Tallinna Ülikool

Digitehnoloogiate Instituut

Üheleherakenduse loomine Ruby on Rails raamistiku abil

Bakalaureusetöö

Autor: Kaspar Tint

Juhendaja: Jaagup Kippar

Autor: .....,,.....,,2016

Juhendaja: .....,,.....,,2016

Instituudi direktor: .....,,.....,,2016

Tallinn 2016

Autorideklaratsioon

Deklareerin, et käesolev bakalaureusetöö on minu töö tulemus ja seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on viidatud.

....

(kuupäev) (autor)

Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemisek

Mina .....(sünnikuupäev:.....)

1. Annan Tallinna Ülikoolile tasuta loa (lihtlitsentsi) enda loodud teose

   .....

   .....

   .....

   .....

   mille juhendaja on .....,

   säilitamiseks ja üldsusele kättesaadavaks tegemiseks Tallinna Ülikooli Akadeemilise Raamatukogu repositooriumis.

2. Olen teadlik, et punktis 1 nimetatud õigused jäävad alles ka autorile.

3. Kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest tulenevaid õigusi.

Tallinnas/Haapsalus/Rakveres/Helsingis, ...

Sisukord

Sissejuhatus

Üheleherakendus ehk SPA on veebirakendus mis ei vaja täielikku lehe uuesti laadimist sellel navigeerimisel. Sellisest rakendusest võib mõelda kui suuremahuliselt kliendist, mis laetakse alla veebi serverist nii, et kasutajal ei ole vahepeal vaja enam uusi elemente juurde laadida. (Mikowski & Powell, 2013)

Ühelehe rakendusi oli vanasti keeruline kirjutada, kuna ei eksisteerinud veel häid kliendipoolseid Javascript teeke, millega kiirelt selline rakendus valmis kirjutada. Lisaks oli vanasti Interneti kiirused kehvavõitu, mis tähendas seda, et üheleherakenduse kasutaja oleks pidanud väga kaua ootama, et esialgne leht lahti saada. Samas on vahepeal tekkinud erinevaid kliendipoolseid raamistikke, mis kasutavad enamasti Javascript keelt. Mõned tuntumad nendest on Angularjs, Backbonejs ja Emberjs. Need raamistikud teevad programmeerija töö oluliselt lihtsamaks kui on soov vahetada `HTML` faile kliendi poolel ilma, et peaks veebi serverilt neid igakord eraldi küsima. See muidugi tähendab, et arendajal on vaja rohkem tööd teha Javascript programmeerimise keelt kasutades kuid samas teeb üheleherakendus kasutaja kogemuse antud veebilehel oluliselt paremaks, kuna ta ei pea pidevalt ootama, et veebileht ennast uuesti täielikult ära laeks, enne kui uus sisu on nähtav.

Sellisel lahendusel on muidugi ka omad miinused. Nimelt peab veebirakenduse kasutaja ootama pisut kauem esialgse lehe laadimisel kui tegu on keerulise ja mahuka lehega. Lisaks peab arendaja palju Javascript koodi kirjutama, et lehel vajaminevad ressursse dünaamiliselt serverilt pärida vastavalt vajadusele. See aga võib tekitada mõneks sekundiks lehel teatud anomaaliaid, kuna vajalikke andmeid pole veel millega kõiki tühimikke täita. Samas annab selliseid probleeme ilustada erinevate graafiliste lahendustega, mis annavad kasutajale edasi infot selle kohta, et midagi laeb veel.

Siinses bakalaureusetöös saab loodud üks üheleherakendus, mis täidab lihtsa sotsiaalvõrgustiku rolli. Uuritakse, millised probleemid esinevad seda sorti rakenduse loomisel ja milliseid tehnoloogiaid oleks vaja kasutada töö jooksul.

1. Ühelerakendus

Üheleherakenduse ajalugu on hägune. Väidetavalt oli 2005 aastal juhtumeid, kus mõned arendajad hakkasid seda sorti rakenduse loomisega eksperimenteerima. Kuna aga sellel ajal veel ei eksisteerinud häid Javascript teeke mis oleksid aidanud seda sorti rakenduste arendamisel kaasa, siis tollel ajal nende rakenduste arendamine veel hoogu sisse ei saanud. Tänapäeval on aga selliste rakenduste loomine aina populaarsemaks muutuv.

Üheleherakendus on veebirakendus, kus sirvija enamasti laeb esimesel laadimisel kõik vajalikud HTML, Javascript ja CSS failid brauserisse. See võimaldab pakkuda kasutajale parema kogemuse, kuna ei ole vajadust oodata peale igat tegevust, kuni server terve uue lehe veebilehitsejale annab. Harilikult küsitakse jooksvalt muutuvad andmed üheleherakenduses XMLHttpRequest abil. XMLHttpRequest on rakendusliides veebilehitsejates mille abil saab dünaamiliselt küsida serverilt erinevas vormingus infot. Toetab see rakendusliides JSON, HTML , XML ja lihtteksti vormingut. Üheleherakenduste puhul kasutatakse enamasti protokollina JSON vormingut kuna seda on väga lihtne lugeda ja kasuta Javascript koodis.

Levinud on kaks põhilist moodust, kuidas üheleherakendust kasutajale kättesaadavaks teha. Esimene neist on viis kus kasutaja esmasele serveri päringule tagastatakse HTML leht mille sees on mainitud ära kõik Javascript ja CSS failid mida peaks veel lisaks alla tõmbama. Kui kasutaja lehitseja on suutnud kõik Javascript failid alla laadida, siis hakkab üheleherakenduse kood juba tööle, laadides veel lisaks kõik vajalikud HTML lehed alla ja muutes need mugavasse vormingusse, et neid hoiustada niikaua kui on vaja. Teise lahenduse puhul kasutatakse kahte eraldi veebiserverit. Üks veebiserver ehk back-end mis tegeleb ainult XMLHttp päringutega ja teine server ehk front-end mis serveerib kasutajale ainult HTML, Javascript ja CSS faile. Mõlemal lahendusel on omad head ja vead.

Kuna standardselt on mõeldud tuntumad veebiraamistikud kuvama serveri poolt täidetud HTML lehti, siis ei ole lihtne esimest moodust üles seada. Tuleb arendajal ise mõelda, kuidas serveerida ainult esimene päring serveri poolt täidetud HTMLi ja järgnevate päringute puhul JSON vormingut. Samas kui selle probleemiga on ühele poole saadud, siis jätkub arendus nagu ikka. Teise moodusega aga tekib probleem, kus on vaja hallata kahte täiesti erinevat projekti. Üks on siis kliendipoolne rakendus ja teine veebiserver. Lisaks tuleb arvestada, et kliendipoolse rakenduse peamine programmeerimise keel on Javascript

ning kahjuks ei ole selle keele teegid veel piisavalt küpsed, et neid täielikult usaldada. Sellise projekti struktuuri haldamiseks oleks kindlasti vaja täiesti eraldi kliendipoole ja serveri poolseid arendajaid kuna pidevalt kahte erinevat projekti hallata on keeruline ja aega nõudev. Siiski kui selline lahendus juba valida, siis võiks eeldada, et projektil on väga suur rõhk front-endi poolel ja on tähtis, et kasutajatel oleks hea kogemus lehte külastades. On aga raske leida arendajaid kes oleksid korraga pädevad nii veebiserveri arenduses kui ka ilusa disaini välja mõtlemises.

