Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Pekka Pekkonen

Tulo–meno-laskurin suunnittelu ja toteutus modernein JavaScript-tekniikoin

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Insinöörityö

2.6.2019

Tekijä Otsikko Sivumäärä Aika	Pekka Pekkonen Tulo–meno-laskurin suunnittelu ja toteutus modernein JavaScript-tekniikoin 56 sivua + 1 liite 2.6.2019
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tietotekniikka
Ammatillinen pääaine	Ohjelmistotekniikka
Ohjaaja	Lehtori Simo Silander

Tämän insinöörityön tarkoituksena on esitellä sovelluksen tuottaminen suunnittelusta toteutukseen moderneja JavaScript-tekniikoita käyttäen. Työssä toteutetaan tekijän henkilökohtaisten tulojen ja menojen kirjanpitoon tarkoitettu web-sovellus.

Aluksi esitellään Excel-tiedostot, jotka uusi sovellus korvaa. Samalla esitellään sovelluksen vaatimusmäärittelyn taustalla olevaa pohdintaa sekä rautalankamallit eli karkean tason graafinen suunnitelma sovelluksen sisällöstä ja rakenteesta. Rautalankamalleissa on kiinnitetty erityisesti huomiota käytettävyyteen.

Ohjelmointi toteutettiin kokonaan JavaScript-pohjaisilla tekniikoilla, jotka mahdollistavat nopean sovelluskehityksen. Sovellukseen tuotettiin graafinen käyttöliittymä ja se yhdistettiin tietokantaan, joten erilaisia tekniikoita on käytössä useita. Asiakaspuoli on toteutettu React-ohjelmistokehyksellä, palvelinpuoli Node.js-ympäristöllä ja tietokanta on MongoDB. Node.js-ympäristöön on kytketty lisäksi Express-ohjelmistokehys ja MongoDB:hen Mongoose-mallintaja. Työssä valotetaan tekniikoiden toimintaa sekä esitetään esimerkkejä niiden käytöstä sovelluksessa.

Insinöörityön tuloksena saatiin toimiva kirjanpitosovellus, jonka tekijä voi ottaa käyttöönsä. Työssä toteutettiin kaikki perusominaisuudet: kirjauksia voi tarkastella, lisätä, muokata ja poistaa. Rautalankamalleissa esitetyt käytettävyyteen liittyvät ratkaisut saatiin toteutettua suureksi osaksi sovellukseen. Sovellus tehtiin modulaariseksi, mikä helpottaa jatkokehitykseen jätettyjen sekä kokonaan uusien ominaisuuksien toteuttamista.

Avainsanat	sovelluksen käytettävyys, web-sovellus, JavaScript, React, Node.js, MongoDB

Author	Pekka Pekkonen
Title	Designing and Implementing an Income and Outcome Tracker with Modern JavaScript Technologies
Number of Pages	56 pages + 1 appendix
Date	2 June 2019
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructor	Simo Silander, Senior Lecturer

The purpose of this thesis is to present a software development case from planning to implementation using modern JavaScript technologies. The developed software is an income and outcome tracker web application for author's personal use.

The application replaces Excel files used previously for author's personal accounting purposes. First the Excel files are presented along with a discussion about software requirements for the application. Based on those requirements approximate drawings for user interface called wireframes are presented with special attention to usability.

The development was made with JavaScript based technologies that enable fast application production. There is a graphical user interface and a database in the software. Hence many different technologies are used. The client side is developed with React software framework, the server side is developed with Node.js environment and the database is MongoDB. Also, Express software framework and Mongoose modeler are used. The thesis discusses about the used technologies and gives examples of their usage in the context of the application.

The work resulted in a functioning income and outcome tracking application that can be taken into use. All the basic functionalities were implemented: entries can be viewed, added, modified and removed. Usability solutions introduced with the wireframes were implemented for the most part in the application. The software is constructed in a modular manner that aids future development of new functionalities.

Keywords	application's usability, web application, JavaScript, React, Node.js, MongoDB

Sisällys

Lyhenteet

CSS	Cascading Style Sheets. Merkkauskielien, kuten HTML:n, ulkoasun määrittelyyn tehty kieli.
HTML	HyperText Markup Language. Verkkosisältöjen rakenteen ja merkityksen merkkauskieli.
HTTP	HyperText Transfer Protocol. Tiedonsiirtoprotokolla, jonka avulla Internet-selaimet ja WWW-palvelimet kommunikoivat keskenään.
JSON	JavaScript Object Notation. JavaScriptin syntaksista johdettu, mutta kieliriippumaton tiedonmuotostandardi.
JSX	JavaScript XML. JavaScriptin laajennos, jonka avulla JavaScriptiin voidaan yhdistää HTML:n kaltaista rakenteen ja merkityksen kuvausta.
URL	Uniform Resource Locator. Merkkijono, joka kertoo Internetissä olevan resurssin sijainnin sekä protokollan, jonka kautta resurssi on saatavissa.

1 Johdanto

Työn tavoitteena on suunnitella ja toteuttaa sovellus henkilökohtaisten tulojen ja menojen seurantaan. Sovellus korvaa aikaisemmin samaan tarkoitukseen käytetyt Excel-tiedostot.

Aluksi esitellään aikaisempaa järjestelmää ja pohditaan sen heikkouksia, joita uusi järjestelmä voisi parantaa. Sen jälkeen suoritetaan vaatimusmäärittely uudelle järjestelmälle. Lisäksi esitellään vaatimusmäärittelyyn perustuva karkea graafinen suunnitelma eli niin sanottu rautalankamalli. Painopiste rautalankamallissa on käytettävyydessä.

Suunnitteluvaiheen jälkeen esitellään valittuja tekniikoita, joista tärkeimpiä ovat JavaScript-ohjelmointikieli, React-ohjelmistokehys, Node.js-ympäristö ja MongoDB-tietokanta. Tekniikoista valotetaan toimintaperiaatteita ja esitetään niiden käyttöä sovelluksessa. Lopuksi esitellään sovellus, joka työssä syntyi.

2 Nykyinen järjestelmä

Tarkastellaan nykyistä Excel-pohjaista järjestelmää käytännön tilanteen kautta. Kun järjestelmää käyttävä henkilö maksaa jotain, hän painaa summan mieleensä siihen saakka, kunnes kirjaa menon järjestelmään. Vaihtoehtoisesti hän voi käyttää jotain apuvälinettä kuten verkkopankin tapahtumalistausta tai puhelimen muistiota. Sopivan hetken tullen hän avaa tietokoneellansa Excel-tiedoston ja kirjaa menon.

Nykyistä järjestelmää olisi mahdollista käyttää myös toisella tavalla: käyttäjä voi kirjata menon ollessaan vielä liikkeessä, sillä Excel-sovellus on saatavilla mobiililaitteille. Todennäköisesti moni kuitenkin käyttää vastaavaa järjestelmää ensin kuvatulla tavalla.

Nykyisessä järjestelmässä menot kirjataan kuukauden mukaan: kullakin kuukaudella on oma väliotsikkonsa, jonka alle kyseisen kuukauden kirjaukset merkataan. Kunkin kirjauksen kohdalle merkataan lisäksi tarkka päivämäärä. Poikkeuksen tästä muodostavat säännölliset menot -otsakkeen alla olevat kirjaukset, joiden kohdalla en ole pitänyt päivämäärän merkkaamista oleellisena. Kyseiseen osioon sijoitetaan kuukausittain toistuvat menot, mikä helpottaa kiinteiden menojen seuraamista.

Menoista voi syöttää päivämäärän, summan, maksajan, artikkelin, tyypin ja vapaamuotoista tietoa (kuva 1). Lisäksi on kirjattu tieto toistuvuudesta. Se on kuukauden lisäksi käytännössä ainoa kohta, johon on välttämätöntä ottaa kantaa kirjausta syötettäessä. Tuloista on voinut tallettaa samat tiedot lukuun ottamatta artikkeli-kohtaa. Olen lisännyt taulukkoon sarakkeita sitä mukaan, kun tarvetta on ilmennyt. Kunkin sarakkeen tiedolla on oma arvonsa. Esimerkiksi vapaamuotoisen tiedon kenttä on syntynyt tarpeesta kirjata jokin harvoin toistuva tieto. Tähän liittyy yksi nykyisen järjestelmän hyvistä puolista: kirjanpidon rakennetta on ollut suhteellisen helppo muuttaa tilanteen mukaan, esimerkiksi vaatimusten kasvaessa.

Joulukuu	summa	maksaja	artikkeli	tyyppi	muuta
säännölliset menot					
	€1,99	minä	Pilvitallennustila	tietotekniikka	
	€57,00	minä	kuntosali	liikunta	
	€58,99				
vaihtelevat menot					
02/12/2018	€60,00	isä	tankkaus	auto	Kävimme tankk
	€60,00				
04/12/2018	€4,20	isä	seutulippu	liikkuminen	Japanin jatkoku
	€4,20	isä	seutulippu	liikkuminen	Japanin jatkoku
	€8,40				
yhteensä	€127,39				
omia	€58,99				

Kuva 1. Nykyisessä järjestelmässä menoista on voinut tallettaa tiedon päivämäärästä, summasta, maksajasta, nimikkeestä (artikkeli), kategoriasta (tyyppi) ja vapaamuotoista tietoa (muuta). Lisäksi menon toistuvuus käy ilmi kirjauksen sijoittelusta.

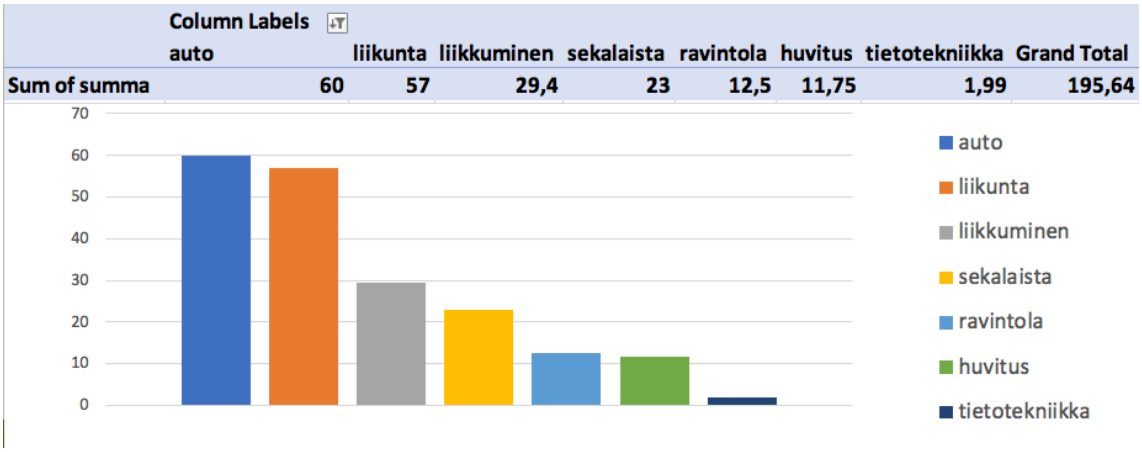
Kunkin yksittäisen päivän menojen summa on kirjattu erikseen kuten myös säännöllisten menojen summa. Tästä on useita hyötyjä. Yksi on se, että näkee, minkälainen summa kertyy yksittäisistä menoista. Esitystapa auttaa havaitsemaan konkreettisesti, että pienehköt yksittäiset menot voivat kumuloitua suhteellisen suureksi yhteissummaksi. Lisäksi vaikkapa kahden samankaltaisen päivän yhteismenoja voi verrata vaivattomasti keskenään. Samoista syistä kuukauden lopuksi on laskettu kyseisen kuukauden menojen yhteissumma. Olen kirjannut lisäksi niiden menojen yhteissumman, jossa olen ollut maksajana erotuksena muiden maksamista menoista. Esimerkiksi vanhempien kanssa asuvalle tällainen ominaisuus on käyttökelpoinen.

Nykyisellä järjestelmällä on lukuisia hyviä puolia. Se on mielestäni varsin helppolukuinen. Helppolukuisuutta lisäävät ainakin tarkoituksenmukainen rakenne sekä värien käyttö. Kuukauden menot erilaisine tietoineen ovat sen kautta käytettävissä useisiin tarkoituksiin, kuten vaikkapa kulujen karsimiseen tai ostettujen tuotteiden hintakehityksen seuraamiseen.

Kuukausi tarkastelujaksona on ollut mielestäni toimiva. Täten tulee sopivissa jaksoissa ajankohtia, jolloin tehdään yhteenvetoa menoista. Tämä parantaa käyttäjän kuvaa niin menojen kokonaismäärästä kuin niiden kehityssuunnasta tai vaikkapa omien menojen ja vuodenajan korrelaatiosta.

Järjestelmän toiminta tukee sitä, että käyttäjä on yleensä kiinnostunut vain tuloista tai menoista kerrallaan. Kuten mainittua, tulot ja menot esitetään eri taulukoissa.

Kustakin kuukaudesta on mahdollista tehdä erilaisia yhteenvetonäkymiä Excelin pivot-taulukoiden avulla. Kuvassa 2 on esitetty kategoriayhteenvetonäkymä, jossa on kuukauden menot kategorian mukaan pienimmästä suurimpaan. Lisäksi näkymässä on näistä tietueista automaattisesti muodostettu kuvaaja.



Kuva 2. Nykyisen järjestelmän esitys kuukauden menoista kategorioittain.

Yhteenvetonäkymät ovat hyödyllisiä. Esimerkiksi kategoriayhteenvetonäkymästä näkee sen, kuinka paljon rahaa on käytetty joukkoliikenteeseen. Tällaista käsitystä ei välttämättä muutoin synny, sillä kategoriaan kuuluvia tapahtumia on kuukaudessa lukuisia.

Nykyisen järjestelmän tarkastelun jälkeen voidaan todeta, että käyttäjä pystyy tekemään sen avulla riittävästi toimintoja.

Nykyisessä järjestelmässä on hyvien ominaisuuksien lisäksi joitain asioita, jotka voisivat toimia paremmin. Järjestelmän käytettävyys ei ole optimaalinen, sillä useimpien toimintojen teko vaatii käyttäjältä runsaasti toimenpiteitä. Ehkä useimmiten tällainen tilanne toistuu kirjauksia lisätessä. Lisäyksen jälkeen täytyy tehdä erilaisia muotoiluita: yhdistää soluja, luoda viiva ja värjätä tausta. Lisäksi tarvitaan erilaisia rivien lisäyksiä ja summalausekkeiden luontia, jotka nekin tehdään suurelta osin käsin.

3 Uusi järjestelmä

Uusi järjestelmä luodaan edellä mainitun järjestelmän seuraajaksi. Tarkoituksena on sisällyttää siihen useimmat hyvinä pidetyt vanhan järjestelmän ominaisuudet, mutta toisaalta poistaa edellisessä luvussa mainitut käytettävyyden heikkoudet.

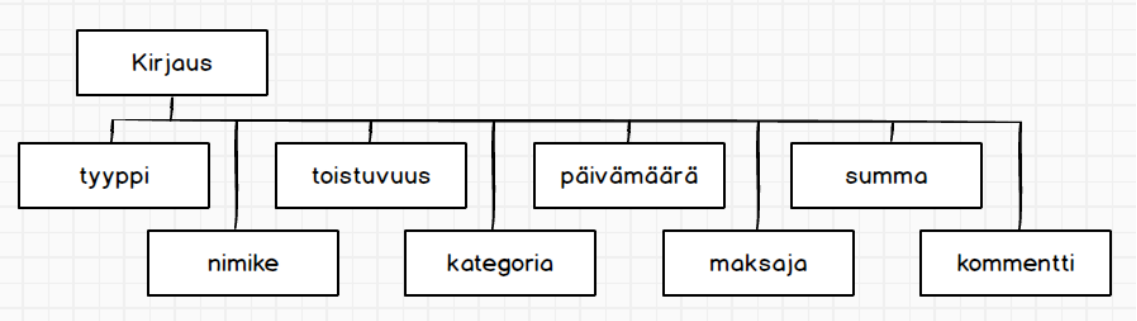
Uusi järjestelmä ohjelmoidaan itse, mikä avaa mahdollisuuksia lisätä myöhemmässä vaiheessa myös aikaisemmin vaikeasti toteutettavia tai mahdottomia toimintoja.

