisse. Kohe kui andmed uuenevad, saavad neid uuendusi kasutada ka teised kaardirakendused, mis võtavad oma andmed MapServerist.

Excelis on küll võimalik geoandmeid salvestada ja neid isegi lisa pistikprogrammide abil visualiseerida, näiteks GIS.XL [28] või Excel Power Map [29] abil. Küll aga oma limiteeritud võimekuse tõttu on palju mõistlikum kasutada andmebaasi. Liidestuse loomine aitaks täita seaduses esitatud infoturbe standardeid andmete hoidmisel ja serveerimisel [30]. Spetsiaalne ruumiandmetele mõeldud töökohatarkara teeb inimesele geoandmete visualiseerimise ja halduse mugavaks.

Kui kasutajad vahetaksid andmeid eelpoolmainitud faili formaatides omavahel näiteks ESRI Shapefile formaadis, siis võib tekkida oht erineva versiooni ja ajakohasuse tekkimisele. Lisaks nõuab selline andmete parendamine ja töötlemine pidevat käsitööd ning kommunikatsiooni teiste osapooltega, mis on ka teatud määral turvarisk ning loob ohu ebaadekvaatse ajalise andmete seisu tekkimiseks. Kui antud ruumiandmestikke kasutaks üks kuni paar kasutajat, oleks tavapärased ruumiandmete faili formaadid ka sobilikud. Kasutajate arvu suurenemisel, aga ei oleks võimalik neid failidena, nii Exceliga kui ka GeoJSON-ga, turvaliselt ja korrektselt hallata. Keskne andmebaas aitab neid riske maandada ning loob võimaluse mitmetel kasutajatel tegeleda sama andmestikuga.

2.2 Arendusele esitatavad nõuded ja kasutuslood

- Andmebaasiga suhtlus peab turvanõuete tõttu käima läbi liidestuse
- Kasutaja peab saama uusi regioone lisada
- Kasutaja peab saama importida ruumiandmeid regioonile
- Kasutaja peab saama ruumiandmeid muuta ja salvestada
- Regioonide ja ruumiandmete pärimine peab toimuma kasutajatele õigustepõhiselt

3 Andmed ja metoodika

Käesolevas diplomitöös arendas töö autor kolm erinevat versiooni liidestusest, kasutades kolme erinevat tehnoloogiat – PostGIS, GeoTools ja jts2geojson. Lahenduste analüüsimiseks teostati erinevate lahenduste kiiruse mõõtmised. Kiirust mõõdeti andmete pärimisel ja salvestamisel töökohatarkvarast QGIS 3.12. Andmete pärimine ja salvestamine toimus antud töö käigus loodud pistikprogrammi abil. Mõõtmisel kasutati Python koodis *time* teeki ja *time()* funktsiooni. Mõõtmisi korrati kolm korda. Erinevate lahenduste mõõtmise vahel teostati arvutile taaskäivitus, et välistada kõikumisi taustaprotsesside tõttu tulenevast koormusest. Saadud tulemusi analüüsiti ning valiti vastavalt kiirusele ja tellija poolt esitatud nõuetele sobivaim lahendus.

Mõõtmise teostamiseks kasutati andmestikku, mis on pärit KEMIT-st. Andmestikuks oli jahipiirkonnad, 2020 septembrikuu seisuga. Sisaldas 330 jahipiirkonda, kus iga ala oli kujutatud *MultiPolygon* geomeetriaga.

Töökohatarkvara valikul lähtus autor tellija soovist kasutada QGIS-i. Antud töös kirjeldab ja analüüsib autor siiski ka alternatiive QGIS-le, mille põhjal on võimalik langetada isiklik otsus töökohatarkvara valikul.

4 Tagarakendus

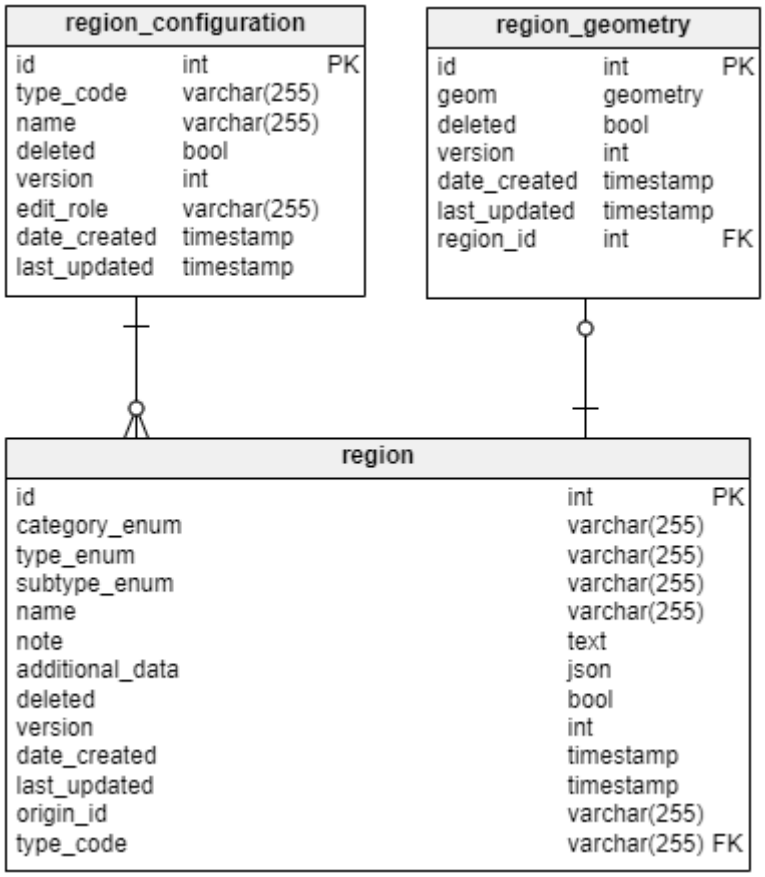
Tagarakenduse ülesandeks on võimaldada geograafiliste alade lisamist, geoandmestiku importimist, serveerimist, muutmist ja kustutamist. Antud töö käigus arendati selline võimekus olemasoleva rakenduse külge lisa funktsionaalsusena. Tagarakendus oli kirjutatud Java 10 tehnoloogial ja Grails 4 raamistikul. Andmebaasiks oli kasutusel PostgreSQL 10.

4.1 Nõuded tagarakendusele

- Liidestusele tehtavad päringud on õigustepõhised
- Liidestus võimaldab uusi regioone lisada
- Liidestus võimaldab importida geoandmeid regioonile
- Liidestus võimaldab geoandmeid salvestada, muuta

4.2 Andmemudel

Antud töö käigus sai lõplikuks andmebaasimudeliks selline lahendus, kus regioonide kohta käiv info salvestatakse *region* tabelisse, ruumikujud *region_geometry* tabelisse ja nimekiri regioonidest ja muutmisõigustest *region_configuration* tabelisse. Andmemudel on välja toodud Joonis 2



Joonis 2. Andmemudel

Geomeetria salvestamise nõue eraldi tabelisse tuli rakenduse arhitektilt. Diplomitöö käigus arendas töö autor ka lahenduse, kus geomeetria oli salvestatud *region* tabelis. Selle tõttu probleeme või kõrvalekaldeid ei esinenud, mistõttu julgeb autor soovitada ruumiandmete hoidmist koos tavaandmetega. Käesoleva probleemi lahenduses oli oluline, et uusi regioone saaks dünaamiliselt, ilma rakenduse koodi muutmata, juurde lisada. Selle jaoks disainis töö autor mudeli *region_configuration* tabeliga. Nimetatud tabelisse on võimalik salvestada regioonide klassifikaatoreid nagu näiteks: *REGION_HUNTING_AREA*. Klassifikaatorite lisamist saab teha veebiliidesest ja neid on võimalik läbi liidestuse pärida, mille vastusest saab järeldada, millised kihid on saadaval. Tavalises *region* tabelis on sama klassifikaatori väli, mille abil saab pärida andmeid spetsiifilise regiooni kohta.

4.3 Regioonide lisamise arendus

Regioonid on näiteks jahipiirkonnad, koristusalad, politseipiirkonnad. Antud alad sisaldavad endas geograafilist komponenti. Näiteks kui kasutaja pärib jahipiirkonnad, siis

tagastatakse talle ruumikujud Eesti jahipiirkondadest ja vajadusel ka kaasas käiv informatsioon kindla piirkonna kohta, nagu vastutav isik.

Regiooni lisamine käib läbi veebiliidese. Veebiliides oli antud diplomitöö käigus juba olemas. Autor täiendas rakendust regiooni lisamise funktsionaalsusega. Uue regiooni salvestamine toimus läbi lihtsa *REST* kontrolleri ja teenusega, mis salvestas andmed andmebaasi.

Regioone saab lisada ja muuta ainult isik, kellel on vastavad õigused. Ala lisamisel, peab sisestama regiooni tüübi/klassifikaatori, mille abil rakenduses erinevaid regioone eristatakse. Samuti peab lisama piirkonna pealkirja ja määrama rolli, ehk õigused, kes võivad antud kihti pärida ja muudatusi teha. Ala lisamise veebiliides on välja toodud Joonis 6

4.4 Ruumiandmete serveerimise arendus

Kuna tagarakenduse olemasolev liidestus serveeris andmeid JSON formaadis, siis sai valitud ka geoandmestiku serveerimiseks JSON-ga sarnane formaat, GeoJSON. Andmestiku pärimiseks andmebaasist ja vastuse kokkuehitamist sai proovitud läbi erinevate lahenduste. Sobiv lahendus valiti andmestiku serveerimise kiiruse ja rakenduse arhitekti nõudmiste põhjal.

4.4.1 PostGIS lahendus

Kuna rakendus kasutas andmebaasiks PostgreSQL 10, sai sinna juurde lisatud PostGIS laiendus, mis võimaldas salvestada geometry tüüpi väljaga geoandmeid. Seejuures tuleb PostGIS laiendusega kaasa palju sisseehitatud andmebaasi funktsioone, mis on abiks geoandmete töötlusel [31]. PostgreSQL andmebaasis on olemas ka *Geometric* väljatüüp, mis võimaldab salvestada ruumiandmeid, kuid on oma võimekuselt limiteeritud. Näiteks ei ole võimalik määrata projektsiooni [32].

