Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Miro Tolkkila

Verkkosovelluksen kehittäminen MERN-pinon avulla

Metropolia Ammattikorkeakoulu

Insinööri (AMK)
Tieto- ja viestintätekniikka
Insinöörityö
22.5.2020

Tekijä Otsikko Sivumäärä Aika	Miro Tolkkila Verkkosovelluksen kehittäminen MERN-pinon avulla 28 sivua 22.5.2020
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Ohjelmistotuotanto
Ohjaajat	Ohjaava opettaja Juha Kämäri

Nykyään verkkosovelluksien kehittämiseen löytyy useita eri teknologioita. Erityiseen suosioon on noussut Node.js-ympäristö, minkä avulla JavaScript-koodi voidaan suorittaa suoraan palvelimella. Tämä mahdollistaa JavaScriptin käyttämisen kaikissa verkkosovelluksen osissa, mikä helpottaa kehittämistä.

Verkkosovellus koostuu eri osista: selain- ja palvelinpuolesta (front-end ja back-end). Lisäksi yleensä tarvitaan tietokanta. Full-stack -kehittäjällä tarkoitetaan henkilöä, joka hallitsee verkkosovelluksen jokaisen osan. Full-stackin suosion myötä on syntynyt erilaisia valmiita pinoja, jotka helpottavat verkkosovelluksien kehittämistä.

Insinöörityön tavoitteena on kehittää oma verkkosovellus Node.js-alustaan muodostuneen MERN-pinon avulla. Sovelluksen on tarkoitus auttaa käyttäjää hahmottamaan omaa rahankäyttöä. Käyttäjä voi sovelluksessa kirjata ylös tuloja, menoja ja budjetteja. Verkkosovellus näyttäisi erilaisia kaavioita, jotta rahankäytön tulkinta olisi helpompaa.

Työssä käydään läpi MERN-pinon sekä sen ulkopuoliset teknologiat, vertaillaan erilaisia kaaviokirjastoja sekä esitellään sovelluksen toimintaperiaate.

Avainsanat	MERN, MongoDB, Express.js, React.js, Node.js, kaaviokirjasto

Author Title Number of Pages Date	Miro Tolkkila Developing Web Application With MERN Stack 28 pages 22 May 2020
Degree	Bachelor of Engineering
Degree Programme	Information and communication technology
Professional Major	Software engineering
Instructors	Juha Kämäri, Project Instructor

Today there are several different technologies available for developing web applications. One technology has shown to be extremely popular amongst the others. That technology is Node.js, which allows JavaScript code to be run directly on a server. This allows JavaScript to be used in all parts of the web application, which facilitates development.

Web application consists of different parts: browser and server side (front-end and back-end). In addition, a database is usually required. A full-stack developer is a person who controls every part of a web application. With the popularity of the full-stack, a variety of ready-made stacks have emerged to facilitate the development of web applications.

The aim of this thesis is to develop a web application using MERN stack that has formed on the Node.js environment. The application is intended to help the user to perceive their own use of money. The user can record income, expenses and budgets in the application. This data would then be displayed on the web application by using various charts, which makes it easier to interpret the use of money.

On this thesis I will present MERN stack and other necessary technologies to create the web application. I will also compare a few different chart libraries and present the operating principle of the application.

Keywords	MERN, MongoDB, Express.js, React.js, Node.js, chart library

Sisällys

Lyhenteet

1 Johdanto 1

2 MERN-pino 1

2.1 Node.js 2
2.2 React.js 3
2.3 Expess.js 5
2.4 MongoDB 6

3 Lisäosat 6

3.1 Passport.js 6
3.2 Redux 7
3.3 Kaaviokirjastot 7
3.3.1 Victory 7
3.3.2 Chartist.js 9
3.3.3 React-Vis 11
3.3.4 Päätelmät kaaviokirjastoista 12

4 Verkkosovelluksen toteutus 12

4.1 Kehitysympäristö 13
4.2 Sovelluksen rakenne 13
4.3 Palvelinpuoli 14
4.4 Tietokanta 16
4.5 Kirjautuminen 18
4.6 Käyttöliittymä 20

5 Yhteenveto 26

Lähteet 27

Lyhenteet

CRUD	Create, Read, Update, Delete. Lyhenne perusfunktioille, joita käytetään tietojenhallintaan.
DOM	Document Object Model. Malli, jonka avulla määritellään dokumentissa olevat elementit
HTTP	Hyper Text Transfer Protocol. Selainten ja verkkosivujen välisessä tiedonsiirrossa käytettävä protokolla.
JSON	JavaScript Object Notation. Avoimen standardin tiedostomuoto tiedonvälitykseen.
NPM	Node Package Manager. Paketinhallintajärjestelmä Node.js-ympäristön paketteja varten
ORM	Object-Relational Mapping. Oliomallin mukaisen esityksen kuvaus relaatiomallin mukaiseksi esitykseksi.
SPA	Single Page Application. Verkkosovellus, jonka kaikki komponentit näytetään yhden html-tiedoston sisällä.
SVG	Scalable Vector Graphics. XML-pohjainen tekstiformaatti, jonka määritellään, miten kuvan tulee piirtyä.

1 Johdanto

Verkkosovelluksien kehittämiseen tarjotaan nykyisin paljon erilaisia ympäristöjä sekä kehyksiä. Vaihtoehtojen määrä voi tuntua sekaannuttavalta aloittelevalle kehittäjälle. Teknologioiden joukosta yksi on noussut suureen suosioon: Node.js. Suuresta suosiosta kertoo erityisesti se, että monet tunnetut yritykset, kuten Netflix, eBay, LinkedIn ja PayPal, ovat ottaneet käyttöönsä Node.js -alustan. [1.]

Verkkosovellukset muodostuvat eri osista: selainpuolesta (front-end), palvelinpuolesta (back-end) ja yleensä myös tietokannasta. Full-stack -kehittäjällä tarkoitetaan henkilöä, joka hallitsee verkkosovelluksen jokaisen osan. Full-stackin suosion myötä on kehittynyt valmiita teknologiapinoja, jotka sisältävät kehyksiä verkkosovelluksien osiin. Tässä insinöörityössä keskitymme yhteen suosittuun Node.js-alustaan kehittyneeseen pinoon. Tavoitteena on opetella verkkosovelluksen kehittämistä hyödyntäen MERN-pinoa, joka sisältää seuraavat teknologiat: MongoDB, Express.js, React.js ja Node.js. Työssä esitellään MERN-pinossa käytettävät sekä sen ulkopoliset teknologiat ja käydään läpi sovelluksen toteuttamisen vaiheet.

Jotta teknologioiden käytön tutkiminen olisi mielekästä, pyrin kehittämään verkkosovelluksen, jolle voisi olla yleistä mielenkiintoa. Tästä syystä valitsin verkkosovellukseksi budjetointityökalun, joka helpottaisi käyttäjää hahmottamaan omaa rahankäyttöä. Verkkosovelluksessa tulee olemaan erilaisia kaavioita, joita voidaan rakentaa ja näyttää erilaisten kaaviokirjastojen avulla. Yksi insinöörityön tavoitteista on myös vertailla muutamaa eri kaaviokirjastoa.

2 MERN-pino

Nykyisin jokainen verkkosovellus toteutetaan käyttämällä useita teknologioita. Näiden teknologioiden kokoelmaa kutsutaan pinoksi (stack). Yksi tunnetuimmista pinoista on LAMP (Linux, Apache, MySQL, PHP), joka sisältää kokoelman avoimen lähdekoodin ohjelmia. Vasan Subramanian kertoo kirjassaan, että verkkosovelluksien kehityksen myötä sovelluksien ja käyttäjien välisestä vuorovaikutuksesta on tullut tärkeää, mikä on johtanut SPA-sovelluksien suosioon. [2.]

Perinteisessä verkkosovelluksessa jokainen datan muutos luo pyynnön palvelimelle ja aiheuttaa käyttäjälle koko verkkosivun uudelleen lataamisen. SPA-sovellukset pyrkivät välttämään kyseisen uudelleen lataamisen. Kun SPA-sovelluksessa tapahtuu datan muutos, verkkoselain lähettää palvelimelle pyynnön ja saa takaisin dataa, joka päivitetään dynaamisesti verkkoselaimen näkymään. Tämä tekniikka on lisännyt selaimen tekemää työtä, mikä on johtanut uusien front-end -teknologioiden kehittymiseen. Nämä uudet teknologiat ovat korvanneet LAMP-pinon.

Yksi nykyisin suosituimmista pinoista on MEAN (MongoDB, Express.js, Angular.js, Node.js). MEAN-pino on Node.js -alustaan muodostunut JavaScript-pohjainen pino, joka soveltuu erinomaisesti SPA-ohjelmien kehittämiseen. Tässä insinöörityössä keskitymme kuitenkin MERN-pinoon, jossa Angular.js korvataan React.js-teknologialla.

2.1 Node.js

Node.js on Ryan Dahlin vuonna 2009 kehittämä avoimen lähdekoodin alusta, joka on rakennettu Google Chromen V8 JavaScript -moottorin ympärille. Node.js:n avulla voidaan kehittää JavaScript-kielisiä palvelinpuolen sovelluksia, jotka voidaan ajaa Node.js:n omalla alustalla käyttöjärjestelmästä riippumatta. [3.] JavaScript-kielinen palvelin on yksi syy Node.js:n suosioon. Front-end kehittäjät, jotka ovat tottuneet käyttämään JavaScriptiä, voivat nyt myös kirjoittaa palvelinpuolen koodia opettelematta uutta ohjelmointikieltä.

