jamk

SolidJS-kirjasto: Uusia mahdollisuuksia frontend-kehitykseen

Aki Berglund

Opinnäytetyö, AMK
Huhtikuu 2025
Tradenomi (AMK), Tietojenkäsittelyn tutkinto-ohjelma

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

jamk

Kuvailulehti

Berglund, Aki

SolidJS-kirjasto: Uusia mahdollisuuksia frontend-kehitykseen

Jyväskylä: Jyväskylän ammattikorkeakoulu. Huhtikuu 2025, 58 sivua.

Tietojenkäsittelyn tutkinto-ohjelma. Opinnäytetyö AMK.

Julkaisun kieli: suomi

Julkaisulupa avoimessa verkossa: kyllä

Tiivistelmä

Web-kehityksen ala on kehittynyt jatkuvasti ja uusia teknologioita ja kirjastoja on syntynyt vastaamaan kehittäjien ja yritysten muuttuvia tarpeita. SolidJS on suhteellisen uusi, moderni frontend-kirjasto, joka on saanut huomiota kehittäjien keskuudessa sen reaktiivisen ohjelmoinnin lähestymistavan ja lupaavan suorituskyvyn vuoksi. Tutkimuksen tavoitteena oli tarkastella, kuinka SolidJS-kirjasto käsittelee frontend-kirjastoille keskeisiä ominaisuuksia. Teoreettisessa osuudessa selvitettiin, kuinka SolidJS-kirjasto käsittelee näitä ominaisuuksia ja empiirisessä osuudessa pyrittiin selvittämään minkälaisia arkkitehtuurillisia eroja se jakaa muiden JavaScript-kirjastojen kanssa.

Tutkimuksessa käytettiin kvalitatiivista tutkimusmenetelmää, jonka avulla pystyttiin tarjoamaan kokonaisvaltainen ymmärrys SolidJS-kirjaston toimintatavoista ja -periaatteista. Käytössä oli koko tutkimuksen ajan ennalta laadittu vertailukehys, jonka avulla pyrittiin antamaan systemaattinen ja tasavertainen kuvaus, kuinka SolidJS-kirjastossa toteutetaan modernien frontend-kirjastojen keskeisiä ominaisuuksia. Vertailukehystä käytettiin myös vertailussa muiden suositumpien JavaScript-kirjastojen kanssa.

Tuloksina saatiin kokonaisvaltaisesti tietoa siitä, kuinka SolidJS-kirjaston avulla toteutettiin modernien frontend-kirjastojen keskeisiä ominaisuuksia, miten arkkitehtuuri ja toiminta erosivat muiden JavaScript-kirjastojen lähestymistavoista ja minkälaisia vahvuuksia ja heikkouksia SolidJS-kirjaston käytössä esiintyi.

Johtopäätöksenä voitiin todeta SolidJS-kirjaston olevan suorituskykyinen ja lupaava työkalu modernien web-sovelluksien luomiseen. Sen suosio on jatkuvassa kasvussa ja sen innovatiivinen reaktiivinen lähestymistapa herättää kiinnostusta yhä useammissa kehittäjissä. Tutkimuksessa todettiin, että SolidJS-kirjastolla on potentiaalia avata tulevaisuudessa uusia mahdollisuuksia web-sovellusten ja käyttöliittymien kehityksessä.

Avainsanat (asiasanat)

SolidJS, Web-sovellukset, Frontend-kehitys, Käyttöliittymäkehitys, Modernit JavaScript-kirjastot

Muut tiedot (salassa pidettävät liitteet)

Opinnäytetyö ei sisällä salassa pidettäviä liitteitä.

jamk

Description

Berglund, Aki

SolidJS Library: New Possibilities for Frontend Development

Jyväskylä: JAMK University of Applied Sciences, April 2025, 58 pages

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

Permission for open access publication: Yes

Language of publication: Finnish

Abstract

The field of web development has continuously evolved, with new technologies and libraries emerging to meet the changing needs of developers and businesses. SolidJS is a relatively new and modern frontend library that has gained attention among developers for its reactive programming approach and promising performance. The objective of this study was to examine how SolidJS addresses key features essential to frontend libraries. The theoretical part explored how SolidJS handles these features, while the empirical part aimed to determine the architectural differences it shares with other JavaScript libraries.

The study utilized qualitative research methods, which enabled a comprehensive understanding of SolidJS's operational principles and methodologies. Throughout the study, a pre-defined comparative framework was employed to provide a systematic and balanced analysis of how SolidJS implements the core features of modern frontend libraries. This comparative framework was also applied when benchmarking SolidJS against other popular JavaScript libraries.

The findings provided comprehensive insights into how SolidJS implements the core features of modern frontend libraries, how its architecture and functionality differ from other JavaScript libraries' approaches, and what strengths and weaknesses were observed in SolidJS during use.

In conclusion, SolidJS was found to be a highly performant and promising tool for creating modern web applications. Its popularity is steadily growing, and its innovative reactive approach continues to attract interest among developers. The study concluded that SolidJS has the potential to open up new possibilities for web applications and user interface development in the future.

Keywords/tags (subjects)

SolidJS, Web Applications, Frontend Development, User Interface Development, Modern JavaScript Libraries

Miscellaneous (Confidential information)

The thesis does not include any confidential information.

Sisältö

Kuviot

Taulukot

Käsitteet

AJAX: Asynchronous JavaScript and XML, tekniikka, jonka avulla verkkosovellus voi lähettää ja vastaanottaa tietoa palvelimelta taustalla ilman, että koko sivua tarvitsee ladata uudelleen.

API: Application Programming Interface, sovellusohjelmointirajapinta, joka määrittelee säännöt ja protokollat, joiden avulla eri ohjelmistot voivat kommunikoida ja vaihtaa tietoa keskenään.

CSS: Cascading Style Sheets, tyyliohjeet, joita käytetään HTML-elementtien ulkoasun määrittelyyn verkkosivulla.

DOM: Document Object Model, rakenteellinen esitys HTML- tai XML-dokumentista, jossa jokainen elementti esitetään objektina.

Fetch API: JavaScript-rajapinta, joka mahdollistaa resurssien hakemisen verkkopalvelimilta.

HTML: HyperText Markup Language, verkkosivujen merkintäkieli, jota käytetään sisällön rakenteen ja merkintöjen luomiseen.

HTTP: HyperText Transfer Protocol, protokolla, joka mahdollistaa tiedonsiirron verkkosivuston ja käyttäjän selaimen välillä.

JSON: JavaScript Object Notation, kevyt tiedonvälitysmuoto, jota käytetään usein verkkopalveluiden ja sovellusten välillä.

JSX: JavaScript XML, on syntaksilaajennus JavaScriptille, jonka avulla voi kirjoittaa HTML:ää muistuttavaa koodia suoraan JavaScript-tiedostoihin.

RESTful API: Representational State Transfer API, ohjelmointirajapinta, joka perustuu REST-arkkitehtuurityyliin. RESTful API käyttää HTTP-metodeja (GET, POST, PUT, DELETE) resurssien käsittelyyn.

RIA: Rich Internet Application, verkkosovellus, joka tarjoaa käyttökokemuksen, joka on lähellä perinteisten työpöytäsovellusten toiminnallisuutta.

SPA: Single Page Application, yksisivuinen sovellus, jossa koko sisältö ladataan aloitussivulla, ja sisällön päivitys tapahtuu dynaamisesti ilman, että koko sivua tarvitsee ladata uudelleen.

TypeScript: JavaScriptin laajennus, joka tuo mukanaan vahvan tyypityksen ja kehittyneitä ominaisuuksia, tehden koodista luotettavampaa ja helpommin ylläpidettävää erityisesti suurissa projekteissa.

URL: Uniform Resource Locator, verkkosivun tai resurssin osoite internetissä, joka määrittää resurssin sijainnin ja siirtymismekanismin.

XML: Extensible Markup Language, merkintäkieli, jota käytetään tiedon tallentamiseen ja siirtämiseen.

1 Johdanto

Web-kehityksen ala on jatkuvassa muutoksessa, uudet teknologiat ja kirjastot nousevat esiin vastatakseen kehittäjien ja yritysten muuttuviin tarpeisiin. Yksi näistä uusista tulokkaista on SolidJS, joka on saanut huomiota web-kehittäjien keskuudessa sen reaktiivisen ohjelmoinnin lähestymistavan ja lupaavan suorituskyvyn vuoksi. SolidJS on suhteellisen uusi frontend-kirjasto, joka on saavuttanut suosiota viime vuosien aikana ja se tarjoaakin houkuttelevan vaihtoehdon perinteisille frontend-kirjastoille, kuten Reactille.

Web-kehityksessä on jatkuva tarve hyödyntää uusimpia työkaluja ja teknologioita parantaakseen web-sovellusten suorituskykyä ja käyttökokemusta. Tässä valossa SolidJS:n kaltaiset uudet kirjastot tarjoavat mahdollisuuden kehittää moderneja ja tehokkaita web-sovelluksia, jotka vastaavat nykypäivän vaatimuksia.

Opinnäytetyön tarkoituksena on tutkia, kuinka SolidJS-kirjastossa toteutetaan modernien frontend-kirjastojen keskeisiä ominaisuuksia, kuten reaktiivisuutta ja tilanhallintaa. Tarkoituksena on antaa lukijalle käsitys modernien frontend-kirjastojen keskeisistä ominaisuuksista ja havainnollistaa kuinka SolidJS-kirjastossa toteutetaan nämä ominaisuudet. Lisäksi työssä suoritetaan vertailua SolidJS:n, Reactin sekä Svelten välillä, minkä tavoitteena on antaa lukijalle käsitys SolidJS:n eroavaisuuksista näihin kirjastoihin verrattuna. Tutkimuksessa hyödynnetään ennalta määriteltyä vertailukehystä, joka mahdollistaa SolidJS:n tarkastelun ja vertailun suorittamisen systemaattisesti ja tasapuolisesti.

Työtä lähestytään kvalitatiivisella tutkimusmenetelmällä, jonka tavoitteena on tarjota kokonaisvaltainen ymmärrys SolidJS-kirjaston toimintatavoista. Tämä saavutetaan keskittymällä SolidJS:n arkkitehtuurilliseen rakenteeseen ja sääntöihin sekä vertailemalla niitä muiden kirjastojen kanssa. Laadullinen lähestymistapa mahdollistaa perusteellisen analyysin SolidJS:n toteutustavoista modernien frontend-kirjastojen keskeisten ominaisuuksien osalta. Tämä lähestymistapa on erityisen hyödyllinen, koska SolidJS on suhteellisen uusi aihe, josta ei vielä ole toteutettu vielä paljon tutkimusta.

2 Tutkimusasetelma

Tässä luvussa esitetään opinnäytetyön tavoitteet ja niiden rajaukset sekä käytetyt tutkimusmenetelmät. Lisäksi tarkastellaan aineistonhankintaa ja sen rajauksia sekä esitetään tutkimusongelma sekä -kysymykset. Luvussa esitellään myös työssä käytössä oleva vertailukehys, jonka pohjalta SolidJS:ää tutkitaan ja verrataan muihin kirjastoihin.

2.1 Tutkimuksen tavoitteet ja rajaukset

Tutkimuksen tavoitteena on tarkastella, miten SolidJS-kirjasto käsittelee frontend-kirjastojen keskeisiä ominaisuuksia. SolidJS on suhteellisen uusi frontend-kirjasto, mutta sen ajankohtaisuus ja erityisen positiivinen vastaanotto web-kehittäjien keskuudessa tekee siitä perustellun tutkimuskohteen nykyisen käyttöliittymäkehityksen saralla. SolidJS:ää tarkastellaan sekä vertaillaan muihin frontend-kirjastoihin ennalta laaditun vertailukehyksen avulla, joka on kuvattuna luvussa 2.5.

Tutkimuksen teoreettisessa osuudessa selvitetään, kuinka nämä frontend-kirjastojen keskeiset ominaisuudet toteutetaan SolidJS:ssä. Aiheen uutuuden vuoksi lähdemateriaalin saatavuus voi olla rajallista, joten työn teoreettisessa osuudessa hyödynnetään myös koodiesimerkkejä. Empiirisessä osuudessa taas selvitetään SolidJS eroavaisuuksia toimintatavoissa muihin frontend-kirjastoihin vertailtuna tekstin sekä koodin muodossa. Vertailu tullaan suorittamaan React- sekä Svelte-kirjastojen kanssa, saman vertailukehyksen avulla. Vertailun suorittaminen juuri näihin kahteen kirjastoon on perusteltua, sillä React on vakiinnuttanut asemansa frontend-kehityksessä ja Svelte puolestaan on ollut samankaltaisessa tilanteessa SolidJS:n kanssa, mistä se on onnistunut nousemaan muiden suosituiden kirjastojen rinnalle. Opinnäytetyössä SolidJS:stä käytetään versiota 1.9, Reactista versiota 18 ja Sveltestä versiota 3.

Lisäksi tavoitteena on analysoida sen toimintatapoja sekä SolidJS-kirjaston nykytilannetta sekä potentiaalia tulevaisuudessa. Aiheen valinta on soveltuva sekä ajankohtainen ja se tarjoaa SolidJS:stä kiinnostuneille kehittäjille mahdollisuuden tutustua SolidJS:n keskeisiin toimintatapoihin frontend-kehityksessä.

Tutkimus on rajattu koskemaan SolidJS-kirjaston arkkitehtuurin ja keskeisten toiminnallisuuksien havainnointia teoreettisesta sekä käytännöllisistä näkökulmista. Työssä ei suoriteta tämän vuoksi

minkäänlaisia suorituskykymittauksia tai vertailevia testejä kirjastojen välillä. Työn rajaus perustuu tarkoitukseen säilyttää opinnäytetyölle optimaalinen pituus.

Kestävään kehitykseen liittyen aihetta voidaan perustella energiatehokkuuden lisäämisen kannalta. Verkkosivustojen ja -sovellusten energiatehokkuus on suoraan riippuvainen niissä käytetyistä teknologioista ja rakenteista. SolidJS:n tehokas mutta kevyt lähestymistapa voi auttaa edistämään verkkosivujen sekä -sovellusten energiatehokkuutta ja auttaa myös rakentamaan kestävämpää digikehitystä. Aihetta voi myös lähestyä toisesta, sosiaalisen ja taloudellisen kestävän kehityksen näkökulmasta. SolidJS:n hyvien dokumentaatio sivujen vuoksi järjestelmien ylläpito ja niiden kehittäminen voi olla yksinkertaisempaa, mikä edistää esimerkiksi niiden pitkäikäisyyttä ja vähentää niiden luomiseen kuluvia resursseja.

2.2 Tutkimusmenetelmät

Opinnäytetyössä käytetään kvalitatiivista tutkimusmenetelmää. Kvalitatiivinen tutkimus on lähestymistapa, jonka avulla pyritään ymmärtämään ilmiöitä syvällisesti ja kokonaisvaltaisesti. Tämä tapahtuu usein tarkastelemalla niiden monimutkaisia konteksteja, merkityksiä ja vuorovaikutuksia. Kvalitatiivisessa tutkimuksessa painotetaan yleensä laadullisia havaintoja ja syväluotaavaa ymmärrystä ilman tilastollisia tai määrällisiä menetelmiä. Tällainen lähestymistapa on erinomainen erityisesti silloin, kun tutkitaan aiheita, jotka eivät ole vielä saaneet paljon tutkimusta. (Kananen 2017, 32–34.)

2.3 Tutkimusongelma ja -kysymykset

Opinnäytetyön tutkimusongelma on ”Mikä on SolidJS-kirjaston nykyinen asema käyttöliittymäkehityksen alalla?”

Tutkimuksessa pyritään vastaamaan seuraaviin kysymyksiin:

1. Miten SolidJS-kirjastossa toteutetaan modernien käyttöliittymäkirjastojen keskeisiä ominaisuuksia?
2. Millä tavoin SolidJS arkkitehtuuri ja toiminta eroavat Reactin ja Svelten lähestymistavoista?
3. Mitkä ovat SolidJS merkittävimmät vahvuudet ja heikkoudet kehittäjien näkökulmasta?