Peale eelnevate arendus probleemide tuleb SPA rakenduste puhul kindlasti ka arvesse võtta erinevaid veebilehitsejaid. Tuntumad neist on Google Chrome, Mozilla Firefox ja Microsoft Internet Explorer. Esimese kahe puhul on Javascript keelega probleeme vähem aga kuna Internet Explorer, mis enam uusi täiendusi ei saa, on samuti väga populaarne lehtitseja siis tuleb mõelda täpselt millisele rahvahulgale on rakendus suunatud. Kui jõutakse järeldusele, et oleks ka vaja Internet Explorer lehitsejat toetada, siis paraku tuleb leppida sellega, et arendajal on vaja kasutada vanemat versiooni Javascript keelest. See aga paraku võib nii mõnegi komponendi arenduse keerukamaks teha, kuna keele uued täiendused oleksid teinud töö märksa lihtsamaks.

1.1 Tehnoloogiate valik üheleherakenduse loomiseks

Kuna väga suur rõhk saab olema ka kliendipoolsel programmeerimisel, mitte ainult HTML ja CSS failide kirjutamisel, siis oleks vaja arendajal oma tööd võimalikult palju lihtsamaks teha. Javascript keelt kasutades üheleherakendust on aeganõudev ja raske tegevus - on vaja kirjutada väga palju koodi ja arvestada erinevate Javascript keele eripäradega. Sellepärast on mõistlik teha uuringut teekide kohta, mis aitaksid seda protsessi kiirendada ja lihtsustada. Kui uurida Internetist, millised on preagusel ajal populaarsemat Javascript veebiraamistikud, siis võib leida nimesid nagu Ember, Backbone ja Angular. Kindlast on neid nimesi palju rohkemgi aga kõikide eksisteerivate raamistike arvustamine pole antud töö ülesanne. Lisaks tuleb kindlast arvestada sellega, et raamistikud Javascript maastikul mis täna on populaarsed ei pruugi enam olla seda mõne aasta pärast. Nimelt on antud programmeerimis keele ümber toimetav kultuur väga kiirest edasi arenev - uusi teeke ja raamistikke tekib väga tihti juurde. Antud töö raames aga saab lühidalt uuritud eelnevalt välja toodud raamistikke mis on preagusel ajal populaarsed.

Angular (Freeman, 2014) on populaarse otsingumootori looja Google poolt loodud Javascript raamistik. Angular üritab arendajat panna rakendama ka kliendipoolsel rakendusel MVC mustrit. Kuna paljud veebiarendajad on juba tuttavad MVC mustriga raamistikest nagu Ruby on Rails ja Spring, siis võtab Angulari põhi ideedest aru saamine ka vähem aega nende jaoks. Lisaks on Angular raamistiku suur pluss see, et see on arendatud usaldatava arendaja poolt. Firmad kes võtavad ette otsuse hakata Angular raamistikku kasutama oma uues rakenduses ei pea kartma, et järgmine päev meeskonda kes antud raamistikku arendas ei ole enam. Tuttav arendusmuster ja usaldusväärne raamistiku haldaja on suur trump Angulari nimel. Sellepärast pole ka ime, et just see raamistik on nii populaarseks saanud arendajate seas.

Ember on rohkem tundud väga kiire arengu poolest. Selle teegi arendajad lisavad uut funktsionaalsust väga kiiresti ja samas lõhuvad ka ühilduvust vanemate versioonidega. Lisaks veel dokumentatsioon, mis ei suuda kiire arenguga kaasas käia. Kui aga sellised miinused kõrvale jätta, siis on Ember tõsine valik üheleherakenduste arendajatele. Ember pakub süntaksit mis on väga sarnane Angular teegi süntaksile ja see teeb uutele kasutajatele väga lihtsaks Ember raamistikku ära õppimise kuna väga paljud on just alustanud Google arendatud produktiga.

Backbone ei ole täielik raamistik võrreldes eelnevatega kuna ta ei kasuta C osa ehk kontrollerit MVC mustrist. Backbone pakub arendajatele mudeli ja vaate loomiseks vajalikud tööriistad. Samas on ka nendel oma head ja vead. Backbone eeldab, et veebirakenduse serveri pool ja kliendi osa on sarnaselt kirjutatud. Ruby on Rails arendajatel on sellepärast väga mugav alustada Backbone kasutusega, kuna paljud rakenduse disaini mustrid mida Ruby on Rails raamistik arendajale soovitab, klapivad Backbone arendus mustritega. Samas ei ole mingi probleem kasutada Backbone teeki mõne teise veebiraamistikuga - on vaja lihtsalt rohkem läbi mõelda enne rakenduse loomist kuidas kliendipool ja serveri pool hästi omavahel suhtlema saada.

Peale raamistike valiku peaks arendaja mõtlema millist moodust ta kasutab, et rakenduse kasutajale kliendipoolne rakendus serveerida. Kui seda tehakse veebirakenduse serveri poolelt, siis palju lisa mõtlemist harilikult ei ole vaja. Samas kui on plaan hoida mõlemat poolt rakendusest eraldi arvutitel, siis oleks

vaja ka kaaluda millist Javascript paketihaldurit ja ehitus süsteemi kasutada. Enamus backend veebiraamistikud on kasutusel programmeerimise keelte juures, mille kommuun on juba kaua eksisteerinud ja küpsed tööriistad nendele keeltele loonud. Javascript keelele on aga alles hiljuti alustatud tööriistade loomist kuna varem lihtsalt ei pandud nii suurt rõhku nagu tänapäeval veebilehtedel kliendipoolsele programmeerimisele. Hetkel on tuntumad Javascript teekide haldurid NodeJs raamistiku poolt arendatud npm programm ja bower. Mõlemad programmid aitavad arendajal lihtsalt kirjutada teatud manifest faili kõik rakenduse poolt vajaminevad teegid ja kui programmile käsk anda siis ta laeb need Internetist kõik alla kontrollides samaaegselt, et ei tekiks versioonide konflikte. Nii npm kui ka bower lahendavad sisuliselt sama probleemi - lihtsalt mõlemal on omad eripärad mis võivad aga ei pruugi mõnele arendajale meeldida. Sellepärast oleks ka tark teha enne arenduse alustamist põhjalik analüüs mõlema paketihalguri kohta.