Node.js-sovellus ajetaan yhdessä prosessissa, eikä pyynnöt luo uusia säikeitä. Ideana on, että Node.js palvelin ei jää koskaan odottamaan pyynnöstä saatavaa dataa, vaan palvelin siirtyy heti seuraavaan pyyntöön suoritettuaan edellisen. Palvelin saa takaisin ilmoituksen aiemmin lähetetystä pyynnöstä, kun se on valmis, ja jatkaa operaatioita tämän jälkeen. [4.]

Kuva 1. Havainnollistava kuva Node.js-palvelimen säikeen toimintaperiaatteesta. [5]

Kuvan 1 mukainen tekniikka nopeuttaa sovelluksen toimintaa, koska palvelin ei tuki säiettä missään vaiheessa. Se myös mahdollistaa tuhansien samanaikaisten yhteyksien pyörittämisen yhdellä palvelimella ilman säikeiden hallitsemista. Kyseistä tekniikkaa varten Node.js tarjoaa asynkronisia funktioita sen standardi kirjastossa. [5.]

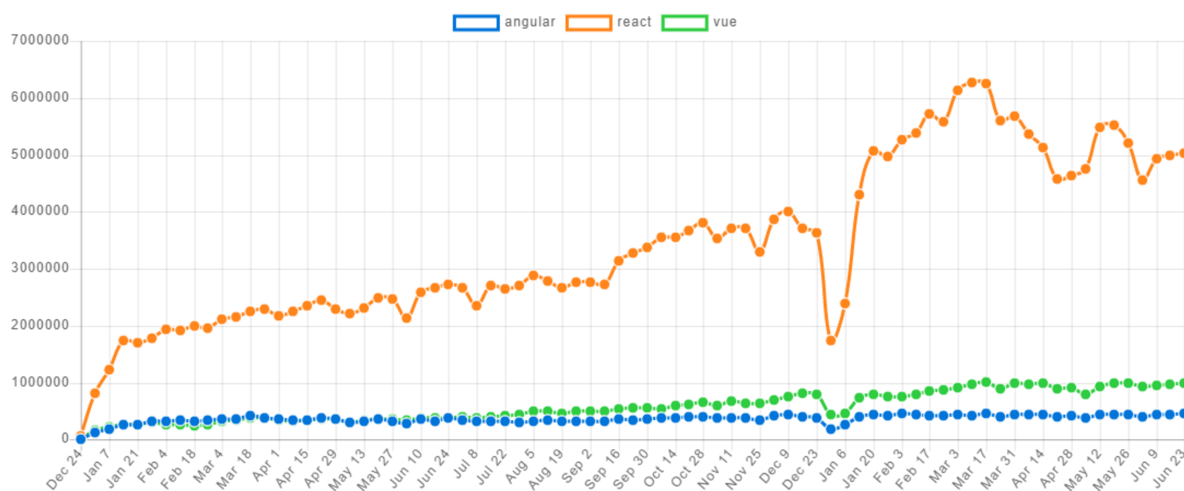
Tärkeä osa Node.js-alustaa on NPM, eli Node Package Manager. Se on verkossa toimiva varasto avoimen lähdekoodin Node.js-projekteille, jotka ovat yleensä erilaisia viitekehyksiä. NPM toimii myös komentorivitulkkina, jolla voidaan asentaa kyseisiä viitekehyksiä. [6.]

2.2 React.js

React.js on Facebookin kehittämä JavaScrpit-kirjasto, mitä käytetään verkkosovelluksien näkymien tekemiseen. Front-end -teknologioiden kärkikolmikkona on pitkään ollut AngularJS, Vue.js ja React.js. Toisin kuin AngularJS tai Vue.js, React.js ei ole kehys, joten se ei määrittele sovelluksen mallia. Yleensä React.js-kirjastoa käytetään renderöimään MVC-mallin View-komponenttia.

Yksi suuri hyöty ohjelmoijan kannalta on se, että React.js päivittää muutokset HTML DOM -malliin automaattisesti. React.js-kirjaston näkymät ovat deklaratiivisia, eli React määrittää saadun datan perusteella itse, miltä näkymät näyttävät. Ohjelmoija ei määrittele todellista DOM-mallia, vaan virtuaalisen rakenteen. React.js pystyy laskemaan virtuaalisen DOM-mallin ja todellisen DOM-mallin erot tehokkaasti, ja päivittää vain tarvittavat muutokset todelliseen malliin. [2.]

Muutaman viime vuoden aikana React.js on noussut hyvin suureen suosioon front-end -teknologioiden joukossa. On vaikea määrittää tarkasti, mikä teknologioista on suosituin, mutta suuntaa-antavia tuloksia saadaan vertailemalla NPM-pakettien latauskertoja sekä Google-hakuja.



Kuva 2. NPM-pakettien latausmäärät viimeiseltä kahdelta vuodelta. [7]

Latausmäärien perusteella React.js on tällä hetkellä ehdottomasti suosituin front-end -teknologia. NPM-pakettien latausmäärä ei kuitenkaan kerro sitä, onko teknologia käytössä jossakin oikeassa projektissa vai onko se ladattu vain testausta varten.

Kuva 3. Googlen hakutermien määrä viimeiseltä viideltä vuodelta. [8]

Myös Googlen hakutermien määrästä nähdään, että React.js on saavuttanut suurta mielenkiintoa parin viime vuoden aikana. Google trends ei myöskään ole täysin luotettava lähde, koska erilaisilla hakutermeillä voi saada toisenlaisia tuloksia. Kuvat 2 ja 3 kuitenkin viittaavat siihen tulokseen, että React.js on relevantti teknologia tällä hetkellä.

2.3 Expess.js

Node.js on vain alusta, jossa voidaan suorittaa JavaScriptiä. Se ei itsessään toimi palvelimena. Express.js on kehys, joka yksinkertaistaa palvelimen koodin kirjoittamista. Expressin omilla kotisivuilla kehystä kuvaillaan seuraavasti:

> ”Express on minimaalinen ja joustava Node.js verkkosovelluskehys, joka tarjoaa vakaat toiminnot verkko- ja mobiilisovelluksille."[9.]

Expressin avulla voidaan määrittää reittejä, jotka kertovat verkkosovellukselle, miten vastataan tiettyyn asiakkaan lähettämään HTTP-pyyntöön. Express.js tarjoaa CRUD-operaatiot, joiden kautta voidaan määrittää toiminnallisuuksia. Seuraavassa esimerkkikoodissa näytetään, miten määritellään yksinkertainen reitti.

```
app.get('/', function (req, res) {
      res.send('Hello World!')
})
```

Esimerkkikoodi 1. Yksinkertaisen reitin määrittäminen Express.js kehyksen avulla.

Esimerkkikoodissa 1 verkkosovellus vastaa asiakkaalle ”Hello World!”, kun asiakas lähettää GET-pyynnön kotisivulle. Expressin avulla voidaan myös jäsentää HTTP-pyyntöjen parametrit.

2.4 MongoDB

MongoDB on avoimen lähdekoodin NoSQL-tietokanta, joka eroaa relaatiotietokannasta siten, että tietojenkäsittely perustuu objekteihin. MongoDB-tietokannassa tiedot tallennetaan JSON-pohjaisiin dokumentteihin. JSON-muotoinen data helpottaa sen käsittelyä sovelluksen koodissa, eikä kehittäjän myöskään tarvitse käyttää ylimääräistä ORM-kerrosta kuten relaatiotietokannoissa tarvitsisi. Perinteisessä relaatiotietokannassa on taulukot ja rivit, mitkä määräävät datan rakenteen ja tekevät datasta konsistentin. Koska MongoDB-tietokannan data on objektimuotoinen, voi data olla epäkonsistenttia. Datan rakennetta voidaan myös muokata helposti. [10.]

3 Lisäosat

3.1 Passport.js

Passport.js on väliohjelmisto Node.js.sovelluksille, jonka avulla voidaan todentaa käyttäjät. Kirjautuminen on tärkeä osa verkkosovellusta. Passport.js-ohjelmiston avulla voidaan selkeyttää ja helpottaa kirjautumisen todentamista. Passport.js tukee useita erilaisia strategioita tunnistautumiseen. Sen avulla voidaan käyttää muun muassa Googlen ja Facebookin tarjoamia tunnistautumisstrategioita. [11.]

Tässä projektissa käytän passport-local-strategiaa, jonka avulla sovellukseen tunnistaudutaan käyttäjätunnuksen sekä salasanan avulla. Käyttäjätunnus sekä salasana on projektissani tallennettu MongoDB-tietokantaan.

3.2 Redux

React.js-kirjastossa käytetään datan tallennukseen niin sanottuja tiloja, jotka ovat tietoa sisältäviä objekteja. Tiedonkulku on yksisuuntainen React.js-sovelluksissa: pääkomponentti jakaa tilan sen alikomponenteille, jotka jakavat sen edelleen omille alikomponenteilleen. Yksisuuntainen tiedonkulku auttaa pitämään komponenttien toiminnan simppelinä. Se kuitenkin muodostuu ongelmaksi isommissa sovelluksissa, joissa data täytyy kuljettaa monen alikomponentin kautta. Ongelman ratkaisee Redux-kirjasto, joka toimii säilönä halutuille sovelluksen tiloille. Säilöön liitetyt komponentit voivat hakea tiloja säilöstä itsenäisesti, eikä tilaa tarvitse kuljettaa usean eri komponentin kautta. [12.]

