Antti Reponen

Full-Stack: MERN-pino

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikan koulutusohjelma

Insinöörityö

7.11.2018

Tekijä Otsikko Sivumäärä Aika	Antti Reponen Full-Stack: MERN-pino 28 sivua 7.11.2018
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tietotekniikka
Ammatillinen pääaine	Software Engineering
Ohjaajat	Osaamisaluepäällikkö Janne Salonen

Vuonna 2009 Ryan Dahl kehitti uudenlaisen kehitysympäristön nimeltä Node.js. Kyseisen alustan avulla pystytään ajamaan JavaScript-koodi palvelinpuolella ensimmäistä kertaa. Tämä nopeuttaa, yksinkertaistaa ja yhtenäistää web-sovelluksien kehitystä, kun ainoana kehityskielenä toimii JavaScript kaikissa sovelluksen osa-alueissa.

Insinöörityössä keskitytään yhteen Node.js-alustan ympärille muodostuneista Full-Stack JavaScript -pinoista nimeltään MERN. Työssä esitellään pinon eri tekniikat ja kerrotaan, miten ne kommunikoivat keskenään.

Tavoitteena on luoda pinon tekniikoiden pohjalta prototyyppisovellus ja käydä läpi kokonaisen verkkopalvelun toteutuksen eri vaiheet. Työssä esitellään sovelluksen luomiseen tarvittavat työkalut ja havainnollistetaan käyttöliittymä- ja palvelinpuolen toiminnot sekä rakenne.

Web-sovellusten eri osa-alueiden kehitys yhtenäistyy, kun sen toteutuksessa käytetään pelkästään JavaScript-kieltä. React.js-pohjainen käyttöliittymä tuo kontrollia ja skaalautuvuutta. API-palvelin yhdistettynä objektimallisen NoSQL-tietokannan kanssa tarjoaa vaihtehtoisen tavan tiedon säilytykselle. Näiden yhdistelmällä pystytään luomaan dynaamisia ja suorituskyvyltään laadukkaita web-sovelluksia.

Avainsanat	MERN stack, MongoDB, Express.js, React.js, Node.js, REST, API, web-sovellus

Metropolia

Author Title Number of Pages Date	Antti Reponen Full-Stack: MERN-stack 28 pages 7 November 2018
Degree	Bachelor of Engineering
Degree Programme	Information and Communication Technology
Professional Major	Software Engineering
Instructors	Head of the Department Janne Salonen

In the year 2009 Ryan Dahl developed a new kind of development environment named Node.js, which allowed JavaScript code to be run server side for the first time ever. This speeds up, simplifies and unifies the web-applications development process as JavaScript language is used in all parts of the software.

The thesis focuses on one Full-Stack JavaScript -stack called MERN that has formed around the Node.js platform. The different parts of the stack will be introduced and explained how they communicate with each other creating dynamic web-applications.

The goal is to create a prototype application with MERN and to review the different phases of the development of an entire online service. The thesis presents the tools necessary for creating this application as well as illustrates the user to server-side functions and structure.

The development process will be more unified, when all parts of a web-application are written in JavaScript. A user interface created with React.js gives control and scalability. The API-server combined with an object-based NoSQL-database provides an alternative to store data. With this combination it is possible to create dynamic web-applications with good performance.

Keywords	MERN stack, MongoDB, Express.js, React.js, Node.js, REST, API, web-application

Sisällys

Lyhenteet

API	Application Programming Interface -ohjelmointirajapinta, jonka avulla voidaan vaihtaa tietoja sovelluksessa.
REST	Representational State Transfer – HTTP-protokollaan perustuva arkkitehtuurimalli ohjelmointirajapintojen luomiseen.
JSON	JavaScript Object Notation – Tiedostomuoto, jossa käytetään attribuuttiarvomäärittelyä datan esittämiseen.
npm	Node Package Manager – Node.js-ympäristön paketinhallintajärjestelmä.
OAuth	Todentamisen tekniikka, joka mahdollistaa käyttäjän tunnistautumisen kolmannen osapuolen sovelluksen tai palvelun kanssa.
DOM	Document Object Model - Dokumenttioliomalli, jonka tarkoituksena on määritellä dokumentissa olevat elementit ja viittaukset niihin.
Git	Yksi suosituimmista versionhallintaohjelmistoista.
CRUD	Create, read, update & delete - Akronyymi neljälle perusfunktiolle, joita käytetään mm. tietokantojen manipulointiin.
SPA	Single Page Application - Sovellus, jonka eri komponentit näytetään samalla sivulla.
MVC	Model-View-Controller - Kolmesta osasta koostuva ohjelmistoarkkitehtuurityyli, jota käytetään käyttöliittymien suunnittelussa ja ohjelmoinnissa.

1. Johdanto

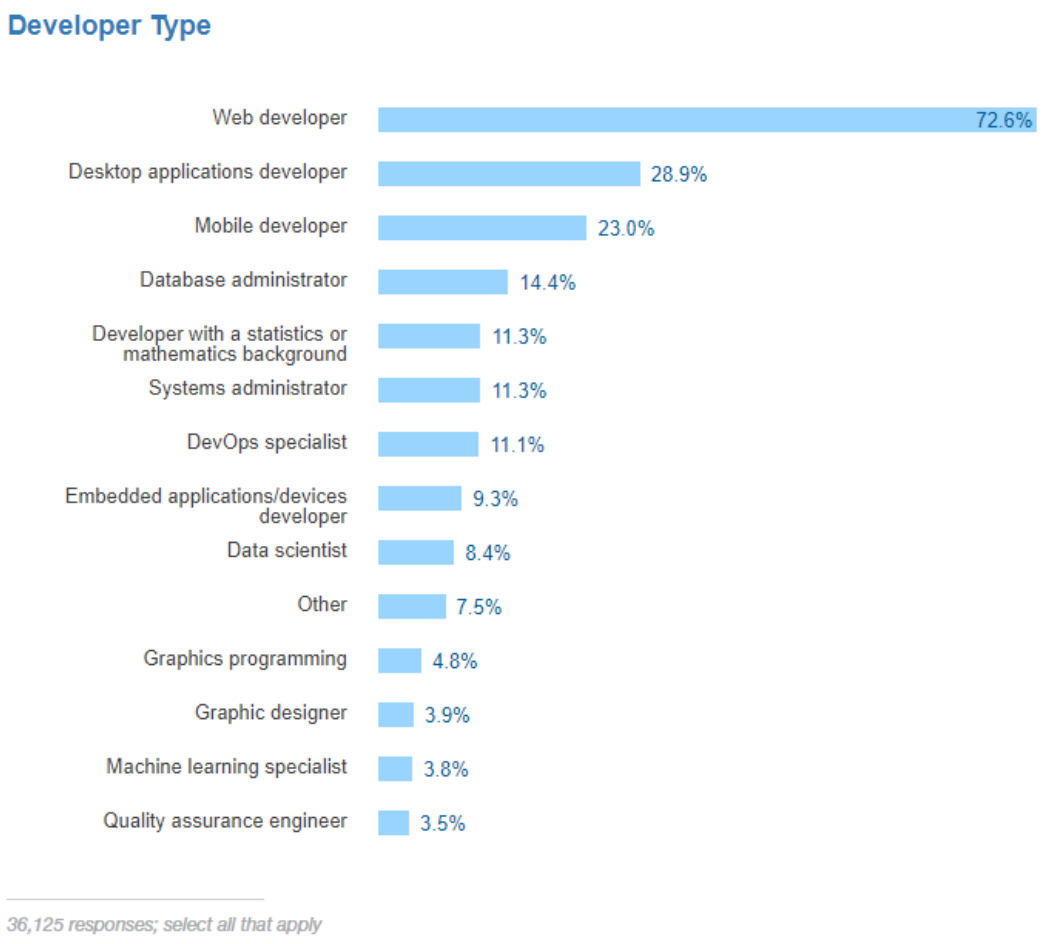
Yhdeksi suosituimmista web-sovelluskehitysalustaksi on noussut Ryan Dahlin vuonna 2009 kehittelemä Node.js. Sen tarjoama suorituskyky ja skaalautuvuus on huomattu myös tunnetuissa yrityksissä, jotka ovat sen käyttöönsä valinneet. Esimerkkeinä näistä mainittakoon Netflix, Uber ja PayPal. Nykyään on muodostunut käsite Full Stack -ohjelmistokehittäjä, joka on kuin ohjelmoinnin yleismies. Käsitteestä voi olla montaa mieltä, mutta Laurence Gellert niminen ohjelmistokehittäjä kuvaili asiaa blogissaan näin:

> “Minulle Full Stack -ohjelmistokehittäjä on joku, jolla on tuntemusta kaikista kehityksen osa-alueista, ellei jopa vahvaa osaamista osasta niistä, ja jolla on aito kiinnostus ohjelmistokehitystä kohtaan.” [1.]

