TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

Siim Salin 143029IABB

ANGULAR 2 JA REACTJS KLIENDIPOOLSETE RAAMISTIKKUDE ANALÜÜS JA VÕRDLUS VÄIKSEMATE ÜHELEHEVEEBIRAKENDUSTE KORRAL

Bakalaurusetöö

Juhendaja: Inna Švartsman
MSc
Lektor

Tallinn 2017

Autorideklaratsioon

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

Autor: Siim Salin

22.05.2017

Annotatsioon

Käesoleva bakalaurusetöö „Angular 2 ja ReactJs kliendipoolsete raamistikkude analüüs ja võrdlus väiksemate üheleheveebirakenduste korral“ eesmärgiks on teha selgeks, mida tuleks enne Angular 2 ning ReactJs raamistike kasutuselevõttu arendajal meeles pidada ning leida, milline raamistik sobib paremini lihtsamate üheleheveebirakenduste korral.

Töös koostatakse kaks sama funktsionaalsusega Todo veebirakendust kasutades Angular 2 ning ReactJs raamistikke. Rakenduste implementatsiooni käigus uuritakse erinevaid aspekte, mis toovad välja antud raamistike plussid ning miinused.

Töö tulemusena tekib ülevaade, mille abil on võimalik hinnata raamistike kasutatavust ning raamistiku juurutamisest tulenevaid eeliseid ja puudusi.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 39 leheküljel, 7 peatükki, 36 joonist, 1 tabelit.

Abstract

The analysis and comparison of Angular 2 and ReactJs front-end frameworks in smaller single-page web applications

The main purpose of this bachelor thesis, „The analysis and comparison of Angular 2 and ReactJs front-end frameworks in smaller single-page web applications“, is to find out, what to keep in mind before starting developing in Angular 2 or ReactJs framework and which framework is better for smaller single-page web applications.

In this thesis, two Todo web applications will be implemented using Angular 2 and ReactJs frameworks. During the implementation, different aspects will be covered, which will bring out pros and cons of each framework.

The result of the thesis will give an overview, which helps to measure the usability of these frameworks.

The thesis is in Estonian and contains 39 pages of text, 7 chapters, 36 figures, 1 tables.

Lühendite ja mõistete sõnastik

AJAX	*Asynchronous JavaScript And XML* Kogum omavahel seotud veebiarenduse tehnikaid, mis on kasutuses rakenduse kliendi poolel, et luua interaktiivseid veebirakendusi.
API	*Application Programming Interface* Reeglistik olemasoleva valmisprogrammiga suhtlemiseks
CBA	*Component-Based Architecture* Komponentidel põhinev veebirakenduse arhitektuur.
CLI	*Command Line Interface* Programmiga suhtlemise liides, mis kasutab käsurida.
HTTP	*Hypertext Transfer Protocol* Protokoll teabe edastamiseks arvutivõrkudes.
MIT	*Massachusetts Institute of Technology* Vabavara tarkvara litsents, mis on pärit Massachusettsi Tehnoloogiainstituudist.
MVC	*Model-View-Controller* Arhitektuurimudel, mida kasutatakse veebirakendustes.
REST	*Representational State Transfer* Tarkvaraarhitektuuri stiil, mida kasutatakse veebis.
SPA	*Single-Page Application* Üheleheveebirakendus, kus muudetakse komponente ilma lehte uuesti laadimata.
JSX	*Java Serialization to XML* Programmeerimiskeel, mida kasutatakse ReactJs raamistikus.
XML	*eXtensible Markup Language* W3C väljatöötatud ja soovitatud standardne üldotstarbeline märgistuskeel.

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

Kliendipoolsed raamistikud on muutunud uueks trendiks veebirakenduste arenduses. Viimasel ajal on tekkinud mitmeid uusi JavaScript'i raamistikke, mis põhinevad MVC (*Model-View-Controller*) ja/või CBA (*Component-Based Architecture*) struktuuridel. Kaks populaarsemat raamistikku on Angular 2 ning ReactJs. Raamistiku valik määrab ära edasise töö sujuvuse, kuluva aja ning lihtsuse, kuid see võib olla tarkvaraarendajale keeruline ning segadust tekitav.

Töö eesmärgiks on teha selgeks, mida tuleks enne Angular 2 ning ReactJs raamistike kasutuselevõttu meeles pidada ning leida, milline raamistik sobib kõige paremini lihtsamate SPA (*Single-Page Application*) rakenduste korral.

Töö käigus implementeeritakse struktureeritud Todo veebirakendus kasutades Angular 2 ning ReactJs raamistikke. Mõlema raamistiku rakenduses kasutatakse Redux'i struktuurihaldusteeki ning rakendust testitakse kasutades Karma testimisraamistikku, mida detailselt antud uurimuses ei kajastata. Rakenduse implementeerimise käigus uuritakse järgmisi aspekte:

- Kogukonna tugi
- Projekti loomise lihtsus
- Rakenduse arhitektuur
- Koodi struktuur
- Õppimise kiirus
- Vigade leidmine
- Rakenduse kiirus
- Ühiktestimine

Uuringu tulemusena tekib ülevaade, mille abil on võimalik hinnata raamistike kasutatavust ning raamistiku juurutamisest tulenevaid eeliseid ja puudusi.

2 JavaScript ja raamistikud

JavaScript on Netscape'i loodud programmeerimiskeel, mida kasutatakse peamiselt veebilehtede skriptimiseks. JavaScript toetab programmeerimist objektorienteeritult ning on ECMAScripti standardiga defineeritud keele ülemhulk [1]. JavaScript'i kompileeritakse kliendipoolel, mida saab kasutada selleks, et määrata, kuidas veebilehekülg reageerib sündmustele. AJAX'i (*Asynchronous JavaScript And XML*) ehk asünkroonse veebirakenduse tehnoloogia kasutuselevõtuga, saavad rakendused saata ja vastu võtta andmeid asünkroonselt ilma kasutajaliidest muutmata. AJAX päringuid tehakse RESTful teenuste abil, milles lühend REST (*Representational State Transfer*) tähistab üle HTTP (*Hypertext Transfer Protocol*) toimuva kliendi ja serveri vahelise suhtluse arhitektuuri [2]. 2006. aasta veebruaris lisati jQuery raamistikku AJAX funktsionaalsus, mis võimaldab dokumendiobjektide mudeliga (DOM) efektiivsemalt manipuleerida ning AJAX funktsionaalsuse jaoks kergemini koodi luua [3]. Üheleheveebirakenduste (SPA) implementeerimine lihtsustus. SPA kujutab endas veebirakendust, kus lehekülje loogika on saadaval ühe laadimisega. Rakenduse elemendid uuendatakse dünaamiliselt. Rakendust ei laadita uuesti terviklikult vaid muudetakse ainult elemente, mis peavad muutuma vastavalt sündmusele. Selline tarkvaralahendus toob kliendi poolele palju koodi ning loogikat, mille efektiivseks haldamiseks läheb vaja disainimustrit. Selle probleemi lahendamiseks on JavaScript'i maailma jõudnud mitmed erinevad arhitektuurimustrid – tuntuimad neist mudel-vaade-kontroller (MVC) ja komponentidel põhinev arhitektuur (CBA).

2.1 MVC ja CBA struktuurid

JavaScript'i populaarsuse tõusu tõttu on kliendipoolsed rakendused muutunud keerulisemaks kui varem. Et koodi struktuur oleks loogiline ning seda oleks kerge hallata, on loodud erinevad disainimustrid, mida kasutatakse rakenduse loomisel. Tuntuimad neist on MVC (*Model-View-Controller*) ning CBA (*Component-Based Architecture*).

2.1.1 MVC

MVC on tarkvara disainimuster veebirakendustes, mis koosneb kolmest osast: mudel, vaade ja kontroller. Järgnevalt täpsustatakse, millega antud osad tegelevad.

Mudelid haldavad rakenduse andmeid. Neid ei peeta kasutajaliidese ega ka vaate kihtideks, kuid aitavad hallata andmeid, mida eelnimetatud kihid võivad kasutada. Mudeli muutusest teavitatakse ka selle kasutajaid [4].

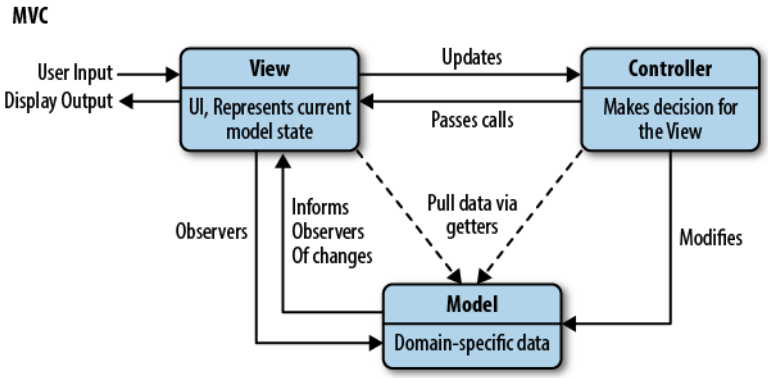
Kasutades mudeleid äriloogilistes rakendustes, on tavaliselt tahetud, et mudelid oleksid püsivad – see võimaldab muuta ja uuendada mudeleid andmetega, mida konkreetseks tegevuseks vaja on [4].

Tüüpiliseks mudeli kasutajaks on vaade, mis võib muutuda vastavalt sündmusele. Vaateks nimetatakse visuaalse presentatsiooni kihti, mis kuvab filtreeritud vaadet mudelite hetkeolekust. JavaScript'i vaated koosnevad DOM (*Document Object Model*) elementidest, mis jälgib mudelit – kui mudel muutub, siis vaadet teatatakse sellest, mis võimaldab vaatel uuendada ennast vastavalt mudeli muutusele [4].

Kui rakenduse kasutaja kutsub esile muudatuse läbi vaate, siis selle realiseerimise ülesanne jääb kontrollerile. Kontroller teeb otsuse, mida kasutaja tehtud muudatusega edasi teha [4]. Selline arhitektuur soodustab koodi taaskasutavust, sest igal vaate komponendil on oma kindel roll.

Kontroller on mudeli ja vaate vahendaja, mis on vastutav mudeli uuendamise eest, kui kasutaja annab vastava käsu vaatelt. Enamus JavaScript MVC raamistikke erineb tavapärastest MVC raamistikest, mida tunneme Spring MVC raamistikust, sest tavapärastes MVC raamistikes on kontrollerid serveri poolel, kuid JavaScript MVC puhul on kontroller tavaliselt kliendipoolel [4].

Järgnevalt on toodud joonis MVC osade suhtluse kohta (Joonis 1).



Joonis 1. MVC osade suhtlus [4].

Antud joonis võtab kokku osade vahelise suhtluse ning nende rollid, mida kasutatakse MVC põhistes raamistikes.