Javascript kogukonnal on ka kaks tuntumat programmi ehitus süsteemi - Grunt ja Gulp. Need tööriistad suudavad automeerida tüüp probleeme millega arendajad muidu igapäevaselt kokku peaksid puutuma. Mõlemad programmid saavad hakkama projekti suuruse vähendamisega mis on protsess kus kaotatakse ära tühikud programmi sees. See on vajalik kuna veebirakenduste maailmas on iga bitt ja bait väga oluline kuna sellest oleneb kui kiiresti kasutaja oma veebilehitsejas lehte nägema hakkab. Lisaks tegelevad mõlemad programmid SASS ja LESS stiilifailide ümber kompileerimisega CSS failideks ja samuti ka Coffeescript ja TypeScript programmide ümber kompileerimisega Javascript failideks. Nendel on ka veel palju lisa funktsioone mida arendaja võib soovi korral kasutada. Nii Grunt kui Gulp programmid lahendavad samuti sama probleemi aga vahe on lihtsalt selles, kuidas neid programme üles seadma peab. Arendaja peab mõlema programmi puhul hoolikalt lugema nende dokumentatsiooni, et aru saada kuidas programmile selgeks teha, milliseid rakenduse osi ja kuidas, muuta või kasutada. Kuna seda sorti programmid oma olemuselt tegelevad paljude aeganõudvate tegevustega on tähtis, et arendaja teeks omale selgeks nende tarkvarade põhimõtted ja ülesanded ning tutvuks hoolikalt dokumentatsiooniga, enne rakenduse arendust. Kui valida õige Javascript keele ehitus tööriist, siis on võimalik arendajal säästa suurel hulgal aega korduma kippuvate ülesannete arvelt ja keskenduda koodi kirjutamisele.

2. Rakenduse idee

Rakenduse ideeks on lihtne sotsiaalvõrgustik, kuhu inimesed saaksid siseneda oma kasutajatega ning peale seda jälgida oma uudiste voogu ning jagada sõpradega pilte või mõtteid. Kuna tuntumad sotsiaalvõrgustikud ei ole kasutanud SPA lähenemist oma rakenduste arendusel, siis oleks huvitav proovida midagi sellist luua just nimelt üheleherakendusena. Reaalset kasutajate koormust on loodavale rakendusele muidugi raske saada aga on võimalik siiski eelnevalt juba läbi mõelda millised osad rakendusest hakkavad rohkem kasutust saama ja mida teha, et kõikidel tulevastel kasutajatel oleks mugav ja kiire kogemus loodud lehel. Kuna populaarsemad võrgustike arendajad nagu Facebook, Instagram, Twitter ja Google+ jagavad väga tihti oma kogemusi läbi erinevate blogide, siis on võimalik juba ette arvestada teatud vajadustega loodava rakenduse puhul. Sellepäras, saavad ka mõned nendest tehnoloogiatest mida antud arendajad kasutavad, kaasatud siinse töö raames loodava rakenduse sisse. Saab töö jooksul läbi võetud ka nende tehnoloogiate kasutamisega seonduvad probleemid ja mugavused.

Loodava lahenduse põhiline komponent saab olema kasutaja ise. Inimene saab luua omale konto antud lehel ning lisada juurde peale konto loomist täiendavat infot enda kohta. Peale oma enda konto loomist ja täiendamist saab kasutaja luua sõprus suhteid teiste kontodega. See võimaldab kasutajal veebirakenduse pealehel näha uudisevoogu, kus on näha kõikide tema sõprade tegevused. Antud lehel võimalikud tegevused on järgnevad:

1. Piltide üles laadimine. Kasutaja saab jagada oma sõpradega pilte endast.
2. "Meeldib" märgi ehk `like` jätmine piltidele, millega saab näidata pildi omanikule oma seisukohta antud sisu kohta.
3. Erinevate kasutajade poolt jagatud pildite kommenteerimine.
4. Mõtete jagamine. Kasutaja saab jagada sõpradega lihtsalt oma mõtteid millegi kohta.

Kui kõik eelnevad komponendid kokku panna ühele lehele, siis olekski võimalik luua algeline sotsiaalvõrgustik. Vaja oleks lisaks veel muidugi ideed millega konkurentide seast välja paista ja kasutajaid kes lehte külastaksid.(Boyd & Ellison, 2007)

3. Arenduse plaan

Enne rakenduse arendamist on vaja luua plaan. Plaan peaks sisaldama endas lehe komponentide loomise järjekorda, tehnoloogiate nimistut mida oleks vaja kasutada ja infot sellekohta, kuidas plaani realiseerida üheleherakendusena. Kuna antud töö jooksul on juba läbi käidud üheleherakenduse olemus ja sellega kaasnevad arenduse probleemid ning muu info, siis viimast punkti ei ole tarvis eraldi kirjeldada enam.

Komponentide arendamise järjekord on aga tähtis osa. Kuigi antud arendus ei ole mõne erafirma poolt tehtav ning kuna puuduvad ka ajaliimid siis peab ikkagi arvestama sellega, et mõned komponendid on mõistlik arendada ennem kui teised. Kõige esimene osa rakenduse arendusest on lihtsa baas struktuuri loomine ning kõikide seadistuste korrektseks kirjutamine. Oleks vaja saada üles lihtne "Tere, maailm!" stiilis veebirakendus, mis tõestaks samas, et esialgne andmebaas ja muud tehnoloogiad töötavad. Rakenduse back-end arenduseks on kasutusel Ruby on Rails veebiraamistik mille ülesseadmist on autor ka oma varasemas tööd kirjeldanud (Tint, 2015). Võib arvestada, et baas tehnoloogia kogum on sama - Ruby on Rails raamistik rakenduse serveri poolel ja relatsiooniliseks andmebaasi kliendiks on PostgreSQL, mis on tuntud vabavaraline andmebaasi lahendus. Kui eelnev osa on paigas ja lihtne leht on kuvatud, oleks edasi vaja mõelda kliendipoolse arenduse kohta. Front-end rakenduse loomiseks saab kasutusse võetud Angular ja selle raamistiku peale koodi kirjutamine hakkab toimuma `Coffeescript` dialektis. Coffeescript on keel mis enne lõppkasutajale andmist kompileeritakse tagasi Javavascript keeleks. Selline keel sai valitud kuna tema süntaks sarnaneb väga palju Ruby süntaksile. See teeb koodi sees navigeerimise lihtsamaks, kuna on vähem vaja harjuda iga erineva faili avamisel teise keelega. Stiili poole pealt saab valitud SASS keel. SASS võimaldab komaktsemalt kirjutada stiili faile ning kasutada ka muutujaid, mida CSS niisama ei luba. SASS kompileeritakse samuti lõpuks CSS failiks mis tähendab, et lõpp koodis kõik muutujaid kasutavad kohad asendatakse selle reaalse väärtusega. See aga ei ole tähtis kuna SASS keele eelis on just see, et arendajal on kõvasti lihtsam lugeda stiili faile.

