React-sovellusten arkkitehtuurin hyvät käytänteet

Ville Kuusela

Opinnäytetyö, AMK
Toukokuu 2023
Tietojenkäsittelyn tutkinto-ohjelma (AMK)

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

jamk

Kuvailulehti

Kuusela, Ville

React-sovellusten arkkitehtuurin hyvät käytänteet

Jyväskylä: Jyväskylän ammattikorkeakoulu. Toukokuu 2023, 42 sivua.

Tietojenkäsittelyn tutkinto-ohjelma. Opinnäytetyö AMK.

Julkaisun kieli: suomi

Julkaisulupa avoimessa verkossa: kyllä

Tiivistelmä

Tutkimuksessa perehdytään React-sovellusten arkkitehtuurin hyviin käytänteisiin ja tarkastellaan niiden hyötyjä sovelluskehityksessä. Interaktiivisten web-sovellusten kysynnän kasvaessa React on noussut yhdeksi suosituimmista JavaScript-kirjastoista responsiivisten käyttöliittymien luontiin. Ilman hyvin suunniteltua sovellusarkkitehtuuria React-sovellusten ylläpitämisestä ja jatkokehittämisestä voi kuitenkin tulla haasteellista.

Tutkimus toteutettiin tutkimuksellisena kehitystyönä. Tutkimuksen yhteydessä kehitettiin web-sovellus käyttäen Reactia ja soveltaen siihen monia yleisiä hyviksi väitettyjä arkkitehtuurikäytänteitä. Näin voitiin arvioida, kuinka hyvin käytänteet sopivat kyseiseen projektiin ja kuinka paljon niistä oli hyötyä sovelluksen kehittämisessä.

Arkkitehtuurikäytänteet on hyvin laaja alue, joten sitä ei pystytty tutkimaan täysin perusteellisesti, mutta tutkittaviksi valituista käytänteistä saatiin hyviä tuloksia. Tutkimuksessa saatiin selville, että valitut arkkitehtuurikäytänteet sopivat luotuun esimerkkisovellukseen hyvin. Käytänteiden noudattamisesta oli huomattavia hyötyjä sovelluksen kehittämisessä, ja sovelluksen mahdollinen ylläpitäminen ja jatkokehitys nähdään helppoina toteuttaa käytänteiden ansiosta.

Kehitetty esimerkkisovellus on suhteellisen yksinkertainen, mutta suurin osa siihen sovelletuista arkkitehtuurikäytänteistä nähdään hyödyllisinä myös monimutkaisemmissa ja laajemmissa sovelluksissa. Kaiken kaikkiaan tutkimus osoittaa, millaisia hyötyjä hyvästä sovellusarkkitehtuurista on, ja kuinka jokaisessa sovellusprojektissa on tärkeää valita projektiin parhaiten sopivat käytänteet.

Avainsanat (asiasanat)

React, sovellusarkkitehtuuri, sovelluskehitys, kehitystutkimus

Muut tiedot (salassa pidettävät liitteet)

jamk

Description

Kuusela, Ville

Best practices for React architecture

Jyväskylä: JAMK University of Applied Sciences, May 2023, 42 pages.

Degree Programme in Business Information Technology. Bachelor´s thesis.

Permission for open access publication: Yes

Language of publication: Finnish

Abstract

The study centers around the best practices in React application architecture and examines their benefits in software development. As the demand for interactive web applications grows, React has become one of the most popular libraries for creating responsive user interfaces with JavaScript. However, without a well-planned architecture maintaining and developing React applications further can become difficult.

The study utilized a development research approach. As part of the study, a web application was developed using React and applying many common architectural practices considered good. This allowed an evaluation of how well the practices fit the particular project and how much they benefited the application development.

Architectural practices are a very broad topic, so they could not be thoroughly researched, but good results were obtained from the selected practices. The study found that the selected architectural practices fit well with the example application created. Following the practices provided significant benefits in the application's development, and the application's possible future maintenance and further development are seen as easy thanks to the practices.

The example application developed is relatively simple, but most of the architectural practices applied to it are seen as useful in more complex and extensive applications as well. Overall, the study highlights the importance of architectural planning in React app development, and the need to evaluate the best practices for each application project.

Keywords/tags (subjects)

React, software architecture, software development, development research

Miscellaneous (Confidential information)

Sisältö

1 Johdanto ... 6
2 Sovellusarkkitehtuuri ... 6
2.1 4 + 1 -malli ... 7
2.2 Arkkitehtuurimallit ... 8
2.3 Suunnittelumallit ... 10
2.4 React ... 11
2.4.1 Komponentit ... 13
2.4.2 Komponentin tila ja tilanhallinta ... 14
2.4.3 Hookit ... 16
3 Tutkimusasetelma ... 19
3.1 Opinnäytetyön tavoitteet ... 19
3.2 Tutkimusmenetelmät ... 20
3.3 Rajaukset ... 20
3.4 Tutkimuskysymykset ... 21
4 Esimerkkisovelluksen toteutus ... 21
4.1 Sovelluksen kuvaus ... 21
4.2 Kansiorakenne ... 22
4.3 Absoluuttinen importointi ... 24
4.4 Komponentit ... 25
4.5 Tilanhallinta ... 30
4.6 Hookit ... 32
4.7 Reititys ... 34
5 Tulokset ... 35
6 Pohdinta ... 37
6.1 Tulosten arviointi suhteessa tietoperustaan ... 37
6.2 Luotettavuus ja eettisyys ... 38
6.3 Johtopäätökset ja kehittämisehdotukset ... 39
Lähteet ... 41

Kuviot

Kuvio 1. 4 + 1 -malli (Kruchten 1995, 43, muokattu) ... 7
Kuvio 2. MVC-arkkitehtuuri (Martin 2023a, muokattu) ... 9
Kuvio 3. MVVM-arkkitehtuuri (Martin 2023b, muokattu) ... 10

Kuvio 4. Yksinkertainen otsikkoelementin renderöivä komponentti 13
Kuvio 5. Datan kuljettaminen lapsikomponentille propsina 14
Kuvio 6. Painikkeen klikkauksia laskeva luokkakomponentti 15
Kuvio 7. Tekstisyöte talletetaan useState-hookilla luotuun tilaan 17
Kuvio 8. Datan hakeminen useEffect-hookissa 18
Kuvio 9. Esimerkkisovelluksen kansiorakenne 23
Kuvio 10. Import-lauseiden vertailu 25
Kuvio 11. Komponentti, joka renderöi logon 26
Kuvio 12. Emotion-kirjastolla tyylitetty div-elementti 27
Kuvio 13. Asetussivun logiikan sisältävä konttikomponentti 28
Kuvio 14. Komponentin renderöivän funktion syöttäminen propsina 29
Kuvio 15. Render-funktion propsina saava komponentti 30
Kuvio 16. Redux-siivu sovelluksen asetusten tilanhallintaan 31
Kuvio 17. Esimerkkisovellukseen luotu custom hook 32
Kuvio 18. Hookin kutsuminen komponentissa ja ehdollinen renderöinti 33
Kuvio 19. Esimerkkisovelluksen reittien määrittäminen 34
Kuvio 20. Linkkielementti hakutuloksen renderöivässä komponentissa 35

1 Johdanto

Sovellusten arkkitehtuurilla on merkittävä vaikutus niiden kehittämiseen ja ylläpitoon. Erityisesti sovellusten kasvaessa ja monimutkaistuessa hyvän sovellusarkkitehtuurin merkitys korostuu, sillä se takaa sovelluksen laadun ja ylläpidettävyyden. React on tunnettu JavaScript-kirjasto käyttöliittymien luomiseen. Se on noussut hyvin suureen suosioon, minkä vuoksi on tärkeää, että sovelluskehittäjät ovat perillä yleisistä React-sovelluksiin liittyvistä arkkitehtuurikäytänteistä.

Opinnäytetyössä perehdytään React-sovelluskehitykseen ja tutkitaan yleisimpiä React-sovelluksille hyviksi väitettyjä arkkitehtuurikäytänteitä. Aihetta tutkitaan kehittämällä esimerkkisovellus, johon valittuja käytänteitä sovelletaan. Näin niiden hyödyllisyyttä voidaan arvioida käytännössä. Työn tavoitteena on tutkia, kuinka hyvin valitut käytänteet sopivat kehitettävään esimerkkisovellukseen ja millaisia hyötyjä niiden noudattamisesta on.

Opinnäytetyön tarkoituksena on tarjota hyödyllistä tietoa React-sovelluskehityksestä ja siihen liittyvistä yleisistä arkkitehtuurikäytänteistä. Työn teoriaosuudessa käsitellään sovellusarkkitehtuuria yleisesti, tarkastellaan tärkeimpiä siihen liittyviä termejä, ja perehdytään Reactin keskeisimpiin ominaisuuksiin. Työn käytännön osuudessa kerrotaan esimerkkisovelluksen kehittämisprosessista ja käydään läpi siihen sovellettuja erilaisia arkkitehtuurikäytänteitä. Työn lopussa esitellään työn tulokset ja niihin liittyvät pohdinnat.

2 Sovellusarkkitehtuuri

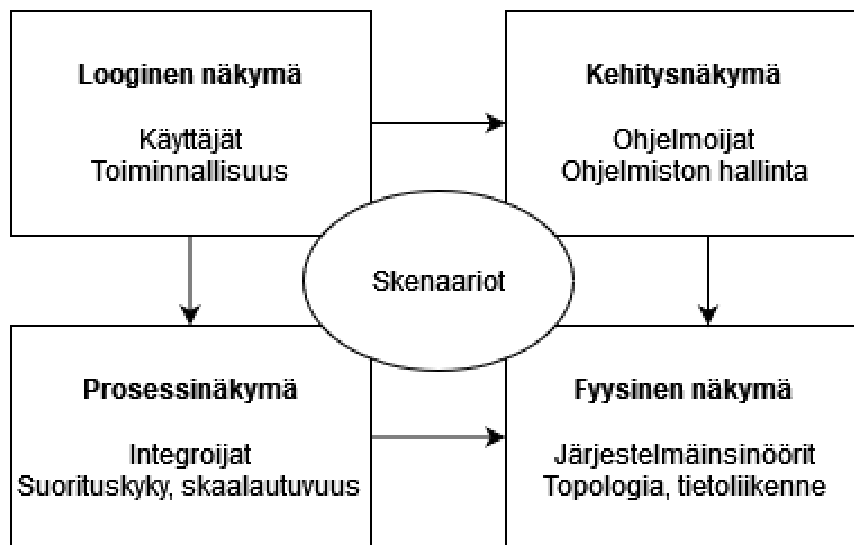
Sovellusarkkitehtuurilla ei ole täysin yksiselitteistä määritelmää. Gortonin (2011, luku 1.2) mukaan kansainvälisen tekniikan alan järjestö IEEE:n standardissa sovellusarkkitehtuurilla tarkoitetaan ohjelmiston yleistä rakennetta, sen eri osia ja niiden välistä vuorovaikutusta, sekä kehitykseen liittyviä periaatteita. Yksinkertaisimpien tietokoneohjelmien luominen ei vaadi merkittävää suunnittelua, mutta mitä suurempaa ja monimutkaisempaa ohjelmistoa kehitetään, sitä tärkeämpää sovellusarkkitehtuurin huolellinen suunnittelu on.

Sovelluksen suunnitteleminen ja toteuttaminen hyvin on kannattavaa, koska siten sovelluksesta saadaan luotua parempi monin eri tavoin. Sovelluksen laatua voidaan mitata ja arvioida monien eri ominaisuuksien kannalta. Näitä ovat muun muassa toimivuus, muokattavuus, skaalautuvuus,

ylläpidettävyys ja turvallisuus. Hyvin suunnitellulla sovellusarkkitehtuurilla voidaan helpottaa tällaisiin ominaisuuksiin liittyvien tavoitteiden saavuttamista. (Gorton 2011, luku 3.1.)

2.1 4 + 1 -malli

Yleinen ja tunnettu tapa sovelluksen kokonaisvaltaisen arkkitehtuurin kuvaamiseen on Philippe Kruchtenin kehittämä 4 + 1 -malli (ks. kuvio 1). Mallissa sovellusarkkitehtuuri jaetaan neljään erilaiseen päänäkymään, sekä ne yhdistävään skenaarionäkymään. Neljä päänäkymää ovat looginen näkymä, kehitysnäkymä, prosessinäkymä ja fyysinen näkymä. Mallin päänäkymät havainnollistavat, mitä eri osia sovelluksen arkkitehtuuriin kuuluu. Malli on ns. geneerinen, eli sen kanssa voidaan käyttää haluttuja notaatio- ja suunnittelutapoja. (Kruchten 1995, 42–43.)