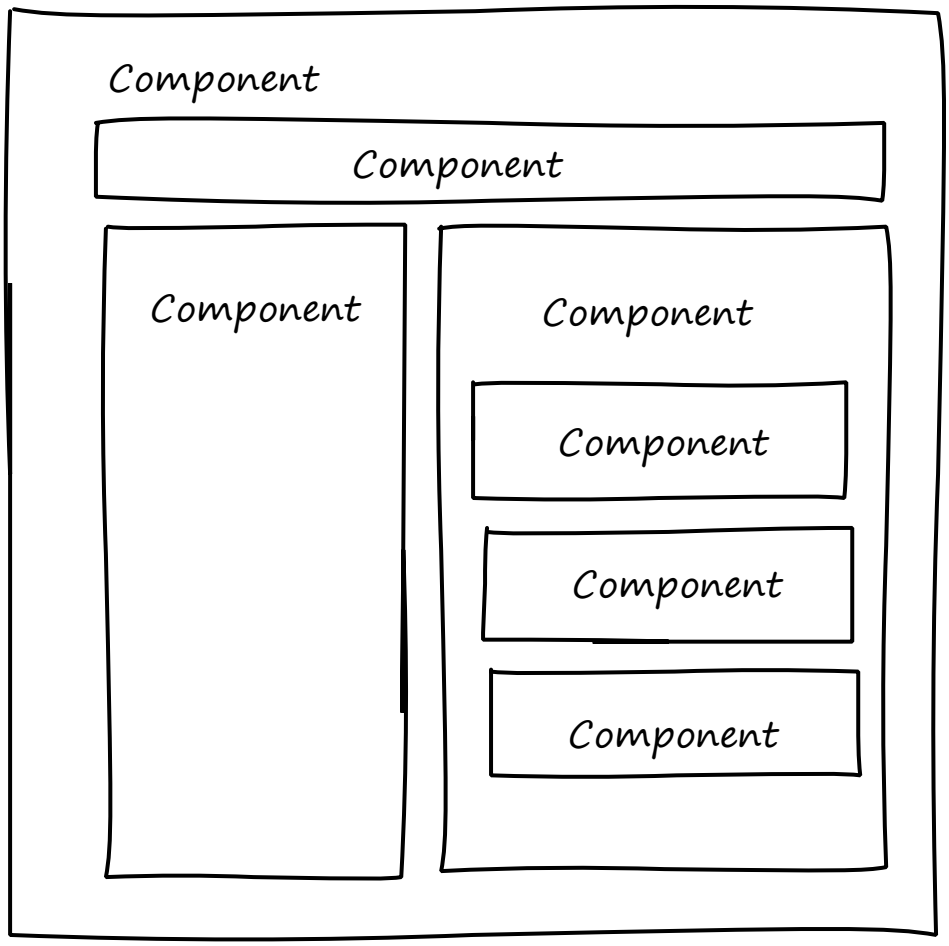
2.1.2 CBA

2013. aastal andis Facebook välja uue teegi ReactJs, mis tõi JavaScript'i maailma CBA, mis on arhitektuuri meetod suure kasutajaliidese kapseldamiseks väiksemateks iseseisvateks mikrosüsteemideks, mida kutsutakse komponentideks [5]. Komponendid annavad võimaluse kirjutada väikestest osadest koosneva järjepideva API (*Application Programming Interface*), mida saab kergesti taaskasutada.

Komponentide loogika sarnaneb AJAX päringutele, kus päring saadetakse serverile kliendi poolelt, mis võimaldab uuendada DOM-i dünaamiliselt ilma lehte uuesti laadimata. Kõikidel komponentidel on oma liides, millega saab teha päringuid serverile. Kuna komponendid on iseseisvad, saab üks komponent uueneda ilma teist komponenti segamata [5].

CBA nõuab, et kõik meetodid ja API-d, mis on seotud antud komponendiga, peavad olema selle komponendi struktuuris [5].

Järgnevalt on illustreeritud komponentidest koosnev veebileht (Joonis 2).



Joonis 2. Komponentidel põhinev veebileht.

Antud joonis (Joonis 2) illustreerib, kuidas saab komponentidel põhineva veebilehe üles ehitada.

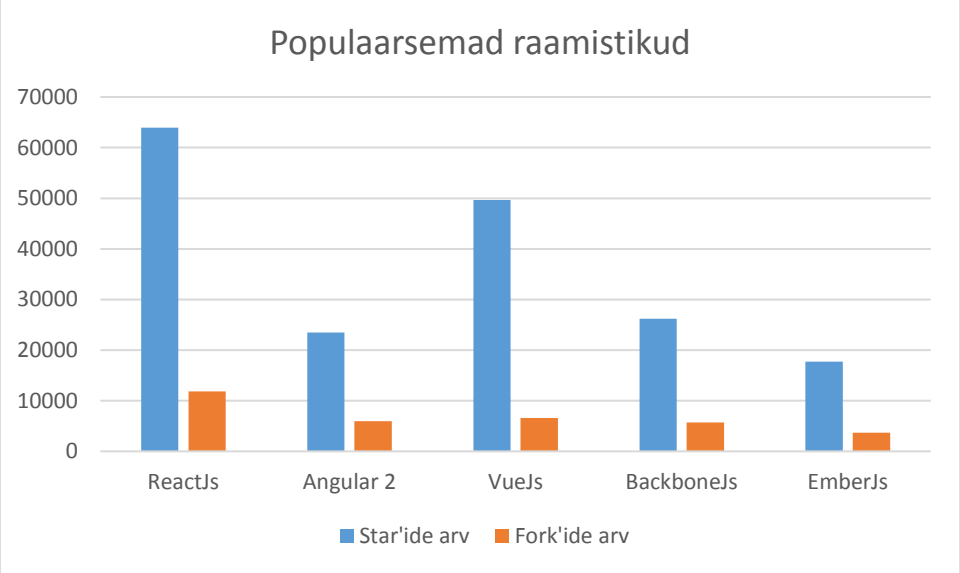
2.2 Kaasaegsete raamistike võrdlus

Järgnevalt võrreldakse raamistikke, mis on kasutajate hulgas saavutanud populaarsuse ning mis kasutavad MVC ja/või CBA disainimustrit, kasutades GitHub'i, Google'i ning Stack Overflow andmeid.

GitHub on veebipõhine Git repositooriumite hostimise teenus. Järgnevalt on kasutatud GitHub'i ressursi kontekstis olevaid mõisteid *rating* ja *fork*. *Rating* näitab inimeste arvu, kes on jätnud positiivse tagasiside hinnangu, vajutades „star" nupule GitHub'i veebikeskonnas [6]. *Fork* näitab inimeste arvu, kes tegid projektist koopia eesmärgiga

seda edasi arendada. *Fork* on repositooriumi kopeerimine, mis võimaldab eksperimenteerida muudatustega ilma originaalprojekti muutmata [7].

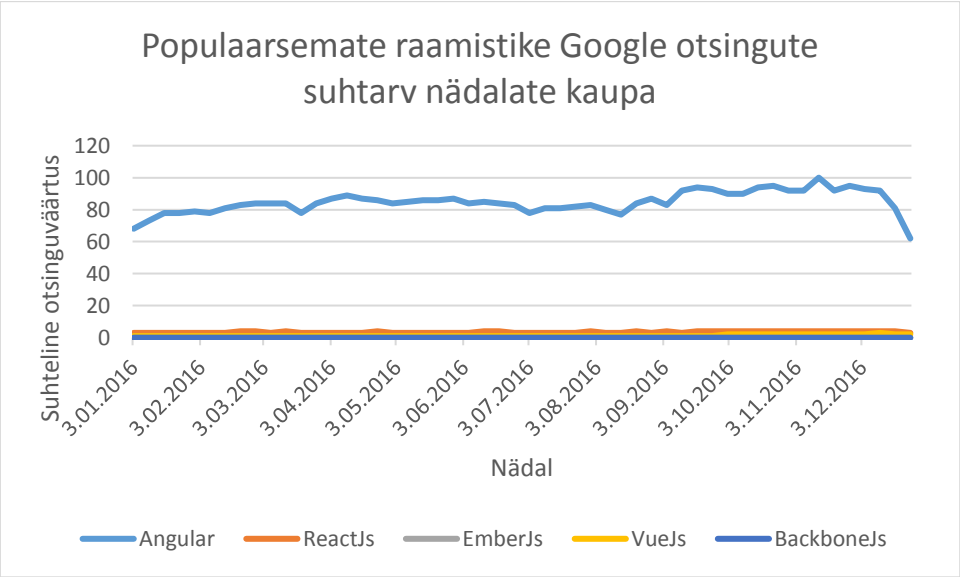
Järgnevalt on toodud *star*'ide ning *fork*'ide arv vastavalt GitHub'i populaarsematele raamistikele, mis kasutavad MVC ja/või CBA disainimustrit (Joonis 3).



Joonis 3. Populaarsemad raamistikud vastavalt *star*'ide ning *fork*'ide arvule [8].

Kõige populaarsem raamistik *star*'ide ning *fork*'ide arvult on ReactJs.

Järgnevalt on toodud statistika Google'i otsingumootori kasutamise kohta 2016. aasta jooksul. Selleks kasutati Google Trends võrdlusfunktsiooni (Joonis 4).

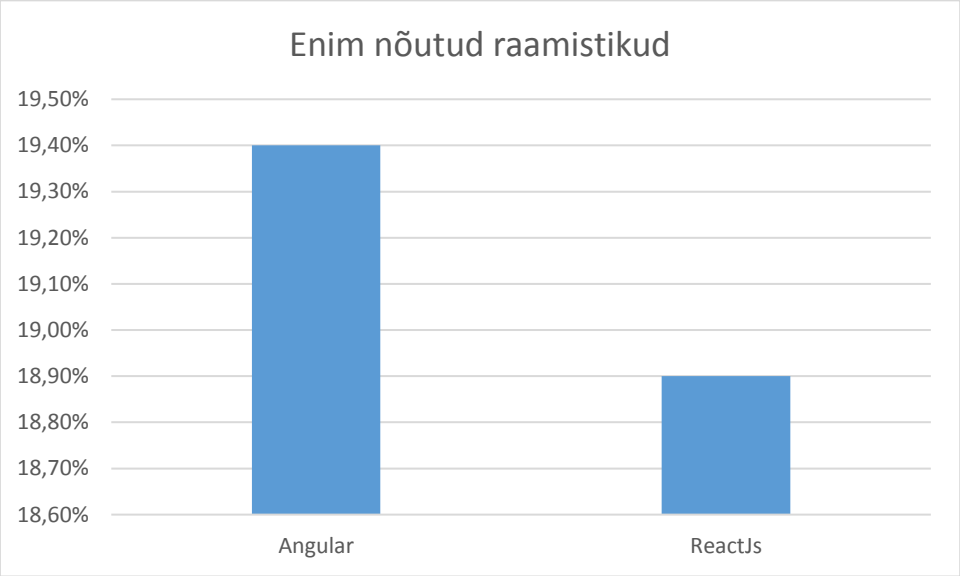


Joonis 4. Populaarsemate raamistike Google otsingute suhtarv nädalate kaupa [9].

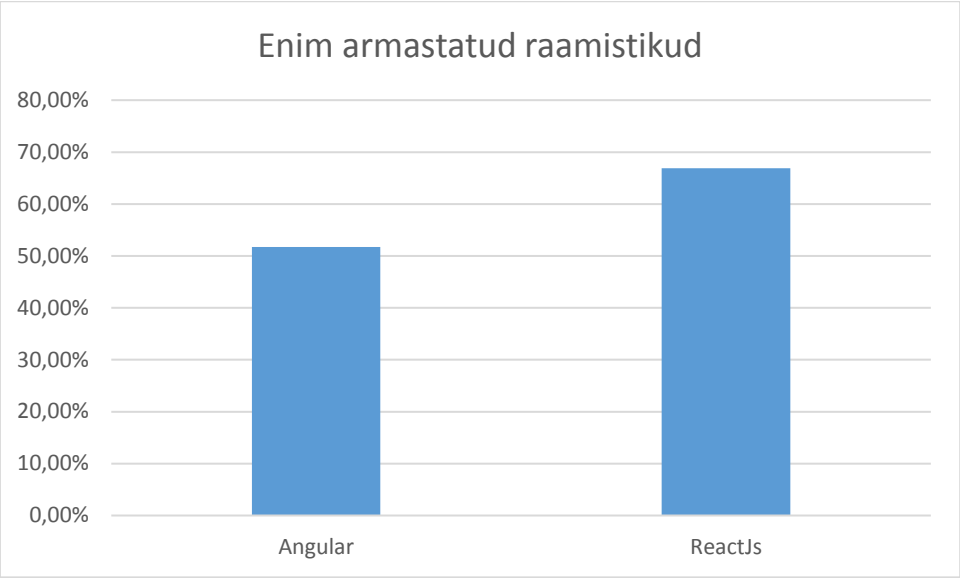
Kõige populaarsem otsingusõna on „angular“, mille otsinguväärtus kahanes 2016. aasta lõpul, kuid siiski hoiab kindlalt oma kohta. Sellele järgneb otsingusõna „reactjs“, mille otsingute arv on olnud aasta jooksul stabiilne.

