TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Robert Krikk 179022IADB

REGRESSIOONITESTIMISE AUTOMATISEERIMINE ETTEVÕTTELE PROEKSPERT AS

Bakalaureusetöö

Juhendaja: German Mumma
Magistrikraad

Tallinn 2020

Autorideklaratsioon

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

Autor: Robert Krikk

11.05.2020

Annotatsioon

Käesoleva töö eesmärk oli luua veebirakenduse automaattestimissüsteem ettevõttele Proekspert AS, mis oleks baasiks järgnevatele süsteemidele. Lahendus loodi ettevõtte kliendile, kellelt võeti üle olemasolev süsteem ning loodi sellele edasiarendus. Süsteemi loomisel kasutati automaattestide arendamiseks Javascripti raamistikku Cypress, mille testid jooksutatakse perioodiliselt läbi pideva integratsiooni tööriista Jenkins.

Töö käigus võrreldi süsteemis kasutusel olevaid raamistikke levinumate testimisraamistikega, mille tulemusena valiti süsteemi edasiseks arendamiseks Cypress raamistik. Testimisraamistiku väljavahetamise kasulikkust hinnati A/B testimise meetodiga, mille kaudu ilmnes, et uue raamistikuga jooksevad testlood ~8% kiiremini, omades samaaegselt suuremat veebilehitsejate tuge.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti kolmekümne viiel leheküljel, seitse peatükki, kaheksateist joonist, viis tabelit.

Abstract

Regression Testing Automation for Proekspert AS

The purpose of this thesis was to create an automated test system for Proekspert AS, which simplifies future test automation projects. The system was created for one of the enterprise's clients as an improvement of existing automated test system. The system's automated tests were developed in Cypress and integrated with Jenkins.

The most popular testing frameworks were compared, and as a result, Cypress framework was chosen. Decision to replace previous testing framework with Cypress was justified, as the speed of test runs improved 8%. Furthermore, the new framework is supported by a larger number of web browsers.

The thesis is in Estonian and contains thirty-five pages of text, seven chapters, eighteen figures, five tables.

Lühendite ja mõistete sõnastik

ATLM	*Automated test life cycle methodology*, automaattestimise elutsükli metoodika
Automaattestimissüsteem	Süsteem, mille käivitamisel testitakse rakenduse funktsionaalsust automatiseeritud testlugudega ning tagastatakse detailne raport
Bitbucket	Git repositooriumide haldussüsteem
Build	Tarkvara ehitamise protsess, mille käigus luuakse lähtekoodi failidest tarkvararakendus
Camel case	Sõne, kus sõnade eraldamiseks kasutatakse uue sõna algul suurtähte
Code review	Koodi ülevaatamine. Kaaskolleegi poolt sooritatav koodi ülevaatamine, veendumaks, et loodud kood on veavaba ja otstarbekas
CRUD	*Create Read Update Delete, r*akenduse kasutajaliideses kasutatav muster olemite loomiseks, lugemiseks, uuendamiseks ja kustutamiseks andmebaasist
CSS	*Cascading Style Sheet*, küljendusel kasutatav märgistuskeel, mida kasutatakse veebilehe kujundamiseks
Cypress dashboard	Cypress raamistiku juhtlaud
Data- atribuut	Veebielementide atribuut, kus talletatakse andmeid
Debug	Koodibloki samm-sammuline läbikäimine ehk silumine
Devops insener	Amet, mille peamine tööülesanne on toodangu- ja arenduskeskkondade ülespanek ning seadistamine pideva integratsiooni tööriistadega
Eesrakendus	Kasutajale nähtav osa veebirakendusest
Environment variable	Arvutis asetsev dünaamiline objekt, mis omab väärtust, mida kasutatakse ühes või mitmes tarkvaraprogrammis
Feature request	Uue funktsionaalsuse soov tarkvararakenduse kasutajate poolt
Git	Versioonihalduse tarkvara
Headless	Rakenduse kasutajaliidese testimine ilma graafilise pooleta – testide jooksutamisel ei kuvata visuaalset samm-sammulist testi läbikäimist
hyphen-case	Sõne, kus sõnade separaatoriteks on kasutatud sidekriipsu

JSON	*JavaScript Object Notation,* Javascripti vorming andmete hoidmiseks objektidena
Kasutajaliidese test	Rakenduse kasutusvoo realiseerimine ja võrdlemine oodatud tulemusega
Kvaliteediinsener	Amet, mille peamine tööülesanne on vastutada projekti kvaliteedi eest läbi testimise
Lowercase	Sõne, mis sisaldab ainult väiketähti
Master branch	Git projektide juurhoidla või peaharu
NPM	*Node Package Manager*, repositooriumi sõltuvuste haldamiseks kasutatav tööriist
Osanik	Kliendi ettevõttes osalust omav isik
Pipeline	Kogum jadamisi ühendatud andmete töötlemise elemente
Regressioonitestimine	Testimisviis, mille käigus veendutakse, et muudatus tarkvaras ei ole tekitanud tõrkeid varasemale funktsionaalsusele
Repositoorium	Failihoidla
Tagarakendus	Kasutajale mittenähtav osa rakendusest, mis sisaldab rakenduse loogikat – andmebaasiga suhtlemine ja andmete käitlemine
Testilugu	Detailne kirjeldus testi sisenditest, lähtetingimustest, protseduurist ja oodatud tulemusest
UI	*User interface*, kasutajaliides
Uurivtestimine	Testimisviis, kus testimise põhjaks pole testilugu, vaid testija isiklik kogemus testitava rakendusega. Samaaegselt on kombineeritud õppimine, testi disainimine ja testi käivitamine
Veebiseanss	Interaktiivne informatsioonivahetus veebirakenduse serveri ja kasutaja(te) vahel
Ühiktest	Tarkvara testimisviis, kus testitakse üht väikest osa rakendusest ehk ühikut

Sisukord

1 Sissejuhatus 11
2 Taust 12
  2.1 Tarkvara testimine 12
  2.2 Automaattestimine 12
  2.3 Automaattestimise protsess 14
    2.3.1 Skoobi määramine 15
    2.3.2 Tööriistade valimine 15
    2.3.3 Testide planeerimine ja implementatsioon 16
    2.3.4 Automaattestide käivitamine 16
    2.3.5 Analüüs ja tulemuste raporteerimine 17
  2.4 Automaattestimissüsteemi hallatavus 17
  2.5 Ettevõtte Proekspert taust testimise automatiseerimisel 18
    2.5.1 Projekti taust 18
    2.5.2 Rakenduse taust 20
    2.5.3 Automaattestimissüsteemi puudujäägid 20
3 Automaattestimissüsteemi kavandamine 21
  3.1 Skoobi määramine 21
  3.2 Tööriistade valimine 21
  3.3 Testide planeerimine ja implementatsioon 24
  3.4 Automaattestide jooksutamine ja raporteerimine 27
4 Automaattestimissüsteem 29
  4.1 Süsteemi seadistamine 29
  4.2 Süsteemi funktsionaalsus 30
    4.2.1 Testandmete loomine ja kasutamine 30

4.2.2 Automaattestid ... 31
4.2.3 Testide jooksutamine ... 33
4.2.4 Testide raporteerimine ... 34
4.3 Automaattestide integreerimine Jenkinsi ... 36
5 Tulemuste analüüs ... 38
5.1 Veebilehitsejate toe võrdlus ... 38
5.2 Testide ülesehituse võrdlus ... 39
5.3 Testide kaetavuse võrdlus ... 40
5.4 Testkomplektide jooksutamise kiirus ... 40
5.5 Arhitektuuri võrdlus ... 41
6 Edasised tegevused ... 44
6.1 Automaattestide paralleelne käivitamine ... 44
6.2 Veebielementide identifikaatorite parandamine ... 45
6.3 Projekti mõju edasistele projektidele ... 45
7 Kokkuvõte ... 46
Kasutatud kirjandus ... 47
Lisa 1 – Github repositooriumi link ... 49

Jooniste loetelu

Joonis 1. Automaattestimise elutsükkel 14
Joonis 2. Kasutajaliidese automaattestide kaetavus 19
Joonis 3. Testitava rakenduse kasutus erinevate veebilehitsejatega 22
Joonis 4. Cypress raamistiku võrdlus teiste tööriistadega 23
Joonis 5. Automaattestide repositooriumi arhitektuur 25
Joonis 6. Automaattesti 3 faasi 27
Joonis 7. Terminalist testide jooksutamise lühendatud käsklus 29
Joonis 8. Testandmete genereerimine Javascriptis faker.js teegiga 31
Joonis 9. Rakenduse testimise alla kuuluvad olemid 32
Joonis 10. Cypress'i testkomplekti sooritamine graafilise kasutajaliidesega 34
Joonis 11. Mochawesome testkomplekti raport 35
Joonis 12. Testkomplektide jooksutamise tulemused Cypress dashboard'is 36
Joonis 13. Cypress testi ülesehituse eelnevas süsteemis 39
Joonis 14. Cypress testi ülesehitus uues süsteemis 39
Joonis 15. Uue ja vana süsteemi kasutajaliidese automaattestide kaetavused 40
Joonis 16. Cypress ja Puppeteer testide sooritusaja võrdlus 41
Joonis 17. Varasema süsteemi arhitektuur 42
Joonis 18. Uue süsteemi arhitektuur 43

Tabelite loetelu

Tabel 1. Testvarad selgitustega 13
Tabel 2. UI testimisraamistike võrdlus 22
Tabel 3. Cypress raamistiku globaalsed seadistused 30
Tabel 4. Automaattestide jooksutamise seadistused 33
Tabel 5. Raporteerimise konfiguratsioon 35

1 Sissejuhatus

Tarkvara testimisel on oluline osa tarkvara kvaliteedi tagamisel [1]. Selle tulemusena väheneb tarkvara tõrgete tekkimise risk. Agiilses arenduses on aga lühikesed iteratsioonid, mis jätab testimiseks vähe aega [2]. Mahukate projektide manuaalne testimine on see eest aeganõudev protsess. Probleemi lahenduseks on regressioonitestimise automatiseerimine, mille käigus väheneb inimressursi testimise aeg ja eksimuste arv. Automaattestimissüsteemide loomisel mängib suurt rolli loodava süsteemi arhitektuur, et saada maksimaalne kasu automatiseerimisest.

Käesoleva töö eesmärk on välja töötada automaattestimissüsteem, mis on arhitektuuriliselt baasiks teistele projektidele ettevõttes Proekspert AS. Ettevõttes Proekspert AS puudub üks kindel arhitektuuriline lähenemisviis testide automatiseerimisel, mille tõttu on vajadus pilootprojekti järgi. Töö autor loob automaattestimissüsteemi ühe projektipõhiselt, kuid dokumentatsiooni põhjal on võimalik kasutada samu tehnikaid ka teistes projektides. Tulemusena võimaldab antud lahendus lihtsustada automatiseerimist teistes projektides, mis omakorda vähendab manuaaltestimise osakaalu ja annab testijale rohkem aega uute funktsionaalsuste testimiseks.

Töö on jaotatud seitsmeks peatükiks. Esimeses peatükis antakse ülevaade tehtavast tööst ja eesmärgist. Teises peatükis kirjeldatakse töö taust – ettevõtte ja projekti taust ning automaattestimise valdkonna tutvustus. Kolmandas peatükis töötatakse välja lahendus vastavalt teise peatüki taustale ning antakse ülevaade tööriistadest ja protsessist. Neljas peatükk annab ülevaate loodud automaattestimissüsteemist. Viiendas peatükis analüüsitakse loodud süsteemi. Kuuendas peatükis tutvustatakse loodud süsteemi edasiseid tegevusi. Viimases peatükis on esitatud kokkuvõte tehtud tööst.

