Opinnäytetyö (AMK)

Tietojenkäsittely

2021

Karel Haapasalo

JAVASCRIPTIN SOVELLUSKEHYS VERKKOSOVELLUKSEN KÄYTTÖLIITTYMÄN TOTEUTUKSESSA

– React vs. Svelte

TURKU AMK

TURKU UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tietojenkäsittely

2021 | 38 sivua

Karel Haapasalo

JAVASCRIPTIN SOVELLUSKEHYS VERKKOSOVELLUKSEN KÄYTTÖLIITTYMÄN TOTEUTUKSESSA

- React vs. Svelte

Verkkokehityksessä käytettävän JavaScript-ohjelmointikielen sovelluskehysten ja kirjastojen määrä on kasvanut kovaa vauhtia. Kehyksiä löytyy niin käyttöliittymien kehitykseen kuin asiakasohjelman ja jopa palvelinpuolen ohjelmointiin. Sovelluskehykset tarjoavat valmiita toiminnallisuuksia ja luettavampaa koodia, mikä tekee kehittämisestä nopeampaa ja sulavampaa.

Opinnäytetyössä tarkasteltiin kahta JavaScriptin sovelluskehystä, joita käytetään erityisesti käyttöliittymien tekemiseen. Tavoitteena oli selvittää onko Svelte varteenotettava vaihtoehto Reactille. Tavoitteena oli myös rakentaa kaksi versiota verkkosovelluksen käyttöliittymästä käyttäen React- ja Svelte-kehyksiä. Rakentamisen yhteydessä katsottiin kehysten tapoja, joilla komponenttilähtöisiä käyttöliittymiä voidaan rakentaa. Valmiita käyttöliittymiä vertailtiin systemaattisesti keskenään. Lisäksi katsottiin kehitysyhteisön vaikutusta kehysten elinvoimaisuuteen.

Työn lopputuloksena onnistuttiin rakentamaan kaksi versiota sovelluksen käyttöliittymästä. Päällisin puolen niistä tuli täysin identtisiä. Näitä onnistuttiin myös testaamaan. Testauksen ja systemaattisen vertailun perusteella pystyttiin toteamaan, että Svelte on pienen kokonsa, nopeutensa ja yksinkertaisuutensa takia varteenotettava vaihtoehto Reactille. Sen kehittäjien yhteisö ja resurssit ovat silti yhä vähäiset Reactiin verrattuna. Svelte saattaakin tarvita vielä muutaman vuoden lisää, vahvistaakseen pysyvyyttään ja kasvattamaan yhteisöään, jotta se voi tosissaan haastaa isommat sovelluskehykset.

ASIASANAT:

Sovelluskehys, verkkokehitys, verkkosovellus, käyttöliittymä

BACHELOR´S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Business Information Technology

2021 | 38 pages

Karel Haapasalo

JAVASCRIPT FRAMEWORK IN THE IMPLEMENTATION OF A WEB APPLICATION USER INTERFACE

- React vs. Svelte

The amount of JavaScript frameworks and libraries used in web development has grown at a rapid pace. Frameworks can be used with user interface development, as well as client- and even server-side programming. They offer ready-made functionalities and more readable code, which makes developing much faster and smoother.

There were two JavaScript frameworks that have been examined in this thesis, which are being used mostly for making user interfaces. The objective of this thesis was to find out if Svelte is a worthy alternative for React. The objective was also to build two versions of a web application's user interface using React- and Svelte-frameworks. During implementation, the ways of which these component-based user interfaces can be built were compared. Complete solutions were also compared to each other systematically. In addition, the impact of a development community was also looked at, regarding each framework's vitality.

Two versions of application's user interface were successfully built as an endproduct of the thesis. On the surface, they are both identical. These were also successfully tested. Based on testing and systematic comparison, Svelte was discovered to be a worthy alternative to React because of it's small size, performance and simpicity. It's development community and resources are still relatively small compared to React. Svelte might need a few years to strengthen it's permanence and grow it's community, to really challenge bigger frameworks.

KEYWORDS:

Framework, web development, web application, user interface

SISÄLTÖ

KÄYTETYT LYHENTEET 6

1 JOHDANTO 7

2 VERKKOSOVELLUS JA JAVASCRIPT 9

2.1 Verkkosovelluksen määritelmä 9

2.2 Käyttöliittymä 9

2.3 JavaScript-ohjelmointikieli 10

2.4 Document Object Model-ohjelmointirajapinta 11

2.5 Sovelluskehykset 11

2.5.1 Määritelmä 11

2.5.2 React-sovelluskehys 12

2.5.3 Svelte-sovelluskehys 12

3 VERTAILUN MÄÄRITTELY 14

3.1 Ohjelmistojen valintaan vaikuttavat tekijät 14

3.2 Vertailun rajaus 15

3.3 Vertailukohdat 15

4 SOVELLUKSEN TOTEUTUS 17

4.1 Suunnittelu 17

4.2 Toteutus 19

4.2.1 Käyttöönotto 19

4.2.2 Komponentit 21

4.2.3 Propsit 23

4.2.4 Tilanhallinta 24

4.2.5 Tyylitys 27

4.2.6 Lopputuote 28

5 VERTAILUN SUORITTAMINEN 30

5.1 Testaus Lighthousella 30

5.2 Vertailutaulukko 31

5.3 Elinvoimaisuus 32

5.4 Kolmansien osapuolten tuki 34

6 POHDINTA 35

LÄHTEET 37

KUVAT

Kuva 1. Sovelluksen prototyyppi Adobe XD:llä. 17
Kuva 2. Jako komponentteihin. 18
Kuva 3. Create React App-kansiorakenne. 20
Kuva 4. Valmis käyttöliittymä. 28
Kuva 5. Reactin tulokset Lighthousella. 30
Kuva 6. Svelten tulokset Lighthousella. 30

TAULUKOT

Taulukko 1. Käyttöliittymien vertailu. 31

KÄYTETYT LYHENTEET

API	Ohjelmointirajapinta (Application Programming Interface).
CSS	Tyyliohjeiden laji (Cascading Style Sheet).
DOM	Puunomainen rakenne kuvaamaan esimerkiksi HTML-dokumenttia (Document Object Model).
HTML	Hypertekstin merkkauskieli (Hypertext Markup Language).
JS	Ohjelmointikieli, jolla web-sivuille luodaan toiminnallisuutta (JavaScript).
JSON	Tiedostomuoto tiedon välitykseen (JavaScript Object Notation).
JSX	Laajennus, joka helpottaa HTML-kielen lisäämistä Reactiin (JavaScript language extension).
NPM	Pakettimanageri JavaScript-kielelle (Node Package Manager).
SPA	Sovellus, jossa sivuja ei ladata uudelleen, vaan sisältö päivittyy uudelleen piirtämällä (Single Page Application).
SRP	Periaate, jossa komponentilla tulisi olla vain yksi tehtävä tai yksi syy muuttua (Single Responsibility Principle).

1 JOHDANTO

Yhä useamman asian siirtyessä verkkoon verkkosovellusten ja -sivujen tärkeys korostuu entisestään. Vaikka staattisia verkkosivuja on yhä olemassa, kehitetään uudet ratkaisut lähes poikkeuksetta dynaamisiksi. Verkkosovelluksia voisikin määritellä juuri dynaamisiksi verkkosivuiksi. Niiden kehityksessä JavaScript-ohjelmointikieli on keskeisessä asemassa, ja sitä pidetäänkin yleisesti verkon kielenä. Käyttöliittymiä tehtäessä käytössä on usein jokin JavaScriptin sovelluskehys helpottamaan kehittäjien työtaakkaa.

Käyttöliittymien rakentamisen tulisi olla mahdollisimman vaivatonta ja sujuvaa työtä. Tästä syystä hyvän sovelluskehyksen käyttö niiden rakennusvaiheessa on avainasemassa. Koska ohjelmointi-alalla toimintatavat ja tekniikat kehittyvät huimaa vauhtia, voi kehityksen kelkasta jäädä helposti pois. Monet yritykset haluavatkin kääntää katseensa tulevaan, ja olla ajan hermolla uusista käyttöliittymien rakentamiseen käytettävistä sovelluskehyksistä. Uuden teknologian käyttöönottamisen tulisi silti olla aina harkittu päätös. Kehittäjillä ei välttämättä ole entuudestaan kokemusta uudesta tekologiasta, jolloin heidän tulee opetella kehyksen toimintatavat ja parhaat käytännöt. Muutos vaatii siis aina aikaa ja resursseja.

Opinnäytetyön tavoite on pyrkiä vastaamaan siihen, onko Svelte-sovelluskehys varteenotettava vaihtoehto jo asemansa vakiinnuttaneelle React-sovelluskehykselle käyttöliittymien rakentamisessa. Tutkimuskysymys pyrkii vastaamaan toimeksiantajan haluun saada lisätietoa Svelte-sovelluskehyksestä. Tutkimusta tehdään osittain teoreettisena, osittain empiirisenä tutkimuksena. Empiiristä havaintoaineistoa saadaan opinnäytetyön toiminnallisesta osuudesta, jossa rakennetaan verkkosovelluksen käyttöliittymä kahdella tapaa. Toteutuksissa käytetään kumpaakin edellä mainutta sovelluskehystä. Havaintoaineistoa saadaan lisäksi kehittäjäyhteisöjen tekemistä tutkimuksista ja tilastoista.

Toteutettavana käyttöliittymänä on simppeli verkkokaupan tuotenäkymä, jota voi suodattaa, ja jonka tuotteita voidaan lisätä ostoskoriin. Rakennettavat esimerkit riittävät käymään läpi sovelluskehysten perustoiminnan, mutta edistyneempiin toimintoihin ei paneuduta. Teoreettista taustaa pyritään tukemaan käytännön koodiesimerkkien avulla, jotka nousevat käyttöliittymien koodipohjasta. Työssä ei perehdytä muihin verkkosovellusten olennaisiin osa-alueisiin, kuten sovellusarkkitehtuuriin tai HTML- ja CSS-kieliin, eikä myöskään palvelinpuolen asioihin. Tämä johtuu siitä, että lukijalla oleteaan olevan pohjalla tietämystä näistä asioista.

