TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

Erko Teor 135066IAPB

REACT NATIVE MOBIILIRAKENDUSTE KASUTAJALIIDESTE AUTOMAATTESTIMINE – PÕHIMÕTTED NING JUURUTAMINE

Bakalaureusetöö

Juhendaja: Inna Švartsman
MSc

Tallinn 2019

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: Erko Teor

20.05.2019

Annotatsioon

Agiilsed arendusmeetodid on saanud tänapäeva tarkvara arendamise lahutamatuks osadeks ja sellest tulenevalt jäävad vanad testimispraktikad ajale jalgu. Traditsiooniline manuaalne testimine ei ole enam mõistliku aja ning ressursiga tulema kaasa agiilsete tarkvara arenduse tiimide koodi pideva uuenemisega ja sellest tulenevalt on mõistlik võtta kasutusele automaatika.

Antud lõputöö eesmärk on seadistada Detox testimise raamistik ning luua kasutajaliidese testid React Natives arendatavale Minu Telia mobiilirakendusele ning need käivitada pideva integratsiooni keskkonnas Bamboos koodimuudatuste üleslaadimisel Bitbucketi keskkonda.

Lõputöös kirjeldatakse ära testimispüramiid, mis annab ülevaate testide loomise ning võimaliku automatiseerimise loogikast. Automaatsete kasutajaliidese testide loomisel tuleks arvestada mõningate heade tavade ning praktikatega, millest tuleb antud lõputöös juttu.

Kogu süsteem nõuab omakorda sobivat süsteemi, milles joosta. Antud süsteem jookseb Bamboo keskkonnas, mis on järjepideva integratsiooni keskkond, mida kasutatakse tarkvara testtsüklite käivitamiseks ning testraportite loomiseks.

Lõputöö on kirjutatud Eesti keeles ning sisaldab teksti 31 leheküljel, 5 peatükki, 18 joonist, 2 tabelit.

Abstract

Automated testing for React Native mobile application user interfaces - principles and implementation

Agile development methodologies have become integral part of software development processes and therefore old testing practices are not able to cope with the frequent changes that come with it. Traditional manual testing is not capable of coping with agile software teams and their continuous and frequent change of code in reasonable time and effort and therefore it is reasonable to automate the process.

The goal of this research paper is to setup Detox testing framework and create automated user interface tests for Minu Telia mobile application developed in React Native and run them in Bamboo continuous integration environment after uploading code to Bitbucket code management environment.

In this research paper the testing pyramid is used to illustrate the possible creation and automation logic of creating automated user interface tests. It is reasonable to take into account some of the good practices and processes when creating automated user interface tests and this research paper will give some insight into that matter.

The entire system will requite a suitable host system to run in. The given system will be run in Bamboo environment, which is a continuous integration environment, which is used to run software test cycles and to create test raports.

The thesis is in Estonian and contains 31 pages of text, 5 chapters, 18 figures, 2 tables.

Lühendite ja mõistete sõnastik

Agiilne (väle) tarkvaraarendus	Tarkvara arendamise metoodika, kus tarkvarale seatud nõuded ning lahendused arenevad iseorganiseeruvate ja multifunktsionaalsete tiimide kollektiivse panuse ning nende klientide koostööl.
Kernel	Arvutiprogramm, mis on arvutisüsteemi operatsioonisüsteemi tuum, millel on täielik kontroll kogu süsteemi funktsionaalsuse üle.
End-to-end	Testimismetoloogia, mida kasutatakse selleks, et kontrollida rakenduse flow toimimmist algusest lõpuni sellisena, nagu ta on disainitud.
XML (Extensible Markup Language)	Märgistuskeel dokumendi struktuuri loomiseks. Struktuur pannakse paika paarissiltide abil, mida kasutaja saab ise luua. Siltide loomine sarnaneb HTML'ile, mis näeb vaeva andmete välise kuju eest.
Gray box testimine	Gray box testimine on tarkvara debugimise strateegia, kus testijal on testitava rakenduse kohta piiratud teadmine.
API	API ehk rakendusliides on arvutiprogrammides alamprogrammi määratluste, protokollide ja tööriistade komplekt rakendustarkvara ehitamiseks.
webview	Brauser, mis on paigutatud mobiilirakenduse sisse ning mis võimaldab kuvada veebilehe sisu mobiilirakenduses
Happy path	Tarkvara arenduse või informatsiooni mudeldamise kontekstis on happy path üks vaikimisi stsenaariumitest, mis ei sisalda endast mitte ühtegi veatingimust/-olukorda
Z-index	Omadus, mis kirjeldab ära objekti järjestuse horisontaalsel tasandil.
Teek	Funktsioonide, makrode (alamprogrammide), klasside, moodulite või muude sarnaste komponentide kogu, mida saab programmis vajadust mööda kasutada.
Breakpoint	Tahtlik programmi töö katkestamise või seiskamise koht, mille eesmärk on aidata arendajat programmi vea otsimisel.
Driver	Programm, mis kontrollib füüsilist seadet.
Endpoint	Seade (arvuti, server), mis suhtleb edasi-tagasi võrguga, millega ta on ühendatud.

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

Agiilses tarkvaraarenduses käib koodibaasi uute muudatuste, paranduste lisamine kiirelt, kuna suures plaanis muudetakse väikeseid tarkvara funktsionaalsuseid ning integreeritakse muudatused juba olemasolevasse süsteemi. Kuidas aga veenduda selles, et lisatud muudatus, parandus ei teinud midagi süsteemis kui tervikus katki?

Suurte koodibaasidega suurtes ettevõtetes ei tundu kuidagi mõistlik hakata pärast iga muudatust süsteemi toimimise kontrollimiseks tervet süsteemi manuaalselt üle testima. Selline lähenemine on liiga aeglane, vigadele vastuvõtlik ning lõpptulemusena kuluks väga suur osa ressursist testimise peale ja sellest tulenevalt ei jõuakski keegi enam arendada. Tarkvaratööstuse õnneks on aga loodud automaattestid, mis jooksevad eraldiseisvalt koos uue versiooni lisamisega ja kontrollivad juba olemasoleva funktsionaalsuse toimimist.

Käesoleva bakalaureusetöö eesmärk on anda ülevaade ühest mobiilirakenduse testimise raamistikust ning mobiilirakenduste testimise eripäradest ja lõpptulemusena kirjutada kasutajaliidese automaattestid React Native's kirjutatud Minu Telia mobiilirakendusele.

Bakalaureusetöö sisuline pool on jaotatud kolmeks osaks. Teises peatükis antakse ülevaade React Native'st ning Detoxist. Kolmandas peatükis antakse ülevaade testimisest ning testimise rollist ja headest tavadest tarkvaraarenduse protsessis ja tutvustatakse erinevaid testimistasemeid kirjeldavat testimispüramiidi. Neljandas peatükis toimub praktiline osa ehk rakenduse seadistamine ning seejärel testimine Detox raamistikuga.

Lõputöö kokkuvõttes kirjeldatakse töö käigus õpitu, millele tuginedes saab hiljem luua paremini töötavaid süsteeme ja teste.

1.1 Väle arendusprotsess ja testimine

Käesolevas töös keskendutakse testimisele väledas arendusprotsessis. Väle arendusprotsess tähendab seda, et süsteemi kallal töötavad mitmed erinevad arendajad üheaegselt ning muudatusi, parandusi lisatakse pidevalt ja kiiresti.

Väledas arendusprotsessis ei ole testimine omaette faas, vaid sellega tegeletakse pidevalt. Tihtipeale ei ole ka klassikalistes väledates arendusmeeskondades eraldi testijaid vaid rakenduse testimisega tegeleb terve meeskond. Rakenduse kvaliteediga pidev tegelemine välistab olukorra, kus vead kuhjuvad ning tuleks hakata tegelema nende prioritiseerimisega ja suure hulga vigade parandamisega.

Samuti ei teki olukorda, kus vigade algpõhjuse leidmine võtab palju aega, sest automaatne testimine annab koheselt märku vea põhjustanud koodimuudatusest.