Kuvio 1. 4 + 1 -malli (Kruchten 1995, 43, muokattu)

Mallin loogisessa näkymässä kuvataan sovelluksen loogista rakennetta, mihin kuuluu muun muassa sovelluksen erilaiset komponentit, komponenttien tehtävät sekä niiden väliset yhteydet. Näkymässä keskitytään järjestelmälle asetettuihin funktionaalisiin vaatimuksiin. Niillä tarkoitetaan toimintoja, joita sovelluksessa tulisi olla loppukäyttäjän näkökulmasta. Looginen näkymä tarjoaa oikein toteutettuna selkeän kuvauksen järjestelmän keskeisimmistä toiminnallisuuksista. (Mts. 43–44.)

Prosessinäkymässä kuvataan, kuinka järjestelmän eri osat ja prosessit toimivat käytännössä ja kuinka se käsittelee erilaisia toimintoja. Näkymässä keskitytään järjestelmän non-funktionaalisiin vaatimuksiin. Tällaisia ovat muun muassa sovelluksen suorituskyvylle ja saatavuudelle asetetut vaatimukset. Prosessinäkymän kuvaaminen auttaa hahmottamaan järjestelmän toimintaa käytännössä, ja kuinka toimintaa voitaisiin mahdollisesti parantaa. (Mts. 44–45.)

Kehitysnäkymässä kuvataan ohjelmistomoduulien, eli ohjelmiston erikseen kehitettävien osien, yleistä rakennetta, hierarkiaa ja niiden tehtäviä ohjelmistossa. Näkymän voidaan siis ajatella kuvaavan ohjelmistoa sen ohjelmoijien näkökulmasta. Siinä otetaan huomioon, kuinka järjestelmän koodi on organisoitu eri tehtävistä vastaaviin tasoihin ja kuinka eri tasot ovat yhteydessä toisiinsa. Kehitysnäkymän kuvaaminen auttaa esimerkiksi projektin sisäisten vaatimusten, kuten sovelluksen kehityksen ja ylläpidettävyyden helppouden saavuttamisessa. (Mts. 45–46.)

Fyysisessä näkymässä kuvataan järjestelmässä käytettäviä fyysisiä laitteita ja kuinka ne ovat yhteydessä toisiinsa. Näkymässä kuvataan myös, kuinka järjestelmän ohjelmistopuoli on yhdistetty laitteistoon. Näkymä kuvaa siis ohjelmistoarkkitehtuuria järjestelmäinsinöörien näkökulmasta. (Mts. 46–47.)

4 + 1 -malli sisältään neljän päänäkymän lisäksi niin kutsutun skenaarionäkymän. Skenaarioilla tarkoitetaan mallissa yksittäisiä, yleisiä käyttötapauksia. Skenaarioilla voidaan kuvata järjestelmän neljän päänäkymän toimintaa yhtenä kokonaisuutena erilaisissa loppukäyttäjän näkökulmasta luoduissa toiminnallisuusketjuissa. (Mts. 47.)

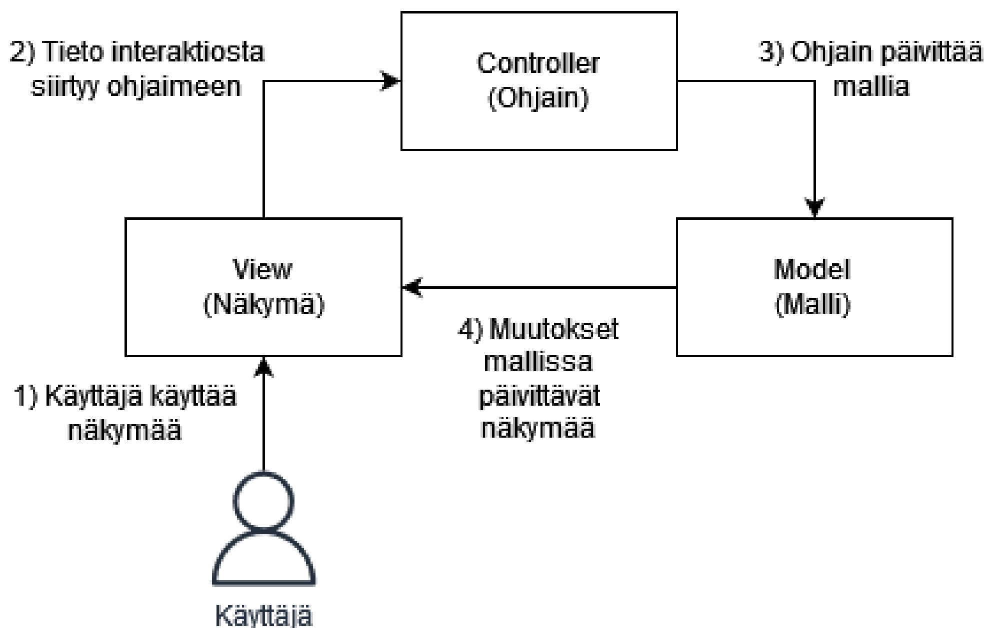
2.2 Arkkitehtuurimallit

On olemassa monenlaisia uudelleenkäytettäviä ratkaisuja sovelluksen arkkitehtuurin toteuttamista varten. Näitä ratkaisuja kutsutaan arkkitehtuurimalleiksi (architectural patterns). Mallit kuvaavat erilaisia uudelleenkäytettäviä, ennalta määrättyjä tapoja arkkitehtuurin toteuttamiseen ja sovelluksen koodin jakamiseen pienempiin lohkoihin. (Dhaduk 2020.)

Erilaisia arkkitehtuurimalleja on paljon, ja ne ovat hyödyllisiä erilaisissa tilanteissa. Niistä tunnetuimpiin ja yleisimmin käytettyihin kuuluvat muun muassa MCV-malli, MVVM-malli, asiakas-palvelinmalli, mikropalveluarkkitehtuuri, tapahtumapohjainen arkkitehtuuri ja kerrostettu arkkitehtuuri.

Kullakin mallilla on omat vahvuutensa ja heikkoutensa, ja niiden valinta riippuu projektin vaatimuksista ja tavoitteista. (Mt.)

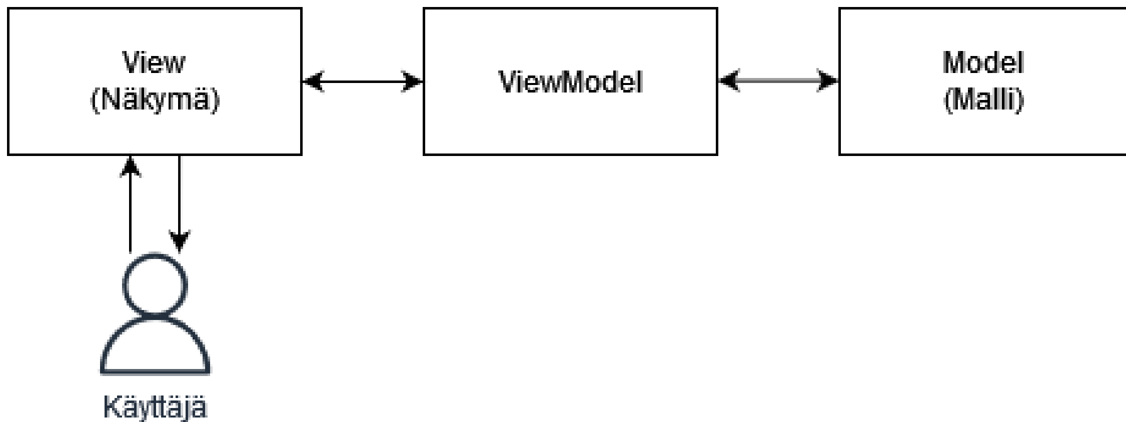
Web-sovelluksissa erittäin yleiseksi noussut arkkitehtuuriratkaisu on MVC-arkkitehtuuri, eli Model-View-Controller (ks. kuvio 2). Siinä ohjelma jaetaan kolmeen kerrokseen, jotka ovat malli (model), näkymä (view) ja ohjain (controller). Mallikerros sisältää sovelluksen käyttämän datan ja sen käsittelyyn liittyvän logiikan. Näkymä sisältää sovelluksen visuaalisen käyttöliittymän, jossa sovelluksen data esitetään. Ohjainkerroksen metodeissa käsitellään toimintoja, joita käyttäjä suorittaa ollessaan interaktiossa käyttöliittymän kanssa, ja nämä metodit muokkaavat mallia. Kun malli muuttuu, näkymää muokataan sen mukaisesti. (Martin 2023a.)



Kuvio 2. MVC-arkkitehtuuri (Martin 2023a, muokattu)

MVC-arkkitehtuurin kaltaisten, koodin eri osiin pilkkovien ratkaisujen käyttämisessä on yleensä monia hyötyjä. Kun koodi on jaettu useisiin eri tehtävää ajaviin osiin, eri osia voidaan kehittää muista itsenäisesti. Tästä jaosta puhuttaessa käytetään yleisesti termiä ”separation of concerns”. Koodin jakaminen osiin mahdollistaa myös yksittäisten osien kehittämisen lisäksi niiden helpon muokkauksen ja testaamisen. (Martin 2023a.)

MVC-mallia muistuttava arkkitehtuurimalli on MVVM-malli (ks. kuvio 3), jossa ohjain on korvattu ViewModel-nimisellä kerroksella. Siinä missä MVC:n ohjainkerros muokkaa mallia, ViewModel-osa toimii kaksisuuntaisena välipisteenä mallin ja näkymän välillä. Se ei vain muokkaa mallia, vaan myös vastaanottaa siltä dataa ja välittää sen näkymään. (Martin 2023b.)



Kuvio 3. MVVM-arkkitehtuuri (Martin 2023b, muokattu)

MVVM on MVC-mallia uudempi arkkitehtuurimalli, mutta sekin on noussut suosioon web-kehityksessä. Esimerkiksi monet JavaScriptin suositut ohjelmistokehykset hyödyntävät MVVM-arkkitehtuuria tai sitä muistuttavaa muunnosta. Tällaisia ohjelmistokehyksiä ovat esimerkiksi Vue ja Knockout (The Vue Instance n.d.; Knockout n.d.).

MVC ja MVVM ovat tunnettuja ja yleisesti käytettyjä, mutta ne ovat kuitenkin vain kaksi esimerkkiä sovellusarkkitehtuurimalleista. Arkkitehtuurimalleja on paljon erilaisia, ja niistä jokaisella on omat heikkoutensa ja vahvuutensa. Ei ole mitään yhtä mallia, mikä sopisi mihin tahansa projektiin, vaan on hyvä pohtia projektikohtaisesti, kuinka sovelluksen arkkitehtuuri toteutetaan.

2.3 Suunnittelumallit

Arkkitehtuurimallin lisäksi tärkeä termi ohjelmistokehityksessä on ohjelmistosuunnittelumalli (software design pattern). Suunnittelumallilla tarkoitetaan ratkaisua johonkin tiettyyn yksittäiseen ongelmaan. Suunnittelumalli ei tarkoita valmista koodia, joka voidaan ottaa käyttöön sellaisenaan, vaan se on yleistettävä malli ongelman ratkaisuun. (Gamma, Helm, Johnson & Vlissides 1994, 2–4.)

Suunnittelumallien historiassa hyvin merkittävä kirja on vuonna 1994 julkaistu Design Patterns, jossa Gang of Four -nimelläkin tunnettu neljän kirjailijan ja ohjelmistoalan asiantuntijan ryhmä listaa ja esittelee 23 erilaista suunnittelumallia. Teoksessa suunnittelumallit on jaettu kolmeen kategoriaan niiden tarkoituksen mukaan. Nämä kategoriat ovat luontimallit, rakennemallit ja käyttäytymismallit. (Mts. 9–11.)

Luontimalleilla (creational patterns) tarkoitetaan suunnittelumalleja, jotka antavat ratkaisuja ohjelmiston rakenneosien, kuten olioiden ja luokkien luomiseen. Luontimallien tarkoituksena on auttaa käsittelemään monimutkaisia luomisprosesseja ja tehdä ohjelmistosta joustavampaa. Yleisiä luontimalleja ovat muun muassa Builder ja Prototype. (Mts. 81.)

Rakennemallit (structural patterns) kuvaavat tapoja, joilla yksittäisistä rakenneosista voidaan luoda suurempia rakennelmia, ja kuinka ne saadaan toimimaan yhdessä parhaiten. Kategorian mallit auttavat luokkien yhteensopivuuden ja koodin uudelleenkäytettävyyden lisäämisessä. Rakennemalleja ovat esimerkiksi Adapter, Bridge ja Decorator. (Mts. 137.)

Käyttäytymismallit (behavioral patterns) ovat suunnittelumalleja, jotka kuvaavat ohjelman rakenneosien omia vastuita ja tehtäviä, sekä osien välistä vuorovaikutusta. Niitä käytetään selkeyttämään monimutkaisia yhteyksiä osien välillä. Esimerkkejä yleisistä käyttäytymismalleista ovat muun muassa Observer ja Template Method. (Mts. 221.)

