Tampereen ammattikorkeakoulu

Rinnakkaisten näkymien hallinta React-sovelluksessa

Anton Höglund

OPINNÄYTETYÖ
Joulukuu 2021

Tietojenkäsittelyn tutkinto-ohjelma
Ohjelmistotuotanto

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittelyn tutkinto-ohjelma
Ohjelmistotuotanto

HÖGLUND; ANTON:
Rinnakkaisten näkymien hallinta React-sovelluksessa

Opinnäytetyö 33 sivua, joista liitteitä 3 sivua
Joulukuu 2021

Opinnäytetyö käsittelee verkkosovelluksen taustajärjestelmää, joka hallitsee sovelluksen näkymien tilaa ja reititystä. Opinnäytetyön toimeksiantaja on Vinka Oy, henkilölogistiikan ratkaisuihin erikoistunut ohjelmistoyritys. Opinnäytetyön taustalla on Tuomi Logistiikka Oy:n tilaama projekti, jonka tarkoituksena on tuottaa verkkosovellus, jolla hallitaan joukkoliikennepalvelun aikatauluja.

Tavanomaisesti verkkosovellukset reititetään näyttämään yhtä pääasiallista näkymää kerrallaan. Tämä opinnäytetyö käsittelee React-kirjastolla toteutettua näkymienhallintajärjestelmää, joka mahdollistaa useamman rinnakkaisen näkymän näyttämisen samanaikaisesti verkkosovelluksessa. Opinnäytetyön tavoitteena on selvittää, miten haluttu järjestelmä voidaan toteuttaa React-kirjaston ja URL-standardin ominaisuuksien ja rajoitteiden puitteissa.

Opinnäytetyössä päädytään käyttämään Reactin Context-rajapintaa näkymien tilan hallintaan. Context-rajapinta mahdollistaa näkymien luonnin useasta eri sovelluksen komponentista, vaikka näkymien tila on keskitetty yhteen komponenttiin. URL-standardin kyselyparametreja käytetään sovelluksen reititykseen URL-polkujen sijaan, jolloin kaikki sovelluksen näkymät voivat ilmetä rinnakkain URL:ista.

Asiasanat: ohjelmistokehitys, verkkosovellukset, URL, React, React Context

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Business Information Systems
Software Development

HÖGLUND, ANTON:
Managing Parallel Views in a React Application

Bachelor's thesis 33 pages, appendices 3 pages
December 2021

The thesis explores the design and development of a view management and routing system for a web application developed with the React framework. Traditionally web applications are routed to display one main view at a time, but the view management system developed in this thesis must be capable of routing and managing the state of multiple parallel views. The thesis studies the React and the URL standard, in order to find out how the system can be implemented.

The client of the thesis is Vinka Oy, a software company specializing in solutions for passenger logistics. The project behind the thesis was developing an application for managing timetables of a public transport service operated by Tuomi Logistiikka Oy.

The thesis uses the React Context API to manage the state of the views in a centralized manner, so that views may be created from any component of the application with ease. The implemented view management system routes the views from the URL by using query parameters. This allows the application to represent the views as multiple parallel values, which would not be possible using the URL path.

Key words: software development, web applications, URL, React, React Context

SISÄLLYS

LYHENTEET JA TERMIT

DOM	Document Object Model, web-selaimissa käytetty dokumentteja esittävä tietorakenne
ECMAScript	Ohjelmointikieli, joka toimii JavaScriptin standardina
HTML	Hypertext Markup Language, merkintäkieli web-selaimissa esitettäville dokumenteille
JSX	JavaScript XML, Reactissa käytettävä JavaScriptin syntaksilaajennus
TypeScript	JavaScriptiin pohjautuva ohjelmointikieli, jossa voi määritellä tietotyyppejä
URI	Uniform Resource Identifier, standardi palvelimen resurssien tunnistamiselle
URL	Uniform Resource Locator, standardi palvelimen resurssien paikantamiselle

1 JOHDANTO

Ohjelmistokehityksessä standardit protokollissa, rajapinnoissa ja ohjelmointikielissä määrittelevät ominaisuudet ja rajoitukset, joiden puitteissa ohjelmistoja voidaan kehittää. Ajan kanssa standardien ympärille muodostuu konventioita, eli yleisesti hyväksyttyjä tapoja, miten standardeja hyödynnetään. Konventioiden ansiosta kehitetään apukirjastoja ja ohjelmistokehyksiä, joiden käyttö nopeuttaa kehitystä huomattavasti.

Perinteisissä web-sovelluksissa selaimen URL-osoite näyttää käyttäjälle yhden tiedoston, joka yleensä tuottaa yhden näkymän, ja sovellusta navigoitaessa haetaan selaimeen uusia tiedostoja. Moderneilla web-sovelluskehyksillä, kuten Reactilla kehitetyissä sovelluksissa on yleensä vain yksi HTML-tiedosto, johon sisältöä lisätään ohjelmallisesti, jolloin URL-osoite pysyy koko ajan samana. React-sovelluksissa voidaan kuitenkin saada sovellus mukailemaan konventionaalista web-sovelluksen reititystä apukirjastojen avulla. Tämänlaisilla apukirjastoilla URL-osoitteessa heijastuu sovelluksen aktiivinen näkymä. Vastaavasti selaimesta kopioidulla linkillä voi apukirjaston ansiosta viedä käyttäjän aina samaan näkymään.

Tässä opinnäytetyössä käsitellään Reactilla toteutettua sovellusta, jossa konventionaaliset apukirjastot eivät ole avuksi, jolloin näkymien hallinta ja reititys on toteutettava itse. Kyseessä on web-sovellus, jossa on samanaikaisesti useampi samanarvoinen näkymä esillä, ja jonka näkymien tila pitää pystyä reitittämään URL-osoitteen kautta. Opinnäytetyötä taustoitetaan teknisesti tutkimalla Reactia ja URL-standardia, ja sovelluksen toteutusta tutkitaan käytännönläheisesti.

Opinnäytetyön taustalla on Pali-palveluliikenteen aikataulujenhallintasovelluksen toteutusprojekti. Pali-palveluliikenne on Tuomi Logistiikka Oy:n toteuttama joukkoliikenteen tapainen kuljetuspalvelu, jossa haetaan asiakkaita tilauksesta ovelta ovelle heidän lähipalveluiden piiriin. Opinnäytetyön toimeksiantaja Vinka Oy toimii tämän projektin ohjelmistotoimittajana.

2 REACT

2.1 Reactin esittely

React on Facebookin kehittämä JavaScript-pohjainen ohjelmistokehys web-sovellusten kehittämiseen. Stack Overflow -sivuston (2021) teettämän kyselyn mukaan React on yleisimmin käytetty web-sovelluskehys.

Yksi Reactin tunnusomaisista piirteistä on deklaratiivisuus. Tämä tarkoittaa sitä, että vaikka kehitettäisiin sovellusta, jossa käyttöliittymän tila päivittyy dynaamisesti, niin sovelluksen kehittäjän tarvitsee ainoastaan määritellä mitä käyttöliittymässä halutaan kokonaisuudessaan näyttää, ja React huolehtii, että vain muuttuneet osat käyttöliittymästä päivittyvät DOM-rajapintaan. (React n.d.)

2.2 DOM ja JSX

DOM on selaimissa toimiva muistinvarainen puumainen tietorakenne, jolla voidaan esittää dokumenttien rakennetta ja sisältöä. DOM-rajapintaa käytetään tavanomaisesti HTML-dokumenttien muokkaamiseen JavaScriptillä. DOMin puun haarat päättyvät solmuihin, jotka sisältävät Node-rajapinnan toteuttavia olioita. HTML-dokumentin tapauksessa HTML-elementit esitetään DOMin tietorakenteessa olioina, jotka toteuttavat mm. HTMLElement -rajapinnan Node-rajapinnan lisäksi. (Document Object Model (DOM) n.d.) Kuvassa 1 on esimerkki, miten JavaScriptissä lisätään HTML-dokumentin div-elementtiin h1-elementti DOMia käyttäen.

