TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Kristjan Marcus Sulaoja 213096IAIB
Andreas Saarep 206827IAIB

GitLabi projektide metaandmete analüüsi ökosüsteemi Cognate'i kasutajamugavuse parandamine ja arendustöö lihtsustamine

Bakalaureusetöö

Juhendaja: Ago Luberg
[PhD]

Tallinn 2024

Autorideklaratsioon

Kinnitame, et oleme 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.

Autorid: Kristjan Marcus Sulaoja, Andreas Saarep

30.05.2024

Annotatsioon

Käesoleva lõputöö raames teostatakse edasiarendus olemasolevale GitLabi projektide metaandmete analüüsi ökosüsteemile Cognate. Rakendust kasutatakse infotehnoloogia teaduskonna tiimiprojektide ainetes, et analüüsida tudengite aktiivsust ja leida üles need tiimid, kes on maha jäämas ning võivad abi vajada. Rakendus võimaldab läbi viia ka täpsemat tiimiliikme põhist hindamist.

Analüüsiks ning hindamiseks vajaminevad andmed kogutakse projektide repositooriumitest kasutades GitLab API-t. Seejärel päritud andmed töödeldakse ja hoiustatakse ning kuvatakse kasutajatele kasutajaliideses läbi mitmete visualiseerimistehnikate. Samuti võimaldatakse samale infole ligipääsu ka tudengitele, kes tiimiprojekti aines osalevad.

Lõputöö esimeses osas kirjeldatakse projektiga seotud äriloogilisi ja tehnoloogilisi probleeme ning arendusmetoodikat. Teine osa käsitleb juurde arendatud äriloogilisi komponente, tehtud valikuid ning probleeme, millega kokku puututi. Seejärel antakse ülevaade projekti tehnoloogilisest lahendusest, varasematest puudujääkidest ning nende kõrvaldamisest. Järgmisena antakse läbilõige projekti keskkondadest ning nende haldamisest. Viimasena analüüsitakse lõputöö tulemusi nii autorite kui ka kliendi silmade läbi ning pannakse paika edasise arenduse sammud.

Lõputöö on kirjutatud eesti keeles keeles ning sisaldab teksti 43 leheküljel, 6 peatükki, 16 joonist, 2 tabelit.

Abstract

Improving User Experience and Simplifying Development for GitLab Project Metadata Analysis Ecosystem Cognate

As part of this thesis, the further development of the existing GitLab projects metadata analysis ecosystem Cognate is carried out. The application is used in the Infotechnology faculty's subjects with team projects to analyze student activity and find teams that are lagging behind and may need help. The application also allows you to perform more detailed individual team member assessments.

The data needed for analysis and evaluation are collected from the teams' project repositories using GitLab API. The inherited data is then processed, stored and displayed to users in the user interface through a number of visualization techniques. Access to the same information is also provided to students who participate in the subject.

The first part of the thesis describes the development methodologies used and the project's problems related to business logic and technological solutions. The second part describes the new business logic components developed, choices made and problems encountered along the way. An overview is then given about technological solutions of the project, previous shortcomings and the ways they were addressed and eliminated. Next a cross-section of project environments and their management is given. Finally, the results of the thesis are analyzed through the eyes of both the authors and the client, and ideas for future development will be established.

The thesis is written in Estonian and is 43 pages long, including 6 chapters, 16 figures and 2 tables.

List of Abbreviations and Terms

API	*Application Programming Interface*, rakendusliides
Django	Tagarakenduse raamistik
DoD	*Definition of Done*, seis agiilses arenduses, kus tehtud töö vastab kõikidele vastuvõtu kriteeriumitele
E2E	*End to End*, testimise metoodika, mis kontrollib süsteemi töökorda algusest lõpuni
HTTP	*Hypertext Transfer Protocol*, protokoll teabe edastamiseks arvutivõrkudes
Live	Keskkond, kus süsteem on lõppkasutajatele igapäevaseks kasutamiseks antud
MR	*Merge Request*, koodiharude mestimistaotlus
MVC	*Model-View-Controller*, arhitektuurimuster, mis jagab tarkvararakenduse kolmeks omavahel seotud osaks: mudel, vaade, kontroller
MVT	*Model-View-Template*, arhitektuurimuster, mis jagab tarkvararakenduse kolmeks omavahel seotud osaks: mudel, vaade, mall
ORM	*Object Relational Mapper*, Objekt-relatsiooniline kaardistamine
Python	Programeerimiskeel
UI	*User interface*, kasutajaliides

Sisukord

List of Figures

List of Tables

1. Sissejuhatus

1.1 Taust

Käesoleva lõputöö raames teostatakse edasiarendus olemasolevale Gitlabi projektide metaandmete analüüsi ökosüsteemile Cognate. Rakendust kasutatakse infotehnoloogia teaduskonna tiimiprojektide ainetes, et analüüsida tudengite aktiivsust ja leida üles need tiimid, kes on maha jäämas ning võivad abi vajada. Rakendus võimaldab läbi viia ka täpsemat tiimiliikme põhist hindamist.

Selle teostamiseks kasutatakse GitLab API-t [1], et koguda projektide repositooriumitest erinevaid andmeid, näiteks *issue*, *milestone*, *commit* [2]. Seejärel päritud andmed töödeldakse ja hoiustatakse ning kuvatakse kasutajaliideses läbi erinevate visualiseerimistehnikate.

1.2 Probleem

Kõige aktuaalsemaks probleemiks Cognate'i juures on vajadus seda õppeainepõhiselt käsitsi konfigureerida, mis on tülikas ja aeganõudev. Seetõttu tuleks vähendada käsitsi konfigureerimise vajadus võimalikult väikeseks. Samuti tehakse topelttööd hinnete sisestamisel, neid lisatakse käsitsi nii Cognate'i kui ka Moodle'isse. Selle parandamiseks tuleb luua liidestus Moodle'iga, mis võimaldaks hindeid Cognate'ist vähese vaevaga Moodle'isse üle viia. Kuna rakenduse kogutavad andmed võiksid olla kasulikud ka tudengite õppeprotsessi parandamiseks, siis tuleb võimaldada ka tudengitele ligipääs Cognate keskkonda. Hetkel selline võimalus puudub. Projekti jätkusuutlikkuse eesmärgil tuleks tagarakenduses läbi viia ka koodibaasi refaktoreerimine. Hetkeseisuga ei ole tagarakenduse koodibaasis võimalik ilma pikaajalise keskendumiseta ja otsimiseta efektiivselt orienteeruda. Leidub mitmesajarealisi faile ja meetodeid, mis tuleks väiksemateks tükkideks teha, sobivatesse kaustadesse liigutada ja intuitiivselt nimetada. Selleks, et tootekeskkonnas tekkinud vigu kergemini üles leida ja siluda, tuleb tagarakenduses implementeerida läbimõtestatud logimine. Antud hetkel on kasutusel ainult Djangosse sisseehitatud HTTP-päringute logimine. Koodibaasi refaktoreerimise tõttu oleks vaja luua testid, et tagada tehtud muudatuste korrektsus.

Järgnevalt saame kirja panna töö oodatavad tulemused:

- käsitsi konfigureerimise vajadust esineb võimalikult vähe

- hindeid on võimalik eksportida Cognate'ist Moodle'isse
- tudengitel on võimalik logida sisse Microsofti kontoga ning jälgida enda hetkeseisu
- läbi on viidud suuremate tükkide refaktoreerimine ning optimeerimine
- implementeeritud on logimine ja erindite töötlemine
- loodud on testid

Nende täiendustega saame oluliselt parandada rakenduse kasutuskogemust nii tudengite kui ka õppejõudude jaoks.

1.3 Arendusmetoodika

Arendajate, juhendaja ja kliendi esindaja vaheline suhtlus toimub Discordi keskkonnas, sest kõik osapooled on keskkonnaga tuttavad ning kasutavad seda igapäevaselt. Projektihaldus, ajajälgimine ja dokumenteerimine toimub GitLabis.

Kasutatakse agiilse arenduse põhimõtteid. Arendus toimub kahenädalastes sprintides, mis algavad planeerimisega, mille käigus pannakse paika sprindi jooksul tehtavad tööd ning jaotatakse need arendajate vahel laiali. Sprintide implementeerimiseks kasutatakse GitLabi *milestone*'e. Siiski tuleb arvestada sellega, et autorid käivad täiskohaga tööl ning tegelevad lisaks antud lõputööle ka teiste koolitöödega. Seega ei pruugi sprintide mahud olla ühtlased, mis teeb pideva tarnimise keeruliseks.

Iga uus arenduspilet (*issue*) dokumenteeritakse GitLabis, kuhu pannakse kirja kasutajalood (*user story*) [3], täpsem kirjeldus ja äriloogika vastuvõetavuse kriteeriumid. Kui pileti arendus on lõppenud, siis luuakse algsele koodibaasile muudatuste juurde liitmiseks mestimistaotlus ehk *merge request* (MR). Arendajal on lubatud muudatused juurde liita alles siis, kui töö vastab kriteeriumitele ehk *definition of done*'ile (DoD).

1.3.1 DoD

DoD [4] on seis agiilses arenduses, kus tehtud töö vastab kõikidele nii tehnilistele kui ka mittetehnilistele vastuvõtu kriteeriumitele. Antud projekti arenduspileti puhul tähendab seda, et koodimuudatused:

- vastavad arenduspiletis sätestatud äriloogika vastuvõtu kriteeriumitele
- testid ning stiilikontrollid on läbitud vigadeta
- uuele funktsionaalsusele on lisatud mõistlikkuse piires teste
- teise arendaja poolt on tehtud koodi ülevaatus [5] ning sellest tulenenud kommen-

taarid on lahendatud.

2. Rakenduse ülevaade

Cognate'i eesmärk on lihtsustada tiimiprojektide ainetes õppejõudude tööd analüüsides ja visualiseerides projekti kulgu, et vähendada käsitsi repositooriumitest andmete otsimist, mis nõuab suurt ajalist ressurssi.

Kasutaja saab luua õppeaineid esindavaid gruppe sarnaselt GitLab *groups*'ile [6], kus õppeainega seotud tudengite projekte hoiustatakse. GitLabis grupile pääsmiku seadistades ning selle väärtuse Cognate'isse lisades saab pärida GitLab API abil projektide kohta erinevaid andmeid, nagu tähtpunkt (*milestone*), arenduspilet (*issue*), koodiridade arv, kulunud aeg jm.

Päritud andmeid töödeldakse, hoiustatakse ning kuvatakse kasutajale. Andmete põhjal on kasutajal võimalik hinnata tiimiliikmeid üksikisikuliselt, nii manuaalselt kui ka automatiseeritud kujul. Üksikisikuline hindamine on tähtis, sest tihtipeale ei ole tiimides tudengite vahel koormused ühtlaselt jaotatud.