Toisin sanoen hänen pitäisi pystyä hahmottamaan, mitä kaikissa pinon eri osa-alueissa tapahtuu, kun sovellus suorittaa toiminnon. Tässä työssä tarkoituksena on opetella yhden Node.js-alustan ympärille muodostuneen Full Stack -pinon eri osa-alueet ja luoda sen pohjalta prototyyppisovellus. Sovellusprojektin tavoitteena on käydä läpi kokonaisen verkkopalvelun toteutuksen eri vaiheet ja käyttää monipuolisesti MERN-pinoa sekä sen mahdollisia ulkoisia lisäosia.

2. Web-sovelluskehitys ja sovelluspinot

Suurin osa meistä käyttää internetiä ja sen avulla toimivia sovelluksia sekä palveluita. Harva meistä kuitenkaan pohtii, miten web-sovelluksia käytännössä toteutetaan ja mitä kaikkea siihen vaaditaan. Web-sovelluskehitys käsittää kaikki toteutukset, joihin pääsemme esimerkiksi selaimella käsiksi. Kehityksen vaiheisiin liittyy monipuolisesti visuaaliset, funktionaaliset, käytännölliset ja luovat ratkaisut. Kuvasta 1 voimmekin nähdä Web-kehittäjän olleen Stack Overflow -kehittäjäyhteisön suosituin tehtävänimike heidän vuonna 2017 teettämässä kyselyssä. [2.]



Kuva 1. Web-sovelluskehittäjät kattoivat noin kolmanneksen vastanneista. Monet heistä kertoivat myös työskentelevänsä työpöytä- tai mobiilisovellusten parissa. [2.]

Verkkopalvelujen eri toteutuksia on laaja valikoima ja oikean sovelluspinon valintaan vaikuttaa moni asia. Omassa tiimissä täytyy miettiä osaamisalueet ja vahvuudet, jonka pohjalta suurin osa alustoista ja tekniikoista määräytyvät. Lisäksi sovelluksen suorituskyvyn tarve sekä mahdollinen skaalautuvuus tulevaisuudessa ovat avaintekijöitä oikean pinon valinnassa. Tunnetuimmat sovelluspinot ovat esiteltynä alla:

- LAMP (Linux, Apache, MySQL, PHP) on yksi vanhimmista pinoista, joka toimii loistavasti dynaamisten verkkosovellusten luomiseen. Se on perinteisin malli ja erittäin vakaa. Kyseisestä pinosta löytyy myös eri variaatioita, jotka toimivat eri käyttöjärjestelmien ja kielien kanssa.
- MEAN (MongoDB, Express.js, AngularJS, Node.js), kokonaan JavaScriptiin pohjautuva moderni pino, joka haastaa LAMP-pinon. Sen JSON-tiedostomuotoa käyttävä NoSQL-tietokanta ja Node.js-alustan päälle rakennettu pino tarjoaa paljon joustavuutta luoda SPA- (Single Page Application) tai MPA- (Multi Page Application) sovelluksia.
- Ruby Stack (Ruby/Ruby on Rails, RVM (Ruby Virtual Machine), MySQL, Apache, PHP) -pino tulee valmiin kehitysympäristön kanssa. Suosittujen ominaisuuksien takia sovellusten luominen on helppoa ja nopeaa. Ruby on hyvin yhteensopiva MySQL:n kanssa.
- Django Stack (Python, Django, Apache, MySQL) on Pythoniin perustuva oliopohjainen ohjelmointiympäristö, jossa kuluu vähemmän aikaa palvelinpuolen konfiguroinnissa ja voi keskittyä enemmän palvelun logiikkaan. [3.]

3. MERN-pino

Keskitymme tässä työssä vain MERN-nimeä kantavaan pinoon, joka hieman poikkeaa edellä mainitusta MEAN-pinosta. Kummatkin pinot ovat kokonaan JavaScript-pohjaisia. Sovellusprojektissa käytetyn MERN-pinon eri tekniikat sekä työn kannalta keskeisimmät kolmannen osapuolen lisäosat esitellään.

3.1. React.js

MERN-pinon käyttöliittymän toteutukseen käytetään AngularJS-kehyksen sijasta React.js-kirjastoa, joka ei pohjaudu MVC-arkkitehtuurimalliin. Alun perin Facebook käytti MVC-mallia käyttöliittymän funktionaalisuuden toteutukseen, joka koostuu kolmesta eri osasta: [4]

1. Malli (model) kuvaa tiedon tallentamisen ja käsittelymallin.
2. Näkymä (view) määrittää käyttöliittymän tiedon näyttämisen sekä ulkoasun.
3. Käsittelijä (controller) vastaanottaa käyttäjän toiminnot ja muuttaa mallia tai näkymää. [4.]

MVC-mallissa näkymä kuuntelee mallin muutoksia, ja ne vastaavat muutoksiin päivittämällä itsensä. Ongelmaksi voi muodostua erittäin kompleksissa ohjelmassa tilanne, jossa muutos aiheuttaa päivityksen tarpeen, joka aiheuttaa toisen päivityksen (jokin muuttui päivityksestä johtuen) ja edelleen johtaa uuteen päivitystilanteeseen. Tällaista vaiheittaisia tapahtumasarjaa on vaikea ylläpitää, koska näkymän päivittäminen vaatii vain hienovaraisen eron koodissa riippuen päivityksen aiheuttajasta. React.js-kirjastoa käytetään vain näkymän renderöimiseen (V-kirjain MVC-mallista), eikä siinä noudateta MVC-mallia. Ohjelman muiden osuuksien toteutus on täysin kehittäjän oman näkemyksen tai käyttötarkoituksen mukainen. React.js on komponenttipohjainen, jonka näkymät ovat itsestään toteutuvia. Toisin sanoen sovelluskehittäjän ei tarvitse huolehtia muutoksista näkymän tilassa eikä datassa. Perinteisesti datan muuttuessa manipuloidaan DOM:ia esimerkiksi jQuery-kirjastoa käyttäen, mutta React.js:llä toteutettu käyttöliittymä osaa itse muokata nykyistä näkymää muutoksen jälkeen. Tämä tekee näkymistä yhtenäisiä, hyvin ennakoitavia ja helppoja ymmärtää. React.js osaa verrata muutoksia

omassa Virtual DOM:ssa hyvin tehokkaasti ja pystyy päivittämään vain tarvittavat muutokset. [5.]

```
class LoginButton extends React.Component {
  constructor(props) {
  super(props);
  this.state = { login: true };

     loginFunction(){
      this.setState({ login: false });
}
     logoutFunction(){
      this.setState({ login: true });
}

  renderContent() {
    if(this.state.login) {
        return (
         <button
          type="button"
          onClick={this.loginFunction}
           >Login</button>
        );
     }else {
        return (
          <button
          type="button"
          onClick={this.logoutFunction}
           >Logout</button>
        );
    }
  }

  render() {
    return (
      <div>
           {this.renderContent()}
      </div>
    );
  }
}
```

Esimerkkikoodi 1. Koodissa tila määrittelee Login- ja Logout-linkin näyttämisen. Tilan muutos aiheuttaa komponentin päivityksen eli render()-funktio ajetaan uudelleen ja komponentti päivittyy.

3.2. Node.js

Ryan Dahlin kehittämä Node.js-ympäristö on yksi maailman suosituimpia. Monet suuret yritykset ovat tunnistaneet sen hyvät ominaisuudet ja käyttäneet tekniikkaa heidän omissa sovelluksissaan. Node.js nimittäin pohjautuu Chrome V8 -moottoriin ja sen tarkoitus on pystyä ajamaan JavaScript-koodia itsenäisesti. Yleisemmin JavaScript-koodi ajetaan suoraan selaimessa ja rajoittuu sen takia käyttöliittymäpuolelle. Node.js-ympäristö mahdollistaa JavaScript-kielellä kirjoitetun koodin ajon myös palvelinpuolella. Kyseinen ominaisuus yhtenäistää käyttöliittymä- ja palvelinpuolen kielen. Tämä helpottaa monia sovelluskehittäjiä, koska sovelluksen eri osa-alueita voidaan kirjoittaa samalla kielellä. [6.]