Valmiita toteutuksia vertaillaan testaamalla ja mittaamalla niitä eri osa-alueilla. Lopuksi vertailujen tulokset kootaan taulukkoon ja niistä tehdään johtopäätöksiä. Vertailussa otetaan myös kehysten elinvoimaisuus huomioon. Kehitysyheisön vaikutus, varsinkin avoimen lähdekoodin tapauksissa, on huomattavan oleellinen teknologioiden tulevaisuudennäkymien kannalta.

2 VERKKOSOVELLUS JA JAVASCRIPT

2.1 Verkkosovelluksen määritelmä

Verkkosovellukset ovat palveluita, useimmiten ohjelmia, joita tarjotaan käyttäjille verkon yli. Ne sijaitsevat palvelimilla ja niihin päästään käsiksi selainten avulla, riippumatta käyttöjärjestelmästä tai päätelaitteesta. Verkkosovelluksina voidaan pitää kaikkia sellaisia verkkosivustoja, jotka ovat dynaamisia, ja jotka toimivat yhdessä palvelinpuolen ja tietokannan kanssa (Kohan 2021).

Verkkosovelluksia ei pidä sotkea niin sanottuihin natiivisovelluksiin, jotka pitää yleensä ladata laitteelle. Näitä ovat esimerkiksi puhelin- tai työpöytäsovellukset. Lisäksi nämä ovat monesti käyttöjärjestelmäkohtaisia. Windows-käyttöjärjestelmälle tehdyt natiivisovellukset eivät toimi Linux- tai macOS-käyttöjärjestelmillä. Tästä poikkeuksena ovat progressiiviset verkkosovellukset, jotka toimivat natiivisovelluksen kaltaisesti, mutta ovat silti selainpohjaisia.

Nykyään verkkosovelluksia tehdään paljolti yhden sivun sovelluksina (SPA). Tämä tarkoittaa sitä, että sovellus lataa alustavasti kaikki tarvittavat toiminnallisuudet ja resurssit, mutta sovelluksen osia näytetään käyttäjän interaktiivisuuden mukaan (Madhuri, Balkrishna & Anushree 2015). Täten sivuja ei tarvitse ladata uudelleen sisällön muuttuessa, vaan se tapahtuu dynaamisesti uudelleen piirtämällä (engl. rendering).

2.2 Käyttöliittymä

Sitä verkkosovelluksen osaa, minkä käyttäjä näkee, ja minkä kanssa hän kommunikoi, kutsutaan käyttöliittymäksi. Monilla asioilla voi olla käyttöliittymiä, mutta pääpiirteittäin ne voidaan mieltää kosketuspinnoiksi koneiden ja ihmisten välillä (Jaye 2019). Olennaisia osia käyttöliittymien suunnittelussa ovat responsiivisuus, selainyhteensopivuus ja riittävä nopeus (Järvenpää 2021). Käyttöliittymien tulisi ensisijaisesti olla selkeitä ja helposti ymmärrettäviä, jotta käyttäjät pystyvät suorittamaan haluttuja tehtäviä ja toimia vaivattomasti.

Nykyään verkkopalveluiden käyttöliittymäsuunnittelu yrittää tasapainotella visuaalisten elementtien ja teknisen toimivuuden välillä. Verkkosivu tai -sovellus viestii käyttäjille

usein yritysten brändistä. Visuaalisesti miellyttävät sekä teknisesti hyvin toteutetut ratkaisut parantavatkin yritysten digitaalista näkyvyyttä ja luotettavuutta. (Järvenpää 2021.)

2.3 JavaScript-ohjelmointikieli

JavaScript on ohjelmointikieli, minkä varaan lähestulkoon jokainen verkkosovellus on rakennettu. Se on tehnyt mahdolliseksi sisällön päivittymisen verkkosovelluksissa ilman sivujen uudelleenlatausta (Haverbeke 2018). JavaScript vastaa siis verkkokehityksessä asiakasohjelman puolen toiminnallisuudesta.

Kieli on dynaamisesti tyypitetty, mikä tarkoittaa sitä, ettei muuttujia luodessa päätetä muuttujan datatyyppiä, vaan tämä tapahtuu vasta ajon aikana. Tästä syystä sillä on mahdollista tehdä asioita, mitä monella muulla vahvasti tyypitetyllä kielellä ei voida saavuttaa. Vaikka JavaScript onkin aloittelijaystävällinen ohjelmointikieli, vaikeuttaa dynaaminen tyypitys esimerkiksi ongelmien löytämistä rakennetuissa järjestelmissä. (Haverbeke 2018.) Kieltä voi käyttää tätä nykyä myös palvelinpuolen ohjelmointiin node.js-teknologian avulla.

JavaScriptiä ei tarvitse erikseen ladata, toisin kuin monia muita ohjelmointikieliä. Se on sisäänrakennettuna jokaiseen selaimeen riippuen tietenkin selaimen ja JavaScriptin versioista. Selaimet eivät tue muita skriptauskieliä. Stack Overflow -verkkosivuston vuonna 2020 teettämän kyselyn mukaan JavaScript on ollut sen hetken käytetyin ohjelmointikieli (Stack Overflow 2020). Kyselyyn vastasi noin 60 000 koodaajaa ympäri maailmaa.

Ensimmäinen versio JavaScriptistä ilmestyi jo vuonna 1997. Kieltä on tämän jälkeen päivitetty aktiivisesti ja ensimmäinen isompi päivitys tapahtui vuonna 2009, jolloin ES5 (ECMAScript 5) julkaistiin. Silloin lisättiin JSON-tuki, sekä monia uusia sisäänrakennettuja metodeita.

Seuraava ja nykyisin käytössä oleva ES6-versio julkaistiin vuonna 2015, jota on sen jälkeen päivitetty miltei vuosittain. ES6 myötä parantui esimerkiksi muuttujien määrittely, joka auttoi näkyvyysalueiden selkeydessä. Muita hyödyllisiä lisäyksiä ovat olleet asynkroniset funktiot ja lupaukset (engl. promises). Näiden myötä ohjelmat voivat suorittaa asioita samanaikaisesti, joutumatta odottamaan muiden ohjelman osien suoritusta loppuun. Molemmat React sekä Svelte mahdollistavat ES6-version käytön.

2.4 Document Object Model-ohjelmointirajapinta

Document Object Model (DOM) on ikään kuin hierarkinen tiekartta, minkä selain luo verkkosivusta. Siinä olevat elementit ovat ohjelmoitavia olioita, joihin voidaan viitata skriptissä. (Goodman 2005, 11–13.) Kun halutaan manipuloida sivua JavaScriptillä, tarvitaan tätä puumaista karttaa, minkä perusteella tiedetään mikä objekti milloinkin on kyseessä. DOM auttaa myös hahmottamaan sivustojen rakennetta.

Virtuaalinen DOM on konsepti, jossa alkuperäisestä Document Object Modelista luodaan virtuaalinen representaatio. Tätä muistissa pidettävää versiota voidaan verrata alkuperäiseen, jonka perusteella muutoksia tehdään. Jotkut ovat verranneet virtuaalista DOMia varjo DOMiin. Nämä ovat kuitenkin eri asioita, sillä varjo DOM keskittyy enemminkin näkyvyysalueisiin. (React Docs. 2021.) Se on kuitenkin yhtälailla ominaisuus Reactissa.

2.5 Sovelluskehykset

2.5.1 Määritelmä

Sovelluskehys (engl. framework) saatetaan monesti sekoittaa kirjastoon. Kirjastot ovat kuitenkin useimmiten vain kokoelma luokkia ja metodeita, joita ohjelmoija voi käyttää uudelleen ja uudelleen. Sovelluskehys on monimutkaisempi asia, mutta se voidaan mieltää luurangoksi, jonka sovellus itse täyttää kutsumalla ohjelmoijan kirjoittamaa koodia. Ohjelmoijan koodi kutsuu kirjastoissa olevaa koodia, mutta vastaavasti sovelluskehys kutsuu ohjelmoijan koodia, ja lisäksi se sisältää kirjastoja. (Programgeek 2011.) Tästä huolimatta termejä käytetään usein ristiin toistensa kanssa. Tässä työssä puhutaan Reactin ja Svelten kohdalla selkeyden vuoksi sovelluskehyksistä eikä kirjastoista.

JavaScriptille on tehty sovelluskehyksiä moniin eri tarkoituksiin. Käyttöliittymien tekemisen lisäksi kehyksiä on saatavilla myös esimerkiksi palvelinpuolen kehitykseen ja testaukseen. Sovelluskehyksiä käytetään, koska ne nopeuttavat ja helpottavat kehittäjien työtä merkittävästi (Arora 2021). Monet käyttöliittymien rakentamiseen käytetyistä kehyksistä ovat komponenttilähtöisiä, mikä tarkoittaa sitä, että ne pyrkivät jakamaan sovelluksen uudelleen käytettäviin osiin eli komponentteihin. Tämä puolestaan tekee

sovelluksista selkeämpia ja helpommin hallittavampia. Myös koodin toistaminen vähenee, sillä komponentit ovat uudelleenkäytettäviä.

2.5.2 React-sovelluskehys

React on Facebookin vuonna 2013 julkaisema sovelluskehys interaktiivisten käyttöliittymien luomiseen. Se on kirjoitettu JavaScriptillä, ja nykyään sitä pitää Facebookin lisäksi yllä yksittäisten kehittäjien ja yhtiöiden yhteisö. React toimii parhaiten yhden sivun sovellusten pohjana. Se on komponenttilähtöinen ja käyttää virtuaalista DOMia päivittäessään osia sovelluksesta (React Docs. 2021). Joissakin lähteissä React luokitellaan kirjastoksi (Tecci 2020).

