TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Informaatika instituut
Infosüsteemide õppetool

JavaScript raamistikude analüüs ja võrdlus struktureeritud SPA rakendustes

Magistritöö

Üliõpilane: Jevgeni Rumjantsev
Üliõpilaskood: 144252IAPM
Juhendaja: lektor Raul Liivrand

Tallinn
2015

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.

---------- ----------

(*kuupäev*) (*allkiri*)

Annotatsioon

Magistritööl on kaks eesmärki:

1. Analüüsida tänapäeval populaarsemaid *JavaScript* raamistikke eesmärgiga arendada erinevate suurustega rakendusi. Analüüs tehakse abstraktsel näiterakendusel, mis käsitleb taaskasutatavat põhifunktsionaalsust ja raamistikuga seotud probleeme, produktiivsust, rakenduse hallatavust ja paindlikkust.

2. Võrrelda sarnase funktsionaalsusega *JavaScript* raamistikud eesmärgiga aru saada, millist raamistiku on parem kasutada. Lisaks ka aru saada, kas analüüsitud *JavaScript* raamistikuga ehitatud rakendust on võimalik toetada lähi- ja kaugtulevikus.

Rakendust pannakse kokku kogukonna poolt valmis tehtud komponentidest. Autor uurib, kuidas mõjutab paigaldatud komponendid rakenduse laadimiskiirust ja töövõimet. Samuti peetakse silmas ohtusid raamistikuga arendamises: rakenduse lahtisus-kinnisus, arhitektuur ja selle paindlikkus ja haldamine, raamistiku õppimise raskus, rakenduse testimine ning kui hästi on otsingumootoritega leitav.

Töö käigus on avastatud, et tänapäeva populaarsemateks *JavaScript* raamistikuteks on *AngularJS*, *EmberJS*, *ReactJS* ja *BackboneJS*. Kogukonna toetus mängib kriitilist rolli komponentide väljaotsimisel, mis pakuvad lahendusi enamlevinutele probleemidele. *AngularJS*-l on kõige suurem toetus kogukonna poolt ja vaatamata raamistiku puudustele on valminud nõrku külgi parandatavad komponendid. *EmberJS* raamistik tihedalt kasutab tänapäeva parimaid praktikaid, kuigi komponentide hulga ja kvaliteedi poolest on kõige nõrgem raamistik. *BackboneJS* raamistik on hea nii väikeste kui suurte rakenduste ehitamiseks ning on heas koostöös *jQuery* teegiga. Kõige värskem tehnoloogia – *ReactJS* – omab kõige suuremat potentsiaali, kuid võrreldes teistega kõige vähem toetatud ja ebastabiilsem. Töö lõpus on toodud üldised parimad praktikad, mida on võimalik rakendada iga raamistikuga arendamisel.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 73 leheküljel, seitse peatükki, 36 joonist, 15 tabelit.

Abstract

There are two goals in Master's degree:

1. Analyze nowadays most popular *JavaScript* frameworks in order to develop scalable applications. Analysis is based on an abstract example application, which handles reusable functionality and framework's problems, productivity, application management and agility.

2. Compares *JavaScript* frameworks with similar functionality with the aim to understand, which framework would be optimal to use. In addition, another main question is, will it be possible to maintain application in near or far future, which was built using an analyzed *JavaScript* framework.

Application is composed of components built by community. Author studies how do components affect the application's loading speed and performance. There are also considered some dangerous points in application development: application's openness, architecture and its agility and management, learning curve, testing and search engine optimization.

During the research, it was found out that nowadays most popular frameworks are considered *AngularJS*, *EmberJS*, *ReactJS* and *BackboneJS*. Community plays critical role when searching for existing solutions. *AngularJS* has the biggest community and therefore it helps to cover the weak spots by supplying helpful components. *EmberJS* framework often uses best practices but overall quality and quantity of components is low. *BackboneJS* is well optimized for building as small as big applications and works well with *jQuery* and a great deal of other components. *ReactJS* is the newest technology and it has the biggest potential but has relatively small number of components and unstable. In the end of the thesis, author writes about best practices, which could be used nowadays when developing using any kind of framework

The thesis is in Estonian and contains 73 pages of text, seven chapters, 36 figures, 15 tables.

Lühendite ja mõistete sõnastik

SPA	***Single Page Application*** Üheleheline rakendus, mis töötab ainult ühel lehel eesmärgiga anda kasutajatele lauarvuti rakendustega sarnast sujuvat kogemust, kus kõik vajalik kood – *JavaScript*, CSS, HTML – on saadud ühe laadimisega.
MVC	***Model View Controller*** Metoodika nimi või disain muster, mis seob edukalt ja tõhusalt kasutajaliidese loogikat andmed mudelitega. MVC mustrit kasutatakse laialdaselt programmide kirjutamisel, kasutades programmeerimiskeeli.
MVVM	***Model View ViewModel*** Tarkvara arhitektuuriline muster kasutajaliidese implementeerimiseks. MVVM on variatsioon Martin Folwer'i *Presentation Model* mustrist, kus MVVM tekitab abstraktse kihi vaate oleku ja käitumise vahel, kuid teeb seda selliselt, mis ei seo sõltuvusse kindlaid kasutajaliidese platvorme.
ES	***EcmaScript*** *EcmaScript* on skriptitud keel, mis on standardiseeritud Ecma International-i poolt ECMA-262 spetsifikatsioonis ja ISO/IEC 16262. Keel on laialt kasutatud kliendipoolsete skriptide kirjutamiseks veebis, mis esineb erinevate implementatsioonidena, nagu JavaScript, JScript ja ActionScript.
DOM	***Document Object Model*** Platvormiülene ja keelest sõltumata konvensioon esinamideks ja toimimiseks objektidega HTML-s, XHTML-s ja XML-s.
MIT Licence	***The license of Massachusetts Institute of Technology*** Lubav tasuta tarkvara litsents, mis lubab korduvalt kasutada vastava litsentsiga litsentseeritud tarkvara koopiaid koos MIT litsentsi tingimustega ja autoriõiguse teatega.

GNU LGPL	**GNU Lesser General Public License** Tasuta tarkvara litsents, mis võimaldab arendajatel ja firmadel kasutada ja integreerida LGPL tarkvara nende isiklikusse tarkvarasse.
BSD	**Berkeley Software Distribution** *Unix*-i operatsioonsüsteemi tuletis, mis on arendatud ja levitatud *Kalifornias* asuvas *Berkelei* ülikooli *Computer System Research Group*-i poolt .
UI	***User Interface*** Ruum, kus toimub koostoime inimese ja masina vahel. Koostoime eesmärgiks on võimaldada efektiivset opereerimist ja kontrolli masina üle.
HTML	***HyperText Markup Language*** Standardne märgekeel, mida on kasutatud veebilehtede ehitamiseks. HTML on kirjutatud kasutades HTML elemente, mis koosnevad nurksulgudega ümbritsetud *tag*-dest.
SEO	***Search Engine Optimization*** Protsess veebilehe nähtavuse mõjutamiseks otsingumootori tulemuste jaoks. Populaarsemad otsingumootori poolt väljastatavad tulemused ilmuvad rohkem nähtaval kohal tulemuste nimekirjas.
AMD	***Asynchronous Module Definition*** *JavaScript*-i spetsifikatsioon, mis defineerib API koodi moodulite defineerimiseks ja nende laadimiseks, millal vaja.
API	***Application Programming Interface*** Funktsioonide, protseduuride, meetodite või klasside kogum, mis on kasutatud programmi poolt teenuste küsimiseks operatsioonsüsteemilt, tarkvara teegilt või mingilt teiselt teenuse tarnijalt, mis jookseb arvutil.
DRY	***Don’t Repeat Yourself*** Tarkvara arenduses – „ära korda ennast“ – on printsiip, mille eesmärgiks on vähendada igasuguse info kordusi, kasulik mitmekihilises arhitektuuris.

Jooniste nimekiri

Tabelite nimekiri

Sisukord

1. Sissejuhatus

Igast aastast areneb *HTML5* ja suureneb jõudlus kaasaegsetes brauserites. Arendajate poolt on loodud palju *JavaScript* raamistike ja teeke, et aidata arendajatel ehitada suuri ja rikkaid kliendipoolseid rakendusi. Need raamistikud ja teegid andsid arendajatele laia valiku tööriistadest, et arendada kuni ettevõtte keerukuseni kliendipoolseid rakendusi.

1.1 Taust ja probleem

Arenduses tuleb kokku puutuda otsusega, millist raamistiku valida, kui hakata arendama rakendust. Miks ei saa kasutada ühte raamistiku kui hõbekuulina ning miks on olemas internetis palju erinevaid lahendusi? Tuleb kokku puutuda otsusega, millist raamistiku valida. Alguses ei pruugi arendajad aru saada, millised halvad küljed on ühel või teisel raamistikul. Halvema valiku puhul hakkab arendaja „võitlema“ raamistikuga, kulutades palju aega ja ressursse.

Antud töös üritatakse vastata põhiküsimusele, kui hästi suudab valitud MVC raamistik täita põhiülesandeid võrreldes teiste *Model-View-Controller* (MVC) ja *Model-View-ViewModel* (MVVM) *JavaScript* raamistikkudega. Samuti, antud töö annab pildi, milliseid võimalusi ja piiranguid seob antud raamistik ettevõtte suurusega ja kindlate eesmärkide täitmiseks arendatav rakendus.

Töö tulemused saaksid tuua kasu nii füüsilistele, kui ka juriidilistele isikutele. Tuleb arvestada, et aeg on piiratud ressurss ning arendada rakendust võimalikult kiiresti ja tõhusalt, kuid valesti rakendatud tehnoloogia tooks vastupidist tulemust. Töös on võetud arvesse arendajate oskuste tasemed. Seega peab arendajatel olema kas piisav motivatsioon raamistiku õppimiseks või õppimise protsess peab olema kerge.

Rakenduste arendamisega veebikeskkonnas puudutakse nii ülikooli projektides, töötades vabakutselisena või firmades. Ülikooli projektides, kus kliendipoolse rakenduse raamistiku nõuded ei piirdu ühe raamistikuga, siis parem on valida raamistiku, millega oleks parem arendada. Vabakutseliselt või firmas töötades on vaja jälgida, et projekti suurus ei takistaks kasutajamugavust rakenduse laadimise ajaga.

1.2 Ülesande püstitus

Antud töö on kirjutatud eesmärgiga uurida, millised on *JavaScript*-i rakendusraamistike olulisemad omadused, milliseid omadusi tuleks põhjalikumalt uurida enne selliste raamistike kasutuselevõtmist. Uuritavateks ja hinnatavateks omadusteks on raamistiku õppimise keerukus, juurutamise keerukus, arendusega seotud kogukonna aktiivsus, raamistikku kasutava rakenduse komponendid, arhitektuur ja testimine.

Töös oodatavaks tulemuseks on omaduste loetelu, mille alusel oleks võimalik hinnata raamistike kasutatavust ja raamistiku juurutamisest tulenevaid eeliseid ja puudusi. Eesmärgiks on ka uurida ja leida võimalikke raamistike kasutamisega seotud probleeme, mida ei ole võimalik märgata raamistiku rakendamise algfaasis.

1.3 Metoodika

Kliendipoolset rakenduse osa ehitatakse mitu korda erinevate raamistikuidega ja/või teekidega. Töös uuritakse järgmiseid põhiaspekte:

1. **Komponentide kasutamine**. Komponendid võimaldavad juurutada teiste arendajate poolt valmis kirjutatud lahendused populaarsetele probleemidele, millega puudutakse pidevalt kokku. Tähtis on, et raamistik oleks pidevas toetuses arendajate poolt.

2. **Arhitektuur ja selle loetavus ja haldamise paindlikkus**. Rakendus peab olema hallatav ja selle kood loetav. Õigesti ehitatud arhitektuuriga saab võita aega rakenduste osade välja otsimises ning kasutada modulaarsust.

3. **Rakenduse kiirus.** Uuritakse, kui palju mälu vajavad raamistik ja rakendatud komponendid. Kuna tegemist on *Single Page Application* (SPA) rakendustega, siis esimene laadimine on esmatähtis. Samuti on uuritud, kui hästi saavad rakendused hakkama jõudlusega, kui lehele on vaja korraga laadida palju kasutajaliidesega seotud loogikat ja äriloogikat.

4. **Lihtsus arendamisel**. Enne raamistiku rakendamist tuleb õppida, kuidas see töötab ja milliseid põhimõtteid kasutab. Kui õppimiseks kulub palju aega, siis see on halb näitaja firmades, kui rakendus on arendatud kindla raamistikuga. Igale uuele arendajale tuleb kulutada aega väljaõpetamisele.

1.4 Ülevaade tööst

Alguses vaadeldakse, millised *JavaScript* raamistikud ja teegid on saanud suure populaarsuse kasvu. *JavaScript* raamistikud on valitud kasutades *Google Trends*, kus on olemas infot raamistikude huvi tundmise statistika inimeste poolt üle terve maailma. Kuna valik on lai, võrdluses kasutatakse antud töös nelja kõige populaarsemat MVC struktuuriga *JavaScript* raamistiku: *AngularJS*, *EmberJS*, *ReactJS* ja *BackboneJS*.

Järgmisena tutvustatakse need raamistikud ning näidatakse põhifunktsionaalsust, mis iseloomustab vastavaid raamistikke. Võrdluse ajal on võimalik selgeks teha, kui palju raamistikud teineteisest erinevad.

Pärast on tehtud neli rakendust, igaüks oma raamistikuga. Iga rakenduse juures kirjeldatakse kasutatud komponente ja seletatakse, miks on neid rakenduses vaja. Komponendid on samuti valitud komponentide populaarsuse ja toetuse järgi.

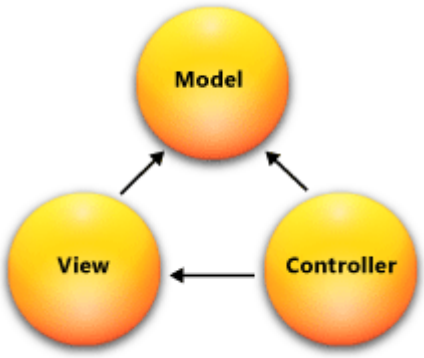
Lõpuks tehakse võrdlused kliendipoolsete rakenduste vahel ja tehakse järeldused. Järelduse andmeid on võimalik kasutada uute *JavaScript* rakenduste loomisel kas samade või teiste raamistikkudega.

2. JavaScript raamistikude valik

2.1 MVC ja MVVM struktuuridest

Mida rohkem loogikat peab brauser täitma, seda suuremaks peab *JavaScript*-i kasutajapoolne rakenduse osa kasvama. Rakenduse kasvuga tuleb rohkem mõelda rakenduse struktuurile, kuidas seda produktiivselt hallata. Üks viis rakenduse haldamisega seotud probleemide lahendamiseks on kasutada MVC ja/või MVVM struktuuriga teegid.

Objektorienteeritud programmeerimise areng, MVC on metoodika nimi või disain muster, mis seob edukalt ja tõhusalt kasutajaliidese loogikat andmed mudelitega. MVC mustrit kasutatakse laialdaselt programmide kirjutamisel. MVC mustrit on kuulutatud paljude arendajate poolt kui kasulikuks mustriks objektorienteeritud koodi taaskasutamiseks ja mustriks, mis võimaldab oluliselt vähendada aega, mis kuluks rakenduste arendamisele, kasutades kasutajaliideseid. [1]



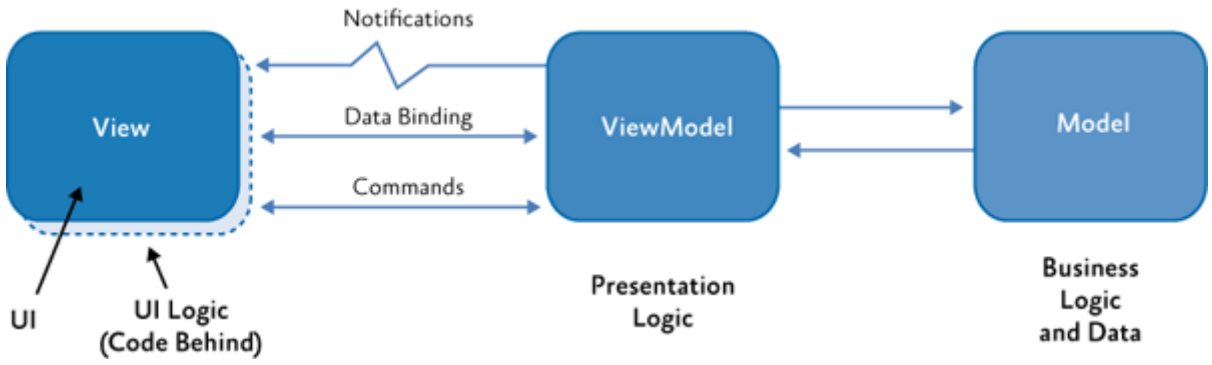
Joonis 1. Model-View-Controller [45]

MVC muster pakub kolme peamist komponenti või objekti, mida kasutatakse tarkvara arendamisel:

- Modell – esindab loogilist struktuuri andmetest tarkvara rakenduses. Vastav objekt ei kanna mitte mingisugust informatsiooni kasutajaliidese kohta [1]
- Vaade – klasside kogum, mis esindavad kasutajaliidese elemente. [1]
- Kontroller – esindab klasse, mis ühendavad modelli ja vaadet, ja kontrollerit on kasutatud kommunikatsiooni klassina modelli ja vaate vahel. [1]

MVVM muster aitab puhtalt eraldada rakenduse äriloogikat ja esindusloogikat kasutajaliidese loogikast. Puhta eralduse haldamine äriloogika ja kasutajaliidese vahel aitab hakkama saada mitme arenduse- ja disainitööga tekkivaid probleeme ning võimaldab rakendust kergemini

testida, hallata ja arendada. Samuti annab muster võimalusi koodi taaskasutamiseks ja lubab arendajatel ja kasutajaliidese disaineritel teha koostööd oma rakenduste osade arendamisel. [2]



Joonis 2. Model-View-ViewModel [2]

MVVM tulenes MVC mustrist ning mõlemad omavad palju ühiseid jooni. Mustri eesmärgiks on eraldada vaadete olekud ja käitumised. Sellisel viisil „vaade“ täidab ainult ühte ülesannet – info kuvamine – ja „vaade“ ei teagi, kes täidab äriloogikat.

2.2 Teegi ja sellele komponentide kättesaadavus

Antud töös on määratud eesmärk, et *JavaScript* teek

- peab olema avatud lähtekoodiga
- peab omama suurt kogukonda

Open Source software– avatud lähtekoodiga tarkvara, mida võib vabalt kasutada, muuta ja jagada (modifitseeritud või modifitseerimata kujul) kõikide poolt [3]. *Git* on tasuta ja avatud lähtekoodiga jaotatud versioonikontrollsüsteem, mis on arendatud eesmärgiga kiirelt ja efektiivselt hallata arendust nii väikeste, kui ka suurte projektide jaoks [4]. *Gihub* on veebipõhine *Git* repositooriumite hostimise teenus, mis pakub terve *Git*-i jaotatud versioonikontrollsüsteemi ja lähtekoodi haldamise funktsionaalsust ning lisaks ka enda poolt lisatud funktsionaalsust [4]. Teegi jaoks peavad olema kättesaadavad ka komponendid, mis peaksid ka olema avatud lähtekoodiga. Tasuta ja avatud lähtekoodiga hankimise allikaks on kasutatud maailma populaarseima veebiressursi - *Github*.