1.2 Mobiilirakendused

Maailm on muutunud väga mobiilseks. Kõikvõimalikke seadmeid on võimalik mugavalt ning lihtsa vaevaga kaasas kanda. Sellest tulenevalt on saanud ka väga populaarseks kõikvõimalike rakenduste loomine nendele seadmetele. Antud lõputöö kontekstis võtame uurimise all olevateks objektideks mobiiltelefonid ja nendele rakenduste arendamise ja testimise arendamise eripärad.

Mobiilidele mõeldud rakenduste käivitamiseks on laias laastus kolm võimalikku varianti - füüsilised seadmed ning arvutis jooksutatavad emulaatorid ning simulaatorid. Kuna erinevate seadistuste, parameetrite ning riistvaraliste eripäradega füüsilisi seadmeid on maailmas väga palju, siis on reaalse tarkvara arendamise ning testimise seisukoha pealt mõistlik kasutada eelmainitud arvutis jooksutatavaid keskkondasid. Emulaatorid ning simulaatorid võimaldavad arvutis jäljendada reaalsete seadmete käitumist ja kontrollida suures osas seda, kas rakendus toimib ootuspäraselt ka reaalses seadmes.

2 Tehnoloogiate ülevaade

Käesolevas peatükis antakse lugejale kiire ülevaade arendamiseks kasutatavast React Native raamistikust ning testimise eesmärgil projekti juurutatavast Detox raamistikust.

2.1 React Native

React Native[1] on Javascripti raamistik, mis on loodud native rakenduste arendamiseks paralleelselt iOSile ning Androidile. React Native baseerub Reactil, mis on Facebooki poolt loodud Javascripti teek kasutajateliideste kiireks ning efektiivseks arendamiseks. Peamine erinevus seisneb selles, et React on loodud brauseris kasutatavate rakenduste jaoks, aga React Native on mõeldud mobiilsete platvormide jaoks. Küll aga on suur osa kirjutatud koodi jagatav mõlema platvormi vahel ning sellest tulenevalt on lihtne arendada ühel ajal Androidile ning iOSile.

Sarnaselt veebi jaoks loodud Reactile kirjutatakse React Native rakendused Javascripti ning XML-iliku markup keele seguna, mida tuntakse JSX nime all. Taustal kutsutakse välja native renderdamise APIsid Objective-C-s iOSi jaoks ning Javas Androidi jaoks. Selle tulemusena renderdab rakendus päris mobilli kasutajaliidese komponente, mitte webviewsid ja näeb välja ning käitub nagu iga teine mobiilirakendus. Täiendavalt võimaldab React Native kasutada platvormi põhiseid rakendusliideseid ja selle tulemusena saavad React Native rakendused kasutada näiteks ka kaamerat või kasutaja asukohta. Hetkel toetab React Native nii Androidi kui iOSi. React Native renderdab kasutades host platvormi standardseid renderdamise APIsid ja see võimaldab sellel teistest olemasolevatest platvormi ülestest raamistikest välja paista. Seda nimelt põhjusel, et olemasolevad raamistikud mobiilsete rakenduste kirjutamiseks kasutades Javascripti, HTMLi ja CSSi renderdatakse kasutades webviewsid. Tavaliselt kaasnevad sellega puudused, eriti jõudluse vaatest. React Native aga transleerib JSXis kirjutatud

1 React Native, https://facebook.github.io/react-native/ (20.05.2019)

markupi reaalseteks Native UI elementideks, mis võimaldavad võimendada olemasolevaid renderdamise võimalusi sõltumata kasutatavast platvormist.

React toimib eraldi peamisest kasutajaliidese lõimest ning selle tulemusena säilitab rakendus kõrge jõudluse ilma võimekust ohverdamata. React Native uuendustsükkel on sarnane Reactile - kui rakenduse muutujad (props) või rakenduse olek (state) muutub, siis React Native renderdab vaated uuesti. Peamine vahe võrreldes Reactiga on see, et React Native teeb seda täiendades kasutajaliidese teeke selle asemel et kasutada HTMLi ja CSSi markup segu. React Native lisab tavapärasele mobiilirakenduste arendamisele veel täiendusi kahes vallas: arendaja kogemus ning platvormide ülene arendamise potentsiaal.
Kuna React Native on reaalsuses lihtsalt Javascript, tähendab see seda, et muudatuste nägemiseks ei ole vaja kogu rakendust uuesti ehitada vaid piisab rakenduse värskendamisest emulaatoris või simulaatoris ning saab näha tulemust sarnaselt veebiarenduses toimuvale. See aitab arendusprotsessis hoida kokku suure osa ajast, sest ei ole vaja pidevalt oodata minuteid rakenduse uuesti nullist ehitamist vaid piisab mõnest hetkest.

Täiendavalt võimaldab React Native ära kasutada intelligentseid debugimise tööriistu ning vigade logimist. Näitena saab tuua erinevate brauserite (Chrome, Safari, Firefox) debugimise tööriistad, mida saab kasutada mobiilirakenduste arendamisel.
Sarnaselt on Javascripti muutmiseks võimalik kasutada ükskõik millist tekstiredaktorit kuna React Native ei eelda toimimiseks kindlat keskkonda nagu xCode, et arendada iOSi peale või Android Studiot, et arendada Androidi platvormi peale.
React Native võimaldab vähendada kasutatavate ressursside hulka, mis on vajalikud mobiilirakenduste arendamiseks. Seda nimelt põhjusel, et suurt osa koodibaasist saab taaskasutada. Muidugi ei ole kõike võimalik taaskasutada ning on juhte, kus on vaja arendada platvormi spetsiifilisi asju. Näitena saab välja tuua disainimise koha pealt z-indexi toimimise Androidide peal ja nende mittetoimimise iOSi peal.[2]

2 Bonnie Eisenman, “What is React Native?", https://www.oreilly.com/library/view/learning-react-native/9781491929049/ch01.html (20.05.2019)

2.2 Detox

Detox[3] on gray box end-to-end testimise ning automatiseerimise teek mobiilsete rakenduste jaoks.

Suurel kiirusel toimuv mobiilirakenduse arendamine nõuab arendajatelt pideva integratsiooni töövoolude kasutuselevõtmist, mis tähendab seda, et sõltuvus manuaalsest kvaliteedi tagamisest peab märkimisväärselt vähenema. Detox testib rakendust siis, kui see jookseb reaalses seadmes, simulaatoris ning käitub sellega justkui päris kasutaja seda teeks. Kõige keerulisem osa automatiseeritud testimisest mobiilides on järgnevas peatükis kirjeldatud testimispüramiidi tipp - end-to-end. End-to-end testide tuumprobleemiks on nende ebakindlus - testid ei ole deterministliku iseloomuga ja võivad minna teadmata põhjustel katki. Seda silmas pidades on Detox võtnud omaks gray box lähenemise.

Detoxi iseloomustavad:

1. Platvormiülesus: saab kirjutada platvormist sõltumatuid JavaScripti teste. Hetkel toetab iOSi ning Androidi.

2. Toimib erinevates seadmetes: annab kindluse, et test käitub rakendusega just nagu päris kasutaja.

3. Automaatne sünkronisatsioon: vähendab olulisel määral testide ebakorrapärast käitumist ja nende ebaõnnestumist jälgides rakenduses toimivaid asünkroonseid operatsioone ning tegutsedes vastavalt saadud tagasisidele.

4. Jätkuva integratsiooni tugi: võimaldab end-to-end teste jooksutada eraldiseisvatel platvormidel nagu näiteks Travis.

5. Ei ole testi käivitajast sõltuv: kasutada võib Mochat, AVAt, või ükskõik millist meelepärast JavaScripti testide käivitajat.

6. Debugitav: modernne asünkroonne-oota (async-await) API võimaldab asünkroonsetes testides breakpointidel töötada ootuspäraselt.