Alun perin React otettiin vastaan skeptisesti, sillä tuohon aikaan sen esittämät käytännöt saattoivat tuntua jokseenkin hullun kuuloisilta. Se salli HTML-tyylisen kielen kirjoittamisen suoraan JavaScriptiin. Lisäksi se painotti funktionaalista ohjelmointia olio-pohjaisen ohjelmoinnin sijaan, joka ei suoranaisesti ollut ominaista JavaScriptille. (Banks & Porcello 2020, 1–2.)

Nykyään React on suosituin käytössä oleva sovelluskehys, jota on käytetty esimerkiksi Facebookin, Redditin sekä AirBnB:n rakentamiseen (AnyforSoft 2019). Se antaa kehittäjilleen enemmän aikaa keskittyä modernin JavaScriptin kirjoittamiseen ja vähentää kirjastokohtaisesta koodista huolehtimista. Se on myös suhteellisen nopea virtuaalisen DOMin ja erilaisten renderöintioptimointien takia. Reactin kanssa joudutaan usein käyttämään erillistä kirjastoa tilanhallintaan, vaikka React 16.8 versiossa esiteltiinkin koukut (engl. hooks) helpottamaan tilanhallintaa.

Reactin oppimista pidetään yleisesti ottaen monia muita suosittuja sovelluskehyksiä vaikempana. JSX-syntaksin opettelu voi olla aluksi haasteellista, varsinkin uusille koodareille. Lisäksi usein on parasta opetella myös erillisten kirjastojen, kuten Reduxin käyttö, mikä myös pidentää oppimiskaarta.

2.5.3 Svelte-sovelluskehys

Svelte on vuonna 2016 julkaistu avoimen lähdekoodin JavaScript-sovelluskehys. Se on kirjoitettu TypeScriptillä, ja alkuperäisenä kirjoittajana on ollut Rich Harris. Reactin tavoin myös Svelteä käytetään yhden sivun sovellusten tekemisessä. Se on

komponenttilähtöinen ja ei käytä virtuaalista Document Object Modelia, vaan generoi sovelluksen rakennusvaiheessa koodia, jolla DOMia manipuloidaan. Svelte seuraa muutoksia ylemmän tason komponenttien muuttujissa (Volkmann 2021). Uudelleen piirtäminen tapahtuu vain osissa, joihin muutokset ovat vaikuttaneet (Volkmann 2021). Tästä syystä Svelte on yleensä suorituskykyisempi käynnistyksessä ja ajon aikaisesti.

Svelte pyrkii siihen, että koodipohja pysyisi mahdollisimman pienenä ja hallittavana. Yleensä bugien määrä ja projektien kesto kasvaa, mitä suuremmaksi koodipohja kasvaa. (Svelte Docs. 2021.) Svelte on oikeastaan kääntäjä (engl. compiler), toisin kuin monet muut suositut sovelluskehykset, jotka ovat enemmänkin ajon aikaisia kirjastoja (Volkmann 2021).

Svelten oppimiskaarta pidetään suhteellisen pienenä. Sen kanssa ei välttämättä tarvitse käyttää erillisiä kirjastoja, ja se omaa suhteellisen simppelin tilanhallinta systeemin. Lisäksi sen syntaksi on käytännössä identtistä HTML-, CSS- ja JS-syntaksien kanssa. Sveltеä voi kirjoittaa myös TypeScriptillä, mikä puolestaan lisää oppimiskaaren kestoa. Tämä ei kuitenkaan ole pakollista.

3 VERTAILUN MÄÄRITTELY

3.1 Ohjelmistojen valintaan vaikuttavat tekijät

Koska ohjelmistojen kehitykseen käytettävä tekniikka muuttuu jatkuvasti, yrityksen on aina silloin tällöin hyvä katsoa, olisiko parempia vaihtoehtoja saatavilla. Näillä päätöksillä on pitkän aikavälin vaukutuksia niin palkkaukseen, kulttuuriin, kuin tuotettujen tuotteiden elinkelpoisuuteen (Halstead 2017).

Lähiaikojen trendinä on ollut se, ettei yrityksissä enää käytetä pelkästään yhtä ohjelmointikieltä. Projektien ja tuotteen luonne vaikuttaa enenevissä määrin myös valittuihin teknologioihin tai kieliin. Tärkeintä on löytää vaihtoehdot, joilla haluttuja ratkaisuja ja lopputuloksia voidaan edes rakentaa (Lazarevich 2018). Jos asiakkailla on tarve tietyntyyppisille sovelluksille, kuten mobiilisovelluksille, on työkalut ja kielet valittava sen mukaisiksi.

Seuraavia asioita on hyvä miettiä, kun valitaan ohjelmointikielten tai kirjastojen väliltä:

- yrityksessä jo käytössä olevat kielet
- sisäiset taidot ja kulttuuri
- paikallinen osaaminen
- tuottavuus
- kielen ekosysteemi ja tuki
- kielen pitkäkestoisuus (Halstead 2017).

Jos yrityksessä on käytetty tiettyä kieltä hyvällä menestyksellä, kannattaa kysyä onko uuden tilalle ottaminen oikeasti tarvittavaa. Uudet asiat vaativat aina resursseja, aikaa ja rahaa. Aluksi kannattaakin tarkistaa, pystytäänkö jo käytössä olevilla välineillä saavuttamaan samat lopputulokset. (Halstead 2017.)

Yrityksen kulttuurilla on myös vaikutusta siihen, millaisia teknologioita siellä saatetaan käyttää. Jos ollaan totuttu avoimen lähdekoodin ratkaisuihin, voidaan katsoa muita avoimen lähdekoodin vaihtoehtoja. On myös hyvä kartoittaa työntekijöiden osaamista ja kiinnostuksen kohteita, sekä paikallista osaamista ylipäätänsä. Tuottavuus on kenties tärkein aihe, mikä tulee kielten ja kirjastojen valintaan. Tuottavuutta on kuitenkin vaikea mitata objektiivisesti, sillä siihen vaikuttavat niin monet eri asiat. (Halstead 2017.) Jos

tuottavuus on juuri kyseisellä hetkellä huonoa, on silloin tärkeää pohtia, olisiko uudesta kielestä sen tiimoilta apua.

Kielen ekosysteemi ja sen yhteisö vaikuttavat laajalti sen kehitykseen ja pitkäkestoisuuteen. Jos kieli on ollut käytössä vasta pari vuotta, on sen pysyvyydestä vaikea sanoa mitään. Myös käyttäjien ja kehittäjien määrät ovat huomioitavia tekijöitä. Muita valintaan vaikuttavia tekijöitä voivat olla turvallisuus asiat ja skaalautuvuus.

3.2 Vertailun rajaus

Tässä opinnäytetyössä haluttiin keskittyä verkkosovellusten käyttöliittymäpuoleen, jonka myötä valittiinkin React ja Svelte. Svelte valittiin, koska se on uudempi ja lupaava sovelluskehys, joka hoitaa asioita erilailla, mutta joka on kuitenkin perusperiaatteiltaan samanlainen Reactin kanssa. Valintaan vaikuttivat myös toimeksiantajan kiinnostus Svelteen. Monelle toimeksianaja-yrityksen kehittäjistä React oli jo varsin tuttu, mutta Sveltestä ei vielä juurikaan ollut kokemusta.

Kun on päätetty mitä vaihtoehtoja vertaillaan, on hyvä tarkastella asioita ja kohtia joita itseasiassa vertaillaan. Ohjelmistojen välillä tehtävää vertailua voidaan tehdä monella tapaa. Kuinka tehokkaita ja suorituskykyisiä ne ovat, mikä on niiden suosion taso ja mitä kaikkea ne mahdollistavat. Sovelluskehysten kohdalla yksi kenties parhaista tavoista, on rakentaa samanlainen sovellus eri teknologioilla.

3.3 Vertailukohdat

Tässä opinnäytetyössä kyseistä lähestymistapaa on käytetty rakentamalla sovelluksen käyttöliittymä kahdella tapaa. Käyttäjältä katsoen käyttöliittymä on tehty identtiseksi. Sovelluksen rakentamisen avulla selviää, miten sovelluskehys otetaan käyttöön ja kuinka sillä itseasiassa voidaan rakentaa käyttöliittymiä.

Itse sovelluksen käyttöliittymän sisältö ja toteuttaminen on esitelty luvussa 4. Tämän jälkeen on esitelty lopputuote, jolle tehdään testausta luvussa 5. Testitulokset ja vertailukohteet ovat koottuna taulukkoon. Tuloksia pohditaan siltä kannalta, onko Sveltestä haastamaan Reactia.

Systemaattinen vertailu suoritetaan seuraavissa osa-alueissa:

- käyttöönotto
- sovelluksen rakentamisen vaiheet ja tavat
- sovelluksen koko
- suorituskyky
- koodirivien määrä
- elinvoimaisuus.

Käyttöönotossa tarkastellaan vain yhtä tapaa ottaa ohjelmisto käyttöön kunkin sovelluskehyksen kohdalla. Useimmiten saattaa olla monta tapaa, joilla näitä voidaan ottaa käyttöön. Huomioitavaa käyttöönottoon liittyen on sen nopeus ja helppous, sekä samalla luotu kansiorakenne.