Design Patterns -kirjassa esitellyt suunnittelumallit ovat tunnettuja ja yleisesti käytettyjä, mutta suunnittelumalleja on paljon enemmän kuin mitä teos sisältää. Suunnittelumallit ovat tärkeä osa hyvän sovellusarkkitehtuurin luomisessa. Oikein käytettynä ne auttavat luomaan koodia tehokkaammin, ja varmistamaan, että luotu koodi on selkeää sekä helposti ylläpidettävää ja uudelleenkäytettävää.

2.4 React

React, joka tunnetaan myös nimellä React.js, on interaktiivisten käyttöliittymien luomiseen tarkoitettu JavaScript-kirjasto. Facebookin (nykyään Meta) kehittämän Reactin kehitys alkoi vuonna 2011, kun Facebook tarvitsi paremman tavan sovelluksensa ylläpidettävyyden varmistamiseen.

Tuolloin Jordan Walke loi prototyypin, josta React sai alkunsa. React julkaistiin avoimena lähdekoodina vuonna 2013. (Hámori 2022.)

Julkaisunsa jälkeen Reactin suosio sovelluskehittäjien parissa on kasvanut suuresti. Tällä hetkellä React on yksi suosituimmista web-kehitykseen liittyvistä teknologioista sekä käyttäjämäärältään että käyttäjiensä tyytyväisyysasteelta. Tässä asemassa React on ollut jo vuosia, ja kiinnostus sitä kohtaan on yhä korkealla. (2022 Developer Survey 2022.)

React luokitellaan yleensä sovelluskehykseksi, ja sitä verrataan usein eri sovelluskehyksiin. (Ks. esim. Elliott 2023; Sugandhi 2023.) Reactia ei kuitenkaan voida pitää varsinaisena sovelluskehyksenä, sillä siitä puuttuu monien sovelluksissa usein käytettyjen omaisuuksien tekemiseen vaaditut keinot, jotka ovat varsinaisissa sovelluskehyksissä sisäänrakennettuina. Tällaisten puutteiden paikkaamiseksi Reactille on kehitetty lukuisia erilaisia kirjastoja, jotka lisäävät siihen sovelluskehityksessä tärkeitä ominaisuuksia. Kirjastojen avulla Reactia käytetäänkin usein kokonaisten sovellusten tekemiseen oikeiden sovelluskehysten tavoin. (Barger 2021b.)

Toinen ero moniin JavaScriptin sovelluskehyksiin on se, että React antaa kehittäjille hyvin vapaat kädet tehdä asioita kuten he haluavat. Tästä voi olla suurta hyötyä, mutta laajat vapaudet voivat helposti johtaa myös huonoihin ratkaisuihin, jos kehittäjä ei tiedä mitä tekee. Tämän vuoksi React-kehittäjien on tärkeä ymmärtää, kuinka React-sovellusten arkkitehtuuri luodaan parhaiden käytänteiden mukaisesti.

Reactilla luodut käyttöliittymän komponentit ja niiden välinen vuorovaikutus ovat osa sovelluksen loogista arkkitehtuuria, joten 4 + 1 -mallin näkökulmasta katsottuna Reactin voidaan ajatella liittyvän mallin loogiseen näkymään. Looginen näkymä keskittyy sovelluksen toiminnallisuuksiin sen loppukäyttäjien näkökulmasta, joten käyttöliittymää voidaan pitää näkymän keskeisimpänä osana. React on myös osa sovelluksen kehitysympäristöä, joten se liittyy myös mallin kehitysnäkymään.

MVC:n ja MVVM:n kaltaisia arkkitehtuurimalleja noudattavissa sovelluksissa Reactin käyttäminen vastaisi selkeimmin mallin näkymää eli view-kerrosta. Näkymäkerrokseen luodaan ohjelmiston käyttöliittymä, jota päivitetään mallikerroksesta saatavan datan mukaan. Reactilla luodut interaktiiviset käyttöliittymät sopivat tähän tarkoitukseen hyvin.

Reactin kontekstissa arkkitehtuurilla tarkoitetaan Reactilla luodun sovelluksen ja sen eri osien yleistä rakennetta ja suunnittelua, sekä kaikkia tärkeimpiä niihin vaikuttavia toimintoja. React-sovellusten arkkitehtuurien suunnittelussa tulee ottaa huomioon monia erilaisia asioita, kuten käyttöliittymän eri komponentit, projektin kansiorakenne, tilanhallinta, hookit, ja eri osiin sovellettavat suunnittelumallit. Hyvin suunniteltu arkkitehtuuri on tärkeä osa React-kehittämistä, sillä se tekee sovelluksen kehittämisestä, ylläpitämisestä, laajentamisesta ja testaamisesta helpompaa. (Gospel 2023.)

2.4.1 Komponentit

Reactilla luodut käyttöliittymät perustuvat komponentteihin. Komponenteilla tarkoitetaan käyttöliittymän itsenäisiä, uudelleenkäytettäviä osia. Funktiomuotoiset komponentit ovat JavaScript-funktioita, jotka palauttavat käyttöliittymässä näytettäviä elementtejä. Yksinkertaisimmillaan React-komponentti palauttaa vain yksinkertaisen käyttöliittymäelementin, eikä sisällä mitään muuta koodia (ks. kuvio 4). (Components and Props n.d.)

```
1  function Example() {
2    return <h1>Hello World!</h1>;
3  }
4
5  export default Example;
6
```

Kuvio 4. Yksinkertainen otsikkoelementin renderöivä komponentti

React käyttää elementtien luomiseen JSX-syntaksia. JSX on JavaScriptin laajennus, jolla voidaan luoda HTML:n kaltaisia käyttöliittymäelementtejä JavaScript-koodin sisällä. JSX:n käyttäminen Reactin yhteydessä ei ole välttämätöntä, mutta se on hyvin suositeltavaa, koska se helpottaa elementtien luomista sekä niiden päivittämistä tilan muuttuessa. (Introducing JSX n.d.)

Komponentit voivat käyttöliittymäelementtien palauttamisen lisäksi pitää sisällään paljon muutakin koodia, vaikka monimutkaistakin JavaScript-logiikkaa. Komponentit voivat myös saada dataa

toisilta komponenteilta niin kutsuttuina propseina. Propsit lähetetään komponenttiin kirjoittamalla ne komponentin luovaan elementtiin, ja ne saadaan komponentissa käyttöön sen luovan funktion parametreina (ks. kuvio 5).

```
1   function Child(props) {
2     return <h1>Hello {props.string}!</h1>;
3   }
4
5   function Example() {
6     return <Child string="World" />;
7   }
8
9   export default Example;
10
```

Kuvio 5. Datan kuljettaminen lapsikomponentille propsina

React-komponentit voidaan luoda funktiolauseiden sijaan myös class-lauseella. Nämä luokkakomponentit ovat funktiotyyppisiin komponentteihin verrattuna hieman monimutkaisempia, mutta niihin voidaan sisällyttää toiminnallisuuksia, jotka eivät toimi funktiotyyppisissä komponenteissa sellaisenaan. Tällaisia toiminnallisuuksia ovat muun muassa elinkaarimetodit (lifecycle methods). Ne mahdollistavat koodin osien suorittamisen esimerkiksi yksinomaan silloin, kun komponentti tuodaan näkyviin tai poistetaan näkymästä. Elinkaarimetodeista voi olla hyötyä monimutkaisemmissa käyttöliittymissä. (State and Lifecycle n.d.)

2.4.2 Komponentin tila ja tilanhallinta

Reactilla luoduissa komponenteissa tapahtuu usein muutoksia, joiden pitää heijastua myös käyttöliittymään. Esimerkiksi tekstikentän sisällön tulee päivittyä, kun käyttäjä kirjoittaa siihen merkkejä. Jos Reactissa tämä yritetään toteuttaa pelkällä JavaScript-muuttujalla, jonka arvo näytetään käyttöliittymäelementissä ja jota päivitetään, näkymä ei päivity. Tämä johtuu siitä, että paikallisten muuttujien arvojen muuttuminen ei saa komponenttia renderöitymään uudestaan. Muuttujien mahdolliset uudet arvot eivät myöskään säily komponentin uudelleenrenderöityessä. (State: A Component’s Memory n.d.)

Jotta käyttöliittymässä näkyviä arvoja saadaan päivitettyä ilman kahta edellä mainittua ongelmaa, Reactin komponenteilla on tila (state). Tilalla tarkoitetaan sellaisia arvoja, joiden päivittyminen saa näkymän renderöitymään uudestaan, ja joiden muutokset säilyvät uuteen renderöitymiseen. Class-tyyppisessä komponentissa tilan määrittäminen tapahtuu luokan rakentajassa (constructor, ks. kuvio 6). Funktiotyyppisissä komponenteissa tilan määrittäminen tapahtuu sen sijaan hieman toisella tavalla, sillä siihen käytetään Reactin sisäänrakennettuja hookeja. (Mt.)

```
3   class Counter extends React.Component {
4     constructor(props) {
5       super(props);
6       this.state = {
7         count: 0,
8       };
9     }
10
11    render() {
12      return (
13        <div>
14          <p>Count: {this.state.count}</p>
15
16          <button onClick={() => this.setState({ count: this.state.count + 1 })}>
17            Click
18          </button>
19        </div>
20      );
21    }
22  }
```

Kuvio 6. Painikkeen klikkauksia laskeva luokkakomponentti

Kun sovellus kasvaa ja sen komponenttien tilat vaativat monimutkaisempia toimintoja, on tärkeää suunnitella huolellisesti, mikä on paras tapa tilojen käsittelyyn. Tätä kutsutaan tilanhallinnaksi (state management). Monesti komponenttien tiloja täytyy esimerkiksi kuljettaa komponenteista toisiin, mikä voidaan toteuttaa esimerkiksi propsien avulla. Laajoissa sovelluksissa, joissa lukuisat komponentit ovat laajasti vuorovaikutuksessa toistensa kanssa, tämä ratkaisu voi kuitenkin johtaa vaikeaselkoiseen koodiin. (Managing State n.d.)

React-sovelluksissa tilanhallinta on usein järkevää toteuttaa globaalisti. Globaalia tilaa ei säilötä komponenteissa, vaan niistä erillisessä varastossa, jota kutsutaan myös storeksi. Storessa oleva tila

voidaan tuoda suoraan mihin tahansa komponenttiin. React tarjoaa tiettyjä omia keinoja globaalin tilanhallinnan toteuttamiseen, mutta usein siihen käytetään ulkopuolista kirjastoa. Tunnettuja tilanhallinnan kirjastoja ovat muun muassa Redux, Recoil, Jotai ja Zustand. Näistä Redux on kaikkein suosituin. (Wieruch 2023.)

2.4.3 Hookit

Reactin luokkakomponentit ovat usein monimutkaisempia ja enemmän koodia vaativia kuin funktiotyyppiset komponentit. Jotta funktiokomponenteissa voitaisiin toteuttaa sellaiset ominaisuudet, jotka olivat aluksi mahdollisia vain luokkakomponenteilla, Reactin kehittäjät lisäsivät Reactiin erilaisia hookeja. Hookit ovat funktioita, jotka mahdollistavat monien Reactin ominaisuuksien käytön ilman luokkia. (Minnick 2022, luku 11.)

Reactin hookeilla on kaksi sääntöä, joita niiden on noudatettava toimiakseen oikein. Ensimmäinen sääntö määrittää, että hookeja voidaan kutsua ainoastaan funktiotyyppisissä komponenteissa tai itse luoduissa custom hookeissa. Toinen sääntö on, että hookeja voidaan kutsua vain komponentin ylimmällä tasolla, eikä esimerkiksi komponentin koodissa olevasta ehtolauseesta, silmukasta tai sisäisestä funktiosta. Toinen sääntö auttaa varmistamaan, että hookit kutsutaan aina samassa järjestyksessä ja tasan yhden kerran komponentin renderöityessä, mikä on vaatimus sille, että hookit toimivat oikein. (Minnick 2022, luku 11.)

Funktiokomponentteihin saa lisättyä tilan useState-hookilla (ks. kuvio 7). UseState-hookissa on kaksi osaa: tilan arvo, ja funktio, jolla arvoa muutetaan. Hookilla luotu tila toimii samalla tavalla kuin luokkakomponenteissa: tilan arvon päivittäminen saa komponentin renderöitymään uudestaan, ja tilan päivitetty arvo säilyy uuteen renderöitymiseen.

```
import { useState } from 'react';

function Example() {
  const [inputValue, updateInputValue] = useState('');

  return (
    <div>
      <input
        value={inputValue}
        onChange={(e) => updateInputValue(e.target.value)}
      />

      <p>Input Value: {inputValue}</p>
    </div>
  );
}
```

Kuvio 7. Tekstisyöte talletetaan useState-hookilla luotuun tilaan

UseState-hookiin voidaan kirjoittaa aloitusarvo, joka tulee tilan arvoksi, kun komponentti ladataan. Kuviossa 7 aloitusarvona annetaan tyhjä merkkijono (''). Aloitusarvona voi olla merkkijonon lisäksi esimerkiksi numero, totuusarvo tai objekti. Kun tilaan talletetaan objekti, siinä voidaan säilöä useita arvoja samaan aikaan, sillä objektin sisäisiä arvoja voidaan päivittää myös yksittäin.