3 Detox, https://github.com/wix/Detox (20.05.2019)

Detox toimib gray box testimisstrateegial, mitte black box põhimõttel ning see võimaldab testraamistikul monitoorida rakendust seespoolt ning reaalselt hoida teste rakendusega sünkroonis. Gray box oma loomuselt kasutab koodi, mis on juba rakendusse paigaldatud ja sellest tulenevalt võimaldab näha ja jälgida rakenduse sees toimuvat.

Erinevalt black box testimisest jookseb gray box samas protsessis, omab ligipääsu mälule ning suudab monitoorida rakenduse käivitamise ning käimise protsesse. Mälu lugemise võimekus annab võimaluse näha ning tuvastada, mis toimub rakenduse protsesside sees. Näiteks, kas parajasti on võrgupäringuid käimas, kas pealõim on ooterežiimis, teised lõimed on ooterežiimis, animatsioonid on lõpetanud oma töö või sild, mis ühendab React Nativet on ooterežiimis. Sellest tulenevalt saab ta joosta pealõimel, et kindlustada see, et kui Detox teeb mingeid tegevusi, siis ei juhtu samaaegselt midagi kasutajaliidese hierarhias.

Küll aga ei tule Detox päris ilma miinuspooleta - tavaliselt gray box testimise raamistikega testimisel tuleb arvestada sellega, et rakendus käib läbi pisut erinevad kompileerimise ning käivitumise protsessid, sest rakendus vajab täiendavat koodi, mis käivitatakse protsessi siseselt.

Detox kasutab sisemiselt EarlGreyd ja Espressot, mis on Google poolt arendatavad draiverid. EarlGrey on iOSi jaoks ning Espresso Androidi jaoks.Need raamistikud suudavad rakendusega sünkroniseerida, mis tähendab seda, et nad tegutsevad rakendusega ainult siis, kui rakendus on ooterežiimis. Sünkroniseerimise mehhanism, mida kasutadakse ka teistes gray box raamistikes toimib järgnevalt. Selle asemel et proovida tegevusi, muudatusi kasutajaliidese peal, proovivad sünkroniseerimise mehhanismid pärida sisemisi ressursse iga mõne millisekundi tagant või kuulata nende callbacke, mis ütleksid testimise mehhanismile, et nüüd on rakendus üle läinud ooterežiimi.Testid ei jätku enne, kui kõik päritavad ressursid annavad testidele rohelise tule ning kui rakendus on parajasti ooterežiimis, siis hakkavad kasutajaliidesega toimetama.[4]

4 Rotem Mizrachi-Meidan, „Detox: Gray Box End to End Testing Framework for Mobile Apps”, https://hackernoon.com/detox-gray-box-end-to-end-testing-framework-for-mobile-apps-196ccd9564ce (20.05.2019)

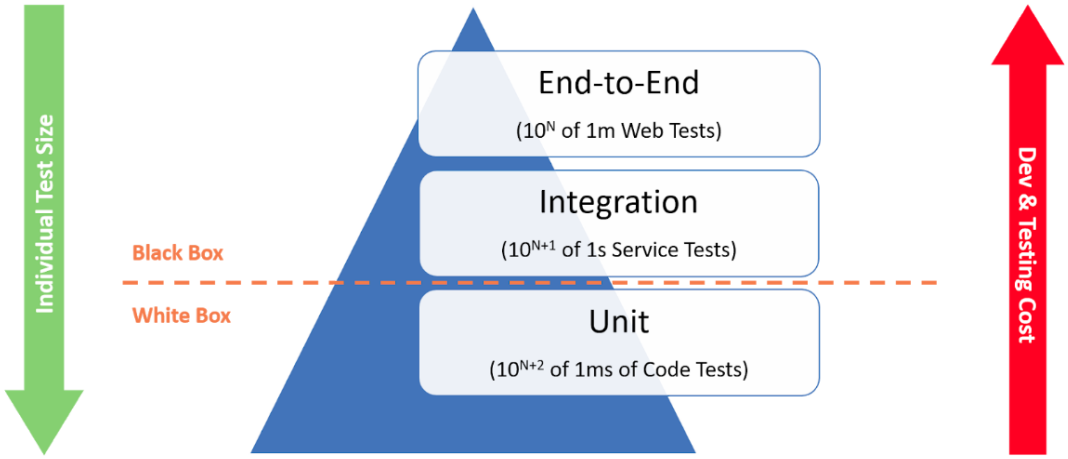
Täiendavalt ei sõltu Detox WebDriverist. Nimelt seetõttu, et tegemist ei ole veebiga. Detox suhtleb oma Native Driveriga, mis laiendab EarlGreyd ja Espressot kasutades JSONil põhinevat peegeldamise mehhanismi, mis võimaldab tavapärasel JavaScripti implemetatsioonil käivitada native meetodeid otse seadmes.

Detox on ehitatud native mobiilirakendust silmas pidades ning omab väga head tuge React Native rakendustele.

3 Testimine

Käesolevas peatükis kirjeldatakse testimispüramiidi, automaattestimise häid tavasid ning praktikaid ja kasutajaliideste automaattestide arendamise ning haldamise protsessi. Täiendavalt kirjeldatakse võimalike testandmete loomist.

3.1 Testimispüramiid



Joonis 1. Testimispüramiid

Testimispüramiid on tarkvara tööstuse standardiks saanud juhend funktsionaalseks testide arendamiseks.Testimispüramiid koosneb kolmest peamisest kihist - ühiktestid, integratsioonitestid /API/komponentide testid ning end-to-end testid.

Ühiktestid on testid, mis käivad otse tarkvara koodibaasi pihta ning kontrollivad koodi ühikuliste elementide korrektset toimimist. Tüüpiliselt kontrollitakse funktsioonide, meetodite ning klasside toimimist. Ühiktestide eripäraks võib veel välja tuua selle, et nad peaksid olema lühikesed, konkreetsed ning testima ühe konkreetse asja toimimist. Näiteks, et kas meetod tagastab etteantud tingimustel listi või sõne või mõne muu soovitud elemendi.

Integratsioonitestidega kaetakse ära kahe erineva tarkvara osa koostöö ning toimimine. Tihtipeale testitakse integratsioonitestidega kontroller-teenus-repositoorium mudelis repositooriumi toimimist. Näiteks, et reaalsest endpointist tuleksid kindlat tüüpi andmed tagasi ja nii edasi. End-to-end testidega kaetakse süsteemi süsteemi ja selle osade kui terviku toimimine. Kasutajaliidese testid on siinkohal heaks näiteks, sest nõuavad tihtipeale terve funktsionaalsuse/süsteemi olemasolu, et saada reaalne tulemus. Testid käituvad justkui päris kasutaja.

Testimispüramiid on kolmnurkse kujuga väga kindlal põhjusel - püramiidi alumises osas peaks olema rohkem teste ning püramiidi tipus vähem teste.

Seda nimelt neljal põhjusel:

1. Kaugus reaalsest koodist:

Ideaalses maailmas peaksid testid püüdma vead kinni nii lähedal juurpõhjusele, kui võimalik. Ühiktestid on esimeseks võrguks ning kaitseliiniks esmaste vigade avastamisel. Lihtsad asjad nagu vormingu vead, arvutuslikud vead või nullpointeri vead on ühiktestide käigus lihtsasti avastatavad, kuid palju keerulisemad avastada hilisemates faasides nagu näiteks integratsioonitestides või end-to-end testides.

2. Käivitamise aeg:

Ühiktestid on väga kiired, end-to-end testid on väga aeglased. Veebirakenduste puhul kehtiva “ühtede reegli” kohaselt kulub ühiktestile suurusjärk 1 millisekund, integratsioonitestile suurusjärk 1 sekund ning veebiliidese kasutajaliidese testile suurusjärk 1 minut. Kui testimise kogumitel on sadu kuni tuhandeid teste testimispüramiidi ülemises osas, siis tähendab see seda, et testidel võib minna tunde testi sooritamiseks. Järjepideva integratsiooni kontekstis on tunde kestvad testid vastuvõetamatud.