3.1 Siirrettävät ominaisuudet ja niihin liittyvät lisämääritykset

Kirjaukset on tarkoitus esittää kuukauden mukaan jaoteltuna myös uudessa järjestelmässä. Kuten aiemmin on mainittu, kuukauden mittaisiin ajanjaksoihin jaottelulla on hyviä puolia.

Toinen siirrettävä ominaisuus on tulojen ja menojen pitäminen erillään. Se heijastelee nykyisen järjestelmän yhteydessä tehtyä havaintoa: miltei aina ollaan kiinnostuneita yhdestä kirjaustyypistä kerrallaan. Lisäksi summatiedot ja tietyt yhteenvetonäkymät halutaan ottaa uuteen järjestelmään.

Kirjauksista on tarkoitus tallentaa likimain samat tiedot kuin aikaisemmin. Muutoksia on kaksi. Tulojen ja menojen kentät yhdenmukaistetaan eli myös tuloihin lisätään nimike-kenttä. Nimike-kenttä arvioidaan hyödylliseksi myös tulokirjauksissa, ja kirjausten käsittely ohjelmakoodissa helpottuu. Toinen muutos on se, että tietokenttien nimiä muutetaan selkeämmiksi. Esimerkiksi tyyppi on muutoksen jälkeen kategoria. Kirjauksen sisältämät tiedot on esitetty kuvassa 3.



Kuva 3. Kirjaus sisältää kahdeksan tietokenttää.

Aikaisemman järjestelmän luonteesta johtuen ei ole ollut tarvetta määritellä sääntöjä tietojen pakollisuudelle. Uudessa järjestelmässä kirjausten sisältöä käsitellään suoraan ohjelmakoodissa. Tästä seuraa se, että ohjelmakoodissa täytyy ottaa huomioon, mikä kenttä voi saada tyhjän arvon.

Uudessa järjestelmässä osa tiedoista on määritetty pakollisiksi. Tyyppi ja toistuvuus on määritetty pakollisiksi. Lisäksi summa määritetään pakolliseksi. Linjaus helpottaa ohjelmointia, sillä summia tultaneen käsittelemään ohjelmakoodissa usein. Ratkaisua tukee niin ikään se, että summa on lähes aina tiedossa. Toki käyttäjältä voi olla tarkka summa unohtunut, mutta hän voi kuitenkin kirjata arvion tai jättää kirjauksen kokonaan syöttämättä. Päivämäärätieto määritetään pakolliseksi samoista syistä kuin summa.

Uuden järjestelmän käyttöliittymästä pyritään tekemään ainakin yhtä selkeä kuin vanhan järjestelmän käyttöliittymä. Selkeyttä on vaikea mitata, mutta rautalankamallien avulla eri suunnitteluratkaisuja voidaan hahmotella, kunnes käyttöliittymä on tyydyttävä.

Käytän nykyjärjestelmää pöytätietokoneella. Olen ajatellut tehdä samoin myös järjestelmän vaihdon jälkeen. Ottaen lisäksi huomioon, että järjestelmä tulee henkilökohtaiseen käyttööni, en aseta tässä vaiheessa vaatimuksia uuden järjestelmän mobiilikäytettävyydelle.

3.2 Uudet ominaisuudet

Edellisen järjestelmän heikkouksiksi tunnistettiin käytettävyyteen liittyviä ominaisuuksia. Esimerkiksi solujen muotoilu sekä luonti vaativat runsaasti napsautuksia. Uudessa järjestelmässä on tavoitteena, että tätä ei tarvitse tehdä, vaan järjestelmä hoitaa ne automaattisesti. Samoin summatiedot ja yhteenvetonäkymät muodostettaisiin täysin automaattisesti.

Järjestelmän uusiminen tarjoaa hyvän mahdollisuuden tarkastella tarvetta uusille ominaisuuksille. Vaihe on luonnollinen, sillä suunnitteludokumentteja tehdessä järjestelmän toimintaa käydään läpi tarkasti. Suunnitelmaa tehdessä kävi niinkin, että mieleen on tullut ominaisuuksia, joita olisi voinut toteuttaa aikaisempaankin järjestelmään. Teknistä rajoitetta siihen ei ole.

Toisaalta uusi järjestelmä mahdollistaa sellaisten uusien ominaisuuksien kehittämisen, jotka vanhalla järjestelmällä eivät olisi olleet mahdollisia. Yksi tällainen on käyttäjänhallinta. Järjestelmä voitaisiin laajentaa yhden henkilön järjestelmästä useamman henkilön järjestelmäksi.

Seuraavassa on esitetty kolme uutta ominaisuutta. Järjestelmä näyttää kuukauden tulojen ja menojen yhteenlasketun summan. Täten käyttäjä näkee heti, onko kuukauden tulot vai menot suuremmat ja kuinka paljon ero on. Toinen uusi ominaisuus on se, että tulonäkymästä näkee tulotapahtumien yhteismäärän ja vastaavasti menonäkymästä näkee menotapahtumien yhteismäärän. Tämä auttaa arvioimaan kuukauden rahankäytön aktiivisuutta uudesta näkökulmasta verrattuna pelkkiin euromääräisiin summiin. Tiedon avulla voi laskea esimerkiksi, kuinka paljon yksi menokirjaus on kustantanut keskimäärin. Kolmantena käyttäjälle näytetään ilmoitus onnistuneesta kirjauksen lisäyksestä, muokkauksesta ja poistosta tai näiden epäonnistumisesta. Tarkoituksena on pitää käyttäjä tietoisena toimintansa lopputuloksesta hyvien käytäntöjen mukaisesti.

Uudet ominaisuudet arvioidaan yksinkertaisiksi toteuttaa, joten ne voidaan sisällyttää uuden järjestelmän vaatimuksiin. Yksinkertaisuudestaan huolimatta ne ovat hyödyllisiä. Kaksi ensimmäistä on samalla esimerkkejä ominaisuuksista, jotka olisi voinut toteuttaa

jo aikaisempaan järjestelmään, mutta idea toteuttamisesta on tullut vasta uutta järjestelmää suunniteltaessa.

Edellä esitetty pohdinta on tiivistetty vaatimusmäärittelyksi, joka on liitteenä 1. Toiminnallisia vaatimuksia määrittelyssä on yhteensä 12 kappaletta.

4 Käytettävyys

Käytettävyys määritellään usein laveasti tehokkuuden ja miellyttävyyden yhdistelmänä [1, s. 17]. Sovelluksen tapauksessa tehokkuutta on esimerkiksi se, että haluamansa asian saa tehtyä nopeasti. Miellyttävyyttä on sen sijaan se, että haluttu toiminto löytyy helposti ja käyttäjältä vaadittavien painallusten määrä on minimoitu. Myös näkymänvaihteluiden minimointi lisää miellyttävyyttä, koska jokaisen vaihdoksen jälkeen huomio täytyy kohdistaa uudelleen.

Tehokkuutta ja miellyttävyyttä koskevat monet aspektit, ja ne pyritään ottamaan huomioon sovellusta suunniteltaessa. Tähän on edellytyksiä, koska järjestelmä on tarkoitettu melko rajoitettuun käyttötarkoitukseen, tulojen ja menojen kirjaukseen sekä niiden seuraamiseen. Tämä on uuden järjestelmän etu verrattuna edelliseen. Edellinen järjestelmä perustuu Exceliin, joka on tehty suhteellisen monen tyyppisiin tehtäviin. Tällainen moninaisuus heikentää käytettävyyttä. [2.]

Uusi järjestelmä ei tuo juurikaan uusia ominaisuuksia, mutta käytettävyyden on tarkoitus olla merkittävästi parempi. Vaatimusmäärittelyn pohjalta tehtiin sovelluksen sisältöä ja rakennetta kuvaavat rautalankamallit, joissa kiinnitettiin erityistä huomiota käytettävyyteen. Rautalankamalleja suunniteltaessa apuna käytettiin muun muassa Jakob Nielsenin kymmentä käytettävyyden heuristiikkaa käyttöliittymille [3].

Seuraavassa on esitelty rautalankamalleja ja taustalla olevaa käytettävyysajattelua osa osalta.

4.1 Navigaatio

Sovellus on jaettu kahtia tulojen ja menojen osuuksiin. Kuvan 4 mukaisesti navigaatio heijastelee tätä jaottelua.

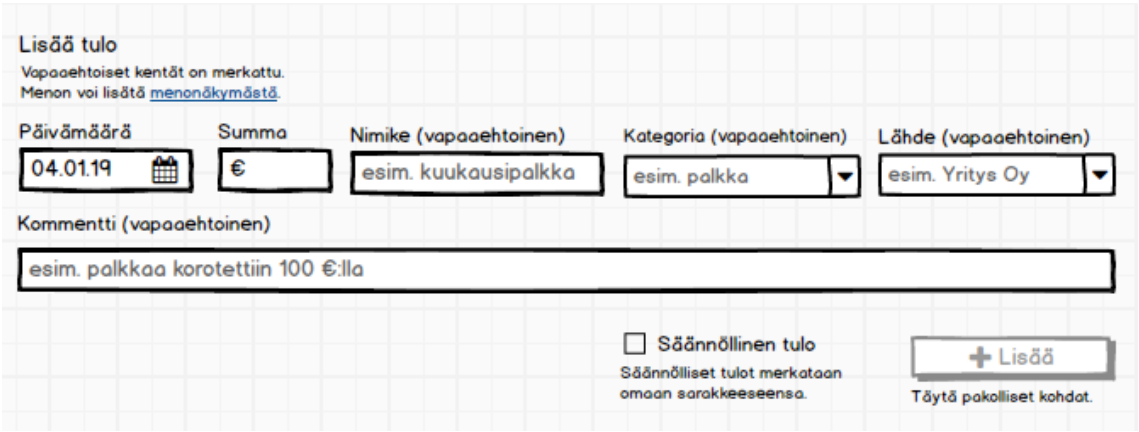
Kuva 4. Navigaatio heijastaa sovelluksen jakautumista kahteen osaan: tuloihin ja menoihin.

Navigaatio on sijoitettu yleisen tyylin mukaisesti ylälaitaan. Oletus on, että useimmat käyttäjät ovat tottuneet etsimää navigaatiota sieltä. Sovellus aukeaa menonäkymässä, koska sen arvioidaan olevan enemmän käytetty näkymä.

Nielsenin esteettinen ja minimalistinen suunnittelu -heuristiikka kertoo, että harvoin tarvittava tieto kannattaa jättää pois. Se kilpailee näkyvyydestä hyödyllisen tiedon kanssa, ja hyödyllisen tiedon näkyvyys laskee. [3.] Tämän vuoksi navigaatiosta on pyritty tekemään mahdollisimman yksinkertainen. Nykyinen sijainti on navigaatiossa tavallisella tekstillä. Toiseen sijaintii johtava teksti on linkki-tyylinen, jotta käyttäjä tietäisi sen olevan painettavissa. Tällaiseen yhteyteen ei ole vakiintuneita symboleita. Vakiintumattomien symbolien tuoma lisäarvo tekstin rinnalla arvioidaan pieneksi, joten niitä ei käytetä. [4.]

4.2 Kirjauksen lisäys

Uuden kirjauksen lisääminen on yksi sovelluksen keskeisimmistä toiminnoista. Tämän vuoksi kirjauksen lisäysosio on sijoitettu välittömästi navigaation alapuolelle ja se on aina näkyvissä kuvan 5 mukaisesti.



Kuva 5. Kirjauksen lisäysosio on navigaation alapuolella.

Kenttiä on suhteellisen monta, minkä vuoksi pohdittiin keinoja vähentää niitä. Kirjauksen tyyppi-kenttä poistettiin. Tyyppi määräytyy sen mukaan, ollaanko tarkastelemassa tuloja vai menoja. Käyttäjälle tyyppi näytetään kenttien yläpuolella. Tulonäkymässä siinä lukee ”Lisää tulo” ja vastaavasti menonäkymässä ”Lisää meno”. Samassa on ohje, kuinka toista tyyppiä oleva kirjaus voidaan lisätä.

Kunkin kentän päällä on selite, joka kertoo kentän nimen. Selite auttaa ymmärtämään, mitä kenttään on tarkoitus syöttää. Alun perin tarkoitus oli näyttää selitteet placeholder-teksteinä. Tämä olisi kuitenkin ongelmallista tilanteissa, jossa käyttäjä on jo kirjoittanut kenttään. Käyttäjä saattaa olla epävarma halutusta syötteestä, ja hän joutuu joko muistelemaan placeholder-tekstiä tai poistamaan kirjoittamansa syötteen. Sovelluksessa päädyttiin käyttämään aina näkyviä selitteitä Nielsenin heuristiikan perusteella: enemmän visuaalista sisältöä on parempi vaihtoehto kuin muistin kuormitus. [5.] Kentän päällä näkyvä selite vapauttaa placeholder-tekstin muuhun käyttöön. Placeholder-tekstiin on sijoitettu esimerkki sopivasta syötteestä.

Kenttien pakollisuuden merkkaamiseen liittyen pohdittiin kolmea eri vaihtoehtoa: merkkaamatta jättämistä, työkaluvinkki-ratkaisua ja kiinteitä tekstejä. Käydään ensin läpi vaihtoehdot, joihin ei päädytty. Merkkaamatta jättäminen hylättiin, koska käyttäjä joutuisi toteamaan tilanteen yrityksen ja erehdyksen kautta. Voi myös olla, että hän ajattelisi kaikkien kenttien olevan pakollisia, kuten perinteisissä paperilomakkeissa on tapana. Tässä tapauksessa hän saattaisi päätyä kirjaamaan järjestelmään sellaista tietoa, jota ei todel-

lisuudessa koe tarpeellisena. Ratkaisu on vastoin näkyvyyden heuristiikkaa [6]. Työkaluvinkki-ratkaisussa tiedon olisi saanut symbolisesti tähtimerkkien avulla ja kirjallisesti viemällä kursorin sopivaan kohtaan. Ratkaisussa on se etu, että kokeneet käyttäjät ymmärtänevät symbolin merkityksen eikä tilaa kulu paljoa. Toisaalta vähemmän kokeneet käyttäjät eivät välttämättä ymmärrä pelkkää symbolia. Lisäksi erityisesti heille tuottanee vaivaa etsiä selitys työkaluvinkeistä. Voikin käydä niin, että käyttäjä päätyy tekemään omia, mahdollisesti virheellisiä tulkintoja. Ratkaisu ei ole siis optimaalinen. Työkaluvinkit sopinevat paremmin sellaisten lisätietojen esitykseen, jotka eivät ole sovelluksen käytön kannalta erityisen olennaisia. [6.]

Edellä esitetyn perusteella paras ratkaisu on käyttää kiinteitä tekstejä, jolloin tieto on aina näkyvissä erikseen etsimättä. Tämä on mahdollista toteuttaa ainakin kahdella tavalla: kertoa joka kentän kohdalla kyseisen kentän pakollisuus tai vaihtoehtoisesti mainita asiasta vain jommankumman tyyppisten kenttien kohdalla. Tilan säästämiseksi päädyttiin jälkimmäiseen vaihtoehtoon. Vain pakollisten kenttien näyttämisestä ja vain vapaaehtoisten kenttien näyttämisestä toteutettiin oma rautalankamalli. Vaihtoehtojen välillä ei havaittu suurta eroa. Esimerkiksi tilan tarve on suunnilleen sama, joten molemmat vaihtoehdot todettiin tilanteeseen sopiviksi. Formulate-sivuston artikkelin perusteella jälkimmäisellä vaihtoehdolla on kuitenkin etu. Tätä perustellaan muun muassa sillä, että kenttien täyttäminen on oletettu toiminta. Vain tästä poikkeavia kohtia, eli vapaaehtoisia kenttiä, kannattaa merkata. Argumenttien perusteella päädyttiin ratkaisuun, jossa merkataan ainoastaan vapaaehtoiset kentät. Lisäksi käyttäjää informoidaan tästä merkkaustavasta tekstillä: "vapaaehtoiset kentät on merkattu". [7.]