HTML

```html
<!DOCTYPE html>
<html>
<head>
  <title>||Working with elements||</title>
</head>
<body>
  <div id="root"></div>
</body>
</html>
```

JavaScript + No-Library (pure JS)

```javascript
document.body.onload = addElement;

function addElement () {
  // Luo h1-elementin
  const h1 = document.createElement("h1");

  // Luo teksti-node
  const newContent = document.createTextNode("Otsikkoteksti");

  // Lisää teksti-node h1-elementtiin
  h1.appendChild(newContent);

  // Lisää h1-elementti div-elementtiin, jonka id on root
  const currentDiv = document.getElementById("root");
  currentDiv.appendChild(h1);
}
```

KUVA 1. DOMin muokkaaminen JavaScriptillä.

Reactissa voidaan käyttää JSX:ää, joka on JavaScriptin syntaksilaajennus. JSX:llä voidaan luoda React-elementtejä HTML-merkintää muistuttavalla syntaksilla. (Introducing JSX n.d.) React-elementit ovat käytännössä tavallisia olioita. Ne toimivat vastineena DOMin HTML-elementeille, ja ovat myös Reactin dokumentaation mukaan tehokkaampia käyttää. React-elementit päivitetään Reactin omaan virtuaaliseen DOMiin, jonka React synkronoi HTML-dokumentin varsinaisen DOMin kanssa. (Rendering Elements n.d.) Kuvassa 2 esitetään, miten Reactissa päivitetään DOMia JSX:ää käyttäen vastaavalla tavalla kuin kuvassa 1.

```
HTML ▼

<!DOCTYPE html>
<html>
<head>
  <title>||Working with elements||</title>
</head>
<body>
  <div id="root"></div>
</body>
</html>

React + No-Library (pure JS) ▼

const element = <h1>Otsikkoteksti</h1>;
ReactDOM.render(element, document.getElementById('root'));
```

KUVA 2. DOMin muokkaaminen Reactissa.

2.3 Komponentit ja tilanhallinta

Reactin komponenteilla voidaan pilkkoa käyttöliittymä uudelleenkäytettäviin osiin. Komponentit tuovat hallittavuutta sovelluksen lähdekoodiin, kun erilliset loogiset osat käyttöliittymästä esiintyvät omina yksikköinään. Komponentit koostuvat React-elementeistä. Komponentteja voidaan määritellä JavaScriptin funktioina, tai vaihtoehtoisesti luokkina. (Components and Props n.d.) Tässä opinnäytetyössä käsitellään kuitenkin vain funktiokomponentteja.

Komponentin ilmentymän luoja voi määritellä komponentin tilan asettamalla komponentille props-parametreja (Components and Props n.d.). JSX-syntaksilla tämä näyttää samankaltaiselta kuin attribuuttien määritteleminen elementtiin HTML-merkintäkielessä, kuten kuvassa 3 demonstroidaan.

```
function Component(props) {
  return <h1>{props.title}</h1>;
}

ReactDOM.render(
  <Component title="Otsikko" />,
  document.getElementById('root')
);
```

KUVA 3. Komponenttien määrittely props-parametreillä

Komponentti voi myös itse hallita tilaansa tilamuuttujilla. Tilamuuttujan saa luotua Reactin ”useState”-funktiolla, joka palauttaa muuttumattoman tilamuuttujan sekä funktion, jonka kautta tilamuuttujaa saa muuttaa. Kutsumalla tilanmuuttajafunktiota React osaa havaita, että komponentin tila on muuttunut. Huomioi, että komponentin tilanhallinnassa ei voi käyttää tavallisia JavaScriptin muuttujia. (Using the State Hook n.d.) Kuvassa 4 demonstroidaan tilanhallintaa komponentissa, jossa on otsikko ja painike. Painiketta painettaessa kasvatetaan numeerista tilamuuttujaa, joka näytetään osana otsikkoa.

```
4   function Component() {
5     const [ counter, setCounter ] = useState(0)
6     return (
7       <div>
8         <h1>Clicked {counter} times</h1>
9         <button onClick={() => setCounter(counter + 1)}>Click me</button>
10      </div>
11    );
12  }
```

KUVA 4. Komponentin tilanhallinta

React-kehityksessä kohdataan usein tilanteita, joissa lapsikomponentin tarvitsee muuttaa vanhempikomponenttinsa tilaa. Tämä voidaan ratkaista niin, että lapsikomponentti ottaa vastaan props-parametreissä callback-funktion, jota kutsutaan, kun tila muuttuu. Callback-funktion tarjonnut vanhempikomponentti voi tarvittaessa muuttaa tilaansa, kun funktiota kutsutaan.

Kuvassa 5 on vanhempikomponentti Parent ja lapsikomponentti Child. Lapsikomponentti on yksinkertainen lomake, jossa on tilanhallintaa. Kun lapsikomponentin lähetyspainiketta painetaan, niin kutsutaan props-parametrina annettua callback-funktiota. Vanhempikomponentti päivittää tilaansa, kun callback-funktiota kutsutaan.

```
4   function Child(props) {
5     const [ text, setText ] = useState('');
6     return (
7       <div>
8         <input onChange={(e) => setText(e.target.value)} />
9         <button onClick={() => props.onChange(text)}>Submit</button>
10      </div>
11    );
12  }
13
14  function Parent() {
15    const [ heading, setHeading ] = useState('');
16    return (
17      <div>
18        <h1>Heading: {heading}</h1>
19        <Child onChange={(value) => setHeading(value)} />
20      </div>
21    );
22  }
```

KUVA 5. Komponenttien välinen tiedonvälitys callback-funktioilla

2.4 Taulukot ja key-parametri

Taulukoita voidaan muuttaa React-elementeiksi JavaScriptin map-metodilla. Tällä tavalla esimerkiksi taulukko, jossa on merkkijonoja, voidaan muuttaa taulukoksi React-elementtejä, jotka esittävät tekstin kappaleita. Taulukosta luotaviin React-elementteihin kuuluu tällöin määritellä yksilöllinen key-niminen props-parametri, jotta React pystyy pitämään kirjaa taulukon lisätyistä, poistuneista tai päivitetyistä elementeistä (Lists and Keys n.d.). Key-parametrin käyttöä demonstroidaan kuvassa 6.

```
35  function List() {
36    const paragraphs = ['Something', 'else', 'entirely'];
37
38    return (
39      <div>
40        {paragraphs.map((paragraph, index) => <p key={`p-${index}`}>{paragraph}</p>)}
41      </div>
42    );
43  }
44
```

KUVA 6. Key-parametrin käyttö taulukossa React-elementtejä.

2.5 React Context

Vaikka tilaa on mahdollista jakaa komponenttien kesken callback-funktioilla, se ei ole kuitenkaan aina paras ratkaisu lähdekoodin hallittavuuden kannalta. Ongelmia hallittavuudessa voi ilmetä esimerkiksi tapauksessa, jossa komponenteilla on syvä hierarkia, jolloin tila pitää välittää useamman lapsikomponentin props-parametrien kautta.

React tarjoaa tämänkaltaisten ongelmien ratkaisuun Context-rajapinnan. Context-rajapinnan avulla voidaan keskittää tila yhteen komponenttiin, ja käsitellä tilaa useammassa komponentissa ilman, että tilaa tarvitsee välittää hierarkkisesti. Context-rajapinnan ytimessä on Context-oliot, joiden kautta voidaan luoda Provider ja Consumer -komponentteja. Provider-komponentit tarjoavat Contextin tilan, ja Consumer-komponentit reagoivat tilan muutoksiin. Komponentit, jotka käyttävät Contextin tilaa tulee olla Provider-komponentin alla komponenttihierarkiassa. (Context n.d.)