2 Taust

2.1 Tarkvara testimine

Tarkvara testimine on lai mõiste ning hõlmab endas mitmeid erinevaid protsesse. Kõige paremini saab testimise võtta kokku läbi testimise eesmärkide: [1]

- Defektide ärahoidmine läbi töötulemuste hindamise, mille alla kuuluvad tarkvara nõuded, kasutuslood, disain ja kood
- Ettenähtud nõuetele vastavuse kontrollimine
- Testitava objekti täielikkuse kontrollimine lõppkasutaja ja osaniku vaatenurgast
- Kindlustunde tekitamine testitava objekti kvaliteedi suhtes
- Defektide ja tõrgete leidmine, et langetada puuduliku kvaliteedi riski
- Küllaldase info tagamine osanikele, et nad saaksid teha äriliselt õigeid otsuseid tarkvara kvaliteedi põhjal
- Tarkvara kinnipidamine lepingulistele, legaalsetele või regulatoorsetele nõuetele

Eelnevalt mainitud punktide realiseerimiseks kasutatakse nii manuaalset kui automatiseeritud lähenemisviisi. Töö autor keskendub viimasele, et tuua järgmistes peatükkides välja automaattestimise head ja vead.

2.2 Automaattestimine

Automaattestimine on alguse saanud organisatsioonide soovist testida oma tarkvara võimalikult lühikeses ajavahemikus, saades samaaegselt põhjaliku ülevaate rakenduse testitavatest teenustest [3]. Definitsiooni järgi on automaattestimine tarkvara kasutamine testide käivitamiseks ning prognoositavate tulemuste võrdlemiseks tegelike tulemustega [4].

Testimise automatiseerimine aitab jooksutada mitmeid testilugusid järjepidevalt ja korduvalt, kasutades testitava tarkvara erinevaid versioone ning keskkondasid. See on protsess, mis hõlmab endas järgnevate testvarade disainimist: [5]

Tabel 1. Testvarad selgitustega

Testvara	**Testvara kirjeldus**
Testilood	Detailne kirjeldus testi sisenditest, lähtetingimustest, protseduurist ja oodatud tulemusest
Dokumentatsioon	Testitava tarkvara nõuded, mille põhjal luuakse testilood
Tarkvara	Testilugude imiteerimise jaoks kasutatav tehnoloogia
Testkeskkonnad	Toodangust eraldiseisvad keskkonnad, kus toimub ainult testitava tarkvara testimine.
Testandmed	Testlugude jaoks loodud sisendandmed, mida kasutatakse testide käivitamisel

Testilood jagunevad positiivse ning negatiivse stsenaariumiga testlugudeks. Positiivse stsenaariumiga antakse testi sisenditeks korrektsed testandmed, mille käigus peaks rakendus töötama sihipäraselt. Negatiivse stsenaariumi puhul antakse testandmeteks vigased andmed ning kontrollitakse kas rakendus kuvab veateate ega tee midagi, mida ta tegema ei peaks [1].

Testvara on vajalik, et teostada põhiliseid testimise tegevusi:

- Testi eeltingimuste üles seadistamine ja kontrollimine
- Testide käivitamine
- Oodatavate ja tegelike tulemuste võrdlemine ning analüüs

Peamised põhjused manuaaltestimise automatiseerimiseks on: [3]

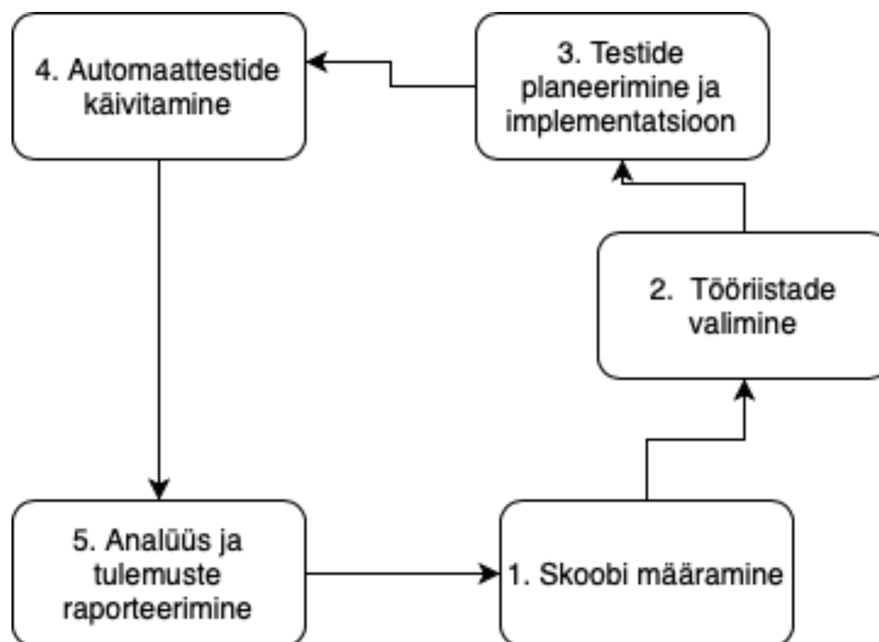
- Manuaaltestimine on aeganõudev
- Manuaalsed protsessid on veaohtlikud

- Automatiseerimine annab testijale rohkem aega uurivtestimiseks
- Varajane ning sage tagasiside
- Automatiseeritud regressioonitestid kindlustavad testitava rakenduse töökindluse

Silmas tuleb pidada, et kõikide rakenduste automaatne testimine pole alati võimalik või arukas. Rakendused peavad toetama testimisraamistikke ning olema ülesehituselt testitavad. See tähendab, et rakenduse mingi komponendi muutumisel ei tohiks liialt palju aega kuluda testide korrastamisele.

2.3 Automaattestimise protsess

Testimise automatiseerimisel lähtutakse ATLM metoodikast [4]. See on struktuurne lähenemine automaattestide implementeerimisele ja rakendamisele, mis on omakorda korduv protsess. See tähendab seda, et seda protsessi rakendatakse uuesti iga kord, kui testitaval rakenduse arendatakse välja uus funktsionaalsus, mis vajab testimist.



Joonis 1. Automaattestimise elutsükkel

Joonise 1 põhjal saab automatiseerimise protsessi jagada viieks etapiks: [4], [6]

1. Skoobi määramine
2. Tööriistade valimine

3. Testide planeerimine ja implementatsioon
4. Automaattestide käivitamine
5. Analüüs ja tulemuste raporteerimine

Paljud materjalid käsitlevad ATLM protsessis veel testkeskkonna loomist eraldi sammuna, kuid kuna puudub ühene määratlus, kus kohas see protsessis asub, siis lähtub töö autor E. Dustini käsitlusest, kus käsitletakse seda alamosana testide planeerimisell [4]. Käesolevas peatükis tutvustab töö autor igat etappi lähemalt ning kirjeldab etapiga seotud tegevusi.

2.3.1 Skoobi määramine

ATLM protsess algab testija valikutega, mida automatiseerida ning mida mitte. See on etapp, kus tuleb vastus leida küsimustele: [6]

- Millised osad rakendusest on võimalik katta automaattestidega?
- Mida on otstarbekas automatiseerida?

Tavapäraselt automatiseeritakse ära esmalt testilood, mis on seotud rakenduse kõige kriitilisemate teenustega – teenused, mis on lõppkasutajale kõige olulisemad. Neid teste käivitatakse tihti ning need on baasiks regressioonitestimisele.

2.3.2 Tööriistade valimine

Pärast skoobi määramist tuleb testijal otsustada, mis tehnoloogiat ning raamistikku või teeke kasutada testilugude implementeerimiseks. Soovitav on kasutada tehnoloogiat, mida kasutatakse ka rakenduse arendamisel. Sellisel juhul saavad tiimi arendajad pakkuda tuge automaattestide arendamisel tekkivatele küsimustele. Lisaks *code review'*de käigus kontrollivad arendajad, et toodangusse jõuaks otstarbekas kood [2].

Hea tava on valida kohe alguses raamistik, millega saab testida kõik olemasolevad ja teadaolevad tulevased funktsionaalsused. Nii välditakse tööriistade rohkust, mis lisab süsteemile komplekssust.

2.3.3 Testide planeerimine ja implementatsioon

Pärast tööriista valimist tuleb teha kõik vajalik, et oleks võimalik käivitada eelnevalt valitud testlood automatiseeritult. Nende tegevuste alla kuuluvad: [4]

- Testkeskkonna loomine
- Testandmete loomine
- Automaattestide arendamine testiloo sammude põhjal eelnevalt valitud tehnoloogiaga

Testkeskkonna loomisel luuakse toodangus olevale rakendusele sarnane keskkond. Sageli on tegu koopiaga, mis kasutab erinevat andmebaasi serverit toodangus oleva rakendusega võrreldes. Seal on samuti reaalsete andmete asemel testandmed [6].

Testandmete loomisel lähtutakse sarnaselt testimisele nii manuaalselt kui ka automatiseeritult. Hea tava on kasutada testandmete genereerimise tööriistu, mis võimaldavad luua valdkonnaspetsiifilisi andmeid. Nii luuakse realistlik testkeskkond kasutamata toodangus olevaid andmeid, mis muudab testimise turvalisemaks.

Automaattestide arendamisel kehtivad samad tavad, mis tavalise tarkvaraarenduse puhul. Lisaks sellele tuleks hoida automaattestid sõltumatutena kasutajaliidese muudatustest. Vastasel juhul mõjutavad muudatused testide tulemusi, mille tulemusena kulub rohkelt ressurssi testide korrashoiule [7]. Üks viis selleks on eesrakenduse elementide sildistamine spetsiaalsete atribuutidega, mis ennetavad testide ebaõnnestumist pärast disainimuutusi [8].

2.3.4 Automaattestide käivitamine

Automaattestide käivitamise koosneb kahest osast:

- Testlugude automaatne jooksutamine
- Testlugude jooksutamise logimine

Testlugude automaatne jooksutamine algab esmalt seadistamisega. Seadistamise käigus valitakse, mis testlugusid soovitakse jooksutada. Võimalik on jooksutada nii üht testlugu kui ka mitut testlugudest koosnevat testkomplekti. Lisaks sellele võimaldavad osad

raamistikud seadistada testlugude paralleelset jooksutamist veebiprojektidele. See tähendab, et testlugusid on võimalik jooksutada samaaegselt, kasutades erinevaid veebilehitsejaid [9].

Testlugude jooksutamisel logitakse testloo sammud. Sellega tõstetakse koodi jälgitavust, mis on eelkõige vajalik testide ebaõnnestumisel. Logimise kaudu veendutakse testi ebaõnnestumise põhjuses – vahel võib ette tulla, et test vajab lihtsalt hooldamist mõne komponendi muutuse tõttu või on viga koodis, mitte rakenduses.

Testide käivitamine lõppeb tagasiside saamisega. Tagasiside võib olla [4]:

- Õnnestumine – kõik testsammud täideti edukalt
- Ebaõnnestumine – üks või rohkem testsammu ebaõnnestus
- Vahele jäetud – testloo eeltingimused ei olnud täidetud või testija poolt määratud sihilik testloo vahele jätmine

Testlugude käivitamise puhul on hea tava seadistada testide jooksutamine pideva integratsiooni tööriistade kaudu automaatseks. Pidev integratsioon on tarkvaraarenduse tava, kus rakenduse arendajad panevad üles oma koodi ühisesse hoidlasse. Igat muudatust kontrollitakse automaatse protsessi kaudu – arendaja muudatuste liitmisel ühishoidlasse luuakse uus versioon tarkvarast, mida testitakse ühiktestide ja kasutajaliidese testidega. Nii tuvastatakse rakenduse tõrked varakult ja säästetakse aega veaparanduselt [10].

2.3.5 Analüüs ja tulemuste raporteerimine

Automaattestide tulemuste põhjal luuakse testimise raport, mida jagatakse ka osanikega. Heas raportis on ära näidatud iga testloo tulemused koos sammudega. See on baasiks analüüsile, mille põhjal otsustatakse kas on vaja edasisi testimise tegevusi või mitte [2].

