Tampereen yliopisto

Jaakko Humalajoki

SVELTEN OMINAISUUDET JA SEN VERTAILU REACTIIN

Kandidaatintyö
Informaatioteknologian ja viestinnän tiedekunta
Tammikuu 2022

TIIVISTELMÄ

Jaakko Humalajoki: Svelten ominaisuudet ja sen vertailu Reactiin
Kandidaatintyö
Tampereen yliopisto
Tietotekniikan tutkinto-ohjelma
Tammikuu 2022

Verkkokehityksen alalla kehysympäristöt ovat teknologioita, jotka antavat sovellukselle valmiin rungon ja toimintamallin, jolla on helpompi rakentaa moderneja verkkosivuja. Sopivan kehysympäristön valitseminen verkkosovelluksen rakentamiseen on tärkeää ennen sovellusprojektin aloittamista, sillä valitun kehysympäristön ominaisuudet voivat merkittävästi vaikuttaa projektin valmistumisen nopeuteen ja lopputulokseen. Tästä syystä kehysympäristöjä on hyvä tutkia etukäteen ja verrata niitä keskenään. Tässä työssä tutkittiin Svelteä ja sen ominaisuuksia, sillä Svelte on yksi uudempia tulokkaita verkkokehityksen alalla. Svelten vertailukohteeksi valittiin React, sillä React on tällä hetkellä suosituin kehysympäristö, ja alalla työskentelevät todennäköisesti tuntevat Reactin ennestään.

Työn lähteinä on käytetty suurimmaksi osaksi kehysympäristöjen omaa dokumentaatiota sekä kehysympäristöistä kirjoitettuja oppikirjoja. Työssä käytiin läpi ensin verkkokehityksen perusteita, jonka jälkeen tutkitaan Svelten ominaisuuksia, ja lopulta Svelteä verrataan tarkemmin Reactiin. Svelteä ja Reactia objektiivisesti vertailevia lähteitä oli hyvin vaikea löytää, joten erityisesti oppimateriaalin laatuun ja ohjelmointisyntaksin helppouteen liittyvät havainnot ovat suurimmaksi osaksi kirjoittajan omia mielipiteitä kehysympäristöjen opiskelun ja tutkimisen ajalta. Työssä on kuitenkin tutkittu myös kehysympäristöjen tehokkuutta, jossa on käytetty laajoja julkisia mittaustuloksia Svelten ja Reactin vertailuun.

Työssä havaittiin, että Svelten pakollinen käännösvaihe on monin tavoin hyödyllinen sovelluskehityksessä. Käännöksen aikana Svelte optimoi sovelluksen mahdollisimman nopeaksi ja pieneksi, eikä sovelluksen ajon aikana prosessointia tarvita enää kehysympäristön ylläpitoon. Käännösvaihe erottaa Svelten ohjelmointisyntaksin lopullisesta ohjelmakoodista, mikä mahdollistaa yksinkertaisemman ja lyhyemmän ohjelmointisyntaksin. Työssä käytettyjen tutkimusten perusteella Sveltellä luodut sovellukset ovat tehokkaampia ja koodirivillisesti lyhyempiä kuin Reactin vastaavat toteutukset.

Avainsanat: Svelte, React, Javascript, Kehysympäristö, Verkkosovellus

SISÄLLYSLUETTELO

LYHENTEET JA MERKINNÄT

API	Ohjelmointirajapinta (engl. Application Programming Interface)
CSS	Porrastetut tyyliarkit (engl. Cascading Style Sheets)
DOM	Dokumenttioliomalli (engl. Document Object Model)
HTML	Hypertekstin merkintäkieli (engl. Hypertext Markup Language)
IDE	Ohjelmointiympäristö (engl. Integrated Development Environment)
NPM	Noden paketinhallintajärjestelmä (engl. Node Package Manager)

1. JOHDANTO

Verkkosivujen kehitystyökalut ovat vaihtuneet ja muuttuneet paljon kahden viime vuosikymmenen aikana. Javascript-kirjastoja ja kehysympäristöjä julkaistaan jatkuvasti lisää, minkä takia verkkosivuohjelmoijien on vaikeata pysyä uusien teknologioiden kehityksen perässä. Viime vuosina tilanne on kuitenkin hiukan rauhoittunut, kun kehittäjien suosioon vakiintui kolme kehysympäristöä: React, Angular ja Vue. Näistä suosituin on React, mikä näkyy sekä viikottaisten latausten määrässä [1] että alan mittavassa mielipidekyselyssä [2].

Kehysympäristöjen kehittäjät kuitenkin jatkavat innovointia, ja todennäköisesti jokin päivä jokin uusi kehysympäristö tulee korvaamaan nykyiset teknologiat. Yksi uusista tulokkaista on Svelte, joka on nostanut suosiotaan viime vuosina ja joka nousi uusimman mielipidekyselyn mukaan pidetyimmäksi kehysympäristöksi [2]. Svelten suosion syinä ovat muun muassa sen tehokkuus sekä sen hyvä kehittäjäkokemus.

Kehysympäristöjen vertailu on hyödyllistä mietittäessä, mikä niistä soveltuu parhaiten uuden sovelluksen kehittämistyökaluksi. Tekemällä tarkkaan harkittuja valintoja ennen työn aloittamista voidaan säästyä monelta vaikeudelta sovelluskehityksen aikana, sillä kehysympäristöjen eri ominaisuudet voivat merkittävästi hidastaa tai nopeuttaa työn etenemistä. Pahimmassa tapauksessa sovelluskehitys joudutaan aloittamaan uudestaan toisella kehysympäristöllä, mikäli vakavia ongelmia ilmenee kesken työskentelyn.

Tämän työn tarkoituksena on käydä läpi Svelten ominaisuuksia ja selvittää, miten ne auttavat verkkosivujen kehityksessä. Svelteä verrataan Reactiin, sillä React on suosituimpana kehysympäristönä parhaiten tunnettu teknologia tällä alalla ja todennäköisesti alalla työskentelevät joko käyttävät Reactia tai tietävät sen perusteet. Reactista puhuttaessa keskitytään erityisesti sen funktiosyntaksiin, joka on modernimpi ja helpompi tapa kirjoittaa Reactia.

Toisessa luvussa kerrotaan lyhyesti verkkosivukehityksen historiaa ja kehysympäristöjen merkitystä. Kolmannessa luvussa kerrotaan Sveltestä ja sen ominaisuuksista. Neljännessä luvussa käydään tarkempaa vertailua Reactin ja Svelten välillä muun muassa koodin luettavuuden ja tehokkuuden kannalta. Viidennessä luvussa pohditaan Svelteä ja sen tulevaisuutta. Viimeisessä luvussa on tiivistettynä työn tärkeimmät havainnot.

2. KEHYSYMPÄRISTÖJEN PERUSTEET

2.1 Verkkokehityksen historia

Verkkosivut koostuvat tyypillisesti neljästä eri tiedostotyypistä: HTML:stä, CSS:stä, Javascriptistä ja mediatiedostoista. HTML-tiedostot sisältävät verkkosivun rungon ja sisällön, ja ne ovat verkkosivun rakentamisen ainut pakollinen osa. CSS-tiedostot määrittelevät verkkosivun ulkonäköön liittyvät säännöt, joilla verkkosivun sisältöä voi muokata paremman näköiseksi. Mediatiedostot ovat joko kuvia, videoita tai äänitiedostoja, joita esitetään verkkosivulla. Javascript-tiedostot sisältävät ohjelmakoodia, jonka selaimen sisäinen Javascript-tulkki pystyy ajamaan ja jonka avulla verkkosivuun voi tehdä muutoksia esimerkiksi tekstin päivittämistä varten tai lisätä interaktiivisuutta käyttäjän toiminnoille.

Ensimmäinen HTML-spesifikaatio julkaistiin vuonna 1991, mutta Javascript kehitettiin vasta myöhemmin vuonna 1995, jolloin se lisättiin Netscape-selaimeen. Tätä ennen kaikki verkkosivut olivat staattisia, eikä niitä voinut ohjelmoida. Javascript lisättiin vuotta myöhemmin vuonna 1996 mukaan Ecman kansainväliseen standardiin nimellä ECMAScript, ja tätä standardia on ylläpidetty ja päivitetty siitä lähtien standardina kaikille selaimille. [3] Nykypäivänä lähes kaikki modernit selaimet tukevat ECMAScriptin 2015-versiota ES6:a kokonaan [4], ja selainkehittäjät lisäävät uusia ominaisuuksia uudemmista standardeista jatkuvasti.