Kuvassa 7 demonstroidaan Context-rajapinnan käyttöä sovelluksen tekstin värin hallintaan. App-komponentissa alustetaan Provider-komponentin avulla Contextin arvo, jossa on muuttuja tekstin värille sekä funktio värin muuttamiseen. ThemedComponent-komponentissa käytetään Contextin Consumer-komponenttia, ja muutetaan otsikon väriä Contextin tilan perusteella. ThemeToggle-komponentissa on painike, jota painettaessa muutetaan Contextin tekstivärimuuttujaa. ThemeToggle-komponentin tapauksessa käytetään useContext-funktiota Consumer-komponentin sijaan. Nämä kaksi tapaa päästä Contextiin käsiksi ovat kuitenkin toiminnallisesti samanlaisia.

```
4   const ThemeContext = React.createContext();
5
6   function App() {
7     const [textColor, setTextColor] = useState('red');
8
9     return (
10      <ThemeContext.Provider value={{ textColor, setTextColor }}>
11        <ThemedComponent />
12        <ThemeToggle />
13      </ThemeContext.Provider>
14    );
15  }
16
17  function ThemedComponent() {
18    return (
19      <ThemeContext.Consumer>
20        {(state) => <h1 style={{ color: state.textColor }}>My themed text</h1>}
21      </ThemeContext.Consumer>
22    );
23  }
24
25  function ThemeToggle() {
26    const context = useContext(ThemeContext);
27    const onClick = () => context.setTextColor(
28      context.textColor === 'red' ? 'green' : 'red'
29    );
30    return <button onClick={onClick}>Toggle theme</button>;
31  }
```

KUVA 7. Context-rajapinnan käyttö

3 URL-osoitteet

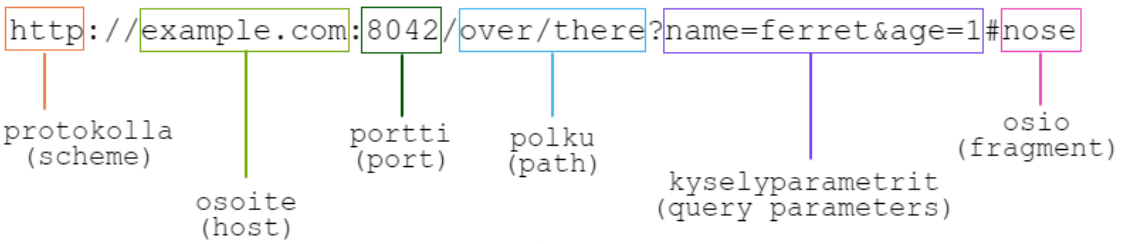
3.1 URL-standardi

Web-sovellukset koostuvat tiedostoista, jotka julkaistaan web-palvelimella. Nämä tiedostot, tai geneerisemmällä termillä resurssit, voidaan tunnistaa ja paikantaa URI-standardiin pohjautuvalla URL-standardilla (Gourley, D., Totty, B., Sayer, M., Aggarwal, A. & Reddy, S. 2002. Kappale 1.3).

3.2 URL-osoitteiden syntaksi

Kuvassa 8 esitetään URL-osoitteiden syntaksia. URL-osoitteessa polkua edeltävät osat liittyvät yhteydenottoon palvelimelle, eikä niinkään resurssin paikantamiseen, joten ne eivät ole relevantteja tässä opinnäytetyössä.

URL-osoitteen polku merkitsee resurssin sijaintia palvelimella, ja muistuttaa usein hierarkkisen tiedostojärjestelmän rakennetta. Polku saattaa vastata palvelimella julkaistavien tiedostojen tiedostorakennetta suoraan, mutta palvelinohjelma ratkaisee, miten polkua tulkitaan. Kyselyparametrit ovat sarja avain-arvo-pareja, jotka erotellaan &-merkillä. Palvelimet käyttävät näitä yleensä rajaamaan pyydettyä resurssia. Jos pyydetty resurssi on esimerkiksi jonkinlainen lista, niin voidaan kyselyparametreillä rajata listassa esiintyviä entiteettejä. URL-osoitteen osio-osa on viite johonkin osaan resurssista. Tätä käytetään usein viittaamaan johonkin otsikkoon HTML-dokumenteissa. (Gourley, D. ym. 2002. Kappale 2.2)



KUVA 8. URL-standardin syntaksi. Kuvasta jätetty pois ennen osoiteosaa ilmenevä vaihtoehtoinen pääsytieto-osa.

3.3 Erikoismerkit URL-osoitteissa

URL-osoitteiden sisällössä on rajattu sallitut merkit osaan US-ASCII-merkistöstä. Käytännössä numerot ja kirjaimet ovat sallittuja, ja useimmat erikoismerkit ovat varattuja URL-osoitteiden syntaksiin. Esimerkiksi kysymysmerkki ei saa esiintyä polussa, koska se on varattu URL-osoitteen kyselyparametriosan erottamiseen. (Gourley D. ym. 2002. Kappale 2.4)

Näiden rajoitusten kiertämiseksi voidaan käyttää URL-standardin prosenttikoodausta (engl. Percent-encoding). Kielletty merkki ilmaistaan suojausnotaatiolla, joka koostuu prosenttimerkistä ja kielletyn merkin ASCII-koodista heksadesimaalimuodossa. (Gourley D. ym. 2002. Kappale 2.4) Kansainväliset merkit, kuten skandinaaviset merkit, koodataan vastaavalla tavalla kuin ASCII-merkistön merkit, mutta tällöin käytetään koodausta UTF-8-merkistön koodeilla (Berners-Lee, T., Fielding, R. & Masinter L. 2005).

Prosenttikoodausta demonstroidaan taulukossa 1. Ensimmäiset kaksi riviä demonstroivat ASCII-merkistöstä löytyvien merkkien koodausta, ja viimeinen rivi demonstroi UTF-8-merkistön merkin koodausta.

TAULUKKO 1. Kiellettyjen merkkien koodaus URL-osoitteeseen.

Merkki	Koodi	Esimerkki
VÄLILYÖNTI	0x20	http://verkkosivu.fi/yhdys%20sana
%	0x25	http://verkkosivu.fi/100%25vastaajista
ö	0xC3 0xB6	http://verkkosivu.fi/merkist%C3%B6

4 SOVELLUKSEN NÄKYMIEN SUUNNITTELU

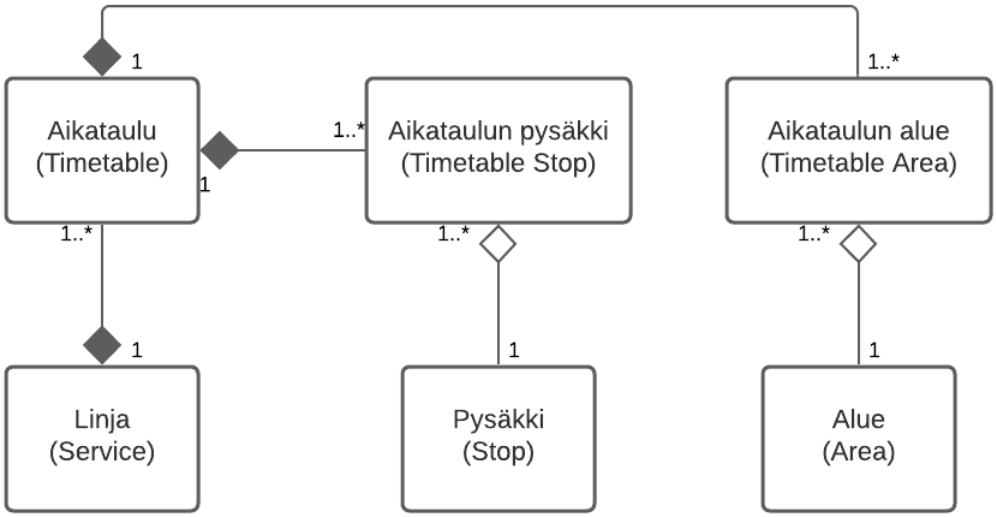
4.1 Tietorakenteet

Aikataulujenhallintasovelluksen tarkoituksena on luoda ja ylläpitää aikatauluja, joista muodostetaan Vinkan ohjelmistoalustassa reittejä, joiden ohjeiden mukaisesti liikennöitsijä ajaa pisteestä toiseen. Reiteissä käydään kiinteillä linja-autopysäkkien tapaisilla pysäkeillä sekä myös tiettyjen alueiden sisällä ovelta ovelle hakemaan asiakkaita tilauksesta.