2.1 Kasutaja mugavuse parandamine

Cognate'i kasutajamugavuse parandamine ja intuitiivsemaks muutmine seati üheks tähtsaimaks eesmärgiks. Varasemalt oli rakenduse kasutajatel, kes enamasti on abiõppejõud, tekkinud kasutajaliidese kohta palju küsimusi, millele ei osatud vastata ilma arendajate abita. Aga kuna kõik rakenduse arendajad on siiamaani tegelenud projektiga lõputöö raames, siis on tekkinud sügissemestrisse ajavahemik, kus küsimustele vastuseid saada on keeruline. Samuti on intuitiivne kasutajaliides tähtis järgmiste tiimide sujuvamaks sisseelamiseks.

Oluline samm selles suunas oli selgituste lisamine erinevatele nuppudele, vormiväljadele jt kohtadesse. Lisaks selgituste lisamisele vähendati nuppude arvu, kus võimalik, ning muudeti tegevusi automaatsemaks. Näiteks käsitsi gruppide, projektide, repositooriumite lisamisel pidi kasutaja lehte manuaalselt värskendama, et uuendused oleksid kasutajaliideses kuvatud. Samuti lisati vormiväljadele veateated, mis aitavad kasutajal mõista ja parandada mittesobivat sisendit.

Lisaks täiendati registreerimisvormi. Kasutaja registreerimisel suunatakse kasutaja nüüd otse sisselogimislehele, andes sellega märku, et registreerimine õnnestus.

2.1.1 Andmete pärimine

GitLabist uute andmete pärimine oli segadusttekitav. Kasutajaliideses oli kasutuses kaks pealtnäha sarnast, kuid väga erinevate tööpõhimõtetega nuppu, mille kasutusjuhud ei olnud piisavalt arusaadavalt ära kirjeldatud. Üks nuppudest võimaldas andmete tõmbamist ainult nende liikmete põhjal, kes olid lisatud projektide alla. Teine nupp ei eristanud liikmete kuuluvust grupi ja projektide vahel[1].

Sellest tulenevalt tekkis probleem nuppude vaheldumisi kasutamisega, mistõttu ei uuendatud kasutajate andmeid korrektselt. Kuna autorid ei näinud vajadust võimaldada mõlemat viisi andmete uuendamist ühe grupi raames, siis leiti sobivaks lahenduseks grupi lisamise vormi täiendamine, kus võimaldatakse määrata, kas andmeid tõmmatakse projekti- või grupiliikmete põhiselt.

2.2 Hindamissüsteem

Hindamissüsteem on oluline osa platvormist, mis võimaldab hinnata ja jälgida tudengite panust ning edusamme nende projektides. Seetõttu on tähtis hoida hindamissüsteemi loogika võimalikult lihtsana, kuid seejuures säilitada konfigureerimisvõimalused.

Hindamine toimub tähtpunkti kaupa, mille alla õppejõud saab määrata, milliseid kriteeriume hinnatakse igatähtpunkti all. Hindamiskriteeriumid tähtpunkti all võimaldavad õppejõududel jälgida ja hinnata tudengite edusamme erinevate kategooriate alusel läbi terve õppeaine kestuse. Kategooriateks võivad olla näiteks ajaline panus, projekti dokumentatsiooni kvaliteet, meeskonnatöö oskus jne.

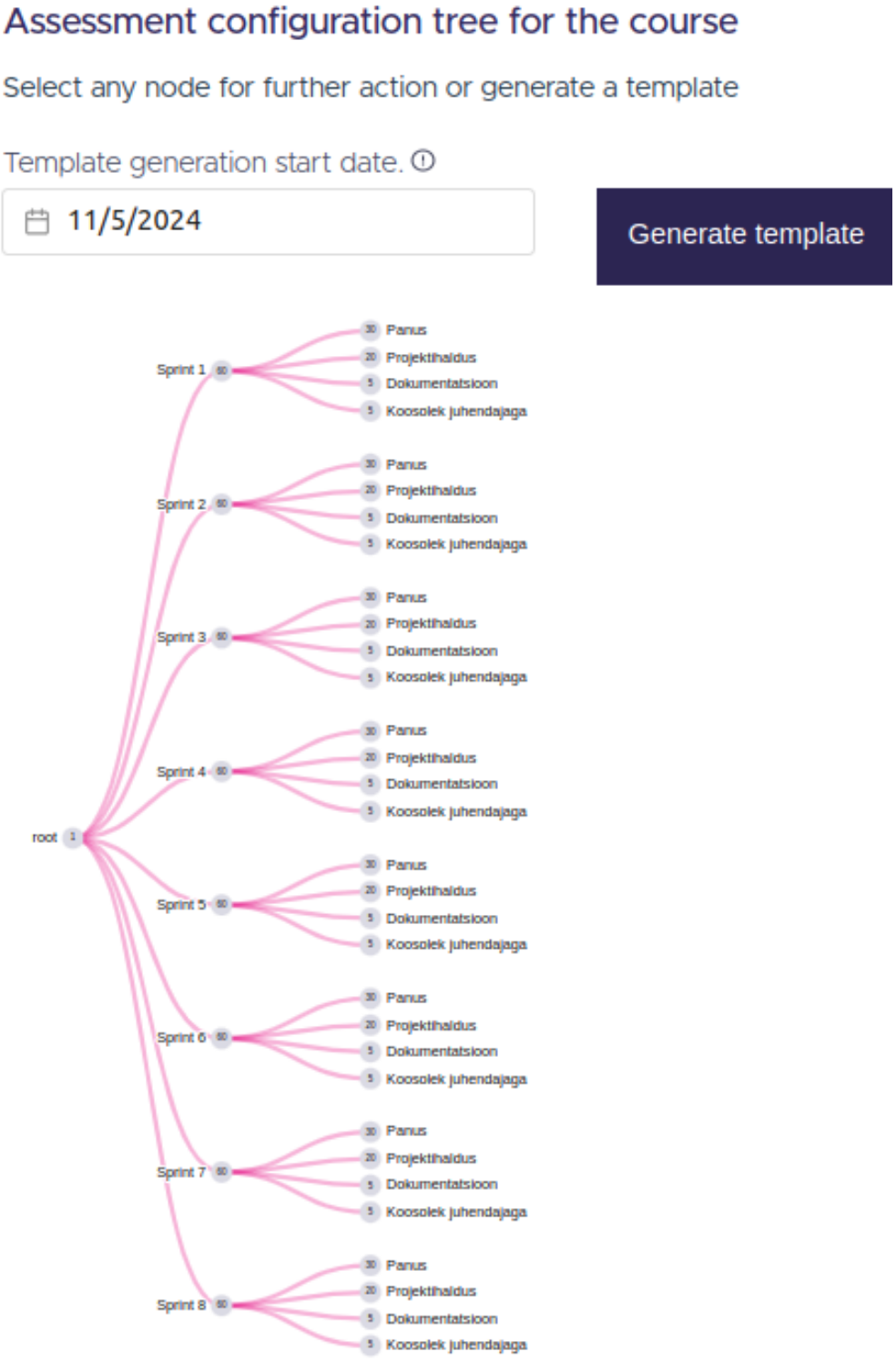
2.2.1 Automaatne hindamine

Kriteeriumi lisamisel saab valida, kas hinne arvutatakse automaatselt süsteemi poolt või lisatakse käsitsi. Automaatse hindamise tüübid on ettemääratud ning sisaldab valikuid nagu koodirida arv, panustatud ajakulu jt. Süsteem arvutab GitLabist saadud andmete põhjal punktid, võttes arvesse kasutaja poolt määratud lävendit. Tudeng saab maksimaalsed punktid, kui lävend on täidetud, vastasel juhul arvutatakse punktid proportsionaalselt puuduoleva osaga.

[1] GitLabis saab liikmeid lisada grupi alla, kuid ka eraldi ainult projekti. Nii saab kasutajatele jagada ligipääsu vastavalt kas kõikidele grupis olevatele projektidele või ainult valitud projektile

2.2.2 Hindamispuu genereerimine

Hindamispuu käsitsi loomine on tülikas ja ajamahukas. Peale selle on hindamispuu struktuur erinevate õppeainete vahel väga sarnane, mistõttu loodi platvormile hindamispuu JSON-mallist genereerimise võimalus. Kasutajal on võimalus valida esimese tähtpunkti algusaeg, mille põhjal genereeritakse kahe nädalased tähtpunktid. Kuigi selle struktuuriga hindamispuu ei sobi alati igale projektile, annab see siiski kasutajale aimu selle kohta, kuidas hindamispuu võiks välja näha ning seda seejärel sobivaks muuta. Mallipõhine genereerimine võimaldab suuremat paindlikkust ja mugavust hindamisprotsessis ning aitab kaasa kasutajamugavusele. Komponendi lahenduses arvestati võimaliku tulevase täiendusega, mis hõlmab JSON-mallide salvestamise funktsionaalsust. Genereeritud hindamispuu on kujutatud joonisel 1.