Järgmine suurem etapp on kasutajate loomise ja nende autentimise funktsionaalsuse loomine. Kuna kõik järgnevad komponendid mis on vaja luua sõltuvad kasutajate olemaolust on tähtis selline osa eelnevalt korralikult valmis teha. Ruby on Rails kogukond pakub siin kohal aga suurt mugandust. On valmis kirjutatud gem nimega `devise`,

mis tegeleb kõigega mis on kasutajate kontodega seotus. Selle paigaldamise õpetus on lihtne käskude käivitamine tänu millele lõpuks tekib tabel andmebaasi mille nimi vaikimisi on `users`. Vajadusel annab erinevaid lisa funktsionaalsusi lasta devise teegil luua nagu kasutajatelt email autentimise nõudmine ja pika ajalise sessiooni loomise võimalus. Kuna devise teek on kogukonna poolt arendatud spetsiaalselt tegelema kõigega mis seondub kasutaja kontodega siis võib usaldada, et on targem kasutada seda teeki kui hakata ise uut lahendust sellise probleemi jaoks välja mõtlema. Kui vaadata näiteks Spring raamistiku poole siis tuleb välja, et seal on küll autentimine Spring raamistiku poolt pakutud aga paroolide krüpteerimine ja ja dekrüpteerimine on vaja ise kirjutada. Samuti on vaja ise mõelda kuidas andmebaasi muudatusi automatiseeritud viisil teha või hakata Internetist otsima eraldi teeke mis sellise probleemiga tegeleksid. Kuna tegemist saab olema üheleherakendusega siis tuleb kasutada devise alammoodulit `devise token auth`. Selle teegi abil hakkab loodav rakendus kasutama nii öelda turvamärki, mille kliendi veebilehitseja endale rakenduse serverilt saab. See turvamärk on kogum teksti mis sisaldab kogu vajalikku infot, et teha kindlaks autentija isik. Turvamärgi sees olev tekst on krüpteeritud kujul, et vältida probleeme kus mõni pahatahtlik isik soovib kasutaja sessiooni lehel üle võtta. Devise token auth teek töötab väga hästi ka Angular raamistikuga - on olemas mugav rakendusliides mille abil saab teada anda kas oleks vaja kasutaja luua, autentida või sessioon lõpetada.

Peale kasutajate loomise komponendi arendamist on võimalik jätkata üleanud rakenduse idees välja toodud osadega. Oleks mõistlik esiteks luua võimalus kasutajatel luua postitusi mis sisaldaksid esialgu lihtsalt teksti. Kui see on tehtud siis saab teha edasiarenduse kus on võimalik ka lisada pilte postituse juurde. Piltide serverisse saatmise funktsionaalsuse loomisel tekib võimalus ka igal kasutajal lasta omale profiili pilt üles laadida. Lõpetuseks oleks vaja lasta kasutajatel üksteist sõbraks lisada ja muidugi lasta igal kasutajal panna `like` erinevatele tegevustele ja ka kommentaare lisada.

3.1 Ruby on Rails valik

Sai valitud Ruby on Rails raamistik kuna see pakub koheselt palju lahendusi, mille abil on lihtsam üheleherakendusi ehitada. Märkimisväärsem eelise annab `asset pipeline` tehnoloogia mis tuleb vaikimisi kaasa Ruby on Rails raamistikul loodud rakenduse loomisel. See tehnoloogia võimaldab kliendipoolset koodi, olgu see siis Javascript või CSS, faili suuruse vähendamise eesmärgil töödelda erinevate

algoritmide abil. Seda funktsionaalsust ei ole mõistlik kasutada rakenduse arendamise faasis kuna sellisel juhul muutub väga tülikaks kliendi poolses koodis olevate vigade tuvastamine. Samas, on see funktsionaalsus väga oluline, kui rakendus lõpuks üldsusele kättesaadavaks teha - kuna siiani on tähtis, et veebilehtede külastajatel oleks võimalik leht avada võimalikult kiiresti siis on väga oluline, et kogu kliendipoolne kood oleks võimalikult hästi tihendatud ehk kiiresti kättesaadav. Lisaks annab asset pipeline arendajale võimaluse kasutada erinevaid kliendipoolseid keeli nagu Coffeescript, Less ja SASS. Asset pipeline suudab nendest keeltest aru saada ja need lõpuks tagasi kas Javascript- või CSS koodiks muuta. Kuna on tähtis, et arendaja saaks oma koodi kiiresti luua siis paraku on selline tehnoloogia väga suur trump Ruby on Rails raamistiku juures. Kui võrrelda Ruby on Rails raamistikku näiteks tuntud Java baasil loodud `Spring` raamistikuga, siis tuleb välja, et Spring seda sorti mugandusi ei paku. Arendajal jääb üle kas ise vastav funktsionaalsus kirjutada või hakata otsima mõnda eraldi teeki. Java puhul on olemas teek nimega Wro4j ehk siis veebi resursside optimiseerija java keelele kuid antud teeki on keeruline üles seada ja lisaks ei suuda ta vaikimise hakkama saada tehnoloogiatega nagu SASS ja Coffeescript.

Rails kommuun on loonud ka veebi lahenduse aadressil https://rails-assets.org/, kus arendajatel on võimalik antud lahendusele ette sööta mõni Git tehnoloogial põhinev hoidla mis sisaldab näiteks Javascript teeki. See rakendus laeb alla koodi sellisest hoidlast ja loob sellest Ruby gem teegi. Peale seda protseduuri on arendajatel koheselt võimalik hakata kasutama seda uut teeki oma Ruby baasil loodud rakenduses. Kuna kliendi poolne arendus maailm on väga kiiresti arenev ja uusi teeke tekib väga tihti, siis on selline lahendus erakordselt mugav arendajate jaoks. Ei ole vaja oodata teiste inimeste taga, et nemad üldsusele mõne teegi uue väljalaske avalikustaks, vaid kui on näha, et uus versioon pole veel kuskilt kätte saadav Ruby gem teegina siis on tarvis lihtsalt eelnevalt mainitud lehele minna ja talle mainida, et oleks vaja uus gem luua teatud hoidlas oleva koodi abil. Muidugi saab ka selle rakenduse poolt loodud gem teeke kasutada ka teistes Ruby keelega loodud rakendustes. Sarnane võimalus on ka olemas Java keelel kuid kuna Java ei paku peale selle paigaldamist vaikimise paketi haldus tarkvara siis tuleb enne eraldi uurida kuidas kasutada võimalusi nagu Maven ja Gradle ja hiljem siis üritada sama funktsionaalsus tööle saada Java keelt kasutades.

Ruby on Rails puhul on samuti väga vähe muudatusi vaja teha, et oleks võimalik seda raamistikku kasutada üheleherakenduste jaoks. Kuna on plaanis arendada rakendus mis peab arvestama, et kliendi esimese päringu puhul tuleb tagastada html leht ja järgnevate päringute puhul JSON vastused, siis on tarvis teatud raamistiku osi muuta, et selline funktsionaalsus üldse toimida saaks. Samuti peaks lehe uuesti laadimine ükskõik mis marsuudil server alati tagastama veebilehe algelise seisu. Kui klient on kõik vajalikud failid lõpuks alla laadinud, saab Angular kood tööle hakata ja taastada soovitava seisu sisestatud marsuudi järgi. Ruby on Rails rakenduse puhul on võimalik esialgu luua väga lihtne filter, mis püüab kinni kõik HTML tüüpi päringud ja viitab need avalehele. Kuna avaleht saab sisaldama koodi mis käivitab Angular koodi, siis on see kõik mis on vaja, et kasutajale luua sisenemis punkt peale igat HTML päringut. (Koodinäide 1)

Koodinäide. 1: ./app/contollers/application_controller.rb

```
before_filter :intercept_html_requests
...
def intercept_html_requests
   render('visitors/index') if request.format == Mime::HTML
end
...
```

Spring raamistiku puhul aga tuleb arendajal ise üles otsida, millist klassi oleks vaja laiendada ja millist meetodit üle kirjutada, et samasugust funktsionaalsust saavutada. Kui uurida Internetist teiste arendajate kogemuste kohta siis tuleb välja, et iga arendaja pakub pisut erinevaid lahendusi selle probleemi lahendamiseks. Neid kõiki lahendusi proovides tuleb välja, et nii mõnigi ei ole täielikult probleemi lahendav. Teatud juhtudel võib ikkagi HTML päring serverini jõuda nii, et klient saab vastuseks lõpuks midagi ootamatut. Kindlasti on võimalik Springi baasil töötav rakendus korralikult valmis seada üheleherakenduse jaoks, aga siia maani ei ole Internetis head õpetust selle kohta ja ise antud probleemi lahendada võib olla aeganõudev tegevus. Spring raamistik on väga mahukas ja keeruline ning ise nende koodi baasis ringi uurimine, et saada teada kuidas mingit funktsionaalsust ümber kirjutada, ei ole lihtne tegevus.