3. Arenduskulud:

Testimispüramiidi tipus olevaid teste on keerulisem kirjutada, sest nad katavad rohkem asju ning sellest tulenevalt on nad pikemad ning vajavad täiendavaid tööriistu ning

sõltuvusi, et oma tööd teha. Vastupidiselt on aga ühikteste võimalik kerge vaevaga muuta ning lisada, sest keskenduvad ühele väikesele osale süsteemist.

4. Töökindlus:

Püramiidi tipus olevad testid ei näe tihtipeale terves süsteemis toimuvat ja sellest tulenevalt võib tekkida ebakorrektsest sünkroniseerimisest erinevad juhud, kus tulemus sõltub sellest, millist funktsiooni esmalt välja kutsutakse ja sellest tulenevalt keskkonna võib esineda suuremat ebakindlust. Püramiidi alumises olevatel testidel aga on täpne ülevaade testitavast funktsionaalsusest ja on seega ka töökindlamad. [5]

3.2 Kasutajaliideste automaattestide kirjutamise head praktikad

Testimise eesmärk on kontrollida olemasoleva koodibaasi korrektset toimimist. Heade praktikate jälgimisel on testide lisamine, muutmine ning haldamine lihtsam ja ärilisest vaatest odavam.

Head testi iseloomustavad:

1. Loetavus:

Testi kavatsus peaks olema selge. Hea test ütleb täpselt ära, mida ta teeb ning sellest tulenevalt peaks olema lihtne mõista, millist stsenaariumi testitakse. Kui test ebaõnnestub, siis peaks olema võimalik kiiresti aru saada, kuidas probleemi peaks lahendama. Hea testiga on võimalik viga parandada ilma koodi debugimata.

2. Usaldusväärsus:

Test peaks ainult siis ebaõnnestuma, kui testitavas süsteemis on mingi viga. See tundub pealtnäha üsna ilmselge, aga arendajad sattuvad aeg-ajalt ikkagi olukorda, kus testid ebaõnnestuvad isegi siis kui mingeid uusi vigasid koodi ei lisatud.

Näiteks võivad testid õnnestuda kui neid käivitada ühekaupa, aga ebaõnnestuda kui käivitada terve testibaas. Sellised olukorrad viitavad olulisele veale disainis. Head testid

5 Automation Panda, „The Testing Pyramid", https://automationpanda.com/2018/08/01/the-testing-pyramid/ (20.05.2019)

peaksid olema korratavad ning eraldiseisvad välistest mõjutajatest nagu keskkond või käivitamise järjekord.

3. Kiirus

Teste kirjutatakse selleks, et neid oleks võimalik korduvalt käivitada ning kontrollida, et ühtegi viga ei ole koodi juurde lisatud.Kui testid on aeglased, siis on suurem tõenäosus, et testide käivitamine jäetakse vahele näiteks kohalikes masinates. Üks aeglane test ei oma suurt tähtsust, aga kui lisada veel sadu või tuhandeid aeglaseid teste, siis tuleb lõpuks testide käivitamist oodata väga pikalt.

4. Keskendub ühele asjale

Tähendab, et test teeb täpselt seda, mida test ütleb, et ta teeb ning ei toimu mingit peidetud kõrvaltegevust. Näiteks kui test ütleb, et loen andmed, siis ei toimu taustal tegelikult veel andmete kirjutamist. Või kui testitakse sisendivälja, siis ei testita sealsamas sisendivälja kõrval olevat nuppu.

5. Lihtsasti muudetav

Test peaks olema lihtsa vaevaga muudetav või ajakohastatav. Kergesti muudetav test on disainitud niimoodi, et näiteks sisendandmeid on võimalik välisest failist muuta selliselt, et test toimib jätkuvalt. Tulemusena saab kontrollida funktsionaalsuse toimimist hoopis teistsuguseid andmeid kasutades.[6]

3.3 Kasutajaliideste automaattestide arendamise ning haldamise protsess

Põhjalik testimine on tarkvara edu seisukoha pealt väga oluline. Kui tarkvara ei toimi korralikult, siis on hea tõenäosus ka sellel, et inimesed ei ole nõus antud tarkvara kasutama või ostma. Siinkohal aitab testimine ning just automatiseeritud testimine, mis käivitab detailseid, korduvaid ning andmeterohkeid teste automaatselt, et saada kiire ülevaade võimalikest probleemidest.

6 Sergey Kolodiy, „Unit Tests, How to Write Testable Code and Why it Matters", https://www.toptal.com/qa/how-to-write-testable-code-and-why-it-matters (20.05.2019)

Arendamisele ning haldamisele aitavad kaasa:

1. Otsustamine, milliseid testjuhte automatiseerida.

Kogu testimist on võimatu ja ka ebapraktiline automatiseerida ning sellest tulenevalt on oluline aru saada ning määrata, millised testjuhud tuleks esimesena automatiseerida. Automatiseeritud testimise kasu on tihedalt seotud sellega, kui mitu korda on võimalik testi mõistliku aja või ressursiga korduvalt käivitada. Teste, mida käivitatakse ainult mõned korrad on pigem mõistlikum jätta manuaalse testimise jaoks.Head testjuhud automatiseerimiseks on need, mida käivitatakse sageli ning mis nõuavad suurt hulka andmeid mingi soovitud tegevust sooritamiseks.

Kõige suurema kasu automaattestimisest on võimalik saada automatiseerides:

1. Korduvad testid, mis jooksevad mitmetes buildides

2. Testid, mis põhjustavad inimvigasid

3. Testid, mis nõuavad mitmeid andmeallikaid

4. Tihedasti kasutatud funktsionaalsuse, mis toob endaga kaasa suured riskid

5. Testid, mida on võimatu manuaalselt teha

6. Testid, mis jooksevad mitmel erineval riistvara või tarkvara platvormil või seadistusel.

7. Testid, mis nõuavad manuaalse testimise korral väga palju aega ning pingutust.[7]

Edu testide automatiseerimises nõuab hoolikat planeerimist ning disainimist. Alustada tuleks automatiseerimise plaani loomisega, see võimaldab tuvastada esialgse grupi testidest, mida automatiseerida ning mis omakorda on juhiseks edasistele testidele.

Esimesena tuleks seada automaattestimise eesmärgid ning otsustada, millist tüüpi teste automatiseerida. Kuna on olemas erinevat tüüpi testimisi ning igal ühel on oma koht testimisprotsessis, tuleks ka seda arvesse võtta. Näiteks ühiktestid, mida kasutatakse testitava rakenduse väikese osa testimiseks. Et testida aga kindlat osa rakenduse

7 Smartbear, „Automated Testing Best Practices and Tips", https://smartbear.com/learn/automated-testing/best-practices-for-automation/ (20.05.2019)

kasutajaliidesest, tuleks kasutada funktsionaalset või graafiliste kasutajaliideste testimist. Pärast eesmärgi seadmist ning otsustamist, et milliseid teste automatiseerida, tuleks otsustada, milliseid tegevusi automaattestid tegema hakkavad. Ei ole mõistlik kirjutada teste, mis testivad rakenduse erinevaid aspekte ühes testis. Suured, keerulised automaattestid ei ole kergesti hallatavad ega debugitavad. Kõige mõistlikum on jaotada testid mitmeks loogiliselt grupeeritud väiksemaks testiks. See muudab testkeskkonna kergemini mõistetavaks ning võimaldab jagada koodi, testandmeid ning protsesse. Kergem on uuendada ka oma automaatteste lisades juurde väikseid teste, mis tutvustavad uut funktsionaalsust. Funktsionaalsust on mõistlik testida nii nagu seda lisatakse, selle asemel, et oodata, et terve funktsionaalsus oleks valmis ehitatud. Testide loomisel on oluline meeles pidada, et testid tuleks hoida võimalikult väiksena ning keskendatuna ühele eesmärgile või ülesandele. Näiteks erinevad testid read-only funktsionaalsustele ja read/write testidele. See võimaldab neid teste kasutada korduvalt ilma, et oleks vaja nad lisada igasse automaattesti. Kui juba on tekkinud mitmed lihtsad automaattestid, siis saab hakata teste grupeerima ühte suuremasse automaattesti. Automaatteste on näiteks võimalik organiseerida rakenduse funktsionaalsest seisukohast lähtuvalt, ühiste funktsioonide põhjal või ühiste testandmete põhjal.