Kuviossa 1 esitetään sovelluksen tietorakenteiden hierarkiaa, joka pohjautuu liitteen 1 domain-mallikuvaukseen. Hierarkiassa korkeimmalla on linja, joka kuvastaa yhtä palvelulinjaa. Tämä on rinnastettavissa linja-autolinjaan, jolla on tietty nimi tai numero, mutta jonka aikataulut muuttuvat ajan saatossa. Sovelluksen linja on käytännössä vain olio, johon kootaan useampi aikataulu yhteen taulukkoon.

Aikataulut kuvaavat linjan aikataulua tiettynä ajanjaksona. Aikataululla on muun muassa voimassaoloaika ja kuvaus siitä, millä pysäkeillä ja alueilla liikennöitsijä käy tiettyinä aikoina päivästä. Liitteen 1 domain-mallikuvauksesta poiketen aikataulut ja pysäkit eivät kuulu suoraan aikataululle, vaan näiden väliselle suhteelle on omat välirakenteensa. Pysäkit ja alueet ovat uudelleenkäytettävissä eri linjojen eri aikataulujen välillä, ja näin ollen ne kuvastavat ainoastaan maantieteellisiä sijainteja. Aikataulun pysäkki viittaa pysäkkiin, mutta kertoo myös, että mihin aikaan pysäkillä käydään aikataulun mukaan. Aikataulun alue toimii samalla periaatteella kuin aikataulun pysäkki.

Kaikissa tietorakenteissa on oma yksilöivä tunniste, joka ilmenee tietorakenteiden olioissa id-avaimessa. Tämä tunniste asetetaan olioille sovelluksen palvelinohjelmiston tietokannassa.



KUVIO 1. Sovelluksen tietorakenteiden hierarkia

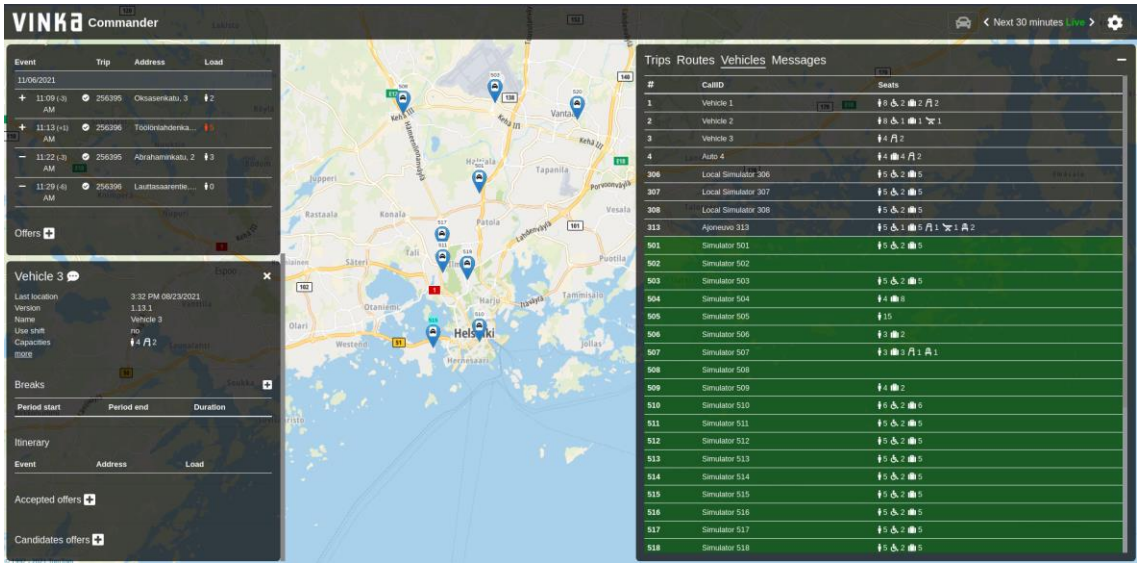
4.2 Sovelluksen näkymät

Sovelluksessa on neljä keskeistä tietorakennetta: linja, aikataulu, pysäkki ja alue. Sovelluksessa tullaan tarvitsemaan näkymiä, joissa yhtä tietorakenneyksikköä voi katselmoida ja muokata, joihin viitataan termillä yksikkönäkymä. Lisäksi tarvitaan listanäkymiä, eli näkymiä, joissa tietorakenneyksiköitä voi listata. Neljää keskeisistä tietorakennetta varten tarvitaan yhteensä seitsemän näkymää, koska linjan yksikkönäkymä toimii aikataulujen listanäkymänä. Aikataulun pysäkit ja alueet näytetään osana aikataulun yksikkönäkymää.

4.3 Näkymien asettelu

Opinnäytetyön toimeksiantajan, Vinka Oy:n, yksi vaatimus sovelluksen ulkoasulle oli, että sovelluksen taustan täyttää interaktiivinen kartta. Näkymät saavat tulla kartan päälle, mutta kartan tulee olla käytettävissä pysäkkien sijaintien valitsemiseen ja alueiden piirtämiseen kartalle. Tämän vuoksi sovelluksen näkymien asetellussa päädyttiin samantapaiseen asetteluun kuin Vinka Oy:n Commander-sovelluksessa, jota näytetään kuvassa 9. Commanderin käyttöliittymän vasemmassa laidassa näytetään kaikki yksikkönäkymät, ja oikeassa laidassa kaikki listanäkymät. Käyttöliittymän keskiosa on jätetty vapaaksi kartan näyttämiseen.

Commanderin tapauksessa käyttöliittymän keskiosaan asetetaan myös näkymiä joissain erikoistapauksissa, joten projektin sovelluksessa tulee myös mahdollistaa keskiosaan asetettavan luokittelemattomia näkymiä, jotka eivät ole lista- tai yksikkönäkymiä. Luokittelemattomille näkymille ei sovelluksen suunnitteluvaiheessa havaittu käyttötapausta, mutta siihen varaudutaan.



KUVA 9. Kuvakaappaus Vinka Oy:n Commander-sovelluksesta

5 NÄKYMIEN HALLINNAN TOTEUTUS

5.1 Teknologiat

Projektissa käytetään Reactin kehityspalvelimen ajoympäristönä Node.js:än versiota 14.5.3. Node.js mahdollistaa JavaScript-koodin suorittamisen palvelimella (OpenJS Foundation. n.d). Sovellus on tehty ECMAScript 2021 -standardin mukaisella JavaScriptillä. JavaScriptin sijaan lähdekoodissa käytetään TypeScript-ohjelmointikielen versiota 4.1.2. Reactista käytetään versiota 17.0.2.

