Petri Nykänen

PÄIVYSTYSKALENTERISOVELLUS METATAVU OY:LLE

Opinnäytetyö

Liiketalouden ammattikorkeakoulututkinto

Tietojenkäsittelyn koulutus

2024

Kaakkois-Suomen ammattikorkeakoulu

Tutkintonimike	Tradenomi (AMK)
Tekijä/Tekijät	Petri Nykänen
Työn nimi	Päivystyskalenterisovellus Metatavu Oy:lle
Toimeksiantaja	Metatavu Oy
Vuosi	2024
Sivut	48 sivua
Työn ohjaaja(t)	Jukka Selin

TIIVISTELMÄ

Päivystyskalenteri on mikkeliläisen Metatavu-ohjelmointiyrityksen verkkosovellustoimeksianto. Sovelluksella seurataan Metatavun päivystävän henkilökunnan ajankohtia sekä päivystysvuoron palkanmaksun tilaa kalenterinäkymässä. Metatavulla oli sovellusta edeltävä toteutus, joka oli listausnäkymä päivystysviikkonumeroista ja -henkilöistä.

Kyseistä sovellusta käytettiin pohjana uuden päivystyskalenterin toteutuksessa. Uusi toteutus lisättiin osaksi vastikään perustettua Metatavu Home -työajan seuranta-alustaa. Päivystyskalenterisovellus toteutettiin web-kehityksessä standardisoituneita teknologioita käyttäen, kuten TypeScript-ohjelmointikieltä sekä tämän React-kirjastoa ja Serverless-sovelluskehystä. Kokonaisuudessa hyödynnetään myös Amazon Web Services -pilvipalvelualustaa (AWS). AWS S3 -palvelua käytetään päivystystietoja sisältävien JSON-tiedostojen säilyttämiseen sekä AWS Lambda -funktioita näiden käsittelemiseen ja lähettämiseen käyttöliittymään esitettäväksi.

Työssä käydään läpi kyseisiä teknologioita. TypeScriptia verrataan JavaScriptiin, josta TypeScript on staattiset tyypitykset sisältävä laajennus. JavaScript dynaamisena ja heikosti tyypitettynä kielenä ei vaadi tietotyyppien määrittämistä funktioiden parametreille tai muuttujille. Se mahdollistaa muuttujien arvojen uudelleenmäärittämisen mihin tahansa tietotyyppiin, mikä vähentää tarvittavaa syntaksia, mutta voi altistaa sovelluksen tyyppivirheille.

Työn toteutusvaiheessa esitellään projektin alustava suunnitelma, vanhan toteutuksen kehittämiskohteet ja uuden toteutuksen rautalankamalli. Palvelinpuolen toteutuksessa rakennettiin kolme AWS Lambda -funktiota. Yksi päivystystietojen hakemiseen, joka aktivoidaan kalenterinäkymään navigoidessa. Yksi päivystysvuoron palkanmaksun tilanpäivitykseen, joka aktivoidaan käyttöliittymän kautta. Yksi funktio päivystystietoja sisältävien tiedostojen luomiseen AWS S3 -palveluun, jonka aktivointi ajastetaan viikon välein. Funktiot määritettiin Serverless-sovellukseen, joka puolestaan tekee tarvittavat määritykset AWS-pilvialustalle.

Käyttöliittymä toteutettiin Metatavu Home -kehitysympäristöön käyttäen React-kirjastoa ja tämän Material UI -alikirjaston komponentteja. Komponentit määritettiin esittämään AWS Lambda -funktioista vastaanotettu data. Työn lopputulos vastasi toteutusvaiheen alussa tehtyä suunnitelmaa ja rautalankamallia.

Asiasanat: sovellus, AWS, React, TypeScript, Serverless

South-Eastern Finland University of Applied Sciences

Degree title	Bachelor of Business Administration
Author (authors)	Petri Nykänen
Thesis title	On-Call Calendar for Metatavu Oy
Commissioned by	Metatavu Oy
Time	2024
Pages	48 pages
Supervisor	Jukka Selin

ABSTRACT

The on-call calendar is a web application commissioned by the programming company Metatavu in Mikkeli. The application tracks the schedules of the on-call staff at Metatavu, as well as the status of payments for on-call shifts in a calendar view. Prior to this application, Metatavu had an earlier implementation which was a list view of on-call week numbers and personnel.

This previous application served as the basis for the development of the new on-call calendar. The new implementation was integrated into the recently established Metatavu Home time management platform. The on-call calendar application was developed using standardized web development technologies, such as the TypeScript programming language, React library and Serverless application framework. Additionally, the application utilizes the Amazon Web Services (AWS) cloud platform. AWS S3 service is used for storing JSON files containing on-call information, and AWS Lambda functions are used for processing and sending this information to the user interface.

During the project, the technologies used were examined. TypeScript was compared to JavaScript, it being an extension of JavaScript with static typing. JavaScript, being a dynamically and weakly typed language, does not require type definitions for function parameters or variables, and allows variable values to be reassigned to any data type, which reduces the required syntax but can expose applications to type errors.

During the implementation phase, an initial plan for the project, improvement areas for the previous implementation, and a wireframe for the new implementation were presented. Three AWS Lambda functions were built for the server-side implementation: one for retrieving on-call data, which is activated when navigating to the calendar user interface; one for updating the payment status of on-call shifts, activated through the user interface; and one function for creating files containing on-call data in AWS S3, with its activation scheduled weekly. These functions were defined in the Serverless application, which in turn created the necessary configurations on the AWS cloud platform.

The user interface was implemented in the Metatavu Home development environment by utilizing the React library and its Material UI component sublibrary. Components were configured to display the data received from AWS Lambda functions. The results of the project matched the initial plan and the wireframe established during the implementation phase

Keywords: application, AWS, React, TypeScript, Serverless

SISÄLLYS

1 JOHDANTO

Opinnäytetyön aiheena on esitellä päivystyskalenteriprojektin työvaiheet sekä tämän kehittämisessä käytetyt teknologiat ja käytännöt. Päivystyskalenteri on internetselaimessa käytettävä verkkosovellus, jossa esitetään Metatavun henkilökunnan päivystysvuoroviikot ja näiden lisätiedot kalenterinäkymässä.

Metatavu on Mikkelissä toimiva ohjelmistoalan yritys, joka tuottaa asiakkailleen pääasiassa sovelluspohjaisia ratkaisuja ja palveluita. Asiakaskunta vaihtelee kunnista pienyrityksiin ja asiakkaita on huomattava määrä, jolloin on muodostunut tarve päivystävälle henkilökunnalle mahdollisia ongelmatilanteita varten. Päivystysvuorossa on viikoittain yksi henkilö kerrallaan ja päivystysvuoroja on aikaisemmin seurattu verkossa toimivalla sisäisellä työkalulla, joka on listaus päivystysviikon numeroista yhdistettynä päivystävään henkilöön.

Metatavu on vastikään uudelleenrakentanut sisäisiä työaika- ja henkilöstöhallintotyökalujaan kondensoiden nämä osaksi yhteistä Metatavu Home -alustaa. Projektin toimeksiantona on nykyisen päivystyskalenterin päivittäminen, uudelleenkehitys sekä sisällyttäminen osaksi tätä kokonaisuutta.

Työkalun kehitysvaiheisiin kuuluvat palvelinpuolen siirtäminen ja mukauttaminen uuteen sovelluskehykseen, käyttöliittymän uudelleentoteutus sekä määrittäminen Metatavu Home -ympäristössä käytettäväksi. Sovellus toteutetaan käyttäen kirjoitushetkellä web-kehityksessä standardisoituneita teknologioita ja käytäntöjä, mainittavimpina TypeScript-ohjelmointikieltä, tämän React-käyttöliittymäkirjastoa ja Amazon Web Services -pilvipalveluita.

Työssä esitellään sovellukseen kehittämiseen käytetyt teknologiat luvussa 2, minkä jälkeen näitä sovelletaan luvun 3 toteutusvaiheessa. Luvussa 3 käydään läpi myös vanhan toteutuksen kehittämiskohteet, joista muodostetaan rautalankamalli uutta toteutusta varten. Työn loppuvaiheessa esitellään projektin tulokset, eli valmis sovellus ja lisäksi pohdinta sen kehittämisprosessista.

2 OPINNÄYTETYÖSSÄ KÄYTETYT TEKNOLOGIAT

Tässä luvussa esitellään projektissa käytettyjä keskeisimpiä teknologioita ja näiden käytäntöjä. Päivystyskalenteri on TypeScript-ohjelmointikielellä kirjoitettu verkkosovellus, joka Serverless-sovelluskehyksen kautta käyttää Amazon Web Services -pilvipalvelualustaa tietojen, kuten päivystysvuorojen ja henkilökunnan, tallentamiseen ja hakemiseen. Sovelluksen käyttöliittymä toteutetaan Reactilla, joka on käyttöliittymäkehitykseen tarkoitettu TypeScriptin kirjasto.

2.1 TypeScript

TypeScript on Microsoftin kehittämä ja ylläpitämä staattiset tyypitykset sisältävä laajennus JavaScript-ohjelmointikielestä. JavaScript on verkkosovellusten kehittämiseen käytetty ohjelmointikieli, joka lisää interaktiivisia osia ja logiikkarakenteita sivuston toiminnallisuuksiin tekemällä muutoksia dokumentin, eli tässä tapauksessa verkkosivun, rakenteeseen. JavaScript muodostaa yhdessä tyyli- ja merkintäkielien (CSS ja HTML) kanssa kokonaisuuden, jota hyödyntää 98 % kaikista WWW-sivuista (W3Techs s.a.). JavaScriptia voidaan siis pitää yhtenä internetin ydinteknologiana, jolla voidaan toteuttaa kaikenlaisia verkkosovellusten ratkaisuja palvelinpuolen toteutuksista käyttöliittymiin.

JavaScript ja DOM

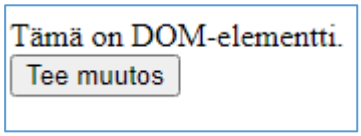
JavaScriptin tuomat interaktiivisuudet verkkosivuilla perustuvat DOM:in manipuloimiseen. DOM, eli Document Object Model, on rajapinta, joka määrittää dokumentin, kuten verkkosivun rakenteen sekä sen sisällä olevien elementtien hierarkian. (Document Object Model s.a, Introduction to the DOM: What is the DOM?). HTML-elementit ovat esimerkiksi painikkeita ja tekstipalstoja, jotka esitetään dokumentissa elementtiä kuvaavalla nimellä suljettuna ”< >”-tageilla. Tagit esitetään aina pareina, joista sulkeva tagi erotetaan vinoviivalla. Tagien sisälle rajataan elementin sisältö, joka on useimmiten tekstiä tai lapsielementti. Esimerkiksi Lihavoitu teksti.

Elementtien hierarkia HTML-dokumentissa voidaan hahmottaa puurakenteena, jossa on <HTML>-juurielementti, joka kattaa koko dokumentin. Tämän sisällä olevat elementit ovat lapsia, joiden sisällä voi puolestaan olla myös lapsielementtejä. (Document Object Model s.a, Using the Document Object Model: What is a DOM tree?).

Kuva 1. HTML-dokumentin rakenne ja JavaScriptin vaikutukset Visual Studio Code -näkymässä

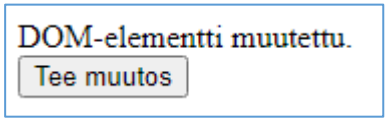
Kuvassa 1 on yksinkertainen HTML-dokumentti, jossa on kuvattu muutama sisäkkäinen elementti. JavaScriptin osuus dokumentissa on script-tagien sisällä riveillä 9–11 määritetty ”muutos”-funktio, joka muuttaa id-tunnisteella kohdistetun div-elementin tekstisisällön. Id-tunniste, joka on tässä tapauksessa ”dom”, poimitaan JavaScriptiin sisäänrakennetulla getElementById-metodilla. Funktiota kutsutaan rivillä 20 määritetyllä painikkeella (button).

Suurin osa HTML DOM:in rakenteesta esitetään internetselaimessa graafisina elementteinä, kuten aiemmin mainitut painikkeet, tekstikentät ja -palstat sekä otsikot. Kun DOM-rakennetta ladataan selaimeen esitettäväksi, kutsutaan tätä renderöinniksi (kuva 2).



Kuva 2. Div-elementti renderöitynä selaimessa ennen JavaScriptin tekemää muutosta

DOM-rakenteessa tehdyt muutokset päivittyvät selainnäkymässä, jolloin puhutaan uudelleenrenderöinnistä (kuva 3).