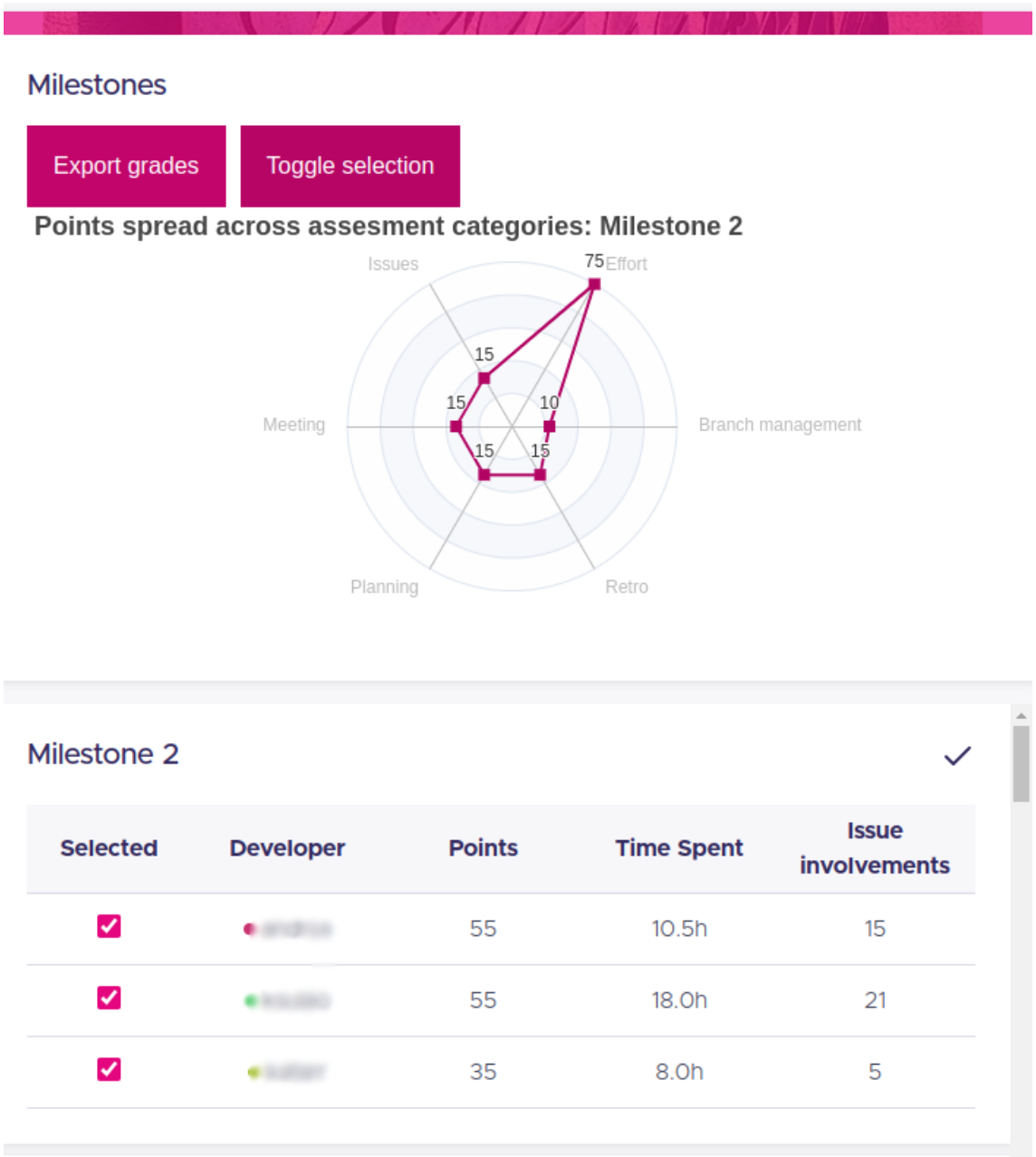
Joonis 1. Genereeritud hindamispuu

2.2.3 Hinnete eksportimine

Siiamaani on olnud vaja rakenduses lisatud hindeid käsitsi Moodle'isse ümber kirjutada. Selle aeganõudva käsitöö kaotamiseks lõid autorid uue funktsionaalsuse hinnete eksportimiseks CSV failiformaadis. Koostatud faili on seejärel võimalik importida Moodle'i kasutajaliideses.

Kuna tihtipeale on projektiainete läbiviimisel abiks mitmed abiõppejõud, kellele igaühele määratakse juhendamiseks mitu projektitiimi, siis on tähtis, et hindeid oleks võimalik eksportida projektipõhiselt. Projekti ülevaate vaatel on võimalik valida, millise tudengi ja millist tema tähtpunkti hinnet tulemustesse lisada. Lisaks arendati võimalus eksportida hindeid tähtpunkti põhiselt, kus tulemuseks on kõikide grupis olevate tudengite hinded valitud tähtpunkti kohta.

Õppeaine ehk grupi kõiki hindeid ei ole võimalik korraga eksportida. Seda selleks, et vältida Moodle'is olevate hinnete ülekirjutamist olukorras, kus Moodle'is tehtud hinnete muudatusi ei ole sünkroniseeritud Cognate'iga. Joonisel 2 on kujutatud hinnete eksportimist projekti ülevaate vaatel ja joonisel 3 CSV-faili eksporditud hinnetega.



Joonis 2. Hinnete eksportimisvaade

Meiliaadress	Milestone 2	Milestone 3
[redacted]	55.00000	40.00000
[redacted]	55.00000	55.00000
[redacted]	35.00000	

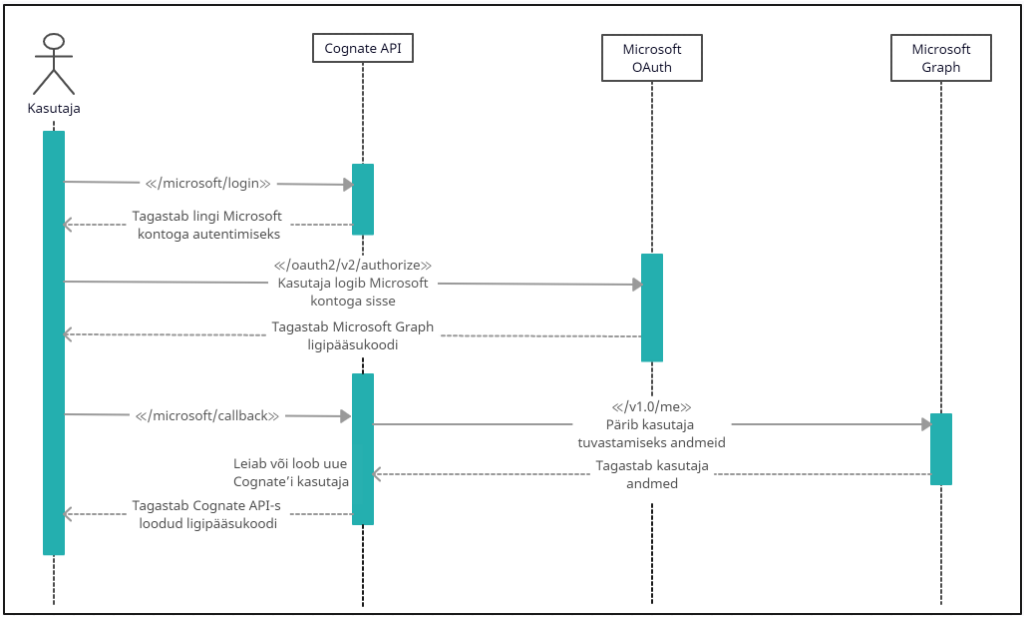
Joonis 3. Eksporditud hinnete CSV fail

2.3 Microsoft OAuth autentimine

Tavapärane kasutatajanime ja parooli põhine konto registreerimine raskendab Cognate'i andmete sidumist TalTechi Microsoft Entra ID [7] platvormi kasutajatega, ehk tudengitega

ja kooli personaliga. Selle probleemi lahendamiseks loodi võimalus autentimiseks kooli Microsoft kontoga läbi Microsoft OAuth2 [8] teenuse. Seeläbi saavad kasutajad sisse logida ning automaatselt registreeruda ilma vorme täitmata. Kasutajaliidesest valitakse sisselogimiseks valik *Login with Microsoft*, misjärel suunatakse kasutaja sisse logima oma Microsoft kontole. Lubatud on vaid TalTechi Microsoft Entra ID *directory* kontod. Eduka sisselogimise korral tagastatakse pääsmik (*token*), mida kasutades päritakse OAuth teenuselt Microsoft Graph [9] teenuse pääsmik. Seejärel küsitakse Graph teenuselt andmed, mille abil on võimalik sisselogivat isikut tuvastada.

Esimest korda sisselogides luuakse kasutajale Cognate'i konto Graphist saadud andmete põhjal, kõikidel järgmistel kordadel leitakse juba andmetele vastav konto rakenduse enda andmebaasist. Seejärel genereeritakse kasutajale autentimiseks JSON Web Token (JWT) ning jätkub tavapärane autentimine. Autentimise voogu on kujutatud joonisel 4.



Joonis 4. Microsoft kontoga autentimise voog

Kusjuures, Microsoft kontoga kasutaja automaatsel loomisel genereeritakse Cognate'i kontole parool, millega ei ole võimalik sisselogimist teostada. See on vajalik, sest Djangosse sisseehitatud *User* andmemudel, mis kasutajate andmeid haldab, nõuab parooli olemasolu. Sealjuures kasutajale hiljem parooli määramist ka ei võimaldata. Selle põhjuseks on plaan täielikult eemalduda käsitsi registreerimisest ning parooliga sisselogimisest. Antud lähenemisega välditakse paroolipõhiseid rünnakuid automaatselt loodud kontodele.

2.3.1 Tulemus

Kasutajaliidese tulemuste valideerimiseks esitasid autorid kliendile küsimused rakenduse täienduste kohta. Lisa 2-s on välja toodud esitatud küsimused, millest osad on hinnatavad 5 punkti skaalal, kus 5 punkti tähistab "väga hea" ja 1 punkt "väga halb".

Klient hindas hinnete genereerimise protsessi arusaadavust ja kasutusmugavust 4 punktiga, märkides, et malli puhul võiks olla kirjas, mis projekti aine järgi see mall tehtud on. Microsofti sisselogimist hindas klient 5 punktiga, olles selle toimivusega väga rahul. Gruppide ja projektide lisamise protsessi arusaadavust ning kiirust võrreldes varasemaga hindas klient 5 punktiga. Tänu selgitustele on kogu protsess arusaadavam ning kiiremini teostatav.

Lisaks märkis klient, et rakendus on üleüldiselt kiirem ja kasutajasõbralikum tänu lisatud selgitustele ning teatud tegevuste automatiseerimisele. Hinnete eksportimise kasutusmugavust ja efektiivsust on hinnatud 5 punktiga ning täiendatud hüpikteateid samuti 5 punktiga.

3. Tehnoloogiline lahendus

Järgmisena kirjeldatakse projekti vältel tehtud tehnoloogiliste valikute analüüsi ja lahendusviise.

3.1 Kasutatud tehnoloogiad

Järgnevalt tuuakse välja autorite arvamus varasemalt valitud tehnoloogiate osas.

Projekti esirakendus on kirjutatud JavaScript raamistikus Vue.js koos TypeScript toega, mis võimaldab komponendipõhist kasutajaliidese loomist. Autorite arvamused raamistikuga arendamise kohta lähevad lahku. Selle põhjuseks võib tuua erinevuse autorite igapäevaselt kasutuses olevates tehnoloogiates.

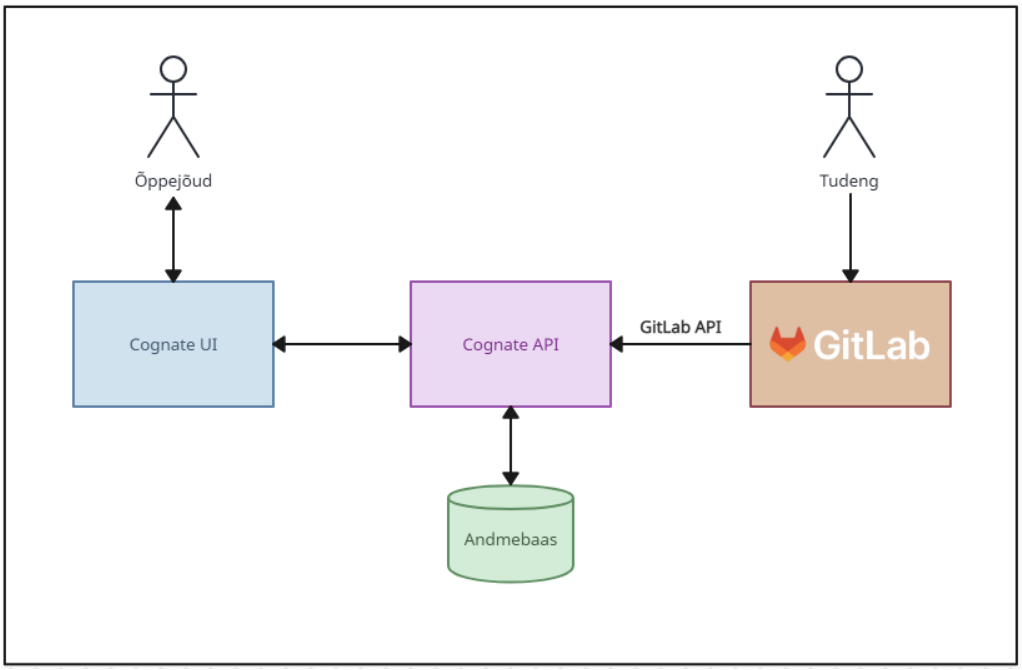
Üks autoritest kasutab igapäevaselt React teeki, millel on autori sõnul palju põhimõttelisi erinevusi Vue raamistikuga, mistõttu on raske ümber harjuda. Teine autor aga kasutab igapäevaselt Angulari ning leiab, et Vue kasutusele võtmine ei olnud konti murdev. Kuigi Angulari ning Vue vahel on süntaktilisi ja tööpõhimõttelisi erinevusi, siis on üldiseid arenduspõhimõtteid võimalik kasutada ilma suuremate muudatusteta. Siiski on mõlemad autorid nõus, et Vue kasutamine Reacti asemel ei ole olnud mõistlik otsus, sest TalTechi kasutajaliidese *commons* teegid on kirjutatud Reactis.