Modernien käyttöliittymäkirjastojen keskeisiä ominaisuuksia käsitellään luvussa 3.3. Ensimmäiseen tutkimuskysymykseen vastataan tarkastelemalla näitä keskeisiä ominaisuuksia SolidJS:ä luvussa 4. Toiseen tutkimuskysymykseen vastataan luvussa 5, missä käsitellään SolidJS:n, Reactin ja Svelten tyypillisiä ominaisuuksia frontend-sovelluksissa. Nämä tyypilliset piirteet määritellään luvun 5 alussa. Tutkimuskysymykseen kolme vastataan pohtimalla SolidJS:n vahvuuksia ja heikkouksia luvussa 6.

2.4 Aineistonhaun kuvaus ja rajaus

Aineistonhaku on toteutettu hakemalla tietoa eri hakupalveluita käyttäen kuten Skillsoft Books ITPro ja Janet Finna -palvelu. Työssä on myös käytetty lähteinä kirjastojen omia dokumentaatioita. Tutkimuksen aiheen takia lähteet ovat enimmäkseen kansainvälisiä. Työssä on pyritty tarkastelemaan lähteitä kriittisesti sekä tutkimaan erilaisia lähteitä. Aiheen uutuuden vuoksi, tutkimusartikkeleita oli haastavaa löytää. Lähteinä on hyödynnetty kirjoja, E-kirjoja, SolidJS:n dokumentaatiota ja verkkolähteitä. Hakusanoina on käytetty aiheeseen liittyvää sanastoa, kuten JavaScript, web-applications, framework ja SolidJS.

2.5 Vertailukehys

Tässä työssä käytettävä vertailukehys on laadittu aiempien vertailujen pohjalta, joissa on vertailtu JavaScript-kehyksiä. Erityisesti vertailukehyksen pohjana on käytetty vertailuja, jotka käsittelevät frontend-kehityksen keskeisiä osa-alueita. Aiemmin tarkastelemani vertailut (Choosing the Best JavaScript Framework, 2024.), (Key Considerations for Choosing the Top Front-End Frameworks, 2024.) ja (Framework main features, 2024.) ovat tarjonneet hyödyllisiä näkemyksiä siitä, mitkä osa-alueet ovat keskeisiä JavaScript frontend-kirjastojen arvioinnissa. Näitä lähteitä soveltamalla ja yhdistämällä olen rakentanut oman vertailukehykseni.

Kehys kattaa seuraavat osa-alueet:

- Syntaksi
- Komponentit ja tapahtumien hallinta
- Reaktiivisuus
- Tilanhallinta
- Reititys

Näitä teknisiä ominaisuuksia tarkastellaan käytännön esimerkein ja selityksin, jotta voidaan havainnollistaa, miten SolidJS-kirjasto toteuttaa niitä ja kuinka sen toimintatavat eroavat Reactin ja Svelten toimintatavoista. Lisäksi olen ottanut vertailuun mukaan yhteisön ja ekosysteemin merkityksen, koska laaja tuki ja aktiivinen kehittäjäyhteisö ovat tärkeitä tekijöitä pitkäaikaista käyttöä ja kirjastojen jatkuvaa kehitystä arvioitaessa. Tämä vertailukehys mahdollistaa systemaattisen ja tasapuolisen tavan tarkastella SolidJS:n toimintatapoja sekä vertailla sen eroavaisuuksia Reactin ja Svelten kanssa.

Menetelmän vaiheet:

1) Vaiheessa yksi opinnäytetyön alussa määritellään vertailukehys luomalla se soveltuvien lähteiden perusteella ja esittelemällä sen sisältö sekä käyttö.
2) Vaiheessa kaksi, vertailukehystä käytetään apuna SolidJS:n tarkemmassa esittelyssä ja pureudutaan aiheeseen kehyksen mukaisesti.
3) Vaiheessa kolme suoritetaan vertailukehyksen avulla ja koodi esimerkkejä hyödyntäen vertailu SolidJS:n, Reactin sekä Svelten välillä.
4) Vaiheessa neljä pyritään hyödyntämään vertailukehystä johtopäätösten laatimisessa ja tulosten analysoinnissa.

Vertailukehyksen käyttö tutkimusmenetelmänä on perusteltua, jotta voidaan saavuttaa opinnäytetyöhön liittyvät tavoitteet. Näiden tavoitteiden saavuttamiseksi vertailun avulla saadaan osviittaa samankaltaisuuksista ja eroavaisuuksista muihin kirjastoihin ja pystytään luomaan lukijalle kokonaisvaltaisempi ymmärrys aiheesta.

3 Frontend web-sovellukset

Tässä luvussa perehdytään web-sovellusten toiminnan perusperiaatteisiin ja frontend teknologioiden toimintaan. Tavoitteena on tarjota lukijalle perusymmärrystä, kuinka web-sovellukset toimivat sekä mistä teknologioista JavaScript frontend-kirjastot pääasiassa muodostuvat. Luvun aikana perehdytään aihealueisiin kuten SPA, JavaScript sekä modernin frontend-kirjaston perusominaisuudet.

3.1 Web-sovellusten toiminta

World Wide Web eli lyhyesti Web on jatkuvasti laajeneva tietotila, joka rakentuu Internetin päälle. Web-resurssit ovat saavutettavissa niiden osoitteen, URL:n, kautta ja voivat sisältää hyperlinkkejä muihin resursseihin. Kaikki nämä toisiinsa linkitetyt resurssit muodostavat valtavan verkon, joka on vertauskuvallisesti kuin hämähäkin verkko. Verkkoon soveltuvia asiakirjoja kutsutaan verkkosivuiksi. Ne on ryhmitelty yhteen verkkosivustoilla ja niitä tarkastellaan erityisen ohjelmiston, selaimen, avulla. (Pesquet, 2023.)

Web-sovellus on internetissä toimiva verkkosivu, joka käyttää selainta käyttöliittymän esittämiseen. Web-sovellus keskittyy yleensä yhteen tiettyyn aiheeseen tai tehtävään, ja siinä on käyttöliittymä, joka mahdollistaa käyttäjän toiminnan siellä yhdellä tai useammalla tavalla. Nämä muistuttavat toiminnaltaan natiiveja työpöytä- tai mobiilisovelluksia. Tämä eroaa tavallisesta verkkosivusta, joka yleensä käsittelee useita aiheita tai tehtäviä ja on käyttöliittymältään sellainen, joka pääasiassa mahdollistaa vain sivustolla navigoinnin. Pääsääntöisesti web-sovellukseen kuuluu kolme keskeistä osa-aluetta, jotka ovat frontend, backend sekä tietokannat. Frontend on se osa verkkosivustoa, jonka verkkoselain esittää käyttäjän selainikkunassa ja jonka kautta käyttäjä näkee ja ohjaa verkkosivuston tai sovelluksen toimintaa. Backend taas on se verkkosivun osa, joka sijaitsee verkkopalvelimella. Se huolehtii verkkosivun toiminnasta ja tietokannat tietojen hallinnasta. (McFedries, 2018.)

Usein web-sovellusten tai sivustojen frontend puolen rakentamisesta puhuttaessa mainitaan kolme kieltä: HTML, CSS ja JavaScript. HTML toimii sivuston sisällön rakenteena antaen sille jäsennyksen ja semantiikkaa. CSS puolestaan määrittelee säännöt HTML-rakenteille ohjaillen siten, miten sisältö esitetään sivustolla. JavaScript tuo sivustolle dynaamisuutta ja mahdollistaa käyttäjän vuorovaikutuksen. (Duckett, 2014)

Tavallisessa web-kehitysskenaariossa noudatetaan synkronista vaihtoa. Tässä tilanteessa, kun sivustolla suoritetaan tapahtuma, selain lähettää palvelimelle pyynnön, joka palauttaa täysin uuden verkkosivun vastauksena pyyntöösi. Tämän vuoksi tällä tavoin rakennetuilla verkkosivuilla esiintyy pidempiä latausaikoja sekä rajoitettua vuorovaikusta, kun jokainen tapahtuma aiheuttaa koko sivun uudelleen latautumisen. Toisessa web-kehitysskenaariossa taas noudatetaan asynkronista vaihtoa. Tässä mallissa pyritään välttämään uuden kokonaisen sivun lähettämistä jokaisen sivuston

tapahtuman toimesta. Jokaisen tapahtuman toimet sivulla välitetään JavaScript-tapahtumankäsittelijöiden kautta ja HTTP-pyynnöt lähetetään palvelimelle ilman, että sivun navigointi keskeytyy. Vain tarvittavat osat sivusta päivitetään pyyntöjen tuloksilla. Vaikka asynkronista vaihtoa käyttävän sivuston rakentaminen voi olla haasteellisempaa tämä voi johtaa lyhyempiin latausaikoihin, parempaan käyttäjäkokemukseen ja vuorovaikutukseen, mikä on verrattavissa natiivisovelluksiin. (Pesquet, 2023.)

SPA-sovellukset toimivat yksisivuisena verkkosivuna, eli aina kun käyttäjä tekee sivustolla jotain, se ei näytä uutta HTML-sivua, vaan se näyttää tiettyjä DOM:n osia. Sen esityskerroksen logiikka on siis siirretty palvelimelta asiakas puolelle JavaScriptin avulla. Tämä arkkitehtuuri tarjoaa sujuvamman käyttökokemuksen, sillä sivun päivityksiä ei tarvita, ja tietopyynnöt tehdään asynkronisesti, usein JSON-muodossa. (Emmit, Scott, 2015.) SPA-sovelluksia tukevat modernit JavaScript-kehykset, kuten React ja Angular, jotka helpottavat monimutkaisten käyttöliittymien rakentamista.

Tekniikkakokonaisuutta mikä mahdollisti näiden web-sovellusten luomisen, kutsuttiin nimellä AJAX. Kun perinteisessä web-kehitysskenaariossa käytettiin synkronisia pyyntöjä, jossa sivuston käyttö oli estetty odottaen palvelimen vastausta, niin AJAX-malli puolestaan käytti asynkronisia pyyntöjä. Tämä mahdollisti datan noutamisen taustalla tarvittaessa ja mahdollisti paremman käyttäjäkokemuksen. (Pesquet, 2023.)

Aiemmin XML oli yleisesti käytetty tiedonsiirtomuoto AJAX-pyyntöjen käsittelyssä. Kuitenkin nykyään JSON on yleistynyt huomattavasti ja korvannut XML:n monissa sovelluksissa. JSON:n suosion taustalla ovat sen keveys, helppolukuisuus ja -kirjoitettavuus. Sitä käytetään laajasti RESTful API -palveluissa ja sen käyttö on lisääntynyt verkkokehityksessä. Tämä muutos johtuu osittain siitä, että JSON on luonnollinen valinta JavaScript-sovelluksissa, mikä helpottaa sen integrointia ja käsittelyä JavaScript-pohjaisissa sovelluksissa. Tästä syystä AJAX-tekniikan käyttö on vähentynyt ja modernimpi Fetch API on noussut suosituksi vaihtoehdoksi verkkopalvelupyynnöissä. Fetch API tarjoaa kehittäjille paremman käyttökokemuksen ja modernimman käyttöliittymän verrattuna vanhempaan XMLHttpRequest-olioon, mikä tekee siitä houkuttelevan vaihtoehdon nykyaikaisille web-sovelluksille. (MDN Web Docs. 2024)

3.2 Frontend-teknologiat

JavaScript on korkean tason, dynaamisesti tyypitetty, prototyyppipohjainen ohjelmointikieli, joka sisältää olio-ohjelmointiin liittyviä piirteitä. Sen kehitti alkujaan Netscape Communications yrityksessä työskentelevä Brendan Eich vuonna 1995. Se luotiin alun perin tuomaan verkkosivustoille dynaamisuutta ja interaktiivista vuorovaikutusta käyttäjien kanssa. Web 2.0:n ja AJAX:in esiin-tulo 2000-luvun alussa oli merkittävä JavaScriptin roolituksen kannalta web-kehityksessä. JavaScript muuttui kielestä, millä luotiin dynaamisuutta ja interaktioita verkkosivustoille, kieleen, millä pystyi luomaan RIA-sovelluksia eli web-sovelluksia, jotka käyttäytyvät samaan tapaan kuin natiivit työpöytä sovellukset. (Wirfs-Brock, 2020.)

JavaScript on nykyään laajalti käytössä kaikkialla, eikä se rajoitu enää pelkästään verkkosivustojen dynaamisuuden ja interaktiivisuuden tuomiseen. NodeJS:n myötä JavaScript-sovellusten rakentaminen on mahdollista myös selaimen ulkopuolella, avaten oven palvelinpuolen kehitykselle JavaScriptin avulla. Älypuhelinten ja tablettien suosion kasvu on tuonut mukanaan useita eri käyttöjärjestelmiä, kuten iOS ja Android. Tämä on johtanut Cross-Platform-sovelluskehitys työkalujen nousuun, jotka mahdollistavat yhden mobiilisovelluksen kehittämisen kaikille eri käyttöjärjestelmille. Useimmiten nämä työkalut perustuvat JavaScriptiin. (Pesquet, 2023.) Nykyään JavaScript onkin Stackoverflown (2023) julkaiseman kyselyn mukaan kahdennettatoista vuotta peräkkäin käytetyin ohjelmointikieli kehittäjien keskuudessa.

JavaScript-kirjastot tarjoavat valmiiksi kirjoitettua koodia, mikä helpottaa kehittäjiä nopeuttamaan kehitysprosessiaan, yksinkertaistamaan koodaustehtäviä ja parantamaan verkkosovellustensa toiminnallisuutta ja vuorovaikutteisuutta. Hyödyllisyyskirjastoista, kuten Lodashista, aina datan visualisointikirjastoihin, kuten D3.js:iin, on laaja ja syvä valikoima valittavana. Jos aiot rakentaa jotain tyhjästä, on todennäköistä, että ainakin yksi avoimen lähdekoodin kirjasto on jo olemassa. (Frisbie, 2024)

HTML on standardoitu tekstipohjainen merkkauskieli, jota käytetään elementtien, kuten tekstin, kuvien ja hyperlinkkien esittämiseen verkkosivustoilla. Lisäksi HTML-dokumentit voivat sisältää muita metatietoja, kuten avainsanoja, jotka auttavat kuvaamaan sivun sisältöä hakukoneille.