Kuva 3. Div-elementtiin kohdistunut muutos uudelleenrenderöitynä

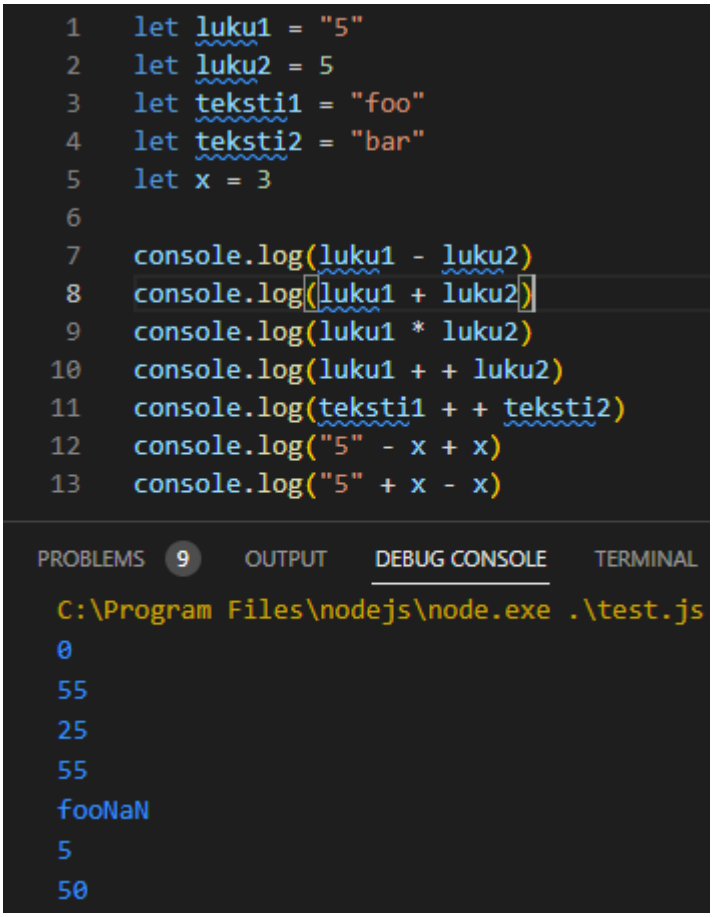
Kun verkkosivua päivitetään, renderöidään tällöin DOM-puu niin kuin se on alun perin esitetty dokumentissa, jolloin näkymä palautuu muutosta edeltävään tilaan.

JavaScriptin ja TypeScriptin eroavaisuudet

JavaScript määritellään samanaikaisesti dynaamisesti ja heikosti tyypitetyksi kieleksi. Heikko tyypitys tarkoittaa, että JavaScript ei vaadi tietotyyppien, kuten esimerkiksi numero tai merkkijono, määrittämistä muuttujille, funktioiden parametreille tai palautusarvoille, vaan JavaScript tulkkaa tietotyypit koodin kääntämisvaiheessa suoraan muuttujien arvoista (The Basics s.a). Dynaaminen tyypitys taas mahdollistaa, että samalle muuttujalle voidaan määrätä uusi arvo riippumatta sen tietotyypistä, kun taas päinvastaisesti vahvasti tyypitetyissä ohjelmointikielissä muuttujille tulee määrittää tämän arvon lisäksi sen tietotyyppi, jota myös uudelleenmääritettyjen arvojen tulee noudattaa. (JavaScript s.a, Intermediate: JavaScript data structures: Dynamic and weak typing.)

Heikossa tyypityksessä kahden eri tietotyypin välisiä operaatioita suoritettaessa toinen tyyppi yhdenmukaistetaan. JavaScriptissä käytetään esimerkiksi

plusmerkkiä matemaattisen yhteenlaskun lisäksi myös merkkijonojen ketjuttamisessa (eng. concatenation). Kuvassa 4 määritetään riveillä 1–5 erilaisia merkkijono- (eng. string) sekä numerotyypin (eng. number) muuttujia ja suoritetaan näiden välisiä operaatioita keskenään riveillä 7–13.



Kuva 4. Riveillä 1–5 muuttujien määritys. Riveillä 7–13 operaatioiden suoritus. Lopuksi tulokset näkyvät konsolissa

Tästä voidaan huomata, että rivin 8 yhteenlaskuoperaatioissa, jossa toinen muuttuja on numerotyyppiä ja toinen on merkkijonotyyppiä, saadaan tulokseksi 55. Kuitenkin rivin 7 operaatiossa miinusmerkillä samoilla muuttujilla suoritetaankin matemaattinen vähennyslasku ja saadaan tulokseksi 0.

Tämä johtuu siitä, että JavaScript suorittaa kääntämisvaiheen tulkkauskonflikteissa ensisijaisesti merkkijonotyypeille tarkoitettuja operaatioita, eli tässä tapauksessa tekstin ketjuttamisen (Type Coercion s.a). Miinusmerkillä suoritettavia operaatiota ei ole olemassa merkkijonoille, joten JavaScript tulkitsee tämän matemaattiseksi vähennyslaskuksi, kuten myös kertolaskun rivillä 9.

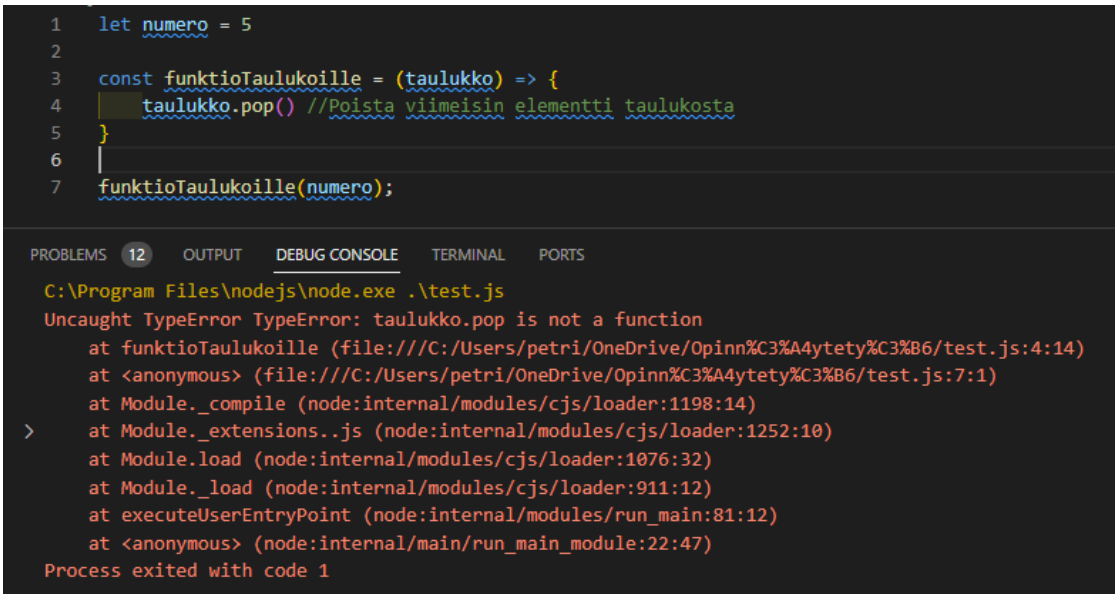
Kahden peräkkäisen plusoperaattorin kanssa asiat monimutkaistuvat hieman. Rivillä 10 saadaan numero 5:n ja merkkijono ”5”:n tulokseksi ketjutettu ”55”, kun suoritetaan näiden välinen operaatio plusmerkillä, joka on odotettu lopputulos aiemmin mainitun merkkijono-operaatioiden priorisoinnin perusteella. Rivillä 11 toistetaan sama operaatio, tosin tällä kertaa kahden merkkijonomuuttujan välillä, joiden arvot ovat ”foo” ja ”bar”.

Lopputulos voi olla hämmentävä, sillä noudattaen JavaScriptin operaatioiden priorisointijärjestystä tulisi lopputuloksen olla kyseisten merkkijonomuuttujien arvot ketjutettuina, eli ”foobar”. Konsolissa lukee kuitenkin ”fooNaN”. Tämä johtuu siitä, että JavaScript tulkitsee operaatiossa jälkimmäisen plusoperaattorin unaarioperaattoriksi.

Unaarioperaattorilla ilmaistaan matemaattisessa kontekstissa luvun polariteetti, eli onko luku positiivinen vai negatiivinen. JavaScriptissä lisäksi unaarioperaattori muuntaa operandinsa numerotyypiksi. Koska kyseessä oleva operandi, johon unaarioperaattori kohdistuu, ei ole numeraalinen merkkijono, muuntaminen epäonnistuu, jolloin JavaScript tulostaa muuttujan virhearvoksi NaN. Tämä on lyhenne sanoista ”Not a Number”, eli muuttujan arvo ei ole numero. (Unary Plus (+) s.a, Description.) Lopputulos on merkkijonon ”foo” ja tämän virhearvon ”NaN” ketjutus.

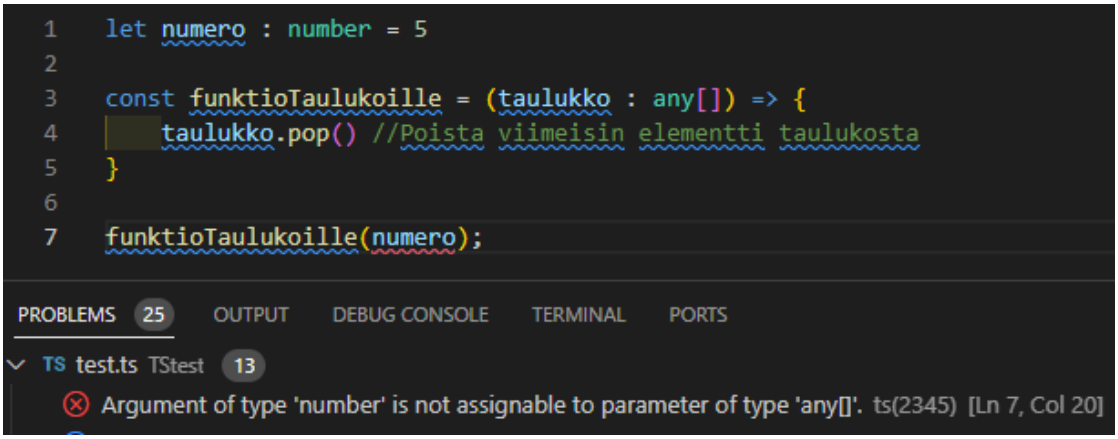
JavaScript tulkitsee tietotyypit koodia ajettaessa, joten tyyppivirheiden aiheuttamat ongelmat sekä näistä johtuvat mahdolliset ohjelman kaatumiset ilmenevät vasta koodin suoritusvaiheessa. Tällaisia tilanteita ovat esimerkiksi funktiossa määritetyt operaatiot, jotka eivät ole yhteensopivia sille parametrina syötetyn tietotyypin kanssa. Esimerkkinä tästä on kuvassa 5 esiintyvä funktio, joka ottaa parametrina taulukon (eng. array) ja joka suorittaa ainoastaan taulukoille soveltuvia operaatiota tai metodeja.

Taulukot ovat muuttujia, joiden sisälle voidaan tallentaa useita arvoja tai kuvaavammin alkioita. Taulukoiden operaatiot ovatkin tyypillisimmin sen sisällä olevien alkioiden manipulointi, poistaminen tai lisääminen. Koska JavaScriptissä ei määritetä tyyppejä funktioiden parametreille, voidaan funktioon antaa parametrina mikä tahansa muuttujatyyppi, vaikkapa tässä tapauksessa primitiivityyppi numero.



Kuva 5. Vääräntyyppinen muuttuja funktiossa aiheuttaa kaatumisen. Virhe ilmenee vasta koodin suorittamisessa

Funktion vastaanottaessa sille epäyhteensopivan numeromuuttujan parametrina yrittää se tästä huolimatta suorittaa taulukon alkion poiston kyseiseen muuttujaan. Sovellus kaatuu tyyppivirheeseen, koska funktion suorittamaa operaatiota ei voida suorittaa numerotyypin muuttujalle. (TypeError s.a.)



Kuva 6. Varoitus tyyppivirheestä VSCode-ohjelmointiympäristössä

TypeScript-versiossa funktioiden parametrien tyypit tulee määrittää, joten funktiossa suoritettavien operaatioiden ja parametrina vastaanotettujen tietotyyppien välisistä konfliktitilanteista varoitetaan moderneissa ohjelmointiympäristöissä (engl. integrated development environment, lyh. IDE) jo ennen koodin ajamista (kuva 6).