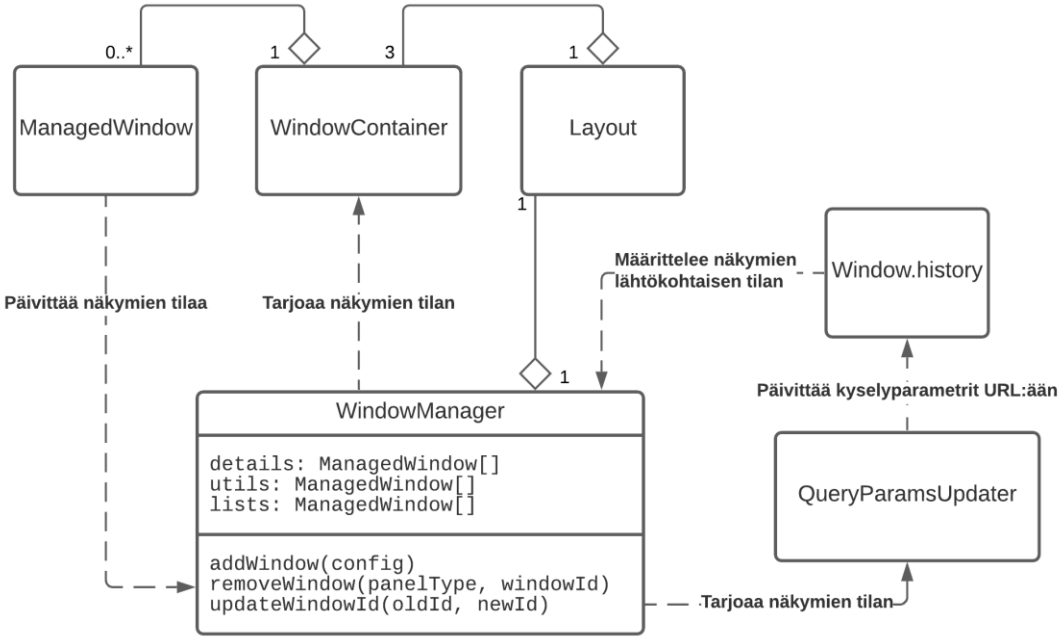
5.2 Arkkitehtuuri

Näkymien tilaa hallitaan Reactin Context -rajapintaa käyttäen, ja näkymienhallintajärjestelmän keskeinen Context on WindowManagerContext. WindowManager-niminen komponentti on näkymienhallinnan ydin, ja käyttää WindowManagerContextin Provider-komponenttia. Kuviossa 2 esitetään WindowManager-komponentin rajapintaa sekä muiden komponenttien vuorovaikutusta sen kanssa. WindowManagerin tilassa säilötään näkymiä kolmessa eri taulukossa, jotka ovat jaoteltu yksikkönäkymiin (details), listanäkymiin (lists), ja luokittelemattomiin näkymiin (utils).

React-elementtien hierarkiassa WindowManager-komponentilla on kaksi lapsikomponenttia, Layout ja QueryParamsUpdater. Layout-komponentti määrittelee koko sovelluksen yleisen visuaalisen asettelun, ja sisältää muun muassa kolme WindowContainer-komponenttia. WindowContainer-komponentit näyttävät kukin näkymät yhdestä WindowManager-komponentin näkymätaulukosta, ja pohjautuvat WindowManagerContext.Consumer-komponentteihin.

Näkymät on esitetty nimellä ManagedWindow kuviossa 2, vaikka tämän nimistä komponenttia ei ole. ManagedWindow on lähdekoodissa nimi rajapintatyypille, joka sisältyy kuvan 10 koodinäytteeseen. ManagedWindow-olion tulee olla React-elementti, jolla on tarvittavat props-parametrit, jotta WindowManager voi yksilöidä näkymän.

WindowManager-komponentin toisen lapsen, eli QueryParamsUpdater-komponentin vastuulla on päivittää URL-osoitteiseen näkymien yksilöintitunnukset. WindowManager lukee sovelluksen käynnistyessä ja selaimen ikkunaa virkistettäessä URL-osoitteesta näkymien yksilöintitunnukset ja luo niiden perusteella näkymät.



KUVIO 2. Näkymienhallintajärjestelmän osien vuorovaikutus

```typescript
import { ReactElement } from 'react';

/**
 * Entity types that WindowManagerService can create windows of.
 *
 * Note: these must NOT be kebab-cased.
 */
export type WindowEntityType = 'area' | 'service' | 'stop' | 'timetable';

/**
 * Names of query parameters that WindowManagerService applies to the url search.
 */
export type QueryParam = keyof WindowPanelMap;

/**
 * Defines which keys of WindowManagerService hold ManagedWindows
 */
export interface WindowPanelMap {
    details: ManagedWindow[];
    utils: ManagedWindow[];
    lists: ManagedWindow[];
}

/**
 * Custom options for windows
 */
export interface CustomWindowOptions {
    serviceId?: string;
    copyFromId?: string;
}

/**
 * A configuration with which WindowManagerService can open a window.
 */
export interface ManagedWindowConfig {
    panelType: keyof WindowPanelMap;
    entityType: WindowEntityType;
    entityId?: string;
    isNew?: true;
    custom?: CustomWindowOptions;
}

/**
 * Properties of a window managed by window manager.
 */
export interface ManagedWindowProps {
    key: string;
    windowId: string;
    custom?: CustomWindowOptions;
}

/**
 * Properties of a "detail" type window.
 */
export interface ManagedDetailWindowProps extends ManagedWindowProps {
    entityId: string;
    isNew: true | undefined;
}

/**
 * Type of a window component.
 */
export type ManagedWindow = ReactElement<ManagedWindowProps>;

/**
 * The properties of WindowManagerContext state.
 */
export interface WindowManagerState extends WindowPanelMap {
    addWindow: (config: ManagedWindowConfig) => void;
    removeWindow: (panelType: keyof WindowPanelMap, windowId: string) => void;
    // Update window id for a window with an entityId
    updateWindowId: (oldWindowId: string, newEntityId: string) => void;
}
```

KUVA 10. Näkymienhallintajärjestelmän rajapinnat TypeScript-koodina.

5.3 Näkymien käsittely

5.3.1 Näkymien luontirajapinta

Näkymät luodaan WindowManagerin ulkopuolelta addWindow-metodin kautta. Tälle metodille syötetään argumenttina ManagedWindowConfig-rajapinnan toteuttava olio. Rajapinnan koodi löytyy kuvasta 10. Rajapinnan tarkoitus on sisältää kaikki tarvittava informaatio näkymän yksilöintiin ja luomiseen. Rajapinta sisältää myös tarvittavia arvoja näkymäkomponentin props-parametreille. Olioon tulee määritellä minkä tyyppinen näkymä on, eli onko se yksikkö-, lista- vai luokittelematon näkymä. Lisäksi näkymän käsittelemä tietorakenne tulee määritellä, jonka vaihtoehtoina ovat linja, aikataulu, pysäkki ja alue.

Yksikkönäkymäkomponenteille välitetään props-parametrit, joiden perusteella komponentti päättelee, että tuleeko sen hakea komponenttiin olemassa olevan tietorakenneyksikön tiedot, vai tuleeko komponentissa luoda uusi tietorakenneyksikkö. Luontirajapinnassa on tämän vuoksi arvoina tietorakenneyksikön yksilötunniste ja totuusarvomuuttuja merkitsemään, onko näkymässä käsiteltävä tietorakenneyksikkö uusi.

5.3.2 Näkymien yksilöiminen

Ennen näkymän komponentin luontia WindowManagerin addWindow-metodissa generoidaan näkymälle yksilöivä tunniste. Tunniste on käytännössä näkymän luontirajapinta merkkijonomuodossa. Tämän avulla WindowManager voi tunnistaa näkymän. Tunnistetta voidaan käyttää näkymäkomponenttien key-parametrina, jotta React pystyy pitämään kirjaa näkymistä, kun ne lisätään DOMiin taulukon kautta. Lisäksi tunniste on myös hyödyllinen näkymien esittämiseen URL-osoitteessa.

Listanäkymän tunniste on yksinkertaisesti näkymään liittyvän tietorakenteen nimi, esimerkiksi ”timetable”, eli aikataulu. Yksikkönäkymän tunniste muodostuu tieto-

rakenteen nimestä, viivasta, sekä tietorakenneyksikön omasta tunnisteesta. Yksilöivä tunniste on tällöin esimerkiksi ”timetable-1”. Jos kyseessä on uusi tietorakenneyksikkö, niin käytetään yksikön oman tunnisteen sijaan nykyhetken aikaa millisekunteina . Tunnisteeseen lisätään myös merkintä, että kyseessä on uusi näkymä. Esimerkki tämänlaisesta tunnisteesta on ”timetable-new-1636808684847”. Uuden tietorakenneyksikön näkymätunnisteen on tarkoitus olla väliaikainen, kunnes yksikkö on tallennettu, jolloin se saa kiinteän yksilöllisen tunnisteen.