Tagarakenduse jaoks on kasutatud programmeerimiskeelt Python koos veebiraamistikuga Django. Autorid leiavad, et Django ei ole kõige mugavam raamistik REST API kirjutamiseks. Seda seetõttu, et raamistik on autorite arvates suuresti mõjutatud MVT (Model-View-Template) [10] arhitektuurimustrist, kuid see ei takista raamistiku kasutamist REST API-na. Kindlasti on autorite seisukoht mõjutatud igapäevasest programmeerimiskeele Java ning veebiraamistiku Spring kasutusest, mis nõuab küll veidi rohkem koodi, kuid on võrreldes Pythoniga veakindlam tänu staatilisele tüüpimisele. Siiski ei leitud, et oleks piisavalt põhjuseid tagarakenduse ümber kirjutamiseks Springi peale. Seda peamiselt tulenevast ajakulust ja autorite endi ajalisest ressursist.

Andmebaasina kasutatakse PostgreSQL andmebaasi haldamise süsteemi, mis on tuntud oma turvalisuse ja laiapõhjalise funktsionaalsuse poolest. PostgreSQL oli juba varasemalt tuttav tehnoloogia, millega tundsid autorid ennast mugavalt. Andmebaasi skeemi versioneerimiseks, korrektseks ehitamiseks ning muudatuste jälgimiseks on kasutusel Django

Migrations moodul. Palju sarnasusi on võimalik tuua samuti muudatuslogide põhimõttel töötava Liquibase'iga. Suurimaks erinevuseks ning Django Migrations tugevuseks saab tuua muudatuslogide automaatse genereerimise Django Models mooduli abil. Tuleb vaid Models moodulit kasutades defineerida andmemudelid soovitud kujul ning jooksutada käsku *makemigrations*, misjärel võrreldakse mudelites tehtud muudatusi andmebaasi skeemi vana versiooniga.

Joonisel 5 on välja toodud rakenduse arhitektuuri diagramm.



Joonis 5. Rakenduse arhitektuur

3.2 Koodibaasi hügieen

Koodibaasi hügieen viitab koodi puhtusele, organiseeritusele ning arusaadavusele. See hõlmab endas erinevaid praktikaid, mille korrektsel kasutamisel on võimalik tagada parem koodi hallatavus, arendusprotsessi efektiivsus ning kvaliteetsem lõpptulemus. Halb koodibaasi hügieen võib põhjustada järgnevaid probleeme:

- kood on raskesti loetav
- vigade tekkimise tõenäosuse suurenemine
- arenduseks kuluva aja pikenemine
- uuenduste ja muudatuste tegemise keerukuse tõus

Koodihügieeni hoidmiseks tuleks järgida järgnevaid põhimõtteid [11]:

- koodi ülemäärase keerukuse ja sügavuse vältimine, et hoida seda lihtsa ja arusaadavana
- keele ja raamistiku parimate tavade järgimine, et tagada koodi kvaliteet ja vastavus standarditele
- erindite korrektne käsitlemine, et säilitada rakenduse stabiilsus
- testide loomine, et tagada koodi usaldusväärsus
- koodilõikude taaskasutamine, et vähendada dubleerimist ning seeläbi muudatuste lihtsam sisseviimine
- rakenduse sõltuvuste pidev uuendamine, et teekidest turvavigade leidmisel ei tuleks ümber kirjutada suurt osa koodist, kuna rakendust "katki tegevaid" muudatusi on teekides palju

3.2.1 Tagarakenduse refaktoreerimine

Tagarakenduse koodibaasi segadus takistas arendusprotsessi, kuna failide ning funktsioonide liialt pikaks veniv struktuur muutis koodibaasi mõistmise ning edasise arendamise keeruliseks ja aeganõudvaks protsessiks. Refaktoreerimise protsessis toetuti suurel määral Pythoni ja Django kommuuni arvamustele ning väljakujunenud parimatele tavadele.

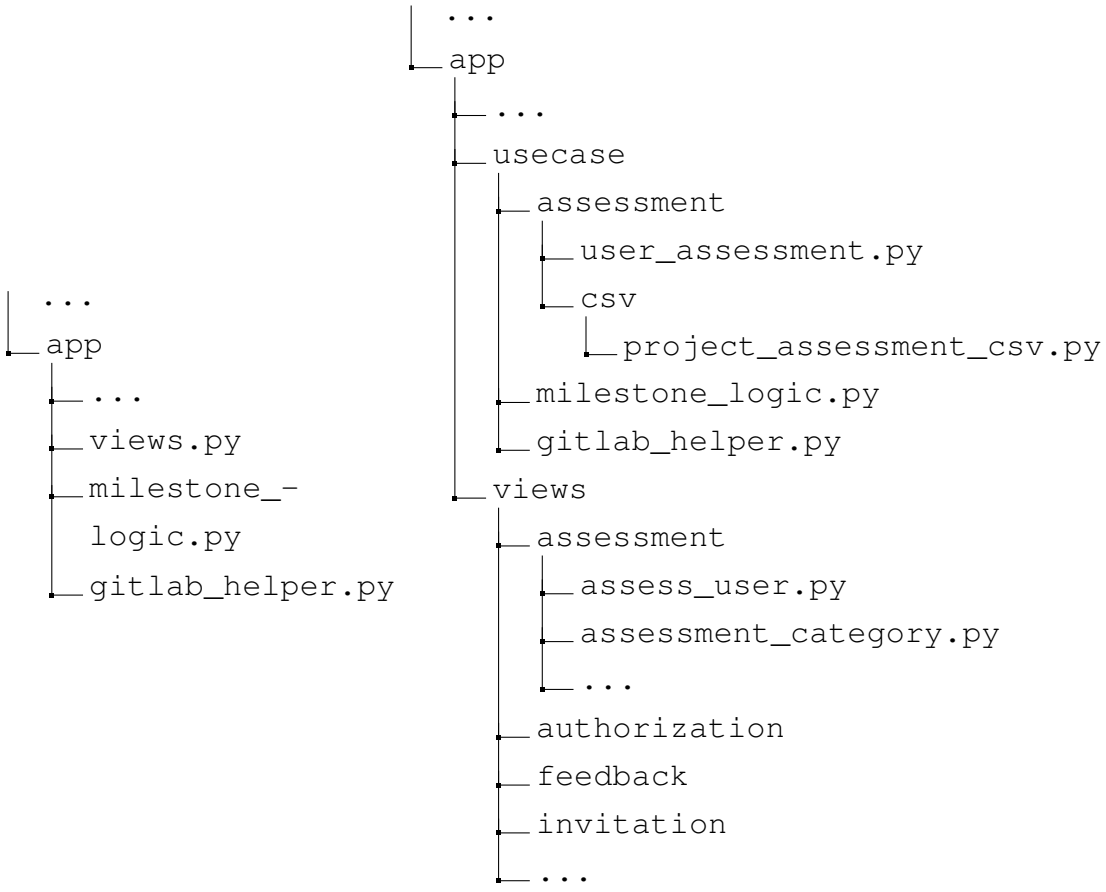
Esimese sammuna võeti ette *views.py*, mis oli üle tuhande rea pikkune ning hoidis endas väga suurt osa rakenduse loogikast. *views.py* faili tükeldamist lihtsustas asjaolu, et kasutusel olid Django REST raamistiku [12] klassipõhised lõpp-punktid. Klassipõhine lõpp-punkt vastab ühele kindlale aadressile ning võimaldab koondada ühte Python klassi kõik vastava aadressi pihta käivad HTTP päringutüübid. Kõik *views.py* failis asuvad lõpp-punktid jaotati eraldi failidesse. Selleks loodi *views* kaust, mille sees omakorda loodi eraldi alamkaustad vastavalt funktsionaalsusele. Seejärel loodi iga funktsionaalsuse ning vaate jaoks eraldi fail ja paigutati sobivasse alamkausta.

Teisena, tükeldati lõpp-punktid MVC [13] põhimõtetel. MVC on tarkvara arhitektuuri muster, kus rakendus jaotatakse kolmeks osaks:

- *Model* - tegeleb andmete manipuleerimisega ja äriloogika teostamisega
- *View* - kuvab andmeid ja haldab kasutaja sisendit. Seda ülesannet täidab esirakendus
- *Controller* - vahendab andmeid *Model* ja *View* kihtide vahel, autoriseerib päringud

Selleks, et kood mustrile vastaks, tuli eraldada äriloogika ja andmete manipuleerimine *APIView* klassidest, mis täidavad kontrolleri ülesannet. Loodi kaust *usecase* ning selle sisse andmemudeleid ja funktsionaalsust arvesse võttes alamkaustad. Samadel põhimõtetel loodi alamkaustadesse äriloogikat ja andmete manipuleerimist sisaldavad failid. Autorid ei

näinud vajadust luua eraldi *repository* kihti, et eraldada äriloogikat andmete manipuleerimisest, sest Django andmemudelite tõttu on mõlemad omavahel tihedalt seotud ning Django eesmärk on olla võimalikult lihtne. Joonisel 6 on kujutatud kaustastruktuuri enne ja pärast.