Järgnevalt vaadeldakse Stack Overflow andmeid võrreldes ReactJs ning Angular raamistikku.

Stack Overflow on maailma suurim veebikommuun, mis on mõeldud programmeerijatele õppimiseks, koodi jagamiseks ning karjäärinõustamiseks. Seal on üle 7 miljoni kasutaja [10]. Järgnevalt on toodud Angular'i ja ReactJs'i võrdlus Stack Overflow küsitluses, mida viiakse läbi iga aasta Jaanuaris (Joonis 5, Joonis 6). 2017. aastal oli vastanuid 46 000 kasutajat [11].



Joonis 5. Enim nõutud raamistikud Stack Overflow küsimustiku põhjal [11].



Joonis 6. Enim armastatud raamistikud Stack Overflow küsimustiku põhjal [11].

Vastanute arvates on enim nõutud JavaScript'i raamistikuks Angular, kuid enim armastatud raamistikuks ReactJs.

Eelnevate andmete põhjal on sügavamaks uurimiseks ning Todo rakenduse implementatsiooniks valitud ReactJs ning Angular 2 raamistikud koos Redux struktuurihaldusteegiga, mida kasutatakse peamiselt koos ReactJs raamistikuga. Implementatsiooni ühtlustamiseks kasutatakse Redux struktuurhaldusteeki ka Angular 2 raamistikuga tehtavas Todo rakenduses.

3 Valitud raamistike funktsionaalus

Antud peatükis uuritakse Angular 2, ReactJs raamistikke ning Redux struktuurihaldusteeki, mis sobib hästi Todo rakenduse olekustruktuuriks.

3.1 Raamistik Angular 2

Angular 2 on TypeScript'i põhjal loodud avatud lähtekoodiga veebiraamistik, mida arendab Angular'i meeskond Google's. Versioon 2.0.0 avaldati 14. Septembril 2016. Tänaseks on väljas ka Angular 4.0.0 stabiilne versioon, mis avaldati 18. Märtsil 2017 [12].

Antud töös kasutatakse Angular 2.4.0 versiooni, mis oli kõige uuem stabiilne versioon Todo rakenduse loomise hetkel. Järgnevates peatükkides on toodud Angular 2 tähtsaimad omadused, arhitektuur ja selle osad ning metoodikad realiseerimiseks.

3.1.1 Raamistiku programmeerimiskeel - TypeScript

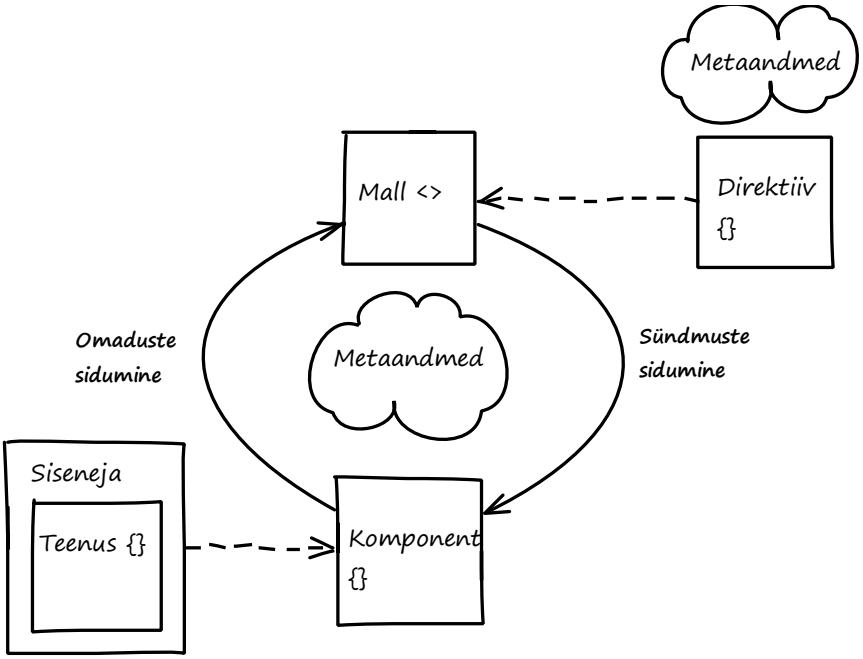
TypeScript on programmeerimiskeel, mis põhineb JavaScript'il ning mida arendatakse Microsoft'i poolt. TypeScript lisab tüübid, klassid ning moodulid JavaScript'i. Toetatud on kõik tuntuimad veebibrauserid ning operatsioonisüsteemid. TypeScript'i kood kompileeritakse loetavaks ning standarditele vastavaks JavaScript koodiks [13].

3.1.2 Angular 2 arhitektuur

Lihtsustades kirjutatakse rakendus järgnevate osadena [14]:

- HTML mallid (*templates*), milles kasutatakse Angular'i metaandmeid.
- Komponendid (*components*), mis haldavad malle.
- Teenused (*services*), kus hoitakse rakenduse loogikat.
- Kokkuvõtvad moodulid (*modules*), mis kasutavad komponente ning teenuseid.

Seejärel jooksutakse rakendust käivitades juurmoodulit. Angular teeb ülejäänud töö ning rakendus on saadaval kliendis [14].



Joonis 7. Lihtsustatud Angular 2 raamistiku arhitektuur [14].

Järgnevalt seletatakse põhjalikumalt Angular 2 arhitektuuri osasid.

Moodulid

Angular'i rakendused koosnevad moodulitest – tavaliselt nimetusega „NgModules". Igal Angular'i rakendusel on juurmoodul – tavaliselt nimetusega „AppModule" [14]. Moodulid on hea viis organiseerida rakendust ning laiendada seda teekidega. Mitmed teegid on moodulid – näiteks *FormsModule*, *HttpModule* ja *RouterModule* [15]. Väiksemad rakendused võivad koosneda ainult juurmoodulist, kuid enamustel on veel mitu moodulit – iga neist on mingi konkreetse domeeni jaoks [14].

Moodulid on tähistatud *@NgModule* dekoraatoriga. Angular'il on mitu dekoraatorit, mis lisavad metaandmeid klassidesse, et Angular saaks aru, mida need klassid tegema peaksid [14]. *@NgModule* on dekoraator funktsioon, mis võtab metaandmete objekti, mille omadused iseloomustavad moodulit [14]. Järgnevalt tuuakse välja *@NgModule* dekoraator objekti tähtsamad omadused:

- Deklaratsioonid (*declarations*) – vaate klassid, mis kuuluvad sellesse moodulisse [14].

- Ekspordid (*exports*) – alamhulk deklaratsioonidest, mis peavad olema nähtavad ja kasutatavad teiste moodulite komponentide mallides [14].

- Impordid (*imports*) – moodulid, mille väljutatud klassid on vajalikud nende komponentide mallides, mis on deklareeritud vastavas moodulis [14].

- Tarnijad (*providers)* – teenused, mida moodul lisab globaalsete teenuste hulka – neid teenuseid on võimalik kasutada rakenduse igas osas [14].

- Eellaadimine (*bootstrap*) – juurkomponent, mis koosneb kõikidest teistest rakenduse vaadetest. Ainult juurkomponendil peaks olema eellaadimise väli [14].

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
@NgModule({
    imports:      [ BrowserModule ],
    providers:    [ Logger ],
    declarations: [ AppComponent ],
    exports:      [ AppComponent ],
    bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

Joonis 8. Angular 2 juurmooduli näide.

Antud näites (Joonis 8) on toodud juurmoodul, mis kasutab eelnimetatud *@NgModule* dekoraatorit koos metaandmetega. Et Angular'i kasutada, tuleb importida vajalikud teegid.

Angular'i teegid

Angular koosneb JavaScript'i moodulite kollektsioonist. Neid nimetatakse teegi (*library*) mooduliteks. Iga Angular'i teegi nimi algab *@angular* eesliitega, mida saab installida *npm* paketiga ning importida „import" käsuga [14].

```
import { Component } from '@angular/core';
```

Joonis 9. Angular 2 komponendi dekoraatori importimise näide.

Importida saab ükskõik millist Angular'i teeki kasutades näites (Joonis 9) toodud struktuuri.

Komponendid

Komponent (*component*) on kontrolleri klass koos malliga, mis tegeleb vaate ning rakenduse loogikaga. Komponendiga on võimalik seadistada stiili malle, mida kasutab vaade [16].

Komponendi registreerimiseks kasutatakse *@Component* dekoraatorit, mis võtab sisendiks metaandmeid komponendi kohta [16].

Järgnevalt on toodud näide komponendi kohta rakenduses (Joonis 10).

```
import { Component } from '@angular/core';

@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.scss'],
})
export class AppComponent {

    constructor() {};

    ngOnInit() {}
}
```

Joonis 10. Angular 2 komponendi näide.

Angular loob, värskendab ning hävitab komponente vastavalt kasutaja sisendile. Rakendus võib muutuda vastavalt elutsükli konksude implementatsioonile [16]. Ülalnäidatud näites (Joonis 10) on kasutatud *ngOnInit* konksu.

Komponendi vaade defineeritakse kasutades malle (*templates*), mis ütlevad Angular'ile, kuidas kuvada komponenti [16].

Mallid

Järgnevalt on toodud näide Angular 2 mallist (Joonis 11).

```
<app-todo
        *ngFor="let todo of todos"
        [todo]="todo"
</app-todo>
<p *ngIf="!todos.length">Nothing to do</p>
```

Joonis 11. Angular 2 malli näide.

Kuigi mallid kasutavad tavalisi HTML elemente nagu „<p>“, siis Angular’i mallides kasutatakse ka spetsiifilist koodi. Antud koodinäites (Joonis 11), on direktiivid „*ngIf“, „*ngFor“ defineeritud Angular’i malli süntaksis. „[todo]=’todo’“ kasutatakse andmete sidumiseks. „<app-todo>“ on element, mis viitab antud juhul Todo komponendile. Antud element sulandub kokku HTML baaselementidega samas mallis. Andmete kasutamiseks mallis on vajalik andmete sidumine (*data binding*).

Andmete sidumine

Andmete sidumine on andmete koordineeritud liigutamine allika ning HTML elementide vahel. Andmete sidumise funktsionaalsus seob mallide osad vastavate osadega komponendis [16]. Angular’is on olemas neli andmete sidumise võimalust. Igal võimalusel on suund. Järgnevalt on toodud andmete sidumise võimalused.

- Interpolatsioon (*interpolation*).

  Järgnevalt on toodud näide interpolatsioonist (Joonis 12), milles kuvatakse HTML elemendi „<p>“ sees *todo* objekti muutujat „text“.

```
<p>
    {{todo.text}}
</p>
```

Joonis 12. Angular 2 interpolatsiooni näide.