2.3 Kaasaegsed raamistikud

Antud töö kontekstis on võrdlusesse pandud kaasaegsed tehnoloogiad. Edaspidi on toodud nimekiri raamistikkudest, mis on kasutajate hulgast saavutanud 2015 aasta seisuga suure populaarsuse. Arvesse ei võeta raamistikud, mida arendajad enam ei toeta ning millel puudub MVC muster. Näiteks *jQuery*-t ei võetud võrdlusesse, sest sellel puudub kindel MVC muster.

Järgnevalt on kasutatud *Github* ressursi kontsekstis olevad mõisted „Reiting“ ja „*Fork*“. „Reiting“ tähendab inimeste arvu, kes on jätnud positiivse tagasiside hinnangu, vajutades „star“ nupule [6]. „*Fork*“ näitab arvu, kui palju inimest tegi projektist koopia eesmärgiga seda edasi arendada. *Fork* on repositooriumi kopeerimine, mis võimaldab eksperimenteerida muudatustega ilma originaalprojekti muutmata [7].

Tabel 1. Kaasaegsed JavaScript teegid ja raamistikud MVC ja/või MVVM struktuuriga [8]

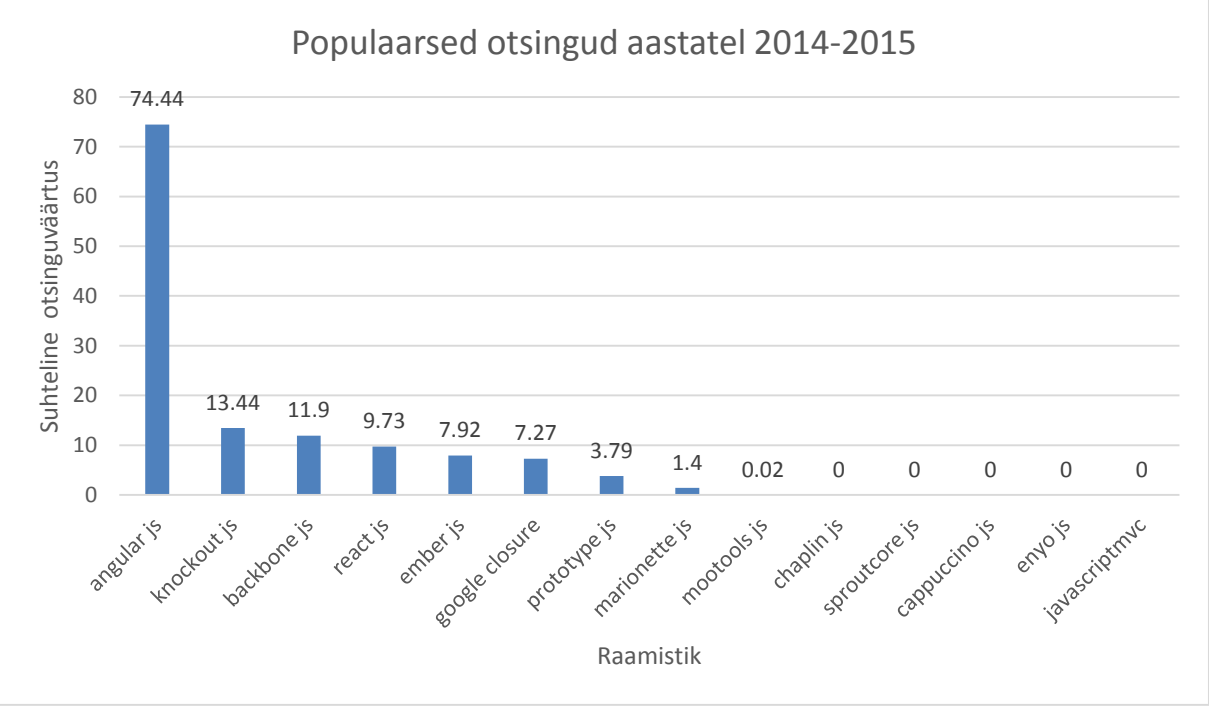
Nimi	**Liik**	**Reiting**	***Fork*-de arv**	**Litsents**
AngularJS	Raamistik	38105	15962	MIT
BackboneJS	Teek	21597	4814	MIT
ReactJS	Raamistik	21351	2984	BSD
EmberJS	Raamistik	13573	2903	MIT
KnockoutJS	Raamistik	6327	1085	MIT
BackboneJS MarionetteJS	Raamistik	6125	1160	MIT
Prototype JavaScript Framework	Raamistik	2976	496	MIT
ChaplinJS	Raamistik	2899	254	MIT
Echo	Raamistik	2294	245	Mozilla Public License
SproutCore	Raamistik	2117	306	MIT
MootoolsJS	Raamistik	2105	395	MIT
CappuccinoJS	Raamistik	2069	336	GNU LGPL
Enyo	Raamistik	1845	279	Apache 2.0

Nimi	Liik	Reiting	*Fork*-de arv	Litsents
Google Closure	Raamistik	850	270	Apache 2.0
JavaScriptMVC	Raamistik	575	128	MIT

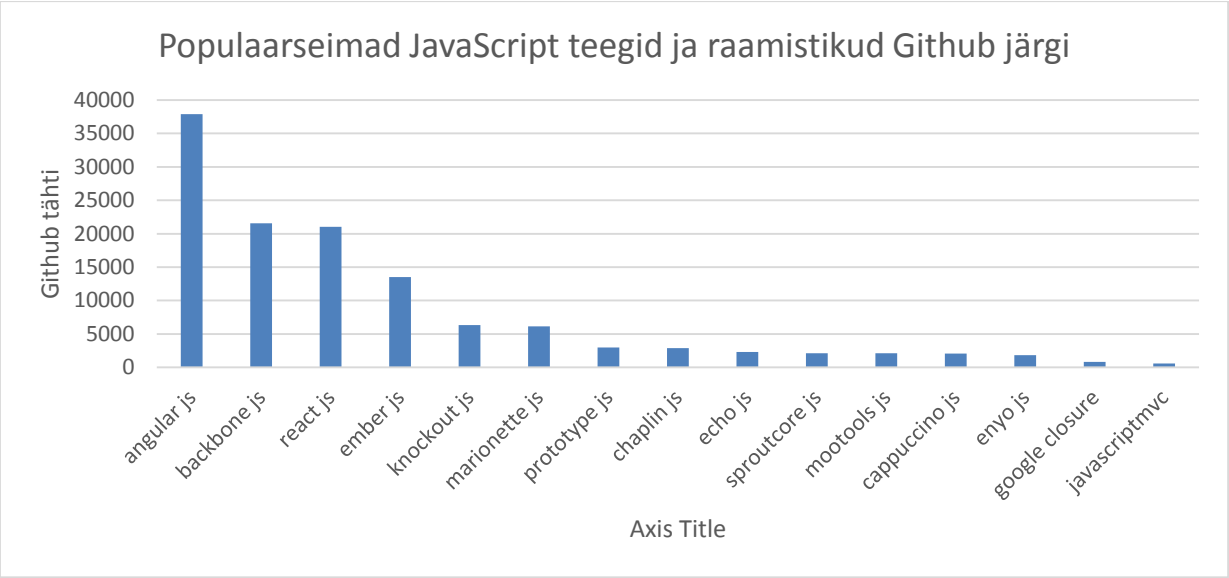
2.4 Võrdlus populaarsuse järgi.

Google Trends on arvuline ja ajalooline esitus Google poolt otsitud suhtelistest mahtudest [5]. Raamistiku populaarsuse kättesaamiseks on kasutatud *Google Trends*-i ja *Github*-i. *Google Trends* analüüsib protsenti *Google* veebiotsingutest välja selgitamiseks, mitu otsingut analüüsis sisestatud võtmesõna kohta võrreldes kõikide otsingu arvuga *Google* otsingumootoris selle aja jooksul [9]. Andmed on grupeeritud maailma regioonide ja keelte järgi. Otsingusõnaks on kasutatud raamistiku nime, siis tühik ja siis võtmesõna „*js*“, mis on lühend sõnast *JavaScript*.

Kõige populaarsem raamistik viie aasta jooksul on seni olnud *AngularJS*, mis järeldub *Google* otsingumootoriga otsitud arvu numbrist.



Joonis 3. JavaScript raamistikude ja teekide suhteline populaarsus Google Trends alusel aastatel 2014-2015



Joonis 4. JavaScript raamistikude ja teekide populaarsuste võrdlus Github reitingute järgi aastal 2015

Järgnevalt on valitud *JavaScript* raamistikud sügavamaks analüüsiks:

1. *AngularJS*
2. *EmberJS*
3. *ReactJS*
4. *BackboneJS*

Uuringu alla on *BackboneJS* asemel võimalik võtta *Marionette*, sest raamistiku arendajate järgi sisaldab kõiki samu *BackboneJS* omadusi ning samuti omab parema MVC struktuuri.

3. Võrreldavate raamistikute põhifunktsionaalsuse kirjeldus

3.1 Raamistik AngularJS

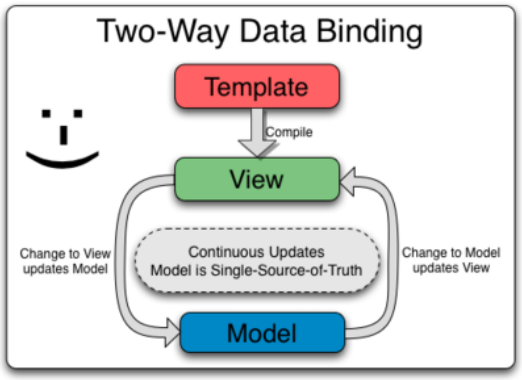
AngularJS on avatud lähtekoodiga veebiraamisik, mis on arendatud ja hooldatud *Google* ja arendajate ja firmade poolt eesmärkida leida lahendused SPA rakenduse arendamisel. *AngularJS* on arendatud *Google* tarkvara inseneride poolt. Esimene väljalase on toimunud aastal 2009 *Miško Hevery'i* ja *Adam Abrons'i* poolt. [10]

AngularJS raamistiku eesmärgiks on lihtsustada arendamist ja testimist, edastades MVC arhitektuuri. *AngularJS* eelistab deklaratiivset programmeerimist imperatiivsele. Deklaratiivne programmeerimine on kasutatud kasutajaliideste ehitamiseks. Imperatiivne programmeerimine on suunatud rakenduse äriloogika defineerimisele. [10]

Raamistik hakkas saama suure populaarsuse alates aastast 2011. SPA rakendus laadib kõik vajalikud komponendid ühe laadimisega ning järgnevalt suhtleb kliendipoolne rakenduse osa serveriga ainult asünkrooniliste päringutega.

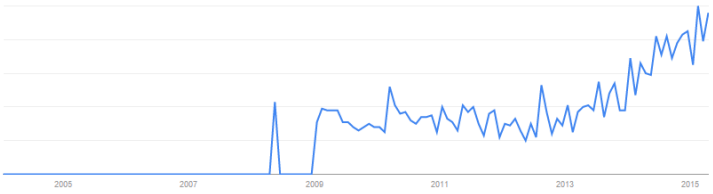
3.1.1 Kahesuunaline andmete sidumine

Kahesuunaline sidumine AngularJS rakendustes on andmete automaatne sünkroniseerimine modelli ja vaate komponentide vahel. Nii, nagu *AngularJS* implementeerib andmete sidumine, võimaldab arendajal käsitleda modelli kui ainuõigeks andmete allikaks rakenduses. Vaade on projektsioon modellist. Kui modell muutub, vaade peegeldab muudatuse, ja vastupidi. [11]



Joonis 5. Kahesuunaline sidumine [10]

Internetis hakati aktiivselt tegema *Google* otsingumootoris päringuid kahesuunalise sidumise kohta aastast 2009. Samal aastal tuli esimene *AngularJS*-i versioon.



Joonis 6. Kahesuunalise sidumise trend Google statistika järgi

3.1.2 Skoop

Skoop on objekt, mis viitab rakenduse modellile. Skoop on käivituse kontekst *expression*-te jaoks. Skoobid on paigutatud hierarhilises struktuuris, mis peegeldavad rakenduse DOM struktuuri. Skoobid võivad jälgida *expression*-e ja jälgida sündmusi. [11]

Skoobis muutuja „*test*“ initsialiseerimine *JavaScript* failis:

```
$scope.test = 'Test';
```

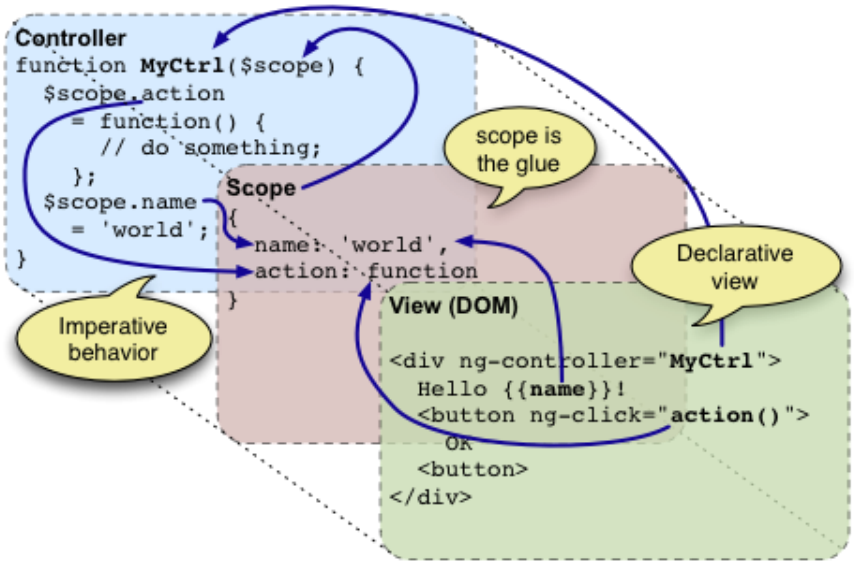
Sama muutuja *HTML* failis on kajastatud, kasutades *expression*-e:

```
{{test}}
```

3.1.3 Kontroller

AngularJS-s on kontroller *JavaScript*-i konstruktori funktsioon, mida kasutatakse *AngularJS*-i skoobi laiendamiseks. Kui kontroller on ühendatud DOM-ga, kasutades direktiivi *ng-controller*, *AngularJS* initsialiseerib uut kontroller-i objekti. Uus alamskoop on saadaval

injektsiooni parameetrina kontrolleri konstruktor funktsioonis *$scope* muutujana. Kontrollereid tuleb kasutada $scope muutuja initsialiseerimiseks ja $scope muutujale käitumiste lisamiseks. [14]



Joonis 7. AngularJS kontrolleri ja vaade seos skoobi muutuja kaudu [33]

Üheks silmapaistvaks erisuseks *AngularJS* kontrolleril on DOM manipuleerimise puudumine. Vastava strateegia eesmärgiks on kindla rakenduse struktuuri loetavuse säilitamine nii vaates, kui ka kontrolleris.

Kontroller *JavaScript*-s:

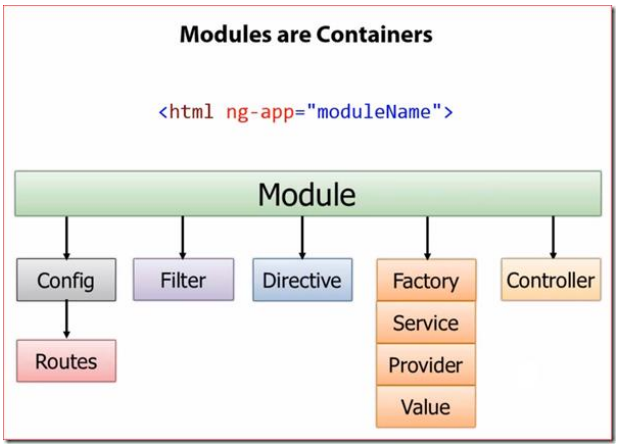
```
function MyCtrl($scope) {    …    }
angular.module('myApp', []).controller('MyCtrl', MyCtrl);
```

Kontroller HTML-s on deklareeritav kasutades *AngularJS* arendajate poolt defineeritud ng-controller direktiiviga:

```
<div ng-controller="MyCtrl">…</div>
```

3.1.4 Moodul

Moodul on konteiner erinevate rakenduste osade jaoks. *AngularJS* rakendustel määratakse deklaratiivsel viisil, kuidas AngularJS rakendust pannakse külge. Vastav lähenemine võimaldab parendada loetavust, taaskasutada mooduleid, laadida mooduleid igas järjekorras ja kergem testida. [15]



Joonis 8. AngularJS-i moodul [34]

Mooduleid on võimalik kapseldada. HTML-s peab *AngularJS*-ga seotud loogika asuma lubatud DOM *tag*-de vahel, millel atribuudiks on *ng-app='moduleName'*.

Moodul *JavaScript*-s:

```
angular.module('moduleName', []);
```

Moodul HTML-s:

```
<html ng-app="myApp">…</html>
```

3.1.5 Mall

AngularJS-s on mallid kirjutatud HTML-s ja kasutavad *AngularJS*-i spetsiifilist elemente ja atribuute. AngularJS kombineerib malli informatsiooniga modellist ja kontrollerist eesmärgiga kompileerida dünaamilist vaadet, mida lõppkasutajad näevad brauseris. Mallis saab kasutada kindlaid *AngularJS* elemente, nagu direktiive, *expression*-e, filtreid ja vormi komponente. [11]

Vaade on deklaratiivne HTML leht eesmärgiga olla lihtne ja loetav. Parima praktika järgselt ei sisalda vaated äriloogikaga seotud funktsionaalsust. Paindlikkus on võimaluses ühendada null või mitu vaadet ühe kontrolleriga. Vaade ei pea olema kontrolleriga ühendatud.

3.1.6 Teenus

AngularJS teenused on asendatavad objektid, mida seotakse kokku, kasutades *dependency injection*-t. Teenuseid on võimalik kasutada rakenduse loogika organiseerimiseks ja jagamiseks terve rakenduse ulatuses. AngularJS teenused, mis initsialiseeruvad laisalt ehk ainult siis, kui

neid on vaja kasutada. Samuti on nad *singleton*-id, kus teenusest iga sõltuv komponent viitab üksikule objektile, mis on genereeritud teenuste tehase poolt. [11]

Teenused kasutatakse näiteks andmebaasiga suhtlemiseks, kergete või keerukate algoritmite arvutamiseks, konstantsete või muutuvate andmete hoidmiseks ja vahetamiseks kindla mooduli piires, konfiguratsioonide hoidmiseks, mittetriviaalseks arvutuseks ja muu äriloogika jaoks. Teenuseid on viit liik: *constant*, *value*, *factory*, *service* ja *$provide*.