HTML:llä luotuja dokumentteja voi näyttää millä tahansa verkkoselaimella, minkä vuoksi sitä pidetään verkkosivujen perustana. Koska HTML on puhdas tekstipohjainen kieli, dokumentteja voidaan muokata ja tallentaa millä tahansa tekstieditorilla. (Wolf. 2023)

CSS kielen tarkoituksena on yksinkertaistaa verkkosivujen esittämisen prosessia. Se mahdollistaa tyylien soveltamisen HTML-dokumentteihin ja kuvailee selaimelle kuinka eri HTML-elementit tulisi näyttää sivustolla. CSS antaa kehittäjille ja suunnittelijoille mahdollisuuden määrittää jokaisen elementin uniikilla tai globaalilla tavalla. (Grant. 2018)

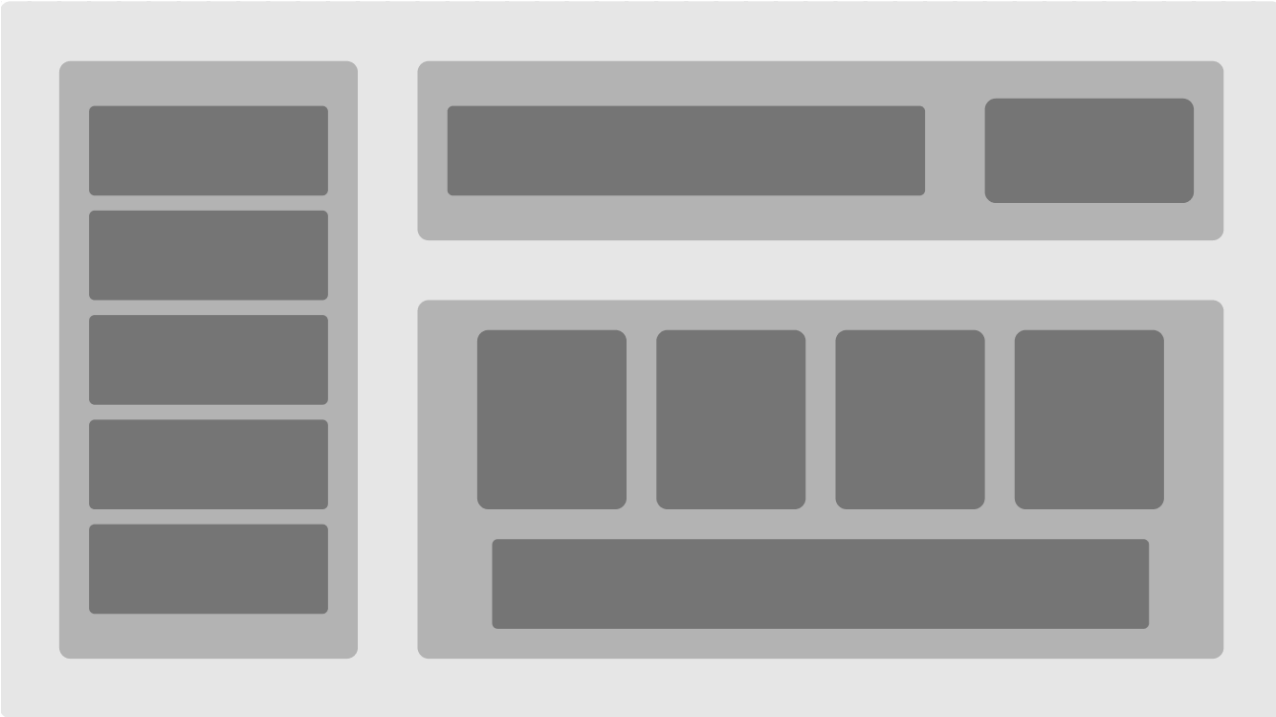
3.3 Frontend-kirjastot

Ohjelmistokehykset (framework) ovat valmiita malleja tietyn ohjelmointikielen käyttöön, ja niiden avulla voidaan rakentaa sovelluksia. Kehysten käyttö mahdollistaa sovellusten ohjelmointikoodin yhtenäisen ylläpidon, kun ne laajenevat. Nämä kehykset tarjoavat valmiiksi rakennettuja, vahvoja mekanismeja yleisiin tehtäviin, kuten komponenttien määrittelyyn, uudelleenkäyttöön, datan liikkeen ohjaukseen ja reititykseen. Niiden päätarkoituksena on yksinkertaistaa ohjelmistokoodia tehden siitä helpommin ylläpidettävää. (Frisbie, 2024.)

Frontend-kehykset ovat valmiiksi kirjoitettuja kokoelmia tai kirjastoja, jotka sisältävät standardoituja HTML-, CSS- ja JavaScript-koodeja. Ne tarjoavat joukon työkaluja ja erilaisia käytäntöjä esimerkiksi käyttöliittymän rakentamiseen, tapahtumien käsittelyyn tai tilanhallintaan. Niiden avulla kehittäjät voivat rakentaa sivustoja tai sovelluksia tehokkaammin, vähentää koodin monimutkaisuutta ja keskittyä enemmän itse sovelluslogiikkaan ja sovelluksen kehittämiseen, sen sijaan että keksisivät pyörän uudelleen. (Mdn web docs, 2024.)

Seuraavaksi käydään läpi yleisimpiä JavaScriptin frontend-kirjastojen avainominaisuuksia, jotka parantavat web-kehityksen tehokkuutta ja käytettävyyttä. Näitä ominaisuuksia ovat muun muassa komponenttipohjainen arkkitehtuuri, virtuaalinen DOM, tilanhallinta, tiedonsidonta, sekä reititys. Näiden avulla kehittäjät voivat luoda dynaamisia ja interaktiivisia sovelluksia, jotka ovat helposti ylläpidettäviä ja suorituskykyisiä.

Komponenttipohjaisessa arkkitehtuurissa käyttöliittymä jaetaan pienempiin, itsenäisiin komponentteihin. Tämä lähestymistapa selkeyttää koodin rakennetta ja tekee siitä helpommin luettavaa. Jokainen komponentti vastaa pienestä osasta käyttöliittymää ja sisältää siihen liittyvän logiikan. Tämä mahdollistaa komponenttien uudelleenkäytön eri osissa sovellusta, mikä vähentää koodin toistoa ja tekee muutosten tekemisestä yksinkertaisempaa ja vähemmän virhealtista. Komponenttipohjainen lähestymistapa helpottaa erityisesti monimutkaisten ja laajojen käyttöliittymien hallintaa ja ylläpitoa. (Hands on React, 2024.)



Kuvio 1: Komponenttipohjainen arkkitehtuuri

DOM on strukturoitu esitys verkkosivun tai sovelluksen HTML-elementeistä, ja se edustaa sovelluksesi tai verkkosivusi käyttöliittymää. DOM esitetään puumaisena tietorakenteena, jossa on solmuja. Jokainen solmu vastaa verkkodokumentissa olevaa käyttöliittymäelementtiä. Tämän ansiosta web-kehittäjät voivat helposti muokata sisältöä JavaScriptin avulla. (Immukul, 2023)

Virtuaalinen DOM on kuin kevyt kopio oikeasta DOM-puusta. Jokaista alkuperäisessä DOM:ssa olevaa objektia kohden on vastaava objekti virtuaalisessa DOM:ssa. Se on täysin samanlainen, mutta sillä ei ole suoraa kykyä muuttaa dokumenttia. Oikean DOM:n päivittäminen on hitaampaa kuin virtuaalisen DOM:n. Tämä johtuu siitä, että joka kerta kun sovelluksen tila muuttuu tai siihen

lisätään jotain uutta, luodaan virtuaalinen DOM-puu. Tässä puussa jokainen sovelluksen elementti on puun solmu. Aina kun minkä tahansa elementin tila muuttuu, luodaan uusi virtuaalinen DOM-puu. Tätä uutta virtuaalista DOM:ia verrataan edelliseen virtuaaliseen DOM:iin, ja näiden perusteella tehdään muutokset oikeaan DOM:iin, joka renderöidään sivulle. Virtuaalisen DOM:n hyöty on siinä, että se minimoi suorat päivitykset oikeaan DOM:iin, mikä parantaa suorituskykyä. (Narayn, 2022.)

Tilanhallinta on keskeinen prosessi, jonka avulla sovelluksen tila ja siihen liittyvät syötteet hallitaan ja ylläpidetään johdonmukaisesti. Sovelluksen tila viittaa tallennettuun tietoon, kuten käyttäjän syötteisiin tai käyttöliittymän tiloihin, joita käytetään sovelluksen toiminnan ohjaamiseen ja hallintaan. Tilanhallinnan avulla kehittäjät voivat varmistaa, että tilamuutokset päivittyvät oikein sovelluksen eri osissa, mikä parantaa sovelluksen johdonmukaisuutta ja toimintakykyä. (Bigelow, 2024.)

Tiedonsidonta yhdistää käyttöliittymäelementit datamalleihin, mahdollistaen dynaamisten ja responsiivisten sovellusten luomisen. Se yksinkertaistaa tilanhallintaa ja käyttöliittymän päivitystä synkronoimalla automaattisesti datan muutokset mallin ja käyttöliittymän välillä. Tämä vähentää tarvittavan koodin määrää ja pienentää virheiden riskiä. (Das, 2024.)

Yleisesti, on olemassa kahta erilaista tiedonsidonta tapaa:

- **Yksisuuntaisessa tiedonsidonnassa** data virtaa yhdessä suunnassa, eli mallista käyttöliittymään. Kun data mallissa muuttuu, käyttöliittymä päivittyy automaattisesti vastaamaan muutoksia. Tämä malli ei kuitenkaan salli käyttöliittymän muutosten vaikuttavan takaisin dataan mallissa. (Das, 2024.)

- **Kaksisuuntaisessa tiedonsidonnassa** data virtaa molempiin suuntiin. Muutokset datamallissa päivittävät käyttöliittymän, ja samalla käyttöliittymässä tehdyt muutokset päivittyvät automaattisesti takaisin datamalliin. Tämä mahdollistaa dynaamisen ja interaktiivisen käyttäjäkokemuksen erityisesti lomakekentissä ja muissa syötteiden käsittelyssä. (Das, 2024.)

Reititys mahdollistaa sovelluksen sisäisen navigoinnin ilman, että sivua tarvitsee ladata uudelleen. Tämä saavutetaan lataamalla vain yksi HTML-kehikko, jonka DOM-rakennetta päivitetään dynaa-

misesti käyttäjän navigoidessa eri URL-osoitteiden välillä. Reitityksellä voidaan hallita, mitä komponentteja tai näkymiä näytetään kullekin URL-osoitteelle. Vaikka reitityksen voi rakentaa käyttämällä JavaScriptin ja selaimen natiiveja ominaisuuksia, useimmissa ohjelmistokehyksissä reititys on integroitu kehyksen sisään tai saatavilla erillisinä kirjastoina, jotka tekevät kehitysprosessista helpompaa. Nämä kirjastot tukevat myös selainhistoriaa, mikä mahdollistaa käyttäjien navigoinnin takaisin ja eteenpäin sovelluksessa samalla tavoin kuin perinteisillä verkkosivustoilla. (MDN Web Docs, 2024.)

4 SolidJS

Tässä luvussa tarkastellaan SolidJS:ää luvun 2.5 vertailukehyksen mukaisesti, mikä mahdollistaa järjestelmällisen ja johdonmukaisen analyysin. Tämän avulla voidaan arvioida SolidJS:n ominaisuuksia yhtenäisellä ja samalla objektiivisella etäisyydellä, kuin myöhemmin suoritettavissa vertailuissa muiden kirjastojen välillä. Tämän avulla säilytetään tutkimuksessa johdonmukaisuus ja lisätään samalla sen luotettavuutta sekä selkeyttä.

SolidJS on moderni JavaScript ohjelmistokehys, jonka pääkehittäjänä on toiminut Ryan Carniato. Kehitys on alkanut jo yli 5 vuotta sitten ja on viimeisten vuosien aikana herättänytkin kovasti innostusta kehittäjien keskuudessa. Se on suunniteltu rakentamaan toimivia ja responsiivisiä ratkaisuja käyttöliittymissä. Se sopii erinomaisesti kaiken tasoisille kehittäjille sen yksinkertaisen ja ennustettavan kehityskokemuksen perusteella. (Overview, 2024)

4.1 Syntaksi

SolidJS hyödyntää JSX-syntaksia, joka on JavaScriptin laajennus. JSX mahdollistaa HTML- koodin kirjoittamisen suoraan JavaScript-tiedostoon, yhdistäen renderöintilogiikan ja sisällön yhteen paikkaan. Tämä tekee komponenttien luomisesta tiivistä ja selkeälukuista. SolidJS tarjoaa myös TypeScript-tuen sen käyttämiseen yhdessä JSX:n kanssa. SolidJS käyttää JSX:ää palauttamaan DOM-elementtejä suoraan ilman välivaiheita, mikä tekee siitä erityisen tehokkaan. Tämän lisäksi JSX:n avulla voidaan käyttää dynaamisia lausekkeita suoraan HTML-koodissa sulkemalla muuttujat ja funktiot aaltosulkeisiin, mikä lisää joustavuutta ja helpottaa datan ja logiikan käsittelyä käyttöliittymässä. SolidJS hyödyntää myös hienojakoista reaktiivisuutta, missä päivitetään vain tarvittavat osat DOM:sta kun tilassa tapahtuu muutoksia. (Understanding JSX, n.d) (ks. Kuvio 2.)

```
export default function Component() {
  const person = { firstName: 'John', lastName: 'Johnsson' };

  return (
    <>
      <p>
        Hi! My name is {person.firstName} {person.lastName}.
      </p>
    </>
  );
}
```

Kuvio 2: Syntaksia

4.2 Komponentit ja tapahtumien hallinta

Komponentti on itsenäinen yksikkö, jota voidaan käyttää uudelleen eri projekteissa tai tilanteissa. Komponentit on suunniteltu toimimaan itsenäisesti ja niillä on selkeät toimintatavat, miten kommunikoida muiden komponenttien kanssa. Yleensä komponentit koostuvat useista moduuleista, jotka yhdessä muodostavat toimivan kokonaisuuden, mikä tekee ohjelmistokehityksestä tehokkaampaa ja modulaarisempaa. (What is a software component?, 2024)

SolidJS sovellus hyödyntää komponenttipohjaista arkkitehtuuria, missä komponentit järjestetään hierarkkiseen rakenteeseen, mitä kutsutaan komponenttipuuksi. Tässä rakenteessa sovelluksen pääkomponentti kuuluu ylimpään kerrokseen, jonka alle lapsikomponentit sijoittuvat. Näillä voi myös olla omia alikomponentteja ja tätä jatkamalla sovellus voidaan jakaa pienempiin osiin, jotka ovat uudelleen käytettäviä. (Component trees, n.d)

Komponenttien elinkaari eroaa huomattavasti muista kehyksistä, sillä se perustuu reaktiivisuuteen ja se suoritetaan vain kerran luomisen yhteydessä. Tämän jälkeen, komponenttia ei ajeta enää uudelleen. Luomisen yhteydessä Solid asettaa reaktiivisen järjestelmän, mikä seuraa tilamuutoksia, ja päivittää komponenttia vain tarvittavin osin ilman uudelleenajamista. On siis tärkeää, että komponentti alustetaan oikein heti alusta alkaen, sillä sen logiikkaa ei käydä jatkuvasti läpi. `onMount`-funktion avulla toiminto voidaan suorittaa heti, kun komponettin on renderöty ja kiinnitetty DOM-puuhun. Solidissa on myös `onCleanup`-funktio, jonka avulla voidaan poistaa esimerkiksi tapahtumankuuntelijoita sen jälkeen, kun komponentti on poistettu DOM:sta tai kun sen reaktiivinen tila on hävitetty. (Component lifecycles, n.d)

SolidJS tarjoaa kolme pääasiallista tapaa käsitellä tapahtumia:

- Luonnolliset tapahtumat
- Delegoidut tapahtumat
- Mukautetut tapahtumat.