Esimene lahendus valmiski tavalise SQL päringuna, kasutades ka PostGIS laienduse abi. Lahenduse tööpõhimõte oli teha tavaline *SELECT* päring vajalikele andmeveergudele ja konverteerida saadud vastus ümber *jsonb* formaati, kasutades selleks PostgreSQL funktsiooni *to_jsonb().* Seejärel sai tehtud PostgreSQL funktsioon *rowjsonb_to_geojson()* [33], mis võttis sisendiks *SELECT* päringust saadud *jsonb*

formaadis vastuse. Andmebaasi funktsiooni kood on välja toodud Lisas 2. Funktsiooni sees kasutati PostGIS funktsiooni *ST_AsGeoJSON()*, mis konverteeris *geometry* väljatüübi GeoJSON formaati. Samuti pandi ka ülejäänud andmeväljad õigesse formaati. Tulemuseks oli GeoJSON kujul vastus, mida sai liidestuse abil nüüd tagastada. Tagarakenduse kood on välja toodud Lisas 3.

4.4.2 GeoTools lahendus

Teise lahenduse jaoks sai proovitud GeoToolsi. GeoTools on vabavaraline, avatud lähtekoodiga Java teek, mis annab vahendid ruumiandmestiku käsitlemiseks [34].

Lahenduseks tegi töö autor ühe Java funktsiooni, mis võttis sisendiks regiooni tüübi ja tagastas GeoTools teegi funktsioone kasutades GeoJSON formaadis vastuse. Tulemi saamiseks, oli vaja GeoToolsil luua ühendus andmebaasiga. Seejärel sai ette anda tabeli nime ja veerud, kust infot pärida. Samuti luua ka filtrid, mille abil kindla regiooni tüübiga andmestik pärida. GeoToolsiga tehtud lahenduse programmikood on välja toodud Lisas 4.

4.4.3 Java + jts2geojson lahendus

Kolmanda lahendusena sai proovitud ka natuke lihtsamat lähenemist võrreldes GeoToolsiga. Kuna GeoJSON-is on *FeatureCollection*, milles on kogum *Feature*-dest, mis on kõik tavalise JSON kujuga, siis tegelikult saab regiooni andmed ja ruumiandmed küsida läbi Java Grails domeeni objekti. Saadud andmed saab sõnedena kokku liita GeoJSON formaati ja siis serveerida. Ainus keerukus seisneb *geometry* andmeväljast vastuse konverteerimine õigesse formaati. Siin kohal saab kasutada *org.wololo.jts2geojson* teegis olevat *GeoJSONWriter* klassi. Mainitud klassis on funktsioon *write()*, mille sisendparameetriks on *geometry* välja väärtus. Funktsiooniga on võimalik tagastada antud lahenduse puhul vajaminevaid koordinaate. Kui kõik andmed päritud ja *Feature*-teks kokku ehitatud, siis on võimalik sõne liitmise abil valmis ehitada *FeatureCollection* ja seda serveerida GeoJSON-na. Valminud lahenduse programmikood asub Lisas 5.

4.5 Ruumiandmete salvestamise arendus

4.5.1 GeoTools lahendus

Kuna geoandmete serveerimisel oli GeoToolsil põhinev lahendus antud diplomitöö käigus proovitud lahendustest kiireim, siis arendas töö autor esimese lahendusena ka salvestamise samal tehnoloogial. Salvestamise funktsioonile antakse sisendiks QGIS-st saadetud GeoJSON, mis loetakse *FeatureCollection* tüüpi muutujasse. Seejärel oli võimalik andmed *id* põhjal jagada kahte *Feature* tüüpi *List*-i – uued ja muudetud andmed. Järgmisena loob GeoTools andmebaasiga ühenduse ning toimub andmete salvestamine. Antud diplomitöö käigus tuli aga GeoToolsi lahendust arendades välja probleem. Tabelis, kuhu GeoTools üritas andmeid salvestada, olid mõned väljad tüübiga *tsvector*. See väljatüüp sundis GeoToolsi andmebaasiühenduse lugemisrežiimi, mistõttu andmeid baasi ei salvestatud. Valminud programmikood on välja toodud Lisas 6.

4.5.2 Lõplik lahendus

GeoToolsi puudujäägist tulenevalt oli vaja autoril välja mõelda teine lahendus. Idee oli kasutada salvestamise jaoks tavalist Java Grails domeeni haldust. Sissetulevad andmed lugeda domeeniobjekti külge, misjärel saata andmed andmebaasi. Sellise lahenduse puhul pole vaja eraldi luua andmebaasi ühendust kuna selle on raamistik juba ära teinud ja kogu suhtlus käib läbi selle kanali. Niimoodi on toetatud ka *tsvector* väljad, mis olid rakenduses olulise tähtsusega, kuid mitte kuidagi seotud käesoleva töö teemaga. Sellise viisi keerukuseks oli sissetulnud ruumiandmete käsitlemine. GeoToolsi abil said ruumiandmed loetud *FeatureCollection* muutujasse, misjärel oli võimalik andmed üle itereerida, domeeniga siduda ning salvestada. Antud lahenduse puhul oli vaja WKT kujul olev geomeetria konverteerida õigele kujule, et seda saaks domeeniga siduda ja andmebaasi salvestada. Selle jaoks sai kasutatud *org.locationtech.jts.io* teegis olevat funktsiooni *WKTReader()*, mis tegi WKT kujul oleva geomeetria *Geometry* tüübiks. Programmikood on välja toodud Lisas 7.

4.6 Lahenduste analüüsimine

Lahenduste analüüsimiseks teostas käesoleva töö autor erinevate lahenduste kiiruse mõõtmised. Kiirust mõõdeti andmete pärimisel ja salvestamisel töökohatarkvarast QGIS 3.12. Andmete pärimine ja salvestamine toimus antud töö käigus loodud pistikprogrammi abil. Mõõtmisel kasutati Python koodis *time* teeki. Aja mõõtmise näidiskood on välja toodud Joonis 3. Mõõdetud ajavahemikud on pärimise või salvestamise funktsiooni väljakutsumisest kuni funktsiooni lõpetamiseni. Funktsiooni sisse jääb tagarakendusele tehtav päringu aeg ja lisaks QGIS-is alade laadimise aeg kaardile.

```
import time

start = time.time()
# do stuff
end = time.time()
print('Time: ' + str(end - start))
```

Joonis 3. Andmete laadimise ajamõõtmise funktsioon

Mõõtmiseks kasutatud andmestik on pärit KEMIT-st, 2020 septembrikuu seisuga. Andmestikuks on jahipiirkonnad, kus on 330 ala, kus iga ala kujutatakse *MultiPolygon* geomeetriaga.

Mõõtmine viidi läbi Windows 10 süsteemil, mille riistvara on järgnev: Intel i7-6600U, 2x10GB RAM 2133MHz, 512GB M.2 mudel AH6661.

Tabel 1 on välja toodud jahipiirkondade pärimise kiirused. Mõõdetud kiirus kujutab olemasoleva regiooni ruumiandmete pärimist tagarakendusest ja seejärel laadimist QGIS kaardile. Mõõdetud kiirustest on näha, et PostGIS lahendus on kolmest proovitud lahendusest kõige aeglasem ja GeoTools on kiireim. Kuigi GeoToolsil põhinev lahendus oli kiireim, sai siiski antud diplomitöös probleemi lahendamiseks kasutatud Java ja *jts2geojson* lähenemist. GeoTools loob andmebaasiga eraldiseisva ühenduse, mida rakenduse arhitekt soovis vältida. 4-5 sekundiline ajakadu suuremahulise andmete pärimisel ei olnud Java ja *jts2geojson* lahenduse puhul probleemiks.

Tabel 1. Jahipiirkondade pärimise kiirus tagarakendusest QGIS-ga

Tehnoloogia	**Mõõtmine 1**	**Mõõtmine 2**	**Mõõtmine 3**
PostGIS	20.8 s	21.3 s	21.7 s
GeoTools	13.7 s	13 s	12.7 s
Java + jts2geojson	17 s	16.8 s	18 s

Tabel 2 on välja toodud jahipiirkondade importimise kiirused. Mõõdetud kiirus kujutab endast regiooni ruumiandmete saatmist QGIS-st tagarakendusse, mis salvestab selle andmebaasi. Seejärel toimub salvestatud andmete pärimine andmebaasist ja kaardile laadimine QGIS-is. Mõõdetud tulemuste põhjal saab järeldada, et GeoToolsil põhinev lahendus on jällegi kiirem. Kuna GeoToolsiga tulid töö tegemise käigus välja rakendust lõhkuvad puudused, sai aga kasutatud Java ja GeoToolsil põhinevat lahendust.

Tabel 2. Jahipiirkondade importimise kiirus

Tehnoloogia	**Mõõtmine 1**	**Mõõtmine 2**	**Mõõtmine 3**
GeoTools	25 s	25.3 s	23.5 s
Java + GeoTools	34.3 s	33.7 s	32.2 s

.

5 Esirakendus

Esirakenduse ülesanne on kasutajatele geograafiliste alade importimine ja ruumikujude muutmine kasutajasõbralikumaks tegemine.

5.1 Nõuded esirakendusele ja pistikprogrammile

- Võimekus liidestuda tagarakendusega
- Kasutaja peab saama end autentida
- Kasutaja peab saama pärida geograafilisi alasid õigustepõhiselt
- Kasutaja peab saama alasid importida
- Kasutaja peab saama alasid muuta
- Kasutaja peab saama alasid kustutada

5.2 Esirakenduse valik ja alternatiivid

Rakendusi, mis spetsialiseeruvad ruumiandmete kuvamisele ja töötlemisele on mitmeid. Mõne eest peab maksma litsentsitasu, nagu ESRI ArcGIS [35] või MapInfo [36]. Tasuliste tarkvarade puhul on positiivne see, et nendega kaasneb ka tehniline tugi [37], [36]. Samuti võivad tasulised rakendused pakkuda paremat kasutajakogemust tänu lihtsamale kasutajaliidesele, spetsiaalsetele tööriistadele, olemasolevale GIS andmestikule ja palju muud [38]. On olemas ka vabavaralisi ja avatud lähtekoodiga rakendusi, nagu QGIS. Kuigi QGIS on vabavaraline, on ta käesoleva töö ilmumise hetkel aktiivses arenduses kogukonna poolt [10]. Kuigi QGIS-ga ei kaasne rakendusele spetsiifilist tehnilist tuge, on võimalik Google abil tänu QGIS-i aktuaalsusele oma probleemidele kerge vaevaga lahendused leida. QGIS on oma vahendite poolest väga võimekas ja hästi dokumenteeritud [39], [40]. Antud diplomitöö raames on suureks plussiks see, et QGIS-i jaoks on võimalik luua pistikprogramme Python-ga, kasutades *Qt* ja *qgis* teeke. Pistikprogrammi näol saab rakendusele juurde lisada väga personaalsetele

nõuetele vastavat võimekust. Teatud määral oleks see ka võimalik litsentsi tarkvaras, aga QGIS-s on avatud lähtekoodi ja üldise pistikprogrammide lähenemise tõttu see märgatavalt lihtsam ning kasutajasõbralikum.