Suorituskyky ja skaalautuvuus ovat Node.js-ympäristön valttikortteja, mutta nämä yhdistettynä sen omaan npm-paketinhallintajärjestelmään, tekevät siitä hyvin suositun ja toimivan alustan. Npm on laaja JavaScript-työkaluja sisältävä kirjasto, jonka kautta on hyvin helppo asentaa ja käyttää tarvittavia ominaisuuksia eri sovellusprojekteissa. Osassa näistä työkaluista saattaa kuitenkin puuttua standardisoitua koodia, joita kaupallisissa projekteissa ei välttämättä kannata hyödyntää tietoturvasyistä. [7.]

3.3. Express.js

Express.js on sovelluskehys Node.js-alustalle. Sen avulla palvelinpuolen toimintoja voidaan kirjoittaa vähemmällä määrällä koodia. Tämä nopeuttaa sovelluksen kehitystä, helpottaa koodin lukemista ja sen ymmärtämistä. Express.js asennetaan npm-paketinhallintajärjestelmän kautta, jota käytetään sovelluksessa. Kyseisen työkalun koko toiminnallisuus voitaisiin kirjoittaa myös tavallista JavaScriptiä käyttäen. Lopputuloksena olisi kuitenkin suurempi määrä koodia.

```
const express = require('express')
const app = express()
const port = 3000

app.get('/', (req, res) => res.send('Hello World!'))
app.listen(port, () => console.log(`Example app listening on port ${port}!`))
```

Esimerkkikoodi 2. Express.js määritellään osaksi sovelluksen toiminnallisuutta ja käytetään funktioiden tavoin. [8]

Esimerkkikoodin 2 ohjelma käyttää Express.js-työkalua, joka on käytettävissä app-nimisestä muuttujasta. Express.js osaa tavallisten CRUD-operaatioiden kautta palauttaa informaatiota tai suorittaa toiminnallisuuksia. Esimerkissä get()-funktiolle annetaan parametreinä polku ja takaisinkutsufunktio, joka taas saa parametreikseen kyselyn datan req sekä vastaamiseen tarkoitetun muuttujan res. Get-pyyntö kuuluu HTTP-protokollaan ja vastaa CRUD-operaatioiden Read-osaa. Viimeisellä rivillä ohjelma määritellään käyttämään porttia 3000, joten get-pyyntö palvelimen juureen palauttaa “Hello World!” -tekstin vastauksena. Paikallisesti toimivalla palvelimella kyseinen kysely lähetettäisiin localhost:3000 osoitteeseen. [8.]

3.4. MongoDB

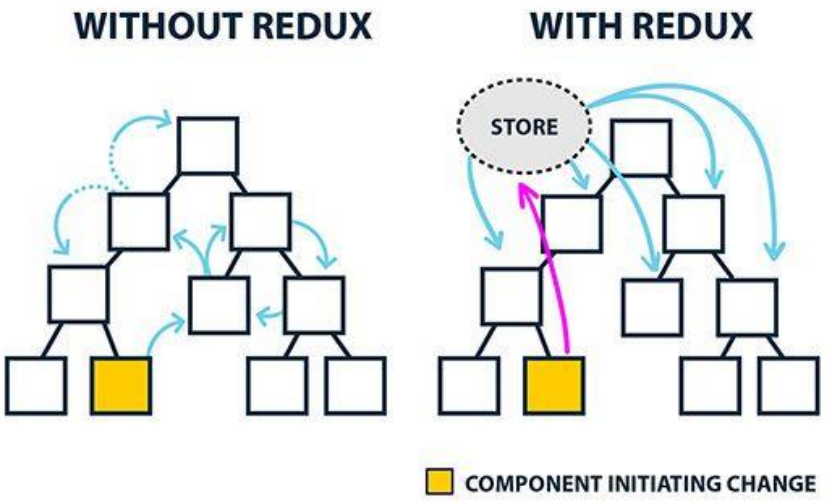
MongoDB on avoimen lähdekoodin NoSQL (Not only SQL) -tietokantaohjelma. NoSQL eroaa perinteisemmästä relaatiomallista siten, että kaikki tieto ja sen käsittely perustuu objekteihin. Relaatiotietokannoissa tieto koostuu useista taulukoista, joiden välille luodaan relaatio eli yhteys. Taulukoiden sisältämä tietorakenne pyritään pitämään samana, jotta taulukoiden väliset relaatiot säilyvät. Isot muutokset saattavat vaatia koko tietokannan luomisen uudelleen ja sen takia onkin tärkeää, että tietokanta suunnitellaan huolellisesti. NoSQL-tietokannassa voidaan tietorakenteita muuttaa jälkikäteen, sillä tietoa käsitellään objekteina. Tallennettava tieto ei myöskään rajoitu pelkästään järjestelmälliseen tietoon vaan sinne voidaan tallentaa myös kuvia, videoita tai muita medioita. Objektimallisen luoteen takia se on myös helposti skaalautuva ja hyvin yhteen sopiva eri olio-ohjelmointikielien kanssa. Suurissa NoSQL-tietokannoissa datan tallennus vaatii palvelimelta enemmän resursseja kuin perinteinen relaatiotietokanta. [9.]

4. Ulkoiset lisäosat

Sovelluksia suunniteltaessa ja kirjoittaessa törmäämme toisinaan haastaviin tilanteisiin. Näiden ratkaisuun tai helpottamiseen löytyy yleensä valmiita toiminnallisuuksia, joita voi sovelluksessa hyödyntää. Node.js-ympäristön npm-paketinhallintajärjestelmä on laaja kirjasto, joka sisältää juurikin tämänkaltaisia hyödykkeitä. Npm-kirjastosta löytyy laaja valikoima erilaisia työkaluja, mutta tässä työssä esitellään vain kaksi niistä. Nämä kaksi lisäosaa ovat isossa roolissa sovelluksen loogista tiedonkulkua ja helpottavat näiden toteutusta.

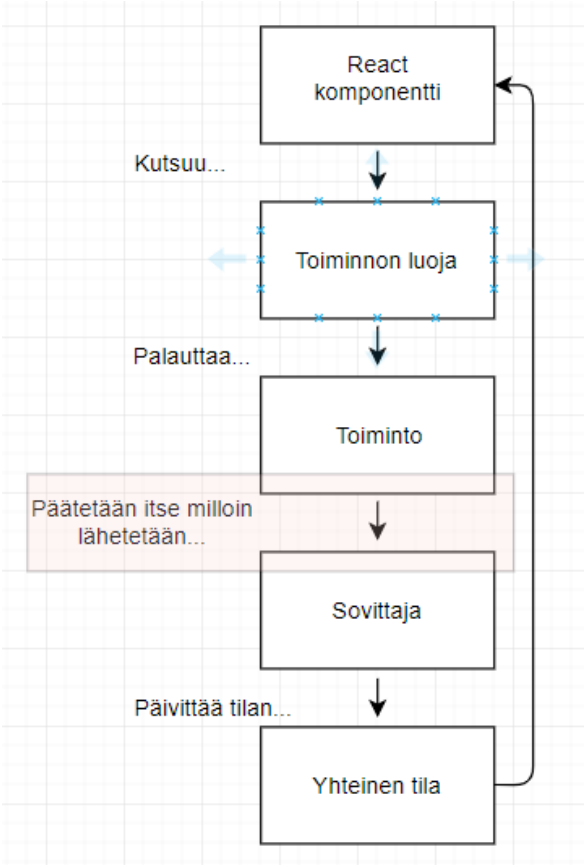
4.1. Redux ja Redux-Thunk

Redux on JavaScript-kirjasto, jota käytetään sovelluksessa tilan tallennusta varten. Tila on objekti, joka sisältää sovelluksessa tai sen komponenteissa tarvittavaa tietoa. Sitä käytetään mm. React.js- ja AngularJS-kirjastojen kanssa tuomaan helpompaa ja selkeämpää tiedonkäsittelyä sovellusten komponenttien välille. React.js on tilan tiedonkulun osalta normaalisti hierarkkinen. Tämä tarkoittaa sitä, että komponentit voivat siirtää tilan informaation kaikille alikomponenteille. Informaatio siirtyy helposti niin sanotussa puumaisessa rakenteessa ylhäältä alas lapsikomponenteille ja niistä taas heidän isäntäkomponenteille. Tällaisessa rakenteessa kommunikaatiota lapsikomponenttien välille on työlästä rakentaa ja kompleksissa sovelluksessa koodista saattaa tulee vaikeasti luettavaa. Redux-kirjastolla luodaan yksi säilytyspaikka tilojen tallennusta varten ja ne komponentit, joiden tarvitsee keskustella muiden komponenttien kanssa, voidaan liittää tähän yhteen tilojen säilytyspaikkaan, jota kutsutaan nimellä säilö. Näin ollen tarvittava lapsikomponenttien välinen tiedonkulku on selkeästi toteutettavissa keskitetyllä tallennuspaikalla. [10.]