5.3.3 Näkymän luominen

Kun näkymän tunniste on luotu, tarkistetaan, löytyykö tunnisteella tilasta näkymää. Jos näkymä on jo olemassa, niin muutetaan näkymän paikkaa taulukon alkuun, jolloin se näkyy ylimpänä sovelluksessa.

Jos näkymä ei löydy, niin se luodaan. Näkymärajapinnan oliosta luodaan näkymäkomponentti ehtolauseiden avulla, tarkistamalla olion näkymätyyppi ja tietorakennetyyppi. Komponentille annetaan oliosta myös muut tarvittavat parametrit, kuten näkymän tunniste ja tarvittaessa tietorakenneyksikön tunniste. Näkymän luonnin toteuttava funktio löytyy kuvasta 11. Kun näkymä on luotu, lisätään se tällöin myös taulukon alkuun.

Näkymän luonti tapahtuu pääasiassa listakomponenteissa, jotka saavat viittauksen WindowManagerin rajapintaan Reactin useContext-funktion kautta. Tämän viittauksen avulla kutsutaan WindowManagerin addWindow-metodia.

```
const configToWindow = (config: ManagedWindowConfig, windowId: string): ManagedWindow => {
    let component: ManagedWindow | undefined;

    const baseProps: ManagedWindowProps = {
        windowId,
        key: windowId,
        custom: config.custom,
    };

    const detailProps: ManagedDetailWindowProps = {
        ...baseProps,
        entityId: config.isNew ? windowId : config.entityId!,
        isNew: config.isNew,
    };

    if (config.panelType === 'details' && !detailProps.entityId) {
        throw Error(
            `No entity id set for detail window of ${config.entityType}. Is new: ${!!config.isNew}`
        );
    }

    // STOP
    if (config.panelType === 'lists' && config.entityType === 'stop') {
        component = <windows.StopsList {...baseProps} />;
    } else if (config.panelType === 'details' && config.entityType === 'stop') {
        component = <windows.StopDetail {...detailProps} />;
    }
    // SERVICE
    else if (config.panelType === 'lists' && config.entityType === 'service') {
        component = <windows.ServicesList {...baseProps} />;
    } else if (config.panelType === 'details' && config.entityType === 'service') {
        component = <windows.ServiceDetail {...detailProps} />;
    }
    // TIMETABLE
    else if (config.panelType === 'details' && config.entityType === 'timetable') {
        component = <windows.TimetableDetail {...detailProps} />;
    }
    // AREA
    else if (config.panelType === 'lists' && config.entityType === 'area') {
        component = <windows.AreasList {...baseProps} />;
    } else if (config.panelType === 'details' && config.entityType === 'area') {
        component = <windows.AreaDetail {...detailProps} />;
    }

    if (component) {
        return component;
    } else {
        throw Error('Could not create window with config ' + JSON.stringify(config));
    }
};
```

KUVA 11. Näkymän luonti näkymänluontirajapinnan avulla.

5.3.4 Näkymien dynaaminen tilankäsittely

Koska funktiokomponenteissa tilan lukemisessa ja muuttamisessa käytetään jokaista tilamuuttujaa varten erillistä muuttujaa ja funktioita, on haastavaa käsitellä tilaa dynaamisesti. WindowManager-komponentissa tämä on ratkaistu luomalla välillinen tilaolio, jonka koodi löytyy kuvasta 12. Tässä tilaoliossa käytetään Ja-

vaScriptin getter- ja setter-syntaksia, joiden avulla tilaolion avaimien arvoja voidaan lukea ja muuttaa kuin tavanomaista olion avain-arvoparia, mutta taustalla käytetään useState-funktion luomia muuttujia ja funktioita.

WindowManagerin addWindow-metodissa, joka löytyy myös kuvasta 12, päästään oikeaan näkymätaulukkoon käsiksi käyttämällä näkymärajapintaoliosta saatua näkymätyyppiä avaimena tilaolioon. Tällä tavoin vältytään ehtolauseista, joilla tarkistettaisiin staattisesti, mitä tilamuuttujaa halutaan käsitellä.

```
95  export function WindowManager({ children }: PropsWithChildren<any>) {
96      const [details, setDetails] = useState<ManagedWindow[]>(getInitialWindows('details'));
97      const [utils, setUtils] = useState<ManagedWindow[]>(getInitialWindows('utils'));
98      const [lists, setLists] = useState<ManagedWindow[]>(getInitialWindows('lists'));
99
100     // Helper for making state iterable and dynamically accessible
101     const panelState: WindowPanelMap = {
102         get details() {
103             return details;
104         },
105         set details(newDetails) {
106             setDetails(newDetails);
107         },
108         get utils() {
109             return utils;
110         },
111         set utils(newUtils) {
112             setUtils(newUtils);
113         },
114         get lists() {
115             return lists;
116         },
117         set lists(newLists) {
118             setLists(newLists);
119         },
120     };
121
122     const addWindow: WindowManagerState['addWindow'] = (config) => {
123         const windowId = encodeWindowId(config);
124         const panelWindows = panelState[config.panelType];
125         const existingWindow = panelWindows.find((w) => w.props.windowId === windowId);
126
127         if (existingWindow) {
128             panelState[config.panelType] = [
129                 existingWindow,
130                 ...panelWindows.filter((w) => w.props.windowId !== windowId),
131             ];
132         } else {
133             const managedWindow = configToWindow(config, windowId);
134             panelState[config.panelType] = [managedWindow, ...panelWindows];
135         }
136     };
```

KUVA 12. WindowManagerin addWindow-metodi ja dynaaminen tilankäsittely

5.4 Näkymien tila URL-osoitteessa

Koska URL-osoitteen polku, jolla verkkosovellukset tavanomaisesti reititetään, viittaa vain yhteen resurssiin, käytetään sovelluksen näkymienhallintajärjestelmässä sen sijaan kyselyparametrejä näkymien reititykseen. Näkymätyypit esiintyvät kyselyparametreissä avaimina. Näkymien tunnisteet yhdistetään pilkuilla yhdeksi merkkijonoksi ja asetetaan avaimen arvoksi.

Tämä tapahtuu QueryParamsUpdater-nimisessä Consumer-komponentissa, joka reagoi WindowManagerin näkymien tilan muutoksiin. Kuvassa 13 esitetään komponentin koodi kokonaisuudessaan. Kun komponentti saa tiedon WindowManagerin päivittyneestä tilasta, luodaan URLSearchParams-olio, joka alustetaan URL-osoitteessa olemassa olevilla kyselyparametreilla. URLSearchParams on rajapinta, joka sisältää apumetodeja kyselyparametrien käsittelyyn, mutta ei itsessään päivitä URL-osoitteeseen kyselyparametreja (URLSearchParams n.d.). Tähän olioon päivitetään kyselyparametrien avaimet ja arvot WindowManagerin tilan mukaan.

Kun avain-arvoparit on lisätty olioon, muunnetaan kyselyparametrit merkkijonoksi olion toString-metodilla, joka käsittelee samalla muun muassa erikoismerkkien prosenttikoodauksen URL-osoitetta varten. Kyselyparametrimerkkijono asetetaan URL-osoitteeseen käyttäen Window.history-oliota. Window.history on rajapinta, jolla voi käsitellä selaimen ikkunan historiaa (Window.history n.d.).