Samas on Ruby on Rails raamistikul ka teatud halbu punkte. Kuna see raamistik

on kirjutatud kasutades Ruby programmeerimise keelt, siis paraku ei ole rakendused loodud antud raamistiku abil väga kiired võrreldes kompileeritud keeltega nagu Java, C# ja GO. Twitter sotsiaalvõrgustik võitles selle probleemiga kirjutades ümber oma Ruby rakenduse Scala rakenduseks, mis jookseb sama virtuaalmasina peal nagu Java. Twitter arendajad on samas rääkinud, et selline otsus oli mõistlik ainult hiljem nende jaoks. Kuna nende arendus algas väga väikese meeskonnaga, siis oli mõistlik kasutada alguses Ruby on Rails raamistikku sest see aitas neil väga luua töötav rakendus mida oleks võimalik üldsusel kasutada. Ruby ja Ruby on Rails raamistik on arenenud jõudsalt oma viimastes versioonides jõudluse poole pealt kuid siiski oleks arendajal vaja enne läbi mõelda, mida on vaja hakata arendama. Kui rahalisi ressursse on piisavalt siis on võimalik, et Spring raamistiku peal kohe alguses arendamine on mõistlik tegevus. Kindlasti on võimalik ka Ruby baasil rakendusi kohandada nii, et need suudaks tohutut kasutust taluda, aga see ei pruugi olla mõistlik, kui on olemas keeled millega sama asja saavutamine võib lõpuks olla soodsam.

4. Rakenduse loomine

Rakenduse esialgsel loomisel on vajalik läbida samad sammud mida siinse töö koostaja läbis oma seminaritöös. Vaja on luua algne rakendus `rails new` käsu abil ning peale seda lisada külge ka `PostgreSQL` andmebaasi suhtlus. Järgnevalt peaks looma kasutajate tabel andmebaasi ja koodi poolele `devise` gem abil vastav Ruby kood, mis laseb luua, autentida ja muuta kasutajaid ning paigaldada ka devise alam-moodul mis lubab kasutada turvamärki autentimiseks. Lisaks on vaja lisada Angular raamistik rakendusse. Angular raamistikku saab projekti lisada gem failist kuna `rails-assets` loob igast uuest Angular versioonist uue gem väljalaske. Kuna rails-assets pole Ruby poolt vaikimise toetatud hoidla siis tuleb Gemfile sees ära mainida selle hoidla asukoht (Koodinäide 2).

Koodinäide. 2: ./Gemfile

```
source 'https://rails-assets.org' do
  ...
  gem 'rails-assets-angular', '~> 1.4.1'
  ...
end
```

Koodinäites on näha, et annab luua eraldi alamosa rails-assets hoidla kohta, mille sees annab ära määrata kõik soovitud teegid just sellest hoidlast. Nüüd kui käivitada `bundle install` käsk projekti juur kaustas siis laetakse esiteks alla kõik teegid `ruby gems` hoidlast ja peale seda kõik vajalik rails-assets hoidlast. Angular kood tuleks paigutada `./app/assets/javascripts` kausta. Selle rakenduse puhul on `javascripts` kausta sisse tehtud veel eraldi `angular` kaust, et oleks lihtne aru saada kus kohas on just Angular raamistikku kasutav kood. Kuna tegemist on üheleherakendusega, siis on ka vaja Angular koodile ette anda kõik vajalikud HTML failid. Selle jaoks sai tehtud `.app/views` kasuta algne avaleht kus laetakse Angular kood HTML lehe külge ja samuti sai selles kaustas loodud ka navigeerimise riba HTML failid. Kuna rakenduse server tagastab lehe uuesti laadimisel alati esiteks avalehe, siis on selline lahendus mõistlik kuna on võimalik veel ära kasutada Ruby on Rails asset-pipeline teegi poolt pakutavaid mugandusi, mis laseb kliendile vajalikke ressursse ära defineerida avalehel mugaval kujul kasutades `stylesheet_link_tag` ja `javascript_include_tag` meetodeid (Koodinäide 3). Need meetodid vaatavad rakenduse `.app/assets/` kaustas olevaid `application.js.coffee` ja `application.css.scss` faile ja lisavad serveri poolel HTML koodi kõik nendes failides välja toodud sõltuvused. Sellisel viisil ei ole arendajal eraldi vaja HTML koodi pidevalt

uusi kliendipoolse koodi faile välja kirjutada.

Koodinäide. 3: ./app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html lang="en" ng-app="Fit">
<head>
   <base href="/">
   <meta name="viewport" content="width=device-width, initial-scale=1">
   <title><%= content_for?(:title) ? yield(:title) : "Example" %></title>
   <meta name="description"
   content="
   <%= content_for?(:description) ? yield(:description) : "Example" %>
   ">

   <%= stylesheet_link_tag 'application', media: 'all' %>
   <%= csrf_meta_tags %>
</head>

<body>
   <%# Load current user into Angular session service %>
   <div ng-controller="SessionCtrl"
   ng-init="init_current_user(<%= current_user.to_json %>)">
      <header>
         <%= render 'layouts/navigation' %>
      </header>