2. Varajane testimine ning tihe testimine

Et saada automatiseeritud testimisest maksimaalset kasu, tuleks alustada testimisega nii vara kui võimalik ning käivitada neid teste nii tihti kui võimalik. Mida varem testid kaasatakse projekti elutsüklisse, seda parem. Mida rohkem teste, seda rohkem vigasid on potentsiaalselt võimalik leida. Varajaselt avastatud vead on palju odavamad parandada, kui hiljem juba kasutajate poolt kasutatavas rakenduses avastatud vead.

3. Heade testandmete loomine

Head testandmed on väga kasulikud andmetest sõltuvatel või andmetest ajendatud testimisel. Andmed, mida sisestatakse sisendiväljadele on üldiselt paigutatud välisesse faili.Neid andmeid võib lugeda andmebaasist või mõnest teisest andmeallikast nagu teksti-, või XML või JSON failidest. Hea automaattestimise tööriist saab andmefailide sisust aru ning käib automaattesti käivitamise korral faili sisust üle.Kasutades väliseid andmeid saab muuta automaattestid taaskasutatavaks ning kergemini hallatavaks. Täiendavate stsenaariumite lisamiseks piisab sellest, et olemasolevaid andmefaile saab

kerge vaevaga laiendada ilma, et oleks vajadust muuta olemasolevat automaattesti. Automaattestidele testandmete loomine ei pruugi olla alati kõige põnevam tegevus, aga see on kindlasti väärt seda ajalist investeeringut ning pingutust, et luua hästi struktureeritud andmed, sest heade testandmete olemasolul muutub automaattestide kirjutamine palju lihtsamaks. Ehk siis mida varajasemas faasis luua hea kvaliteediga andmed, seda kergem on laiendada olemasolevaid automaatteste koos rakenduse arendamisega.

4. Automattestide loomine, mis on vähetundlikud muutustele kasutajaliideses.

Automaattestid, mis on loodud skriptidena või märksõna põhiste testidena on sõltuvad testitavast rakendusest. Rakenduse kasutajaliides võib erinevate arenduse versioonide vahel muutuda, eriti projekti varajastes staadiumites. Need muutused võivad mõjutada testide tulemusi ning sellest tulenevalt rakenduse tulevaste versioonidega enam mitte toimida. Probleem seisneb üldiselt selles, et automaattestide tööriistad kasutavad erinevaid koodis olevaid omadusi, et tuvastada otsitav objekt. Mõnikord sõltub testimise tööriist hoopis mingi testitava objekti asukohast. Näiteks, kui on kadunud otsitav omadus või on muudetud asukohta, siis test enam ei leia soovitud objekti ning ebaõnnestub. Automaattestide edukaks käivitamiseks võib olla sellest tulenevalt vajalik muuta vanad nimetused uuteks terves projektis, enne kui käivitatakse testid uue muudetud rakenduse vastu. Kui aga kohe alguses luua unikaalsed nimed kontrollelementidele, siis muudab see rakenduse vähem tundlikuks kasutajaliidese muudatustele ning tagab selle, et automaattestid toimivad edasi ilma testis muudatusi tegemata. See eemaldab ka testimistööriista vajaduse tugineda asukohal, mis on vähem stabiilne ja läheb kergesti katki.

3.4 Testandmed

Andmed, mida kasutatakse testimises, kirjeldavad ära esialgsed tingimused testile ning esindavad meediumi, läbi mille testija mõjutab tarkvara. Testandmed on tarkvarale ette antud sisendiks. See hõlmab endast andmeid, mis mõjutavad või on mõjutatud spetsiifilise mooduli käivitamisest. Mõningaid andmeid võidakse kasutada selleks, et

saada kinnitust sellele, et testitav funktsionaalsus tagastab oodatud tulemuse. Teisi andmeid võidakse kasutada selleks, et kontrollida programmi võimekust tulla toime niiöelda halbade juhtudega, kus näiteks süsteemi väline osa või mõni sõltuvus tagastab vigast sisendit ja sellest tulenevalt programm ilma nende vigu püüdvate osadete (failsafe) kokku jookseb.

Halvasti disainitud testandmed ei pruugi testida kõiki võimalikke testimise stsenaariume ning see võib suures plaanis mõjutada tarkvara kvaliteeti, sest koodiread on justkui ülevaate põhjal kaetud ja testid rohelised, aga tegelikult on andmeid, mille puhul süsteemi töö ebaõnnestub.

Olenevalt testimiskeskkonna iseloomust võib olla vajalik ise luua testitavaid andmeid, mida tuleb teha suuremal osal juhtudest või siis tuleb leida olemasoleva seast juba sobivad andmed ning need kasutusele võtta. Testandmeid on võimalik luua manuaalselt, koopiana rakenduse live andmetest testkeskkonda, koopiana "pärand" (legacy) klientsüsteemidest või automatiseeritud testiandmete loomise tööriistade abil. Üldiselt peaks olema testandmed olemas juba enne testimise alustamist, sest muidu on raske hiljem kirjutada õigeid teste.[8]

Testandmeid võiks valida pidades silmas järgnevaid asju:

1. Eelistatult soovitakse katta nii palju harusid kui võimalik ja sellest tulenevalt on mõistlik valida ja luua testandmed sellisena, et kõiki testitavaid harusid käsitletakse vähemalt korra.

2. Kaetakse ära erinevad programmi teekonnad - testandmete ettevalmistus peaks olema selline, et katta võimalikult palju kasutusjuhte.

3. Ebakorrektsete andmetega API testimine - testandmed võivad sisaldada ebakorrektseid parameetrite tüüpe, millega erinevaid meetodeid välja kutsutakse. Või siis võivad testandmed sisaldada väära kombinatsiooni argumentidest, millega programmi meetodeid välja kutsutakse.

Testandmed kiirusetundlikute testide tarbeks

8 Guru99, „Test Data Generation: What is, How to, Example, Tools", https://www.guru99.com/software-testing-test-data.html (20.05.2019)

Kiirusetundlikud testid on testid, mis määravad ära süsteemi kiiruse ning töövõime kindla töökoormuse all. Taoliste testide eesmärgiks ei ole leida vigu vaid avastada nii-öelda pudelikaelasid, mis süsteemi aeglustavad. Kiirusetundlike testide andmed peavad olema võimalikult lähedased "päris" andmetele. Kuna reaalsuses on palju andmeid, mis on tundlikud kliendiandmed, nagu näiteks krediitkaardi info, pangadetailid, siis on hea tava neid andmeid ka anonümiseerida.

Testandmed turvalisuse testide tarbeks

Turvalisuse testimine on protsess, mis aitab selgust saada selles, kas süsteem kaitseb süsteemis olevaid andmeid pahatahtlike tegevuste eest. Testandmed turvalisuse testide tarbeks peaksid katma ära järgnevad teemad:

1. Konfidentsiaalsus

Kogu klientide poolt tulev informatsioon peaks olema kõige rangemate meetoditega kaitstud ning neid andmeid ei tohiks kellegi kolmanda osapoolega jagada. Näitena võib tuua rakenduse, mis kasutab SSLi. Selle puhul saab disainida andmed, mis kontrollivad seda, et andmed enkrüpteeritakse korrektselt.

2. Andmete terviklikkus

Tuleks kindlaks teha, et süsteemi poolt tulevad andmed on õiged. Et luua sobivaid testandmeid, tuleks vaadata süsteemi disaini, koodi, andmebaase ning failistruktuure.

3. Autentimine

Autentimine on kasutaja identiteedi kindlaks tegemise protsess. Testandmed saab disainida erinevate kombinatsioonidena kasutajanimedest ja paroolidest ning selle kõige eesmärgiks on kontrollida, et ainult lubatud ning vastavaid õigusi omavad kasutajad saavad tarkvara süsteemile ligi.