Teenuse initsialiseerimine *JavaScript*-s:

```
angular.module('myModule', []).service('serviceId', function() {});
```

3.1.7 Direktiivid

Direktiivid on DOM elementide markerid, nagu atribuut, elemendi nimi, kommentaar või CSS klass, mis suunavad *AngularJS*-s HTML kompileerijale käske ühendada kindlat käitumist selle DOM elemendiga või transformeerida DOM elementi ja selle alamelemente. *AngularJS* raamistikus on olemas mitu eeldefineeritud direktiivi, mis on eristatavad prefiksi „*ng*“ järgi, kuid samuti on olemas arendajal luua ja taaskasutada omad direktiivid. [18]

Direktiiv *JavaScript*-s:

```
angular.module('docsSimpleDirective', [])
.controller('myController', function() {}])
.directive('myDirective', function() {  return { template: 'Test' };});
```

Direktiiv HTML-s:

```
<div ng-controller="myController">
  <div my-directive></div>
</div>
```

3.1.8 Sõltuvuse injektsioon

Sõltuvuse injektsioon on tarkvaraarenduse muster, mis implementeerib tarkvarateekide inversiooni kontrolli. Sõltuvuse injektsioon võimaldab programmi disainil jälgida sõltuvuse inversiooni printsiibi, kus moodulid on nõrgalt seotud. Sõltuvuse injektsiooni kasutamisel ei ole vaja teada mooduli või teenuse loomise juures kõiki detaile ning vastavaid mooduleid ja teenuseid on kerge asendada ilma et peaks vaadet muuta. [11]

AngularJS-s on olemas kaks liiki sõltuvuse injektsioonist – selge (ingl. k. *explicit*) või kaudne (implicit). Kaudse injektsiooni puhul ei ole vaja täiendavalt määrata, millist moodulit või teenust juurutatakse.

```
angular.module('app').service('service1', function (service2) { … });
```

Siin juurutatakse teenust „*service2*“. Antud praktika miinuseks on, et koodi minimeerimisel see enam ei tööta. Seega parem on kasutada selget juurutamist:

```
angular.module('app').service('service1', ["service2", function (service2) { … }]);
```

või

```
angular.module('app').service('service1', service1);
service1.$inject = ["service2"];
function service1(service2) { … }
```

3.2 Raamistik EmberJS

EmberJS on avatud lähtekoodiga kliendipoolne MVC mustril põhinev struktuuriga veebirakenduste arendamiseks raamistik. *EmberJS* võimaldab ehitada suuri SPA rakendusi, kasutades kahesuunalist sidumist, eelarvutatud parameetrid, automaatselt uuendatavaid malle, kasutades teeki *HTMLBars*, ja oma ümbersuunamise loogikat. [12]

EmberJS peab omama ainult ühte *Ember.Application* objekti. Rakendus on defineeritav, kasutades muutujat *Ember.Application*:

```
window.App = Em.Application.create({ rootElement: '#myapp });
```

Järgnevalt on selles peatükis käsitletud muutujat *App* kui *EmberJS*-i defineeritud rakenduse objekt.

3.2.1 Mall

Mall nimega „*application*“ on vaikimisi seadistatud mall, mida kuvatakse *EmberJS* rakenduse käivitamisel. Peamallise pannakse kõik elemendid, mis hakkavad korduma terve rakenduse ulatuses. [12]

Peamall peab sisaldama *outlet* komponenti:

```
<div>{{outlet}}</div>
```

Outlet komponenti täidetakse ruuteri poolt edastatud sisuga vastavalt hetkel olevast URL-st. [12]

Vaadete kompileerimiseks on kasutatud teek nimega *HTMLBars*, mis on teegi *Handlebars* eelkäia, mis on omaette teegi *Moustache* eelkäia. *HTMLBars* on kindlate reeglitega defineeritud vaadete konverteerimise teek, mis laiendab HTML vaadete omadusi.

Vaadete mallid on võimalik defineerida järgnevalt:

- Malle on võimalik panna eraldi failidesse, mille laienditeks on „*hbs*“
- Mallide kood on võimalik lisada ühele lehele ning panna igat vaadet „script*“ tag*-sse. Vastavad „script“ *tag*-d peavad sellisel juhul omama kahte atribuuti – *type =„text/x-handlebars“* ja *data-template-name=“malli_nimi“*.

3.2.2 Ruuter

Ruuter suhtleb pidevalt rakendusega, liikudes ühest olekust teisse. *EmberJS* annab tööriistu, mis võimaldavad hallata olekuid, mis võimaldaksid teha koostööd suurete rakendustega. [12]

Kui rakendus käivitub, ruuter vastutab mallide kuvamise, andmete laadimise ja rakenduse olekute seadistamise eest. *EmberJS*-i ruuter teeb seda võrreldes hetkel olevat URL-i ja eeldefineeritud URL-dega. [24]

Ruuteri URL-de seoseid defineeritakse „*map*“ meetodiga:

```
App.Router.map(function () {
    this.route("home", { path: "/" });
    this.route("about");
});
```

Kui külastada URL-i „*./about*“, siis EmberJS otsib malli nimega „*about.hbs*“ ja kuvab kasutajale. [12]

3.2.3 Kontroller

EmberJS-s saavad mallid andmeid kontrolleritest, mis omaette dekoreerivad modelle. See tähendab, et mallid on teadlikud kontrolleritest ja kontrollerid on teadlikud modellidest, aga mitte vastupidi. Kontroller on defineeritud, kasutades *EmberJS*-i *Controller* objekti laiendamise teel [12].

```
App.MyController = Ember.Controller.extend({ … })
```

3.2.4 Mudel

EmberJS-s on igal *route*-l olemas oma assotsiatsioon modelliga. Lihtsamate rakenduste puhul on võimalik kasutada näiteks *jQuery*-t JSON andmete laadimiseks serverist ning kasutada

JSON objekte modellidena. Kuigi parem oleks kasutada modellide haldamiseks kindlat teeki, mis võimaldaks otsida, muuta ja salvestada modelle. Selleks on võimalik kasutada *Ember Data* teeki, mis integreerub *EmberJS*-sse ja võimaldab ilma konfliktideta teha koostööd. Modellide defineerimiseks on võimalik kasutada *Ember Data* teegist tulenevat *DS*-nimelist nimeruumi. Vastav moodul defineerib andmetega töötamiseks vajalikku *JavaScript* objekti [12].

```
DS.Model.extend({ … });
```

Üldiselt on modellidel olemas parameetrid, mida kasutatakse serverile salvestamiseks, samal ajal kui kontrolleris olevaid parameetre ei salvestata serverile. [12]

3.2.5 Komponent

EmberJS komponendid on kapseldatud rakenduse osa, mida on võimalik taaskasutada, säilitades *Don't Repeat Yourself* (DRY) printsiipi. Komponentide defineerimiseks tuleb luua mall, mis asuks kaustas nimega „components" ning selle sees luua komponendi mall. Faili nimi mängib olulist rolli ning peab olemas sama, mis komponendi nimi. Komponendi nimes peab olema sidekriips. [12]

Näiteks komponendi „*blog-post*" loomiseks on vaja luua fail *„/templates/components/blog-post.hbs*":

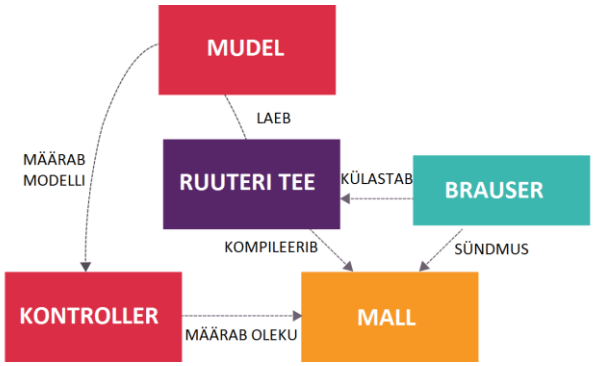
```
<article class="blog-post">
  <h1>Komponendi sisu</h1>
</article>
```

Järgnevalt on vaja defineerida *JavaScript*-s sama komponent, luues faili *„/components/blog-post.js*":

```
Ember.Component.extend({ … });
```

3.2.6 *EmberJS*-i elutsükkel

EmberJS omab vastava elutsükli mudeli:



Joonis 9. EmberJS rakenduse elutsükkel

3.3 Raamistik ReactJS

ReactJS on *Facebook*-i ja *Instagram*-i poolt arendatud avatud lähtekoodiga teek eesmärgiga kergendada interaktiivsete, olekutega, taaskasutatavate kasutajaliidese komponentide loomist. ReactJS esindab V-tähte lühendis MVC, kus V tähendab vaadet. *ReactJS*-i on võimalik teiste teekide ja raamistikudega koos kasutada. *ReactJS* arendajad on loonud raamistiku ülesandega lahendada probleemi, mis on seotud suurete rakenduste ehitamiseks, kus andmed muutuvad pidevalt. [13]

3.3.1 Virtuaalse DOM-i terminoloogia

ReactJS-s on olemas viis terminoloogiat, mida on tähtis omavahel eristada:

1. **ReactElement**. *ReactJS*-i element on primaarne tüüp, millel puuduvad igasugused meetodid ja prototüübid. *ReactJS* elementidest koosneva elemendi kuvamiseks *HTML* DOM-s tuleb vastavat elementi edastada „*React.render*“ meetodile. [14]

2. **ReactElement factory**. *ReactJS* elementide tehas on funktsioon, mis genereerib kindla tüübiga *ReactJS* elemente. *ReactJS* raamistikul on olemas hulk eeldefineeritud tehaseid tihti kasutatavate HTML elementide jaoks. JSX süntaksi kirjutamisel ei ole vajadust tehaseid kasutada. [14]

3. **ReactNode**. *ReactJS*-i sõlm võib olla kas *ReactJS* element, string, number, või ReactJS sõlmede massiiv. [14]

4. **ReactComponent**. *ReactJS* komponendid on kapseldatud varjatud olekutega JavaScript klassid. [14]

5. **ReactComponent factory**. *ReactJS*-i komponent on modulaarne taaskasutatav rakenduse osa. Igal komponendil on olemas oma olek, mis hoiab komponendi kontekstis olevaid väärtusi. Samuti atribuutide kaudu edastatavad väärtused on kättesaadavad komponendi skoobis abiobjekti „*props*“ kaudu. *ReactJS*-i komponendi oleku muutmisel kasutatakse „*setState*“ meetodid, mis käivitab uuendamise kasutajaliideses. [17]

Virtuaalse DOM-i kirjutamiseks on võimalik kasutada tavalist *JavaScript* või JSX süntaksit. JSX on *JavaScript*-i XML süntaksi transformeerimine. JXS-ga saab kirjutada HTML-ile sarnase koodi *JavaScript*-s, aga tuleb pöörata tähelepanu kindlatele komponentidele, et „*class*“ asemel tuleb kirjutada „*className*“ ja „*for*“ asemel „*htmlFor*“. [15]

Järgnevalt on demonstreeritud, kuidas näeb välja *ReactJS*-i DOM:

```
var HelloMessage = React.createClass({
  render: function() {
    return <div>Hello {this.props.name}</div>;
  }
});
React.render(<HelloMessage name="John" />, mountNode);
```

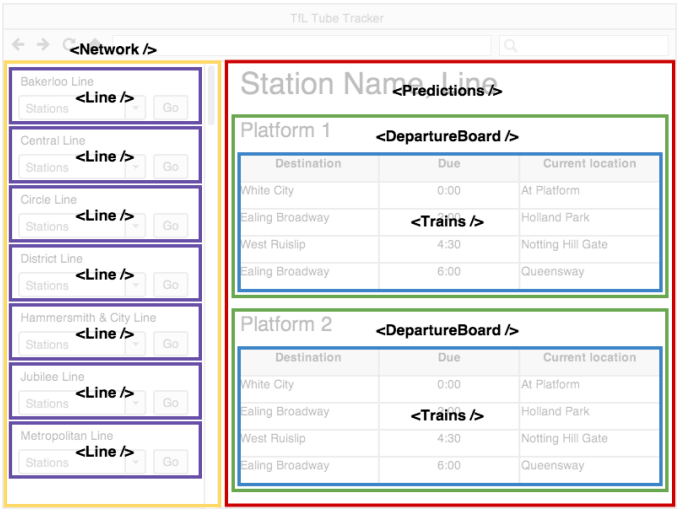
ReactJS DOM-i transkompileerimisel saab eelnev kood järgmise kuju:

```
var HelloMessage = React.createClass({displayName: "HelloMessage",
  render: function() {
    return React.createElement("div", null, "Hello ", this.props.name);
  }
});
React.render(React.createElement(HelloMessage, {name: "John"}), mountNode);
```

Seega kui on vaja kirjutada *ReactJS DOM* elemendi, nagu „*div*“ *tag*, või mingi teise elemendi asemel puhta *JavaScript* kujulise elemendi, siis see käib kasutades funktsiooni

```
React.createElement(„div“, null, „sisu“)
```

Järgnevalt on näide, kuidas vaadet on võimalik ReactJS komponentidest kokku panna:



Joonis 10. ReactJS komponentidest koosnev vaade. [36]

3.3.2 Elutsükli meetodid

ReactJS-i komponendid kasutavad elutsükli meetodeid. Elutsükli meetod on funktsioon, mis käivitub komponendi kindlaksmääratud eluhetkel [18]. Meetodid on järgmised:

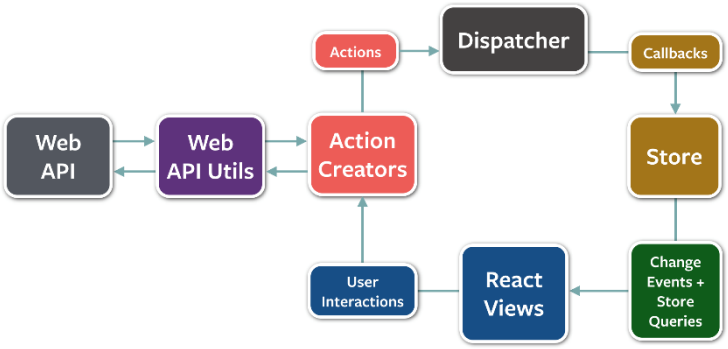
- *componentWillMount* – käivitub üks kord enne kompileerimist, nii kliendi kui ka serveri poolel [18].
- *componentDidMount* – käivitub üks kord ainult kliendi pool pärast kompileerimist [18].
- *shouldComponentUpdate* – tagastatav väärtus määrab, kas komponenti on vaja uuendada [8].
- *componentWillUnmount* – käivitub üks kord enne komponendi lahutamist rakendusest [18].

Lisaks elutsükli meetoditele on olemas järgmised meetodid:

- *getInitialState* – tagastab vaikimis oleva oleku [18].
- *getDefaultProps* – tagastab vaikimisi seadistatud muutuja „*props*“, kui see pole edastatud [18].
- *mixins* – objektide massiiv, mida on kasutatud komponendi funktsionaalsuse laiendamiseks [18].
- *render* – meetod, kus defineeritakse väljastatava *ReactJS DOM*-i osa ning mida käivitatakse komponendi igal uuendusel [18].

3.3.3 Rakenduse struktureerimisteek *Flux*

Raamistiku kindla struktuuri defineerimiseks on kasutatud „*Flux*“-nimeline raamistik, mis on arendatud Facebook arendajate poolt. „Flux“ aitab hallata suuri JavaScript rakendusi ja nende keerukust [19]. *Flux* teek sisaldab endas parimate praktikate kogumit rakenduse struktureerimises, kasutab MVC-st mõnevõrra teise lähenemise ning struktuur on järgmine:



Joonis 11. Flux rakenduse struktuur [19]

Põhikomponendiks on siin *View*, *Store*, *Dispatcher* ja *Action* objektid.

- *Dispatcher* – tervikprotsessi haldur, mis võtab vastu sündmused (*Action*) ning edastab sündmusi ja andmeid registreeritud *callback*-desse [19].
- *Store* – haldab rakenduse olekuid kindla domeeni jaoks rakenduse piires. *Store*-s tuleb hoida andmeid, andmete töötlemise meetodeid, rakenduse olekuid ja dispetšeri callback meetodeid [19].
- *Actions* – tegevuste ehk sündmuste defineerimiseks kasutatav objekt, mis võib sisaldada mitu alamtegevust. Tegevused kutsutakse välja vaadetes ning suunatakse edasi *Dispatcher*-sse. Sündmuste eristamiseks on kasutatud konstante ning sellega seoses tekib vajadus konstantide hoidmiseks objekti tekitamiseks ja importimiseks [19].
- *View* – *ReactJS*-i komponendid, mis ootavad muudatusi sündmuste käivitamiseks ja võtavad rakenduse olekuid *Store*-st. Kõik andmed on edastatavad alamkomponentidele „*props*“ kaudu [19].

Callback on käivitatav kooditükk, mida edastatakse teisele koodile argumendina, millest on oodatud vastava kooditükki käivitamist teisele koodile mugaval ajal [20].

Tegevuste defineerimiseks kasutatakse objekti *Action*, mis võib sisaldada mitu alamtegevust. *Action*-it võib käsitleda kui sündmusi, mida käivitab vaade. Kõiki tegevusi registreeritakse *Dispatcher*-is. *Dispatcher* muudab kõiki *Action*-ist tulnud sündmusi *promise*-teks, järjestatakse ning töödeldakse igaühte. Sündmuste järjekorras hoidmine eemaldab sündmuste omavahelise konflikti ohtu. *Dispatcher* saadab kõiki *callback*-e *Store*-isse käivitamiseks. *Store*-des hoitakse andmeid. *Store* näeb välja, nagu *AngularJS*-i kontroller, kuid tegelikkuses käitub nagu *AngularJS*-i teenus. [19]

Rakenduse struktuur omab järgmise kuju:

```
app/
    actions/
    components/
    constants/
    dispatcher/
    stores/
    app.js
```

3.4 Raamistik BackboneJS

BackboneJS on kergekaaluline *JavaScript* teek, mis lisab kindla struktuuri rakendusele. *BackboneJS* kergendab koodi haldamist võttes arvesse, et rakenduse suurus võib kergelt kasvada. Teek on kasutatud SPA rakenduste ehitamiseks. Teegi eesmärg on anda arendajale abifunktsionaalsust andmete ja päringute manipuleerimiseks, pigem kui *JavaScript* objekti haldusloogikale jalgratta leiutamine. Suurimaks tunnusjooneks *BackboneJS*-l on peetud tema suurus, mis võimaldab käivitada veebilehti nõrga ühendustega võrkudes. [21]

BackboneJS-l on olemas kindlad omadused ja tunnusjooned:

- Eesmärgiks on anda arendajale hulk põhilist abifunktsionaalsust, millega puudutakse arenduses kokku igapäevaselt.
- *BackboneJS* ei nõua, et rakendus peaks olema SPA struktuuriga, vaid on samuti optimeeritud terve lehe ümberlaadimisega.
- Kasutatud on imperatiivne viis kliendipoolse rakenduse loogika kirjutamiseks.

- *BackboneJS* töötab hästi nii väikeste, kui ka suurte rakendustega.
- *BackboneJS* annab valiku, kas arendaja saab kasutada kahte-suunalist sidumist või mitte.

Samuti soovitavad teegi arendajad kasutada *BackboneJS* objekte koos teegiga *underscore.js*. See annab juurde funktsionaalsust andmete mugavamaks manipuleerimiseks. [21]

3.4.1 Mudel ja kollektsioon

Andmete esindamiseks ja haldamiseks on kasutatud *BackboneJS* mudelid. Mudeleid saab luua, valideerida, ära hävitada ning salvestada serverile. Mudelite andmete manipuleerimisloogika on jagatud järgmisteks suurteks osadeks: konventsioonid, validaatorid, arvutatud omadused ja juurdepääsu kontroll. [21]

Iga uus mudel on loodav järgmiselt:

```
var myModel = Backbone.Model.extend({ … });
```

Kollektsioonid on sorteeritud mudelite hulgad. Nende abil on kergem manipuleerida suured andmehulgad, et saavutada kergemat sorteerimist, filtreerimist, otsimist, grupeerimist ja teisi tegevusi. Paindlikkust andmehulkade töötlemiseks lisab abistav teek *underscore.js*. [21]

3.4.2 Ruuter

Arendajatel on tihti vajadus jagada omavahel URL linke tähtsatele asukohtadele rakenduses. *Backbone.Router*-i komponent võimaldab navigeerimist kliendipoolses rakenduses ning ümbersuunamisloogikat on võimalik kokku ühendada tegevustega ja sündmustega. Uutel brauseritel kasutab *Backbone.Router History* API-t ning vanematel brauseritel, mis ei toeta *History* API-t, on olemas alternatiivne töötav funktsionaalne komponent navigeerimise tagamiseks. [21]

Rakenduse laadimise alguses tuleb aktiveerida ümbersuunamisfunktsionaalsust järgmise käsuga:

```
Backbone.history.start();
```

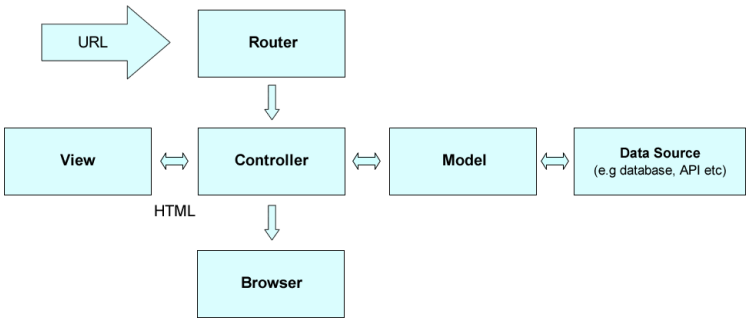
3.4.3 Sündmus

Sündmus on komponent, mida saab kokku sulatada iga teise objektiga, mis annab vastavale objektile siduda ja käivitada arendajate poolt nimetatud sündmusi. Sündmused ei pea olema deklareeritud enne, kui nad on seotud kindlate objektidega. [21]

Sündmused on laiendatavad abiteegi *underscore.js* abil.

