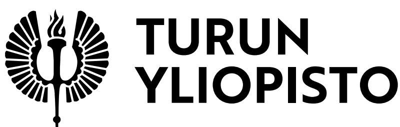


React-kirjaston vaikutus sovelluskehityksen tehokkuuteen verrattuna pelkkään JavaScriptiin

Tietotekniikka

Tietotekniikan laitos, Teknillinen tiedekunta

Kandidaatintutkielma

Tuukka Maunu

Toukokuu 2026

Turun yliopiston laatujärjestelmän mukaisesti tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -järjestelmällä.

Kandidaatintutkielma

Tutkinto-ohjelma, oppiaine: Tietotekniikka

Tekijä(t): Tuukka Maunu

Otsikko: React-kirjaston vaikutus sovelluskehityksen tehokkuuteen verrattuna pelkkään JavaScriptiin

Sivumäärä: 29 sivua

Päivämäärä: Toukokuu 2026

Tässä kandidaatintutkielmassa tarkastellaan React-kirjaston vaikutusta web-sovelluskehityksen tehokkuuteen verrattuna pelkän JavaScriptin käyttöön. Tutkimuksen tavoitteena on selvittää, millä tavoin Reactin tarjoamat rakenteelliset ja tekniset ominaisuudet, kuten komponenttipohjaisuus, tilanhallintamekanismit ja virtuaalinen DOM, vaikuttavat kehitysprosessiin, koodin ylläpidettävyyteen ja sovellusten suorituskykyyn. Tutkimus toteutettiin vertailevana analyysina, jossa rakennettiin yksinkertainen käyttöliittymäsovellus kahdella eri tavalla: ensin perinteisellä JavaScriptillä ja tämän jälkeen Reactilla. Vertailua varten hyödynnettiin kirjallisuutta, dokumentaatiota sekä toteutettujen esimerkkien analyysiä.

Tulosten perusteella React tehostaa kehitystyötä erityisesti tilanteissa, joissa sovellus on laaja tai sisältää paljon dynaamisia vuorovaikutuksia. Komponenttipohjainen arkkitehtuuri parantaa koodin modulaarisuutta ja helpottaa sen uudelleenkäyttöä, kun taas virtuaalinen DOM nopeuttaa käyttöliittymän päivityksiä vähentämällä tarpeettomia renderöintejä. Lisäksi Reactin tarjoamat tilanhallintaratkaisut selkeyttävät sovellusrakennetta verrattuna pelkän JavaScriptin hajautettuun logiikkaan. Päätelmänä voidaan todeta, että React soveltuu erityisen hyvin skaalautuviin ja pitkäikäisiin projekteihin, joissa ylläpidettävyys ja kehitystehokkuus ovat keskeisiä vaatimuksia.

Avainsanat: React, JavaScript, web-sovelluskehitys, komponenttipohjaisuus, virtuaalinen DOM, tilanhallinta

Sisällysluettelo

1 Johdanto 4

2 HTML, CSS ja JavaScript 5

2.1 Document Object Model 7

2.2 JSX 8

3 ReactJS 11

3.1 ReactJS & Vanilla JavaScript kehitys- ja ohjelmistoympäristö 12

3.2 React-sovelluksen luominen 13

3.3 React komponentin arkkitehtuuri 15

3.4 Virtuaalinen DOM reactissa 17

3.5 Tilanhallinta 17

3.5.1 UseState-hook 18

3.5.2 UseReducer-hook 18

3.5.3 UseContext-hook 19

3.5.4 Kolmannen osapuolen kirjastot tilanhallintaan 20

3.5.5 Redux 20

3.5.6 MobX 22

3.5.7 Recoil 23

3.5.8 Jotai 24

3.5.9 Signia 26

3.5.10 XState 26

3.6 Deklaratiivinen ohjelmointi 27

3.7 Uudelleenkäytettävyys React-komponentissa 30

3.8 React developer tools 30

4 Yhteenveto 35

Lähteet 36

1 Johdanto

Mobiililaitteiden kasvanut suosio ja jatkuva teknologinen kehitys ovat lisänneet tarvetta tehokkaille ja käyttäjäystävällisille sovelluksille. Verkkoliikenteestä yli 50 % tapahtuu mobiililaitteilla, ja tämän kehityksen myötä sovelluskehitykselle on asetettu uusia vaatimuksia. Sosiaalisen median ja digitaalisen kulttuurin asiantuntijatoimisto We Are Social raportoi huhtikuussa 2020, että Android- ja iOS-puhelinten käyttäjiä oli 5,16 miljardia, joista 4,57 miljardia käytti internetiä.[1]

Sovelluskehittäminen voidaan jakaa front-end- ja back-end-kehitykseen. Tässä työssä keskitytään front-end-kehitykseen, erityisesti JavaScript-pohjaiseen lähestymistapaan ja sen tueksi kehitettyihin kirjastoihin. Näistä ReactJS on yksi suosituimmista ja vaikutusvaltaisimmista. React on avoimen lähdekoodin JavaScript-kirjasto, jonka tavoitteena on helpottaa käyttöliittymien rakentamista komponenttipohjaisella arkkitehtuurilla.

Tutkimuksen tavoitteena on selvittää, miten ReactJS helpottaa sovelluskehitystä verrattuna pelkkään JavaScriptiin. Tarkastelussa ovat erityisesti Reactin tarjoamat lisäominaisuudet ja hyödyt kehitystyössä.

Työn tutkimuskysymykset ovat:

- Mitä lisäominaisuuksia React tarjoaa verrattuna pelkkään JavaScriptiin?
- Miten Reactin ominaisuudet vaikuttavat sovelluskehityksen tehokkuuteen ja hallittavuuteen?

Tutkimusmenetelmänä käytetään vertailevaa analyysiä, jossa toteutetaan yksinkertainen sovellus ensin pelkällä JavaScriptillä ja sen jälkeen Reactilla. Tuloksia verrataan laadullisesti käytettävyyden, koodin ylläpidettävyyden ja kehitystyön tehokkuuden näkökulmasta. Työ perustuu kirjallisuuteen, dokumentaatioon sekä käytännön toteutukseen.

2 HTML, CSS ja JavaScript

HTML (HyperText Markup Language) ja CSS (Cascading Style Sheets) muodostavat verkkosivujen rakenteellisen ja visuaalisen perustan [5]. HTML kuvaa verkkosivun sisällön rakenteen, kuten tekstin, kuvien ja hyperlinkkien paikat, kun taas CSS määrittelee näiden elementtien ulkoasun, kuten värit, fonttikoot ja asettelun. JavaScript puolestaan täydentää näitä tarjoamalla mahdollisuuden lisätä verkkosivuille dynaamista ja vuorovaikutteista toiminnallisuutta. Esimerkiksi tekstin muuttaminen tai kuvan suurentaminen käyttäjän toiminnan perusteella on tyypillistä JavaScriptin tuottamaa dynaamisuutta.

HTML-kielen ensimmäisen version kehitti CERN:ssä työskennellyt Sir Tim Berners-Lee vuonna 1991 [8]. HTML 1.0 julkaistiin virallisesti vuonna 1993, ja sen tarkoituksena oli jakaa luettavissa olevaa tietoa web-selaimien kautta. Kielen kehitys eteni nopeasti: HTML 2.0 julkaistiin vuonna 1995, sisältäen uusia ominaisuuksia edeltäjänsä lisäksi. HTML 3.0 ja 4.0 seurasivat vuonna 1997, ja nykyisin käytössä oleva HTML5 julkaistiin vuonna 2012 [8].

CSS-kieli syntyi tarpeesta erottaa verkkosivun sisältö sen ulkoasusta. Ensimmäiset ideat CSS:n kehittämiseksi esitti Håkon Wium Lie vuonna 1994, kun HTML:stä puuttui tyylimäärityksiä tukevat ominaisuudet [9]. Lie sai kehitystyöhön tuekseen Argo-verkkoselainta kehittävän Bert Bosin. He esittelivät kieltä useissa alan konferensseissa, ja CSS1 julkaistiin vuonna 1996. Tämän jälkeen kielen kehitystä on jatkanut W3C:n työryhmä, ja CSS2 julkaistiin vuonna 1998. Nykyisin käytössä on CSS3, jota kehitetään edelleen. Kehitystyön laajuus on kasvanut merkittävästi: vuonna 1999 kehittäjiä oli 15, mutta vuonna 2016 jo 115 [9].

JavaScriptin kehitti Netscapen palveluksessa työskennellyt Brendan Eich vuonna 1995. Netscape havaitsi tarpeen tehdä verkkosivuista vuorovaikutteisempia ja ryhtyi kehittämään uutta ohjelmointikieltä, jolla olisi samankaltaisuuksia suosittuun Java-kieleen [10]. Eich kirjoitti ensimmäisen version JavaScriptistä – alun perin nimeltään

Mocha – vain kymmenessä päivässä. Kielen nimi vaihtui samana vuonna ensin LiveScriptiksi ja lopulta JavaScriptiksi [10].