2.4 Automaattestimissüsteemi hallatavus

Veebirakenduse automaattestimissüsteemi planeerimisel tuleb jälgida loodava süsteemi hallatavust. See tähendab, et arendataval süsteemil ei tohiks kuluda liiga palju ressurssi automaattestide muutmisele pärast testitava rakenduse kasutajaliidese muutumist [11].

Peamine põhjus automaattestide muutmiseks pärast kasutajaliidese muutumist on testide ebaõnnestumine. Automaattestide ebaõnnestumise ärahoidmiseks pärast kasutajaliidese muutumist tuleb arendajatel või testijatel lisada rakenduse eesrakendusse spetsiaalsed atribuudid. Parim praktika selleks on *data-test* atribuutide lisamine veebielementidele. Eelmainitud atribuudid on mõeldud ainult veebielementide tuvastamiseks automaattestides, mis tõttu arendajad neid ei muuda. Sellega välditakse testide ebaõnnestumist, mis on tingitud veebielementide disaini või teiste atribuutide muutustest [8].

Vahel võivad kasutajaliidese muudatused olla ulatuslikumad ja muuta tuleb *data-test* atribuutide väärtusi või testandmete väärtusi. Sellistel juhtudel oleks arukas hoida veebielementide identifikaatoreid ja testandmeid kindlates eraldiseisvates failides, kust käiakse väärtusi küsimas testide käigus. Selle tulemusena peab muutma väärtusi vaid vastavates failides.

2.5 Ettevõtte Proekspert taust testimise automatiseerimisel

Proekspert AS on infotehnoloogilisi teenuseid pakkuv ettevõtte, mis on asutatud aastal 1993. Ettevõttega on seotud mahukad manussüsteemi- ja veebiprojektid, mis eeldavad täpsemaid taustteadmisi kindlates valdkondades. See on üks põhjustest, miks ei ole välja kujunenud üht kindlat kvaliteediinseneride tiimi, kes vastutaks kogu ettevõtte projektide testimise eest. Samuti ei ole välja kujunenud üht kindlat automaattestimise arhitektuuri manussüsteemide ja veebisüsteemide erinevusest tingituna. Antud töö pakub välja ühe lähenemisviisi veebiprojektide testimise automatiseerimiseks.

2.5.1 Projekti taust

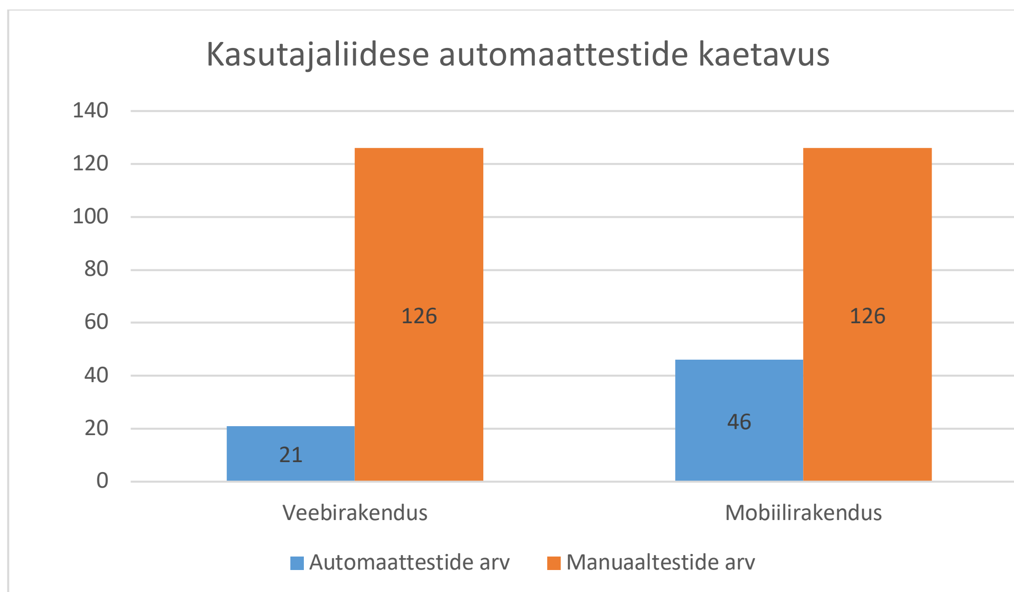
Töö autor realiseerib praktilise poole Proekspert AS kliendile, kes tegeleb farmaatsiatööstusega. Kliendi ärilistel kaalutlustel ei soovi ettevõte oma nime avaldada, mille tõttu kasutab töö autor pöördumisel terminit „klient“. Projekt on üle võetud teiselt ettevõttelt, mis tähendab, et Proekspert AS ei ole antud projekti nullist arendanud.

Kliendiprojekt on mõeldud veterinaariaametitele, kes käivad loomafarmides raporteid koostamas. See sisaldab endas veebi- ja mobiilirakendusi. Antud töö keskendub veebiprojektile automaattestimisesüsteemi loomisele. Veebiprojekt sisaldab endas järgnevaid tehnoloogiaid:

- Tagarakendus – NodeJS, MongoDB
- Eesrakendus – EmberJS
- Automaattestid – Puppeteer/Jest, Cypress

Veebirakenduse automaattestid on arendatud eelneva firma arendajate poolt. See kajastub ka automaattestide arendamisel – kuna esi- ja tagarakenduse arendus on toimunud JavaScripti raamistikega, on valitud sama tee ka automaattestide arendamisel. Puppeteer/Jest teekide kooslust on kasutatud automaattestides, mis on kõik seotud rakenduse kasutamise stsenaariumitega. Cypress'i on kasutatud vormiväljade eksisteerimise ja kasutaja sisselogimise funktsionaalsuse testimiseks. Cypressi automaatestide arendamisel on arendajad rõhku pannud veebipäringute valideerimisele. See tähendab, et kontrollitud on veebipäringute vastavusi, kuid mitte päringuga kaasnevat funktsionaalsust.

Põhiline fookus testimisel on olnud mobiilirakenduse testimise automatiseerimisel, mille automaattestide kaetavus on oluliselt suurem veebiprojekti omast. Kokku on 126 unikaalset testlugu, millest veebirakendusel on automatiseeritud 21 testi ning mobiilirakendusel 46 testi. Stsenaariumid, mida on testitud mobiilirakenduses, on jäetud veebirakenduses testimata, kuna on eeldatud, et ühise tagarakenduse tõttu pole vaja seda testi veebis korrata.



Joonis 2. Kasutajaliidese automaattestide kaetavus

2.5.2 Rakenduse taust

Eelnevalt mainituna põhineb rakenduse töövoog raportide koostamisel. Nende koostamiseks tuleb aga algselt luua rakenduses farmid, mis seostatakse riigi ja kindla kliendiga. Sellele järgneb loomakarjade lisamine farmi tõugude ja muu olulise info järgi. Pärast seda on veterinaaridel võimalik luua külastusi farmidesse, mille käigus luuakse raportid loomakarjade heaolu ja tervise kohta. Külastuse raportid koosnevad kümnetest moodulitest, mille põhjal antakse tagasisidet iga looma tervisele ja üleüldisele farmi seisule. Igal mooduli kohta luuakse skoor, mis iseloomustab looma hetkeolekut ning mille põhjal analüüsivad veterinaariaametid farmi vastavust nõuetele. Raportite põhjal luuakse ka erinevaid graafikuid jälgimaks farmide seisu läbi aja, mille põhjal saavad farmi omanikud samuti tagasisidet farmi kohta.

2.5.3 Automaattestimissüsteemi puudujäägid

Veebirakenduse praegusesse automaattestimissüsteemi süvenemisel ilmneb mitu probleemi:

- Mitmete raamistike/teekide kasutamine, mis ei loo süsteemile lisaväärtust
- Madal automaattestide kaetavus
- Automaattestide puudulik valideerimine

Järgnevas peatükis pakub töö autor välja lahenduse kavandi, mis täiustaks hetkest automaattestimissüsteemi. Analüüsitakse erinevaid UI raamistikke ning automaattestide parimaid praktikaid, et luua süsteem, mis oleks pilootprojektiks edasistele veebiprojektide automaattestimissüsteemidele.

3 Automaattestimissüsteemi kavandamine

Käesoleva töö autor pakub välja loodava automaattestimissüsteemi eelmises peatükis käsitletud ATLM metoodika kaudu. Sarnaselt ATLM metoodikale tehakse läbi kõik protsessi etapid ning töötatakse välja arhitektuur automaattestimissüsteemile.

3.1 Skoobi määramine

Automaattestimise ulatuse määrab hetkese veebirakenduse kasutajaliidese automaattestidega kaetavus, mis moodustab alla poole mobiilirakenduse automaattestidest. Mobiilirakenduses on 46 erineva positiivse stsenaariumiga testlugu, mis ei ole samuti ideaalne. Testida tuleks samuti ka negatiivseid stsenaariume [1]. Üks sellistest stsenaariumidest on näiteks blokeeritud kontoga sisselogimine, mis puhul kuvatakse kasutajale, et sisselogimine pole võimalik.

Esialgne plaan oleks saavutada võrdne testide kaetavus veebirakenduse ja mobiilirakenduse testimisel. Oluline on testida kõik kasutajaliidese funktsionaalsused veebis erinevate veebilehitsejatega, sest iga veebilehitseja interpreteerib rakendust erinevalt.

3.2 Tööriistade valimine

Tööriistade valimisel võrdleb töö autor kasutatavamaid testimisraamistikke või teeke projektis kasutatavaga. Selle tulemusena jätkatakse süsteemi arendamist raamistikuga või teegiga, mis täidab enim süsteemi nõudeid. Võrdlemiseks luuakse tabel, millele on vaja eelnevalt selgitavat legendi:

- Roheline ruut – Vastab tingimustele
- Kollane ruut – Vastab osaliselt tingimustele
- Punane ruut – Ei vasta tingimustele
- Tuntumad veebilehitsejad – Chrome, Safari, Edge, Firefox
- Tuntumad tehnoloogiad – Java, C#, Javascript, Python

Tabel 2. UI testimisraamistike võrdlus [12], [13], [14], [15], [16]

Tööriist	**Toetatud tehnoloogiad**	**Toetatud veebilehitsejad**	**Valideerimine**	**Raport**
Selenium	Kõik tuntumad	Kõik tuntumad	Lisateegid (JUnit, testNG)	Lisateegid
Selenide	Java	Kõik tuntumad	Sisseehitatud	Lisateegid
Cypress	Javascript	Kõik tuntumad v.a Safari	Sisseehitatud	Sisseehitatud
Puppeteer	Javascript	Chrome, Chromium	Lisateegid (Jest)	Lisateegid
Robot framework	Python, Jython (Java), IronPython (.NET)	Kõik tuntumad	Sisseehitatud	Sisseehitatud aga infovaene
Nightwatch.js	Javascript	Kõik tuntumad	Sisseehitatud	Lisateegid

Lähtudes tabeli 2. tulemustest saab selgeks, et projektis kasutatav tööriist Puppeteer töötab vaid Chrome ja Chromium veebilehitsejaga. Uurides Google Analytics'ist veebirakenduse kasutatavust veebilehitsejate kaupa, ilmneb, et oluline on katta ära Chrome, Firefox ja Edge brauserid. Mainitud 3 brauseri sessioonid moodustavad kokku 47.93% kõigist sessioonidest. Statistika puhul tuleb arvestada, et kaasatud on mobiilirakenduste sessioonid, mis moodustavad 45,38%.