Kuva 2. Tiedonkulkutilanne, kun komponentti aiheuttaa päivityksen tarpeen. [11]

Redux-Thunk on Reduxin omaa tiedonkulkua muokkaava lisäosa, jolla voimme päättää, milloin väliohjelmisto jatkaa toimintoa. Tällaista toimintaperiaatetta tarvitaan usein asynkronisissa tapahtuma sarjoissa, joissa halutaan varmistaa tietyn toiminnon jatkaminen vasta, kun haluttu data on vastaanotettu. Näin esimerkiksi vältytään lataamasta tyhjiä komponentteja käyttöliittymäpuolella ja taataan mahdollisimman hyvä käyttökokemus. Samalla periaatteella toimii myös palvelinpuolen toiminnot, jossa tiettyyn resurssiin lähetetty kutsu vastaa sitten, kun sen toiminto on valmis. [12.]



Kuva 3. Kuvassa havainnollistetaan Redux-kirjastolla toteutetun ohjelmiston normaali tiedonkulku ja minkä automaattisen toiminnon manualisointi Redux-Thunk:lla tehdään.

4.2. Passport.js

Passport.js on Node.js-ympäristön yksi valinnainen väliohjelmisto. Se vastaa sovelluksessa tapahtuvista todennuksista ja auttaa niiden toteuttamisessa. Kirjautumisten tai oikeuksien todentaminen on yleisin käyttökohde web-sovelluksissa, jonka toteutus selkiytyy huomattavasti Passport.js-väliohjelmiston avulla. Se tukee myös suosittujen palveluiden, kuten Google, Twitter ja Facebookin tunnistautumisstrategioita. Esimerkkinä mainittakoot OAuth, joka on menetelmä identiteetin varmistamista varten. Kyseistä menetelmää käyttäen pystytään turvallisesti kirjautumaan web-sovellukseen mm. käyttämällä Google-tiliä, eikä omia tunnuksia tällöin tarvita. Passport.js-väliohjelmistoa voidaan käyttää myös sessioiden ja evästeiden kautta tapahtuvaan tunnistautumiseen, jolloin uudelleen kirjautumista ei määritellyn ajan sisällä tarvitse tehdä. [13; 14.]

```
app.post('/login', passport.authenticate('local', {
```

```
successRedirect: '/',
failureRedirect: '/login'
}));
```

Esimerkkikoodi 3. Express.js:llä toteutettu API-polku määritellään käyttämään Passport.js-väliohjelmistoa.

5. Sovellusprojektin toteutus

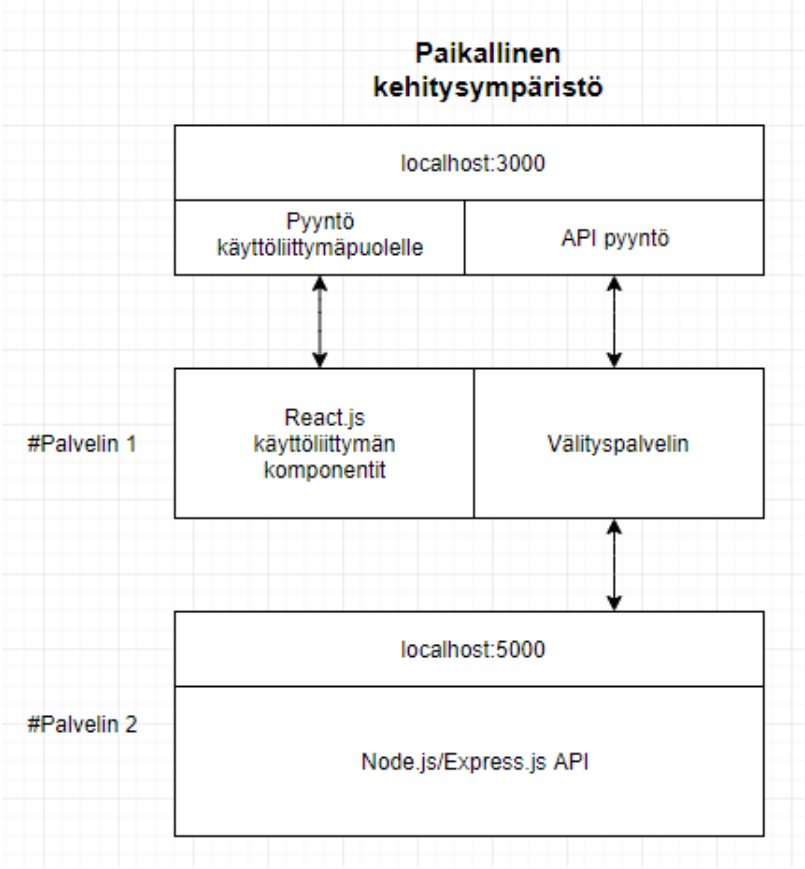
Projektina toteutetaan web-sovellus MERN-pinoa käyttäen. Tarkoituksena on havainnollistaa verkkosovellusten kehitysprosessia ja samalla luoda web-sovellus. Tärkeimmät kehitysvaiheet käydään läpi ja niihin otetaan kantaa. Sovelluksen kirjastojen, komponenttien sekä sovelluskehyksien tehtävät kuvataan lyhyesti. Kehityksen tuloksena saatu sovellus pystyy todentamaan kirjautumisen Google-tiliä käyttäen. Todennettu käyttäjä voi lisätä tehtäviä ja niille muistutuksia. Muistutukset lähetetään Google-kalenterin toimesta sähköpostiin.

5.1. Kehitysympäristö

Web-sovellusten kehitysympäristönä toimii yleensä paikallinen tietokone, jossa Linux-, Windows- tai macOS-käyttöjärjestelmä asennettuna. Kehitysympäristö voidaan ajaa myös virtuaalikoneissa, jotka taas voivat sijaita esimerkiksi pilvipalveluissa. Käyttöjärjestelmä valikoituu joko käyttötottumuksen tai kehitettävän sovelluksen mukaan. Tässä projektissa käytetään paikallista ympäristöä Windows 10 -käyttöjärjestelmällä. Koodin ja kansiorakenteiden hallintaan on valittu Visual Studio Code -ohjelmisto, joka mielestäni on erittäin selkeä ja tukee monia lisäosia. Kyseessä on ilmainen IDE-ympäristö (Integrated development environment). IDE-ympäristössä on kehitystä helpottavia ominaisuuksia kuten debuggaus, koodin väritys ja integroitu komentorivi, mikä päihittää normaalit tekstinkäsittelyohjelmat. Kehitysympäristön tarkoituksena on simuloida kehitettävän sovelluksen eri osa-alueet. Full-Stack -sovelluskehityksessä ajetaan paikallisesti palvelimen, ja käyttöliittymän toiminnot.

5.2. Suunnittelu ja rakenne

Valitsemamme sovellusprojekti toteutetaan siten, että kehittämisen eri vaiheita on helppo hallinnoida. Paikallisessa ympäristössä ajetaan kahta eri palvelinta: toinen käyttöliittymää ja toinen palvelinpuolen toiminnallisuuksia varten. Kyseisellä menetelmällä erotellaan nämä kaksi kokonaisuutta omikseen, ja niitä voidaan myös erinäisinä kehittää. Käyttöliittymäpuolen kehittäjän ei tarvitse ladata kuin käyttöliittymäpuolen kokonaisuus, jota hän pystyy ajamaan paikallisesti. Toisena isona etuna tällaisessa rakenteessa on välttyä ongelmilta selainten saman alkuperän käytännön kanssa. Tämä käytäntö on selaimissa turvallisuussyistä käytössä. Sen tarkoituksena on estää resurssien hakeminen erinimisestä verkkotunnuksesta kuin mistä kutsu lähetettiin alun perin. Ongelma pystytään ohittamaan määrittelemällä CORS (Cross-Origin Resource Sharing), jolloin sallimme erikseen kaikki tarvittavat resurssit. CORS:n määrittäminen vaatii kuitenkin lisätyötä koodin ja ylläpidon osalta, joten lähestymme ongelmaa hieman eri tavalla. Pyrimme säilyttämään koodissa kaikkien resurssien polut sellaisessa muodossa, että ne toimivat saman verkkotunnuksen alta. Kahden palvelimen paikallisessa kehitysympäristössä määritellään frontend-palvelimelle resurssit, joihin tulevat kutsut ohjataan Node.js/Express.js -palvelimelle. [14; 15.]