Kommenttikenttä poikkeaa muista kentistä siinä, että siihen saatetaan kirjoittaa pitkä teksti. Tämä otettiin huomioon siten, että kommenttikenttä on omalla rivillään.

Sovellus pyrkii vähentämään tarvetta käyttäjän napsautuksille. Seuraavassa on esitetty keinoja, joilla tätä tavoitetta on saavutettu kirjauksen lisäysosiossa. Yksi tyypillinen tilanne on, että käyttäjä lisää kirjauksen nykyiselle päivälle. Käyttötilannetta on helpotettu siten, että nykyinen päivämäärä on esitäytetty kirjaukseen. Samoin summakentässä on esitäytetty euromerkki. Käyttäjän ei tarvitse syöttää sitä joka kerta itse, ja toisaalta järjestelmä tukee ainakin aluksi ainoastaan euroja. Kategoria- ja lähdekentät toimivat vaih-

toehtoisesti pudotusvalikkoina. Käyttäjä näkee valikoissa aikaisemmin syöttämiään arvoja, joten hänen ei tarvitse syöttää toistuvia tekstejä yhä uudelleen. Arvot haetaan jo järjestelmään tallennetuista samaa tyyppiä olevista kirjauksista. Kaikkia arvoja ei esitetä tiedon relevanssin vuoksi: oletetaan, että harvoissa tilanteissa sama taho aiheuttaa sekä menoja että tuloja.

Lisää-painikkeessa on käytetty plus-merkkiä. Tämä antaa käyttäjille vihjeen painikkeen toiminnasta jo ennen tekstin lukemista. Suosittuna käytäntönä sovelluksissa on assosioida plus-merkki jonkin asian lisäykseen. Symbolin käyttö ei ole kuitenkaan minkään standardin alaista, eikä se toimi kaikissa sovelluksissa samalla tavalla. Sen vuoksi symbolin rinnalla on lisäksi seliteteksti. [4.]

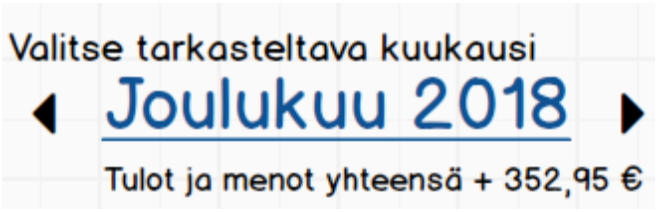
Säännöllinen meno -valintaruudun yhteydessä on aina näkyvissä lyhyt selite, joka kertoo, miten valinta vaikuttaa kirjauksen käsittelyyn. Nielsenin järjestelmätilan näkyvyys -heuristiikan mukaisesti käytettävyyttä parantaa tieto, joka on käyttötilanteelle olennaista [8].

Tulon lisäys on muuten yhdenmukainen, mutta meno-sanat on korvattu tulo-sanoilla. Yhdenmukaisuus sovelluksen sisällä helpottaa käyttäjän toimintaa. [9.]

4.3 Kuukausivalitsin

Käyttäjä voi valita kuukauden kahdella tapaa. Hän voi selata kuukausia yksi kerrallaan eteen- ja taakse-nuolen avulla. Toinen vaihtoehto on napsauttaa kuukauden tai vuoden nimeä, mikä avaa kuukausivalitsimen. Käyttäjällä on täten valinnanvaraa, joka lisää käytettävyyttä. Ensiksi mainittu tapa on tehokas läheisten kuukausien tarkasteluun, kun taas jälkimmäinen tapa on tehokas etäisten kuukausien tarkasteluun.

Kuvan 6 mukaisesti kuukausikontrollien päällä on selite, jotta osion tarkoitus käy mahdollisimman selväksi. Lisäksi kuukauden nimessä ja vuosiluvussa on korostus, joka vihjaa käyttäjälle vuorovaikutuksen mahdollisuutta.



Kuva 6. Tarkasteltavan kuukauden voi valita kahdella tapaa.

Kuukausivalitsimen alla näkyy kuukauden tulojen ja menojen yhteissumma. Sijoittelulla on pyritty korostamaan summan liittyvän yllä olevaan kuukauteen.

4.4 Kirjaukset

Kirjaukset ovat luonteeltaan taulukkomuotoista tietoa, joten taulukko on paras vaihtoehto esittää ne sovelluksessa [10]. Taulukon paikaksi valittiin sivun alaosa. Mikäli taulukko on suuri, sen jälkeen tulevat elementit ovat vaikeasti tavoitettavissa. Sijoittelua puoltaa myös se, että on tyypillistä valita tarkasteltava kuukausi ennen itse sisällön tarkastelua. Rautalankamalli kirjausten esitystavasta on kuvassa 7.

Kuva 7. Kirjaukset esitetään taulukkomuodossa.

Aikaisemman järjestelmän tapaan uudessa järjestelmässä on todettu luontevaksi sijoittaa säännölliset ja epäsäännölliset kirjaukset erikseen. Sijoittelu heijastaa tarvetta erotella kirjaukset toisistaan. Tällaisen erottelun hyvä puoli on siinä, että toisen tai kummat-

kin osiot voi piilottaa helposti. Käyttäjä saa vihjeen osioiden piilottamisen mahdollisuudesta otsake-palkin oikeasta laidasta, jossa lukee "piilota osio" tai "näytä osio". Toinen vihje on se, että kursori muuttuu käsi-symboliksi kohdissa, joissa toiminto on käytössä.

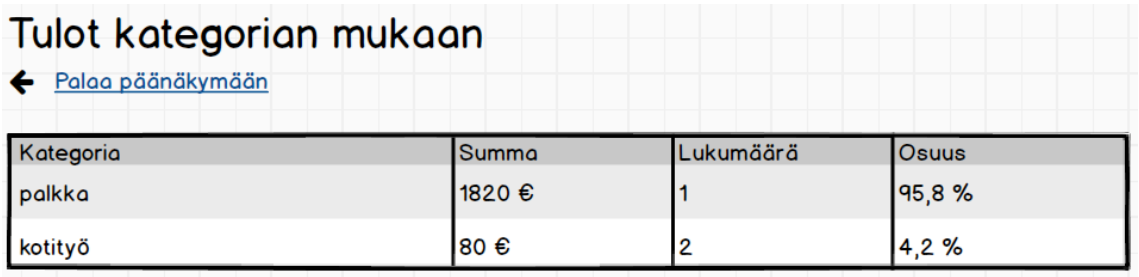
Mikäli samalla päivällä on allekkain useampi kirjaus, näytetään päivämäärä vain ylimmällä rivillä. Taulukko on siistimpi ilman, että informaatiosisältö pienenee. Tarpeeton toisteisuus vähenee, joka on esteettisyys ja minimalistiset toiminnallisuudet -heuristiikan mukaista. [3.] Sen sijaan muilla sarakkeilla tätä ei tehdä, koska sisältö ei ole järjestetty niiden perusteella. Käyttäjä voisi luulla, että kentät ovat tyhjiä vain siitä syystä, että arvoa ei ole syötetty.

Sarakeotsikot ja tyypin väliotsikko toteutetaan aina näkyvinä. Muutoin ne saattavat jäädä näytön ulkopuolelle käyttäjän selatessa sivua alaspäin. Aina näkyvät otsikot vähentävät käyttäjän muistikuormitusta.

4.5 Yhteenvetonäkymät

Kuvassa 7 summa, kategoria ja lähde on merkattu linkeiksi. Ne viestivät sitä, että kukin linkki johtaa erilaiseen yhteenvetonäkymään.

Kuvassa 8 on esitelty kategoriayhteenvetonäkymä. Siinä, kuten muissa yhteenvetonäkymissä, rivit on järjestelty summan mukaan suurimmasta pienimpään.


Tulot kategorian mukaan

Palaa päänäkymään

Kategoria	Summa	Lukumäärä	Osuus
palkka	1820 €	1	95,8 %
kotityö	80 €	2	4,2 %

Kuva 8. Ohjelmassa on kolme erilaista yhteenvetonäkymää, joista yksi on kuvassa esitetty kategorianäkymä.

Yhteenvetonäkymissä ei ole mahdollisuutta vaikuttaa rivien järjestykseen esimerkiksi siten, että ne olisivat summan mukaan käänteisessä järjestyksessä. Resurssienkäyttö on suurin syy, mutta lisäksi vaikuttaa se, että ominaisuuden ei arvioida tuovan juurikaan lisäarvoa. Tuote on myös sitä käytettävämpi, mitä vähemmän ominaisuuksia siinä on [11]. Täten on tärkeää tunnistaa oleelliset ominaisuudet ja keskittyä niihin. Samoista syistä yhteenvetonäkymiä ei tehdä tässä vaiheessa nimike- tai kommenttikentän perusteella.

Käyttäjälle tarjotaan mahdollisuus palata päänäkymään taulukon yläpuolella. Tämä navigaatio-ominaisuus on tärkeä, koska se on ainoa tapa palata päänäkymään. Oletettavasti käyttäjä haluaa tehdä niin usein, joten toiminnon on hyvä olla keskeisellä paikalla.

4.6 Muokkausnäkymä

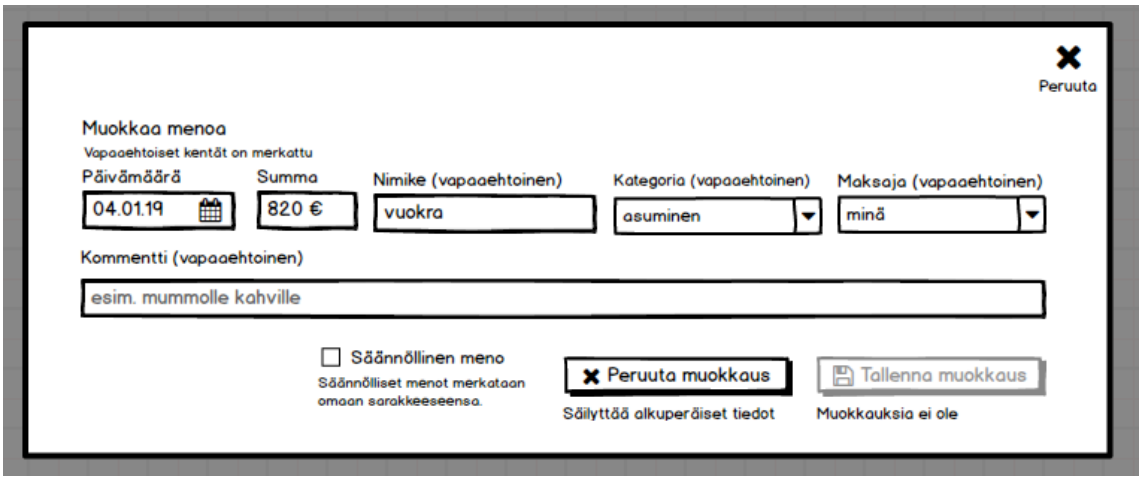
Kuhunkin kirjaukseen on liitettävä muokkaustoiminto. Yleisesti ottaen, mitä lähempänä muokkauksen voi tehdä itse muokattavaa kohdetta, sitä parempi on käytettävyys. Optimaalisin tilanne on siis mahdollistaa muokkaus samaan tapaan kuin Excelissä: tekstiä muokataan suoraan esityspaikassa. Sovelluksen tapauksessa esiin nousee kuitenkin erilaisia teknisiä kysymyksiä. Jos kohdetta muokataan suoraan, tallennetaanko jokainen muutos heti tietokantaan. Toisekseen, kuinka ratkaista kommenttikentän pieni tila muokkauksessa. Kommenttia muokatessa olisi hyvä nähdä kommentti kokonaisuudessaan. Ottaen huomioon se, että muokkauksia ei oletettavasti tehdä kovin usein, mietittiin helpommin teknisesti toteutettavia vaihtoehtoja.

Vaihtoehtoisia ratkaisuja pohdittaessa on edelleen tärkeää se, että muokkauksen voisi tehdä mahdollisimman lähellä itse muokkauskohdetta. Lähimpänä suoraa muokkausta taitaa olla kullekin riville sijoitettu muokkauspainike, joka avaa muokkausnäkymän. Muokkauspainikkeen toteutukselle pohdittiin kahta eri vaihtoehtoa: tilannesidonnaisesti tai aina näytettävää painiketta. Ensimmäinen vaihtoehto tarkoittaa sitä, että rivin muokkauspainike näkyy vain, kun kursori on rivin päällä. Hyvänä puolena tässä ratkaisussa on se, että visuaalinen kuorma vähenee. Toisaalta toiminnon löydettävyys heikkenee. Toiminnollisuus pitää joko tietää tai siitä pitää olla ohjeteksti. Ensiksi mainittua ei voi

käyttäjiltä vaatia. Toinen vaihtoehto, ohjeteksti, on parempi, mutta sekin vaatii usein jonkin verran etsimistä. Parempi siis on, jos painike on aina nähtävillä. Oman arvioni mukaan visuaalinen kuorma ei kasva liikaa, joten aina näkyvä painike on perusteltu. Selitteen lisääminen painikkeeseen veisi suhteessa paljon tilaa. Tämän vuoksi se näkyy vain hiiren ollessa rivin päällä. Sen sijaan taulukon yläpuolella on koko ajan näkyvillä yleisselitys, miten rivejä voi muokata. Tällaisen ratkaisun tavoitteena on yhdistää tilannesidonnaisen ja staattisen käyttöliittymän parhaat puolet. Kuva 9 havainnollistaa ratkaisua.

Kuva 9. Muokkaustoiminnossa on pyritty yhdistämään tilannesidonnaisen ja staattisen käyttöliittymän parhaita puolia.

Muokkaa-painike on sijoitettu ensimmäiseksi, koska muokkaus on avautuvista valinnoista vähemmän vahva toiminto. Painikkeen painaminen avaa muokkausnäkymän, jonka toteuttamistavaksi mietittiin kahta vaihtoehtoa: modaalia tai uutta sivua. Vaihtoehdoista päädyttiin modaaliin, sillä se vaatii käyttäjältä oletettavasti vähemmän uudelleen orientoitumista. Modaalin rajoite on se, että tilaa on vähemmän. Rautalankamallista voidaan kuitenkin arvioida, että tässä tapauksessa tilaa on riittävästi. Modaali on esitetty kuvassa 10.



Kuva 10. Muokkausnäkymä toteutetaan modaalina.

Modaalin rautalankamallissa seurattiin yleistä käytäntöä, että muu sivu tummennetaan taustalla. Kontrasti lisääntyy modaali-ikkunan ja muun sivun välillä, mikä saa aikaan syvyysvaikutelman. Vaalea modaali näyttää olevan lähempänä katsojaa. Lähempänä katsojaa olevat asiat kiinnittävät katsojan huomion, mikä on tilanteessa tarkoituksenmukaista: käyttäjän halutaan keskittyvän modaaliin. [1, s. 85–86, 128.]

Modaalin ja kirjauksen luontiosion sisältö on lähes identtinen. Täten muokkausnäkymä on käyttäjälle pääpiirteissään tuttu. Luonnollisesti lisää-painiketta ei kuitenkaan ole, vaan sen sijaan näytetään tallenna muokkaus ja peruuta muokkaus -painike. Tallenna muokkaus -painiketta ei voi painaa, jos muokkauksia ei ole tehty. Täten käyttäjälle ei tule epätietoisuutta, onko hän tehnyt muutoksia vai ei. Kummankin toimintonäppäimen alla on lisäksi seliteteksti. Peruuta muokkaus -painikkeen alla on teksti: säilyttää alkuperäiset tiedot. Täten käyttäjä saa varmuutta painaa painiketta, jos hän on tehnyt joitain muutoksia, joita ei halua säilyttää. [8.]