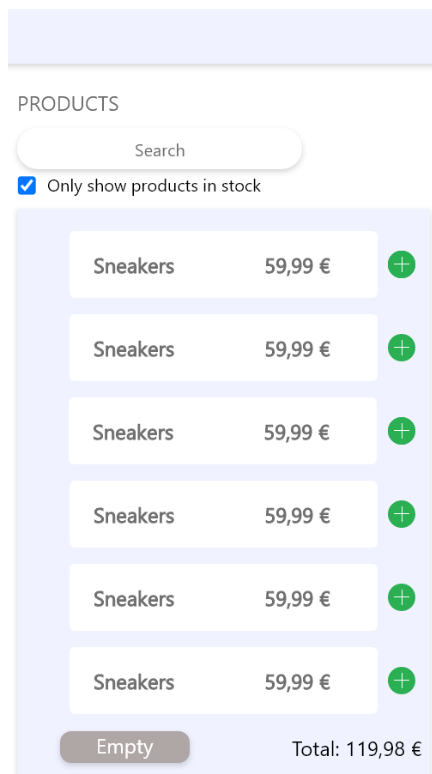
Sovelluksen rakentamisen yhteydessä vertaillaan sitä, kuinka komponentteja rakennetaan, kuinka tilaa ja elinkaarta pystytään hallitsemaan, ja kuinka tyylittelyä voidaan toteuttaa. Valmiiden sovellusten koko ja koodirivien määrät mitataan, koska nämä vaikuttavat aina sovelluksen nopeuteen. Suorituskykyä testataan Google Chromen kehittämistyökaluissa olevalla Lighthouse Audit -työkalulla, joka antaa pisteitä eri kategorioissa.

Viimeisenä vertailukohtana on sovelluskehyksen kehitysyhteisön vaikutus kehyksen elinvoimaisuuteen. Kehitysyhteisön tuki ja kehityksen jatkuminen sovelluskehyksen kohdalla on hyvin oleellista sen kannattavuuden puitteissa.

4 SOVELLUKSEN TOTEUTUS

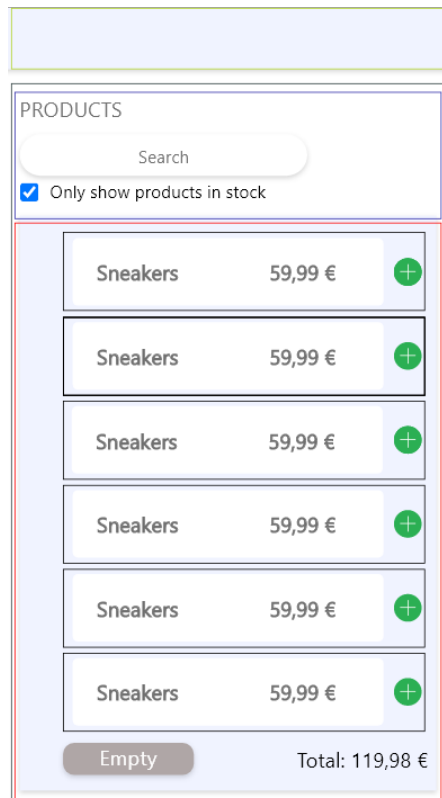
4.1 Suunnittelu

Sovelluksen käyttöliittymä, jonka avulla vertailua tullaan tekemään, on minimalistinen verkkokaupan tuotenäkymä (Kuva 1). Hakupalkista voidaan hakea tuotteita, jolloin ne suodattuvat hakusanan mukaan. Lisäksi voidaan valita, näytetäänkö vain varastossa olevat tuotteet. Ostoskorin summa päivittyy sitä mukaa, kun koriin lisätään tuotteita. Se voidaan myös tyhjentää empty-napista.



Kuva 1. Sovelluksen prototyyppi Adobe XD:llä.

Reactin ja Svelten kaltaisia käyttöliittymien rakentamiseen suunniteltuja sovelluskehyksiä käyttäessä on tärkeä jakaa sovellus komponentteihin jo suunnitteluvaiheessa. Yhtä tapaa, jolla voi tulkita mikä on komponentti, kutsutaan Single Page Responsibilityksi (SPA) (React Docs. 2021). Tämä tarkoittaa sitä, että sovelluksen komponentilla pitäisi olla vain ja ainoastaan yksi tarkoitus, tai vain yksi syy muuttua (Martin 2014). Kuvassa 2 on hahmoteltu piirrettyjen laatikoiden avulla, mitkä asiat toimivat komponentteina sovelluksessa.



Kuva 2. Jako komponentteihin.

Sovelluksen komponentteina toimivat tässä tapauksessa seuraavat asiat:

- Header (vihreä)
- FilterableProductsTable (tummanharmaa)
    - SearchBar (sininen)
    - ProductsTable (punainen)
        - Product (musta)

Sovelluksen tietorakenne koostuu JS-objektista, joka vastaa API-kutsusta palautettavaa JSON-payloadia (Esimerkkikoodi 1). Tätä tietorakennetta käytetään myöhemmin, kun piirretään sovelluksen tuotenäkymä, joka koostuu yksittäisistä Product-komponenteista.

Esimerkkikoodi 1. Sovelluksen tietorakenne.

```
const PRODUCTS = [
{ price: 49.99, stocked: true, name: 'Sneakers'},
{ price: 9.99, stocked: true, name: 'Baseball Cap'},
{ price: 29.99, stocked: false, name: 'Hoodie'},
{ price: 69.99, stocked: true, name: 'Training Joggers'},
{ price: 19.99, stocked: false, name: 'Training Shirt'},
{ price: 29.99, stocked: true, name: 'Sweatpants'},
{ price: 25.99, stocked: true, name: 'Training Shorts'},
];
```

Objekti sisältää jokaisen tuotteen tiedot. Hintana on double-tyyppinen numeerinen arvo, varastossa arvo on totuusarvo, ja nimi on merkkijono-tyyppiä. Tietorakenne on pyritty pitämään simppelinä tämän työn esimerkissä.

4.2 Toteutus

4.2.1 Käyttöönotto

React

React-sovelluksen voi luoda monella eri tapaa, mutta tässä työssä käytetään vain yhtä, Reactin Create React App-tapaa. Lisäksi luodaan vain paikallinen kehitysympäristö, jolla pääsee rakentamaan ja testaamaan sovelluskehystä. React vaatii node.js-ajoympäristöä toimiakseen. Node.js:n mukana tulee hyödyllisiä ohjelmistoja, kuten pakettimanageri NPM. Komento *npx create-react-app app-name* luo uuden React sovelluksen kansioineen ja tiedostoineen, asentaa tarvittavia riippuvuuksia ja paketteja, sekä alustaa paikallisen Git-repositoryn.

Tällä komennolla luodaan siis pelkästään asiakasohjelman pohja, eikä ollenkaan palvelinpuolen ohjelmoinnin pohjaa, jolloin kehittäjä voi valita, miten sen haluaa toteuttaa (React Docs. 2021). Asennettuista moduuleista mainittakoon Babel, joka sallii HTML-syntaksia muistuttavan JSX-syntaksin kirjoittamisen suoraan JS-tiedostoihin. Toinen mainittava moduuli on Webpack, joka kokoaa kaikki JS-tiedostot yhdeksi ajettavaksi bundle-tiedostoksi.

Kansiorakenteeseen luotu public-kansio sisältää index.html-tiedoston, joka toimii sivupohjana sovellukselle (Kuva 3). Muut kyseisessä kansiossa olevat tiedostot, kuten kuvat, ovat ainoastaan kyseisen index.html-tiedoston saatavilla. Src-kansioon tulevat kaikki ne

JavaScript- ja CSS-tiedostot, jotka Webpack kokoaa sovelluksen rakennusvaiheessa. Tässä kansiossa oleva index.js-tiedosto on muiden JS-tiedostojen tulokohta. Sovelluksessa tulee lisäksi oletuksena mukana esimerkiksi testaamista auttavia tiedostoja ja skriptejä.

```
./   .git/       build/         package.json       public/   src/
../  .gitignore  node_modules/  package-lock.json  README.md
```

Kuva 3. Create React App-kansiorakenne.

Oletuksena sovellus käynnistyy kehitys-versiona komennolla *npm start*. Mikäli tässä vaiheessa koodiin tehtäisiin muutoksia, piirrettäisiin sivun muutokset automaattisesti selaimessa tallennuksen yhteydessä. Juuri tämän dynaamisen piirtämisen takia Reactia käytetään. Sovelluksen voi ajaa myös testi- tai tuotantoympäristössä. Tuotannossa sovelluksen suorituskyky optimoidaan.

Svelte

Myös Svelte-sovelluksen voi asentaa ja luoda ainakin kahdella eri tapaa. Tässä työssä se tehdään jälleen taas pakettimanageri NPM:ää käyttäen. Myös Svelte toimii Noden päällä, joten sen tulee olla asennettuna. Komento *npx degit sveltejs/template app-name* lataa Svelte repositoryn, joka sitten asennetaan *npm install* komennolla. Tämä luo kansiorakenteen, lataa tarvittavat tiedostot ja riippuvuudet.

Sveltessä tärkeimmät aloitustiedostot ovat:

- public/index.html
- src/main.js
- src/App.svelte.

Index.html-tiedosto sisältää HTML-pohjan sekä viittaukset kahteen CSS-tiedostoon global.css ja bundle.css. Lisäksi tiedoston body-osiossa on tuotuna bundle.js-skripti, johon kootaan JS-tiedostot sovelluksen rakennuksen aikana. Tiedosto main.js piirtää App-komponentin ja määrittelee kohteen, jonne komponentti piirretään. App.svelte-tiedoston loppupääte .svelte, määrittelee tiedoston Svelte-komponentiksi. App-tiedoston tulisi olla sovelluksessa ylimmän tason komponentti. (Volkmann 2021.)

Kun Svelte sovellus rakennetaan se kokoaa sovelluksen koodin yhdeksi optimoiduksi JS-tiedostoksi, joka sisältää vain pienen määrän itse kehyksen koodia. Tästä syystä käytökseltään identtiset sovellukset ovat Sveltellä kooltaan paljon pienempiä Reactiin verrattuna. (Volkmann 2021.)