Browser	Acquisition			Behavior		
	Sessions	% New Sessions	New Users	Bounce Rate	Pages / Session	Avg. Session Duration
	5,322 % of Total: 100.00% (5,322)	23.24% Avg for View: 23.21% (0.16%)	1,237 % of Total: 100.16% (1,235)	23.13% Avg for View: 23.13% (0.00%)	13.88 Avg for View: 13.88 (0.00%)	00:13:59 Avg for View: 00:13:59 (0.00%)
1. Android Webview	1,891 (35.53%)	25.91%	490 (39.61%)	24.11%	9.06	00:13:33
2. Chrome	1,648 (30.97%)	20.21%	333 (26.92%)	17.23%	17.71	00:14:02
3. Safari (in-app)	524 (9.85%)	18.89%	99 (8.00%)	28.05%	11.23	00:09:28
4. Firefox	459 (8.62%)	14.81%	68 (5.50%)	30.28%	25.36	00:26:29
5. Edge	444 (8.34%)	23.42%	104 (8.41%)	22.30%	13.22	00:13:12
6. Safari	202 (3.80%)	49.01%	99 (8.00%)	28.71%	12.24	00:06:32
7. Samsung Internet	117 (2.20%)	32.48%	38 (3.07%)	35.04%	6.91	00:06:04
8. Opera	21 (0.39%)	4.76%	1 (0.08%)	4.76%	34.62	00:26:24

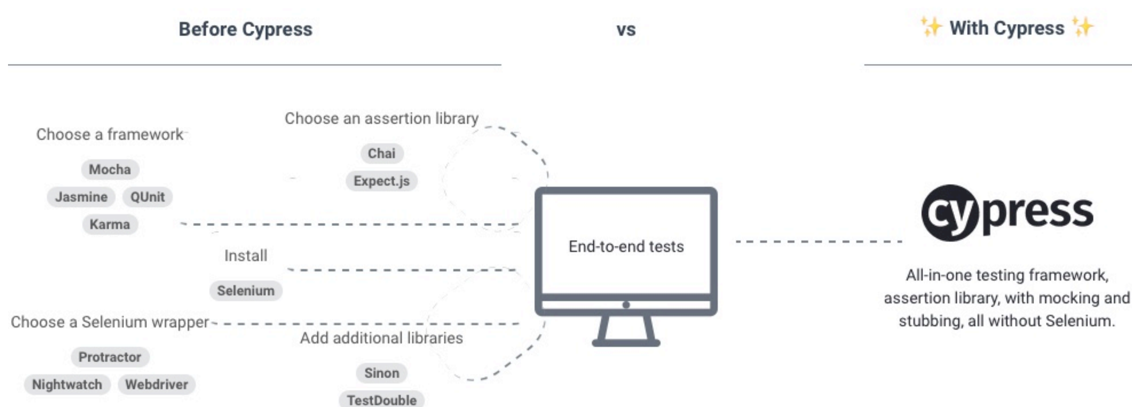
Joonis 3. Testitava rakenduse kasutus erinevate veebilehitsejatega

Vastupidiselt Puppeteerile täidab Cypress peaaegu kõik nõuded. Cypress on kasutajaliidese testimise raamistik, millesse on integreeritud kõik testimiseks vajaminevad teegid. Raamistiku eripära teiste tööriistadega võrreldes on ainulaadne arhitektuur – paljud teised tööriistad on Selenium raamistiku edasiarendused, mis omavad ühiseid probleeme.

Enne Cypressi raamistikku tuli kasutajaliidese automaattestimiseks sooritada järgmised tegevused [17]:

- Testimisraamistiku valimine
- Valideerimise ja/või raporteerimise teegi valimine ja integreerimine
- Seleniumi allalaadimine
- Testmeetodite arendamine asünkroonselt, et testid jookseksid kiiremini

Cypress raamistiku näitel on loodud raamistik, mis omab ainulaadset arhitektuuri ega sõltu Selenium raamistikust. Raamistiku arhitektuuri üheks osaks on sisseehitatud meetodite asünkroonne kasutusviis, mis teeb testide läbimise kiiremaks kui Selenium raamistikuga. Cypress'i on integreeritud levinumad valideerimise ja raporteerimise teegid. Süsteemi loomisel ei kulu aega erinevate teekide integreerimisele ning ei teki ka ühilduvuse probleeme, kuna raamistiku arendusel on eelnevalt testitud kokkusobivusi [17].



Joonis 4. Cypress raamistiku võrdlus teiste tööriistadega [17]

Lisaks sellele on raamistikul veel implementeeritud [15]:

- Graafiline kasutajaliides – lihtsustatud testide jooksutamine ning jälgimine
- Ajaränne – ekraanitõmmis igast testloo sammust koos logiga
- Automaatne ootamine – testides ei ole vaja implementeerida ootamist veebielementide järgi
- Mugavdatud *debug'imine*
- Kohene värskendus – pärast koodi muutmist käivitatakse eelnevalt jooksutatud testid uuesti

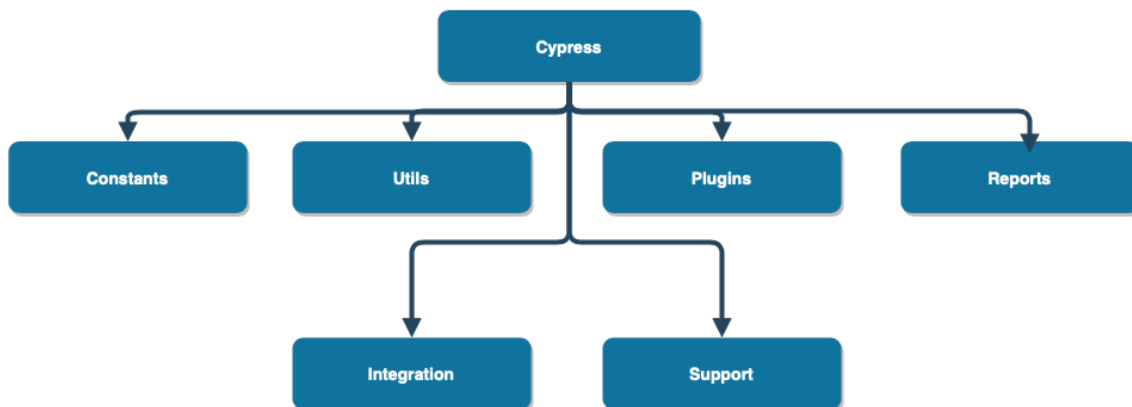
Töö autor jätkab automaattestimissüsteemi testide arendamist Cypress raamistikuga, sest sellega on võimalik katta kõik rakenduse testimiseks vajalikud nõuded ning ükski teine tööriist eelnevas võrdluses ei viita raamistiku väljavahetamise vajadusele. Rakenduse enda arendustiimis on Javascripti teadmised, mis kindlustab, et läbi *code review*'de jõuab toodangusse otstarbekas kood. Ainus põhjus teiste raamistike valimiseks oleks Safari veebilehitseja toetus, kuid see probleem on juba raamistiku arendajatele tõstatatud kasutajate poolt ning *feature request*'le määratakse hetkel prioriteeti. Samuti on projektis kaasatud vanem kvaliteediinsener, kes sooritab tihti uurivtestimist Safari's, mis teeb veebilehitsejast tingitud tõrgete jõudmise toodangusse väga madalaks.

3.3 Testide planeerimine ja implementatsioon

Testide planeerimist alustatakse testkeskkonna üles seadistamisega. Testkeskkond on loodud projekti DevOps inseneri poolt kasutades majutusteenusena Google Cloudi ja pidevintegratsiooni tööriista Jenkinsit.

Testandmete genereerimiseks kasutab töö autor javascripti teeki *faker.js*, mis on üks levinumaid teeke andmete genereerimiseks. Teegiga saab genereerida realistlikke andmeid piirkonnaspetsiifiliselt. Näiteks saab genereerida ettevõtte ja persooninimesid, mis on omased Tšehhi kultuurile. Nii saab kontrollida rakenduse vormide *unicode'i* tuge ja nõuetele vastavust eri kultuuriruumides [18].

Testide implementeerimisel jälgitakse Cypress'i dokumentatsiooni, raamistiku arendajate soovitusi ning testlugude implementeerimise parimaid praktikaid. Veebitestide repositooriumi loomisel jälgitakse Cypressi dokumentatsiooni poolt soovitatavat arhitektuuri [19]:



Joonis 5. Automaattestide repositooriumi arhitektuur

Kaustas *Constants* hoitakse igas failis vastava kasutajaliidese funktsionaalsuse konstante. Need sisaldavad testandmeid, mis on kas töö autori poolt eelnevalt defineeritud või genereeritud teegi faker.js abil. Lisaks on seal funktsionaalsusega seotud veebielementide identifikaatorid, mille kaudu tuvastatakse elemendid automaattestides.

Utils sisaldab endas testimiseks vajaminevaid abimeetodeid, mis on jagatud ära erinevatesse failidesse rakenduse funktsionaalsuse põhiselt. Lähtutud on Gleb Bahmutov'i kasutajaliidese testimisest abimeetodite kaudu [20]. Meetodid sisaldavad endas kasutajat imiteerivaid tegevusi. Töös käsitletava rakenduse mõistes kuuluvad sinna alla CRUD kontrollerite tegevused, milleks on näiteks farmikülastuste loomine, muutmine, kustutamine ja lugemine.

Plugins kaust on Cypressi installeerimisel loodud kaust, kuhu kuuluvad testimiseks vajalikud pluginad. Töö autor lisab sinna plugina, millega kustutakse peale testimist kasutajaliidese poolt loodud andmed.

Support kaustas hoitakse raamistiku enda abimeetodeid ning globaalset konfiguratsiooni. Raamistiku enda abimeetodite alla lisatakse meetodid, mis on laialt kasutatavad ja ei ole

otseselt seotud ühegi teenusega. Üheks selliseks funktsionaalsuseks on kasutaja autentimine sisselogimisel.

Integration kaustas asuvad kõik testlugudega täidetud testkomplektide kaustad, mis on failide kaupa ära jaotatud sõltuvalt funktsionaalsusest, mida testitakse. Pärast testlugude jooksutamist luuakse *Reports* kausta testide raportid.

Automaattestide implementeerimisel järgitakse standardset struktuuri, mis koosneb 3 faasist [19]:

- Rakenduse algseisundi sätestamine
- Tegevuse imiteerimine
- Rakenduse lõppseisundi valideerimine

Joonisel 6 näidatakse automaattesti 3 faasi Cypressi raamistiku näitel. Rakenduse algseisundi sätestamine toimub before ja beforeEach meetodites. Tegevuse imiteerimine toimub kommentaari "Take an action" all, kus teostatakse farmi loomise negatiivne stsenaarium. Lõpus valideeritakse vormi väljade veateated kommentaari "Make an assertion about the resulting application state" all.

```
describe('Add new farm', () => {
  /**
   * Set the state before running all tests
   */
  before(() =>  {
    cy.login('user8');
  });

  /**
   * Set the state before every test
   */
  beforeEach(() => {
    cy.restoreLocalStorage();
    cy.server();
    cy.visit('/farms/add');
    createRoute('POST', 'farms', 'farm');
    clearFarmFields();
  });

  it('As a user, I cannot add a farm without a name, company and region.',
() => {
    // Take an action
    const farm = faulty_farms.farm1;
    addFarm(farm, contacts);

    // Make an assertion about the resulting application state.
    cy.getByTestId(farmTestIds.presentError('name')).should('exist');
    cy.getByTestId(farmTestIds.presentError('company')).should('exist');
    cy.getByTestId(farmTestIds.presentError('region')).should('exist');
  });
});
```

Joonis 6. Automaattesti 3 faasi

3.4 Automaattestide jooksutamine ja raporteerimine

Automaattestide jooksutamiseks seatakse vastavalt võimalustele üles üks või mitu masinat, kus teste jooksutatakse. Mitme masina olemasolul on võimalik sooritada paralleelset testide jooksutamist. Testide jooksutamiseks kasutatakse pideva integratsiooni tööriista Jenkinsit, mille kaudu veendutakse, et iga uue tarkvaraversiooni

tekkel käidaks läbi ka kogu rakenduse automaattestid. Testide tulemused raporteeritakse Cypress'i testide juhtimislaud (ingl. keeles *dashboard*'), milles sisaldub:

- Õnnestunud, ebaõnnestunud, vahele jäetud testide arv
- Ebaõnnestunud testide täielik vealogi (inglise keeles *stack trace*)
- Ekraanitõmmised ebaõnnestunud testi sammudest
- Video kogu testide jooksutamisest brauseris
- Masinate ülevaade (paralleelse testimise puhul)

Järgnevas peatükis tutvustab töö autor loodud automaattestimissüsteemi, mis on loodud selle peatüki kavandi põhjal.