JavaScript on kevyt, olio-orientoitunut komentosarjakieli, jota käytetään yleisesti verkkosivujen toiminnallisuuden ohjelmointiin. Sen käyttö ei rajoitu vain verkkosivuihin, vaan sitä hyödynnetään myös työpöytäsovelluksissa, mobiilisovelluksissa ja peleissä. Verkkosivujen kontekstissa JavaScriptin tärkein ominaisuus on kyky lisätä dynaamista vuorovaikutusta. Käyttäjän toiminnan perusteella sivulle voidaan luoda, muuttaa tai poistaa HTML-elementtejä ja niiden attribuutteja, käsitellä tyylimäärittelyjä sekä hyödyntää 2D- tai 3D-grafiikkaa [10].

Verkkosivujen kehitys aloitetaan kirjoittamalla HTML-rakenteet koodieditorissa. HTML, CSS ja JavaScript ovat kaikkien nykyaikaisten koodieditorien tukemia kieliä. HTML-elementit määritellään tunnistettavilla tageilla, jotka muodostavat rakenteen sisällön ympärille. CSS:n tehtävänä on ohjata näiden HTML-elementtien visuaalisia ominaisuuksia, kuten väriä, kokoa ja sijaintia. Yhdessä nämä kielet muodostavat verkkosivun ulkoasun ja sisällön kokonaisuuden. Kaikki verkkosivujen dynaaminen käyttäytyminen, kuten tapahtumat, päivitykset ja graafiset elementit, toteutetaan JavaScriptin avulla [7].

Kun verkkosivun koodi on valmis tai sitä halutaan esikatsella, se avataan verkkoselaimessa, joka renderöi koodin automaattisesti visuaaliseksi näkymäksi. Kuvassa 1 esitetään HTML-, CSS- ja JavaScript-koodia, jossa painikkeen painaminen suurentaa tekstielementin fonttikokoa. Kuvassa 2 havainnollistetaan tämän koodin toiminnallista tulosta selaimessa: vasemmalla puolella alkuperäinen näkymä ja oikealla tilanne sen jälkeen, kun käyttäjä on painanut painiketta. Toiminnallisuus toteutetaan JavaScriptin avulla, ja tyylimuutos määritellään CSS:n kautta.

```
<!DOCTYPE html>
<html>
<body>

<h2>HTML elementin suurennus</h2>

<p id="demo">Suurennettava elementti</p>
<button type="button"
onclick="document.getElementById('demo').style.fo
ntSize='35px'">Click Me!</button>

</body>
</html>
```

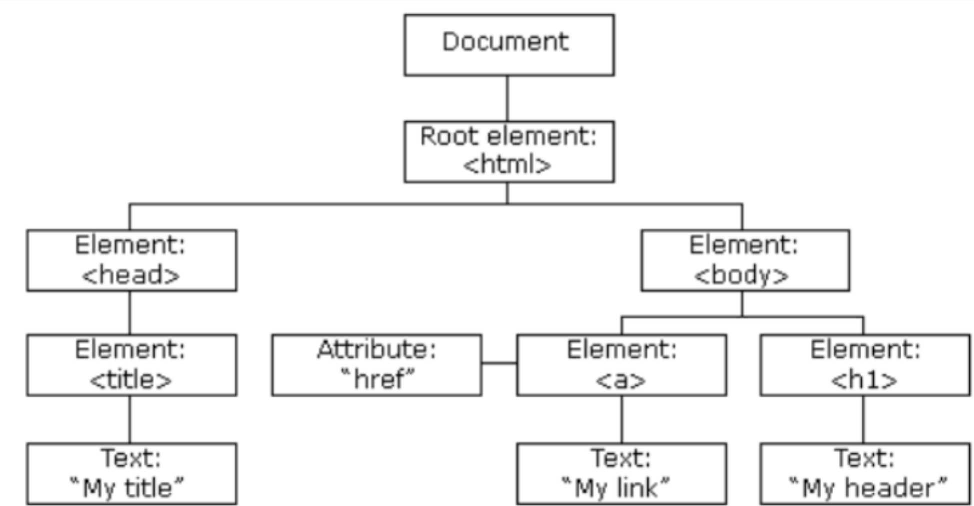
Kuva 1. Näyte HTML-koodista, jossa JavaScriptiä käytetään DOM-elementin fonttikoon muuttamiseen.

Kuva 2. Verkkosivun renderöity näkymä, jossa DOM-manipulaatio muuttaa tekstin fonttikokoa käyttäjän vuorovaikutuksen seurauksena.

2.1 Document Object Model

DOM (Document Object Model) on W3C:n (World Wide Web Consortium) määrittelemä ohjelmointirajapintastandardi, joka mahdollistaa dokumenttien rakenteen ja sisällön käsittelemisen ohjelmallisesti. DOM jakautuu kolmeen osa-alueeseen: Core DOM, XML DOM ja HTML DOM. Core DOM määrittelee yleiset säännöt kaikille dokumenttityypeille, kun taas XML DOM on suunnattu XML-dokumenttien ja HTML DOM HTML-dokumenttien käsittelyyn.

Kun verkkosivu ladataan selaimeen, muodostetaan sen pohjalta DOM-malli, jossa dokumentti jäsennetään haarautuvaksi objektipuuksi. Tämä puumainen rakenne mahdollistaa sen, että selaimet voivat lukea, muokata ja päivittää verkkosivun sisältöä dynaamisesti. Kuvassa 3 on esitetty HTMLdokumentin rakenne DOM-mallina, jossa elementit, attribuutit ja tekstisisällöt esitetään hierarkkisessa muodossa.



Kuva 3. Havainnollistus DOM-mallin puumaisesta rakenteesta, jossa HTML-elementit, attribuutit ja tekstisisällöt jäsentyvät loogiseen kokonaisuuteen.

JavaScriptin avulla DOM:iin voidaan vaikuttaa ohjelmallisesti. Tämä mahdollistaa muun muassa HTML-elementtien ja -attribuuttien lisäämisen, poistamisen tai muuttamisen sekä CSS-tyylien muokkaamisen. Lisäksi JavaScriptillä voidaan luoda uusia HTML-elementtejä reaaliaikaisesti, mikä mahdollistaa verkkosivun rakenteen ja sisällön mukauttamisen käyttäjän toiminnan perusteella. [13]

2.2 JSX

React-kirjastossa HTML:ää ja JavaScriptiä voidaan yhdistää toisiinsa erityisen syntaksin, JSX:n (JavaScript XML), avulla. Perinteisessä web-kehityksessä HTML ja JavaScript pidetään yleensä erillään, mikä johtaa usein laajoihin ja vaikeasti hallittaviin JavaScript-tiedostoihin, jotka sisältävät kaiken toiminnallisuuden. React kuitenkin rikkoo tätä perinteistä erottelua ja mahdollistaa käyttöliittymäkomponenttien rakenteen ja toiminnan yhdistämisen yhteen loogiseen kokonaisuuteen JSX-syntaksin avulla [20].

JSX kehitettiin nimenomaan helpottamaan React-pohjaisten sovellusten rakentamista. Sen avulla voidaan kirjoittaa HTML-elementtejä JavaScript-koodin sisälle ja sijoittaa ne Document Object Modeliin (DOM) ilman tarvetta käyttää perinteisiä menetelmiä, kuten createElement() tai appendChild() [14]. Tämä lähestymistapa parantaa koodin

luettavuutta, modulaarisuutta ja ylläpidettävyyttä erityisesti suurissa ja monimutkaisissa käyttöliittymäsovelluksissa.

Tässä esimerkissä luodaan React-elementti JSX-syntaksin avulla. Muuttujaan myElement sijoitetaan HTML:ää muistuttava <h1>-elementti, joka sisältää tekstin "I Love JSX!". JSX mahdollistaa HTMLrakenteen kirjoittamisen suoraan JavaScriptin sisään.

ReactDOM.createRoot() alustaa juurielementin (root), joka sijoitetaan HTML-tiedoston id='root'elementtiin. Tämän jälkeen root.render(myElement) piirtää JSX-elementin käyttöliittymään.

```
const myElement = <h1>I Love JSX!</h1>;

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

Kuva 4. React-komponentin luominen JSX-syntaksilla. JSX mahdollistaa HTML-muotoisen koodin kirjoittamisen suoraan JavaScriptissä, mikä parantaa koodin luettavuutta.

Tässä esimerkissä sama lopputulos saavutetaan ilman JSX-syntaksia. Reactin createElement()metodilla luodaan h1-elementti, jolle ei anneta attribuutteja ({}), mutta sisältötekstiksi asetetaan "I do not use JSX!".
Tämän jälkeen luotu elementti piirretään juurielementtiin samalla tavalla kuin edellisessä esimerkissä. Tämä tapa on matalamman tason lähestymistapa JSX:ään verrattuna ja antaa hyvän käsityksen siitä, mitä JSX käytännössä abstrahoi.