Luonnolliset tapahtumat liitetään suoraan tiettyihin DOM-elementteihin. Ne tarjoavat tarkkaa kontrollia, mutta suurissa sovelluksissa ne voivat kuormittaa suorituskykyä huomattavasti. Delegoidut tapahtumat taas ovat yhteydessä ylemmälle tasolle, kuten dokumenttiin, mistä ne delegoidaan alaspäin. Tämä parantaa suorituskykyä sekä vähentää muistinkulutusta suurissa tapahtumamäärissä. Mukautetut tapahtumat antavat mahdollisuuden luoda omia tapahtumatyyppejä ja käsitellä niitä erikseen. Niissä voidaan esimerkiksi hyödyntää `on:` -syntaksia, mikä tarjoaa joustavuutta tilanteesta riippuen. (Event handlers, n.d)

Tapahtumien sitominen voidaan optimoida välittämällä taulukko tapahtumankäsittelijän sijaan, jolloin toinen taulukon elementti välitetään käsittelijälle ensimmäiseksi argumentiksi. Tämä vähentää ylimääräistä suorituskykykuormaa, kuten JavaScriptin `bind`-metodin käyttöä. Tapahtumankäsittelijät eivät ole osa reaktiivista järjestelmää, joten ne eivät automaattisesti päivity signaalien muutoksiin. Käsittelijät kuitenkin voidaan suunnitella kutsumaan reaktiivisia lähteitä tarvittaessa ilman, että ne itse ovat reaktiivisia. (Event handlers, n.d)

Propsien avulla voidaan välittää dataa vanhemmalta komponentilta lapsikomponentille. Ne mahdollistavat komponenttien uudelleenkäytön eri tilanteissa ja dynaamisen sisällön näyttämisen muuttamatta komponentin sisäistä logiikkaa. Ne toimivat ikään kuin komponentin ”syötteenä”, jota komponentti käyttää tiettyjen näkymien tai toimintojen tuottamiseen. (Props, n.d) (ks. Kuvio 3.)

```
function Card(props) {
  return (
    <div>
      <h2>{props.title}</h2>
      <p>{props.content}</p>
    </div>
  );
}

export default function App() {
```

```
  return (
    <div>
      <Card
        title="First Card"
        content="This is the content of the first card."
      />
      <Card
        title="Second Card"
        content="Here is the second card with different content."
      />
      <Card
        title="Third Card"
        content="This card also uses the same component."
      />
    </div>
  );
}
```

Kuvio 3: Propsien välitys

SolidJS:ssä elementtien tyylittäminen muistuttaa pitkälti HTML-elementtien tyylitystä `class`- ja `style`-atribuuttien kanssa. `Class`-attribuutilla voi tyylittää useita elementtejä, jotka käyttävät samoja CSS-sääntöjä. Tämän avulla tyyleistä saadaan uudelleenkäytettäviä sekä helpommin ylläpidettäviä. `Style`-attribuutin avulla taas voi tyylittää sisäisesti yksittäistä elementtiä merkkijonona tai objektina. (Class and style, n.d)

Dynaamisen tyylittelyn avulla voidaan muokata komponentin ulkoasua tilan tai muiden tekijöiden perusteella. Tämä voidaan toteuttaa esimerkiksi käyttämällä `props`-arvoja tai `classList`-atribuuttia, mikä helpottaa useiden luokkien käsittelyä. `ClassList` on tehokkaampi tapa hallita luokkia, sillä se päivittää vain tarvittavat luokat. (Class and style, n.d)

4.3 Reaktiivisuus

Reaktiivisuudella tarkoitetaan järjestelmän kykyä reagoida automaattisesti datan tai tilan muutoksiin ilman manuaalista väliintuloa. SolidJS on suunniteltu reaktiiviseksi, jonka avulla varmistetaan, että sovellus pysyy jatkuvasti synkronoituna sen taustalla olevan datan kanssa. Reaktiivisuus varmistaa käyttöliittymän ja sovellustilan ajantasaisuuden, mikä vähentää tarpeen manuaalisille päivityksille. Automaattiset, reaaliaikaiset päivitykset parantavat käyttäjäkokemusta, tehden siitä sujuvamman ja vuorovaikutteisemman. (Intro to reactivity, n.d)

SolidJS:ä reaktiivisuuden ylläpito perustuu kahteen keskeiseen periaatteeseen: signaaleihin ja tilaajiin. Signaalit mahdollistavat sekä datan säilyttämisen että siihen liittyvien muutosten seuraamisen sovelluksessa. Signaalissa käytetään kahta funktiota: getter ja setter. (Signals, n.d)

```
const [count, setCount] = createSignal(0);
//     ^ getter  ^ setter
```

Kuvio 4: Signaalin luominen (Signals, n.d)

Yllä oleva koodiesimerkki (ks. Kuvio 4) esittelee signaalin luonnin SolidJS:ää. Signaali luodaan käyttämällä `createSignal`-funktiota, ja se palauttaa kaksi arvoa: getter-funktion `count()` ja setter-funktion `setCount()`. Getter-funktiota käytetään lukemaan signaalin nykyinen arvo. Tämä funktio kutsutaan aina, kun halutaan päästä käsiksi signaalin tallennettuun tietoon komponentin sisällä. Setter-funktio puolestaan vastaa signaalin arvon päivittämisestä. Kun signaalin arvo päivittyy, se laukaisee reaktiivisen päivityksen niihin osiin sovellusta, jotka ovat riippuvaisia tästä signaalista. (Signals, n.d)

Signaalien lisäksi, Solidissa on toinen perustekijä, tilaajat. Ne ovat vastuussa muutoksien seuraamisesta signaaleiden arvoissa sekä järjestelmän päivittämisestä. Ne ovat automaattisia vastaajia pitämään järjestelmä ajan tasalla viimeisimpiin datan muutoksiin. Näillä on kaksi perustoiminnallisuutta, havainnot ja reaktiot. Tilaajat ovat valmiina havainnoimaan signaalissa tapahtuvat muutokset seuraamalla niitä jatkuvasti. Kun signaalissa tapahtuu muutos, se laukaisee tilaajan reagoimaan tähän muutokseen. Tämä voi tarkoittaa esimerkiksi käyttöliittymän päivitystä tai ulkoisten toimien kutsumista. (Subscribers, n.d)

Yleisin tilaaja on `createEffect`, mikä reagoi siihen sidottujen signaalien arvojen muutoksiin. Tämä on hyödyllistä, kun halutaan päivittää käyttöliittymää suoraan tai kutsua muita toimintoja muutoksien yhteydessä. Toinen tärkeä tilaaja solidJS:ssä on `createMemo`. Sen avulla voidaan laskea ja tallentaa siihen sidottujen signaalien arvot aina sen päivittyessä. Lisäksi solidJS tarjoaa myös `createResource`-tilaajan, jota käytetään yleisesti asynkronisten tietojen hakemisessa kuten API-kutsuissa. Tämän avulla käyttöliittymä pysyy ajantasaisena datan saapuessa. (Basic Reactivity, n.d)

```
function Counter() {
  const [count, setCount] = createSignal(0);
```

```
  createEffect(() => {
    console.log('The count is now', count());
  });

  return <button onClick={() => setCount(count() + 1)}>Click Me</button>;
}
```

Kuvio 5: CreateEffect-funktio

Yllä olevassa esimerkissä (ks. Kuvio 5) `createEffect`-funktio toimii tilaajana, joka seuraa `count`-muuttujan arvoa. Sen sisälle on määritelty lauseke, joka tulostaa konsoliin `count`-muuttujan nykyisen arvon. Tämä tulostus tapahtuu automaattisesti aina, kun `count`-muuttujan arvo päivittyy tässä tapauksessa silloin, kun käyttäjä painaa "Click Me" -painiketta. Näin varmistetaan, että sovellus reagoi välittömästi käyttäjän toimintaan ja pitää tilan synkronoituna konsolin kanssa.

SolidJS reaktiivinen järjestelmä on suunniteltu vastaamaan datan muutoksiin. Vastaukset näissä voivat tapahtua välittömästi tai viiveellä, riippuen tilanteesta. Näitä kutsutaan synkroniseksi- sekä asynkroniseksi reaktiivisuudeksi. Synkroninen reaktiivisuus tarkoittaa, että järjestelmä reagoi muutoksiin välittömästi ja ennustettavasti. Kun signaali muuttuu, kaikki siihen liittyvät tilaajat päivitetään heti tietyssä järjestyksessä. Tämä varmistaa, että tilaajat, jotka ovat riippuvaisia toisista signaaleista, saavat oikeat arvot oikeaan aikaan. Asynkroninen reaktiivisuus viittaa järjestelmään, joka reagoi muutoksiin viiveellä tai epälineaarisesti. Tilaajat päivitetään vasta tiettyjen tehtävien tai tapahtumien päätyttyä, mikä on hyödyllistä tilanteissa, joissa tilaajien tulee odottaa useiden signaalien päivittymistä ennen reaktiota, jotta vältetään datan epäjohdonmukaisuudet. (Synchronous vs. asynchronous, n.d)

SolidJS on erityisen tehokas renderöinnissä, koska se hallinnoi DOM-puuta optimoidusti. Toisin kuin joissakin muissa frontend-kirjastoissa, joissa koko komponentti voidaan renderöidä uudelleen tilamuutoksen myötä, SolidJS seuraa tarkasti yksittäisten signaalien muutoksia. Kun signaalin arvo päivittyy, vain ne DOM-elementit, jotka ovat riippuvaisia kyseisestä signaalista, renderöidään uudelleen. Tämä lähestymistapa minimoi turhat päivitykset ja parantaa suorituskykyä, erityisesti monimutkaisissa käyttöliittymissä, joissa on paljon vuorovaikutteisia elementtejä tai suuria tietomääriä. Tämä tekee SolidJS:stä erittäin suorituskykyisen, sillä tarpeettomat uudelleenrenderöinnit vältetään ja selainresurssit käytetään tehokkaasti. (Fine-grained reactivity, n.d)

4.4 Tilanhallinta

Tilanhallintaan kuuluu kolme peruselementtiä, mitkä ovat tila, näkymä ja toiminnot. Tila tarkoittaa dataa, jota käytetään määrittämään, mitä sisältöä käyttäjälle näytetään. Se on sovelluksen "totuus", jonka perusteella käyttäjä näkee erilaisia elementtejä. Näkymä havainnollistaa käyttäjälle visuaalisesti tilaa. Se on se osa käyttöliittymää, jossa tila esitetään, kuten teksti tai kuvat, jotka päivittyvät tilan muuttuessa. Toiminnot viittaavat kaikkiin tapahtumiin, jotka muuttavat tilaa. Esimerkiksi napin klikkaaminen voi lisätä tai vähentää laskurin arvoa, mikä vaikuttaa siihen, mitä käyttäjä näkee. (State managment, n.d)

Esimerkiksi laskurikomponentin (ks. Kuvio 6) tila luodaan `createSignal`-funktion avulla, jonka alkuarvoksi asetetaan 0. Nykyinen tila näytetään kutsumalla `count`-funktiota. Komponentissa tilaa muutetaan `increment`-funktion avulla, joka kasvattaa arvoa yhdellä napin painalluksessa.

```
import { createSignal } from 'solid-js';

function Counter() {
  const [count, setCount] = createSignal(0);

  const increment = () => {
    setCount((prev) => prev + 1);
  };

  return (
    <>
      <div>Current count: {count()}</div>
      <button onClick={increment}>Increment</button>
    </>
  );
}
```

Kuvio 6. Perustilan hallinta (State managment, n.d)

Johdetulla tilalla tarkoitetaan uusien arvojen laskemista olemassa olevien tilan arvojen perusteella. Se on hyödyllistä, kun halutaan näyttää tilan arvon muunnos ilman alkuperäisen tilan muuttamista (Derived state, n.d.). Esimerkiksi voimme laskea kaksinkertaisen arvon käyttäen:

```
const doubleCount = () => count() * 2;
```

Kuvio 7. Johdettu tila (Derived state, n.d)

Tämä voi olla kuitenkin tehotonta suuremmissa sovelluksissa, joissa `doubleCount`-tilaa käytetään useita kertoja, sillä se arvioidaan uudelleen jokaisella kutsulla. Näissä tapauksissa on suositeltavaa käyttää `createMemo`-funktiota, minkä etuna on, että laskenta suoritetaan vain kerran ja arvoa voidaan käyttää useita kertoja ilman uudelleenarviointia. (Derived state, n.d)

```
const doubleCountMemo = createMemo(() => count() * 2);
```

Kuvio 8. createMemo-funktion käyttö (Derived state, n.d)

Toinen tärkeä tilanhallinnan osa-alue ovat storet. Kun signaaleilla hallitaan yksittäisiä tilakappaleita, storet tarjoavat keskitetyn paikan useiden tilakappaleiden hallintaan. Storet muodostavat joukon reaktiivisia signaaleja, joista jokainen vastaa tietystä osa-alueesta, mikä helpottaa monimutkaisen tilan hallintaa ja vähentää koodin toisteisuutta. (Stores, n.d)

```
const [users, setUser] = createStore({
   user: [
     {
       id: 0,
       name: 'John Doe',
       age: 23,
     },
     {
       id: 1,
       name: 'Sarah Poe',
       age: 34,
     },
   ],
 });
```

Kuvio 9: Storen luominen

Kuviossa 10 luodaan store käyttämällä `createStore`-funktiota. Se toimii samaan tapaan kuin signaali, eli se saa getterin ja setterin, joiden avulla voidaan näyttää sekä päivittää storen tilaa (ks. Kuvio 5). Tässä tapauksessa storessa on taulukko nimeltä `user`, joka sisältää kaksi objektia, missä on käyttäjien tietoja.

```
import { users } from './userStore';

function App() {
```

```
  return (
    <>
      <h1>User List</h1>
      <ul>
        {users.user.map((user) => (
          <li key={user.id}>
            {user.name}, {user.age} years old
          </li>
        ))}
      </ul>
    </>
  );
}

export default App;
```

Kuvio 10. Storen arvojen käsittely

Storen arvoihin päästään käsiksi importoimalla store haluttuun komponenttiin, kuten tässä tapauksessa `userStore` tiedostosta. Komponentissa `users`-store sisältää `user`-taulukon, joka pitää sisällään käyttäjien tiedot. Näitä tietoja voidaan käsitellä esimerkiksi silmukan avulla, kuten kuviossa 11 `map()`-metodilla luodaan käyttäjälistaus, jossa kullekin käyttäjälle muodostetaan listaelementti. Jokaiselle listan riville annetaan myös yksilöllinen avain käyttäjän ID perusteella. Tällöin jokaisen käyttäjän nimi ja ikä saadaan renderöityä dynaamisesti käyttöliittymässä.

4.5 Reititys

Solidin ekosysteemistä löytyy universaali reititin, Solid-Router, joka toimii sekä asiakas- että palvelinpuolella. Reititin mahdollistaa näkymän muuttamisen käyttäjän URL-osoitteen perusteella, jolloin sovellus voi simuloida perinteisen MPA-sivuston toimintaa SPA-sovelluksessa. Solid-Routerissa määritellään komponentteja, joita kutsutaan `Routeiksi`. Nämä komponentit reagoivat URL-osoitteen muutoksiin ja reititin hoitaa niiden vaihtamisen automaattisesti. Tämän ansiosta Solidilla voidaan rakentaa moderniin internettiin sopivia SPA-sovelluksia. Solid-Router yhdistää React Routerin ja Ember Routerin lähestymistapoja, mikä tekee siitä joustavan ja monipuolisen ratkaisun reititystarpeisiin. (Overview, n.d)

```
import { render } from 'solid-js/web';
import { Router, Route, A } from '@solidjs/router';

import Home from './pages/Home';
```

```
import Users from './pages/Users';
import NotFound from './pages/NotFound';

const App = (props) => (
  <>
    <nav>
      <A href="/">Home</A>
      <A href="/users">Users</A>
    </nav>
    <h1>Site Title</h1>
    {props.children}
  </>
);

render(
  () => (
    <Router root={App}>
      <Route path="/" component={Home} />
      <Route path="/users" component={Users} />
      <Route path="*paramName" component={NotFound} />
    </Router>
  ),
  document.getElementById('root')
);
```

Kuvio 11: Reititys

Solid-routerin dynaamiset reitit mahdollistavat joustavan reitityksen käyttämällä polun osia parametreina. Dynaamisia reittejä voi luoda käyttämällä kaksoispistettä (`:`) reitityksen polussa, jolloin osa polusta, kuten käyttäjän ID, voi vaihdella. Tämä mahdollistaa komponenttien renderöinnin eri parametreilla URL-osoitteen perusteella. Reittiparametreihin pääsee käsiksi `useParams`-funktion avulla, jonka jälkeen niitä voidaan hyödyntää esimerkiksi tiedon hakemiseen tai käyttöliittymän päivittämiseen. (Path parameters, n.d)

```
<Route path="/users/:id" component={User} />;
```

Kuvio 12: Dynaamisen reitin määrittely (Path parameters, n.d)

Pesiytetyt reitit tarkoittaa sitä, että reitit voivat olla toistensa sisällä, jolloin alireitit voidaan määrittää osaksi laajempaa reittiä. Tämän avulla voidaan hallita monimutkaisia reitityksiä ja mahdollistaa hierarkkinen järjestely. Pesiytetyissä reiteissä ylempi reitti toimii "kehyksenä", johon alireitit

sijoittuvat. Tämä mahdollistaa sen, että pääreitin komponentti voi sisällyttää alireittien sisällön, mikä lisää joustavuutta sovelluksen rakenteessa. (Nesting Routes, n.d)

```
<Route path="/users" component={Users} />
<Route path="/users/:id" component={User} />

// or

<Route path="/users">
  <Route path="/" component={Users} />
  <Route path="/:id" component={User} />
</Route>
```

Kuvio 13: Nested routing (Limitations, n.d)

5 Vertailu

Tässä luvussa tarkastellaan SolidJS:n, Reactin ja Svelten välisiä eroja ja yhtäläisyyksiä keskeisten teknisten ominaisuuksien pohjalta. Näiden kolmen JavaScript -kirjaston vertailu tarjoaa syvemmän ymmärryksen siitä, miten ne eroavat toisistaan ja millaisia yhtäläisyyksiä niillä on eri osa-alueilla.