Joonis 6. Kaustastruktuur enne (vasakul) ja pärast (paremal)

Viimase sammuna teostati äriloogiliste meetodite refaktoreerimine, mis viidi läbi nii, et funktsioonid oleksid:

- hallatava pikkusega, seejuures ei minda üle piiri funktsioonide võimalikult lühikeseks tegemisega
- konkreetsete ülesannetega
- taaskasutatavad
- tähistatud pääsupiirikutega (*access modifier*)
- nimetatud intuitiivselt
- kergelt testitavad

3.2.2 Erindite töötlemine

Erindite töötlemine on oluline osa arendusest, sest see võimaldab vigade tekkimisel jätkata rakenduse sujuvat tööd, anda lõppkasutajale arusaadavaid ja informatiivseid veateateid ning aidata arendajaid vigade silumises. [14]

Vigade töötlemine nii rakenduse serveri- kui ka kliendipoolses osas oli ebapiisav. Enamikel juhtudel tagastati serveripoolse vea tekkel kasutajale vastuseks, et päring oli edukas, mis tegi vigade silumise keeruliseks ning andis kasutajatele eksliku ettekujutuse tegevuste õnnestumisest. Gitlab API-st andmete pärimise õnnestumist hinnati vastuse formaadi järgi. Järjend (*List*) tüüpi vastused loeti õnnestunuks, samas kui sõnastik (*dict*) tüüpi vastused loeti ebaõnnestunuks. Selle asemel tuleks päringuid kontrollida staatusekoodi alusel, lugedes vahemikus 200-299 olevad koodid õnnestunuks ja vahemikus 400-599 ebaõnnestunuks [15].

Django võimaldas kasutusele võtta kohandatud erindite käitleja [16], mis töötleb rakenduses visatud vigu ja tagastab vastavalt veatüübile korrektse veakoodiga vastuse. Ärireeglite vigade tarbeks loodi `BusinessLogicError` erind. Erind võtab kohustusliku sisendina `ErrorCode` *enum* väärtuse, mis tähistab vastavat ärireegli viga ning mida saab kasutajaliideses kasutada võtmena. Valikulise parameetrina saab lisada ka kasutajaliidese välja nime, kus veateade kuvada tuleks.

Kasutajaliidese käitumine oleneb sellest, kas välja nimi on tagarakenduses lisatud või mitte. Olemasolul kuvatakse välja juures veateade, selle puudumisel aga kuvatakse *toast*'i teavitus. See lähenemine tagab, et kasutajaliideses kuvatakse vead selgelt ja arusaadavalt, aidates kasutajal probleemidega toime tulla. Lisaks koondati kasutajaliideses vigade haldamine ühte meetodisse, vastupidiselt varasemale olukorrale, kus vigade käitlemine toimus igas komponendis omamoodi.

Kõik andmeid manipuleerivad lõpp-punktid viidi transaktsiooni sisse. See tähendab, et kui lõpp-punkti tegevuses esineb viga, mida ei püüta kinni, siis kõik transaktsiooni sees tehtud andmebaasi muudatused võetakse tagasi. Selline vigade käsitlemine hoiab ära osaliselt uuendatud andmebaasi tekkimist. Transaktsiooni implementeerimiseks kasutati Django *@transaction.atomic* dekoraatorit [17].

Tabelis 1 on välja toodud tagarakenduse poolt visatavad veakoodid, viskamise tingimus ja vastav käitumine kasutajaliideses.

Staatus	Vea viskamise tingimus	Kasutajaliidese käitumine
401	Kasutaja sisselogimise andmed on valed	Kuvab *toast*'i teavituse.
403	Kasutajal puudub õigus tegevuseks	Kuvab *toast*'i teavitus.
404	Lehte või ressurssi ei leitud, lehe olemasolu varjatakse kasutaja eest	Suunab "not found" lehele.
422	Ärireeglitega tekib viga	Visatakse *toast* või kuvatakse vormivälja juures vastav veateade.
500	Kõikidel muudel juhtudel	Kuvab *toast*'i teavitus.

Tabel 1. Rakenduse veakoodid

3.2.3 Logimine

Logimine on rakenduse jooksutamise vältel toimunud sündmuste ning tegevuste ülesmärkimine eesmärgiga jälgida ja analüüsida rakenduse käitumist.

Rakenduse logid

Kuna tagarakenduses puudus läbimõtestatud logimine ning seotud reeglid otsustati kasutusele võtta Djangosse sisseehitatud *logging* teek [18]. Django `settings.py`'s saab määrata alates millise tasemega logisid konsoolis kuvatakse. Djangos on 5 tüüpi tasemeid: *DEBUG*, *INFO*, *WARNING*, *ERROR*, *CRITICAL*. Tasemed on järjestatud madalaimast tasemest DEBUG kuni kõrgeima tasemeni CRITICAL.

Tabelis 2 on näha rakenduses kasutatavad logimisetasemed ja kasutuskohad.

Tase	Kasutuskoht
ERROR	Vigade esinemisel
INFO	Üldine info tähtsamate rakenduses toimuvate sündmuste ning tegevuste kohta
DEBUG	Detailsem info, mida kasutatakse tõrkeotsinguks. Vaikimisi *debug* tasemega logisid ei kuvata, et vältida liigset ressursikasutust

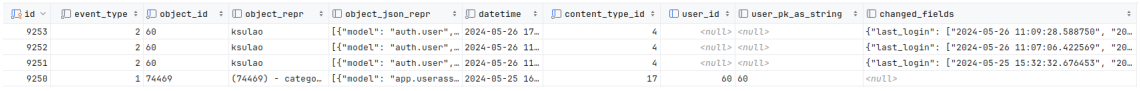
Tabel 2. Tagarakenduse logimistasemed

Baasipäringute auditlogid

Auditi logimine on süsteemi tegevuste dokumenteerimise protsess, mille käigus salvsetatakse sündmus, toimumise aeg ja vastutav kasutaja.

Lisaks rakenduste logidele otsustati kasutusele võtta auditlogid, kasutades selleks *django-*

easy-audit teeki [19], mis võimaldab logida kõik CRUD ja sisselogimise sündmused andmebaasi tabelitesse. Joonistel 7 ja 8 on näited andmebaasis salvestatud sündmustest. Teek kasutab Django signaale [20] sündmuste kuulamiseks, näiteks sisselogimise sündmuse registreerimiseks ootab *django-easy-audit user_logged_in* signaali.



id	event_type	object_id	object_repr	object_json_repr	datetime	content_type_id	user_id	user_pk_as_string	changed_fields
9253	2	60	ksulao	[{"model": "auth.user",…	2024-05-26 17…	4	<null>	<null>	{"last_login": ["2024-05-26 11:09:28.588750", "20…
9252	2	60	ksulao	[{"model": "auth.user",…	2024-05-26 11…	4	<null>	<null>	{"last_login": ["2024-05-26 11:07:06.422569", "20…
9251	2	60	ksulao	[{"model": "auth.user",…	2024-05-26 11…	4	<null>	<null>	{"last_login": ["2024-05-25 15:32:32.676453", "20…
9250	1	74469	(74469) - catego…	[{"model": "app.userass…	2024-05-25 16…	17	60	60	<null>

Joonis 7. CRUD sündmuste auditlogide tabel

id	login_type	username	datetime	user_id	remote_ip
54	2	test	2024-05-26 17:44:38.656645 +00:00	<null>	193.40.252.91
53	2	admin	2024-05-26 17:44:36.889850 +00:00	<null>	193.40.252.91

Joonis 8. Sisselogimise sündmuste tabel

3.3 Päringute optimeerimine

Suur osa kasutajaliidese mugavusest on seotud sellega, kui kiiresti rakendus suudab reageerida kasutaja tegevustele [21]. Rakenduse reageerimisvõimekust mõjutavad erinevad tegurid, nagu näiteks interneti kiirus, riistvara võimekus, ajakohane tarkvara jt. Kuna üldiselt tarkvara arendajatel selliste väliste tegurite üle kontrolli ei ole, siis nähakse palju vaeva sellega, et rakendus ise oleks võimeline reageerima võimalikult kiiresti.

Jõudluse aspekti tuleb arvesse võtta alates projekti algusest saadik. Tähtsad on tehnoloogilised valikud nagu kasutatavad programmeerimiskeeled ja raamistikud ning ka projekti ja rakenduste arhitektuur. Üldjuhul saavad siiski otsustavaks faktoriks kas taga- või esirakenduse kood ning baasipäringud. Cognate puhul oli reageerimiskiiruse pudelikaelaks tagarakenduse kood ning Djangosse sisseehitatud objekt-relatsioonvastanduse ehk ORM-i (*object-relational mapping*) valekasutus.

3.3.1 ORM

ORM on programmeerimistehnika, mille abil saab siduda objekt-orienteeritud programmeerimise koodi andmebaasi relatsioonilise andmemudeliga. ORM vastandab objektid ning nende atribuudid tabelitele ning nende väljadele [22].

Lisaks andmete vastandamisele võimaldab ORM andmebaasist andmete lugemist ja manipuleerimist ilma puhast SQL-i kirjutamata. Tänu sellele on võimalik vähendada objektide ning tabelite sidumiseks ja andmetega manipuleerimiseks vajaminevat koodi ning kasutada SQL-i süntaksi õppimisele ja kirjutamisele kuluvat aega mõne muu otstarbe jaoks.

Ohukohad

Siiski kaasnevad ORM-iga ka teatud negatiivsed küljed. Esiteks, arendaja ei ole sunnitud mõtlema selle üle, kuidas oleks mõistlik tabelit koostada [1] või päringut kirjutada SQL-is. Seetõttu on keerukus andmebaasi struktuuris või vead baasipäringute koostamises kerged tekkima. Teiseks, ilma liidest süvitsi tundmata on keeruline olla veendunud selles, kas ORM-i poolt loodud päring teeb täpselt seda, mida arendaja on ette näinud ning kas päring on optimaalne või mitte. Tihtipeale juhtub ka, et arendaja ei ole teadlik või on lihtsalt ära unustanud, et ilma andmeid ORM-i mudelist ümber kirjutamata ei ole tegemist täielikult rakenduse mälus asuvate objektidega, vaid ka otseste viidetega seotud andmebaasitabelite kirjetele.

N+1 probleem

Viimasena kirjeldatud olukord võib suuresti pärssida rakenduse jõudlust. Nimelt võib probleem tekkida, kui otsitakse andmeid mingi olemiga (*entity*) seotud olemite kohta. Probleemi illustreerimiseks kujutame ette internetipoe rakendust, kus on vaja leida kasutaja kõik ostukorvid ning nende ostukorvide sisu.