      <main role="main">
         <%= yield %>
      </main>
   </div>
   <%= javascript_include_tag 'application' %>
</body>
</html>
```

Kui Angular on kliendi poolt lõpuks laetud, on võimalik hakata kasutama HTML `partial` faile. Partial HTML failid ei sisalda endas kogu veebilehe kirjeldust vaid ainult teatud osa mingisugusest komponendist. Angular oskab selliseid osalisi HTML faile vajalikesse

kohtadesse panna serveri poolt saadud esilehe HTML failis. Üheleherakenduse puhul on tüüpiline, et eksisteerib üks peamine sisu ala mille sees vahetatakse selliseid osalisi HTML faile pidevalt, et kasutajale erinevat sisu näidata. Ruby on Rails rakenduse struktuuris on kaust `./public`, kuhu soovitatavalt oleks vaja paigutada kõik staatilised failid, mida ei ole vaja jooksu pealt serveri rakenduse poolel muuta. HTML faili osakud mida Angular kasutab on täpselt sellised failid. Angular kood küsib need serverilt ilma, et serveri rakendus neid muutma hakkaks. Selline funktsionaalsus ongi üheleherakenduse üks peamistest eesmärkidest - võtta ära HTML failide töötlemise kohustus veebirakenduselt, et tagada kasutajale sujuvam veebilehel navigeerimine.

Peale Angular raamistiku integreerimist rakendusse, on vaja devise moodul tööle saada Angular koodiga. Selleks on kasutusel `angular-devise` teek mille annab Angular koodi laadida moodulina. Angular-devise annab Coffeescript koodi poolel mugava rakendusliidese mille abil saab vähese koodiga saata Ruby on Rails rakendusele kas kasutaja autentimise, sisse- või välja logimise ja kasutaja loomise päringuid. Samuti suudab see teek ise meelde jätta kas kasutaja on sisse logitud või mitte. See aitab lihtsasti kliendipoolse rakenduse sees kontrollida kas teatud tegevusi peaks lubama teha või mitte kuna ei oleks mõistlik lasta kliendipoolses koodis sisse logimata kasutajal luua uusi kommentaare kellegi pildi kohta. Selline tegevus küll ei läheks läbi veebiserveri poolel kuna ei ole võimalik lisada kommentaari ilma autorita kuid siiski on vaja selline võimalus ära kaotada ka kliendirakenduse poolel. Angular-devise laseb seda teha `Auth.isAuthenticated()` käsuga, mis tagastab kas `true` või `false`. Selle abil saab luua lihtsaid kontrolle millega teha kindlaks kas on vaja teatud osa rakendusest näidata või mitte.

Järgenevalt on vaja luua üleanud vajalikud komponendid rakenduse serveri poolel milleks on pildid, sõprus suhted, meeldimised piltidel ja kommentaarid. Need kõik objektid annab serveri poolel valmis luua samal ajal ka vastavaid andmebaasi tabeleid ja kontrollereid luues. Testida, et kõik töötab annab Ruby on Rails poolt pakutud `debug` lahendusega. Enne serveri käima panemist on tarvis rakendusele ette anda parameeter `--debugger` mille abil on võimalik konsooli aknas jooksva rakenduse sees kirjutada tavalist Ruby koodi. Niivisi saab teha päringuid nagu `puts(User.all)`, mis kirjutab jooksva rakenduse logisse kõik andmebaasis olevad kasutajad. Selline viis arenduseks on hea kuna saab enne kliendipoolse osa kirjutamist juba veenduda, et kogu kirjutatud kood töötab nii nagu

vaja. Muidugi on lisaks veel väga soovitav rakendada testide põhist arendusmustrit, kus arendaja kirjutab enne uue komponendi loomist selle kohta testi milles on kirjeldatud kõik oodatavad tulemused ja alles siis loob reaalse komponendi koodi. Selle töö raames aga testide juurde ei minda.

4.1 Andmete ühtsele standardile viimine

Kuna andmeobjektid, mida Ruby on Rails rakendus Angular koodile saadab sisaldavad endas sõltuvusi ka teistele andmeobjektidele siis on mõistlik, et arendaja valib täpselt välja, millist osa andmetest JSON kujule viies Angular rakendusele tagastada. Java maailmas kasutatakse sellest `service layer` komponenti. Arendaja kirjutab iga andmebaasi ehk `PO` objekti kohta ka ühe andme edastus objekti ehk `DTO`. Kui andmed on andmebaasist küsitud ja andmebaasi objekt täidetud saadud andmetega, siis muudetakse selline objekt ümber andme edastus objektist, enne kui see kliendile saadetakse. Ruby on Rails raamistik on kirjutatud aga viisil, mis teeb sellise arendus käigu keeruliseks. Arendaja peab ise hakkama looma keerulist koodi, et saada sealsed `ActiveRecord` tüüpi objektid lihtsamateks objektides. Sellepärast on Ruby on Rails kogukond kirjutanud eraldi teegi sellise probleemi lahendamiseks ja selle teegi nimeks on `active model serializers`. Active model serializer teek laseb arendajal kirjutada lihtsaid Ruby objekte, mis kirjeldavad ActiveRecord objekti lihtsamalt. Siinse bakalaureuse töö raames loodavas rakenduses on olemas `User` object mis sõltub ActiveRecord klassist. See objekt sisaldab endas infot kasutaja kohta nagu email, nimi ja roll. Samas sisaldab see objekt ka infot mida alati ei ole kliendipoolel vaja nagu kõiki selle objektiga seotud teisi objekte milleks võivad olla kommentaarid ja pildid mida see kasutaja on rakendusse sisestanud. Et hoida Ajax päringud võimalikult väikesed, tuleb luua active model serializers teeki kasutades lihtsamas kuvandid sellistest objektidest. Seda sorti lihtsamaid objekte võib olla mitmeid ühe ActiveRecord objekti kohta, arendaja asi on otsustada kus ja millal vastavaid objekte kliendile JSON kujul tagastada. Loodava rakenduse näitel on olemas ühe kasutaja kohta kaks sellist objekti: `user_simple_serializer.rb` ja `user_serializer.rb` (Koodinäide 4). Esimene objekt sisaldab endas kasutaja andmebaasi id koodi, nime, emaili, rolli ja avatari mis on kasutaja pilt. Teine objekt aga sisaldab peale eelnevalt mainitud andmetele ka kasutaja poolt rakendusse sisestatud pilte ja kommentaare. Tuleb ka ära mainida, et kommentaarid ja pildid mida sellise objektiga kaasa saadetakse on

samamoodi omaette active model serializers tüüpi objektid ja antud juhul on nende nimed rakenduses `comment_serializer.rb` ja `medium_serializer.rb`. Et hoida ära juhtumeid, kus rakendus lakkab töötamast kuna tekib lõputu tsükkel andmete JSON kujule viimisel, peaks kommentaaride ja piltide lihtsustatud objektid mitte sisaldama endas täielikku viidet neid loonud kasutajale. Selle asemel saab sellistes objektides lihtsalt välja tuua näiteks kasutaja id ja nime.

Koodinäide. 4: ./app/serializers/user_simple_serializer.rb