5.1 Vertailun toteutus

Vertailu suoritetaan ennalta laaditun luvussa 2.5 esitetyn vertailukehyksen mukaisesti, jotta analyysi olisi mahdollisimman tasapainoinen ja kattava. Tavoitteena on käydä läpi jokainen vertailukehyksen osa-alue ja havainnollistaa, kehyksien keskeiset toimintatavat ja ratkaisut.

On tärkeää huomioida, että tässä työssä suoritetut vertailut ovat suuntaa antavia. Ne tarjoavat kattavan yleiskuvan Solidin, Reactin ja Svelten toimintatavoista, mutta eivät käsittele kaikkia mahdollisia teknisiä ominaisuuksia. Ohjelmistokehityksessä on usein useita eri toteutustapoja, joten on tärkeää huomioida, että tässä tutkimuksessa käytetyt ratkaisut voidaan toteuttaa myös vaihtoehtoisilla menetelmillä. Vertailukehys on suunniteltu antamaan selkeä ja jäsennelty tapa tarkastella kunkin kehyksen toiminnallisuuksia.

Kunkin osa-alueen kohdalla esitellään lyhyesti, miten kyseinen osa-alue toteutetaan SolidJS:ssä, Reactissa ja Sveltessä. Tämä auttaa lukijaa ymmärtämään näiden kehyksien väliset eroavaisuudet

ja yhtäläisyydet. Vertailu tarjoaa yleiskuvan siitä, miten kunkin kehyksen lähestymistavat eroavat toisistaan, mutta syvällisempien tarkastelujen ja erityisten käyttötapausten analysointi saattaa vaatia jatkotutkimusta tai tarkempaa perehtymistä.

React on Facebookin kehittäjien kehittämä JavaScript-kirjasto, minkä avulla voidaan rakentaa dynaamisia käyttöliittymiä. Se hyödyntää deklaratiivisuutta sekä komponenttipohjaista arkkitehtuuria, minkä vuoksi käyttöliittymä voidaan jakaa pieniin uudelleenkäytettäviin komponentteihin. Tällä hetkellä React on sekä suosituin että tunnetuin JavaScript-kirjasto ja se on vakiinnuttanut asemansa kehittäjäyhteisöissä ja yrityksissä.

Svelte on moderni käyttöliittymäkirjasto, joka eroaa muista perinteisemmistä kehyksistä ja kirjastoista, sillä se käyttää kääntäjää. Sen avulla, se kääntää koodin JavaScriptiksi jo rakennusvaiheessa, mikä vähentää selaimessa tarvittavaa laskentaa. Svelte on erittäin suosittu sen nopeuden ja pienen koodimäärän takia, mikä helpottaa kehitystyötä. Svelten suosio on ollut kasvussa jo useamman vuoden ja se alkaakin pikkuhiljaa saavuttamaan paikkaansa käyttöliittymäkehityksen alalla.

5.2 Syntaksi

Luodaan yksinkertainen lomakekomponentti, jossa käyttäjä voi syöttää nimensä ja lomakkeen lähettämisen jälkeen selain näyttää ilmoituksen, jossa nimi näkyy. Lomake on hyvä vertailukomponentti, sillä se on olennainen osa monia verkkosovelluksia, ja sen toteutus paljastaa kirjastojen väliset eroavuudet konkreettisesti ja selkeästi. Käyttäjän syötteen reaaliaikainen näyttäminen ja tilan päivittäminen käyttöliittymässä ovat keskeisiä toimintoja, joiden toteutus vaihtelee kirjaston mukaan.

SolidJS

Komponentin rakentaminen aloitetaan luomalla signaali `name`-muuttujalle käyttämällä `createSignal`-funktiota. Tämä signaali on kaksisuuntaisessa tiedonsidonnassa tekstikentän ja tilan välillä, jolloin käyttäjän syöte päivittyy reaaliajassa `onInput`-tapahtuman avulla. Kun lomake lähetetään, suoritetaan `handleSubmit`-funktio, joka estää lomakkeen oletuslähetyksen (preventDefault) ja näyttää selaimessa ilmoituksen, jossa näkyy käyttäjän syöttämä nimi. Tämä ta-

pahtuu käyttämällä name-funktiota, joka palauttaa signaalin sen hetkisen arvon. JSX-syntaksin mukaisesti <p>-elementissä käytetään aaltosulkeita {name()}-arvon lukemiseen ja näyttämiseen, ja SolidJS päivittää näkymää automaattisesti aina, kun signaalin arvo muuttuu. (ks. Kuvio 11)

```
import { createSignal } from 'solid-js';

export default function FormComponent() {
  const [name, setName] = createSignal('');

  const handleSubmit = (e) => {
    e.preventDefault();
    alert(`Hello, ${name()}!`);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label for="name">Name: </label>
      <input
        id="name"
        type="text"
        value={name()}
        onInput={(e) => setName(e.target.value)}
      />
      <button type="submit">Submit</button>
      <p>Your name is: {name()}</p>
    </form>
  );
}
```

Kuvio 14. SolidJS syntaksi

React

Kuviosta 12 nähdään, että React käyttää myös JSX-syntaksia ja on toiminnaltaan hyvin samankaltainen kuin SolidJS. SolidJS:än signaalien sijaan React käyttää useState-hookkia, jonka avulla voidaan palauttaa sekä päivittää tilan arvo. SolidJS:stä poiketen, React ei tue suoraan kaksisuuntaista tiedonsidontaa. Sen sijaan, aina name-muuttujan arvon päivittyessä, se renderöin komponentin uudelleen, minkä avulla syöte päivittyy käyttäjälle reaaliajassa. Tapahtumankäsittelyssä React käyttää onChange-tapahtumaa syötteen päivitykselle, mikä tarkoittaa, että syötteen muutos vaatii erillisen päivityksen tilan arvoon. Reactissa ei käytetä tilaan viitattaessa name-funktiota, vaan suoraan name-muuttujaa, mikä tarkoittaa, että tilan arvoon pääsee käsiksi ilman funktiokutsua.

```jsx
import React, { useState } from 'react';

export default function FormComponent() {
  const [name, setName] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault();
    alert(`Hello, ${name}!`);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label for="name">Name: </label>
      <input
        id="name"
        type="text"
        value={name}
        onChange={(e) => setName(e.target.value)}
      />
      <button type="submit">Submit</button>
      <p>Your name is: {name}</p>
    </form>
  );
}
```

Kuvio 15. React syntaksi

Svelte

Jo koodia silmäillessä huomaamme, että Svelte eroaa SolidJS syntaksista merkittävästi. Suurin ero on varmasti JSX-syntaksin puuttuminen, Svelte erottelee HTML-koodin JavaScriptin logiikasta. Sveltessä ei tarvita erillisiä signaaleja muuttujien arvojen näyttämiseen tai päivittämiseen, sillä tilanhallinta tapahtuu suoraan muuttujien avulla `let name = ' ';`. Muuttujan sitominen syötekenttään tapahtuu `bind`-metodilla, jolloin muutokset päivittyvät automaattisesti. (ks. Kuvio 16.)

```svelte
<script>
  let name = '';

  function handleSubmit(event) {
    event.preventDefault();
    alert(`Hello, ${name}!`);
  }
</script>
```

```
<main>
  <form on:submit={handleSubmit}>
    <label for="name">Name: </label>
    <input id="name" type="text" bind:value={name} />
    <button type="submit">Submit</button>
    <p>Your name is: {name}</p>
  </form>
</main>

<style>
</style>
```

Kuvio 16. Svelte syntaksi

Vertailun aikana havaittiin, että vaikka SolidJS:n syntaksi ei ole yhtä yksinkertainen kuin Sveltellä, niin JSX-syntaksi on kuitenkin Reactin laajan käytön ansiosta monille kehittäjille entuudestaan tuttu, minkä vuoksi SolidJS:n omaksuminen voi olla helpompaa näille henkilöille. SolidJS tarjoaa myös kattavan TypeScript-tuen, mikä on tärkeä etu, sillä TypeScript on yleisesti ottaen turvallisempi vaihtoehto JavaScriptin sijaan ja se onkin käytössä monissa yrityksissä erityisesti sen luotettavuuden ja vakauden vuoksi. Tämä on lupaava asia SolidJS:n tulevaisuuden kannalta, sillä se voi madaltaa yritysten kynnystä ottaa SolidJS käyttöön.

5.3 Komponentit ja tapahtumien hallinta

Kuviossa 17 esitetään `TimerButton`-funktio, joka luo yksinkertaisen ajastimen painikkeen, jonka käyttäjä voi käynnistää ja pysäyttää. Ajastimen toiminta perustuu kahteen signaaliin, mistä yksi hallinnoi aikaa ja toinen ilmaisee, onko ajastin käynnissä vai pysäytettynä.

Ajastimen tilaa hallitaan luomalla tapahtumankäsittelijä, joka kytkee `running`-signaalin arvon toden ja epätoden välillä. Tämän lisäksi ajastin käynnistetään automaattisesti komponentin latautuessa ensimmäisen kerran `onMount`-funktion avulla. Tilan muutosten hallintaan on käytetty `createEffect`-funktiota, joka tarkkailee `running`-signaalin arvoa. Kun ajastin on käynnissä, `setInterval`-funktio kasvattaa `time`-arvoa sekunnin välein. Jos ajastin pysäytetään, `clearInterval`-funktio keskeyttää ajastimen päivitykset. Tilan siivouksen varmistaa `onCleanup`-funktio, joka myös pysäyttää ajastimen, jos komponentti poistetaan käytöstä.

Ajastimen ulkoasu päivittyy dynaamisesti `running`-tilan perusteella. Tyyliasetus on toteutettu asettamalla button-elementille CSS-luokka, joka määräytyy `running`-tilan mukaan (`timer-btn running` tai `timer-btn stopped`). Lisäksi button-elementtiin on sidottu `onClick`-tapahtuma, joka käynnistää ja pysäyttää ajastimen. Tyylittelyyn käytetyt asetukset välitetään `props`-objektin avulla, ja tyylit ladataan erillisestä CSS-tiedostosta.

```
import { createSignal, createEffect, onCleanup, onMount } from 'solid-js';
import './TimerButton.css';

function TimerButton(props) {
  const [time, setTime] = createSignal(0);
  const [running, setRunning] = createSignal(false);

  const toggleTimer = () => {
    setRunning(!running());
  };

  onMount(() => {
    setRunning(true);
  });

  createEffect(() => {
    let timer;
    if (running()) {
      timer = setInterval(() => setTime(time() + 1), 1000);
    } else {
      clearInterval(timer);
    }
    onCleanup(() => clearInterval(timer));
  });

  return (
    <button
      class={running() ? 'timer-btn running' : 'timer-btn stopped'}
      onClick={toggleTimer}
      style={{ background: props.color }}
    >
      {running() ? `Running: ${time()}s` : `Stopped: ${time()}s`}{' '}
    </button>
  );
}

export default TimerButton;
```

Kuvio 17: SolidJS ajastin-komponentti

React

Ajastimen tilaa hallitaan Reactissa luomalla tapahtumankäsittelijä, joka vaihtaa `running`-tilan arvon toden ja epätoden välillä `useState`-hookin avulla. Ajastin käynnistetään automaattisesti komponentin ensimmäisellä latauskerralla `useEffect`-hookin avulla, joka vastaa SolidJS:n onMount-funktiota. Tilan muutosten seuraamiseen käytetään toista useEffect-hookia, joka tarkkailee `running`-tilan arvoa riippuvuuslistan avulla. Kun ajastin on käynnissä, `setInterval`-funktio kasvattaa `time`-tilaa sekunnin välein. Jos ajastin pysäytetään, `clearInterval`-funktio keskeyttää ajastimen päivitykset. Tilan siivous varmistetaan `useEffect`-hookin palauttamalla `cleanup`-funktiolla, joka pysäyttää ajastimen aina, kun komponentti poistetaan DOM:sta tai `running`-tila muuttuu. (ks. Kuvio 18).

```
import { useState, useEffect } from 'react';
import './TimerButton.css';

function TimerButton({ color }) {
  const [time, setTime] = useState(0);
  const [running, setRunning] = useState(false);

  const toggleTimer = () => {
    setRunning(!running);
  };

  useEffect(() => {
    setRunning(true);
  }, []);

  useEffect(() => {
    let timer;
    if (running) {
      timer = setInterval(() => setTime((prevTime) => prevTime + 1), 1000);
    } else {
      clearInterval(timer);
    }
    return () => clearInterval(timer);
  }, [running]);

  return (
    <button
      className={running ? 'timer-btn running' : 'timer-btn stopped'}
      onClick={toggleTimer}
      style={{ background: color }}
    >
```

```
      {running ? `Running: ${time}s` : `Stopped: ${time}s`}
    </button>
  );
}

export default TimerButton;
```

Kuvio 18: React ajastin-komponentti

Svelte

Sveltessä voidaan käyttää suoria muuttujia ajan ja ajastimen tilojen seuraamiseen, ilman tarvetta hookeille tai signaaleille. Tämän myötä, myös tilan käsittely onnistuu ilman erillisiä funktiokutsuja. Tässä esimerkissä ajastimen tilan seuraamiseen on käytetty `writable`-store, mutta se ei ole välttämätöntä, sillä tilan hallinta onnistuu myös suoraan muuttujien avulla. Kuten SolidJS:ssä, myös Sveltessä on myös käytössä `onMount`-funktio, jonka avulla ajastin voidaan käynnistää, heti komponentin latautuessa sivulle. Siivous tapahtuu `return`-lausekkeella, joka suoritetaan heti, kun komponetti on poistettu DOM:sta. Svelte tarjoaa myös `onDestroy`-funktion, jonka avulla, komponentti voidaan tuhota jo ennen sen poistamista DOM:sta. (ks. Kuvio 19).

```
<script>
  import { onMount } from 'svelte';
  import { writable } from 'svelte/store';

  let time = 0;
  let running = false;

  const timerStore = writable(0);

  const toggleTimer = () => {
    running = !running;
  };

  onMount(() => {
    running = true;
    const interval = setInterval(() => {
      if (running) {
        time += 1;
        timerStore.set(time);
      }
    }, 1000);

    return () => clearInterval(interval);
```

```
  });
</script>

<main>
  <button
    class={running ? 'timer-btn running' : 'timer-btn stopped'}
    on:click={toggleTimer}
  >
    {running ? `Running: ${time}s` : `Stopped: ${time}s`}
  </button>
</main>

<style>
  .timer-btn {
    padding: 10px 20px;
    font-size: 16px;
    border: none;
    border-radius: 5px;
    color: white;
    cursor: pointer;
  }

  .running {
    background-color: green;
  }

  .stopped {
    background-color: red;
  }
</style>
```