JavaScriptissä, ja täten myös TypeScriptissä, tunnetaan seitsemän erilaista primitiivityyppiä:

- string - merkkijono
- number - numero
- bigint – suuri kokonaisluku
- boolean – boolean-arvo
- undefined - määrittämätön
- symbol - symboli
- null – nolla, tyhjä

Primitiivillä tarkoitetaan yksinkertaista tietotyyppiä, jolla ei ole sisäänrakennettuja ominaisuuksia, verrattuna esimerkiksi olioihin (eng. object). Oliot ovat ohjelmoinnissa yleisiä tietorakenteita, jotka sisältävät primitiiveistä koostuvia avain-arvo-pareja tai funktioita, tai tarkemmin metodeja, kun funktio on osana oliota. (Kuva 7.) (MDN Web Docs Glossary: Definitions of Web-related terms s.a, Primitive.)

```
let numero = 5;

let merkkijono = "teksti";

let olio = {
    numero : 3,
    totuusarvo : false,
    merkkijono : "kirjoitus",
    funktio(arvo : string){console.log("Funktioon syötetty merkkijono: " + arvo)},
}
```

Kuva 7. Primitiivimuuttujat ja olio

Sekä JavaScript että TypeScript pystyvät päättelemään primitiivityypit suoraan muuttujan arvoista. TypeScriptillä ei siis tarvitse määrittää muuttujalle primitiivityyppejä erikseen, vaikkakin funktioiden parametreissa ja palautusarvoissa se on pakollista.

JavaScript ja TypeScript ovat vakiintuneita teknologioita verkkosovelluskehityksessä. Vuonna 2023 JavaScript oli suosituin ohjelmointikieli Github-versionhallinta-alustalla. Samana vuonna TypeScript syrjäytti Java-ohjelmointikielen kolmanneksi suosituimpana ohjelmointikielenä. (Daigle 2023, The Most Popular Programming Languages.)

Sovelluskehykset ja kirjastot

JavaScriptilla ja TypeScriptilla on kehitetty lukuisia kirjastoja (eng. library) ja sovelluskehyksiä (eng. framework) eri käyttötarkoituksiin. Ohjelmoinnissa kirjastolla tai sovelluskehyksellä tarkoitetaan ohjelmointikielellä luotuja valmiita, uudelleenkäytettäviä kokonaisuuksia, työkaluja tai yleisimmin komponentteja. Näillä kahdella termillä on paljon päällekkäisyyksiä ja näitä usein käytetäänkin vaihdellen samassa asiayhteydessä. Määrittävimpänä erona on se, että kirjasto sisältää valmiiksi ohjelmoituja yksittäisiä osia, usein funktioita, kun taas sovelluskehys näiden lisäksi sisältää valmiin rakenteen, rungon tai käytäntömallin, jota noudatetaan sovellusta kehitettäessä (Chris 2023).

Eräs JavaScriptin ja TypeScriptin kirjasto on esimerkiksi Luxon, joka sisältää valmiita funktioita ja metodeja päivämäärien ja ajan esittämiseen sekä muuntamiseen. Sovelluskehyksistä esimerkiksi ExpressJS on tarkoitettu JavaScriptillä (Node.js) luodulle palvelinpuolen verkkosovellukselle, jolla hallinnoidaan sovelluksen http-pyyntöjä.

2.2 React

Verkkosovellusten käyttöliittymienkin kehittämisessä on erilaisia vaihtoehtoja kirjastojen ja sovelluskehysten välillä. Suosittuja sovelluskehysvaihtoehtoja ovat muun muassa Googlen ylläpitämä Angular, Vue.js ja Svelte. Kirjastoista puolestaan React on kenties kaikkein tunnetuin. Reactia käytetäänkin myös tässä projektissa. React on vuonna 2013 Facebookin (nyk. Meta) luoma avoimen lähdekoodin kirjasto JavaScriptille ja sittemmin myös TypeScriptille.

JSX

Reactilla luodaan funktionaalisia käyttöliittymäkomponentteja, jotka renderöivät JSX-elementtejä. JSX on laajennus HTML-merkintäkielestä ja se mahdollistaa JavaScript-koodin kirjoittamisen suoraan HTML-elementin sisällä. React-komponentti on funktio, jonka palautusarvo on JavaScript-koodia JSX-tagien ympäröimänä, joka renderöidään internetselaimeen esitettäväksi. (Writing Markup with JSX s.a, JSX: Putting markup into JavaScript.)

Tämä eroaa hieman tavallisen JavaScriptin toimintavasta, jossa JavaScript tekee kohdistetusti erilliseen HTML-tiedostoon muutoksia, jotka sitten renderöidään selaimessa. React hyödyntää lisäksi Virtual DOM -tekniikkaa, jossa DOM-rakennetta simuloidaan sisäisesti Reactissa ennen kuin muutokset yhdenmukaistetaan varsinaisen DOM:in kanssa. Tämä prosessi nopeuttaa DOM:in päivittämistä ja renderöimistä. (Virtual DOM and Internals s.a, What is the Virtual DOM?.)

Hookit ja tilamuuttujat

Keskeisimpiä käytäntöjä Reactissa verrattuna tavalliseen JavaScriptiin on ”hook”-ominaisuuksien käyttö. Nämä ovat Reactin sisäänrakennettuja funktioita, joita kuvataan ”use”-etuliitteellä. Yleisimpiä hookeja on esimerkiksi useState, jolla määritetään tilamuuttujia.

Kun React renderöi JavaScript-koodia selaimeen, kaikki paikalliset muuttujat alustetaan siihen arvoon, johon ne on koodissa alun perin määritetty, vaikka nämä olisivatkin saaneet uusia arvoja sovelluksen käytön aikana. Näin tapahtuu myös näkymää uudelleen renderöidessä. Lisäksi React ei huomioi paikallisiin muuttujiin tehtyjä muutoksia, eikä nämä täten laukaise uudelleen renderöintiä. Tämä aiheuttaa sen, että selainnäkymässä ei havaita päivitettyjä muutoksia, vaikka koodipuolella nämä olisivatkin tehty. (State: A Component’s Memory s.a, When a regular variable isn’t enough.)

```
1   import React, { useState } from "react";
2
3   function Komponentti() {
4     const muuttuja = "Merkkijono 1";
5     const [tilamuuttuja, setTilamuuttuja] = useState("Merkkijono 2");
6
7     setTilamuuttuja(muuttuja);
8
9     return <div>{tilamuuttuja}</div>;
10  }
11
12  export default Komponentti;
13
```

Kuva 8. React-komponentin rakenne

Tilamuuttujat säilyttävät arvonsa myös uudelleenrenderöintien aikana ja näihin tehdyt muutokset myös käynnistävät uudelleen renderöinnin. Tilamuuttujat koostuvat taulukkoon määritetyistä arvopareista, jotka alustetaan useState-hookilla kuvan 8 rivin 5 mukaisesti. Taulukon ensimmäinen alkio on itse tilamuuttujan arvo ja toinen alkio on funktio, jolla tilamuuttujan arvoa muutetaan. Tätä funktiota kutsutaan myös ”setter”-nimellä ja yleisenä käytäntönä tämä nimetään ”set”-etuliitteellä. (State: A Component’s Memory, Anatomy of useState.)

Muita hookeja ovat esimerkiksi useEffect, johon voidaan määrittää renderöinnin jälkeisiä toimintoja, kuten esimerkiksi funktioiden käynnistys perustuen tilamuuttujan arvojen muutoksiin (Synchronizing with Effects, What are Effects and how are they different from events?). React-sovelluksen suorituskyvyn optimoimiseen käytetään useMemo- ja useCallback-hookeja, joilla vähennetään tarpeetonta uudelleenrenderöintiä (Built-in React Hooks, Performance Hooks).

Material UI

Reactille on saatavissa alikirjastoja, jotka koostuvat valmiiksi rakennetuista React-komponenteista. Tässä projektissa käytetään muun muassa Material UI -alikirjastoa, joka sisältää valmiita käyttöliittymäelementtejä, kuten tyylitellyt painikkeet ja erityisesti työssä käytetty kalenteripohja.

2.3 Amazon Web Services

Amazon Web Services (AWS) on Amazonin kehittämä ja ylläpitämä pilvipalvelualusta, joka tarjoaa palvelimettomia, automaattisesti skaalautuvia ja käyttötasoon mukautuvia palvelinpuolen kehittämiseen tarkoitettuja palveluja ja näin tarjoavat vaihtoehdon tavanomaiselle palvelinpuolen toteutukselle. Tällaisia palveluita ovat muun muassa tietokannat sekä pilvisäilytystila. (Klems 2018, 6.) Opinnäytetyössä hyödynnetään AWS Lambda - ja AWS S3 -palveluita.

AWS Lambda

AWS Lambda mahdollistaa koodin ja funktioiden ajamisen pilviympäristössä tavanomaisen palvelinpuolen sovelluksen sijasta, joka tavallisesti vaatisi kehittäjältä muun muassa oman ylläpidettävän palvelimen. Lambdan tapauksessa AWS hoitaa ylläpidon automaattisesti ja skaalautuvasti, jolloin resursseja ja laskentatehoa käytetään sekä veloitetaan käytön mukaan. (What is AWS Lambda? s.a, When to use Lambda.)

AWS Lambda -funktiot määritetään AWS:n omassa konsolissa, johon kirjauduutaan AWS-tunnuksilla palvelun verkkosivuilla (What is AWS Lambda? s.a, Getting started with Lambda: Create a Lambda function with the console). AWS Lambda -funktioiden aktivointi kytketään tapahtumakriteereihin (engl. events), eli funktiolle voidaan määritellä jokin tietty tapahtuma, josta funktion kutsu käynnistyy. Tällaisia ovat yleisesti http-pyynnöt, tyypillisimmin GET-pyynnöt käyttäjän vieraillessa verkkosivulla. AWS Lambda -funktiot voidaan määrittää reagoimaan säännöllisesti tiettyinä ajankohtina tai vaikka havaitessaan muutoksen jossain muussa AWS-palvelussa. (What is AWS Lambda? s.a, Lambda foundations: Concepts: Event.)

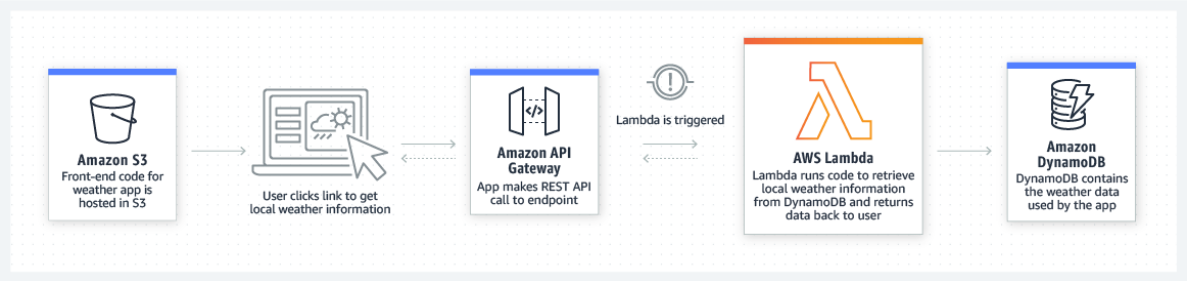
AWS S3

AWS S3 (Simple Storage Service) on pilvisäilytyspalvelu, jossa käyttäjä voi ladata ja säilöä mitä tahansa, jopa viiden teratavun suuruisia tiedostoja kuten mediaa, lokeja ja sovellustietoja (Uploading objects s.a). Tiedostot säilötään avain-arvo-periaatteella ”ämpäreihin” (engl. bucket) (Golden 2013, 61). Tämä voidaan hahmottaa perinteisenä tiedostokansiorakenteena, jossa bucket on juurikansio ja avain on tiedoston nimi.

Tallennustilan lisäksi AWS S3 tarjoaa versionhallintaominaisuuksia. Tiedostot voidaan määrittää replikoitumaan useammalle eri bucketille. Tämä parantaa tiedostojen saatavuutta tilanteissa, jossa tiedosto yhdessä bucketissa on vioittunut tai kyseinen bucket on jostain syystä toimimattomana, jolloin sama tie-

dosto on saatavissa muista bucketeista (Replicating objects s.a). Bucketit voidaan myös määrittää automaattisesti säilyttämään tiedostojen eri versiot (What is Amazon S3? s.a, How Amazon S3 works: S3 Versioning).

AWS S3 mahdollistaa tiedostojen luokittelun käytön ja tarpeen perusteella (storage classes), jolloin tiedostojen säilytystilan hinta skaalautuu dynaamisesti. Aktiivisimmissa tiedostoissa säilytyshinta on suurempi ja täten myös näiden saatavuus on parhainta luokkaa. Alemmilla luokituksilla tiedostojen säilytystilan hinnoittelu pienenee samalla kun tiedostojen noutamiseen käytetty viive kasvaa. (What is Amazon S3? s.a, Features of Amazon S3: Storage classes.) Lisäksi bucketeissa on mahdollisuus käyttöoikeusasetuksiin, jolloin haltija voi rajata ne tahot, joilla on pääsy bucketeihin ja lisäksi yksittäisiin tiedostoihin. Julkinen pääsy on pois päältä oletuksena.