4 Automaattestimissüsteem

Regressioonitestimise automatiseerimiseks ettevõttele Proekspert AS on töö autor loonud automaattestimissüsteemi, mille arhitektuuri saab kasutada ka teistes veebiprojektides. Loodud süsteem ei ole lõplik ega täielik testide kaetavuse poolest, kuid piisav, et olla näidisprojekt edasiste automaattestimissüsteemide loomiseks.

4.1 Süsteemi seadistamine

Sarnaselt testitava rakenduse ees- ja tagarakenduse repositooriumile on testsüsteemi sõltuvuste haldamiseks kasutatud *Node Package Manager*'i. Repositooriumi juurkaustas asub fail *package.json*, kus asuvad kõik teekide sõltuvused, mida süsteem kasutab. Süsteemi seadistamiseks tuleb kasutajal minna terminalis projekti juurkausta ja jooksutada käsku:

```
npm install
```

Süsteemi sõltuvuste edukal installeerimisel kuvatakse kasutajale terminalis teegid ning teekide koguarv, mis installeeriti failist. Lisaks hoitakse *Package.json* failis *scripts* objekti all testide terminalist jooksutamise käsklusi. Käsklused on võti-väärtus paarid, kus võtmeteks on kasutaja poolt defineeritavad nimetused ja väärtusteks käsklused, mida jooksutatakse terminalis. Neid kasutatakse lühendamaks täispikkasid käsklusi, mis sisaldavad endas testide jooksutamise konfiguratsiooni. Järgnevalt on toodud näide *scripts* objektist failis, mis sisaldab endas Cypress'I testide jooksutamist Google Chrome veebilehitsejaga.

```
"scripts":   {
       "cy:run:chrome":  "cypress  run  --record  --browser  chrome  --spec
‘cypress/integration/**’"
       }
```

Joonis 7. Terminalist testide jooksutamise lühendatud käsklus

Vastavat käsklust on hiljem võimalik jooksutada terminalist käsuga:

```
npm run cy:run:chrome
```

Raamistiku enda konfigureerimiseks on juurkaustas fail *cypress.json*, kus on järgmised globaalsed seadistused:

Tabel 3. Cypress raamistiku globaalsed seadistused

Parameeter	**Parameetri tüüp**	**Parameetri väärtus**	**Parameetri kirjeldus**
baseUrl	Tekst	"http:/localhost:4200"	Testide jooksutamise keskkonna URL'i vaikimisi väärtus
reporter	Tekst	"mocha-multi-reporters"	Raamistiku poolt kasutatav teek raporteerimiseks
reporterOptions	JSON	configFile	Raporteerimise seadistused
configFile	Tekst	"reporter-config.json"	Raporteerimise seadistuste failinimi
defaultCommandTimeout	Täisarv	12000	Raamistiku päringuviivituse aeg millisekundites
requestTimeout	Täisarv	15000	Veebipäringute päringuviivituse aeg millisekundites
projectId	Tekst	4wx3rz	Unikaalne Cypress projekti identifikaator

4.2 Süsteemi funktsionaalsus

Järgnevas peatükis tutvustab töö autor automaattestimissüsteemi funktsionaalsust, mis sisaldab endas:

- Testandmete loomist ja kasutamist
- Automaattestide testkomplektide loomist
- Automaattestide jooksutamist
- Tulemuste raporteerimist ja salvestamist

4.2.1 Testandmete loomine ja kasutamine

Testandmete loomine toimub *Constants* kausta Javascripti failides, kus andmed on ära jagatud kasutajaliidese funktsionaalsuse kaupa. Andmed hoitakse JSON objektis võti-väärtus paaridena, kus võtmeks on identifikaator ja väärtuseks omakorda JSON objekt,

mis sisaldab faker.js teegi poolt genereeritud andmeid või töö autori poolt eelnevalt defineeritud andmeid.

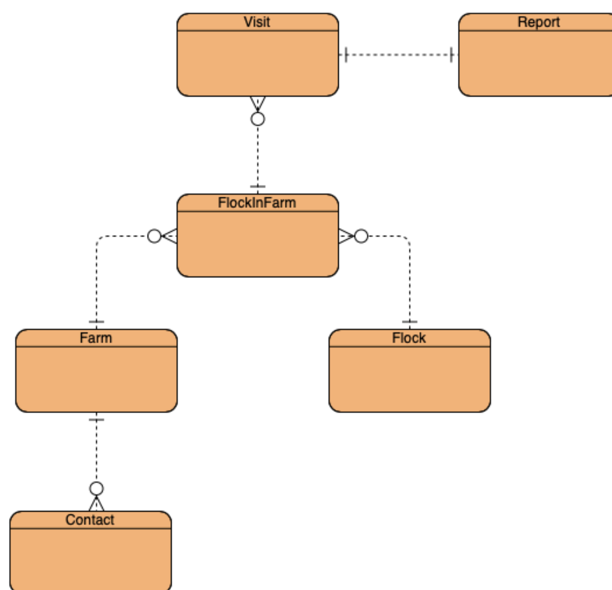
```
export const contacts = {
  contact1: {
    firstName: faker.name.firstName(),
    lastName: faker.name.lastName(),
    telephone: faker.phone.phoneNumber(),
    email: faker.internet.email()
  },
  contact3: {
    firstName: faker.name.firstName(),
    lastName: faker.name.lastName(),
    telephone: '+37255556666',
    email: faker.internet.email()
  }
```

Joonis 8. Testandmete genereerimine Javascriptis faker.js teegiga

Samades failides hoitakse identifikaatoreid, millega määratakse veebielementide asukohad ning täidetakse need testandmetega. Veebielementide tuvastamine käib raamistiku *getByTestId* meetodi kaudu, mis võtab argumendiks sõne kujul identifikaatori. Meetod otsib veebist elemente, mille küljes on atribuut *data-test-id* ning mis vastab meetodi argumendi sõnele. Elemendi tuvastamisel sooritatakse elemendiga tegevus, milleks võib olla näiteks joonisel 8. loodud testandmetega välja täitmine.

4.2.2 Automaattestid

Testimissüsteemi kasutaja tegevuste imiteerimise osa asub *Utils* kaustas, mis sisaldab endas taaskasutatavaid meetodeid. Neid meetodeid kasutatakse testlugude sammudes, testlugude eeltingimuste ning testijärgsete tegevuste realiseerimiseks. Meetodite mõistmiseks on töö autor koostanud ERD skeemi rakenduse olemitest, mida kasutab rakenduse tavakasutaja. Reaalsuses on süsteem oluliselt keerulisem.



Joonis 9. Rakenduse testimise alla kuuluvad olemid

Automaattesti ja testi eeltingimuste realiseerimist Cypress'is on näidatud joonisel 6 Testkomplektide nimetamiseks kasutatakse meetodit *describe*, mille esimeseks argumendiks on sõne ning teiseks argumendiks meetod, mis sisaldab endas testi eeltingimusi, automaatteste ning testijärgseid tegevusi.

Testide põhiliseks eeltingimuseks on kasutaja sisselogimine, mille funktsionaalsus on realiseeritud *Cypress Custom Commands*'ide kaudu. See kujutab endast raamistikule meetodite lisamist, mis ei ole otseselt ühegi kasutaja poolt kasutatava funktsionaalsusega seotud. Nende meetodite alla kuuluvad veel veebiseansi salvestamise ja taastamise meetodid, sest raamistik kustutab seansi peale igat testi või *before, beforeEach, after, afterEach* meetodit [21].

Testi eeltingimuste seadmiseks kasutatakse meetodeid *before* ja *beforeEach.* Enne kõikide testide jooksutamist realiseeritakse kõigepealt *before* meetod, kus joonisel 6 realiseeritakse rakendusse sisselogimine. Sellele järgneb *beforeEach* meetod, mida jooksutatakse enne igat testi. Meetodiga määratakse ära iga testi alguspunkt ja taastatakse sisselogitud kasutaja veebiseanss meetodist *before.*

Testid realiseeritakse meetodiga *it*, mille esimeseks argumendiks on testi nimetuse sõne ning teiseks argumendiks meetod, mis sisaldab endas automaattesti ülejäänud kahte faasi

– testimise tegevused ja valideerimine. Tegevused realiseeritakse *Utils* kausta abimeetodite kaudu. Raamistiku valideerimisel tuvastatakse elemendid *getByTestId* või *queryByTestId* meetodite kaudu, mida hiljem valideeritakse *should* meetodiga, mille argumendiks on tingimus, mis peab olema täidetud testi läbimiseks. Joonise 6 näitel on kasutatud meetodit *should* argumendiga '*exist'*, mis kontrollib veebielemendi eksisteermist. Veebielementide mitteeksisteerimist valideeritakse raamistiku *queryByTestId* meetodi kaudu, sest *getByTestId* valideerimine ebaõnnestub automaatselt, kui elementi ei suudeta tuvastada.

4.2.3 Testide jooksutamine

Testide jooksutamiseks on projekti juurkaustas *package.json* failis määratud scripts objektis käivitamise lühendatud käsklused. Käsklused sisaldavad endas jooksutamise seadistusi, mida seadistatakse kujul (Vaja läheb NPM 5.2+ versiooni):

```
npx cypress run --<seadistuse nimetus> <seadistuse väärtus>
```

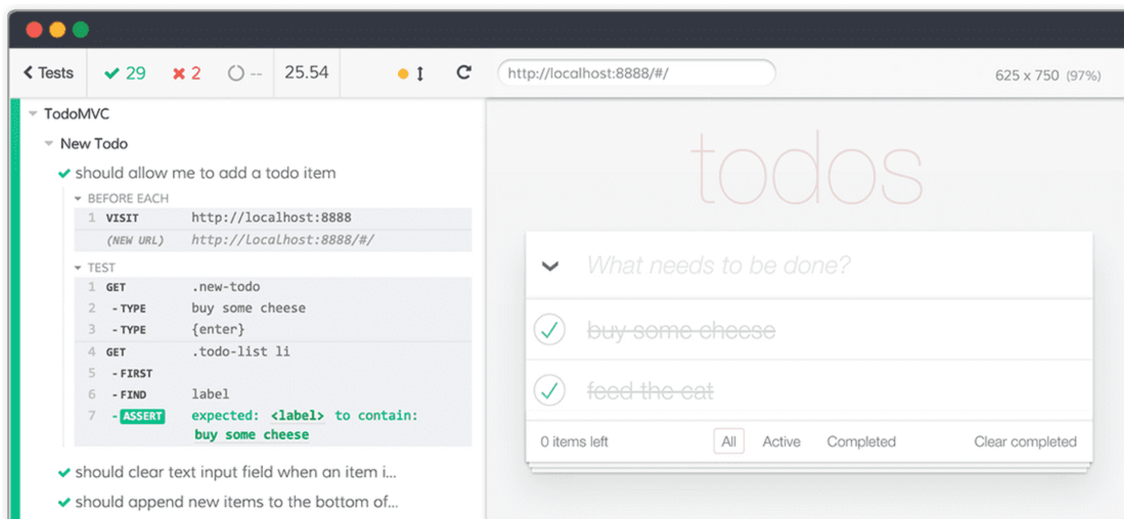
Järgnevalt on välja toodud töö raames kasutatavad seadistused koos seletustega:

Tabel 4. Automaattestide jooksutamise seadistused

Parameeter	**Parameetri tüüp**	**Parameetri näiteväärtus**	**Parameetri kirjeldus**
config-file	Tekst	'qa-common-config.json',	Testide käivitamise konfiguratsiooni faili asukoht juurkausta suhtes, mis kirjutab üle *cypress.json* parameetrite väärtused
record	-	-	Testi jooksutamise tulemuste salvestamine
spec	Tekst	'cypress/integration/ /**'	Testi(de) asukoht juurkausta suhtes
browser	Tekst	chrome	Veebilehitseja määramine testideks
headless	-	-	Testide jooksutamine veebilehitsejat kuvamata

Parameetrit config-file kasutatakse seadistamaks süsteemi vastavalt keskkonnale, kus seda kasutatakse. Töö autor kasutab juurkaustas asuvat faili ’qa-common-config.json’, kus on sätestatud testkeskkonna konfiguratsioon, mida kahjuks kliendi soovide tõttu avaldada ei saa, sest veebilehe URL’i parameeter *baseUrl* viitab kliendi ettevõtte nimele.