Toinen hyvin tärkeä hook Reactissa on useEffect, joka mahdollistaa elinkaarimetodien käytön funktiotyyppisissä komponenteissa. UseEffect-hookin sisään voidaan kirjoittaa koodia, joka ajetaan vain tietyissä tapauksissa, esimerkiksi vain komponentin latautuessa ensimmäisen kerran, tietyn tilan päivittyessä, tai komponentin purkautuessa. UseEffectiä käytetään tyypillisesti ulkoisten palveluiden tai komponentin tilan hallinnassa. Esimerkiksi jos komponentissa ladataan ulkoista dataa, joka talletetaan komponentin tilaan, sen lataus on hyvä suorittaa useEffect-hookissa (ks. kuvio 8). (Minnick 2022, luku 11.)

```
import { useState, useEffect } from 'react';

function Example() {
  const [fact, setFact] = useState('');

  useEffect(() => {
    fetch('https://catfact.ninja/fact')
      .then((response) => response.json())
      .then((data) => setFact(data.fact));
  }, []);

  return <p>Random cat fact: {fact}</p>;
}

```

Kuvio 8. Datan hakeminen useEffect-hookissa

Se, milloin useEffect-hookissa oleva koodi ajetaan, voidaan määrittää sen toisessa argumentissa. Kuviossa 8 hookin toisena argumenttina on tyhjä taulukko ([]), mikä tarkoittaa useEffectin koodin ajettavan vain silloin, kun komponentti ladataan näkyviin ensimmäisen kerran. Jos taulukon sisälle laitettaisiin jokin tila, hookin koodi ajettaisiin vain silloin, kun kyseisen tilan arvo päivittyy.

Jos useEffect-hookin toinen argumentti jätetään pois kokonaan, eli taulukkoa ei lisätä ollenkaan, hookissa oleva koodi ajetaan aina kun komponentti latautuu tai päivittyy. Kuviossa 8 esitetyssä komponentissa tämä johtaisi loputtomasti toistuviin datapyyntöihin, sillä kun komponentin tilaa päivitetään latauksessa saadulla datalla, komponentti uudelleenrenderöityisi ja useEffectissä oleva pyyntö suoritettaisiin jälleen. Tämän virheen välttäminen on tärkeää useEffectiä käytettäessä.

UseMemo on hook, joka säilöö sisältämänsä funktion palautusarvon. Funktio ajetaan vain silloin, kun komponentti ladataan ensimmäisen kerran, tai kun hookin riippuvuustaulukkoon kirjoitettu arvo on muuttunut komponentin latautuessa uudelleen. UseMemosta on hyötyä esimerkiksi silloin, jos komponentissa tulee suorittaa raskasta ja aikaa vievää koodia, jota ei haluta ajaa joka kerta kun komponentti ladataan. (Built-in React Hooks n.d.)

UseRef-hook sen sijaan on samantapainen kuin useState, koska sen avulla voidaan tallettaa dataa, joka säilyy komponentin latautuessa uudestaan. Ero useStaten ja useRefin välillä on siinä, että

useRefin arvon päivittäminen ei aiheuta komponentin uudelleenlatautumista. Yleinen käyttötarkoitus useRef-hookille on komponentissa luotaviin käyttöliittymäelementteihin viittaaminen. (Mt.)

UseContext-hookilla komponenteissa voidaan vastaanottaa dataa niiden yllä olevista komponenteista ilman datan kuljettamista komponentista toiseen propseina. Sen avulla voidaan päivittää tilaa kerralla useassa eri komponentissa, jotka ovat samasta tilasta riippuvaisia. UseContextia käytetään usein muun muassa siinä tapauksessa, jos sovelluksen teemaa voidaan vaihtaa, esimerkiksi tumman ja vaalean teeman välillä. UseContextilla tieto siitä, mikä teema on valittuna, voidaan välittää kaikkiin komponentteihin, joissa kyseistä tietoa tarvitaan. UseContextia voidaan siis käyttää globaalin tilan luomiseen. (Mt.)

Kaikki edellä mainitut hookit ovat tärkeitä ja yleisesti Reactissa käytettyjä, mutta Reactissa on vielä monia muitakin hookeja. Tämän lisäksi Reactissa on mahdollista luoda omia hookeja, eli custom hookeja. Niillä tarkoitetaan uudelleenkäytettäviä funktioita, jotka hyödyntävät Reactin sisäänrakennettuja hookeja. Custom hookeilla koodia voidaan erottaa komponenteista, ja viedä siten sama koodi käytettäväksi useisiin komponentteihin. (Minnick 2022, luku 11.)

3 Tutkimusasetelma

3.1 Opinnäytetyön tavoitteet

Opinnäytetyössä tutustutaan moniin yleisiin erilaisista lähteistä löytyviin ja hyviksi väitettyihin React-sovellusten arkkitehtuurikäytänteisiin. Työssä arvioidaan niiden käyttämisen vaikutuksia, hyötyjä ja mahdollisia haittoja käytännön sovelluskehitysprojektissa. Työn tavoite on selvittää, ovatko tutkittavat käytänteet niin hyviä kuin mitä niistä väitetään, kuinka hyvin ne sopivat kehitettävään esimerkkisovellukseen, ja sopisivatko ne paremmin toisenlaisiin sovellusprojekteihin.

Aihe on tärkeä ja ajankohtainen Reactin suuren suosion ja yleisen käytön vuoksi. React antaa myös kehittäjille paljon vapauksia toteuttaa sovelluksensa kuten he haluavat, mikä voi johtaa huonoihin ratkaisuihin, jos kehittäjä ei tunne arkkitehtuurin hyviä käytänteitä. Myös tämän vuoksi aihe on tärkeä tutkittavaksi, sillä opinnäytetyön tulokset voivat antaa hyödyllisiä vinkkejä tuleville React-projekteille sovellusarkkitehtuuriin liittyen.

Koska React on niin suosittu, ja hyvät arkkitehtuuriratkaisut ovat merkittävä osa Reactia, aiheesta on kirjoitettu jo lukuisia artikkeleita ja muuta kirjallisuutta. Tästä huolimatta Reactin arkkitehtuurin hyvät käytänteet eivät ole mitään itsestäänselvyyksiä. Monet lähteet esittävät hieman erityyppisiä näkemyksiä siitä, millaisia käytänteitä React-sovelluksissa tulisi noudattaa, eivätkä kaikki hyviksi väitetyt käytänteet välttämättä toimi yhtä hyvin kaikissa erilaisissa sovelluksissa.

3.2 Tutkimusmenetelmät

Toikon ja Rantasen (2009) mukaan tutkimuksellinen kehitystyö tuo yhteen kehittämistoiminnan ja tutkimustoiminnan. Siinä etsitään vastauksia tutkimuksellisiin ja käytännöllisiin kysymyksiin soveltamalla tietoa konkreettiseen kehitystyöhön. Tutkimuksellisia menetelmiä käytetään ohjaamaan kehittämistä, jolloin lopputuloksena syntyy jokin tuotos. React-sovellusten erilaisten arkkitehtuurikäytänteiden arviointi onnistuu parhaiten konkreettisessa sovelluskehitystyössä, joten opinnäytetyön tutkimusmenetelmäksi valittiin kehittämistutkimus. (Toikko & Rantanen 2009, 19–23.)

Opinnäytetyössä arkkitehtuurikäytänteiden arviointi tapahtuu soveltamalla niitä työn yhteydessä kehitettävään esimerkkisovellukseen, ja arvioimalla niiden toimivuutta kyseisessä sovelluksessa. Arvioinnin yhteydessä pohditaan lisäksi, toimisivatko tutkittavat käytänteet paremmin jossain toisenlaisessa sovellusprojektissa. Arviointi tulee mahdollisesti olemaan osittain subjektiivista, koska kyseessä on kirjoittajan itse kehittämä sovellus ja omat kokemuksensa sen kehittämisestä, mutta arvioinnissa pyritään kuitenkin olemaan kriittinen ja mahdollisimman objektiivinen, jotta tuloksista saataisiin yleistettäviä.

3.3 Rajaukset

Sovellusarkkitehtuuri on hyvin laaja ja monipuolinen käsite, myös Reactin kontekstiin rajattuna. Hyviä tapoja sovellusarkkitehtuurin toteuttamiselle on valtavasti, eikä niitä kaikkia ole mahdollista käsitellä yhdessä työssä. Tästä syystä opinnäytetyössä ja sen yhteydessä kehitettävässä sovelluksessa tullaan syventymään vain tiettyihin yleisimpiin eri lähteistä löytyviin hyviksi väitettyihin arkkitehtuurikäytänteisiin.

Jotta tutkittavia käytänteitä voitaisiin arvioida mahdollisimman perusteellisesti, niitä tulisi kokeilla monissa erityyppisissä ja -kokoisissa sovellusprojekteissa. Käytänteiden käytön hyötyjen arvioinnissa olisi myös ihanteellista, jos sama esimerkkisovellus voitaisiin luoda sekä käytänteitä noudattaen että ilman niitä. Näin perusteellinen tutkimus ei kuitenkaan ole mahdollinen opinnäytetyön yhteydessä ajan ja muiden resurssien rajallisuuden vuoksi. Hyvä sovellusarkkitehtuuri helpottaa myös sovelluksen kehittämistä tiiminä, mutta opinnäytetyössä kehitettävä sovellus joudutaan luomaan yksin.

3.4 Tutkimuskysymykset

Opinnäytetyön tavoitteiden ja sen aiheelle asetettujen rajausten perusteella tutkimuskysymyksiksi valittiin seuraavat:

- Millaisia erilaisia hyviksi väitettyjä käytänteitä React-sovellusten arkkitehtuurin toteuttamiseen on?
- Kuinka hyvin tutkittavaksi valitut arkkitehtuurikäytänteet sopivat esimerkkisovellukseen?
- Sopisivatko tutkittavat käytänteet paremmin johonkin toisenlaiseen sovellukseen?

4 Esimerkkisovelluksen toteutus

4.1 Sovelluksen kuvaus

Tutkittavia käytänteitä arvioitiin kehittämällä yksinkertainen verkkosovellus, jossa voidaan hakea eri sijaintien säätietoja. Säätietojen hakemisen lisäksi sovelluksen käyttäjä voi asettaa hakemiaan sijainteja ”suosikkeihinsa”, jolloin niiden säätiedot näyttäville sivuille pääsee suoraan sovelluksen etusivulta. Sovellukseen luotiin myös asetuksia, joilla käyttäjä voi muokata sen sisältöä ja ulkoasua.

Sovelluksen käyttöliittymä luotiin käyttäen Reactia. JavaScriptin sijaan sovelluksen kehittämiseen valittiin kielen tyypitetty versio TypeScript. Säätiedot ja sijainnit sovellus saa Open-Meteo-sivuston vapaasti käytettävistä API-rajapinnoista. Sovelluksessa käytettiin Reactin lisäksi myös muutamia muita kirjastoja sen eri toiminnallisuuksien toteuttamiseen. Esimerkiksi globaalin tilanhallinnan luomiseen valittiin Redux-kirjasto, ja sovelluksen reititys toteutettiin React Routerilla.