Oletusarvoisesti Redux lähettää toimenpiteet synkronoidusti, mikä voi aiheuttaa ongelmia, jos Rexud joutuu esimerkiksi odottamaan tietokannasta saatavaa dataa. Jotta toimenpiteet saadaan lähetettyä asynkronoidusti, tarvitsee Reduxin kanssa käyttää väliohjelmistoa. Tässä projektissa käytämme Redux Thunk -kirjastoa, jonka avulla voidaan lisätä asynkroninen logiikka. [13.]

3.3 Kaaviokirjastot

Tämän insinöörityön verkkosovellus tulee sisältämään paljon kaavioita. React.js-sovelluksia varten on olemassa paljon erilaisia kaaviokirjastoja ja niistä on vaikea valita paras omaa sovellusta varten. Seuraavaksi esittelen kolme eri kaaviokirjastoa ja pyrin käymään läpi niiden käyttöönotettavuuden, muokattavuuden sekä suorituskyvyn.

3.3.1 Victory

Victory on Formidablen kehittämä modulaarinen kaaviokirjasto React.js-sovelluksille. Se on täysin päällekirjoitettava, eli muokattava, ja soveltuu vuorovaikutteiseen käyttöön. Victory-kirjasto tarjoaa yleisimmin käytettyjä diagrammeja kuten viiva-, pylväs- ja ympyrädiagrammeja. Niiden lisäksi kirjastosta löytyy muita erilaisia kaavioita, eli kaavioiden tarjonta on hyvin kattava. Victoryn kaaviot tukevat suurentamista ja panorointia (kaavion liikuttamista hiirellä). Kirjasto tarvitsee toimiakseen vain React.js-projektin, joten mitään

ylimääräisiä väliohjelmia ei tarvitse asentaa. Victoryn kotisivuilta löytyy erittäin hyvät dokumentoinnit sekä useita esimerkkejä, mikä helpottaa kirjaston käyttöönotettavuutta. [14.]

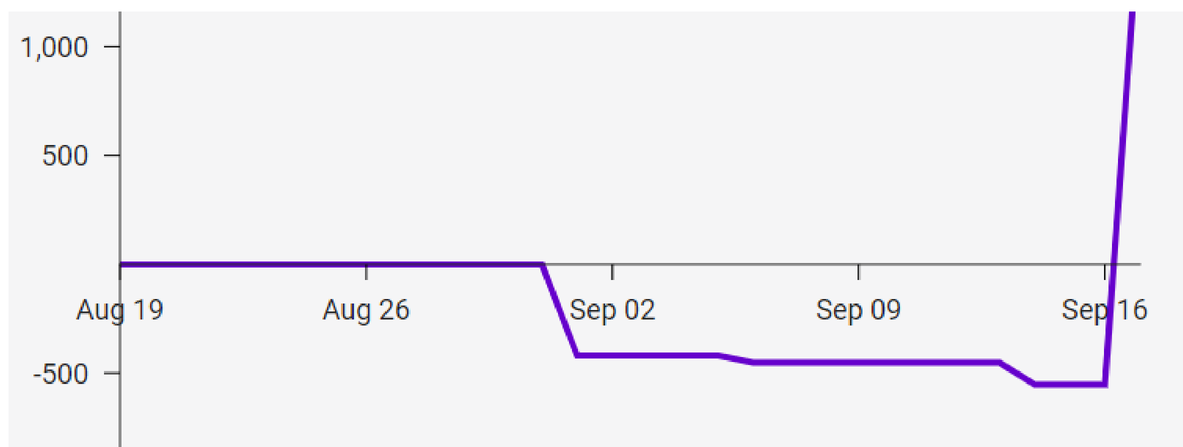
Värien vaihtaminen onnistuu helposti Victoryn kaavioissa. Esimerkiksi ympyrädiagrammille voidaan antaa värejä sisältävä taulukko, josta diagrammi käyttää värejä järjestyksessä:

```
const customColorScale = [
      'rgba(255, 0, 0, 0.8)',
      'green',
      '#e69500',
]

<VictoryPie
      colorScale={customColorScale}
/>
```

Esimerkkikoodi 2. Väritaulukon syöttäminen Victoryn ympyrädiagrammille

Diagrammi käyttää satunnaista väriä, mikäli taulukon värit loppuvat kesken. Värit voidaan antaa ainakin nimenä sekä HEX- tai RGBA-värikoodeina.

Victoryn kaaviot myös tukevat päivämääriä koordinaatiston akseleissa. Päivämäärät annetaan kaavioille JavaScript Date -objekteina, ja kaaviot määritetään käyttämään aikaa asteikkona.



Kuva 4. Päivämäärien käyttäminen Victoryn viivadiagrammissa.

Viivadiagrammiin oli helppo lisätä Victory-kirjaston avulla tooltip, eli hiiren osoittimessa näkyvä tekstikenttä, joka näyttää osoittimen kohdalla olevan päivämäärän sekä arvon. Tooltip-ominaisuutta on mahdollista muokata erittäin paljon Victory-kirjaston avulla. Ominaisuuden liittämistä tähän projektiin helpotti kattava dokumentaatio.

Kaavioiden suorituskyky on hyvä Victory-kirjaston avulla. Suoritin testin, jossa käyttöliittymässä on samaan aikaan 20 ympyrädiagrammia ja verkkosivu latautui nopeasti, eikä diagrammien renderöinnissä ilmennyt ongelmia. En näe hyödylliseksi suorittaa testiä suuremmassa mittakaavassa, koska realistisen verkkosovelluksen tuskin tarvitsee renderöidä edes 20 diagrammia samaan aikaan. Victory-kirjasto tarjoaa myös kaavioihin animaatioita. Animaatiot ovat tasaisia ja näyttävät hyviltä, kun animoitavia diagrammeja on samaan aikaan muutama. Jos animoitavia kaavioita on samaan aikaan 10, niin animaatiot alkavat pätkimään. Testin mukaan Victory-kirjaston animaatioiden käyttö on järkevää, kun animoitavia kaavioita on alle 10.

3.3.2 Chartist.js

Chartist.js on avoimen lähdekoodin kirjasto, jonka tarkoituksena on tarjota yksinkertaisia, kevyitä sekä responsiivisia kaavioita. Chartist.js-kirjaston kaaviot piirretään verkkosivuun SVG:tä hyödyntäen. Chartist.js-kirjastoa ei ole tehty pelkästään React.js-sovelluksia varten, vaan sitä voi myös käyttää muun muassa Angular.js-alustan kanssa. Chartist.js-kirjaston lisäksi täytyy asentaa väliohjelma riippuen alustasta. React.js-sovelluksia varten täytyy asentaa react-chartist-väliohjelma. Chartist.js-kirjasto tarjoaa yleisimmin käytettyjä diagrammeja kuten Victory-kirjasto. Yleisimpien diagrammien lisäksi Chartist.js ei kuitenkaan tarjoa muita kaavioita, eli kaavioiden tarjonta jää vähäiseksi. Chartist.js-kirjaston kotisivuilta löytyy kohtalaisen hyvät dokumentoinnit natiiviin kirjastoon. Sivuilla on paljon erilaisia esimerkkejä kaavioista mutta joistakin parameteristä sekä funktioista on huonot dokumentoinnit. React-chartist-välihojelman dokumentointi on myös vähäinen, mikä vaikeuttaa Chartist.js-kirjaston käyttöönotettavuutta React.js-sovelluksiin. [23.]

Kaavioiden muokkaaminen tapahtuu muokkaamalla Chartist.js-kirjaston mukana tullutta CSS-tiedostoa. Sen avulla onnistuu muun muassa värien, viivojen tyylien sekä paksuuden muokkaaminen. Chartist.js tarjoaa myös vaihtoehdon määrittelemiseen, miten kaavio muokkautuu responsiivisesti.

Metropolia

```
var data = {
  labels: ['Jan', 'Feb', 'Mar', 'Apr', 'Mai', 'Jun', 'Jul', 'Aug',
'Sep', 'Oct', 'Nov', 'Dec'],
    series: [
    [5, 4, 3, 7, 5, 10, 3, 4, 8, 10, 6, 8],
    [3, 2, 9, 5, 4, 6, 4, 6, 7, 8, 7, 4]
  ]
};

var options = {
  seriesBarDistance: 15
};

var responsiveOptions = [
  ['screen and (min-width: 641px) and (max-width: 1024px)', {
    seriesBarDistance: 10,
    axisX: {
      labelInterpolationFnc: function (value) {
        return value;
      }
    }
  }],
  ['screen and (max-width: 640px)', {
    seriesBarDistance: 5,
    axisX: {
      labelInterpolationFnc: function (value) {
        return value[0];
      }
    }
  }]
];

new Chartist.Bar('.ct-chart', data, options, responsiveOptions);
```

Esimerkkikoodi 3. Responsiivisien asetuksien määritteleminen Chartist.js-kirjaston kaavioihin. Esimerkissä on määriteltynä pylväsdiagrammi, jossa on kuukauden lyhenteet x-akselin arvona. Responsiivisiin asetuksiin määritellään, miten x-akselin arvot skaalautuvat näytön leveyden mukaan. Näytön leveyden pienetessä vähennetään ensiksi x-akselin arvojen riviväliä. Jos näytön leveys tippuu alle 640 pikseliin, niin x-akselin arvoina näytetään vain kuukauden ensimmäinen kirjain. [24.]