Kuva 9. Erilaisten AWS-palveluiden välinen vuorovaikutus (AWS Lambda s.a)

AWS S3:n vahvuuksia on sen synergia muiden AWS-palveluiden kanssa. Kuvassa 9 havainnollistetaan AWS S3:n vuorovaikutusta AWS Gatewayn ja AWS Lambdan kanssa, jossa AWS Lambda -funktio aktivoituu tapahtumakriteerin täyttyessä, kun käyttäjä klikkaa AWS API Gatewayhyn kytkettyä linkkiä. API Gatewayhyn vastaanotettu pyyntö käynnistää AWS Lambda -funktion, joka kuvan esimerkissä noutaa säätiedot Amazon DynamoDB -tietokantapalvelusta käyttäjälle näytettäväksi. Opinnäytetyössä hyödynnetään samaa vuorovaikutusta päivystystietoja sisältävien JSON-tiedostojen kanssa, jotka tallennetaan ja noudetaan bucketista.

2.4 Serverless-sovelluskehys

Serverless on sovelluskehys, jolla kehitetään pilvipalvelualustoja hyödyntäviä sovelluksia. Tässä projektissa käytetyn AWS:n lisäksi Serverlessin tukemia muita tunnettuja pilvipalvelualustoja ovat Azure, Google Cloud ja Cloudflare

(Serverless Infrastructure Providers s.a). Lisäksi sovelluksia voidaan kehittää useammalla kielellä: Node.js:n (JavaScript) lisäksi tuettuja kieliä ovat muun muassa Python ja Java (Serverless Framework Concepts s.a).

Serverless-viitekehyksellä luodaan palvelinpuolen (eng. back end) osuus sovelluksesta. Node.js:llä toteutettu Serverless-sovellus eroaa tyypillisestä back-end Node.js -sovelluksesta eniten toimintatavaltaan ja rakenteeltaan. Tyypillisessä palvelinpuolen sovelluksessa määritetään suoraan päätepisteet, jotka reagoivat tuleviin pyyntöihin palauttamalla vastauksena esimerkiksi kyseisellä päätepisteellä olevan resurssin. Kyseiset sovellukset vaativat lisäksi myös omat palvelimet, joiden ylläpito ja hallinta on kehittäjän vastuulla.

Serverless-sovellus määrittää automaattisesti tarvittavat päätepisteet pilvialustalle. Sovellus kytketään pilvipalvelualustassa olevalle käyttäjätilille ja määritetään reagoimaan tapahtumiin (events), joita esimerkiksi voivat olla pilvipalvelun vastaanottamat pyynnöt. Päätepisteiden lisäksi Serverless-sovellus tekee automaattisesti muut tarvittavat määritykset pilvipalvelualustan puolella. (AWS Credentials s.a.)

YAML-kuvaus

YAML on datan serialisointikieli. YAML-lyhenne tulee sanoista ”Yet Another Markup Language” tai sittemmin uudelleenmääritellystä muodosta ”YAML Ain’t Markup Language”. Nimityksen muutoksella on haluttu painottaa YAML:n tarkoitusta datan serialisointiin käytettynä kielenä eikä merkintäkielenä, kuten esimerkiksi HTML:ää, jolla kuvataan dokumentteja. (ingydotnet 2013.)

Datan serialisoinnilla tarkoitetaan tietojen muuttamista tilapäisesti helpommin käsiteltävään tai luettavaan tiedostomuotoon, joka voidaan palauttaa alkuperäiseen muotoonsa myöhempää käyttöä varten. Tällaisia ovat esimerkiksi skenaariot, jossa tiedosto halutaan lähettää päätelaitteelta toiselle verkon välityksellä, jolloin tiedosto serialisoidaan lähetystä varten ja muunnetaan tämän jälkeen takaisin alkuperäiseen muotoonsa. YAML:llä kuvataan tässä tapauksessa myös Serverless-sovelluksen konfiguraatiota, kuten funktioita, tietotyyppejä ja yhteyksiä ihmisille luettavassa muodossa.

Opinnäytetyössä kuvaukseen vaaditaan muun muassa tiedot AWS-ympäristöstä, AWS Lambda- funktioiden määrityksiä ja AWS S3 -bucketien nimistä sekä alueista. Kuvauksen perusteella Serverless muodostaa yhteyden ja rakentaa sovelluksen AWS-ympäristöön. Serverlessissä YAML-kuvaus voidaan korvata muilla kielillä (Serverless Framework Concepts s.a, Alternative configuration format). Esimerkiksi tässä projektissa Serverless-konfiguraatio toteutetaan TypeScriptillä erillisessä serverless.ts-tiedostossa. YAML:ää hyödynnetään muualla opinnäytetyössä esimerkiksi käyttöliittymän funktioiden määrityksessä.

Kirjautuminen ja käyttäjätodennus

Serverlessin YAML-kuvaukseen voidaan määrittää myös käyttäjien todentaja (engl. authorizer). Metatavu käyttää Keycloakia kirjautumiseen ja käyttäjien todentamiseen. Keycloak on Red Haltin kehittämä avoimen lähdekoodin identiteetti- ja pääsynhallintaratkaisu, joka tarjoaa minimalistisen ja helppokäyttöisen vaihtoehdon käyttäjien todennukselle.

Keycloak-instanssia ylläpidetään virtuaalikoneympäristössä tai konttialustalla (engl. container, containerization platform) kuten Dockerilla tai Kubernetesillä, eikä se täten vaadi erillisen todennuslogiikan ohjelmoimista projektille (Thorgersen 2020, Keycloak Overview). Keycloakissa voidaan määrittää lisäksi käyttäjäryhmät- ja roolit, tyypillisesti pääkäyttäjät (admin) ja tavalliset käyttäjät (mt. Roles). Metatavu Homen kontekstissa esimerkiksi admin-käyttäjillä on oikeus hyväksyä tai hylätä työntekijöiden hakemia lomia.

2.5 Päivystyskalenterin rakenne

Päivystyskalenteri on tiivistettynä TypeScriptillä kirjoitettu sovelluskokonaisuus, joka muodostuu Reactilla toteutetusta käyttöliittymästä sekä Serverlessillä toteutetusta palvelinpuolesta, ja jota ylläpidetään Amazon Web Services -pilvipalvelussa. Serverlessiin kirjoitetaan kolme AWS Lambda -funktiota: päivystystietojen haku AWS S3 -bucketista, päivystyksen palkanmaksun tilan päivitys sekä päivystävän henkilökunnan viikoittainen tarkistus. Näistä kaksi ensimmäistä kytketään REST API -tapahtumakriteeriin.

Funktiot käynnistyvät, kun AWS:ssä olevat päätepisteet (AWS API Gateway) vastaanottavat http-pyynnön, eli kun käyttäjä lähettää nämä pyynnöt joko vierailemalla verkkosivulla tai suorittaa käyttöliittymässä olevia toimintoja. Viikoittainen tarkistus sen sijaan määritetään käynnistymään aikataulutetusti viikon välein.

Koska päivystyskalenteri on ominaisuus Metatavu Home -palvelussa, käyttöliittymän kehittäminen hyödyntää jo tässä palvelussa käytettyinä olevia teemoja, tyylittelyjä ja kirjastoja. Metatavu Homen käyttöliittymän elementit ovat komponentteja Material UI -alikirjastosta, jolloin on yhdenmukaista käyttää myös tämän paketin kalenterikomponenttia, joka määritetään esittämään lambda-funktioiden vastaanottama data.

3 TOTEUTUSVAIHE

Metatavulla on jo olemassa oleva ratkaisu päivystysvuorojen seurantaan (kuva 10). Kyseessä on verkkosovellus, joka on erillään Metatavu Home -kokonaisuudesta ja joka toimii omana verkkosivunaan. Nykyinen päivystyskalenteri toimii samoilla periaatteilla kuin tämän uudelleentoteutuskin. Eli siinä on palvelinpuoli, joka määrittää lambda-funktiot AWS:ään. Ne hakevat tarvittavat tiedot S3 bucketeista, jotka esitetään lopulta käyttöliittymässä.

Suurimmat tekniset eroavaisuudet löytyvät käytetyistä sovelluskehyksistä: nykyisen kalenterin palvelintoteutus on rakennettu SST:llä sekä käyttöliittymä Sveltellä. Nykyinen päivystyskalenteri on lisäksi full stack -sovellus, eli sekä palvelin- että käyttöliittymäpuoli on toteutettu samaan kehitysympäristöön.

Kuva 10. Metatavun päivystyskalenterin edellinen toteutus

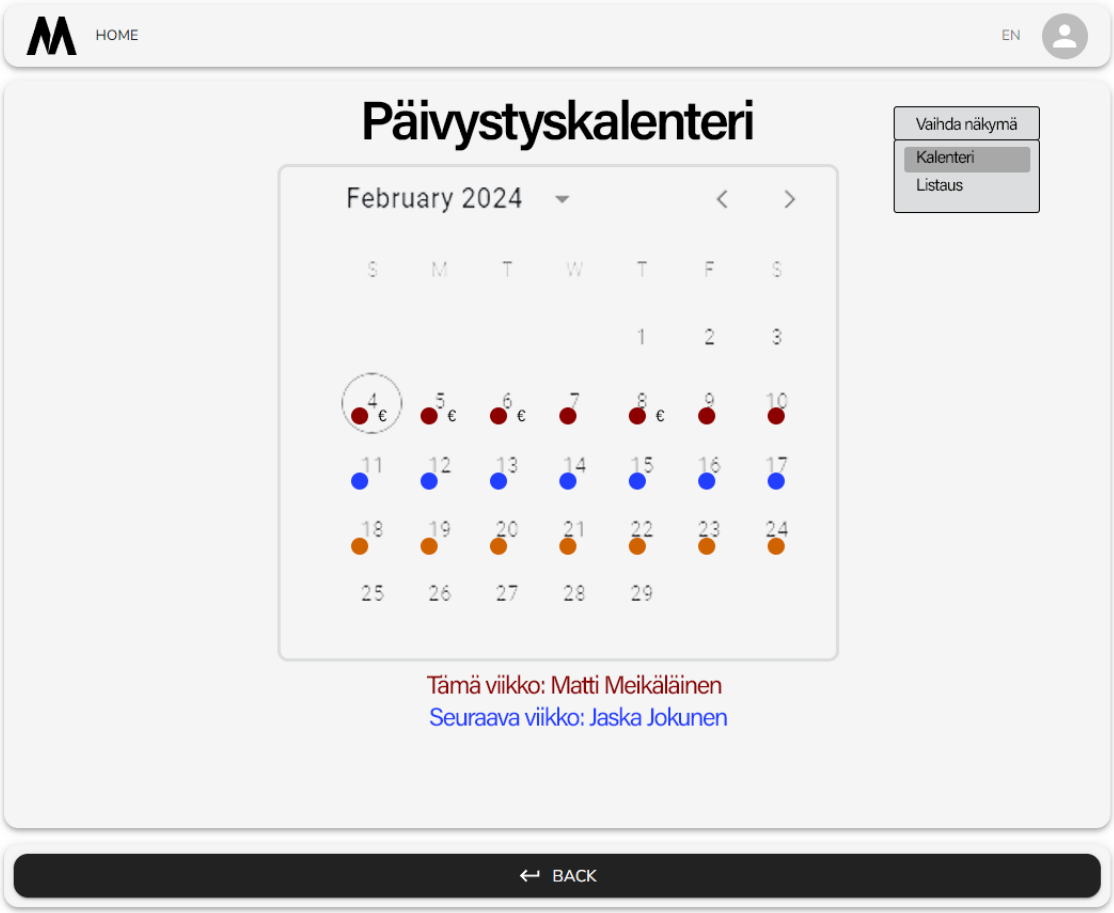
Kuvassa 10 on kuvankaappaus sovelluksen käyttöliittymästä. Näkymä on kolmen pystyrivin listaus, jossa on päivystysviikon palkanmaksun tila, viikkonumero ja päivystävä henkilö.

3.1 Kehittämiskohteet

Edellinen toteutus on helppokäyttöinen ja ajaa käyttötarkoituksensa mainiosti, joten kehityskohteet ovatkin tämän myötä minimaaliset. Vanha toteutus tarjoaa ajan seurannan vain viikkoformaatissa, mikä poikkeaa yleisistä kalenteritottumuksista ja -formaateista vaatien toisinaan käyttäjää tarkastelemaan viikkonumeroa vastaavat päivät perinteisestä kalenterista. Luomalla perinteisen kalenterinäkymän, jossa on viikot, päivät sekä kuukaudet saavutetaan universaalimpi käyttökokemus.