3.4.4 Rakenduse struktuur

BackboneJS rakenduse elutsükkel on järgmise kujuga:



Joonis 12. BackboneJS rakenduse elutsükkel [21]

Optimaalsema rakenduse *MVC* struktuuri on võimalik defineerida, kasutades raamistiku Marionette. *Marionette*-ga on rakenduse loogikat võimalik struktureerida.

Tabel 2. Marionette struktureerimise elementide definitsioonid [22]

Funktsionaalsuse nimetus	**Kirjeldus**
Application	Konteiner terve rakenduse jaoks, mille hulk peab olema vähemalt üks. Sisaldab rakenduse alguskohta „start“, mis käivitab tervet rakendust.
AppRouter	Ümbersuunamiste konfigureerimise abimeetod, mis võimaldab käsitleda ümbersuunamiste sündmusi.
Behaviour	Isoleeritud DOM elementide kogum, mida saab kokku segada vaadetega eesmärgiga teha vaated loetavamaks ja esinduslikumaks. Üritatakse säilitada DRY printsiip.
CollectionView	Komponent, mis võimaldab itereerida moodulite massiivi, mis on defineeritud kindlas kollektsioonis ning väljastada igaüht alamvaatele.
CompositeView	Komponent, mis laiendab CollectionView-d eesmärgiga esindada vaated puustruktuurina.
Functions	Komponent, mis varustab arendajat abifunktsioonidega, mida on kasutatud üldiste tegevuste lihtsustamiseks terve raamistiku raames.

Funktsionaalsuse nimetus	Kirjeldus
ItemView	Komponent, mis esindab vaadet üksiku elemendi jaoks. Elemendiks võib olla nii *Backbone.Model* või *Backbone.Collection* ning on käsitletud üksikuna elemendina.
LayoutView	Komponentide *Region* objektide kollektsiooni ja *ItemView* hübriid, mis on ideaalsed rakenduste *layout*-de kompileerimisega, mis koosnevad mitmetest alamregioonidest, mis on hallatavad kindlaksmääratud regioonhalduri poolt.
Module	Komponent, mis võimaldab luua modulaarselt kapseldatud loogikat. See võimaldab suured failid luua mitmeteks väikesteks failideks ning luua individuaalseid komponente rakenduste jaoks.
Object	Baasklass, millest laiendavad kõik teised klassid. *Object* sisaldab mitu *Backbone* kokkuleppeid ja kasutab meetodeid, nagu „*initialize*“ ja *Backbone.Events*.
Region	Komponent, mis varustab arendajat järjekindlate meetoditega vaadete haldamiseks, kuvamiseks ja kustutamiseks oma rakenduses ja *layout*-des. Vaated kasutavad jQuery teeki nende kuvamiseks õiges kohas.
RegionManager	Komponent, mis varustab arendajat järjekindlate meetoditega regioonide haldamiseks rakenduse piires. *RegionManager* on mõeldud kasutamiseks teiste objektide poolt, et lihtsustada lisamist, salvestamist, objektide kättesaamist ja nende kustutamist.
Renderer	Komponent, mis on väljavõetav *ItemView* kompileerimisprotsessist eesmärgiga luua järjekindlat ja taaskasutatavat meetodit mallide kompileerimiseks andmetega või ilma.
TemplateCache	Komponent, mis varustab arendajat vahemäluga mallide kättesaamiseks skriptide blokkidest HTML-s.
View	Komponent, mis on kasutatud vaadete defineerimiseks, kus vaated on kasutatud kindla lehesisu kuvamiseks.

4. Tarkvara realisatsioon

Tarkvara realisatsioon on jagatud serveri- ja kliendipoolseks osaks. Antud töös on realiseeritud üks serveripoolne rakenduse osa ja mitu kliendipoolset rakenduse osa, igaüks erineva tehnoloogiaga, aga sarnase tulemusega.

4.1 Komponendid

4.1.1 Taustinfo

Arendamisel on vaja lahendada sarnaseid probleeme, millega puutuvad arendajad üle terve maailma iga päev. Probleemide lahendamisel tehakse valmis rakenduse osad ja antakse ühisesse kasutusse teistele arendajatele oma rakenduses vabaks kasutamiseks ja/või selle edasiseks arendamiseks. Niipea kui arenduses tekib vajadus arendada kindlat funktsionaalsust, tuleks alguses otsida internetis, kas keegi on juba sarnast funktsionaalsust valmis teinud. Kui sarnase funktsionaalsusega komponent on olemas ja sellel on palju häid hinnanguid, siis komponenti võib lugeda usaldusväärseks oma projekti juurutamiseks.

Iga komponendi suurus on tähtis. Rakendust võib lugeda suureks, kui selle üldine suurus ületab 1 MB. Alati tuleb minimeerida kõiki kasutatavaid komponente, enne kui laadida rakendust serverile. Komponentide minimeerimine arenduskeskkonnas ei ole aga mõistlik, kuna võib aeglustada arendusprotsessi sellega, et vigade olemasolul ei kuvata vigu loetaval kujul.

4.1.2 Ühised komponendid

Igal raamistikul peaksid olemas olema põhikomponendid, mis on arendajate poolt laialt kasutatud. Iga komponent on võetud *Github* ressursist. Komponenti on võimalik hinnata järgmiste parameetrite järgi:

- **Tähtede** (ing. k. *star*) **arv**. Peegeldab, kui paljud inimesed on jätnud positiivse tagasiside. Mida suurem arv, seda parem.
- ***Fork*-ide arv**. Peegeldab teegi aktiivsust. Mida suurem arv, seda parem. Näitab, kui palju inimesi on valmis teeki toetama.

Vastavate komponentide liikide loetelu, mida on kasutatud näidisrakenduste loomise juures:

- **Tabeli komponent**. Komponent peab võimaldama andmeid sorteerida ja filtreerida. Samuti on eeldatud, et tabelid suudavad mahutada andmeid optimeeritumal viisil, et joonistuste arv brauseris ei oleks ilma vajaduseta suur.

- **Graafikute komponent**. Graafikud on vajalikud andmete visualiseerimiseks. Graafikute on võrreldes teiste komponentidega kõige suurem joonistamistsüklite arv. Graafikute komponendid baseeruvad teegile d3, mis võimaldab kasutada standardiseeritud tehnoloogiaid *SVG*, *HTML* ja *CSS* eesmärkiga toetada võimalikult suurim hulk brausereid.

- **Kalendri komponent**. Kalender on sündmuste kuvamiseks, grupeerimiseks ja haldamiseks vajalik komponent.

- **Rakenduse struktureerimise abikomponent.** Rakendus võib kasvada suureks ning võivad tekkida raskused selle haldamisel, kui kasutada raamistiku vaikimisi struktureerimisreegleid. Rakenduse struktureerimise soodustamiseks, kui võimalik, tuleks kasutada rakenduse struktureerimise abikomponendi.

- **Abikomponent.** Komponent, mida on vaja rakendusse integreerida, kus põhjuseks võib olla

    - Sõltuvus teisest komponendist

    - Võrreldaval raamistikul realiseeritud rakenduse funktsionaalsustaseme saavutamiseks

Komponentide alla laadimiseks ja installimiseks on kasutatud paketihaldureid *NPM* ja *Bower*, kus antud töö kontekstis on primaarsem eelistus *Bower*-le. Mõlemad paketihaldurid töötavad sarnasel viisil. Paketihaldur töötab selliselt, et laeb internetist ja installib raamistiku, teeki vm faili [24]. *NPM* kasutab „*package.json*“ ja *Bower* „*bower.json*“ manifest faile, mis sisaldavad rakenduses kasutatavate pakettide loetelu. Käsuga „*npm install*“ ja „*bower install*“ otsivad paketihaldurid manifesti failist kõiki seal nimetatud pakette ja tõmbavad ning installivad neid rakenduse kataloogi.

4.2 Raamistikude ühiktestimisel kasutatavad raamistikud

Testimiseks on võimalik kasutada populaarseid testimisteeke. Järgnevalt tehakse ülevaatlik hinnang testimise raamistikkudele.

Tabel 3. JavaScript-i testimisraamistikud

Nimetus	**Reiting**	***Fork*-de arv**	**Suurus (KB)**	**Suurus minimeerituna (KB)**
Jasmine	8704	1384	84	42
Mocha	6522	1033	142	59

Nimetus	Reiting	*Fork*-de arv	Suurus (KB)	Suurus minimeerituna (KB)
Karma	5078	832	1	1
QUnit	3156	646	114	40
Intern	2772	191	2	1
Sinon	2005	321	100	38
TestSwarm	959	162	9	3
Buster	377	61	19	8

Ühiktestimine. Kõige populaarsem on *Github* hinnangute järgi teek *Jasmine*.

End2End testimine. Tekib vajadus kasutajaliidese interaktsiooni testimiseks ning see protsess peab olema automatiseeritav. *End2End* testimiseks on võimalik kasutada *Protractor*-it, mida soovitavad *AngularJS* arendajad kõige rohkem. *Protractor* kasutab *Selenium* veebidraiverit kasutajaliidese testimiseks [25].

4.3 Kliendipoolne realisatsioon kasutades *AngularJS*

4.3.1 Komponendid

AngularJS komponentide juurutamiseks ei ole vaja midagi muud kui *AngularJS*-i, välja arvatud komponentide otseselt nõutud sõltuvusteeke.

Tabel 4. AngularJS-i abikomponendid

Nimetus	Komponendi liik	Reiting	Fork-de arv	Kirjeldus
ng-grid	tabel	2467	1278	Andmete kuvamine kasutajasõbralikus formaadis võimalustega andmeid sorteerida ja filtreerida.
ui-calendar	kalender	712	313	Andmete ja sündmuste kuvamiseks kasutatav kalendrikomponent
angular-google-maps	kaardid	1617	648	Google kaartidel põhinev kaartide komponent asukohtade visualiseerimiseks

Nimetus	Komponendi liik	Reiting	Fork-de arv	Kirjeldus
angular-charts	graafikud	862	222	D3 teegil põhinevad graafiku komponendid andmete visualiseerimiseks
angular-ui-bootstrap	abikomponendid	8689	4567	Vajavaminevate abikomponentide kogum

Kõikide komponentide hankimiseks on võimalik kasutada paketihaldurit *Bower* kasutades käsku

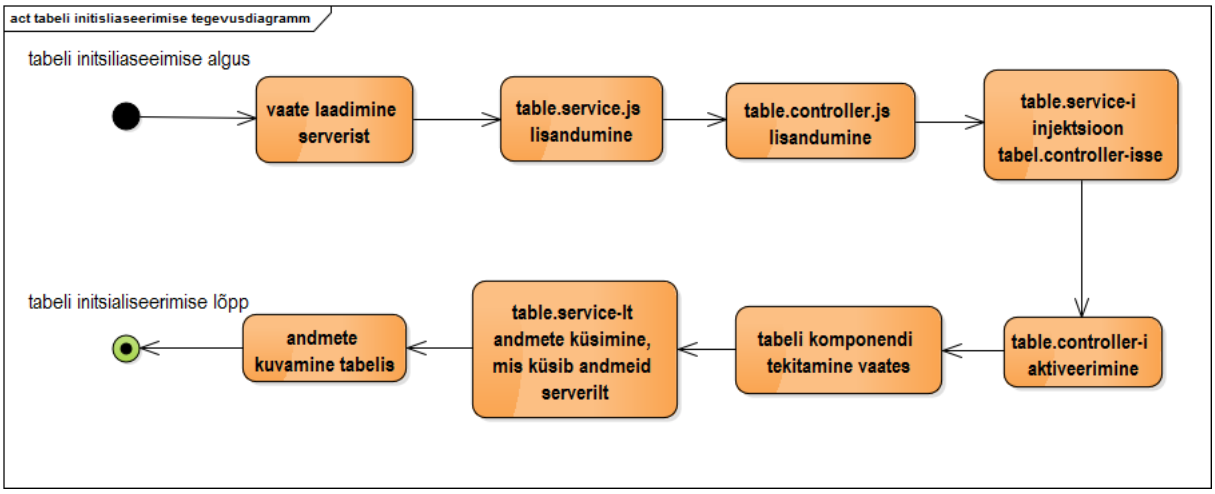
```
bower install --save ng-grid ng-table angular-charts angular-ui-calendar angular-google-maps angular-bootstrap
```

4.3.2 Rakenduse struktuur

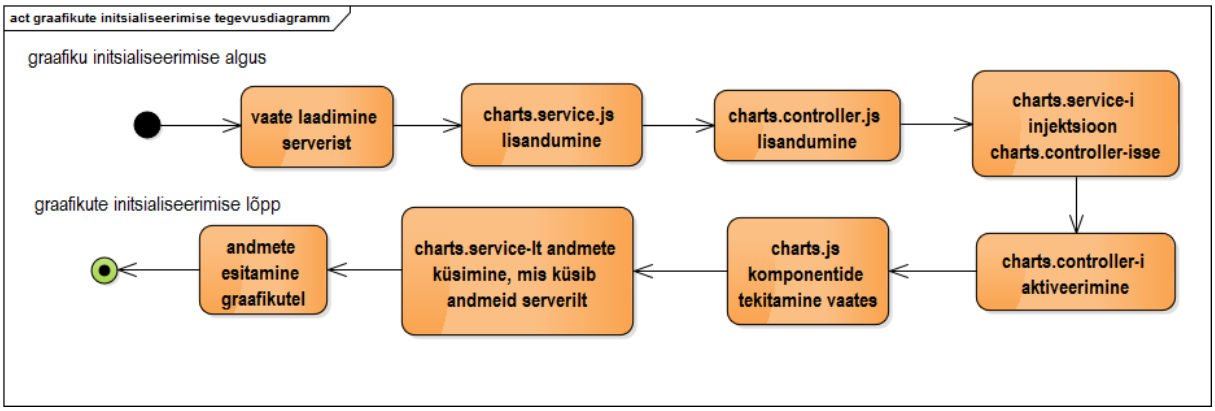
Antud töös on rakenduse suurus keskmine ning vastavalt sellele tehakse ka optimaalne struktuur.

```
angular/
    constants/
        app.constants.js
    controllers/
        calendar.controller.js
        charts.controller.js
        performance.controller.js
        table.controller.js
    services/
        calendar.service.js
    views/
        main.html
        calendar.html
        charts.html
        index.html
        performance.html
        table.html
    app.config.js
    app.js
```

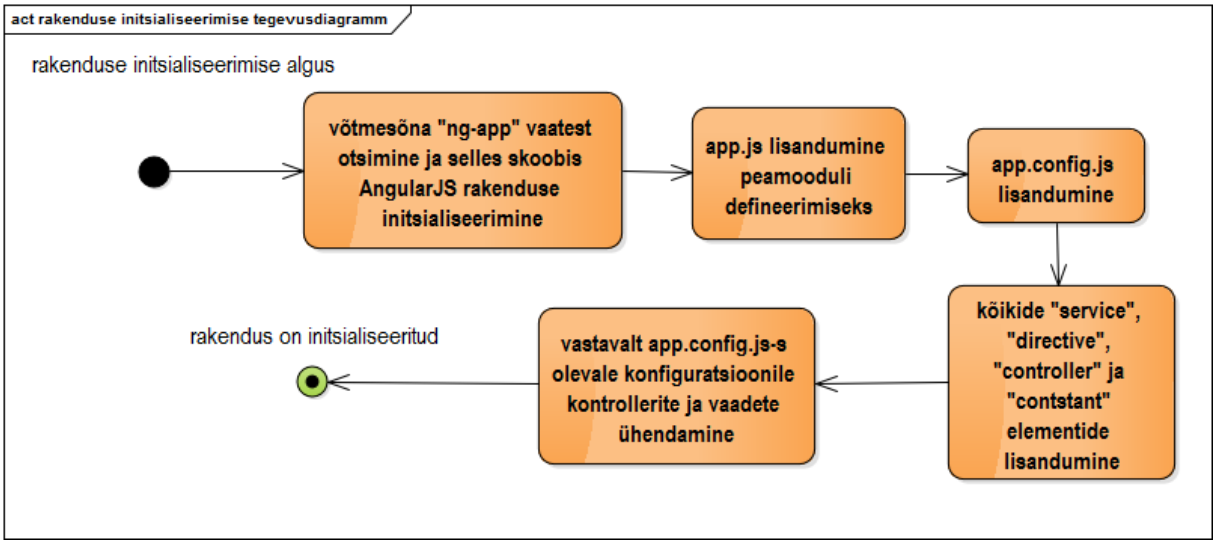
4.3.3 Diagrammid



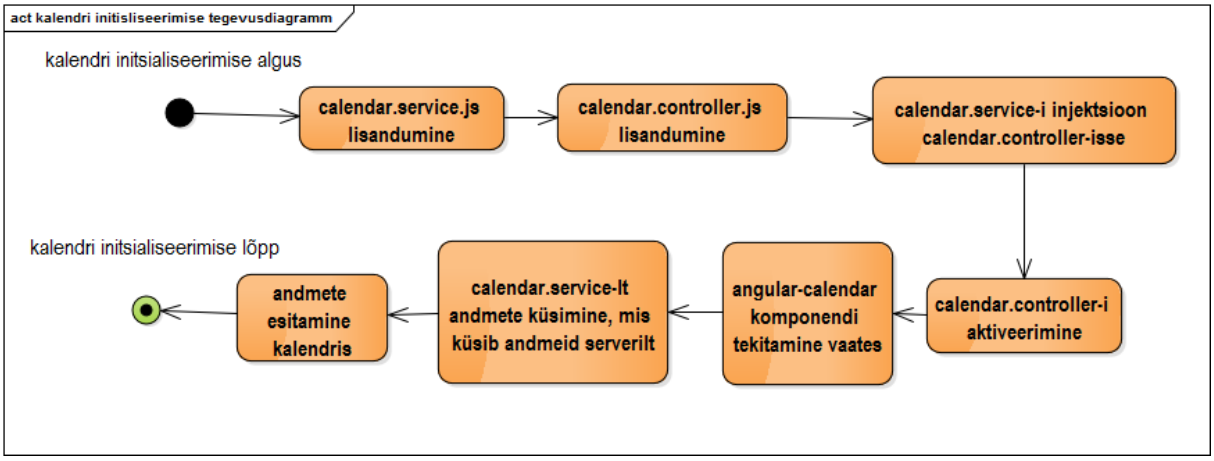
Joonis 13. Rakenduse initsialiseerimine



Joonis 14. Tabeli initsialiseerimine



Joonis 15. Graafikute initsialiseerimine



Joonis 16. Kalendri initsialiseerimine

4.4 Kliendipoolne realisatsioon kasutades EmberJS

EmberJS juurutamiseks on vaja rahuldada järgnevad sõltuvused: *jQuery*, *HTMLBars*, *EmberJS Data*.