Chartist.js-kirjasto tarjoaa hyvät mahdollisuudet muokata kaavioita mutta kaavioiden asetuksien määritteleminen tuntuu kömpelöltä. Oma kokemukseni kyseisestä kirjastosta on, että se on helppo ottaa käyttöön mutta kaavioiden muokkaaminen on hankalaa. Muokkaamista vaikeutti erityisesti react-chartist-väliohjelman huono dokumentaatio.

Suorituskyky on hyvä Charist.js-kirjaston kaavioissa. Suoritin saman testin, minkä tein myös Victory-kirjastolle. Verkkosivu latautuu nopeasti ja ilman ongelmia, vaikka käyttö-

liittymässä on 20 diagrammia samaan aikaan. Myös Chartist.js-kirjasto tarjoaa animaatioita sen kaavioihin. Chartist.js-kirjaston animaatiot ovat suorituskyvyltään parempia verrattuna Victory-kirjaston animaatioihin: 20 samanaikaisesti animoitavan diagrammin kohdalla Chartis.js-kirjaston animaatiot eivät näyttäneet pätkivän ollenkaan.

3.3.3 React-Vis

React-Vis on Uberin kehittämä kaaviokirjasto, joka on kehitetty toimimaan suoraan React.js-sovelluksissa ilman ylimääräisiä väliohjelmia. React-Vis-komponentit on suunniteltu toimimaan kuten tavalliset React.js-komponentit: niillä on ominaisuuksia, lapsikomponentteja sekä takaisinkutsufunktioita. React-Vis väittääkin kotisivuillaan, että jos kehittäjä tuntee normaalien React.js-komponenttien toimintaperiaatteet, niin osaa hän myös React-Vis-komponenttien toimintaperiaatteet. [25.] Kuten Victory-kirjasto, tarjoaa React-Vis-kirjasto yleisimpien diagrammien lisäksi muita kaavioita, eli kaavioiden tarjonta on kattava.

Kaavioiden muokkaamiseen on muutama eri tapa React-Vis-kirjastossa. Yksi tapa on muokata kirjaston mukana tulevaa CSS-tiedostoa. React-Vis-kaaviokomponentit myös hyväksyvät className-ominaisuuden, jonka avulla kaavioita voidaan muokata itsetehdyn CSS-tiedoston avulla. Viimeinen tapa muokata kaavioita on komponenttien style-ominaisuus, joka hyväksyy CSS-ominaisuuksia objektin muodossa. Style-ominaisuus toimii siis samalla periaatteella kuin perus-DOM-elementtien tyylittäminen React.js-komponenteissa.

Kuten Victory-kaaviokirjasto myös React-Vis -kaaviokirjasto tukee vuorovaikutteisia kaavioita. React-Vis tukee muun muassa kaavioiden suurentamista sekä tooltip-ominaisuutta.

Kaavioiden suorituskyky ei poikkea Victory- ja Chartist.js-kirjastoista. React-Vis-kirjaston kaaviot renderöityvät nopeasti ja ongelmitta vaikka käyttöliittymässä on 20 ympyrädiagrammia samaan aikaan. Kuten Victory- ja Chartist.js-kirjastot myös React-Vis-kirjasto tarjoaa animaatioita sen kaavioihin. Suorituskyvyltään animaatiot ovat samaa luokkaa Chartist.js-kirjaston kanssa, eli React-Vis pystyy tuottamaan 20 samanaikaiseen diagrammiin animaatiot ilman huomattavaa ongelmaa.

3.3.4 Päätelmät kaaviokirjastoista

Kaikki kolme vertailussa ollutta kaaviokirjastoa tarjoavat kattavat kaaviot, kuten viiva-, ympyrä- ja pylväsdiagrammit. Victory- ja React-Vis -kirjastot myös tarjosivat näiden kaavioiden lisäksi muita harvinaisempia vaihtoehtoja. Kaaviot olivat hyvin muokattavissa mutta Chartist.js-kirjasto hävisi tässä kategoriassa Victory- ja React-Vis -kirjastoille. Chartist.js ei myöskään tarjonnut yhtä hyviä mahdollisuuksia vuorovaikutteisten kaavioiden tekemiseen. Chartist.js-kirjasto toisaalta tarjosi hyvät mahdollisuudet responsiivisiin kaavioihin. Kaavioiden luonti ja muokkaaminen tuntui miellyttävimmältä Victory-kirjaston avulla.

Kirjastojen suorituskyvyissä ei ole suurta eroa paitsi Victory-kirjaston animaatioiden kohdalla. Victory-kirjasto ei kykene tuottamaan animaatioita ongelmitta yhtä moneen samanaikaisesti renderöitävään kaavioon verrattuna Chartist.js- ja React-Vis-kirjastoihin. Tämän ei pitäisi kuitenkaan olla ongelma useimmissa verkkosovelluksissa.

Verrattujen kirjastojen dokumentaatioissa oli suuria eroja. Victory-kaaviokirjaston dokumentaatiot olivat erittäin kattavat, mikä erityisesti helpotti kirjaston käyttöönottoa ja sen avulla työskentelyä. Vertailun tulosten perusteella tulen käyttämään Victory-kaaviokirjastoa projektin verkkosovelluksessa.

4 Verkkosovelluksen toteutus

Insinöörityön projektina on toteuttaa verkkosovellus MERN-pinon avulla. Projektista käydään läpi tärkeimmät suunnittelu- ja kehitysprosessit. Lopullisen sovelluksen olisi tarkoitus toimia eräänlaisena budjetointityökaluna, jonka avulla käyttäjä voisi syöttää tuloja, menoja ja budjetteja. Verkkosovellus näyttäisi syötetyistä tiedoista erilaisia kaavioita, jotka helpottaisivat rahankäytön tarkastelua. Sovelluksessa täytyy olla mahdollista rekisteröidä käyttäjä sekä todentaa käyttäjä sisäänkirjautumisen avulla.

4.1 Kehitysympäristö

Verkkosovelluksia voidaan kehittää useassa eri ympäristössä. Suosittuja käyttöjärjestelmiä ovat Linux, macOS ja Windows. Tämän insinöörityön projekti tullaan kehittämään Windows 10 -käyttöjärjestelmällä. Ohjelmointia varten olen valinnut ilmaisen Visual Studio Code -editorin, joka on tarkoitettu koodin editointia, debuggausta ja versionhallintaa varten. Visual Studio Code -editoriin on olemassa monia eri liitännäisiä, jotka helpottavat muun muassa koodin rivitystä ja automaattista täydennystä.

Node.js täytyy asentaa kehitysympäristöön, jotta sitä voidaan käyttää. Latasin Windows 64-bit -version projektiani varten verkkosivulta https://nodejs.org/en/download/. Asennuksen jälkeen Node.js-komentoja voidaan käyttää Visual Studio Coden omassa komentorivissä.

4.2 Sovelluksen rakenne

Jotta verkkosovelluksen rakenteen hallinnointi pysyy helppona, kannattaa erottaa sovelluksen palvelinpuoli ja käyttöliittymä toisistaan. Niistä tehdään omat kokonaisuudet, jotka ajetaan omilla palvelimillaan. Tämä helpottaa kehittämistä, koska palvelinpuolen kehittäjän ei tarvitse välittää käyttöliittymästä, ja voi ladata pelkästään palvelinpuolen kokonaisuuden. Koska kokonaisuudet ajetaan omilla palvelimillaan, voivat kehittäjät ajaa omaa kokonaisuuttaan paikallisesti.

Yksi suuri etu kyseisessä rakenteessa on myös ongelmien välttäminen selainten saman alkuperän käytännön (same-origin policy) kanssa. Saman alkuperän käytäntö on tärkeä turvallisuusmekanismi, joka estää resurssien lataamista erinimisestä verkkotunnuksesta kuin mistä kutsu lähetettiin alunperin. Tämä auttaa estämään vahingollisten resurssien lataamista. Saman alkuperän käytäntö voidaan ohittaa määrittelemällä CORS (Cross-Origin Resource Sharing), minkä avulla voidaan erikseen sallia halutut resurssit. [15.] Sen määrittäminen aiheuttaa toisaalta paljon lisätyötä. Pääsemme ongelmasta eroon toisella tavalla: front-end palvelimella tehdyt kutsut ohjataan Node.js -palvelimelle, jolloin kaikki resurssit toimivat samassa verkkotunnuksessa. Esimerkiksi kun käyttäjä lisää uuden tulonlähteen, niin front-end-palvelin lähettää kutsun Node.js-palvelimelle, joka puolestaan lähettää kutsun edelleen tietokantaan. Tietokanta palauttaa tiedot takaisin

Node.js-palvelimelle, josta tiedot palautetaan lopulta front-end-palvelimeen. Tällöin kaikki resurssit liikkuvat Node.js-palvelimen kautta, eikä ohjelma aiheuta ongelmia saman alkuperän käytännön kanssa.