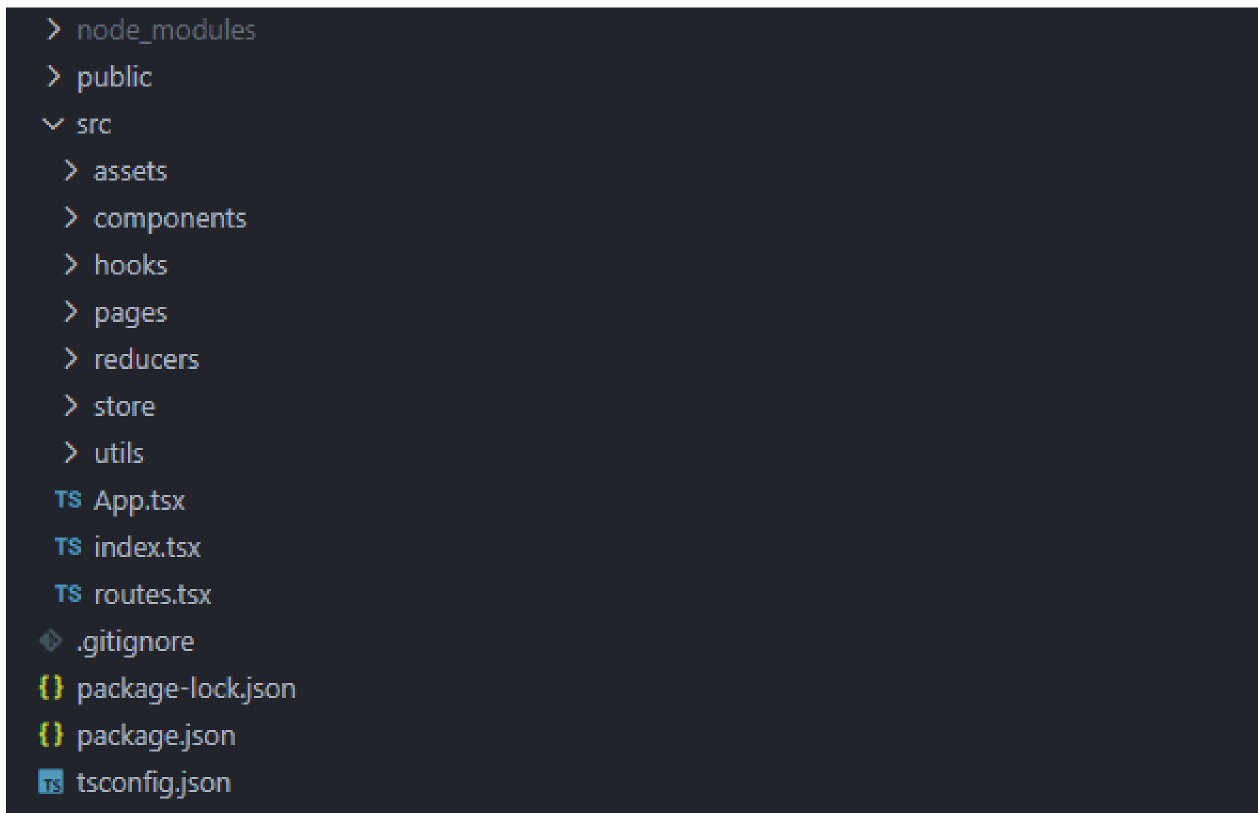
4.2 Kansiorakenne

Yksi ensimmäisistä asioista, joita on hyvä suunnitella React-sovelluksen kehittämistä aloittaessa on sovelluksen kansiorakenne. Oikein toteutetut React-sovellukset koostuvat lukuisista tiedostoista, joista jokaiseen on kirjoitettu tietyn tehtävän toteuttavaa koodia. Mitä enemmän näitä tiedostoja kerääntyy, sitä tärkeämpää on organisoida ne omiin kansioihinsa hyvin.

React antaa sitä käyttäville kehittäjille hyvin vapaat kädet jakaa sovellustensa koodin tiedostoihin kuten haluavat, ja säilöä nämä tiedostot valitsemallaan tavalla. Tästä syystä on tärkeää valita omaan projektiinsa sopiva kansiorakenne jo hyvissä ajoin, jotta tiedostojen organisointi pysyy luontevana ja selkeänä myös niiden määrän kasvaessa. Hyvä kansiorakenne mahdollistaa myös tiedostojen sijaintien helpon muuttamisen tarvittaessa.

Ei ole yhtä oikeaa tapaa organisoida React-sovellusten tiedostoja. Monia erilaisia kansiorakenteita on suositeltu, mutta niissä kaikissa toistuu yksi yhteinen teema: tiedostojen jakaminen niiden tehtävien mukaan. React-sovellus voi sisältää monia erilaisia tiedostoja, jotka suorittavat erilaisia tehtäviä. Kun samanlaisia tehtäviä ajavat tiedostot löytyvät yhden hakemiston alta, niiden hallinta helpottuu, mikä otettiin huomioon myös esimerkkisovelluksen tiedostojen organisoinnissa.

Reactissa sovelluksen varsinainen lähdekoodi luodaan src-nimiseen kansioon. Sen sisällä olevat tiedostot voidaan sitten jakaa eri kansioihin niiden tehtävän mukaan. Myös esimerkkisovellukseen valittiin tätä periaatetta noudattava kansiorakenne (ks. kuvio 9).



Kuvio 9. Esimerkkisovelluksen kansiorakenne

Ensimmäisenä src-kansion sisältä löytyy assets-niminen kansio. Sen tarkoituksena on sisältää kaikki koodiin liittymättömät tiedostot. Esimerkkisovelluksessa siihen laitettiin sovelluksen globaalit tyylit sisältävä CSS-tiedosto, sekä sovelluksen logon sisältävä kuvatiedosto. Tällaisten tiedostojen lisäksi sinne voitaisiin laittaa muun muassa mahdolliset fonttitiedostot.

Seuraavana src-hakemiston sisällä on components-kansio, joka sisältää sovelluksen komponentit. React-sovelluksissa käyttöliittymä on hyvä jakaa hyvin pieniin komponentteihin, minkä vuoksi niitä syntyy nopeasti hyvin suuri määrä. Tällöin niiden kaikkien sijoittaminen suoraan komponenteille tarkoitetun kansion sisään ei ole järkevää, vaan ne on hyvä ryhmitellä vielä tarkemmin.

Esimerkkisovelluksessa komponentteja luotiin noin 30, mikä on vielä suhteellisen pieni määrä, mutta silti parhaaksi nähtiin jakaa ne components-kansion sisällä oleviin pienempiin kansioihin, joista jokainen sisältää tiettyyn sovelluksen sivuun tai ominaisuuteen liittyvät komponentit. Esimerkiksi kaikki sovelluksen asetussivuun liittyvät komponentit sijoitettiin omaan kansioonsa. Näin

components-kansio pysyy organisoituneena ja selkeänä navigoida. Jokainen komponentti on tämän lisäksi laitettu omaan kansioonsa, jonka nimi on sama kuin komponentilla itsellään. Tämän ansiosta jokaiseen yksittäiseen komponenttiin liittyvät tiedostot, kuten mahdolliset tyylitiedostot ja yksikkötestit, voidaan säilyttää suoraan komponentin yhteydessä.

Components-kansion jälkeen src-kansiosta löytyy kansio sovelluksen custom hookeille. Niitä ei esimerkkisovellukseen tullut kovin montaa, joten ne laitettiin suoraan hooks-kansion alle. Jos hookeja olisi enemmän, niidenkin ryhmitteleminen erilaisiin kansioihin voisi olla järkevää.

Seuraava kansio src-kansion sisällä on nimeltään pages. Reactissa sovelluksen eri sivut luodaan komponentteina, joiden sisään laitetaan sivulle kuuluvat komponentit. Nämä sivukomponentit ovat selkeyden vuoksi hyvä erottaa muista komponenteista, minkä vuoksi niille tehtiin oma kansionsa.

Seuraavat kansiot, store ja reducers, liittyvät esimerkkisovelluksessa käytettyyn tilanhallintakirjasto Reduxiin. Store-kansio on tarkoitettu Reduxin storelle ja siihen liittyville tiedostoille. Store sisältää Reduxilla luodun tilan. Reducers-kansioon sijoitettiin funktiot, joita käytetään muokkaamaan Reduxilla luotua tilaa.

Viimeisenä kansiona src-hakemiston alla on utils-kansio. Siihen laitettiin kaikki sellaiset funktiot ja muut kooditiedostot, jotka oli selkeyden vuoksi hyvä erottaa komponenteista. Se sisältää muun muassa TypeScriptin vaatimaan tyypitykseen käytetyt rajapinnat eli interfacet. Kaikki utils-kansion sisältämät tiedostot ryhmiteltiin vielä alikansioihin niiden tehtävän mukaan.

4.3 Absoluuttinen importointi

Kun React-sovelluksessa on paljon kansioita, tiedostojen koodin tuominen toisiin tiedostoihin voi kasvaa monimutkaiseksi, etenkin jos tiedosto täytyy hakea monen kansion päästä. Tiedostojen tuomisen eli importoimisen helpottamiseksi on kuitenkin käytänne, jota kutsutaan absoluuttiseksi importoinniksi (absolute imports). Se tarkoittaa, että kun tiedostoon tuodaan jotain toista tiedostoa, importoitavan tiedoston polkua ei aloiteta importoivan tiedoston sijainnista, vaan sovelluksen juurikansiosta.

React-sovelluksissa absoluuttinen importointi on hyvä asettaa alkamaan src-kansiosta, koska kaikki sovelluksessa käytettävät kooditiedostot löytyvät tyypillisesti sen alta. Esimerkkisovelluksessa tämä määritettiin sovelluksen juurikansiosta löytyvässä tsconfig.json-tiedostossa. Se on tiedosto, jossa määritetään TypeScript-kääntäjään liittyvät asetukset.

Komponenttien importointipolkuja yksinkertaistettiin myös toisella yleisellä ratkaisulla. Jokainen components-kansion sisällä oleva, tiettyyn ominaisuuteen liittyvät komponentit sisältävä kansio, sisältää myös index.tsx-nimisen tiedoston, jossa jokainen kyseisen kansion sisältämä komponentti exportoidaan uudestaan. Koska tiedostoja importoidessa index-nimisiä tiedostoja ei tarvitse sisällyttää polun loppuun, importoinnissa käytettävä polku yksinkertaistuu (ks. kuvio 10).

```
1   // Yksinkertaistamaton, relatiivinen import-lause
2   import WeatherDataWind from '../../components/Weather/WeatherDataSun';
3
4   // Yksinkertaistettu, absoluuttinen import-lause
5   import { WeatherDataSun } from "components/Weather";
6
```

Kuvio 10. Import-lauseiden vertailu

Absoluuttisen importoinnin tarkoituksena on tehdä koodista helpommin luettavaa ja ylläpidettävää. Sen tärkeys korostuu etenkin sovelluksen monimutkaistuessa, kun sen tiedosto- ja kansiomäärät kasvavat. Käytänne auttaa myös välttämään virheitä, kun tiedostoja täytyy siirtää tai nimetä uudestaan, ja mahdollistaa useiden komponenttien importoinnin samasta sijainnista yhdellä import-lauseella.

4.4 Komponentit

Kaikkein yleisin ja tärkein React-sovellusten komponentteihin liittyvä ohje lienee niiden pitäminen yksinkertaisina. Ihanteellisesti yksi komponentti vastaa yhden asian toteuttamisesta. Näin sovelluksen koodi pysyy selkeänä ja yksinkertaisena. Myös funktiokomponentteja on usein hyvä suosia luokkakomponenttien sijaan tästä syystä.

Esimerkkisovelluksessakin käyttöliittymä pyrittiin jakamaan hyvin pieniin komponentteihin, jotta niistä jokainen olisi mahdollisimman yksinkertainen ja helppolukuinen. Esimerkiksi sovellukseen tehty navigaatiopalkki, jossa on sovelluksen logo, hakukenttä ja linkki asetuksiin, on rakennettu yhteensä seitsemästä komponentista. Sovelluksen logo, joka toimii myös linkkinä takaisin etusivulle, on esimerkiksi oma yksittäinen komponenttinsa (ks. kuvio 11).

```
1   import styled from '@emotion/styled';
2   import { Link } from 'react-router-dom';
3
4   const Logo = styled.img({
5     width: '100%',
6     maxWidth: '300px',
7   });
8
9   const NavBarLogo = () => {
10    return (
11      <Link to="/">
12        <Logo src={require('assets/logo.png')} alt="Logo" />
13      </Link>
14    );
15  };
16
17  export default NavBarLogo;
18
```

Kuvio 11. Komponentti, joka renderöi logon

Kun koko navigaatiopalkki on jaettu vastaavanlaisiin pieniin komponentteihin, navigaatiopalkin varsinainen komponentti voidaan koostaa ilman yhtään sovelluslogiikkaa; kaikki navigaatiopalkkiin kuuluva logiikka löytyy sen alikomponenteista. Samanlaista periaatetta pyrittiin noudattamaan sovelluksen muissakin osissa. Tämä auttaa pitämään kaikki yksittäiset komponentit selkeinä.

Kuviossa 11 on kuvattu toinenkin esimerkkisovelluksen komponentteihin sovellettu käytänne, eli komponentin elementtien tyylittelyn kirjoittaminen tiedoston koodiin. Tätä suunnittelumallia kutsutaan nimellä ”CSS-in-JS”. Yksi mallin hyödyistä on siinä, että siten kirjoitetut tyylit ulottuvat vain niihin komponentteihin, joissa niitä käytetään. Tavallisiin CSS-tiedostoihin luodut tyylisäännöt ulottuvat koko sovelluksen laajuisesti, vaikka tiedostot tuotaisiinkin vain tiettyyn komponenttiin, mikä voi johtaa ongelmiin tyylisääntöjen nimeämisen kanssa.