4.4.1 Komponendid

Tabel 5. EmberJS-i abikomponendid

Nimetus	**Liik**	**Reiting**	***Fork*-de arv**	**Kirjeldus**
ember-table	tabel	1341	201	Andmete kuvamine kasutajasõbralikus formaadis võimalustega andmeid sorteerida ja filtreerida
ember-charts	graafik	590	91	D3 teegil põhinevad graafiku komponendid andmete visualiseerimiseks
ember-calendar	kalender	76	21	Andmete ja sündmuste kuvamiseks kasutatav kalendrikomponent
ember-leaflet	kaartide komponent	143	31	Google kaartidel põhinev kaartide komponent asukohtade visualiseerimiseks
ember-components	abikomponentide kogum	150	10	Komponentide kogum, mis sisaldab järgmisi komponente: vormide, nupu, modaalakna, puu-, graafikute, sakkide, lehe ekskursiooni, WYSIWYG, akordioni ja nimekirja komponent

Kõikide komponentide hankimiseks on võimalik kasutada paketihaldurit *Bower* kasutades käsku

```
bower install --save ember-table ember-charts ember-calendar ember-bootstrap ember-compoents
```

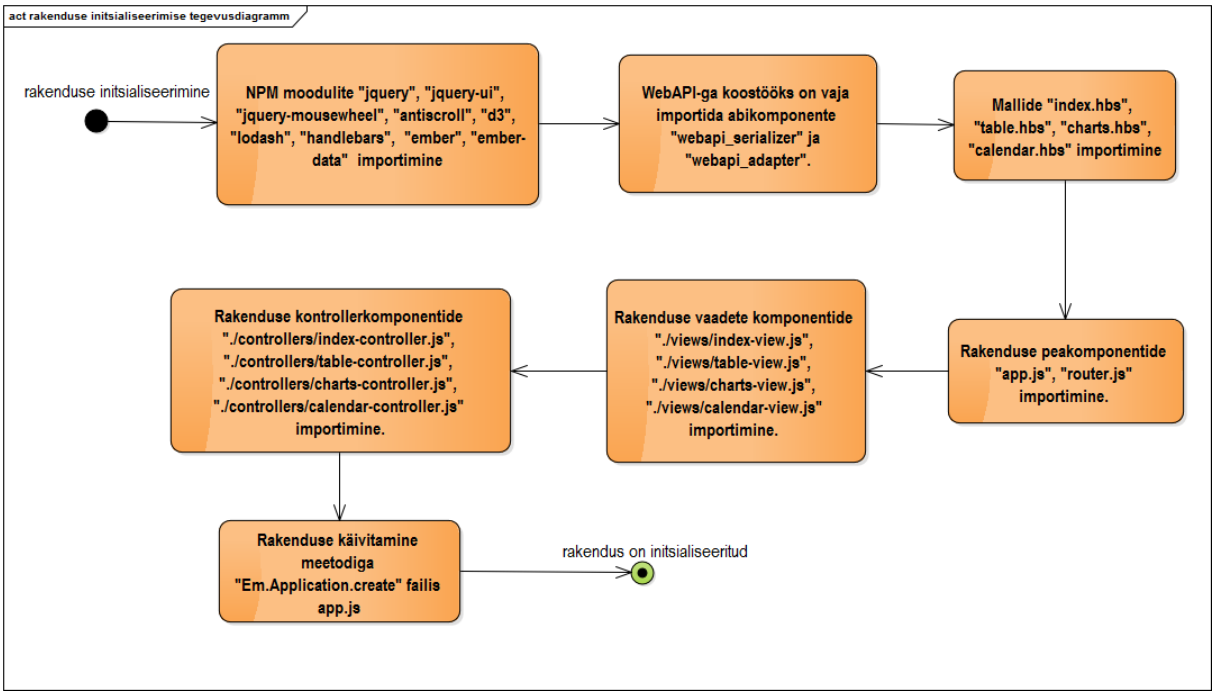
4.4.2 Rakenduse struktuur

Rakenduse struktuuri eeskujuks on võetud tööriista „*ember-cli*“ poolt genereeritava struktuuri. „*Ember-cli*“ on arendajate poolt arendatud tööriist, mis sisaldab parimaid praktikaid rakenduse struktureerimise osas. Genereerida saab teste ja testimise abikomponente, adapterit, komponenti, kontrollerit, abiobjekti, ruuteri objekte, mudelit, malle, teenuseid, serverit, teeki, ressursi, serialiseerijat, transformeerijat, utiliite ja vaateid [27].

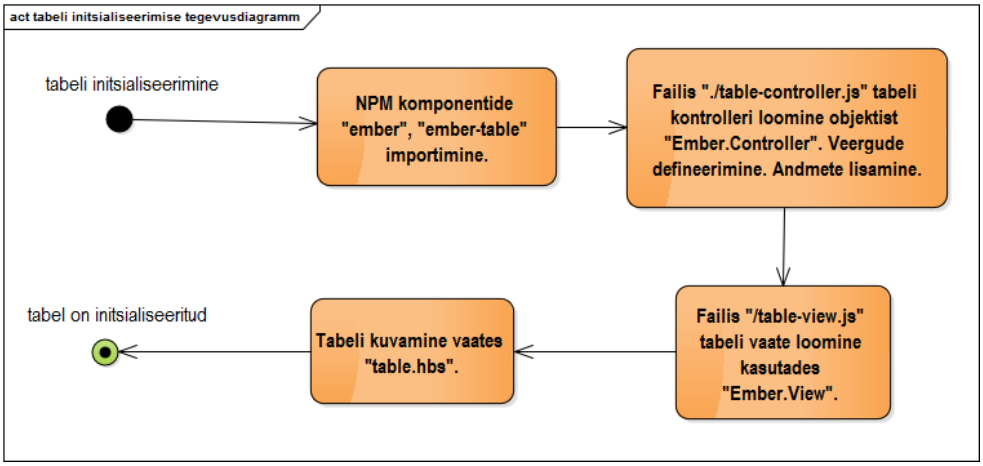
```
emberapp/             -      rakendus
    components/       -      komponentide jaoks
    controllers/      -      kontrollerid
        index-controller.js
        table-controller.js
        charts-controller.js
        calendar-controller.js
    helpers/          -      abiobjektid
    routes/           -      ruuterid
        index-route.js
        table-route.js
        charts-route.js
        calendar-route.js
    styles/           -      CSS stiilid
        …             -      stiilide kogum
    templates/        -      mallid, mida kompileeritakse ja kuvatakse lõppkasutajatele
        index.hbs
        table.hbs
        charts.hbs
        calendar.hbs
    views/            -      vaated, mis kompileerivad malle ja esitavad lõppkasutajatele
        index-view.js
        table-view.js
        charts-view.js
        calendar-view.js
    app.js            -      rakenduse tuum
    index.html        -      peavaade, kuhu laaditakse kõik vaated
    router.js         -      juhib ümbersuunamisloogikat; kasutatakse uute ruuter objektide
                             registreerimiseks
tests/                -      ühiktestid
    …                 -      igale komponendile tuleb üks või rohkem teste
vendor/               -      internetist tõmmatud komponendid
```

4.4.3 Diagrammid

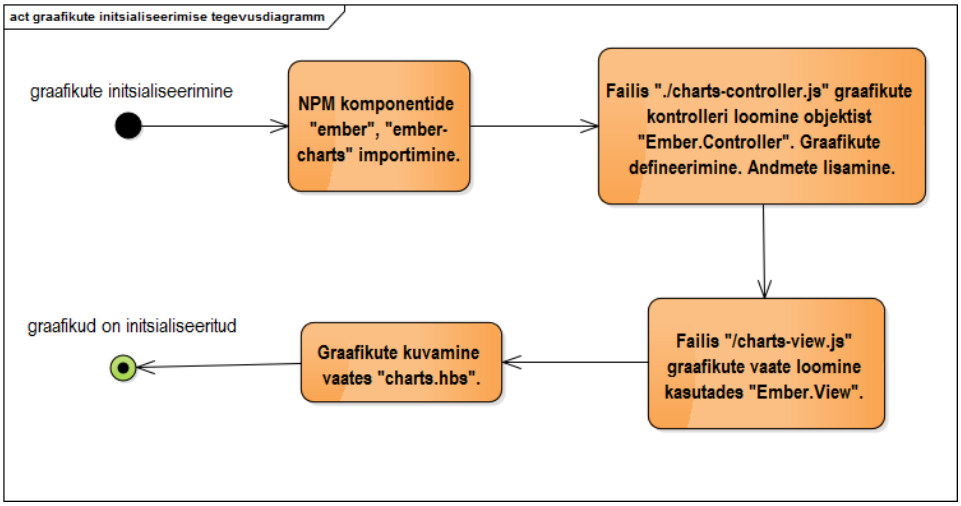
Vastavalt on kirjeldatud rakenduse, tabeli, graafikute ja kalendri initsialiseerimise protsessid.



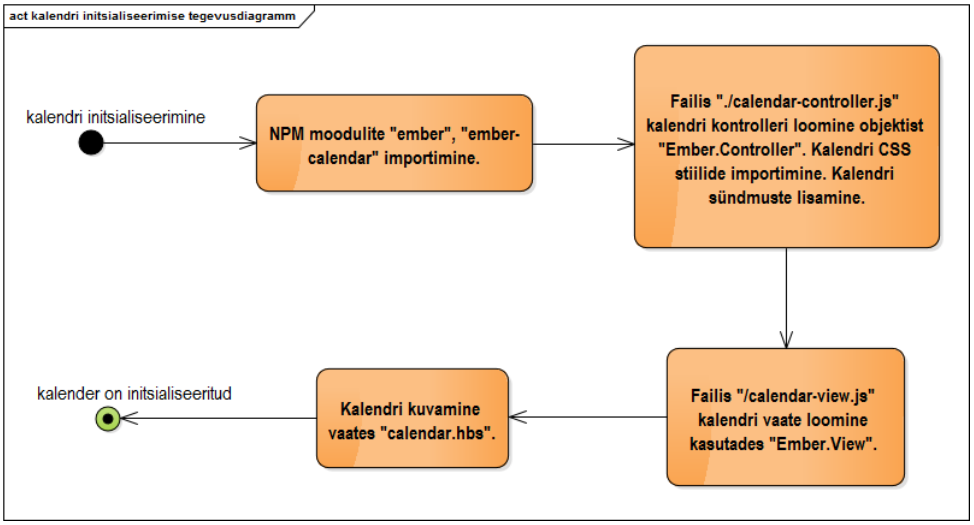
Joonis 17. Rakenduse initsialiseerimine



Joonis 18. Tabeli initsialiseerimine



Joonis 19. Graafikute initsialiseerimine



Joonis 20. Kalendri initsialiseerimine

4.5 Kliendipoolne realisatsioon kasutades ReactJS

4.5.1 Komponendid

ReactJS komponendid on enamasti optimeeritud, kirjutades rakendusi modulaarsel viisil, kasutades *AMD* või *CommonJS*-i. Paljud ReactJS komponendid on kirjutatud eeldusel, et arenduskeskkonnaks oleks *NodeJS*. *NodeJS* on platvorm, mis on ehitatud *Chrome*-i JavaScript mootoril eesmärgiga ehitada kiireid ja suuri võrgurakendusi [30]. *NodeJS* toetab ning *ReactJS* komponendid kasutavad võtmesõnu „*require*“ ja „*module.exports*“, mis kasutavad *CommonJS*-i. Samuti, kui tekib vajadus arendada teises keskkonnas ja toetada neid võtmesõnu, Seega tuleb rakendust arendada *NodeJS* keskkonnas või kasutada abikomponendina *Browserify*.

Tabel 6. ReactJS-i abikomponendid

Nimetus	**Komponendi liik**	**Reiting**	***Fork*-de arv**	**Kirjeldus**
reactable	tabeli komponent	483	53	Andmete kuvamine kasutajasõbralikus formaadis võimalustega andmeid sorteerida ja filtreerida
react-router	ruuter	4027	552	Rakenduses navigeerimise komponent, mille idee põhineb *EmberJS* ruuteril.
flux	rakenduse struktureerimise komponent	5919	986	Rakenduse struktuuri ülesehitamiseks vajalik komponent, mis on optimeeritud ka suurte rakendustega töötegemiseks.
react-google-maps	kaartide komponent	224	48	*Google* kaartidel põhinev kaartide komponent asukohtade visualiseerimiseks
react-d3	graafikute komponent	85	15	*Chart.js* teegil põhinevad graafiku komponendid andmete visualiseerimiseks
browserify	abikomponent	7185	631	JavaScript rakenduse ehitamine modulaarsel viisil. Paljud *ReactJS* komponendid kasutavad *CommonJS*-i modulaarset lähenemist, seega tuleb kasutusele võtta „*browserify*“ teek.
react-bootstrap	abikomponendid	2134	325	Väikeste tähtsate komponentide kogum, mis põhinevad *Twitter Bootstrap* komponentidel.

Kõikide komponentide hankimiseks on võimalik kasutada *Node Package Manager* (NPM) paketihaldurite käsud

```
npm install --save flux reactable react-chartjs react-router browserify-loader react-bootstrap react-google-maps
```

4.5.2 Rakenduse struktuur

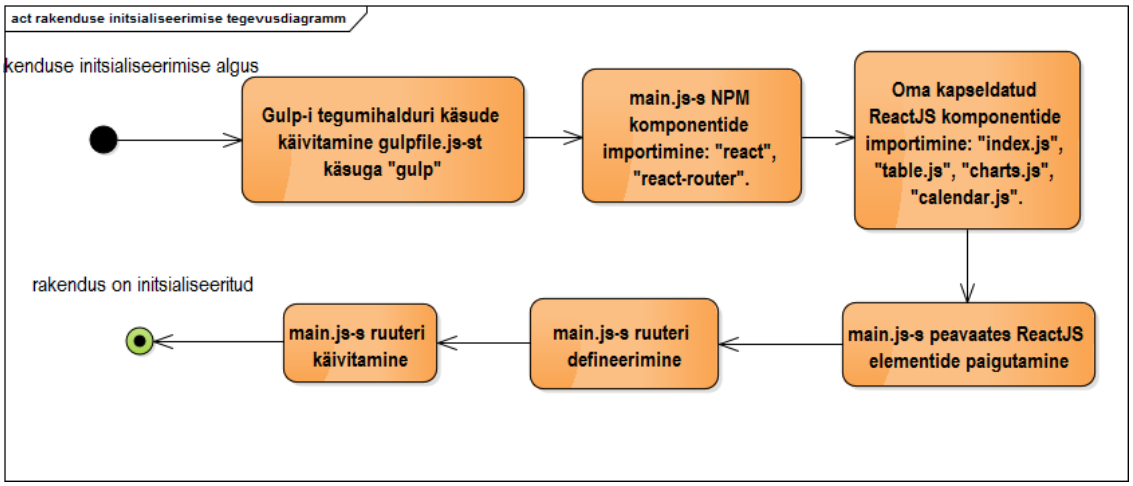
Keeruliste rakenduste ehitamiseks tuleb *ReactJS*-i kasutada koostöös *Flux* teegiga. Antud töö uurimise käigus on koostatud järgmine *Flux* teegi reeglitega vastavuses oleva rakenduse struktuur:

```
react/
    actions/
    build/
    components/              -      ReactJS elemendid ja klassid
        about.js
        calendar.js
        charts.js
        index.js
        maps.js
        performance.js
        table.js
    constants/               -      JavaScript objektid, mis käituvad enumeraatoritena
        app-constants.js
    dispatchers/             -      Sündmuste suunajad
        app-dispatcher.js
    stores/                  -      Andmete hoidja ja töötleja
        app-store.js
    gulpfile.js              -      Rakenduse konfiguratioonnid
    main.js                  -      Rakenduse sisenemiskoht
```

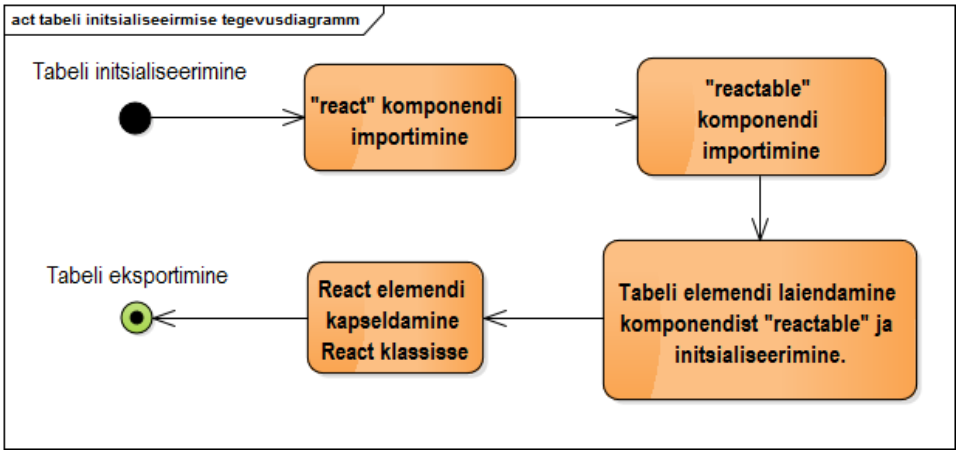
Tegumihaldurina on kasutatud *Gulp* ja modulaarse rakenduse ehitamiseks on kasutaud *CommonJS*-l põhinev teek - *Browserify*. *Browserify* on *NodeJS* stiilis kasutatud „*require()*“ meetod moodulite laadimiseks rakendusse [29].

4.5.3 Diagrammid

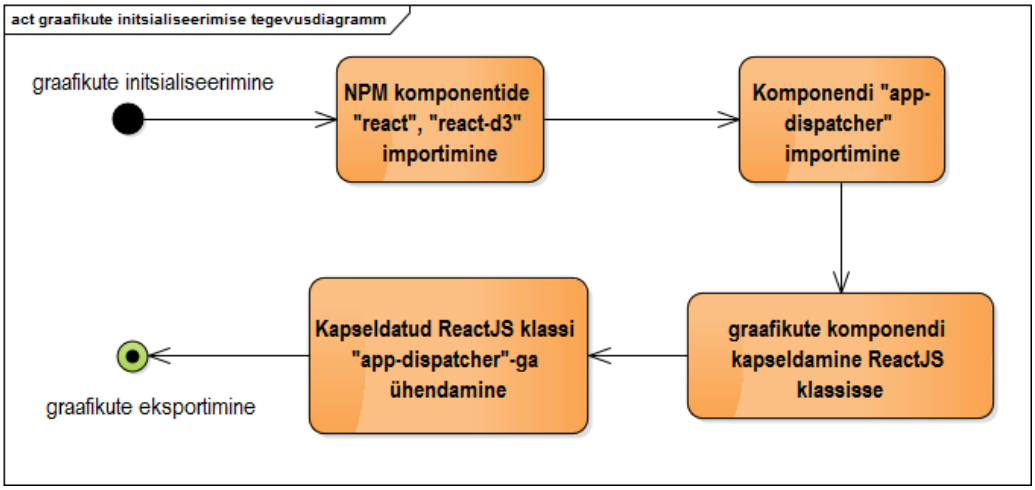
Vastavalt on kirjeldatud rakenduse, tabeli, graafikute ja kalendri initsialiseerimise protsessid.