Luon projektin pääkansion alle oman kansiot palvelimelle (back-end). Käyttöliittymän (front-end) kansio tullaan luomaan myöhemmin React.js-kirjaston yhteydessä. Projektissa tarvittavat kirjastot tullaan myös asentamaan kyseisiin alikansioihin eikä globaalisti. Tämä helpottaa pitämään projektin organisoituna.

4.3 Palvelinpuoli

Sovelluksen palvelin tulee vastaamaan kirjautumisesta, HTTP-pyyntöihin vastaamisesta sekä kommunikoinnista MongoDB-tietokannan kanssa. Node.js-ympäristö alustetaan komennolla ”npm init”, joka luo uuden package.json-nimisen tiedoston. Se sisältää erilaista projektiin liittyvää metadataa sekä tiedot projektiin kuuluvista NPM-lisäosista. Node.js-alustan lisäosat asennetaan komennolla ”npm install”, joka myös uusimmissa NPM-versioissa lisää automaattisesti tiedot package.json-tiedostoon. NPM-versiota 5.0.0 vanhemmissa versioissa lisäosien tiedot täytyi lisätä package.json -tiedostoon manuaalisesti tai käyttämällä komentoa ”npm install –save lisäosan_nimi”. Lisäosat asentuvat node_modules-kansioon. Package.json-tiedostoon listatut lisäosat asennetaan automaattisesti, kun joku asentaa projektin NPM:n avulla, eikä node_modules-kansion tiedostoja tarvitse esimerkiksi tallentaa Git-versionhallintaan.

```
backend > {} package.json > ...
{
  "name": "backend",
  "version": "1.0.0",
  "description": "",
  "main": "server.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "body-parser": "^1.19.0",
    "connect-mongo": "^3.0.0",
    "cors": "^2.8.5",
    "express": "^4.16.4",
    "express-session": "^1.16.2",
    "mongoose": "^5.5.8",
    "passport": "^0.4.0",
    "passport-local": "^1.0.0"
  }
}
```

Kuva 5. Kuvassa Node.js-palvelinpuolen package.json-tiedosto. Se sisältää projektiin liittyvää metadataa sekä listan projektissa käytettävistä NPM-lisäosista (dependencies).

Seuraavaksi määrittelen käytettävän palvelimen. Asennan Express.js-kehyksen ja HTTP-pyyntöjä varten tarvittavan body-parser lisäosan komennolla ”npm install express body-parser”. Palvelimen juurikansioon luon tiedoston server.js, johon määritän palvelimen asetukset seuraavan koodiesimerkin mukaisesti:

```
const express = require('express')
const app = express()
const bodyParser = require('body-parser')
const PORT = 4000

app.use(bodyParser.json())

app.listen(PORT, function() {
    console.log("Server is running on Port: " + PORT)
})
```

Esimerkkikoodi 4. Palvelimen määrittäminen server.js-tiedostossa.

Nyt verkkosovelluksen palvelin toimii Express.js-kehyksen avulla. Tällä hetkellä palvelin täytyy aina käynnistää manuaalisesti uudelleen, jos koodiin tehdään muutoksia. Nodemon lisäohjelman avulla palvelin voidaan käynnistää automaattisesti uudelleen muutoksien jälkeen. [16.]

4.4 Tietokanta

Projektissa käytettävää MongoDB-tietokantaa voidaan käyttää palveluntarjoajan kautta verkossa tai se voidaan asentaa paikallisesti. Tässä projektissa tulen käyttämään paikallisesti asennettua tietokantaa. Ohjeet MongoDB-tietokannan asentamisesta Windowsille löytyy MongoDB:n virallisilta verkkosivuilta. [17.] Asentamisen jälkeen luon yhden tietokannan verkkosovellusta varten.

Tietokannan luomisen jälkeen täytyy server.js-tiedostoon määrittää toiminnallisuus palvelimen ja tietokannan välille. Toiminnallisuus voidaan toteuttaa mongoose-lisäosan avulla. Se on objektien mallintamiseen tarkoitettu työkalu, jonka avulla voidaan keskustella MongoDB-tietokannan kanssa. Mongoose voidaan asentaa komennolla ”npm install mongoose”. Projektissa tietokantaan tullaan tallentamaan käyttäjätiedot sekä käyttäjän merkitsemät tulot, menot ja budjetit. Ensiksi luon uuden alikansion malleille, jonka jälkeen määrittelen mongoosen avulla mallit jokaiselle tallennettavalle datalle. Käyttäjän malliin tullaan liittämään muut mallit, kuten tulot ja menot.

```
const mongoose = require("mongoose")
const Schema = mongoose.Schema
const Income = require("./income.model.js")
const Expense = require("./expense.model.js")

let User = new Schema({
    user_name: {
        type: String,
        required: true
    },
    user_email: {
        type: String,
        required: true
    },
    user_password: {
        type: String,
        required: true
    },
    user_createdAt: {
```

```
        type: Date,
        default: Date.now
    },
    user_incomes: [Income.schema],
    user_expenses: [Expense.schema]
})

module.exports = mongoose.model("User", User)
```

Esimerkkikoodi 5. Käyttäjää varten luotu malli mongoosen avulla. Objektille määritetään arvot JSON-muodossa. Malliin on myös liitetty tulojen sekä menojen omat mallit.

Mongoose tarjoaa tavanomaiset CRUD-operaatiot, joita voidaan kutsua sovelluksessa mallin kautta. Tässä projektissa mallin funktioita tullaan käyttämään Express.js-kehyksen reiteissä. Seuraavassa esimerkkikoodissa näytän, miten uusi käyttäjä tallennetaan tietokantaan, kun Express.js:n avulla määritetty reitti vastaanottaa HTTP-pyyntönä lähetetyn JSON-objektin uudesta käyttäjästä.

```
router.route('/add').post(function(req, res) {
    let newUser = new User(req.body)
    User.findOne({ user_email: newUser.user_email }).then(user => {
        if (user) {
            res.json({
                user: 'Email is already registered',
                redirect: '/register'
            })
        } else {
            newUser
                .save()
                .then(newUser => {
                    res.status(200).json({
                        user: 'user added succesfully',
                        redirect: '/login'
                    })
                })
                .catch(err => {
                    res.status(400).send('adding new user failed')
                })
        }
    })
})
```

Esimerkkikoodi 6. Uuden käyttäjän tallentaminen tietokantaan mongoose-lisäosan avulla.

Mongoosen avulla tarkistetaan ensiksi, onko uuden käyttäjän syöttämä sähköposti jo käytössä findOne-funktion avulla. Jos sähköposti on käytössä, niin käyttäjä ohjataan uudestaan rekisteröitymissivulle. Muussa tapauksessa uusi käyttäjä tallennetaan tietokantaan save-funktiolla, jonka jälkeen käyttäjä ohjataan sisäänkirjautumissivulle.

Palvelimeen täytyy myös määrittää yhteys tietokannan välille. Yhteys hoidetaan mongoosen avulla. Se onnistuu lisäämällä seuraava esimerkkikoodi server.js-tiedostoon.

```
const mongoose = require('mongoose')

mongoose.connect('mongodb://127.0.0.1:27017/budget', { useNewUrlParser: true
})
const connection = mongoose.connection
```

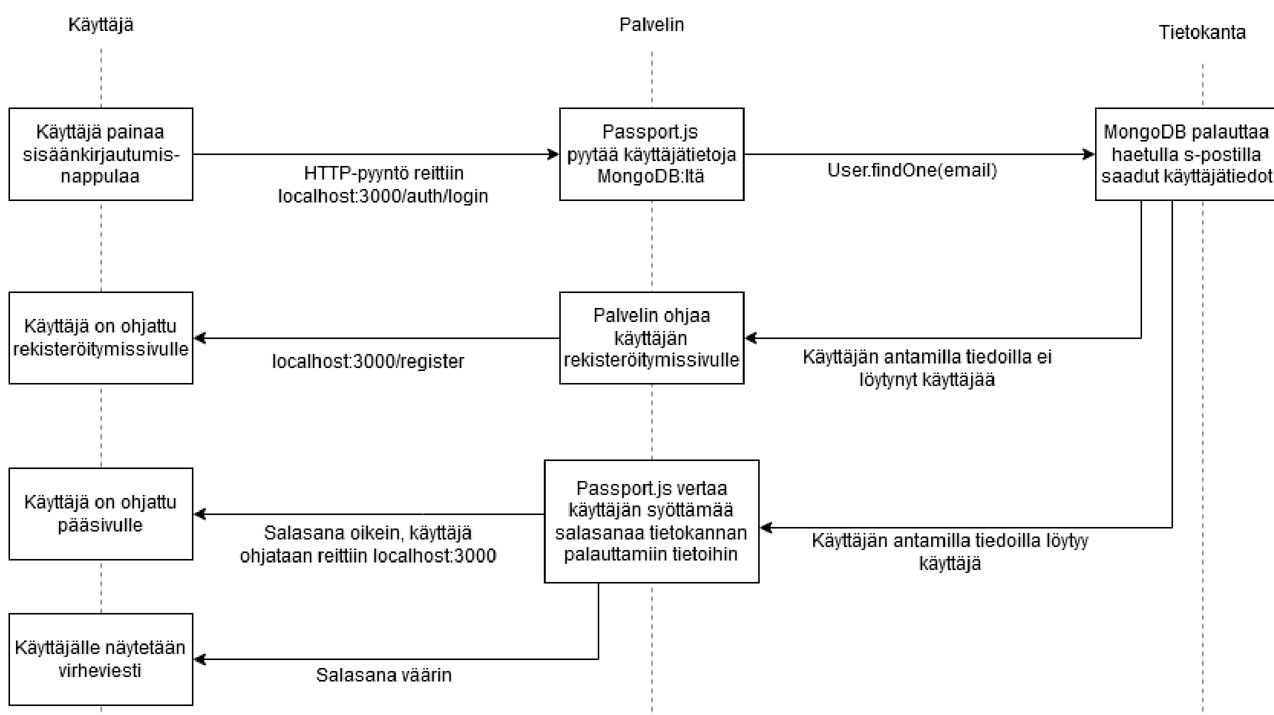
Esimerkkikoodi 7. Palvelimen ja MongoDB-tietokannan välinen yhteys määritettynä mongoosen avulla.