```
const myElement = React.createElement('h1', {}, 'I do not use JSX!');

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

Kuva 5. React-komponentin luominen ilman JSX-syntaksia käyttäen createElement()-metodia. Tämä lähestymistapa vastaa JSX:n toimintaa, mutta vaatii eksplisiittisemmän määrittelyn.

3 ReactJS

React (tunnetaan myös nimellä ReactJS) on avoimen lähdekoodin JavaScript-kirjasto, jonka Facebook kehitti käyttöliittymien rakentamista varten. Reactia pidetään yhtenä merkittävimmistä ja vaikutusvaltaisimmista teknologioista modernissa web-kehityksessä [3]. Sen suosio kasvoi nopeasti sen jälkeen, kun se julkaistiin avoimena lähdekoodina vuonna 2013. Reactin suosion taustalla ovat erityisesti sen tarjoama yksinkertainen ja tehokas tapa rakentaa käyttöliittymäkomponentteja sekä suorituskykyä parantava arkkitehtuuri.

React hyödyntää niin sanottua virtuaalista DOM:ia (Virtual DOM), joka on JavaScriptillä toteutettu olioesitys sovelluksen todellisesta DOM-rakenteesta. Virtuaalinen DOM mahdollistaa tehokkaan renderöinnin vertaamalla muutoksia nykytilan ja uuden tilan välillä ja päivittämällä vain ne komponentit, jotka ovat muuttuneet [3]. Tämä tekee renderöinnistä huomattavasti nopeampaa verrattuna perinteisiin JavaScript-kirjastoihin, joissa koko DOM saatetaan renderöidä uudelleen.

Reactin juuret ulottuvat vuoteen 2010, jolloin Facebook integroi XHP-nimisen PHP-laajennuksen front-end-kehitykseen helpottaakseen käyttöliittymäkoodin hallintaa. Vuonna 2011 Jordan Walke, Facebookin ohjelmistokehittäjä, kehitti Reactin ensimmäisen prototyypin nimeltä FaxJS ratkaisuksi ongelmiin, joita Facebook kohtasi mainostenhallintasovelluksensa laajentuessa. Sovelluksen jatkuva kasvu vaikeutti sen ylläpitoa, mikä johti tarpeeseen uudenlaiseen lähestymistapaan komponenttipohjaisessa kehityksessä. Tästä tarpeesta syntyi React [4].

React julkaistiin avoimena lähdekoodina vuonna 2013 Jordan Walken esiteltyä sen JSConf US konferenssissa. Vuonna 2014 React alkoi saavuttaa laajempaa huomiota ja käyttöä kehittäjäyhteisössä. Facebook pyrki vastaamaan yhteisön epäilyihin vakauden suhteen lanseeraamalla viestin "How is React stable?" ja julkaisemalla siihen liittyviä työkaluja, kuten React Developer Tools ja React Hot Loader sekä tukemalla Atom-editoria ja ReativeX.io:ta [4].

Vuonna 2015 Reactista julkaistiin uusi versio mobiilikehitystä varten, nimeltään React Native. Se esiteltiin React.js-konferenssissa ja mahdollisti React-komponenttien hyödyntämisen iOS- ja Androidsovelluksissa. Samana vuonna suuret teknologiayritykset, kuten Netflix ja Airbnb, siirtyivät käyttämään Reactia, mikä vahvisti sen asemaa vakaan ja luotettavan teknologian asemassa [4]. Vuonna 2016 Reactista tuli valtavirran teknologia web-kehityksessä. Kehityksen tueksi julkaistiin useita uusia työkaluja ja kirjastoja, kuten MobX tilanhallintaan, Draft.js tekstieditoreihin, React Storybook komponenttien kehittämiseen, virheenkäsittelyn Error Code System sekä käyttöliittymäkomponenttikirjasto Blueprint [4]. Tästä vuodesta lähtien React on säilyttänyt asemansa suosituimpana JavaScript-kirjastona [3].

React on suunniteltu toimimaan saumattomasti yhdessä muiden nykyaikaisten web-teknologioiden kanssa, kuten Reduxin, TypeScriptin ja GraphQL:n. Lisäksi sitä voidaan helposti yhdistää muihin ohjelmointiympäristöihin, kuten Node.js- ja Express-pohjaisiin back-end-ratkaisuihin, mikä tekee siitä erittäin joustavan valinnan sekä yksinkertaisiin että laajoihin sovelluskokonaisuuksiin.

3.1 ReactJS & Vanilla JavaScript kehitys- ja ohjelmistoympäristö

Vaadittava ohjelmointiympäristö, eli käyttäjän tietokone sekä pakolliset ladattavat ohjelmat Reactin pyörittämiseksi ovat laajat. Tekstieditoreja on paljon, suurin osa on ilmaisia sekä lähdekoodiltaan avoimia, kuten Visual Studio Code. VS Code on mahdollista ladata Windows, Mac sekä Linux käyttöjärjestelmille VS Coden nettisivuilta. Valitsin tähän tutkielmaan koodieditoriksi VS Coden, sillä se oli jo entuudestaan tuttu. VS Code on käytössä työmarkkinoilla sekä koodarien keskuudessa. Kehitysympäristönä toimii alustariippumaton Node.js JavaScript-tulkki, joka on mahdollista ladata Windows, Mac sekä Linux käyttöjärjestelmille. Node.js asennuksen mukana tulee node package manager npm, joka mahdollistaa erilaisten pakettien asentamisen npmrekisteristä. Noden sekä npm package manager ovat pakollisia asennuksia React-kirjaston käyttämiseen.

3.2 React-sovelluksen luominen

React sovelluksen voi luoda kuudella eri tavalla, jotka ovat Create-React-App (CRA), Next.js, Gatsby, Nwb, Razzle sekä <script> tagin lisääminen HTML sivulle. [12] Luomme sovelluksen tässä työssä CRA käyttäen. CRA on kehys, joka tekee Reactin SPA helppoja sekä yksinkertaisia. [12] CRA on hyvä valinta aloittelijoille Reactin kanssa, sillä se muodostaa projektille kansiorakenteen ja asentaa kaikki tarvittavat paketit. CRA haetaan npm-rekisteristä ja voidaan suorittaa ilman erillisiä asennuksia käyttämällä npx komentoa. Tässä tutkielmassa hyödynnetään Create React App -työkalua (CRA), ja React-sovellukselle annetaan nimeksi kandiapp, jolloin komentoriville kirjoitetaan npx create-reactapp kandiapp (ks. Kuva 6).

```
C:\Users\tmaun>npx create-react-app kandiapp

Creating a new React app in C:\Users\tmaun\kandiapp.

Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts with cra-template..
```

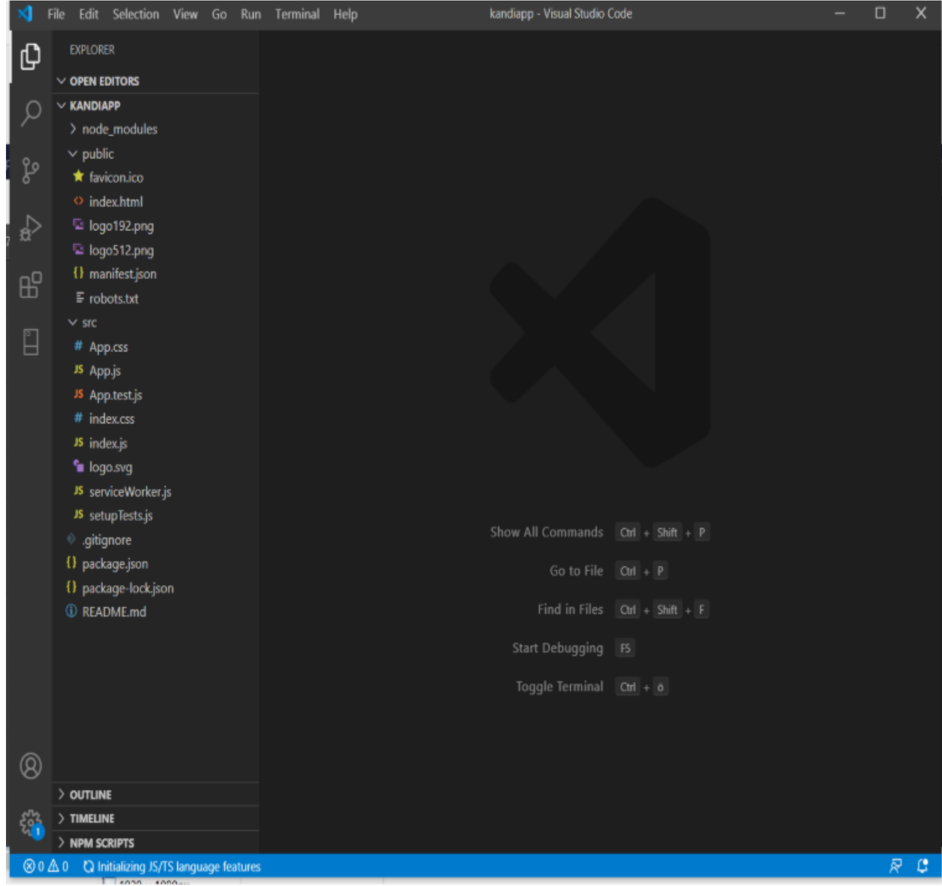
Kuva 6. React-sovelluksen luominen komennolla npx create-react-app kandiapp.