4.2.2 Komponentit

Komponentit ovat osia, jotka jakavat käyttöliittymän yksittäisiin uudelleenkäytettäviin palasiin. Komponentteja voi ajatella myös funktioiden kaltaisiksi, sillä ne voivat ottaa sisäänsä tietoja, jotka vastaavat funktioiden parametrejä. Ne myös palauttavat elementtejä. (React Docs. 2021.) Koodiesimerkkeinä toimii sovelluksessa käytettävä Header-komponentti, sekä Product-komponentti.

Reactissa komponentteja voidaan luoda kahdella eri tapaa. Ensimmäinen tapa on luoda luokkapohjaisia komponentteja. Tällöin luokan tulee periytyä React.Component-luokasta ja sen täytyy sisältää render-metodi (Esimerkkikoodi 2).

Esimerkkikoodi 2. Luokkapohjainen komponentti.

```
class Header extends React.Component {

    render() {
        const headerstyle = {
            width: "100%",
            backgroundColor: "#F0F3FF",
            textAlign: "center",
            boxShadow: "0 4px 8px 0 rgba(0, 0, 0, 0.2), 0 6px 20px 0 rgba(0, 0, 0, 0.19)"
        };

        return <div style={headerstyle}>
            <div className="headerText">
                <h2>Sporting Goods</h2>
            </div>
        </div>
    }
}

export default Header;
```

Header-luokassa on luotu render-metodin sisälle myös tyyli, jota käytetään palautettavassa HTML-elementissä. Huomattava osa komponenteissa on siitä palautettava HTML-elementti. Reactissa näitä voidaan luoda joko JSX-syntaksilla tai ilman. JSX-syntaksia käytetään lähes poikkeuksetta, sillä se tekee koodista luettavampaa, ja muistuttaa jo

valmiiksi HTML-koodia. Ajon alkaessa Babel kääntää tämän koodin tavalliseksi JavaScriptiksi.

Toinen tapa luoda komponentteja Reactissa on tehdä funktionaalisia komponentteja. Tämä on uudempi ja modernimpi tapa, ja niin sanottujen koukkujen myötä myös tilan hallitseminen on tullut helpommaksi. Funktionaaliset komponentit ovat kirjaimellisesti JS-funktioita (React Docs. 2021). Alla on kuvattu, miten funktionaalisia komponentteja luodaan, ja miten price-muuttujaa on käytetty JSX-syntaksin sisällä (Esimerkkikoodi 3).

Esimerkkikoodi 3. Funktionaalinen komponentti.

```
function Product() {

    let price = 19.90;

    return  <div className="product">
                <h3>Sneaker {price} €</h3>
            </div>
}

export default Product;
```

Sveltessä komponentteihin, jotka yleensä koostuvat HTML- CSS- ja JS-koodista, voidaan suoraan kirjoittaa HTML-syntaksia (Esimerkkikoodi 4). Tätä osaa komponentista, johon myös tyylitys ja JavaScript-koodi vaikuttavat, kutsutaan sen näkymäksi tai pohjaksi (Gasanov 2021).

Esimerkkikoodi 4. Svelte komponentti.

```
<script>
    let price = 19.90;
    let stocked = true;
    let name = 'Sneakers';
</script>

<style>
    .green-button {
    background-color: #4CAF50;
    border: none;
    color: white;
    padding: 8px 16px;
    }
</style>

<table>
    <tr>
        <td>{name}</td>
        <td>{price}</td>
        {#if stocked}
        <button class="green-button">Add</button>
        {/if}
    </tr>
</table>
```

Sekä React että Svelte mahdollistavat JavaScriptin sulauttamisen elementtien sisään. Koodi tulee tässä tapauksessa kirjoittaa aaltosulkeisiin. Yllä olevassa esimerkissä on käytetty if-ehtolausetta elementin sisässä. Tätä kutsutaan konditionaaliseksi piirtämiseksi. Elementti piirretään vain, jos tietty ehto toteutuu. Sveltessä tämä on tehty helpoksi, sillä ehdollisia lauseita voidaan kirjoittaa suoraan HTML-koodin sekaan. Reactissa tätä ei voi suoraan tehdä JSX-syntaksin sisällä. Kyseinen asia saavutettaisiin esimerkisi kutsumalla funktiota, joka tarkistaa ehdon ja piirtää, tai on piirtämättä sen mukaan elementtejä.

4.2.3 Propsit

Komponentit ottavat tietoa vastaan niin sanottujen propsien kautta. Propsit määritellään attribuuteiksi niitä kutsuttaessa ylemmän tason komponenttien toimesta. (Volkmann 2021.) Molemmat React sekä Svelte mahdollistavat tämän yhden suunnan tiedonvälityksen. Molemmissa kehyksissä propseja lähetetään samalla tavalla, laittamalla ne attribuutiksi komponentin kutsumisen yhteydessä. Koodissa propseina on käytetty PRODUCTS-objektia, joka välitetään App-komponentilta FilterableProductsTable-komponentille (Esimerkkikoodi 5).

Esimerkkikoodi 5. Propsien lisäys Reactissa.

```
  return (
    <div className="container">
      <Header />
      <FilterableProductsTable products={PRODUCTS} />
    </div>
  );
}
```

Koska komponentti, jolle propsit annetaan on luokkapohjainen, päästään siinä propseihin käsiksi luomalla konstruktori, joka ottaa vastaan parametrin props. Tämä alustaa ne käytettäväksi tästä eteenpäin kyseisessä komponentissa. Funkionaalisessa komponentissa propsit alustettaisiin laittamalla funktion sulkeiden sisään props-parametri. Komponentteja rakentaessa sääntönä on, ettei komponenttien tulisi koskaan muuttaa omien propsiensa arvoja, vaan komponenttien tulisi säilyä ”puhtaana”. (React Docs. 2021).

Sveltessä propsien käyttöön saaminen tapahtuu hiukan eri tavalla. Alemman tason komponentissa, eli tässä tapauksessa FilterableProductsTable-komponentissa tulee käyttää export-avainsanaa deklaratiivisen avainsanan edessä, jotta propsit saadaan käyttöön. Tämä luo muuttujan, jossa tiedot nyt ovat, ja joita voidaan käyttää viittaamalla muuttujan nimeen.

4.2.4 Tilanhallinta

Tilan hallitseminen on yksi tärkeimmistä asioista, kun kehitetään verkkosovelluksia sovelluskehyksillä. Tila kertoo sen, minkälainen näkymä ja mitä tietoa milloinkin tulee näyttää. Reactissa tilanhallintaa voi tehdä monella eri tapaa. Perinteisin tapa on käyttää luokkapohjaisia komponentteja, jotka mahdollistavat sisäänrakennetun state-objektin käytön. Esimerkkikoodissa 6 on käytetty kyseistä lähestymistapaa.

Esimerkkikoodi 6. Tilan asettaminen luokkapohjaisessa React-komponentissa.

```
class FilterableProductsTable extends React.Component {
    constructor(props) {
      super(props);
      this.state = {
        filterText: '',
        inStockOnly: false,
        totalAmount: 0.0
      };

      this.handleFilterTextChange = this.handleFilterTextChange.bind(this);
      this.handleInStockChange = this.handleInStockChange.bind(this);
      this.handleAddButtonClick = this.handleAddButtonClick.bind(this);
      this.handleEmptyButtonClick = this.handleEmptyButtonClick.bind(this);
    }
```

Esimerkissä state-objektiin on lisätty kolme ominaisuutta. Todellisuudessa näitä ominaisuuksia voisi olla rajaton määrä. Kun jokin näistä muuttuu, komponentti uudelleen-piirretään. Tilan muuttamiseksi tulee käyttää setState-metodia, sillä tämä kutsuu automaattisesti render-metodia. Tässä tapauksessa setState-metodia kutsutaan niin sanotun handler-metodin sisällä (Esimerkkikoodi 7).

Esimerkkikoodi 7. Tilan muuttaminen Reactissa.

```
handleFilterTextChange(filterText) {
  this.setState({
    filterText: filterText
  });
}
```

Reactissa ylemmän ja alemman tason komponentit eivät tiedä toisistaan sitä, onko niillä tilaa, tai onko komponentti määritelty luokkana vai funktiona (React Docs. 2021). Yllä oleva metodi, samoin kuin itse tila, voidaan kuitenkin välittää lapsikomponenteille propseina. Suosituksena on myös kiinnittää (engl. bind) Handler-metodit komponenttihin, jotta ne olisi oikein määritelty, kun niitä käytetään.

Modernimpi tapa hallita tilaa Reactissa, on käyttää funkionaalisia komponentteja, sekä niihin suunniteltuja koukkuja. Tämä lähestymistapa auttaa erityisesti tilaan liittyvän logiikan uudelleenkäytettävyydessä (React Docs. 2021). Tilan hallitsemien avuksi voi käyttää myös siihen suunniteltua erillistä kirjastoa. Suosituin Reactin tilanhallintaan käytetty kirjasto on Redux. Tämän työn toteutuksessa on kuitenkin pitäydytty vain luokkapohjaisessa lähestymisessä.

Sveltessä tilan hallitseminen on tehty paljon yksinkertaisemmaksi, eikä erillisiä kirjastoja tarvita. Esimerkissä komponentin script-osion sisällä on määritelty kolme muuttujaa, jotka vastaavat tilaa (Esimerkkikoodi 8).

Esimerkkikoodi 8. Tilan asettaminen Sveltessä.

```
let inStockOnly = false;
let filterText = '';
let totalAmount = 0.0;

const handleFilterTextChange = value => {filterText = value};
const handleInStockChange = checked => {inStockOnly = checked};
```

Näihin muuttujiin viitatessa ei tarvitse käyttää this-avainsanaa, toisin kuin Reactin luokkapohjaisessa lähetymisessä. Handler-funktioita ei myöskään tarvitse kiinnittää komponenttiin. Tämä tapa muistuttaa paljon enemmän perinteistä JavaScriptiä. Tila voidaan myös Reactin tavalla lähettää lapsikomponenteille propseina. Tila voidaan sitoa suoraan esimerkiksi lomakkeen kenttiin, jolloin se pysyy synkronoituna kentän arvoihin nähden.