Kuva 4. Paikallinen kehitysympäristö noudattaa kaavaa, jossa kaikki pyynnöt lähetetään käyttöliittymäpuolen palvelimelle. Palvelin 1 osaa ohjata tiettyyn resurssiin lähetetyt API-kutsut toiselle palvelimelle. [15.]

Kun sovellus julkaistaan, pakataan palvelimen 1 kokonaisuus kahteen eri tiedostoon, jolloin jäljelle jää vain yksi palvelin. Palvelin 2 on määritelty siten, että julkaistussa alustassa se palauttaa nämä kaksi tiedostoa, jotka sisältävät kaiken informaation käyttöliittymäpuolen ominaisuuksista ja toiminnoista. Julkaistuna sovellus toimii vain yhden verkkotunnuksen alta, jolloin ongelmia saman alkuperän käytännön kanssa ei pääse syntymään. [15.]

5.3. Versionhallinta

Versionhallinta on yksi sovelluskehityksen tärkeimmistä asioista. Koodiin tehdyt muutokset täytyy pystyä jäljittämään ja tarvittaessa palauttamaan. Käytämme tämän toiminnallisuuden saavuttamiseksi Git-nimistä versionhallintaohjelmaa. Git-ohjelman avulla tallennetaan tiedostoista ns. kuvankaappauksia sovelluksen eri vaiheista. Sovelluksen eri vaiheet ja ominaisuudet voidaan kehittää omissa versio haaroissa, jotka sulautetaan sitten valmiiseen ohjelmarunkoon. Ohjelman runkoa ja eri kehityshaaroja säilytetään yleensä keskitetyssä tallennuspaikassa esimerkiksi Github-nimisessä palvelussa. Sovelluksemme tekeminen alkaa asentamalla Git-ohjelma Windows-koneeseen ja määrittelemällä tietty kansio Git-versionhallinnan alle. Tämä tapahtuu ajamalla komento “git init”, luodussa projektikansiossa. Tästä lähtien Git-ohjelma seuraa uusia muutoksia, ja ne voidaan lisätä osaksi valmista ohjelmaa sitä mukaan, kun se valmistuu. Jos tiedostoja halutaan sulkea pois tästä seurannasta, voidaan halutut kansiot tai tiedostojen nimet lisätä tiedostoon nimeltä “.gitignore”. Yleisimmin käytetyt komennot Git-ohjelmassa ovat: [16.]

- git add valitsee tiedostoja lisättäväksi.
- git commit lisää tiedostot nykyiseen julkaisukokonaisuuteen.
- git push työntää julkaisukokonaisuuden keskitettyyn tallennustilaan esim. github.com.
- git pull --rebase, lataa uusimman version nykyiseen kehitys haaraan ja asettaa omat muutoksesi uudempien edelle.
- git status, näyttää nykyisen version tilanteen. [16.]

5.4. Palvelinpuoli ja kirjautuminen

Voidaksemme toteuttaa kirjautumisen, todentamisen ja sovelluksessa käytettävien tietojen tallennuksen tarvitsemme palvelimen, joka hallinnoi näitä toiminnallisuuksia. Valmistelemme projektikansion Node.js-ympäristöä varten ajamalla komennon “npm init”, joka lisää hakemistoon tiedoston nimeltä “package.json”. Tiedosto sisältää tiedot Node.js-ympäristöä varten, kuten käytettävät lisäosat ja käynnistysskriptit. Node.js-ohjelmien paikallinen asentaminen tapahtuu joko ajamalla “npm install --save paketin_nimi” tai ajamalla “npm install”, jos kaikki tarvittavat tiedot ovat jo package.json-tiedostossa. Moduulien tiedostoja ei myöskään tarvitse siirtää kehitys- tai julkaisuympäristöjen välillä, vaan ne voidaan asentaa siellä, missä niitä tarvitaan.

```
{
  "name": "server",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "engines": {
    "node": "8.9.1",
    "npm": "5.5.1"
  },
  "scripts": {
    "start": "node index.js",
    "server": "nodemon index.js",
    "client": "npm run start --prefix client",
    "dev": "concurrently \"npm run server\" \"npm run client\"",
    "heroku-postbuild": "NPM_CONFIG_PRODUCTION=false npm install --prefix client && npm run build --prefix client"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "axios": "^0.18.0",
    "body-parser": "^1.18.2",
    "concurrently": "^3.5.1",
    "cookie-session": "^2.0.0-beta.3",
    "express": "^4.16.2",
    "google-calendar": "^1.3.2",
    "mongoose": "^5.0.6",
    "nodemon": "^1.15.1",
    "passport": "^0.4.0",
    "passport-google-oauth20": "^1.0.0",
    "passport-oauth2-refresh": "^1.0.0",
    "q": "^1.5.1",
    "reactjs-popup": "^1.2.0"
  }
}
```

Kuva 5. Kuvassa esiteltynä palvelinpuolen package.json-tiedoston sisältö. Tiedostossa ovat määriteltynä Node.js-sovelluksen perustiedot kuten versiot lisäosista ja käynnistysskriptit.

Index-nimistä tiedostoa etsitään vakiona ajettavaksi eri web-ympäristöissä. Näin ollen projektikansion konfiguraation jälkeen lisätään esimerkkikoodi 2:n mukainen koodinpätkän index.js-nimiseen tiedostoon juurihakemistossa. Yksinkertainen Express.js-palvelin on nyt käytettävissä. Pinoon kuuluva MongoDB-tietokanta voidaan asentaa paikallisesti tai vaihtoehtoisesti käyttää jotakin valmista palveluntarjoajaa. Käytämme tässä projektissa jälkimmäistä ratkaisua helpon käytettävyyden takia sekä välttääksemme liiallisia asennuksia paikallisessa ympäristössä. mLab.com tarjoaa käyttäjätilin luoneille 500 MB ilmaista tallennustilaa, johon perustamme kaksi NoSQL-tietokantaa. Yhden kehitystä varten ja toisen käytettäväksi ohjelman julkaistussa versiossa. Index.js-tiedostoa jatketaan lisäämällä siihen tietokannan kanssa keskusteleva toiminnallisuus. Tämän toiminnallisuuden aikaansaamiseksi tarvitsemme paketin nimeltä “mongoose”. Kyseessä on objektien mallinnus työkalu, joka pystyy keskustelemaan NoSQL-tietokannan kanssa. Jokaiselle tallennettavalle datalle määritellään etukäteen malli ja kokoelma, mihin se

kuuluu. Määriteltyä mallia voimme hyödyntää ohjelmassa objektin tavoin, jolle löytyy funktiot tavallisten CRUD-operaatioiden suorittamista varten.

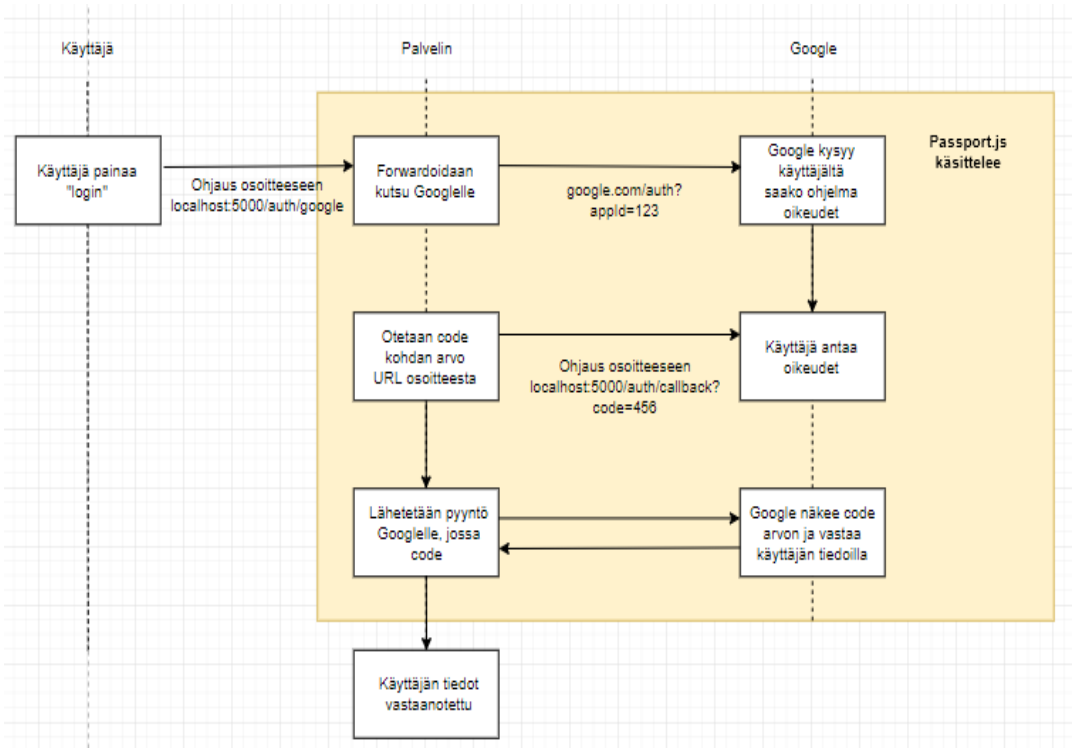
```
const mongoose = require("mongoose");
const { Schema } = mongoose;

const userSchema = new Schema({
  userName: String,
  googleId: String,
  refreshToken: String,
  accessToken: String,
  accessTokenExpiresIn: Date
});

mongoose.model("users", userSchema);
```

Esimerkkikoodi 4. Koodissa luodaan uusi mallin nimeltä userSchema, jolle määritellään objektin tarvitsemat attribuutit JSON-muodossa.