Tämän jälkeen pakettien asentaminen alkaa, ja muutaman minuutin kuluttua, kun noin 68 pakettia on asennettu, React-sovellus on valmis aloitettavaksi. Komentoriville kirjoitetaan ehdotusten mukaisesti cd kandiapp ja npm start sovelluksen käynnistämiseksi (ks. Kuva 7).

```
We suggest that you begin by typing:

  cd kandiapp
  npm start

Happy hacking!

C:\Users\tmaun>cd kandiapp

C:\Users\tmaun\kandiapp>npm start

> kandiapp@0.1.0 start C:\Users\tmaun\kandiapp
> react-scripts start

i ｢wds｣: Project is running at http://172.20.10.2/
i ｢wds｣: webpack output is served from
i ｢wds｣: Content not from webpack is served from C:\Users\tmaun\kandiapp\public
i ｢wds｣: 404s will fallback to /
Starting the development server...
Compiled successfully!

You can now view kandiapp in the browser.

  Local:            http://localhost:3000
```

Kuva 7 cd kandi app, npm start

jonka jälkeen sovellus aukeaa välilehteen localhost:3000 osoitteeseen, jossa näkyy tehdyt muutokset, kun tekstieditoria päivitetään. Avataan tekstieditori VS Code ja kansio kandiapp. Alla olevassa kuvassa näkyy kuinka CRA on asentanut tarvittavat paketit.



Kuva 8 CRA projektin hakemistorakenne VS Codessa.

3.3 React komponentin arkkitehtuuri

React keskittyy käyttöliittymien rakentamiseen komponenttipohjaisella arkkitehtuurilla. Tämän avulla kehittäjät voivat rakentaa monimutkaisia verkkosovelluksia yhdistämällä pieniä, modulaarisia komponentteja. Jokainen React-sovelluksen komponentti edustaa itsenäistä ja uudelleenkäytettävää käyttöliittymäelementtiä, joka vastaa oman tilansa hahmontamisesta ja hallinnasta. Reactin komponenttipohjainen arkkitehtuuri mahdollistaa paremman koodin organisoinnin ja kapseloinnin, mikä helpottaa sovellusten hallintaa ja ylläpitoa [18]. Kun komponentit on rakennettu hyvin, sovelluksen käyttöliittymä voidaan jakaa pienempiin, modulaarisiin osiin, joita on helpompi kehittää, testata ja korjata. Tämä lähestymistapa verkkosovellusten rakentamiseen on nopeasti saavuttanut suosion kehittäjien keskuudessa, ja sitä

pidetään mullistavana käyttöliittymäkehityksen maailmassa [18]. Komponenttipohjaisen arkkitehtuurin hyödyntäminen Reactissa tuo mukanaan useita merkittäviä etuja.

- **Parempi koodin organisointi ja modulaarisuus**: Reactin komponenttipohjainen arkkitehtuuri jakaa sovelluksen käyttöliittymän pienempiin, uudelleenkäytettäviin komponentteihin. Tämä modulaarisuus johtaa parempaan koodin organisointiin ja puhtaampaan, ylläpidettävämpään koodikantaan [18].
- **Komponenttien uudelleenkäytettävyys:** Komponentit voidaan jakaa sovelluksen eri osien tai useiden sovellusten kesken, mikä vähentää koodin päällekkäisyyttä ja edistää parhaita käytäntöjä, kuten DRY (Don’t repeat yourself) -periaatetta [21].
- **Parannettu testaus ja ylläpito:** Koska komponentit ovat pieniä ja keskittyneitä, yksittäisten toimintojen testien kirjoittaminen ja ylläpito helpottuu. Lisäksi yhden komponentin päivittämisellä ei ole ei-toivottuja sivuvaikutuksia muille, mikä lisää koko sovelluksen vakautta [20].
- **Huolenaiheiden erottaminen:** Jokainen React-sovelluksen komponentti on vastuussa omasta renderöinnistään ja tilanhallinnastaan. Tämä pakottaa huolenaiheiden selkeän erottamisen ja antaa kehittäjille mahdollisuuden käsitellä käyttöliittymän monimutkaisuutta pala kerrallaan [18].
- **JavaScript:** Sovelluksen rakentaminen pelkästään JavaScriptillä voi olla monimutkainen ja vähemmän modulaarinen prosessi, erityisesti suurissa sovelluksissa. Koko käyttöliittymä voi olla hajallaan, mikä tekee koodista vaikeasti ylläpidettävää ja testattavaa.
- **React:** React käyttää komponenttipohjaista lähestymistapaa, jossa käyttöliittymän osat (komponentit) ovat erillisiä ja itsenäisiä. Komponentit voivat olla pieniä, uudelleenkäytettäviä ja helposti hallittavia. Tämä parantaa koodin selkeyttä, uudelleenkäytettävyyttä ja ylläpidettävyyttä [20].

3.4 Virtuaalinen DOM reactissa

Yksi ReactJS:n keskeisimmistä ominaisuuksista on sen hyödyntämä virtuaalinen DOM (Virtual DOM, VDOM). Virtuaalinen DOM on ohjelmointimalli, jossa käyttöliittymän rakenteellinen esitys pidetään muistissa ja synkronoidaan todellisen selaimen DOMin kanssa kirjaston, kuten ReactDOM:n, avulla [11]. Tämän mallin ansiosta React kykenee optimoimaan DOM-päivitykset: kun komponentin tila muuttuu, React ei suoraan päivitä selaimen DOMia, vaan se luo virtuaalisen kopion ja vertaa sitä aiempaan versioon. Ainoastaan muuttuneet osat päivitetään todelliseen DOMiin, mikä parantaa sovelluksen suorituskykyä merkittävästi, erityisesti laajoissa ja monimutkaisissa käyttöliittymissä [11]. Reactin käyttöön ottama VDOM mahdollistaa myös deklaratiivisen ohjelmointityylin. Kehittäjä määrittelee komponenttien halutun tilan, ja React huolehtii siitä, että käyttöliittymä vastaa kyseistä tilaa. Tämä lähestymistapa vähentää manuaalisten DOM-muokkausten, attribuuttien hallinnan ja tapahtumakäsittelyn tarvetta [15].

Teknisen toteutuksen osalta Reactin virtuaalinen DOM ei rajoitu pelkkiin React-elementteihin, vaan se sisältää myös sisäisiä olioita, joita kutsutaan fibereiksi. Fiber-arkkitehtuuri, joka esiteltiin React 16 versiossa, toimii täsmäytysmoottorina ja mahdollistaa komponenttien asteittaisen ja joustavamman renderöinnin [11].

Pelkällä JavaScriptillä toteutetussa kehityksessä DOMin päivittäminen voi aiheuttaa suorituskykyongelmia, erityisesti suurissa sovelluksissa, joissa DOM-rakenne on monimutkainen. Jokainen muutos DOMissa johtaa potentiaalisesti koko rakenteen uudelleenrenderöintiin, mikä kasvattaa resurssien kulutusta. Reactin virtuaalinen DOM ratkaisee tämän haasteen tehokkaasti optimoimalla päivitykset komponenttitasolla ja vähentämällä tarpeettomia muutoksia todelliseen DOMiin [11].

3.5 Tilanhallinta

Tila (State) on Javascript objekti, joka voi muuttaa komponentin toimintaa käyttäjän toimesta. Reactsovellukset on rakennettu komponenteista, jotka hallitsevat oman tilansa. Tämä toimii pienissä sovelluksissa, mutta sovelluksen monimutkaisuuden

kasvaessa komponenttien välisten jaettujen tilojen käsittely muuttuu yhä monimutkaisemmaksi ja ongelmallisemmaksi. [16]

3.5.1 UseState-hook

useState-hook on yksinkertaisin Reactin tarjoama API vuorovaikutukseen komponentin tilan kanssa. `const [count, setCount] = useState(0);`

Tämä koodirivi luo tilamuuttujan nimeltä 'count' ja tilaa päivittävän funktion nimeltä 'setCount'. Tilan alkuarvoksi on asetettu arvo 0.

```
setCount(count + 1);
```

Yllä oleva koodi kasvattaa tilamuuttujan arvoa yhdellä.

```
<p>You clicked {count} times</p>
```

Tässä HTML-elementissä hyödynnetään tilamuuttujaa 'count' näyttämään käyttäjälle montako kertaa painiketta on klikattu.

