Saana Valkama

Reactista Angulariin – Oppimisen siirtovaikutus ohjelmistokehysten välillä

Metropolia Ammattikorkeakoulu

Tekniikan ammattikorkeakoulututkinto

Tieto- ja viestintätekniikka

Opinnäytetyö

9.3.2026

Tiivistelmä

Tekijä(t):	Saana Valkama
Otsikko:	Reactista Angulariin – Oppimisen siirtovaikutus ohjelmistokehysten välillä
Sivumäärä:	54 sivua + 1 liitettä
Aika:	9.3.2026
Tutkinto:	Tekniikan ammattikorkeakoulututkinto
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ohjaaja(t):	Lehtori Minna Kivihalme

Tämän opinnäytetyön tehtävänä on tarkastella Reactin ja Angularin keskeisiä eroja sekä oppimisen siirtovaikutusta tilanteessa, jossa React-taustainen kehittäjä alkaa omaksumaan Angularia. Aihe on ajankohtainen, sillä molemmat kehykset ovat suosittuja yksisivusovelluksien työkaluja.

Menetelmänä on projektipohjainen tapaustutkimus, jossa sama sääsovellus toteutettiin identtisissä ympäristöissä sekä Reactin että Angularin avulla. Analyysi on laadullinen ja havainnoi oppimisen siirtovaikutusta Perkinsin ja Salomonin siirtovaikutusteorian pohjalta, määrittäen positiivisen ja negatiivisen siirtovaikutuksen sekä lähi- ja etäsiirtovaikutuksen.

Tulokset osoittavat, että kehyksillä on useita eroja niin konkreettisesti kuin arkkitehtuuritasolla. Myös sekä positiivista että negatiivista siirtovaikutusta havaittiin projektin aikana. Esimerkiksi komponenttipohjainen kehittäminen, sovelluslogiikan abstrahointi ja riippuvuuksien injektion omaksuminen Context API:n kautta siirtyvät positiivisesti. Negatiivisesti puolestaan siirtyi reaktiivinen ohjelmointi ja moduulipohjainen kehittäminen. React-tausta voi vaikuttaa merkittävästi siirtymiseen Angular-kehykseen.

Asiasanat: React, Angular, SPA, Oppimisen siirtovaikutus, Tapaustutkimus

---

Opinnäytetyön alkuperä on tarkastettu Turnitin Originality Check -ohjelmalla.

Abstract

Author(s):	Saana Valkama
Title:	From React to Angular – Learning Transfer Between Software Frameworks
Number of Pages:	54 pages + 1 appendices
Date:	09 March 2026
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Instructor(s):	Minna Kivihalme, Senior Lecturer

The purpose of this thesis is to examine the key differences between React and Angular and to analyze learning transfer in a situation where a developer with a React background begins to adopt Angular. The topic is relevant, as both frameworks are widely used tools for developing single-page applications. The research method is a project-based case study in which the same weather application was implemented in identical environments using both React and Angular.

The analysis is qualitative and examines learning transfer based on Perkins and Salomon's theory of transfer of learning, identifying positive and negative transfer as well as near and far transfer.

The results show that the frameworks differ in several ways, both in practical implementation and at the architectural level. Both positive and negative learning transfer were observed during the project. For example, component-based development, abstraction of application logic, and the adoption of dependency injection through the Context API were transferred positively. In contrast, reactive programming and module-based development showed negative transfer. A React background can therefore significantly influence the transition to Angular.

Keywords: React, Angular, SPA, Transfer of learning, Case study

---

This thesis has been checked using Turnitin Originality Check service.

Sisällys

1 Johdanto 1

2 Kirjallisuuskatsaus 2

2.1 Yksisivusovellukset 2

2.1.1 Yksisivusovellusten ominaisuudet 2

2.1.2 Yleiskatsaus SPA-sovelluksen kehyksiin 4

2.2 React 5

2.2.1 Historia ja kehitys 5

2.2.2 Keskeiset käsitteet 6

2.2.3 Kirjastoympäristö 8

2.2.4 Edut ja haasteet 8

2.3 Angular 9

2.3.1 Historia ja kehitys 9

2.3.2 Keskeiset käsitteet 10

2.3.3 Ekosysteemi 11

2.3.4 Edut ja haasteet 12

2.4 Oppimisen siirtovaikutus 12

2.4.1 Oppimisen siirtovaikutus ilmiönä 12

2.4.2 Positiivinen ja negatiivinen siirtovaikutus 14

2.4.3 Lähi- ja etäsiirtovaikutus 14

2.4.4 Oppimisen siirtovaikutus ohjelmoinnissa 15

3 Tutkimusmenetelmä 16

3.1 Tutkimusasetelma 16

3.2 Aineisto 16

3.3 Analyysitapa 17

3.4 Menetelmän haasteet ja rajoitteet 17

4 Sovelluksen kehittäminen 18

4.1 Sovelluksen toiminnalliset periaatteet 18

4.2 Tärkeimmät toiminnalliset vaatimukset, ei-toiminnalliset vaatimukset ja rajaukset 19

4.3 React-sovelluksen toteutus 20

4.3.1 Projektin alustus 20

4.3.2 Arkkitehtuuri ja kansiorakenne 21

4.3.3 Keskeiset komponentit, palvelut ja tilanhallinta 22

4.3.4 Käytetyt työkalut ja kirjastot 24

4.3.5 Haasteet 24
4.4 Angular-sovelluksen toteutus 25
4.4.1 Projektin alustus 25
4.4.2 Arkkitehtuuri ja kansiorakenne 26
4.4.3 Keskeiset komponentit, palvelut ja tilanhallinta 27
4.4.4 Käytetyt työkalut ja kirjastot 28
4.4.5 Haasteet 29
5 Vertailu 30
5.1 Projektirakenne ja komponenttimalli 30
5.2 Datanhallinta ja reititys 31
5.3 Käyttöliittymän toteutus ja muut erot 33
5.4 Yleinen kehyskohtainen yhteenveto 35
5.5 Kehittäjäkokemus ja oppiminen 39
6 Oppimisen siirtovaikutuksen analyysi 40
6.1 Positiivinen siirtovaikutus 40
6.2 Negatiivinen siirtovaikutus 42
6.3 Lähi- ja etäsiirtovaikutus 43
7 Johtopäätökset 43
7.1 Keskeiset havainnot Reactin ja Angularin eroista 43
7.2 Oppimisen siirtovaikutus Reactista Angulariin 46
7.3 Työn rajaukset ja jatkokehityksen mahdollisuudet 48
Lähteet 50
Liitteet 54
Github linkit 54

1 Johdanto

Kuten muu teknologia, myös web-sovelluskehitys on ollut jatkuvan kehityksen kohteena. Traditionaalisten monisivusovellusten rinnalle on syntynyt interaktiivisia ja intuitiivisesti toimivia yksisivusovelluksia eli SPA-sovelluksia. SPA-sovellukset ovat suosittuja, ja moni sovellus valitsee nykyään SPA-arkkitehtuurin (Prathap & Saravanan, 2019).

SPA-sovellusten kehittämiseen käytetään yleisesti kehyksiä. Erilaisia kehyksiä löytyy melko laajasti, ja suosituimpia kehyksiä ovat React, Angular ja Vue (Emmanni, 2023). Usean kehyksen taitaminen voi olla kehittäjälle etu työmarkkinoilla. Kehyksillä on omat vahvuutensa ja heikkoutensa, joiden tunnistaminen edistää kykyä valita sopivin kehys sovelluksen koon ja tarpeiden mukaan. Myös eri kehittäjäryhmään siirryttäessä kehittäjä saattaa kohdata uuden kehyksen, jolloin sen nopea omaksuminen on hyvin tärkeää.

Tällä opinnäytetyöllä on kaksi eri tavoitetta. Ensimmäinen tavoite on havainnoida kehysten välisiä eroja. Tämä on tärkeää, koska erot vaikuttavat vahvasti sisäiseen arkkitehtuuriin ja kehyksen käyttötapaan. Toinen tavoite on tarkastella oppimisen siirtovaikutusta, kun React-pohjainen kehittäjä alkaa omaksua uutta kehystä, Angularia. Ilmiötä tarkastellaan Perkinsin ja Salomonin siirtovaikutusteorian viitekehyksen kautta, ja mielenkiinto kohdistuu etenkin siihen, mitkä React-taidot siirtyvät Angulariin ja minkälaiset asiat saattavat aiheuttaa hankaluuksia.

Työn keskeinen tavoite on toimia eräänlaisena tukimateriaalina React-perusteet osaavalle kehittäjälle, joka haluaa tehostaa Angularin käyttöönottoa. Työ pyrkii tuottamaan tietoa React-kehittäjän siirtymästä Angulariin, ja näin ollen olemaan osa ohjelmistokehysten oppimista koskevaa keskustelua. Taustatutkimuksen perusteella moni kehyksiin liittyvä tutkimus on tekninen ja suorituskykyä vertaileva, eikä oppimisprosessi ole ollut tutkimuksien keskiössä. Tämä oppinäytetyö

on toteutettu projektipohjaisena tapaustutkimuksena erojen ja siirtovaikutuksen esiintuomiseksi.

Tavoitteiden pohjalta tämän opinnäytetyön tutkimuskysymyksiksi valikoituivat seuraavat:

- Millaisia eroavaisuuksia React- ja Angular-kehysten välillä ilmenee saman sovelluksen toteutuksessa?
- Miten aiempi perustason React-osaaminen vaikuttaa Angularin oppimiseen?
- Millaiset React-taidot siirtyvät positiivisesti tai negatiivisesti Angulariin?

Luvussa 2 lukijalle annetaan yleiskatsaus yksisivusovelluksiin ja oppimisen siirtovaikutukseen, sekä käydään yksityiskohtaisesti läpi React ja Angular. Luku 3 taustoittaa tapaustutkimuksen. Luku 4 esittelee opinnäytetyössä kehitetyn sääsovelluksen ja sen toteutuksen niin Reactilla kuin Angularilla. Tämän jälkeen kehysten välisiä eroja käsitellään yksityiskohtaisesti luvussa 5. Luvussa 6 analysoidaan oppimisen siirtovaikutusta, jota ilmeni projektin aikana. Johtopäätökset keskittyvät kehysten välisiin eroihin ja oppimisen siirtovaikutukseen yleisellä tasolla, sekä tutkimuksen rajoihin.

2 Kirjallisuuskatsaus

2.1 Yksisivusovellukset

2.1.1 Yksisivusovellusten ominaisuudet

Yksisivusovellusten eli SPA-sovellusten kehitys alkoi 2000-luvun alkupuolella (Narender, 2025:1220, Scott, 2016). Kehitys on johtanut siihen, että nämä websovellukset muistuttavat käyttökokemukseltaan yhä enemmän työpöytäsovelluksia (Gupta, n.d.). Tämä kehityssuunta on käyttäjien kannalta myönteinen, sillä

se tarjoaa entistä sujuvamman ja intuitiivisemman käyttökokemuksen — sovellukset ovat interaktiivisia ja latautuvat ilman kokonaista sivunlatausta (Vivek, 2022:272).

Vaihtoehtona SPA-ratkaisuille ovat monisivusovellukset eli MPA-sovellukset, joissa jokainen sivu ladataan erikseen palvelimelta. Arkkitehtuurin valinnalla on keskeinen merkitys, sillä se vaikuttaa merkittävästi sovelluksen suorituskykyyn, käyttäjäkokemukseen ja skaalautuvuuteen. Molemmilla arkkitehtuureilla on vahvuutensa ja heikkoutensa, jotka vaikuttavat arkkitehtuurin valintaan. MPA-ratkaisua suositellaan sovelluksissa, joissa hakukoneoptimoinnin (SEO) maksimointi on tärkeää, sivustolla on laaja ja hierarkkisesti jäsennelty sisältö, ja tietoturva halutaan toteuttaa keskitetysti. MPA-mallin haasteina voidaan pitää vähemmän intuitiivista käyttökokemusta jatkuvien sivunlatauspyyntöjen myötä. Myös yhtenäisen käyttöliittymän luominen on haastavampaa. (Vivek, 2022)

Sovellusrajapinnoilla (API) on keskeinen merkitys sovelluskehityksessä, ja niin myös SPA-sovelluskehityksessä. Sovellusrajapinnat mahdollistavat erilaisten ohjelmistojen kommunikoinnin, datan välityksen ja yhteistyön ilman, että ne paljastavat omaa sisäistä arkkitehtuuriaan. Tämä on tärkeää, sillä se edistää monimutkaisten ohjelmistojen integraatiota kokonaisuudeksi. Sen lisäksi sovellusrajapinnat edistävät modulaarista skaalautuvuutta ja luovat monipuolisia toiminnallisuuksia hajautetuissa järjestelmissä. Sovellusrajapinnat on suunniteltava huolellisesti, sillä ne vaikuttavat käyttäjäkokemukseen, kehityksen vaivattomuuteen ja palveluiden ketteryyteen (Rion & Any, 2025). Kokonaisuudessaan SPA-sovelluksissa on asiakaspuoli, joka sisältää sovelluksen näkymäkerroksen, ja palvelinpuoli, joka käsittelee asiakaspuolelta saapuvia pyyntöjä ja hallitsee dataa. Asiakaspuoli ja palvelinpuoli kommunikoivat sovellusrajapinnan kautta (Kornienko et al., 2021).

SPA-sovelluksilla on omat etunsa ja haasteensa. Vivekin (2022) mukaan SPA-arkkitehtuurin etuina on sujuva interaktiivisuus, vähäinen palvelinkuormitus sekä mahdollisuus offline-käyttöön Progressive Web App (PWA) -integraation avulla. Samassa tutkimuksessa Vivek tuo esiin myös SPA-sovellusten heikkouksia,

joita on hidas ensimmäinen lataus, hakukoneoptimoinnin haasteet ilman lisäkonfigurointia ja lisääntynyt muistinkäyttö asiakaspuolella. Myös Sangarsu (2019, s. 286–287) tarkastelee SPA-sovellusten haasteita ja tukee monilta osin Vivekin havaintoja. Sangarsu nostaa esiin hakukoneoptimoinnin haasteet ja ensimmäisen latauksen hitauden. Vanhemmat selaimet eivät välttämättä tue kaikkia moderneja JavaScript-ominaisuuksia tai tyylejä, jolloin selainten yhteensopivuus voi muodostua ongelmaksi. Tästä syystä Sangarsu suosittelee fallback-tekniikoiden eli vararatkaisujen hyödyntämistä. Lisäksi Sangarsu käsittelee SPA-sovelluksiin liittyviä tietoturvahaasteita, erityisesti Cross-Site Request Forgery (CSRF) -hyökkäyksiä, ja korostaa kehittäjien vastuuta kirjasto- ja kehysversioiden säännöllisessä päivittämisessä sekä tietoturvapäivitysten hyödyntämisessä.

Nykyisin yhä useammat verkkosovellukset toteutetaan SPA-arkkitehtuurin mukaisesti (Prathap & Saravanan, 2019, s. 141). SPA-sovelluksia voidaan käyttää erittäin monipuolisesti. Ideaaleja käyttökohteita SPA-sovelluksille ovat sosiaalisen median sovellukset, interaktiiviset hallintapaneelit ja verkkotyökalut (Vivek, 2022: 275).

2.1.2 Yleiskatsaus SPA-sovelluksen kehyksiin

SPA-sovellusten kehitys perustuu yleisesti sovelluskehysten käyttöön. Kehityksen tueksi on syntynyt useita kehyksiä, jotka pyrkivät hallinnoimaan sovellusten dynaamista arkkitehtuuria (Hutagikar & Hedge, 2020). SPA-sovellukset ovat saavuttaneet laajaa suosiota, ja suosioon on vaikuttanut kehysten jatkuva sopeutuminen sovelluskehityksen vaateisiin. Kehysten tarkoituksena on tarjota monipuolisia työkaluja sovellusten kehittämiseen. Suosituimpia kehyksiä ovat Angular, React ja Vue (Emmanni, 2023).