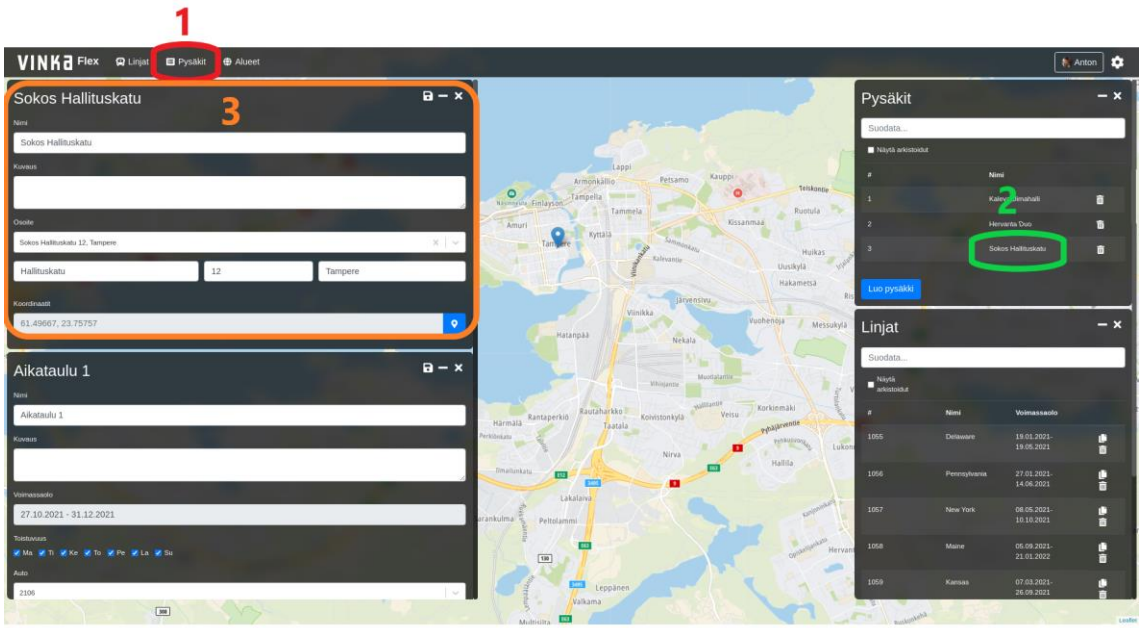
```
export default function QueryParamsUpdater() {
    return (
        <WindowManagerContext.Consumer>
            {(state) => {
                const params = new URLSearchParams(window.location.search.slice(1));

                // Update query params when a panel's windows change.
                for (const panelType of allPanels) {
                    if (state[panelType].length) {
                        params.set(
                            panelType,
                            state[panelType].map((w) => w.props.windowId).join(',')
                        );
                    } else {
                        params.delete(panelType);
                    }
                }

                const paramString = params.toString();
                const url = paramString ? '?' + paramString : location.pathname;
                window.history.pushState(null, '', url);

                return <></>;
            }}
        </WindowManagerContext.Consumer>
    );
}

```

KUVA 13. QueryParamsUpdater-komponentti.

5.5 Sovelluksen käyttö

Kuvassa 14 demonstroidaan navigointia sovelluksessa käyttäen näkymienhallintajärjestelmää. Kuvassa avataan pysäkin yksikkönäkymä, johon kuuluu kolme eri vaihetta. Ensin pysäkkien listanäkymä avataan yläreunan navigointipalkista löytyvällä linkillä. Seuraavaksi avataan pysäkkien listanäkymästä pysäkin yksikkönäkymä. Sen jälkeen avautuu näkymä, jossa pysäkkiä voi katselmoida ja muokata.

Kuvan alareunassa näkyy URL-osoite, jonka kyselyparametreissa heijastuu näkymien tila. Sovelluksessa on pysäkin lista- ja yksikkönäkymien lisäksi avoinna myös linjan lista- ja yksikkönäkymät sekä yhden aikataulun yksikkönäkymä, jotka näkyvät myös URL-osoitteessa.



http://localhost:3010/?lists=stop%2Cservice&details=stop-3%2Ctimetable-3115%2Cservice-1067

KUVA 14. Pysäkin yksikkönäkymän avaaminen sovelluksessa.

6 POHDINTA

Opinnäytetyössä saatiin ratkaisu epäkonventionaalisen näkymien reitityksen toteutukseen, ja näkymien tilanhallintaan. Näkymien reitittäminen URL-osoitteen kyselyparametrien kautta mahdollisti rinnakkaisten näkymien reitittämisen. Tilanhallinnassa React Context osoittautui helposti laajennettavaksi tavaksi sekä keskittää tila yhteen komponenttiin, että päästä käsiksi keskitettyyn tilaan.

Sovelluksen tavoiteltu visuaalinen asettelu vaikutti laajalti näkymienhallintajärjestelmän tarpeisiin ja toteutukseen. Visuaalinen asettelu oli ottanut inspiraationsa toimeksiantajan aikaisemmin toteuttamasta sovelluksesta. Visuaalisessa asettelussa olisi voinut tutkia järjestelmällisemmin sovelluksen käytettävyyttä, ja kuinka hyvän käyttäjäkokemuksen toteutettu näkymien asettelu todellisuudessa tuottaa. Tällöin työn tuloksena olisi voinut olla hyvin erilainen näkymienhallintajärjestelmä, tai että sovelluksessa olisi voitu käyttää valmista apukirjastoa näkymien hallintaan ja reititykseen.

Näkymienhallintajärjestelmän toteutus oli suoraan sidottu sovelluksessa luokiteltuihin näkymätyyppeihin sekä sovelluksessa käytettyihin tietorakenteisiin. Uudelleenkäytettävyyttä ajatellen tulisi tutkia, miten järjestelmästä saisi tietotyypeiltään geneerisemmän, jotta se soveltuisi muihinkin sovelluksiin. Jos järjestelmää haluttaisiin käyttää muissa sovelluksissa, on aiheellista olettaa, että näkymätyyppejä ei voisi luokitella vain kolmeen rinnakkaiseen eri kategoriaan. Voisi tutkia, miten järjestelmää käyttävä sovellus voisi yksinkertaisesti määritellä minkälaisia näkymätyyppejä tulisi olla, ja järjestelmä tuottaisi näille dynaamisesti tilanhallinnan. Jotta järjestelmää ylipäätään voitaisiin helposti käyttää eri sovelluksissa, tulisi tutkia miten järjestelmästä voitaisiin tehdä itsenäinen kirjasto, joka voitaisiin asentaa muihin React-sovelluksiin.

Työssä keskityttiin kuitenkin vain yhden sovelluksen ongelman ratkaisuun. Jos haluttaisiin tehdä näkymienhallintajärjestelmän toteutuksesta geneerisempi eri käyttötapauksiin, olisi tunnistettava nämä eri käyttötapaukset yksityiskohtineen, jotta geneerisen toteutuksen soveltuvuutta voitaisiin arvioida eri käyttötapauksissa.

LÄHTEET

Berners-Lee, T., Fielding, R. & Masinter L. 2005. RFC3986. Verkkosivu. Viitattu 28.10.2021.
https://datatracker.ietf.org/doc/html/rfc3986

Components and Props. n.d. Facebook Inc. Verkkosivu. Viitattu 21.10.2021.
https://reactjs.org/docs/components-and-props.html

Context. n.d. Facebook Inc. Verkkosivu. Viitattu 12.10.2021.
https://reactjs.org/docs/context.html

Document Object Model (DOM). n.d. Mozilla. Verkkosivu. Viitattu 14.10.2021.
https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model

Gourley, D., Totty, B., Sayer, M., Aggarwal, A. & Reddy, S. 2002. HTTP: The Definitive Guide. O'Reilly. Viitattu 14.10.2021. Vaatii käyttöoikeuden.
https://learning.oreilly.com/library/view/http-the-definitive/1565925092/

Introducing JSX. n.d. Facebook Inc. Verkkosivu. Viitattu 21.10.2021.
https://reactjs.org/docs/introducing-jsx.html

Lists and Keys. n.d. Facebook Inc. Verkkosivu. Viitattu 13.11.2021.
https://reactjs.org/docs/lists-and-keys.html

OpenJS Foundation. n.d. Node.js. Verkkosivu. Viitattu 7.11.2021.
https://nodejs.org/en/about/