3.5.2 UseReducer-hook

useReducer-hookin avulla voidaan hallita komponentin tilaa lähettämällä toimintoja (actioneita) ja käsittelemällä ne reducer-funktiossa.

```
const [state, dispatch] = useReducer(reducerFunc, 0);
```

Yllä oleva koodi kutsuu useReducer-hookia, jolle annetaan kaksi argumenttia: reducer-funktio ja alkutila (tässä 0). Hook palauttaa nykyisen tilan (state) ja dispatch-funktion, jonka avulla tilaa voidaan päivittää. Tilapäivitys tehdään kutsumalla dispatch-funktiota action-objektilla, joka kuvaa haluttua toimintaa.

```
dispatch({ type: 'INCREMENT' })
```

React kutsuu reducer-funktiota edellä esitetyllä action-objektilla, jonka perusteella tila päivitetään.

```
function reducerFunc(state, action) {     switch (action.type) {        case 'INCREMENT':        return state + 1;      case 'DECREMENT':        return state - 1;      case 'RESET':        return 0;      case 'SET':
       return action.val;
   }
}
```

Yllä oleva reducer-funktio vastaanottaa nykyisen tilan ja action-objektin. Sen perusteella se päivittää tilan eri toiminnon mukaan: kasvattaa, pienentää, nollaa tai

asettaa tiettyyn arvoon. Tapahtumaketjun lopussa reducerin palauttama arvo on uusi tilamuuttujan arvo.

3.5.3 UseContext-hook

useContext hook tulee kuvaan tilanhallinnassa. Yleisenä ajatuksena on tallentaa tieto, jota komponentit voivat käyttää, mutta malli ja käyttötapaukset ovat erilaisia.

Kun useState ja useReducer käytetään komponenti tilan hallintaan, useContextia käytetään komponenttien kesken jaetun tilan hallintaan. Sen päätarkoitus on välittää komponentilta data sen lapsikomponentille, joka on useita tasoja komponenttipuuta alempana. Yksi kontekstin ensisijaisista käyttötapauksista on sovelluksen teeman hallinta.

Komponenttipuun yläosassa olevaan komponenttiin luodaan konteksti: `export const themeContext = createContext('light');`

Yllä olevassa esimerkissä luodaan uusi konteksti nimeltä 'themeContext', jonka oletusarvoksi on asetettu merkkijono 'light'. Tämä tehdään sovelluksen ylemmällä tasolla, jotta arvoa voidaan jakaa komponenttipuun läpi.

```
import { useContext } from 'react';
import { ThemeContext } from './App.js';

export default function Button({ children })
{    const theme = useContext(ThemeContext);
// ...
}
```

Tässä esimerkissä komponentti 'Button' tuo ThemeContextin ja hyödyntää Reactin useContext-hookia käyttääkseen contextin arvoa. Tämä mahdollistaa teemaan liittyvän tilan hyödyntämisen missä tahansa komponenttipuun kohdassa.

```
<ThemeContext.Provider value={theme}>
   {children}
</ThemeContext.Provider>
```

Kontekstin käyttö edellyttää, että komponenttipuu kääritään Provider-komponenttiin, joka tarjoaa context-arvon sen alikomponenteille. Kun Providerin 'value'-prop päivitetään, kaikki sitä käyttävät alakomponentit päivittyvät automaattisesti uuteen arvoon. [16]

3.5.4 Kolmannen osapuolen kirjastot tilanhallintaan

Tilanhallintaratkaisu on järkevä, kun tilan laajuus on vain yksi komponentti (useStaten ja useReducerin tapauksessa) tai kun tilaa on tarkoitus käyttää loppupään komponentissa (useContextin tapauksessa), nämä rajoitukset vaikeuttavat näiden ratkaisujen käyttöä suuremmissa sovelluksissa, joissa tilan luomis- ja kulutusjärjestystä ei ole määritelty. Tässä kolmannen osapuolen kirjastot tulevat käyttöön.[16]

3.5.5 Redux

Redux on yksi ensimmäisistä ja edelleen yleisesti käytetyistä tilanhallintakirjastoista Reactekosysteemissä. Se tarjoaa keskitetyn tavan hallita sovelluksen tilaa yhdistämällä useita tilaviipaleita (state slices) yhdeksi globaaliksi tilaksi (engl. store). Reduxissa keskeisiä käsitteitä ovat store, reducerit ja toiminnot (actions), joiden avulla tila saadaan hallittavaksi ja ennustettavaksi [23].

Redux Toolkitin `configureStore` -funktiota käytetään usein Redux-sovelluksen storessa yhdistämään useita reducereita, joita koko sovellus voi hyödyntää. Reducerit määritellään ns. "slice" -olioissa, joissa asetetaan alkutila ja tilaa muuttavat funktiot. Esimerkiksi todosReducer ja userReducer voidaan yhdistää yhdeksi tilahierarkiaksi. Tämä mahdollistaa modulaarisen rakenteen, jossa jokainen slice vastaa tietystä osasta sovelluksen tilaa.

Koko React-sovellus kääritään `Provider` -komponenttiin, joka mahdollistaa Reduxin tarjoaman tilan saatavuuden kaikkialla sovelluksen komponenttipuussa. Tilaan päästään käsiksi `useSelector` -hookin avulla, ja muutoksia tehdään `useDispatch` -hookilla. Näiden hookien avulla komponentit voivat lukea ja muokata tilaa ilman suoraa sidosta sen rakenteeseen.

Tämä malli parantaa koodin ylläpidettävyyttä ja skaalautuvuutta. Kun uusia ominaisuuksia lisätään, niitä varten voidaan luoda uusia sliceja, jotka yhdistetään

olemassa olevaan storeen. Reduxin deterministinen toimintamalli ja debuggauksen mahdollistavat työkalut, kuten Redux DevTools, ovat tehneet siitä suositun valinnan suurissa ja monimutkaisissa React-projekteissa [24].

Reduxin käyttöönotto: koodiesimerkkejä

```
import { configureStore } from '@reduxjs/toolkit';
import todosReducer from '../features/todos/todosSlice';
import userReducer from '../features/user/userSlice';

export const store =
configureStore({   reducer: {
todos: todosReducer,     user:
userReducer
   }
});
```

Yllä oleva esimerkki näyttää, kuinka useita slice-reducereita yhdistetään Redux-storeen käyttämällä Redux Toolkitin configureStore-funktiota.

```
import { Provider } from 'react-redux';
import { store } from './app/store';

<Provider store={store}>
   <App />
</Provider>
```

Sovelluksen juurikomponentti kääritään Provider-komponenttiin, jotta Reduxin tila on käytettävissä kaikissa alikomponenteissa.

```
import { useSelector, useDispatch } from 'react-redux';
import { increment } from './features/counter/counterSlice';

const count = useSelector((state) => state.counter.value);
const dispatch = useDispatch();

<button onClick={() => dispatch(increment())}>
   Increment
</button>;
```

Tässä esimerkissä tilan arvoa luetaan useSelector-hookin avulla ja tilaa päivitetään useDispatchhookin kautta. Toimintoa dispatchataan painikkeen tapahtumankäsittelijässä.

3.5.6 MobX

MobX on reaktiivinen tilanhallintakirjasto, jota käytetään usein vaihtoehtona Reduxille Reactsovelluksissa. Sen keskeinen periaate on tilan automaattinen synkronointi käyttöliittymän kanssa: kun havaittava tila (observable) muuttuu, kaikki sitä käyttävät näkymät (komponentit) päivittyvät automaattisesti. Tämä saavutetaan MobX:n reaktiivisen ohjelmointimallin avulla, jossa muutokset havaitaan ja välitetään riippuvuuksien kautta eteenpäin [25].

MobX tekee eron tilan, näkymän ja toiminnan (state, view, action) välillä. Havaittava tila määritellään `observable`- avainsanalla, ja sitä muutetaan `action`-funktioiden avulla. Reaktiivinen näkymä voidaan toteuttaa käyttämällä `observer` -funktiolla käärittyjä komponentteja, jotka seuraavat automaattisesti tilan muutoksia.

MobX mahdollistaa yksinkertaisen ja intuitiivisen tavan hallita tilaa ilman tarvetta reducer-rakenteisiin tai toimintojen manuaaliseen käsittelyyn. Tämä tekee siitä helposti lähestyttävän erityisesti pienemmissä tai keskisuurissa sovelluksissa, joissa ei tarvita monimutkaista tilarakennetta tai historian seurantaa.