Tellija vaatest on QGIS perfektne lahendus avatud lähtekoodi tõttu. Avatud lähtekood võimaldab näha, kuidas arvutisse paigaldatud rakendus tegelikult käitub. Samuti puudub litsentsitasu, mis hoiab asutusel kulusid madalamal. Kasutaja vaatest on QGIS võimekas ning pistikprogrammide abil laiendatav. Diplomitöö autori vaatest on QGIS-i võimalik juurde arendada väga spetsiifilise nõudmisega võimekust tänu heale dokumentatsioonile ja erinevatele teekidele. Kuna QGIS on vabavaraline, avatud lähtekoodiga aktuaalne lauatarkvara, võimaldab tellimuspõhilise pistikprogrammi arendust ja on familiaarne tellija kasutajatele, sai antud töös ja arenduses valitud just QGIS.

5.3 Esirakenduse pistikprogrammi arendus

QGIS-is on võimalik pistikprogramme arendada Pythonis ja C++ [41]. Arendamise hõlbustamiseks on loodud *Qt* ja *qgis* teegid, mis pakuvad erinevaid funktsioone, mille abil kasutada QGIS tööriistu või ehitada kasutajaliidest [42], [43]. Antud töö raames kirjutati pistikprogramm Pythonis. Põhjuseks on diplomitöö autori vähene kogemus C++ keeles, mille õppimine oleks käesoleva töö probleemi lahenduseks olnud ajaliselt liiga kulukas. Pythoniga on autor varasemalt kokku puutunud.

Pistikprogrammi arenduses sai kasutajaliidese disainimiseks ja genereerimiseks kasutatud Qt Creator tarkvara. Seejärel oli pistikprogrammi koodis võimalik kasutada *Qt* teegi kasutajaliidesega seotud funktsioone ning esile kutsuda muudatusi kasutajaliideses.

Pistikprogrammi kasutamiseks on vajalik kasutajal end autentida. Antud töö raames tuli autoril pistikprogramm liidestuda autentimisliidesega. Seejärel oli võimalik tagarakenduselt *GET* päringuga vastavalt kasutaja õigustele küsida regioonide nimistu ning kuvada nimekiri kasutajaliideses *Qt* teegi *addItem()* funktsiooniga. Regioonide nimekirja laadimise funktsioon on välja toodud Joonis 4.

```
def loadLayers(self):
	# Get layers (names and typeCodes)
	self.getRegionConfiguration()
	# Add layers to UI list
	self.layerListUI.clear()
	if self.json:
		for layer in self.json:
			item = QtWidgets.QListWidgetItem()
			item.setText(layer['name'])
			item.setData(1, layer['typeCode'])
			self.layerListUI.addItem(item)
		self.show()
```

Joonis 4. Kood regioonide nimekirja pärimiseks ja laadimiseks kasutajaliidesesse

Kui kasutaja on kihi või kihid valinud, saab ta alustada regiooni laadimist kaardile. Regiooni laadimisel teostatakse *GET* päring tagarakendusele, kus antakse päringuga kaasa regiooni klassifikaator, mille abil saab kindla regiooni ruumiandmed pärida. Andmed laetakse QGIS-i vektorkihina *qgis* teegi abil, kasutades selleks *QgsVectorLayer()* funktsiooni. Ruumiandmete pärimine ja vektorkihi initsialiseerimine toimuvad funktsioonides *downloadAndInitLayer()* ja *initLayers()*, mis on leitavad Lisas 8. Seejärel saab vektorkihi kaardile laadida, kasutades autori poolt loodud *ProjectLayer* klassi ja seal olevat *loadLayerAsProject()* funktsiooni, mida võib näha Joonis 5.

```
def loadLayerAsProject(self):
	self.memLayer = QgsVectorLayer("multipolygon?crs=epsg:4326",
self.layerName, "memory")
	tempFeatures = [feat for feat in self.vlayer.getFeatures()]
	# initialize dataprovider, so we can query which fields are present
	memLayerData = self.memLayer.dataProvider()
	# get feature fields
	attr = self.vlayer.dataProvider().fields().toList()
	memLayerData.addAttributes(attr) # add fields to data
	self.memLayer.updateFields()  # update memLayer with fields
	memLayerData.addFeatures(tempFeatures)  # add features to data
	# Custom type to differentiate between custom and user personal layers
	self.memLayer.setCustomProperty('type', 'custom')
	# Add this memory layer to project for editing
	QgsProject.instance().addMapLayer(self.memLayer)
	# Add layer id so we can differentiate between instances later
	self.layerId = self.memLayer.id()
	# In memory layer where modified and new features will be added
	self.loadModifiedFeaturesLayer(self.layerName, attr)
	# Hide fields we don't want user to see or modify
	self.hideFields()
	self.layerColor = self.memLayer.renderer().symbol().color()
```

Joonis 5. Kood ruumiandmete lisamisest QGIS kaardile

Kaardikihi infot – uued, eemaldatud, muudetud ruumiandmed, regiooni klassifikaator, hoiustatakse *ProjectLayer* klassis, mille kood on välja toodud Lisas 9. Samas klassis on ka funktsioonid, mis kuulavad muutusi kaardil ja on abiks kaardikihi laadimisel tagarakendusse.

Kaardikihi üleslaadimisel vajutab kasutaja selleks üleslaadimise nuppu. Funktsioon *loadProjectLayers()* kogub kokku kihid, millel on tehtud muudatusi. Järgmisena kuvatakse need kasutajale nimekirja, kust kasutaja saab neid ühe või mitmekaupa andmebaasi salvestada. Nimekirjast valitud kihtides olevad andmed konverteeritakse *qgis* teegis oleva *QgsJsonExporter()* abil GeoJSON formaati, funktsioonis *uploadLayer()*. Seejärel saadetakse andmed *POST* päringuga tagarakendusse, mis salvestab need andmebaasi. Kood on välja toodud Lisas 10.

6 Tulemus ja testimine

Käesoleva diplomitöö raames valmis liidestus ja pistikprogramm, mis täidavad kõik töös püstitatud nõuded.

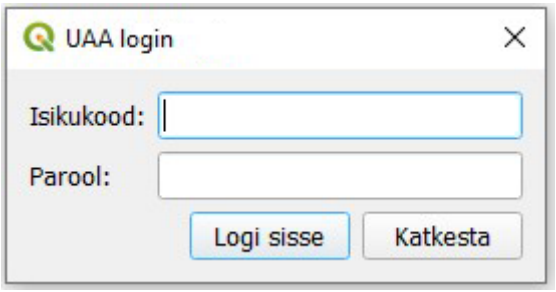
6.1 Uue regiooni lisamine dünaamiliselt

Geograafiliste regioonide haldamise puhul oli tähtis, et oleks võimalik lisada uusi regioone dünaamiliselt. Regioonide lisamine sai võimalikuks läbi veebiliidese vormi, mis on näha Joonis 6. Vastavate õigustega isik defineerib liidese kaudu regiooni klassifikaatori. Klassifikaatori abil on võimalik regioone üksteisest eristada. Defineeritakse ka illustratiivne nimi ja määratakse *Muutja* roll, mis määrab õigused, kes saab regiooni ruumiandmeid näha ja hallata.

Joonis 6. Regiooni klassifikaatori lisamine veebiliidesest

6.2 Regioonide pärimine

Regioone on kasutajal võimalik pärida läbi töö raames valminud pistikprogrammi. Selleks on esimesena vajalik kasutaja autentimine, mida saab pistikprogrammi abil QGIS-is teha, näidatud Joonis 7.



Joonis 7. Kasutaja autentimine läbi QGIS pistikprogrammi

Pärast kasutaja autentimist, kuvatakse talle nimekiri regioonidest, mida ta on autoriseeritud nägema ja muutma, näidatud Joonis 8.

Joonis 8. Nimekiri kasutaja halduses olevatest regioonidest

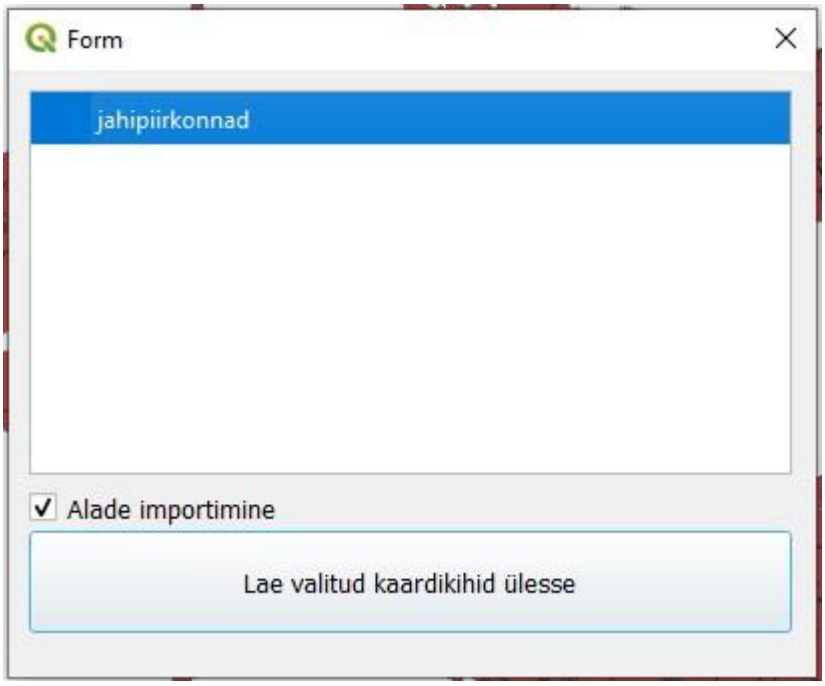
6.3 Ruumiandmete pärimine

Tagarakenduse liidestuse andmete päringu osa sai arendatud kasutades Grails domeeni haldust ja ruumiandmete konverteerimiseks *jts2geojson* teeki. Antud lahendus ei olnud testitud lahendusest kiireim, kuid sobis rohkem rakenduse arhitektuuriga. GeoToolsil põhinev lahendus tekitas andmebaasiga eraldi ühenduse, mida rakenduse arhitekt soovis vältida.