- Omadusi siduv (*property binding*).

  Järgnevalt on toodud näide omadusi siduvast andmete sidumisest (Joonis 13), milles antakse komponendi objekt „todo“ edasi alamkomponendile.

```
<app-todo
    *ngFor="let todo of todos"
    [todo]="todo"
</app-todo>
```

Joonis 13. Angular 2 omadusi siduva andmete sidumise näide.

- Sündmusi siduv (*event binding*).

Järgnevalt on toodud näide sündmusi siduvast andmete sidumisest (Joonis 14), milles kutsutakse välja komponendi funktsioon *toggleTodo*, kui elemendi „div“ peal klikitakse.

```
<div class="todo" (click)="toggleTodo()"
    <p>{{ todo.text }}</p>
</div>
```

Joonis 14. Angular 2 sündmusi siduva andmete sidumise näide.

- Kahepoolne andmete sidumine (*two-way data binding*).

  Järgnevalt on toodud näide kahepoolsest andmete sidumisest (Joonis 15), mis ühendab omadusi ning sündmusi siduvad meetodid kasutades *ngModel* direktiivi.

```
<input [(ngModel)]="todo.text">
```

Joonis 15. Angular 2 kahepoolse andmete sidumise näide.

Direktiivid

Direktiiv (*directive*) on klass *@Directive* dekoraatoriga. Angular'i komponent on laiendatud direktiiv koos mallile vajalike tunnustega [14]. Direktiive jagatakse kaheks: struktuurne direktiiv ja atribuudi direktiiv [14].

Struktuurne direktiiv lisab ja kustutab DOM elemente [14]. Järgnevalt on toodud näide struktuursest direktiivist (Joonis 16)

```
<app-todo
    *ngFor="let todo of todos"
    [todo]="todo"
</app-todo>
```

Joonis 16. Angular 2 struktuurse direktiivi näide

Antud näites (Joonis 16) kasutatakse **ngFor* direktiivi, mis käitub nagu klassikaline for-tsükkel, moodustades igast *todo* objektist uue „<app-todo>“ elemendi.

Atribuudi direktiiv oskab muuta olemasoleva elemendi käitumist ja kuvatavust [14]. Järgnevalt on toodud näide atribuudi direktiivist (Joonis 17).

```
<input [(ngModel)]="todo.text">
```

Joonis 17. Angular 2 atribuudi direktiivi näide.

Antud näites (Joonis 17) implementeerib *ngModel* direktiiv kahepoolset andmete sidumist, mis muudab olemasoleva „<input>“ elemendi nähtavat väärtust vastavalt sündmusele.

3.2 Raamistik ReactJs

ReactJs (edaspidi React) on Facebook'i poolt arendatud avatud lähtekoodiga teek, eesmärgiga kergendada interaktiivsete olekutega taaskasutatavate kasutajaliidese komponentide loomist. React esindab V tähte lühendis MVC, kus V tähistab vaadet. React kasutab mallides JSX (*Java Serialization to extensible markup language*) programmeerimiskeelt. React'i kasutatakse tihti koos Redux struktuurihaldusteegiga [17]. Järgnevates peatükkides on toodud React'i peamised omadused, arhitektuuri osad ning metoodikad nende realiseerimiseks.

3.2.1 Raamistiku programmeerimiskeel - JSX

JSX on staatiliselt kirjutatud, objekt orienteeritud programmeerimiskeel, mis on disainitud veebibrauseritele. JSX'i arendatakse DeNA's uurimistööna [18]. Järgnevalt on toodud JSX'i iseloomustavad karakteristikud [18].

- Kiire – JSX teostab optimiseerimisprotsesse, kui lähtekoodi JavaScript'i koodiks kompileeritakse. Genereeritud kood jookseb kiiremini kui tavaline JavaScript'i kood.
- Turvalisem – JSX on staatiliselt kirjutatud ja tüübikindel. Arendusprotsess on turvalisem, sest mitmed vead leitakse juba kompileerimisprotsessis.
- Lihtsam – JSX pakub klassi arhitektuuri.

3.2.2 ReactJs arhitektuur

React esindab vaate osa MVC arhitektuuris. Vaade on jaotatud komponentideks. Järgnevalt on toodud React'i arhitektuuri peamised osad ning nende funktsionaalsus.

Virtuaalne DOM

DOM-i manipuleerimine on modernsete ning interaktiivsete veebilahenduste aluseks. Siiski on see palju aeglasem kui enamus JavaScript'i operatsioone.

React'is kasutatakse virtuaalse DOM-i (*virtual DOM*) metoodikat, kus igale DOM objektile on moodustatud ka virtuaalne DOM-i objekt. Virtuaalne DOM-i objekt esindab originaali [19]. Virtuaalsel DOM-il on kõik samad omadused, mis on ka originaalil, kuid selle manipuleerimine on palju kiirem [19].

DOM-i muutmisel peab React teadma, millal andmed uuendatakse ning millist DOM-i uuendada. React kasutab kuulajate mudelit, et teada, millal andmeid uuendati. Selle tulemusena ei pea React kalkuleerima mida muudeti vaid see info on peale sündmust kohe olemas. DOM-i muutmiseks ehitab React puu skeemiga DOM-ide sõltuvuse ning arvutab välja, millist DOM elementi peaks muutma. Sellega tagatakse, et uuendatakse ainult neid DOM elemente, mida tarvis [20].

Komponendid

React kasutab komponentidel põhinevat arhitektuuri, mis koosneb taaskasutavatest komponentidest (*components*), millest iga komponent teeb ühte sõltumatut tegevust [20].

Järgnevalt on toodud näide React'i komponendi klassi loomisest, milles kasutatakse *React.createClass* funktsiooni (Joonis 18).

```
var MyComponent = React.createClass({
    render: function () {
        return (
            <div>
                <p>{this.props.text}</p>
            </div>
        )
    }
});
```

Joonis 18. React komponendi klassi loomise näide.

Järgnevalt on toodud näide (Joonis 19), kuidas saab eelnevalt koostatud (Joonis 18) komponenti kuvale lisada.

```
ReactDOM.render(
    <MyComponent/>,
    document.getElementById('app')
);
```

Joonis 19. React komponendi kuvale lisamise näide.

React komponendid kasutavad elutsükli meetodeid. Elutsükli meetod on funktsioon, mis käivitub komponendi kindlaksmääratud eluhetkel [21].

Järgnevalt on toodud React komponentide elutsükli meetodid:

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

Lisaks elutsükli meetodile on olemas ka teisi komponentide meetodeid [21].

- *getInitialState* – tagastab vaikimisi oleva oleku.
- *getDefaultProps* – tagastab vaikimisi seadistatud muutuja *props*, kui seda pole edastatud.
- *mixins* – objektide massiiv, mida on kasutatud komponendi funktsionaalsuse laiendamiseks.
- *render* – meetod, kus defineeritakse väljastatava React DOM-i osa ning mida käivitatakse komponendi igal uuendusel.

Props

Komponendi defineerimisel on võimalik lisada komponendile atribuute, mida kutsutakse *props*'ideks. *Props*'idega on võimalik lisada dünaamilisi andmeid [22]. Järgnevalt on toodud näide *props*'i kasutamisest (Joonis 20).

```
var MyComponent = React.createClass({
    render: function () {
        return (
            <div>
                <p>{this.props.text}</p>
            </div>
        )
    }
});
ReactDOM.render(
    <MyComponent text="testText"/>,
    document.getElementById('app')
);
```

Joonis 20. React *props*'ide kasutamise näide.

Antud näites (Joonis 20) kasutatakse mallis *props*'i nimetusega „text", millele antakse väärtus „testText" *ReactDOM.render* meetodis.

Sündmused

Sündmused (*events*) on „sündmused", mis tehakse HTML elementidega. Sündmuseks võib nimetada näiteks klikki HTML elemendi peal.

React'is lisatakse sündmused komponendi omadustena ning need käivitavad meetodi, mis defineeritakse objekti spetsifikatsioonides [23]. Järgnevalt on toodud näide sündmuse kontrollimisest (Joonis 21).

```
export var AddTodo = React.createClass({
    handleSubmit: function (e) {
        e.preventDefault();
        console.log(this.refs.todoText.value);
    },
    render: function () {
        return (
            <form onSubmit={this.handleSubmit}>
                <input type="text" ref="todoText">
                </input>
                <button>Add todo</button>
            </form>
        )
    }
});
```

Joonis 21. React sündmuse kontrollimise näide.

Antud näites (Joonis 21) kasutatakse *onSubmit* sündmust, mida kontrollib *handleSubmit* meetod.

Ilma suunata andmete liikumine

React'is liiguvad andmed ilma suunata kasutades *state* ning *props* objekte. Mitme komponendi hierarhias peaks iga komponent andma edasi olekut läbi *props*'ide [22].

Kahepoolne andmete sidumine on React'is võimalik, kuid sellega kaasneb lisakeerukus. Tüüpilises React'i rakenduses liiguvad andmed ülemkomponendist alamkomponenti, kuid muutes andmeid alamkomponendis, siis ülemkomponendis neid ei muudeta. Et realiseerida kahepoolset andmete sidumist, peab alamkomponent ülemkomponenti teavitama muutusest [24].

3.3 Redux

Redux on JavaScript'i teek, millega hallatakse rakenduse olekuid. Redux on saanud oma ideed Flux teegist, mille funktsionaalsust on oluliselt lihtsustatud. Redux kuulub MIT (*Massachusetts Institute of Technology*) litsentsi alla ning algne autor on Dan Abramov [25].

Redux'it kasutatakse koos teiste raamistikega, et luua rakendusi. Kõige enam on levinud React koos Redux'iga. Antud töös kasutatakse Redux'it nii Angular 2 Todo rakenduses kui ka React Todo rakenduses.

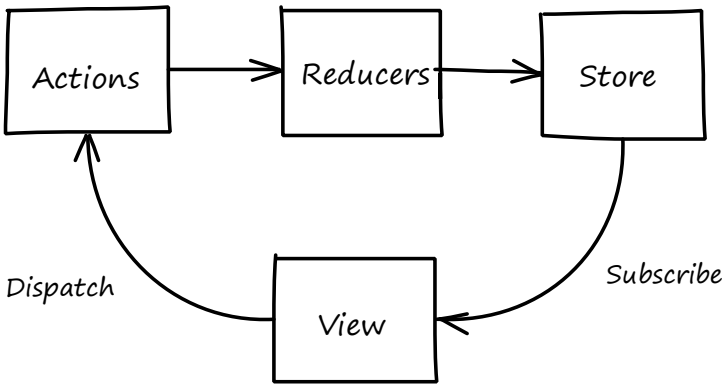
3.4 Redux’i arhitektuur

Redux’i andmete liikumine on rangelt ühesuunaline. Kõik andmed järgivad sama elutsükli reeglistikku, mis muudab rakenduse arusaadavamaks ning paremini prognoositavaks [26].

Andmed liiguvad kasutades järgnevat elutsükli reeglistikku [26]:

- Kutsutakse välja *store.dispatch* meetod, mis võtab sisendiks tegevuse (*action*).
- Redux’i olekuladu (*store*) kutsub välja reduktor (*reducer*) funktsiooni, mille sisendiks on olekupuu ning tegevus.
- Juurreduktor kombineerib väljakutsutud reduktori väljundi üheks olekupuuks.
- Redux’i olekuladu salvestab juurreduktori tagastatud olekupuu.

Järgnevalt on toodud joonis Redux’i arhitektuuri osade sõltuvuste kohta (Joonis 22).



Joonis 22. Redux’i arhitektuuri osade sõltuvus [26].