React. n.d. Facebook Inc. Verkkosivu. Viitattu 21.10.2021.
https://reactjs.org/

Rendering Elements. n.d. Facebook Inc. Verkkosivu. Viitattu 21.10.2021.
https://reactjs.org/docs/rendering-elements.html

Stack Overflow. 2021. Developer Survey 2021. Verkkosivu. Viitattu 21.10.2021.
https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-web-frameworks

URLSearchParams n.d. Mozilla. Verkkosivu. Viitattu 13.11.2021.
https://developer.mozilla.org/en-US/docs/Web/API/URLSearchParams

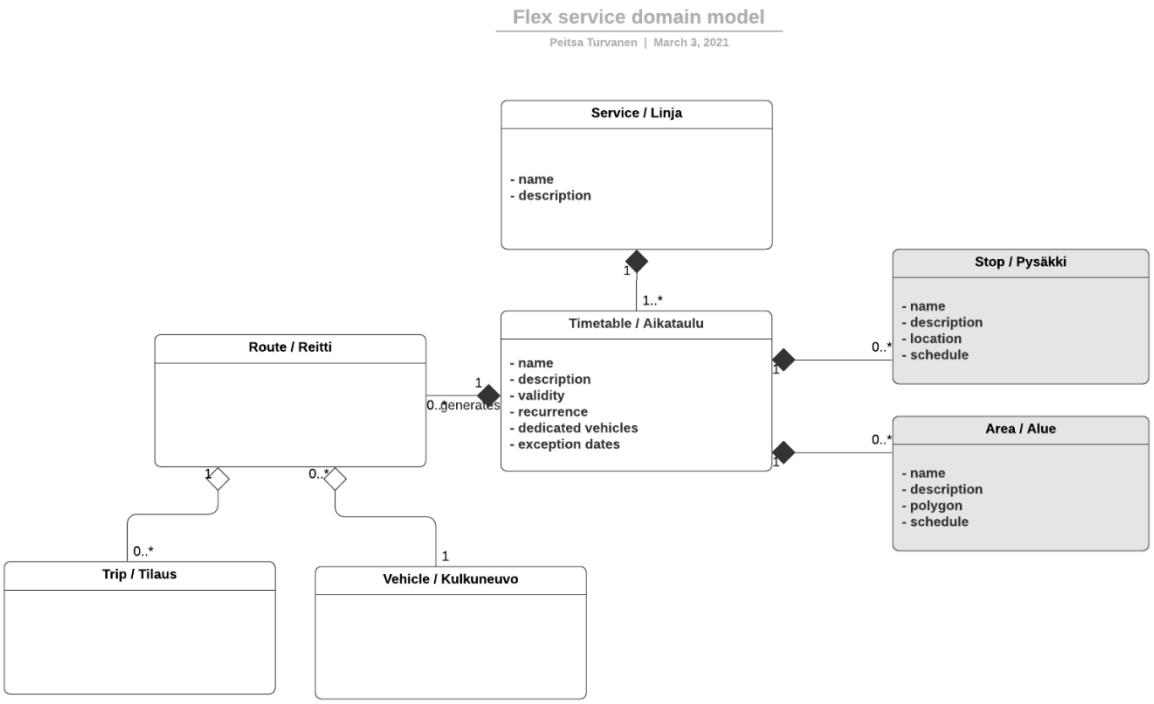
Using the State Hook. n.d. Facebook Inc. Verkkosivu. Viitattu 14.10.2021.
https://reactjs.org/docs/hooks-state.html

Window.history n.d. Mozilla. Verkkosivu. Viitattu 13.11.2021.
https://developer.mozilla.org/en-US/docs/Web/API/Window/history

LIITTEET

Liite 1. Projektin käyttötapausdokumentin domain-malli

Palvelulinjatoteutuksen domain-malli on kuvattu alla olevassa kaaviossa ja mallin objektit ja relaatiot on selitetty seuraavissa kappaleissa.



Linja

Linja on yhden palvelulinjakuvauksen kokoava yläkäsite. Linja kokoaan tiedot mm. pysäkeistä, alueista, aikatauluista, voimassaoloajoista, toistuvuudesta ja kuljetuskalustosta. Yhden palvelulinjan määritykset tehdään yhden linjaobjektin alle.

Aikataulu

Aikataulu kuvaa palvelulinjan muodon ja aikataulun. Yhdellä palvelulinjalla voi

olla useampia aikatauluja, joista jokaisella voi olla eri muoto, aikataulu, voimassaoloaika ja operointiin käytettävä kuljetuskalusto. Esim. yhden palvelulinjan muutokset eri vuodenaikoina voidaan kuvata erillisillä aikatauluilla.

Aikataulun muoto kuvataan pysäkkeinä ja alueina. Aikataulussa saattaa olla yksi tai useampi pysäkki ja/tai alue, joiden sijainnit ja aikataulut määrittävät palvelulinjan operointimallin kullekin aikataululle.

Aikataulu kuvaa suunnitelman, jonka mukaan palvelulinja toteutetaan. Suunnitelmasta luodaan kuljetuskaluston operoitavia reittejä.

Pysäkki

Pysäkki on pysähdyspiste aikataulun sisällä. Pysäkillä on sijainti ja aikataulu. Aikataulun määrittävä viimeinen mahdollinen pysäkilletuloaika ja ensimmäinen mahdollinen pysäkiltä lähtöaika

.

Pysäkkejä on erilaisia. Pysäkin tyyppi määräytyy sen mukaan, käydäänkö pysäkillä aina vai ainoastaan tilauksesta ja sen mukaan, onko pysäkki pistemäinen vai tietyltä alueelta matkustajia kokoava. Erilaiset pysäkkityypit on esitetty alla olevissa kappaleissa.

Kiinteä pysäkki

Kiinteällä pysähdytään voimassa olevan aikataulun puitteissa aina, riippumatta siitä, onko pysäkille tehty tilausta vai ei. Kiinteällä pysäkillä voidaan nousta kyytiin / kyydistä tilaamalla matka etukäteen tai ilmestymällä pysäkille aikataulun puitteissa.

Joustava pysäkki

Joustavalla pysäkillä on aikataulu, jonka puitteissa pysäkillä käydään, mutta ainoastaan ennakkotilauksesta. Jos pysäkille ei ole tehty tilausta, pysäkillä ei pysähdytä.

Virtuaalipysäkki

Virtuaalipysäkillä ei ole aikataulua ja sillä pysähdytään ainoastaan tilauksesta. Virtuaalipysäkki kokoaa tietyltä alueelta tulleet osoitteet yhteen lähtöpisteeseen siten, että tilauksia ei operoida toivottuihin lähtö- ja kohdeosoitteisiin vaan virtuaalipysäkille. Esimerkiksi torilta tai torille tehdyt tilaukset voidaan kerätä kaikki yhdelle linja-autopysäkille tai taksitolpalle.

Alue

Alue määrittää maantieteelliset ja aikataululliset rajat palvelulinjan operoinnille. Palvelulinjan aikataulu saattaa sisältää useita alueita, joilla vieraillaan aikataulun mukaisesti. Alueiden sisällä tarjotaan tyypillisesti ovelta ovelle -palvelua, mutta alueen sisällä saattaa olla myös erityyppisiä pysäkkejä.

Reitti

Reitti pitää sisällään aikataulun määrittelemät säännöt, joiden puitteissa reitille voidaan lisätä matkustajien tilauksia. Reitti on optimoitu, aikataulutettu, autojen operoitavissa oleva kuvaus palvelulinjasta ja pitää sisällään suunniteltujen pysäkkien lisäksi reitin pysäkeille tai alueille reititetyt tilaukset.

Tilaus

Matkustajan tekemä tilaus, joka sovitetaan reittiin. Tilauksia voidaan tehdä pysäkeille tai alueille.

Auto

Reittiä operoivan auton perustiedot. Määrittää mm. reittien kapasiteettirajoitteen ja varustelutason. Autotieto voidaan määrittää aikataulutasolla siten, että sama auto operoi kaikki aikataulusta luodut reitit tai jokaiselle reitille erikseen.