Kasutaja algatab ruumiandmete pärimist läbi QGIS-i pistikprogrammi. Valides pistikprogrammis regiooni, laetakse ruumiandmed QGIS-i, lisatakse regioon kihtide loetellu ja alad joonistatakse kaardivaatesse.

6.4 Ruumiandmete salvestamine

Kui kasutaja on soovitud muudatused kaardikihil ära teinud, on võimalik tehtud muudatused salvestada andmebaasi. Selleks tuleb vajutada üleslaadimise nuppu, valida kiht, mille andmeid salvestada soovitakse ja vajutada nupul *Lae valitud kaardikihid ülesse*, näidatud Joonis 9.



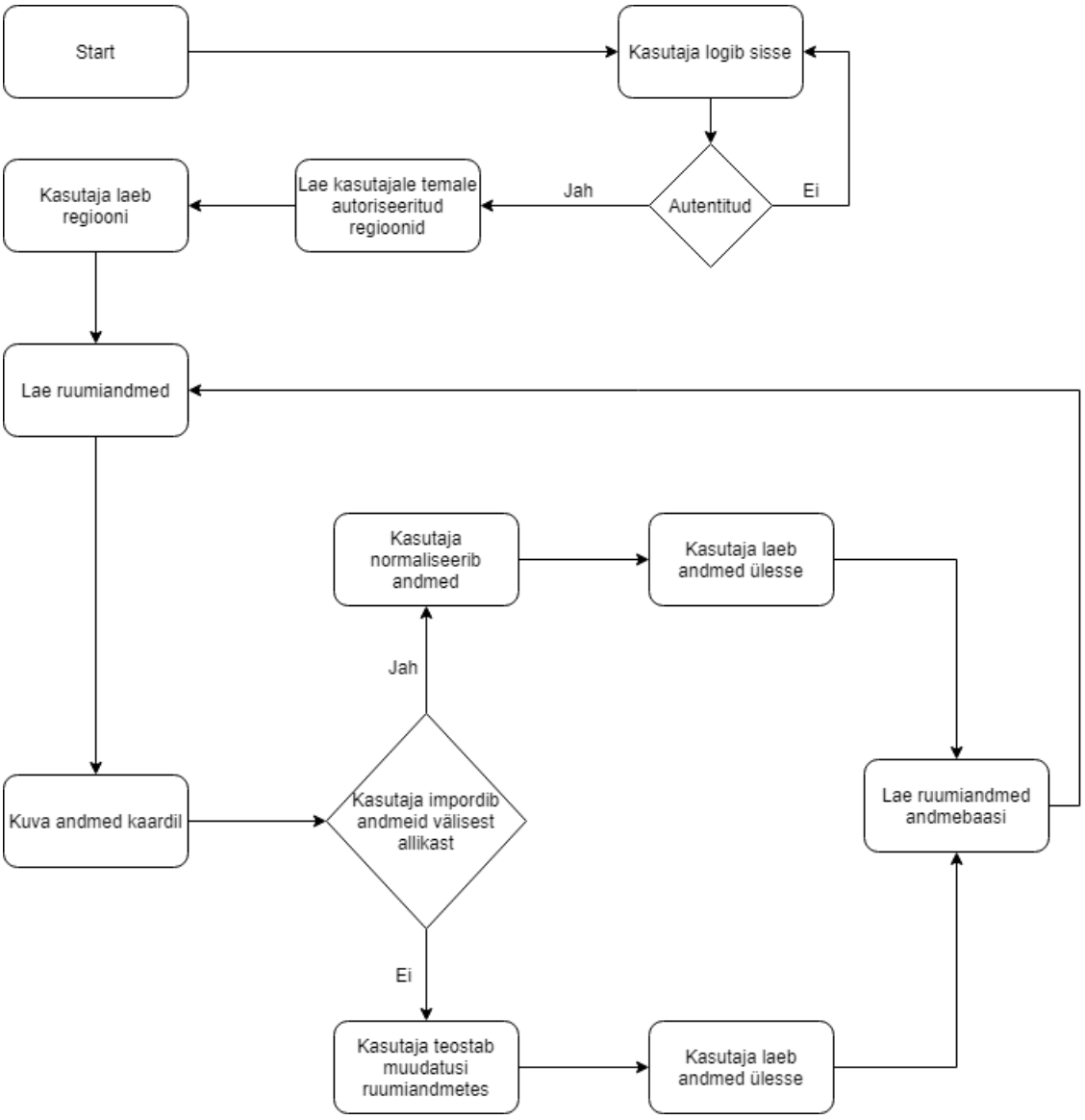
Joonis 9. Kihil tehtud muudatuste üleslaadimine andmebaasi

Andmed salvestatakse andmebaasi, täidetakse auditi nõue, salvestades ka kihil tehtud muudatuste logi ja muud.

6.5 Skeem kasutaja ja esirakenduse töövoost

Joonis 10 on lihtsustatult näidatud, kuidas näeb välja kasutaja ja esirakenduse töövoog. Töövoog jaguneb vahepeal kaheks, olenevalt andmete päritolust. Näiteks võib andmestik pärit olla KEMIT-st. Sel juhul haldab KEMIT alade piirjooni, nimetusi ja muud. Kuna välisest allikast tulnud andmed võivad erineda, peab kasutaja manuaalselt väljanimesid ja andmeid vajadusel normaliseerima. Töö autor suutis jahipiirkondade importimisel

andmed QGIS-is normaliseerida 2 minutiga. Kui importi ei toimu, vaid muudetakse asutuse enda poolt loodud regiooni ruumiandmeid, siis on kasutaja töövoog rakenduses lihtsam. QGIS tööriistadega viiakse läbi muudatused, misjärel kasutaja need ülesse laeb.



Joonis 10. Kasutaja ja esirakenduse töövoo skeem

7 Kokkuvõte

Diplomitöö käigus arendas töö autor Java Grails, GeoTools, *jts2geojson*, PostGIS tehnoloogiatel põhineva liidestuse, mis võimaldab õigustepõhiste ruumiandmete vahendamist teistele rakendustele. Samuti sai arendatud ka QGIS töökohatarkvara jaoks pistikprogramm, mis kasutab seda liidest. Pistikprogrammi abil on kasutajal võimalik end autentida ning hallata ainult temale määratud regioone. QGIS teeb oma kaardi tööriistadega alade muutmise ja visualiseerimise kasutajasõbralikumaks. Diplomitöö tulemus hõlbustab kasutajatel ruumiandmeid hallata ja neid ajakohasena hoida. Kasutatud tehnoloogiad ja töökohatarkvara olid diplomitöö loomise hetkel vabavaralised, mis võimaldab ettevõtetel kulud madalamal hoida.

Edasiarendusena saaks ruumiandmeid saata MapServerisse, kus nende andmete põhjal saaks ehitada kaardikihi. Valminud kaardikihti oleks võimalik serveerida teistele teenustele üle veebi. Kuna ruumiandmeid hoiustatakse ühes kohas ja nende põhjal toimub MapServeris kaardikihi ülesehitamine, siis andmete uuendamisel uueneks kihid automaatselt ka kõikides teistes teenustes, mis antud kaardikihi infot oma rakenduses kuvada soovivad. Edasiarendusena oleks vaja ka uurida võimalusi pistikprogrammi üleslaadimist repositooriumi, kus kasutajal oleks seda lihtne endale QGIS-i paigaldada. Samuti saaks lihtsustada kasutaja töövoogu, automatiseerides väljade normaliseerimist, kui toimub välisest andmeallikast uue regiooni ruumiandmete salvestamine andmebaasi.

Kasutatud kirjandus

[1] C. Hope, „ASCII,“ 06 03 2020. [Võrgumaterjal]. Available: https://www.computerhope.com/jargon/a/ascii.htm. [Kasutatud 04 12 2020].

[2] C. Hoffman, „What Is a CSV File, and How Do I Open It?,“ 17 04 2018. [Võrgumaterjal]. Available: https://www.howtogeek.com/348960/what-is-a-csv-file-and-how-do-i-open-it/. [Kasutatud 04 12 2020].

[3] „GeoJSON,“ [Võrgumaterjal]. Available: https://geojson.org/. [Kasutatud 04 12 2020].

[4] „HTTP Request Methods,“ [Võrgumaterjal]. Available: https://www.w3schools.com/tags/ref_httpmethods.asp. [Kasutatud 04 12 2020].

[5] „Geography Markup Language,“ [Võrgumaterjal]. Available: https://www.ogc.org/standards/gml. [Kasutatud 13 09 2020].

[6] „What is JSON?,“ [Võrgumaterjal]. Available: https://www.w3schools.com/whatis/whatis_json.asp. [Kasutatud 04 12 2020].

[7] „KML,“ [Võrgumaterjal]. Available: https://www.ogc.org/standards/kml/. [Kasutatud 09 13 2020].

[8] „Welcome to MapServer,“ [Võrgumaterjal]. Available: https://mapserver.org/. [Kasutatud 04 2020 2020].

[9] „About OGC,“ [Võrgumaterjal]. Available: https://www.ogc.org/about. [Kasutatud 04 12 2020].

[10] „QGIS GitHub,“ [Võrgumaterjal]. Available: https://github.com/qgis/QGIS. [Kasutatud 20 11 2020].

[11] „What is REST?,“ [Võrgumaterjal]. Available: https://www.codecademy.com/articles/what-is-rest. [Kasutatud 01 01 2021].

[12] „Extensible Markup Language (XML),“ [Võrgumaterjal]. Available: https://www.w3.org/XML/. [Kasutatud 04 12 2020].

[13] „Well-known text representation of geometry,“ [Võrgumaterjal]. Available: https://en.wikipedia.org/wiki/Well-known_text_representation_of_geometry. [Kasutatud 04 12 2020].