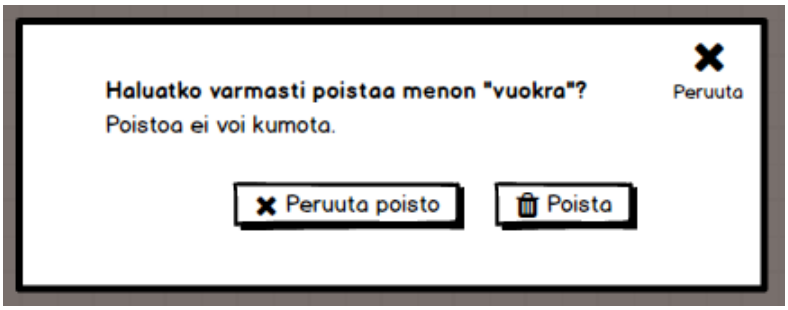
Yleisenä käytäntönä modaalien oikeassa yläkulmassa on painike, josta modaalin voi sulkea. Tätä käytäntöä seurataan, koska käyttäjät ovat tottuneet siihen. Painike toimii samoin kuin peruuta muokkaus -painike. Sen yhteyteen ei ole mahtunut yhtä pitkää selvennystä kuin painikkeen yhteyteen, mutta saman symbolin ja sanan käyttö antaa varmuutta samasta toiminnasta. Kokeneet käyttäjät uskaltanevat painaa painiketta näillä tiedoilla, ja arvostavat tarjottua lisävaihtoehtoa. Varman päälle ottavat käyttäjät voivat käyttää peruuta muokkaus -painiketta.

Mikäli muutoksia on tehty, peruuta muokkaus -painikkeen painaminen aiheuttaa varmistuksen. Toiminto pienentää riskiä peruuttaa muutokset vahingossa.

4.7 Kirjauksen poisto

Kuten kuvasta 9 nähdään, kirjauksen poisto-toiminnallisuus on sijoitettu samaan paikkaan kuin muokkaus. Kyseessä on läheisesti toisiinsa liittyvät toiminnot, joten sijoittelu on tehty samalla logiikalla.

Poiston kaltaiset vahvat tapahtumat on syytä aina varmistaa, jotta toiminto ei tapahtuisi vahingossa. Tiedetään, että vahinkopainalluksia tapahtuu kaikissa järjestelmissä. [3.] Varmistus on esitetty kuvassa 11.

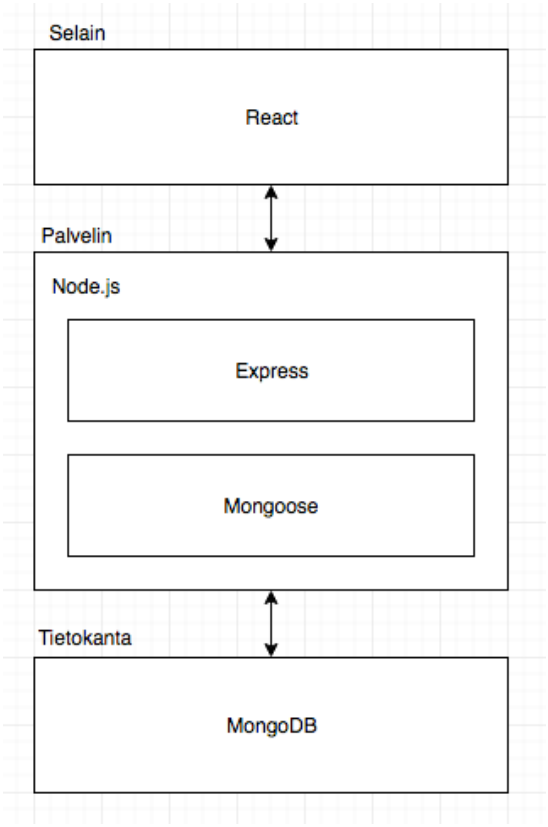


Kuva 11. Kirjauksen poisto varmistetaan käyttäjältä, jotta toiminto ei tapahtuisi vahingossa.

Poiston varmistavassa modaalissa on mainittu kirjauksen nimike, jos se on olemassa. Vaihtoehtoisesti näytetään kirjauksen summa. Tarkoituksena on pitää käyttäjä mahdollisimman tietoisena, mitä kirjausta toiminto koskee. Järjestelmätilan näkyvyys -heuristiikan mukaisesti mainitaan lisäksi, että poistoa ei voi kumota. [8.]

5 Sovelluksen tekniikat

Sovellus koostuu kolmesta osasta: asiakaspuolesta, palvelinpuolesta ja tietokannasta, ja ohjelmointikielenä on kokonaisuudessaan JavaScript. Asiakaspuoli on tehty React-ohjelmistokehyksellä. Palvelinpuoli on ohjelmoitu Node.js-ympäristöllä ja Express-ohjelmistokehyksellä. Sovelluksen tietokanta on MongoDB, ja sitä käytetään Mongoose-mallintajan kautta. Tekniikat on esitetty kuvassa 12.



Kuva 12. Sovelluksen tärkeimpiä tekniikoita.

JavaScriptin yksinomainen käyttö mahdollistaa nopean kehityksen. Merkittäviä syitä on ainakin kaksi. Ensiksi ohjelmoijan ei tarvitse perehtyä useampaan ohjelmointikieleen. Toiseksi JavaScriptin avulla on nopea tehdä sovelluksia, sillä se hoitaa ohjelmoijan puolesta useita asioita automaattisesti.

5.1 ECMAScript 2015 -standardi

JavaScriptiin tuli runsaasti ominaisuuksia ECMAScript 2015 (lyhennettynä ES2015 ja ES6) -version myötä [12]. Laskurisovelluksen hyödyntämiä ominaisuuksia on esitetty seuraavana.

5.1.1 Let-avainsana

Muuttujat määritettiin ennen ES2015-versiota var-avainsanalla. Var-muuttuja on määritelty kaikkialla funktiossa muuttujan alustamisen jälkeen, myös muuttujaa ympäröivän lohkon ulkopuolella. [13.]

Uudemmalla let-avainsanalla määritetty muuttuja on voimassa vain muuttujaa ympäröivän lohkon sisäpuolella [13]. Ominaisuus vähentää ohjelmointivirheitä. Ohjelmoija ei voi käyttää vahingossa toiseen lohkoon tarkoitettua muuttujaa. Tämän vuoksi laskurisovelluksessa on käytetty let-avainsanaa var-avainsanan sijaan.

Muuttujan näkyvyyttä voidaan pitää merkittävimpänä erona var- ja let-muuttujien välillä, mutta muitakin eroja on. Yksi ero on se, että let-muuttujaa ei voi määrittää kahta kertaa näkyvyysalueessa [14]. Sen sijaan saadaan virheilmoitus. Tämä estää määrittämästä epähuomiossa kahta samannimistä muuttujaa yhteen näkyvyysalueeseen.

Sovelluksessa käytetään myös ES2015:sta const-avainsanaa. Let- ja const-avainsanalla määritetyt muuttujat toimivat muutoin samoin, mutta const-muuttuja täytyy alustaa muuttujan esittelyn yhteydessä eikä const-muuttujan arvoa voi muuttaa.

5.1.2 Nuolifunktiot

Nuolifunktioilmaisu tarjoaa tiiviimmän tavan määrittää funktioita. Function-avainsanan sijaan riittää nuolta kuvaava merkkiyhdistelmä (=>), josta ilmaisun nimi tulee. Kuvassa 13 on kaksi samalla lailla toimivaa funktiota: kumpikin laskee kaksi lukua yhteen ja palauttaa summan. Ylempänä on määritetty funktio function-avainsanalla ja alempana nuolifunktioilmaisulla.

```
3   const funktio1 = function (parametri1, parametri2) {
4     return parametri1 + parametri2
5   }
6
7   const funktio2 = (parametri1, parametri2) => {
8     return parametri1 + parametri2
9   }
```

Kuva 13. Kuvan funktiot laskevat kaksi numeroa yhteen. Ylempi on määritetty function-avainsanalla ja alempi nuolifunktioilmaisulla.

Nuolifunktioilmaisua on mahdollisuus lyhentää entisestään, jos funktio sisältää ainoastaan return-lauseen. Tällöin voidaan jättää pois aaltosulkeet sekä itse return-avainsana. Nuolen jälkeen sijoitettu ohjelmakoodi suoritetaan ja se palautetaan funktion tuloksena. Kuvassa 14 on esitetty yhteenlaskufunktio lyhennetyssä nuolifunktiomuodossa.

```
9
10  const funktio3 = (parametri1, parametri2) => parametri1 + parametri2
11
```

Kuva 14. Nuolifunktioilmaisusta voidaan jättää return-avainsana ja aaltosulkeet pois, mikäli halutaan palauttaa nuolen oikealla puolella oleva arvo.

Keskeinen ero function-avainsanalla määritetyn funktion ja nuolifunktion välillä on this-viitteen käyttäytymisessä. Nuolifunktiolle ei määritellä this-viitettä kutsumistavan perusteella. Sen sijaan this määräytyy hierarkiassa ylempien lohkojen perusteella tavallisten muuttujien näkyvyyssääntöjen mukaisesti. Jos nuolifunktio sijaitsee luokan sisällä class property -syntaksin mukaisesti, this-avainsana viittaa luokkaan. Tästä on hyötyä Reactin kaltaisessa luokkaperustaisessa ohjelmoinnissa: funktiossa olevaa this-avainsanaa ei tarvitse sitoa luokkaan erikseen. [15.]

5.1.3 Promiset

JavaScriptin yksi keskeinen ominaisuus on asynkroniset kutsut. Asynkroniset kutsut käynnistävät toimintoja, joiden loppuunsuorittamista ei odoteta. Odottelun sijaan voidaan

suorittaa muita komentoja ja kutsun käynnistämään palvelupyyntöön palataan myöhemmin. [16.] JavaScript-ohjelmakoodissa ylempänä oleva toiminto ei tule siis välttämättä suoritetuksi kokonaan ennen kuin alempi rivi tulee suoritukseen [17]. Yleensä asynkronisella tyylillä on toteutettu siirräntää sisältävät toimenpiteet kuten verkon yli kommunikointi tai tietokannan käyttö. Tyylin tarkoitus on estää pitkät odotukset, joiden aikana sovellus muuttuu havaittavasti epäresponsiiviseksi. [18.]

Asynkroniselle kutsulle annetaan yleensä parametrina niin sanottu takaisinkutsufunktio (engl. callback function). Takaisinkutsufunktio ajetaan silloin, kun asynkroninen palvelupyyntö on saatu suoritettua kokonaisuudessaan. Mekaniikka on tarpeen, koska usein vasta asynkronisen palvelupyynnön suorituksen jälkeen on käytössä takaisinkutsufunktion ajamiseksi tarvittava tieto. Esimerkiksi tietokantaa käytettäessä täytyy tietää, että tietokantaoperaatio on onnistunut ja välittää tietokannalta saatu data takaisinkutsufunktiolle. Takaisinkutsufunktiossa voidaan tehdä myös uusi asynkroninen palvelupyyntö, jonka valmistuttua suoritetaan siihen liittyvä takaisinkutsufunktio. Asynkronisia kutsuja ja niihin liittyviä takaisinkutsufunktioita voi olla vastaavalla tavalla hyvin montakin peräkkäin. Tästä voi seurata ohjelmakoodi, johon tulee useita ohjelmakoodin lohkoja sisäkkäin kuvan 15 tapaan. Sisäkkäisten takaisinkutsufunktioiden luettavuus ei ole hyvä, jonka vuoksi tilannetta kutsutaan englanninkielisellä lempinimellä *callback hell*. [18.]

```
70  asynchronousCall1( (error, dataToCallback1) => {
71    // ensimmäisen takaisinkutsufunktio
72    if(error) {
73      // virhetilanteen käsittely
74    } else {
75      // onnistuneen tilanteen käsittely,
76      // jossa tehdään lopuksi toinen asynkroninen palvelupyyntö
77      asynchronousCall2( (error, dataToCallback2) => {
78        // toinen takaisinkutsufunktio
79        if(error) {
80          // virhetilanteen käsittely
81        } else {
82          // onnistuneen tilanteen käsittely
83        }
84      })
85    }
86  })
```

Kuva 15. Useampi sisäkkäinen asynkroninen kutsu ja niihin liittyvät takaisinkutsufunktiot voivat johtaa vaikeasti luettavaan ohjelmakoodiin, niin sanottuun callback helliin.

ES2015 sisältää uuden mekaniikan, jossa hyödynnetään promiseiksi kutsuttuja olioita. Tekniikkaa käytettäessä takaisinkutsufunktioita ei kytketä suoraan asynkronisiin kutsuihin, mikä selkeyttää syntaksia. Sen sijaan asynkroninen kutsu palauttaa välittömästi promise-olion, joka välittää tietoa asynkronisesta palvelupyynnöstä takaisinkutsufunktiolle. Takaisinkutsufunktiot liitetään promiseen. Promisella on kolme tilaa, joiden nimet ovat vapaasti suomennettuna odottaa, valmis ja hylätty. Odottaa-tilassa asynkronisen palvelupyynnön suoritus on vielä kesken. Valmis-tilassa asynkronisen palvelupyynnön suoritus on päätetty onnistuneesti, ja sen paluuarvo on tallennettu promise-olioon. Hylätty-tilassa asynkronisen palvelupyynnön suoritus on päätynyt virheeseen ja virheilmoitus on tallennettu promiseen. Promise-olio pitää siis sisällään oleelliset tiedot, jotka liittyvät asynkroniseen palvelupyyntöön. Takaisinkutsufunktio voidaan liittää promiseen missä tahansa vaiheessa. Kytkentä on siis mahdollinen, vaikka asynkroninen palvelupyyntö olisi suoritettu loppuun, mistä on hyötyä tietyissä tilanteissa. [19.]

Promise-tekniikan käyttäminen on yksi tapa välttää callback hell [17]. Syntaksi pysyy kuvan 16 kaltaisesti yhdessä tasossa, ja luettavuus on parempi. Selkeyttä tuo myös se, että onnistuneen ja epäonnistuneen tilanteen takaisinkutsufunktiot on eriytetty toisistaan.

Onnistuneen tilanteen takaisinkutsufunktio liitetään then-metodilla. Epäonnistuneen tilanteen takaisinkutsufunktio liitetään catch-metodilla tai vaihtoehtoisesti kaksiparametrisella then-metodilla. Promise palautaa aina uuden promisen, jota voidaan käyttää promise-ketjun luomisessa. Täten voidaan kytkeä useampi asynkroninen kutsu toisiinsa siten, että jälkimmäinen kutsu tulee suoritukseen vain, jos aiempi palvelupyyntö on suoritettu onnistuneesti. Kuvan 16 esimerkin tapaan ensimmäisessä takaisinkutsufunktiossa asynkroninen kutsu (asynchronousCall2) täytyy merkata return-avainsanalla, jotta se kytkeytyy toiseen promiseen (promise2). Virhetilanne täytyy merkata throw-avainsanalla kuvan mukaisesti, jotta virhe välittyy promise-ketjussa. Myös promise2 palauttaa uuden promisen, mutta ohjelmakoodissa sitä ei oteta muuttujaan talteen. Mikäli promisessa ei käytetä return-avainsanaa, ketjussa seuraava promise sisältää arvon undefined. [19.]

```
88  const promise = asynchronousCall1()
89  // Ensimmäinen asynkroninen kutsu on tehty
90  // Lisätään takaisinkutsufunktiot promise1-olioon
91  const promise2 = promise
92      .then(dataToCallback1 => {
93          // ensimmäisen takaisinkutsufunktion sisältö, jossa
94          // tehdään lopuksi toinen asynkroninen palvelupyyntö
95          return asynchronousCall2()
96      })
97      .catch(error => {
98          // ensimmäisen asynkronisen palvelupyynnön virhetilanteen käsittelijä
99          throw error
100     })
101     // Toinen asynkroninen kutsu on tehty
102     // Lisätään takaisinkutsufunktiot promise2-olioon
103 promise2
104     .then(dataToCallback2 => {
105         // toisen takaisinkutsufunktion sisältö
106     })
107     .catch(error => {
108         // toisen asynkronisen palvelupyynnön virhetilanteen käsittelijä
109     })
```

Kuva 16. Promise-tekniikkaa voidaan käyttää tekemään asynkronisesta ohjelmakoodista helppolukuisempaa.