Kehysten valintaan vaikuttavat muun muassa projektin tarpeet, suorituskyky ja kehittäjien kokemus kehyksistä. Oikean kehyksen valitseminen on tärkeää, koska sillä on myönteinen vaikutus kehityksen tehokkuuteen, suorituskykyyn ja käyttäjäkokemukseen. Suosituimmat kehykset, Angular, React ja Vue eroavat

toisistaan arkkitehtuurillisesti, ja niillä on omat vahvuutensa ja heikkoutensa. Angular on kokonaisvaltainen kehys, joka soveltuu erityisesti suurten sovellusten kehitykseen sen tyyppisuojan ja työkalujen ansiosta. React puolestaan mahdollistaa yksilöllisten käyttöliittymien luomisen nopeasti laajan ekosysteeminsä ja joustavuuden ansiosta. Vue:ta pidetään helpoimpana oppia, ja se soveltuu erityisesti pienille projekteille. (Emmanni, 2023)

2.2 React

2.2.1 Historia ja kehitys

Alun perin React kehitettiin Facebookin sisäiseksi työkaluksi. Reactin alkuperäisenä tavoitteena oli tarjota työkalu, joka helpottaa monimutkaisten käyttöliittymien rakentamista sovelluksiin, joissa tila muuttuu dynaamisesti ajan myötä. (Gackenheimer, 2015:1)

Reactin kehittäjänä pidetään Jordan Walkea, Facebookin ohjelmistoinsinööriä, joka loi ensimmäisen prototyypin nimeltä FaxJS. Prototyyppi sai vaikutteita XHP:stä — PHP- ja komponenttipohjaisesta HTML-laajennuksesta. Työkalua hyödynnettiin aluksi Facebookin sisäisissä järjestelmissä, ja sitä käytettiin muun muassa Facebookin uutisvirrassa vuonna 2011 sekä Instagramissa vuonna 2012. React julkaistiin avoimena lähdekoodina JSConf US -konferenssissa vuonna 2013, jolloin myös ulkopuoliset kehittäjät pääsivät hyödyntämään sitä. (Komperla et al., 2022:1710)

Ensimmäisessä julkaisussa vuonna 2013 React esitteli JSX-syntaksin, jossa JavaScriptiä voidaan yhdistää HTML:ään sekä komponenttipohjaisen arkkitehtuurin. Myöhemmissä versioissa Reactiin sisäistä arkkitehtuuria on päivitetty merkittävästi, kuten Virtual DOM-ratkaisun lisääminen vuonna 2014 ja Fiber-arkkitehtuuriin siirtyminen versiossa 16. Yksi merkittävimmistä kehitysaskeleista tapahtui vuonna 2019, kun Reactiin lisättiin hookit, jotka muuttivat ratkaisevasti tilan ja elinkaaren hallinnan toteutustapoja. (GeeksforGeeks, 2025)

Reactin kehittäjät tutkivat jatkokehitysmahdollisuuksia aktiivisesti. Tuotantokäyttöön valittavat ratkaisut valitaan huolellisesti, ja niiden toimivuudesta tulee olla käytännön näyttöä ennen kuin ne sulautetaan osaksi Reactia. React on kehittynyt versioon 19. (Meta Platforms, n.d.)

2.2.2 Keskeiset käsitteet

Tässä kappaleessa keskitytään Reactin keskeisimpiin ominaisuuksiin, joista yksi merkittävimmistä on JSX. JSX (JavaScript XML) on merkintäkieli, jota käytetään yleisesti React-sovelluskehityksen yhteydessä, vaikka sen käyttö ei ole pakollista. JSX toimii JavaScriptin laajennoksena ja mahdollistaa käyttöliittymäkomponenttien kuvaamisen HTML:n kaltaisella syntaksilla. Tämä mahdollistaa elementtien luomisen DOM-rakenteeseen (Document Object Model). JSX helpottaa kehitystyötä helppokäyttöisyydellään. Selain ei ymmärrä JSX:ää sellaisenaan, vaan se muutetaan selaimen ymmärtämään muotoon Babelin avulla. (Narayn, 2022)

Komponentit ovat React-sovellusten rakenteellinen pohja, sillä koko käyttöliittymä rakennetaan erillisistä komponenteista. Komponentit tukevat modulaarisuutta ja uudelleenkäytettävyyttä, jotka ovat keskeisiä periaatteita React-kehyksessä. Komponentit sisältävät sekä näkymän että logiikan yhdessä kapseloidussa kokonaisuudessa. Komponentit voidaan rakentaa niin, että ne vastaavat omista tehtävistään. (Narayn, 2022)

React-komponentit ovat itsenäisiä ja uudelleenkäytettäviä yksiköitä, mikä vähentää toiston tarvetta ja parantaa koodin ylläpidettävyyttä. Niiden itsenäisyyden ansiosta sovellusta on myös helpompi laajentaa tarvittaessa. Komponentit voidaan toteuttaa joko funktiopohjaisesti tai ES6-luokkapohjaisesti, joista funktiokomponentit ovat nykyisin suositeltu ja modernimpi lähestymistapa. Funktionaaliset komponentit ovat luonteeltaan puhtaita funktioita, eli ne eivät aiheuta sivuvaikutuksia renderöinnin aikana. Komponenttien välinen yksisuuntainen tiedonsiirto luodaan propsien avulla. (Meta Platforms, n.d.)

Tila ja sen hallinta ovat tärkeitä SPA-sovellusten luomisessa. Tila kuvaa sovelluksen senhetkistä tilaa, joka muuttuu esimerkiksi käyttäjän toiminnan seurauksena. Komponentin tila on paikallinen ja yksityinen. Käytännössä tämä tarkoittaa sitä, että kahden saman komponentin tilat ovat erillisiä, ja yläkomponentit eivät ole tietoisia alakomponenttiensa tilasta. (Meta Platforms, n.d)

Reactin renderöintiprosessi koostuu ensimmäisestä renderöinnistä ja uudelleenrenderöinnistä. Ensimmäinen renderöinti tapahtuu createRoot-funktion avulla. Funktio auttaa renderöimään sovelluksen valittuun DOM-solmuun. Uudelleenrenderöinti käynnistyy, kun React havaitsee tilan arvojen muuttuneen edellisen ja nykyisen renderöinnin välillä. Vasta sen jälkeen React vaihtaa commit-vaiheeseen, jossa se tekee mahdollisimman vähäiset operaatiot, jotta DOM vastaa viimeisintä renderöinnin tulosta. DOM-päivityksen jälkeen selain piirtää päivitetyn käyttöliittymän ruudulle. (Meta Platforms, n.d)

React-sovelluksen tilanhallinnan voi toteuttaa usealla eri tavalla. Yksi tapa on käyttää useState-hookia, joka palauttaa tilan nykyisen arvon ja setter-funktion, jolla tilaa voidaan muuttaa (Meta Platforms, n.d.). Tilanhallinnassa voidaan käyttää myös keskitettyjä tilanhallintaratkaisuja, kuten Redux, MobX, Recoil tai Reactin oma Context API. Nämä ratkaisut sopivat suurille sovelluksille, joissa on monimutkainen tila. Toteutustavoilla on omat vahvuutensa ja heikkoutensa (Narender, 2025).

React Hookit tulivat uutena ominaisuutena Reactiin versiossa 16.8. Reactin sisäänrakennetut hookit tarjoavat monipuolisen työkalupakin kehittäjälle. Niiden avulla voidaan hallita tilaa (useState, useReducer), sivuvaikutuksia (useEffect), kontekstia (useContext) sekä muita sovelluskehityksessä hyödyllisiä toimintoja. Hookeihin liittyy myös tiettyjä käyttöperiaatteita: niitä voidaan kutsua vain komponentin ylimmällä tasolla ja ainoastaan Reactin funktiokomponenteista tai muista hookeista. Kehittäjät voivat myös luoda omia hookeja yhdistelemällä sisäänrakennettuja hookeja ja muuta logiikkaa. (Meta Platforms, n.d.)

Reactin ytimessä on Virtual DOM, joka toimii virtuaalisena esityksenä todellisesta selaimen DOM-rakenteesta. Virtual DOMin avulla kehittäjät voivat keskittyä Reactille tyypilliseen kuvailevaan tapaan sovellusta rakennettaessa. Virtual DOM hyödyntää Fiber-arkkitehtuuria, joka määrittää, miten ja missä järjestyksessä DOM-puun muutokset suoritetaan. Kun sovelluksen tila tai ominaisuudet muuttuvat, React luo uuden virtuaalisen DOM-puun ja vertaa sitä edelliseen versioon. Tätä vertailuprosessia kutsutaan diffing-menetelmäksi, ja sen avulla React tunnistaa muuttuneet elementit ja päivittää ne tehokkaasti todelliseen DOM-rakenteeseen (Meta Platforms, n.d.)

2.2.3 Kirjastoympäristö

React ei ole kehys, vaan JavaScript-kirjasto. Vertailun sujuvoittamiseksi sitä kuitenkin viitataan kehyksenä tässä työssä. Reactin sisäänrakennetut ominaisuudet keskittyvät käyttöliittymän rakentamiseen. Ulkoisten kirjastojen avulla pyritään helpottamaan kehitystyötä ja laajentamaan käyttöliittymän toiminnallisuutta. Ulkoisten kirjastoiden hallitseminen on tärkeää kehittäjälle, joka työskentelee Reactin parissa (Elrom, 2021). React on kirjasto, joka ei sisällä kaikkia keskeisiä ominaisuuksia, kuten sisäänrakennettua reititystä, edistynyttä tilanhallintaa tai API-hallintaa (Kushawa, Kumar & Dhankar, 2023).

Ulkoisten kirjastojen valikoima on laaja, ja samaan ongelmaan on kehitetty useita ratkaisuja. Kirjastot tarjoavat ratkaisuja esimerkiksi tyyppitarkastamiseen, CSS-esiprosessointiin, tilanhallintaan, CSS-kehyksiin, reititykseen, testaukseen ja koodin laadun tarkistamiseen. (Elrom, 2021)

2.2.4 Edut ja haasteet

Reactilla on useita vahvuuksia. Komponenttipohjaisen arkkitehtuurin ja syntaksin ansiosta React on suhteellisen helppo oppia. Kehitystyö on nopeaa, koska komponentit ovat uudelleenkäytettäviä ja Virtual DOM-ratkaisu päivittää tehokkaasti käyttöliittymää. Reactin avulla voidaan luoda visuaalisesti näyttäviä ja in-

teraktiivisia käyttöliittymiä. Monet suuret yhtiöt, kuten Netflix, hyödyntävät Reactia, mikä vahvistaa sen mainetta kyvykkäänä sovelluskehityksen työkaluna. Reactin ympärille on muodostunut laaja yhteisö, joka tukee kehittäjiä dokumentaation, tutoriaalien ja avoimen lähdekoodin projektien kautta. (Kushawa, Kumar & Dhankar, 2023)

Reactin heikkouksiin kuuluu tarve ulkoisille kirjastoille, sillä monimutkaisempi sovelluslogiikka on vaikeampaa toteuttaa Reactin sisäänrakennetuilla ratkaisuilla. Reactin edistyneemmät konseptit saattavat olla haastavia oppia. (Kushawa, Kumar & Dhankar, 2023)

React suoriutuu hyvin suorituskyky- ja skaalautuvuustesteissä verrattuna muihin SPA-sovellusten kehyksiin. React saavuttaa Angulariin verrattuna paremmat tulokset latausajassa ja renderöintinopeudessa. Lisäksi sillä on alhaisempi korkein huppumuistinkäyttö, ja vähemmän muistivuotoja ja pienempi suorittimen käyttö. Vue puolestaan on näistä kolmesta kehyksestä tehokkain, kun verrataan kaikkia edellä mainittuja kategorioita. (Piastou, 2023)

2.3 Angular

2.3.1 Historia ja kehitys

Angularin historia alkoi vuonna 2009, kun kehittäjät työskentelivät Google Feedback-projektin yhteydessä. Projektissa ilmeni haasteita etenkin kehitysnopeudessa ja testattavuudessa. Yksi kehittäjistä, Misko Hevery, esitti ratkaisuksi oman projektinsa, jonka avulla kehitystyötä voitaisiin nopeuttaa huomattavasti. Heveryn demonstroitua projektinsa toimivuuden muut kehittäjät huomasivat työkalun potentiaalin, mikä johti AngularJS:n kehityksen alkuun. Angular on Googlen ylläpitämä ohjelmistokehys. (Green & Shyam, 2013:vii)

Tämän seurauksena AngularJS julkaistiin vuonna 2010. Sen MVC-pohjainen arkkitehtuuri oli yksi varhaisimmista ratkaisuista, joka tarjosi laajan valikoiman

työkaluja SPA-sovelluksien kehittämiseen. Kehys sisälsi sisäänrakennettuja ratkaisuita muun muassa reititykseen, lomakkeiden validiointiin ja HTTP-pyyntöihin. Sen keskeisimpiä innovaatioita oli kaksisuuntainen tiedonsidonta, joka mahdollisti automaattisen datan synkronoinnin mallin ja näkymän välillä ilman suoraa DOM manipulointia. (Vishnuvardhan, 2015:45)

Siirtyminen Angular 2-versioon toi mukanaan merkittäviä muutoksia kehyksen arkkitehtuuriin. Angular siirtyi käyttämään pääasiallisena kehyskielenä tyyppiturvallista TypeScriptiä. Angular siirtyi myös Reactin tapaan komponenttipohjaiseen rakenteeseen. Uusi versio tarjosi AngularJS:ään verrattuna paremman suorituskyvyn ja skaalautuvuuden. (TestKarts, n.a)

Google ja Angular pyrkivät mukautumaan web-kehityksen jatkuvasti kasvaviin vaatimuksiin säilyttäen samalla positiivisen kehittäjäkokemuksen. Kehitys Angularin nykyiseen versioon on tuonut mukanaan muutoksia, jotka pyrkivät pitämään sitä vahvana vaihtoehtona kehittäjille, jotka haluavat kehittää sovelluksia kehyksen avulla, joka tarjoaa kaikki työvälineet modernien sovelluksien kehitykselle. (TestKarts, n.a)

2.3.2 Keskeiset käsitteet

Tässä kappaleessa tarkastellaan Angularin keskeisiä käsitteitä, kuten TypeScriptiä, moduuleja, palveluita ja riippuvuuden injektiota.

Angularin suositeltu kehyskieli on TypeScript. TypeScript on JavaScriptin superset, eli se laajentaa JavaScriptin ominaisuuksia. Se on staattisesti tyypitetty kieli, joka käännetään lopulta JavaScriptiksi. TypeScript edistää kehittäjien tehokkuutta, sovelluksen ylläpidettävyyttä, ja virheiden ehkäisyä. Sen edut korostuvat etenkin isoissa sovelluksissa. (George, 2020)

Angularissa sovellus voidaan jakaa moduuleihin, mikä on arkkitehtuurillinen valinta. Sovellus voi koostua useista moduuleista, jotka muodostavat oman yksikkönsä kääntämisvaiheessa. Moduulit määrittävät kuinka komponentit tulisi

kääntää. Moduuleja käytettäessä komponentit eivät ole itsenäisiä, vaan ne kuuluvat tiettyyn moduuliin. Moduulit mahdollistavat laiskan latauksen (Savkin & Cross, 2017). Moduuli määrittää, mitkä komponentit, direktiivit ja piiput se omistaa, mitä ulkoisia osia se tuo käyttöönsä, ja mitkä sen osat ovat muiden komponenttien ja moduuleiden käytettävissä. Se määrittää myös mitkä riippuvuuksien tarjoajat ovat komponenttien käytettävissä, ja mitkä moduulin osat ovat välittömästi ladattavana ensimmäisenä. Moduulien sijasta Angular suosittelee itsenäisten komponenttien käyttöä uusissa sovelluksissa (Angular Documentation, n.a).