4. Autoriseerimine

Määrab ära kasutajale antud ning kasutajale omased õigused. Testandmed võivad sisaldada erinevaid kombinatsioone kasutajatest, rollidest ja tegevustest eesmärgiga teha

kindlaks see, et ainult piisavate õigustega kasutajad saavad mingit konkreetset tegevust sooritada.

4 Praktika

Peatükis kirjeldatakse React Native's kirjutatud Minu Telia mobiilirakenduse kasutajaliidese erinevate osade automaattestimist ja uute koodimuudatuste lisamisel nende testide automaatset käivitamist Bamboos. Esmalt antakse ülevaade süsteemi üldisest struktuurist ning seejärel kirjeldatakse kasutatavaid sõltuvusi. Edasi liigutakse kasutajaliidese testide tasemele.

4.1 Süsteemi ülevaade

React Native's kirjutatud Minu Telia mobiilirakendusele kirjutatakse tavaliselt uue funktsionaalsuse lisamisel ka ühiktestid. Lõputöö kontekstis lisame täiendavalt kasutajaliidese testid. Testimine põhiline fookus on rakenduse enda kvaliteedil ning integratsioon teiste süsteemidega on pigem teisejärguline.

Ühiktestid kasutavad täielikult simuleeritud andmeid, kasutajaliidese testid kasutavad rakenduse praeguses valmisoleku faasis samuti simuleeritud andmeid.

Kasutajaliidese testid asuvad koos rakenduse koodiga ühes repositooriumis. See lihtsustab testide haldamist ja testide sünkroonis olemist arendusega, sest teste on võimalik sama koha pealt muuta ning hallata.

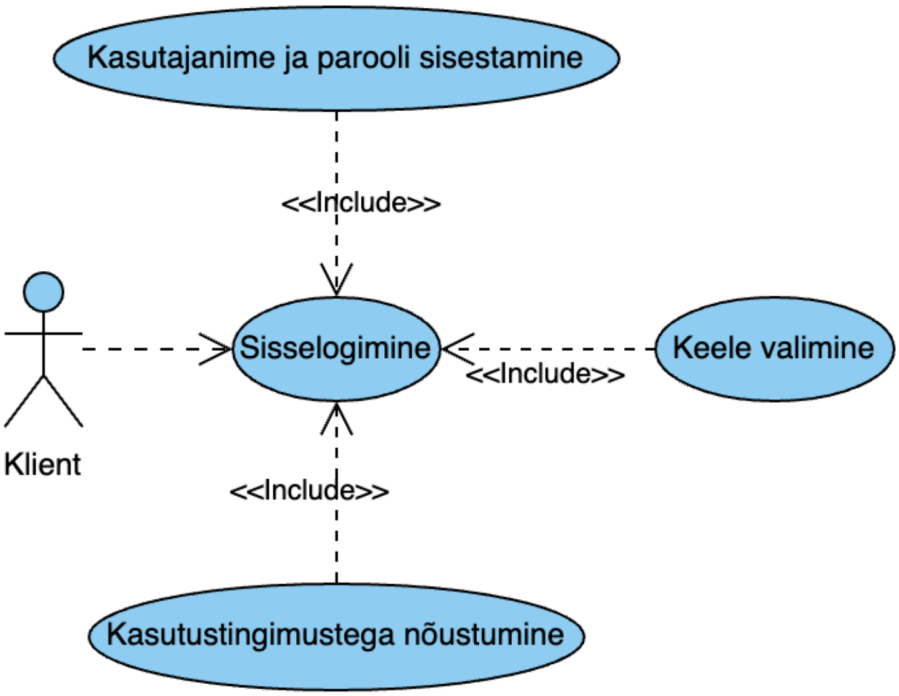
Praegune rakendus on alles arendusjärgus ning seega on võimalik testida piiratud osa süsteemi kui terviku funktsionaalsusest. Näiteks esmasel käivitamisel keele valimist, sisselogimist, menüüpunktide vahel liikumist, erinevatesse Telia esindustesse aegade reserveerimist ja kõige lõpuks veel rakenduse sisesele seadete menüüle omaseid elemente nagu keelevahetust, kasutajatingimuste vaatamist ja ka välja logimist.

4.2 Testimise näidisplaan

Testimise puhul on hea, kui on olemas plaan, mida jälgida ning kirjeldatud testjuhtumid, mida testida.

Minu Telia rakenduses olevate funktsionaalsuste näitena tuuakse siinkohal ära mõned võimalike testitavate kasutusjuhtude diagrammid ning nende võimalikud testjuhtumid:

1. Kasutajanime ja parooli sisestamise kasutusjuhu diagramm

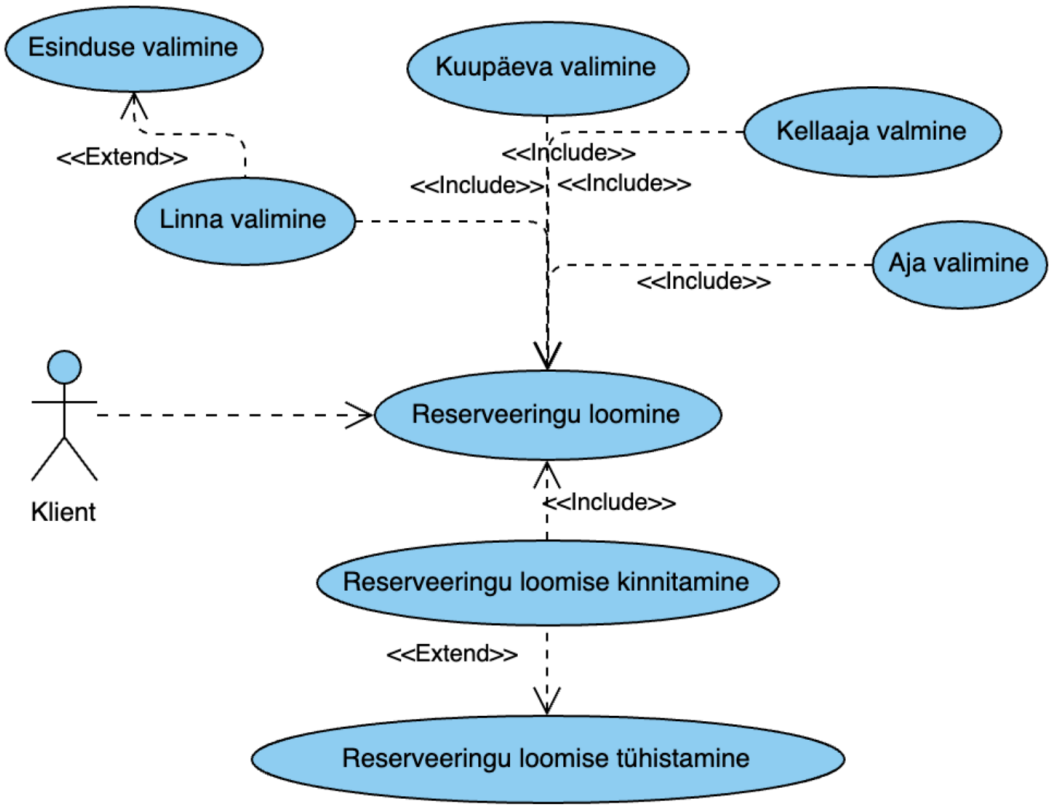


Joonis 2. Kasutajanime ja parooli sisestamise kasutusjuhu diagramm

Tabel 1. Kasutajanime ja parooli sisestamise testjuhtude kirjeldus

Number	Funktsionaalsed testjuhud	Testjuhtumi tüüp – negatiivne / positiivne
1	Kinnita, et kasutajale kuvatakse rakenduse esmakordsel käivitamisel keele valimise vaadet	Positiivne
2	Kinnita, et kasutaja saab õige kasutajanime ja parooliga sisse logida.	Positiivne
3	Kinnita, et kasutaja ei saa vale kasutajanime või parooliga sisse logida.	Negatiivne
4	Kinnita, et kasutaja parooli on peidetud tärnide taha.	Positiivne
5	Kinnita, et “logi sisse” nupu vajutamisel kasutaja logitakse rakendusse sisse.	Positiivne
6	Kinnita, et kasutale kuvatakse kasutustingimustega nõustumise vaadet esmakordsel sisselogimisel	Positiivne

2. Reserveeringu loomise protsessi kasutusjuhu diagramm. Antud juhul on eelduseks see, kui kõik õnnestub (ei teki päringutel vigu ega muud säärast). Ehk siis nii-öelda “happy path” kirjeldus süsteemi funktsionaalsuse osast.