[14] C. Dempsey, „What is the Difference Between GIS and Geospatial?,“ 14 02 2014. [Võrgumaterjal]. Available: https://www.gislounge.com/difference-gis-geospatial/. [Kasutatud 12 09 2020].

[15] „Andmete vektorkuju,“ [Võrgumaterjal]. Available: http://www.geo.ut.ee/kooligeo/EGCD/opik/juts/gis/vektor.html. [Kasutatud 03 12 2020].

[16] „Andmete rasterkuju,“ [Võrgumaterjal]. Available: http://www.geo.ut.ee/kooligeo/EGCD/opik/juts/gis/raster.html. [Kasutatud 03 12 2020].

[17] „GIS & Geospatial Data,“ [Võrgumaterjal]. Available: https://libguides.library.arizona.edu/GIS/about-gis. [Kasutatud 12 09 2020].

[18] „What is raster data?,“ [Võrgumaterjal]. Available: https://desktop.arcgis.com/en/arcmap/10.3/manage-data/raster-and-images/what-is-raster-data.htm#:~:text=In%20its%20simplest%20form%2C%20a,pictures%2C%20or%20even%20scanned%20maps. [Kasutatud 03 12 2020].

[19] „Mis on GIS?,“ [Võrgumaterjal]. Available: http://www.gispaev.ee/avaleht/mis-on-gis/. [Kasutatud 12 09 2020].

[20] „Geospatial queries,“ [Võrgumaterjal]. Available: https://docs.mongodb.com/manual/geospatial-queries/. [Kasutatud 12 09 2020].

[21] „PostGIS,“ [Võrgumaterjal]. Available: https://postgis.net/. [Kasutatud 12 09 2020].

[22] Simon, „Why you should use GeoPackage instead of Shapefile,“ 13 03 2018. [Võrgumaterjal]. Available: https://www.gis-blog.com/geopackage-vs-shapefile/. [Kasutatud 12 09 2020].

[23] „GIS Data,“ [Võrgumaterjal]. Available: https://mangomap.com/gis-data. [Kasutatud 12 09 2020].

[24] „GIS Standards,“ [Võrgumaterjal]. Available: https://www.gistandards.eu/gis-standards/. [Kasutatud 12 09 2020].

[25] „A Look at Standards,“ 01 01 2013. [Võrgumaterjal]. Available: https://www.esri.com/about/newsroom/arcnews/a-look-at-standards/. [Kasutatud 13 09 2020].

[26] „The Ultimate List of GIS Formats and Geospatial File Extensions,“ 06 03 2020. [Võrgumaterjal]. Available: https://gisgeography.com/gis-formats/. [Kasutatud 13 09 2020].

[27] A. Hershy, „Excel vs SQL: A Conceptual Comparison,“ 03 08 2019. [Võrgumaterjal]. Available: https://towardsdatascience.com/excel-vs-sql-a-conceptual-comparison-dcfbee640c83. [Kasutatud 13 09 2020].

[28] „Spatial Data & Maps in Excel,“ [Võrgumaterjal]. Available: http://gisxl.com/. [Kasutatud 13 09 2020].

[29] „Get started with Power Map,“ [Võrgumaterjal]. Available: https://support.microsoft.com/en-us/office/get-started-with-power-map-88a28df6-8258-40aa-b5cc-577873fb0f4a. [Kasutatud 13 09 2020].

[30] „ISKE juhendid ja materjalid,“ [Võrgumaterjal]. Available: https://www.ria.ee/et/kuberturvalisus/iske/juhendid-ja-materjalid.html. [Kasutatud 27 12 2020].

[31] „PostGIS reference,“ [Võrgumaterjal]. Available: https://postgis.net/docs/reference.html. [Kasutatud 16 11 2020].

[32] „Geometric Types,“ [Võrgumaterjal]. Available: https://www.postgresql.org/docs/10/datatype-geometric.html. [Kasutatud 16 11 2020].

[33] P. Ramsey, „GeoJSON Features from PostGIS,“ 27 03 2019. [Võrgumaterjal]. Available: http://blog.cleverelephant.ca/2019/03/geojson.html. [Kasutatud 16 11 2020].

[34] „GeoTools The Open Source Java GIS Toolkit,“ [Võrgumaterjal]. Available: https://www.geotools.org/. [Kasutatud 16 11 2020].

[35] „ArcGIS Online Pricing,“ [Võrgumaterjal]. Available: https://www.esri.com/en-us/arcgis/products/arcgis-online/buy?rmedium=esri_com_regex&rsource=arcgis-online. [Kasutatud 20 11 2020].

[36] „MapInfo Pro,“ [Võrgumaterjal]. Available: https://www.precisely.com/product/precisely-mapinfo/mapinfo-pro. [Kasutatud 19 10 2020].

[37] „Esri Support Services,“ [Võrgumaterjal]. Available: https://support.esri.com/en/other-resources/SupportServices. [Kasutatud 20 11 2020].

[38] „The Differences Between QGIS and ArcGIS,“ 03 01 2021. [Võrgumaterjal]. Available: https://gisgeography.com/qgis-arcgis-differences/. [Kasutatud 06 01 2021].

[39] „QGIS Features,“ [Võrgumaterjal]. Available: https://docs.qgis.org/3.16/en/docs/user_manual/preamble/features.html. [Kasutatud 20 11 2020].

[40] „QGIS Documentation,“ [Võrgumaterjal]. Available: https://www.qgis.org/en/docs/index.html. [Kasutatud 20 11 2020].

[41] „Development in QGIS,“ [Võrgumaterjal]. Available: https://www.qgis.org/en/site/getinvolved/development/development.html. [Kasutatud 20 11 2020].

[42] „the QGIS Python API documentation,“ [Võrgumaterjal]. Available: https://qgis.org/pyqgis/3.6/. [Kasutatud 20 11 2020].

[43] U. Gandhi, „Building a Python Plugin (QGIS3),“ [Võrgumaterjal]. Available: https://www.qgistutorials.com/en/docs/3/building_a_python_plugin.html. [Kasutatud 20 11 2020].

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

Mina, Indrek Nurja

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose „Liidestuse ja töökohatarkvara pistikprogrammi arendus geograafiliste alade haldamiseks“, mille juhendaja on Meelis Antoi
   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.

06.01.2021

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 – PostGIS andmebaasikood ruumiandmete pärimiseks

```
CREATE OR REPLACE FUNCTION geo.rowjsonb_to_geojson(
  rowjsonb jsonb,
  geom_column text DEFAULT 'geometry'::text)
  RETURNS text
  LANGUAGE 'plpgsql'

  COST 100
  IMMUTABLE STRICT
AS $BODY$
DECLARE
  json_props jsonb;
  json_geom jsonb;
  json_type jsonb;
BEGIN
  IF NOT rowjsonb ? geom_column THEN
    RAISE EXCEPTION 'geometry column ''%'' is missing', geom_column;
   END IF;
  json_geom := ST_AsGeoJSON((rowjsonb ->> geom_column)::geometry)::jsonb;
  json_geom := jsonb_build_object('geometry', json_geom);
  json_props := jsonb_build_object('properties', rowjsonb - geom_column);
  json_type := jsonb_build_object('type', 'Feature');
  return (json_type || json_geom || json_props)::text;
END;
$BODY$;
```

Lisa 3 – PostGIS programmikood ruumiandmete pärimiseks

```
String getGeoJson(String typeCode) {
	def sql = Sql.newInstance(dataSource)
	def result = ''
	sql.eachRow('SELECT geo.rowjsonb_to_geojson(' +
		'to_jsonb(sub), \'geometry\'::text) as feature FROM' +
		'(SELECT id, category_enum, type_enum, name, deleted,' +
		'version, date_created, creator_session_name,' +
		'creator_session_id_code, last_updated, editor_session_name,' +
		'editor_session_id_code, geometry, origin_id' +
		'FROM geo.region WHERE type_code = :type_code' +
		'and deleted = false) sub;',
		type_code: typeCode) { GroovyResultSet rs ->
		if (rs.isFirst() && rs.isLast() || rs.isLast()) {
			result += rs.getString('feature')
		} else {
			result += rs.getString('feature') + ','
		}
	}
	sql.close()
	return '{"type": "FeatureCollection","features": [' + result + ']}'
}
```

Lisa 4 – GeoTools programmikood ruumiandmete pärimiseks andmebaasist

```
String getGeoJson(String typeCode) {
	Map<String, Object> connectionParams = initDevConnectionParams()
	DataStore dataStore = DataStoreFinder.getDataStore(connectionParams)

	// Get specified properties from table using filter and build a
	// featurecollection
	SimpleFeatureSource source = dataStore.getFeatureSource('region')
	String[] props = ['geometry', 'name', 'deleted',
			'date_created','last_updated','version',
			'creator_session_id_code', 'creator_session_name',
			'editor_session_id_code','editor_session_name']
	FilterFactory ff = CommonFactoryFinder.getFilterFactory(null)
	Filter deletedFilter = ff.equals(
				ff.property('deleted'),
				ff.literal(false)
				)
	Filter typeCodeFilter = ff.equals(
				ff.property('type_code'),
				ff.literal(typeCode)
				)
	Filter andFilter = ff.and(deletedFilter, typeCodeFilter)
	Query query = new Query('region', andFilter, props)
	SimpleFeatureCollection simpleFeatureCollection =
					source.getFeatures(query)

	// Return features from simpleFeatureCollection as GeoJSON
	FeatureJSON featureJSON = new FeatureJSON()
	featureJSON.setEncodeNullValues(true) //include null value props
	String geojson
	// If we dont have any features yet, create one fake feature
	// This is needed so all the necessary properties/fields
	// are sent to QGIS
	// Otherwise when adding a new feature in QGIS,
	// the feature will not have any properties
	if (simpleFeatureCollection.size() > 0) {
		println 'size more than 0'
		geojson = featureJSON.toString(simpleFeatureCollection)
	} else {
		final SimpleFeatureType TYPE =
			DataUtilities.createType(
				'Region',
```

```
            'geometry:Point,'
                  +
                  'name:String,'
                  +
                  'deleted:Boolean,'
                  +
                  'date_created:java.util.Date,'
                  +
                  'last_updated:java.util.Date,'
                  +
                  'version:Integer,'
                  +
                  'creator_session_id_code:String,'
                  +
                  'creator_session_name:String,'
                  +
                  'category_enum:String,'
                  +
                  'type_enum:String,'
                  +
                  'subtype_enum:String,'
                  +
                  'type_code:String,'
                  +
                  'editor_session_id_code:String,'
                  +
                  'editor_session_name:String'
      )
List<SimpleFeature> features = []
GeometryFactory geometryFactory = new GeometryFactory()
SimpleFeatureBuilder featureBuilder =
                                new SimpleFeatureBuilder(TYPE)

Point point = geometryFactory.createPoint(
                                new Coordinate(28.26, 59.12)
                                )
featureBuilder.add(point)
featureBuilder.add('')
featureBuilder.add(false)
featureBuilder.add(new Date())
featureBuilder.add(new Date())
featureBuilder.add(0)
featureBuilder.add('')
featureBuilder.add('')
featureBuilder.add('')
featureBuilder.add('')
SimpleFeature feature = featureBuilder.buildFeature(null)
features.add(feature)

SimpleFeatureCollection collection =
                  new ListFeatureCollection(TYPE, features);
```

```
            geojson = featureJSON.toString(collection)
        }
        dataStore.dispose()
        return geojson
}
```