Angular-komponentit ovat sovelluksen keskeisiä rakennuspalikoita ja Angular määrittelee miten ne toteutetaan. Komponentit muodostuvat sisäisistä konfiguraatioista, kuten komponentin selektorista, HTML- ja CSS-pohjista, sekä mahdollisista riippuvuuksien tarjoajista. Komponentti tarvitsee oman luokkansa, jossa määritetään sen logiikka (Angular Documentation, n.a).

Angular käyttää riippuvuuksien injektioita ominaisuuksien jakamiseen sovelluksen eri osien välillä. Riippuvuus voi olla esimerkiksi objekti tai palvelu, jota komponentti ei luo itse. Riippuvuuksien injektio toimii siten, että riippuvuus joko tarjotaan tai injektoidaan komponentille. Riippuvuuden injektiolla on useita etuja, ja se pyrkii parantamaan koodin ylläpidettävyyttä, skaalautuvuutta ja testattavuutta. (Angular Documentation, n.a)

2.3.3 Ekosysteemi

Angular tarjoaa laajasti ominaisuuksia ja sillä on laaja ekosysteemi. Ekosysteemi tarjoaa monipuolisesti työkaluja sovelluskehitykseen, kuten Angular CLI, Angular Material ja testaukseen tarkoitettuja työkaluja. Ominaisuuksiensa ja ekosysteemin ansiosta Angularilla on mahdollista rakentaa suuria ja monimutkaisia sovelluksia. (Emmanni, 2024)

Angular CLI on komentorivityökalu, jonka avulla Angular sovelluskehityksen vaiheita voidaan automatisoida. Angular CLI tarjoaa laajasti komentoja sovelluksen

elinkaaren ajalle, kuten esimerkiksi sovelluksen ja ominaisuuksien luomiseen, testaukseen ja sovelluksen rakentamiseen (van de Moere, 2018). Angular CLI on ollut jatkuvan kehityksen alla, ja sen tarkoituksena on helpottaa kehitysprosessia ja lisätä kehittäjien tuottavuutta (Kodali, 2024).

2.3.4 Edut ja haasteet

Kuten muillakin kehyksillä, Angularilla on omat etunsa ja haasteensa. Yksi sen keskeisistä vahvuuksista on mahdollisuus rakentaa suuria ja monimutkaisia sovelluksia. Angularin sisäänrakennetut ominaisuudet mahdollistavat edellä mainitun vahvuuden. Kaksisuuntainen tiedonsidonta on todettu johtavan hyvään käyttäjäkokemukseen. Riippuvuuden injektio puolestaan auttaa rakentamaan modulaarisen rakenteen sovellukselle. Angular CLI vaikuttaa positiivisesti tuottavuuteen, koska se automatisoi tehtäviä. (Waite, 2023)

Angularin haasteina pidetään sen jyrkkää oppimiskäyrää, etenkin aloitteleville kehittäjille. Sillä on myös suuri pakettikoko, joka vaikuttaa suorituskykyyn hidastavasti. Angular päivittää kehystä jatkuvasti, mikä voi lisätä ylläpitotyötä. (Waite, 2023)

Suorituskyvyssä, kehittäjien tuottavuudessa ja ekosysteemin tuessa Angular jää hieman Reactille ja Vuelle vertailussa. Se sopii etenkin isojen sovelluksien kehittämiseen, jossa on dynaamisia näkymiä ja ominaisuuksiltaan laaja käyttöliittymä. (Emmanni, 2023)

2.4 Oppimisen siirtovaikutus

2.4.1 Oppimisen siirtovaikutus ilmiönä

Oppimisen siirtovaikutus tapahtuu, kun aiemmin opittua käytetään ja sovelletaan uuden asian oppimisessa. Uusi asia voi olla joko samankaltainen tai merkittävästi erilainen. Tämä ilmiö on hyvin tärkeä, sillä se luo perustan oppimiselle, ajattelulle ja kognitiolle. Etenkin koulutuksessa siirtovaikutusta pidetään tärkeänä, vaikka ilmiön toteutuminen ei aina onnistu käytännössä. (Haskell, 2011)

Vaikka ilmiötä on tutkittu laajasti, niin yhtä yleisesti hyväksyttävää teoriaa oppimisen siirtovaikutuksesta ei ole muodostunut (Haskell, 2011). Tämän takia tämän opinnäytetyön teoria pohjautuu Perkinsin ja Salomonin teoriaan oppimisen siirtovaikutuksesta. Teoria on valittu, koska se antaa selkeät raamit siirtovaikutuksen tutkimiselle.

Perkins ja Salomon määrittelevät oppimisen siirtovaikutusta kontekstiin ja materiaaliin riippuvaiseksi ilmiöksi. Heidän mukaansa oppiminen voidaan osoittaa, kun opittua voidaan demonstroida myöhemmin. Yksiselitteistä viivaa ei voida vetää sen suhteen, tapahtuiko oppiminen siirtovaikutuksen vai normaalin oppimisen pohjalta. Työssään he jakavat oppimisen siirtovaikutukset positiiviseen ja negatiiviseen siirtovaikutukseen, sekä lähi- että etäsiirtovaikutukseen. Näitä tarkastellaan lähemmin tulevissa kappaleissa. (Perkins & Salomon, 1999)

Perkins ja Salomonin (1999) mukaan positiivinen oppimisen siirtovaikutus tapahtuu erilaisten olosuhteiden vallitessa. Oppimisen siirtovaikutus on todennäköisempää aiheesta, joka osataan perinpohjaisesti ja monimuotoisesti. Näitä joustavia ja osin automatisoituja taitoja on helpompi tuoda esiin uusissa tilanteissa, jotka vaativat erilaisia taitoja. Toinen siirtovaikutusta tukeva kyky on tietoinen abstrahointi. Myös aktiivinen metakognitiivinen itsehavainnointi on tärkeää, sillä kyky havainnoida omia ajatusprosesseja edistää positiivista siirtovaikutusta. Oppimisen siirtovaikutuksen kannalta on myös tärkeää pystyä käyttämään metaforia ja analogioita. Perkins ja Salomon mainitsevat myös tietoisuustaidot, jotka ovat tehokkaampia siirtovaikutuksen kannalta verrattuna passiiviseen oppimiseen.

Työssään Perkins ja Salomon (1999) esittävät kolme mekanismia oppimisen siirtovaikutukselle. Ensimmäinen mekanismi on siirtovaikutus havaittujen toimintamahdollisuuksien kautta. Jos oppimistilanteet ovat toimimismahdollisuuksiltaan samankaltaisia, oppija voi käyttää adaptoitua toimintamallia uuden oppimiseen. Toinen mekanismi on niin sanotut korkean ja matalan tien siirtovaikutukset. Matalan tien siirtovaikutus tapahtuu kun samankaltaisessa kontekstissa

tapahtuu osin automaattisia toimintoja, jotka helpottavat uuden oppimista. Korkean tien siirtovaikutus edellyttää tietoisuutta, yhteyksien löytämistä ja abstrahointia.

2.4.2 Positiivinen ja negatiivinen siirtovaikutus

Positiivinen siirtovaikutus tapahtuu, kun yhdessä kontekstissa opittu tieto tai taito lisää todennäköisyyttä onnistua positiivisesti toisen kyvyn oppimisessa. Tästä esimerkkinä on esimerkiksi uuden kielen omaksuminen, jos se on rakenteeltaan samankaltainen kuin oppijan äidinkieli. (Perkins & Salomon, 1999)

Negatiivinen siirtovaikutus puolestaan voi vaikeuttaa uuden asian omaksumista. Esimerkiksi vieraan kielen oppiminen voi olla haastavampaa, jos tukeutuu oman kielensä sanajärjestykseen tai muihin ominaisuuksiin, jotka poikkeavat opittavassa kielessä. Negatiivista siirtovaikutusta esiintyy usein oppimisen alkuvaiheessa, ja kokemuksen kautta oppijat pystyvät korjaamaan virheellisiä siirtoja. (Perkins & Salomon, 1999)

2.4.3 Lähi- ja etäsiirtovaikutus

Lähisiirtovaikutus viittaa siirtovaikutukseen, joka tapahtuu toisilleen samankaltaisissa olosuhteissa. Oppija voi esimerkiksi soveltaa osaamistaan koekysymyksiin, jotka muistuttavat tehtäviä joita hän on tehnyt kotiläksynä. Etäsiirtovaikutuksissa olosuhteet voivat olla merkittävästi poikkeavia, mutta siirtovaikutus tapahtuu silti. Esimerkiksi shakin pelaaja voi käyttää pelistä oppimiaan strategioitaan ominaisuuksiltaan erilaisiin tilanteisiin, kuten politiikkaan. Lähisiirtovaikutus on todennäköisempää verrattuna etäsiirtovaikutukseen (Perkins & Salomon, 1999)

Perkins ja Salomon mainitsevat, että nämä termit ovat suhteellisia, ja niitä ei voi määrittää täysin tarkasti. Ne ovat kuitenkin isommassa mittakaavassa hyödyllisiä oppimisen siirtovaikutuksen tutkimisessa. (Perkins & Salomon, 1999)

2.4.4 Oppimisen siirtovaikutus ohjelmoinnissa

Ohjelmoinnin siirtovaikutusta käsitteleviä tutkimuksia on runsaasti, ja monet niistä tarkastelevat ohjelmoinnin oppimisen vaikutuksia kognitiivisiin taitoihin, kuten tietokoneajatteluun, ongelmanratkaisukykyyn ja luovuuteen. Siirtovaikutuksen on arvioitu tapahtuvan, koska ohjelmoinnilla ja näillä kognitiivisilla taidoilla on konseptuaalisia yhteneväisyyksiä. Tämä aihe on saanut kritiikkiä, koska useat väitteet perustuu rajalliseen näyttöön kokeellisista tutkimuksista. Empiiriset tutkimukset ovat myös vanhahkoja, ja perustuvat 1980- ja 1990-luvuille. Joillakin osa-alueilla, kuten ohjelmoinnin vaikutus ongelmanratkaisukykyyn, näyttö on osin ristiriitaista. Aiheeseen peräänkuulutetaan enemmän ajankohtaista empiiristä näyttöä, sillä myös teknologia on uusiutunut ja mennyt eteenpäin. (Scherer, 2016)

Eräs meta-analyysi tarkastelee tietokoneajattelun siirtovaikutusta muille osa-alueille, erityisesti STEM-aloille. Tämä alue sisältää luonnontieteet, teknologian, tekniikan ja matematiikan. Vaikka tietokoneajattelu ja ohjelmointi eivät ole identtisiä käsitteitä, ne kuitenkin sisältävät paljon samoja ominaisuuksia, kuten abstrahointia, ongelmanratkaisua ja algoritmista ajattelua. Meta-analyysin mukaan tietokoneajattelulla on siirtovaikutuksia niin kognitiivisiin kuin ei-kognitiivisiin ominaisuuksiin. Tutkimus raportoi että lähisiirtovaikutus on todennäköisempää kuin etäsiirtovaikutus. Lisäksi ei-kognitiiviset taidot todettiin siirtyvän useammin kuin kognitiiviset. Voimakkaimpia siirtovaikutuksia havaittiin matematiikkaan, fysiikkaan, tuotteliaisuuteen sekä tietokoneajattelun käsitteelliseen ymmärrykseen liittyvissä taidoissa. (Li & Oon, 2024)

Ohjelmoinnin oppimisen siirtovaikutusta on tutkittu myös siirryttäessä lohkopohjaisista ohjelmistoympäristöistä tekstipohjaisiin ohjelmistoympäristöihin. Näiden ympäristöjen välillä on huomattu olevan oppimisen siirtovaikutukseen liittyviä haasteita, ja oletettu siirtovaikutus ei ole aina automaattista. Lohkopohjaisten ympäristöjen haasteena on, että oppilaat omaksuvat ohjelmoinnin alkeita, joilla mahdollistetaan pääosin yksinkertaisten ohjelmistojen toteuttaminen. Tutkimuksessa on kuitenkin havaittu, että lohko- ja tekstiympäristön piirteitä yhdistävä

hybridimallit tukevat siirtymää lohkopohjaista mallia paremmin. Hybridipohjaista mallia käyttäneet oppilaat osoittivat parempaa kykyä oppia, muokata koodia, vähäisempää syntaksivirheiden määrää, parempaa ohjelmistokäskyjen muistamista ja kokonaisuudessaan helpompaa siirtymistä tekstipohjaisiin ympäristöihin. (Alrubaye, Ludi & Mkaouer, 2019)

3 Tutkimusmenetelmä

3.1 Tutkimusasetelma

Opinnäytetyön tarkoituksena on havainnollistaa React- ja Angular-kehysten välisiä eroja ja oppimisen siirtovaikutusta, joita ilmenee projektin aikana. Projekti on valittu tarkoituksenmukaisesti siten, että se mahdollistaa useiden teknologisten osa-alueiden vertailun sekä monipuolisen oppimisprosessin. Tutkimusasetelman lähtökohtana on, että tekijällä on aiempaa kokemusta React-kehyksestä, kun taas Angular on uusi kehys.

Tutkimus toteutetaan projektipohjaisena tapaustutkimuksena. Projektissa toteutetaan sama sovellus kahden eri kehyksen avulla, joka muodostaa tutkimuksen raamit. Vertailuasema on kontrolloitu niin, että sovellukset käyttävät samaa palvelinpuolen sovellusta, jolloin toteutuksen erot johtuvat pelkästään asiakaspuolen teknologisista ratkaisuista. Tutkimus perustuu yhden kehittäjän kokemukseen. Tutkimuksen tavoitteena on dokumentoida siirtymä kehyksestä toiseen monipuolisesti ja läpinäkyvästi.

3.2 Aineisto

Aineisto perustuu kahteen itse rakennettuun sovellukseen ja toteutuksien vertailuun. Suorituskyky ei ollut yksi päätöksentekoon vaikuttava tekijä, sillä työ painottui lopulta enemmän oppimisprosessiin. Konkreettisesti aineisto syntyi esimerkiksi tilanhallintaan, reititykseen ja arkkitehtuuriin vaikuttavista päätöksistä, joita pyrittiin tekemään mahdollisimman johdonmukaisesti.

Aineiston rajaus tarkentui kehitysprosessin edetessä. Sovelluksen toiminnalliset ja ei-toiminnalliset vaatimukset, joita käsitellään seuraavassa luvussa, olivat merkittäviä rajoittavia tekijöitä. Tarkastelun pääpainopiste kohdistuu arkkitehtuurillisesti merkittävimpiin ratkaisuihin, kuten tilanhallintaan, reititykseen ja komponenttirakenteeseen.

Aineisto kerättiin sovellusten ollessa valmiita. Oppimisen siirtovaikutusta analysoitiin retrospektiivisesti. Aineiston keräämiseen vaikutti se, että tutkimuskysymys tarkentui vasta sovelluskehityksen loppuvaiheilla.

3.3 Analyysitapa

Analyysi perustuu siihen, että kummallakin kehyksellä tehtyjä ratkaisuita verrataan rinnakkain. Erojen, samankaltaisuuksien ja retrospektiivisen analyysin avulla pyritään tunnistamaan positiivinen ja negatiivinen siirtovaikutus.