Antud näites (Joonis 22) toodud arhitektuuri kasutavad kõik Redux’i olekustruktuuri kasutavad rakendused. Järgnevalt on seletatud põhjalikumalt Redux’i arhitektuuri osasid.

Tegevused

Tegevus (*action*) on informatsiooni kogu, mis saadetakse rakendusest olekulattu kasutades *store.dispatch* meetodit [27]. Tegevused on tavalised JavaScript'i objektid, millel peab olema *type* väli, mis määrab ära, millise tegevusega on tegu [27].

```
export var setText = (text) => {
    return {
        type: 'SET_TEXT',
        text
    };
};
```

Joonis 23. Redux'i tegevuse näide.

Antud näites (Joonis 23) on tegevuse tüübiks on „SET_TEXT" ning tegevus tagastab tegevuse tüübi ning sisendina saadud „text" muutuja.

Reduktorid

Tegevused defineerivad ära, et „midagi" juhtus, kuid ei täpsusta, kuidas peab rakenduse olek muutuma. Seda teevad reduktorid (*reducers*).

Reduktor on funktsioon, mis võtab sisendiks eelmise oleku ning tegevuse. Reduktor funktsioon ei tohi muuta sisendiks antud objekti, vaid peab moodustama uue objekti ning tagastama selle. Reduktor ei tohiks teha midagi muud [28].

```
export var textReducer = (state = '', action) => {
    switch (action.type) {
        case 'SET_TEXT':
            return action.text;
        default:
            return state;
    };
};
```

Joonis 24. Redux'i reduktori näide.

Antud näites (Joonis 24) reageerib reduktor siis, kui sisendiks on „SET_TEXT" tüübiga tegevus ning tagastab tegevuse objekti muutuja „text" väärtuse uue olekuna.

Olekuladu

Igal rakendusel peaks olema ainult üks olekuladu (*store*) [29]. Järgnevalt on toodud tegevuste nimekiri, mille eest olekuladu vastutab [29].

- Rakenduse oleku hoidmine.
- Olekule juurdepääsu tagamine kasutades *getState* meetodit.
- Oleku uuendamise tagamine kasutades *dispatch* meetodit.
- Kuulajate registreerimine kasutades *subscribe* meetodit.

Järgnevalt on toodud näide olekulao loomisest (Joonis 25).

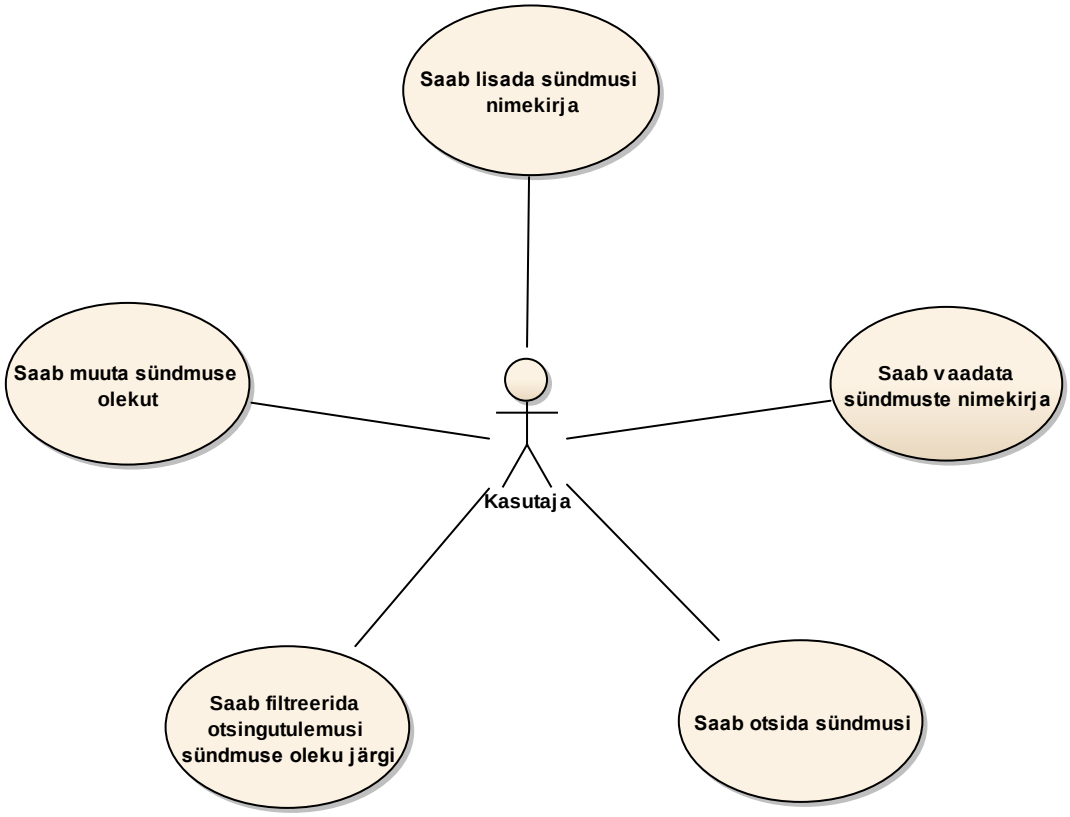
```
var store = redux.createStore(reducer, initialState);
```

Joonis 25. Redux'i olekulao loomise näide [29].

4 Rakenduse realisatsioon

Antud töö käigus realiseeritakse Todo rakendus kasutades Angular 2 ning ReactJs raamistikke. Todo rakendus on mõeldud tegemata ning tehtud sündmuste haldamiseks.

Järgnevalt on toodud kasutusjuhtude diagramm Todo rakenduse kohta (Joonis 26).



Joonis 26. Todo rakenduse kasutusjuhtude diagramm.

Kasutusjuhtude kirjeldused

Kasutusjuht: Saab lisada sündmusi nimekirja.

Tegutseja: Kasutaja.

Eeltingimused: Kasutaja on avanud rakenduse brauseris.

Järeltingimused: Sündmus on lisatud sündmuste nimekirja.

Põhistsenaarium: Kasutaja lisab sündmuse teksti sisendväljale ning vajutab nuppu „Add todo“.

Kasutusjuht: Saab vaadata sündmuste nimekirja.

Tegutseja: Kasutaja.

Eeltingimused: Kasutaja on avanud rakenduse brauseris.

Järeltingimused: Kasutaja näeb sündmuste nimekirja.

Põhistsenaarium: Kasutajale kuvatakse sündmuste nimekiri pealehel.

Kasutusjuht: Saab otsida sündmusi.

Tegutseja: Kasutaja.

Eeltingimused: Kasutaja on avanud rakenduse brauseris.

Järeltingimused: Kasutaja näeb vastavalt otsingu sisendile sündmuste nimekirja.

Põhistsenaarium: Kasutaja sisestab otsingulahtrisse otsitava märksõna või tähe.

Kasutusjuht: Saab filtreerida otsingutulemusi sündmuse oleku järgi.

Tegutseja: Kasutaja.

Eeltingimused: Kasutaja on avanud rakenduse brauseris.

Järeltingimused: Kasutaja näeb vastavalt olekusisendile sündmuste nimekirja.

Põhistsenaarium: Kasutaja valib, mis olekuga sündmusi kuvatakse.

Kasutusjuht: Saab muuta sündmuse olekut.

Tegutseja: Kasutaja.

Eeltingimused: Kasutaja on avanud rakenduse brauseris. Olek on nähtav nimekirjas.

Järeltingimused: Sündmuse olek muutub vastupidiseks.

Põhistsenaarium: Kasutaja vajutab sündmuse rea peale.

Järgnevates alampeatükkides on kirjeldatud Todo rakenduse implementatsiooni vastavalt raamistikule ning nende omadustele.

4.1 Rakenduse realisatsioon kasutades Angular 2 raamistikku.

Antud peatükis kirjeldatakse Todo rakenduse implementatsiooni Angular 2 raamistikuga. Kogu töö käigus valminud Angular 2 Todo rakenduse lähtekood on saadaval järgneval aadressil: https://github.com/habe33/angular2-todo. Oleku hoidmiseks kasutatakse Redux'i olekustruktuuriteeki ning arenduskeskkonnaks kasutati IntelliJ IDEA programmi. Andmebaasiks kasutatakse brauseri lokaalset mälu. Rakenduse jooksutamise abivahendina kasutatakse NodeJs käitussüsteemi, mis võimaldab JavaScript'i abil luua veebiservereid ja võrgutööriistu [30]. Angular 2 implementatsiooni kergendamiseks kasutatatakse Angular CLI (*Angular Command Line Interface*) käitussüsteemi, mis võimaldab lisada rakendusse üldtuntuid Angular 2 osasid kasutades käsurida [31].

Projekti loomiseks kasutatakse järgnevat Angular CLI käsku:

```
ng create angular2-todo –style=scss
```

Sõltuvuste lisamiseks muudetakse konfiguratsiooni faili „package.json" objekte nagu on näidatud alltoodud joonises (Joonis 27).

```
 "dependencies": {
    "@angular/common": "^2.4.0",
    "@angular/compiler": "^2.4.0",
    "@angular/core": "^2.4.0",
    "@angular/forms": "^2.4.0",
    "@angular/http": "^2.4.0",
    "@angular/platform-browser": "^2.4.0",
    "@angular/platform-browser-dynamic": "^2.4.0",
    "@angular/router": "^3.4.0",
    "core-js": "^2.4.1",
    "foundation-sites": "^6.3.1",
    "redux": "^3.6.0",
    "zone.js": "^0.7.6"
  },
  "devDependencies": {
    "@angular/cli": "1.0.0-rc.1",
    "@angular/compiler-cli": "^2.4.0",
    "@types/jasmine": "2.5.38",
    "@types/node": "~6.0.60",
    "codelyzer": "~2.0.0",
    "jasmine-core": "~2.5.2",
    "jasmine-spec-reporter": "~3.2.0",
    "karma": "~1.4.1",
    "karma-chrome-launcher": "~2.0.0",
    "karma-cli": "~1.0.1",
    "karma-coverage-istanbul-reporter": "^0.2.0",
    "karma-jasmine": "~1.1.0",
    "karma-jasmine-html-reporter": "^0.2.2",
    "protractor": "~5.1.0",
    "ts-node": "~2.0.0",
    "tslint": "~4.4.2",
    "typescript": "~2.0.0"
  }
}
```

Joonis 27. Angular2-todo „package.json“ sõltuvused.