Esimerkkisovelluksessa komponenttien tyylien kirjoittaminen koodiin toteutettiin Emotion-nimistä kirjastoa käyttäen. Sillä voidaan luoda styled-elementtejä, joihin voidaan kirjoittaa CSS:n kaltaisia tyylisääntöjä (ks. kuvio 12). Kaikki esimerkkisovelluksen yksittäisiä komponentteja koskevat tyylit luotiin Emotionilla, mutta laajasti ympäri sovellusta käytettävät tyylit kirjoitettiin sovelluksen globaalit tyylit sisältävään index.css-tiedostoon.

```
15
16    const NavigationBarContent = styled.div({
17      width: '95%',
18      height: '100%',
19      maxWidth: '600px',
20      display: 'flex',
21      alignItems: 'center',
22      justifyContent: 'space-between',
23      boxSizing: 'border-box',
24      margin: 'auto',
25    });
26
```

Kuvio 12. Emotion-kirjastolla tyylitetty div-elementti

Toinen esimerkkisovelluksessa sovellettu komponentteihin liittyvä suunnittelumalli on komponenttien jakaminen sovelluslogiikan sisältäviin konttikomponentteihin (container component) ja käyttöliittymän elementtejä esittäviin presentaatiokomponentteihin (presentational component). Konttikomponentit käsittelevät vain dataa. Ne eivät suoraan palauta käyttöliittymäelementtejä. Sen sijaan ne sisältävät presentaatiokomponentteja, jotka käyttävät niiden dataa. Kyseinen malli voidaan luokitella rakennemallien kategoriaan, koska se kuvaa, kuinka yksittäisistä osista voidaan rakentaa suurempia kokonaisuuksia.

Esimerkkisovelluksessa tämä suunnittelumalli on käytössä muun muassa sovelluksen asetussivun komponenteissa. Asetusten arvojen muuttamisesta vastaava logiikka löytyy SettingsPageContainer-nimisestä komponentista, joka toimii konttikomponenttina (ks. kuvio 13). Komponentti ei suoraan palauta elementtejä sovelluksen käyttöliittymään, vaan se palauttaa vain kaksi SettingsToggler-komponenttia, joille se antaa propseina niiden käyttämän datan. Presentaatiokomponenttina

toimiva SettingsToggler sen sijaan ei pidä sisällään yhtään logiikkaa, vaan se vain renderöi tekstiä sekä painikkeen, jolla tiettyä asetusta voidaan vaihtaa.

```
1  import { SettingsToggler } from 'components/SettingsPage';
2  import { useDispatch, useSelector } from 'react-redux';
3  import { setTheme, setTempUnit } from 'reducers/settingsSlice';
4  import { Settings } from 'utils/interfaces';
5
6  const SettingsPageContainer = () => {
7    const { useDarkTheme, useFahrenheit } = useSelector(
8      (state: Settings) => state.settings
9    );
10   const dispatch = useDispatch();
11
12   const handleToggle = (setting: string) => {
13     if (setting === 'theme') {
14       dispatch(setTheme(!useDarkTheme));
15     } else if (setting === 'unit') {
16       dispatch(setTempUnit(!useFahrenheit));
17     }
18   };
19
20   return (
21     <>
22       <SettingsToggler
23         title="Dark theme"
24         desc="Use dark theme instead of light"
25         value={useDarkTheme}
26         onChange={() => handleToggle('theme')}
27       />
28
29       <SettingsToggler
30         title="Temperature unit"
31         desc="Use Fahrenheit instead of Celsius"
32         value={useFahrenheit}
33         onChange={() => handleToggle('unit')}
34       />
35     </>
36   );
37 };
38
39 export default SettingsPageContainer;
40
```

Kuvio 13. Asetussivun logiikan sisältävä konttikomponentti

Konttikomponenttien ja presentaatiokomponenttien tarkoituksena on auttaa luomaan separation of concerns -periaatteen mukaista koodia. Kun koodi on jaettu osiin sen eri tehtävien mukaan, koodin ylläpitämisestä, muokkaamisesta ja testaamisesta tulee helpompaa. Tiettyyn tehtävään luodut komponentit ovat myös helpommin uudelleenkäytettäviä. (Paralkar 2022.)

Toinen esimerkkisovelluksessa kokeiltu malli, jonka tarkoitus on jakaa koodia osiin eri tehtävien mukaan, on nimeltään ”render props”. Siinä komponentille syötetään propsina funktio, joka määrittelee, mitä kyseinen komponentti renderöi. Oikein toteutettuna tämä mahdollistaa datan monimutkaisemman käsittelyn irrallaan sen esittävästä komponentista, mikä lisää komponentin joustavuutta. Render props voidaan luokitella käyttäytymismalleihin, sillä se liittyy osien omiin vastuisiin ja osien väliseen vuorovaikutukseen.

Esimerkkisovelluksessa render props -mallia käytettiin näyttämään käyttäjän suosikkeihin asettamien sijaintien lista sovelluksen etusivulla. Etusivun sivukomponentissa näytetään FavoritesListContainer-komponentti, jolle annetaan propsina funktio, joka renderöi listan varsinaisen sisällön (ks. kuvio 14). Funktioon on lisätty myös items-parametri, johon FavoritesListContainer-komponentissa voidaan sitten syöttää arvoja.

```
7   const HomePage = () => {
8     return (
9       <>
10        <h3>Favorite locations:</h3>
11
12        <FavoritesListContainer
13          render={(items: Location[]) => <FavoritesListContent items={items} />}
14        />
15      </>
16    );
17  };
18
```

Kuvio 14. Komponentin renderöivän funktion syöttäminen propsina

FavoritesListContainer-komponentti sen sijaan ei suoraan palauta käyttöliittymäelementtejä itse (ks. kuvio 15). Sen palautusfunktiossa kutsutaan sen propsina saamaa render-lausetta, joka sitten luo siihen määritetyn komponentin. Renderöitävälle komponentille annetaan argumenttina items-niminen taulukko, joka sisältää tallennetut suosikkisijainnit.

```
 9
10  const FavoritesListContainer = ({ render }: Props) => {
11    const items = useSelector((state: FavoritesList) => state.favorites.items);
12
13    return render(items);
14  };
15
```

Kuvio 15. Render-funktion propsina saava komponentti

Render props -suunnittelumallin käyttämisen ansiosta suosikkisijainnit sisältävää taulukkoa ja sen hakemisen toteuttavaa lausetta voidaan käsitellä täysin irrallaan sijainnit renderöivästä komponentista. Tämän lisäksi ratkaisu tekee suosikkisijainnit renderöivästä komponentista helpommin uudelleenkäytettävän, sillä se ei ole millään tavalla yhteydessä datalähteeseensä. Jos sekä datan käsittely että esittäminen toteutettaisiin samassa komponentissa, komponenttia ei voisi käyttää mihinkään muuhun tarkoitukseen.

4.5 Tilanhallinta

Tilanhallinta on tärkeä osa sovelluskehitystä. Oikeaoppisen tilanhallinnan tärkeys korostuu etenkin komponentteihin perustuvissa ohjelmistoissa, kuten Reactilla luoduissa sovelluksissa, koska tilan arvot ovat usein monien komponenttien käytössä. Kuten myös komponentit, on tilanhallintakin hyvä toteuttaa selkeällä ja yksinkertaisella tavalla.

Esimerkkisovelluksessa tilanhallintaan käytetään sekä Reactin sisäänrakennettuja tilanhallinnan keinoja, että Redux-nimistä tilanhallintakirjastoa. Reduxilla luotiin keskitetty, globaali tila sen omaan storeen, kun taas Reactin tilanhallinnan hookeja, kuten useStatea, käytettiin toteuttamaan tietyissä komponenteissa vaadittu paikallinen tila. Reactin hookien avulla luotiin myös custom hookeja, joissa käsitellään sovelluksen tilaa.

Esimerkkisovelluksen Redux-storeen tallennetaan muun muassa sovelluksen asetuksissa muutettavat arvot. Asetussivun konttikomponentissa (ks. kuvio 13) on handleToggle-niminen funktio, joka käsittelee muutoksia sovelluksen asetuksissa. Funktiossa asetusten uudet arvot lähetetään Reduxin storeen dispatch-lauseita käyttäen. Storesta asetuksien arvot voidaan sitten pyytää missä tahansa sovelluksen komponentissa.

Koska esimerkkisovelluksen asetussivu on toteutettu kontti- ja presentaatiokomponenteilla, ja konttikomponentista muokataan Reduxilla hallittavaa tilaa, joka puolestaan muokkaa näkymää, vastaa kyseinen toteutus MVC-arkkitehtuuria. Tässä tapauksessa presentaatiokomponentti vastaa näkymäosuutta (view), konttikomponentti ohjainta (controller) ja Reduxin store mallia (model). Tämänkaltaista arkkitehtuuria pyrittiin noudattamaan muuallakin sovelluksessa.

Arvot tallennetaan Reduxin storeen ja niitä muokataan siivuiksi (slice) kutsutuissa moduuleissa. Siivuihin määritetään alkutila, jota sitten voidaan muokata niihin kirjoitetuissa reducer-funktioissa. Siivujen käyttäminen ja luominen Redux Toolkit -lisäosaan kuuluvalla createSlice-funktiolla on suositeltavaa, koska niiden luominen on yksinkertaista ja siten voidaan vähentää tarpeetonta koodia. (Redux Style Guide n.d.)

On myös suositeltavaa, että yhdessä siivussa käsitellään vain yhteen ominaisuuteen liittyvää tilaa, jotta tilanhallinta pysyy yksinkertaisena. Esimerkkisovelluksessa luotiin kaksi siivua: yksi asetusten tilanhallintaan (ks. kuvio 16) ja toinen käyttäjän suosikkeihinsa lisäämien sijaintien hallintaan. Siivut yhdistetään toisessa tiedostossa yhteen Redux-storeen, joka sitten viedään koko sovelluksen käytettäväksi Provider-elementissä sovelluksen index.tsx-juuritiedostossa.

```
import { createSlice, PayloadAction } from '@reduxjs/toolkit';

type SettingsState = {
  useDarkTheme: boolean;
  useFahrenheit: boolean;
};

const initialState: SettingsState = {
  useDarkTheme: false,
  useFahrenheit: false,
};

const settingsSlice = createSlice({
  name: 'settings',
  initialState,
  reducers: {
    setTheme: (state, action: PayloadAction<boolean>) => {
      localStorage.setItem('useDarkTheme', JSON.stringify(action.payload));
      state.useDarkTheme = action.payload;
    },
    setTempUnit: (state, action: PayloadAction<boolean>) => {
      localStorage.setItem('useFahrenheit', JSON.stringify(action.payload));
      state.useFahrenheit = action.payload;
    },
  },
});

export const { setTheme, setTempUnit } = settingsSlice.actions;

export default settingsSlice.reducer;
```

Kuvio 16. Redux-siivu sovelluksen asetusten tilanhallintaan

4.6 Hookit

Reactin hookit ovat hyvä keino yksinkertaistaa ja parantaa sovellusten logiikkaa. Reactin sisäänrakennetut hookit auttavat luomaan komponenteissa tarvittuja toiminnallisuuksia, mutta myös omista custom hookeista on usein hyötyä. Custom hookeja luomalla sovelluksen komponentteja voidaan selkeyttää erottamalla niistä uudelleenkäytettävää koodia, ja edistää siten separation of concerns -periaatteen mukaista ohjelmointia.

Esimerkkisovellukseen luotiin muutama custom hook. Ne ajavat sellaista koodia, jonka on hyvä olla erillään komponenteista. Yksi esimerkkisovellukseen luoduista hookeista on useFetchWeather, joka hakee ja palauttaa tietyn sijainnin säätiedot Open-Meteon API-rajapinnasta sijainnin koordinaattien mukaan (ks. kuvio 17).

```
1   import { useState, useEffect } from 'react';
2   import { Settings } from 'utils/interfaces';
3   import { useSelector } from 'react-redux';
4   import axios from 'axios';
5
6   const useFetchWeather = (latitude: string, longitude: string) => {
7     const [data, setData] = useState(null);
8     const [loading, setLoading] = useState(true);
9     const [error, setError] = useState(null);
10
11    const useFahrenheit = useSelector(
12      (state: Settings) => state.settings.useFahrenheit
13    );
14    const tempUnit = useFahrenheit ? 'fahrenheit' : 'celsius';
15
16    useEffect(() => {
17      axios
18        .get(
19          `https://api.open-meteo.com/v1/forecast?latitude=${latitude}&longitude=${longitude}&d
20        )
21        .then((res) => {
22          setData(res.data);
23          setLoading(false);
24        })
25        .catch((error) => {
26          setLoading(false);
27          setError(error);
28        });
29    }, [latitude, longitude, tempUnit]);
30
31    return { data, loading, error };
32  };
```

Kuvio 17. Esimerkkisovellukseen luotu custom hook

Custom hookien nimi on hyvä aloittaa use-sanalla, kuten kaikkien Reactin omienkin hookien nimet alkavat. Tämä käytäntö ei ole pakollinen, mutta se on suositeltu, koska sen ansiosta React pystyy erottamaan hookit tavallisista funktioista. Näin React voi tarkistaa, että myös custom hookit noudattavat samoja sääntöjä, joita myös sisäänrakennettujen hookien on noudatettava toimiakseen oikein.