Mongoose.connect-funktioon annetaan parametrina tietokannan osoite. Tässä projektissa tietokanta ajetaan paikallisesti, joten osoitteena on annettu localhost sekä tietokannan nimi.

4.5 Kirjautuminen

Käyttäjän sisäänkirjautumista varten käytän projektissa Passport.js-ohjelmistoa passport-local-strategian avulla. Jotta käyttäjän ei tarvitsisi kirjautua sisään aina uudelleen, lisään verkkosovellukseen ominaisuuden evästeiden tallentamista varten. Tätä varten täytyy asentaa express-session ja connect-mongo-lisäosat.

Passport-local-strategiaa varten luon tiedoston nimeltä ”passport.js”, johon määrittelen Passport.js-ohjelman konfiguraation. Tiedostoon täytyy määrittää attribuutti, jonka perusteella käyttäjä etsitään tietokannasta. Tässä projektissa on järkevintä etsiä tietokantaan tallennettua käyttäjää sähköpostin avulla, koska sähköpostiosoite on mahdollista liittää vain yhteen käyttäjään. Passport.js-tiedostoon määritetään myös takaisinkutsufunktiot riippuen siitä, löytyikö haettu käyttäjä ja oliko salasana oikein vai väärin. Konfiguraation lisäksi minun täytyy määritellä Express.js:n avulla reitti, jossa sisäänkirjautumis HTTP-pyynnöt käsitellään. Luon ”auth.js”-nimisen tiedoston, johon määrittelen käytettäväksi passport.js-tiedostossa olevaa strategiaa, kun Express.js vastaanottaa käyttäjältä kutsun sisäänkirjautumiseen. [18.]



Kuva 6. Kulkukaavio sisäänkirjautumisesta sekä Passport.js-ohjelman toiminnallisuus riippuen tietokannan palauttamista käyttäjätiedoista.

Kuvan 6 mukaisesti sisäänkirjautuminen toimii seuraavalla tavalla: ensiksi käyttäjä syöttää tiedot ja painaa sisäänkirjautumisnappulaa. Front-end lähettää HTTP-kutsuna käyttäjän syöttämät tiedot, jotka Express.js vastaanottaa. Määritellyn passport-local-strategian mukaan Passport.js lähettää kutsun MongoDB-tietokannalle, joka etsii tallennettua käyttäjää annetun sähköpostin perusteella. MongoDB palauttaa löydetyt tiedot Passport.js-ohjelmalle, joka vertaa tietokannasta saatuja tietoja käyttäjän antamiin tietoihin. Vertailun mukaan Express.js ohjaa käyttäjän oikealle verkkosivulle tai ilmoittaa virheellisestä salasanasta.

Evästeet tallennetaan kirjautumisen yhteydessä express-session- ja connect-mongo-lisäosien avulla. Käyttäjän evästeen tiedot tallennetaan MongoDB-tietokantaan. Evästeeseen voidaan määrittää aika, milloin eväste vanhentuu. Mikäli käyttäjän verkkoselaimessa on voimassa oleva eväste, ei hänen tarvitse syöttää kirjautumistietoja uudelleen, vaan verkkosovellus kirjautuu automaattisesti sisään.

4.6 Käyttöliittymä

Projektin verkkosovelluksen käyttöliittymä tullaan toteuttamaan React.js-kirjaston sekä muutaman lisäosan avulla. Tavoitteena on tehdä yksinkertainen ja selkeä käyttöliittymä, jonka kautta käyttäjä voi suorittaa palvelimen toimintoja. Aloitan ensiksi suorittamalla komennon ”npx create-react-app frontend”, minkä avulla luodaan valmiiksi konfiguroitu paketti ”frontend”-kansioon. Nyt verkkosovelluksessa on omat kansiot palvelimelle sekä käyttöliittymälle. Create-react-app-komennon avulla luotu paketti on määritelty käynnistämään palvelin osoitteessa http://localhost:3000. React.js palvelimen voi käynnistää komennolla ”npm start”. Osoitteen voi avata omaan verkkoselaimeen, josta käyttöliittymään tehtyjä muutoksia voi tarkastella reaaliajassa. Luodussa paketissa tuli mukana index.html-tiedosto, joka toimii React.js-sovelluksen juurena. Tiedosto sisältää yleiset html-tiedoston määritykset sekä div-elementin, jolle on valmiiksi määritelty id-attribuutti arvolla ”root”. React.js-komponentit tullaan renderöimään kyseisen div-elementin sisälle. [19.]

Asennan vielä verkkosovellukseen Redux- ja Redux-Thunk -kirjastot, mikä helpottaa tietojen kuljettamista eri React.js-komponenttien välillä. Redux-kirjasto vaatii myös ylimääräisen kirjaston React-Redux toimiakseen React.js-sovelluksessa. Kirjastot täytyy lisätä sovelluksen ylätasoon, jolloin ne ovat käytössä koko sovelluksessa. Ensiksi luon ”store.js”-nimisen tiedoston, mihin määrittelen Redux- ja Redux-Thunk-kirjastojen avulla käytettävän säilön.

```
import { createStore, applyMiddleware } from 'redux'
import thunk from 'redux-thunk'
import rootReducer from './reducers/index'

const store = createStore(rootReducer, applyMiddleware(thunk))

export default store
```

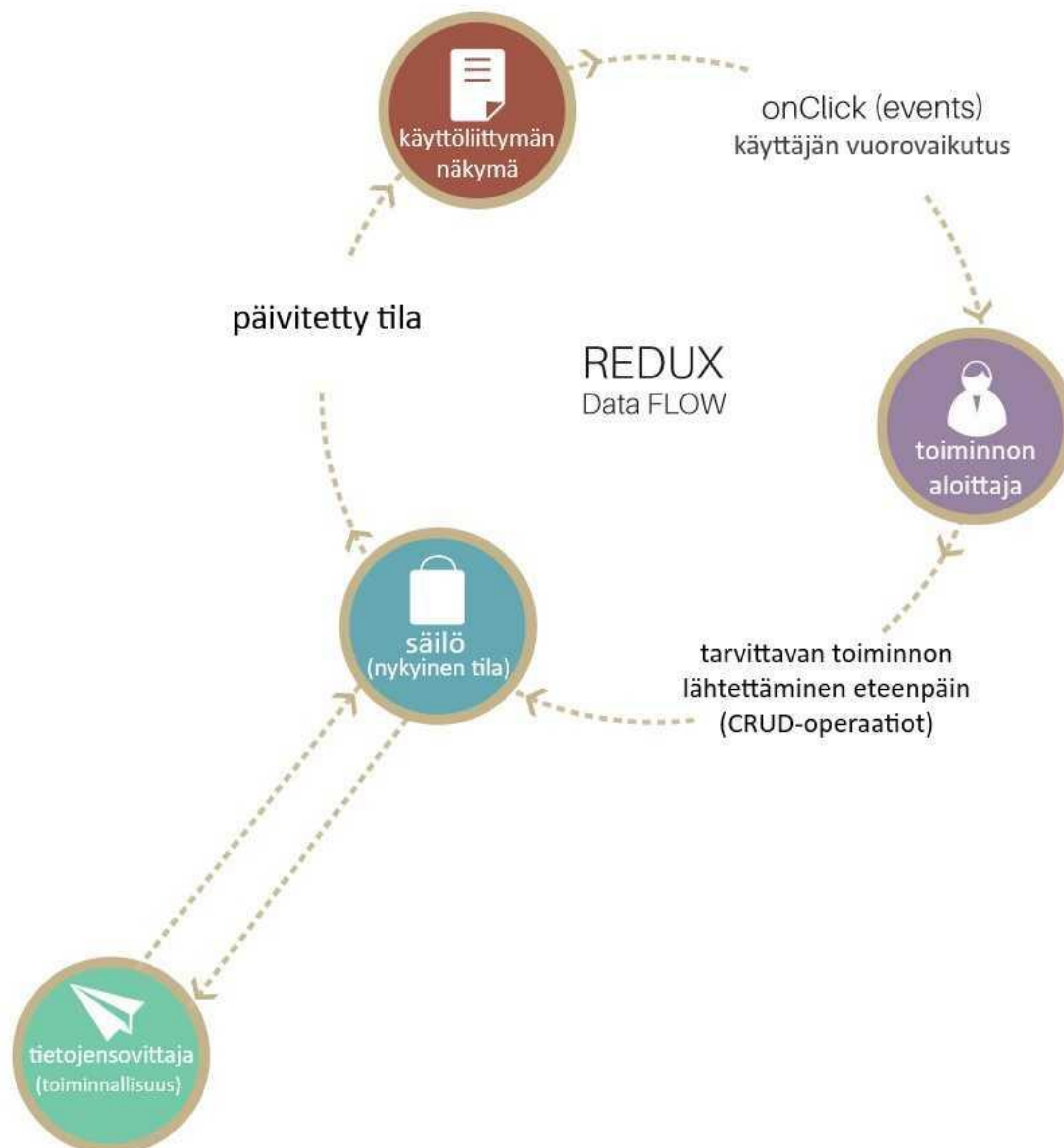
Esimerkkikoodi 8. Redux- ja React-Thunk-kirjastojen avulla määritellään säilö React.js-sovellukselle. Redux-kirjasto toimii säilönä ja siihen liitetty väliohjelmisto Redux-Thunk lisää säilöön erilaisia toimintoja, kuten asynkronisia tietojen päivityksiä. CreateStore()-funktiolla luodaan uusi säilö nimeltä store, johon komponenttien tiedot voidaan säilöä. Tämä säilö saa parametrikseen tietojensovittajat (reducers) sekä väliohjelmiston Redux-Thunk. [20.]