Koska selaimet kehittyivät eri tahtiin ja selaimien tuki eri ECMAScriptin versioille oli vaihtelevaa, alkuaikojen verkkosivuohjelmointi oli erityisen hankalaa. Verkkosivujen tekijöiden piti itse ottaa huomioon eri selaimet ja varmistaa, että sovellus toimi oikein kaikilla selaimilla. Vuonna 2006 tätä ongelmaa korjaamaan tuli jQuery-niminen Javascript-kirjasto, jonka yksi perusideoista oli hoitaa selainten välinen yhteensopivuus kehittäjien puolesta sekä helpottaa verkkosivujen interaktiivisuuden koodaamista. [5] JQueryn voi sanoa olleen ensimmäinen askel kohti nykypäivän kehysympäristöjä, mutta jQuery ei itse ole kehysympäristö. JQuery ei ota kantaa koko sovelluksen rakenteeseen, vaan tarjoaa monia yksittäisiä ominaisuuksia, joita kehittäjä voi itse käyttää haluamallaan tavalla.

2.2 Kehysympäristöjen merkitys

Kehysympäristöt auttavat kehittäjiä verkkosovelluksien rakentamisessa antamalla valmiin rungon sovelluksen tekemiseen ja automatisoimalla vaikeita ominaisuuksia koodissa. Jokaisella kehysympäristöllä on omat vahvuutensa ja erikoisominaisuutensa, mutta yleisesti niiden voi sanoa helpottavan kehittäjän työtä ja nopeuttavan verkkosovelluksien valmistumista. Verkkosivuilta vaaditaan yhä enemmän, ja nykyään kehittäjien olisi mahdotonta tehdä kaikkea itse ilman kehysympäristöjen ja Javascript-kirjastojen apua.

Kehysympäristöjen hyötypuolena on myös niiden ympärille kasvaneet kehittäjäyhteisöt. Isoissa projekteissa kehittäjien pitää pystyä toimimaan yhdessä ja ymmärtää toistensa kirjoittamaa koodia, ja tässä kehysympäristöjen tuoma standardointi koodaustyylissä ja käytännöissä on suuri etu. Suosituimpien kehysympäristöjen ympärille on myös muodostunut valtava määrä ilmaista opiskelumateriaalia, ja moniin yleisiin sovellusongelmiin löytyy valmiita ratkaisuja.

2.3 Modernien kehysympäristöjen historia

2.3.1 Angular

AngularJS kehitettiin vuonna 2010 Googlen sisäisessä kehitystiimissä helpottamaan verkkokehitystä Googlen sovellusprojekteissa. Myöhemmin samana vuonna AngularJS julkaistiin avoimen lähdekoodin projektina kaikkien käytettäväksi. Sen tarjoamia ominaisuuksia olivat muun muassa kaksisuuntainen datan linkitys ja sovelluksen reitityksen hallinta yhden sivun sovelluksissa, ja se nousi nopeasti verkkokehittäjien suosioon. [6]

AngularJS:n jatkokehitys kuitenkin alkoi muuttua hankalaksi, kun kehittäjät alkoivat kaipaamaan yhä monimutkaisempia ominaisuuksia kehysympäristöiltä. AngularJS:ää ei ollut alun perin suunniteltu yleiseen käyttöön, ja lopulta sen kehittäjät päättivät uudelleenkirjoittaa sen kokonaan alusta. Google julkaisi vuonna 2016 Angularin eli Angular 2:n. Se ei ollut enää taaksepäin yhteensopiva vanhan AngularJS:n kanssa, ja vanhalla versiolla kirjoitettuja sovelluksia ei voinut päivittää uuteen versioon. Tämä suututti monet AngularJS:n käyttäjät, jotka päättivät tehdä jatkossa sovelluksensa eri kehysympäristöillä. [6]

Angular 2 ei ole versionvaihdoksen jälkeen enää yltänyt yhtä suureen suosioon kuin AngularJS:n aikoina. Stack Overflowin vuoden 2021 kehittäjäkyselyn mukaan Angularia käyttää vakituisesti 22 prosenttia vastaajista [2], ja se on neljäntenä Reactin, jQueryn ja Expressin jälkeen. Kirjoitushetkellä Angularia on ladattu Noden paketinhallintajärjestelmän (NPM) datan mukaan viikon aikana 2,8 miljoonaa kertaa [7]. NPM on yleisin tapa ladata Javascriptin paketteja ja lisäosia.

2.3.2 React

Reactin kehitti Facebookin ohjelmistoinsinööri Jordan Walke vuonna 2011, jolloin sitä käytettiin Facebookin uutisvirrassa. Se julkaistiin kaksi vuotta myöhemmin avoimena lähdekoodina muiden kehittäjien käyttöön. Sen suosio kasvoi, ja vuonna 2015 Netflix ilmoitti käyttävänsä Reactia käyttöliittymien tekoon. Samoihin aikoihin julkaistiin React Router, Redux- ja MobX-lisäosat, jotka hoitavat sovelluksen reitittämisen ja tilan hallinnan. [8]

Vuonna 2018 Reactista julkaistiin versio 16.8, jonka mukana tuli React Hooks -ominaisuus. React Hooks mahdollisti tilan säilömisen funktionaalisissa komponenteissa. Tätä ennen lähes kaikki Reactin komponentit kirjoitettiin luokkasyntaksilla, mutta Hooks mahdollisti yksinkertaisemman funktiosyntaksin käytön. Reactin kehittäjät suosittelevat funktiosyntaksiin vaihtamista hitaasti ajan myötä, ja molempia voi käyttää rinnakkain [9].

Nykyään React on suosituin kehysympäristö sovelluskehityksen alalla. Kehittäjäkyselyyn vastanneista 40 prosenttia vastasi käyttävänsä Reactia vakituisesti [2], ja NPM:n datan mukaan Reactia on kirjoitushetkellä ladattu viikon aikana yli 12 miljoonaa kertaa [1].

3. SVELTE

3.1 Svelten perusteet

Svelten kehitti alun perin Rich Harris vuonna 2016, ja se on sen jälkeen avattu avoimen lähdekoodin projektiksi, jolla on vapaaehtoisesti auttava kehitystiimi. Svelten versio 2.0 tuli valmiiksi vuonna 2018, ja vuotta myöhemmin vuonna 2019 Sveltestä julkaistiin versio 3.0, joka teki Svelten syntaktista yksinkertaisempaa.

Svelten suurin ero muihin kehysympäristöihin on, että se tekee suurimman osan kehysympäristön työstä käännösvaihessa. Svelte on käännettävä kieli, joten se ei sellaisenaan toimi ennen käännöstä. Svelten kääntäjä muuttaa sovelluksen koodin pieneksi optimoiduksi paketiksi, joka ei enää sovelluksen ajon aikana tarvitse ylimääräistä prosessointia kehysympäristön ylläpitämistä varten [10]. Sveltellä toteutetut sovellukset ovat tehokkaita ja pienikokoisia, mikä on hyödyksi erityisesti heikommilla mobiililaitteilla käytettävyyden takaamiseksi.

Sovelluskehitys Sveltellä perustuu Reactin tavoin komponentteihin, jotka ovat uudelleenkäytettäviä itsenäisiä osia sovelluksesta. Ohjelmassa 3.1 on esimerkki Sveltellä toteutetusta komponentista. Kaikki yksittäiseen komponenttiin liittyvä koodi kirjoitetaan yhden .svelte-tiedoston sisälle. Yhteen komponenttitiedostoon tulee HTML-määrittelyjen lisäksi toiminnallisuutta lisäävää Javascriptiä `<script>`-tunnisteen sisään sekä mahdollisesti CSS-tyylimäärittelyjä `<style>`-tunnisteen sisään.