Neid sõltuvusi läheb vaja Todo rakenduse implementatsiooniks. Sõltuvuste installeerimiseks kasutatakse alljärgnevat käsku:

```
npm install
```

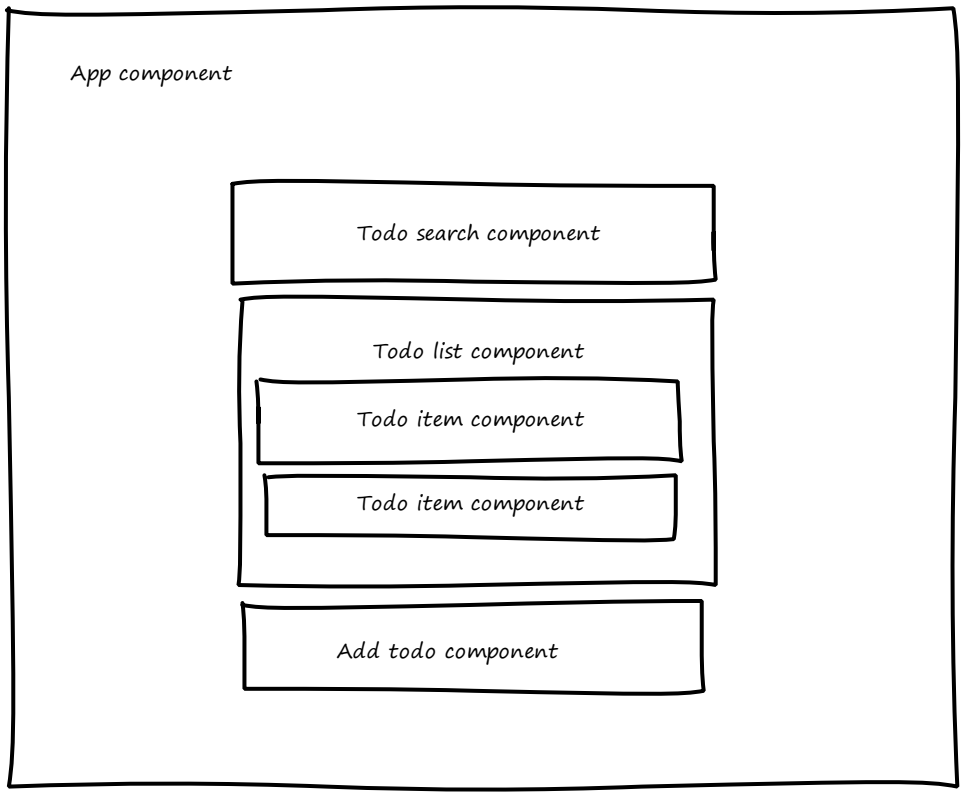
Antud käsk loob *node_modules* kausta koos kõikide defineeritud sõltuvuste failidega. Tehtud toimingutega on lihtsasti moodustatav alusprojekt loodud.

Enne Redux’i olekusüsteemi lisamist on tarvis defineerida *Todo* objekti muutujad ning nende tüübid, et olekusüsteem teaks, millist tüüpi *Todo* objekti laos hoida. Redux’i olekusüsteemi lisamiseks kasutatakse Angular 2 TypeScript’i funktsionaalsust moodustades tüüpidega defineeritud liidesed. Tulemuseks on loogiline ning veakindel

olekusüsteem, mida on lihtne hallata. Lisatakse Redux'i arhitektuuri osad – tegevused, reduktorid ning olekulao abifail. Rakenduse olekulaos kasutatakse Redux'i abifunktsiooni *combineReducers*. Olekuladu moodustatakse kasutades Redux'i abifunktsiooni *createStore*.

Et klassidele ka mujalt rakenduses ligi pääseks, tuleb kasutada sõltuvuste süstimist. Klassile tuleb lisada *@Injectable* dekoraator ning samuti tuleb lisada see klass rakenduse mooduli *@NgModule* dekoraatori metaandmetesse.

Komponentide lisamiseks tuleb visualiseerida rakenduse kasutajaliides komponentidena (Joonis 28).



Joonis 28. Todo rakenduse komponentide visualisatsioon

Komponentide lisamiseks kasutatakse Angular CLI käske:

```
ng generate component addTodo
ng generate component todo
ng generate component todoList
ng generate component todoSearch
```

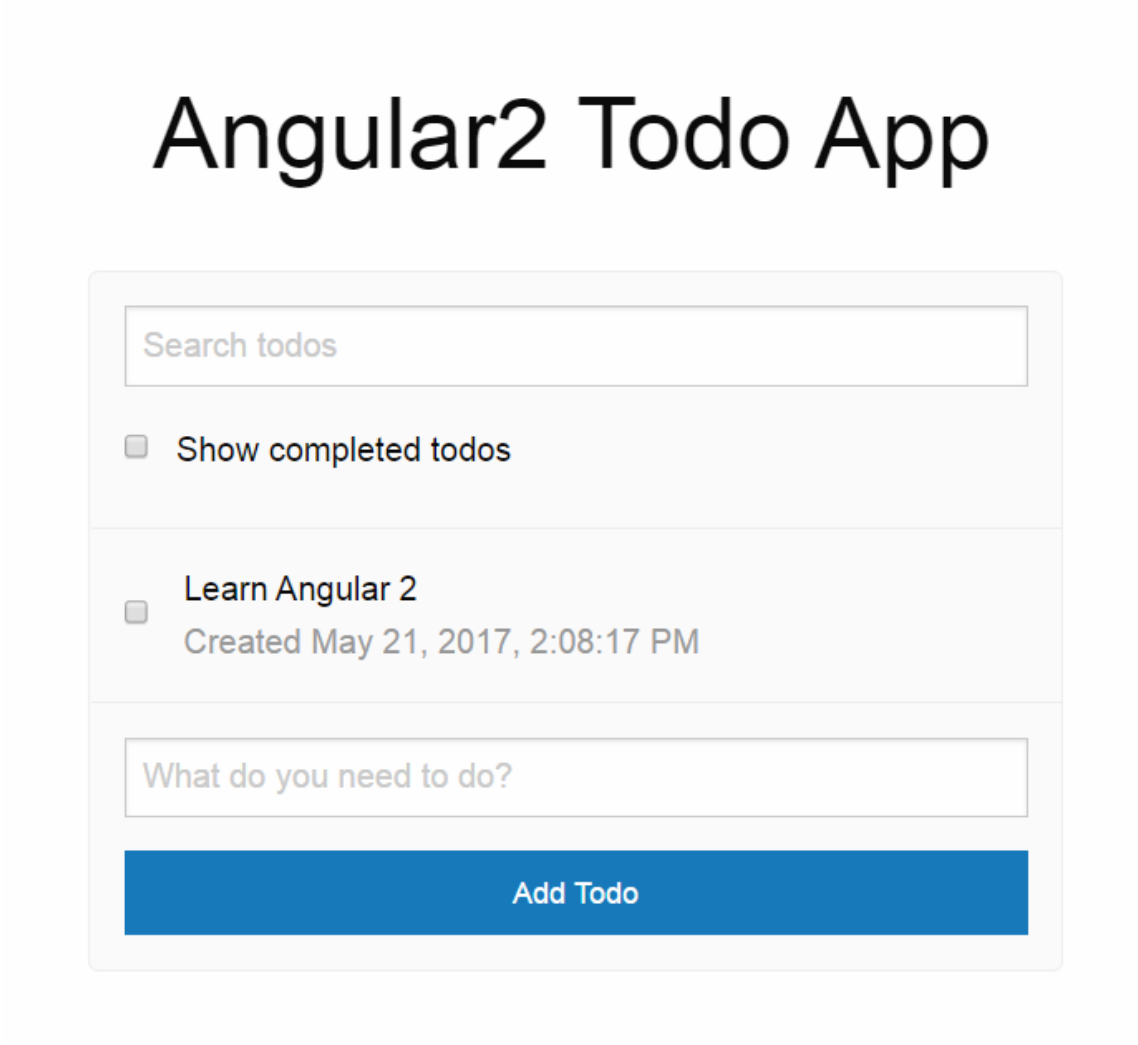
Komponentidele lisatakse mallid, mille implementeerimisel kasutatakse Angular 2 direktiive ning andmete sidumise funktsionaalsust. Lisatakse rakenduse juurkomponent, mis laeb igal laadimisel brauseri lokaalsest mälust *Todo* objektid, ning lisab need Redux'i olekulattu. Samuti kuulab kõiki olekulao muutusi ning lisab muudatuse korral uue *Todo* objekti lokaalsesse mälusse. *Todo* objekti väljad on rangelt määratud kasutades TypeScript'i funktsionaalsust. Juurkomponendi mallis määratakse ära komponentide paigutus.

Juurmoodulis defineeritakse ära, kuidas rakendus töötama hakkab – millised komponendid deklareeritakse, milliseid teeke kasutada, milliseid klasse rakenduses nähtavaks teha ning millist komponenti kasutada eellaadimisel.

Todo rakenduse implementatsiooni tegi kergemaks Angular CLI teek, mis muutis osade lisamise lihtsamaks. Samuti on rakendus loogiliselt ning „puhtalt" struktureeritud. Rakenduse jooksutamiseks kasutatakse järgnevat Angular CLI käsku:

```
ng serve
```

Minnes brauseri aadressile *localhost:4200* näeme implementeeritud Todo rakendust. Järgnevalt on toodud kuvatõmmis valmis Angular 2 Todo rakendusest (Joonis 29).



Joonis 29. Kuvatõmmis Angular 2 Todo rakendusest.

4.2 Rakenduse realisatsioon kasutades ReactJs raamistikku

Järgnevalt kirjeldatakse Todo rakenduse implementatsiooni ReactJs raamistikuga. Kogu töö käigus valminud React Todo rakenduse lähtekood on saadaval järgneval aadressil: https://github.com/habe33/react-todo. Oleku hoidmiseks kasutatakse Redux'i olekustruktuuri teeki ning arenduskeskkonnaks kasutati Atom tekstiredaktorit. Andmebaasiks kasutatakse brauseri lokaalset mälu.

React Todo rakendus on koostatud Andrew Mead'i veebikursuse „The Complete React Web App Developer Course" järgi [32]. Samuti on kasutatud Andrew Mead'i koostatud React rakenduse aluskoodi [33], millega on mugav alustada arendamist. Sarnaselt Angular 2 Todo rakendusele kasutatakse rakenduse jooksutamise abivahendina NodeJs käitussüsteemi.

Sõltuvused lisatakse „package.json“ konfiguratsiooni faili (Joonis 30).

```
 "dependencies": {
    "axios": "^0.9.1",
    "express": "^4.13.4",
    "react": "^0.14.7",
    "react-addons-test-utils": "^0.14.7",
    "react-dom": "^0.14.7",
    "react-router": "^2.0.0"
  },
  "devDependencies": {
    "babel-core": "^6.5.1",
    "babel-loader": "^6.2.2",
    "babel-preset-es2015": "^6.5.0",
    "babel-preset-react": "^6.5.0",
    "babel-preset-stage-0": "^6.5.0",
    "css-loader": "^0.23.1",
    "deep-freeze-strict": "^1.1.1",
    "expect": "^1.14.0",
    "foundation-sites": "^6.2.0",
    "jquery": "^2.2.1",
    "karma": "^0.13.22",
    "karma-chrome-launcher": "^0.2.2",
    "karma-mocha": "^0.2.2",
    "karma-mocha-reporter": "^2.0.0",
    "karma-sourcemap-loader": "^0.3.7",
    "karma-webpack": "^1.7.0",
    "mocha": "^2.4.5",
    "moment": "^2.12.0",
    "node-sass": "^3.4.2",
    "node-uuid": "^1.4.7",
    "react-redux": "^4.4.1",
    "redux-thunk": "^2.2.0",
    "redux": "^3.3.1",
    "sass-loader": "^3.1.2",
    "script-loader": "^0.6.1",
    "style-loader": "^0.13.0",
    "webpack": "^1.12.13"
  }
}
```

Joonis 30. React-todo „package.json“ sõltuvused.