Kuvio 19: Svelte ajastin-komponentti

5.4 Reaktiivisuus

SolidJS

Alla olevassa esimerkissä havainnollistetaan reaktiivisuuden toimintaa SolidJS-kirjastossa. Reaktiivisuus ilmenee siten, että aina kun käyttäjä painaa "+1" -painiketta, se muuttaa `count`-muuttujan arvoa signaalin avulla, samalla automaattisesti päivittäen käyttöliittymän reaktiiviset osat. `createSignal`-funktio luo reaktiivisen signaalin `count`, jonka arvo päivittyy `setCount`-funktiolla. Tämän lisäksi `createMemo`-funktiota käytetään luomaan muuttuja `doubledCount`, joka riip-

puu suoraan `count`-signaalista. Kun `count`-arvo muuttuu, myös `doubledCount` päivittyy automaattisesti ilman erillistä päivityskomentoa. Tämä eliminoi tarpeen manuaalisille päivityksille ja tekee sovelluksesta tehokkaan ja reaktiivisen. (ks. Kuvio 20)

```
import { createSignal, createMemo } from 'solid-js';

function Counter() {
  const [count, setCount] = createSignal(0);

  const doubledCount = createMemo(() => count() * 2);

  return (
    <div>
      <div>Count: {count()}</div>
      <br />
      <div>Doubled Count: {doubledCount()}</div>
      <br />
      <button onClick={() => setCount(count() + 1)}>+1</button>
    </div>
  );
}

export default Counter;
```

Kuvio 20: SolidJS reaktiivisuus

React

Reactin toiminnallisuus muistuttaa pitkälti Solid.js reaktiivista mallia, mutta sillä on omat erikoispiirteensä. SolidJS:n signaalien sijasta Reactissa käytetään `useState`-hookia, joka palvelee samaa tarkoitusta: tilan hallintaa ja komponentin uudelleenrenderöintiä aina, kun tila muuttuu. SolidJS `createSignal` ja Reactin `useState` toimivat siten samankaltaisesti. Johdettujen arvojen laskenta tapahtuu Reactissa samalla tavalla kuin SolidJS:ssä, mutta siihen käytetään `useMemo`-hookia. Esimerkiksi, jotta laskettaisiin kaksinkertainen arvo (kuten SolidJS:n `createMemo`), Reactissa voitaisiin hyödyntää `useMemo`-hookia seuraavasti:

```
const doubledCount = useMemo(() => count * 2, [count]);
```

Kuvio 21: React useMemo-hook

Svelte

Svelte eroaa huomattavasti SolidJS:stä ja Reactista reaktiivisuuden hallinnassa. Sveltessä muuttujat, kuten `count` (ks. kuvio 22), ovat reaktiivisia ilman erillisiä signaaleja. Tämä tarkoittaa, että aina kun `count` muuttuu, käyttöliittymä päivittyy automaattisesti. Johdettu arvo, kuten `doubledCount`, lasketaan reaktiivisen lausekkeen avulla, joka määritellään dollari-symbolilla `$:`. Tämä lauseke kertoo, että `doubledCount` on aina `count`-muuttujan arvo kerrottuna kahdella, ja se päivittyy automaattisesti aina, kun `count`-muuttujan arvo muuttuu. Svelte hoitaa reaktiiviset päivitykset ilman erillisiä signaaleita tai hookkeja.

```
<script>
  let count = 0;

  $: doubledCount = count * 2;

  function increment() {
    count += 1;
  }
</script>

<div>
  <div>Count: {count}</div>
  <br />
  <div>Doubled Count: {doubledCount}</div>
  <br />
  <button on:click={increment}>+1</button>
</div>
```

Kuvio 22: Svelte reaktiivisuus

SolidJS:n hallitun reaktiivisuuden avulla, se päivittää vain käyttöliittymän tietyt osat, mikä tekee siitä todella suorituskykyisen erityisesti tilanteissa, missä käyttöliittymä päivittyy usein. Tämä jättää kehittäjille täyden kontrollin siitä, mitkä komponentit päivitetään milloinkin, jonka avulla minimoidaan tarpeettomat DOM:n päivitykset ja säästetään resursseja. Sen sijaan virtuaalisen DOM:iin perustuva lähestymistapa Reactissa voi käydä raskaaksi komponenttipuulle, mistä voi seurata esimerkiksi suorituskykyongelmia. Vaikka SolidJS:n reaktiivinen kehitysympäristö tarjoaa paljon merkittäviä etuja, niin se vaatii kehittäjältä tuntemusta sekä toimenpiteitä, kuten tilaajien asettamista seuraamaan muuttujia. Tästä syystä se ei kuitenkaan yllä Svelten tasolle poikkeuksellisen yksinkertaisessa reaktiivisuuden hallinnassaan. Huomiona, Svelten uusimassa versiossa 5, reaktiivisuus on

toteutettu signaalipohjaisesti riimujen (runes) avulla, joka muistuttaa pitkälti SolidJS:n reaktiivisuutta.

5.5 Tilanhallinta

Luodaan skenaario, joka havainnollistaa tilanhallinnan toimintaa sovelluksessa. Usein sovelluksen käynnistyessä on tarpeellista hakea tietoa palvelimelta tai muista lähteistä. Tällaiset tiedot kannattaa tallentaa esimerkiksi storeen, jolloin niitä ei tarvitse hakea uudelleen aina, kun käyttäjä navigoi sovelluksessa, vaan tiedot ovat aina saatavilla storesta. Tämä parantaa suorituskykyä ja vähentää turhia pyyntöjä palvelimelle. Tässä luvussa tiedot haetaan ilmaisesta API:sta nimeltä JSONPlaceholder, joka tarjoaa niin kutsuttua ”feikki dataa”, jota voidaan käyttää sovelluksen testaamiseen sekä simuloimiseen.

SolidJS

Kuviossa 23 esitetään `createUserStore`-funktio, joka luo `userStore`-storen sekä asynkronisen `loadUsers`-funktion. `UserStore` alustetaan tyhjällä taulukolla, johon myöhemmin tallennetaan APIhaetut käyttäjätiedot. `LoadUsers`-funktiossa käyttäjät haetaan `fetchUsers`-funktion avulla, joka suorittaa hakupyynnön palvelimelle. Saadut käyttäjät tallennetaan `users`-muuttujaan. Tämän jälkeen `users`-muuttujan arvot lisätään `userStoreen` käyttämällä `setUserStore`-funktiota, mikä mahdollistaa storeen päivityksen.

```
import { onMount } from 'solid-js';
import { createUserStore } from './userStore';
import { fetchUsers } from './api';

export default function App() {
  const { userStore, setUserStore } = createUserStore();

  const loadUsers = async () => {
    try {
      const users = await fetchUsers();
      setUserStore('users', users);
    } catch (error) {
      console.error('Failed to load users:', error);
    }
  };
```

```
  onMount(() => {
    loadUsers();
  });

  return (
    <div>
      <h1>User List</h1>
      <div className="user-list">
        {userStore.users.map((user) => (
          <div key={user.id} className="user-card">
            <h2>{user.name}</h2>
            <p>
              <strong>Username:</strong> {user.username}
            </p>
            <p>
              <strong>Email:</strong> {user.email}
            </p>
          </div>
        ))}
      </div>
    </div>
  );
}
```

Kuvio 23: SolidJS tilanhallinta

React

Toisinkuin SolidJS:ssä, Reactissa ei ole sisäänrakennettua tilanhallintajärjestelmää, mutta siihen on saatavilla useita lisäosia. Kuviossa 24 on käytetty React Reduxia, joka on yksi suosituimmista tilanhallintaratkaisuista. Reduxissa tilaa voidaan päivittää sekä näyttää `useDispatch`- ja `useSelector`-hookkien avulla. Reactissa ei myöskään ole sisäänrakennettua elinkaarifunktiota, kuten SolidJS:n `onMount`-funktio, mutta vastaava toiminnallisuus voidaan saavuttaa käyttämällä `useEffect`-hookia. Jättämällä `useEffect`-funktion riippuvuuslistan tyhjäksi merkillä `[]`, saavutetaan sama toiminnallisuus kuin `onMount`-funktiossa. Näin koodi suoritetaan vain kerran, kun komponentti ladataan ensimmäisen kerran. Myös Reactin `useState`-hookin käyttö olisi ollut mahdollista tässä yhteydessä korvaamaan Reduxin toimintaa, sillä se mahdollistaa useiden muuttujien varastoimisen ja hallinnan komponenttien sisällä. (ks. Kuvio 24)

```
import { useEffect } from 'react';
import { useDispatch, useSelector } from 'react-redux';
```

```
import { setUsers } from './store/userSlice';
import { fetchUsers } from './api';

export default function App() {
  const dispatch = useDispatch();
  const users = useSelector((state) => state.users.users);

  const loadUsers = async () => {
    try {
      const fetchedUsers = await fetchUsers();
      dispatch(setUsers(fetchedUsers));
    } catch (error) {
      console.error('Failed to load users:', error);
    }
  };

  useEffect(() => {
    loadUsers();
  }, []);
```

Kuvio 24: React tilanhallinta

Svelte

Svelte tarjoaa sisäänrakennetun tilanhallintaominaisuuden, jota voi käyttää store-toiminnallisuuden avulla. Tässä yhteydessä käytetään writable-storea, joka mahdollistaa tilan lukemisen ja päivittämisen sovelluksen eri osista. Tilaa voidaan päivittää kutsumalla userStore.set(users), jolloin koko tila päivittyy ilman erillisiä avain-arvopareja. Lisäksi Svelte tarjoaa $-merkinnän automaattiseen tilaamiseen, kun store muuttuu, käyttöliittymä päivittyy automaattisesti. Toisin kuin SolidJS:ssä, Sveltessä $-syntaksi tekee komponentista automaattisesti reaktiivisen ilman ylimääräistä tilausta, kun taas Solidissa tila täytyy asettaa seurantaan erillisellä tilaajalla, kuten createEffect-funktiolla. (ks. Kuvio 25)

```
<script>
  import { onMount } from 'svelte';
  import { userStore } from './userstore';
  import { fetchUsers } from './api';

  const loadUsers = async () => {
    try {
      const users = await fetchUsers();
      userStore.set(users);
    } catch (error) {
```

```
      console.error('Failed to load users:', error);
    }
  };

  onMount(() => {
    loadUsers();
  });
</script>

<main>
  <h1>User List</h1>
  <div class="user-list">
    {#each $userStore as user (user.id)}
      <div class="user-card">
        <h2>{user.name}</h2>
        <p>
          <strong>Username:</strong>
          {user.username}
        </p>
        <p>
          <strong>Email:</strong>
          {user.email}
        </p>
      </div>
    {/each}
  </div>
</main>
```

Kuvio 25: Svelte tilanhallinta

SolidJS tarjoaa sisäänrakennettuna storen, createStore-ominaisuuden avulla. Tämä voi yksinkertaistaa tilanhallinnan rakentamista sovelluksessa, kun ei ole tarvetta integroida kolmannen osapuolen kirjastoja. Vaikka Solid-store on kevyt ja hyvin suorituskykyinen, se voi kuitenkin käydä liian suppeaksi isommissa projekteissa, mihin esimerkiksi Reactin Redux on tarkoitettu. Pienempiin projekteihin Solid-store on erinomainen valinta, sillä se integroituu luonnollisesti SolidJS:n omaan signaalipohjaiseen reaktiivisuusmalliin.

5.6 Reititys

SolidJS

Alla on esimerkki (ks. Kuvio 26) reitityksen toteutuksesta yksinkertaisessa SolidJS SPA-sovelluksessa, joka hyödyntää @solidjs/router-kirjastoa. Sovelluksessa on useita reittejä, jotka määrittävät, mikä komponentti renderöidään kunkin URL-polun perusteella.

Pääkomponenttina toimii `Router`, joka käärii kaikki reitit sisäänsä ja käyttää sovelluksen juuritason asetteluna RootLayout-komponenttia, tarjoten näin yhtenäisen rakenteen kaikille sivuille. `RootLayout` sisältää `Navbar`-komponentin, joka näkyy kaikilla sivuilla sekä `props.children`-ominaisuuden, jonka kautta se renderöi jokaisen reitin, joka on määritelty `Route`-komponentin alle. (ks. Kuvio 26)

ProductsLayout-komponentti määrittelee sisäkkäisen asettelun kaikille tuotteisiin liittyville reiteille. Se sisältää navigaatiokomponentin, mikä näkyy kaikilla /products-reiteillä, mahdollistaen käyttäjälle sujuvan navigoinnin eri tuotteiden hallintasivujen välillä. Dynaamisien reittien avulla voidaan näyttää yksilöityjä tuotteita yhden reittiparametrin avulla. Esimerkiksi reitti /:id mahdollistaa tietyn tuotteen näyttämisen sen yksilöivän tunnisteen perusteella. Jos käyttäjä navigoi polulle "/products/123", ProductDisplay-komponentti näyttää tuotteen, jonka id on 123. (ks. Kuvio 26)

```
import { Router, Route } from '@solidjs/router';

// Pages
import Home from './pages/Home.jsx';
import About from './pages/About.jsx';
import Search from './pages/products/Search.jsx';
import ListProducts from './pages/products/ListProducts.jsx';
import AddProduct from './pages/products/AddProduct.jsx';
import ProductDisplay from './pages/products/ProductDisplay.jsx';
import NotFound from './pages/NotFound.jsx';

//Layouts
import ProductsLayout from './layouts/ProductsLayout.jsx';
import RootLayout from './layouts/RootLayout.jsx';

export default function App() {
  return (
        <Router root={RootLayout}>
```

```
          <Route path="/" component={Home} />
          <Route path="/about" component={About} />
          <Route path="/products" component={ProductsLayout}>
            <Route path="/list" component={ListProducts} />
            <Route path="/search" component={Search} />
            <Route path="/add" component={AddProduct} />
            <Route path="/:id" component={ProductDisplay} />
          </Route>
          <Route path="/*" component={NotFound} />
        </Router>
  );
}
```

Kuvio 26. SolidJS Reititys

Solid-router tarjoaa oman <A>-komponentin, joka toimii samalla tavalla kuin alkuperäinen HTML <a>-tagi. <A>-komponentilla on kuitenkin joitakin erityispiirteitä, kuten automaattinen perus-URL-polkujen sekä suhteellisten polkujen tuki. Lisäksi se tarjoaa end-ominaisuuden, jonka avulla voit määrittää tarkat reitit, joihin href-ilmaisu viittaa. (ks. Kuvio 27.)

```
<A href="/products/search">Search</A>
```

Kuvio 27. SolidJS reititys <A>-komponetti

React

Kuviossa 28 on toteutettu täysin samanlainen SPA-sovellus, mutta Reactilla. Tässä käytetään react-router-kirjastoa, joka on yleisin reitityskirjasto React-sovelluksissa ja johon myös solid-router perustuu. React Router tarjoaa useita erilaisia reititysratkaisuja eri tilanteisiin. Tässä esimerkissä käytämme BrowserRouter-komponenttia, joka on suositeltava kaikissa verkkosovellusprojekteissa, sillä se hyödyntää DOM History APIURL-osoitteen päivittämiseen sekä selainhistorian hallintaan.

Reactissa reittien määrittämiseen käytetään element-propeja, kun taas SolidJSä käytetään component-propeja. React hyödyntää myös Routes-komponenttia, jonka sisälle kaikki reitit määritellään. Tämä komponentti toimii hallintapisteenä, joka varmistaa, että vain yksi reitti renderöidään kerrallaan nykyisen URL-polun perusteella. SolidJSä Router-komponentti hoitaa reittien renderöinnin suoraan ilman erillistä Routes-rakennetta. Reactissa käytetään index-proppia reitille osoitta-

maan oletusreitti (esim. /), kun taas SolidJS ä tämä määritetään suoraan polkuominaisuuden sisään. Dynaamiset reitit ja sisäkkäinen reititys toimivat molemmissa kirjastoissa samalla periaatteella. React-router tarjoaa oman <a>-tagin korvikkeen nimeltä Link. Tämä komponentti mahdollistaa myös sisäisen navigoinnin React-sovelluksessa ilman sivun uudelleenlatausta. (ks. Kuvio 28.)

```
import { BrowserRouter as Router, Route, Routes } from 'react-router-dom';

// Pages
import Home from './pages/Home';
import About from './pages/About';
import Search from './pages/products/Search';
import ListProducts from './pages/products/ListProduct';
import AddProduct from './pages/products/AddProduct';
import NotFound from './pages/Notfound';
import ProductDisplay from './pages/products/ProductDisplay';

// Layouts
import ProductsLayout from './layouts/ProductsLayout';
import RootLayout from './layouts/RootLayout';

export default function App() {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<RootLayout />}>
          <Route index element={<Home />} />
          <Route path="about" element={<About />} />
          <Route path="products" element={<ProductsLayout />}>
            <Route path="list" element={<ListProducts />} />
            <Route path="search" element={<Search />} />
            <Route path="add" element={<AddProduct />} />
            <Route path=":id" element={<ProductDisplay />} />
          </Route>
          <Route path="*" element={<NotFound />} />
        </Route>
      </Routes>
    </Router>
  );
}
```

Kuvio 28. React reititys

Svelte

Sveltessä ei ole sisäänrakennettua reititystä, joten reititys toteutetaan usein kolmannen osapuolen kirjastolla, kuten svelte-spa-routerilla. Tämä on hash-pohjainen reitityskirjasto, joka on suunniteltu SPA-sovelluksien luomiseen. Toisin kuin SolidJSä, jossa RootLayout voidaan määritellä osana reititystä, Sveltessä layout-komponentti kuten RootLayout sijoitetaan manuaalisesti komponentiksi, jonka sisään reitittimen sisältö renderöidään.