Ühe baasipäringuga otsitakse üles kõik sellised kasutaja ostukorvi andmed, mis asuvad ostukorvi esindavas tabelis. Kuid selle asemel, et kohe ka ostukorvi sisu seotud tabelitest välja pärida, teostatakse otsing for-tsükli sees iga ostukorvi jaoks eraldi. Situatsiooni on kujutatud Django koodilõigu näitel joonisel 9.

```
orders = Order.objects.filter(buyer_id=buyer_id)
for order in orders:
    items = OrderItem.objects.filter(order_id=order.id)
```

Joonis 9. ORM-i valekasutus

Sellist viga nimetatakse N+1 probleemiks [23], kus 1 tuleneb kõikide ostukorvide üksikust päringust ning N ostukorvide hulgast, mille üle itereeritakse. Probleemi vältimiseks tuleks kasutada virka laadimist (*eager loading*) ehk koos ostukorvide päringuga tuleks pärida ka kõik ostukorvidega seotud kaubad. Eelnevalt kujutatud baasipäringu parandatud versioon on näha joonisel 10.

```
orders = (Order.objects.prefetch_related("orderitem_set")
                       .filter(buyer_id=buyer_id))
```

Joonis 10. Parandatud baasipäring

[1] Django ORM loob andmebaasi struktuuri koodi põhjal

3.3.2 Päringute optimeerimine projektide loendvaate näitel

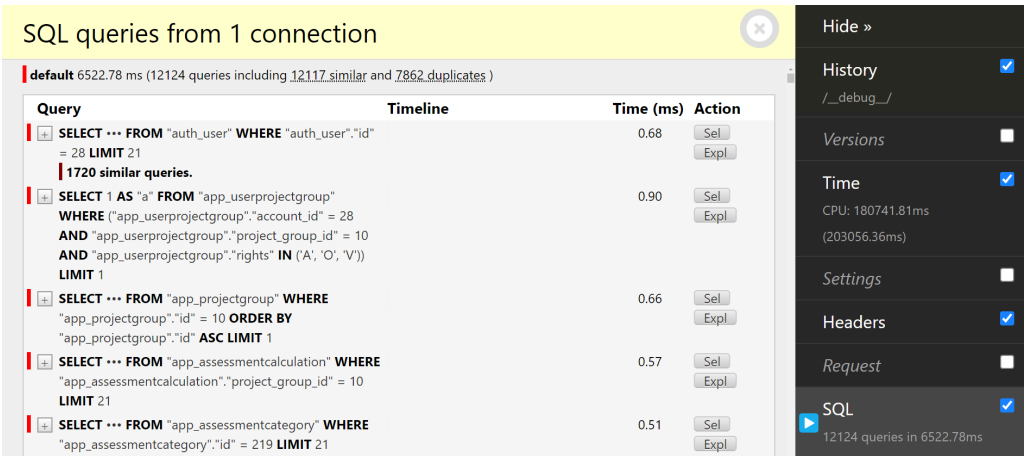
Projektide listvaatel kuvatakse kõiki grupi alla kuuluvaid projekte. Kuna tegemist on vaatega, mis on osa väga paljudest kasutusjuhtude voogudest, siis on tähtis, et see oleks võimalikult kiiresti reageeriv. Varasemalt on üritatud projektide loendvaate päringu aeglust ära peita tulemuste salvestamisega brauseri *local storage* salvestusruumi. Selline lahendus on aga petlik, kuna on võimalik, et kasutajale kuvatakse aegunud informatsiooni.

Algseis

Päringu efektiivsuse mõõtmise aluseks võeti sarnane seis tootekeskkonna andmebaasiga, kus ühe grupi all oli 63 projekti ning hindamissüsteemis oli seadistatud 8 tähtpunkti. Rakenduses toimuva mõistmiseks võeti kasutusele Django Debug Toolbar [24], mille abil on võimalik jälgida HTTP päringu jooksul tehtud funktsioonide väljakutseid, baasipäringuid, *cache*'itud andmeid jms.

Tööriista on võimalik kasutada nii lokaalselt kui ka välises serveris, mille seadistamiseks autorid vajadust ei näinud. Lisaks kaasnevad tööriista kasutamisega väga kõrged lisakulud jõudlusele, mida ei soovitud testkeskkonda tekitada.

Selgus, et 63 projekti puhul tehakse 12124 erinevat baasipäringut, mille kogupikkus jäi keskmiselt vahemikku 6400-6700ms. Sealjuures 12117 neist olid märgitud sarnasteks, millest omakorda 7862 olid duplikaadid. Joonisel 11 on kujutatud tootekeskkonna andmete põhjal tehtud baasipäringud enne optimeerimist.

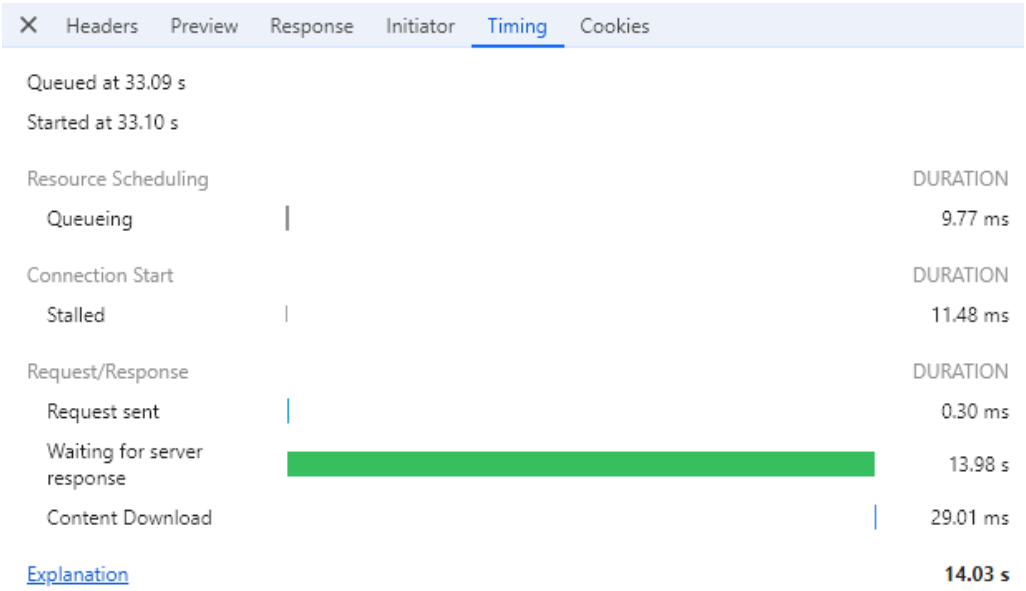


Joonis 11. Baasipäringud enne optimeerimist

Kogu aeg lokaalselt, mis kulus alates päringu vastuvõtust kuni vastuse saatmiseni[2], jäi

[2]*Total CPU time*, andmete saatmine üle võrgu ei ole sisse arvestatud

vahemikku 20-22 sekundit. Tootekeskkonna kogu ajakulu oli vahemikus 12-14 sekundit, mida on kujutatud joonisel 12.



Joonis 12. Ajakulu tootekeskkonnas enne optimeerimist

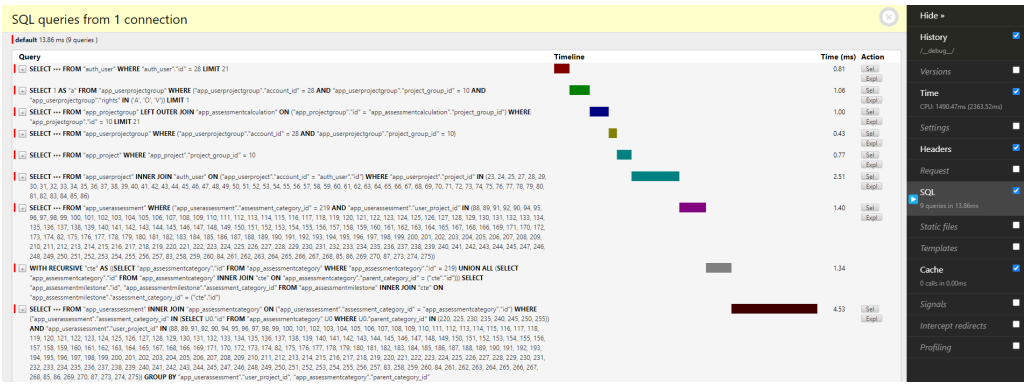
Lahendus

Lõpplahendus kujunes välja mitmete muudatuste tulemusena. Kõige suurema mõjuga neist oli baasipäringute viimine tsüklitest väljapoole. Varasemalt tehti tsüklites kokku mitutuhat duplikaadist baasipäringuid. Tsüklites päringute tegemine tõi esile ka jõudlust suuresti mõjutava N+1 probleemi, mis lahendati baasolemiga seotud andmete kohese baasist välja pärimisega. Lisaks üritati võimalikult palju eemaldada Django Model [25] tüüpi objektide kasutust, kuna nende loomine on väga kulukas protsess. Selle asemel küsiti välja ning kasutati andmeid primitiivsete väärtustena.

Parima efektiivsuse saavutamiseks prooviti läbi mitmeid Django ORM-i kasutusvõimalusi ning liigutati andmetega opereerimist andmebaasitasandile.

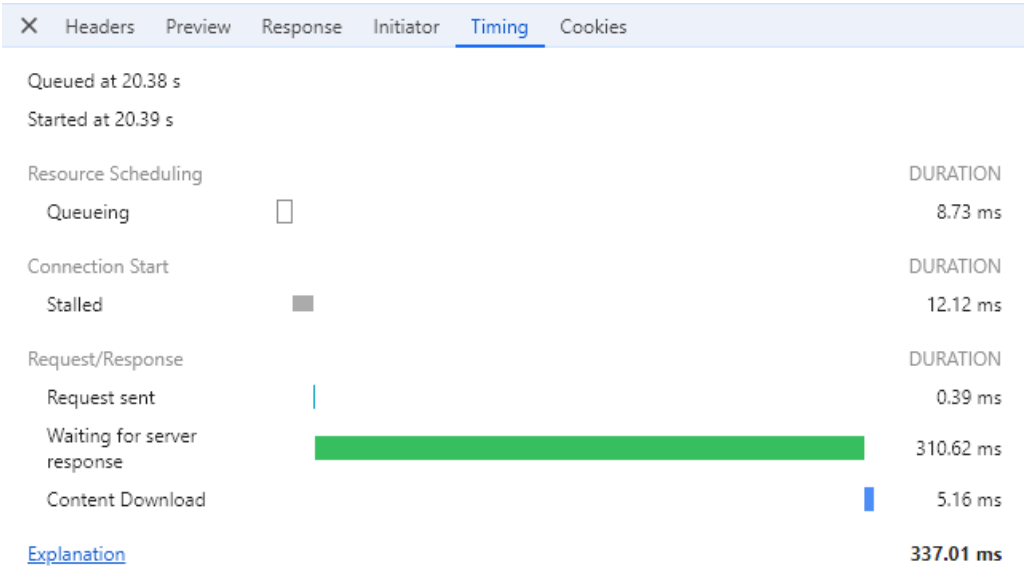
Tulemus

Optimeerimise tulemusena toodi baasipäringute arv alla 9-le päringule, mille tegemiseks kulus 10-14ms. Kusjuures enam ei tehta mitte ühtegi sarnast või duplikaadist baasipäringut. Loendvaate optimeeritud baasipäringud on kujutatud joonisel 13.



Joonis 13. Baasipäringud pärast optimeerimist

Lokaalselt kulub varasema 20-22 sekundi asemel päringule nüüd 140-160ms. Tootekeskkonnas aga veidi üle 300ms, mis tähendab, et päring on ligi 70 korda kiirem. Tootekeskkonna ajakulu on kujutatud joonisel 14.