Useimmat kehysympäristöt toimivat Javascriptin päällä, joten niiden syntaksi on suurimmaksi osaksi myös Javascriptiä. Svelte sen sijaan on ikään kuin oma ohjelmointikielensä, joka käännetään myöhemmin selaimen ymmärtämään muotoon. Svelten syntaksi näyttää hyvin paljon Javascriptiltä, mutta se toimii osittain eri tavoin kuin normaalisti. Esimerkiksi Sveltessä ei tarvitse välttää avainsanan `class` käyttöä HTML-elementin luokan määrittelyssä, vaikka se onkin varattu sana Javascriptissä.

```
1  <script>
2    export let name = "world";
3  </script>
4
5  <div>
6    <h1>Hello {name}!</h1>
7    <p>Esimerkkikappale</p>
8  </div>
9
10 <style>
11   h1 {
12     color: blue;
13     font-size: 3em;
14   }
15 </style>
```

Ohjelma 3.1. *Svelten esimerkkikomponentti*

Komponentille voi antaa alikomponentteja, ja komponentteja yhdistelemällä luodaan koko verkkosivun ulkoasu. Komponenttien tarvitsee usein kommunikoida keskenään, ja Svelte tarjoaa Reactin tavoin tavan siirtää dataa komponentista alikomponenteille propsien avulla. Props on komponentin sisäinen muuttuja, jonka arvon yläkomponentti voi vapaasti muuttaa. Sveltessä alikomponentin muuttuja määritellään propsiksi kirjoittamalla sen eteen `export`, kuten nähdään ohjelmassa 3.1. Tämän jälkeen yläkomponentti voi kutsua komponenttia esimerkiksi `<Component name="Programmer"/>` ja näin vaihtaa tekstiä, jota komponentti kirjoittaa ruudulle.

3.2 Svelten erikoispiirteet

3.2.1 Kehittäjäystävällisyys

Svelten käännösvaihe muuttaa Sveltellä kirjoitetun sovelluksen puhtaaksi Javascriptiksi. Käännösvaihe on hyödyllinen yksinkertaistamaan Svelten syntaksia ja vähentämään toistuvaa koodia, sillä Svelte voi käännöksen aikana hoitaa automaattisesti monimutkaisempien rakenteiden muodostamisen.

Yksi usein tarvittava osa verkkosovellusta on tiedon kaksisuuntainen linkitys. Sovelluksen kysyessä käyttäjältä nimeä, pitää käyttäjän selaimessa kirjoittama teksti saada päivittymään myös sovelluksen sisäiseen muuttujaan. Tähän Svelte tarjoaa keinon hoitaa linkitys automaattisesti käännöksen aikana lisäämällä koodiin `bind:` ennen elementin muuttujaa. Esimerkki kaksisuuntaisesta linkityksestä näkyy ohjelmassa 3.2. Linkityksen jälkeen komponentin teksti päivittyy automaattisesti, kun käyttäjä tekee muutoksia kirjoituskenttään.

```
1 <script>
2   let name = "world";
3 </script>
4
5 <h1>Hello {name}!</h1>
6 <input bind:value={name}/>
```

Ohjelma 3.2. *Svelten kaksisuuntainen linkitys*

Vastaavanlaisen komponentin luominen Reactilla vaatii enemmän koodia, josta osa on toisteista, eli se toistuu samanlaisena komponentista toiseen (englanninkielinen termi boiler-plate). Esimerkki Reactin vertauskomponentista on esitelty ohjelmassa 3.3. Ero Svelten ja Reactin välillä kuitenkin kapenee, mikäli käyttäjän syötettä ei voida tallentaa sellaisenaan, jolloin myös Sveltellä joutuu kirjoittamaan vastaavanlaisen rakenteen syötteen muokkaamiseksi ennen tallennusta.

```
1  export default App = () => {
2    const [name, setName] = useState("world");
3
4    const handleChange = (event) => {
5      setName(event.target.value);
6    }
7
8    return (
9      <>
10       <h1>Hello {name}!</h1>
11       <input value={name} onChange={handleChange} />
12     </>
13   );
14 };
```

Ohjelma 3.3. *Reactin kaksisuuntainen linkitys*

Svelten yksi tavoite on vähentää kirjoitettavan koodin määrää nopeuttaakseen sovelluksen kehittämistä, mutta myös vähentääkseen koodausvirheiden riskiä [11]. Koodausvirheiden uskotaan kasvavan eksponentiaalisesti koodimäärään nähden, joten koodin pitäminen lyhyenä ja selkeänä vähentää myös virheiden korjaamiseen kuluvaa aikaa.

3.2.2 Optimointi

Kehysympäristöjen tehtävänä on helpottaa sovelluskehittäjän työtä muun muassa automatisoimalla verkkosivun osien päivittämisen kun sovelluksen data muuttuu. Selaimet kuvastavat verkkosivun sisältöä dokumenttioliomallilla (DOM), jonka elementit on puumaisesti linkitetty toisiina. Monet kehysympäristöt, kuten myös React, ylläpitävät sovelluksen ajon aikana virtuaalista DOM:ia, mihin datan muutokset tehdään. Tätä muutettua

virtuaalista DOM:ia verrataan verkkosivun edelliseen tilaan, ja vertailun perusteella kehysympäristöt tekevät tarpeelliset muutokset oikeaan DOM:iin, jotta muutokset näkyvät myös sivun käyttäjälle. Virtuaalinen DOM on tullut suosituksi tavaksi ylläpitää käyttöliittymän tilaa, sillä muutoksien teko ja tilojen vertailu keskusmuistin sisällä on hyvin nopeata oikeisiin DOM-operaatioihin verrattuna. [12]

Svelten kehittäjä Rich Harris kuitenkin huomauttaa, että virtuaalisen DOM:in ylläpito ja muokkaus on ylimääräistä työtä prosessorille, sillä sen jälkeen pitää kuitenkin tehdä muutokset myös oikeaan DOM:iin ruudun päivittämiseksi [13]. Svelte ei käytä tilan hallintaan virtuaalista DOM:ia, vaan Svelten kääntäjä muokkaa käännöksen yhteydessä sovelluksen siten, että datan muutokset päivittyvät suoraan oikeaan DOM:iin. Svelten perusajatus on tehdä käännöksen aikana mahdollisimman paljon optimointia, jotta ajonaikainen suorituskyky olisi mahdollisimman hyvä. Svelten tehokkuusmittauksia käydään tarkemmin läpi luvussa 4.4.

Käännöksen aikana kääntäjä karsii mahdollisimman paljon käyttämätöntä koodia pois minimoiden lopullisen tiedostokoon. Kääntäjä poistaa turhien CSS-määrittelyjen ja komponenttien lisäksi myös Svelten omasta pohjakoodista käyttämättömät ominaisuudet. Mark Volkmannin mukaan pieneen tai keskikokoiseen sovellukseen jää keskimäärin noin 500 riviä Svelten koodia, mikä on hyvin vähän verrattuna muihin yleisiin kehysympäristöihin [14, luku 1.1.5].

Svelten optimointi toimii parhaiten moderneilla selaimilla, jotka tukevat ES6-standardia. Tämä tarkoittaa lähes kaikkia nykypäivän selaimia paitsi Internet Exploreria. Jos sovelluksen vaaditaan toimivan myös Internet Explorerilla, pitää sovellus transpiloida eli kääntää uudesta Javascriptistä vanhempaan, jotta Svelte voi käyttää ES6:n ominaisuuksia ilman selaimen natiivia tukea. Tämä transpilointi kasvattaa sovelluksen tiedostokokoa valtavasti. [15, luku 1]

3.2.3 Tyylimäärittelyjen kapselointi

Svelte-tiedoston sisäiset CSS-määrittelyt on automaattisesti rajoitettu muotoilemaan pelkästään kyseisen tiedoston komponenttia. Tyylimäärittelyt eivät vuoda ulos vaikuttamaan muihin komponentteihin eikä myöskään sen alikomponentteihin. Tämä vähentää ylimääräistä päänvaivaa, koska komponenteille ei tarvitse miettiä ainutkertaisia tunnisteita tai luokkanimiä tyylimäärittelyjen rajaamista varten.

Tyylimäärittelyt voi jakaa myös komponentin alikomponenteille käyttämällä `:global()`-tunnistetta määrittelyssä. Toinen tapa määritellä koko sovellukselle yhtenäisiä tyylisääntöjä on kirjoittaa ne erilliseen .css-tiedostoon sovelluksen juurikansioon. Globaaleita tyylisääntöjä luotaessa pitää kuitenkin muistaa, että Svelte käyttää tyylien kapselointiin käännöksen aikana annettuja luokkatunnisteita. Tästä syystä komponenttien sisäiset määrit-

telmät ovat spesifisempiä kuin globaalit luokkatunnisteiset määritelmät ja yliajavat ne, vaikka kehittäjä ei itse olisi käyttänyt luokkatunnisteita komponentin sisällä.