MobX:n käyttöönotto: koodiesimerkkejä

```
import { makeAutoObservable } from "mobx";

class CounterStore {
count = 0;

  constructor() {
     makeAutoObservable(this);
   }

  increment() {
this.count++;
   }
}

export const counterStore = new CounterStore();
```

Yllä määritellään MobX-store, jossa on yksi tilamuuttuja (count) ja sen muuttamiseen tarkoitettu metodi (increment). Konstruktorissa kutsutaan makeAutoObservable, joka tekee kaikista kentistä ja metodeista automaattisesti havaittavia ja toimintakelpoisia.

```
import { observer } from "mobx-react-lite";

const CounterView = observer(() =>
{   return (     <div>
       <p>Count: {counterStore.count}</p>
       <button onClick={() => counterStore.increment()}>Increment</button>
     </div>
   );
});
```

Tässä komponentti CounterView on kääritty observer-funktiolla, jotta se päivittyy automaattisesti aina, kun counterStore.count muuttuu. Näin saavutetaan reaktiivinen käyttäytyminen ilman manuaalista tilan käsittelyä.

3.5.7 Recoil

Recoil on Reactin tilanhallintaan tarkoitettu kirjasto, joka perustuu atomiseen tilanhallintamalliin. Toisin kuin Redux tai MobX, joissa tila hallitaan keskitetysti yhdessä rakenteessa tai luokassa, Recoil hajottaa tilan itsenäisiin osiin nimeltä atomit. Atomi on tilan perusyksikkö, jota voidaan lukea ja muokata mistä tahansa komponentista käsin. Kun komponentti lukee atomin arvon, se tilaa itsensä kyseiselle atomille ja renderöidään automaattisesti uudelleen aina, kun atomin arvo muuttuu. [25]

Ennen kuin Recoil-atomeja voidaan käyttää, React-sovelluksen juurikomponentti on käärittävä RecoilRoot-komponentin sisään. Tämä toimii samalla tavalla kuin Reduxin Provider-komponentti ja mahdollistaa Recoil-tilan käytön kaikissa alikomponenteissa. [25]

Koodiesimerkkejä Recoilin käytöstä

```
import { atom } from 'recoil';

const nameState =
atom({   key:
'nameState',
default: '', });
```

Tässä määritellään Recoil-atomi nimeltä nameState, joka tallentaa merkkijonon.

Atomille määritetään uniikki key ja oletusarvo.

```
import { useRecoilState } from 'recoil';

function NameInput() {   const [name, setName] =
useRecoilState(nameState);

  return <input value={name} onChange={(e) => setName(e.target.value)} />;
}
```

Komponentti NameInput lukee ja päivittää atomia useRecoilState-hookin avulla.

Käyttäjän syöttämä arvo tallentuu automaattisesti globaaliin tilaan.

```
import { selector } from 'recoil';

const greetingState = selector({
key: 'greetingState',   get: ({
get }) => {     const name =
get(nameState);
     return `Hello, ${name || 'stranger'}!`;
   },
});
```

Selectorin avulla voidaan johdetusti laskea arvoja atomien perusteella. Tässä palautetaan tervehdys, joka perustuu nameState-atomin arvoon.

```
import { useRecoilValue } from 'recoil';

function Greeting() {
   const greeting =
useRecoilValue(greetingState);   return
<h1>{greeting}</h1>; }
```

Greeting-komponentti lukee selectorin arvon Recoilin useRecoilValue-hookin avulla.

Käyttöliittymä päivittyy automaattisesti, kun tila muuttuu.

3.5.8 Jotai

Jotai on kevyt ja minimaalinen tilanhallintakirjasto Reactille, joka tarjoaa yksinkertaisen API:n atomipohjaiseen tilanhallintaan. Se on suunniteltu erityisesti moderneihin React-sovelluksiin, joissa halutaan hallita komponenttikohtaista tilaa ilman monimutkaista konfiguraatiota tai riippuvuutta ulkoisista työkaluista [26].

Jotai toimii samalla atomiperiaatteella kuin Recoil, mutta keskittyy yksinkertaisuuteen. Jokainen atomi määrittelee yksittäisen tilaelementin. Komponentit voivat lukea ja muuttaa atomeja `useAtom`-hookin avulla. Lisäksi Jotai tukee johdettua tilaa käyttämällä toisten atomien arvoista laskettavia arvoja. Tämän ansiosta tilojen välinen riippuvuus voidaan kuvata suoraan funktionaalisesti.

Jotai ei vaadi käärittävää provider-komponenttia, ja se integroituu hyvin Reactin natiivien hookien kanssa. Tämä tekee siitä houkuttelevan vaihtoehdon erityisesti pieniin sovelluksiin, joissa halutaan säilyttää yksinkertaisuus ilman tilanhallintarakenteiden ylikuormitusta.

Koodiesimerkkejä Jotai käytöstä

```
import { atom } from 'jotai';

const countAtom = atom(0);
```

Yksinkertainen atomi, jonka oletusarvona on kokonaisluku 0.

```
import { useAtom } from 'jotai';
import { countAtom } from './store';

function Counter() {
   const [count, setCount] = useAtom(countAtom);
return (
     <>
       <h1>{count}</h1>
       <button onClick={() => setCount(count + 1)}>Increment</button>
     </>
   );
}
```

Counter-komponentti lukee ja päivittää countAtomin arvoa. Komponentti päivittyy automaattisesti, kun tila muuttuu.

```
import { atom } from 'jotai';

const doubleCountAtom = atom((get) => get(countAtom) * 2);
```

Johdettu atomi käyttää toisen atomin arvoa ja laskee siitä uuden arvon. Tässä esimerkissä arvo tuplataan.

3.5.9 Signia

Signia on uusi ja vaihtoehtoinen kirjasto Reactin tilanhallintaan. Se eroaa muista kirjastoista, kuten Recoil ja Jotai, lähestymistapansa perusteella: Signia ei käytä atomeja tai observable-mekanismeja, vaan hyödyntää signaaleja (signals). Signaali on puhdas, reaktiivinen arvo, jota voidaan tarkkailla, ja joka päivittyy automaattisesti, kun siihen liittyvät arvot muuttuvat. Signia mahdollistaa myös signaaleihin perustuvien atomeiden luomisen. Tilan päivittäminen tapahtuu suoraan kutsumalla atomin omaa metodifunktiota, mikä tekee tilan hallinnasta yksinkertaista ja läpinäkyvää.

Kuten Recoilissa ja Jotaissa, myös Signiassa on tuki laskennalliselle tilalle (computed state). Tämä mahdollistaa johdettujen tilojen luomisen käyttämällä muita signaaleja ja niiden arvoja atom.value viittauksella. Tämä tekee Signiasta erittäin ilmeikkään ja joustavan valinnan tilanhallintaan reaktiivisissa käyttöliittymissä. Signia sopii erinomaisesti yksinkertaisten ja keskenään riippuvaisten tilojen hallintaan, ja tarjoaa hyvän suorituskyvyn sekä selkeän ohjelmointimallin ilman tarvetta ulkoisille riippuvuuksille.[16]

3.5.10 XState

XState on kirjasto tilakoneiden ja tilakaavioiden mallintamiseen JavaScript- ja TypeScriptsovelluksissa. Se perustuu muodolliseen tilakonemalliin (finite state machine, FSM), joka tarjoaa eksplisiittisen ja ennustettavan tavan hallita sovelluksen tiloja ja tilasiirtymiä. Tämä tekee siitä erityisen hyödyllisen sovelluksissa, joissa on paljon käyttäjän vuorovaikutusta tai monimutkaisia tilasiirtymiä [27].

XState integroituu Reactin kanssa tarjoamalla `useMachine` -hookin, jonka avulla komponentit voivat käyttää tilakonetta samalla tavalla kuin Reactin `useState` -hookia. Tilakone määritellään `createMachine` -funktiolla, jossa kuvataan tilat (`states`) ja niiden siirtymät (`transitions`). Tilakone on aina deterministinen, mikä tekee siitä helposti testattavan ja virheensietokykyisen.

Toisin kuin monet muut tilanhallintakirjastot, XState erottelee selkeästi tilan, siirtymän ja tapahtuman, mikä parantaa sovelluksen rakennetta ja tekee sen käyttäytymisestä dokumentoitavaa. Tämä rakenne auttaa erityisesti suurissa sovelluksissa, joissa käyttäjän vuorovaikutuksen virrat ovat monimutkaisia tai niihin liittyy useita sääntöjä.

Koodiesimerkkejä XState-käytöstä

```
 import { createMachine } from
'xstate';

const toggleMachine =
createMachine({   id: 'toggle',
initial: 'inactive',   states: {
inactive: {
       on: { TOGGLE: 'active' }
     },
active: {
       on: { TOGGLE: 'inactive' }
     }
   }
});
```

Tässä määritellään yksinkertainen tilakone, jolla on kaksi tilaa: 'inactive' ja 'active'. Tilan vaihto tapahtuu TOGGLE-tapahtumalla.

```
import { useMachine } from '@xstate/react';
import { toggleMachine } from './toggleMachine';

function ToggleButton() {
   const [state, send] = useMachine(toggleMachine);

  return (
     <button onClick={() => send('TOGGLE')}>
       {state.matches('inactive') ? 'Off' : 'On'}
     </button>
   );
}
```

Komponentissa ToggleButton käytetään useMachine-hookia tilakoneen käynnistämiseen ja tapahtumien lähettämiseen. Komponentti reagoi tilan muutoksiin automaattisesti.