Parameeter *record lisamisel* luuakse lisaks testraportile ka videosalvestus ja ekraanitõmmised testide samm-sammulisest sooritamisest Cypress graafilise kasutajaliidesega, mis lihtsustab testide ebaõnnestumise põhjuste leidmist. Parameetri lisafunktsionaalsustest tuleb lähemalt juttu järgmises alampeatükis.



Joonis 10. Cypress’i testkomplekti sooritamine graafilise kasutajaliidesega [15]

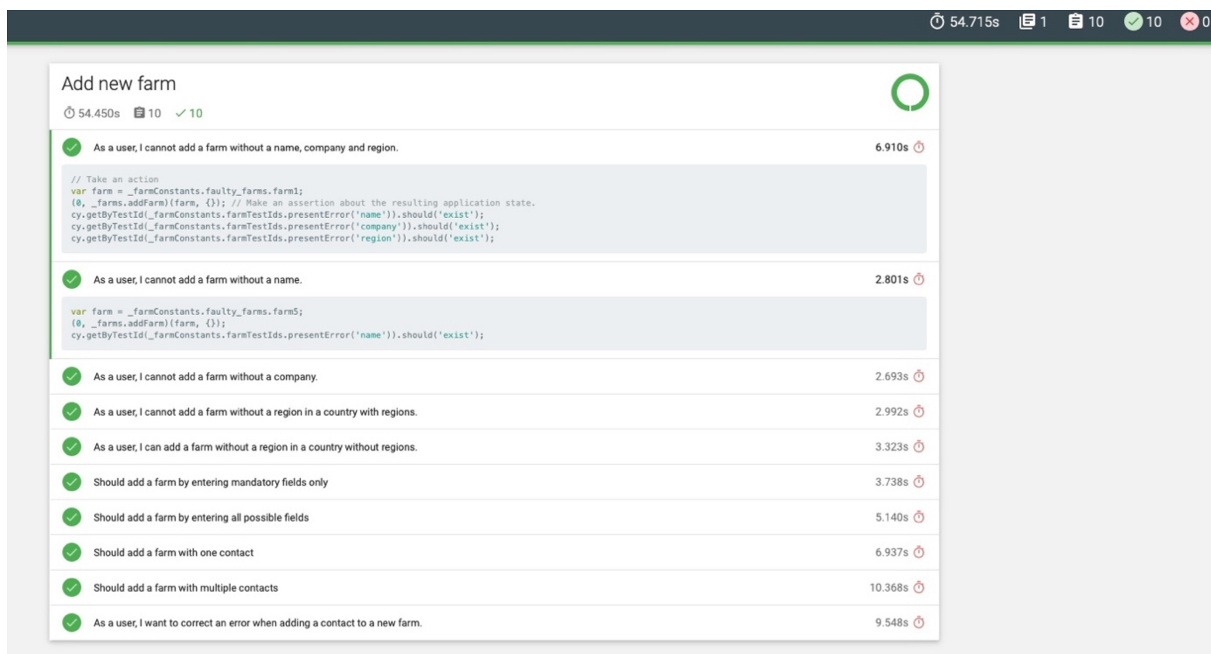
4.2.4 Testide raporteerimine

Raamistikuga saab luua mitmeid raporte, mis seadistatakse *reporter-config.json* failis. Töö raames kasutab autor raamistiku sisseehitatud Spec ja Mochawesome raporteid, mis seadistatakse järgnevalt:

Tabel 5. Raporteerimise konfiguratsioon

Parameeter	**Parameetri tüüp**	**Parameetri väärtus**	**Parameetri kirjeldus**
reporterEnabled	Tekst	“spec, mochawesome”	Raamistiku raporteerimise teekide määraja
mochawesomeReporterOptions	JSON	reportDir, overwrite, json	Mochawesome teegi seadistused
reportDir	Tekst	“cypress/reports”	Raporti loomise kausta asukoht
overwrite	Tõeväärtus	false	Alati luuakse uus raport ega kirjutata vana üle
json	Tõeväärtus	false	Lisaraporti loomine JSON objektina

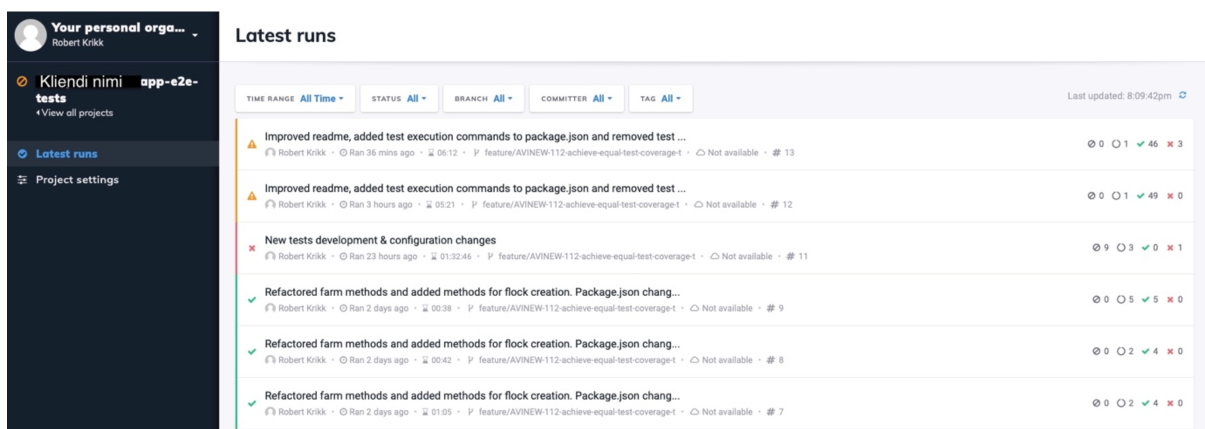
Spec raporteerijaga luuakse raport terminalis kohe pärast testide jooksutamist, mis mugavdab automaattestide arendamisfaasi ja terminalist käivitamise testimist. Mochawesome raporteerijaga luuakse raport html faili kujul, mis on disaini poolt kaunim ning kõlbab projekti huviisikutele jagamiseks pärast testkomplektide jooksutamist.



Joonis 11. Mochawesome testkomplekti raport

Eelnevalt mainitud konfiguratsioonid loovad raportid ainult masinasse, kus testid jooksutatakse. Reaalsuses jooksutatakse kasutajaliidese automaatteste mitmetes erinevates masinates, mis teeb informatsiooni kättesaamise keerulisemaks. Töö autor on selleks seadistanud üles testide salvestamise Cypress *dashboard*'i, kuhu lisatakse kõik testkomplektide jooksutamise tulemused, mis on sooritatud pideva integratsiooni automaattestimise käigus [22].

Testkomplektide salvestamise keskkonna seadistamisel luuakse Cypress projekti identifikaator *projectId package.json* faili ning genereeritakse võti, mis tuleb igal testi käivitamisel anda kaasa *key* parameetriga või seadistada masina *environment variable*'na. See on loodud testide käivitamise ja salvestamise autentimiseks [22].



Joonis 12. Testkomplektide jooksutamise tulemused Cypress dashboard'is

Järgmises peatükis analüüsib töö autor edasiarendatud automaattestimissüsteemi, võrreldes uue süsteemi funktsionaalsust eelmise süsteemi funktsionaalsusega.

4.3 Automaattestide integreerimine Jenkinsi

Testitava rakenduse puhul on oluline kontrollida, et uus muudatus ei tekitaks tõrkeid rakenduse funktsionaalsuses. Projektis testitakse tagarakenduse igat uut muudatust ühiktestidega, mis käivitatakse pideva integratsiooni tööriistas Jenkins. Veendumaks, et muudatus pole tekitanud tõrkeid eesrakenduses, integreeris töö autor loodud kasutajaliidese automaattestid Jenkinsi keskkonnaga.

Testitavat rakendust arendab kokku 8 arendajat, mille tulemusena muudetakse arenduskeskkonna koodi iga tund. Töö autori ja arendajate kokkuleppel käivitatakse kasutajaliidese automaatteste iga 6 tunni tagant, et mitte koormata liigselt masinat, kus

käib lisaks automaattestide käivitamisele ka testitava rakenduse mobiilirakenduste *build* etapp. Tulevikus hangitakse testide jooksutamiseks eraldiseisev masin, millest räägib töö autor lähemalt peatükis „Edasised tegevused“.

Automaattestide käivitamiseks Jenkinsis luuakse keskkonnas uus *pipeline*, mis seotakse automaattestide Bitbucket repositooriumiga. Automaattestide repositooriumis asub juurkaustas fail nimega Jenkinsfile, mille sees määratletakse testide käivitamise masin ja järgnevad kasutajaliidese testimise etapid:

1. Kasutajaliidese automaattestimine Chrome brauseriga
2. Kasutajaliidese automaattestimine Edge brauseriga
3. Kasutajaliidese automaattestimine Firefox brauseriga

5 Tulemuste analüüs

Käesolevas peatükis analüüsitakse loodud süsteemi võimekusi ning võrreldakse neid eelneva süsteemi võimekustega, et anda hinnang tehtud töö kasulikkusele. Selleks sooritatakse kahe süsteemi vahel A/B testimine, kus võrreldakse kumb süsteemidest täidab eesmärki paremini järgmiste parameetrite järgi:

- Veebilehitsejate tugi
- Testide ülesehitus
- Testide kaetavus
- Testkomplektide jooksutamise kiirus
- Eelneva ja hetkese süsteemi arhitektuuri võrdlus

5.1 Veebilehitsejate toe võrdlus

Tabelist 2. lähtudes, ilmneb, et eelnevalt toetas automaattestimissüsteem Chrome'i ja Chromiumi veebilehitsejaid. Automaattestide üleminekul Cypress raamistikule on nüüdsest toetatud süsteemis järgmised veebilehitsejad [23]:

- Google Chrome
- Chromium
- Canary
- Edge (Sealhulgas Edge Beta, Edge Canary, Edge Dev)
- Electron
- Firefox (Sealhulgas Firefox Developer Edition ja Firefox Nightly)

5.2 Testide ülesehituse võrdlus

Projekti eelmised automaattestid olid kirjutatud arendajate poolt, mis kajastusid automaattestide ülesehituses. Osade testide kirjutamisel oli kontrollitud ainult veebipäringu vastust. Üks selline testidest oli korrektse kasutajanime ja parooliga sisselogimine, kus pärast sisselogimist kontrolliti, kas avalehele suunates saadakse veebipäringu staatuskoodiks 200.

```
it('I want to login by pressing Enter', () => {
	cy.get('input[type="email"]').type('user@password.com');
	cy.get('input[type="password"]').type('password{enter}');
	cy.request('/').then(res => {
		expect(res.status).to.eq(200);
	});
});
```

Joonis 13. Cypress testi ülesehituse eelnevas süsteemis

Töö autor viis joonise 14. näitel testid üle kujule, kus testi lõpus valideeritakse lõpptingimused, kontrollides veebielementide eksisteerimist. Iga sisseloginud kasutajale kuvatakse navigeerimiselemendid, mille olemasolul valideeritakse sisseloginud kasutaja seanss. Elementide tuvastamisel kasutatakse *data-* atribuute, sest eesrakendus võib tulevikus muutuda, mille tulemusena testid ebaõnnestuksid. Joonise 13 näite puhul võib olla selliseks muutuseks rakendusse sisselogimise võimaldamine kasutajanimega. Sellisel juhul ei kasutataks enam sama *type* atribuudi väärtust, sest veebivormi välja väärtus ei peaks enam ainult e-maili kujul olema.