3.3 SvelteKit

SvelteKit on Svelten lisäosa, joka lisää korkeamman tason ominaisuuksia sovellukseen kuten reitityksen ja palvelinpuolen renderöinnin. SvelteKit rajattiin erilliseksi paketiksi, jotta Svelten ydinpaketti keskittyisi pelkästään komponenttien rakentamiseen ja pysyisi mahdollisimman pienenä ja helppokäyttöisenä.

Sekä Svelte että SvelteKit tukevat nykyään TypeScriptiä, jolla sovelluksen muuttujien tyypit voi määrittää vahvasti. Javascriptin kielessä muuttujat ovat heikosti määriteltyjä, eli niiden tietotyypit voi vaihtaa vapaasti miksi tahansa ajon aikana. Sovellus ei tiedä etukäteen, onko muuttujan sisällä esimerkiksi teksti vai objekti. Typescriptiä käyttämällä kehittäjä voi lukita muuttujan hyväksymään vain tiettyä tietotyyppiä, jolloin varmistutaan muuttujan pysyvän oikeanlaisena. Tämä muun muassa vähentää sovellusvirheiden riskiä ja helpottaa testausta.

SvelteKitin kehitys on kuitenkin vielä kirjoitushetkellä kesken, eikä tarkkaa julkaisupäivää vielä tiedetä. Sen beta-versiota voi jo alkaa opettelemaan ja käyttämään, mutta SvelteKit saattaa mahdollisesti vielä muuttua ennen ensimmäistä julkaisuversiota. Jos uuden sovelluksen tekee SvelteKitillä, on riskinä, että versiomuutos hajoittaa sovelluksen toiminnan, ja sen päivittäminen saattaa vaatia lisätyötä.

SvelteKit ei ollut ensimmäinen Svelten kehittäjien suunnittelema lisäosa, vaan sitä ennen he kehittivät Sapper-nimistä lisäosaa. Sapperin oli tarkoitus hoitaa verkkosovellusten vaativammat ominaisuudet. Sapper kuitenkin päätyi vaikeuksiin, kun sen pohjakoodi muuttui vaikeaksi jatkokehittää, ja Sapperin kehitys lopetettiin vuonna 2020 [16]. SvelteKit on Sapperin uudelleenkirjoitettu versio, joka toimii hiukan eri tavoin kuin edeltäjänsä. Tämä oli ikävä takaisku henkilöille, jotka olivat jo kirjoittaneet sovelluksensa Sapperin ennakkoversion avulla. Sapperin käytöstä oli myös kirjoitettu yksi oppikirja.

4. SVELTEN JA REACTIN VERTAILU

Tarkasteltaviksi aiheiksi valittiin opetusmateriaalit, tilanhallinnan menetelmät, muut lisäominaisuudet sekä kehysympäristöjen tehokkuus. Tehdyt havainnot ovat suurimmaksi kirjoittajan omia mielipiteitä molempien kehysympäristöjen opettelun ja tutkimisen ajalta, joskin erityisesti viimeisen osion tehokkuusvertailut perustuvat ulkopuoliseen vertailudataan.

Tässä osassa Svelteä verrataan vain Reactin funktiosyntaksiin. Reactin luokkasyntaksi on vaikeampaa käyttää ja oppia, ja tämän työn laajuus kasvaisi liikaa, mikäli Svelteä verrattaisiin molempiin kirjoitustyyleihin. Koodiesimerkeistä on myös poistettu `import`-lauseita tilan säästämiseksi.

4.1 Opetusmateriaalin laatu

Verkkokehityksen alalla on erityisen paljon opeteltavaa, ja uusia teknologioita tulee jatkuvasti lisää, joten on olennaista, että uusien teknologioiden opettelu on mahdollisimman nopeata ja yksinkertaista. Sovelluskehittäjät joutuvat harkitsemaan ajankäyttöä jatkuvasti, sillä uuden teknologian oppimiseen kulunut aika on ajanhukkaa, mikäli uuden teknologian hyödyt eivät korvaa siihen kulutettua aikaa. Tästä syystä sovelluskehittäjät saattavat valita teknisesti heikomman kehysympäristön projektia varten, mikäli projekti tulee nopeammin valmiiksi ja silti täyttää suoritusvaatimukset aiemmin tuttua kehysympäristöä käyttämällä.

Opiskelumateriaali itsessään ei ole osa kehysympäristöä, mutta se on yksi olennaisimmista asioista opettelun aikana. Opiskelumateriaalin laadulla on suora vaikutus kehysympäristön opettelun nopeuteen, joten siksi se on myös yksi tärkeä arviointikriteeri kehysympäristöjä vertaillessa. Kehysympäristöjen kehittäjät yleensä ylläpitävät ajantasaista opetusmateriaalia omilla verkkosivuillaan itseopiskelua varten, mutta usein kolmannet osapuolet tarjoavat myös muita opiskelumuotoja kuten oppikirjoja, videoita tai kursseja. Reactille on olemassa huomattavasti enemmän kolmansien osapuolten tekemää opetusmateriaalia Reactin suosion takia, mutta tässä osiossa arvioidaan vain kehysympäristöjen tekijöiden itse tekemiä materiaaleja.

Svelten sivuilla on interaktiivinen opetusmateriaali [17], joka opettaa Svelten perusteet käyttäen hyväkseen sivuun upotettua koodieditoria. Tämä on erityisen hyvä etenkin aloittelijoille, sillä tätä käyttämällä Svelten opettelun voi aloittaa ilman erillistä ohjelmointiym-

päristöä (IDE) tai muita valmisteluja. Tämä poistaa yhden kynnyksen opiskelun aloittamisesta, mikä helpottaa aloittelijan etenemistä.

Reactia on myös mahdollista opetella selaimella toimivan koodieditorin avulla, mutta tätä ominaisuutta ei ole lisätty Reactin omaan opetusmateriaaliin, vaan koodieditori on muiden palveluntarjoajien sivuilla. Esimerkiksi Reactin aloittelijoille suunnattu opetusmateriaali [18] linkittää CodePenin mallipohjaan, johon sovellus on tarkoitus kirjoittaa materiaalin neuvoja seuraamalla.

Svelten interaktiivinen opetusmateriaali on jaettu pieniin osapaloihin, joissa Svelten ominaisuudet opetetaan loogisessa järjestyksessä pala kerrallaan perusteista alkaen. Jokaisella palalla on oma kooditehtävä, jonka voi ratkaista tekstin viereisellä koodieditorilla. Koodiharjoituksen tekeminen teorian lukemisen ohella vahvistaa opitun tiedon omaksumista. Tehtävien mallivastauksen voi halutessaan nähdä nappia painamalla, mikä on hyödyksi oman vastauksen tarkistamisen lisäksi myös, jos opiskelija ei itse keksi oikeaa vastausta ja kaipaa lisäapua.

Reactin aloittelijoiden opetusmateriaali on kirjoitettu yhdelle pitkälle sivulle, joka opettaa Reactin perusteet ristinollapelin rakentamisen avulla. Materiaali ei kerro miten koko sovelluksen voisi kirjoittaa suoraan valmiiksi, vaan materiaalin aikana opetetaan peruskäsitteitä, ja välillä uudelleenkirjoitetaan aiempia koodilohkoja, kun uusia ominaisuuksia lisätään yksi kerrallaan. Materiaalin välissä on ajoittain linkkejä mallikoodiin, mistä näkee, miltä ristinollasovelluksen koodin pitäisi sillä hetkellä näyttää. Tämä on Svelten versioon verrattuna hankalampaa, sillä tarkistuskohtia on harvemmin ja mallikoodi on eri sivulla omasta koodista. Reactin oppimateriaalista saa kuitenkin kohtuullisen hyvän käsityksen Reactin käytöstä kokonaisen sovelluksen rakentamiseen.

React tarjoaa opetusmateriaalina myös toisen vaihtoehtoisen materiaalin, jossa Reactin ominaisuudet käydään läpi käsite kerrallaan syvällisemmin. Reactin molemmille opetusmateriaaleille on yhteistä, että kumpikin opettaa ensisijaisesti Reactin luokkasyntaksiin perustuvaa komponenttityyliä uudemman funktiosyntaksin sijaan. Luokkasyntaksi on erityisesti aloittelijoille vaikeampi tapa oppia Reactia, sillä sen käyttö vaatii paljon enemmän opettelua ja sen kirjoittaminen enemmän toistuvaa koodia. Reactin kehittäjät uskovat funktiosyntaksin olevan tulevaisuudessa ensisijainen tapa kirjoittaa Reactia [9], joten oppimateriaalin luokkasyntaksipainotteisuus vaikuttaa vanhentuneelta tavalta oppia Reactia.