Kuviossa 17 kuvatussa hookissa käytetään Axios-nimistä kirjastoa säätiedot hakevan http-pyynnön tekemiseen. Axios on React-sovelluksissa suosittu ja yleisesti käytetty kirjasto tähän tarkoitukseen. Verrattuna JavaScriptin omaan Fetch API -ominaisuuteen, Axios mahdollistaa samojen asioiden toteuttamisen vähemmällä koodilla ja paremmilla ominaisuuksilla. (Barger 2021a.)

Esimerkkisovelluksen säätiedot hakeva http-pyyntö toteutetaan useEffect-hookin sisällä, jotta hakua ei ajettaisi kuin kerran silloin kun kyseistä hookia kutsutaan. UseEffectin lisäksi custom hook hyödyntää Reactin omaa useState-hookia säilyttämään haun tuloksena saadun datan. UseStatea käytetään hookissa myös tallettamaan tieto siitä, onko haku vielä kesken, sekä http-pyynnöstä mahdollisesti palautunut virheilmoitus. Hyödyntäen komponenttien ehdollista renderöintiä, näitä tiloja käytetään näyttämään haun keskeneräisyydestä kertovaa viestiä sekä mahdollista virheilmoitusta sovelluksen käyttäjälle (ks. kuvio 18). Tämä on melko yleinen käytänne dataa hakevien custom hookien luomisessa.

```
24  const WeatherDataContainer = ({ latitude, longitude }: Props) => {
25    const { data, loading, error } = useFetchWeather(
26      latitude,
27      longitude
28    ) as Fetch;
29
30    if (loading) return <Loading />;
31
32    if (error) return <Error error={error} />;
33
34    return (
35      <>
36        {data && (
37          <div>
38            <Columns>
```

Kuvio 18. Hookin kutsuminen komponentissa ja ehdollinen renderöinti

4.7 Reititys

Reititys, eli näytettävän sisällön vaihtaminen sivun URL-osoitteen mukaan, on yksi ominaisuus, jota Reactissa ei ole sisäänrakennettuna. Reititys voidaan kuitenkin toteuttaa kirjastojen avulla. Hyvin yleisesti käytetty kirjasto React-sovellusten reititykseen on React Router, ja sitä käytettiin myös esimerkkisovelluksessa.

Reitit määritetään tyypillisesti sovelluksen juurikomponentissa React Routeriin kuuluvan Routes-elementin sisään. Näin tehtiin myös esimerkkisovelluksessa, mutta siinä reitit kirjoitettiin toiseen tiedostoon objektitaulukkoon. Sieltä ne tuodaan juurikomponenttiin, jossa reittitaulukko puretaan map-metodin avulla Routes-elementin sisään (ks. kuvio 19). Kun reitit kirjoitetaan toiseen tiedostoon, juurikomponentti säilyy selkeämpänä, ja uusien reittien lisääminenkin yksinkertaistuu.

```
36
37        <ContentContainer>
38          <Routes>
39            {routes.map((route, index) => (
40              <Route {...route} key={index} />
41            ))}
42          </Routes>
43        </ContentContainer>
44      </BrowserRouter>
45    );
46  };
47
```

Kuvio 19. Esimerkkisovelluksen reittien määrittäminen

React Router mahdollistaa myös parametrien lisäämisen reittien osoitteisiin. Niiden avulla osoitteet voivat sisältää muuttuvia osia, joiden arvoksi voidaan asettaa haluttu merkkijono. Parametrien arvoihin pääsee käsiksi reitin osoitteen mukaisessa komponentissa käyttäen React Routeriin kuuluvaa useParams-hookia. Esimerkkisovelluksessa tätä metodia käytettiin sijaintien URL-osoitteiden muodostamisessa. Sovelluksessa jokaisen sijainnin URL-osoitteeseen laitetaan kyseisen sijainnin koordinaatit, nimi sekä mahdollisen laajemman alueen nimi. Näin sijaintien säätietojen hakemiseen tarvittavat tiedot saadaan komponentteihin pelkästä URL-osoitteesta.

Reittien osoitteisiin siirtyminen tapahtuu React Routeriin kuuluvilla Link-elementeillä. Toisin kuin HTML:ään kuuluvat a-linkkielementit, React Routerin linkit eivät aiheuta sivun uudelleenlatautumista, mikä saisi kaikki sovelluksen senhetkiset tilat palautumaan oletusarvoihinsa. Esimerkkisovelluksessa muun muassa sijaintihaun yksittäisen hakutuloksen renderöivä komponentti sisältää React Routerin linkkielementin (ks. kuvio 20). Siihen kirjoitettuun osoitteeseen (/location/...) lisätään hakutuloksen sijainnin säätietojen hakemiseen tarvittavat tiedot.

```
14  const SearchResultsItem = ({ result }: Props) => {
15    return (
16      <Link
17        to={`/location/${result.latitude}/${result.longitude}/${result.name}${
18          result.admin1 ? '/' + result.admin1 : ''
19        }`}
20      >
21        <Item>
22          <span className="bold">{result.name}</span>
23
24          {result.admin1 ? `, ${result.admin1}` : ''}
25          {result.country ? `, ${result.country}` : ''}
26        </Item>
27      </Link>
28    );
29  };
30
```

Kuvio 20. Linkkielementti hakutuloksen renderöivässä komponentissa

5 Tulokset

Opinnäytetyön tavoitteena oli tutkia yleisiä React-sovellusten arkkitehtuuriin liittyviä hyviä käytänteitä soveltamalla niitä esimerkkisovelluksen kehitykseen. Vaikka esimerkkisovelluksesta jouduttiin tekemään suhteellisen pieni ja yksinkertainen, sen kehityksessä pystyttiin soveltamaan lukuisia yleisiä käytänteitä. Kehitystutkimuksen valitseminen työn tutkimusmenetelmäksi soveltui aiheen tutkimiseen siis hyvin, ja asetettuihin tutkimuskysymyksiin löydettiin vastauksia.

Esimerkkisovelluksen kehityksessä sovelletut käytänteet koettiin pääosin hyviksi. Hyvien käytänteiden noudattaminen saattaa hidastaa sovelluskehitystä sen alkuvaiheissa, mikä huomattiin myös esimerkkisovelluksen kehittämisessä, mutta pitkällä aikavälillä niiden noudattaminen on kuitenkin hyödyllistä myös ajallisesti. Huonosti suunnitellun ja rakennetun sovelluksen kehittäminen ja yllä-

pitäminen voi muodostua hitaaksi ja vaikeaksi, kun taas hyviä arkkitehtuuriratkaisuja noudattavassa sovelluksessa se on paljon helpompaa. Tämä onkin yksi tärkeimmistä syistä, miksi hyvään sovellusarkkitehtuuriin panostaminen nähdään kannattavana.

Esimerkkisovellus on sovelluksena vielä melko pieni ja yksinkertainen, mutta valittu kansiorakenne koettiin sille hyvin sopivaksi. Sen ansiosta sovelluksen tiedostot pysyivät hyvin organisoituina, mikä teki kehittämisestä helpompaa. Myös sovelluksen mahdollisen jatkokehityksen kannalta kyseinen kansiorakenne on toimiva, koska sovelluksen yksittäisiin osasiin voidaan helposti tehdä muutoksia ja korjauksia muuttamatta sovelluksen koko rakennetta. Kyseisen kansiorakenteen koetaan sopivan hyvin esimerkkisovelluksen kokoisille tai hieman suuremmille sovelluksille. Merkittävästi suuremmissa sovellusprojekteissa kannattaa kuitenkin harkita tiedostojen jakamista vielä selkeämmin. Paljon pienemmissä sovelluksissa tällainen kansiorakenne voi sen sijaan olla tarpeettoman yksityiskohtainen.

Esimerkkisovelluksen komponentteihin sovelletut käytänteet ovat sen sijaan pääosin universaaleja siinä, että niiden noudattaminen on järkevää sovelluksen koosta riippumatta. Komponenttien pitäminen mahdollisimman yksinkertaisina helpottaa sekä niiden luomista, muokkaamista että ylläpitämistä. React-komponenteista on parempi tehdä liian pieniä kuin liian suuria ja monimutkaisia.

Komponentteihin sovelletut suunnittelumallit koettiin myös hyviksi. Etenkin komponenttien jako kontti- ja presentaatiokomponentteihin nähtiin erittäin järkevänä ratkaisuna. Sen soveltaminen kannattaa etenkin sellaisiin komponentteihin, joissa vaaditaan paljon koodia. Kun komponenttiin liittyvä sovelluslogiikka on erillään sitä käyttävistä käyttöliittymäelementeistä, koodi pysyy helppolukuisena ja helposti käsiteltävänä. Kaikki muukin separation of concerns -periaatteen mukainen ohjelmointi on hyvin toteutettuna suositeltavaa sovelluskehitysprojekteissa.

CSS:n kirjoittaminen koodiin on toinen malli, jota sovellettiin esimerkkisovellukseen laajasti. Sen käytöstä oli tiettyjä hyötyjä, mutta sitä ei koettu välttämättömäksi ratkaisuksi. Käytänteen mukaisesti luodut tyylit eivät vaikuta koko sovellukseen globaalisti kuten CSS-tiedostoihin luodut tyylit, mikä vähensi virheiden sattumista. Tätä käytännettä voidaan suositella erityisesti keskisuuriin ja suuriin sovelluksiin, joissa käyttöliittymän tyylien kirjoittaminen voi muuten muodostua monimutkaiseksi. Emotion koettiin hyväksi kirjastoksi CSS-in-JS:n toteuttamiseen, koska sillä tyyliteltyjen

elementtien luominen onnistui hyvin intuitiivisella tavalla, ja nämä elementit pystyttiin halutessa erottamaan omiin tiedostoihinsa helposti. Emotion-kirjastoa voidaan suositella sovelluksiin, joissa komponenttien tyylejä halutaan luoda tähän tapaan.

Esimerkkisovellus ei vaatinut erityisen monimutkaista tilanhallintaa, mutta siihen sovelletut tilanhallinnan käytänteet nähtiin kuitenkin hyödyllisiksi. Redux koettiin sovellukseen hyvin sopivaksi kirjastoksi keskitetyn tilanhallinnan luomiseen. Reduxin käyttö vaatii melko paljon koodia, mutta esimerkkisovelluksessa se koettiin kuitenkin hallittavaksi. Reduxiin kuuluvien siivujen hyödyntäminen nähtiin erityisen hyödylliseksi. Jokaisessa sovellusprojektissa on kuitenkin hyvä arvioida, mikä tilanhallintakirjasto sopisi siihen parhaiten, ja pohtia, onko sellainen edes tarpeen. Yksinkertaisimmissa sovelluksissa keskitettyä tilanhallintaa ei välttämättä tarvita.

Hookit ovat yksi Reactin keskeisimmistä ja hyödyllisimmistä ominaisuuksista. Esimerkkisovelluksen custom hookit auttoivat merkittävästi komponenttien ja sovelluslogiikan erottamisessa. Hookien käyttö ei vain yksinkertaistanut ja selkeyttänyt komponentteja, vaan teki myös niistä erotetusta koodista helposti uudelleenkäytettävää. Sekä Reactin sisäänrakennetuista hookeista että custom hookeista voi olla hyötyä missä tahansa React-projektissa, joten niiden käyttöä voidaan suositella.

Kaiken kaikkiaan esimerkkisovellukseen sovellettuja käytänteitä pidettiin erittäin hyödyllisinä sen arkkitehtuurille. Sovelluksen kehittämisen koettiin etenevän selkeästi, ja sovelluksen mahdollinen jatkokehitys ja ylläpito nähdään helppoina toteuttaa. Käytetyt käytänteet nähdään siis sopivina etenkin esimerkkisovelluksen kaltaisiin pieniin tai keskisuuriin React-sovelluksiin. Monet käytänteistä ovat myös yleispäteviä ja sopivat kaikenlaisiin React-sovelluksiin. Jokaisessa projektissa on kuitenkin hyvä pohtia, millaiset käytänteet edistävät sen tavoitteiden saavuttamista parhaiten.

6 Pohdinta

6.1 Tulosten arviointi suhteessa tietoperustaan

Opinnäytetyön tietoperustassa käsiteltiin sovellusarkkitehtuuria yleisellä tasolla, ja hyvin luodun arkkitehtuurin hyötyjä sovelluskehityksessä. Hyvällä arkkitehtuurilla sovelluksesta saadaan luotua helpommin kehitettävä, muokattava ja ylläpidettävä. Kehitetyn esimerkkisovelluksen kohdalla

näin koettiin käyvän, sillä sovelluksen kehittämisen nähtiin etenevän tasaisesti ja selkeästi. Sovelluksen mahdollinen jatkokehitys ja ylläpitäminen nähdään helpompina verrattuna siihen, jos sen kehityksessä ei olisi käytetty tutkittavia arkkitehtuurikäytänteitä.