Sõltuvuste installeerimiseks kasutatakse alljärgnevat käsku, mis tekitab *node_modules* kausta, kus asuvad kõik sõltuvuste failid.

```
npm install
```

Redux'i olekusüsteemi arhitektuuri osad (tegevused ja reduktorid) lisatakse eksporditavate funktsioonidena, mida saab kasutada kõikjal moodustades nendest muutujad. Järgnevalt on toodud näide tegevuse muutuja loomisest (Joonis 31).

```
var actions = require('actions');
```

Joonis 31. Redux tegevuse muutuja loomine

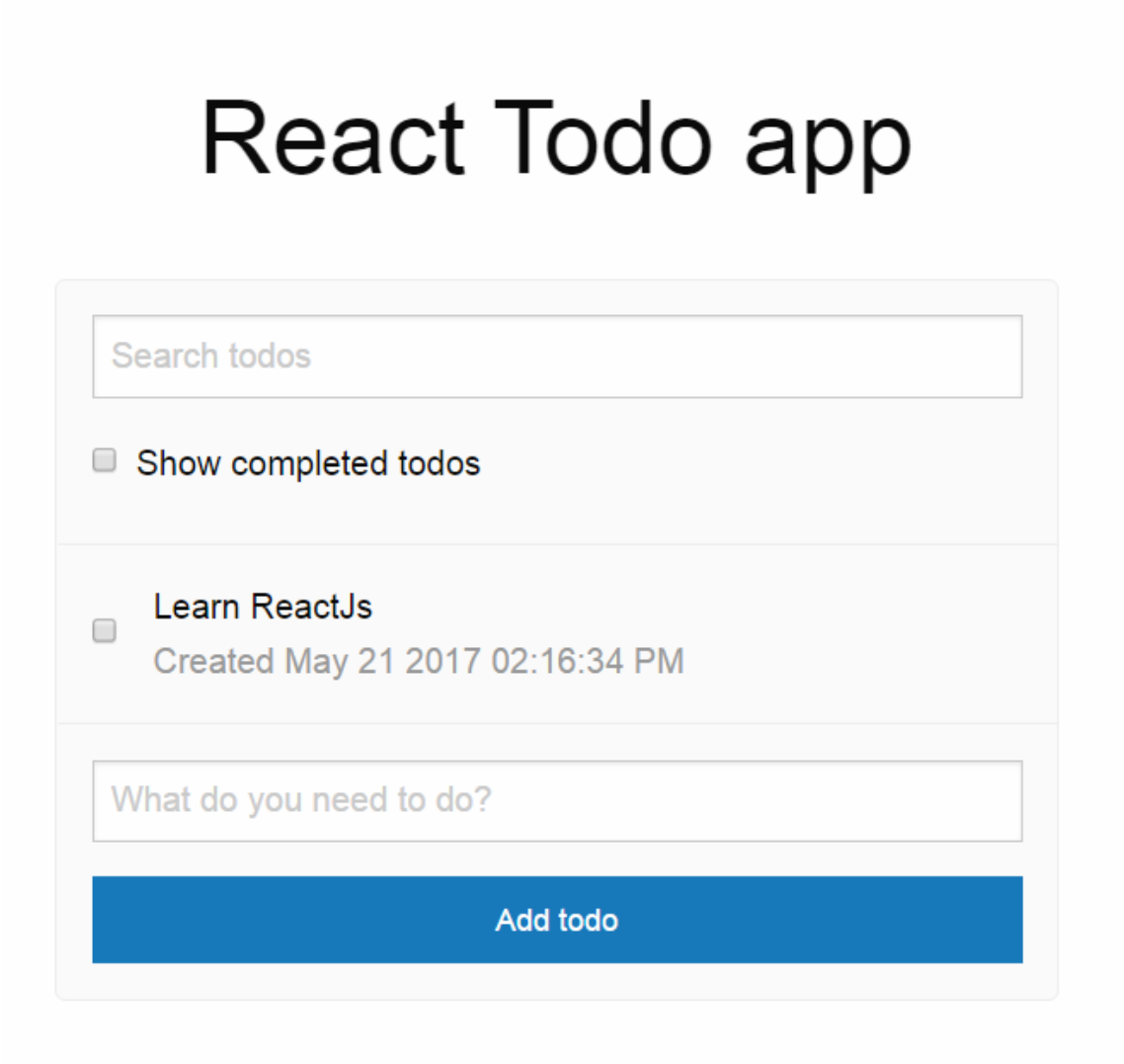
Olekulao moodustamiseks kasutatakse sarnaselt Angular 2 Todo rakendusele Redux'i abifunktsioone *combineReducers* ning *createStore*.

React'i komponendid koostatakse kasutades virtuaalse DOM-i funktsionaalsust. Komponendid on paigutatud nagu Angular 2 Todo rakenduses (Joonis 28). React komponentide mallid on ühes failis komponendi funktsionaalse loogikaga. Komponendi klass moodustatakse *React.createClass* meetodiga, mis võtab sisendiks *render* objekti, kus defineeritakse ära malli ning funktsioonide loogika. Rakenduse juurfailis moodustatakse juurkomponent kasutades *ReactDOM.render* funktsiooni. Juurkomponent määrab ära, kuidas rakenduse komponendid paigutatud on. Sarnaselt Angular 2 Todo rakendusele laeb rakenduse juurfail igal laadimisel brauseri lokaalsest mälust sündmused massiivi ning lisab Redux'i olekulattu. Samuti kuulab rakenduse juurfail kõiki Redux olekulao muutusi ning lisab muudatuse korral uue sündmuste massiivi lokaalsesse mällu.

Rakenduse jooksutamiseks kasutatakse alljärgnevat NodeJs käsku.

```
npm start
```

Peale käsu jooksutamist surutakse kogu rakenduse loogika *bundle* faili ning rakendus on kättesaadav minnes brauseriga aadressile *localhost:3000*. Järgnevalt on toodud kuvatõmmis valmis React Todo rakendusest (Joonis 32).



Joonis 32. Kuvatõmmis React Todo rakendusest.

5 Angular 2 ja ReactJs raamistike võrdlus

Angular 2 ja React on konkureerivad JavaScript'i raamistikud. Selleks, et arendajale oleks raamistiku valik lihtsam, tuleb neid erinevate näitajate alusel võrrelda. Raamistike võrdluseks kasutatud näitajad on valitud tuginedes autori kogemustele. Võrdluses lähtutakse autori tähelepanekutest koostades Todo rakendusi ning kogemustest tarkvara arenduse valdkonnas.

Kogukonna tugi

Kogukonna tugi on oluline, sest tänapäeval on muutunud tarkvara rakendused väga keerukaks ning aeg-ajalt tuleb abiks internetist leitud näited või õpetused. Mida rohkem on raamistiku kasutajaid, seda suurem tõenäosus on leida oma probleemile lahendus. StackOverflow on üks suurimaid kogukondi, kus arendajad otsivad küsimustele vastuseid. Otsides StactOverflow'st „reactjs" märksõna järgi, ilmneb, et selle märksõnaga on seotud 39958 küsimust [34]. Otsides „angular" märksõna järgi, ilmneb, et see märksõna on seotud 47530 küsimusega [35]. Samuti näeb eelmainitud joonise (Joonis 4) järgi, et Angular'i raamistike kohta otsitakse tunduvalt rohkem informatsiooni. See võib tähendada kahte – kas raamistik on keeruline ning tekitab palju küsimusi või siis on raamistik väga populaarne. Angular'i esimene versioon avaldati 3 aastat enne React'i esimest versiooni, mis võib põhjendada ka kogukonna suuruse erinevust. Rakendusi arendades ei märganud autor, et kummagi raamistiku kohta oleks vähem informatsiooni saadaval.

Projekti loomise lihtsus

Projekti loomise lihtsuse all mõeldakse rakenduse ülesseadmist ning konfigureerimist. Angular 2 raamistikule on arendatud Angular CLI, mis on mugav tööriist Angular 2 rakenduste loomiseks. Angular CLI järgib projekti loomisel hea tava reegleid ning koostab projekti kasutades lihtsat ning loogilist struktuuri. Samuti moodustatakse failid kasutades puhta koodi reeglistikku. Komponentide ning muude Angular 2 osade lisamine ning nende sidumine rakendusega on muudetud väga kergeks ning kiireks.

Mõlema rakenduse puhul kasutatakse lisa teeke rakenduse ehitamisel. React raamistik on vaade MVC arhitektuuris. See tähendab, et React'il ei ole määratud standardit, milliseid teeke kasutada. See muudab rakenduse konfigureerimise keerukaks ning aeganõudvaks.

Kasutatavate teekide head ning halvad küljed peab arendaja endale enne valikuid selgeks tegema. Hiljuti koostati ka React'ile *create-react-app* CLI, mis aitab arendajal moodustada React'i baasprojekti, kuid antud CLI dokumentatsioon ning arusaamine oli keeruline, mistõttu otsustas autor seda mahu suuruse tõttu mitte kasutada.

Todo rakenduste tegemisel oleks võtnud React'i projekti moodustamine kordades rohkem aega, kui Angular projekti moodustamine ning seetõttu otsustas autor kasutada Andrew Mead'i arendatud baasprojekti.

Rakenduse arhitektuur

Angular 2 kasutab standardset kahesuunalise andmevoo mudelit. Kui vaates kutsutakse esile sündmus, siis andmed liiguvad teenuse kihti ning sealt tagasi vaatesse. Selline andmete manipuleerimise viis on laialdaselt kritiseeritud, kuna andmete olek võib muutuda mõlemat pidi liikudes, mis võib lõppeda konfliktidega.

Angular 2 raamistikuga on võimalik kasutada ka Redux'i arhitektuuri, mis kasutab ühesuunalist andmete liikumist. Seda ka Todo rakenduse implementatsioonis tehti. Selline kooslus ei ole väga levinud ning võib viia suuremate rakenduste puhul ettenägematute vigadeni. React'i kasutatakse väga laialdaselt koos Redux'iga. Redux oligi algselt tehtud React'i jaoks. Sellist kooslust on palju testitud ning ilmselt on enamus veakohad juba teatavaks tehtud.

Todo rakenduste puhul Redux'i kasutamisega probleeme ei tekkinud. Rakenduse arhitektuur on mõlema raamistiku puhul selge ning arusaadav, sest rakenduse keerukus on väike.

Koodi struktuur

Angular 2 rakendusi kirjutatakse kasutades TypeScript'i, mis pakub tüüpe ning objekt-orienteeritud lähenemist. Selliste omaduste olemasolu on vajalik suuremate projektide haldamiseks. Samuti on võimalik Angular 2 rakendustes lihtsalt tõsta mallid ning funktsionaalne loogika erinevatesse failidesse, mis muudab koodi struktuuri arusaadavamaks ning failid lühemaks.

React rakendusi kirjutatakse kasutades JSX-i. Süntaksit ning struktuuri vaadates näeb JSX rohkem JavaScript'i moodi välja ning HTML kood „sobitub" sinna paremini.

Keerukate komponentide puhul võivad tekkida hiiglaslikud ning arusaamatud komponendid, mis koosnevad nii funktsionaalsest koodist kui ka HTML elementidest.

Autori arvates muudab TypeScript isegi väiksemad rakendused palju loogilisemaks ning arusaadavamaks. Samuti on tüüpidel põhinev TypeScript veakindlam.

Õppimise kiirus

Autori arvates on lihtsam mõelda nagu React kui mõelda nagu Angular 2. Uustulnukad, kes hakkavad õppima Angular 2 raamistikku, peavad rinda pistma uute direktiividega ning süntaksiga nagu näiteks *ng-** direktiivid, kuid React seejuures kasutab JavaScript'ile sarnast süntaksit. Samuti paneb Angular 2 raamistik JavaScript'i koodi HTML koodi, kuid React kasutab HTML koodi hoopis JavaScript'is. Pannes HTML koodi JavaScript'i on lihtsam mõelda ning õppimine läheb sujuvamalt, sest enamus, kes hakkavad neid raamistikke õppima, on varem kasutanud JavaScript'i.

Samuti õppides Angular 2 raamistikku, peab arendaja õppima paralleelselt ka TypeScript'i, sest TypeScript lisab uut süntaksit ning funktsionaalsust JavaScript'i.