Reactin dokumentaatio on saatavilla tällä hetkellä kuudellatoista eri kielellä, ja yhdeksän lisää on työn alla. Reactin materiaali ei kuitenkaan ole saatavilla eikä suunnitteilla suomeksi [19]. Osasyy tälle saattaa olla Helsingin Yliopiston tarjoama ilmainen verkkokurssi, jonka kattava oppimateriaali on avoinna kaikille suomeksi verkossa [20]. Svelten opetusmateriaali on tällä hetkellä vain englanniksi.

4.2 Tilan hallinta

Yksinkertaisissa sovelluksissa datan pitäminen yläkomponentissa ja sen jakaminen alikomponenteille propsien avulla saattaa olla riittävä ratkaisu, mutta sovelluksen kasvaessa datan pitäisi kulkea jokaisen komponentin läpi ja kokonaistilanteen hallinta hankaloituu. Sekä Svelte että React tarjoavat tähän kehittyneempiä ratkaisuita.

4.2.1 Konteksti

Svelte ja React molemmat sisältävät kontekstiominaisuuden, jonka avulla komponentin datan voi julkaista sen kaikille alikomponenteille riippumatta siitä, kuinka syvällä ne ovat komponenttipuussa. Käyttämällä kontekstia sovelluksen juurikomponentissa, saa datan globaalisti käyttöön kaikkiin komponentteihin. Kontekstin sisältämä data voi olla mitä tahansa.

Sveltessä konteksti kirjoitetaan suoraan yläkomponentin sisälle käyttäen `setContext`-funktiota. Funktiolle annetaan parametrina tunnisteavain sekä jaettava data. Tunnisteavain voi olla mitä vain, ohjelmassa 4.1 on käytetty tekstiä tunnisteena, mutta myös objekti tai muu tietotyyppi toimii. Alikomponentit saavat kontekstin sisältämän datan ulos kutsumalla `getContext`-funktiota oikealla tunnisteavaimella.

```
1  // tiedosto app.svelte
2  <script>
3    svelte.setContext("text", "Hello World!");
4  </script>
5
6  <Header/>
7
8  // tiedosto header.svelte
9  <script>
10   const text = svelte.getContext("text");
11 </script>
12
13 <h1>{text}</h1>
```

Ohjelma 4.1. *Esimerkki kontekstista Sveltessä*

Reactissa konteksti pitää kirjoittaa erilliseen tiedostoon, josta se importataan kaikkiin sitä käyttäviin komponentteihin. Konteksti luodaan `createContext`-funktiolla, jonka jälkeen se asetetaan yläkomponenttiin käyttämällä `Provider`-elementtiä paluuarvon ympärillä. Alikomponentit saavat kontekstin datan käyttöön kutsumalla `useContext`-funktiota käyttäen parametrina alkuperäistä kontekstioliota. Ohjelmassa 4.2 on esimerkki Reactin kontekstin käytöstä.

```
1  // tiedosto context.js
2  export const TextContext = react.createContext();
3
4  // tiedosto App.js
5  export default function App() {
6    return (
7      <TextContext.Provider value="Hello World!">
8        <Header/>
9      </TextContext.Provider>
10   );
11 };
12
13 // tiedosto Header.js
14 export default function Header() {
15   const text = react.useContext(TextContext);
16   return <h1>{text}</h1>;
17 };
```

Ohjelma 4.2. *Esimerkki kontekstista Reactissa*

Ohjelmista 4.1 ja 4.2 näkee, että Svelten koodi on lyhyempi ja yksinkertaisempi. Reactin esimerkki olisi vieläkin pitempi, mikäli siinä olisi ollut mukana pakolliset `import`-lauseet. Svelten mahdollisuus tunnistaa oikea konteksti ilman referenssiä ulkoiseen tiedostoon vähentää toisteista koodia. Sveltessä on kuitenkin mahdollista sekoittaa kaksi eri kontekstia keskenään, mikäli ohjelmoija käyttää vahingossa samaa tunnisteavainta molemmille. Tämän voi välttää käyttämällä Reactin tavoin ulkoista tiedostoa, josta haetaan tunnisteavaimeksi objekti. Objektit ovat Javascriptissä yhtäläisiä pelkästään itseensä, eikä sekaantumisen riskiä ole.

4.2.2 Svelte Store ja React Recoil

Store on toinen tapa hallita sovelluksen tilaa Sveltessä. Se on kontekstista poiketen aidosti globaali koko sovellukselle riippumatta komponenttien sijainnista. Storen arvoa voi muuttaa mistä tahansa komponentista, ja sen muutokset päivittyvät automaattisesti muihin Storen arvoa käyttäviin komponentteihin.

Reactissa ei ole sisäänrakennettuna samanlaista ominaisuutta, mutta kolmannet osapuolet ovat luoneet Reactille useita eri tilanhallinnan lisäosia. Yksi niistä on Recoil, joka on

yksi uudempia tilanhallinnan työkaluja. Recoil toimii lähes samoin tavoin kuin Svelten Store, ja siksi se valittiin tähän lukuun Storen vertauskohteeksi.

Sveltessä Store kirjoitetaan erilliseen tiedostoon, josta se importoidaan sitä käyttäviin komponentteihin. Store luodaan `writable`-komennolla, jolle annetaan parametriksi alustusarvo. Store on objekti, joka sisältää `subscribe`-, `set`- ja `update`-metodit, joiden avulla Storen arvoa voi seurata ja päivittää. Storen päivityksiin ei kehittäjän itse tarvitse ilmoittautua `subscriben` avulla, sillä kutsumalla Storea dollarimerkin kanssa Svelte antaa Storen arvon suoraan ja hoitaa päivityksiin ilmoittautumiset ja katkaisun automaattisesti. `Update`-komennolla Storen arvo muuttuu ja samalla päivittyy kaikkialle, missä sitä käytetään. Ohjelmassa 4.3 on esimerkki tätä tekniikkaa käyttävästä komponentista.

Svelten `derived` on erikoistapaus Storesta, jonka arvo riippuu toisen Storen arvosta. Sen arvoa ei voi manuaalisesti muuttaa, ja se päivittyy automaattisesti kun sen seuraama Store muuttuu. Ohjelmassa 4.3 `countSquared` laskee ajantasaisesti `count`-Storen luvun neliön.

```
1  // tiedosto stores.js
2  export const count = writable(0);
3
4  export const countSquared = derived(
5      count,
6      $count => $count * $count
7  );
8
9  // tiedosto store.svelte
10 <script>
11   import { count, countSquared } from './stores.js';
12 </script>
13
14 <h1>{$count} squared is {$countSquared}</h1>
15 <button on:click={() => count.update(n => n - 1)}> -1 </button>
16 <button on:click={() => count.update(n => n + 1)}> +1 </button>
```

Ohjelma 4.3. *Svelte Storen esimerkki*

Reactin Recoilissa tilaa kutsutaan atomiksi, ja se myös tallennetaan erilliseen tiedostoon. Atomin sisältämän arvon saa kutsumalla sitä esimerkiksi `useRecoilState`-funktiolla, ja sen syntaksi on lähes identtinen Reactin hooksien `useState`-syntaksiin. Tämän jälkeen atomin arvoa voi muokata vapaasti, ja Recoil hoitaa kaikkien sitä käyttävien komponenttien päivittämisen.

Toisesta atomista riippuvainen atomi on nimeltään `selector`, joka on täysin vastaava kuin Svelten `derived`. Ohjelmassa 4.4 `countSquared` automaattisesti laskee `count`-atomin neliön aina sen muuttuessa.