Sveltessä tilan hallintaan on myös monia muita keinoja. Niin sanottu konteksti (engl. context) on yksi keino hallita tilaa ja dataa, ja siihen tulisi olla pääsy ainoastaan periytyvillä komponenteilla. Konteksti voi sisältää mitä tahansa arvoja aina merkkijonoista funktioihin tai objekteihin asti. Komponentit näkevät datan, joka on kontekstin sisällä ainoastaan siinä vaiheessa, kun komponentti alustetaan, eli ne eivät ole reaktiivisia. (Volkmann 2021.)

Varasto (engl. store) mahdollistaa tilan hallitsemisen Sveltessä, ilman että se olisi sidottu komponenttiin. Konteksti ja varasto eroavat toisistaan siinä määrin, että varastot ovat saatavilla mille tahansa osaa sovelluksesta, kun taas konteksti on saatavilla ainoastaan komponentille ja sen jälkeläisille (Svelte Docs. 2021). Sveltellä on kolme erilaista varastoa, jotka ovat writable store, readable store ja derived store.

Saadakseen varaston käyttöön komponentissa, se tulee lähettää propsina tai hakea kontekstista. Sen voisi myös importoida. Lisäksi Sveltessä on mahdollisuus luoda omia varastoja. Tämä tekee Sveltestä erittäin joustavan ja mukautuvan.

Myös Reactissa pystytään käyttämään kontekstia React Contextilla, joka toimii hyvin samalla tavalla kuin Svelten konteksti. Kontekstia käytetään usein esimerkiksi säilyttämään tietoa käyttäjien autentisoinnista, teemoista tai kieliasetuksista (React Docs. 2021). Kun Reactissa halutaan käyttää varastoja, tarvitaan siihen erillistä kirjastoa.

Molempien sovelluskehysten ominaisuuksiin kuuluvat myös niin sanotut elinkaari metodit. Nämä ovat metodeita, joita ajetaan komponentin eri elinkaaren vaiheissa. Vaiheita ovat komponentin piirto, DOMin päivitys ja komponentin tuhoaminen. Sveltellä on huomattavasti vähemmän elinkaaren vaiheisiin liittyviä metodeita kuin Reactilla.

Opinnäytetyön toteutuksessa varastoa ja kontekstia ei ole käytetty, koska on haluttu pitää tilan hallitseminen mahdollisimman simppelinä. Myöskään elinkaarimetodeita ei tässä tapauksessa ole tarvittu.

4.2.5 Tyylitys

React yhteisössä on käyty paljon keskustelua siitä, miten tyylitystä tulisi käyttää. Vaihtoehtoina ovat esimerkiksi omat erilliset CSS-tiedostot ja niiden tuominen komponentteihin, style-attribuutin kautta tyylittäminen suoraan elementtiin, tai komponenttikohtaisen tyylityksen luominen JavaScript-objektina. Esimerkkikoodissa on käytetty komponenttikohtaista tyylitystä, jota käytetään div-elementissä (Esimerkkikoodi 9).

Esimerkkikoodi 9. Tyyliobjektin luominen ja käyttö Reactissa.

```
render() {
    const headerstyle = {
        width: "100%",
        backgroundColor: "#F0F3FF",
        textAlign: "center",
        boxShadow: "0 4px 8px 0 rgba(0, 0, 0, 0.2), 0 6px 20px 0 rgba(0, 0, 0, 0.19)"
    };

    return <div style={headerstyle}>
        <div className="headerText">
            <h2>Sporting Goods</h2>
        </div>
    </div>
}
```

Myös Sveltessä voi käyttää erillisiä ja globaaleja tyylitiedostoja, jotka importoidaan tarvittaviin osiin. Tämä onkin usein hyvä tapa säilyttää tyylittelyä, jota tarvitaan useissa osissa sovellusta. Komponenttikohtaisen tyylityksen Svelte on tehnyt todella helpoksi sen komponentin rakenteen ansiosta. Komponenteissa on oma tyylitunniste, jonka sisään komponenttikohtainen tyylittely tulee (Esimerkkikoodi 10).

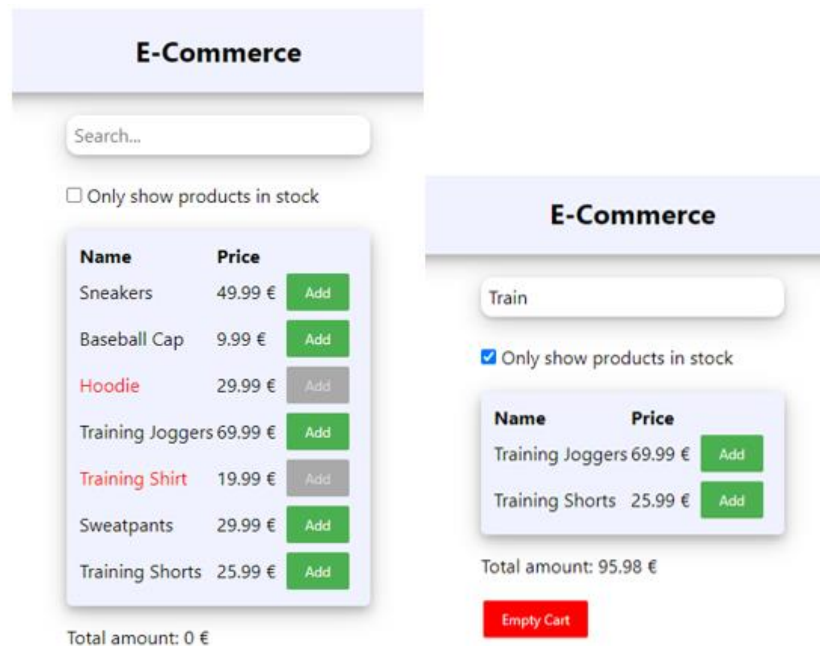
Esimerkkikoodi 10. Tyylittely Svelten komponentissa.

```
<style>

    form {
      display: flex;
      flex-direction: column;
    }

    .searchBar {
      border-radius: 10px !important;
      border: none;
      margin-top: 20px;
      box-shadow: 0 2px 4px 0 rgba(0, 0, 0, 0.2), 0 6px 20px 0 rgba(0, 0, 0, 0.19);
    }
  </style>
```

Tunnisteiden sisään voi kirjoittaa suoraan CSS-koodia, jossa on helppo viitata elementtien luokkiin tai id-tunnisteisiin. Tämä selkeyttää tyylittelyn lähestymistapaa huomattavasti ja muistuttaa hyvin paljon tavallisen HTML-dokumentin tyylitunnisteiden käyttöä.

4.2.6 Lopputuote

Lopputuloksena saatiin toteutettua toimiva käyttöliittymä (Kuva 4). Käyttöliittymä on täysin saman näköinen molemmissa versioissa, ja käyttäytyy päällisin puolin samalla tavalla. Taulukossa näkyvät tuotteet, niiden hinta ja tuotteen lisäys nappi. Mikäli tuotetta ei ole varastossa, näytetään tuotteen nimi punaisella ja lisäys nappi on toimintakyvytön.



Kuva 4. Valmis käyttöliittymä.

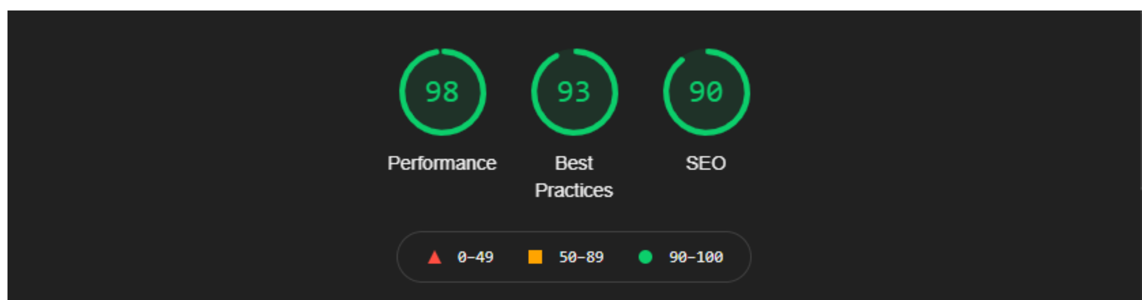
Kuvan oikealla puolella on suodatetun taulukon näkymä. Hakupalkkiin on kirjoitettu tekstiä, jolloin kolme tuotetta vastaavat hakusanaa. Lisäksi on valittu näytettäväksi vain varastossa olevat tuotteet, jolloin yhtä hakusanalla vastaavaa tuotetta ei näytetä, koska sitä ei ole varastossa. Ostoskoriin on lisätty tuotteita, jolloin Empty Cart-nappi tulee näkyviin. Napista painamalla summa tyhjenee ja nappi katoaa.

Lopullisen version tyylittely erosi hiukan alkuperäisestä suunnittelusta, mutta asialla ei ole vaikutusta sovelluksen toimintaan. Toiminnallisuuksiltaan toteutus on täysin suunnitelman mukainen.