Lisa 5 – Java ja jts2geojson programmikood ruumiandmete pärimiseks

```
String getGeoJson(String typeCode) {
	def regions = Region.createCriteria().list {
		and {
			eq('deleted', false)
			eq('typeCode', typeCode)
		}
		fetchMode 'regionGeometry', FM.JOIN
	}

	GeoJSONWriter writer = new GeoJSONWriter()
	def features = regions.collect() {
		[
			'type':     'Feature',
			'geometry': ['type' : 'MultiPolygon',
				     'coordinates': writer.write(
				     (MultiPolygon)it?.regionGeometry?.geom)
				     .getCoordinates()
			],
			'properties': [
				'id': it?.id,
				'name': it?.name ?: '',
				'deleted': it?.deleted,
				'origin_id': it?.originId,
				'version': it?.version,
				'type_enum': it?.typeEnum,
				'date_created': it?.dateCreated,
				'last_updated': it?.lastUpdated,
				'category_enum': it?.categoryEnum,
				'subtype_enum': it.subtypeEnum
			]
		]
	}

	// Create a fake feature if there are no features under correlating
	// region type. Fake feature enables user
	// to add properties to the first feature that will be added into the
	// region type in QGIS
	if (features.size() < 1) {
		features = [[
			'type':     'Feature',
			'geometry': ['type' : 'MultiPolygon', 'coordinates': []],
```

```
                'properties': [
                        'id': -1,
                        'name': 'fakeFeature',
                        'deleted': false,
                        'origin_id': null,
                        'version': 0,
                        'type_enum': null,
                        'date_created': null,
                        'last_updated': null,
                        'category_enum': null,
                        'subtype_enum': null
                ]
        ]]
    }

    def featureCollection = '{"type": "FeatureCollection",
                        "features":' + JsonOutput.toJson(features) + '}'
    return featureCollection
}
```

Lisa 6 – GeoTools programmikood ruumiandmete salvestamiseks

```
def save(Object json, Map params) {
	// Convert incoming geojson to featureCollection
	String str = json
	InputStream is = new ByteArrayInputStream(str.bytes)
	FeatureJSON fJSON = new FeatureJSON()
	FeatureCollection featureCollection = fJSON.readFeatureCollection(is)

	// Separate new and modified features into separate Lists
	List<SimpleFeature> newFeatures = []
	List<SimpleFeature> modifiedFeatures = []
	SimpleFeatureIterator iterator =
			featureCollection.features() as SimpleFeatureIterator
	try {
		while (iterator.hasNext()) {
			SimpleFeature feature = (SimpleFeature) iterator.next()
			if (feature.getProperty('id').value == null) {
				feature.setAttribute('date_created', new Date())
				feature.setAttribute('last_updated', new Date())
				feature.setAttribute('creator_session_name',
							params.userFullName)
				feature.setAttribute('creator_session_id_code',
							params.userIdCode)
				feature.setAttribute('editor_session_name',
							params.userFullName)
				feature.setAttribute('editor_session_id_code',
							params.userIdCode)
				feature.setAttribute('category_enum',
					RegionCategoryEnum.REGION_CATEGORY_REGION)
				feature.setAttribute('type_enum',
					RegionTypeEnum.REGION_TYPE_ENVIRONMENT)
				feature.setAttribute('subtype_enum',
					RegionSubtypeEnum
						.REGION_SUBTYPE_HUNTING_AREA
					)
				feature.setAttribute('type_code',
					'REGION_HUNTING_AREA')
				feature.setAttribute('version', Integer.valueOf(1))
				feature.setAttribute('deleted', Boolean.FALSE)
				newFeatures.add(feature)
			} else if (feature.getProperty('id').value != null) {
				feature.setAttribute('last_updated', new Date())
```

```
                    feature.setAttribute('editor_session_name',
                                        params.userFullName)
                    feature.setAttribute('editor_session_id_code',
                                        params.userIdCode)
                    feature.setAttribute('version',
                                (int) feature.getAttribute('version')
                                + 1)
                    modifiedFeatures.add(feature)
             }
       }
} finally {
       iterator.close()
}

// Connect to database and a table
Map<String, Object> connectionParams = initDevConnectionParams()
DataStore dataStore = DataStoreFinder.getDataStore(connectionParams)
SimpleFeatureType featureType =
                    dataStore.getFeatureSource('region').schema
FeatureSource featureSource = dataStore.getFeatureSource('region')

// Check for write access and then add and modify features
if (featureSource instanceof FeatureStore) {
       SimpleFeatureStore featureStore =
                           (SimpleFeatureStore) featureSource

       // Create feature collections from new and modified
       // feature lists
       // These are used for saving to database
       SimpleFeatureCollection newFeaturesCollection =
              new ListFeatureCollection(featureType, newFeatures)
       SimpleFeatureCollection modifiedFeaturesCollection =
              new ListFeatureCollection(featureType, modifiedFeatures)

       Transaction t = new DefaultTransaction(
                    'Adding/Modifying features in table: region'
                    )
       featureStore.setTransaction(t)
       try {
              // Add new features to database
              if (newFeatures.size() > 0) {
                    featureStore.addFeatures(newFeaturesCollection)
              }

              // Modify existing features in database
              // Filter gets feature from database by ID
              // Change features on that ID
              FilterFactory ff =
                    CommonFactoryFinder.getFilterFactory(null);
              if (modifiedFeatures.size() > 0) {
                    SimpleFeatureIterator it =
```

```
        modifiedFeaturesCollection.features()
try {
        while (it.hasNext()) {
                SimpleFeature modifiedFeature =
                        (SimpleFeature) it.next()
                // Build filter for filtering
                // features with ID
                Filter idFilter = ff.id(
                        Collections.singleton(
                                ff.featureId(
                                modifiedFeature
                                .getProperty('id')
                                .value
                                .toString()
                                )
                        )
                )
                // Set fields to be updated, get
                // values and update feature based
                // on feature ID
                String[] names = [
                        'name',
                        'geometry',
                        'last_updated',
                        'editor_session_name',
                        'editor_session_id_code',
                        'version'
                ]
                Object[] attributes = [
                        modifiedFeature
                                .getAttribute('name'),
                        modifiedFeature
                                .getAttribute(
                                'geometry'
                                ),
                        modifiedFeature
                                .getAttribute(
                                'last_updated'
                                ),
                        modifiedFeature
                                .getAttribute(
                                'editor_session_name'
                                ),
                        modifiedFeature
                                .getAttribute(
                                'editor_session_id_code
                                ),
                        modifiedFeature
                                .getAttribute('version')
                ]
```

```
                                        featureStore.modifyFeatures(
                                              names, attributes, idFilter
                                              )
                                  }
                            } finally {
                                  it.close()
                            }
                      }
                      t.commit()
                } catch (IOException ex) {
                      println 'Error while writing features to database: '
                                  + (ex.printStackTrace() as CharSequence)
                      t.rollback()
                } finally {
                      t.close()
                }
          }
          dataStore.dispose()
          is.close()
}
```

Lisa 7 – Java ja GeoTools programmikood ruumiandmete salvestamiseks

```
def saveGeoJson(Object json, Map params) {
	// Read incoming GeoJSON into a FeatureCollection where we can easily
	// go over all the features, attributes, values
	FeatureCollection featureCollection = geoJsonToFeatureCollection(json)

	FeatureIterator iterator = featureCollection.features()
	try {
		while (iterator.hasNext()) {
			SimpleFeature feature = (SimpleFeature) iterator.next()
			MultiPolygon mPoly =
				wktGeomToMultiPolygon(
					feature.getAttribute('geometry').toString(),
					4326)
			// No id then save as new region, existing id then update
			// origin_id check is for preventing accidental import
			// when user forgot to choose import mode
			// (imported new features have origin_id,
			// new features drawn by user don't)
			if (feature.getAttribute('id') == null &&
				feature.getAttribute('origin_id') == null) {
				def uuid = UUID.randomUUID()
				def region = new Region()
				region.id = uuid
				region.categoryEnum =
					RegionCategoryEnum.REGION_CATEGORY_REGION
				region.typeEnum =
					RegionTypeEnum.REGION_TYPE_ENVIRONMENT
				region.subtypeEnum = getRegionSubtypeEnum(
					feature.getAttribute('subtype_enum')
					?.toString(),
					region)
				region.typeCode = params?.typeCode?.toString()
				region.name = feature.getAttribute('name')
				region.originId = feature.getAttribute('origin_id')
				region.deleted = false
				def regionGeom = new RegionGeometry()
				regionGeom.id = UUID.randomUUID()
				regionGeom.geom = mPoly
				region.regionGeometry = regionGeom

				region.save(flush: true)
```

```
                        regionGeom.save(flush: true)
                    } else if (feature.getAttribute('id') != null) {
                        Region region =
                            Region.get(
                                feature.getAttribute('id').toString()
                                )
                        RegionGeometry regionGeom =
                                RegionGeometry.findByRegion(region)
                        regionGeom.geom = mPoly
                        region.regionGeometry = regionGeom
                        region.name =
                            feature.getAttribute('name')?.toString()
                                == '' ? '' :
                            feature.getAttribute('name')?.toString()
                                ?: region.name
                        region.subtypeEnum = getRegionSubtypeEnum(
                                feature.getAttribute('subtype_enum')
                                ?.toString(), region
                        )
                        region.save(flush: true)
                        regionGeom.save(flush: true)
                    }
                }
        } catch (Exception ex) {
                LOG.error('Error while saving Region: ', ex)
        } finally {
                iterator.close()
        }
}

private static MultiPolygon wktGeomToMultiPolygon(String geometry, Integer
srid) {
        //Build Geometry object from which we can build MultiPolygon
        WKTReader reader = new WKTReader()
        Geometry jtsGeom = null
        jtsGeom = reader.read(geometry)

        //Build MultiPolygon from Geometry
        //If Geometry is Polygon, promote it to MultiPolygon
        MultiPolygon mPoly = null
        GeometryFactory geometryFactory = new GeometryFactory()
        if (jtsGeom instanceof Polygon) {
                Polygon[] polys = new Polygon[1]
                polys[0] = (Polygon) jtsGeom
                mPoly = geometryFactory.createMultiPolygon(polys)
        } else {
                mPoly = (MultiPolygon) jtsGeom
        }
        mPoly.setSRID(srid)
        return mPoly
}
```

```
private static FeatureCollection geoJsonToFeatureCollection(Object json) {
    String str = json
    InputStream is = new ByteArrayInputStream(str.bytes)
    FeatureJSON fJSON = new FeatureJSON()
    FeatureCollection featureCollection = fJSON.readFeatureCollection(is)
    return featureCollection
}
```