Siirtovaikutuksen analyysi perustuu myös Perkinsin ja Salomonin teoriaan oppimisen siirtovaikutuksesta. Tarkastelu painottui positiiviseen ja negatiiviseen siirtovaikutukseen. Myös lähi- ja etäsiirtovaikutusta tutkittiin niin, että teknologisesti samankaltaiset toteutukset tulkittiin tässä kontekstissa lähisiirtovaikutukseksi, ja erilaiset toteutukset etäsiirtovaikutukseksi.

Analyysi pohjautuu havaittuihin eroihin sekä omaan reflektioon, joka pyrkii tuottamaan laadullista analyysiä. Tutkimuksessa ei käytetty määrällisiä mittareita, sillä tavoitteena oli laadullinen ja teorialähtöinen tarkastelu.

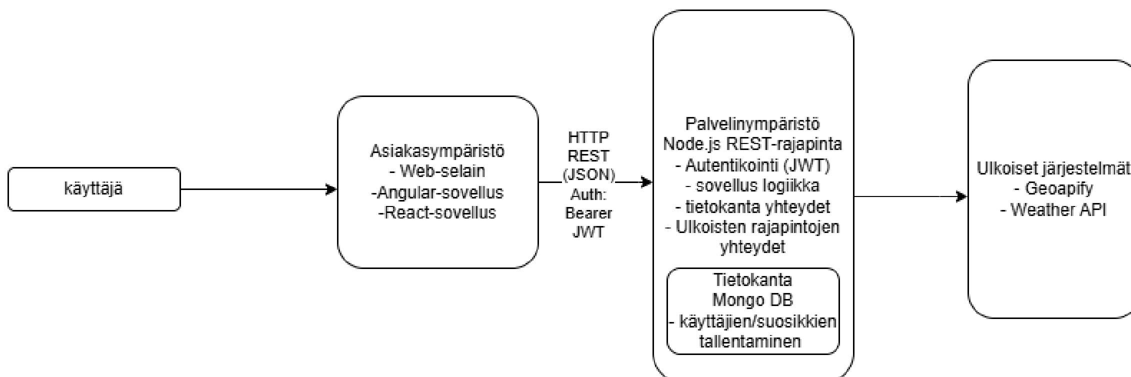
3.4 Menetelmän haasteet ja rajoitteet

Tämän työn tutkimusasetelma on projektipohjainen tapaustutkimus. Ensimmäinen haaste liittyy yhden kehittäjän asetelmaan. Tulkinnat pohjautuvat merkittävästi aiempaan React-kokemukseen, johon kuuluvat esimerkiksi mentaalimallit ja kirjastopreferenssit. Analyysin retroperspektiivisyys pohjautuu muistiin, ja se voi johtaa väärentymiin ja hankaloittaa tarkkoja havaintoja projektin aikana.

Yleistettävyys on hyvin rajattua tutkimuksen muodon johdosta. Se ei myöskään ota kantaa suorituskykyyn ja optimointiin.

4 Sovelluksen kehittäminen

4.1 Sovelluksen toiminnalliset periaatteet



Kuva 1. Projektin kokonaisarkkitehtuuri

Opinnäytetyöhön valittiin projektiksi sääsovellus. Tarkoituksena oli valita mahdollisimman yksinkertainen projekti, joka kuitenkin pystyy tuomaan esiin Reactin ja Angularin eroja ja demonstroimaan niiden perustoimintoja. Projektin kokoon ja laatuun vaikutti myös opinnäytetyön asetelma, eli miten React-perusteet hallitseva kehittäjä oppii Angularin. Työ rajattiin käsittelemään kehysten perustoimintoja, ja tuotantotason ominaisuudet, kuten istunnonhallinta, laaja optimointi ja monimutkainen arkkitehtuuri, jätettiin tarkastelun ulkopuolelle. Tähän rajaukseen vaikutti se, että työn keskiössä on oppimisprosessi, eikä tuotantotason sovelluksen kehittäminen.

Sääsovellus valittiin perustellusti. Se vaatii huolellista tilanhallinnan suunnittelua, koska sovellus on riippuvainen useista tilamuuttujista, mutta samalla se ei muutu vaikeaksi hallita ja hahmottaa. Tavoitteena oli arvioida juuri sopivin tilanhallintaratkaisu molemmille sovelluksille, sillä molemmat kehykset tarjoavat useampia vaihtoehtoja tilanhallintaan. Reititys sisällytettiin projektiin lisäämällä useampi sivunäkymä. Näin ollen sovellus on lähempänä tuotantotason ratkaisua useampine sivunäkymineen. Suosikkilista valittiin sovellukseen, koska se luo tarpeen monissa sovelluksissa löytyvistä CRUD-toiminnoista (luonti, haku, päivitys, poisto), jotka vaativat palvelin- ja asiakaspuolen integroitua tilanhallintaa.

Työhön kehitettiin oma palvelinpuolen sovellus, jotta sovelluksen toiminta olisi lähempänä realistista ratkaisua. Palvelinpuoli on Express-pohjainen Node.js-sovellus. Palvelinpuolen tärkein toiminnallisuus perustui käyttäjätoimintoihin ja kommunikointiin kolmansien osapuolten rajapintojen kanssa. Käyttäjien autentikointiin käytettiin token-pohjaista autentikointia JSON Web Tokenin avulla. Tietokantana käytettiin MongoDB:tä, jolloin kokonaisuuksista muodostuu MERN- ja MEAN-teknologiapinot. Tietokantaa käytettiin tallentamaan käyttäjien tiedot, kuten hashattu salasana ja käyttäjän omat paikkasuosikit.

Kuva 1 näyttää kokonaisarkkitehtuurin. On huomionarvoista, kuinka erilliset sovellukset toimivat muuten identtisissä ympäristöissä. Tämä johtaa siihen, että opinnäytetyössä havaitut erot tulevat ainoastaan asiakaspuolen sovelluksien teknologisista ratkaisuista. Palvelinpuoli vastaa sovelluslogiikasta, kuten datan hakemisesta ulkoisista rajapinnoista ja datan käsittelystä, kun taas asiakaspuolen tehtävänä on visualisoida dataa, vastata käyttäjävuorovaikutuksesta ja välittää pyynnöt palvelimelle. Näin ollen saadaan kokonaisuus, jossa vastuut on eroteltu selkeästi.

4.2 Tärkeimmät toiminnalliset vaatimukset, ei-toiminnalliset vaatimukset ja rajaukset

Toiminnalliset vaatimukset:

- käyttäjä voi rekisteröityä ja kirjautua sisään
- käyttäjä voi hakea säätietoja sijainnin perusteella
- käyttäjä voi vaihtaa sijainnin joko kartasta tai automatisoiduista ehdotuksista
- käyttäjä voi tarkastella sääennustetta
- kirjautunut käyttäjä voi tallentaa paikkoja suosikkilistalle tai poistaa paikan sieltä
- säätiedot, automatisoidut ehdotukset ja käyttäjään liittyvät toiminnot haetaan ulkoisista rajapinnoista

Ei-toiminnalliset vaatimukset

- sovellus on SPA-sovellus
- sovellus käyttää URL-pohjaista reititystä

- sovellus on kehitetty modulaarisesti
- tilanhallinta on toteutettu projektiin skaalaan sopivalla tavalla
- osa näkymistä on suojattu reittien avulla
- osa komponenteista käyttää lazy loading-tekniikkaa
- sovelluksessa on selkeät lataus- ja virhenäkymät
- sovellus ajetaan lokaalisti kehitysympäristössä, eikä tuotantoon vientiä ole toteutettu osana tätä työtä

Rajaukset

- ei tuotantotason suorituskykyoptimointia
- ei istunnonhallintaa
- ei monimutkaista käyttöoikeusmallia
- ei laajaa virheenkäsittelyä
- ei testejä

4.3 React-sovelluksen toteutus

4.3.1 Projektin alustus

Projektin alustus alkoi kehitysympäristön valitsemisella. Kehitysympäristöksi valittiin aiemman kokemuksen perusteella Vite. Tämän jälkeen valittiin projektin rakenne. Ratkaisu pohjautui lähteisiin keskisuuren projektin kansiorakenteesta sekä omasta mieltymyksestä. Tavoitteena oli luoda modulaarinen kansiorakenne, joka on helposti navigoitavissa.

Sovelluksen suunnittelu koostui tärkeimpien komponenttien määrittämisestä ja niiden tiloista. Tästä johdettiin sovelluksen koko tila, joka jaettiin loogisesti omiin ryhmiinsä. Projektissa päädyttiin käyttämään Reactin tarjoamaa kontekstia, koska useat komponentit olivat riippuvaisia eri kontekstien arvoista. Ratkaisun arviointiin selkeyttävän komponentteja, sekä vähentävän prop drillingin tarvetta, eli ominaisuuksien siirtoa komponentista sen lapsi komponentteihin.

Kontekstin tilanhallinta valittiin useStaten ja useReducerin väliltä sen mukaan, kuinka paljon erillisiä tiloja tarvittiin, ja kuinka paljon tilaa tulisi päivittää samanaikaisesti. Ulkoisen kirjaston, kuten Reduxin, sisällyttämistä projektiin ei koettu tarpeellisena, koska projekti oli suhteellisen pieni.

Tilan perusteella määritettiin palvelut. Suunnitteluvaiheessa ulkoista kirjastoa kuten Axios ei koettu tarpeellisena, koska rajapinnoille tehtävien kutsujen logiikka oli suunniteltu hyvin yksinkertaiseksi.

Myös omien hookien tarve määritettiin, ja automatisoidut ehdotukset tunnistettiin sopivaksi kohteeksi toteuttaa omalla hookilla. Valinta perustui siihen, että vain yksi komponentti tarvitsi tämän tilan, ja että se selkeyttäisi komponentin rakennetta.

Ulkoisten kirjastojen tarve rajattiin reititykseen ja karttaan. Aiemman kokemuksen ja suosion perusteella React Router Dom valittiin reititykseen ja Leaflet-kirjasto kartan ja sen toimintojen toteuttamiseen. Molemmat kirjastot tarjoavat komponenttipohjaisen ratkaisun, joka koettiin projektiin sopivaksi.

Kaikki valinnat tehtiin projektin koon, tarkoituksen ja oman osaamisen perusteella. Suunnitteluprosessi mahdollisti selkeän rakenteen ja modulaarisuuden, mikä sujuvoitti kehittämisprosessia. Sovelluksen arkkitehtuuriin vaikutti myös pyrkimys erottaa näkymälogiikka ja sovelluslogiikka selkeästi toisistaan.

4.3.2 Arkkitehtuuri ja kansiorakenne

Reactissa kansiorakenteen määrittäminen on suurilta osin kehittäjän omalla vastuulla. Projektissa pyritään noudattamaan keskisuuren projektin kansiorakennetta, joka erittelee sovelluslogiikan helposti hahmotettavaksi kokonaisuudeksi.

Suurin osa sovelluksen logiikkaa on määritetty context-kansiossa, ja se on hajautettu käyttäjienhallinnan ja autentikoinnin, suosikkien, paikan, sään ja asetus-

ten konteksteihin. Kontekstin sisällä määritetään Provider, joka jakaa kontekstista tilan arvot ja niihin liittyvät funktiot, sekä oma hook, jonka avulla komponentit pääsevät käsiksi kontekstin arvoihin ja funktioihin ilman toistuvaa perusrakennetta.

Palvelut-kansio vastaa rajapinnoille tehtävistä kutsuista, ja myös sen sisältämät tiedostot on jaettu loogisesti toiminnan mukaan. Palveluita käytetään joko kontekstissa tai omissa hookeissa.

Hooks-kansiossa määritellään omat hookit. Syötteiden ehdotuksille luotiin oma kustomoitu hook. Kontekstin tavoin tällä varmistettiin, että sovelluslogiikka pidetään erillään komponenteista, mikä tekee niistä selkeämmin esittäviä osia sovelluksesta.

Sovelluksen näkymä on pilkottu komponentteihin. Yksittäinen komponentti sisältää mahdollisimman vähän näkymälogiikkaa. Globaalin tyylin sijaan joka komponentilla on myös oma modulaarinen tyylitiedosto. Nämä komponentit kasataan kokonaisuuksiksi Pages-kansion sivunäkymissä, jotka renderöidään reitteihin.

4.3.3 Keskeiset komponentit, palvelut ja tilanhallinta

Sovelluksen keskeiset komponentit sisältävät eniten kontekstiin liittyvää logiikkaa. Käyttäjänhallintaan liittyvät komponentit, kuten käyttäjän sisäänkirjautumiska rekisteröitymiskomponentit ovat keskeisimmässä roolissa käyttäjänhallintatoimintojen kannalta. Sovelluksen tärkein toiminta sijaitsee näkymästä, joka koostuu kartasta, säänäytöstä ja automatisoiduista paikkasuosituksista. Nämä on edelleen pilkottu pienempiin komponentteihin. Esimerkiksi säänäyttö koostuu sijaintiin, nykyhetken säähän ja sääennusteeseen liittyvistä komponenteista. Lisäksi tärkeitä komponentteja löytyy asetuksista sekä suosikkinäkymästä.

Sovelluksen tilanhallinta on toteutettu käyttäen useita eri ratkaisuita. Muutamien komponenttien tila on toteutettu niin, että tila on rajattu yksittäisen komponentin

käyttöön. Näitä ovat esimerkiksi rekisteröitymiseen ja sisäänkirjautumiseen käytätettävät komponentit, jotka tarvitsevat paikallisen tilan lomakehallintaan.

Merkittävä osa sovelluksen keskeisintä tilalogiikkaa on toteutettu kontekstien avulla, jotka on jaettu loogisiin kokonnaisuuksiin. Tila on määritetty joko useState- tai useReducer-hookin avulla sen perusteella, kuinka paljon tilan osien on päivityttävä samaan aikaan. Tila ja siihen liittyvät funktiot voidaan ottaa helposti käyttöön komponenteissa, joissa niitä tarvitaan.

Kolmas lähestymistapa on tilan määrittäminen omien hookien yhteydessä. Automatisoitujen ehdotusten tila toteutettiin omalla hookilla, koska se on riippuvainen tilanmuutoksista ja sivuvaikutuksista. Tämä hook on käyttötarkoitukseltaan rajattu, mutta käytännössä omat hookit ovat kontekstin tapaan uudelleenkäytettävissä.

Myös URL:ää hyödynnetään tilana, jossa se sisältää totuuden sen hetkisestä sijainnista. Tietyt käyttäjän toiminnot käynnistävät navigoinnin, joka muuttaa URL-parametreja. Säänäyttökomponentti seuraa näitä parametrejä ja päivittää sijainnin sivuvaikutuksena niiden muuttuessa, mikä johtaa näkymän päivittymiseen.

Kommunikointi rajapinnoille tapahtuu palveluiden kautta. Palvelut vastaavat käyttäjän autentikointiin, suosikkilistaan, sään hakemiseen ja automatisoitujen ehdotuksien hakemiseen palvelimelta. Kommunikointiin on käytetty yksinkertaista fetch-mekaniikkaa.

Reactin tarjoama joustavuus tilanhallinnassa lisää vaihtoehtoja, mutta samalla se edellyttää tietoista päätöksentekoa. Päätöksentekoon rakennettiin malli, jossa ensimmäiseksi määriteltiin onko tila tarpeellinen useammissa komponenteissa. Mikäli tila oli paikallista eikä vaatinut uudelleenkäyttöä, se eristettiin yksittäiseen komponenttiin. Tila määriteltiin kontekstissa, jos useampi komponentti oli riippuvainen tilasta. Omia hookeja hyödynnettiin tilanteissa, joissa tila oli rajattu yhden komponentin ja sen lapsikomponenttien käyttöön tai kun logiikka koostui useammasta hookeista.