Joonis 14. Ajakulu tootekeskkonnas pärast optimeerimist

3.4 Testimine

Testimine aitab tagada rakenduse kvaliteeti, avastades vigu ja probleeme enne nende jõudmist tootesse. See on osa koodi hügieeni hoidmisest. Kuna rakenduses puudusid testid ja esines palju vigu, otsustati lisada testid. Testid võimaldavad teha juurdearendusi, viia sisse parandusi ning refaktoreerida olemasolevat koodi, tagades samal ajal koodi ettenähtud käitumist.

Rakenduse testimiseks rakendati Pythoni *unittest*'i teeki [26].

3.4.1 Üksustestid

Üksustestimise [27] käigus testitakse süsteemi väiksemaid osasid, et tagada üksikute komponentide korrektsus. See võimaldab testida rakenduse erinevaid komponente, nagu vaated, mudelid ja vormid. Otsus luua üksusteste tulenes uute funksionaalsuste ja paranduste sisseviimise keerukusest. Manuaalselt testides ei olnud võimalik olla kindel kas komponendid jätkavad töötamist ootuspäraselt.

3.4.2 E2E testid

E2E [28] testide eesmärk on kontrollida süsteemi funktsionaalsust algusest lõpuni, simuleerides kasutusstsenaariume. See aitab kontrollida, kas erinevad komponendid toimivad koos. Kuna tagarakendusse loodi põhjalik erindite käitlemine, viidi läbi suuremahuline refaktoreerimine ja esines situatsioone, mida üksustestimisega ei saanud kontrollida, siis lisati tagarakendusse ka E2E testid.

Mock

Mock on simulatsioon reaalsest objektist. Seda on vaja, kui testitaval objektil on välised sõltuvused ning soovitakse keskenduda testitavale koodile, mitte väliste sõltuvuste käitumisele. [29]. E2E testide jaoks oli tarvis luua GitLab API *mock*, selle jaoks kasutati Pythoni *unittest.mock* teeki [30].

3.4.3 Tulemus

Testide jooksutamiseks kasutati *coverage* teeki, mis võimaldab kuvada testide katvust. *.coveragerc* failis määrati failid ja read, mis pole katvuse jaoks olulised ning mida testimise raportis ei arvestata.

Kokku loodi 138 testi, mis katavad ära 62% koodist. Kindlasti tuleks edaspidi olemasolevaid teste pidevalt täiendada ning uusi lisada. Testide katvust on kujutatud joonisel 15.

app/views/project_group/project_group_update.py	33	3	91%
app/views/project_group/project_group_user.py	42	4	90%
app/views/project_milestone/project_milestone_connections.py	21	2	90%
app/views/project_milestone/project_milestone_data.py	10	0	100%
app/views/project_milestone/project_milestone_time_spent.py	19	1	95%
app/views/repository/add_new_repo.py	35	19	46%
app/views/repository/repository_set_project.py	24	12	50%
app/views/repository/repository_update.py	17	4	76%
app/views/user_project/user_projects.py	15	7	53%
TOTAL	2612	996	62%

Joonis 15. Testide katvus

4. Keskkonnad

Järgnevalt tuuakse esile projektis kasutusel olevad keskkonnad ning seotud põhimõtted.

4.1 Rakenduse avalikustamine

Projektis on kasutuses 2 keskkonda, *test* ning *live*, mille jaoks on seadistatud pideva integreerimise ja tarnimise protsessid. Keskkonnad jooksevad TalTechi Ubuntu serverite peal ning rakendused on konteineriseeritud kasutades Dockerit.

Kuna oli vajadus keskkonna järele, kus saaks eksperimenteerida serveri konfigureerimisega, puhastamisega ja pideva integreerimise ning tarnimise protsessidega, siis otsustati *test* ja *live* keskkonnad eraldi serveritesse tõsta. Lisaks tõstab keskkondade eraldatus tootekeskkonna stabiilsust ja turvalisust.

Projekti ülevõtmisel esines probleem serverite mälupuudusega, mis takistas rakenduse uute versioonide tarnimist ning serverite uuendamist. Selle ajutiseks lahenduseks viidi serverites läbi puhastus, kustutati kõik tarbetud Dockeri ressursid ning serveri failid. Samuti uuendati GitLab Runnerit ning Dockerit, et tagada keskkondade stabiilne töövoog.

4.2 Pideva integreerimise ja tarnimise protsessid

Pideva integreerimise ja tarnimise protsessid [31] on automaatselt konveierpõhimõttel täidetavad käsud, mis võimaldavad kontrollida, et tehtud muudatused ei lõhuks ära juba olemasolevat funktsionaalsust, aitavad hoida koodibaasi hügieeni ning tarnida rakenduse uut versiooni.

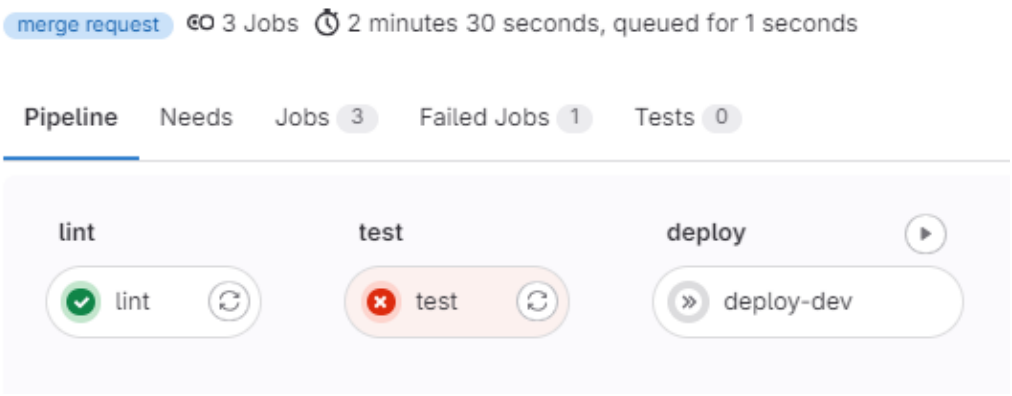
Projektis on selle jaoks kasutusel GitLab CI [32] tööriist, mida on võimalik konfigureerida vastavalt vajadustele. Konveier jaguneb etappideks ja need omakorda töödeks, kus on ära defineeritud käivitatavad käsud. Käske täidab rakendus Gitlab Runner, mille instantsid seati üles nii toote- kui ka arenduskeskkonnas. Tootekeskkonna *runner* seadistati käivituma vaid siis, kui koodimuudatusi tehti *main* haru pihta.

4.2.1 Tagarakenduse konfiguratsioon

Tagarakenduse konfiguratsioon koosneb kolmest etapist:

1. *lint* - koodi staatiline stiilikontroll ning lihtsamate vigade otsimine;
2. *test* - testide jooksutamine;
3. *deploy* - manuaalselt käivitatav rakenduse uue versiooni paigaldamine valitud keskkonda.

Tagarakenduse konveierit on kujutatud joonisel 16.



Joonis 16. Tagarakenduse konveier

Tööriistade valik

Lint etapi teostamiseks otsiti tööriista, mis suudaks teostada nii stiilikontrolli kui ka lihtsamate turva- ja koodivigade otsimist. Põgusa otsingu tulemusena otsustati Prospectori [33] kasuks, mis koondab enda alla mitmeid erinevaid staatilise koodianalüüsi tööriistu, nagu Pylint [34], millel on ka eraldi pistikprogramm Django raamistiku jaoks, pycodestyle jt.

Lokaalne ülesseadmine ning konfigureerimine õnnestus sujuvalt, kasutusele võeti tööriistad Pylint, pycodestyle, Pyflakes, Dodgy, ning Bandit. Lisaks Prospectorile loodi eraldi konfiguratsioonifail Pylintile, ülejäänud tööriistu seadistati läbi Prospectori konfiguratsioonifaili. Pideva integratsiooni konveieris tööriista katsetades ilmnes probleem, kus Prospector ei suutnud leida Pylinti konfiguratsioonifaili. Pikema uurimistegevuse tulemusena tuvastati viga versioonide erinevuses. Runneris installeeriti Pylinti uus *major* väljalase, kuigi Prospectori sõltuvustes see lubatud ei ole. Leiti, et uues versioonis oli ära eemaldatud Prospectoris kasutusel olev konfiguratsioonifaili otsiv meetod. Rakenduses Pylint versiooni määramisega viga kadus.

Kontrolli ranguseks määrati keskmine tase, millega leiti kõikide kasutuses olevate tööriistade pealt kokku ligi 210 viga, mis vajasid parandamist.

4.2.2 Esirakenduse konfiguratsioon

Testide puudumise tõttu koosneb esirakenduse konfiguratsioon kahest etapist:

1. *lint* - koodi staatiline stiilikontroll, mida teostab ESLint;
2. *deploy* - manuaalselt käivitatav rakenduse uue versiooni paigaldamine valitud keskkonda.

5. Analüüs

5.1 Tulemused

Töö tulemuseks on lihtsustatud kasutajaliides, mida on täiendatud hinnete eksportimisvõimalusega Moodle'isse, Microsoft kontoga sisselogimisega ja malli põhjal hindamispuu genereerimisega. Tagarakenduses tehti suuremahuline refaktoreerimine, mis hõlmas koodibaasi struktuuri muutmist, erindite käitlemist, rakenduse protsesside ja tegevuste logimist ja andmebaasi päringute optimeerimist. Tehtud töö valideerimiseks loodi E2E ja üksustestid ning võrreldi päringute kiirust enne ja pärast muudatusi.

Lisaks eraldati üksteisest toote- ja testkeskkonnad, hooldati servereid ning loodi korralik pideva integreerimise ja tarnimise protsess.

Projekti ülevõtmine osutus keerulisemaks kui esialgu arvata võis, kuna koodibaas oli struktureerimata ning esines palju vigu, mille parandamine oli testide ja logide puudumise tõttu keeruline.

Refaktoreerimisele suurema tähelepanu pööramine oli seega äärmiselt oluline samm, mis ei olnud algselt sellises mahus plaaneeritud. See aitas mitte ainult parandada olemasolevaid vigu ja koodibaasi struktuuri, vaid ka tagada, et tulevased arendajad saaksin kiiremini projekti sisse elada. Korralik koodibaas võimaldab järgmistel tiimidel mõista rakenduse toimimist, keskenduda äriloogiliste komponentide arendamisele ning kiiremini leida ja parandada vigu. Samuti autorite tehtud koodihügieeni muudatused on eeskujuks, mida peaksid järgmised arendajad järgima.

5.2 Autorite hinnang tehtud tööle

Arendustöö algusperiood oli keeruline, kuna korralikku projekti üleandmist ei toimunud ning esines palju arendust takistavaid probleeme keskkondade haldamisega. Autorite ajalise ressursi puudumise tõttu ei jõutud õigeaegselt tegeleda tootekeskkonna vigade silumisega, et toetada abiõppejõude ning läbi nende kogemuste leida viise, kuidas Cognate'i kasutuskogemust edasi arendada.