```
class UserSimpleSerializer < ActiveModel::Serializer

  attributes :id,
             :name,
             :email,
             :role

  has_one :avatar, serializer: AvatarSerializer

  def avatar
    object.get_avatar
  end
end
```

4.2 Probleemid rakenduse arendamisel

Arenduse etapis, kus sai loodud piltide sisestamine kui kasutaja postitusena ja nende piltide kommenteerimine, muutus Angular koodi haldamine raskeks. Esimene suurem probleem oli viis kuidas osalised HTML failid sai paigutatud. Kuna suurem osa arendusest toimus Coffeescript ja HTML faile muutes, siis paraku oli väga ebamugav pidevalt liikuda `./public` ja `./app/assets/javascripts/angular` kasutade vahel. Ei teinud paremaks ka seisu see, et oli otsus kõik failid mis käisid teatud komponendi alla, paigutada eraldi sellise komponendi nimelisse kausta. Lõpuks muutus väga aega nõudvaks iga kasuta vahel liikumine, et mingit teatud faili leida. Muidugi pakkus JetBrains firma poolt loodud arendus keskkond `RubyMine` palju mugandusi selle probleemi lahendamiseks. Üheks selliseks muganduseks oli terve projekti seest

kiirelt faili leidmine selle nime järgi. Samas kuna faile tekkis väga palju siis tekkis tihti olukordi, kus faili nimi ei tulnud meelde ja oli vaja ise vastavad kaustad avada, et vajalik fail üles leida. Tulevikus oleks hea viis leida lahendus, kuidas Ruby on Rails rakenduses hoida ka osalisi HTML faile `./app/assets/javascripts/angular` kaustas. Probleemi lahenduseks oleks vaja paigutada kõik teatud komponendiga seonduvad failid ühte kausta, et saaks kiiresti leida vajalikke HTML failile ja nendele vastavaid Coffeescript faile. Reaalset lõpptulemust selline lahendus ei muudaks, kuna enne rakenduse üldsusele avalikustamist Internetis, tuleb Ruby on Rails rakenduse kliendipoolne kood alati asset-pipeline teegil lasta ära tihendada mis tähendab seda, et lõpp tulemus paigutatakse automaatselt `./public` kausta ja Ruby on Rails rakendus kasutab sellisel juhul vaikimisi hoopis tihendatud CSS ja Javascript faile. Kindlasti on võimalik pisut funktsionaalsust arendajal ümber kirjutada sellel koha pealt ja lasta asset-pipeline teegil ka vajalikud HTML failid `public` kausta paigutada.

Samuti tuli välja, et Coffeescript on küll pealtnäha mugav keel mille süntaks on väga sarnane Ruby keelele aga paraku muutus selliste failide haldamine kiiresti väga tülikaks. Kuna Angular ise on kirjutatud Javascript keeles, siis paraku on ka Internetis olevad õpetused selle raamistiku kasutamiseks just Javascript keele baasil. See aga tekitas arendus käigus palju probleeme kui oli vaja mõne spetsiifilise probleemi kohta infot leida ja lahendus sellele oli kirjutatud Javascript keeles. Tihti peale aga ei ole väga lihtne olemas olevat Javascript koodi ümber tõlkide Coffeescript koodiks. Eriti tekkis selles osas probleeme Angular `$resource` mooduli kasutamisel. Coffeescript süntaks üldiselt ei soosi sulgude kasutamise koodis, aga $resource moodulit kasutades oli tihti peale see siiski vajalik. Kui sulge ei kasutatud hakkas veebilehtiseja Javascript silur teavitama vigadest. Paraku ei olnud aga vea teates väga probleemi kirjeldavad vaid viitasid enamasti täiesti mitte seonduvale koodile ja vahel ei viidanud üldse kuskile vaid teatasid lihtsalt kasutajat stiilid "Midagi läks katki". Koodinäites 5 on näha koodi, mis lakkas töötamast niipea kui sai eemaldatud meetodi järgsed sulud. Samas on läbi rakenduse palju teisi näiteid, kus $resource mooduli kasutamine toimib ilma sulgudeta.

Koodinäide. 5: ../angular/controllers/content/likes/likeForm.coffee

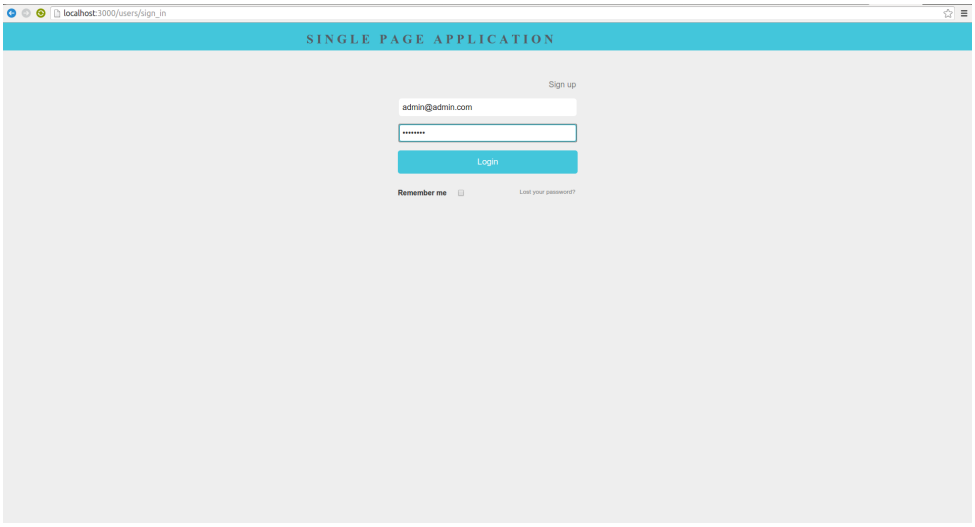
```
LikeService.like({
   id: $routeParams.id,
   user_id: Common.get_current_user.id
}, (response) ->
      medium.likes.push(user_id: response.id)
)
```

4.3 Rakenduse käivitamine lokaalselt

Rakenduse lähtekoodi on võimalik alla laadida autori Github hoidlas, aadressil: `https://github.com/Veske/fit-rails`. Enne rakenduse käivitamist tuleks veenduda, et arvutisse on paigaldatud Ruby programmeerimis keele väljalase `2.3.0` ja Ruby on Rails raamistiku versioon `4.2.6`. Samuti tuleks kindlasti veenduda, et arvutisse on paigutatud Redis andmehoidla ja PostgreSQL mida rakendusel on vaja, et andmeid hoiustada. Andmehoidlate puhul tasub valida kõige uuem stabiilne versioon. Kui on soov rakendusse ka lokaalselt pilte sisestada mõne kasutaja alt, siis oleks vaja ka paigaldada arvutisse ImageMagick programm, mis tegeleb piltide tihendamisega enne nende kõvakettale salvestamist. Antud programm on ainult Linux kernelit kasutavatele operatsioonisüsteemidele mõeldud tarkvara.

Hoidlast olevat koodi on võimalik arvutisse paigutada, kasutades GIT programmi käsku `git clone https://github.com/Veske/fit-rails.git`. Enne Ruby on Rails spetsiifiliste käskude käivitamist oleks vaja lasta Ruby gem programmil alla laadida kõik Gemfile failis välja toodud teegid käsuga `bundler install`. Järgnevalt oleks vaja käivitada Redis andmehoidla kirjutades käsureale `redis-server`. Lisaks oleks vaja liikuda käsureal arvutisse paigaldatud koodi kausta ja käivitada seal `rake db:create` `rake db:schema:load` ja `rake db:seed` käsud. Nende abil luuakse andmebaasi skeem, tabelid ja lõpuks täidetakse tabelid test andmetega, et oleks võimalik veebilehel kohe andmeid näha. Lõpetuseks oleks vaja käivitada rakendus käsu `rails s` abil. Kui rakenduse käivitamise käsk ei kirjuta veateateid käsureale, siis võib avada lokaalselt töötava rakenduse veebilehitsejas, aadressil `http://localhost:3000/`. Rakendus kuvab esiteks lehe, kus on vaja sisestada kasutaja autentimiseks vajalikud andmed (Pilt 1). Email väljale võib kirjutada admin@admin.com ja parooli väljale changme, et oleks võimalik sisse logida

rakendusse. Rakenduse Github hoidlas on olemas ka põhjalikum käivitamise õpetus inglise keeles, mis selgitab kuidas rakendust käivitada ka operatsiooni süsteemidel nagu Microsoft Windows.



Pilt. 1: Autentimise leht

Kokkuvõte