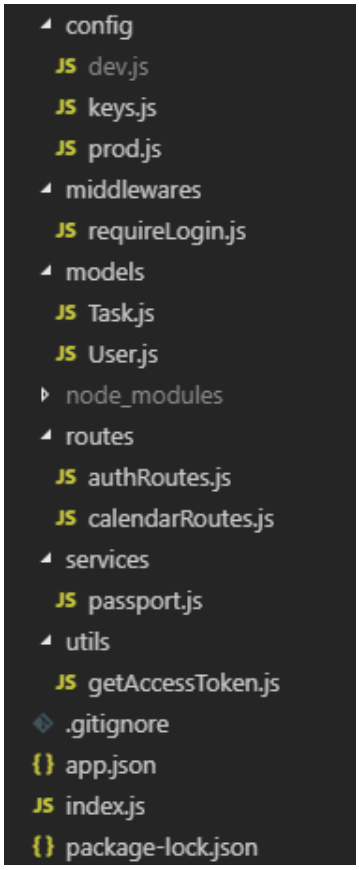
Kirjautumista ja käyttäjän todennusta varten valitaan OAuth-menetelmän, jolla käyttäjä voi kirjautua sovellukseen Googlen tunnuksia käyttäen. Tällä tavoin käyttäjän on helppo aloittaa sovelluksen käyttö eikä käyttäjätunnusten ja salasanojen tallennuksesta tarvitse huolehtia. Tallennamme siis vain käyttäjäkohtaisen tunnisteen tietokantaan, jolla voimme identifioida käyttäjän. Käytämme Passport.js-väliohjelmistoa helpottamaan OAuth-menetelmän toteuttamista.



Kuva 6. Kirjautumisen todentamiseen käytetty OAuth-menetelmä ja missä määrin Passport.js osallistuu sen eri vaiheissa. [17]

Kuten voimme kuvasta 6 havaita, on todentamisessa monia eri vaiheita ennen kuin sovellus on sallittu vastaanottamaan käyttäjän tietoja. Passport.js tarvitsee toimiakseen strategian Googlen tunnistautumista varten. Strategiassa määritellään takaisinkutsu URL-osoite sekä sovelluksen käyttämät API-avaimet, jotka saamme Googlelta rekisteröimällä sovellus Googlen Developer Console -alustassa. Kun käyttäjä kirjautuu, ohjaa palvelimemme hänen pyyntönsä Googlelle, joka näyttää käyttäjälle suostumusnäkymän. Suostumuksen annettuaan saa palvelin vastauksena koodin, jota vastaan saamme Googlelta käyttäjän profiilitiedot uutta kyselyä vastaan. Vastaanotettuamme käyttäjän tiedot ohjaamme käyttäjän sovelluksessa haluttuun osioon ja voimme lisätä uuden dokumentin tietokantaan käyttäjän myöhempää tunnistamista varten. Kirjautumisen yhteyteen lisätään vielä ominaisuus evästeiden asettamista varten, jotta käyttäjän ei tarvitse heti kirjautua uudelleen. Evästeiden asettaminen toteutetaan cookie-session lisäosaa käyttäen. Cookie-session saa attribuutteina evästeen voimassaoloajan sekä salausavaimen. Google-kalenterin toiminnallisuus vaatii käyttäjältä hyväksynnän kalenteriin. Pas-

sport.js-väliohjelmiston todennus kyselyyn voi lisätä vaihtoehdon, joka aiheuttaa käyttäjälle kyselyn hyväksyä sovelluksen pääsy lukemaan ja muokkaamaan kalenteria. [18; 19.]



Kuva 7. Palvelinpuolen kansiorakenne. Config-kansiossa sijaitsevat julkaisu- ja kehitysympäristöjen käyttämät avaimet. Middlewares-kansio sisältää toiminnallisuuksia, joita haluamme putkittaa esimerkiksi API-kutsuun. Models-kansiossa sijaitsevat määrittelyt MongoDB-tietokantaa varten. Eriteltynä ovat myös API-resurssit routes-kansiossa kuten myös kirjautumiseen ja todentamiseen tarvittavaa koodia services- ja utils-nimisissä kansioissa.

5.5. Käyttöliittymä

Web-sovelluksemme pystyy vastaamaan kutsuihin ja suorittamaan toiminnallisuuksia, mutta tarvitsemme ihmisille suunnatun käyttöliittymän. Käyttöliittymä toteutetaan käyttämällä React.js-kehystä lisäosineen. Projektissa käytetään pohjana Facebook:in julkaisemaa pakettia nimeltä create-react-app. Paketti on konfiguroitu valmiiksi, jotta React.js-sovelluksen kehittämisen voi aloittaa helposti. Yhtenä tärkeänä ominaisuutena on käyttöliittymäpuolen julkaisuversion luonti, joka pakkaa koko käyttöliittymäpuolen kokonaisuuden muutamiin tiedostoihin. Create-react-app on konfiguroitu käynnistämään komennosta palvelin osoitteeseen http://localhost:3000, jossa omia muutoksiaan voi testata reaaliajassa. React.js-sovellus rakentuu ensin määrittelemällä index.html-tiedosto sisältäen div-elementin, jolle on määritelty id-attribuutti. Id-attribuutti voi olla mikä tahansa, mutta yleisesti React.js-sovelluksissa sille annetaan arvoksi root. Tähän root id:llä varustetun div-elementin sisälle renderöidään React.js-komponentit. Tässä projektissa lisätään vielä Redux- ja Redux-Thunk-kirjastot sovelluksen ylätasoon, jotta ne ovat käytettävissä koko sovelluksen tiedonkulussa. [20.]

```
import React from "react";
import ReactDOM from "react-dom";
import { Provider } from "react-redux";
import { createStore, applyMiddleware } from "redux";
import reduxThunk from "redux-thunk";
const store = createStore(reducers, {}, applyMiddleware(reduxThunk));

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.querySelector("#root")
);
```

Esimerkkikoodi 5. Koodissa renderöidään App-niminen komponentti react-dom-pakettia käyttäen. React-dom tarjoaa funktioita DOM-manipulointia varten ja sitä käytetään vain sovelluksen ylätasossa. Redux.js-kirjastosta käyttöön otettu createStore()-funktioilla määrittelemme uuden tilojen säilön nimeltä store. Uusi tilojen säilö saa parametreikseen sovittajat sekä applyMiddleware()-funktion avulla redux-thunk-väliohjelmiston. Redux.js ei ole suoraan yhteensopiva React.js-kehyksen kanssa, joten react-redux-niminen kirjasto vaaditaan, jotta React.js-komponentit voivat keskustella Redux.js-säilön kanssa. [21.]

React.js-sovellus perustuu erillisiin komponentteihin, joita näytetään tarvittaessa. Näistä tilan muutoksesta johtuen renderöityvistä komponenteista rakennetaan sovelluksessa tarvittavat osiot. Kaikkien tarvittavien komponenttien välille saadaan yhteys liittämällä ne Redux.js-kirjastolla luotuun yhteiseen säilöön. Tilojen tiedonkulku sovelluksessa on tällöin kuvan 3 mukainen ja koostuu kolmesta päävaiheesta:

- Toiminnon luoja on funktio tai toiminnallisuus, joka palauttaa toiminnon.
- Toiminto on sen toimintoa kuvaavalla nimellä nimetty ja voi sisältää toimintoa varten dataa.
- Sovittaja näkee kaikki toiminnot ja asettaa tilan.