4.3.4 Käytetyt työkalut ja kirjastot

React-sovelluksen reititykseen valittiin ulkoinen React Router Dom-kirjasto. Se noudattaa Reactin filosofiaa, sillä reitit määritetään komponentteina, ja navigointi toteutetaan komponenttien lisäksi kirjaston tarjoamien hookien avulla. Reitit määritetään Route-komponenteissa, jossa määritetään renderöitävä komponentti. Navigointi tapahtuu Link ja NavLink-komponenttien avulla tai ohjelmoinnillisesti useNavigate-hookilla. Kirjasto tarjoaa myös useSearchParams-hookin, jonka avulla päästään suoraviivaisesti käsiksi URL:ään tallennettuihin koordinaatteihin. Koordinaatteja hyödynnetään sovelluksen tilanhallintalogiikassa.

Sovelluksen karttatoiminnallisuus toteutettiin Leaflet-kirjastolla. Myös Leaflet tukee komponenttipohjaista lähestymistapaa ja tarjoaa valmiita komponentteja sekä hookeja kartan rakentamiseen. Kartta koostuu useista kirjaston tarjoamista komponenteista, ja sen toiminnallisuus toteutettiin useMap ja useMapEvents-hookien avulla.

React itsessään tarjoaa keskeiset työkalut sovelluslogiikan toteuttamiseen. Tilanhallintaan käytettiin useReducer ja useState-hookeja. React tarjoaa myös kontekstin luomiseen tarvitut funktiot, komponentit ja hookit. Sivuvaikutuksien hallinta toteutettiin useEffect-hookilla. Rajapintakutsut toteutettiin selaimen sisäänrakennetun fetch-rajapinnan avulla.

Kirjastot valittiin aikaisemman kokemuksen perusteella. Ideaalitilanteessa kokemusta olisi useammasta vaihtoehdosta, jolloin toteutusta olisi voitu vertailla eri kirjastojen välillä ennen lopullista valintaa.

4.3.5 Haasteet

Suurin haaste liittyi koordinaattitietojen hallintaan. Alkuperäisen suunnitelman mukaisesti sijainti eristettiin omaksi tilakseen, mutta kehityksen aikana ratkaisu osoittautui tarpeettomaksi ja sovelluksen logiikkaa monimutkaistavaksi. Sään

hakeminen muuttuvien URL-parametrien perusteella olisi yksinkertaistanut toimintalogiikkaa.

Kokemus korosti huolellisen suunnittelun merkitystä sekä valmiutta muuttaa valittua strategiaa, mikäli se ei tue sovelluksen rakenteellista selkeyttä.

4.4 Angular-sovelluksen toteutus

4.4.1 Projektin alustus

Aiemman kokemuksen rajallisuuden vuoksi suunnitteluprosessi ja projektin alustus erosivat React-sovelluksesta. Angularin dokumentaatio ohjasi ajattelutapaan, jossa projektiin alustus on vahvasti Angular CLI:n vastuulla, luoden kehitysympäristön perustan. Kansiorakenteen valinnassa hyödynnettiin ulkoisia lähteitä, jotka vastasivat keskisuuren Angular-sovelluksen rakennetta. Kansiorakenne toteutettiin feature-pohjaisena.

Keskeisten komponenttien määrittelyssä sovellettiin samaa periaatetta kuin React-sovelluksessa. Oletus Angularin rakenteellisuudesta johti siihen, että suunnitellussa painotettiin toiminnallisuuksien selkeää sijoittamista oikeisiin palveluihin, tilanhallinnanratkaisuihin ja komponentteihin.

Tarvittavat työkalut hahmotettiin projektin vaatimusten ja Angularin sisäänrakennettujen ominaisuuksien perusteella. Projektin alustus sisälsi myös perehtymisen Angularin tarjoamiin työkaluihin, kuten reitityskirjastoon, sen ominaisuuksiin ja API-rajapintaan.

Tilanhallintaan valittiin mahdollisemman yksinkertainen mutta riittävä ratkaisu, koska Angular oli kehitysympäristönä uusi. Toteutus rajattiin signaalien ja Observable-mekanismin käyttöön.

Projektia päätettiin kehittää asteittain: ensin luotiin näkymät, sen jälkeen lisättiin reititys, ja lopuksi muu sovelluslogiikka iteratiivisesti. Suunnitelma jätettiin joustavaksi, sillä oppimisen oletettiin vaikuttavan toteutuksen suuntaan projektin edetessä.

4.4.2 Arkkitehtuuri ja kansiorakenne

Angularin virallisen suosituksen perusteella moduulien sijasta sovelluksessa käytettiin komponenttipohjaista arkkitehtuuria. Komponenttipohjaisen arkkitehtuurin valintaan vaikutti myös sen yksinkertaisuus moduuleihin verrattuna, jotka lisäävät abstraktiokerroksen sovellukseen ja monimutkaistavat sovelluksen kehitystä. Päätökseen vaikutti myös se, että moduuleihin koettiin tarvitsevan tarkemman suunnittelun koko sovelluksen rakenteen kannalta, ja uusien ominaisuuksien lisääminen voisi olla vähemmän ketterää kuin komponenttipohjaisessa ratkaisuissa.

Tilanhallinta pyrittiin toteuttamaan ensiksi reaktiivisesti, RxJs-kirjaston Observables avulla. Etenkin yhdistetyn logiikan kohdalla tämä muuttui kuitenkin hankalaksi. Ratkaisuksi vaihdettiin signaalit, joidenka avulla tilan hallinta muuttui selkeämmäksi, koska niiden syntaksi oli suoraviivaisempaa. Signaalit ovat myös lähempänä React-kirjaston natiivia tilanhallinta logiikkaa, mikä saattoi helpottaa niiden omaksumista ja käyttöä. Koska kyseessä oli tekijän ensimmäinen Angular-projekti, painottui ratkaisussa ensisijaisesti koodin ymmärrettävyys ja hallittavuus. Tuotantoympäristössä tämä valinta vaatisi parempaa osaamista tilanhallinnasta, jossa myös suorituskyky ja skaalautuvuus vaikuttavat päätöksentekoon.

Kansiorakenne pyrkii noudattamaan keskisuuren sovelluksen kansiopohjaratkaisua. Tämä osoittautui suunnitteluvaiheessa loogiseksi ja helposti navigoitavaksi ratkaisuksi, mikä johti sen noudattamiseen. Core-kansiosta löytyy sovelluksen keskeisin logiikka, ja se sisältää muun muassa palvelut, tilanhallinnan, data mallit, palvelupyyntöjen välikerroksen ja reittien suojauksen. Keskeisimmät

komponentit löytyvät features-kansiosta, joka sisältää sovelluksen kannalta tärkeimmät komponentit ja kaikki sivut. Shared-kansiosta löytyy uudelleenkäytettävät komponentit joita hyödynnetään features-komponenteissa, sekä sivujen yhtenäiseen ulkoasuun liittyvät komponentit. Angular-sovelluksessa on lisäksi omat sisäänrakennetut kansiorakenteet, esimerkiksi reititystä ja sovelluksen käynnistystä varten.

4.4.3 Keskeiset komponentit, palvelut ja tilanhallinta

React-sovelluksen tapaan sovellus muodostui komponenteista. Kansiorakenteen mukaisesti features-kansio sisälsi kaikki keskeiset toiminnallisuuteen liittyvät näkymät ja komponentit.

Sääsivu koostui kolmesta eri pääkomponentista: kartasta, sijaintiehdotuksesta ja paikallisesta säätiedosta. Lisäksi näkymä sisälsi linkit suosikkilistaan ja asetuksiin. Suurin osa sovelluksen keskeisestä logiikasta liittyi tähän ominaisuuteen. Muut sovelluslogiikkaa hyödyntävät komponentit olivat käyttäjän rekisteröintiin ja kirjautumiseen liittyvät komponentit.

Sovellusta varten luotiin kolme palvelua, jotka vastasivat autentikointiin, kaupunkien hakuun ja säätietojen hakemiseen liittyvistä Http-pyynnöistä. Palvelut palauttivat vastaukset Observable-muodossa ja muodostivat kerroksen, joka vastasi palvelinpuolen kanssa tapahtuvasta kommunikoinnista.

Sovelluksen tila hajautettiin siten, että kukin store vastasi tilakokonaisuudestaan ja siihen liittyvästä toiminnallisuudesta. Erilliset tilat määriteltiin säälle, autentikoinnille, asetuksille, käyttäjänhallinalle, ja paikkaehdotuksille. Yleisimmin signaalin arvot johdettiin Observable-objektista, joita komponentit hyödynsivät. Tilat toteutettiin injektoitaviksi, jotta niitä voitiin käyttää sovelluksen muissa osissa.

Joissakin tilanteissa komponenteissa hyödynnettiin kahta eri palvelua yhdistämällä niiden toiminnallisuudet. Jälkikäteen arvioituna valmiiden funktioiden määrittäminen suoraan tilakerrokseen olisi kuitenkin selkeyttänyt rakennetta ja vähentänyt komponenttien vastuuta.

4.4.4 Käytetyt työkalut ja kirjastot

Projektissa hyödynnettiin monipuolisesti Angularin sisäänrakennettuja ominaisuuksia. Tässä kappaleessa käsitellään niistä keskeisimpiä.

Angular core-kirjasto tarjoaa sovelluslogiikan kannalta keskeiset toiminnallisuudet. Sen avulla toteutetaan Angularin keskeinen tiedonvälitysmekanismi eli riippuvuuksien määrittely ja injektointi. Kirjasto sisältää myös sovelluksen tilanhallintaan käytetyt signaalit, sekä effect-funktion, jonka avulla voidaan määritellä sivuvaikutuksia. Lisäksi komponentit määritellään tämän kirjaston tarjoamien rakenteiden avulla.

Common/http-kirjastoa käytettiin palvelinpuolelle tehtävien pyyntöjen käsittelyyn. HttpClient-luokan avulla määriteltiin HTTP-pyynnöt, ja kirjaston tarjoamilla työkaluilla voitiin asettaa tarvittavat http-parametrit. Kirjasto tarjosi myös ratkaisun palvelupyyntöjen välikerroksen toteuttamiseen, jonka avulla autentikointi token lisättiin pyyntöihin ilman, että se täytyi määritellä erikseen joka kutsussa.

Angular tarjoaa reitityksen sisäänrakennettuna. Router-kirjasto määrittelee mallin, jonka mukaisesti sovelluksen reitit konfiguroidaan. Reitit mahdollistivat myös suojausmekanismien, kuten canActivate-guardin, käytön. RouterOutlet-direktiivin avulla sovellukseen määritellään, mihin reittien komponentit renderöidään. Sovelluksessa käytettiin ohjelmallista reititystä näkymien välillä, joka toteutettiin Router-objektin avulla.

Reaktiivisesta RxJs-kirjastosta hyödynnettiin Observable-mallia, jonka avulla verkkopyynnöt käsiteltiin asynkronisesti. Observables tarjoavat useita operaatioita datan muokkaamiseen ja siirtämiseen, ja niitä käytettiin tiedon välittämiseen signaaleihin.

Common-kirjastosta käytettiin CommonModule-moduulia, joka mahdollistaa direktiivien, kuten *ngIf, käytön. Nämä direktiivit mahdollistavat ehdollisen renderöinnin, jonka avulla näkymiä voitiin muokata sovelluksen tilan mukaiseksi.

Angular CLI-komentorivityökalua käytettiin palveluiden ja komponenttien luomisessa, jonka avulla luotiin tarvittavat tiedostot Angularin mallin mukaisesti.

Leaflet-kirjastoa käytettiin myös tässä sovelluksessa, joka oli sen yleinen versio, eikä Reactille luotu versio. Toteutustapa poikkesi React-versiosta, sillä Angular ei perustu hook-malliin. Tämä teki karttatoiminnallisuuden toteutuksesta teknisesti erilaisen.

4.4.5 Haasteet

Sovelluksen tilanhallinta Observables-objektien avulla osoittautui haastavaksi, erityisesti tilanteissa, joissa yksi tila vaikutti toiseen. Ainoastaan automatisoidut ehdotukset toteutettiin kokonaan Observables-muodossa muuttamatta niitä signaaleiksi. Synkroninen tilanmuutos toimi laukaisijana asynkroniselle prosessille, jonka hallinta Observables-mallissa vaati monimutkaista operaattoriketjutusta ja sivuvaikutusten huolellista käsittelyä. Tämän vuoksi suurin osa tilanhallinnasta vaihdettiin signaaleihin. Erityisesti tilan sivuvaikutusten hallinta osoittautui selkeämmäksi signaalien effect-funktion avulla.

Angular vaatii logiikan hajauttamista. Suunnitteluvaiheessa tämä tuotti haasteita, sillä kokonaisuuden hahmottaminen oli aloittelijalle haastavaa. Vasta kehityksen loppuvaiheessa tapa koota sovelluslogiikka komponentteihin ja palveluihin alkoi tuntua intuitiiviselta. Haaste ei liittynyt niinkään abstraktiokykyyn, vaan kokonaisrakenteen hahmottamiseen.

5 Vertailu

5.1 Projektirakenne ja komponenttimalli

Molemmat projektit pyrkivät noudattamaan keskisuuren projektin kansiomallia. Angularissa suosittiin ominaisuusperustaista komponenttien erottelua, ja keskeinen sovelluslogiikka sijoittui hajautettuna core-kansioon. Reactissa ominaisuuksien sijaan käytettiin sivupohjaista rakennetta, ja sovelluslogiikka oli eriytettynä omiin kansioihinsa. Kehysten toimintatavat ohjasivat erilaiseen kansiomalliin. Esimerkiksi Reactissa hookeille ja konteksteille oli omat kansiot.

Angularissa komponentit ja palvelut luotiin Angular CLI:n avulla, mikä auttaa pitämään projektin rakenteen Angularin suosittelemanmallin mukaisena. CLI luo komponentille erilliset tiedostot logiikalle, tyyleille, rakenteelle ja testeille. Reactissa komponenttien rakenne ja logiikka on samassa tiedostossa, ja tyylit toteutettiin komponenttikohtaisessa tyylitiedostossa. Näin tyylit pysyivät helposti lokaalina kummassakin toteutuksessa. React tarjoaa myös enemmän vaihtoehtoja tyylien toteuttamiseen.

Angularissa komponentti on osa kehyksen core-kirjastoa. Komponentti on luokka, ja sillä on valmiita sisäänrakennettuja ominaisuuksia, kuten elinkaaritoimintoja. Reactissa modernein tapa on luoda komponentit funktioina.

Molemmissa sovelluksissa komponenttien rooli muodostui hyvin samankaltaiseksi. Niiden keskeinen tehtävä oli vastata näkymälogiikasta ja reagoida käyttäjän toimintoihin, jotka aiheuttivat tilan muutoksia. Angularissa komponentin roolitus syntyi automaattisemmin, koska riippuvuuksien injektointi ohjasi sovelluslogiikan eriyttämiseen. Reactissa komponentin roolitus on vapaampaa, ja siihen vaikuttivat erityisesti kontekstien ja hookien käyttö, jotka eristävät sovelluslogiikkaa komponentista.

Vaikka molemmat sovellukset käyttivät komponenttipohjaista mallia, Reactin lähestymistapa tuntui hieman ketterämmältä ja tuki käyttöliittymän pilkkomista

pienempiin osiin. Päätöksentekoa ohjasi ennen kaikkea se, kuinka paljon logiikkaa yksi komponentti sisälsi. Mikäli logiikkaa kertyi paljon, se pyrittiin pilkkomaan erilliseksi komponentiksi. Myös funktionaaliset komponentit tukivat ketterää kehitystapaa. Angularissa hyvin pienten komponenttien toteuttaminen oli hieman raskaampaa, mikä johti mieltymykseen käyttää hieman suurempia komponentteja. Komponenttien toteutus Angularissa vaatii myös enemmän määrittelyä.