Sveltessä reitit määritellään erillisessä muuttujassa, esimerkiksi routes, ja annetaan sen jälkeen Router-komponentille propsina. Tämä eroaa SolidJS:stä, jossa reitit voidaan suoraan määrittää Router-komponentin sisään, mutta Solidissa on mahdollisuus toimia myös näin. Samalla periaatteella myös tuotteisiin liittyvät reitit voidaan kääriä ProductsLayout-komponenttiin, joka tarjoaa tuotteisiin liittyvän navigoinnin ja asettelun, ja sisältää loput reitit samalla tavalla. Dynaamiset reitit määritellään Sveltessä samalla tavalla kuin SolidJSä käyttämällä /products/:id-tyylistä syntaksia, jolloin voidaan käsitellä yksilöityjä URL-polkuja. Näin sisäkkäiset ja dynaamiset reitit voidaan toteuttaa molemmissa kehyksissä samanlaisin menetelmin.

```
<script>
  import Router from 'svelte-spa-router';
  import Navbar from './components/Navbar.svelte';
  import Home from './pages/Home.svelte';
  import About from './pages/About.svelte';
  import NotFound from './pages/NotFound.svelte';

  import Rootlayout from './layouts/RootLayout.svelte';
  import Productslayout from './layouts/Productslayout.svelte';

  const routes = {
    '/': Home,
    '/about': About,
    '/products/*': Productslayout,
    '*': NotFound,
  };
</script>

<main>
  <Rootlayout>
    <Router {routes} />
  </Rootlayout>
```

```
</main>
```

Kuvio 29: Svelte reititys

SolidJS:n ekosysteemistä löytyvään Solid-Router-reititys ominaisuuteen, pätee sama kuin Solid-storeen, se voi helpottaa sovelluksen rakentamisessa, kun ei tarvitse integroida kolmannen osapuolen kirjastoja projektiin. Kuitenkin siinä piilee samainen ongelma, sillä vaikka se tarjoaakin työkalut perinteisten SPA-sovelluksien rakentamiseen, voi se mahdollisesti jäädä liian pieneksi, esimerkiksi React-routerin rinnalla. React-router tarjoaa monipuoleisemmin työkaluja, kuten erilaisia komponentteja ja hookkeja, mitkä helpottavat laajempien reititys ratkaisujen luomisessa.

5.7 Yhteisö ja ekosysteemi

SolidJS on nuori, nopeasti kasvava JavaScript-kirjasto, joka on tunnettu erityisesti sen suorituskyvystä sekä reaktiivisesta rakenteesta. Vaikka SolidJS on vielä uusi, se on kerännyt laajalti huomiota sekä kiinnostusta kehittäjien keskuudessa. StateofJS:n suorittaman kyselyn mukaan (2023) tietoisuus SolidJS:stä oli vuonna 2021 kehittäjien keskuudessa 37 % ja viime vuonna luku oli jo 76 %, mikä osoittaa kirjaston kasvavaa näkyvyyttä ja kiinnostavuutta. GitHubissa SolidJS on onnistunut keräämään itselleen jo 32,400 tuhatta tähteä ja noin 56 tuhatta käyttäjjää.

SolidJS-yhteisö on vielä suhteellisen pieni, mutta se on aktiivinen ja jatkuvassa kasvussa. Yhteisöllä on monia aktiivisia kokoontumispaikkoja, kuten Discord, GitHub, Reddit sekä X. SolidJS virallisella Discord-kanavalla kehittäjät keskustelevat aktiivisesti, jakaen omia projektejaan ja auttaen toisiaan. GitHubissa SolidJS:n kehittäjät voivat osallistua kirjaston kehitykseen ja seurata uusimpia päivityksiä. Lisäksi yhteisö on aktiivinen Reddit:ssä ja X:ssä, joissa keskustellaan ajankohtaisista julkaisuista ja jaetaan mielipiteitä.

SolidJS:n ekosysteemi on myös kehittymässä vauhdilla ja se tarjoaakin jo monia työkaluja sekä kirjastoja, jotka helpottavat kehitystyötä. Näitä on saatavilla SolidJS:n virallisilta nettisivuilta, mistä löytyy mm. lisäosia, reitittimiä, testausvälineitä sekä UI-elementtejä. Osa näistä on merkitty ”official” merkinnällä, mikä kertoo, että se on SolidJS:n virallisesti julkaisema, mutta sieltä löytyy myös monia aktiivisten kehittäjien rakentamia lisäosia. SolidJS tukee jo monia suosittuja kolmannen osa-

puolen kirjastoja ja palveluita, sekä sen ekosysteemi laajenee jatkuvasti, mikä parantaa sen integroitavuutta ja laajentaa sen käyttökohteita eri kehitysympäristöissä. SolidJS tarjoaa myös kattavat ja hyvät dokumentaatiot.

Toisin kuin SolidJS, React on yksi tunnetuimmista ja laajimmin käytetyistä JavaScript-kirjastoista, joka on saavuttanut valtavan suosion ja menestyksen kehittäjien sekä yritysten keskuudessa. React on huomattavasti SolidJS:ää vanhempi, sillä se julkaistiin jo vuonna 2013 ja on ollut merkittävä osa web-sovellusten kehitystä. Samaisen vuonna 2023 julkaistun StateofJS-kyselyn mukaan, React oli kaikista frontend kirjastoista tunnetuin, sekä käytetyin. Kiinnostus Reactia kohtaan on kuitenkin selvässä laskussa, sillä vuonna 2016 75 % piti Reactia kiinnostavan, mutta vuonna 2023 kiinnostus on pudonnut 42 prosenttiin. Reactin suosio näkyy myös GitHubissa, missä se on kerännyt itselleen jo 229 tuhatta tähteä sekä sitä on käyttänyt miltei 25 miljoona eri GitHub käyttäjää.

Reactilla on valtava ja aktiivinen yhteisö ja se on vakiinnuttanut asemansa kansainvälisesti. Myös React yhteisö on aktiivinen eri alustoilla kuten GitHubissa, Redditissä, Discordissa ja monilla muilla foorumeilla. React järjestää myös universaaleja konferensseja ja tapaamisia ympäri maailmaa, jotka ovat avoinna kehittäjille. Yhteisön laajuuden ansiosta, kehittäjät voivat löytää todella paljon resursseja kuten avoimen lähdekoodin projekteja, dokumentaatiota tai ohjeita, jotka ovat tukena kehittämisessä. Facebookin tuki on ollut myös merkittävässä asemassa Reactin kehityksessä ja vakauden ylläpitämisessä.

React-ekosysteemi on todella laaja ja sille on saatavilla erittäin paljon erilaisia työkaluja sekä kirjastoja. Kuten myös SolidJS:lle, myös Reactille on saatavilla todella paljon erilaisia lisäosia ja valmiita koodeja. React integroituu lähes kaikkien suosituimpien kolmannen osapuolen kirjastojen kanssa ja saatavilla on myös todella paljon yksinomaan Reactille tarkoitettuja kirjastoja. Se integroituu myös erityisen hyvin muiden modernien web-teknologioiden kanssa, mikä tekee siitä todella monipuolisen työkalun. Myös React tarjoaa hyvät dokumentaatio sivut, Reactin opetteluun. Reactin lisäksi, se tarjoaa myös vaihtoehdot React Native sekä Next.js, mitkä ovat todella suosittuja omissa käyttötarkoituksissaan.

Svelte muistuttaa hyvin paljon SolidJS:ää, mikä on julkaistu muutaman vuoden aikaisemmin kuin SolidJS. Svelte on alkanut vakiinnuttamaan paikkaansa käyttöliittymäkehityksen alalla viimeisten

vuosien kuluessa. Vuonan 2023 suoritetun StateofJS-kyselyn mukaan, kehittäjien mielestä Svelte herätti kaikista kirjastoista eniten kiinnostusta vuodesta 2019 alkaen. Sveltestä ollaan siis hyvinkin kiinnostuneita ja sen tietoisuus kehittäjien keskuudessa on noussut vuoden 2019 75 prosentista 96 prosenttiin. Svelte on saanut GitHubissa 79,800 tuhatta tähteä sekä sitä on käyttänyt jo noin 342 tuhatta eri käyttäjää.

Svelten yhteisö on hieman laajempi kuin SolidJS:llä. Se on aktiivinen ja jatkuvassa kasvussa. Kehittäjät keskustelevat Svelten liittyvistä asioista erinäköisillä alustoilla ja foorumeilla kuten GitHubissa, Redditissä sekä Discordissa. Näiden avulla yhteisön jäsenet voivat keskustella Sveltestä, jakaa resurssejaan sekä ohjata toisiaan meneillään olevissa projekteissa. Samoin kuin SolidJS:llä, myös Sveltellä avoimen lähdekoodin luonne ja sen ympärillä oleva aktiivinen yhteisö ovat merkittävässä roolissa sen jatkuvassa kehityksessä ja uusien ominaisuuksien lisäämisessä.

Kuten myös SolidJS:lle, myös Sveltelle on saatavilla useita työkaluja ja kirjastoja mitkä auttavat kehitystyötä. Myös Svelte-ekosysteemi on kasvava ja laajentumassa koko ajan suuremmaksi. Svelten iso askel eteenpäin käyttöliittymäkehityksen saralla tapahtui SvelteKitin julkaisemisen myötä. SvelteKit on Svelte-kirjaston päälle rakennettu full-stack kehys, joka tarjoaa kehittäjille tehokkaat työkalut ja toiminnallisuudet web-sovellusten luomiseen. Svelte on myös hyvin integroituva suosittujen kolmannen osapuolen kirjastojen sekä lisäosien kanssa ja se tarjoaa myös hyvät dokumentaatio sivut.

Yhteenvetona voidaan todeta, että SolidJS:n yhteisö ja ekosysteemi ovat samankaltaisia Svelten kanssa: ne ovat avoimia ja lähestyttäviä. SolidJS:n yhteisö on hieman epämuodollisempi ja maanläheisempi verrattuna Reactin virallisempaan yhteisöön. Reactilla on paljon virallisia tapahtumia sekä konferensseja, joista osa on Metan tukemia. SolidJS puolestaan on enemmän yhteisölähtöinen ja esimerkiksi SolidJS:n virallisella Discord-kanavalla voi suoraan keskustella muiden kehittäjien sekä itse SolidJS:n pääkehittäjä, Ryan Carniaton kanssa. Tämä tekee SolidJS:stä helposti lähestyttävän, erityisesti uusille käyttäjille, joille apu on helposti saatavilla. Vaikka SolidJS tarjoaakin erinomaisen suorituskyvyn ja yksinkertaisemman lähestymistavan, sen pienempi ekosysteemi voi aiheuttaa haasteita suurissa projekteissa, joissa tarvitaan laajempaa tukea erilaisille kirjastoratkaisuille. Pienemmän ekosysteemin takia, SolidJS:lle voi olla vaikeampaa löytää esimerkiksi valmiita komponentteja, jotka nopeuttaisivat kehitystyötä.

6 Pohdinta

Tässä luvussa esitetään opinnäytetyön keskeiset johtopäätökset, arvioidaan työn luotettavuutta ja eettisyyttä sekä esitellään jatkokehitysehdotuksia. Luvun tarkoituksena on tarkastella aiemmissa luvuissa esiin nousseita havaintoja, joiden avulla voidaan muodostaa opinnäytetyöstä perustellut johtopäätökset. Luotettavuuden sekä eettisyyden arvioinnissa keskiössä on erityisesti vertailukehyksen käyttö tutkimuksen runkona sekä lähteiden valinta. Jatkokehitysehdotuksissa nostetaan esiin ajatuksia ja pohdintoja siitä, miten opinnäytetyötä voisi tarkastella syvällisemmin sekä laajentaa sen näkökulmia tulevassa tutkimuksessa tai kehitystyössä.

6.1 Johtopäätökset

Opinnäyteyön tavoitteena oli selvittää kuinka SolidJS-kirjasto käsittelee frontend-kirjastojen keskeisiä ominaisuuksia. Tutkimuksessa SolidJS-kirjastoa esiteltiin, tutkittiin sekä vertailtiin sen toimintatapoja Svelten sekä Reactin kanssa, ennalta laaditun vertailukehyksen avulla. Opinnäytetyön tavoite saavutettiin ja asetettuihin tutkimuskysymyksiin saatiin vastaus tietoperustan, vertailukehikon avulla toteutetussa vertailussa sekä pohdinnan avulla.

SolidJS toteuttaa luvussa 3.3 mainittujen modernien käyttöliittymäkirjastojen keskeisiä ominaisuuksia yhdistämällä reaktiivisen ohjelmoinnin ja vahvan suorituskyvyn. SolidJS:n tapa käyttää signaaleja ja muita reaktiivisia rakenteita, mahdollistaa suoran reaktiivisen datan muutoksen DOM-päivityksiin. SolidJS-sovellus hyödyntää komponenttipohjaista arkkitehtuuria, jolla vähennetään koodin toistoisuutta, sekä tehdään komponenteista uudelleenkäytettäviä. Komponentit ovat kevyitä ja yksinkertaisia, mikä tukee ylläpidettävyyttä sekä modulaarisuutta. SolidJS:stä löytyy myös moderneille kirjastoille tyypillisiä ratkaisuja kuten luvussa 3.3 on esitetty, näitä ovat mm. sisäänrakennettu reititys, tilanhallinta sekä storet. Lisäksi TypeScript-tuen tarjoaminen on myös erittäin tärkeää nykyään modernien web-sovelluksien rakentamisessa, sillä se mahdollistaa tyypitetyn ja turvallisen kehitystyön.

Kuten luvussa 5 suoritetussa vertailussa ilmiää, SolidJS:n arkkitehtuuri eroaa Reactista ja Sveltestä erityisesti sen suoraviivaisessa reaktiivisuuden hallinnassa ja lähestymistavassa DOM-päivityksiin. Luvussa 5.4 esitettyjen vertailujen perusteella voidaan todeta, että suurimmat erot Reactiin löytä-

vät reaktiivisuuden hallinnasta. React käyttää virtuaalista DOM:ia, jonka avulla se suorittaa muutoksia DOM:iin, kun taas SolidJS suorittaa tarkat päivityksen suoraan oikeaan DOM:iin. Muutoksien sattuessa React usein renderöi koko DOM:in uudelleen, mikä voi aiheuttaa turhaa laskentaa, vaikka vain pieni osa DOM:sta muuttuisi. SolidJS sen sijaan suorittaa päivitykset DOM:iin vain niiden osien kohdalla, jotka todella muuttuvat. Tämä tekee SolidJS:stä erittäin suorituskykyisen, erityisesti sovelluksissa, joissa päivityksiä tapahtuu jatkuvasti.