React Recoil vaatii yhden ylimääräisen osan Svelten Storeen verrattuna. Recoil ei toimi, ellei sen ylimmän tason komponenttia ole kääritty `<RecoilRoot>`-elementin sisälle. Tämä toimii kontekstin tavoin sijaintina, jonne Recoilin tila tallennetaan ja josta se jaetaan kaikille alikomponenteille. Esimerkkiohjelmassa 4.4 `<RecoilRoot>` on kääritty koko sovelluksen ympärille index.js-tiedostossa.

```
1  // tiedosto index.js
2  ReactDOM.render(
3      <RecoilRoot>
4        <App />
5      </RecoilRoot>,
6    document.getElementById('root')
7  );
8
9  // tiedosto atoms.js
10 export const countState = atom({
11   key: "count",
12   default: 0,
13 });
14
15 export const countSquaredState = selector({
16   key: "countSquared",
17   get: ({ get }) => {
18     const count = get(countState);
19     return count * count;
20   },
21 });
22
23 // tiedosto app.js
24 export default function App() {
25   const [count, setCount] = useRecoilState(countState);
26   const countSquared = useRecoilValue(countSquaredState);
27
28   return (
29     <>
30       <h1>{count} squared is {countSquared}</h1>
31       <button onClick={() => setCount(count - 1)}>-1</button>
32       <button onClick={() => setCount(count + 1)}>+1</button>
33     </>
34   );
35 };
```

Ohjelma 4.4. *React Recoilin esimerkki*

Svelten ja Reactin esimerkkiohjelmia vertaamalla huomataan, että Svelten koodi on tässä tapauksessa puolet lyhyempi Reactiin verrattuna. Sveltessä koodin olisi voinut tiivistää vieläkin pienempään jos Storen muuttamiseen olisi nappien sijasta käyttänyt `<input`

`type="number"bind:value={$count}/>`, jolloin `bind`-avainsana olisi automaattisesti tallentanut käyttäjän tekemät muutokset Storeen.

4.2.3 React Redux

Reactilla on useita muitakin tilanhallinnan lisäosia, kuten esimerkiksi React Redux, joka on ollut suosituin tilanhallinnan lisäosa jo usean vuoden ajan. Redux on huomattavasti monimutkaisempi käyttää Recoiliin verrattuna, mutta Redux auttaa varmistamaan tilan pysyvän halutunlaisena. Redux estää tilan muuttamisen mielivaltaisesti, ja tilaa voi muuttaa vain etukäteen suunniteltuilla metodeilla. Nämä metodit ovat puhtaasti funktionaalisia, eli ne toimivat aina identtisesti samoilla parametriarvoilla. Tämä helpottaa tilanhallinnan testaamista huomattavasti, ja estää huolimattomien sivuvaikutuksien tapahtumista.

Reduxia vastaavaa tilan hallintaa ei ole Sveltessä, vaan oletusarvoisesti kehittäjällä on aina vapaat kädet muokata sovelluksen tilaa miten vain. Kehittäjien pitää itse sopia omat säännöt tilan muokkaustavoista tai luoda oma Store-olio, joka ei anna komponenttien kutsua suoraan `set`-komentoa, vaan sisältää itse koodattuja metodeita, joilla tilaa voi muokata rajoitetusti. Tämä on kuitenkin kehittäjän omalla vastuulla toteuttaa oikein, eivätkä nämä metodit ole yhtä helposti testattavissa.

4.3 Lisäominaisuudet

Omaisuuksien määrässä React on ylivertainen Svelteen verrattuna. Reactia on kehitetty kymmenen vuotta, ja Reactin takana on Facebook, jolla on varaa ja resursseja jatkaa sen kehitystä. Svelte sen sijaan on viisi vuotta sitten yksityishenkilön aloittama avoimen lähdekoodin projekti, jonka jatkokehitys on vapaaehtoistyön varassa. React ja sen lisäosat kattavat lähes kaikki käyttötarpeet, kun taas Svelte hoitaa lähinnä ydinasiat.

Reactin suosio on tuonut mukanaan valtavan määrän kolmansien osapuolten lisäosia ja valmiita integrointeja muihin teknologioihin. Verkosta löytyy myös tuhansittain valmiita Reactin komponentteja, joita yhdistelemällä voi merkittävästi vähentää työmäärää omassa projektissa. Svelteä käytettäessä kehittäjät joutuvat usein tekemään komponentit itse vaihtoehtojen puutteen takia.

Toisaalta Reactin valtava ominaisuusmäärä voi myös olla ongelma, sillä kaikkien ominaisuuksien opettelu voi viedä hyvin paljon aikaa. Eri vaihtoehtojen määrä tekee valinnoista vaikeaa, ja jokainen uusi lisäosa kasvattaa opeteltavan määrää. Reactin paketeista koostuva rakenne on sekä hyöty että haitta, sillä sen avulla voi tehdä lähes mitä vain, mutta se samalla kasvattaa kehittäjien omaa valintojen vastuuta.

4.4 Tehokkuus

Kehysympäristöjen tehokkuutta on vaikeaa arvioida yksiselitteisesti, sillä tulokset riippuvat hyvin paljon koodin laadusta ja kirjoitustyylistä. Koodin kehittäjä saattaa panostaa koodin luettavuuteen koodivirheiden vähentämiseksi, mikä taas saattaa johtaa hitaampaan koodiin. Päinvastoin äärimmilleen optimoitu koodi saattaa olla hyvin vaikeaa ymmärtää tai jatkokehittää. Siksi tehokkuusvertailuja tutkittaessa pitää muistaa, että tulokset ovat lähinnä suuntaa-antavia, ja omalla koodaustyylillä on erittäin suuri vaikutus sovelluksen tehokkuuteen.

Tehokkuusvertailun dataksi valittiin kaksi alan mittavaa vertailua, Web Frameworks Benchmark sekä RealWorld Comparison 2020, joissa kummassakin oli mukana yli sata henkilöä. Tutkimuksien koko ja osallistujien määrä vähentävät yksittäisistä koodaajista johtuvia eroja, ja täten tulosten voi olettaa olevan suurin piirtein todenmukaisia. Tutkimukset vertasivat eri asoita ja antavat eri näkökulmia kehysympäristöjen vertailuun.

4.4.1 Web Frameworks Benchmark

Web Frameworks Benchmark tutkii noin 60 eri kehysympäristön ja Javascript-teknologian tehokkuutta. Aika-ajoin julkaistaan uusia vertaustuloksia, joiden välillä sekä teknologioiden versiot että testaukseen käytetty Chrome-selaimen versio on päivitetty ajan tasalle. Testit voi tehdä myös omalla koneella kopioimalla testien pohjakoodi projektin GitHub-sivulta [21].

Testit mittaavat toteutuksien tehokkuutta suorittamalla tuhansia toistoja yksinkertaisen listan päivittämiseen. Toistoihin kulunut aika mitataan millisekunteina, jonka jälkeen aikoja verrataan kategorian parhaaseen, josta saadaan normalisoitu tulos. Jokaisen toteutuksen normalisoidut tulokset lasketaan lopuksi yhteen geometriseksi keskiarvoksi, joka on ikään kuin kokonaisarvosana käytetylle teknologialle. Geometrisessa keskiarvossa pienempi tulos on parempi.

Yksittäisten HTML-metodien testaamisen lisäksi testit mittaavat muistin käyttöä ja käynnistymisnopeutta, mutta nämä tulokset eivät ole yhtä oleellisia tämän työn kannalta. Käynnistymisnopeudesta saadaan hyödyllisempiä tuloksia seuraavassa osiossa, jossa kehysympäristöjä verrataan tekemällä isompi sovellus.

Kirjoitushetkellä uusimman testin tulokset on nähtävillä sen omilla verkkosivuilla [22]. Svelten lisäksi mittauksessa oli yli kymmenen eri Reactin versiota, joissa oli käytetty eri tekniikoita muun muassa tilan hallintaan. Tämän työn kannalta oleellisimmat tulokset on kerätty taulukkoon 4.1, johon valittiin kolme eri Reactin versiota. Vertailuun valittiin kaksi Hookseihin perustuvaa toteutusta sekä yksi Recoil-toteutus, sillä näistä tekniikoista on puhuttu jo aiemmin tämän työn aikana. Vertailuun otettiin myös puhtaalla Javascriptillä

kirjoitettu toteutus, sillä se oli kaikista testeistä paras toteutus. Puhdas Javascript on paras siksi, että se on käsin optimoitu mahdollisimman tehokkaaksi, ja siinä ei ole kehysympäristön lisäämää ylimääräistä taakkaa.