3.6 Deklaratiivinen ohjelmointi

Imperaktiivinen lähestymistapa on, kun annat vaiheittaiset ohjeet halutun käyttöliittymän saavuttamiseksi. Imperaktiivinen ratkaisu jossa määrittelemme jokaiselle vuorovaikutukselle vaiheittaiset DOM-mutaatiot halutun käyttöliittymän tilan saavuttamiseksi.

```
const container = document.querySelctor(“.container”)
const btn = document.querySelector(“.btn”)

const listener1 = () => {
  btn.style.backgroundColor = “#DB2777”;
  btn.style.innerText = “Are you sure?”
  btn.removeEventListener(“click”, listener1)
  btn.addEventListener(“click”, listener2)
}

const listener2 = () => {
  container.innerHTML = ‘🦄’
}

btn.addEventListener(“click”, listener1)
```

Kuva 9 deklaratiivinen ohjelmointi

Deklaratiivinen lähestymistapa on, kun kuvaat halutun käyttöliittymän lopullisen tilan. Deklaratiivinen React-ratkaisu. Sen sijaan, että määriteltäisiin vaiheittain halutun käyttöliittymän saavuttamiseksi, määrittelen lopullisen käyttöliittymän, jonka haluan kullekin scenelle.

```
cconst [scene, setScene] = useState('button')

if (scene === 'button') {
  return (
    <Button
      blue
      onClick={() => setScene('question')}>
      Show the unicorn
    </Button>
  )
}

if (scene === 'question') {
  return (
    <Button
      pink
      onClick={() => setScene('unicorn')}>
      Are you sure?
    </Button>
  )
}

if (scene === 'unicorn') {
  return (
    <span>🦄</span>
  )
}
```

Kuva 10 deklaratiivinen ohjelmointi

Mikä tekee siitä deklaratiivisen, on se, että määrittelemme lopullisen tilan käyttöliittymän esitykselle.[19]

JavaScript: Pelkällä JavaScriptillä UI-päivitykset voivat olla imperatiivisia, eli kehittäjän täytyy itse huolehtia siitä, kuinka ja milloin DOM-päivitykset tehdään.
React: React käyttää deklaratiivista lähestymistapaa, jossa määritellään, millainen UI pitäisi olla tietyn tilan perusteella. React huolehtii itse siitä, miten ja milloin DOM-päivitykset tehdään, mikä tekee koodista selkeämpää ja helpommin ymmärrettävää.

3.7 Uudelleenkäytettävyys React-komponentissa

Uudelleenkäytettävyys ohjelmoinnissa tarkoittaa ohjelmakoodin osien suunnittelemista siten, että niitä voidaan käyttää uudelleen eri konteksteissa tai sovelluksissa ilman merkittäviä muutoksia. Reactkomponentit on suunniteltu erityisesti tätä periaatetta silmällä pitäen. Jokainen komponentti voidaan kapseloida, testata ja käyttää uudelleen osana laajempia käyttöliittymärakenteita [28].

Uudelleenkäytettävät komponentit tukevat myös periaatetta "Don't Repeat Yourself" (DRY), mikä vähentää koodin toistoa ja parantaa sovelluksen ylläpidettävyyttä ja skaalautuvuutta. Hyvin suunnitellut komponentit ovat riippumattomia toisistaan ja ottavat vastaan datansa ja toimintonsa propsien kautta. Tämä lisää niiden joustavuutta ja uudelleenkäytettävyyttä eri näkymissä tai jopa muissa projekteissa [29]. Erityisesti suuremmissa sovelluksissa komponenttien uudelleenkäytettävyys mahdollistaa nopeamman kehitystyön ja yhtenäisen käyttöliittymän.

Koodiesimerkkejä uudelleenkäytettävästä komponentista

```
function Button({ label, onClick }) {    return
<button onClick={onClick}>{label}</button>;
}

// Käyttö eri osissa sovellusta
<Button label="Tallenna" onClick={handleSave} />
<Button label="Poista" onClick={handleDelete} />
```

Yllä oleva Button-komponentti on suunniteltu uudelleenkäytettäväksi. Se vastaanottaa tekstin (label) ja toiminnon (onClick) propsien kautta, jolloin sitä voidaan käyttää eri yhteyksissä eri toiminnallisuuksilla.

3.8 React developer tools

Perinteiset selainkehittäjätyökalut on suunniteltu verkkosivujen tarkastamiseen ja virheenkorjaukseen HTML-, CSS- ja JavaScriptin avulla. Niitä ei voi kuitenkaan käyttää React-sovellusten tehokkaaseen tarkastamiseen ja virheenkorjaukseen, vaan siihen tarvitaan ReactDevTools toimiakseen Reactin rakenteen kanssa. Yleisin tapa asentaa ReactDevTools on asentaa se selaimeesi. Chrome selaimeen se asennetaan Chrome Webstoresta haulla “React developer tools” ja painamalla sen jälkeen add to chrome painiketta. Tämän jälkeen se on käytössä chrome selaimessa. Sama tapa toimii Firefoxille ja Edgelle. Muille selaimille, kuten Safarille, ReactDevlTools täytyy asentaa npm-pakettina. Seuraava asennus tapa pätee myös React Nativelle.

```
# Yarn
yarn global add react-devtools

# Npm
npm install -g react-devtools
```

Kuva 10 rect-devtools

Seuraavaksi avataan react-devtools terminaalissa

```
react-devtools
```

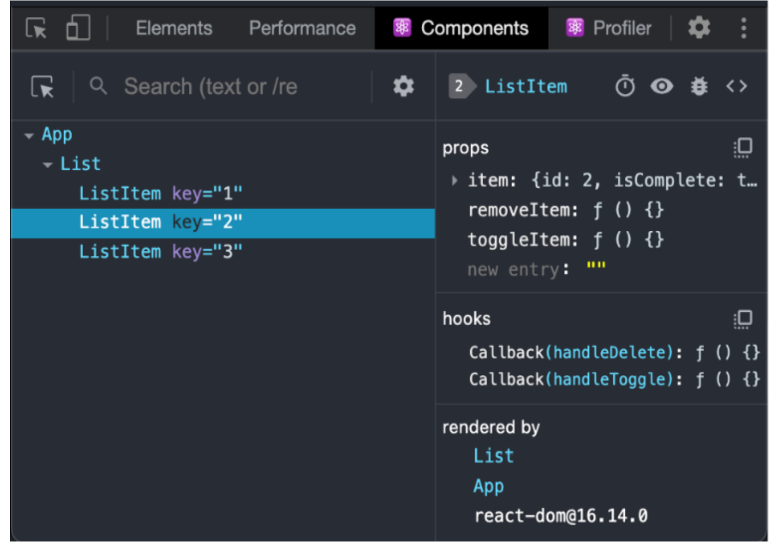
Kuva 11 react-devtools

Seuraavaksi yhdistetään verkkosivulle script-tagin avulla.

```
<html>
  <head>
    <script src="http://localhost:8097"></script>
```

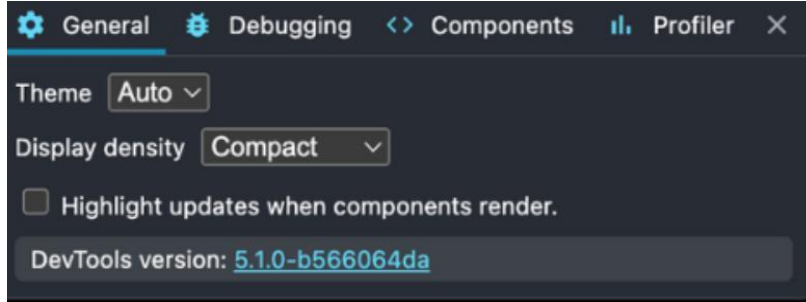
Kuva 12 react-devtools

Kun developer tools on asennettu, navigointi palkkiin ilmestyy kaksi uutta paneelia Components ja Profiler.



Kuva 13 react-devtools

Components välilehdessä on komponentti puu josta pääsee käsiksi jokaisen komponentin hookkeihin ja propseihin. Profiler välilehdessä voi tarkastella sovelluksen suorituskykyä. Välilehdellä voidaan tuoda (import) ja viedä (export) tallennettuja suorituskykyyn liittyviä tietoja. Rataskuvaketta painamalla aukeaa 4 välilehteä, General ja Debugging, Components ja Profiler.



Kuva 14 react-devtools

Näissä neljässä välilehdessä voit säätää profiloijan teemaan, näyttöön, virheenkorjausasetuksiin, komponenttisuodattimiin ja tallennusasetuksiin liittyviä asetuksia.