Käyttökokemuksen tyytyväisyyden maksimoimista varten lisäksi vanha näkymä säilytetään vaihtoehtona sitä suosivalle käyttäjäkunnalle. Näkymän vaihto onnistuisi pudotusvalikon kautta, jossa on vaihtoehdot kahden näkymän välillä. Näkymän vaihto tallentuisi myös selaimen paikallismuistiin, jotta tulevilla käyttökerroilla avautuu suoraan käyttäjän suosima näkymä.

Palkanmaksua seurataan vasemmanpuoleisen pystyrivin rasteilla. Kuka tahansa voi poistaa tai asettaa rasteja vapaasti, eikä muutoksesta tule mitään varoitusta tai vahvistusta. Tämä altistaa sovelluksen huomaamattomille vahinkopainalluksille. Uudessa toteutuksessa on vahvistusikkunat rastin poistamiselle ja asettamiselle.



Kuva 11. Päivystyskalenterin rautalankamalli

Uuden käyttöliittymän keskeisin periaate on pyrkiä ylläpitämään sen jo olemassa olevaa selkeyttä. Vanha toteutus on jo valmiiksi helppo hahmottaa ja ymmärtää, jolloin selkeyttä parantavia muutoksia tarvitsee tehdä vain vähän.

Tavoitteena on, että sovelluksen käyttötarkoitus ymmärretään jo yhdellä silmäyksellä. Tämä saavutetaan korostamalla tärkeimpiä elementtejä selkeillä kokoeroilla ja väreillä niin, että elementtien selkeä hierarkia on erotettavissa, kuten kuvan 11 rautalankamalli esittää.

Rautalankamallista hahmottuvat tärkeimmät elementit, eli itse kalenteri, josta ilmenevät nykyisen viikon sekä lähitulevaisuuden päivystävä henkilökunta. Päivystävät henkilöt pyritään merkitsemään erottuvilla tunnisteilla hyödyntäen ensisijaisesti värejä. Päivystyskalenteriin kuuluu myös toiminto, jossa päivystysviikkovuoron palkanmaksu merkitään tehdyksi.

Palkanmaksun tila esitettäisiin kalenterinäkymässä henkilötunnisteen ohessa omalla tunnisteellaan esimerkiksi valuuttaa tai maksutapahtumaa kuvaavalla v symbolilla, ikonilla tai värityksellä. Itse palkanmaksun tilan muuttamistoimintoa ei ole rautalankamallissa näkyvillä. Tavoitteena olisi, että käyttäjän painaessa tiettyä päivää kalenterinäkymässä avautuisi ikkuna, jossa olisi tarkempien tietojen lisäksi lisätoiminnot, kuten kyseinen palkanmaksun tilan päivittäminen.

Vanhan toteutuksen palvelinpuoli käyttää samaa logiikkaa hyödyntäviä AWS Lambda -funktioita toimintoihinsa. Keskeisin uudelleentoteutus tässä tapauksessa on näiden funktioiden siirtäminen sovelluskehykseltä toiselle, eli SST:stä Serverlessiin. Funktioita ei voida kuitenkaan kopioida sellaisenaan kehitysympäristöstä toiseen, vaan ne tulee ensin mukauttaa Serverlessin määritysten mukaan. Tässä vaiheessa käydään läpi myös funktioiden koodit, joita siistitään tarvittaessa esimerkiksi noudattamalla alan nykystandardien mukaisia nimeämiskäytäntöjä tai lisäämällä virheiden käsittelyjä.

3.2 Palvelinpuolen toteutus

Projektin lähtötilanteessa lambda-funktioiden siirtäminen Serverlessiin oli jo osittain aloitettu. Osaa funktioista oli mahdollista kutsua onnistuneesti paikallisella komennolla palvelinpuolen kehitysympäristössä. Sovellukseen kirjoitetaan kolme AWS Lambda-funktiota: päivystystietojen hakemiseen listOnCallData, palkanmaksun päivittämiseen updatePaid ja päivystystiedostojen luomiseen weeklyCheck.

AWS SDK

Palvelinpuolen kehitysympäristössä käytetään aws-sdk-pakettia (software development kit), jonka metodit mahdollistavat AWS-palveluiden kutsumisen suoraan kehitysympäristössä. Projektissa on määritetty TypeScript-tiedosto s3-utils.ts tiedostojen hakemista ja tallentamista varten. Tiedostossa aws-sdk-paketista tuodaan AWS S3:n luokkailmentymä, jonka metodeja käytetään kyseisten toimintojen suorittamiseen (kuva 12).

Kuva 12. s3-utils.ts, aws-sdk-paketista käytetään S3-luokkailmentymää

Tiedostossa määritetään lisäksi nimiavaruus S3Utils, jonka alaisuuteen kirjoitetaan kaksi funktiota: loadJson ja saveJson. Funktiot suorittavat nimiensä mukaiset toiminnot muodostamalla ensin yhteyden AWS S3 -bucketiin käyttämällä s3-luokkailmentymän asianmukaisia metodeja. loadJson-funktio vastaanottaa parametrina s3-luokkailmentymän lisäksi bucketin ja tiedoston nimen (key). Nämä syötetään s3-luokkailmentymän getObject-metodiin, jolloin parametrien mukainen tiedosto haetaan nimeä vastaavasta bucketista. (Kuva 13.)

Funktion keskivaiheessa kuvassa 13 riveillä 22–25 validoidaan haetut tiedot. Mikäli tietoja ei ole olemassa, funktio keskeytyy ja palauttaa null-arvon, eli tyhjän. Muussa tapauksessa JSON-tiedosto jäsennetään TypeScriptissä käsiteltävään taulukkomuotoon ja asetetaan palautusarvoksi rivillä 27. Koko funktio suoritetaan try-catch-lohkon sisällä mahdollista virheidenkäsittelyä varten.

```
6   namespace S3Utils {
7
8     /**
9      * Loads JSON from S3
10     *
11     * @param s3 S3 client
12     * @param key object key
13     * @returns object data as JSON or null if not found
14     */
15    export const loadJson = async <T>(s3: S3, bucket: string, key: string): Promise<T | null> => {
16      try {
17        const object = await s3.getObject({
18          Bucket: bucket,
19          Key: key
20        }).promise();
21
22        const body = object.Body;
23        if (!body) {
24          return null;
25        }
26
27        return JSON.parse(body.toString());
28      } catch (e: any) {
29        if (e.code === "NoSuchKey") {
30          return null;
31        }
32
33        throw e;
34      }
35    }
```

Kuva 13. s3utils.ts, S3Utils-nimiavaruus: loadJson-funktio

loadJson-funktiossa JSON-tiedoston tallennukseen kutsutaan s3-luokkailmentymän putObject-metodia (kuva 14) rivillä 45. Kuten aikaisemmassa sa-

veJson-funktiossa, kyseinen metodi vastaanottaa parametrina funktiolle syötetyt bucketin ja tiedoston nimen sekä tiedoston sisällön. Eli tässä tapauksessa JSON-dataa, joka stringify-metodilla muutetaan taulukkomuodosta JSON-muotoon. Lopuksi metodi tallentaa tiedoston S3-bucketiin.

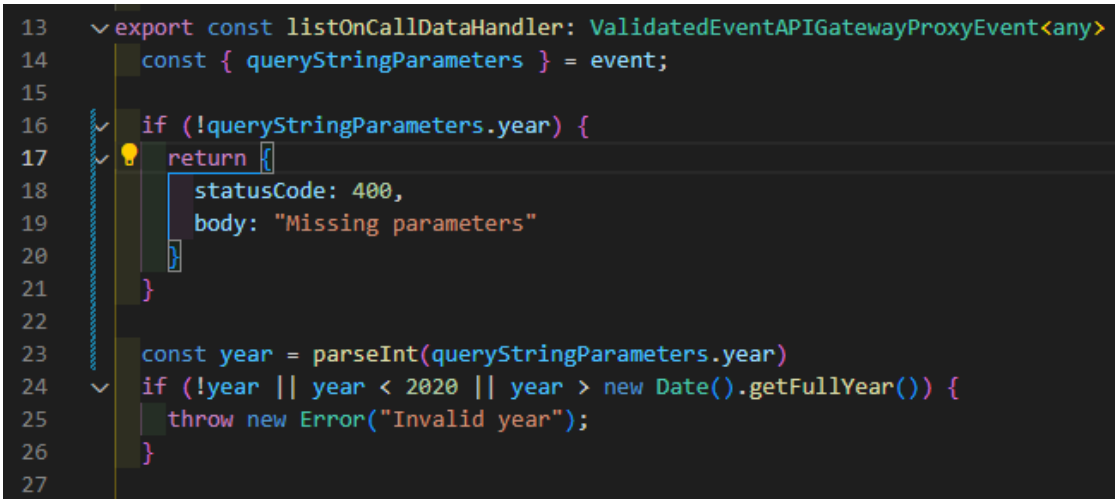
```
37    /**
38     * Saves data as JSON to S3
39     *
40     * @param s3 s3 client
41     * @param key object key
42     * @param data data to be saved as JSON
43     */
44    export const saveJson = async <T>(s3: S3, bucket: string, key: string, data: T): Promise<void> => {
45      await s3.putObject({
46        Bucket: bucket,
47        Key: key,
48        Body: JSON.stringify(data)
49      }).promise();
50    }
51  }
```

Kuva 14. s3utils.ts, S3Utils-nimiavaruus: saveJson-funktio

Nimiavaruus on määritetty ”export”-avainsanaa käyttäen, jolloin se voidaan tuoda myös muihin TypeScript-tiedostoihin. Nimiavaruuden funktioita hyödynnetään AWS Lambda -funktioissa.

Päivystystietojen hakeminen listOnCallData-lambda-funktiolla

Päivystystietojen hakua varten määritetään AWS Lambda-funktio nimeltään listOnCallData. Funktio kytketään AWS API Gateway -päätepisteeseen, jolloin funktiota kutsutaan vain silloin, kun API Gatewayn määrittämä URL-osoite vastaanottaa GET-pyynnön. Koska päivystystietoja on useammalle vuodelle, kohdennetaan pyyntö tietylle vuosiluvulle asettamalla URL-osoitteen loppuun kyselymerkkijono (engl. query string). Kyselymerkkijono muodostuu avain-arvo-parista, jonka avaimena on ”year” (vuosi) ja hakuarvona on vuosiluku.



```
13  export const listOnCallDataHandler: ValidatedEventAPIGatewayProxyEvent<any>
14    const { queryStringParameters } = event;
15
16    if (!queryStringParameters.year) {
17      return {
18        statusCode: 400,
19        body: "Missing parameters"
20      }
21    }
22
23    const year = parseInt(queryStringParameters.year)
24    if (!year || year < 2020 || year > new Date().getFullYear()) {
25      throw new Error("Invalid year");
26    }
27
```

Kuva 15. list-on-call-data/handler.ts, listOnCallDataHandler, parametrien tarkistus

Funktion määritykset alkavat syötettyjen parametrien oikeellisuuden tarkastuksella (kuva 15). Koska vuosiluku poimitaan kyselymerkkijonona URL-osoitteesta, on käyttäjän teoriassa mahdollista jättää hakuarvo kokonaan pois. Tämä johtaa palvelinvirheeseen siinä vaiheessa, kun funktio koettaa käsitellä olematonta tai virheellistä vuosilukua. Kuvassa 15 riveillä 16–20 määritetään lohko, jossa tarkastetaan if-lauseella, onko kyselyparametriä olemassa. If-lauseen ehdon täyttyessä funktio keskeytyy ja palauttaa vastauksen, joka sisältää http-tilakoodin 400, eli puutteellinen pyyntö (engl. bad request), sekä lisäviestin, joka kuvailee virhettä tarkemmin. Jos kyselymerkkijono on olemassa, alustetaan tämä muuttujaksi ja numeeriseksi arvoksi rivillä 23. Lisätarkastus suoritetaan lohkossa riveillä 24–26 siltä varalta, että vuosiluku on väärässä muodossa. Arvon on oltava suurempi kuin nykyinen vuosiluku tai pienempi kuin 2020, sillä Metatavulla ei ole tätä vuotta edeltäviä päivystystietoja.