Tuloksista huomataan, että Svelte oli Reactia parempi kaikissa kategorioissa, joskin vain hyvin niukasti muutamissa tilanteissa. React jäi jälkeen erityisesti yhden rivin valitsemisessa ja kahden rivin vaihtamisessa. Kaikki Reactin toteutukset olivat yli kuusinkertaisesti parasta tulosta hitaampia kahden rivin vaihtamisessa, jonka takia niiden keskiarvotulokset olivat 1,71 tai yli. Svelten vastaava kokonaistulos oli 1,23, eli noin 23 prosenttia puhdasta Javascriptiä hitaampi.

Taulukko 4.1. *Web Frameworks Benchmark - Svelten ja Reactin tulokset*

	Vanillajs	Svelte	React Redux-Hooks	React Hooks	React Recoil
Luo 1 000 riviä (ms)	83,0	99,1	123,1	116,7	117,3
(vertaus parhaaseen)	1,00	1,19	1,48	1,41	1,41
Päivitä kaikki rivit (ms)	79,8	96,8	103,7	97,5	103,5
	1,00	1,21	1,30	1,22	1,30
Päivitä joka kymmenes rivi (ms)	155,3	168,3	200,1	190,8	224,5
	1,00	1,08	1,29	1,23	1,45
Valitse tietty rivi (ms)	19,0	31,2	41,7	67,9	81,1
	1,00	1,64	2,19	3,56	4,26
Vaihda kaksi riviä (ms)	49,2	52,9	330,5	325,7	329,8
	1,00	1,07	6,71	6,61	6,70
Poista rivi (ms)	20,8	21,3	21,8	21,9	28,1
	1,00	1,03	1,05	1,05	1,35
Luo 10 000 riviä (ms)	790,3	1039,1	1352,0	1297,7	1290,4
	1,00	1,31	1,71	1,64	1,63
Lisää 1 000 riviä (ms)	177,2	213,6	219,6	231,0	223,5
	1,00	1,21	1,24	1,30	1,26
Poista kaikki rivit (ms)	49,9	71,3	79,1	71,4	87,5
	1,00	1,43	1,58	1,43	1,75
Vertauksien geom. keskiarvo	1,00	1,23	1,71	1,76	1,93

4.4.2 RealWorld Comparison 2020

RealWorld Comparison vertaili kehysympäristöjä korkeamman tason näkökulmasta rakentamalla saman keskikokoisen projektin jokaisella kehysympäristöllä. Kaikkien kehysympäristöjen piti täyttää samat tekniset vaatimukset sekä toimia identtisesti, ja koodin laatuvaatimus oli olla alan ammatilaisen tekemää tai hyväksymää. Viimeisin testi on vii-

me vuodelta, jolloin kehysympäristöjä verrattiin keskenään suorituskyvyn, sovelluskoon ja koodirivimäärän perusteella [23]. Taulukkoon 4.2 on kerätty kaikki Svelten ja Reactin tulokset.

Suorityskykymittauksessa käytettiin Chromen Lighthouse Audit -työkalua, joka antaa pistearvion 0 ja 100 väliltä, korkeampi luku parempi. Sisäisesti Lighthouse laskee tuloksen mittaamalla kuutta eri aspektia, jotka yhdessä mittaavat toteutuksen nopeutta saada sovellus valmiiksi käynnityksen aikana. Aspekteja ovat muun muassa ensimmäiseen piirtämiseen kulunut aika, sekä kuinka kauan kestää, kunnes sovellus alkaa reagoimaan käyttäjän komentoihin. Lighthouse laskee pisteet vertaamalla tuloksia todellisista verkkosivuista kerättyyn vertailudataan, jossa tehokkaimpien sivujen tulokset asetetaan vastaamaan 99 pisteen tulosta.

Svelte oli suorituskykymittauksen yksi kolmesta parhaasta kehysympäristöstä, joista kaikki kolme sai 99.0 pisteen tuloksen. Reactin molemmat toteutukset jäivät selkeästi jälkeen. Lighthousen kehittäjät suosittelevat kehittäjien pyrkivän vähintään 90 pisteen tulokseen, jotta käyttäjäkokemus olisi mahdollisimman miellyttävä.

Toisessa mittauksessa verrattiin toteutuksien yhteenlaskettua tiedostokokoa. Isompi sovellus kestää kauemmin ladata, jolla on merkitystä erityisesti jos verkkoyhteys on hidas. Tässä kategoriassa Svelte oli testien paras, joka pystyi käännöksen aikaisen optimoinnin avulla karsimaan sovelluksen 15 kB:n kokoiseksi. Reactin sovelluksien koko oli moninkertaisesti enemmän, mutta silti noin keskitasoa kaikkien kehysympäristöjen kesken.

Kolmannessa mittauksessa verrattiin koko sovelluksen koodirivimäärää. Koodiriveihin ei laskettu tyhjiä rivejä tai kommentteja. Tämä tulos on suuntaa-antava merkki kehysympäristöjen kehittäjäystävällisyydestä, sillä lyhyempi koodi on helpompi kirjoittaa ja todennäköisesti sisältää vähemmän koodivirheitä. Testissä Svelte oli kolmanneksi paras, ja Svelten toteutus oli noin puolet lyhyempi Reactin toteutuksiin verrattuna.

Taulukko 4.2. *RealWorld Comparison 2020 - Svelten ja Reactin tulokset*

Kehysympäristö	Suorituskyky (pistearvio)	Latauskoko (KB)	Koodirivimäärä
Svelte	99,0	15,0	1057
React + MobX	82,0	97,2	1917
React + Redux	67,0	193,0	2050

Tuloksia tutkiessa pitää muistaa, että tämä mittaus on jo hiukan vanhentunut, eikä vastaavaa mittausta ole suoritettu tänä vuonna. Erityisesti suorituskykytulokset olisivat todennäköisesti muuttuneet vuoden aikana, sillä päivitykset kehysympäristöihin ja uudemmat vertailudatat voivat molemmat vaikuttaa pisteisiin huomattavasti.

5. POHDINTA

Svelte on hyvä vaihtoehto pienen tai keskikokoisen sovelluksen rakentamiseen. Svelte hoitaa verkkosovelluksen rakentamisen perusasiat paremmin kuin React, mutta Sveltellä ei ole samanlaista kolmansien osapuolien tukea kuin Reactilla. Reactille löytyy paljon valmiita komponentteja, kuten kalentereita tai visuaalisia kaavioita, joita käyttämällä voi säästyä isolta vaivalta. Sveltellä joutuu todennäköisemmin tekemään ominaisuuksia itse.

Svelten tulevaisuudesta ei ole varmuutta, sillä sen jatkokehitys riippuu lähes täysin vapaaehtoisten kehittäjien avusta. Svelte on avoimen lähdekoodin projekti, jolla ei ole isomman yrityksen tukea, kuten Reactilla tai Angularilla on. Tästä syystä Svelten kehitystiimi ei pysty jatkokehittämään kehysympäristöään yhtä tasaisesti tai varmasti kuin React. Toisaalta Vue on päässyt kolmanneksi suurimmaksi kehysympäristöksi, vaikka se on myös avoimen lähdekoodin projekti.

Sveltellä on potentiaalia nousta suurempaan suosioon tulevaisuudessa, sillä sen ominaisuudet ovat hyödyllisiä ja kehittäjillä on mielipidemittauksen mukaan kiinnostusta Svelteen. Svelte kuitenkin tarvitsee vielä jonkin ulkopuolisen sysäyksen, jotta se kasvaisi yhtä suosituksi kuin React, Angular tai Vue. SvelteKitin 1.0-version julkaisu saattaisi olla merkittävä merkkipaalu, joka voisi mahdollisesti aiheuttaa seuraavan kasvupiikin suosiossa.

Svelte voisi soveltua Reactia paremmin verkkokehityksen opiskelijan ensimmäiseksi kehysympäristöksi, sillä Svelten opettelu ja käyttö on huomattavasti Reactia helpompaa. Svelteä käyttämällä saisi kevyemmän oppimiskynnyksen verkkokehityksen alalle, ja moni Svelten kautta opittu tekniikka toimii vastaavalla tavalla Reactissa, kuten huomattiin luvussa 4.2, jossa vertailtiin tilan hallintaa. Molempien kehysympäristöjen Javascript-painotteinen syntaksi myös helpottaa siirtymistä kehysympäristöstä toiseen.

Svelteä ei kuitenkaan kannata opetella ainoana kehysympäristönä, vaan sen rinnalle on hyvä oppia myös React tai jokin muu kehysympäristö. Svelteä käyttäviä yrityksiä on vielä hyvin vähän, ja työpaikan löytäminen vain Svelten avulla olisi vaikeaa.