Kogemused projekti sisseelamisel suunasid autoreid tegelema koodibaasi kvaliteedi tõstmisega. Autorid on arvamusel, et tänu tehtud muudatustele on järgmisel tiimil projektiga

lihtsam alustada.

Autorid jäid tehtud tööga rahule. Õpiti kasutama raamistikke Vue.js ja Django, millega varasem kokkupuude oli väga väikene. Lisaks saadi õpetlik kogemus keskkondade haldamise ning seadistamise osas. Suurimaks õppekohaks kujunes autorite arvates koodihügieeni hoidmise tähtsus.

5.3 Kliendi hinnang

Kliendi tagasiside oli positiivne. Eriti rahul oli klient lehe arusaadavuse ja kasutusmugavusega, märkides, et varasemalt oli rakenduse kasutajaliides liiga keeruline ning ei motiveerinud selle kasutamist. Lisaks tõi klient esile rahulolu võimalusega Microsofti kontoga sisse logida ning märkis, et tegevused võtavad nüüd vähem aega, mis muudab rakenduse kasutamise mugavamaks ja kasutajasõbralikumaks võrreldes varasemaga.

Klient tõi välja, et me oleksime võinud varem arendusega alustada, kuna rakendus oli kehvas seisus. Seetõttu ei saadud rakendust mänguprojekti aines kasutada.

5.4 Edasine arendus

Järgmine samm arendustegevuses oleks luua võimalus hindamispuu malle üles laadida. See võimaldaks kasutajatel laadida üles oma kohandatud mallid ning seejärel hindepuu genereerimisel valida nende mallide hulgast, millist struktuuri nad soovivad kasutada.

Lisaks hindamisüsteemi mallide üleslaadimisele oleks kasulik luua GitLabist statistika tõmbamise perioodiline töö. Kuna see protsess on aeganõudev, oleks mugav, kui see protsess toimuks öösiti automaatselt. Selleks võiks luua järjekorra töö, mis käivitub näiteks iga südaöö, ja mis automaatselt tõmbab värskendatud statistika GitLabist. Sellisel juhul saavad kasutajad alati kätte kõige ajakohasema teabe ilma, et peaksid ise protsessi algatama ja ootama selle lõppemist.

Viimasena võiks luua serverite hooldusskriptid, kuna serveri mälu on piiratud ja aegajalt saab see täis. Skript jookseks serveris regulaarsete intervallide tagant ja puhastaks mäluruumi kasutamata konteineritest ning muudest ebavajalikest failidest.

6. Kokkuvõte

Cognate'i probleemiks oli selle tülikas kasutamine nii õppejõudude kui ka arendajate jaoks. Õppejõudude jaoks tähendas see palju käsitsi konfigureerimist ning topelttööd hinnete sisestamisel. Arendajate jaoks oli keeruline rakendust hallata ja arendada segase koodibaasi, puuduvate logide ning testide tõttu.

Töö põhieesmärk oli muuta rakenduse kasutamine lihtsamaks nii õppejõududele, tudengitele kui ka arendajatele. Selle saavutamiseks vähendati käsitsi konfigureerimise vajadust, mis kulutas vähem õppejõudude aega. Lisaks sellele loodi võimalus hinnete eksportimiseks, milletõttu õppejõud ei pidanud enam hindeid eraldi sisestama nii Cognate'isse kui ka Moodle'isse. Samuti loodi tudengitele võimalus logida sisse Microsofti kontoga ning jälgida enda hetkeseisu.

Arendajate jaoks viidi läbi koodibaasi refaktoreerimine, mis muutis selle struktuuri selgemaks ja paremini hallatavaks. Samuti implementeeriti logimine, mis võimaldab jälgida rakenduse toimimist ja tuvastada probleeme kiiremini. Lisaks sellele loodi testid, tagades sellega rakenduse stabiilsuse ja töökindluse.

Järeldusena võib öelda, et tehtud muudatuste abil on Cognate'i kasutamist oluliselt parandatud, muutes selle nii õppejõudude kui ka arendajate jaoks efektiivsemaks ja kasutajasõbralikumaks.

Kasutatud kirjandus

[1] *REST API*. [Accessed: 30-10-2023]. URL: https://docs.gitlab.com/ee/api/rest/index.html.

[2] *Plan and track work*. [Accessed: 30-10-2023]. URL: https://docs.gitlab.com/ee/topics/plan_and_track.html.

[3] Max Rehkopf. *User stories with examples and a template*. [Accessed: 30-10-2023]. URL: https://www.atlassian.com/agile/project-management/user-stories#:~:text=A%20user%20story%20is%20the,the%20end%20user%20or%20customer..

[4] Derek Huether. *Definition of done*. [Accessed: 30-10-2023]. URL: https://www.leadingagile.com/2017/02/definition-of-done/.

[5] Dan Radigan. *Code review*. [Accessed: 30-10-2023]. URL: https://www.atlassian.com/agile/software-development/code-reviews.

[6] *Groups*. [Accessed: 05-05-2024]. URL: https://docs.gitlab.com/ee/user/group/.

[7] *What is Microsoft Entra ID?* [Accessed: 05-05-2024]. URL: https://learn.microsoft.com/en-us/entra/fundamentals/whatis.

[8] *Microsoft OAuth 2.0*. [Accessed: 16-03-2024]. URL: https://learn.microsoft.com/en-us/entra/architecture/auth-oauth2.

[9] *Overview of Microsoft Graph*. [Accessed: 16-03-2024]. URL: https://learn.microsoft.com/en-us/graph/overview.

[10] *Django MVT*. [Accessed: 05-05-2024]. URL: https://www.javatpoint.com/django-mvt.

[11] German Cocca. *How to Write Clean Code – Tips and Best Practices (Full Handbook)*. [Accessed: 05-05-2024]. URL: https://www.freecodecamp.org/news/how-to-write-clean-code/.

[12] *Class-based Views*. [Accessed: 05-05-2024]. URL: https://www.django-rest-framework.org/api-guide/views/.

[13] *Model–view–controller*. [Accessed: 05-05-2024]. URL: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller.

[14] *Error Handling*. [Accessed: 26-05-2024]. URL: https://www.dremio.com/wiki/error-handling/#:~:text=Error%20Handling%20refers%20to%20the,ensure%20data%20integrity%20and%20reliability..

[15] *HTTP response status codes*. [Accessed: 05-05-2024]. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status.

[16] *Custom exception handling*. [Accessed: 30-04-2024]. URL: https://www.django-rest-framework.org/api-guide/exceptions/.

[17] *Database transactions*. [Accessed: 12-05-2024]. URL: https://docs.djangoproject.com/en/5.0/topics/db/transactions/.

[18] *Logging*. [Accessed: 30-04-2024]. URL: https://docs.djangoproject.com/en/4.2/topics/logging/.

[19] *django-easy-audit*. [Accessed: 10-05-2024]. URL: https://github.com/soynatan/django-easy-audit.

[20] *Signals*. [Accessed: 10-05-2024]. URL: https://docs.djangoproject.com/en/5.0/topics/signals/.

[21] *Web performance is user experience*. [Accessed: 10-05-2024]. URL: https://www.forbes.com/sites/oreillymedia/2014/01/16/web-performance-is-user-experience/.

[22] *ORM*. [Accessed: 10-05-2024]. URL: https://www.theserverside.com/definition/object-relational-mapping-ORM.

[23] *N+1 probleem*. [Accessed: 10-05-2024]. URL: https://www.sqlservercentral.com/articles/how-to-avoid-n1-queries-comprehensive-guide-and-python-code-examples.

[24] *Django Debug Toolbar*. [Accessed: 07-05-2024]. URL: https://django-debug-toolbar.readthedocs.io/en/latest/index.html.

[25] *Django Model*. [Accessed: 20-01-2024]. URL: https://docs.djangoproject.com/en/4.2/topics/db/models/.

[26] *Unit testing framework*. [Accessed: 30-04-2024]. URL: https://docs.python.org/3/library/unittest.html#module-unittest.

[27] *Mis on ühiktestimine*. [Accessed: 10-05-2024]. URL: https://kool.gunnarpeipman.com/programmeerimine-2/uhiktestimine/.

[28] *E2E*. [Accessed: 05-05-2024]. URL: https://katalon.com/resources-center/blog/end-to-end-e2e-testing.

[29] *Unit Testing and Mocking Explained*. [Accessed: 11-05-2024]. URL: https://www.telerik.com/products/mocking/unit-testing.aspx.

[30] *unittest.mock — mock object library*. [Accessed: 11-05-2024]. URL: https://docs.python.org/3/library/unittest.mock.html.

[31] *What is CI/CD?* [Accessed: 05-05-2024]. URL: https://www.redhat.com/en/topics/devops/what-is-ci-cd#:~:text=CI%2FCD%2C%20which%20stands%20for,a%20shared%20source%20code%20repository..

[32] *Get started with GitLab CI/CD*. [Accessed: 05-05-2024]. URL: https://docs.gitlab.com/ee/ci/.

[33] *Prospector*. [Accessed: 30-04-2024]. URL: https://prospector.landscape.io/en/master/index.html.

[34] *Pylint*. [Accessed: 30-04-2024]. URL: https://pylint.readthedocs.io/en/v2.17.7/.

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

Meie, Kristjan Marcus Sulaoja ja Andreas Saarep

1. Anname Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose "GitLabi projektide metaandmete analüüsi ökosüsteemi Cognate'i kasutajamugavuse parandamine ja arendustöö lihtsustamine", mille juhendaja on Ago Luberg
   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. Oleme teadlikud, et käesoleva lihtlitsentsi punktis 1 nimetatud õigused jäävad alles ka autoritele.
3. Kinnitame, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest ning muudest õigusaktidest tulenevaid õigusi.

30.05.2024

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

Lisa 2 - Kliendi tagasiside küsimused

1. Kuidas hindad hinnete genereerimise protsessi arusaadavust ja kasutusmugavust? (1-5p)
2. Milline on sinu kogemus hinnete genereerimise kiiruse ja arusaadavusega?
3. Kuidas hindad Microsofti sisselogimise kasutusmugavust võrreldes käsitsi sisselogimisega? (1-5p)
4. Kuidas hindad grupi ja projektide lisamise protsessi arusaadavust ja kiirust võrreldes varasemaga? (1-5p)
5. Milliseid parandusi oled märganud?
6. Kuidas hindad rakendusse lisatud selgituste kasulikkust? (1-5p)
7. Kas need parandused on aidanud protsessi arusaadavamaks muuta?
8. Kuidas hindad hinnete eksportimise funktsionaalsuse kasutusmugavust ja efektiivsust? (1-5p)
9. Kuidas hindad täiendatud hüpikteadete (nii edukate kui ka veateadete) kasulikkust? (1-5p)