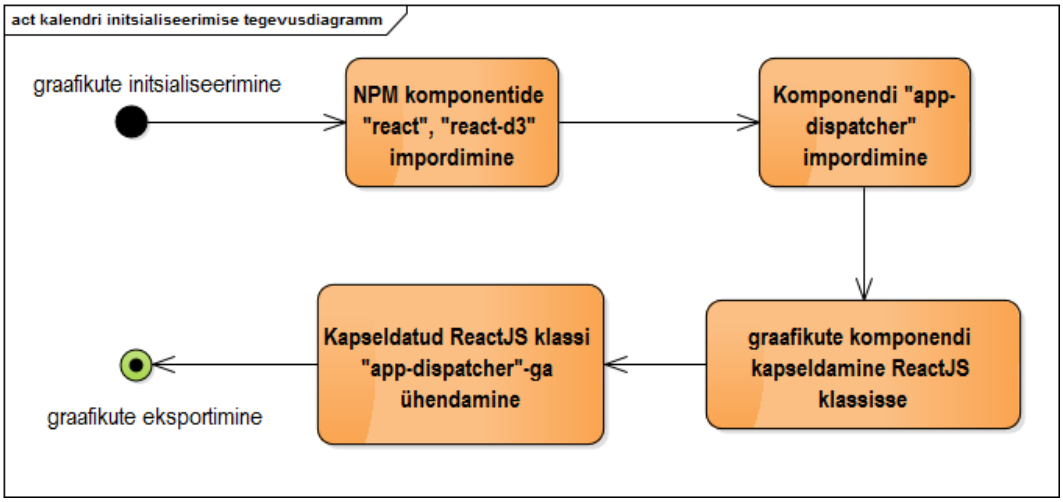
Joonis 21. Rakenduse initsialiseerimine



Joonis 22. Tabeli initsialiseerimine



Joonis 23. Graafikute initsialiseerimine



Joonis 24. Kalendri initsialiseerimine

4.6 Kliendipoolne realisatsioon kasutades BackboneJS

4.6.1 Komponendid

BackboneJS-l on kõikidest võrreldavatest raamistikkudest kõige lihtsama struktuuriga ning puudub vajadus igat komponenti kapseldamiseks. Tänu *BackboneJS* paindlikkusele on *BackboneJS* võimeline tegema koostööd *jQuery* teegiga.

Tabel 7. BackboneJS-ga arendamisel kasutatavad abikomponendid

Nimetus	**Komponendi liik**	**Reiting**	***Fork*-de arv**	**Kirjeldus**
backgrid	tabel	1751	278	Andmete kuvamine kasutajasõbralikus formaadis võimalustega andmeid sorteerida ja filtreerida. Optimeeritud *BackboneJS*-i jaoks.
Chart.js	graafikud	14298	4310	*D3* teegil põhinevad graafiku komponendid andmete visualiseerimiseks.
fullcalendar	kalender	4148	1434	Andmete ja sündmuste kuvamiseks kasutatav kalendrikomponent.
gmaps	kaartide komponent	5285	872	*Google* kaartidel põhinev kaartide komponent asukohtade visualiseerimiseks.
underscore.js	abikomponent	14387	3263	*BackboneJS*-i rakenduse ehitamiseks vajavaminev komponent, mis lihtsustab ja optimeerib tööd andmetega.
bootstrap	abikomponentide kogum	80629	31778	*jQuery*-l põhinevad abikomponendid.
marionette	rakenduse struktureerimise komponent	6123	1157	Rakenduse struktureerimiseks teek, mis lihtsustab *BackboneJS*-i ülesehitatavat rakendust ning pakub üldiseid disaini- ja implementatsiooni mustreid.
node-hbsfy	abikomponent	201	17	Mallide kompileerimiseks kasutatav komponent.
cssify	abikomponent	68	14	CSS komponentide importimine.

Kõikide komponentide hankimiseks on võimalik kasutada paketihaldurit *Bower* kasutades käsku

```
bower install --save slickgrid chart.js fullcalendar gmaps underscore bootstrap marionette
```

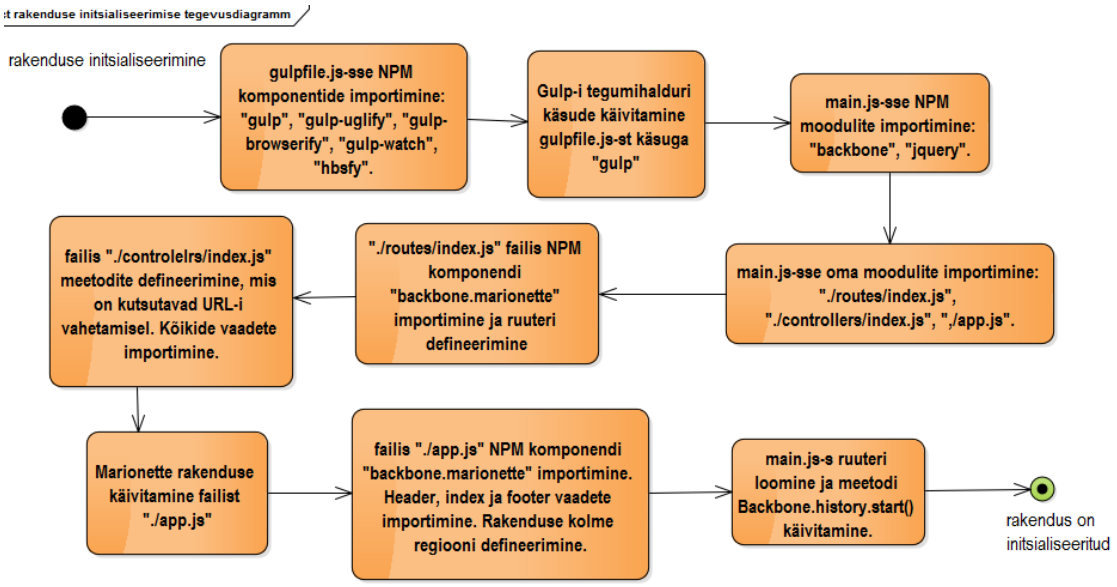
4.6.2 Rakenduse struktuur

Antud töös loodud rakenduse struktuur on järgmine:

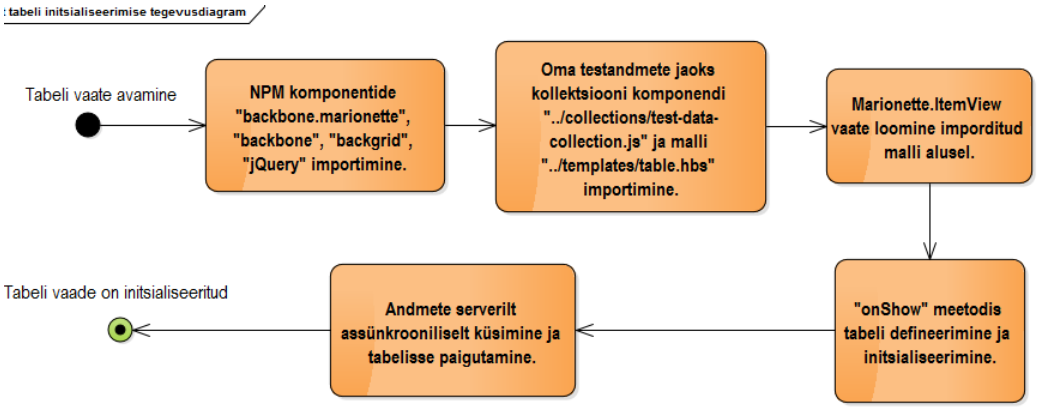
```
backbone/
    build/
        build.js              -       kokkupakitud fail
    collections/
        test-data.collection.js       -       testandmete massiiv
    controllers/
        index.js              -       controller, mis kuulab router/index.js-i
    models/
        chart-model.js
        test-data-model.js
    routers/
        index.js              -       suunab kontrollerisse controllers/index.js
    templates/                -       Handlebars mallid
        calendar.hbs
        charts.hbs
        footer.hbs
        header.hbs
        index.hbs
        notfound.hbs
        table.hbs
    vendor/
        …                 -       sisaldab teeke
    views/                -       Marionette vaadete kogum
        calendar-view.js
        charts-view.js
        footer-view.js
        header-view.js
        index-view.js
        notfound-view.js
        table-view.js
    app.js                -       Marionette rakendus
    gulpfile.js           -       rakenduse konfiguratsioon
    main.js               -       Rakenduse sisenemispunkt
```

4.6.3 Diagrammid

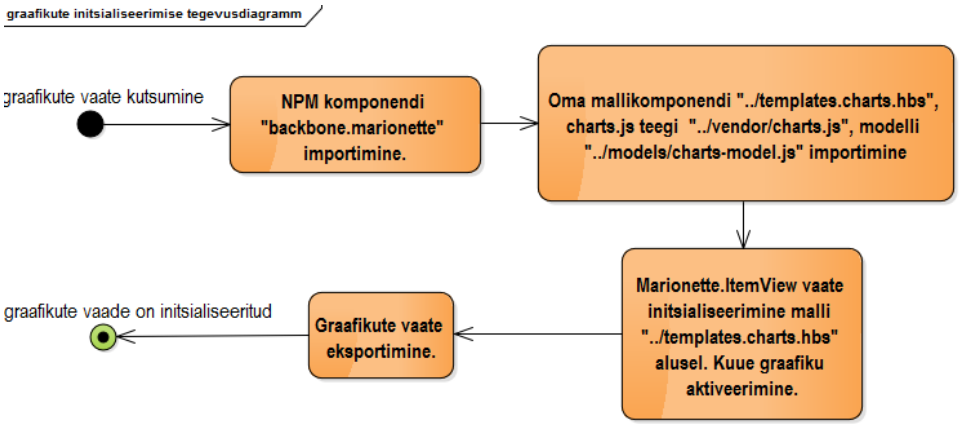
Vastavalt on kirjeldatud rakenduse, tabeli, graafikute ja kalendri initsialiseerimise protsessid.



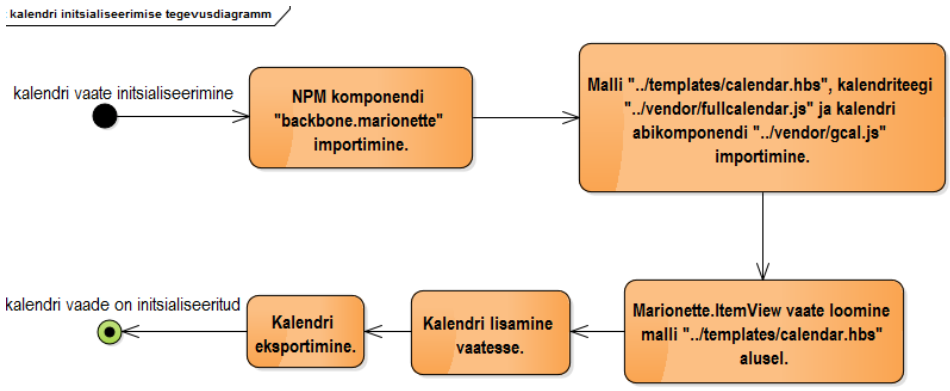
Joonis 25. Rakenduse initsialiseerimine



Joonis 26. Tabeli initsialiseerimine



Joonis 27. Graafikute initsialiseerimine



Joonis 28. Kalendri initsialiseerimine

5. Analüüs

Järgnevalt võetakse ette neli eelnevalt analüüsitud rakendust ning tehakse nende kohta analüüs. Analüüsi käigus uuritakse järgnevat:

- Kui palju faile oli vaja alla laadida vahemälus mitte salvestatud infot
- Kui palju võttis aega lehtede kompileerimine
- Millised on uuritavate raamistikute plussid ja miinused

5.1 *AngularJS* realisatsiooni analüüs

5.1.1 Tulemused

AngularJS-l on kõige suurem hulk pakutavaid komponente, mida on võimalik rakendusse juurutada. Kui vaatluse alla võtta kõike selles töös läbiproovitud komponente, siis võib järeldada, et komponendid on isoleeritud ja globaalsete muutujate kokkupõrked on välditud. Teisest küljest saab ainult juurutada *AngularJS* spetsiifilised komponendid ning vältida kõike teisi. See kaotab automaatselt võimaluse *jQuery* komponentide mugavaks kasutamiseks, sest tekib vajadus alternatiivi otsimises kirjutatuna *AngularJS*-s või ümbritseda direktiivisse, millele kulub aeg ja mida saavad teha ainult kogenumad arendajad.

Koodi loetavuse osas on probleemseks kohaks muutujad vaates. Esialgu valesti kirjutatud kood võib arendajat segamini ajada.

```
<div ng-app=”app” ng-controller=”ctrl”>{{test}}</div>
```

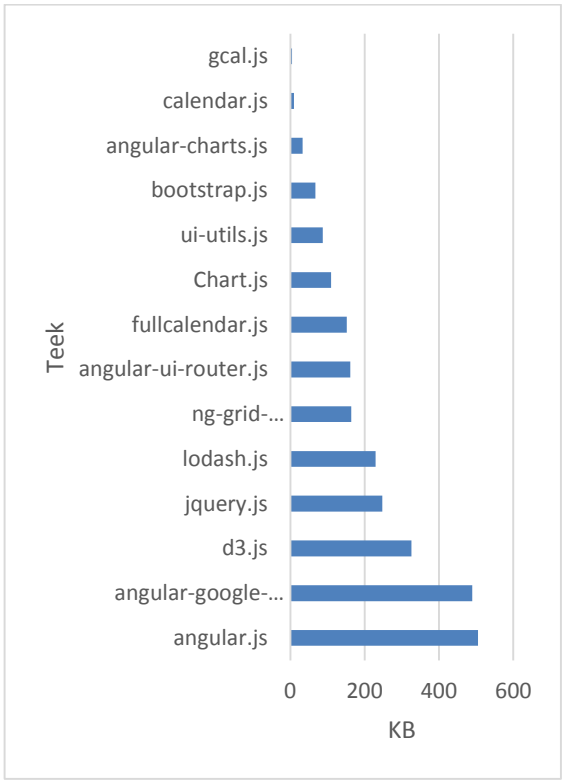
Siin on näha, et muutuja „*test*“ peaks olema kontrolleris nimega „*ctrl*“. Vastav probleem paistab hästi silma, kui muutuja asub osalises vaates ja ei pruugi teada, milline on ülem element. *AngularJS* arendajad pakkuvad kasutada järgmist kirjutamisviisi: „*objektiNimi.objetktiNimi*“.

Samuti on probleemiks, et massiivide itereerimise kiirus *AngularJS*-s on tunduvalt aeglasem võrreldes tavalise *JavaScript*-ga või teiste raamistikkudega. Praktilisest seisukohast, kui rakendus vajab korraga kuvada palju andmeid korraga, siis tabeli komponent saab hästi hakkama ning vaja on kuvada ainult andmed, millised saaks mahutada ekraanile.

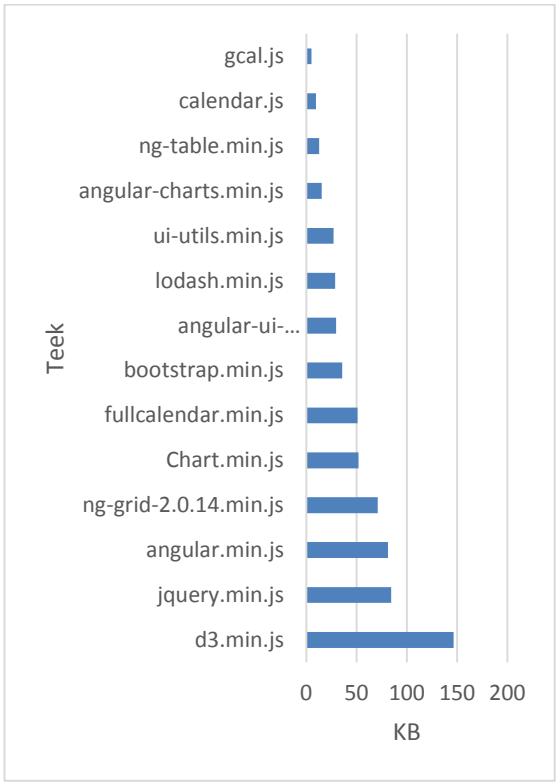
Tuleb pöörata suuremat tähelepanu *AngularJS* failide minimiseerimisele, kuna halvemal juhul võib rakendus katki minna.

5.1.2 Rakenduse suurus

Vastavalt on toodud kõikide komponentide suurused täis- ja minimeeritud kujul.



Joonis 29. AngularJS komponentide suurused



Joonis 30. AngularJS minimeeritud komponentide suurused

Nagu on näha, *jQuery*-t on ikka vaja võtta kasutusele. Samuti kõige tähtsam komponent graafikute ehitamiseks on kõige halvemini minimeeritav. Seega vajavamineva funktsionaalsuse tagamiseks on rakenduse suurus minimeeritud kujul 846 KB.

5.1.3 Tööjõudlus

Tööjõudluse testimiseks on kasutatud massiivi kõikide elementide kuvamise vaates, kasutades direktiivi *„ng-repeat“*. See tähendab, et massiiv kasutab kahesuunalise sidumise vaate ja mudeli vahel. Massiivi sisse lisatakse 5000 objekti ning uuritakse, kuidas saab rakendus hakkama.

Vaade:

```
<span ng-app=“app“ ng-controller=“ctrl“>
    <span ng-repeat="item in items">{{item}}</span>
</span>
```

Mudel:

```
$scope.addItems = function () {
    $scope.list.length = 0;
    for (var i = 0; i < 5000; i++) {
        $scope.list.push(i);
    }
}
```

Teine test oli käivitatud kohe peale esimest testi, mille tulemusena tehti massiiv tühjaks ning täideti uuesti andmetega ilma lehe täieliku ümberlaadimist.

Tabel 8. AngularJS-i tööjõudlus

Objektide arv	**Test 1, ilma tabelita**	**Test 2, ilma tabelita**	**Test 1, tabeliga**	**Test 2, tabeliga**
500	107.9ms	91.8ms	86.5ms	84.4ms
2500	98.5ms	89.0ms	169.9ms	85.1ms
5000	155.0ms	89.3ms	275.7ms	83.7ms

Statistika järgi, *AngularJS* tsükkel käivitab alguses funktsiooni „*$apply()*“, mille sees on olemas operatsioon „*$digest()*“. Viimane operatsioon annab käsu uuendada kõike kasutajaliideses asuvaid elemente. Selle sees käivitatakse omakorda “ *$watchCollectionAction*“, mis käib igat elemendi massiivist ja uuendab elemendi väärtust. *AngularJS* arendajate sõnul on selline viis kõige kiirem ning võimaldab toetada *AngularJS*-i laial hulgal brauseritest.

5.1.4 Plussid

- **Alustamine**. Tuleb lihtsalt lisada põhiteeki, lisada põhilise *tag*-i „*ng-app“* ning rakendus hakkab kohe tööle.

- **Toetus arendajate poolt**. teek on pidevas arenduses ning kellest mitu arendajat on Google arendajad.

- **Suur kogukond**. Arendajad arendavad pidevalt uusi komponente, lihtsustades tööd teiste arendajate jaoks.

- **Rakenduse struktuur**. Rakendust on võimalik ehitada nii väikeste, keskmiste kui ka suurete rakenduste jaoks. Kõikidel juhtudel on kood loetav, kui seda õigesti kirjutada.

5.1.5 Miinused

- **Suur sisenemisbarjäär**. Et aru saada, kuidas on vaja lahendada ühte või teist probleemi, tuleb alguses kulutada palju aega, kui seda nõuaks keskmiselt alternatiivne teek. Kui rakendus on arendatud kasutades *AngularJS*, siis tuleb palgata kas *AngularJS* spetsialiste või tuleb kulutada arendajate peale aja- ja raharessursse nende väljaõpetamiseks.