Projektissa käytämme kyseistä tiedonkulkua mm. päättämään, mitkä elementit käyttäjälle näkyvät. App-komponentissa määritellään sovelluksen kaikki pääkomponentit kuten Header- ja Dashboard-komponentit. Tämä komponentti on liitetty react-redux-kirjaston connect()-funktiolla Redux.js-säilöön. Connect-nimiselle funktiolle annetaan parametreinä Redux.js-tiedonkulussa käytettävät funktiot. Koska kyseessä on sovelluksen pääkomponentti, liitämme toiminnon luojat tässä koko sovelluksen käytettäväksi. Yhtenä toiminnon luojana on fetchUser-niminen funktio, joka hakee API-palvelimelta nykyisen käyttäjän tiedot. Haluamme varmistaa, että asynkronista tapahtumasarjaa jatketaan vasta kun tieto on saatu palvelimelta. Tätä varten redux-thunk on määritelty reduxin väliohjelmistoksi, joka aktivoituu, kun sille palautetaan funktio toiminnon sijaan. Näin ollen tapahtumasarja keskeytyy ja jatkuu vain, jos palautamme itse toiminnon. Koodissa palautamme toiminnon vasta, kun olemme saaneet vastauksen, jolloin viive tai virhe ei vääristä tilan sisältöä. Toiminto siirtyy seuraavaksi sovittajille, joiden tarkoituksena on määrittää tila toiminnon mukaan. [22.]

```
export const fetchUser = () => {
  return function(dispatch) {
      axios
      .get("/api/current_user")
      .then(res => dispatch({ type: FETCH_USER, payload: res.data }));
  };
};
```

Esimerkkikoodi 6. Palautamme funktion, joka ottaa vastaan normaaliin tiedonkulkuun kuuluvan dispatch-funktion. Axios-nimisellä lupaus pohjaisella HTTP-asiakasohjelmalla haetaan API-palvelimelta käyttäjän tiedot. Vasta vastauksen jälkeen toimintoa jatketaan ja palautamme dispatch-funktion toiminnolla ja sen datalla.

```
export default function(state = null, action) {
  switch (action.type) {
    case FETCH_USER:
      return action.payload || false;
    default:
      return state;
  }
}
```

Esimerkkikoodi 7. Sovittaja ottaa vastaan tilan ja toiminnon. Tila voidaan määritellä olevan alussa null, joka vaihdetaan saadun toiminnon mukaan. Esimerkissä tarkistetaan toiminnon nimi ja palautetaan tilan arvoksi null, false tai saatu käyttäjäobjekti.

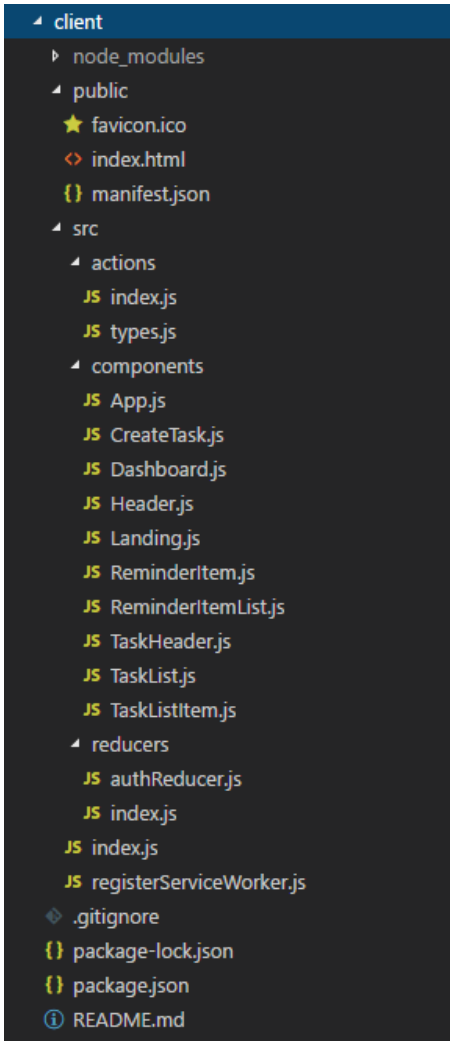
Projektissamme tilan muutos sai alkunsa ajamalla toiminnan luojiin kuuluva fetchUser-funktio App-komponentissa. Sen alla oleva Header-komponentti on myös liitetty osaksi Redux-js-säilöä connect()-funktiota käyttäen. Header-komponentissa määritellään mapStateToProps()-funktio osaksi connect()-funktiota, jolloin kyseinen funktio ajetaan, kun tilassa tapahtuu muutos. Tässä funktiossa voimme palauttaa tälle komponentille tarvittavan tilan, joka liitetään props-objektiin. Props-objekti kuuluu React.js-komponenttien vakio-ominaisuuksiin, joka voi sisältää dataa komponentissa. Redux.js osaa vertailla mapStateToProps()-funktion palauttamaa tilan arvoa ja, kun muutos havaitaan, komponentti päivittyy.

```
class Header extends Component {
   renderContent() {
    switch (this.props.auth) {
      case null:
        return;
      case false:
        return (
          <li>
            <a href="/auth/google">Login With Google</a>
          </li>
        );
      default:
        return (
          <div>
            <li>
              <a href="/main/viewTask">Dashboard</a>
            </li>
            <li>
              <a href="/api/logout">Logout</a>
            </li>
          </div>
        );
    }
  }

  render() {
      return (
      <nav>
        <div className="nav-wrapper">
            <ul className="right">{this.renderContent()}</ul>
        </div>
      </nav>
    );
  }
}

function mapStateToProps(state) {
   return { auth: state.auth };
}
export default connect(mapStateToProps)(Header);
```

Esimerkkikoodi 8. Header-komponentin props-objektiin lisätään auth-niminen muuttuja, joka sisältää tiedon todennuksesta. Todennuksen arvon mukaan päätetään, mikä komponentti näytetään käyttäjälle.



Kuva 8. Käyttöliittymäpuolen kansiorakenne esiteltynä, jossa components-kansion alla olevat komponentit muodostavat näkyvän osan sovelluksesta. Osaksi Redux.js -tapahtumasarjaan kuuluvat actions- ja reducers-kansion alla olevat tiedostot.

5.6. Sovelluksen julkaisu Heroku-alustalle

Verkkopalvelun viimeisiin vaiheisiin kuuluu sovelluksen julkaisu. Projektille on valittu Heroku-niminen julkaisualusta, joka tukee Node.js-sovellusten lisäksi myös muilla tekniikoilla ja kielillä luotuja sovelluksia kuten Ruby, Java, PHP ja Python. Heroku-alusta vaatii sovellukselta pieniä muutoksia, jotta se olisi julkaisukelpoinen kyseiselle alustalle. API-palvelin määritellään käyttämään Heroku-alustan antamaa porttia sekä tarjoamaan käyttöliittymäpuolen tuotantoversiotiedostot. Kyseisen konfiguraation toteuttamiseen käytetään alustan omia muuttujia, joihin pääsemme käsiksi kutsumalla process.env.muuttujan_nimi. Samanlailla hallinnoidaan kehitys ja tuotantoympäristössä käytettäviä avaimia, jotka ovat eriteltyinä omissa tiedostoissaan. [23.]

```
if (process.env.NODE_ENV === "production") {
  app.use(express.static("client/build"));
  const path = require("path");
  app.get("*", (req, res) => {
    res.sendFile(path.resolve(__dirname, "client", "build", "index.html"));
  });
}

const PORT = process.env.PORT || 5000;
```

Esimerkkikoodi 9. Heroku-alusta asettaa NODE_ENV-muuttujan, joka kertoo sovelluksen olevan tuotantoympäristössä. Kun sovellus havaitsee olevansa tuotantoympäristössä, tarjoaa se käyttöliittymän tuotantoversiotiedostot. Sovelluksen käyttämä portti määräytyy asetetun PORT-ympäristömuuttujan mukaan. Jos sellaista ei ole asetettu, käytetään porttia 5000, jolloin ollaan kehitysympäristössä.

```
if (process.env.NODE_ENV === "production") {
  module.exports = require("./prod");
} else {
  module.exports = require("./dev");
}

prod.js tiedosto:

module.exports = {
  googleClientID: process.env.GOOGLE_CLIENT_ID,
  googleClientSecret: process.env.GOOGLE_CLIENT_SECRET,
  mongoURI: process.env.MONGO_URI,
  cookieKey: process.env.COOKIE_KEY
};
```

Esimerkkikoodi 10. Avaimet on tallennettu Heroku-alustan palvelimelle ja ovat käytettävissä sen ympäristömuuttujista.