Angularin tapa rakentaa komponentti sai sen näyttäytymään osana laajempaa järjestelmää, kun taas Reactissa komponentti tuntuu itsenäisemmältä osalta sovellusta. Angularin komponenttimalli ja Angular CLI:n käyttö ohjaavat kehitystä modulaarisempaan suuntaan. Angularissa komponentit voivat olla hyvinkin hienosäädeltyjä, sillä niissä voidaan määrittää esimerkiksi elinkaaritoimintoja.

5.2 Datanhallinta ja reititys

Kommunikointi sovelluksen palvelinpuolelle toteutettiin molemmissa sovelluksissa services-kansion kautta. Reactissa asynkroniset palvelupyynnöt konfiguroitiin manuaalisesti, kun taas Angularissa käytettiin siihen tarkoitettua HttpClient -kirjastoa. Angular-toteutuksessa palvelupyyntöihin lisättiin token välikerroksen avulla, koska tämä oli kirjaston tarjoamien työkalujen ansiosta suoraviivaista. Reactissa token lisättiin pyyntöihin manuaalisesti.

Yleisellä tasolla palvelupyyntöjen toteutuksessa havaittiin eroja etenkin suoraviivaisuudessa ja kehittäjän vastuussa. Reactissa manuaalisesti toteutetut palvelupyynnöt lisäävät virhealttiutta ja edellyttävät hyvää ymmärrystä esimerkiksi virhetilojen käsittelystä. Tämä kasvattaa kehittäjän vastuuta ja voi johtaa siihen, että suuremmissa sovelluksissa päädytään käyttämään ulkoisia kirjastoja palvelupyyntöjen toteuttamiseen. Angularissa HttpClient abstrahoi osan toteutuksesta kehittäjän puolesta, mikä suoraviivaistaa kehitystyötä. Myös palvelupyyntöjen välikerroksen toteuttaminen oli kirjaston avulla yksinkertaista. Lisäksi Angular

johdattelee kehittäjää reaktiiviseen ohjelmointiin, koska palvelupyynnöt palauttavat Observable-objektin. Reactissa palvelupyynnöt eivät samalla tavalla ohjaa tiettyyn tilanhallinnan ratkaisuun.

Molemmissa toteutuksissa palveluja käytettiin hyvin samankaltaisesti. Reactissa niitä kutsuttiin konteksteissa, kun taas Angularissa ne voidaan toteuttaa tiloista vastaavissa tiedostoissa. Kokonaisarkkitehtuurin kannalta tämä johtaa sovelluksissa ratkaisuun, jossa sovelluslogiikka erottuu näkymälogiikasta. Palveluiden käyttötarkoitus pysyi samankaltaisena, mutta samalla niiden käyttö johti kehyskohtaisiin eroihin, kuten Reactissa palveludatan sitomiseen hookeihin ja Angularissa signaaleihin. Angularissa palvelut injektoidaan, mikä tekee niistä selkeämmin modulaarisen osan sovelluslogiikkaa. Myös Reactissa palvelut lisäävät modulaarisuutta, vaikka tämä ei näy yhtä selkeästi rakenteessa.

Molemmat kehykset tarjoavat useita vaihtoehtoja tilanhallintaan. Reactissa tilanhallinta toteutettiin sen sisäänrakennettujen useState- ja useReducer-hookien avulla. Angularissa tila toteutettiin Observables- ja signaalipohjaisesti. Signaalit muistuttavat toiminnaltaan Reactin useState-hookia. Observables edustaa reaktiivista ohjelmointia, joka ei ole osa Reactin ydinkonsepteja, ja vaatii subscribe- ja pipe-mallin ymmärtämistä.

Molemmat kehykset tarjoavat monipuolisia ratkaisuja tilanhallintaan eri kokoisille sovelluksille, mikä lisää kehittäjien vastuuta valita tilanteeseen sopiva ratkaisu. Reactissa tämä voi johtaa ulkoisten kirjastojen käyttöön. Reactissa tila nähdään yksittäisinä tilamuuttujina ja niiden muutoksina, kun taas Angular ohjaa ajattelemaan tilaa virtaavana osana sovellusta. Angularin lähestymistapa voi olla selkeämpi suurissa sovelluksissa, joissa tilanmuutoksia tapahtuu paljon ja ne ovat keskenään riippuvaisia. Reactissa useiden toisiinsa liittyvien tilojen hallinta pelkkien sivuvaikutusten avulla voi muuttua haastavaksi. Tässä projektissa tätä eroa ei kuitenkaan voitu tarkastella syvällisesti käytännössä, koska sovelluksen tilanhallinta pysyi suhteellisen yksinkertaisena.

Angular tarjoaa reitityksen osana kehystä. Reactissa reititys toteutettiin lisäämällä React Router Dom-kirjasto. Angularissa reitit määritettiin siihen tarkoitetussa tiedostossa taulukkopohjaisesti, tarjoten hyödyllisiä ominaisuuksia, kuten lazy loading ja reitin aktivointi. Reactissa reitit ovat komponentteja, jotka määriteltiin sovelluksen käynnistävässä komponentissa. Suojatut reitit toteutetaan Reactissa omien komponenttien avulla, kun taas Angularissa tämä voidaan toteuttaa funktiolla, jota käytetään reittien määrittelyn yhteydessä. Molemmissa sovelluksissa lazy loading on helppo toteuttaa osana reittejä. Reactissa etuna on kuitenkin nopea ja joustava tapa integroida vaihtoehtoinen näkymä, jota voidaan hyödyntää esimerkiksi latausnäyttönä.

Angular route-kirjasto tarjoaa monipuolisia työkaluja, kuten aktiivisen reitin seurannan ja ohjelmallisen navigoinnin. Ohjelmallisessa navigoinnissa käytetään parametripohjaista ratkaisua, joka voi vähentää virheitä, koska koko reittiä ei tarvitse kirjoittaa käsin kuten Reactissa. React Router Dom seuraa Reactin filosofiaa ja tarjoaa esimerkiksi hookin ohjelmalliseen navigoimiseen. Angularissa on suora tuki reittiparametrien seuraamiseen, mikä suoraviivaisti sovelluksen toteuttamista, koska URL:ää hyödynnettiin tilana. Reactissa vastaava toiminnallisuus toteutettiin hookin avulla.

Sisäänrakennettu reititys vahvistaa kuvaa Angularista täysvaltaisena kehyksenä. Reactissa reititykseen valitaan erillinen kirjasto, mutta esimerkiksi React-Router Dom on kypsä ratkaisu, joka sopii hyvin Reactin komponenttipohjaiseen filosofiaan.

5.3 Käyttöliittymän toteutus ja muut erot

Käyttöliittymän rakenne eroaa syntaksiltaan kehysten välillä. React käyttää JSX-syntaksia, kun taas Angular hyödyntää HTML-pohjaista templating-kieltä, jota voidaan laajentaa Angularin omalla syntaksilla. Reactin komponenteissa renderöintilogiikka on keskeisessä roolissa, kun taas Angularissa rakenne näyttäytyy osana komponentin laajempaa kokonaisuutta, jossa vastuut ovat eroteltu selkeämmin. Reactin JSX koettiin ketterämmäksi, sillä sen avulla näkymiin oli

helppo lisätä JavaScriptin ominaisuuksia, kuten terniäärisiä operaatioita ja map-funktioita. Tämä lähestymistapa ohjaa ajattelemaan käyttöliittymän rakentamista JavaScriptin kautta. Havainto viittaa siihen, että Reactissa käyttöliittymä rakentuu ohjelmointikielen ehdoilla, kun taas Angularissa käyttöliittymä on osa ennalta määriteltyä ja kattavaa arkkitehtuurirakennetta.

Angularin Forms-kirjasto mahdollistaa lomakkeiden rakentamisen siten, että logiikka on lähestulkoon kokonaan abstrahoitu kehittäjän puolesta. Tämä poistaa tarvetta luoda kontrolloitujen lomakkeiden logiikkaa, kuten tapahtumien rekisteröintiä ja validointia itse. React-sovelluksessa lomakkeiden tila ja validointilogiikka toteutettiin manuaalisesti. Reactissa on kuitenkin mahdollista käyttää ulkoisia kirjastoja, mikäli halutaan valmiimpia lomakeominaisuuksia. Lomakkeet ovat hyvä esimerkki siitä, kuinka Angular toimii kokonaisvaltaisena kehyksenä ja ratkaisee monia ongelmia valmiiksi. Reactissa lomakkeiden manuaalinen toteuttaminen johtaa tila-ajatteluun, koska lomakkeet integroituvat osaksi komponentin tilaa.

Reactin funktionaaliset komponentit eivät tarjoa nimettyjä elinkaaritoimintoja, vaan vastaava logiikka toteutetaan hookien avulla. React-sovelluksessa sivuvaikutuksia hallittiin useEffect-hookilla, joka tarjoaa kontrolloidun ajon renderöinnin jälkeen. Angularissa on sisäänrakennetut elinkaarimetodit, ja ne on selkeästi nimetty. Signaaleihin liittyvät reaktiiviset sivuvaikutukset hoidettiin effect()-funktion avulla, joka muistuttaa ominaisuuksiltaan Reactin useEffect-hookia.

React ei tarjoa erillistä riippuvuuksien injektointijärjestelmää, mutta Context API:n käyttö mahdollistaa jaetun tilan ja logiikan välittämisen komponenttipuussa tavalla, joka muistuttaa riippuvuuksien injektointia. Provider-mekanismin avulla määritetään, mitkä komponentit voivat käyttää kontekstin tarjoamia arvoja ja toimintoja, ja komponentit hakevat nämä riippuvuudet hookien kautta. Angularissa riippuvuuksien injektointi on keskeinen ja sisäänrakennettu osa kehystä. Tämä vahvistaa vaikutusta, että Angular standardoi sen keskeisiä konventioita, kun taas React antaa vapautta valita toteutustapa.

Angular käyttää oletuksena TypeScriptiä, kun taas React-toteutus tehtiin JavaScriptillä. TypeScript tarjoaa tyyppisuojan, joka helpottaa datan rakenteen määrittelyä ja vähentää virheitä datan kulkiessa sovelluksen eri osien välillä. React on vähemmän mielipiteellinen ohjelmointikielen käytössä, mutta myös se tukee TypeScriptin käyttöä.

5.4 Yleinen kehyskohtainen yhteenveto

Kolme edellistä lukua ovat käyneet Angularin ja Reactin välisiä eroja projektipohjaisesta näkökulmasta. Erot ovat osin kontekstisidonnaisia. Tähän lukuun on johdettu ja jalostettu keskeisimmät erot niin, että ne eivät ole vain projektikohtaisia, vaan selvästi kehyskohtaisia eroja. Taulukot ovat jaettu aiempien lukujen mukaisiin teemoihin.

Taulukko 1. Yleiset erot komponenttimallissa ja projektirakenteessa

Ominaisuus	Angular	React	Erojen luonne
Komponenttimalli	Luokka + decorator	Funktiokomponentti (nykyinen suositus	Angular käyttää luokkaa, React

			funktiokomponentteja
Komponentin rakenne	Tyypillisesti erilliset tiedostot, TypeScript-logiikka, HTML-template, CSS-tyylit	Yksi tiedosto sisältäen näkymän ja logiikan	Angularissa modulaarinen lähestymistapa, Reactissa logiikka ja näkymä samassa
Tiedostojen generointi	Angular CLI luo komponentille ennalta määritellyn rakenteen	Ei oletus-generaattoria, rakenne vapaa	Angular ohjaa projektirakennetta CLI:n kautta, Reactissa ei vastaavaa ohjausta
Tyylien käsittely	Komponenttikohtainen CSS, voidaan määrittää myös decoratorissa	Useita vaihtoehtoja: komponenttikohtainen tiedosto, globaali CSS, inline-tyylit	Angular ohjaa tyylien kapselointiin, React antaa useita tpteutustapoja

Taulukko 2. Yleiset erot datanhallinnassa ja reitityksessä

Ominaisuus	Angular	React	Erojen luonne

Http-kerros	Sisäänrakennettu HttpClient	fetch tai ulkoinen kirjasto	Angular tarjoaa valmiin palvelu-pyynnön kerroksen
Palvelupyyntöjen palautusarvo	Observable-olio	Tyypillisesti Promise	Angular käyttää reaktiivista data-virtaa, Reactissa ei vastaavaa ohjausta datan suhteen
Välikerros	Sisäänrakennettu	Toteutus manuaalisesti tai kirjastoilla	Sisäänrakennettu vs. manuaalinen toteutus
Palvelukerros	Kehyksen keskeisiä ominaisuuksia, jotka injektoidaan komponentteihin	Ei palvelu-abstraktiota, logiikka jaetaan custom hookeilla tai context API:n kautta	Angularissa palveluskerros on osa arkkitehtuuria, projektissa se syntyy projekti-kohtaisesti
Tilanhallinnan perusmalli	Signals ja Observables	hooks: useState ja useReducer	Reaktiivinen tila vs. hook-pohjainen tila

Sivuvaikutuksien toteutus	effect(), Observable-operaatiot, lifecycle-metodit	useEffect	Angularissa useita toteutustapoja, Reactissa kaikki sivuvaikutukset keskitetty yhteen mekanismiin

Taulukko 3. Yleiset erot käyttöliittymässä ja muut erot

Ominaisuus	Angular	React	Erojen luonne
Templating / UI-syntaksi	HTML-template + direktiivit	JSX	Angular erottaa näkymän ja logiikan, React yhdistää ne samaan komponenttiin
Lomakehallinta	Sisäänrakennettu Forms-kirjasto	Manuaalinen toteutus tai ulkoiset kirjastot	Angular tarjoaa valmiin lomake-API: n, React siirtää vastuun kehittäjälle

Lifecycle-mekanismit	Nimetyt elinkaarimetodit	Ei nimettyjä elinkaarimetodeja, vastaavat ominaisuudet toteutetaan sivuvaikutuksien avulla	Angular tarjoaa nimetyt elinkaarimetodit, React käyttää hook-mallia
Riippuvuuksien injektointi	Sisäänrakennettu riippuvuuksien injektointi	Ei sisäänrakennettua riippuvuuksien injektointia	Angularissa DI arkkitehtuurillisesti keskiössä
TypeScript	Oletuksena	Vapaaehtoinen, JS täysin tuettu	Angular edellyttää TS-mallia oletuksena, React ei sido kieltä

5.5 Kehittäjäkokemus ja oppiminen

Reactin lähestymistapaa kuvaavat vapaus ja siihen liittyvä vastuu. Kehys tarjoaa vähemmän valmiita rakenteellisia ratkaisuja, mikä antaa kehittäjälle paljon vapautta rakenteellisiin ja arkkitehtuurisiin valintoihin, mutta samalla lisää epävarmuutta siitä, mikä lähestymistapa on tarkoituksenmukaisin. Komponenttipohjainen malli mahdollistaa ketterän kehittämisen ja käyttöliittymän pilkkomisen osiin. Abstrahointi on pitkälti kehittäjän vastuulla.

Angularissa kehys tarjoaa valmiin rakenteen, joka ohjaa kehittäjän tekemiä ratkaisuja. Tämä vähentää tarvetta kyseenalaistaa rakenteellisia valintoja samalla

tavalla kuin Reactissa, koska kehys tarjoaa selkeät toimintamallit ja sisäänrakennettuja ratkaisuja yleisiin ongelmiin. Aloittelijan näkökulmasta nämä valmiit ratkaisut ovat usein riittäviä ja tukevat järjestelmällistä kehittämistä.