- **Halb koostöö teiste teekidega**. Rusikareegli järgi ei tohi kokku segada *jQuery* ja *AngularJS*, muidu võib tekkida rakenduse ettearvamatu käitumine.

- **Kahesuunaline sidumine on halb praktika**. See on potentsiaalne vigade allikas, kus skoopide pärimisel võivad tekkida ootamatud vead.

- **Failide minimeerimine.** Kindlatel juhtudel võib failide minimeerimise protsess terve AngularJS rakenduse katki teha. Tuleb kasutada parimaid praktikaid kontrollerite jm AngularJS objektide defineerimiseks.

- **SEO optimeerimata.** Kirjutades *AngularJS*-ga rakendusi tuleb arvestada, et otsingu robotitel tekivad raskused lehe indekseerimisel. Seega lehe esikohale saamiseks otsingumootorites on tunduvalt raskem ning arendajatel tuleb näha rohkem vaeva, kui võrreldavatel raamistikkudel.

5.2 EmberJS plussid ja miinused

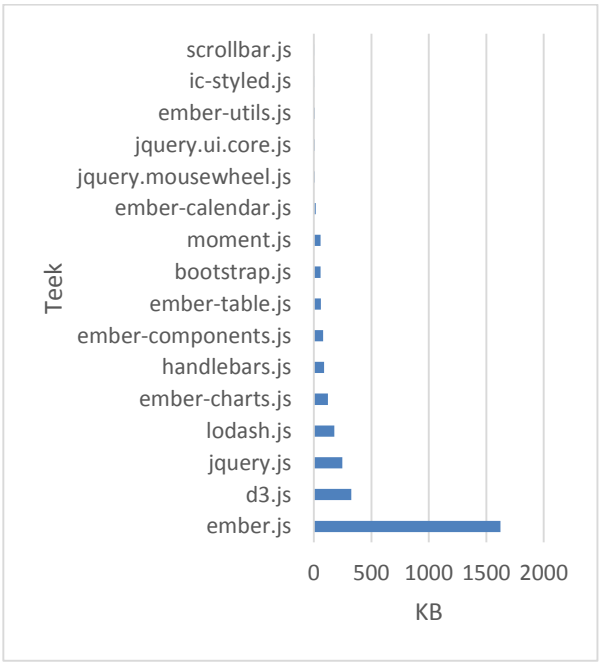
5.2.1 Tulemused

Iseenesest raamistik ei nõua palju teeke minimaalseks tööks. Samas on näiteks tabeli komponent sõltub mitmetest teekidest, mis sunnib need rakendusse sisse laadida, kasvab mälukasutus ning leht laeb aeglasemalt. Kuigi tavaliste kasutusjuhtude puhul on see vastuvõetav, sest vastava rakenduse tööks on mälukasutus alla 1 MB ning *AngularJS*-l ehitatud rakendusega sarnase suurusega.

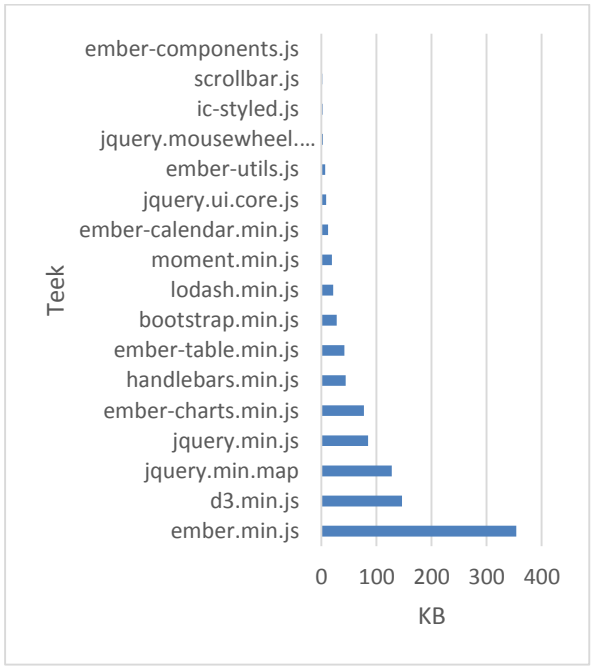
Kõike vajalikke komponente rakenduse ehitamiseks on küll võimalik leida, kuid nende populaarsus ja toetus kogukonna poolt on madalavõitu, mis ohustab rakendust mittestabiilselt töötama.

5.2.2 Rakenduse suurus

Vastavalt on toodud kõikide komponentide suurused täis- ja minimeeritud kujul.



Joonis 31. EmberJS komponentide suurused



Joonis 32. EmberJS minimeeritud komponentide suurused

EmberJS rakenduse suurus minimeeritud kujul on kokku 977 KB.

5.2.3 Tööjõudlus

Vaade:

```
{{#each EMapp.data}}<span>{{this}}</span>{{/each}}
```

Mudel:

```
EMapp = Ember.Application.create({
    rootElement: '#emapp'
});

EMapp.data = Ember.A();
window.EMclear = function () {
    EMapp.data.clear();
};
window.EMpush = function (data) {
    EMapp.data.pushObject(data);
};
```

Tabel 9. EmberJS-i tööjõudlus.

Objektide arv	Test 1, ilma tabelita	Test 2, ilma tabelita	Test 1, tabeliga	Test 2, tabeliga
500	120.9ms	90.5ms	125.9ms	88.5ms
2500	132.2ms	92.0ms	192.2ms	89.3ms
5000	170.1ms	94.3ms	300.7ms	91.2ms

5.2.4 Plussid

- **Võimas ruuter**. *EmberJS* on optimeeritud töötamiseks *URL*-idega, et neid oleks võimalik teha jagatavateks teiste inimeste vahel spetsiifiliste olekute puhul.

- ***Ember-cli* rakenduse komponentide generaator**. *Ember-cli* on võimas generaator, mis võimaldab genereerida EmberJS rakenduste osad, kasutades parimad praktikaid. Samas antud töö kirjutamise ajal puudub *ember-cli* tugi *Java* või *.NET* keskkonnale, kuigi *NodeJS*-i oleks hästi integreeritav.

- ***EmberJS* failide ja objektide nimetamise reeglid on ranged.** *EmberJS* nõuab, et enamuse *EmberJS*-i objektide loomisel oleksid jälgitud kindlad reeglid. Vastav lähenemine automaatselt soodustab

5.2.5 Miinused

- **Nõutud minimaalsed rakenduse teegid on suured**. Mudelite osa moodustab suurema osa *EmberJS*-st, mis sisaldab *HTMLBars* mallide moodustamisteeki ja muid komponente.

- **Suur sisenemisbarjäär**. Enne *EmberJS*-i rakenduste arendamist tuleb lugeda läbi dokumentatsiooni. *EmberJS* on intuitiivne kasutamises ainult siis, kui arendaja on teadlik enamusest või kõikidest reeglitest.

- **Komponentide kvaliteet ja kogus on madal**. Põhifunktsionaalsuse realiseerimiseks on võimalik leida ja juurutada *EmberJS* komponente, kuid nad ei paku palju funktsionaalsust ning alternatiivseid komponente peaaegu polegi olemas.

5.3 *ReactJS* plussid ja miinused

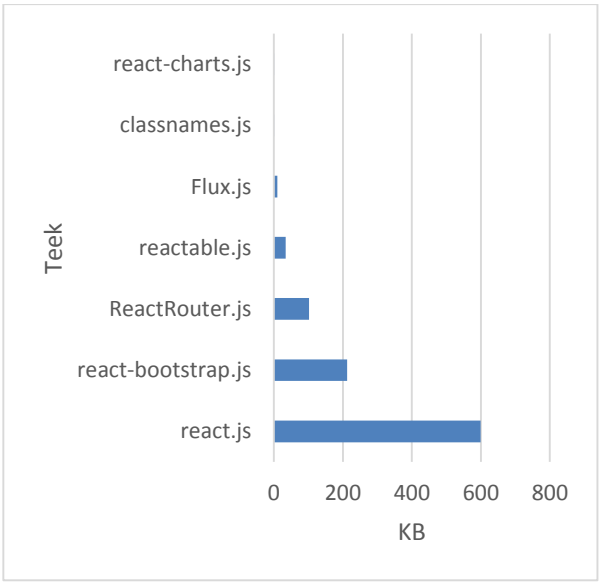
5.3.1 Tulemused

Enamus komponentidest nõuavad modulaarset lähenemist rakenduse ehitamisel. Komponentides on kasutusel võtmesõnad, nagu „*require*“ ja „*module.exports*“, mis eeldavad *CommonJS*-i olemasolu. Selle nõue rahuldamiseks on kasutatud komponent nimega „*browserify*“. Alternatiivselt on võimalik modifitseerida komponenti või võtta kasutusele mitte ReactJS põhilise komponendi ning kapseldada seda *ReactJS* komponendi sisse. Viimaste tegevuste tõttu on mittekogenul arendajal võimalik kergesti eksida ning seega vastavad tegevused ei ole soovitatavad.

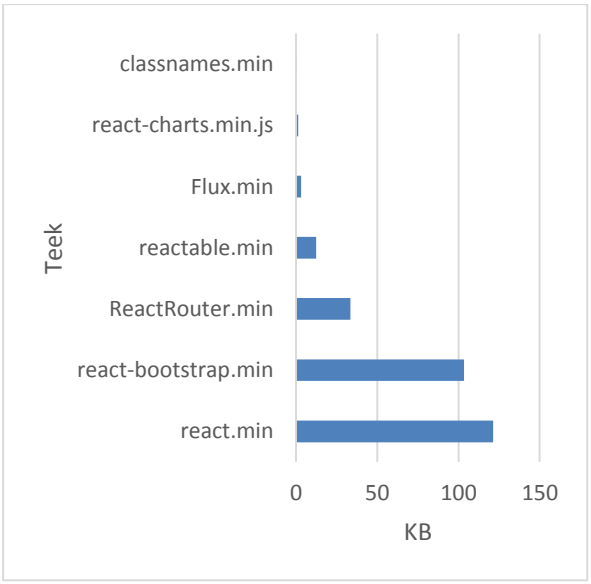
Kindla struktuuri defineerimiseks on kasutatud „*flux*“, mis tõepoolest võimaldab kirjutada suuri rakendusi. Ruuteri spetsiifiliste probleemide lahendamiseks on kasutatud *EmberJS*-i ruuteri funktsionaalsuse ideel põhinev teek.

5.3.2 Rakenduse suurus

Vastavalt on toodud kõikide komponentide suurused täis- ja minimeeritud kujul.



Joonis 33. ReactJS komponentide suurused



Joonis 34. ReactJS minimeeritud komponentide suurused

5.3.3 Tööjõudlus

Tööjõudlust on mõõdetud objektide lisamise teel massiivi. Proovitud on variant, kus kasutajaliides näidatakse ainult andmeid puhtal kujul ja varianti, kui andmete esitamiseks on kasutatud tabel.

Teine test oli käivitatud kohe peale esimest testi, mille tulemusena tehti massiiv tühjaks ning täideti uuesti andmetega ilma lehe täieliku ümberlaadimist.

Tabel 10. ReactJS-i jõudlus

Objektide arv	**Test 1, ilma tabelita**	**Test 2, ilma tabelita**	**Test 1, tabeliga**	**Test 2, tabeliga**
500	107.9ms	91.8ms	86.5ms	84.4ms
2500	98.5ms	89.0ms	169.9ms	85.1ms
5000	155.0ms	89.3ms	275.7ms	83.7ms

5.3.4 Plussid

- **Kiire**. Üks kõige suurematest põhjustest *ReactJS* arendamises on selle töökiirus, mis võimaldab teha tööd suure andmete hulgaga. Samuti ei peab *ReactJS* terve DOM-i ümber ehitama, kui arendaja on teinud väikese muudatuse. *ReactJS* võrdleb virtuaalset ja valmis kompileeritud DOM-i, ning teeb minimaalsed muudatused nende omavaheliseks sünkroniseerimiseks.

- **Kompaktne**. Põhifunktsionaalsuse tööle saamiseks kulub suhteliselt vähe mälu.

- **Brauserite lai tugi**. Virtuaalne DOM võimaldab lahendada sündmuste töötlemisega seotud probleeme, mis võiksid olla erinevalt välja tulnud erinevates brauserites.

- **Koostöö teiste teekidega**. *ReactJS* on hästi integreeritav koos teiste raamistikude ja teekidega, nagu *jQuery*. Kui vajalik teek pole *ReactJS*-le kirjutatud, siis saab ilma suurte raskusteta võtta kasutusele *jQuery*-s kirjutatud teeki ning kapseldada seda *ReactJS* komponenti.

- **SEO optimeeritud**. Isegi kui kasutusel on virtuaalne DOM, otsingumootorid suudavad ikkagi leida nendele vajaliku infot ning soodustavad rakenduse populaarsust.

5.3.5 Miinused

- **JSX süntaks ei ole laialt toetatud**. *ReactJS* kasutab oma vaadete genereerimiseks JSX-failis kirjeldatud virtuaalset DOM-i. Kui arenduskeskkond ei toeta JSX süntaksit, siis koodi kirjutamine võib osutuda probleemseks ning produktiivsus langeb. Alternatiivselt on võimalik kasutada tavalist JavaScript-i, aga selle kasutamise sisenemisbarjäär on suurem ning kood on raskemini loetav.

- **Rakenduse struktureerimise õppimine**. Põhjusel, et kasutusel on „flux“ teek rakenduse struktureerimisel, mis kasutab MVC-st teistsugust printsiibi, tuleb arendajatel kulutada aega rakenduse struktureerimise õppimisele.

- **Mõeldud arendamiseks *NodeJS* keskkonnas.** *ReactJS*-i tööle saamiseks ei ole vahet, mis keskkonnas toimub arendus. Kui asi puudutab komponentide kasutust, sealhulgas ka „*flux*“ rakenduse struktureerimise komponenti, siis need komponendid on orienteeritud *NodeJS*-le. Teisesse arenduskeskkonda integreerida pole võimatu, kuid tuleb vaeva näha.

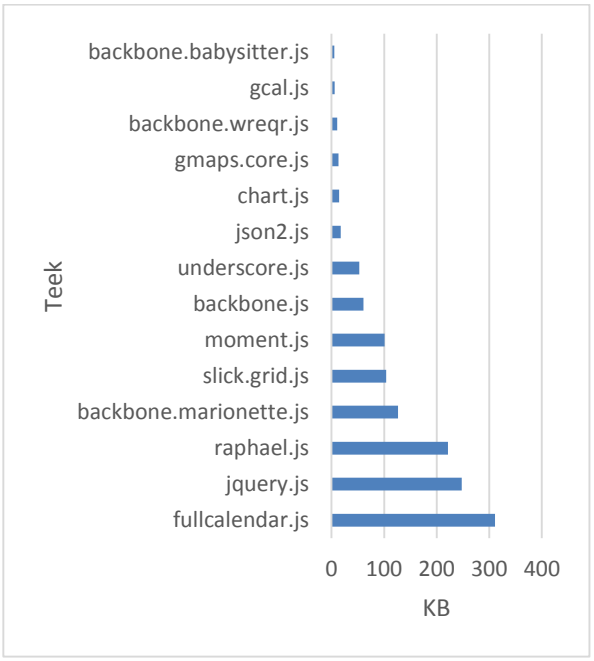
5.4 *BackboneJS* plussid ja miinused

5.4.1 Tulemused

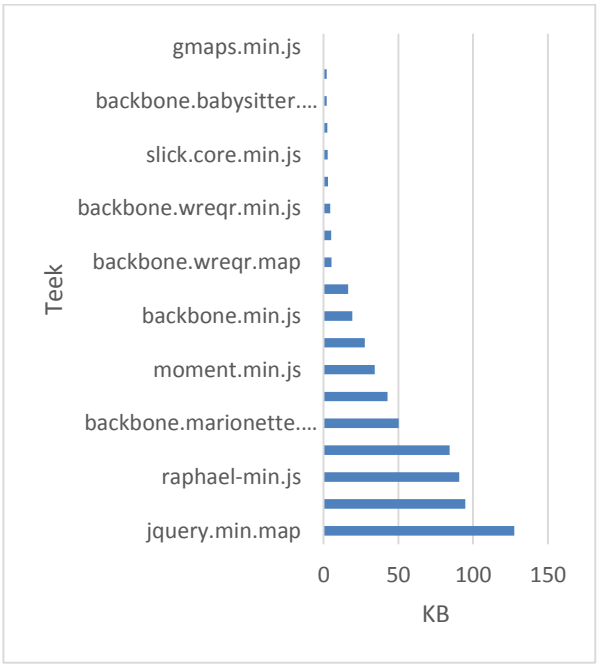
BackboneJS võimaldab ehitada rakendusi, mille suurused on olulised väiksemad kui võrreldavatel raamistikkudel. *BackboneJS* on tihedalt seotud *jQuery* teegiga ning on sellest pidevas sõltuvuses. *Marionette* võimaldab lahendada rakenduse struktureerimisega seotud probleeme.

5.4.2 Rakenduse suurus

Vastavalt on toodud kõikide komponentide suurused täis- ja minimeeritud kujul.



Joonis 35. BackboneJS-i rakenduse jaoks vajalikke komponentide suurused



Joonis 36. BackboneJS rakenduse jaoks vajalikke komponentide suurused minimeeritud kujul

5.4.3 Tööjõudlus

Tabel 11. BackboneJS-i tööjõudlus.

Objektide arv	Test 1, ilma tabelita	Test 2, ilma tabelita	Test 1, tabeliga	Test 2, tabeliga
500	100.9ms	89.3ms	125.9ms	84.5ms
2500	111.2ms	85.2ms	150.2ms	87.3ms
5000	123.1ms	88.6ms	240.7ms	96.2ms

5.4.4 Plussid

- **Väike, stabiilne ja läbi testitud**. Teek annab võimaluse kasutada mudeleid ja vaateid baasfunktsionaalsusena, mis on minimaalne nõue rakenduste ehitamiseks.

- **Väike sisenemisbarjäär**. *BackboneJS*-iga tutvumiseks võib kuluda tunni piires. Samas kulub rakenduse struktureerimisele rohkem aega.

- **Koostöö teiste teekidega**. *BackboneJS* on hästi integreeritav koos teiste raamistikude ja teekidega, nagu jQuery

5.4.5 Miinused

- **Suurte rakenduse jaoks on vaja lisada muid teeke.** *BackboneJS* -i baasfunktsionaalsus hõlmab ainult väikest vabadust rakenduse ehitamises. Et kood oleks loetavam, skaleeruv ja hallatavam, tuleb kasutusele võtta teisi teeke. Mallide mootorina on võimalik kasutada näiteks underscore.js, *HTMLBars* või midagi muud.

5.5 Kiiruste võrdlemine

Järgnevalt on võrreldud massiividesse lisamise kiirused, kus massiivid on seotud kahesuunalise sidumisega mudeli ja vaate vahel. Kiirustesti seadistamine:

- „*ANGclear*“ ja „*ANGPush*“ – esimene meetod tühistab skoobis oleva massiivi ja teine lisab sinna andmeid. Massiivi elemente kuvatakse kasutades „*ng-repeat*“.
- „*EMclear*“ ja „*EMpush*“ – andmete massiivi hoitakse *Ember.Application* muutujas. Esimene meetod tühistab massiivi ja teine lisab sinna uued.
- „*RClear*“ ja „*RPush*“ – andmete massiivi hoitakse *React.Component*-s. Andmeid uuendatakse ReactJS komponendi „*setState*“ meetodiga peale igat lisamist/eemdaldamist.
- „*BClear*“ ja „*BPush*“ – andmete massivi hoitakse *Backbone.View*-s. Kasutatakse *Backbone.View* meetodeid andmete lisamiseks ja eemaldamiseks.