Sovelluksessa käytettävien avainten erottelu tuo tietoturvaa ja hallintaa sovelluksen kehityksessä. Ympäristömuuttujat voi huoletta julkaista esimerkiksi Github-palvelussa, jossa muut kehittäjät saattavat nähdä ne. Kehitysympäristön avaimia käytetään vain paikallisesti ja ne on sen takia jätetty pois Git-versionhallinnasta. Vaikka paikallinen tietokoneemme joutuisi hyökkäyksen kohteeksi olisi tuotantoversio turvassa ympäristömuuttujien ansiosta. Projektimme kehitysympäristössä on tällä hetkellä käytössä kaksi palvelinta. Toinen näistä poistuu, kun pakkaamme käyttöliittymäpuolen tuotanto versioksi. Sovellus konfiguroidaan siten, että Heroku-alusta asentaa kaikki tarvittavat riippuvuudet kuten kirjastot ja kehykset ja luo tuontantoversion omilla palvelimillaan.

```
"scripts": {
    "start": "node index.js",
    "server": "nodemon index.js",
    "client": "npm run start --prefix client",
    "dev": "concurrently \"npm run server\" \"npm run client\"",
    "heroku-postbuild": "NPM_CONFIG_PRODUCTION=false npm install --prefix cli-
ent && npm run build --prefix client"
  },
```

Esimerkkikoodi 11. API-palvelimen package.json-tiedostossa määrittelemme heroku-postbuild -nimisen skriptin, jonka Heroku-alusta ajaa julkaistaessa sovellusta. [24]

Sovellus on nyt valmiina julkaistavaksi ja aivan viimeisenä päätöksenä tarvitsee päättää julkaisutapa. Heroku -alusta tarjoaa oman Git-pohjaisen julkaisutavan ja versionhallinnan. Käytämme projektissa kuitenkin Github-palvelua lähdekoodin tallennukseen ja versionhallintaan. Github-palvelu kuuluu yhtiin Heroku-alustan tuetuista julkaisutavoista, jotka toimivat keskenään hyvin. Sovelluksen eri versiot ladataan Github-palveluun, josta Heroku-alusta määritellään seuraamaan pääkehityshaaraa. Muutoksen havaittuaan alkaa Heroku luoda uutta tuotantoversiota ja julkaisee sen omilla palvelimillaan. Sovelluksemme on näin ollen julkaistu ja kenen tahansa käytettävissä.

6. Yhteenveto

Sovelluskehitys on laaja ja monimuotoinen prosessi, johon ei löydy yhtä ja oikeaa toteutustapaa. Työssä on pyritty havainnollistamaan verkkopalvelun toteutus yhtä JavaScript -pohjaista sovelluspinoa ja sen ympärillä käytettäviä lisäosia käyttäen. MERN-pino ja sen ympärillä oleva kehitys kuuluu uuteen tapaan käyttää JavaScript-kieltä sovelluskehityksen kaikissa osa-alueissa. Suurimmat erot ja hyödyt löytyvät mielestäni käyttöliittymäpuolesta, jossa sovelluksen osat keskustelevat tehokkaasti keskenään luoden toimivampia kokonaisuuksia käyttäjälle. Perinteiseen tapaan verrattuna React.js-kirjastolla luotua käyttöliittymää on yksinkertaisempi ylläpitää sen komponenttipohjaisuuden takia. React.js- ja Redux-yhdistelmän tiedonkulku voi aluksi vierastaa, mutta oikein toteutettuna ratkaisee ongelmia skaalautuvuudessa ja suorituskyvyssä. Yhtenäinen kieli kaikissa osa-alueissa tuo mm. yhtenäisemmät kehitysmetodit ja työkalut. Kehittäjä voi keskittyä yhteen kieleen, mutta voi olla kehittämässä käyttöliittymää sekä palvelinpuolen toimintoja. Objektipohjainen lähestymistapa tiedon tallennuksessa ja JSON-muotoisen tiedon siirtyminen sovelluksen sisällä on helposti ihmisen ymmärrettävissä, jolloin keskittyminen ja luovuus painottuvat ideoiden toteuttamiseen. Koska lähestyimme sovelluskehitystä vain toteuttamisen kannalta ja kävimme läpi tämänkaltaisen sovelluksen sisäistä tiedonkulkua, jäi sovelluskehitykselle ominaisia asioita läpikäymättä. Tämän kokoluokan sovelluksessa mm. testaus onnistuu hyvin paikallisesti, mutta laajemmissa sovelluksissa testaus kannattaa jo automatisoida laadun varmistamiseksi. Tämän insinöörityön teko on auttanut hahmottamaan ja ymmärtämään asioita, joita sovelluskehityksessä pitäisi ottaa huomioon. Sovelluksen luomiseen tarvittavien prosessien ja sovelluksen sisäisten tiedonkulkujen ymmärrys on kehittynyt ja vahvistunut.

Lähteet

1. Gellert, Laurence. 2012. What is a Full Stack developer. Verkkoaineisto: https://www.laurencegellert.com/2012/08/what-is-a-full-stack-developer/. Luettu 23.6.2018.

2. Developer Survey Results 2017. Stack Overflow. Verkkoaineisto: https://insights.stackoverflow.com/survey/2017/. Luettu 3.7.2018.

3. Carey Wodehouse. 2015. Verkkoaineisto: https://www.upwork.com/hiring/development/choosing-the-right-software-stack-for-your-website/. Luettu 23.9.2018.

4. MVC architecture. Mozilla Foundation. Verkkoaineisto: https://developer.mozilla.org/en-US/docs/Web/Apps/Fundamentals/Modern_web_app_architecture/MVC_architecture/. Luettu 2.7.2018.

5. Srinivasan Subramanian. 2017. Verkkoaineisto: https://www.apress.com/gp/blog/all-blog-posts/why-mern/12056000/. Luettu 10.10.2018.

6. 9 code and framework trends to watch in 2018. TechBeacon. Verkkoaineisto: https://techbeacon.com/9-code-framework-trends-watch-2018/. Luettu 10.10.2018.

7. Olga Trad. 2018. Verkkoaineisto: https://www.netguru.co/codestories/should-you-learn-nodejs-in-2018/. Luettu 10.10.2018.

8. Getting started. Express. Verkkoaineisto: https://expressjs.com/. Luettu 20.11.2017.

9. What is MongoDB. MongoDB Inc. Verkkoaineisto: https://www.mongodb.com/what-is-mongodb/. Luettu 20.11.2017.

10. Read Me. Redux. Verkkoaineisto: https://redux.js.org/. Luettu 23.09.2018.

11. Jeroen Savat & Andy Somers. 2017. Verkkoaineisto: https://www.foreach.be/blog/why-the-react-redux-combo-works-like-magic/. Luettu 18.10.2018.

12. Readme.md. Redux. Verkkoaineisto: https://github.com/reduxjs/redux-thunk/. Luettu 01.10.2018.

13. Documentation. Passport. Verkkoaineisto: http://www.passportjs.org/docs. Luettu 23.12.2017.

14. Cross-Origin Resource Sharing (CORS). Mozilla Foundation. Verkkoaineisto: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS/. Luettu 05.10.2018.

15. Stephen Grider. 2017. Verkkoaineisto: https://www.udemy.com/node-with-react-fullstack-web-development/learn/v4/t/lecture/7605040?start=0/. Luettu 23.11.2017.

16. Scott Chacon & Ben Straub. 2009. Verkkoaineisto: https://git-scm.com/book/en/v1/. Luettu 23.9.2017.

17. Stephen Grider. 2017. Verkkoaineisto: https://www.udemy.com/node-with-react-fullstack-web-development/learn/v4/t/lecture/7593706?start=0/. Luettu 23.12.2017.

18. Authorizing Requests to the Google Calendar API. Google LLC. Verkkoaineisto: https://developers.google.com/calendar/auth/. Luettu 15.10.2018.

19. Using OAuth 2.0 to Access Google APIs. Google LLC. Verkkoaineisto: https://developers.google.com/identity/protocols/OAuth2/. Luettu 15.10.2018.

20. Readme.md. Facebook. Verkkoaineisto: https://github.com/facebook/create-react-app/. Luettu 13.09.2018.

21. Getting Started. React. Verkkoaineisto: https://reactjs.org/docs/getting-started.html/. Luettu 20.11.2017.

22. Quick Start. Redux. Verkkoaineisto: https://react-redux.js.org/docs/introduction/quick-start/. Luettu 28.11.2017.

23. Configuration and Config Vars. Heroku Inc. Verkkoaineisto: https://devcenter.heroku.com/articles/config-vars/. Luettu 23.10.2018.

24. Heroku Node.js Support. Heroku Inc. Verkkoaineisto: https://devcenter.heroku.com/articles/nodejs-support/. Luettu 23.10.2018.