Angular on kokonaisvaltainen kehys, jonka filosofian hahmottaminen vaatii enemmän perehtymistä sisäänrakennettuihin ominaisuuksiin ja niiden sovellusrajapintoihin. Reactissa puolestaan kehittäjän on arvioitava itse projektin koko ja tarpeet, sekä valittava sopivat arkkitehtuuriset ratkaisut sen perusteella.

6 Oppimisen siirtovaikutuksen analyysi

6.1 Positiivinen siirtovaikutus

Tässä ja seuraavassa luvussa tarkastellaan tutkimuskysymykseen liittyvää positiivista ja negatiivista siirtovaikutusta Reactista Angulariin.

Yksi merkittävimmistä positiivisista siirtovaikutuksista liittyy kykyyn abstrahoida sovelluslogiikkaa. Etenkin hookien ja kontekstin käyttö tukee tätä kykyä Reactissa. Tämä ajattelumalli helpotti Angularin käyttöönottoa, koska se tukee kokonaisuuden hahmottamista tilanteissa, joissa sovelluslogiikka on kerroksittainen ja hajautettu. Angularin omaksuminen voi kuitenkin tuntua haastavammalta, jos React-kehittäjä on tottunut keskittämään suurimman osan logiikasta komponentteihin. Tällöin mentaalimalli sovelluksen rakenteesta poikkeaa Angularin tavasta jäsentää sovelluslogiikkaa erillisiin osiin, mikä voi lisätä kognitiivista kuormitusta.

Tämä siirtovaikutus saattoi osittain johtua siitä, että toteutukset voitiin rakentaa rakenteellisesti samankaltaisesti. Reactissa palveluita käytettiin kontekstien kautta, kun taas Angularissa niitä olisi voinut kutsua tilanhallintaan liittyvissä palveluissa. Molemmissa ratkaisuissa tilaan liittyvät tiedostot loivat muuttujia ja funktioita, jotka olivat komponenttien käytettävissä kehyksen omien mekanismien kautta. Tämä samankaltaisuus mahdollisti tuttujen ajattelumallien soveltamista uuteen kehykseen ja siten loivensi oppimiskäyrää.

Toinen positiivinen siirtovaikutus liittyi komponenttipohjaiseen kehitysmalliin. Molemmissa sovelluksissa näkymät rakennettiin useista komponenteista, jotka muodostivat yhdessä suurempia kokonaisuuksia. Myös komponenttien rooli osana isompaa kokonaisuutta oli hyvin samankaltainen. Reactissa komponenttien pilkkominen pienempiin osiin on usein ketterää, kun taas Angularissa komponenttien luominen vaatii enemmän rakennetta ja niin sanottua boilerplate-koodia, kuten koristemääreitä. Tämän seurauksena Angular-toteutuksessa osa komponenteista jäi hieman suuremmiksi kuin React-versiossa, mikä kertoo ajattelutavan muutoksesta. Liiallinen React-mentaalimallin soveltaminen Angulariin voi myös johtaa siihen, että osa Angularin tarjoamista ominaisuuksista, kuten komponenttien elinkaaritoiminnot, jää vähemmälle käytölle.

Riippuvuuksien injektointi on Angularissa keskeinen mekanismi. Vaikka Reactin konteksti ei ole suoraan vastaava ratkaisu, se auttoi hahmottamaan ajatusta siitä, että tietyt sovelluslogiikan osat ovat useiden komponenttien käytettävissä. Kummassakin haluttuihin ominaisuuksiin pääsee käsiksi valitut komponentit. Mentaalimalli ei kuitenkaan siirry täysin suoraan, koska riippuvuuksien injektointi on laajempi mekanismi. Injektoinnin merkitys sovelluksen arkkitehtuurissa voi aluksi olla vaikeampi hahmottaa, jos React-kehittäjä on tottunut sijoittamaan suuren osan sovelluslogiikasta komponentteihin.

Siirtovaikutusta havaittiin myös Angularin @Input ja @Output-mekanismien omaksumisessa. Etenkin @Input muistuttaa Reactin propsien välittämistä komponenttipuussa. Yhdessä iteraatiossa URL:n päivittäminen toteutettiin lapsikomponentista vanhempikomponenttiin tapahtumien välityksellä. Vaikka tällainen tiedon kulku poikkeaa Reactin tyypillisestä yksisuuntaisesta datavirrasta, mekanismi oli silti suhteellisen helposti omaksuttavissa.

Myös tilanhallinnassa ja sivuvaikutuksien määrittämisessä havaittiin positiivista siirtovaikutusta. Angularin signaalit muistuttavat tietyiltä osin Reactin useState-tilaa, mikä helpotti niiden omaksumista. Myös niiden sijoittaminen muistuttaa kontekstiin sidottua tilaa. Signaalien effect-funktio puolestaan muistuttaa Reactin useEffect-hookia, mikä teki sivuvaikutusten käsittelystä intuitiivista React-

taustaiselle kehittäjälle. Sen sijaan Observables-malliin perustuva subscribe-mekanismi vaati uudenlaisen mentaalimallin datan virrasta. Tämän seurauksena Observables-arvoja muunnettiin usein signaaleiksi, mikä helpotti niiden käyttöä, vaikka ratkaisu ei välttämättä ollut optimaalinen laajemmissa sovelluksissa.

Kaikki positiivinen siirtovaikutus ei kuitenkaan välttämättä liity pelkästään mentaalimallien siirtämiseen ja teknologioiden samankaltaisuuteen. Myös yleiset kognitiiviset taidot, kuten abstrahointikyky ja ongelmanratkaisutaito, ovat voineet siirtyä aiemmasta osaamisesta uuteen kontekstiin. Esimerkiksi abstraktiot auttoivat hahmottamaan sovelluksen rakennetta ilman, että se kuormitti työmuistia merkittävästi. Tällaisia siirtovaikutuksia on kuitenkin vaikeampi mitata kuin konkreettisia teknisiä havaintoja.

6.2 Negatiivinen siirtovaikutus

Projektin aikana havaittiin myös negatiivista siirtovaikutusta. Reaktiivinen ohjelmointi osoittautui vaikeaksi hahmottaa, koska React ei korosta vastaavaa mallia samalla tavalla. Etenkin alkuvaiheessa liiallinen React-mentaalimallin siirtäminen osoittautui merkittäväksi hidastavaksi tekijäksi reaktiivisen ohjelmoinnin hahmottamiselle. Tämä haaste johti myös teknologisiin ratkaisuihin, kuten signaalien käyttöön.

Toinen negatiivinen siirtovaikutus liittyi Angularin moduuleihin. Moduulien rooli sovelluksen arkkitehtuurissa oli aluksi vaikea hahmottaa, koska vastaavaa rakennetta ei ole Reactissa. React-mentaalimallin sovittaminen tähän rakenteeseen vaikeutti niiden ymmärtämistä, minkä seurauksena projektissa ei hyödynnetty moduulipohjaista rakennetta.

Tässä luvussa esitetty analyysi oppimisen siirtovaikutuksesta vastaa tutkimuskysymyksiin siitä, miten React-tausta vaikuttaa Angular-kehyksen oppimiseen sekä missä tilanteissa havaitaan positiivista tai negatiivista siirtovaikutusta.

6.3 Lähi- ja etäsiirtovaikutus

Perkinsin ja Salomonin (1999) mukaan lähi- ja etäsiirtovaikutusta ei voida määrittää täysin tarkasti. Tämä vuoksi aihe ei ollut yhtä tarkasti havainnoitu projektissa kun positiivinen ja negatiivinen siirtovaikutus.

Projektissa voidaan todeta tapahtuvan sekä lähi- että etäsiirtovaikutusta, jos määrittäväksi tekijäksi otetaan syntaksi ja konseptuaalinen samankaltaisuus. Lähisiirtovaikutuksen piiriin voidaan mieltää esimerkiksi komponenttipohjainen kehittäminen, ja sivuvaikutuksien luominen effect()-funktion avulla. Konseptuaalisen eroavaisuuden takia propsien soveltaminen @Output-mekanismiin ja Context API:n riippuvuuksien injektointiin voidaan mieltää etäsiirtovaikutukseksi. Määrittely riippuu perspektiivistä, ja ei siten ole täysin mielekästä.

7 Johtopäätökset

7.1 Keskeiset havainnot Reactin ja Angularin eroista

Tämä luku havainnollistaa Reactin ja Angularin välisiä eroja yleisemmällä tasolla. Yhdessä vertailuluvun kanssa se muodostaa kokonaisuuden, jossa ensimmäiseen tutkimuskysymykseen vastataan sekä konkreettisten esimerkkien että yleisemmän tarkastelun kautta.

Yksi keskeisimmistä havaituista eroista liittyy vastuuseen sovelluslogiikan abstrahoinnista. Reactissa tämä korostui siten, että merkittäviä arkkitehtuurillisia päätöksiä tehdään jo varhaisessa vaiheessa. Esimerkiksi päätökset hookien ja Context API:n käytöstä vaikuttivat siihen, missä määrin sovelluslogiikkaa abstrahoidaan erilleen komponenteista. Angularissa kehys ohjaa päätöksentekoa selkeämmin. Keskeinen ohjaava tekijä on riippuvuuksien injektointi, joka ohjaa eriyttämään sovelluslogiikkaa komponenteista. Molemmissa kehyksissä abstrahoinnin määrään voi vaikuttaa, mutta Reactissa kehittäjän vapaus on suurempi, kun taas Angularissa kehys ohjaa abstrahointiin jo rakenteellisten periaatteiden kautta.

Reactissa sovelluslogiikka olisi ollut mahdollista toteuttaa pitkälti komponenttien sisällä ilman erillistä abstrahointia. Tämä edellyttäisi huolellista tilan sijoittamista ja tiedon välittämistä propsien avulla niille komponenteille, jotka tilaa tarvitsevat. Tämä ratkaisuperiaate saattaa vähentää kognitiivista kuormitusta, koska kehittäjän ei tarvitse siirtyä useiden tiedostojen välillä ymmärtääkseen yksittäisen näkymän toiminnan, ja siten olla aloittelijoille vähemmän kognitiivisesti kuormittava ratkaisu. Vapaus voi kuitenkin johtaa rakenteellisiin ongelmiin, etenkin sovelluksen kasvaessa. Mikäli tila on alun perin sijoitettu liian matalalle komponenttipuussa, uusien ominaisuuksien lisääminen voi edellyttää tilan nostamista ylemmälle tasolle tai siirtämistä kontekstiin, mikä voi aiheuttaa laajoja muutoksia useisiin komponentteihin ja hidastaa kehitystyötä.

Angularin keskeiset ominaisuudet, kuten palvelut, komponentit ja riippuvuuksien injektointi, tarjoavat heti alussa selkeän rakenteen arkkitehtuurille. Tämä voi vaatia vahvempaa abstrahointikykyä jo alkuvaiheessa, koska sovelluslogiikka jakautuu useisiin tiedostoihin ja rakenteisiin. Valmis rakenne tukee vastuunjakoa ja modulaarisuutta heti alussa. Projektissa havaittiin, että tämä rakenne helpotti uusien ominaisuuksien lisäämistä ilman merkittäviä muutoksia jo olemassa olevaan logiikkaan. Uudet toiminnallisuudet voitiin pääsääntöisesti toteuttaa erillisinä palveluina ja integroida olemassa oleviin komponentteihin, mikä viittaa siihen, että Angularin tarjoama rakenne tukee hallittua laajennettavuutta.

Toinen selkeä ero kehysten välillä liittyy ekosysteemiin. Reactin luonteesta johtuen projekteissa tukeudutaan ulkoisiin kirjastoihin, ja kehys ei ohjaa tätä valintaa. Yleisesti ulkoisia kirjastoja käytetään esimerkiksi reititykseen, verkkopyyntöihin ja lomakkeiden hallintaan. Angular tarjoaa vastaavat työkalut pitkälti sisäänrakennettuina, mikä vähentää valinnan vapautta mutta lisää yhtenäisyyttä. Samalla sisäänrakennetut ratkaisut abstrahoivat osan toteutuksesta kehyksen sisään, jolloin kehittäjän painopiste on syntaksin ja käytäntöjen hallinnassa.

React-projektiin valitut kirjastot olivat kypsiä ja React-filosofian mukaisia, mikä teki niiden integroinnista suoraviivaista. Niihin kuitenkin liittyy ylläpitovastuu. Versioiden päivitykset voivat tuoda mukanaan yhteensopivuusongelmia ja API-

muutoksia, jolloin myös olemassa olevaa koodia joudutaan päivittää. React edellyttää ekosysteemin tuntemista sekä valittujen kirjastojen vahvuuksien ja heikkouksien ymmärtämistä.

Edellä kuvatut havainnot liittyvät myös kehysten ohjaavuuteen. Reactissa kehittäjällä on enemmän vapautta päättää, mihin logiikka sijoitetaan ja mitä työkaluja projektiin sisällytetään. Angular puolestaan ohjaa kehitystä enemmän. Esimerkiksi Angular CLI:n avulla luodut palvelut ja komponentit eroavat rakenteeltaan toisistaan, ohjaten niiden erilaiseen käyttöön. Riippuvuuksien injektio ohjaa vastaavanlaisesti kohti tietynlaista rakennetta. Kehittäjän vastuu valita perustuu Angularissa sen tarjoamiin ominaisuuksiin. Ohjaavuuden seurauksena kaksi React-toteutusta voi poiketa merkittävästi toisistaan eri kehittäjien käsissä, kun taas Angular-sovellukset ovat enemmän standardoidumpia.

Joustavuudella ja ohjaavuudella on omat vahvuutensa ja heikkoutensa. Reactin vapaus korostaa tarvetta yhteisille arkkitehtuuriperiaatteille erityisesti suuremmissa kehittäjätiimeissä. React ei myöskään takaa hyvää arkkitehtuuria. Angularin rakenne voi puolestaan tuntua raskaammalta etenkin pienissä projekteissa. Ohjaavuus saattaa myös olla haaste sovelluksissa, jossa tarvitaan erittäin räätälöityjä ratkaisuita. Lisäksi Angularin käyttöönotto koettiin hitaampana, koska kehityksen alussa oli omaksuttava useita erillisiä kokonaisuuksia, joihin kehys ohjaa.

Neljäs havainto liittyy tilanhallintaan. Angular korostaa reaktiivista ohjelmointia ja siihen liittyviä rakenteita, kuten Observable-mallia sekä palveluiden tilaamista ja tilauksien hallintaa. Reactissa tiedon kulku on pääasiassa hookien, tilan, kontekstin ja propsien vastuulla. Ero näkyi erityisesti datan virtauksessa: Reactin malli koettiin selkeämmäksi ja suoraviivaisemmaksi, kun taas Angularin toteutus sisälsi useampia vaiheita ja rakenteita, mikä koettiin kognitiivisesti raskaammaksi.

Havainnot osoittavat, että React ja Angular edustavat kahta erilaista lähestymistapaa SPA-sovellusten kehittämiseen. Erojen ymmärtäminen auttaa kehystä lähestyttäessä ja tukee teknologisten valintojen tekemistä käyttötarkoituksen ja vaatimusten perusteella. Projektin perusteella voidaan päätellä, että React korostaa arkkitehtuurista kypsyyttä erityisesti silloin, kun sovellusta laajennetaan. Angular puolestaan ohjaa varhaisessa vaiheessa kohti modulaarisuutta ja vastuiden erottelua, mikä tukee uusien ominaisuuksien integrointia. React koettiin ketteräksi ja nopeaksi toteuttaa projektin kaltaisessa sovelluksessa, mikä voi olla etu prototyyppikehityksessä. Angularin rakenne puolestaan tuki johdonmukaista kehitystä, mikä edistää ylläpidettävyyttä pidemmällä aikavälillä. Ohjaavampi rakenne soveltuu myös hyvin isoille kehittäjätiimeille.