Mikäli promise-olioita ei kirjoita näkyviin, ohjelmakoodista tulee entistä siistimpi kuvan 17 mukaisesti. Ohjelmakoodi toimii muuten samoin, mutta käytössä on ainoastaan yksi virhetilanteen käsittelijä, joka on yhteinen koko promise-ketjulle. Yhteistä virheenkäsittelijää

kannattaa käyttää mahdollisuuksien mukaan, sillä myös se tekee ohjelmakoodista helpolukuisemman.

```
139  asynchronousCall1()
140      .then(dataToCallback1 => {
141          // ensimmäisen takaisinkutsufunktion sisältö, jossa
142          // tehdään lopuksi toinen asynkroninen palvelupyyntö
143          return asynchronousCall2()
144      })
145      .then(dataToCallback2 => {
146          // toisen takaisinkutsufunktion sisältö
147      })
148      .catch(error => {
149          // promise-ketjun yhteinen virheenkäsittelijä
150      })
```

Kuva 17. Promise-tekniikalla tehty ohjelmakoodi lyhenee, jos promise-olioita ei laita näkyviin tai käyttää promise-ketjulle yhteistä virheenkäsittelijää.

5.2 Asiakaspuoli

Asiakaspuoli on React-ohjelmistokehyksellä tehty ohjelma, joka toimii Internet-selaimessa. React on käyttöliittymien ohjelmointiin tarkoitettu kehys, ja käyttäjä vuorovaikuttaa laskurisovelluksen kanssa tämän osan kautta.

5.2.1 React

Reactilla kehitetyt ohjelmat ovat yleensä niin sanottuja yhden sivun sovelluksia kuten tämän työn laskurisovellus. Termi tarkoittaa, että sivua ei ladata kokonaisuudessaan kertaakaan uudestaan, vaan sisältöä muutetaan ainoastaan dynaamisesti. Dynaaminen sisältö haetaan taustalla palvelimelta. Varsinkin jos selaimen eteen- ja taakse-painike on toteutettu imitoimaan usean sivun sivustoa, käyttäjä ei välttämättä huomaa, että kyseessä on yhden sivun sovellus. Lähestymistapa vähentää selaimen tarvetta tulostaa sisältöä ja pienentää Internetin kautta haettavan datan määrää, joten sovellus toimii nopeammin. [20.]

Tavallinen React-sovellus vaatii, että selaimen JavaScript-ominaisuus on päällä. Syy on siinä, että käytännössä kaikki sisältö luodaan JavaScript-ohjelmakoodin kautta. Kuvassa 18 on laskurisovelluksen HTML:n body-osuus ennen JavaScript-ohjelmakoodin ajamista. Osuus koostuu kahdesta elementistä. Noscript-elementti ilmoittaa JavaScriptin tarpeellisuudesta. Div-elementti toimii kiinnepisteenä, johon JavaScript-ohjelmakoodi tulostaa tuottamansa sisällön.

```
27  <body>
28    <noscript>You need to enable JavaScript to run this app.</noscript>
29    <div id="root"></div>
30  </body>
```

Kuva 18. Sovelluksen HTML:n body-osuus on miltei tyhjä ennen kuin React-ohjelmakoodilla luotu sisältö sijoitetaan paikalleen.

React voi tulostaa sisältöä HTML-sivulle ReactDOM.render-ilmaisulla. Parametriksi annetaan esimerkiksi tulostettava HTML-elementti tai React-komponentti. Laskurisovelluksen tapauksessa tyhjään div-elementtiin tulostetaan koko sovelluksen sisältävä pääkomponentti, App kuvan 19 mukaisesti. React-dom-moduuli täytyy olla importoitu aina tulostuskomentoa käytettäessä. Mikäli tulostetaan nimenomaan React-komponentti, täytyy importoida myös React-niminen moduuli (kuvan rivi 1). Jos komponenttia ei ole määritetty samassa tiedostossa, myös se pitää importoida (rivi 3).

```
1  import React from 'react';
2  import ReactDOM from 'react-dom';
3  import App from './App';
4
5  ReactDOM.render(<App />, document.getElementById('root'));
```

Kuva 19. ReactDOM.render-ilmaisulla voidaan tulostaa sisältöä HTML-sivulle.

5.2.2 JSX

Reactin ohje suosittelee React-sovelluksiin JSX-nimistä JavaScript-laajennosta [21]. JSX:llä voidaan kuvata sekä rakennetta että toimintalogiikkaa. Rakennetta määritetään HTML-tyyppisillä elementeillä ja toimintalogiikkaa JavaScriptillä.

Kuvassa 20 on JSX-merkkausta. Rakennetta kuvaava syntaksi muistuttaa hyvin paljon HTML-merkkausta. Tärkeimpiä eroja on kaksi. Ensimmäinen ero on se, että osa attribuuttien avainten nimistä on eri. Esimerkiksi HTML:n attribuuttia class vastaa JSX:ssä attribuutti className. Reactin oman ohjeen mukaan syynä tähän on se, että JSX on enemmän JavaScriptiä kuin HTML:ää. Toinen ero on se, että JavaScript-ohjelmakoodia voidaan laittaa HTML-tyylisen merkkauksen sekaan. JavaScript-kohdat merkataan aaltosulkeilla. Kuvassa määritetään tapahtumankäsittelijä JavaScript-muuttujan avulla, mutta aaltosulkeiden sisällä voi olla mikä tahansa JavaScript-lauseke. [21.]

```
41    <input
42      className="submit-button link"
43      type="submit"
44      value="Muokkaa"
45      onClick={updateEntry(entryToBeModified)}
46    />
```

Kuva 20. JSX-laajennos tuo HTML:n kaltaisen merkkauksen JavaScriptiin.

JSX tekee mielestäni Reactin käytöstä miellyttävämpää, mutta mikä tahansa määritys voidaan tehdä myös ilman sitä. JSX käännetään puhtaaksi JavaScriptiksi ennen ajoa. Edellä oleva HTML-elementti voitaisiin luoda Reactin createElement-metodilla ilman JSX:ää kuvassa 21 esitetyllä tavalla. Metodin ensimmäiseksi parametriksi tulee HTML-elementin tai komponentin nimi. Toiseksi parametriksi tulee olio, jossa on attribuuttien tai propertyjen avain-arvoparit. Kolmanneksi tulee mahdolliset lapsielementit tai -komponentit. Null-arvo vastaa sitä, ettei lapsielementtejä tai -komponentteja ole. Kaksi viimeistä parametria ovat vapaaehtoisia. [22.]

```
const modifyButton = React.createElement(
  'input',
  {
    className: 'submit-button link',
    type: 'submit',
    value: 'Muokkaa',
    onClick: updateEntry(entryToBeModified)
  },
  null
)
```

Kuva 21. React-elementin voi määritellä myös ilman JSX-laajennosta.

Esimerkkejä vertaillessa voi havaita, että JSX-merkkaus on helpommin hahmotettavissa. Lisäksi tilanne korostuu, jos elementillä tai komponentilla on lapsia.

5.2.3 Komponentit

React-sovellus muodostuu komponenteista, jotka jäsentävät sovelluksen rakennetta. Ohjelmoija vastaa ohjelmakoodin jakamisesta loogisiksi osiksi omiin komponentteihinsa. Yksi ruudulla näkyvä osa vastaa useimmiten yhtä komponenttia ohjelmakoodin puolella. Esimerkiksi tulo–meno-laskurissa yläreunan navigaatiopalkki on oma komponenttinsa. Komponentteihin perustuva ohjelmointitapa on omiaan vähentämään saman ohjelmakoodin toistuvuutta. Esimerkiksi verkkokauppasovelluksessa yhdestä tuotekortista voitaisiin tehdä komponentti. Sitten korttia voitaisiin käyttää yhä uudestaan useille eri tuotteille. Komponenteille voi välittää parametreja aivan kuten funktioille. React-terminologiassa komponentin parametreja kutsutaan propertyiksi. Propertyillä voidaan kustomoida komponetteja. Tuote-esimerkin tapauksessa tuotteen nimi ja tuotekuvaus voitaisiin viedä komponenttiin propertynä. Lisäksi propertyillä voidaan luoda dynaamisuutta: niiden arvoja voidaan muuttaa kesken ohjelman ajon tai ohjelman eri suoristuskertojen välillä.

Kuvassa 22 on esitetty ohjelmakoodin rivi, jossa otetaan käyttöön Navigation-niminen komponentti. Komponentille välitetään kaksi propertyä: showEntryType ja toggleShowEntryType. JavaScript tukee funktionaalista ohjelmointia, joten property voi olla myös funktio kuten esimerkin toggleShowEntryType-muuttuja. Propertyt määritetään

komponentin sisällä. Yhtäsuuruusmerkin vasemmalla puolella on propertyn nimi ja oikealla puolella arvo. Jos arvo on dynaaminen, se laitetaan aaltosulkeiden sisälle JSX-syntaksin mukaisesti.

```
167     <Navigation
168       showEntryType={this.state.showEntryType}
169       toggleShowEntryType={this.toggleShowEntryType}
170     />
```

Kuva 22. Navigation-komponentti vastaa sovelluksen navigaatiosta ja sillä on kaksi propertyä.

Jotta Navigation-komponenttia voi käyttää, täytyy se olla määritetty tiedostossa itsessään tai olla importoitu. Sovelluksessa pyrittiin modulaarisuuteen, joten komponentit määritettiin omissa tiedostoissaan ja importointiin. React-sovelluksessa import-lauseet on tapana laittaa heti tiedoston alkuun. Kuten kuvasta 23 näkyy, komponentit kirjoitetaan isolla alkukirjaimella. Tämä on tapa erottaa ne HTML-elementeistä. Pienellä kirjoitettuja komponenttinimiä ei hyväksytä.

```
5   import Navigation from './components/Navigation'
6   import AddEntryForm from './components/AddEntryForm'
7   import MonthSelector from './components/MonthSelector'
8   import IncomeExpenseSummary from './components/IncomeExpenseSummary'
9   import EntriesTable from './components/EntriesTable'
10  import ModifyEntryModal from './components/ModifyEntryModal'
```

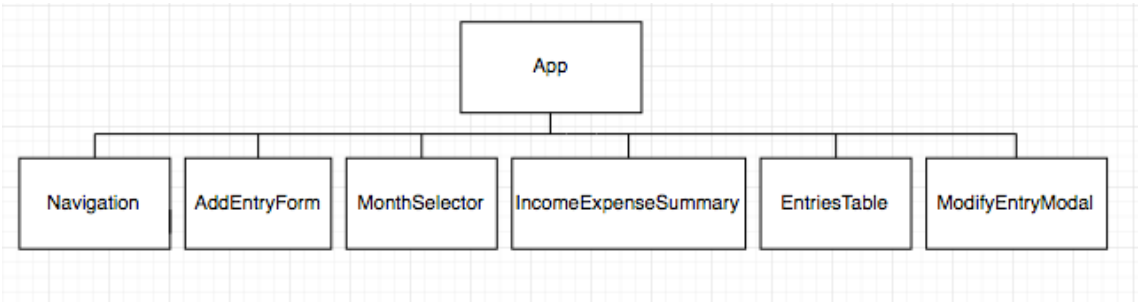
Kuva 23. Komponentin voi importoida toisesta tiedostosta import-lauseella.

Komponentin määrittävässä tiedostossa on oltava export-lause, jos komponenttia käytetään toisessa tiedostossa. Komento sijoitetaan yleensä määrittävän tiedoston loppuun. Kuvassa 24 on esitetty Navigation-komponentin export-lause.

```
11
12    export default Navigation
```

Kuva 24. Export-lause asettaa komponentin muiden tiedostojen saataville.

Tulo–meno-laskurissa kukin komponentti sijaitsee jonkun toisen komponentin sisällä. Toisin sanottuna kukin komponentti on jonkun toisen komponentin lapsikomponentti. Ainoa poikkeus tähän on pääkomponentti, joka ei ole minkään muun komponentin sisällä. Kuvassa 25 on esitetty sovelluksen kaksitasoinen komponenttihierarkia. App on pääkomponentti, jolla on kuusi lapsikomponenttia. Reactissa olisi mahdollista tehdä myös syvempiä komponenttihierarkioita, sillä myös lapsikomponentilla voi olla omia lapsikomponentteja.



Kuva 25. Sovelluksen pääkomponentilla on kuusi lapsikomponenttia.

Komponentteja on kahta tyyppiä: luokka- ja funktioperustaisia. Luokkaperustaisilla komponenteilla on yhteisten toimintojen lisäksi niin sanottu tila ja elinkaarimetodeja, joista on kerrottu edempänä. [23.] Sovelluksen pääkomponentti on luokka-komponentti. Muut komponentit ovat hieman yksinkertaisempia funktio-komponentteja, sillä niissä ei tarvita tilaa tai elinkaarimetodeja.

Komponentin tulostus

Komponentin tulostettava sisältö määritetään luokka-komponentissa render-metodin paluuarvossa ja funktio-komponentissa itse funktion paluuarvossa. Paluuarvo voi olla joko React-elementti tai esimerkiksi taulukko. Palautettava React-elementti saa sisältää lapsia, mutta sisarelementtejä ei saa olla. [23.]

Reactin dokumentaatio suosittelee, että render-metodissa ei muuteta komponentin tilaa ja palautetaan aina sama tulos kutsuajankohdasta riippumatta. Lisäksi suoraa vuorovaikutusta selaimen kanssa ei suositella. [23.]

Kuvassa 26 on laskurisovelluksen erään funktio-komponentin return-lause. Edellä mainitun mukaisesti komponentin paikalle lopulliseen ohjelmakoodiin sijoitetaan return-lauseessa määritetty div-elementti.

```
11    return(
12      <div className="advice-text income-expense-summary--text">
13        Tuloja ja menoja yhteensä:
14        {entries.reduce(getSum, 0) > 0 ? ' +' : ' '}
15        {entries.reduce(getSum, 0).toLocaleString()} €
16      </div>
17    )
```

Kuva 26. Funktio-komponentin paluuarvo tulostetaan komponentin merkkaamaan paikkaan.

React kutsuu komponentin render-metodia tai funktio-komponenttia automaattisesti. Ajo suoritetaan ensimmäisen kerran, kun komponentti tulostetaan. Lisäksi ajon aikana kutsu tehdään oletusarvoisesti uudestaan, mikäli komponentin property tai tila muuttuu. Täten elementti heijastaa propertyjensa ja tilansa nykyistä arvoa. ForceUpdate-käskyllä suorituksen voi kutsua manuaalisesti. Kyseiset käyttötilanteet ovat kuitenkin harvinaisia. [23.]

Komponentin tila

Luokka-komponentilla voi olla tila, joka koostuu yhdestä tai useammasta avain-arvoparista. Tilaolio eroaa tavallisista olioista: React reagoi sen muutokseen. Kun jokin olion kentän arvo muuttuu, osa sovelluksesta tai koko sovellus tulostetaan uudestaan. Ne osat, joiden sisältö muuttuu arvon mukana, päivitetään. React päättelee automaattisesti päivitettävät sisällöt. Ohjelmoijalle tila tarjoaa vaivattoman keinon luoda sovellukseen dynaamisuutta. On kuitenkin huomattava, että tilaa täytyy päivittää setState-metodin kautta, kuten kuvassa 27 esitetään. Metodin parametriksi riittää laittaa pelkästään muutettavat kentät, koska React lomittaa muutokset olemassa olevaan tilaan. Mikäli olion kenttää vaihtaa suoraan, automaattinen uudelleentulostus ei laukea.

```
36
37      this.setState({ showEntryType: 'income' })
38
```

Kuva 27. Komponentin tilaa muutetaan setState-metodilla.