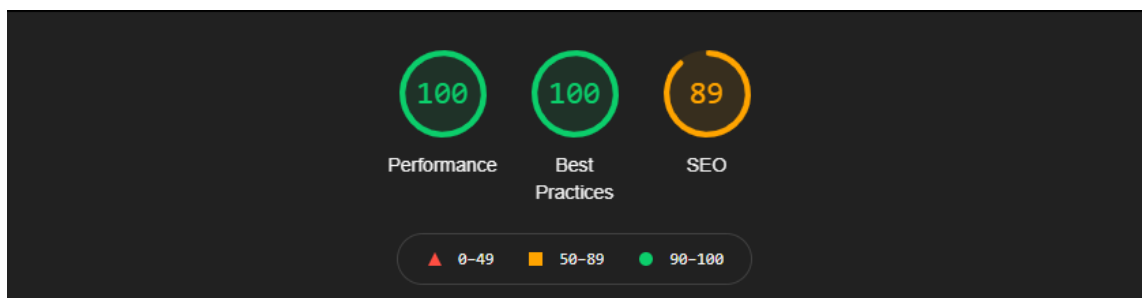
5 VERTAILUN SUORITTAMINEN

5.1 Testaus Lighthousella

Valmiita käyttöliittymiä testataan, jotta saadaan selville onko käyttöliittymissä suuria eroja. Testiympäristönä toimii luonnollisesti selain, tässä tapauksessa Google Chrome. Chromessa on automatisoitu työkalu verkkosivujen testaukseen nimeltä Lighthouse. Sitä voi ajaa kehittäjän työkalujen avulla, komentoriviltä tai Node-moduulina mitä tahansa verkkosivua tai -sovellusta vasten (Web.dev, 2021). Työkalu mittaa muun muassa sivujen suorituskykyä, käytettävyyttä sekä hakukoneoptimointia, ja antaa pisteitä nollan ja sadan välillä näissä kategorioissa. Kuvissa 5 ja 6 on molempien toteutusten pisteytys Lighthousella testattuna.



Kuva 5. Reactin tulokset Lighthousella.



Kuva 6. Svelten tulokset Lighthousella.

Molemmilla versioilla on hyvin samanlaiset pisteet testin perusteella. Tämä johtuu hyvin pitkälti siitä, että sovellukset ovat sen verran pieniä, ettei suuria eroja päästä näkemään. Tärkein kategoria testissä on suorituskyky, jossa Svelte on piirun verran parempi.

Suorituskyvyn mittaaminen kattaa sivuston latausaikoja, sekä aikaa, jolloin sivusta tulee täysin interaktiivinen. Latausaikoja seurataan esimerkiksi ensimmäisestä ja suurimmasta DOM-elementistä, jotka sivulle piirretään. Työkalu laskee myös nopeusindexin sille, koska ensimmäistä visuaalista sisältöä saadaan sivulle. (Web.dev 2021.) Keskimmäinen pisteytys kertoo siitä, onko sovellus toteutettu parhaita käytäntöjä käyttäen. Tämä saattaa vaikuttaa osittain myös suorituskykyyn, mutta ei tässä tapauksessa merkittävästi. Viimeinen tilasto kertoo siitä, kuinka hyvin sivu on hakukoneoptimoitu. Molemmat ovat käytännössä yhtä hakukoneoptimoituja. Arvot ovat sen verran hyviä, että voitaisiin todeta Reactin sekä Svelten olevan hakukone ystävällisiä.

5.2 Vertailutaulukko

Vertailutaulukkoon on tiivistetty tietoa ja tuloksia molemmista käyttöliittyymien versioista (Taulukko 1). Taulukkoon kootut tiedot ovat sellaisia, joista saadaan jokin luku tai arvo selvästi selville. Siihen ei ole otettu mukaan esimerkiksi oppimiskaarta, sillä sen kestoa tai arvoa on hankala numeraalisesti mitata. Taulukko kattaa myös sovelluskehysten kehittäjäyhteisöön liittyviä tilastoja.

Taulukko 1. Käyttöliittymien vertailu.

	React	**Svelte**
Ajoaika projektin luonnille	77 sek	2 sek
Asennettujen tiedostojen koko	153 MB	17 MB
Valmiin sovelluksen koko	174 MB	17,3 MB
Bundle-tiedoston koko	36 KB	11 KB
Koodirivien määrä	382 riviä	328 riviä
Suorituskyky (Lighthouse)	98/100	100/100
Parhaat käytännöt (Lighthouse)	93/100	100/100
SEO (Lighthouse)	90/100	89/100
Edistäjät (GitHub)	1542	396
Käyttäjät (GitHub)	6.5m	47k
Tähdet (GitHub)	169k	47k
Viikottaiset lataukset (NPM)	9.1m	240k

Taulukosta selviää, että Svelte on huomattavasti nopeammin käyttöönotossa. Sen asennukseen meni vain muutama sekunti, kun taasen Reactin asennukseen meni reipas minuutti. Lisäksi Reactin mukana tulleet tiedostot ovat noin 10 kertaa suuremmat kuin Sveltessä.

Projektin tiedostojen koko ei kuitenkaan ole niin merkittävä kuin bundle-tiedoston koko. Bundle on koottu tiedosto, jossa on sisällytettynä koko sovellus. Tämän tiedoston koko on Sveltessä pienempi, johtuen siitä, ettei itse kirjastokohtaista koodia tarvita Svelte sovelluksissa paljoakaan. Svelten tiedosto on noin kolmanneksen React tiedoston koosta. Myös vuonna 2019 tehdyssä vertailututkimuksessa, jossa rakennettiin eri sovelluskehyksillä RealWorld App, todettiin Svelte-version olevan monin verroin React-versiota pienempi (Schae 2019).

Koodirivien määrä on myös hyvä huomioida, sillä mitä vähemmän tarvitsee koodia kirjoittaa, sitä vähemmän aikaa kuluu kehittämiseen. Svelte versio vaati tässäkin tapauksessa vähemmän koodia, ero ei kuitenkaan ole merkittävä. Se tulisi luultavasti esille paremmin suuremman sovelluksen kohdalla. Samaisessa tutkimuksessa listattiin myös vuoden 2019 vertailututkimuksessa rakennetun RealWorld App:n koodirivien määriä. Siinä Svelten koodirivien määrä oli 1116 riviä, kun taas Reactilla se oli noin 2000 riviä (Schae 2019).

Koodirivien pienempi määrä Sveltessä johtuu siitä, että siinä komponenttien rakennus vaatii huomattavasti vähemmän ylimääräistä koodia, eikä toistettavaa koodia esimerkiksi metodien sitomisen takia jouduta tekemään. Sveltessä on pyritty juuri siihen, että komponentit pysyvät helposti luettavina ja mahdollisimman suppeina.

Suorituskyky oli hyvin samaa luokkaa molemmissa käyttöliittymissä. Eroja ei päässyt näin pienessä sovelluksessa juurikaan syntymään. Tästä johtuen ei voi tehdä merkittäviä havaintoja siitä, kumpi on suorituskykyisempi. Tämäkin tulisi paremmin esiin suuremman sovelluksen kohdalla. Voidaan kuitenkin päätellä, että Svelte on hiukan suorituskykyisempi, johtuen myös sen pienemmästä koosta. Lataukseen menevä aika vähenee mitä pienempi sovellus on.

5.3 Elinvoimaisuus

Molempien sovelluskehysten repositoryt löytyvät GitHubista. Tämä antaa hyvän vertailukohdan niiden kehitysyhteisöistä. Molemmilla on myös omilla verkkosivuillaan

dokumentaatio. Nämä dokumentaatiot ovat kuitenkin verrattain suppeita, ja kehittäjien on tukeuduttava myös muihin lähteisiin tiedonhaussa.

Taulukossa esitelty edistäjät (engl. contributors) viittaa kehittäjiin, jotka ovat osaltaan edistäneet Reactin koodia ja luoneet siihen muutoksia. Reactilla on ollut neljä kertaa enemmän edistäjiä, mikä kertoo suuresta kehittäjäyhteisöstä. Lisäksi Reactin elinvoimaisuuden takaa se, että sillä on taustalla iso yhtiö.

Svelten kehittäjäyhteisö on huomattavasti pienempi, mutta se on iältäänkin kolme vuotta nuorempi teknologia. Sveltellä ei ole teknologiajättiä sen taustalla, joten sen elinvoimaisuus ei ole aivan yhtä taattu. Sitä kuitenkin kehitetään aktiivisesti kokoajan, ja se on tullut vuosi vuodelta yhä useamman kehittäjän tietoisuuteen.

Käyttäjämäärät ovat käyttäjiä, joiden projektien repositoryt ovat riippuvaisia Svelten tai Reactin repositorystä. Reactilla käyttäjien määrä on 6 500 000. Tämä kertoo Reactin suosiosta, verrattuna Svelten 47 000 käyttäjään. Suosiota voi mitata osaltaan myös tähdillä, joita voi antaa GitHubissa. Reactilla on tähtiä 169 000 kappaletta, kun taas Sveltellä niitä on 47 000. Reactin tähdet ovat käyttäjämääriin suhteellisen vähäiset. Sveltellä sen sijaan on yhtä paljon tähtiä kuin käyttäjiä. Tämä kertoo siitä, että monet kehittäjät jotka ovat kokeilleet Svelteä, ovat pitäneet siitä.

NPM pakettimanagerin tilastot kertovat, että Reactin viikottaiset lataukset ovat olleet huimat 9 092 242 kappaletta. Viikottaisten latausten määrä on kasvanut 1 500 000 kappaleella vuodessa. (NPM 2021.) React on siis aktiivisesti käytössä lukuisalla joukolla kehittäjiä, ja onkin syystä käytetyin JavaScriptin sovelluskehys.

Svelten kohdalla viikottaiset lataukset ovat 240 000 kappaletta. Vuosi sitten Svelten viikottaiset lataukset olivat vain 67 000 kappaletta. (NPM 2021.) Määrällisesti se ei ole lähelläkään Reactin lukuja, mutta prosentuaalisesti kasvu on ollut paljon parempaa. Svelten latausten määrä on yli kolminkertaistunut viimeisen vuoden kuluessa.