Esimerkkikoodi 8 mukaisen säilön määrittelemisen jälkeen täytyy säilö vielä liittää React.js-sovellukseen. Create-react-app-komennon avulla luotu paketti sisältää ”App.js”-nimisen tiedoston, mihin Redux-kirjaston säilö liitetään.

```
import React from 'react'
import ReactDOM from 'react-dom'

import { Provider } from 'react-redux'
import store from './store'

import App from './App'

const rootElement = document.getElementById('root')
ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  rootElement
)
```

Esimerkkikoodi 9. React-Redux-kirjaston liittäminen React.js-sovellukseen. Esimerkkikoodissa geneerisesti nimetty ”App”-komponentti renderöidään react-dom-paketin avulla index.html-tiedoston div-elementtiin, jossa id-attribuutti on root. React-Redux-kirjaston tarjoama säilö määritellään ReactDOM.render()-funktion sisälle. Kaikki komponentit, jotka jäävät säilön HTML-merkkauksen sisäpuolelle voivat hyödyntää säilöä. Koska säilö on merkattuna ylimmälle tasolle, koko sovellus käyttää säilöä. [21.]

Normaalisti React.js-sovelluksessa tilojen tiedonkulku toimii hierarkkisesti, eli komponentit siirtävät tilanmuutoksen informaation lapsikomponenteille, jotka taas siirtävät informaation omille lapsikomponenteilleen tai isäntäkomponenteilleen. Tällainen kommunikaatio on selkeää yksinkertaisissa ja pienissä sovelluksissa. Mutta kookkaammissa ja monimutkaisemmissa sovelluksissa kyseinen hierarkkinen tiedonsiirto alkaa tehdä sovelluksen rakenteesta hankalampaa, ja koodista tulee vaikeammin seurattavaa. Tämän projektin verkkosovellukseen on nyt Redux-kirjaston avulla määritelty säilö, joka yhdistää kaikki komponentit. React.js-sovelluksen komponenttien tilojen tiedonkulku on nyt seuraavan kuvan mukainen:



Kuva 7. Redux-kirjaston avulla määritellyn säilön tiedonkulku React.js-sovelluksessa. [22]

Kuten kuvasta 7 havaitaan, kun käyttäjä suorittaa käyttöliittymässä toimenpiteen, minkä seurauksena sovelluksessa muuttuu jokin tila, kutsutaan seuraavana säilön kautta tietojensovittajaa. Sovittaja vastaanottaa tilan ja käynnistää toimenpiteen tilan muuttamiseen. Kun tila on muuttunut, käyttöliittymässä renderöidään uudelleen osa, johon tila on liitettynä. Tässä projektissa kyseistä tekniikkaa käytetään muun muassa käyttäjän lisätessä uuden tulonlähteen sekä muissa samankaltaisissa CRUD-operaatioissa.

```
export const addIncome = (userId, newIncome) => dispatch => {
    axios
        .post('/user/' + userId + '/income/add', newIncome)
        .then(res =>
            dispatch({
                type: ADD_INCOME,
                payload: res.data
            })
        )
        .catch(err =>
            dispatch({
                type: GET_ERRORS,
                payload: err.response.data
            })
        )
}
```

Esimerkkikoodi 10. Tulonlähteen lisäämisen käynnistävä funktio addIncome. Axios-kirjaston avulla lähetetään lupaus pohjainen HTTP-pyyntö Express.js-palvelimelle. Pyyntö sisältää käyttäjätunnuksen sekä tiedot uudesta tulonlähteestä, joka tallennetaan MongoDB-tietokantaan. Vastauksen jälkeen kutsutaan dispatch-funktiota, mikä saa parametrikseen toiminnon sekä datan.

```
export default function(state = initialState, action) {
    switch (action.type) {
        case SET_CURRENT_USER:
            return {
                ...state,
                isAuthenticated: !isEmpty(action.payload),
                user: action.payload
            }
        case USER_LOADING:
            return {
                ...state,
                loading: true
            }
        case ADD_INCOME:
            return {
                ...state,
                user: {
                    ...state.user,
                    user_incomes: action.payload
                }
            }
        case ADD_EXPENSE:
            return {
                ...state,
                user: {
                    ...state.user,
                    user_expenses: action.payload
                }
            }
        default:
            return state
    }
```

Esimerkkikoodi 11. Tiedonsovittaja vastaanottaa dispatch-funktion avulla lähetetyn toiminnon sekä tilan. Toiminnon arvo, eli nimi, käydään läpi switch-rakenteessa. Esimerkiksi uutta tulonlähdettä lisätessä suoritetaan ADD_INCOME -tapauksen toiminnot. Ensiksi palautetaan nykyisen säilön tila, minkä jälkeen uusi

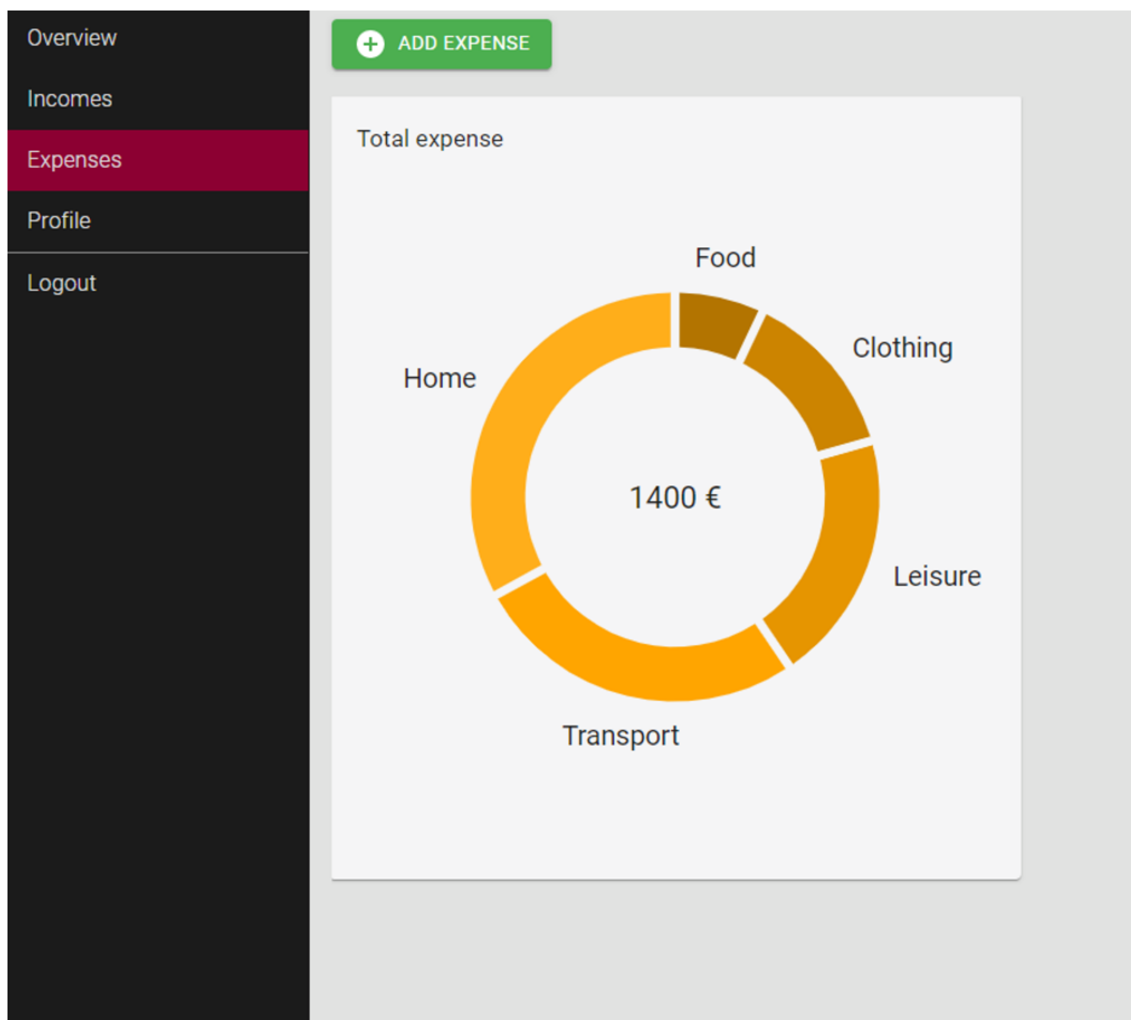
tulonlähde lisätään tilaan. Tiedonsovittaja ei koskaan muuta olemassa olevaa tilaa, vaan se palauttaa aina uuden tilan, johon on voitu kopioida vanhan tilan tiedot.