Esimerkkisovelluksessa käytettiin työn tietoperustassa esiteltyä MVC-arkkitehtuuria muistuttavaa ratkaisua paljon. Koodin jakaminen eri tehtävistä vastaaviin osiin auttaa muun muassa pitämään koodin selkeänä, helpommin muokattavana ja uudelleenkäytettävänä. Nämä hyödyt havaittiin myös esimerkkisovelluksessa, sillä koodin jokaisen yksittäisen osan muokkaaminen onnistui helposti muuttamatta toisten osien koodia. Samaa voidaan sanoa myös monista esimerkkisovelluksessa käytetyistä suunnittelumalleista. Esimerkiksi sovellukseen sovellettu kontti- ja presentaatiokomponenttien rakennemalli, mikä toimi osana MVC:n kaltaisen arkkitehtuurin luomista, auttoi pitämään koodin selkeänä ja helposti muokattavana.

Työn tietoperustassa korostetiin moneen otteeseen myös sitä, kuinka React antaa kehittäjille hyvin vapaat kädet rakentaa sovelluksensa kuten he haluavat. Reactissa käytännössä minkä tahansa asian voi toteuttaa lukuisilla erilaisilla tavoilla. Tämä vapaus on yksi Reactin suurista hyödyistä, mutta sen vuoksi React-kehittäjiltä vaaditaan myös paljon osaamista, jotta sovellusten koodi pysyy laadukkaana. Tämä huomattiin myös esimerkkisovelluksen kehittämisessä ja työn tuloksissa. Vaikka esimerkkisovelluksen kehityksessä tutkitut käytänteet nähtiinkin pääosin hyödyllisinä, on mahdotonta sanoa, olivatko ne valittavissa olleista vaihtoehdoista parhaat, vai olisivatko jotkin muut ratkaisut toimineet vielä paremmin.

6.2 Luotettavuus ja eettisyys

Opinnäytetyössä löydettiin vastauksia siinä määritettyihin tutkimuskysymyksiin, mutta vastausten arvioinnissa on kuitenkin otettava huomioon muutamia seikkoja, jotka voivat vaikuttaa tulosten luotettavuuteen. Ensinnäkin kirjoittajan kokemus React-sovelluskehityksestä on vielä suhteellisen lyhyt. On mahdollista, että kokeneemmalla React-kehittäjällä voisi olla erilaisia vastauksia ja näkökulmia tutkimuskysymyksiin. On myös huomioitava, että tämä oli kirjoittajalle ensimmäinen kerta monien esimerkkisovellukseen sovellettujen käytänteiden käytössä, joten on mahdollista, ettei niitä käytetty parhailla mahdollisilla tavoilla.

Toiseksi on huomioitava, että kirjoittaja työskenteli esimerkkisovelluksen parissa yksin. Yksi hyvän sovellusarkkitehtuurin hyödyistä on siinä, että sovelluksen kehittäminen onnistuu helposti myös tiiminä, jossa jokaisella on omat vastuualueensa. On mahdollista, että työssä määritettyihin tutkimuskysymyksiin olisi saatu vielä perusteellisempia vastauksia, jos esimerkkisovellusta olisi kehitetty tiiminä.

Kolmanneksi on huomioitava, että kehitetty esimerkkisovellus on suhteellisen pieni ja yksinkertainen. Arkkitehtuurikäytänteiden suurimmat hyödyt tulevat parhaiten esiin vasta suurissa sovelluksissa, joissa niiden noudattaminen on huomattavasti tärkeämpää kuin pienemmissä sovelluksissa. Vaikka hyvien arkkitehtuurikäytänteiden noudattamisesta on hyötyä myös pienille sovelluksille, on mahdollista, että esimerkkisovelluksessa käytettyjen käytänteiden toimivuus olisi tullut parhaiten ilmi vasta sitä laajennettaessa suuremmaksi.

Tutkimuksen laadun ja eettisyyden varmistamiseksi opinnäytetyön tekemisessä noudatettiin hyvää tieteellistä käytäntöä. Työhön haettiin tietoa muun muassa Jamkin verkkokirjaston (janet.finna.fi) eri tietokannoista, ja monia lähteitä löydettiin myös avoimesta internetistä. Työssä käytetyt lähteet valittiin huolellisesti, ja niitä arvioitiin kriittisesti. Niiden valinnassa kiinnitettiin huomiota niiden ajankohtaisuuteen, puolueettomuuteen ja tekijöiden asiantuntijuuteen. Lähteistä poimitut tiedot on merkitty lähdeviittauksilla, ja kaikki käytetyt lähteet on listattu lähdeluetteloon.

Opinnäytetyössä ei käsitellä henkilötietoja, eikä siihen liity salassa pidettävää aineistoa. Työllä ei ollut toimeksiantajaa. Työn tekemiseen ei tarvittu tutkimuslupaa.

6.3 Johtopäätökset ja kehittämisehdotukset

Työn johtopäätöksenä voidaan todeta, että sovelletut arkkitehtuurikäytänteet sopivat kehitettyyn esimerkkisovellukseen hyvin. Tutkittujen käytänteiden nähdään sopivan myös muihin esimerkkisovelluksen kaltaisiin React-sovelluksiin, etenkin samankokoisiin. Monet tutkitut käytänteet, kuten separation of concerns -periaate ja siihen liittyvät eri mallit, ovat universaalisti sovelluskehitykseen liittyviä hyviä käytänteitä, joista on hyötyä kaikissa sovellusprojekteissa niiden koosta ja tyypistä huolimatta. Toiset tutkitut käytänteet, kuten kansiorakenne ja tilanhallintakirjaston käyttäminen, ovat sen sijaan enemmän kehitettävän sovelluksen koosta ja tyypistä riippuvaisia. Jokaisessa sovelluksessa on hyvä arvioida, millaiset ratkaisut edistävät sen tavoitteiden saavuttamista parhaiten.

Vaikka opinnäytetyössä esitettyihin kysymyksiin saatiin vastauksia, työssä tehtyä tutkimusta jouduttiin kuitenkin rajaamaan resurssien rajallisuuden vuoksi. Tutkimusta voitaisiin siis kehittää vielä pidemmälle. Hyvien arkkitehtuurikäytänteiden perusteellisempi tutkimus vaatisi esimerkiksi niiden soveltamista useampiin erityyppisiin ja -kokoisiin sovelluksiin. Näin niiden soveltuvuudesta voitaisiin tehdä vielä perustellumpia ja yleispätevämpiä johtopäätöksiä.

Toinen tapa, jolla tutkimuksesta saataisiin vielä parempia tuloksia, olisi sovelluksen kehittäminen tiiminä. Hyvät käytänteet helpottavat monipuolisen tiimin työskentelyä yhden sovelluksen parissa, joten käytänteiden perusteellinen tutkiminen vaatisi niiden arviointia myös tiimityöskentelyn kontekstissa. Opinnäytetyössä tämä ei ollut mahdollista, mutta tutkimusta voitaisiin jatkokehittää luomalla sovellus ryhmässä hyviä käytänteitä noudattaen ja arvioiden.

Lähteet

2022 Developer Survey. 2022. StackOverflow’n vuoden 2022 kehittäjäkysely. Viitattu 15.2.2023. https://survey.stackoverflow.co/2022/.

Barger, R. 2021a. How To Use Axios With React: The Definitive Guide (2021). Artikkeli FreeCode-Camp-sivustolla. Julkaistu 13.7.2021. Viitattu 17.4.2023. https://www.freecode-camp.org/news/how-to-use-axios-with-react/.

Barger, R. 2021b. Is React a Library or a Framework? Here’s Why it Matters. Artikkeli FreeCo-deCamp-sivustolla. Julkaistu 12.4.2021. Viitattu 13.4.2023. https://www.freecode-camp.org/news/is-react-a-library-or-a-framework/.

Built-in React Hooks. N.d. Reactin virallinen dokumentaatio. Viitattu 9.3.2023. https://beta.reactjs.org/reference/react.

Components and Props. N.d. Reactin virallinen dokumentaatio. Viitattu 6.3.2023. https://reactjs.org/docs/components-and-props.html.

Dhaduk, H. 2020. 10 Software Architecture Patterns You Must Know About. Simformin sivustolla julkaistu blogikirjoitus. Julkaistu 4.7.2020. Viitattu 14.3.2023. https://www.simform.com/blog/software-architecture-patterns/.

Elliott, E. 2023. Top JavaScript Frameworks and Technology 2023. Medium-sivustolla julkaistu ar-tikkeli. Julkaistu 20.2.2023. Viitattu 18.4.2023. https://medium.com/javascript-scene/top-ja-vascript-frameworks-and-technology-2023-4e4a06d6be93.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Boston: Addison-Wesley.

Gorton, I. 2011. Essential Software Architecture. Toinen painos. Berliini: Springer. Viitattu 27.2.2023. https://janet.finna.fi, Skillsoft Books ITPro (Skillport Platform).

Gospel, D. 2023. React.js Architecture Pattern: Implementation + Best Practices. Knowledgehut-sivustolla julkaistu blogikirjoitus. Julkaistu 30.1.2023. Viitattu 20.2.2023. https://www.knowledge-hut.com/blog/web-development/react-js-architecture.

Hámori, F. 2022. The History of React.js on a Timeline. RisingStack-sivustolla julkaistu blogikirjoi-tus. Päivitetty 31.5.2022. Viitattu 3.5.2023. https://blog.risingstack.com/the-history-of-react-js-on-a-timeline/.

Introducing JSX. N.d. Reactin virallinen dokumentaatio. Viitattu 6.3.2023. https://reactjs.org/docs/introducing-jsx.html.

Knockout. N.d. Knockoutin virallinen sivusto. Viitattu 16.3.2023. https://knockoutjs.com/.

Kruchten, P. 1995. Architectural Blueprints – The “4+1” View Model of Software Architecture. Julkaisussa IEEE Software. Washington D.C.: IEEE Computer Society, 12, 6, 42–50. Viitattu 1.3.2023. https://janet.finna.fi, IEEE Electronic Library.

Managing State. N.d. Reactin virallinen dokumentaatio. Viitattu 7.3.2023. https://beta.reactjs.org/learn/managing-state.

Minnick, C. 2022. Beginning ReactJS Foundations Building User Interfaces with ReactJS: An Approachable Guide. Hoboken: John Wiley & Sons. Viitattu 8.3.2023. https://janet.finna.fi, Skillsoft Books ITPro (Skillport Platform).

Martin, M. 2023a. MVC Framework Tutorial for Beginners: What is, Architecture & Example. Artikkeli Guru99-sivustolla. Julkaistu 1.2.2023. Viitattu 15.3.2023. https://www.guru99.com/mvc-tutorial.html.

Martin, M. 2023b. MVC vs MVVM – Difference Between Them. Artikkeli Guru99-sivustolla. Julkaistu 28.1.2023. Viitattu 15.3.2023. https://www.guru99.com/mvc-vs-mvvm.html.

Paralkar, K. 2022. Separation of Concerns in React – How to Use Container and Presentational Components. Artikkeli FreeCodeCamp-sivustolla. Julkaistu 6.12.2022. Viitattu 14.4.2023. https://www.freecodecamp.org/news/separation-of-concerns-react-container-and-presentational-components/.

Redux Style Guide. N.d. Reduxin virallinen opas. Viitattu 14.4.2023. https://redux.js.org/style-guide/.

State: A Component’s Memory. N.d. Reactin virallinen dokumentaatio. Viitattu 7.3.2023. https://beta.reactjs.org/learn/state-a-components-memory.

State and Lifecycle. N.d. Reactin virallinen dokumentaatio. Viitattu 6.3.2023. https://reactjs.org/docs/state-and-lifecycle.html.

Sugandhi, A. 2023. Best Front end Frameworks for Web Development. Knowledgehut-sivustolla julkaistu blogikirjoitus. Julkaistu 21.2.2023. Viitattu 18.4.2023. https://www.knowledgehut.com/blog/web-development/front-end-development-frameworks.

The Vue Instance. N.d. Vuen virallinen dokumentaatio. Viitattu 16.3.2023. https://v2.vuejs.org/v2/guide/instance.html.

Toikko, T. & Rantanen, T. 2009. Tutkimuksellinen kehittämistoiminta. Tampere: Tampereen yliopistopaino. Viitattu 18.2.2023. https://trepo.tuni.fi/handle/10024/100802.

Wieruch, R. 2023. React Libraries for 2023. Robin Wieruchin blogi. Julkaistu 21.2.2023. Viitattu 7.3.2023. https://www.robinwieruch.de/react-libraries/.