Tila alustetaan usein luokan konstruktorissa. Konstruktoriin määritetään state-olio, jonka sisään tilan muodostavat avain-arvoparit sijoitetaan. Lisäksi konstruktorin ensimmäisellä rivillä täytyy kutsua yliluokan konstruktoria, jotta sen jälkeen käytettävä this-arvo toimii. Kuten kuvasta 28 nähdään, sovelluksen pääkomponentin tila muodostuu seitsemästä avain-arvoparista. Tilassa on neljää erityyppistä arvoa: merkkijono, taulukko, olio ja totuusarvo.

```
13  class App extends React.Component {
14    constructor(props) {
15      super(props)
16      this.state = {
17        showEntryType: 'expense',
18        viewPeriod: new Date(),
19        entries: [],
20        newEntryDate: '',
21        newEntryAmount: '',
22        entryToBeModified: {
23          date: '',
24          amount: ''
25        },
26        showModifyEntryModal: false
27      }
28      entryService
29        .getAll()
30        .then(entries => this.setState({entries}))
31    }
```

Kuva 28. Sovelluksen tila on keskitetty pääkomponenttiin. Tila alustetaan pääkomponentin konstruktorissa.

Koko sovelluksen tila on keskitetty pääkomponenttiin, vaikka alikomponenteillekin voi määrittää tilan. Suunnittelupäätöksen taustalla on se, että Reactissa tilan voi siirtää vain komponenttihierarkiassa alaspäin. Toisin sanottuna lapsikomponentissa määritettyä tilaa ei voi käyttää äitikomponentissa, mutta toisin päin se onnistuu. Tila voidaan välittää lapsikomponentille propertynä. Samoin äitikomponentin tilaa on mahdollista muokata lapsikomponentissa. Tämän saavuttamiseksi äitikomponenttiin luodaan funktio, joka

muuttaa tilaa. Funktio voidaan antaa propertynä lapsikomponentille ja ajaa siellä tarvittaessa. Kuvassa 29 näkyy äitikomponentin tilaa muuttava funktio, hideModifyEntryModal. Funktio suoritetaan, kun käyttäjä painaa elementin määrittämää painiketta.

```
28    <input
29      className="submit-button link"
30      type="submit"
31      value="Peruuta"
32      onClick={hideModifyEntryModal}
33    />
```

Kuva 29. HideModifyEntryModal on propertynä vastaanotettu funktio, jonka avulla äitikomponentin tilaa voidaan muuttaa.

Kun tila on keskitetty yhteen komponenttiin, muut komponentit yksinkertaistuvat. Ne voidaan toteuttaa tilattomina funktio-komponentteina.

Komponentin elinkaarimetodit

Kaikilla luokka-komponenteilla on niin sanotut elinkaarimetodit, joihin ohjelmoija voi kirjoittaa haluamansa toteutuksen. Nimi viittaa siihen, että kyseiset metodit suoritetaan aina tietyssä vaiheessa komponentin elinkaarta. Esimerkiksi aiemmin mainittu render-metodi on elinkaarimetodi, koska se suoritetaan silloin, kun komponentin sisältö tulostetaan.

Render-metodin lisäksi yleisiä elinkaarimetodeita on componentDidMount, componentDidUpdate ja componentWillUnmount. Ensiksi mainittu suoritetaan, kun komponentti on sijoitettu tietorakenteeseen. Toiseksi mainittu suoritetaan, kun komponentit sisältö on muuttunut tilamuutoksen tai propertyn muutoksen kautta. Kolmas suoritetaan ennen komponentin poistamista pysyvästi sovelluksen tietorakenteesta. Render-metodi on kuitenkin ainoa, jolle on pakko kirjoittaa toteutus. [23.]

Reactissa myös luokan konstruktoria ajatellaan elinkaarimetodina. Konstruktori suoritetaan ennen komponentin ensimmäistä tulostuskertaa. Konstruktoria käytetään tyypillisesti alustamaan komponentin tila. Myös laskurisovelluksen pääkomponentissa tehdään

näin, kuten aiemmasta kuvasta 28 käy ilmi. Pääkomponentin konstruktorissa haetaan lisäksi palvelimelta data, joka näytetään käyttäjälle. Konstruktorin sijaan data voitaisiin hakea teknisesti myös componentDidMount-elinkaarimetodissa. Ensiksi mainitussa tapauksessa komponentin suorituskyky on kuitenkin parempi, koska render-metodi suoritetaan vain kerran. [23.]

5.3 Palvelinpuoli

Palvelinpuoli toteutetaan rajapinta-ajattelun kautta. Toisin sanottuna palvelin tarjoaa rajapinnan, jonka avulla voidaan tarkastella, lisätä, muokata ja poistaa kirjauksia. Rajapintaa ei sidota tiukasti mihinkään tekniikkaan, vaan sen asiakkaana voi olla web-sovelluksen sijaan vaikkapa Android-sovellus. Yleiskäyttöisyys perustuu siihen, että data kulkee rajapinnan ja asiakkaan välillä JSON-muodossa, jota useimmat ohjelmointikielet tukevat. JSON:in eli JavaScript Object Notationin syntaksi on johdettu JavaScriptin olionotaatiosta, ja ne ovatkin syntaksiltaan hyvin samankaltaiset. [24.] Esimerkki JSON-muotoisesta tiedosta on esitetty kuvassa 30.

```
{
    "id": "5cd6de04731d23531821c8dd",
    "type": "expense",
    "amount": 30,
    "date": "2019-04-17T00:00:00.000Z"
}
```

Kuva 30. Data kulkee asiakkaan ja palvelimen välillä JSON-muodossa.

Node.js

Palvelinpuoli ohjelmoidaan JavaScript-pohjaisessa Node.js-ympäristössä, jossa ohjelmakoodi käännetään konekielelle ennen suoritusta [25]. JavaScriptiä ei siis tulkata reaaliajassa, mikä tekee Node.js-ympäristöstä nopeamman. Nopeutta tuo myös se, että

Node.js käyttää asynkronisia palvelupyyntöjä. Niiden kautta voidaan muun muassa käsitellä saapuvia pyyntöjä, vastata pyyntöihin ja käyttää tietokantaa. Kun jokin asynkroninen palvelupyyntö on suorituksessa, muutakin ohjelmakoodia voidaan suorittaa. Ominaisuuden ansiosta Node.js-pohjainen web-palvelin voi palvella tuhansia asiakasohjelmia kerralla, vaikka pääohjelma suoritetaan vain yhdessä säikeessä. Yleistäen voidaan sanoa, että ohjelmoija on vapautettu useista rinnakkaisohjelmoinnin haasteista, mutta suorituskyky on silti hyvä. [26.]

Express

Node.js:llä pystyy käsittelemään HTTP-protokollan kautta tulevia pyyntöjä, joten se sopii sellaisenaan web-palvelimeksi. Ohjelmointi kuitenkin helpottuu, kun käyttää jotain tarkoitukseen tehtyä ohjelmistokehystä. Sen vuoksi työssä on otettu käyttöön suosituin Node.js:ään perustuva web-ohjelmistokehys, Express. [27.]

Node.js perustuu tapahtumien käsittelyyn, ja myös Express-ohjelmistokehyksen käyttö perustuu pitkälti tapahtumankäsittelijöiden käyttöön. Tapahtumankäsittelijälle määritetään seurattava HTTP-pyyntötyyppi ja URL. Mikäli saapuvan pyynnön tiedot täsmäävät määritykseen, suoritus siirtyy tapahtumankäsittelijän ohjelmalohkoon. Laskurisovelluksessa React-ohjelmakoodi lähettää HTTP-pyyntöjä, joihin Express-ohjelmakoodi reagoi.

HTTP-pyyntötyyppejä on tietty ennalta määritetty joukko. Pyyntötyyppi kertoo, mitä asiakas pyytää vastaanottajan tekevän. Taulukossa 1 on esitelty yleisimpiä pyyntötyyppejä ja niiden tarkoituksia. [28.]

Taulukko 1. Yleisimpiä HTTP-pyyntötyyppejä ja niiden tarkoituksia [28].

HTTP-pyyntö-tyyppi	**Tarkoitus**
GET	saada tietoa
POST	luoda tietoa
PUT	päivittää tieto kokonaisuudessaan

PATCH	päivittää tieto osittain
DELETE	poistaa tieto
OPTIONS	pyytää palvelimen asetukset mukaan luettuna pyynnölle kohdistetut vaatimukset

Pyyntötyypit ja niihin liittyvät tarkoitukset perustuvat HTTP 1.1 -spesifikaatioon. On hyvä alleviivata, että käytännön toteutuksesta vastaa ohjelmoija. Esimerkiksi Express ei sisällä minkäänlaista mekanismia, joka estäisi tietyn pyyntötyypin käytön spesifikaatiosta poikkeavalla tavalla. Spesifikaatiota on kuitenkin hyvä seurata, sillä se helpottaa muiden ymmärrystä ohjelmakoodista. Toinen ohjelmoija tietää heti nähdessään esimerkiksi GET-pyynnön, että kyseisen pyynnön tarkoitus saada tietoa idempotentisti ja turvallisesti. Idempotentti tarkoittaa sitä, että sama kysely tuottaa saman lopputuloksen vastaanottajassa suoritettiinpa kysely kuinka monta kertaa tahansa, jos vastaanottajan tila ei ole muuttunut jostain muusta syystä kyselyiden välissä. Turvallinen tarkoittaa sitä, että kysely ei muuta palvelimella olevaa tietosisältöä. [29, s. 42.]

Expressissä saapuviin HTTP-pyyntöihin voi määritellä tapahtumankäsittelijöitä, kun on ottanut käyttöön joko express- tai router-olion express-moduulista. Kuvan 31 mukaisesti laskurisovelluksessa käytetään router-oliota.

```
1
2   const entriesRouter = require('express').Router()
3
```

Kuva 31. Router-olion avulla voi määritellä saapuviin HTTP-pyyntöihin tapahtumankäsittelijöitä.

Express- ja router-olioilla on HTTP-pyyntötyyppien mukaan nimettyjä metodeja. Metodin ensimmäiseksi parametriksi tulee URL ja toiseksi tapahtumankäsittelijä. Tapahtumankäsittelijä suoritetaan, kun pyyntötyyppi ja URL vastaavat saapuneen pyynnön vastaavia tietoja. Kuvassa 32 tapahtumankäsittelijä on sijoitettu suoraan paikalleen ES2015-standardin mukaisen nuolifunktion avulla. Tapahtumankäsittelijä saa kaksi parametria. Ensimmäinen parametri sisältää saapuneen kyselyn ja toisen parametrin avulla lähetetään vastaus asiakkaalle.

```
5
6   entriesRouter.get('/', (req,res) => {
7     // tapahtumankäsittelijän sisältö
```

Kuva 32. Kuvassa on määritetty tapahtumankäsittelijä GET-tyyppiselle pyynnölle, joka kohdistetaan palvelimen osoitteeseen /.

Tulo–meno-laskurissa vastaus-parametriin viitataan res-muuttujalla. Vastaukset halutaan lähettää JSON-muodossa, joten käytetään res-muuttujan json-metodia. Tällöin Express muuttaa parametrina annetun datan JSON-muotoon ja lähettää sen vastauksena. Lisäksi vastauksen otsake Content-type muutetaan muotoon application/json. Kuvassa 33 muuttuja entriesFormalized on JavaScript-taulukko, jonka sisällä on olioita.

```
42
43        res.json(entriesFormalized)
44
```

Kuva 33. JSON-muotoisen vastauksen voi lähettää Expressin vastausolion json-metodilla.

Vastausten tilakoodit

Kuhunkin HTTP-protokollan kautta lähetettyyn vastaukseen liittyy tilakoodi. Sen tehtävänä on antaa yksiselitteistä tietoa, onko pyyntö suoritettu onnistuneesti. Taulukossa 2 on esitetty yleisimpiä tilakoodeja ja niiden selitteitä. [30.]

Taulukko 2. HTTP-vastausten tilakoodeja ja selitteitä [30].

Tilakoodi	**Selite**
200	Pyyntö on käsitelty onnistuneesti.
201	Pyyntö on käsitelty onnistuneesti, ja uusi resurssi on luotu.
204	Pyyntö on käsitelty onnistuneesti, mutta vastaussisältö on jätetty tarkoituksella tyhjäksi.
400	Pyyntö ei ole oikein muotoiltu.
401	Pyynnön suoritukseen liittyvä valtuutus epäonnistui.

403	Pyynnön suoritus on kielletty.
404	Pyynnön kohdetta ei löydy.
500	Pyynnön käsittely keskeytyi palvelimen virheeseen.

Kuten taulukon arvoista voi nähdä, vastaukset on ryhmitelty eri ryhmiin ensimmäisen numeron perusteella. Esimerkiksi kaikki numerolla kaksi alkavat tilakoodit ilmaisevat onnistunutta pyynnön käsittelyä. Numerolla neljä alkavat tilakoodit kertovat pyynnön käsittelyn epäonnistuneen asiakkaasta johtuen ja numerolla viisi palvelimesta johtuen. [30.]

Expressissä oletustilakoodi vastauksille on 200, ja sitä ei tarvitse eksplisiittisesti määrittää. Mikäli halutaan käyttää jotain toista tilakoodia, se määritetään vastausolion statusmetodilla ennen vastauksen lähettämistä. Kuvassa 34 tilakoodi on vaihdettu 400:ksi.

```
203
204   res.status(400).json({ error: 'Entry id is malformatted' })
205
```

Kuva 34. Vastauksen tilakoodia voi muuttaa vastausolion statusmetodilla.

Vastauksen tilakoodi ja sisältö määräytyvät laskurisovelluksessa yksinkertaisten if-lauseiden avulla. Jos palautettavaa on, palautetaan tieto tilakoodilla 200 tai 201. Mikäli palautettavaa ei ole, annetaan tilanteeseen sopiva numero neljällä alkava tilakoodi sekä JSON-muotoinen virheilmoitus.

Kuvassa 35 on kirjauksen poistoon liittyvää ohjelmakoodia. Mikäli on löydetty poistoehtoon sopiva kirjaus, se on poistettu sovelluksen tietokannasta ja lisätty muuttujaan removedEntry. Mikäli ehtoon sopivaa kirjausta ei ole, muuttujan arvo on undefined. Käytetään hyväksi JavaScriptin ominaisuutta, että olioita käsitellään kuten totuusarvoa true ja undefined-arvoa kuten totuusarvoa false. Täten onnistunut poisto johtaa vastaukseen tilakoodilla 200 eli ensimmäiseen vaihtoehtoon. Epäonnistunut poisto johtaa vastaukseen tilakoodilla 404, joka tarkoittaa, ettei pyynnön kohdetta löytynyt.

```
194   if (removedEntry) {
195     res.json(Entry.formalize(removedEntry))
196   } else {
197     res.status(404).json({ error: 'Entry with specified id is not found' })
198   }
```

Kuva 35. Ylemmässä ehtolauseen haarassa lähetetään vastaus oletustilakoodilla 200 ja alemmassa virhetilakoodilla 404.

Expressin väliohjelmat

Express voi käsitellä saapuneita pyyntöjä niin sanottujen väliohjelmien avulla. Väliohjelman käyttö lisää modulaarisuutta. Väliohjelmalle välitetään pyyntö- ja vastausolio, joiden avulla se suorittaa oman tehtävänsä. Tehtävän suorituksen jälkeen kontrolli siirtyy yleensä seuraavaan väliohjelmaan niin kauan kuin väliohjelmia on jonossa. Kontrollin siirrosta vastaa next-funktio, jota kutsutaan väliohjelmassa viimeiseksi. Mikäli next-funktiota ei kuitenkaan kutsuta, pyynnön käsittely päättyy. Mitä ylempänä väliohjelma on otettu käyttöön, sen aikaisemmin se tulee suoritukseen. Käyttöönotto tehdään use-funktiolla, kuten kuvassa 36. Väliohjelma annetaan funktion parametrina.

```
23
24  app.use(cors())
25  app.use(bodyParser.json())
```

Kuva 36. Väliohjelma otetaan käyttöön use-funktiolla.