Joonis 3. Reserveeringu loomise protsessi kasutusjuhu diagramm

Antud juhul saab ära määrata näiteks järgnevad testjuhud:

Tabel 2. Reserveeringu loomise protsessi testjuhtude kirjeldus

Number	Funktsionaalsed testjuhud	Testjuhtumi tüüp – negatiivne / positiivne
1	Kinnita, et kasutaja saab avada reserveeringu loomise vaate.	Positiivne
2	Kinnita, et kasutaja näeb linna valimise vaadet, kui valitakse reserveeringu loomise vaade.	Positiivne
3	Kinnita, et avatakse esinduse valimise vaade, kui antud linnas on rohkem kui üks esindus.	Negatiivne
4	Kinnita, et linna valimise vaatest liigutakse linna valimisel edasi kuupäeva valimise vaatesse.	Positiivne
5	Kinnita, et esinduse valimise vaatest liigutakse esinduse valimisel edasi kuupäeva valimise vaatesse.	Positiivne
6	Kinnita, et kuupäeva valimise vaatest liigutakse kuupäeva valimisel edasi kellaja valimise vaatesse.	Positiivne
7	Kinnita, et kellaaja valimise vaatest liigutakse kellaaja valimisel edasi reserveeringu kinnitamise vaatesse.	Positiivne
8	Kinnita, et reserveeringu kinnitamise vaatest liigutakse reserveeringu kinnitamisel edasi dealeri vaatesse.	Positiivne
9	Kinnita, et reserveeringu kinnitamisel on kasutajal olemas reserveering.	Positiivne
10	Kinnita, et reserveeringu kinnitamise vaatest liigutakse reserveeringu tühistamisel edasi dealeri vaatesse.	Positiivne
11	Kinnita, et reserveeringu loomise tühistamist saab katkestada.	Positiivne
12	Kinnita, et reserveeringu loomise tühistamisel ei ole kasutajal ühtegi reserveeringut.	Positiivne

4.3 Raamistiku poolt pakutavad võimalused testide kirjutamisel

Kasutatav testimise raamistik pakub rakenduse kasutajaliidese automaatseks testimiseks mitmeid võimalusi. Peamised teststsenaariumid näevad ette mobiilirakenduse erinevate vaadete vahel liikumist ning lehtedel asuvate elementide olemasolu kontrollimist. Raamistik võimaldab vaatel asuvate elementidega manipuleerimist, näiteks tekstiväljadele kirjutamist, raadionuppude seadmist, „linnukeste“ (checkbox) seadmist, nuppude vajutamist, nimekirjadest (select list) valikute tegemist jne. [9]

Teste alustatakse tavaliselt eelmise testi järelt „koristamisega“ (näiteks, kui kasutaja ei ole rakendusest välja loginud, peaks seda tegema). Seejärel liigutakse testitavale vaatele ning kontrollitakse lehe teatud sisuelementide olemasolu. Kui element on olemas, siis vaate funktsionaalsus võib eeldada ka sellega manipuleerimist.[10]

Näiteks tekstivälju peab saama täita ning nuppudele vajutamine peab viima järgmisele (alam)vaatele. Testid on tüüpiliselt ehitatud üles kasutajalugude järgi: näiteks peab klient saama luua reserveeringut, tellida lisamahtu, maksta arveid jne.

4.4 Detoxi seadistamise näide iOSi jaoks

Antud juhul on käskude täpsem kirjeldus leitav Detoxi kodulehel. Järgnevalt kirjeldatakse lõputöös vajalikud käsud iOSi jaoks ning tuuakse need joonistena välja. Täiendavalt on lisatud mõne käsu poolt tekitatud failid ning nende täiendav seadistus.

Detoxi seadistus kohalikus masinas iOSi jaoks macOSi peal: [11]

1. Tuleb installida Homebrew kasutades terminalis käsku:

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Joonis 4. Homebrew installimise käsk

9 Detox, API Reference, https://github.com/wix/Detox/blob/master/docs/README.md

10 Henri Perkmann, „Veebirakenduse kasutajaliidese automaatne testimine väleda arendusprotsessi kontekstis – põhimõtted ja implementatsioon” lk 25, https://dspace.ut.ee/bitstream/handle/10062/50453/Perkmann_BA2015.pdf?sequence=1&isAllowed=y

11 Detox, „Getting Started”, https://github.com/wix/Detox/blob/master/docs/Introduction.GettingStarted.md (20.05.2019)

2. Järgnevalt tuleb installida Node.js kasutades terminalis käsku:

```
brew update && brew install node
```

Joonis 5. Node.js installimise käsk

3. Järgnevalt tuleb installida applesimutils kasutades terminalis käske:

```
brew tap wix/brew && brew install applesimutils
```

Joonis 6. Applesimutils installeerimise käsud

4. Järgnevalt tuleb installida Detoxi käsurea tööriistad kasutades terminalis käsku:

```
npm install -g detox-cli
```

Joonis 7. Detox käsurea tööriistade paigaldamise käsk

5. Järgnevalt tuleb soovitud React Native projektis minna juurkausta ning käivitada käsk:

```
npm install detox --save-dev
```

Joonis 8. Developer sõltuvuste lisamise käsk

Pärast selle käsu käivitamist tuleb package.json faili lisada vastavalt projektile täiendus:

```
"detox": {
  "test-runner": "jest",
  "runner-config": "./e2e/config.json",
  "configurations": {
    "ios.sim.debug": {
      "binaryPath": "ios/build/Build/Products/Debug-iphonesimulator/myTelia.app",
      "build": "xcodebuild -project ios/myTelia.xcodeproj -UseNewBuildSystem=NO -scheme myTeliaDev -configuration Debug -sdk iphonesimulator -derivedDataPath ios/build",
      "type": "ios.simulator",
      "name": "iPhone 8"
    },
    "ios.sim.release": {
      "binaryPath": "ios/build/Build/Products/Release-iphonesimulator/myTelia.app",
      "build": "xcodebuild -project ios/myTelia.xcodeproj -UseNewBuildSystem=NO -scheme myTelia -configuration Release -sdk iphonesimulator -derivedDataPath ios/build",
      "type": "ios.simulator",
      "name": "iPhone 8"
    }
  }
}
```

Joonis 9. Package.json faili lisatav täiendus

6. Järgnevalt tuleb projekti juurkaustas käivitada sobiva testraamistikuga järgnev käsk (antud juhul kasutame Jesti):

```
detox init -r jest
```

Joonis 10. Detoxi testraamistiku initsialiseerimise käsk

See käsk seadistab projektis e2e kausta ning loob sinna kausta järgnevad failid:

init.js fail

```
const detox = require( id: 'detox');
const config = require( id: '../package.json').detox;
const adapter = require( id: 'detox/runners/jest/adapter');

jest.setTimeout(120000);
jasmine.getEnv().addReporter(adapter);

beforeAll(async () => {
  await detox.init(config);
});

beforeEach(async () => {
  await adapter.beforeEach();
});

afterAll(async () => {
  await adapter.afterAll();
  await detox.cleanup();
});
```

Joonis 11. init.js fail

config.json fail

```
{
    "setupTestFrameworkScriptFile": "./init.js",
    "testEnvironment": "node"
}
```