Lisa 8 – QGIS pistikprogrammi kood regioonide ja ruumiandmete laadimiseks tagarakendusest

```
import os
import requests

from qgis.PyQt import uic
from qgis.PyQt import QtWidgets
from qgis.PyQt.QtWidgets import QApplication
from qgis.core import *
from .project_layer import ProjectLayer as ProjectLayer
import layers.config as Config
import layers.message_util as message_util

# This loads your .ui file so that PyQt can populate your plugin with the
elements from Qt Designer
# FORM_CLASS, _ = uic.loadUiType(os.path.join(
#     os.path.dirname(__file__), 'layers_dialog_base.ui'))
FORM_CLASS, _ = uic.loadUiType(os.path.join(
       os.path.dirname(__file__), 'layers.ui'))


class LoadLayers(QtWidgets.QDialog, FORM_CLASS):
       def __init__(self, parent=None):
              """Constructor."""
              super(LoadLayers, self).__init__(parent)
              # Set up the user interface from Designer through FORM_CLASS.
              # After self.setupUi() you can access any designer
              # object by doing
              # self.<objectname>, and you can use autoconnect slots - see
              # http://qt-project.org/doc/qt-4.8/designer-using-a-ui-file.html
              # #widgets-and-dialogs-with-auto-connect

              # List of layers that the user can choose and edit
              self.layers = []
              # Raw json from getting available layers
              # Includes type_code which allows
              # to query db for specific layers
              self.json = None
              self.setupUi(self)
              self.load.clicked.connect(self.handleLoadBtnClick)

       def loadLayers(self):
              # Get layers (names and typeCodes)
```

```
            self.getRegionConfiguration()
            # Add layers to UI list
            self.layerListUI.clear()
            if self.json:
                  for layer in self.json:
                        item = QtWidgets.QListWidgetItem()
                        item.setText(layer['name'])
                        item.setData(1, layer['typeCode'])
                        self.layerListUI.addItem(item)
                  self.show()

      def handleLoadBtnClick(self):
            for selectedLayer in self.layerListUI.selectedItems():
                  layerName = selectedLayer.text()
                  typeCode = selectedLayer.data(1)
                  layerExists = False

                  # Check if layer already in project
                  for layer in QgsProject
                              .instance().layerTreeRoot().children():
                        if layer.name() == layerName:
                              layerExists = True
                              confirmationPopup =
                                    message_util
                                          .create_confirmation_popup(
'Hoiatus', 'Kiht: "' + layerName + '" on juba projektis. Kas soovite kihti
üle kirjutada?','Jah', 'Ei')
                              confirmationPopup.exec()
                              # If Yes was pressed, remove layer and load
                              if confirmationPopup
                                    .clickedButton().text() == 'Jah':
                                    QgsProject.instance()
                                          .removeMapLayer(
                                                layer.layerId()
                                                )
                                    self.downloadAndInitLayer(
                                          typeCode, layerName
                                          )
                  # Layer must not already be in project for us to load it.
                  # Prevents loading layer the second time
                  # when it was reloaded in for loop
                  if not layerExists:
                        self.downloadAndInitLayer(typeCode, layerName)

      def downloadAndInitLayer(self, typeCode, layerName):
            # Show loading popup to user before starting to load features
            loadingPopup = message_util.create_message_popup(
                  'Laadimine...', 'Toimub kihi: "' + layerName + '"
laadimine, palun oodake')
            loadingPopup.show()
            QApplication.processEvents()
```

```
        # get region geojson and build a vectorlayer
        geojson = self.getRegionGeoJSON(typeCode)
        if geojson:
                vlayer = QgsVectorLayer(geojson, layerName, "ogr")
                # Build projectLayer object from vector layer
                # and add it to QGIS so user can edit it
                self.initLayers(typeCode, vlayer, layerName)
        loadingPopup.close()

# Create a new in memory layer and add it as a project in QGIS
def initLayers(self, typeCode, vlayer, layerName):
        self.projectLayer = ProjectLayer(typeCode, vlayer, layerName)
        self.projectLayer.loadLayerAsProject()

def reloadFeatures(self, projectLayer, typeCode, layerName):
        geojson = self.getRegionGeoJSON(typeCode)
        vlayer = QgsVectorLayer(geojson, layerName, "ogr")
        projectLayer.reloadFeatures(vlayer)

# fetch the list of layers/regions that the
# user is allowed to edit/view
def getRegionConfiguration(self):
        self.layers = []
        response = requests.get(
                url=Config
                        .ROOT_URL
                        + '/rest/v1/gis/regionConfiguration/list',
                headers={'accept': 'application/json'},
                verify=False,
                cookies={'tokenID': Config.TOKEN_ID}
        )
        if response.status_code == requests.codes.ok:
                self.json = response.json()
                for layer in self.json:
                        self.layers.append(layer['name'])
        else:
                responseJson = response.json()
                errorMsg =
                        'Kihtide nimekirja päring ebaõnnestus: ' \
                        + str(responseJson.get('status')) + ' - ' +
                        responseJson.get('message')
                errorPopup = message_util.create_error_popup(
                                        'Viga', errorMsg)
                errorPopup.exec()

def getRegionGeoJSON(self, typeCode):
        url = Config.ROOT_URL + '/rest/v1/gis/' + typeCode
        response = requests.get(
                url=url,
                headers={'accept': 'application/json'},
                verify=False,
```

```
        cookies={'tokenID': Config.TOKEN_ID}
)
if response.status_code == requests.codes.ok:
        return response.content.decode('utf-8')
else:
        responseJson = response.json()
        errorMsg =
        'Regiooni (' + typeCode + ') pärimine ebaõnnestus: ' \
        + str(responseJson.get('status')) + ' - ' +
        responseJson.get('message')
        errorPopup = message_util.create_error_popup(
                              'Viga', errorMsg)
        errorPopup.exec()
        return None
```