Kun vastaanotettu vuosiluku on validoitu, suoritetaan funktion keskivaiheilla päivystystietojen haku AWS S3:sta (kuva 16). Tätä varten hyödynnetään aikaisemmin esiteltyä S3Utils-nimiavaruutta, jolla kutsutaan AWS S3:n toimintoja. Tässä tapauksessa kuvan 16 esittämällä tavalla riveillä 35–37 haetaan S3:n bucketista kolme JSON-tiedostoa, jotka ovat nimikartta (nameMap), vuositiedosto (yearFile) ja maksutiedosto (paidFile)

```
28    const s3 = new S3();
29
30    const nameMapFile = "name-map.json";
31    const yearFile = `${year}.json`;
32    const paidFile = "paid.json";
33
34    const bucket = Config.get().onCall.bucketName;
35    const nameMap = await S3Utils.loadJson<{ [key: string]: string }>(s3, bucket, nameMapFile) || {};
36    const data = await S3Utils.loadJson<OnCallEntry[]>(s3, bucket, yearFile);
37    const paidData = await S3Utils.loadJson<PaidData>(s3, bucket, paidFile) || {};
38
39    if (!data || !nameMap || !paidData) {
40      throw new Error("No data");
41    }
```

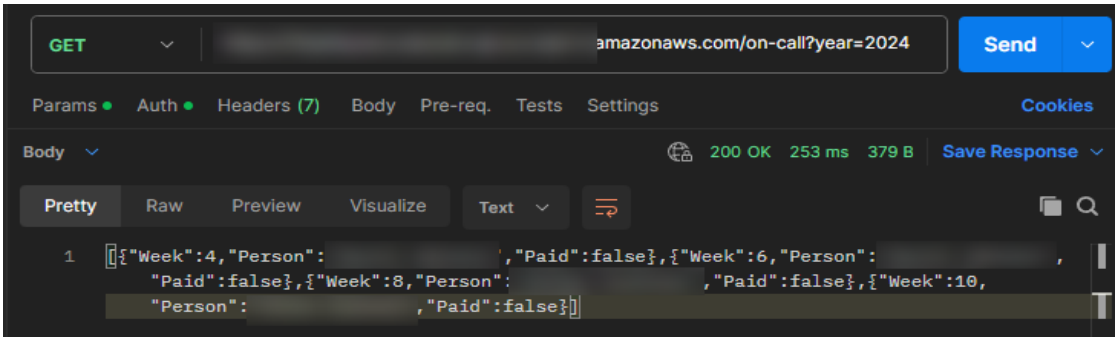
Kuva 16. list-on-call-data/handler.ts, listOnCallDataHandler, Funktion keskivaihe

Kuvassa 17 riviltä 45 alkaen muodostetaan näistä tiedostoista palautusarvo JSON-muotoon. JSON sisältää olioita, jotka koostuvat kolmesta avain-arvo-parista: viikkonumero, päivystävän henkilön nimi ja päivystysvuoron palkanmaksun tila. Kukin olio vastaisi yhtä viikkoa kalenterinäkymässä. Funktio palauttaa JSON-datan ja onnistuneen pyynnön http-tilakoodin.

```
43    return {
44      statusCode: 200,
45      body: JSON.stringify(data.map((entry: any) => {
46        return {
47          ...entry,
48          Person: nameMap[entry.Person] || entry.Person,
49          Paid: paidData[year] && paidData[year][entry.Week] || false
50        }
51      }))
52    };
53  }
```

Kuva 17. list-on-call-data/handler.ts, listOnCallDataHandler. Funktion palautusarvo

Kuvassa 18 demonstroidaan funktion toiminta Postman-sovelluksessa. Funktiota kutsutaan lähettämällä GET-pyyntö tälle määritettyyn AWS API Gateway URL-osoitteeseen, jolloin saadaan vastauksena kuvassa 18 esitelty palautusarvo.



Kuva 18. Postman-sovellus, funktion testaaminen GET-pyynnöllä

Sovelluksen käyttöliittymä hyödyntää samaa periaatetta ja poimii GET-pyynnöstä vastaanotetun datan esitettäväksi kalenterissa, kun käyttäjä vierailee kalenterinäkymässä.

Palkanmaksun tilan päivittäminen updatePaid-lambda-funktiolla

Kun päivystysvuorosta on saatu palkanmaksu, voidaan se merkitä maksetuksi updatePaid-lambda-funktiolla. Kuten päivystystietojenkin haussa, funktion kutsu käyttää REST-rajapintaa. Erona kuitenkin on, että funktiota kutsutaan lähettämällä POST-pyyntö GET-pyynnön sijaan AWS API Gateway URL-osoitteeseen.

```
13  export const updatePaidHandler: ValidatedEventAPIGatewayProxyEvent<any>
14    const { year, week, paid } = event.body as UpdatePaidRequestBody;
15
16    if (!year || year < 2020 || year > new Date().getFullYear()) {
17      throw new Error("Invalid year");
18    }
19
20    if (!week || week < 1 || week > 53) {
21      throw new Error("Invalid week");
22    }
23
24    if (paid === undefined) {
25      throw new Error("Invalid paid status");
26    }
```

Kuva 19. update-paid/handler.ts, updatePaidHandler. Pyynnön tietojen validisointi

Lisäksi tilan päivittämiseen tarvittavia tietoja ei poimita kyselymerkkijonosta, vaan POST-pyynnön rungosta (engl. body), josta poimitaan tarvittavat tiedot, eli vuosiluku, viikkonumero ja palkanmaksun tila (true tai false). Kuten aiemmassa Lambda-funktiossa, myös päivityksen yhteydessä tietojen oikeellisuus

tarkistetaan, jotta funktioon ei pääse virheellisiä tai olemattomia tietoja (kuva 19).

```
28      const s3 = new S3();
29      const bucket = Config.get().onCall.bucketName;
30      const paidFile = "paid.json";
31
32      const paidData = await S3Utils.loadJson<PaidData>(s3, bucket, paidFile) || {};
33      if (!paidData[year]) paidData[year] = {}
34      paidData[year][week] = paid;
35      await S3Utils.saveJson(s3, bucket, paidFile, paidData);
36
37      return {
38        statusCode: 200,
39        body: "Paid status updated"
40      };
```

Kuva 20. update-paid/handler.ts. updatePaid-funktion loppuvaiheet ja palautusarvo

Kuvassa 20 rivillä 32 kutsutaan S3Utils-nimiavaruuden funktiota saveJson. Funktiolla muodostetaan yhteys AWS S3:n bucketiin, johon päivitetyt tiedot tallentaan JSON-muodossa. Funktion palautusarvo on http-vastauksen tilakoodi sekä viesti onnistuneesta päivityksestä.

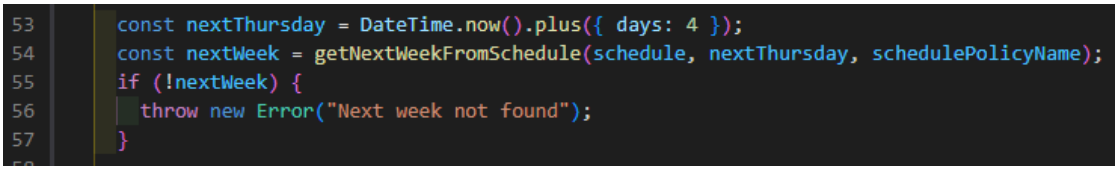
Viikoittainen tietojen tarkistaminen

Aikaisemmin läpikäytiin listOnCall -Lambda-funktio, jolla päivystyskalenteriin haetaan päivystystietoja sisältävät JSON-tiedostot AWS S3:sta, jotka puolestaan muotoillaan käyttöliittymässä näytettäväksi. Kyseiset JSON-tiedostot muodostetaan Splunk-sovellusrajapinnasta (engl. API, application programming interface), jossa päivystysvuorot luodaan. Tätä varten määritetään weeklyCheck -Lambda-funktio, joka muodostaa yhteyden Splunkin sovellusrajapintaan ja hakee sovelluksen kautta luodut päivystysvuorot sekä muuttaa nämä JSON-muotoon (kuva 21).

```
export const weeklyCheckHandler : ValidatedEventAPIGatewayProxyEvent<any> = async () => {

  const { apiId, apiKey, schedulePolicyName, teamOnCallUrl } = Config.get().splunkApi

  const splunkTeamOnCallUrl = teamOnCallUrl

  const schedule = await (await fetch(`${splunkTeamOnCallUrl}/schedule?daysForward=4&daysSkip=3`, {
    headers: {
      'X-VO-Api-Id': apiId,
      'X-VO-Api-Key': apiKey,
      'Accept': 'application/json'
    }
  })).json() as SplunkSchedule;
```

Kuva 21. weekly-check/handler.ts. weeklyCheck-funktion alku, tietojen haku Splunk-rajapinnasta

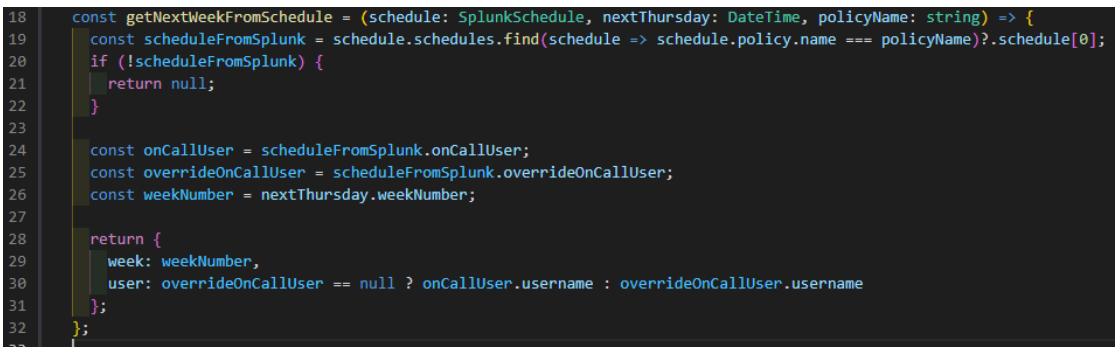
Koska Lambda-funktio on aikataulutettu ja aktivoituu CRON-aikataulun mukaan sunnuntaisin, haetaan seuraavan viikon päivystystiedot päivämäärämuuttujalla rivillä 53 (kuva 22). Muuttujaa käytetään parametrina seuraavalla rivillä funktiossa ”getNextWeekFromSchedule”.



```
53      const nextThursday = DateTime.now().plus({ days: 4 });
54      const nextWeek = getNextWeekFromSchedule(schedule, nextThursday, schedulePolicyName);
55      if (!nextWeek) {
56        throw new Error("Next week not found");
57      }
```

Kuva 22. weekly-check/handler.ts, weeklyCheckHandler-funktio. Seuraavan viikon määritykset

Kyseinen funktio käy läpi aikaisemmin kuvassa 21 Splunkista haetun aikataulun ja muotoilee tästä tulevan viikon päivystystiedot (kuva 23). Funktion palautusarvona on seuraavan viikon Splunkin päivystysaikataulusta muodostettu olio, jossa on viikkonumero ja päivystävän henkilön käyttäjänimi.

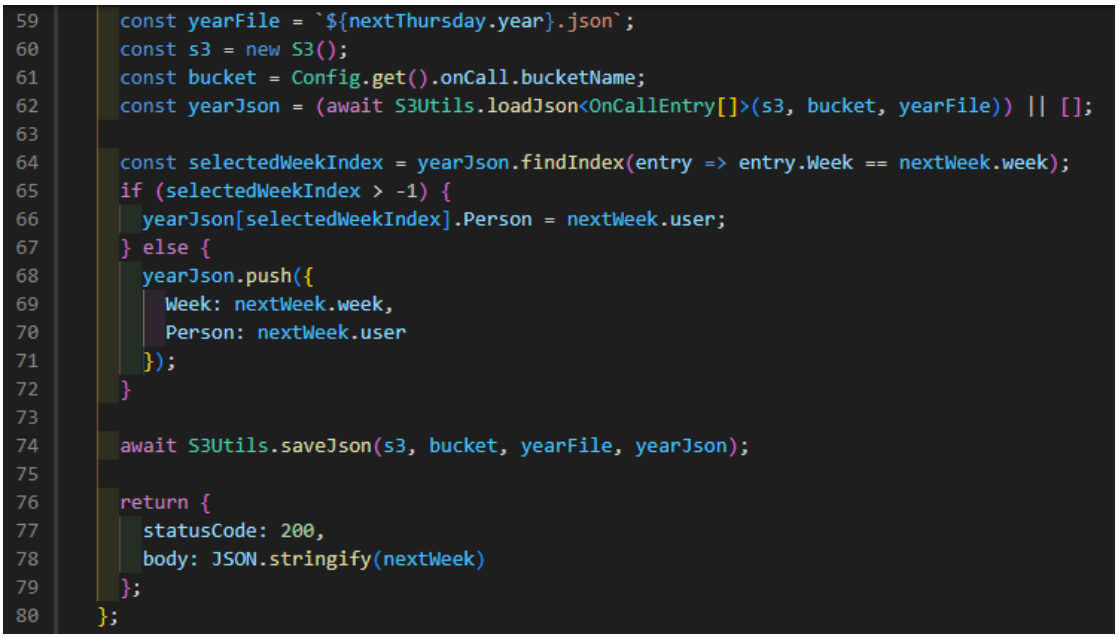


```
18  const getNextWeekFromSchedule = (schedule: SplunkSchedule, nextThursday: DateTime, policyName: string) => {
19    const scheduleFromSplunk = schedule.schedules.find(schedule => schedule.policy.name === policyName)?.schedule[0];
20    if (!scheduleFromSplunk) {
21      return null;
22    }
23
24    const onCallUser = scheduleFromSplunk.onCallUser;
25    const overrideOnCallUser = scheduleFromSplunk.overrideOnCallUser;
26    const weekNumber = nextThursday.weekNumber;
27
28    return {
29      week: weekNumber,
30      user: overrideOnCallUser == null ? onCallUser.username : overrideOnCallUser.username
31    };
32  };
```