Tabel 12. Raamistikute töökiiruste võrdlus

Raamistik/Teek	**Kood**	**Operatsioone sekundis (ümmardatuna 5ni)**

AngularJS	`ANGclear();`
	`for (var i = 0; i < 100; i++)`
	`  ANGpush("nitem" + Date.now());`
		umbes 100

EmberJS	`EMclear();`
	`for (var i = 0; i < 100; i++)`
	`  EMpush("eitem" + Date.now());`
		umbes 80

ReactJS	`RClear();`
	`for (var i = 0; i < 100; i++)`
	`  RPush("ritem" + Date.now());`
		umbes 150

BackboneJS	`BClear();`
	`for (var i = 0; i < 100; i++)`
	`  BPush("ritem" + Date.now());`
		umbes 110

5.6 Raamistiku valik

Antud töö käigus on selgunud iga raamistiku plussid ja miinused. Vaatamata puudustele on kõikide raamistikkudega võimalik arendada suuri veebilehti. Autori seisukohalt oleks kõige optimaalsem arendada *AngularJS* raamistikuga. Põhjuseks on kõige suurem toetus kogukonna poolt. Töö käigus on selgunud, et *AngularJS* komponendid on parema kvaliteediga ning palju suurema lisafunktsionaalsusega võrreldes teiste raamistiku komponentidega. Kogukonna toestus annab samuti võimaluse internetis kergemini üles leida lahendusi probleemidele, millega arendajad puutuvad tihti kokku. Kiiruse poolest on *AngularJS* küll üks aeglasematest, kuid praktilisest seisukohast on lehe laadimise ja andmete töötlemise kiirused mõistlikud, et rakendust oleks võimalik kiirelt ja tõhusalt kasutada. See tähendab, et rakenduse kasutamisel pole võimalik silma järgi aru saada, millise tehnoloogia alusel on rakendus arendatud.

Tabel 13. Autori raamistikute eelistused

	AngularJS	**BackboneJS**	**EmberJS**	**ReactJS**
Kogukond	esimene eelistus	teine eelistus	neljas eelistus	kolmas eelistus
Rakenduse struktureerimine	teine eelistus	neljas eelistus	esimene eelistus	kolmas eelistus
Komponentide kättesaadavus	esimene eelistus	kolmas eelistus	neljas eelistus	teine eelistus
Testimine	esimene eelistus	neljas eelistus	teine eelistus	kolmas eelistus
Rakenduse kiirus	kolmas eelistus	teine eelistus	neljas eelistus	esimene eelistus

Autor arvestab, et üldine pilt võib muutuda järgnevate aastate jooksul, kus turule võivad tulla uued raamistikud. Samas, et need raamistikud jõuaksid esikohtadele maailmas veebiarendajate hulgas, on kulunud aastaid. Vastavalt on võimalik teha hüpoteetiline järeldus, et vastavad raamistikud säilitavad populaarsust, kui arendada nende tehnoloogiatega.

6. Metoodika väljatöötamine

6.1 Produktiivsus

Positiivne produktiivsus on võti raha- ja ajaressursside säästmiseks. Tähtis on pöörata tähelepanu juba projekti algfaasis projekti struktuurile ja mõelda võimalikult palju läbi, milliseid tehnoloogiaid hakata projekti käigus kasutusele võtma. Antud töö kontekstis on produktiivsuse optimeerimise keskpunktiks võetud kliendipoolsete rakenduste arendamine.

Et säilitada produktiivsust, siis tuleks minimiseerida enda jaoks tühja tööd ning võta kasutusele olemasolevaid lahendusi. Kui rääkida teiste arendajate komponentidest, siis tuleks hoolikalt otsida. Kõige populaarsemas avatud lähtekoodiga komponentide ressursis *Github*-s on komponendid hinnatavad erinevate omaduste alusel. Tuleb pöörata tähelepanu, kui palju aega ja jõudu ning kui palju inimest on panustanud komponenti ehitamisse. Samuti tuleb pöörata tähelepanu, kui paljudele on komponent meeldinud, mis peegeldab teiste arendajate arvamust. Mida suurem number, seda kindlam on hinnang.

6.2 Parimad praktikad ja soovitused

JavaScript-i tuleb tunda heal tasemel, et osata teha ükskõik millise raamistikuga või teekiga tööd. Sõltumata raamistiku valikust tuleb silmas pidada hulk reegleid, mida on maailmas laialt kasutusel.

6.2.1 Modulaarsus

Halb praktika on kirjutada halvasti loetavaid suuri *JavaScript* faile, kus loogika pole selgelt eristatav. Samuti tuleb vältida globaalsete muutujate kasutamist, sest nad võivad tekitada omavahelisi konflikte.

Selle jaoks tuleb koodiloogikat organiseerida moodulitesse. Koodi kapseldamise moodulitesse on võimalik teha anonüümse funktsioonide abil ja nende kohesel käivitamisel, kasutades „*( )*“ anonüümse funktsiooni lõpus:

```
(function(){
    // Anonüümse funktsiooni sisu
}());
```

Vastav lahendus ei ole kaugeltki optimaalne, sest rakenduse moduleeritud koodiosa võib tekitada tugevaid sõltuvusi teiste moodulitega. See nõuab failide laadimist õiges järjekorras, millega on kerge vigu tekitada.

2015-nda aasta seisuga on *Github* reitingute järgselt populaarseimad lahendused *AMD* ja *CommonJS*-i printsiibe kasutavad teegid. Asünkroonne mooduli definitsioon, lühidalt *AMD*, on *JavaScript*-i spetsifikatsioon, mis defineerib API koodimoodulite ja nende sõltuvuste defineerimiseks ning vajadusel nende asünkroonilisel juurde laadimisel. *CommonJS*-i eesmärgiks on JavaScript ökosüsteemi spetsifitseerimiseks brauserist väljaspool, nagu serveril või natiivsetel rakendustel. *AMD* ja *CommonJS*-i kasutus võimaldab optimeerida veebilehtede jõudlust, kuna serveril on vaja alla laadida väiksemaid faile. Samuti tuleb sisse nõue, et sõltuvate failide puudumisel ei käivitata ka nendest sõltuvaid faile.

6.2.2 Valideerimisteegi kasutus.

JSLint on staatilise koodi analüüsimise tööriist, mis on kasutaud tarkvara arenduses JavaScript koodi kompileerimisel vastavalt koodireeglitele. *JSLint* võtab viimase *EcmaScript*-i standardi reegleid ning valideerib tarkvara arenduses kirjutatud koodiga. Juhul, kui on leitud vigu, *JSLint* joonib alla vigased kohad. Võimalikeks alternatiivideks on *JSHint* ja *ESLint*.

Üks viisidest, kuidas tõsta valideerimise rangust, on kirjutada *JavaScript* koodi ranges režiimis. Moodulbloki algusesse tuleb panna märksõna '*use strict*', et *JavaScript* oleks valideeritav rangemate reeglite järgi, mis võimaldab töökindlama koodi kirjutamist.

6.2.3 Tegumihalduri kasutamine.

Kõik primitiivsed tegevused, mida on võimalik, tuleb võimalused automatiseerida. Teegid „*grunt*“ ja „*gulp*“ on kõige populaarsemad ning võimaldavad soodustada arendamist. Tegumihaldurite seadistamine on ajakulukas protsess ning võib aeglustada arendamist kõige algsemas faasis, eriti kui arusaam on nõrk. Kuigi nende teekide tasub ennast ära pikemas perspektiivis, kui projekt kasvab suuremaks ning tekib vajadus keeruliste operatsioonide kiireks täitmiseks, nagu failide minimeerimine, konverteerimine ühest formaadist teisse, testide käivitamine, koodi valideerimine ja transkompileerimine, teekide versioonide uuendamine, mikroserverite käivitamine arenduses. Tegumihaldur on heas koostöös paketiihaldussüsteemidega.

Järgnevalt on esitatud nimekiri põhilistest tegumihalduri moodulitest, mida oleks võimalik rakendada:

Tabel 14. Tegumihaldurite põhilised moodulid

Tegevus failidega	**Grunt**	**Gulp**	**Kirjeldus**
Kopeerimine	grunt-copy	gulp-copy	Kopeerib failid ühest määratud kaustast teisse.
Minimeerimine	grunt-uglify	gulp-uglifyjs, gulp-uglifycss	Minimeerib failid laadimiskiiruse optimeerimiseks
Ühendamine	grunt-concat	gulp-concat	Ühendab mitu faili ühte
Valideerimine	grunt-jslint	gulp-jslint	Kontrollib, kas failis on olemas süntaks-vigu
Testimine	grunt-test	gulp-test	Rakenduse testide käivitamine
Server	grunt-server	gulp-connect	Rakenduse käivitamine kasutades sisse ehitatud serverit
Automaati-seeritud tegevuste käivitamine	grunt-contrib-watch	gulp-watch	Tegevushalduri poolt mingi jälgitava faili muutmisel käivituvad kindlad käsud

6.2.4 Paketihaldussüsteemide kasutuselevõtt.

Teekide alla tõmbamiseks ja integreerimiseks projekti on kasutatud paketihaldussüsteemid. Populaarseimad on „*npm*“ ja „*bower*“. Käsureaga on võimalik määrata avalikult saadava teegi nime ja pakettide haldussüsteem tõmbab automaatselt vajalikud failid. Vastav protsess on kiirem ja automaatsem kui brauserist käsitsi teekide väljaotsimine. Samuti lahenduvad versioonidega seotud probleeme. See tähendab, kui mitme teekide versioonid rakenduses vananevad, siis kõiki on võimalik uuendada ühe käsuga „*npm update*" või „*bower update*".

6.2.5 Transkompileeritud *JavaScript*-i kirjutamine.

JavaScript keeles rakenduste kirjutamine tuleb silmas pidada, et keele dünaamilisusele ja range tüübi määratluse puudusele võivad tekkida ootamatud ajakulukad ja tootlikust tapvad vead.

Vigade arvu minimeerimiseks on võimalik jälgida kindlaid reegleid, kirjutamisviise ja kasutada universaalseid komponente.

Transkompileerimine ehk transpileerimine on kompilaatori tüüp, mis võtab programmeerimiskeele lähtekoodi sisendina ühes keeles ja väljastab lähtekoodina teises keeles [31]. Olemas on mitu transkompilaatorit, mis sarnanevad JavaScript keelega, aga suurendavad selle omaduste skoobi. Transkompilaatoriga on vigade kontroll rangem, kus valideeritakse ka muutujate tüübid. Kui rakendust on vaja teha kättesaadavaks tavakasutajatele, siis terve lähtekood peaks olema transpileeritud kujul, et ei tekiks lisaoperatsioone rakenduse kiiruse soodustamiseks. Ühed populaarseimad avatud lähtekoodiga transkompilaatorid-id, mida oleks võimalik kasutusele võtta ja mis soodustaks arendust:

Tabel 15. Transkompilaatorid

	CoffeeScript	**Babel**	**ClosureScript**	**TypeScript**
Reiting	11128	5431	4772	4339

6.2.6 Raamistiku valik ei mängu suurt rolli.

Raamistikutel on omad plussid ja miinused ning nendest peab olema võimalikult palju teadlik enne, kui rakendada uuesse projekti. Probleem, et raamistik aeglustab arendust, võib tekkida iga raamistikuga, kui sellega pole piisavalt tutvutud. *AngularJS*-le tuleb kulutada kõige rohkem aega selle tundmisele, *BackboneJS*-le kõige vähem. Tähtis on teada oma meeskonna liikmete oskused ja tasemed ning võtta kasutusele vastavad raamistikud ja teegid.

6.3 *JavaScript*-i areng tulevikus

Antud töö kirjutamise ajal on alles tulemas uus standard *EcmaScript* 6 ning praegune *EcmaScript*-i versiooni number on 5. *EcmaScript* 6-l on palju uut funktsionaalsust, mis soodustavad rakenduste arendust. ES6 standard esitab lahendusi, millele kirjutavad arendajad teeke. Üks lahendustest hõlmab modulaarsusega seotud probleemide lahendust, kus võetakse kasutusele parimad praktikad *AMD*-st ja *CommonJS*-st. Autori arvates on tähtsamaks uuenduseks uute veebikomponentide defineerimine ja taaskasutamine ES6 standardi järgi. Tulemas on *Angular*-i teine versioon, mis integreerub ES6-ga hästi kokku.

7. Kokkuvõte

Käesoleva magistritöö eesmärgiks on analüüsida tänapäeva populaarsed JavaScript raamistikud eesmärgiga luua struktureeritud *SPA* rakendusi. Võrdlus käib *AngularJS*, *EmberJS*, *ReactJS* ja *BackboneJS* raamistikude vahel.

Järgnevalt on loetletud käesoleva lõputöö tulemused:

- *AngularJS*, *EmberJS*, *ReactJS* ja *BackboneJS* raamistikude analüüs;
- Populaarsete avatud lähtekoodiga komponentide väljaotsimine ning analüüsitud raamistikute alusel prototüüpide arendamine koos väljaotsitud komponentide rakendamisega;
- Prototüüpide sarnase funktsionaalsuse võrdlus ning tugevate ja nõrkade kohtade väljaselgitamine.

Läbiviidud prototüüpide võrdluse tulemusena on väljaselgitatud järgmised punktid:

- Kogukonna toetus mängib kriitilist rolli tehnoloogia valikul ning komponendid on võimalik valida hinnangute järgi;
- Kogukonna toetus on tähtis ning raamistik *AngularJS* omab kõige suuremat toetust kogukonna poolt ja *ReactJS* kõige väiksemat;
- Sõltumata *JavaScript* raamistiku valikust on võimalik rakendada hulk parimaid praktikaid rakenduse tõhusaks arendamiseks.

Edasiseks töödeks on rakenduste arendamiseks uute tehnoloogiate kasutamine, nagu *AngularJS* 2.0 ja *EcmaScript* 6.0, millel on suur potentsiaal.

Summary

The goal of this thesis was to analyze nowadays-popular JavaScript frameworks in order to develop structured SPA applications. Comparison is made between *AngularJS*, *EmberJS*, *ReactJS* and *BackboneJS* frameworks.

The main results of the thesis are the following:

- Analyze *AngularJS*, *EmberJS*, *ReactJS* and *BackboneJS* frameworks;
- Searching of popular open-source components and development of prototypes based on analyzed frameworks with found components;
- The comparison of similar functionality between prototypes and emphasizing strong and the weak points of frameworks.

Results of the research have led to the conclusion of the following points:

- The support of community plays critical role in choosing the technology. Components should be chosen based on the users' rating;
- The support of community is important. *AngularJS* has the most and *ReactJS* has the least 3rd party components and community support;
- Despite of the *JavaScript* framework there are certain best practices, which can be used in productive development of application.

For further development, new technologies like *AngularJS* 2.0 and *EcmaScript* 6.0 have the great potential.

Kasutatud kirjandus

[1] [WWW] Model-View-Controller.
http://whatis.techtarget.com/definition/model-view-controller-MVC (06.05.2015)

[2] [WWW] Model-View-ViewModel.
https://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40).aspx (06.05.2015)

[3] [WWW] What is open source?
http://opensource.com/resources/what-open-source (06.05.2015)

[4] S. Chacon, B. Straub. Pro Git, Second Edition, 2014

[5] Google Trends Data Goldmine.
http://marketingland.com/google-trend-goldmine-117626 (12.05.2015)

[6] [WWW] Github. About Stars.
https://help.github.com/articles/about-stars/ (06.05.2015)

[7] [WWW] Github. Fork A Repo.
https://help.github.com/articles/fork-a-repo/ (06.05.2015)

[8] [WWW] List of JavaScript libraries.
http://www.infoq.com/research/top-javascript-mvc-frameworks (06.05.2015)

[9] [WWW] Google Trends. Where Trends data comes from.
https://support.google.com/trends/answer/4355213?hl=en&ref_topic=4365599
(06.05.2015)

[10] [WWW] Full documentation on Angular framework.
https://docs.angularjs.org (08.05.2015)

[11] A. Lerner. Ng-book – The complete book on AngularJS, 2013

[12] J. Cravens, T. Q. Brady. Building Web Apps with Ember.js, July 2014

[13] [WWW] React. Why React?
https://facebook.github.io/react/docs/why-react.html (06.05.2015)

[14] [WWW] React. React (Virtual) DOM Terminology.
https://facebook.github.io/react/docs/glossary.html (06.05.2015)

[15] [WWW] React. JSX in Depth.
http://thinkingonthinking.com/the-rendering-question/ (12.05.2015)

[16] [WWW] React. Reusable Components.
https://facebook.github.io/react/docs/reusable-components.html (06.05.2015)

[17] [WWW] React.js tutorial - Creating components
http://ghost.stevenisekimart.in/react-js-tutorial-creating-components/ (08.05.2015)

[18] [WWW] Introduction to React.js.
http://devangelist.de/en/react-js/ (12.05.2015)

[19] [WWW] Getting To Know Flux, the React.js Architecture.
https://scotch.io/tutorials/getting-to-know-flux-the-react-js-architecture (06.05.2015)

[20] [WWW] Understand JavaScript Callback Functions and Use Them.
http://javascriptissexy.com/understand-javascript-callback-functions-and-use-them/
(06.05.2015)

[21] [WWW] Backbone. Home page, full documentation.
http://backbonejs.org/ (06.05.2015)

[22] [WWW] Backbone.Marionette v2.4.1, full documentation.
http://marionettejs.com/docs/v2.4.1/ (06.05.2015)

[23] [WWW] Introduction to npm.
http://howtonode.org/introduction-to-npm (12.05.2015)

[24] [WWW] Bower. Getting Started with Bower Package Manager.
https://www.codementor.io/bower/tutorial/beginner-tutorial-getting-started-bower-package-manager (12.05.2015)

[25] [WWW] What is end to end testing and How is it done?
http://www.geekinterview.com/question_details/49899 (06.05.2015)

[26] P. J. Agboado. Ember.js Guides - The guide for building Ambitious Web Applications, 2015.
https://leanpub.com/emberjsguides/read (08.05.2015)

[27] [WWW] Ember-cli. The command line interface for ambitious web applications.
http://www.ember-cli.com/ (06.05.2015)

[28] [WWW] RequireJS. Why AMD?
http://requirejs.org/docs/whyamd.html (06.05.2015)

[29] [WWW] Getting started with Browserify.
http://www.sitepoint.com/getting-started-browserify/ (06.05.2015)

[30] [WWW] NodeJS.
https://nodejs.org/ (06.05.2015)

[31] [WWW] Transparent Compilation.
http://martinfowler.com/bliki/TransparentCompilation.html (12.05.2015)

[32] [WWW] Model-View-Controller.
https://msdn.microsoft.com/en-us/library/dd381412(v=vs.108).aspx (08.05.2015)

[33] [WWW] Fun with AngularJS.
http://devgirl.org/2013/03/21/fun-with-angularjs/ (08.05.2015)

[34] [WWW] AngularJS fundamentals.
https://grekai.wordpress.com/2013/05/07/angular-js/ (08.05.2015)

[35] [WWW] Developing Backbone.js Applications.
http://addyosmani.github.io/backbone-fundamentals/ (08.05.2015)

[36] [WWW] Building robust web apps with React: Part 1, in-browser prototypes
http://maketea.co.uk/2014/03/05/building-robust-web-apps-with-react-part-1.html (08.05.2015)

[37] [WWW] Use ECMAScript 6 Today.
http://code.tutsplus.com/articles/use-ecmascript-6-today--net-31582 (12.05.2015)