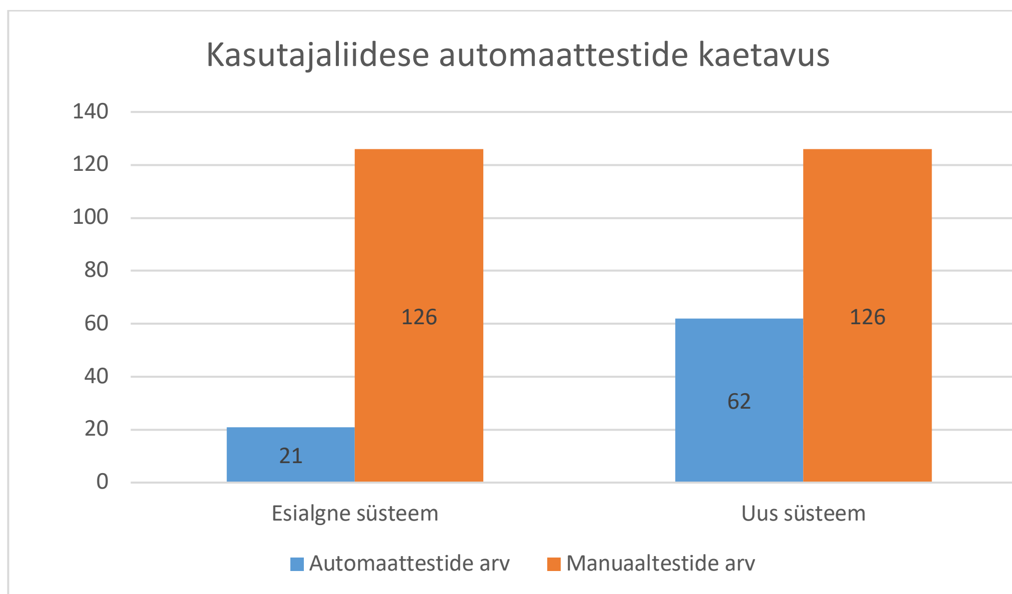
```
it('I want to login by pressing Enter', () => {
	cy.getByTestId('login-email-input').type('user@password.com');
	cy.getByTestId('login-password-input').type('password{enter}');

	cy.getByTestId('navigation-home-button').should('exist');
	cy.getByTestId('navigation-farms-button').should('exist');
	cy.getByTestId('navigation-visists-button').should('exist');
});
```

Joonis 14. Cypress testi ülesehitus uues süsteemis

5.3 Testide kaetavuse võrdlus

Projekti ülevõtmisel oli repositooriumis kokku 21 kasutajaliidese automaattesti – 6 Cypress raamistikus ning 15 Jest/Puppeteer raamistikega. Mobiilirakendustel oli kokku 46 kasutajaliidese automaattesti. Autori esialgne plaan oli saavutada võrdne automaattestide kaetavuse ulatus mobiilirakenduste ja veebirakenduse testimisel.

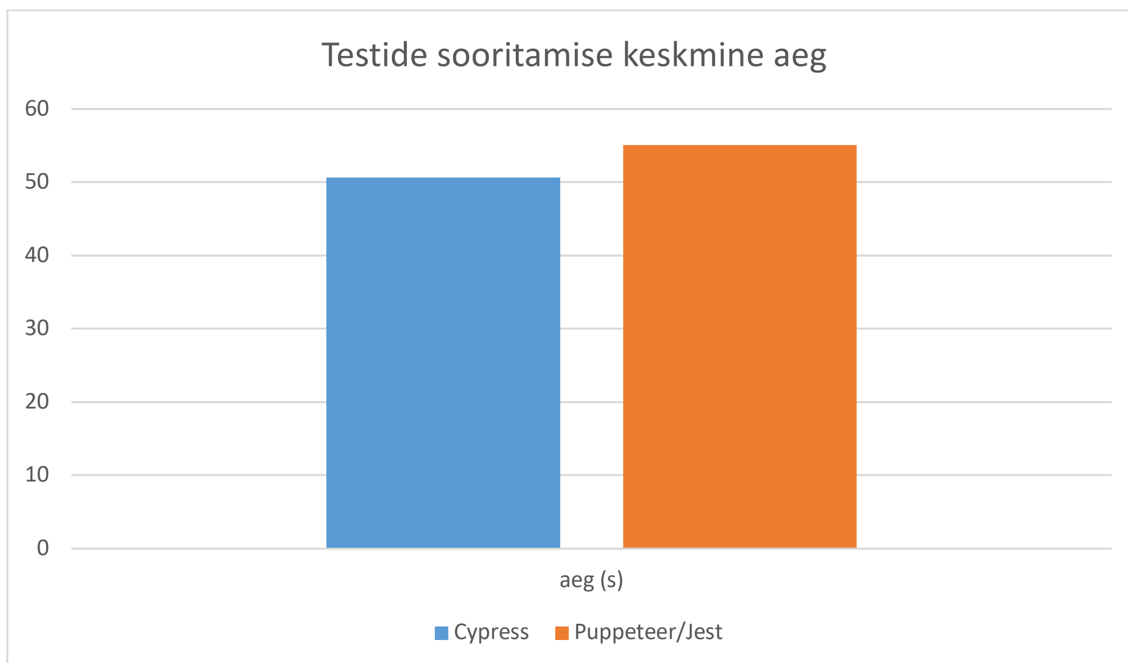


Joonis 15. Uue ja vana süsteemi kasutajaliidese automaattestide kaetavused

Veebirakendusel on kokku 126 manuaaltesti. Esialgses süsteemis oli automatiseeritud 16,7% manuaaltestidest ehk 21 testi. Uues süsteemis on automatiseeritud 49,2% testidest. Võrreldes algse olukorraga on testide kaetavuse ulatus paranenud ligi 3 korda. Automaattestide arendamine jätkub projektis edasi seni kuni kõik olulised funktsionaalsused on kasutajaliidese testidega kaetud.

5.4 Testkomplektide jooksutamise kiirus

Süsteemide võimsuse võrdlemiseks jooksutatakse mõlemas keskkonnas samasid testlugusid. Kokku jooksutatakse mõlemas süsteemis 15 erinevat stsenaariumit kasutades Chrome veebilehitsejat ning võrreldakse testide sooritamise koguaega. Testlugusid jooksutatakse 5 korda ja tulemuste põhjal arvutatakse välja keskmine testide sooritusaeg.

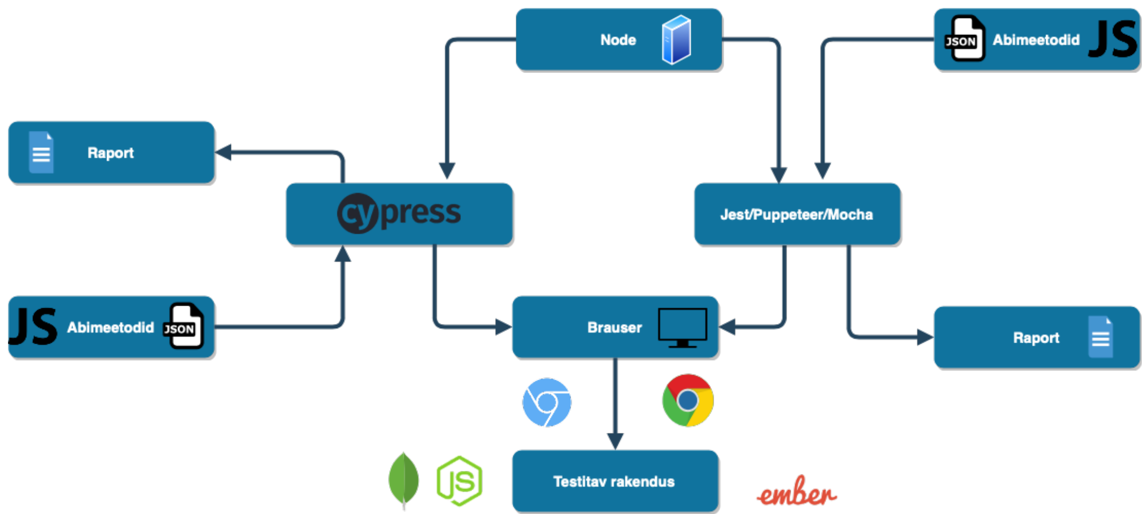


Joonis 16. Cypress ja Puppeteer testide sooritusaja võrdlus

Cypress raamistikku kasutades kulus testide läbimiseks keskmiselt 50.66 sekundit, Puppeteer/Jest teekide kasutamisel kulus 55.08 sekundit. Mõlemas süsteemis oli testide jooksutamine asünkroonne, mis kajastub sooritusaja väheses erinevuses. Töö autori valik kasutada Cypress raamistikku on ennast ära õigustanud - uue raamistikuga jooksevad samad testid keskmiselt 8.1% kiiremini, omades samaaegselt suuremat veebilehitsejate tuge.

5.5 Arhitektuuri võrdlus

Projekti ülevõtmisel Proekspert AS poolt teatas eelnev kliendi arendusfirma, et veebiprojektide automaattestimissüsteem on aegunud ning täpne informatsioon süsteemi kohta puudub. Automaatteste ei kasutatud ning enamik testid ebaõnnestusid. Testid ebaõnnestusid, sest testitava rakenduse kasutajaliides oli uuenenud ning testide ülesehitusel ei arvestatud süsteemi hallatavusega. Töö autor on loonud repositooriumi põhjal skeemi, milline võis olla automaattestimissüsteemi varasemalt.