7.2 Oppimisen siirtovaikutus Reactista Angulariin

Angular ja React poikkeavat toisistaan kehyksinä, vaikka ne molemmat ovat suunniteltu SPA-sovellusten toteuttamiseen. Ne ovat kuitenkin tarpeeksi samankaltaisia, jolloin aiempaa osaamista pystyy siirtämään ja soveltamaan uuteen kehykseen. Molemmat ovat suosittuja teknologioita, joita voi tulla vastaan käytännön projekteissa.

Sovelluskehityksen aikana havaittiin sekä positiivisia että negatiivisia siirtovaikutuksia. Havaintojen perusteella React-kehittäjä pystyy siirtämään merkittävän määrän hyödyllistä osaamista Angulariin, mikä voi tehostaa kehyksen omaksumista. Vaikka tutkimusnäyttö on osin ristiriitaista (Scherer, 2016), Reactin oppiminen saattaa vaikuttaa myös laajempiin kognitiivisiin kykyihin, jotka tukevat uuden teknologian käyttöönottoa. Tämä tutkimus havainnollistaa erityisesti konkreettista ja ei-kognitiivista siirtovaikutusta, mutta kognitiivista siirtovaikutusta ei voida täysin sulkea pois, vaikka sen osoittaminen onkin haastavampaa. Myös negatiivista siirtovaikutusta havaittiin. Perkinsin ja Salomonin (1999) mukaan negatiivinen siirtovaikutus ilmenee etenkin uuden taidon oppimisen alkuvaiheilla. Tämä projekti nostaa esiin negatiivisen siirtovaikutuksen, jolloin siihen voi reagoida tulevaisuudessa.

On mahdollista, että kaikki siirtovaikutukset eivät tulleet projektin aikana esiin, koska kyseessä oli ensimmäinen kokemus Angularista. Pidemmällä aikavälillä ja kokemuksen myötä havainnot saattaisivat tarkentua. Angular-sovellus ei myöskään edusta kaikkien parhaiden käytäntöjen mukaista ratkaisuja, jolloin työn painopiste on enemmän Angular työkalujen käyttöönotto kuin niiden optimaalinen käyttö. Sovelluksen jatkokehitys ja optimointi voisivat tuottaa syvempää ymmärrystä sekä Angularista että oppimisen siirtovaikutuksista.

Lähi- ja etäsiirtovaikusta ei voida määrittää täysin yksiselitteisesti. Koska molemmat kehykset liittyvät SPA-sovelluksien kehittämiseen, React-osaamisen soveltaminen Angulariin voidaan mieltää lähisiirtovaikutukseksi. Jos kehyksiä tarkastelee lähemmin, voidaan jotkut osa-alueet mieltää etäsiirtovaikutukseksi, esimerkiksi reaktiivinen ohjelmointi Angularissa. Sekä Li & Pey Tee Oon, 2004 että Perkins & Salomon,1999 ovat havainneet että lähisiirtovaikutus on todennäköisempää kuin etäsiirtovaikutus. Ratkaisuita tarkemmin tarkasteltaessa tämä tutkimus tukee havaintoa siitä, että lähisiirtovaikutus on todennäköisempää kuin etäsiirtovaikutus.

Angularia mielletään usein vaikeammaksi omaksua kuin React (Franz & Niklasson, 2024). Yhdessä tutkimuksessa oppimisen siirtovaikutus tehostui, kun oppimisiseen käytettiin niin sanottua hybridi ratkaisua (Alrubaye, Ludi & Mkaouer, 2019). Hybridiratkaisu tekee teknologioista enemmän samankaltaiset. Eroavaisuuksien takia Reactia ei voida mieltää varsinaiseksi hybridiratkaisuksi, ja paikoitellen ajatusmallit poikkeavat huomattavasti Angulariin verrattuna. Jonkintapaisen hybridiratkaisun luominen voisi olla mahdollisesti tehokkain tapa auttaa uusia kehittäjiä omaksumaan oppimiskäyrän jyrkkyydestä tunnettu Angular. React-osaamisella on kuitenkin paljon potentiaalia toimina siltana Angularin oppimisessa ja siten tehostaa sen käyttöönottoa.

Projektin havaintojen perusteella React-osaamisen siirtovaikutus voidaan jäsentää kolmeen eri kategoriaan. Osa osaamisesta siirtyy lähes suoraan, kuten komponenttipohjainen ajattelutapa ja sovelluslogiikan abstrahointi. Osa osaamisesta vaatii ajattelutavan lievää mukauttamista, kuten esimerkiksi Context API:n

ja riippuvuuksien injektoinnin välinen ajattelumalli. Kolmannessa kategoriassa Reactin ajattelumalli ei siirry suoraan, vaan edellyttää kokonaan uudenlaisen mentaalimallin rakentamisen, kuten reaktiivisen ohjelmoinnin kohdalla. On myös huomioitavaa, että monissa tapauksissa aiempi osaaminen auttaa omaksumaan uuden aiheen Angularissa, mutta kokonaisuuden ymmärtäminen vaatii usein tuekseen dokumentaation ja kehyksen omien toimintatapojen tarkemman opiskelun.

Toinen tutkimuskysymys koskee sitä, miten aiempi perustason React-osaaminen vaikuttaa Angular-kehyksen oppimiseen. Tämän tutkimuksen perusteella aiempaa osaamista voidaan soveltaa usein varsin automaattisesti uuden teknologian oppimisessa, jolloin siitä muodostuu keskeinen oppimisstrategia. Tällainen lähestymistapa voi sekä edistää että haitata uuden kehyksen omaksumista. On kuitenkin huomioitava, että tämä tutkimus perustuu yhden kehittäjän kokemukseen, ja tulokset saattaisivat olla erilaisia, jos kehyksen oppimista olisi lähestytty toisenlaisen strategian kautta.

Tämä opinnäytetyö pyrkii toimimaan eräänlaisena tukimateriaalina kehittäjille, joilla on React-taustaa ja jotka haluavat oppia Angularin. Havainnollistetut erot auttavat kehittäjää ymmärtämään, mitä Angularilta voi odottaa. Oppimisen siirtovaikutuksen tarkastelu osoittaa, mitä ajatusmalleja kannattaa hyödyntää Angularia käytettäessä ja missä tilanteissa aiempia ajatusmalleja on tarpeen haastaa. Näin työ pyrkii tukemaan kehittäjää Angularin käyttöönoton alkuvaiheessa.

7.3 Työn rajaukset ja jatkokehityksen mahdollisuudet

Tämän projektin tarkoituksena on antaa hyödyllisiä näkökulmia React-taustaiselle kehittäjälle, joka ottaa ensiaskelmia Angularin parissa. Koska kyseessä on tapaustutkimus, tuloksia ei voida suoraan yleistää. React on joustava kehys, joten eri kehittäjillä voi myös olla eri mieltymyksiä sen käytössä, mikä vaikuttaa siihen, miten he omaksuvat uuden kehyksen. Yhden kehittäjän kokemus johtaa

välttämättäkin rajattuun näkökulmaan. Vaikka havaitut erot voivat olla samankaltaisia muille kehittäjille, oppimisen siirtovaikutus on usein henkilökohtaisempaa ja voi siten vaihdella merkittävästi.

Analyysi oppimisen siirtovaikutuksesta tehtiin retrospektiivisesti, koska työn alkuperäinen aihe ei ollut oppimisen siirtovaikutus. Tutkimus olisi voinut hyötyä projektin aikana systemaattisesti dokumentoiduista havainnoista sekä oppimisen siirtovaikutuksen teorian huomioimisesta jo projektin alkuvaiheilla. Tämä olisi voinut edistää esimerkiksi metakognitiivista itsehavainnointia, jonka on todettu tukevan siirtovaikutuksen kehittymistä (Perkins & Salomon, 1999). Lisäksi siirtovaikutuksen mekanismit olisivat voineet tulla selkeämmin esiin, jos niitä olisi tarkasteltu analyysin näkökulmasta jo kehitystyön alusta lähtien.

Projekti on rajattu eikä edusta tuotantokäyttöön tarkoitettua sovellusta. Sovelluksista puuttuu esimerkiksi täysimittainen istunnonhallinta ja suorituskykyoptimointi, mikä rajoittaa joidenkin havaintojen tekemistä. Koska sovelluksen optimointia ei sisällytetty projektin tavoitteisiin, kaikki ratkaisut eivät välttämättä edusta optimoituja tai parhaiden käytäntöjen mukaisia toteutuksia.

Jatkokehityksenä tämänkaltaista tutkimusta voisi laajentaa useamman kehittäjän toteuttamaksi, erityisesti kehittäjien, joilla on työkokemusta vastaavista teknologioista. Tällöin myös tuotantotason näkökulmat voisivat tulla paremmin esiin ja siirtovaikutuksesta saataisiin monipuolisempaa tietoa. Tutkimusta voitaisiin laajentaa myös muihin kehyksiin, kuten Vueen, jolloin vastaavaa asetelmaa voitaisiin tarkastella useamman teknologian välillä. Kaikkien näiden tutkimusten tavoitteena olisi ymmärtää paremmin, kuinka uuden kehyksen käyttöönottoa voidaan tehostaa aiemman osaamisen avulla. Lisäksi tutkimukseen voisi sisällyttää selkeämpiä mittareita, kuten aika uuden ominaisuuden toteuttamiseen, tai Likert-asteikko, jonka perusteella aiemmin opitun vaikutusta voitaisiin arvioida.

Lähteet

Alrubaye, H., Ludi, S. & Mkaouer, M.W., 2019. Comparison of block-based and hybrid-based environments in transferring programming skills to text-based environments. In: Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering (CASCON ’19). IBM Corp., pp. 100–109.

Angular. (n.d). Angular documentation. https://angular.io/docs

Elrom, S. (2022). React and Libraries. Apress.

Emmanni, PS. (2023). Comparative Analysis of Angular, React, and Vue.js in Single Page Application Development. International Journal of Science and Research (IJSR). 12. 10.21275/SR24401230015.

Franz, J. & Niklasson, M., 2024. A learning curve comparison between React and Angular. Degree project report, Chalmers University of Technology

Gackenheimer, C. (2015). Introduction to React. Apress.

GeeksforGeeks, “History and evolution of react,” GeeksforGeeks, https://www.geeksforgeeks.org/reactjs/history-and-evolution-of-react/ (accessed Nov. 4, 2025).

George, J. (2020). Performance Benchmarking: TypeScript vs. JavaScript in Modern Web Development.

Green, B. & Seshadri, S., 2013. AngularJS. Sebastopol, CA: O’Reilly Media

Gupta, S. (n.d.). Single Page Application (SPA): What’s So Good (Or Bad)? Jellyfish Technologies. Available at: https://www.jellyfishtechnologies.com/single-page-application-spa/.

Haskell, R.E., 2001. Transfer of Learning: Cognition, Instruction, and Reasoning. San Diego, CA: Academic Press

Hutagikar, V., & Hegde, V. (2020). Analysis of front-end frameworks for web applications. Int. Research J. of Engineering and Tech, 7(4), 3317-3320.

Komperla, V & Pratiba, D & Ghuli, P & Pattar, R. (2022). React: A detailed survey. Indonesian Journal of Electrical Engineering and Computer Science. 26. 1710. 10.11591/ijeecs.v26.i3.pp1710-1717.

Kornienko, DV et al. (2021). The Single Page Application architecture when developing secure Web services. J. Phys.: Conf. Ser. 2091 012065. DOI 10.1088/1742-6596/2091/1/012065

Karka, NR. (2025). State Management in Large-Scale React Applications: A Comprehensive Analysis. INTERNATIONAL JOURNAL OF ADVANCED RESEARCH IN ENGINEERING & TECHNOLOGY. 16. 108-132.

Kodali, N. (2024). The Evolution of Angular CLI and Schematics : Enhancing Developer Productivity in Modern Web Applications. International Journal of Scientific Research in Computer Science, Engineering and Information Technology. 10. 805-812. 10.32628/CSEIT241051068.

Kushwaha, A.K., Kumar, S. and Dhankhar, A. (2023). Unleashing The Power Of React Js: A Comprehensive Study on Front-End Development and framework Analysis. IJSART, 9(6).

Li, Z. & Oon, P.T., 2024. The transfer effect of computational thinking (CT)-STEM: a systematic literature review and meta-analysis. International Journal of STEM Education, 11, art. 44

Meta Platforms (n.d.) React documentation. Available at: https://react.dev/

Narayn, H. (2022) Just React!: Learn React the React Way. Apress, Berkeley, CA.

Narender, NR. (2025). Best Practices for Building Scalable Single Page Applications (SPAS). INTERNATIONAL JOURNAL OF INFORMATION TECHNOLOGY AND MANAGEMENT INFORMATION SYSTEMS. 16. 1219-1241. 10.34218/IJITMIS_16_01_087.

Perkins, D & Salomon, G. (1999). Transfer Of Learning. 11.

Piastou, M. (2023). Comprehensive Performance and Scalability Assessment of Front-End Frameworks: React, Angular, and Vue.js. World Journal of Advanced Engineering Technology and Sciences, 9(02), 366–376. DOI: 10.30574/wjaets.2023.9.2.0153

Prathap, I & Saravanan, P. (2019). Evolution of the Single Page Application in the modern web application development. Innovative Food Science & Emerging Technologies. 6. 141-145.

Rion, AH & Any, MM. The Role of APIs in Modern Web Development: A Literature Review (July 28, 2025). Available at SSRN: https://ssrn.com/abstract=5380383 or http://dx.doi.org/10.2139/ssrn.5380383

Sangarsu, R.R. (2019). Advancing Web Development with Single Page Applications (SPAs). Journal of Scientific and Engineering Research, 6(3), pp. 284–288

Savkin, V. & Cross, J., 2017. Essential Angular. Birmingham: Packt Publishing

Scherer, R. (2016). Learning from the Past–The Need for Empirical Evidence on the Transfer Effects of Computer Programming Skills. Frontiers in Psychology, 7:1390. doi: 10.3389/fpsyg.2016.01390

Scott, EA Jr. (2016). SPA design and architecture : understanding single-page web applications. Shelter Island, Ny: Manning.

TestKarts (n.d.). Evolution of Angular- AngularJS to Angular 2+, Key Features, History. Available at: https://www.testkarts.com/angularjs

van de Moere, J., Rappl, F., Bodrov-Krukowski, I., Smith, J., Wanyoike, M. & Motto, T., 2018. Learn Angular: Related Tools & Skills. SitePoint. Chapter 1: The Ultimate Angular CLI Reference Guide. Available at: https://www.oreilly.com/library/view/-/9781492068433/

Vivek, J. (2022). A COMPARATIVE ANALYSIS OF SINGLE PAGE APPLICATIONS (SPAS) AND MULTI PAGE APPLICATIONS (MPAS). 7. 271–277. 10.5281/zenodo.14956673.

Vishnuvardhan Reddy Goli, The Impact of Angularjs and React on The Evolution of Frontend Development, International Journal of Advanced Research in Engineering and Technology (IJARET), 6(6), 2015, pp. 44-53 doi: https://doi.org/10.34218/IJARET_06_06_008

Waite, R., 2023. Pros and Cons of Angular Framework You Need to Know. https://www.robinwaite.com/blog/pros-and-cons-of-angular-framework-you-need-to-know

Liitteet

Github linkit

React: https://github.com/saanavalkama/React-Weather

Angular: https://github.com/saanavalkama/Angular_weather

Backend: https://github.com/saanavalkama/Weather_Backend