Svelte eroaa SolidJS:stä hieman enemmän kuin React, erityisesti luvuissa 5.2 ja 5.4 suoritettujen vertailujen pohjalta. Sveltessä reaktiivisuus perustuu muuttujapohjaiseen malliin, joka optimoidaan tehokkaaksi JavaScriptiksi jo käännösvaiheessa. SolidJS puolestaan toteuttaa reaktiivisuuden ajonaikaisesti signaalien avulla, mikä mahdollistaa dynaamisemman tilan hallinnan. Syntaksiltaan Svelte poikkeaa merkittävästi SolidJS:stä, joka käyttää Reactin tavoin JSX-pohjaista lähestymistapaa. Sveltessä kaikki koodi (JavaScript, HTML ja CSS) yhdistetään yhteen tiedostoon, mikä tarjoaa selkeän ja modulaarisen rakenteen sovelluskehitykseen.

Vaikka SolidJS tarjoaakin erittäin tehokkaan ja reaktiivisen alustan käyttöliittymäkehitykseen, sillä on myös heikkouksia. Olen listannut alle SolidJS:n merkittävimpiä vahvuuksia ja heikkouksia kehittäjän näkökulmasta:

Taulukko 1. SolidJS:n merkittävimmät vahvuudet sekä heikkoudet

Vahvuudet	**Heikkoudet**
Suorituskyky: Reaktiivisen ohjelmointi mallin takia, SolidJS on yksi nopeimmista frontend-kirjastoista.	**Pieni ekosysteemi:** Vaikka SolidJS yhteisö on aktiivinen ja kasvava, on se silti vielä kehitysvaiheessa, mikä voi vaikeuttaa esimerkiksi tuen saamista. Myös kirjastojen ja valmiiden komponenttien löytäminen voi olla haasteellisempaa.

Reaktiivisuus: Hienojakoisen reaktiivisuuden ansiosta tilanhallinta on suoraviivaista ja ennakoitua. Signaalien avulla reaktiivisuus on helposti seurattavissa.	**Tuki ja vakaus:** Koska SolidJS on suhteellisen uusi kirjasto, sen pitkäaikainen tuki ja vakaus eivät vielä ole täysin vakiintuneet, mikä voi herättää epävarmuutta yrityksien tuotannossa.
Dokumentaaatio ja kasvava yhteisö: SolidJS tarjoaa hyvät dokumentaatio sivut, mikä madaltaa oppimiskäyrää. SolidJS yhteisö on kasvavassa asemassa ja JSX-syntaksi houkuttelee myös Reactiin tottuneita kehittäjiä.	
Kevyt ja modulaarinen: SolidJS on erittäin kevyt kirjasto ja se tarjoaa myös monien modernien web-kehitystyökalujen, kuten Astron sekä Viten tuen.	

Yhteenvetona voidaan todeta, että SolidJS on erityisen suorituskykyinen ja erittäin lupaava kirjasto modernien web-sovelluksien rakentamiseen ja täyttääkin luvussa 3.3 esitettyjen modernien frontend JavaScript-kirjastojen keskeisiä piirteitä. Vaikka SolidJS on vielä suhteellisen uusi frontend-kirjasto, sen suosio on jatkuvassa kasvussa ja sen innovatiivinen reaktiivinen lähestymistapa herättää kiinnostusta yhä useammissa kehittäjissä. SolidJS:n kasvun myötä sillä on myös potentiaalia avata uusia mahdollisuuksia web-sovellusten ja käyttöliittymien kehityksessä, tarjoten tulevaisuudessa tehokkaan ja yksinkertaisen ratkaisun monimutkaistenkin web-sovellusten kehittämiseen.

On hyvä kuitenkin huomioida, että vaikka SolidJS tarjoaakin poikkeuksellista suorituskykyä ja erinomaisen kehittäjäkokemuksen, sen käyttöönotto voi edelleen olla riski yrityksille. Tämä johtuu sen

suhteellisen nuoresta asemasta ekosysteemissä, rajallisesta yhteisö- ja työkalutuesta sekä epävarmuudesta pitkäaikaisen tuen ja vakauden suhteen. Yritykset saattavat siksi suosia vakiintuneempia vaihtoehtoja, jotka tarjoavat laajemman ekosysteemin ja varmemman vakauden tulevaisuudessa.

6.2 Luotettavuuden sekä eettisyyden arviointi

Vertailun luotettavuutta on varmistettu toteuttamalla se ennalta laaditun vertailu kehyksen avulla, mikä mahdollisti sen toteuttamisen johdonmukaisesti sekä tasavertaisesti. Vertailussa käytetty koodi on pyritty luomaan mahdollisimman yhdenmukaisesti jokaisen kirjaston kohdalla ja niitä on pyritty lähestymään sekä tulkitsemaan mahdollisimman tasapuolisesti. Jokaisen osa-alueen kohdalla on pyritty rakentamaan tilanne, jossa kirjaston toimintatavat näkyvät selkeästi ja vertailukelpoisesti. Kuten luvun 5 alussa mainittiin, vertailut ovat kuitenkin suuntaa antavia, eikä niissä käsitellä kaikkia teknisiä ominaisuuksia tai toimintatapoja. Ohjelmistokehityksessä on useita ratkaisuja eri tilanteisiin, niin on hyvä muistaa, että kyseiset vertailut voidaan toteuttaa myös vaihtoehtoisin menetelmin eivätkä ne sulje muiden menetelmien käyttökelpoisuutta.

Tutkimus on toteutettu pääasiassa vertailukehyksen avulla, joka on esitelty tarkemmin luvussa 2.5. Vertailukehyksen luotettavuutta ja eettisyyttä voidaan arvioida tarkastelemalla sen laatimisprosessia ja käsiteltyjä aiheita. Kehyksen aiheet on valittu niiden merkityksen perusteella, ottaen huomioon modernien frontend-kirjastojen vertailuissa esiin nousseet keskeiset tekijät, ja niitä on muokattu sopimaan tämän tutkimuksen tarpeisiin. Vertailukehys on pyritty suunnittelemaan tutkimuksen tavoitteisiin soveltuvaksi ja mahdollisimman puolueettomaksi. Lisäksi kehyksen on tarkoituksella pyritty säilyttämään riittävä objektiivinen etäisyys, jotta se mahdollistaa kirjastojen ominaisuuksien tarkastelun kokonaisvaltaisesti. On kuitenkin hyvä huomioida, että vertailukehyksen sisältämät osa-alueet eivät välttämättä edusta kaikkien mielestä modernin frontend-kirjaston keskeisimpiä ominaisuuksia, sillä näkemykset voivat vaihdella yksilöllisesti.

Opinnäytetyössä on pyritty hyödyntämään työn aiheeseen soveltuvia ja luotettavia lähteitä. Lähteitä on arvioitu kriittisesti ja niiden soveltuvuutta työssä käsiteltäviin aiheisiin on harkittu huolellisesti. Lisäksi on pyritty käyttämään monipuolisesti erityyppisiä lähteitä. Yksi keskeisimmistä lähteistä on ollut SolidJS:n virallinen dokumentaatio, joka tarjoaa luotettavaa ja aiheeseen parhaiten soveltuvaa tietoa. Työn aiheen rajallisen tutkimuksellisen materiaalin vuoksi joissakin tapauksissa

on käytetty verkkolähteitä. Verkkolähteistä hankittu tieto on kuitenkin pyritty tarkistamaan mahdollisimman huolellisesti ja pyritty varmistamaan sen oikeellisuus vertailemalla useita lähteitä.

Opinnäytetyössä on hyödynnetty tekoälyä (ChatGPT) kieliopin tarkistamiseen ja tekstien kääntämiseen. Eettisyyden ja luotettavuuden kannalta tutkimuksessa on otettu huomioon, että tekoälyllä käännetty tai tarkastettu teksti ei ole virheetöntä ja alkuperäisen tekstin sisältö ei saa olennaisesti muuttua.

Eettisyyden ja luotettavuuden kannalta opinnäytetyössä on pyritty noudattamaan koko tutkimuksen ajan Tutkimuseettisen neuvottelukunnan (TENK) ja yhteistyössä suomalaisen tiedeyhteisön kanssa laatimaa ohjeistusta hyvästä tieteellisestä käytännöstä (HTK). Hyvän tieteellisen käytännön tarkoituksena on ohjata tieteellisen työn tekemistä rehellisesti, avoimesti, huolellisesti ja tarkasti. Siinä myös sovelletaan tieteelliseen tutkimukseen sopivia eettisiä tiedonhankinta-, tutkimus- ja arviointimenetelmiä. (Hyvä tieteellinen käytäntö 2023.)

Tutkimuksen aikana ei kerätty eikä käsitelty henkilötietoja tai muuta arkaluonteista aineistoa. Tämän vuoksi salassapitosopimusta tai tutkimuslupaa ei ollut tarpeen laatia.

6.3 Jatkokehitysehdotukset

Opinnäytetyössä todettiin, että SolidJS on suorituskykyinen ja lupaava kirjasto modernien web-sovelluksien luomiseen. Tutkimuksessa tarkasteltiin kuitenkin vain arkkitehtuurillisia ratkaisuja, eikä tarkempiin mittaustuloksiin paneuduttu lainkaan. Yhtenä jatkokehitysehdotuksena olisikin ehdottomasti tarkempien suorituskykymittauksien toteuttaminen ja näistä syntyvien tulosten analysointi. Tällä tavoin voitaisiin selvittää SolidJS:n suorituskyvyn tehokkuutta tarkemmin ja nähdä, kuinka SolidJS suoriutuu verrattuna muihin suosittuihin frontend-kirjastoihin, kuten tässä opinnäytetyössä käytettyihin Reactiin ja Svelteen.

Tarkemmat mittaukset antaisivat myös näkemyksiä siitä, kuinka suorituskyvyn tehokkuus vaikuttaa konkreettisesti sovelluksen toimintaan ja käyttäjäkokemukseen. Tätä varten voitaisiin rakentaa demosovellus, jonka avulla voitaisiin simuloida todellisia käyttötapauksia ja analysoida, minkälaisia hyötyjä suorituskyvyn tehokkuus tarjoaa sekä syventää ymmärrystä SolidJS:n vahvuuksista.

Lähteet

Allen Wirfs-Brock and Brendan Eich. 2020. JavaScript: The First 20 Years. Proc. ACM Program. Lang. 4, HOPL. Viitattu 20.3.2024. https://www.wirfs-brock.com/allen/jshopl.pdf

Basic reactivity. N.d. Viitattu 12.10.2024. https://docs.solidjs.com/reference/basic-reactivity/create-effect

Bigelow, J. 2024. State management. Viitattu 22.9.2024. https://www.techtarget.com/searchapparchitecture/definition/state-management

Choosing the Best JavaScript Framework. 2024. Viitattu 10.3.2024. https://www.sencha.com/blog/how-to-choose-the-best-javascript-frameworks-in-2023/

Class and style. N.d. Viitattu 15.10.2024. https://docs.solidjs.com/concepts/components/class-style

Component lifecycles. N.d. Viitattu 15.10.2024. https://docs.solidjs.com/concepts/components/basics#component-lifecycles

Component trees. N.d. Viitattu 15.10.2024. https://docs.solidjs.com/concepts/components/basics#component-trees

Das, A. 2024. Data Binding in JavaScript. Viitattu 20.9.2024. https://arunangshudas.medium.com/data-binding-in-javascript-098d850023ca

Desai, J. 2024. 5 Best Frontend Frameworks for Web Development. Viitattu 10.3.2024. https://positiwise.com/blog/best-front-end-frameworks#Compare_React_vs_Angular_vs_Vue_vs_Svelte_vs_Ember

Derived state. N.d. Viitattu 20.10.2024. https://docs.solidjs.com/concepts/components/basics#component-trees

Developers survey. 2023. Viitattu 20.3.2024. https://survey.stackoverflow.co/2023/

Duckett, Jon. JavaScript & JQuery: Interactive Front-End Web Development. John Wiley & Sons. © 2014. Skillsoft. Viitattu 3.3.2024. https://2masteritezproxy.skillport.com/skillportfe/main.action?assetid=58167

Event handlers. N.d. Viitattu 16.10.2024. https://docs.solidjs.com/concepts/components/event-handlers

Fine-grained reactivity. N.d. Viitattu 17.10.2024. https://docs.solidjs.com/advanced-concepts/fine-grained-reactivity

Framework main features. 2024. Viitattu 10.3.2024. https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Main_features

Frisbie, Matt. Professional JavaScript for Web Developers, Fifth Edition. John Wiley & Sons. © 2024. Skillsoft. Viitattu 25.3.2024. https://2masteritezproxy.skillport.com/skillportfe/main.action?assetid=165364

Grant, Keith J. CSS in Depth. Manning Publications. © 2018. Skillsoft. Viitattu 3.3.2024. https://2masteritezproxy.skillport.com/skillportfe/main.action?assetid=147137

Hyvä tieteellinen käytäntö (HTK). 2023. Viitattu 20.11.2024. https://tenk.fi/fi/tiedevilppi/hyva-tieteellinen-kaytanto-htk

Immukul. 2024. ReactJS Virtual DOM. Viitattu 22.3.2024. https://www.geeksforgeeks.org/reactjs-virtual-dom/

Introduction to client-side frameworks. 2024. Viitattu 27.3.2024. https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Introduction

Intro to reactivity. N.d. Viitattu 12.10.2024. https://docs.solidjs.com/concepts/intro-to-reactivity

JavaScript frameworks. 2024. Viitattu 10.3.2024. https://developer.mozilla.org/en-US/curriculum/extensions/a-practical-understanding-of-javascript-frameworks/

JSONPlaceholder. 2024. Viitattu 27.9.2024. https://jsonplaceholder.typicode.com/

Kananen, J. 2017. Laadullinen tutkimus pro graduna ja opinnäytetyönä. Jyväskylä: Jyväskylän ammattikorkeakoulu.

McFedries, Paul. Web Coding & Development All-in-One for Dummies. John Wiley & Sons. © 2018. Skillsoft. Viitattu 3.3.2024. https://2masteritezproxy.skillport.com/skillportfe/main.action?assetid=142528

Narayn, H. Just React! Learn React the React Way. Apress. © 2022. Skillsoft. Viitattu 14.3.2024. https://2masteritezproxy.skillport.com/skillportfe/main.action?assetid=163518

Nested routes. N.d. Viitattu 25.10.2024. https://docs.solidjs.com/guides/routing-and-navigation#nested-routes

Overview. N.d. Viitattu 10.10.2024. https://docs.solidjs.com/#overview

Overview. N.d. Viitattu 25.10.2024. https://docs.solidjs.com/solid-router/

Path parameters. N.d. Viitattu 25.10.2024. https://docs.solidjs.com/solid-router/concepts/path-parameters

Pesquet, B. 2020. The JavaScript Way. Viitattu 1.3.2024. https://thejsway.net/

Props. N.d. Viitattu 16.10.2024. https://docs.solidjs.com/concepts/components/props

Scott, Jr., Emmit A. SPA Design and Architecture: Understanding Single-Page Web Applications. Manning Publications. © 2015. Skillsoft. Viitattu 10.10.2024. https://2masteritezproxy.skillport.com/skillportfe/main.action?assetid=147239

State management. N.d. Viitattu 23.101.2024. https://docs.solidjs.com/guides/state-management

Stores. N.d. Viitattu 23.10.2024. https://docs.solidjs.com/concepts/stores

Subscribers. N.d. Viitattu 18.10.2024. https://docs.solidjs.com/concepts/intro-to-reactivity#subscribers

Synchronous vs. asynchronous. N.d. Viitattu 19.10.2024. https://docs.solidjs.com/concepts/intro-to-reactivity#synchronous-vs-asynchronous

Understanding JSX. N.d. Viitattu 20.9.2024. https://docs.solidjs.com/concepts/understanding-jsx

What is a software component?. 2024. Viitattu 20.11.2024. https://www.geeksforgeeks.org/difference-between-module-and-software-component/

Wolf, Jürgen. HTML and CSS: The Comprehensive Guide. SAP Press. © 2023. Skillsoft. Viitattu 3.3.2024. https://2masteritezproxy.skillport.com/skillportfe/main.action?assetid=164820