Svelte soveltuu hyvin esimerkiksi sulautettujen järjestelmien selainpohjaisiin käyttöliittymiin sen erinomaisen tehokkuuden ansiosta. Svelteä on jo tällä hetkellä käytetty muun muassa halvassa massatuotetussa maksuvälinelaitteessa ja smart-TV:n käyttöliittymässä [24]. Svelten avulla sovellus reagoi nopeasti käyttäjän syötteisiin heikoillakin laitteilla, mikä on tärkeää käyttäjäkokemuksen kannalta. C-pohjaiset kielet olisivat sitäkin tehok-

kaampia, mutta Sveltellä voi tehdä sovelluksia käyttämällä selaimen valmista toiminta-runkoa ja Javascriptin helpompaa syntaksia, mitkä nopeuttavat sovelluskehitystä.

Mikäli Svelte ei itse nouse suosituksi kehysympäristöksi, todennäköisesti jokin toinen kehysympäristö ottaa Svelten perusideat ja toteuttaa ne paremmin. Käännökseen perustuva optimointi ja ohjelmointisyntaksin helpottaminen ovat molemmat hyviä ideoita, jotka todennäköisesti tulevat yleistymään tulevissa kehysympäristöissä.

6. YHTEENVETO

Tämän työn tarkoituksena oli tutkia Svelten ominaisuuksia ja miettiä milloin Svelte sopisi parhaiten sovellusprojektin kehysympäristöksi. Svelte markkinoi itseään tehokkaana ja kehittäjäystävällisenä kehysympäristönä, ja ulkopuoliset mittaukset sekä kirjoittajan omat havainnot tukivat tätä väitöstä. Svelten käännöksen aikainen optimointi vähentää ajonaikaisen prosessoinnin tarvetta ja karsii sovelluksesta pois kaiken käyttämättömän koodin sovelluskoon pienentämiseksi. Tässä työssä tarkastelujen mittauksien mukaan Sveltellä rakennetut sovellukset olivat nopeampia ja pienempiä kuin Reactilla tehdyt vastineet.

Svelten koodauskieli on tuttua kehittäjille, jotka osaavat ennestään Javascriptiä, HTML:ää ja CSS:ää. Svelte kuitenkin käyttää Javascriptin syntaksia hiukan eri tavalla kuin se normaalisti toimisi, sillä Svelte muuttaa käännöksen aikana koodin selaimen ymmärtämään muotoon. Tämä erotus koodauskielen ja lopullisen ohjelmakoodin välillä mahdollistaa Svelten yksinkertaistamaan koodaukseen käytettävää syntaksia ja vähentämään toisteista koodia.

Svelten ja Reactin koodiesimerkkejä vertaillessa huomattiin, että Svelten koodi oli lyhyempää ja helpompi ymmärtää. Yksinkertaisissa tilanteissa Svelten koodi oli parhaimmillaan yli puolet lyhyempi. Ero Svelten ja Reactin välillä kaveni, mikäli ominaisuus muuttui monimutkaiseksi ja vaati erityiskäsittelyä. Työssä käytetyn tutkimuksen mukaan kokonaisia sovelluksia verrattaessa Svelten toteutus oli lähes puolet lyhyempi koodirivimäärässä Reactin toteutukseen verrattuna.

Svelte on hyvä vaihtoehto sovellusprojekteihin, joissa kehittäjät suunnittelevat tekevänsä koko sovelluksen itse. Svelte hoitaa kehysympäristöjen perusasiat paremmin kuin React, mutta Sveltelle ei ole yhtä kattavaa kolmansien osapuolten tukea kuin Reactille. Tästä syystä Sveltelle on vaikeampaa löytää valmiita komponentteja tai lisäominaisuuksia kuin Reactille.

Tämä työ keskittyi Svelten ydinpaketin toimintaan. Jatkotutkimuksen kannalta SvelteKitin lisäominaisuuksien tutkiminen sekä sen vahvuuksien että heikkouksien vertailu olisi hyvä aihe, sillä erityisesti isot sovellusprojektit vaativat monimutkaisempia ominaisuuksia, joita ei ole Svelten ydinpaketissa. Tämän työn perusteella ei vielä pysty kattavasti päättämään, onko Svelte hyvä kehysympäristö kaikille sovellusprojekteille.

LÄHTEET

[1] *NPM - React*. URL: https://www.npmjs.com/package/react (viitattu 13. 11. 2021).

[2] *Stack Overflow Developer Survey 2021*. 2. elokuuta 2021. URL: https://insights.stackoverflow.com/survey/2021.

[3] Dean, J. *Web Programming with HTML5, CSS and Javascript*. 1. tammikuuta 2018. URL: https://learning.oreilly.com/library/view/web-programming-with/9781284091809/.

[4] *caniuse.com - ECMAScript 2015 (ES6)*. URL: https://caniuse.com/es6 (viitattu 24. 10. 2021).

[5] Wanyoike, M. *History of front-end frameworks - LogRocket Blog*. 16. lokakuuta 2018. URL: https://blog.logrocket.com/history-of-frontend-frameworks.

[6] Gavigan, D. *The History of Angular - Medium*. 3. huhtikuuta 2018. URL: https://medium.com/the-startup-lab-blog/the-history-of-angular-3e36f7e828c7.

[7] *NPM - @angular/core*. URL: https://www.npmjs.com/package/@angular/core (viitattu 13. 11. 2021).

[8] Porcello, E. ja Banks, A. *Learning React, 2nd Edition*. 1. kesäkuuta 2020. URL: https://learning.oreilly.com/library/view/learning-react-2nd/9781492051718.

[9] *Hooks FAQ - React*. URL: https://reactjs.org/docs/hooks-faq.html#should-i-use-hooks-classes-or-a-mix-of-both (viitattu 03. 11. 2021).

[10] Segala, A. *Svelte 3 Up and Running*. Packt Publishing, 2020. 168 p. URL: https://learning.oreilly.com/library/view/svelte-3-up/9781839213625/.

[11] Harris, R. *Write less code - Svelte*. 20. huhtikuuta 2019. URL: https://svelte.dev/blog/write-less-code (viitattu 11. 11. 2021).

[12] Elrom, E. *React and Libraries: Your Complete Guide to the React Ecosystem*. 1. maaliskuuta 2021. URL: https://learning.oreilly.com/library/view/react-and-libraries/9781484266960.

[13] Harris, R. *Virtual DOM is pure overhead*. 27. joulukuuta 2018. URL: https://svelte.dev/blog/virtual-dom-is-pure-overhead.

[14] Volkmann, M. *Svelte and Sapper in Action*. Manning Publications, 2020. 456 p. URL: https://learning.oreilly.com/library/view/svelte-and-sapper/9781617297946/.

[15] Libby, A. *Practical Svelte: Create Performant Applications with the Svelte Component Framework*. 1. lokakuuta 2021. URL: https://learning.oreilly.com/library/view/practical-svelte-create/9781484273746/.

[16] *Rich Harris: Futuristic Web Development*. 19. lokakuuta 2020. URL: https://www.youtube.com/watch?v=qSfdtmcZ4d0.

[17] *Svelte Tutorial*. URL: https://svelte.dev/tutorial/basics (viitattu 02. 11. 2021).

[18] *React Tutorial*. URL: https://reactjs.org/tutorial/tutorial.html (viitattu 03. 11. 2021).

[19] *React - Languages*. URL: https://reactjs.org/languages (viitattu 03. 11. 2021).

[20] *Full Stack open 2021*. URL: https://fullstackopen.com/ (viitattu 03. 11. 2021).

[21] *GitHub - JS Framework Benchmark*. URL: https://github.com/krausest/js-framework-benchmark (viitattu 14. 11. 2021).

[22] *JS frameworks benchmark results*. URL: https://krausest.github.io/js-framework-benchmark/2021/table_chrome_96.0.4664.45.html (viitattu 14. 11. 2021).

[23] *A RealWorld Comparison of Front-End Frameworks 2020*. URL: https://medium.com/dailyjs/a-realworld-comparison-of-front-end-frameworks-2020-4e50655fe4c1 (viitattu 18. 10. 2021).

[24] *Talking Svelte with Rich Harris - ShopTalk*. URL: https://shoptalkshow.com/349 (viitattu 19. 10. 2021).