Kuva 23. weekly-check/handler.ts, getNextWeekFromSchedule-funktio

Funktion loppuosassa (kuva 24) riveillä 59–62 määritetään muuttujiksi JSON-tiedoston nimi (yearFile) ja AWS S3:n luokkailmentymä bucketiin tallentamista

varten, jolla tiedostonimeä vastaava JSON-tiedosto haetaan bucketista. Kuvan 24 rivien 64–72 lohkossa käydään läpi bucketista vastaanotettu JSON-tiedosto, joka sisältää kuluvan vuoden päivystysviikot. Tiedosto on muutettu taulukkomuotoon käsittelyä varten. Rivillä 64 alustetaan selectedWeekIndex-muuttuja, jonka arvo on indeksiluku.



```
59    const yearFile = `${nextThursday.year}.json`;
60    const s3 = new S3();
61    const bucket = Config.get().onCall.bucketName;
62    const yearJson = (await S3Utils.loadJson<OnCallEntry[]>(s3, bucket, yearFile)) || [];
63
64    const selectedWeekIndex = yearJson.findIndex(entry => entry.Week == nextWeek.week);
65    if (selectedWeekIndex > -1) {
66      yearJson[selectedWeekIndex].Person = nextWeek.user;
67    } else {
68      yearJson.push({
69        Week: nextWeek.week,
70        Person: nextWeek.user
71      });
72    }
73
74    await S3Utils.saveJson(s3, bucket, yearFile, yearJson);
75
76    return {
77      statusCode: 200,
78      body: JSON.stringify(nextWeek)
79    };
80  };
```

Kuva 24. weekly-check/handler.ts, weeklyCheckHandler-funktio. JSON-tiedoston haku, käsittely ja tallennus

Indeksiluku saadaan etsimällä findIndex-metodilla tiedostosta nykyisen vuoron järjestysnumero käyttämällä viikkoa hakuehtona. Taulukoiden indeksointi alkaa numerosta 0, eli mikäli findIndex ei löydä päivystysviikkoa vastaavaa järjestysnumeroa, metodi palauttaa tulokseksi -1, jolloin kyseistä viikkoa ei löydy tiedostosta.

Tällöin rivien 65–71 if-lauseen else-lohkossa puuttuva viikko ja henkilön nimi lisätään taulukkoon. Jos findIndex-metodi löytää haetun viikon, muokataan tiedostossa kyseisen viikon kohdalle päivystävän henkilön nimitunniste. Käsittelyn jälkeen päivitetty tiedosto tallennetaan AWS S3 -bucketiin ja funktio palauttaa http-tilakoodin onnistuneesta pyynnöstä ja lisäksi bucketiin lisätyn tiedon seuraavan viikon päivystäjästä.

```
5   export default {
6     handler: `${handlerPath(__dirname)}/handler.main`,
7     events: ONCALL_WEEKLY_SCHEDULE_TIMER ? [
8       {
9         schedule: ONCALL_WEEKLY_SCHEDULE_TIMER,
10      }
11    ] : []
12  };
```

Kuva 25. weekly-check/index.ts, tapahtumakriteerin määritys

Funktion tapahtumakriteeri määritetään samassa kansiossa erillisessä index.ts-tiedostossa. weeklyCheck-lambda-funktio käynnistetään sunnuntaisin CRON-aikataulun mukaan, joka määritetään ONCALL_WEEKLY_SCHEDULE_TIMER-ympäristömuuttujan arvoksi (kuva 25).

Google Cloud Build

AWS Lambda -funktioiden määritysten ja oikein toimimisen varmistamisen jälkeen sovelluksesta luodaan Google Cloud Build -sovellus, jossa sovelluksen tuotantoversiota ajetaan ja ylläpidetään. Tällöin Serverless konfiguroi myös AWS Lambda -funktiot AWS-pilvialustalle. Tämän Serverless-sovelluksen repositorioita säilötään Github-versionhallinta-alustalla, jossa toimeksiantoa varten on tehty sovelluksesta erillinen versio (engl. branch) sisältäen luvussa läpikäydyt lambda-funktiot. Google Cloud Build -versio luodaan automaattisesti, kun kyseinen versio yhdistetään pääkehitysversion kanssa (engl. develop branch).

3.3 Käyttöliittymän toteutus

Kun Lambda-funktiot ovat käyttövalmiita ja yhteydessä verkkoon, voidaan käyttöliittymä määrittää kutsumaan kyseisiä funktioita aktivoimalla näissä määritetyt tapahtumakriteerit. Lambda-funktioita on kirjoitettu kolme kappaletta, mutta koska näistä yhden (weeklyCheck) aktivoiminen on sidottu aikatauluun, eikä se täten tarvitse käyttöliittymällistä syöttöä, käyttöliittymä määritetään aktivoimaan vain updatePaid- ja listOnCallData-lambda-funktioita. Serverless-sovellus on konfiguroinut AWS-pilvialustalle API AWS Gateway -URL-osoitteen, jolla näitä kahta lambda-funktiota kutsutaan.

AWS API Gateway hyödyntää REST-rajapintaa, jolloin se voi vastaanottaa GET-pyyntöjä listOnCallData-lambda-funktion kutsumista varten sekä POST-pyyntöjä updatePaid-lambda-funktiota varten

OpenAPI Client Generator

Http-pyyntöjä lähettäviä käyttöliittymäfunktioita voidaan kirjoittaa käyttöliittymän kehitysympäristössä itse, mutta tässä tapauksessa hyödynnetään OpenAPI Client Generator -pakettia, joka generoi automaattisesti tarvittavat funktiot ja koodit tätä varten. Ennen kuin tiedostoja voidaan generoida, luodaan tarvittavista lambda-funktioista ja tietotyypeistä YAML-kuvaus, jonka avulla OpenAPI Client Generator luo asiakasrajapinnan (engl. client) käyttöliittymän kehitysympäristöön (kuva 26). Asiakasrajapinta sisältää generoidut valmiit TypeScript-koodit ja tietotyypit, jotka ajetaan käyttöliittymässä Lambda-funktioiden kutsumiseksi.

```
334   "/on-call":
335     get:
          Try it | Audit
336       operationId: listOnCallData
337       summary: Lists on call personnel data from S3
338       description: Lists on call personnel data from S3
339       tags:
340         - On call
341       parameters:
342         - name: year
343           in: query
344           required: true
345           schema:
346             type: string
347           description: Selected year for on call entries
348       responses:
349         "200":
350           description: A list of on call personnel for a selected year
351           content:
352             application/json;charset=utf-8:
353               schema:
354                 type: array
355                 items:
356                   $ref: "#/components/schemas/OnCall"
357         default:
358           description: Invalid or missing year
```

Kuva 26. YAML-kuvaus listOnCallData -Lambda-funktiosta

Kun käyttöliittymän kehitysympäristöä asennetaan, tässä tapauksessa käyttämällä Node Package Manager -paketinhallintatyökalua (npm), suoritetaan ”npm install”-komennon yhteydessä myös build-client -skripti.

Kuva 27. package.json-tiedosto, NPM skriptit

Skripti sisältää OpenAPI Client Generatorin komennot tiedostojen luomiseksi. (Kuva 27.)

Palvelinpuolen yhdistäminen käyttöliittymään

Metatavu Homen käyttöliittymän kehitysympäristössä on TypeScript-tiedosto api.ts, jossa määritetään yhteys sovellusrajapintoihin ja lambda-funktioihin käyttämällä OpenAPI:n generoimia ilmentymiä. Ilmentymät vaativat parametreikseen pyyntöjen muodostamiseen vaadittavan URL-osoitteen ja lisäksi todennustunnisteen (engl. access token), joka saadaan Keycloak-kirjautumistiedoista. AWS API Gatewayn URL-osoitteella kutsutaan henkilötietoja käsitteleviä Lambda-funktioita ja on täten arkaluontoinen tieto, jolloin määritetään tämä ympäristömuuttujaksi (engl. enviroment variable) muotoon ”config.lambdas.url”. Muuttujan arvo poimitaan kehitysympäristön env-tiedostosta muuttujan nimeä vastaavalla avaimella, mikäli se on asetettu.

Kuva 28. api.ts, OpenAPI Client Generatorin ilmentymät getApiClient-funktion palautusarvona

Kuvassa 28 nähdään getApiClient-funktio, jonka palautusarvoina on OpenAPI Generatorin luomat ilmentymät sovellusrajapinnoille kullekin Metatavu Homen toiminnolle. Ilmentymät sisältävät metodeja esimerkiksi tietojen hakemista ja muokkaamista varten. Muut ilmentymät konfiguroidaan käyttäen sovellusrajapintakohtaisia URL-osoitteita, jolloin riville 43 määritetään onCallApi-ilmentymää varten oma getLambdaConfiguration-muuttuja, johon syötetään AWS API

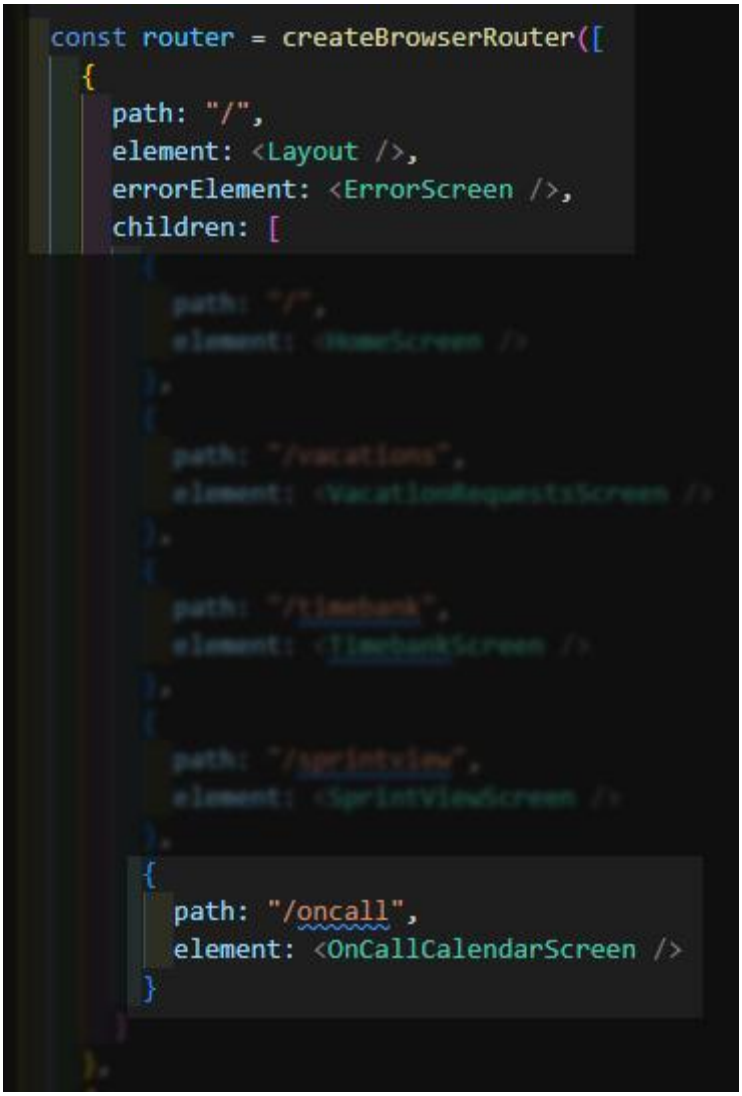
Gateway -URL-osoite. getApiClient-funktio on määritetty ”export”-avainsanaa käyttäen, jolloin sitä voidaan kutsua missä tahansa muussa kooditiedostossa, kuten kuvan 29 React-komponentissa OnCallCalendarScreen.

Kuva 29. on-calendar-screen.tsx, listOnCallData -Lambda-funktiota kutsutaan OnCallCalendarScreen-käyttöliittymäkomponentissa

Komponentissa on getOnCallData-funktio, jossa kutsutaan onCallApi-ilmentymän metodia ”listOnCallData”. Metodi lähettää GET-pyynnön AWS API Gateway -URL-osoitteeseen, jolla kutsutaan metodin nimeä vastaava AWS Lambda-funktio. Itse getOnCallData-funktio kytketään Reactin useEffect-hookiin, jolloin funktiota kutsutaan kalenterisivun renderöityessä.