Yksi React.js-komponenttien ominaisuuksista on props-objekti, jonka avulla voidaan siirtää dataa komponenteissa. Tulonlähteistä vastaavaan React.js-komponenttiin määritellään mapStateToProps()-funktio osaksi säilön liittävää connect()-funktiota. Nyt mapStateToProps()-funktio ajetaan aina kun tilassa tapahtuu muutos. Kyseisessä funktiossa voidaan palauttaa tila, jota tarvitaan liitetyssä React.js-komponentissa. Palautettu tila liitetään props-objektiin, jolloin tarvittua dataa voidaan käsitellä komponentissa.

```
class IncomeMain extends Component {
    render() {
        const emptyPage = (
            <div className={classes.emptyDiv}>
                <h2 className={classes.emptyText}>
                    This page is currently empty. Try adding some incomes.
                </h2>
            </div>
        )

        return (
            <div>
                {this.props.hasIncome ? (
                    <GridListContainer>
                        <IncomePie />
                    </GridListContainer>
                ) : (
                    emptyPage
                )}
            </div>
        )
    }
}

const mapStateToProps = state => {
    if (state.auth.user.user_incomes) {
        if (state.auth.user.user_incomes.length > 0) {
            return { hasIncome: true }
        }
    } else {
        return { hasIncome: false }
    }
}

export default connect(mapStateToProps) (IncomeMain))
```

Esimerkkikoodi 12. Koodinpätkä tulonlähteistä vastaavasta React.js-komponentista. MapStateToProps()-funktion avulla lisätään säilössä olevat käyttäjän tulot props-objektiin. Mikäli käyttäjällä ei ole tulonlähteitä, komponentissa renderöidään emptypage-objektin sisältö. Muussa tapauksessa renderöidään tulonlähteet sisältävä kaavio.

Myös muutkin tämän projektin React.js-komponentit tullaan toteuttamaan esimerkkikoodien 8, 9 ja 10 näyttämällä tavalla. Vaikka Redux-kirjaston säilön konfigurointi vie aluksi aikaa, niin sen avulla on kuitenkin helppo liikuttaa sovelluksessa tarvittavaa dataa, kuten käyttäjän tulot ja menot eri komponenttien välillä.



Kuva 8. Verkkosovelluksen käyttöliittymä. Victory-kirjaston avulla luotuun diagrammiin on liitetty React.js-komponenttien tilat, jolloin myös diagrammit päivittyvät dynaamisesti.

5 Yhteenveto

Verkkosovelluksien kehittäminen on iso prosessi, joka voidaan toteuttaa monella eri tavalla. Tässä insinöörityössä käytiin läpi verkkosovelluksen kehittäminen JavaScript-pohjaisten teknologioiden avulla. MERN-pino on tällä hetkellä suosittu tapa kehittää verkkosovelluksia Node.js-ympäristössä. Pino on helppo ottaa käyttöön ja siihen voi liittää useita kirjastoja, jotka helpottavat verkkosovelluksen kehittämistä. MongoDB-tietokannan JSON-pohjainen tiedontallennus tuntui luontevalta käyttää, koska suurin osa tämän työn verkkosovelluksen datasta on JSON-muodossa. React.js-kirjaston avulla voidaan luoda dynaamisia käyttöliittymiä, joissa tiedonkulku on tehokasta eri komponenttien välillä.

Insinöörityön tavoitteisiin päästiin melko hyvin. MongoDB-tietokanta on yhdistettynä Node.js-palvelimeen. React.js-kirjaston avulla rakennettu käyttöliittymä on toimiva, ja käyttöliittymästä lähetetyt HTTP-pyynnöt vastaanotetaan onnistuneesti Express.js-kirjaston avulla. Käyttäjän syöttämät tiedot tallennetaan onnistuneesti tietokantaan. Verkkosovellus on pääosin toimiva, mutta käyttöliittymä ei ole täysin valmis

Tämän työn verkkosovelluksessa näytettiin dataa kaavioiden avulla. React.js-kirjastoon on olemassa erilaisia kirjastoja, joiden avulla voidaan rakentaa kaavioita. Vertailun kohteena oli kolme eri kaaviokirjastoa: Victory, Chartist.js ja React-Vis. Kaikki kolme kirjastoa tarjosi yleisimmin käytettyjä diagrammeja. Kirjastot oli helppo ottaa käyttöön mutta kaavioiden muokattavuudessa oli eroja. Victory-kirjaston avulla kaavioiden tekeminen ja muokkaaminen tuntui luontevimmalta. Vertailussa olleiden kaaviokirjastojen suorituskyky on normaalioloissa erittäin hyvä ja kaaviot renderöityvät ilman ongelmia. Kun kaavioihin lisätään animaatioita, niin Victory-kirjasto häviää suorituskyvyssä Chartist.js- ja React-Vis-kirjastoille. Responsiivisiin kaavioihin tarjoaa parhaat vaihtoehdot Chartist.js-kirjasto.

React.js-komponenttien tila voidaan liittää verkkosovelluksen kaavioihin. Tämän avulla kaaviot päivittyvät dynaamisesti. React.js-kirjasto soveltuu täten erinomaisesti verkkosovellusten tekemiseen, mikäli sovelluksessa täytyy esittää dataa erilaisten diagrammien avulla.

Lähteet

1 Krawczyk, Ross. 2019. 16 companies which apps were written using Node.js. Verkkoaineisto: https://softwarebrothers.co/blog/companies-that-use-node-js/. Luettu 13.6.2019.

2 Subramanian, Vasan. 2017. Pro MERN stack: Full Stack Web App Development with Mongo, Express, React, and Node. Bangalore, Karnataka, India: Apress.

3 Node.js – Introduction. Verkkoaineisto: https://www.tutorialspoint.com/nodejs/nodejs_introduction.htm. Luettu 19.6.2019.

4 Introduction to Node.js. Verkkoaineisto: https://nodejs.dev/introduction-to-nodejs. Luettu 19.6.2019.

5 Ragaey, Amr. 2017. Shortly, How Node.js works on a single thread? https://medium.com/@amragaey/shortly-how-node-js-works-on-a-single-thread-763fda99f012. Luettu 19.6.2019.

6 What is npm? Verkkoaineisto: https://nodejs.org/en/knowledge/getting-started/npm/what-is-npm/. Luettu 12.8.2019.

7 NPM trends. Verkkoaineisto: https://www.npmtrends.com. Luettu 24.6.2019.

8 Google trends. Verkkoaineisto: https://trends.google.fi/. Luettu 24.6.2019.

9 Express.js. Verkkoaineisto: https://expressjs.com/. Luettu 30.6.2019.

10 What is MongoDB? Verkkoaineisto: https://www.mongodb.com/what-is-mongodb. Luettu 30.6.2019.

11 Passport. Verkkoaineisto: http://www.passportjs.org/. Luettu 24.7.2019.

12 Redux. Verkkoaineisto: https://redux.js.org/. Luettu 12.8.2019.

13 Asynchronous Redux Actions Using Redux Thunk. Verkkoaineisto: https://alligator.io/redux/redux-thunk/. Luettu 14.8.2019.

14 Getting Started with Victory. Verkkoaineisto: https://formidable.com/open-source/victory/docs/. Luettu 14.8.2019.

15 Same-origin policy. Verkkoaineisto: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy. Luettu 7.10.2019.

16 Nodemon. Verkkoaineisto: https://nodemon.io/. Luettu 2.4.2020.

17 Install MongoDB Community Edition on Windows. Verkkoaineisto: https://docs.mongodb.com/manual/tutorial/install-mongodb-on-windows/. Luettu 2.4.2020.

18 Passport-local. Verkkoaineisto: http://www.passportjs.org/packages/passport-local/. Luettu 17.4.2020.

19 Create React App. Facebook. Verkkoaineisto: https://github.com/facebook/create-react-app/. Luettu 19.4.2020.

20 Redux-Thunk. Verkkoaineisto: https://github.com/reduxjs/redux-thunk. Luettu 19.4.2020.

21 Quick Start. Verkkoaineisto: https://react-redux.js.org/introduction/quick-start. Luettu 19.4.2020.

22 Albertorio, Anthony. 2018. Redux. Verkkoaineisto: https://teamtreehouse.com/community/see-this-question-before-you-start-get-a-high-level-overview-of-redux-so-that-you-know-why-this-is-worthwhile. Luettu 22.4.2020.

23 Chartist.js. Verkkoaineisto: https://gionkunz.github.io/chartist-js/. Luettu 28.4.2020.

24 Chartist – Getting started. Verkkoaineisto: https://gionkunz.github.io/chartist-js/getting-started.html. Luettu 29.4.2020.

25 Welcome to React-vis. Uber. Verkkoaineisto: https://uber.github.io/react-vis/documentation/welcome-to-react-vis. Luettu 28.4.2020.