Vigade leidmine

Mõlemale raamistikule on olemas brauseri lisavidinad, mis aitavad vigade leidmisele kaasa. Angular 2 raamistikule on loodud Angular Augury, mis on kõige enim kasutatud vidin. React raamistikule on loodud React Developer Tools vidin, mis hõlpsustab vigade leidmist. Mõlemad vidinad sobituvad Google Chrome veebibrauseriga.

Autori arvates muudab lisaks vidinatele ka TypeScript'i kasutamine vigade leidmise palju lihtsamaks tänu tüüpide defineerimisele.

Rakenduse kiirus

Angular 2 manipuleerib originaal DOM-i otseselt, mis muudab selle muutmise aeglaseks. React kasutab virtuaalse DOM-i metoodikat, mis muudab DOM-i muutmise palju kiiremaks.

Angular 2 kasutab kahesuunalist andmete liigutamist, mis tähendab, et kui muuta väärtust DOM-is, siis muutub nii vaade kui ka mudel. See on võimalik tänu mitmete lisafunktsionaalsustele. Andmete liigutamine nõuab „vaatlemist" ning mida suurem on

rakendus, seda suurem vastutus on „vaatlejatel“ ning seda aeglasemaks rakendus muutub. React’is tuleb kirjutada ise meetod, mis juhib andmete liigutamist ning tänu sellele ning virtuaalsetele DOM-idele on andmete uuendamine kiirem.

Ühiktestimine

Kuna Todo rakendus on konfigureeritud kasutama *webpack* konfiguratsiooni, siis on lihtne seadistada mõlemat raamistikku kasutama Karma testimisraamistikku. Karma raamistik on sisuliselt vahend, mis loob veebiserveri ning jooksutab lähtekoodi testkoodi vastu.

Testide struktuur on loogiline ning testide kirjutamine lihtne. Kvaliteetse tarkvara tagamiseks on testimine väga oluline. Kuna testide kirjutamine on mahukas töö, siis antud uurimuse raames seda ei kajastatud. Raamistike vahel ühiktestide osas olulisi erinevusi testimisel ei tekkinud.

6 Järeldused

Töö käigus selgus, et mõlemad raamistikud sobivad hästi väiksemate rakenduste jaoks nagu selleks on Todo rakendus. Sobivaima raamistiku leidmiseks võetakse arvesse eelnevas võrdluses selgunud tulemused. Võrreldavad näitajad koos tulemustega on toodud tabelis (Tabel 1).

Võrreldavad näitajad	**Angular 2**	**ReactJs**
Kogukonna tugi	Kõrgem	Madalam
Projekti loomise lihtsus	Lihtsam	Keerulisem
Rakenduse arhitektuur	Vähem sobilikum Redux'iga	Sobilikum Redux'iga
Koodi struktuur	Lihtsam	Keerulisem
Õppimise kiirus	Aeglasem	Kiirem
Vigade leidmine	Lihtsam	Keerulisem
Rakenduse kiirus	Aeglasem	Kiirem
Ühiktestimine	Lihtne	Lihtne

Tabel 1. Võrreldavad näitajad koos tulemustega.

Mõlemal raamistikul on tekkinud suur kogukond, kuid Angular 2 kogukond on pisut suurem. Selle põhjuseks võib olla see, et Angular'i esimene versioon väljastati 3 aastat varem kui React'i esimene versioon. Mõlema raamistiku kohta leidub palju usaldusväärset materjali internetis. Samuti on mõlema raamistiku kohta kirjutatud erinevaid raamatuid.

Projekti loomine on Angular 2 raamistikuga korral väga lihtne. Seda tänu Angular CLI süsteemile. React'il on ka olemas CLI süsteem, kuid see ei ole nii populaarne ning lihtne.

Todo rakenduste implementatsioonis kasutati Redux'i olekusüsteemi. Redux on algselt koostatud React'i jaoks, seega on antud teek sobilikum React'ile. Angular 2 rakendusi saab ka koostada koos Redux olekusüsteemiga, kuid sellist lähenemist suuremate rakenduste korral väga palju ei praktiseerita.

Koodi struktuuri muudab lihtsaks ning objekt-orienteerituks Angular 2 Typescript. Samuti on Angular 2 mallid ning komponendi loogika lihtsasti tõstetav erinevatesse failidesse.

React'i õppimine on lihtsam kui Angular 2 õppimine, kui õppijal on kogemusi JavaScript'iga. Üldjuhul õpitakse Angular 2 raamistikku koos TypeScript'iga, mis muudab õppimise keerukamaks.

Vigade leidmine on mõlema raamistiku puhul tehtud lihtsaks tänu brauserite lisavidinatele, mis aitavad vigu leida jälitades rakenduse käiku. Angular 2 raamistik kirjutatakse TypeScript'is, mis hoiab ära tüübivead.

React on kiirem tänu virtuaalsele DOM-ile. Samuti võivad suuremad Angular 2 rakendused muutuda aeglasemaks kahepoolse andmete sidumise tõttu.

Ühiktestimises olulisi raamistike vahelisi erisusi ei tekkinud. Antud töös kasutati Karma testimisraamistikku, mis ühildus mõlema raamistikuga.

Võrdluse käigus selgub, et mõlemal raamistikul on oma koht vastavalt projekti suurusele, loogikale ning keerukusele. Kui rakenduseks on lihtne rakendus, siis sobib selleks pigem React + Redux raamistike arhitektuur. Kui rakendus on keerukam ning hallatav mitme inimese poolt, siis pigem sobib raamistikuks Angular 2. Samuti tuleks arvesse võtta arendaja maitset. Kui arendajale meeldivad pigem struktureeritud ning objekt-orienteeritud lähenemisega rakendused, siis tuleks vaadata pigem Angular 2 raamistiku poole. Kui arendaja on JavaScript'i taustaga, siis võiks pigem vaadata React raamistiku poole.

Autori arvates sobis Todo rakenduse implementatsiooniks rohkem Angular 2 raamistik, sest projekt on struktureeritud loogilisemalt, kood on veakindlam ning loomine oli palju lihtsam kui React raamistikuga.

7 Kokkuvõte

Antud töö eesmärgiks oli hinnata Angular 2 ning ReactJs raamistike kasutatavust ning juurutamisest tulenevaid eeliseid ja puudusi, mille põhjal oleks arendajal lihtsam teha valik nende raamistike vahel.

Töö käigus implementeeriti Angular 2 ning ReactJs raamistikega Todo veebirakendus, mille käigus uuriti erinevaid aspekte, mis mõjutavad arendamise kiirust, loogilisust ning lihtsust.

Tulemustest selgus, et mõlemal raamistikul on oma koht vastavalt projekti suurusele, loogikale, vajadustele ning keerukusele. Tulemuste ning autori arvamuse põhjal võib väita, et väiksemate üheleheveebirakenduste korral on mõistlikum kasutada Angular 2 raamistikku, sest selle ülesseadmine on palju kiirem, projekt on paremini struktureeritud ning kood on veakindlam.

Töö eesmärgid saavutati toetudes välistele infoallikatele, rakenduse implementeerimisele ning autori tähelepanekutele.

Kasutatud kirjandus

[1] „About JavaScript,“ [WWW] https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript. (20.02.2017)

[2] H. M. N. C. F. O. Booth. [WWW] https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#relwwwrest. (20.02.2017)

[3] „History,“ [WWW] https://jquery.org/history/. (20.02.2017)

[4] A. Osmani, Learning JavaScript Design Patterns, O'Reilly Media, Inc., 2012.

[5] D. Shapiro, „Understanding Component-Based Architecture,“ [WWW] https://medium.com/@dan.shapiro1210/understanding-component-based-architecture-3ff48ec0c238. (10.04.2017)

[6] „About Stars,“ [WWW] https://help.github.com/articles/about-stars/. (10.04.2017)

[7] „Fork A Repo,“ [WWW] https://help.github.com/articles/fork-a-repo/. (10.04.2017)

[8] „Front-end JavaScript frameworks,“ [WWW] https://github.com/showcases/front-end-javascript-frameworks?s=stars. (10.04.2017)

[9] „Google Trends,“ [WWW] https://trends.google.com/trends/explore?date=2016-01-01%202016-12-31&q=angular,reactjs,emberjs,vuejs,backbonejs. (26.04.2017)

[10] „Stack Overflow,“ [WWW] https://stackoverflow.com/. (26.04.2017)

[11] „Developer Survey Results 2017,“ [WWW] https://stackoverflow.com/insights/survey/2017. (26.04.2017)

[12] „GitHub Angular,“ [WWW] https://github.com/angular/angular.js/blob/master/CHANGELOG.md. (26.04.2017)

[13] „GitHub TypeScript,“ [WWW] https://github.com/Microsoft/TypeScript. (18.04.2017)

[14] „Architecture overview,“ [WWW] https://angular.io/docs/ts/latest/guide/architecture.html. (18.04.2017)

[15] „NgModules,“ [WWW] https://angular.io/docs/ts/latest/guide/ngmodule.html. (16.04.2017)

[16] „Angular 2 - Architecture,“ [WWW] https://www.tutorialspoint.com/angular2/angular2_architecture.htm. (22.04.2017)

[17] „ReactJS Tutorial,“ [WWW]

https://www.tutorialspoint.com/reactjs/. (18.04.2017)

[18] „What is JSX?," [WWW]
https://jsx.github.io/. (18.04.2017)

[19] „React: The Virtual DOM," [WWW]
https://www.codecademy.com/articles/react-virtual-dom. (26.04.2017)

[20] P. Márton, „The React.js Way: Getting Started Tutorial," [WWW]
https://blog.risingstack.com/the-react-way-getting-started-tutorial/. (20.04.2017)

[21] „React.Component," [WWW]
https://facebook.github.io/react/docs/react-component.html. (20.04.2017)

[22] K. Wheeler, „Learning React.js: Getting Started and Concepts," [WWW]
https://scotch.io/tutorials/learning-react-getting-started-and-concepts.
(20.04.2017)

[23] R. Bez, „Introduction to React.js," [WWW]
http://devangelist.de/en/react-js/. (20.04.2017)

[24] P. Shan, „React Tutorial: Two way data binding," [WWW]
http://voidcanvas.com/react-tutorial-two-way-data-binding/. (20.04.2017)

[25] „GitHub Redux," [WWW]
https://github.com/reactjs/redux. (21.04.2017)

[26] „Data Flow," [WWW]
http://redux.js.org/docs/basics/DataFlow.html. (22.04.2017)

[27] „Actions," [WWW]
http://redux.js.org/docs/basics/Actions.html. (21.04.2017)

[28] „Reducers," [WWW]
http://redux.js.org/docs/basics/Reducers.html. (21.04.2017)

[29] „Store," [WWW]
http://redux.js.org/docs/basics/Store.html. (21.04.2017)

[30] „About Node.js," [WWW]
https://nodejs.org/en/about/. (22.04.2017)

[31] „GitHub Angular-CLI," [WWW]
https://github.com/angular/angular-cli. (22.04.2017)

[32] „The Complete React Web App Developer Course," [WWW]
https://www.udemy.com/the-complete-react-web-app-developer-course/.
(22.04.2017)

[33] „GitHub React Course Boilerplate App," [WWW]
https://github.com/andrewjmead/react-course-boilerplate-app. (26.04.2017)

[34] „Stack Overflow," [WWW]
https://stackoverflow.com/questions/tagged/reactjs. (25.04.2017)

[35] „Stack Overflow," [WWW]
https://stackoverflow.com/questions/tagged/angular. (22.04.2017)

[36] „Angular Augury," [WWW]
https://augury.angular.io/. (10.05.2017)