Luvut kertovat Reactin suuresta käyttäjäkunnasta ja vankasta asemasta. Moni uskookin Reactin säilyttävän suosituimman sovelluskehyksen aseman vielä pitkään. Molemmat sovelluskehykset ovat kasvaneet tasaiseen tahtiin, ja näyttäisikin siltä, että kasvu jatkuu (NPM 2021). Svelten tulevaisuudesta ei voida vielä olla aivan yhtä varmoja, sen pienen kokonsa tähden.

5.4 Kolmansien osapuolten tuki

Kolmannen osapuolen tuki on tärkeässä asemassa, varsinkin avoimen lähdekoodin toteutuksissa. Reactissa on mahdollista käyttää useita eri kirjastoja, kuten jo aiemmin mainittua Reduxia. Kirjastot auttavat saavuttamaan tai parantamaan tiettyjä ominaisuuksia. Reduxin tavoin myös esimerkiksi Zustand kirjasto on suunniteltu auttamaan tilan hallinnassa. React Queryä taas käytetään verkon yli tehtävissä pyynnöissä. Reactilla on yhteensä 72 357 riippuvaisuutta (engl. dependents), jotka ovat juurikin kirjastoja tai apuohjelmia (NPM 2021).

Sveltellä riippuvaisuuksia on 450 kappaletta (NPM 2021). Sovelluskehys onkin suunniteltu siten, että se ei tarvitsisi tiettyihin asioihin, kuten tilanhallintaan kolmansien osapuolten apua. Toki riippuvaisuuksia on usein moneen lähtöön, minkä vuoksi React on tässä kohtaa monipuolisempi.

6 POHDINTA

Työn tavoitteena oli selvittää, onko Svelte sovelluskehyksestä haastamaan asemansa vakiinnuttaneen Reactin verkkosovellusten käyttöliittymien rakentamisessa. Tämä saavutettiin tutkimalla niitä teoriatasolla, ja vertailemalla koodiesimerkein käyttöliittymien rakennuksen yhteydessä kummankin tapoja tehdä asioita. Lisäksi sovelluskehyksiä vertailtiin rakennettujen käyttöliittymien testitulosten ja kokojen kautta. Myös kehitysyhteisöjen vaikutusta kehysten elinvoimaisuuteen vertailtiin.

Tutkimuksen perusteella voidaan todeta, että Svelte on varteenotettava vaihtoehto suositulle Reactille. Se tekee komponenttien kirjoittamisen vaivattomammaksi ja selkeämmäksi. Se muistuttaa tyyliltään perinteisen HTML- CSS- ja JS-koodin kirjoittamista suoraan HTML-dokumenttiin, pysyen kuitenkin juuri komponenttilähtöisenä. Se on myös verrattain helpompi ottaa käyttöön ja oppia, eikä nojaudu niin vahvasti ulkoisiin kirjastoihin saavuttaakseen tiettyjä toiminnallisuuksia.

Lopputuote oli testausten perusteella Svelte-versiossa suorituskyvyltään hiukan parempi ja kooltaan pienempi. Sovellusten käyttöliittymien toteutuksesta ei voi tehdä kuitenkaan absoluuttisia johtopäätöksiä siitä, kumpi olisi suorituskykyisempi tai nopeampi sovelluskehys. Tämä johtuu siitä, että käyttöliittymien koko oli työssä sen verran pieni, etteivät kaikki ominaisuudet päässeet oikeuksiinsa. Lisäksi koodin kirjoittaja vaikuttaa aina siihen, kuinka hyvin koodi toimii. Asioita voidaan tehdä suorituskyvyn kannalta huonosti hyvilläkin välineillä ja toisinpäin.

Kehittäjäyhteisön kannalta Svelte on vielä jäljessä. Se on ollut tykätty sitä kokeilleiden kehittäjien keskuudessa, mutta määrät ovat yhä vähäisiä Reactiin verrattuna. Voi olla, että yritykset haluavat Sveltellä olevan suurempaa jalansijaa myös yritysmaailmassa, ennen kuin ne ottavat sen omaan käyttöönsä. Lisäksi kolmansien osapuolten tuki on Sveltessä Reactiin verrattuna vielä vähäistä, joka vaikuttaa tiettyjen toiminnallisuuksien saatavuuteen.

Työssä onnistuttiin rakentamaan suunnitelman mukainen käyttöliittymä, jota voitiin onnistuneesti myös testata. Käyttöliittymän koodiesimerkkejä pystyttiin myös hyödyntämään vertailussa. Sovelluksen käyttöliittymä oli tässä työssä suhteellisen suppea, jolloin kaikki ominaisuudet eivät päässeet näytille. Toiminnallinen osuus haluttiin kuitenkin pitää suhteellisen suppeana ja keskittyä myös teoriaan sekä kehysten elinvoimaisuuteen.

Tutkimus näyttäisi tukevan sitä ennalta asetettua olettamusta, että Svelte on nopea ja sillä rakennettujen sovellusten koodipohja on keskivertoa pienempi. Lisäksi Svelteä oli mielestäni helppo oppia, sillä en sitä entuudestaan osannut. Tämä on ollut myös yleinen käsitys Sveltestä. Vastaavanlainen tutkimus olisi hyvä tehdä muutaman vuoden päästä uudestaan, sillä kuten aiemmin todettiin, teknologiat kehittyvät huimaa vauhtia. Svelte on osaltaan vielä kasvuvaiheessa, ja sen suosio sekä kannattajakunta tulee mitä luultavimmin kasvamaan. Tämän myötä sitä voi kenties vielä perusteellisemmin verrata Reactin kaltaisiin sovelluskehyksiin.

LÄHTEET

AnyforSoft 2019. 10 Famous Websites Built with React JS. AnyforSoft 27.6.2019. Viitattu 19.3.2021 https://anyforsoft.com/blog/10-famous-websites-built-react-js/

Arora, S. 2021. 10 Best JavaScript Frameworks to Use in 2021. Julkaistu 23.1.2021. Viitattu 31.5.2021 https://hackr.io/blog/best-javascript-frameworks

Banks, A. & Procello, E. 2020. Learning React: Modern Patterns for Developing React Apps. 2nd edition. Sebastobol CA: O'Reilly Media Inc. https://ebookcentral.proquest.com/lib/turkuamk-ebooks/detail.action?docID=6226974&query=Learning+React

Facebook Inc. 2021. React docs. Viitattu 24.3.2021 https://reactjs.org/docs/getting-started.html

Gasanov, T. Lighter and Faster – A Guide to the Svelte Framework. Toptal 2021. Viitattu 26.3.2021 https://www.toptal.com/front-end/svelte-framework-guide

Goodman, D. 2005. Dynamic HTML: The Definitive Reference. 2nd edition. Sebastobol CA: O'Reilly Media Inc. https://books.google.fi/books?id=AuO923YJTX4C

Halstead, B. 2017. How to Choose a Programming Language. Julkaistu 17.8.2017 Cto Craft. Viitattu 7.5.2021 https://medium.com/cto-craft/how-to-choose-a-programming-language-a4b8212d2e37

Haverbeke, M. 2018. Eloquent JavaScript: A Modern Introduction to Programming. 3rd edition. Viitattu 16.3.2021. https://eloquentjavascript.net/Eloquent_JavaScript.pdf

Jaye, H. 2019. What Is A User Interface, And What Are The Elements That Comprise one? Career Foundry. Viitattu 16.3.2021 https://careerfoundry.com/en/blog/ui-design/what-is-a-user-interface/

Järvenpää, L. Käyttöliittymä- & käyttäjäkokemussuunnittelu (UI & UX Design). Itewiki. Viitattu 16.3.2021 https://www.itewiki.fi/opas/kayttoliittymasuunnittelu-ux-user-experience-design-eli-kayttajakokemus/

Kohan, B. Guide to Web Application Development. Viitattu 16.3.2021 https://www.comentum.com/guide-to-web-application-development.html

Lazarevih, V. 2018. How to compare software solutions, framework, libraries and other components. Julkaistu 25.08.2018, Digiteum. Viitattu 30.4.2021 https://www.digiteum.com/how-to-compare-software-solutions-frameworks-libraries-and-other-components/

Madhuri, J., Balkrishna, S. & Anushree, D. 2015. Single Page Application using AngularJ. International Joural of Computer Science and Information Technologies, Vol 6. Viitattu 30.3.2021 https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.736.4771&rep=rep1&type=pdf

Martin, R. 2014. The Clean Code Blog. Viitattu 12.4.2021 https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html

NPM, 2021. Website. Viitattu 24.5.2021 https://www.npmjs.com/

Programgeek 2011. Library vs. Framework? Julkaistu 2.9.2011. Viitattu 17.3.2021 https://www.programcreek.com/2011/09/what-is-the-difference-between-a-java-library-and-a-framework/

Schae, J. 2019. RealWorld Comparison. Julkaistu 8.4.2019. Viitattu 21.5.2021 https://www.freecodecamp.org/news/a-realworld-comparison-of-front-end-frameworks-with-benchmarks-2019-update-4be0d3c78075/

Stack Overflow 2020. 2020 Developer Survey. Viitattu 18.3.2021 https://insights.stackoverflow.com/survey/2020#technology-programming-scripting-and-markup-languages-all-respondents

Svelte 2021. Svelte docs. Viitattu 24.3.2021 https://svelte.dev/docs

Tecci 2020. React vs Angular vs Vue: Mikä JavaScript Kirjasto Valita Vuonna 2020?. Julkaistu 2.9.2020. Viitattu 31.5.2021 https://tecci.fi/2020/09/02/react-vs-angular-vs-vue-mika-javascript-kirjasto-valita-vuonna-2020/

Volkmann, M. Svelte – Web Development Make Easier. Viitattu 24.3.2021 http://mvolkmann.github.io/programming/svelte-article/svelte-article.pdf

Web.dev. 2021. Viitattu 28.4.2021 https://web.dev/lighthouse-performance/