Lisa 9 – QGIS pistikprogrammi kood regiooni ruumiandmete hoiustamiseks ja funktsioonid abistamiseks

```
from qgis.core import *


class ProjectLayer:
	# Instance list so when saving layers,
	# we can differentiate between different layers
	# And query their properties by calling out a specific instance
	instances = []

	def __init__(self, typeCode=None, vlayer=None, layerName=None):
		"""Constructor."""
		super(ProjectLayer, self).__init__()
		# add newly constructed class to instance list
		self.__class__.instances.append(self)
		# Unique identifier for this instance and layer in Qgis
		self.layerId = ''
		# Layer type, need to tell DB which type of layer is being saved
		self.typeCode = typeCode
		# Helpers for building layer in QGIS
		self.vlayer = vlayer
		self.layerName = layerName
		# in memory layer which will be used to display features in QGIS
		self.memLayer = None
		# in memory layer where only modified and new features
		# are added so saving to database is less consuming
		self.modifiedFeaturesLayer = None
		# helpers for building modifiedFeaturesLayer
		self.modifiedFeaturesSet = set()
		self.modifiedFeatures = []
		# Set of removed feature IDs,
		# sent to database for removing features
		self.removedFeaturesSet = set()
		# misc
		self.modified = False

	def resetFeatureSets(self):
		# self.memLayer = None
		# self.modifiedFeaturesLayer = None
		self.modifiedFeaturesSet = set()
		self.modifiedFeatures = []
		self.removedFeaturesSet = set()
```

```
        self.truncateModifiedFeaturesLayer()

    # For reloading features after sending them to backend.
    # This makes it so we don't have to remove layer
    # from project and add it, which changes styles,
    # which might be confusing for the user
    def reloadFeatures(self, vlayer):
        listOfIds = [feat.id() for feat in self.memLayer.getFeatures()]
        self.memLayer.startEditing()
        self.memLayer.deleteFeatures(listOfIds)
        features = [feat for feat in vlayer.getFeatures()]
        self.memLayer.addFeatures(features)
        self.memLayer.commitChanges()
        self.resetFeatureSets()
        self.modified = False

    def truncateModifiedFeaturesLayer(self):
        listOfIds = [feat.id() for
                feat in self.modifiedFeaturesLayer.getFeatures()]
        self.modifiedFeaturesLayer.startEditing()
        self.modifiedFeaturesLayer.deleteFeatures(listOfIds)
        self.modifiedFeaturesLayer.commitChanges()

    def loadLayerAsProject(self):
        self.memLayer = QgsVectorLayer(
                "multipolygon?crs=epsg:4326", self.layerName, "memory")
        tempFeatures = [feat for feat in self.vlayer.getFeatures()]
        # initialize dataprovider, so we can query
        # which fields are present
        memLayerData = self.memLayer.dataProvider()
        # get feature fields
        attr = self.vlayer.dataProvider().fields().toList()
        # add fields to data
        memLayerData.addAttributes(attr)
        self.memLayer.updateFields()  # update memLayer with fields
        memLayerData.addFeatures(tempFeatures)  # add features to data
        # Custom type to differentiate between custom
        # and user personal layers
        self.memLayer.setCustomProperty('type', 'custom')
        # Add this memory layer to project for editing
        QgsProject.instance().addMapLayer(self.memLayer)
        # Add layer identification so we can
        # differentiate between instances later
        self.layerId = self.memLayer.id()
        # Initialize an in memory layer where modified
        # and new features will be added
        self.loadModifiedFeaturesLayer(self.layerName, attr)
        # Hide fields we don't want user to see or modify
        self.hideFields()
        self.layerColor = self.memLayer.renderer().symbol().color()
```

```
                # Declare listeners inside this method,
                # otherwise they will be garbage collected and not work.
                def modifiedFeaturesListener(layerId, geom):
                        self.modified = True
                        layer = QgsProject.instance().mapLayer(layerId)
                        for fid in geom:
                                self.modifiedFeaturesSet.add(fid)
                                self.modifiedFeatures =
                                        layer.getFeatures(
                                                        list(self.modifiedFeaturesSet)
                                                        )

                def addedFeaturesListener(layerId, features):
                        self.modified = True
                        layer = QgsProject.instance().mapLayer(layerId)
                        for feature in features:
                                self.modifiedFeaturesSet.add(feature.id())
                                self.modifiedFeatures =
                                        layer.getFeatures(
                                                        list(self.modifiedFeaturesSet)
                                                        )

                def removedFeaturesListener():
                        if self.memLayer.editBuffer():
                                removedFids =
                                        self.memLayer.editBuffer()
                                                                .deletedFeatureIds()
                                for feature in
                                        self.memLayer.dataProvider()
                                                .getFeatures(QgsFeatureRequest()
                                                        .setFilterFids(removedFids)):
                                        self.modified = True
                                        # If feature was added in QGIS and removed
                                        # again without saving to database,
                                        # remove from modifiedFeatureSet, so it
                                        # doesnt get saved to database
                                        # (it has no ID)
                                        try:
                                                # Sometimes we get type QVariant,
                                                # sometimes type str.
                                                # They have different ways of
                                                # checking Null/None.
                                                # Elif checks for features added and
                                                # removed without uploading
                                                if(
feature.attribute('id').__class__.__name__ == 'QVariant' and not
feature.attribute('id').isNull() or
feature.attribute('id').__class__.__name__ == 'str' and
feature.attribute('id') is not None) and \
feature.attribute('id') != -1:
                                                        self.removedFeaturesSet.add(
```

```
                                                feature.attribute('id')
                                                      .split('.')[-1])
                                          elif feature.attribute('id') is None
                                          and len(self.modifiedFeaturesSet) > 0:
                                                self.modifiedFeaturesSet
                                                      .remove(feature.id())
                                    except KeyError as e:
                                          print(e)

            def removeFromInstanceList():
                  ProjectLayer.instances.remove(self)

            # Connect listeners
            self.memLayer.committedGeometriesChanges
                  .connect(modifiedFeaturesListener)
            self.memLayer.committedFeaturesAdded
                  .connect(addedFeaturesListener)
            self.memLayer.beforeCommitChanges
                  .connect(removedFeaturesListener)
            self.memLayer.willBeDeleted
                  .connect(removeFromInstanceList)

      # Initialize modified features layer, where modified and new
      # features are added for saving later
      def loadModifiedFeaturesLayer(self, layerName, fields):
            self.modifiedFeaturesLayer =
                  QgsVectorLayer(
                        "multipolygon?crs=epsg:4326",
                        layerName + "_modified",
                        "memory")
            modifiedFeaturesLayerAttrs =
                  self.modifiedFeaturesLayer.dataProvider()
            modifiedFeaturesLayerAttrs.addAttributes(fields)
            self.modifiedFeaturesLayer.updateFields()

      # Used for building a modifiedFeaturesLayer which will be
      # used for uploading to database
      # This layer consist only of new and modified features
      def buildModifiedFeaturesLayer(self):
            self.modifiedFeatures =
                  self.memLayer.getFeatures(list(self.modifiedFeaturesSet))
            self.modifiedFeaturesLayer.startEditing()
            self.modifiedFeaturesLayer.addFeatures(self.modifiedFeatures)
            self.modifiedFeaturesLayer.commitChanges()

      def hideFields(self):
            hiddenWidgetField = QgsEditorWidgetSetup('Hidden', {})
            fields = self.memLayer.fields()
            fieldsToHide = {'id', 'deleted', 'date_created', 'last_updated',
```

```
                    'version', 'creator_session_name',
                    'creator_session_id_code', 'editor_session_name',
                    'editor_session_id_code', 'type_enum',
                    'category_enum', 'origin_id'}
        for field in fieldsToHide:
              if fields.indexFromName(field) != -1:
                    self.memLayer.setEditorWidgetSetup(
                          fields.indexFromName(field),
                          hiddenWidgetField)

def getInstances(self):
      return self.instances

def getModifiedFeatures(self):
      return self.modifiedFeatures

def getRemovedFeaturesSet(self):
      return self.removedFeaturesSet

def getModifiedFeaturesSet(self):
      return self.modifiedFeaturesSet

def getTypeCode(self):
      return self.typeCode

def getLayerName(self):
      return self.layerName

def getMemLayer(self):
      return self.memLayer

def getInstance(self):
      return self

def getModifiedFeaturesLayer(self):
      return self.modifiedFeaturesLayer
```

Lisa 10 – QGIS pistikprogrammi kood ruumiandmete üleslaadimiseks

```
import os, tempfile

from qgis.PyQt import uic
from qgis.PyQt import QtWidgets
from qgis.PyQt.QtWidgets import QApplication
from qgis.core import *
from .project_layer import ProjectLayer as ProjectLayer
from .load_layers import LoadLayers as LoadLayers
import layers.config as Config
import layers.message_util as message_util
import requests

# This loads your .ui file so that PyQt can populate your plugin with the
elements from Qt Designer
# FORM_CLASS, _ = uic.loadUiType(os.path.join(
#     os.path.dirname(__file__), 'layers_dialog_base.ui'))
FORM_CLASS, _ = uic.loadUiType(os.path.join(
       os.path.dirname(__file__), 'upload_layers.ui'))


class UploadLayers(QtWidgets.QDialog, FORM_CLASS):
       def __init__(self, parent=None):
              """Constructor."""
              super(UploadLayers, self).__init__(parent)
              # Set up the user interface from Designer through FORM_CLASS.
              # After self.setupUi() you can access
              # any designer object by doing
              # self.<objectname>, and you can use autoconnect slots - see
              # http://qt-project.org/doc/qt-4.8/designer-using-a-ui-file.html
              # #widgets-and-dialogs-with-auto-connect
              self.setupUi(self)
              # Button functions
              self.uploadBtn.clicked.connect(self.uploadLayer)


       def loadProjectLayers(self):
              # Reset UI
              unsavedLayers = []
              self.listWidget.clear()
              self.importCheckBox.setChecked(False)
              # Fetch the all layers in the project
```

```
        projectLayers = QgsProject.instance().layerTreeRoot().children()
        # Loop through layers with 'custom' type
        for layer in projectLayers:
              if str(layer.layer().customProperty('type')) == 'custom':
                    try:
                          # Find out if layer has been modified
                          ins = list(filter(
                                lambda instance: instance.layerId ==
                                layer.layerId(),
                                ProjectLayer.instances))[0]
                    except IndexError:
                          ins = False
                    # If layer isn't saved, add it to list
                    # so we can warn the user later
                    if layer.layer().editBuffer() and
                          layer.layer().editBuffer().isModified():
                          unsavedLayers.append(layer.name())
                    # Only add layer to UI list if it has been modified
                    elif ins and ins.modified:
                          item = QtWidgets.QListWidgetItem()
                          item.setText(layer.name())
                          item.setData(1, layer.layerId())
                          self.listWidget.addItem(item)

        # Prompt to go back so user can save or continue and
        # show list of layers that user has saved
        if unsavedLayers:
              confirmationPopup =
                    message_util.create_confirmation_popup(
                    'Hoiatus', 'Teil on salvestamata kihte: ' +
                    ','.join(unsavedLayers),
                    'Jätka', 'Mine tagasi')
              confirmationPopup.exec()
              if confirmationPopup.clickedButton().text() == 'Jätka':
                    self.show()
        else:
              self.show()

  def uploadLayer(self):
        modifiedFeaturesLayer = None
        # Get instance of project_layer class by
        # currently selected layerName in listWidget
        # This allows us to query the class for
        # modifiedFeatures and other info for help on
        # saving to database and reloading the layer etc..
        for layer in self.listWidget.selectedItems():
              for ins in ProjectLayer.instances:
                    if layer.data(1) == ins.layerId:
                          ins.buildModifiedFeaturesLayer()
                          modifiedFeaturesLayer =
                                ins.getModifiedFeaturesLayer()
```

```
modifiedFeaturesSet =
       ins.getModifiedFeaturesSet()
removedFeaturesSet =
       ins.getRemovedFeaturesSet()
typeCode = ins.getTypeCode()
layerName = ins.getLayerName()
instance = ins.getInstance()

loading =
       message_util
              .create_loading_progress_popup(
       'Palun oodake!', 'Kihi: "' + layerName
       + '" üleslaadimine...')
loading.show()
QApplication.processEvents()

if len(modifiedFeaturesSet) > 0:
       # Build save request and execute it
       featureJson =
              QgsJsonExporter()
              .exportFeatures(
                     modifiedFeaturesLayer
                     .getFeatures())
       response =
              self.executePostRequest(
                     featureJson,
                     typeCode,
              'save?import='+str(self.importC
              heckBox.isChecked()).lower())

if len(removedFeaturesSet) > 0:
       # Build delete query and execute it
       removedFeatures =
              "{\"removedFids\": [" + ','
              .join(str(s) for s in
              removedFeaturesSet) + "]}"
       response =
              self.executePostRequest(
                     removedFeatures,
                     typeCode,
                     'delete')
# Fetches new features with typeCode
# Builds new vectorlayer
# in QGIS with layerName
# Empties current instance and loads
# new features into that instance
# from the new vectorlayer.
LoadLayers().reloadFeatures(
              instance, typeCode, layerName)
loading.close()
```

```
def executePostRequest(self, postData, typeCode, action):
       header = {'Content-type': 'application/json'}
       response = requests.post(
              url=Config
                     .ROOT_URL + '/rest/v1/gis/' + typeCode + '/' +
              action,
              headers=header,
              verify=False,
              data=postData.encode('utf-8'),
              cookies={'tokenID': Config.TOKEN_ID}
       )
       return response
```