Joonis 12. config.json fail

7. Järgnevalt tuleb käivitada käsk rakenduse ehitamiseks:

```
detox build
```

Joonis 13. Detoxi rakenduse ehitamise käsk

8. Ning viimasena testide käivitamiseks:

```
detox test
```

Joonis 14. Detox testide käivitamise käsk

4.5 Näidistestid

1. näidistest kontrollib rakenduse esmasel käivitamisel kasutajale kuvatavate keelevalikute toimimist ning kontrollib seda, et keel valida, siis rakendus liigub edasi sisselogimise vaatesse.

```
describe('Test initial language selection and navigation from it', () => {

  beforeEach(async () => {
    await device.launchApp({delete: true});
  });

  it('initial select language screen should be visible', async () => {
    await expect(element(by.id('InitialLanguageSelectScreen'))).toBeVisible();
  });

  it('should move to sso screen on Estonian language select', async () => {
    await element(by.text('Eesti')).atIndex(1).tap();
    await expect(element(by.id('SsoScreen'))).toBeVisible();
  });

  it('should move to sso screen on English language select', async () => {
    await element(by.text('English')).atIndex(1).tap();
    await expect(element(by.id('SsoScreen'))).toBeVisible();
  });

  it('should move to sso screen on Russian language select', async () => {
    await element(by.text('Русский')).atIndex(1).tap();
    await expect(element(by.id('SsoScreen'))).toBeVisible();
  });
});
```

Joonis 15. 1. näidistest

1. näidistesti õnnestumine. Antud juhul kulus nii palju aega seetõttu, et rakenduse eripärade tõttu pidi testimine iga testi puhul alustama täiesti uue rakendusega.

```
PASS e2e/initialLoginScreenTest.spec.js (42.175s)
  Test Estonian language select button
    ✓ initial select language screen should be visible (6270ms)
    ✓ should move to sso screen on Estonian language select (8735ms)
    ✓ should move to sso screen on English language select (8894ms)
    ✓ should move to sso screen on Russian language select (8541ms)

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        42.224s
Ran all test suites matching /initialLoginScreenTest/i with tests matching "^((?!:android:).)*$".
```

Joonis 16. 1. näidistesti õnnestumine

2. näidistest kontrollib rakenduse liikumist läbi reservatsiooni loomise protsessi ning kontrollib täpsemalt seda, et rakendus jõuaks esmasest protsessi vaatest (DealerScreen) reservatsiooni kinnitamise vaateni (ConfirmReservationScreen).

```
it('should move through reservation creation flow', async () => {
  await expect(element(by.id('DealerScreen'))).toBeVisible();
  await element(by.id('CreateReservationButton')).atIndex(1).tap();
  await expect(element(by.id('SelectCityScreen'))).toBeVisible();
  await element(by.text('Tallinn')).atIndex(1).tap();
  await expect(element(by.id('SelectBranchScreen'))).toBeVisible();
  await element(by.id('SelectBranchFirstBranchButton')).atIndex(1).tap();
  await expect(element(by.id('SelectDateScreen'))).toBeVisible();
  await element(by.id('SelectDateFirstDateButton')).atIndex(1).tap();
  await expect(element(by.id('SelectTimeScreen'))).toBeVisible();
  await element(by.id('SelectTimeFirstTimeButton')).atIndex(1).tap();
  await expect(element(by.id('ConfirmReservationScreen'))).toBeVisible();
});
```

Joonis 17. 2. näidistest

4.6 Bamboo

Bamboo on pideva integratsiooni keskkond, mida saab kasutada tarkvara reliiside haldamise automatiseerimiseks.

Pidev integratsioon on tarkvara arendamise metoloogia, milles tarkvara ehitamine, ühiktestid ning integratsioonitestid käivitatakse iga kord kui uus kood repositooriumisse üles pannakse. See võimaldab leida esmased uute muudatustega seotud vead kiiresti. [12]

Antud töö kontekstis tuleb lisada Bamboo skriptile täiendus projekti koodis oleva Detoxi seadistuse[13] kasutamiseks testide jooksutamisel.

```
detox build --configuration ios.sim.release

detox test --configuration ios.sim.release --cleanup
```

Joonis 18. Bamboo skripti täiendus

12 Atlassian, „Understanding the Bamboo CI Server", https://confluence.atlassian.com/bamboo/understanding-the-bamboo-ci-server-289277285.html

13 Detox, „Step 2: Add build and test commands to your CI script", https://github.com/wix/Detox/blob/master/docs/Guide.RunningOnCI.md

5 Kokkuvõte

Bakalaureusetöös kirjeldatud Minu Telia mobiilirakenduse kasutajaliidese automaattestid ning nende käivitamine Bamboos aitab tagada järjepidevat rakenduse kvaliteeti ning vähendada olulisel määral manuaalse regressioonitestimise mahtu rakenduse pihta.

Järjepidev rakenduse kvaliteedi kontroll on rakenduse pideva arengu ja elutsükli raames väga oluline, sest aitab kokku hoida manuaalset ressurssi ning sellest tulenevalt kulusid. Programeerimisel tekkinud vead on võimalik kiiresti tuvastada ning vajalikud parandused saab kiiresti implementeerida.

Automaatne kasutajaliideste testimine küll aga ei välista täielikult arendajat testimise monitoorimise tsüklist, sest testimise käigus loodud raporteid on vaja jälgida ning analüüsida. Küll aga saab juba analüüsida üsna täpselt tuvastatavat viga ja sellest tulenevalt kiiresti reageerida. Manuaalse regressiooni puhul tuleks ise täiendavalt välja mõelda, et mis täpselt võiks katki minna ja mille tulemusena.

Peamise õppetunnina võib lõputööst kaasa võtta selle, et kasutajaliideste automaatne testimine peaks olema eraldatud välistest allikatest ning püsima rakenduse piirides. Tähenduses siis, et sõltuvused teistest andmeallikatest ning rakendustest tuleks kõrvaldada ja kasutada võimalikult palju heade tavade järgi loodud testandmeid.Välised allikad peaksid olema ise vastutavad oma kvaliteetsuse eest ning sõltuvusi on mõistlikum testida teistel tasanditel kui kasutajaliideste testides. Näitena siis ühiktestides.

Täiendavalt peab kasutajaliidese testide puhul arvestama sellega, et need testid võtavad omajagu aega ning testimisel pole mõtet kõike automatiseerida ja on mõistlik mõelda ka testide optimaalsuse peale.

Kasutatud kirjandus

[1] Bonnie Eisenman, “What is React Native?", https://www.oreilly.com/library/view/learning-react-native/9781491929049/ch01.html (20.05.2019)

[2] React Native, https://facebook.github.io/react-native/

[3] Detox, https://github.com/wix/Detox (20.05.2019)

[4] Rotem Mizrachi-Meidan, „Detox: Gray Box End to End Testing Framework for Mobile Apps”, https://hackernoon.com/detox-gray-box-end-to-end-testing-framework-for-mobile-apps-196ccd9564ce (20.05.2019)

[5] Automation Panda, „The Testing Pyramid”, https://automationpanda.com/2018/08/01/the-testing-pyramid/ (20.05.2019)

[6] Sergey Kolodiy, „Unit Tests, How to Write Testable Code and Why it Matters”, https://www.toptal.com/qa/how-to-write-testable-code-and-why-it-matters (20.05.2019)

[7] Smartbear, „Automated Testing Best Practices and Tips”, https://smartbear.com/learn/automated-testing/best-practices-for-automation/ (20.05.2019)

[8] Guru99, „Test Data Generation: What is, How to, Example, Tools”, https://www.guru99.com/software-testing-test-data.html (20.05.2019)

[9] Atlassian, „Understanding the Bamboo CI Server”, https://confluence.atlassian.com/bamboo/understanding-the-bamboo-ci-server-289277285.html

[10] Detox, API Reference, https://github.com/wix/Detox/blob/master/docs/README.md

[11] Detox, „Step 2: Add build and test commands to your CI script”, https://github.com/wix/Detox/blob/master/docs/Guide.RunningOnCI.md

[12] Henri Perkmann, „Veebirakenduse kasutajaliidese automaatne testimine väleda arendusprotsessi kontekstis – põhimõtted ja implementatsioon” lk 25, https://dspace.ut.ee/bitstream/handle/10062/50453/Perkmann_BA2015.pdf?sequence=1&isAllowed=y