JavaScript: Virheiden ja suorituskykyongelmien jäljittäminen pelkällä JavaScriptillä voi olla aikaa vievää ja hankalaa, varsinkin jos sovellus kasvaa.
React: Reactin ekosysteemiin kuuluu erityisesti "React Developer Tools" -selainsovellus, joka tekee komponenttien tarkastelusta, virheiden jäljittämisestä ja suorituskyvyn optimoinnista huomattavasti helpompaa.

React Developer Tools on selainlaajennus, joka tarjoaa React-kehittäjille syvällisemmän näkymän komponenttipohjaisen käyttöliittymän tilaan ja rakenteeseen. Vaikka Chrome Developer Tools (DevTools) tarjoaa kattavat työkalut HTML:n, CSS:n ja JavaScriptin tarkasteluun ja virheenkorjaukseen, se ei ole suunniteltu näyttämään Reactin sisäistä komponenttihierarkiaa tai tilahallintaa. React Developer Tools täydentää Chrome DevToolsia tarjoamalla seuraavat erityisominaisuudet:

- Komponenttihierarkia: Näyttää React-sovelluksen komponenttipuun sellaisena kuin se on määritelty JSX:ssä, mahdollistaen yksittäisten komponenttien tutkimisen ja niiden tilojen (state, props, context) tarkastelun reaaliajassa.
- Tilan tarkastelu ja muokkaus: Komponentin tilaa voidaan paitsi tarkastella myös muuttaa suoraan käyttöliittymästä, mikä helpottaa testauksessa ja virheenkorjauksessa.

- Renderöintijäljitys: React Developer Tools näyttää, mitkä komponentit renderöityivät uudelleen ja miksi, mikä auttaa optimoimaan suorituskykyä (esim. React.memo, useCallback, jne.).
- Hooks-tuki: Tarjoaa erillisen näkymän useState, useEffect, useReducer jne. hookien arvoista komponenttitasolla, mikä ei ole mahdollista Chrome DevToolsilla ilman manuaalista konsolitulostusta.

Sen sijaan Chrome Developer Tools soveltuu paremmin:

- DOM-puun ja tyylien (CSS) virheenkorjaukseen
- Verkkoanalytiikkaan (Network)
- JavaScript-konsoliin ja virheiden tarkkailuun
- Suorituksen profilointiin

Yhteenvetona voidaan todeta, että Chrome DevTools tarjoaa yleiskäyttöiset selaimen

virheenkorjausvälineet, kun taas React Developer Tools on erikoistunut React-komponenttien ja tilan hallintaan. Paras tulos saavutetaan käyttämällä molempia työkaluja rinnakkain.

4 Yhteenveto

Tässä kandidaatintyössä tarkasteltiin React-kirjastoa ja sen tarjoamia etuja verrattuna pelkkään JavaScriptiin. Työssä esiteltiin Reactin keskeisiä ominaisuuksia, kuten komponenttipohjainen arkkitehtuuri, virtuaalinen DOM, tilanhallintakirjastot sekä deklaratiivinen ohjelmointimalli. Lisäksi vertailtiin eri tilanhallintaratkaisuja (Redux, MobX, Recoil, Jotai, Signia, XState) ja tarkasteltiin, kuinka ne vaikuttavat sovellusten kehitykseen ja ylläpitoon.

Reactin keskeinen lisäarvo pelkkään JavaScriptiin nähden on sen modulaarinen rakenne, joka mahdollistaa käyttöliittymän jakamisen pieniin, uudelleenkäytettäviin komponentteihin. Tämä parantaa koodin selkeyttä ja tukee skaalautuvien ratkaisujen rakentamista. Reactin deklaratiivinen lähestymistapa vähentää virheherkkää manuaalista DOM-manipulointia ja nopeuttaa käyttöliittymän kehitystä.

Sovelluskehityksen tehokkuus kasvaa, kun komponentteja voidaan uudelleenkäyttää eri osissa sovellusta, ja ylläpidettävyys paranee selkeän rakenteen sekä hyväksi todettujen käytäntöjen avulla. Reactin ekosysteemi — sisältäen tilanhallintakirjastot, kehitystyökalut kuten React Developer Tools, ja laajan yhteisön tuen — tekee siitä kilpailukykyisen vaihtoehdon modernien web-sovellusten rakentamiseen.

Yhteenvetona voidaan todeta, että React tarjoaa merkittäviä etuja verrattuna pelkkään JavaScriptiin erityisesti ylläpidettävyyden, suorituskyvyn ja kehityksen tehokkuuden näkökulmasta. Nämä ominaisuudet tekevät Reactista suositun ja perustellun valinnan nykyaikaisten käyttöliittymien kehityksessä

Lähteet

[1] LogRocket. “A guide to choosing the right React state management solution.” https://blog.logrocket.com/guide-choosing-right-react-state-management-solution/ Viitattu: 20.4.2026

[2] Auth0. “A Brief History of JavaScript.” https://auth0.com/blog/a-brief-history-of-javascript/ Viitattu: 20.4.2026

[3] Aston, B. “Lesson 1a: The History of JavaScript.” Medium. https://medium.com/@_benaston/lesson-1a-the-history-of-javascript-8c1ce3bffb17 Viitattu: 20.4.2026

[4] CoderRocketFuel. “6 Different Ways to Create a React Web Application.” https://coderrocketfuel.com/article/6-different-ways-to-create-a-react-web-application Viitattu: 20.4.2026

[5] Framer. “React vs. Vanilla JS.”
https://www.framer.com/blog/posts/react-vs-vanilla-js/
Viitattu: 20.4.2026

[6] Geekboots. “ReactJS and its usability.” https://www.geekboots.com/story/reactjs-and-its-usability
Viitattu: 20.4.2026

[7] LambdaTest. “Best JavaScript Framework 2020.” https://www.lambdatest.com/blog/best-javascript-framework-2020/
Viitattu: 20.4.2026

[8] Mozilla Developer Network. “What is JavaScript?”
https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript
Viitattu: 20.4.2026

[9] Nielsen Norman Group. “Mobile Native Apps.” https://www.nngroup.com/articles/mobile-native-apps/
Viitattu: 20.4.2026

[10] React. “Introducing JSX.”

https://legacy.reactjs.org/docs/introducing-jsx.html
Viitattu: 20.4.2026

[11] React. “Virtual DOM and Internals.” https://legacy.reactjs.org/docs/faq-internals.html
Viitattu: 20.4.2026

[12] FreeCodeCamp. “How to Use React Developer Tools – Explained With Examples.” https://www.freecodecamp.org/news/how-to-use-react-devtools/
Viitattu: 20.4.2026

[13] W3Schools. “React JSX.” https://www.w3schools.com/react/react_jsx.asp
Viitattu: 20.4.2026

[14] RisingStack. “The History of React.js on a Timeline.” https://blog.risingstack.com/the-history-of-react-js-on-a-timeline/
Viitattu: 20.4.2026

[15] Sidorenko, A. “React is Declarative – What Does it Mean?” https://alexsidorenko.com/blog/react-is-declarative-what-does-it-mean
Viitattu: 20.4.2026

[16] Skillcrush. “Skills to Become a Front-End Developer.” https://skillcrush.com/blog/skills-to-become-a-front-end-developer/
Viitattu: 20.4.2026

[17] We Are Social. “Digital Around the World in April 2020.” https://wearesocial.com/blog/2020/04/digital-around-the-world-in-april-2020
Viitattu: 20.4.2026

[18] W3C. “CSS 2.0 History.” https://www.w3.org/Style/CSS20/history.html
Viitattu: 20.4.2026

[19] W3Schools. “HTML Tutorial: History of HTML.” https://www.w3schools.in/html-tutorial/history/
Viitattu: 20.4.2026

[20] W3Schools. “JavaScript HTML DOM.” https://www.w3schools.com/js/js_htmldom.asp

Viitattu: 20.4.2026

[21] AppMaster. “React’s Component-Based Architecture: A Case Study.”
https://appmaster.io/blog/react-component-based-architecture
Viitattu: 20.4.2026

[22] FreeCodeCamp. “How to Build Reusable React Components.”
https://www.freecodecamp.org/news/how-to-build-reusable-react-components/ Viitattu: 20.4.2026

[23] Abramov, D., & Clark, A. (2015). “Redux – A predictable state container for JavaScript apps.”
https://redux.js.org
Viitattu: 20.4.2026

[24] Erikson, M. (2021). “Redux Toolkit Documentation.” https://redux-toolkit.js.org/
Viitattu: 20.4.2026

[25] Egghead.io. (2020). “Introduction to MobX.”
https://egghead.io/courses/manage-complex-state-in-react-apps-with-mobx
Viitattu: 20.4.2026

[26] Jotai Documentation. (2023). https://jotai.org/docs/introduction
Viitattu: 20.4.2026

[27] Ragheb, M., & Florin, C. (2021). “Modeling Finite State Machines in Modern Front-End Applications.” https://stately.ai/docs/ Viitattu: 20.4.2026