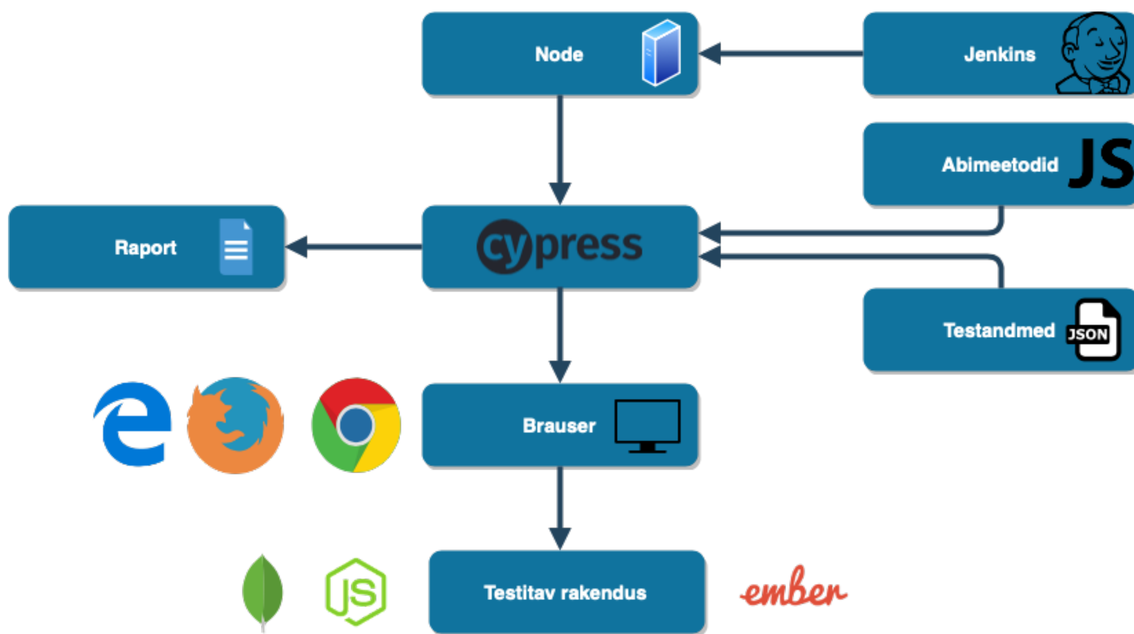
Joonis 17. Varasema süsteemi arhitektuur

Jooniselt 17 ilmnevad eelmise süsteemi puudujäägid – teste ei jooksutatud pideva integratsiooni tööriista kaudu ning kasutajaliidese testimiseks on kasutatud mitmeid erinevaid raamistikke, mis ei loo lisaväärtust. Samuti on kasutatud mõlema raamistiku jaoks eraldi testandmeid ja abimeetodeid. Selle tulemusena oli loodud süsteem ebapraktiliselt kompleksne järgmistel põhjustel:

- Testandmeid ja meetodeid tuli testide ebaõnnestumisel muuta mitmes kohas
- Raamistikud lõid mõlemad eraldiseisvad raportid

Süsteemi lihtsustamiseks viidi süsteem üle ühele raamistikule, mille käigus saavutati arusaadavam ning lihtsam arhitektuur. Selleks viidi kõik testid üle Cypress raamistikule, mille tulemusena ei olnud enam mitmeid testandmete ja abimeetodite hoidlaid. Raamistiku testtulemuste seadistamisel Cypress kontrolllauda saavutati lihtsam jälgitavus – kõik testijooksud on näidatud detailselt ühes keskkonnas.

Automaattestid jooksutatakse uues süsteemis läbi Jenkins keskkonna. Selle tulemusena on paranenud automaattestide ja testitulemuste kättesaadavus – varasemalt tuli testid jooksutada testija masinast, kuhu loodi ka testtulemuste raportid. Samuti on süsteemil suurem automatiseeritus – teste ei tule käivitada manuaalselt, tänu millele on testijal rohkem aega muudeks testimistegevusteks. Uue süsteemi arhitektuur on näidatud joonisel 18.



Joonis 18. Uue süsteemi arhitektuur

6 Edasised tegevused

Loodud automaattestimissüsteem on oluliselt vähendanud projekti ajakulu regressioonitestimisele. Sellest tulenevalt jätkab töö autor süsteemi edasiarendamist, mille järgmisteks etappideks on:

- Automaattestide paralleelne käivitamine
- Testitava eesrakenduse veebielementide identifikaatorite parandamine
- Kasutajaliidese automaattestide kaetavuse laiendamine
- Automaattestide jooksutamine Safari veebilehitsejaga (Esimesel võimalusel kui Cypress peaks välja arendama vastava funktsionaalsuse)

Lisaks projekti edasiarendamisele aitab autor kaasa tulevastes testimise automatiseerimise projektides ettevõttes Proekspert AS. Töö põhjal on autor omandanud teadmised automaattestimise erinevatest raamistikest ja teekidest ning oskab aidata arendustiime automatiseerimise tööriistade valimisel ja rakendamisel. Esimese sammuna teadmiste jagamisel tutvustab töö autor ettevõtte kvaliteediinseridele antud tööd ning raamistikku Cypress, mis ei ole senini tähelepanu saanud.

6.1 Automaattestide paralleelne käivitamine

Aja kokkuhoidmiseks on autoril tulevikus plaan lisada testide jooksutamine paralleelselt. Cypress raamistikuga on see täiesti võimalik, kuid soovitatav on kasutada selleks mitut erinevat masinat [24]. Hetkel on projektis kasutusel testide jooksutamiseks üks spetsiifiline masin, kus toimub lisaks mobiilirakenduste *build* etapp. Lisades masinale testide paralleelse käivitamise tekib oht, et masinal võivad tõrked tekkida ning testkomplektide läbimine või mobiilirakenduste *build* ebaõnnestub.

6.2 Veebielementide identifikaatorite parandamine

Testitava eesrakenduse suurimad probleemid testimise automatiseerimise vaatenurgast on:

- Veebielementide unikaalsete identifikaatorite osaline puudumine (*data-test* atribuudid)
- Olemasolevate identifikaatorite ebaühtlane stiil

Veebielementide unikaalsete identifikaatorite lisamise probleem on projektis juba tõstatatud, mille tulemusena lisavad arendajad identifikaatoreid uutele funktsionaalsustele. Vastavad atribuudid on osaliselt puudu ka hetkestel funktsionaalsustel, mille lisamisele aitab kaasa töö autor.

Olemasolevatel identifikaatoritel on ebaühtlane stiil – identifikaatori sõned on moodustatud kasutades *camel case*, *lowercase, hyphen-case* stiile segamini. Lahenduseks pakub töö autor välja liikuda ühtlasele stiilile, milleks on *hyphen-case* stiil. Enamik eesrakenduse atribuutidest kasutab just seda stiili, mis tõttu oleks arukas viia ka ülejäänud identifikaatorid samale kujule. Uute identifikaatorite lisamisel rakendatakse juba vastavat stiili.

6.3 Projekti mõju edasistele projektidele

Tehtud töö põhjal pakuti autorile võimalust tutvustada automaattestimissüsteemi ettevõtte kvaliteediinseridele. Kvaliteediinseneride koosoleku raames seletab autor lahti töös käsitletud automatiseerimise protsessi põhietapid ning toob näiteid loodud süsteemist. Koosoleku eesmärgiks on laiendada teadmisi seoses testimise automatiseerimisega, et rakendada samu tehnikaid teistes projektides.

Töötoa raames tutvustab töö autor testimisraamistikku Cypress, mille põhjal sooritatakse lihtsamad kasutajaliidese testid. Töötoa eesmärgiks on anda ülevaade raamistiku võimekustest ja näidata ette põhilised funktsionaalsused, et kvaliteediinsenerid kaaluksid raamistikku võimaliku tööriistana tulevikus. Samuti tuuakse välja raamistiku kitsaskohad, et mitte tekitada kallutatud arvamust tööriista suhtes.

7 Kokkuvõte

Käesolevas töös käsitleti probleeme veebirakenduste regressioonitestimise automatiseerimisel. Töö keskendus kasutajaliidese regressioonitestimise automatiseerimisele, mille käigus loodi ettevõtte Proekspert AS projektile automaattestimissüsteem. Töö eesmärk oli luua lahendus, mis oleks pilootprojektiks teistele automaattestimissüsteemidele.

Metoodikas määrati automaattestimise kaetavuse ulatus ja võrreldi erinevaid testimisraamistikke. Võrdluse käigus võrreldi raamistike veebilehitsejate ja tehnoloogia tugesid ning raamistike võimekusi raporteerimisel ja valideerimisel. Võrdluse tulemusena valiti välja raamistik Cypress, mis täitis kõik vajaoleva süsteemi nõuded.

Tulemusena loodi vana automaattestimissüsteemi asemel uus süsteem, mis integreeriti pideva integratsiooni tööriistaga Jenkins. Süsteem võimaldab jooksutada kasutajaliidese teste erinevatest masinatest mitmete veebilehitsejatega, mille tulemusena raporteeritakse tulemused Cypress raamistiku juhtlauda. Testid jooksutatakse perioodiliselt ning protsessi on võimalik näha läbi graafilise juhtlaua kõigil projekti huviisikutel.

Töö tulemusena vähenes manuaalse regressioonitestimise osakaal projektis ja paranes uue automaattestimissüsteemi võimekus eelmise süsteemiga võrreldes. Oluliselt vähenes uue süsteemi komplekssus, sest süsteemis kasutatakse nüüd mitme raamistiku asemel ühte.

Töö põhjal aitab autor edasite projektide regressioonitestimise automatiseerimisel. Selleks viiakse läbi töötubasid ja koosolekuid, kus tutvustatakse ettevõtte kvaliteediinseneridele läbitud protsessi ning sooritatakse lihtsamaid automaatteste Cypress raamistikuga, tutvustades nii raamistiku võimekusi ja kitsaskohti.

Kasutatud kirjandus

1. International Software Testing Qualifications Board. Certified Tester: Foundation Level Syllabus. [WWW] https://www.istqb.org/downloads/send/2-foundation-level-documents/281-istqb-ctfl-syllabus-2018-v3-1.html (25.03.2020)
2. Crispin, L., Gregory, J. Agile Testing: A Practigal Guide for Testers and Agile Teams. Addison-Wesley, 2009.
3. Falk, J., Kaner, C., Nguyen H. Q. Testing Computer Software. Wiley Computer Publishing, 1999.
4. Dustin, E., Paul, J. , Rashka, J. Automated Software Testing. Addison-Wesley, 1999.
5. International Software Testing Qualifications Board. Certified Tester: Test Automation Engineer. [WWW] https://www.istqb.org/downloads/send/48-advanced-level-test-automation-engineer-documents/201-advanced-test-automation-engineer-syllabus-ga-2016.html (26.03.2020)
6. Arya, S. Developing a Testing Strategy With the Automation Testing Life Cycle. [WWW] https://dzone.com/articles/all-you-need-to-know-about-automation-testing-life (29.03.2020)
7. Smartbear. Automated Testing Best Practices and Tips. [WWW] https://smartbear.com/learn/automated-testing/best-practices-for-automation (02.04.2020)
8. Turner, C. Test Your DOM with Data Attributes. [WWW] https://medium.com/@colecodes/test-your-dom-with-data-attributes-44fccc43ed4b (03.04.2020)
9. Smartbear. What Is Parallel Testing? [WWW] https://help.crossbrowsertesting.com/selenium-testing/getting-started/what-is-parallel-testing/ (04.04.2020)

10. Rehkopf, M. What Is Continuous Integration. [WWW] https://www.atlassian.com/continuous-delivery/continuous-integration (18.04.2020)
11. Kaner, C. Improving the Maintainability of Automated Test Suites. [WWW] http://www.kaner.com/pdfs/autosqa.pdf (19.04.2020)
12. Nightwatch.js. [WWW] https://nightwatchjs.org (20.04.2020)
13. Robot Framework Examples. [WWW] https://robotframework.org/#examples (20.04.2020)
14. Puppeteer. [WWW] https://github.com/puppeteer/puppeteer (20.04.2020)
15. Cypress in a nutshell. [WWW] https://docs.cypress.io/guides/overview/why-cypress.html#In-a-nutshell (20.04.2020)
16. Selenide and Selenium Comparison. [WWW] https://selenide.org/documentation/selenide-vs-selenium.html (20.04.2020)
17. How Cypress Works. [WWW] https://www.cypress.io/how-it-works (20.04.2020)
18. Faker.js. [WWW] https://github.com/marak/Faker.js/ (21.04.2020)
19. Writing and Organizing Tests. [WWW] https://docs.cypress.io/guides/core-concepts/writing-and-organizing-tests.html#Folder-Structure (21.04.2020)
20. Stop Using Page Objects and Start Using App Actions. [WWW] https://www.cypress.io/blog/2019/01/03/stop-using-page-objects-and-start-using-app-actions/#page-objects-problems (22.04.2020)
21. Best Practises. [WWW] https://docs.cypress.io/guides/references/best-practices.html#Using-after-or-afterEach-hooks (22.04.2020)
22. Cypress Projects. [WWW] https://docs.cypress.io/guides/dashboard/projects.html#Setup (23.04.2020)
23. Launching Browsers. [WWW] https://docs.cypress.io/guides/guides/launching-browsers.html#Browsers (01.05.2020)
24. Parallelization. [WWW] https://docs.cypress.io/guides/guides/parallelization.html#Overview (01.05.2020)

Lisa 1 – Github repositooriumi link

https://github.com/robert123211/final-thesis/