Sovelluksessa käytettävä cors-väliohjelma vastaa siitä, että palvelimeen nähden ulkoisiin pyyntöihin vastataan. Turvallisuussyistä niin ei oletusarvoisesti tehdä. BodyParser-väliohjelma muuntaa kyselyn mukana tullutta body-dataa helpommin käsiteltävään muotoon. Tässä yhteydessä määritetään vielä, että halutaan data JSON-muotoiseksi.

Edellä esitetyt väliohjelmat on otettu käyttöön valmiina moduulina, mutta väliohjelmia voi ohjelmoida myös itse. Laskurisovelluksessa saapuvien pyyntöjen käsittely tehdään itse ohjelmoidulla väliohjelmalla, jotta sovellus olisi mahdollisimman modulaarinen. Kuvan 37 mukaisesti väliohjelma otetaan käyttöön kaksiparametrisella use-metodilla. Ensimmäi-

nen parametri on kaikille pyynnöille yhteinen URL-osuus ja toinen parametri on varsinainen väliohjelma. Ensimmäinen parametri tekee sen, että väliohjelman ohjelmakoodissa URL-osuuden alkua ei tarvitse enää määrittää.

```
27
28    app.use('/api/entries', entriesRouter)
29
```

Kuva 37. EntriesRouter on itse toteutettu väliohjelma.

Tiedoston ulkopuolella määritetyt väliohjelmat importoidaan tiedostoon samalla tavalla kuin kaikki muutkin moduulit. ES2015:n importointi-ilmausta ei tueta Node.js-ympäristössä, joten käytetään vanhempaa require-syntaksia, joka on esitetty kuvassa 38. Väliohjelmat on importoitu kuvan riveillä kaksi, neljä ja viisi.

```
1    const express = require('express')
2    const entriesRouter = require('./controllers/entries')
3    const mongoose = require('mongoose')
4    const cors = require('cors')
5    const bodyParser = require('body-parser')
```

Kuva 38. Väliohjelma täytyy importoida, mikäli se on määritetty toisessa tiedostossa.

Itse tehty väliohjelma entriesRouter täytyy lisäksi olla laitettu muiden tiedostojen saataville. Tämä on tehty entriesRouterin määrittävässä tiedostossa module-olion avulla. Kuvan 39 mukaisesti eksportoitava väliohjelma laitetaan sijoitusoperaattorin oikealle puolelle. Eksportoinnin komento on hyvin samankaltainen kuin ES2015:sta export-lause, mutta vanhempaa alkuperää. Kuten importoinnin tapauksessa, Node.js ei tue uudempaa syntaksia. [27.]

```
101
102    module.exports = entriesRouter
```

Kuva 39. Itse tehty väliohjelma voidaan laittaa muiden tiedostojen saataville module-olion avulla.

5.4 Tietokanta

Sovelluksessa käytetään tietokantaohjelmaa datan pysyvään varastointiin. Toinen vaihtoehto olisi ollut tallettaa data suoraan esimerkiksi tekstitiedostoon. Tietokannan käyttö tuo kuitenkin etuja. Ohjelmoijan ei tarvitse toteuttaa itse datan varastointiin liittyvää ohjelmakoodia, vaan voidaan käyttää tietokantaohjelman tarjoamaa rajapintaa datan tarkasteluun ja hallinnointiin. Käytetty rajapinta tarjoaa lisäksi mahdollisuuden sijoittaa data eri Internet-sijaintiin kuin muu ohjelmakoodi, mitä laskurisovelluksessa on hyödynnetty.

5.4.1 MongoDB

Sovelluksessa käytetään MongoDB-nimistä tietokantaa, joka on niin sanottu dokumenttitietokanta. Keskeinen ero MongoDB:n ja perinteisemmän relaatiotietokannan välillä on se, että MongoDB:ssä data on tallennettu BSON-muodossa ja esitetään käyttäjälle JSON-muodossa [31]. Datan esitysmuoto on siis hyvin samankaltainen tietokannassa ja ohjelmakoodissa. Lisäksi MongoDB:n ja relatiotietokantojen terminologia poikkeaa toisistaan: esimerkiksi taulun sijaan puhutaan kokoelmasta, sarakkeen sijaan puhutaan kentästä ja rivin sijaan puhutaan dokumentista [32].

MongoDB ei aseta rajoitteita datan muodolle: dokumenteilla voi olla vaihteleva määrä kenttiä ja kentän datatyyppiä ei ole sidottu. Vastuun muodon oikeellisuudesta on tietokantaa käyttävällä ohjelmalla. Ominaisuus saattaa lisätä jossain tapauksissa ohjelmavirheiden mahdollisuutta, mutta etujakin on. Suorituskyky on parempi, kun tietokannan ei tarvitse tehdä muototarkastuksia. Vapaamuotoisesta rakenteesta on myös se hyöty, että sovelluksia voidaan kehittää joustavammin. Sovelluslogiikan muutos riittää eli itse tietokannan määritykseen ei tarvitse koskea. [33.]

5.4.2 Mongoose

MongoDB:n käyttö on suhteellisen työlästä MongoDB:n oman Node.js-ajurin avulla [27]. Otetaan sen vuoksi käyttöön korkeamman tason kirjasto, Mongoose.

Mongoose on olio–dokumentti-muunnin. Sen avulla palvelimella voidaan käsitellä tietokantaan meneviä ja sieltä tulevia dokumentteja tavallisten JavaScript-olioiden tapaan. Lisäksi Mongoose tarkastaa, että tieto vastaa haluttua muotoa ennen tietokantaoperaatioiden suorittamista. Muita Mongoosen tarjoamia toimintoja ovat muun muassa tiedon validointi ja ohjelmoijalle helpompi tietokantakysely-syntaksi.

Skeema

Mongoosea käytettäessä tietokantaan tallennettavalle tiedolle määritellään skeema [27]. Skeema määrittää ohjelmassa luotavaa oliota, ja tätä kautta tietokantaan päätyvää dokumenttia. Skeema tuo dokumenteille muodon. Muoto tehdään ohjelmakoodin puolella, joten tietokannan suorituskyky ei laske ja kehitys on ketterää.

Skeema luodaan Schema-luokan avulla kuvan 40 esittämällä tavalla. Määrittelyssä käytetään olio-notaatiota, jossa avaimena on kentän nimi ja arvona tietotyyppi tai sisäolio. Sisäoliota käytetään, jos halutaan tietotyypin lisäksi määrittää kentälle validaatio.

```
3   const EntrySchema = new mongoose.Schema({
4     type: {
5       type: String,
6       enum: ['income', 'expense'],
7       required: true
8     },
9     date: {
10      type: Date,
11      required: true
12    },
13    amount: {
14      type: Number,
15      min: 0.01,
16      max: 99999999,
17      required: true
18    }
19  })
```

Kuva 40. Mongoosen skeema luodaan Schema-luokan avulla.

Validaatio

Mongoosen avulla voi validoida kentän olemassaolon. Tämä tehdään määrityksellä required: true (ks. kuva 40). Lisäksi numeroille ja päivämäärille on saatavilla valmiina validaatio minimi- ja maksimiarvosta. Merkkijonolle on saatavilla esimerkiksi validaattori, joka tarkastaa, onko merkkijono joku ennalta määrätyistä. Myös säännöllistä lauseketta vastaan validointi on mahdollista. Jos valmiista validaattoreista ei löydy sopivaa vaihtoehtoa, sen voi ohjelmoida itse. [34.]

Validaatio suoritetaan paikallisesti ennen tietokantaoperaation tekemistä. Täten yksi kyselykierros Internetin yli säästyy tapauksissa, joissa tietokanta on eri Internet-sijainnissa ja data on epävalidia. Tämä on tehokasta, sillä yhteyden muodostus on yksi eniten aikaa vievistä osioista sovelluksen toiminnassa. Samalla tietokannan kuormitus laskee, mikä on merkityksellistä suuren kokoluokan sovelluksissa.

Skeeman staattiset funktiot

Skeemaan voi määritellä staattisia funktioita toiminnoille, jotka liittyvät skeemaan kiinteästi. Laskurisovelluksessa käytetään yhtä staattista funktiota. Funktio muuttaa parametrina saamansa olion _id-kentän id-nimiseksi. Staattisen funktion käyttöön päädyttiin, sillä tietokanta antaa tunnisteen automaattisesti _id-kentässä, mutta asiakaspuoli olettaa tunnisteen olevan id-kentässä. Funktiota käytetään aina ennen kuin tunnisteita sisältävä vastaus lähetään palvelinpuolelta asiakaspuolelle. Staattiseen funktioon päästää käsiksi skeeman pohjalta luodun malli-luokan Entry avulla kuten kuvassa 41.

```
52    newEntry
53      .save()
54      .then(savedEntry => {
55        res.status(201).json(Entry.formalize(savedEntry))
56      })
```

Kuva 41. Formalize on skeemaan liitetty staattinen funktio.

Malli

Malli luodaan skeeman avulla. Skeeman ja mallin tarkoitukset voi olla aluksi vaikea erottaa toisistaan. Pääero on kuitenkin se, että mallin kautta suoritetaan tietokantaoperaatioita, kun skeema kuvaa tiedon muotoa. [35.]

Malli luodaan skeemasta model-funktiolla, jossa ensimmäinen parametri on mallin nimi ja toinen skeema, jonka mukaan malli luodaan (kuva 42).

```
29
30    const Entry = mongoose.model('Entry', EntrySchema)
31
```

Kuva 42. Malli luodaan skeemasta model-funktiolla.

Sovelluksen tietokantaoperaatiot

Tietokantaa voi hallita staattisilla funktioilla, jotka Mongoose tarjoaa mallin kautta. Funktioiden avulla voidaan tehdä CRUD-operaatiosta dokumenttien luku, päivitys ja poisto. Dokumenttien luonti tehdään malli-luokan olion avulla. Tietokantafunktiot ovat asynkronisia, ja niille voi antaa takaisinkutsufunktion parametrina. Vaihtoehtoisesti niitä voi käyttää promise-tyylisesti, kuten laskurisovelluksessa on tehty. [36.]

Dokumentteja voi luoda save-metodilla, kuten aiemmasta kuvasta 41 ilmenee. Kuvassa tallennettava newEntry-olio on luotu Entry-mallilla.

Dokumentit haetaan aggregate-funktiolla, johon voidaan sijoittaa MongoDB:n tarjoamia tietokantaoperaatioita taulukkomuodossa (kuva 43). Tämä on tarpeen, sillä Mongoosen omat tietokantafunktiot eivät tue haluttua toimintoa: luodaan haettuihin dokumentteihin kaksi ylimääräistä kenttää. Year-kenttään sijoitetaan vuosi dokumentin date-kentästä ja month-kenttään sijoitetaan kuukausi dokumentin date-kentästä. Jos omia kenttiä ei tarvitse luoda, Mongoosen find-funktio on lyhyempi vaihtoehto.

```
Entry
    .aggregate([
        {
            $addFields: {
                'year': { $year: '$date' },
                'month': { $month: '$date' }
            }
        },
        {
            $match: {
                year: searchedYear,
                month: searchedMonth
            }
        },
        {
            $sort: { date: 1 }
        }
    ])
    .then(entries => {
```

Kuva 43. Aggregate-funktion kautta voidaan käyttää MongoDB:n tietokantaoperaatioita.

Kun kyselyn vastaukseen on luotu kaksi omaa kenttää, niiden avulla voidaan suodattaa dokumentteja. Suodatus on tehty MongoDB:n match-operaatiolla. Match-operaatiolla valitaan ne dokumentit, joissa on haluttu vuosi ja kuukausi. Viimeiseksi ennen kyselyn suoritusta pyydetään tietokantaa järjestämään dokumentit date-kentän perusteella. Ilmaisu $sort: { date: 1 } tekee sen, että dokumentit järjestetään kronologiseen aikajärjestykseen.

Päivitys on tehty Mongoosen findByIdAndUpdate-funktiolla (kuva 44). Ensimmäiseksi parametriksi sijoitetaan päivitettävän dokumentin id, toiseksi muutettavat kentät oliomuodossa ja kolmanneksi asetukset. Asetuksissa validaatio täytyy kytkeä erikseen päälle. Oletusarvoisesti validaatio ei ole päällä päivitysoperaatioissa, jotta funktio säilyy taakse päin yhteensopivana. Ennen versiota 4.0 Mongoose ei tukenut validaatiota päivitysoperaatiossa. Jälkimmäinen asetus, New: true tekee sen, että tietokanta palauttaa dokumentin päivitetyillä tiedoilla. Oletusarvoisesti palautettava dokumentti on vanhoilla tiedoilla.

```
169     Entry
170       .findByIdAndUpdate(
171         body.id,
172         modifiedEntry,
173         { runValidators: true, new: true }
174       )
175       .then(updatedNote => {
```

Kuva 44. Tietokannan dokumentin voi päivittää findByIdAndUpdate-funktiolla.

Poisto on tehty Mongoosen findByIdAndDelete-funktiolla (kuva 45). Parametriksi annetaan poistettavan dokumentin tunniste.

```
87      Entry
88        .findByIdAndDelete(id)
89        .then(removedEntry => {
```

Kuva 45. Dokumentin voi poistaa findByIdAndDelete-funktiolla.

5.5 CSS-määritteet

Sovellukseen lisättiin CSS-määrittelyt lopuksi. Tavoitteena oli toteuttaa ulkoasu rautalankamallien pohjalta.

CSS-vaiheessa tehtiin joitain muutoksia rautalankamalleihin verrattuna, mutta pääpiirteissään suunnitelmassa pysyttiin. Esimerkiksi lisää-painikeen ulkoasua muutettiin. Mielestäni näytti visuaalisesti yhdenmukaisemmalta, kun painikkeen tyylitteli samalla tavalla muiden painettavien elementtien kanssa. Lisäksi visuaalisen erottuvuuden parantamiseksi eri päivien kirjausrivit erotettiin toisistaan isommalla välillä kuin saman päivän kirjausrivit.

6 Työn tulokset ja jatkokehitysideat

Työn tulokseksi saatiin perustoiminnot sisältävä tulo–meno-laskuri. Käyttäjä voi tarkastella, lisätä, poistaa ja muokata tuloja sekä menoja. Lisäksi käyttäjä näkee suunnitelmien mukaisesti uusia yhteenvetotietoja kuukauden kirjauksista (kuukauden meno- ja tulokirjausten määrä sekä kuukauden tulojen ja menojen yhteissumma).

Osa suunnitelluista ominaisuuksista jäi kuitenkin jatkokehittelyyn. Esimerkiksi kirjaukseen ei voi tallettaa tyypin, päivämäärän ja summan lisäksi muita tietoja. Lisäksi yhteenvetonäkymiä, ilmoituksia tai kirjauksen poiston varmistavaa dialogi-ikkunaa ei ole. Kattava lista toteutetuista ja jatkokehittelyyn jätetyistä ominaisuuksista on esitetty taulukossa 3. Taulukon numerot viittaavat vaatimusmäärittelyssä (ks. liite 1) esitettyihin kohtiin. Ohjelmakoodin rakenne pyrittiin tekemään modulaariseksi, mikä helpottaa jatkokehittelyyn jätettyjen sekä uusien ominaisuuksien toteuttamista.

Taulukko 3. Toteutetut ja jatkokehittelyyn jätetyt ominaisuudet.

Toteutetut ominaisuudet	**Jatkokehittelyyn jätetyt ominaisuudet**
1ab, 2a, 3, 4acd, 7cde, 8, 9	1c, 2b, 4befgh, 5, 6, 7ab, 10, 11, 12

Käytettävyyteen liittyvät suunnitelmat saatiin suurelta osin toteutettua. Sovelluksen käyttöliittymästä tuli hyvin rautalankamallien kaltainen. Poikkeuksen tästä muodostaa kirjausrivin muokkaa- ja poista-painike. Ne toteutettiin aina näkyvinä, koska tässä vaiheessa kehitystä kirjausrivillä on runsaasti tilaa. Mielestäni suunnitelmassa esitetyt käytettävyysratkaisut toimivat käytännössä ja sovelluksen käytettävyys on hyvä. Sovelluksen käyttöliittymä on kuvassa 46.