Siinse bakalaureusetöö käigus õnnestus uurida mis on üheleherakendus ja kuidas sellist tüüpi rakendust arendada. Samuti õnnestus lihtsa sotsiaalvõrgustiku prototüübi loomine, mis sai arendatud üheleherakendusena. Üheleherakenduse kohta uurimisel tuli välja, et vajalik on teha põhjalik eeltöö, enne kui arendama asutakse. Arendajal on valida suurel hulgal abistavate teekide ja raamistike seast, mis kõik on üksteisest vähem või rohkem erinevad. Siinse töö käigus sai valitud Ruby on Rails veebirakenduse raamistik, kuna selle abil on võimalik säästa palju aega tüüpiliste arendusprobleemide juures nagu kasutajate autentimine ja andmebaasi muudatuste haldus. Lisaks sai valitud kliendipoolseks raamistikuks Google poolt loodud Angular, mis soosib arendajaid kasutama Ruby on Rails raamistikule väga sarnast arendusmustrit. Uuritud sai ka kahte erinevat moodust, kuidas kasutajale kliendipoolne rakendus serveerida. Moodus, kus kasutatakse NodeJs tarkvara jättis liiga keeruka mulje peale uurimist, kuna sisaldas endas ebaküpseid pakihaldussüsteeme, mis vajavad palju aega, et need üldse tööle saada. Samuti oleks selle lahenduse puhul muutunud keeruliseks arendusprotsess, kuna vajalik oleks olnud kahte eraldi projekti korraga hallata, mis on arendustööd aeglustav tegevus. Sellepärast sai valitud variant, kus kliendipoolne rakendus serveeritakse veebilehitsejale Ruby on Rails rakenduse poolt. See valik osutus õigeks, kuna arendusprotsess kujunes kiireks ja mugavaks - oli võimalik keskenduda põhiliselt funktsionaalsuse arendusele.

Rakenduse loomisel sai tehtud ka mitmeid vigu. Valik kasutada Coffeescript keelt Javascript asemel oli viga, kuna Coffeescript tekitas probleeme Angular raamistiku poolt pakutavate liideste kasutamisel. Lisaks oli raske leida abi tekkinud probleemidega selle keele kasutamisel, kuna Angular dokumentatsioon ja ka erinevad Angular arendajate blogid on kirjutatud kõik eeldusega, et lugeja kasutab arenduses Javascript keelt. Coffeescript aga sai algul valitud, kuna tema süntaks on väga sarnane Ruby keelega, mis oleks võinud Ruby ja Coffeescript failide vahel liikumist sujuvamaks teha. Tuli aga välja, et see ei olnud piisavalt hea põhjus selle keele valikuks. Lisaks tekkis probleeme Coffeescript ja HTML failide paigutamisega. Oli raske liikuda projekti kaustas nende kahe eraldi tüüpi failide vahel, kuna need failid asusid liiga erinevates kohtades.

Töö lõpuks valminud rakendus sisaldab endas funktsionaalsust, mis sai välja toodud rakenduse idee peatükkis. On võimalik liikuda veebilehel, mis kujutab endast lihtsat sotsiaalvõrgustikku. Kuna tegemist on üheleherakendusega, siis tundub arendatud

rakendus silma kiire - puuduvad täielikud veebilehe uuestilaadimised. Siiski oleks rakendusel ruumi edasiarenduseks. Oleks mõistlik luua iga komponendi kohta eraldi kaust kuhu sisse saaks paigutatud nii HTML, kui ka Coffeescript kood ning samuti oleks mõistlik kirjutada Coffeescript failid lõpuks ümber Javascript failideks, et see osa arendusest muutuks edaspidi lihtsamaks. Lisaks jäi selle bakalaureusetöö raames mõõtmata loodud rakenduse jõudlus, kuna ei ole loodud tööriistu, mis suudaks lokaalselt töötava üheleherakenduse veebilehel liikumise kiirust mõõta nii, et tulemused oleks võrreldavad reaalse kasutusega . Info sellist tüüpi rakenduse jõudluse kohta oleks oluline, kuna oleks võimalik teada saada, kas üheleherakendus ka päriselt kiirem on, kui harilik veebirakendus.

Kasutatud kirjandus

Boyd, D. M., & Ellison, N. B. (2007). Social network sites: Definition, history, and scholarship. *Journal of Computer-Mediated Communication*, *13*(1), 210–230. doi: 10.1111/j.1083-6101.2007.00393.x

Freeman, A. (2014). *Pro AngularJS*. doi: 10.1007/978-1-4302-6449-1

Mikowski, M., & Powell, J. (2013). Single Page Web Applications. , 432. Retrieved from `https://d1wshrh2fwv7ib.cloudfront.net/courses/a38264fc-bf10-4d1f-93c5-ca8952408c39/literature{_}entry/f29033b4-009d-4214-b9f1-853ff28e74bc.pdf`

Tint, K. (2015). Ruby on Rails raamistik Veebilahenduse või veebiteenuse koostamine selle abil. Retrieved from `http://www.cs.tlu.ee/teemaderegister/get{_}file.php?id=397{&}name=Ruby{_}on{_}Rails{_}raamistik.Veebilahenduse{_}voi{_}veebiteenuse{_}koostamine{_}selle{_}abil-Kaspar{_}Tint.pdf`

Summary

Title: Creating a Single-Page Application Using Ruby on Rails Framework

The purpose of this Bachelor Thesis was to develop a single-page web application with Ruby on Rails framework while investigating what kind of problems arise when developing it and also what kind of technologies are essential for this type of application. It turned out that choosing the technologies for a single-page application is not an easy task as the Javascript community has created a lot of very different libraries and frameworks that are meant to help the developer out when creating such an application. Also the Javascript libraries tend to often get updates that break backwards compatibility. Because of that it was decided that Angular framework that is developed by Google is to be used when creating the single-page application. One can expect that a successful company like Google will not drop support for their work in the near future.

The idea of the application was to develop a simple version of a social network where users could upload pictures that they wanted to share and then comment on the uploads of other people and also `Like` the content that they want. It was decided that the back end for the application would be written using the Ruby on Rails web application framework and the front end would be developed using Angular framework. Ruby on Rails was used because it helped speed up time consuming and complex tasks like authentication and database migrations. Ruby on Rails community has developed a Ruby gem named `devise` that creates the authentication system into a Ruby on Rails application with a couple of console commands. It was a good choice to use Ruby on Rails framework because most of the application logic had to be written in front end side of the application. A lot of time was saved because most of the back end development was simple and fast to do.

The result of this Bachelor Thesis was a single-page social network application. The author learned a lot about different front end technologies while creating it and was successful at developing a Ruby on Rails back end that was able to serve the initial `html.erb` file which was the entry point for Angular application and at the same time keep the rest of the application respond only to JSON API calls.