Kuva 46. Sovelluksen käyttöliittymä.

Sovelluksen kaikki toiminnot ovat nopeita. Esimerkiksi sovelluksen käynnistys selaimessa kesti testatessa alle sekunnin. Yhden sivun sovellus -periaate näyttäisi odotusten mukaisesti nopeuttavan sovelluksen toimintaa. Seuraavassa on esitetty dynaamisten toimintojen vaatimia aikoja. Uuden kuukauden tiedot sai palvelimelta 0,07–0,12 sekunnissa. Kirjauksen lisäys, muokkaus ja poisto kestivät suunnilleen saman verran, joten kaikki dynaamisesti haettu data on sovelluksen käytössä käytännössä välittömästi.

Yksi myöhemmin toteutettava ominaisuus voisi olla käyttäjähallinta. Täten useampi käyttäjä voi käyttää sovellusta. Samassa yhteydessä olisi mahdollista parantaa tietoturvaa kirjautumistoiminnolla: käyttäjän tulisi antaa käyttäjätunnus ja salasana ennen kuin tiedot näytetään.

Useamman kirjauksen syöttäminen kerrallaan järjestelmään esimerkiksi CSV-muodossa on toinen jatkokehityskohde. Useimmat verkkopankit tarjoavat mahdollisuuden ladata ti-

litapahtumatietoa sivuiltaan. Täten tilitapahtumat saisi siirrettyä kätevästi tulo–meno-laskuriin manuaalisen syöttämisen sijaan. Toisaalta datan voisi tarjota ladattavaksi myös sovelluksesta käyttäjälle. Tämä antaisi mahdollisuuden siirtää tietoa järjestelmästä ulos muissa ohjelmissa käsiteltäväksi. Ladattu tiedosto on samalla varmuuskopio.

7 Yhteenveto

Insinöörityön tavoitteena oli suunnitella ja toteuttaa käyttäjäystävällinen, tekijän henkilökohtaiseen käyttöön soveltuva tulo–meno-laskuri. Aiemmin samaan tarkoitukseen oli käytetty Excel-tiedostoja.

Työ aloitettiin tutustumalla vanhaan Excel-pohjaiseen järjestelmään ja sen ominaisuuksiin. Tätä kautta muodostettiin uudelle sovellukselle vaatimusmäärittely, johon kirjattiin tärkeitä ominaisuuksia vanhasta järjestelmästä sekä joitain uusia ominaisuuksia. Uudessa sovelluksessa painopisteeksi valittiin käytettävyys. Vaatimusmäärittelyn pohjalta toteutettiin rautalankamalli, joista ilmenee sovelluksen sisällöt ja rakenne. Malli esiteltiin käytettävyyden näkökulmasta.

Sovellus päätettiin tehdä JavaScript-pohjaisilla tekniikoilla. Etuna valinnasta oli muun muassa suhteellisen nopea kehitysvaihe.

Varsinainen ohjelmointi aloitettiin asiakaspuolesta, joka toteutettiin React-ohjelmistokehyksellä. React toi etuna muun muassa sen, että dynaamisuutta oli helppo luoda. Ohjelmoijan tehtäväksi jäi lähinnä luoda logiikka tilan muutokselle. React päivittää ajon aikana automaattisesti tilaolion kentästä riippuvaiset sovelluksen osat, kun kentän arvo muuttuu. Lisäksi Reactin avulla ohjelmakoodin voi jakaa erillisiin komponentteihin, jonka avulla laskurisovellukseen saatiin modulaarisuutta.

Palvelinpuoli toteutettiin Node.js-ympäristössä ja ympäristöön perustuvalla Express-ohjelmistokehyksellä. Node.js osoittautui nopeaksi ainakin tulo–meno-laskurin kaltaisen pienen sovelluksen tapauksessa. Express-kehys helpotti tapahtumankäsittelijöiden kytkemistä palvelimelle saapuviin HTTP-pyyntöihin sekä pyyntöjen käsittelyä. Tietokannaksi otettiin käyttöön MongoDB, johon data tallennetaan niin sanottuina dokumentteina. Dokumentit ovat muodoltaan hyvin samankaltaisia ohjelmakoodissa käsiteltyjen JavaScript-olioiden kanssa, mikä helpottaa ohjelmakoodin olioiden ja tietokannan dokumenttien välisen vastaavuuden hahmottamista. Tietokantaa käsitellään ohjelmakoodin puolella Mongoose-mallintajan kautta. Mallintaja virtaviivaisti tietokantakommunikaatiota muun muassa lyhyempien komentojen kautta. Lisäksi sen avulla luotiin skeema, joka

määrittää muodon tallennettavalle datalle. Täten tallennettavan datan oikeasta muodosta voidaan olla varmoja, vaikka MongoDB ei itsessään tarkista datan muotoa.

Insinöörityössä saatiin aikaan perustoiminnot sisältävä sovellus, jonka tekijä voi ottaa käyttöönsä. Lisäksi tekijän mielestä käytettävyystavoitteisiin päästiin. Sovelluksen modulaarinen rakenne helpottaa jatkokehitykseen jätettyjen sekä uusien ominaisuuksilla toteuttamista.

Lähteet

1 Sinkkonen, Irmeli; Kuoppala, Hannu; Parkkinen, Jarmo & Vastamäki, Raino. 2006. Käytettävyyden psykologia. Helsinki: Edita Publishing Oy.

2 Natoli, Joe. Interface Design. Verkkoaineisto. <https://www.udemy.com/user-experience-design-fundamentals/learn/v4/t/lecture/325003?start=309>. Päivitetty toukokuu 2017. Luettu 26.1.2019.

3 Nielsen, Jakob. 1994. 10 Usability Heuristics for User Interface Design. Verkkoaineisto. <https://www.nngroup.com/articles/ten-usability-heuristics/>. 24.4.1994. Luettu 26.1.2019.

4 Harley, Aurora. 2014. Icon Usability. Verkkoaineisto. <https://www.nngroup.com/articles/icon-usability/>. 27.7.2014. Luettu 26.1.2019.

5 Budiu, Raluca. 2014. Memory Recognition and Recall in User Interfaces. Verkkoaineisto. <https://www.nngroup.com/articles/recognition-and-recall/>. 6.7.2014. Luettu 28.1.2019.

6 Babich, Nick. 2016. Tooltips in UI Design. Verkkoaineisto. <https://uxplanet.org/tooltips-in-ui-design-f63e117aa3d1>. 9.11.2016. Luettu 26.1.2019.

7 Required versus optional fields – a new standard?. 2015. Verkkoaineisto. Formulate Information Design. <https://www.formulate.com.au/blog/required-versus-optional-fields-new-standard/>. 16.4.2015. Luettu 26.1.2019.

8 Harley, Aurora. 2018. Visibility of System Status. Verkkoaineisto. <https://www.nngroup.com/articles/visibility-system-status/>. 3.6.2018. Luettu 25.1.2019.

9 Nielsen, Jakob. 1999. Do Interface Standards Stifle Design Creativity?. Verkkoaineisto. <https://www.nngroup.com/articles/do-interface-standards-stifle-design-creativity/>. 22.10.1999. Luettu 26.1.2019.

10 Table (database). Verkkoaineisto. Wikipedia. <https://en.wikipedia.org/wiki/Table_(database)>. Luettu 26.1.2019.

11 Natoli, Joe. Defining the Skeleton. Verkkoaineisto. <https://www.udemy.com/user-experience-design-fundamentals/learn/v4/t/lecture/325002?start=260>. Päivitetty toukokuu 2017. Luettu 26.1.2019.

12 Engelschall, Ralf. ECMAScript 6 - New Features: Overview & Comparison. Verkkoaineisto. <http://es6-features.org>. Luettu 28.4.2019.

13 JavaScript Let. Verkkoaineisto. Refsnes Data. <https://www.w3schools.com/js/js_let.asp>. Luettu 28.4.2019.

14 let. Verkkoaineisto. Mozilla. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let#Redeclarations>. Luettu 28.4.2019.

15 Arrow functions. Verkkoaineisto. Mozilla. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions#No_separate_this>. Luettu 28.4.2019.

16 Tuura, Tommi. Moniajo ja asynkroniset kutsut JavaScriptissä. Verkkoaineisto. <https://www.cs.helsinki.fi/u/ttuura/otk-js/asynkronisuus.html>. Päivitetty 29.1.2019. Luettu 16.5.2019.

17 Olson, Peter. 2019. Introduction to Asynchronous JavaScript. Verkkoaineisto. <https://www.pluralsight.com/guides/introduction-to-asynchronous-javascript>. 17.4.2019. Luettu 16.5.2019.

18 Ogden, Max. Callback hell. Verkkoaineisto. <http://callbackhell.com/>. Päivitetty 4.2.2016. Luettu 12.5.2019.

19 Promise. Verkkoaineisto. Mozilla. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise>. Luettu 28.4.2019.

20 Single-page application vs. multiple-page application. 2016. Verkkoaineisto. Neoteric. <https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58>. 2.11.2016. Luettu 28.4.2019.

21 Introducing JSX. Verkkoaineisto. Facebook. <https://reactjs.org/docs/introducing-jsx.html>. Luettu 9.4.2019.

22 React Without JSX. Verkkoaineisto. Facebook. <https://reactjs.org/docs/react-without-jsx.html>. Luettu 9.4.2019.

23 React.Component. Verkkoaineisto. Facebook. <https://reactjs.org/docs/react-component.html>. Luettu 30.4.2019.

24 JSON. Verkkoaineisto. Wikipedia. <https://en.wikipedia.org/wiki/JSON>. Luettu 12.5.2019.

25 Node.js. Verkkoaineisto. Wikipedia. <https://en.wikipedia.org/wiki/Node.js#V8>. Luettu 28.4.2019.

26 Cantelon, Mike; Meck, Bradley & Young, Alex. 2017. Node.js in Action. E-kirja. Manning Publications.

27 Luukkainen, Matti. osa 3. Verkkoaineisto. <https://fullstackopen.github.io/osa3>. Luettu 28.4.2019.

28 HTTP request methods. Verkkoaineisto. Mozilla. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods>. Luettu 28. 4.2019.

29 De, Brajesh. 2017. API Management. E-kirja. Apress.

30 HTTP response status codes. Verkkoaineisto. Mozilla. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>. Luettu 28.4.2019.

31 Glossary. Verkkoaineisto. Mongo DB, Inc. <https://docs.mongodb.com/manual/reference/glossary/>. Luettu 30.4.2019.

32 MongoDB and MySQL Compared. Verkkoaineisto. MongoDB, Inc. <https://www.mongodb.com/compare/mongodb-mysql>. Luettu 30.4.2019.

33 Hellas, Arto & Luukkainen, Matti. Osa 7. Verkkoaineisto. <https://materiaalit.github.io/tikape-s17/part7/>. Luettu 30.4.2019.

34 Validation. Verkkoaineisto. Mongoose. <https://mongoosejs.com/docs/validation.html>. Luettu 30.4.2019.

35 Lyons, Peter. 2014. Mongoose: Schema vs Model?. Verkkoaineisto. <https://stackoverflow.com/questions/22950282/mongoose-schema-vs-model/22950402#22950402>. 8.4.2014. Luettu 1.5.2019.

36 Queries. Verkkoaineisto. Mongoose. <https://mongoosejs.com/docs/queries.html>. Luettu 30.4.2019.

Vaatimusmäärittely

Tehtävä tulo-menolaskuri tulee tekijän omaan käyttöön. Sen on tarkoitus korvata kaksi Excel-tiedostoa, johon vastaavat tiedot merkitään tällä hetkellä.

Excel-tiedostoihin verrattuna tavoitteena on alla luetellut hyödyt.

- Tietoa ei tarvitse muotoilla, vaan laskuri huolehtii tiedon muodosta.
- Kuukauden tulojen ja menojen summan sekä tulo- ja menokirjausten määrän näkee automaattisesti.
- Kuukauden tapahtumista näkee yhdellä painalluksella erilaisia yhteenvetotietoja.
- Mahdollisuus lisätä aiemmin vaikeasti toteutettavia tai mahdottomia ominaisuuksia.

Toiminnalliset vaatimukset

1. Kirjaukset on jaoteltu
    a. tyypin (tulo tai meno) mukaan
    b. kuukauden mukaan
    c. toistuvuuden mukaan kuukauden sisällä.
2. Tarkasteltavan kuukauden voi valita
    a. yksi kuukausi kerrallaan nuolista edeten
    b. erillisestä kuukausivalitsimesta.
3. Järjestelmä muotoilee käsittelemänsä tiedon automaattisesti.
4. Kirjauksista voi tallettaa seuraavat tiedot. Pakolliset tiedot on lihavoitu.
    a. tyyppi [tulo tai meno]
    b. toistuvuus [kertaluontoinen tai toistuva]
    c. päivämäärä
    d. summa sentin tarkkuudella
    e. nimike (esimerkiksi bussilippu)
    f. kategoria (esimerkiksi liikkuminen)
    g. maksaja
    h. kommentti.
5. Järjestelmä antaa uuteen kirjaukseen seuraavat tiedot esitäytettynä:
    a. päivämääräkenttään nykyinen päivämäärä
    b. summakenttään euromerkki
    c. kategoriakenttään arvon voi valita myös pudotusvalikosta. Pudotusvalikon arvot tulevat tietokantaan tallennetuista samaa tyyppiä olevista kirjauksista.

   d. maksajakenttään arvon voi valita myös pudotusvalikosta. Pudotusvalikon arvot tulevat tietokantaan tallennetuista samaa tyyppiä olevista kirjauksista.
6. Järjestelmä näyttää kirjaustaulukon sarakeotsikot ja tarkasteltavien kirjausten tyypin, vaikka käyttäjä selaa sivua alas.
7. Järjestelmästä näkee:
   a. yksittäisen päivän tulot yhteen laskettuna
   b. yksittäisen päivän menot yhteen laskettuna
   c. yksittäisen kuukauden tulot ja menot yhteen laskettuna
   d. yksittäisen kuukauden tulot ja tulotapahtumien määrä
   e. yksittäisen kuukauden menot ja menotapahtumien määrä.
8. Kirjauksia voi muokata.
9. Kirjauksia voi poistaa.
10. Seuraavat tapahtumat varmistetaan käyttäjältä:
    a. kirjauksen poisto
    b. muokattujen tietojen hylkäys kirjauksen muokkausnäkymässä.
11. Käyttäjä näkee:
    a. ilmoituksen, kun hän on lisännyt tai poistanut tapahtuman tai muokannut tapahtumaa.
    b. virheilmoituksen, jos kirjauksen lisäys, muokkaus tai poisto epäonnistuu puutteellisten tietojen vuoksi tai muusta syystä.
12. Ohjelma tarjoaa kullekin kuukaudelle yhteenvetonäkymän summan, kategorian ja maksajan perusteella.
    a. Tuloilla ja menoilla on toisistaan erilliset yhteenvetonäkymät.
    b. Kussakin yhteenvetonäkymässä rivit on järjestetty summan mukaan korkeimmasta pienimpään.
    c. Summan yhteenvetonäkymä näyttää kirjaukset summan mukaan. Samassa listassa on sekä säännölliset että kertaluontoiset kirjaukset, mutta ne on visuaalisesti toisistaan erotettavissa.
    d. Kategorianäkymä näyttää rivillä kategorian nimen, kategoriaan kuuluvien kirjausten summan, kategoriaan kuuluvien kirjausten lukumäärän ja summan osuuden kuukauden kokonaissummasta.
    e. Maksajanäkymä näyttää rivillä maksajan nimen, maksajaan liittyvien kirjausten summan, maksajaan liittyvien kirjausten lukumäärän ja summan osuuden kuukauden kokonaissummasta.

Muut huomiot

Muutoksia vaatimuksiin tai lisävaatimuksia voidaan määrittää toteutuksen aikana.