Reitti kalenterinäkymään

Käyttöliittymän muodostaessa onnistuneesti yhteyden lambda-funktioihin voidaan määrittää käyttöliittymäkomponentit, jossa lambda-funktioista vastaanotettu data esitetään. Kalenterinäkymälle luodaan oma reittinsä Metatavu Homen juurikomponentissa App.tsx react-router-pakettia hyödyntäen (kuva 30).



Kuva 30. App.tsx-juurikomponentti, reitittimen määritykset

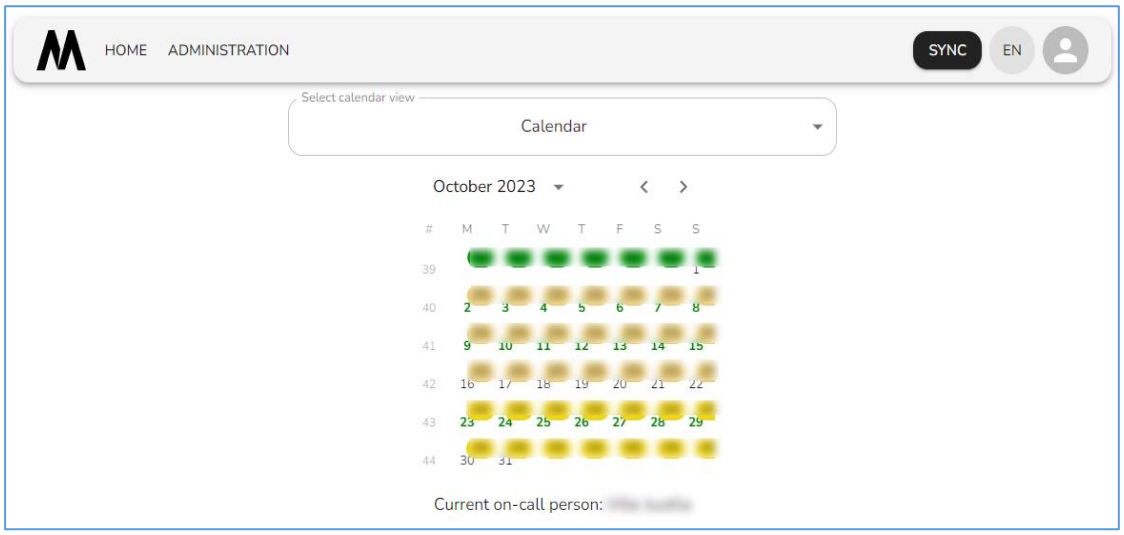
App.tsx renderöi juurikomponenttina kaikki sen palautusarvoksi asetetut lapsikomponentit, kuten react-router-paketin RouterProvider-komponentin, joka mahdollistaa router-muodostimessa määritettyjen komponenttien renderöimisen reittikohtaisesti (kuva 31).

Kuva 31. App.tsx-juurikomponentin palautusarvo, jonne RouterProvider-komponentti sijoitetaan

Kalenterinäkymään voidaan nyt navigoida /oncall-reitin kautta, jolloin selaimeen renderöidään OnCallCalendarScreen.tsx-komponentti.

Kalenterinäkymän määrittäminen

Näkymässä on kalenterikomponentti, jossa päivystysviikot ilmaistaan värillisillä päivystävän henkilön nimikirjainlaatoilla. Kalenteriviikon päivänumeroiden vihreä teksti ja tekstin lihavointi ilmaisee, että päivystysviikon palkka on maksettu. (Kuva 32.)



Kuva 32. OnCallCalendarScreen.tsx-komponentti renderöitynä internet-selaimessa

OnCallCalendarScreen-komponentti koostuu Material UI -lapsikomponenteista, joista osaa renderöidään ehdollisesti, kuten kuvassa 33 riveillä 184 ja

185. Yksi tällainen funktio on näkymän tärkein elementti, eli itse kalenterikomponentti. Funktiossa renderöidään joko perinteinen kalenterinäkymä tai aiempaa päivystyskalenteritoteutusta vastaava listaus. Uudessa näkymässä käytetään Material UI -alikirjaston DateCalendar-komponenttia.

```
160   return (
161     <Box sx={{ display: "flex", flexDirection: "column", justifyContent: "center" }}>
162       <FormControl sx={{ width: "50%", textAlign: "center", margin: "auto" }}>
163         <InputLabel id="calendarSelect">Select calendar view</InputLabel>
164         <Select
165           labelId="calendarSelect"
166           id="calendarSelect"
167           label="Select calendar view"
168           value={isCalendarView ? "Calendar" : "List"}
169         >
170           <MenuItem value={"Calendar"} onClick={() => handleCalendarViewChange(true)}>
171             Calendar
172           </MenuItem>
173           <MenuItem value={"List"} onClick={() => handleCalendarViewChange(false)}>
174             List
175           </MenuItem>
176         </Select>
177       </FormControl>
178       <OnCallHandler
179         open={open}
180         setOpen={setOpen}
181         onCallEntry={selectedOnCallEntry}
182         updatePaidStatus={updatePaidStatus}
183       />
184       {renderCalendarOrList()}
185       {renderCurrentOnCall()}
186       <Button onClick={() => setOpen(true)}>TEST</Button>
187     </Box>
188   );
189 };
```

Kuva 33. OnCallCalendarScreen-komponentin palautusarvo

DateCalendar-komponentin ominaisuuksiin (engl. properties, props) määritetään riveillä 142 ja 143 funktiot tapahtumankäsittelijöihin, kun kuukausi- tai vuosinäkymää vaihdetaan kalenterissa, tässä yhteydessä aktivoituu selectedYear-tilamuuttujan setter-funktio setSelectedYear, joka asettaa selectedYear-tilamuuttujan arvoksi näkymässä olevan vuosiluvun. (Kuva 34.)

```
138  const renderCalendarOrList = () => {
139    if (isCalendarView)
140      return (
141        <DateCalendar
142          onMonthChange={(value) => setSelectedYear(value.year)}
143          onYearChange={(value) => setSelectedYear(value.year)}
144          displayWeekNumber={true}
145          slots={{ day: populateCalendarWeeks }}
146        />
147      );
148
149    return <OnCallListView selectedYear={selectedYear} updatePaid={updatePaidStatus} />;
150  };
```

Kuva 34. Ehdollinen kalenterinäkymän tai listauksen renderöintifunktio

Tämä tehdään siksi, että päivystystietoja hakeva AWS Lambda-funktio on määritetty hakemaan päivystystiedot vuosi kerrallaan, jolloin vuoden vaihtuessa kalenterinäkymässä täytyy tiedot hakea uudelleen kyseiselle vuodelle. Tietojen hakemiseen määritetty käyttöliittymäfunktio getOnCallData kytketään tästä syystä useEffect-hookiin, jonka riippuvuudeksi asetetaan selectedYear-tilamuuttuja (kuva 35). useEffect reagoi tilamuuttujan muutoksiin, jolloin vuoden vaihtuessa haetaan tiedot automaattisesti uudelleen samaa tilamuuttujaa parametrina käyttäen.

```
35  useEffect(() => {
36    getOnCallData(selectedYear);
37  }, [selectedYear]);
```

Kuva 35. Päivystystietojen haku kiinnitettynä useEffect-hookiin

DateCalendar-komponentin ominaisuutena on myös ”slots”, joka näkyy kuvan 34 rivillä 145. Ominaisuuteen voidaan määrittää kalenteripäiville lisäelementtejä, kuten tässä tapauksessa kalenteripäivien nimikirjainlaatat. Nimikirjainlaatat generoidaan populateCalendarWeeks-funktiolla (kuva 36).

Funktiossa alustetaan riveillä 108–112 erilaisia muuttujia, jotka toimivat joko funktion palautusarvona olevien Badge ja PickersDay komponenttien sisältönä tai ehtoina näiden tyylimäärityksille. Kyseiset muuttujat ovat rivin 107 calendarProps-taulukon alkioita ja näiden uudelleenmuotoiluja.

```
102  const populateCalendarWeeks = (
103    props: PickersDayProps<DateTime> & { highlightedDays?: number[] }
104  ) => {
105    const { day, outsideCurrentMonth, ...other } = props;
106
107    const calendarProps = generateOnCallWeeks();
108    const badgeColor = calendarProps.find((item) => item.date === day.toISODate())?.badgeColor;
109    const populatedDay = calendarProps.map((item) => item.date).includes(day.toISODate());
110    const personName = calendarProps.filter((item) => item.date === day.toISODate())[0]?.person;
111    const personInitials = personName?.split(" ")[0][0] + personName?.split(" ")[1][0];
112    const paidStatus = calendarProps.filter((item) => item.date === day.toISODate())[0]?.paid;
113
114    return (
115      <Box>
116        <Badge
117          key={props.day.toString()}
118          overlap={"circular"}
119          sx={{ color: "black", ".MuiBadge-overlapCircular": { backgroundColor: badgeColor } }}
120          badgeContent={populatedDay ? personInitials : undefined}
121        >
122          <PickersDay
123            onClick={() => selectEntryToUpdate(calendarProps, day)}
124            sx={paidStatus ? { color: "green", fontWeight: "bold" } : {}}
125            {...other}
126            outsideCurrentMonth={outsideCurrentMonth}
127            day={day}
128          />
129        </Badge>
130      </Box>
131    );
132  };
```

Kuva 36. OnCallCalendarScreen.tsx, populateCalendarWeeks-funktio

calendarProps-taulukko saadaan palautusarvona generateOnCallWeeks-funktiosta (kuva 37). Koska lambda-funktioilta vastaanotetut päivystystiedot ilmaisevat vain viikkonumeroita, täytyy yksittäiset päivät generoida ohjelmallisesti. Tätä varten on määritetty generateOnCallWeeks-funktio.

Funktio alkaa alustamalla tyhjä taulukko rivillä 64 (kuva 37). Taulukon sisältö täytetään silmukoita käyttäen. Jokaista päivystystietojen elementtiä, eli yhtä viikkoa kohden suoritetaan seitsemän kierroksen silmukka alkaen riviltä 68. Jokainen kierros lisää taulukkoon viikkonumeroa vastaavan yhden viikonpäivän. Tässä yhteydessä liitetään olioon mukaan myös henkilön nimi, päivystysviikon palkanmaksun tila ja nimilaatan väri henkilön nimen perusteella. Lopuksi täytetty taulukko annetaan funktion palautusarvoksi.

```
63  const generateOnCallWeeks = () => {
64    const onCallWeeks: OnCallCalendarEntry[] = [];
65    onCallData.forEach((item) => {
66      const weeks = DateTime.fromObject({ weekNumber: Number(item.Week), weekYear: selectedYear });
67
68      for (let i = 0; i < 7; i++) {
69        onCallWeeks.push({
70          date: weeks.plus({ days: i }).toISODate(),
71          person: item.Person,
72          paid: item.Paid,
73          badgeColor: stringToColor(item.Person)
74        });
75      }
76    });
77    return onCallWeeks;
78  };
```

Kuva 37. OnCallCalendarScreen.tsx, generateOnCallWeeks-funktio

Taulukosta poimitaan esimerkiksi nimilaatan väri ja henkilön nimikirjaimet, jotka syötetään Badge-komponenttiin. Taulukossa on myös palkanmaksun tila, jolla muotoillaan päivänumeroiden tyyliä sen perustella, onko palkkaa maksettu. Itse palkanmaksun päivittämiseen tarkoitettu funktio määritetään PickersDay-komponenttiin, jota klikkaamalla kutsutaan selectEntryToUpdate-funktiota

```
<Box>
  <Badge
    key={props.day.toString()}
    overlap={"circular"}
    sx={{ color: "black", ".MuiBadge-overlapCircular": { backgroundColor: badgeColor } }}
    badgeContent={populatedDay ? personInitials : undefined}
  >
    <PickersDay
      onClick={() => selectEntryToUpdate(calendarProps, day)}
      sx={paidStatus ? { color: "green", fontWeight: "bold" } : {}}
      {...other}
      outsideCurrentMonth={outsideCurrentMonth}
      day={day}
    />
  </Badge>
</Box>
```

Kuva 38. OnCallCalendarScreen.tsx. selectEntryToUpdate-funktio kytkettynä PickersDay-komponentin onClick-tapahtumankäsittelijään

selectEntryToUpdate-funktio ei lähetä itse pyyntöä palkanmaksun päivittämistä varten, vaan asettaa valitun päivystysviikon selectedOnCallEntry-tilamuuttujan arvoksi (kuva 39).

```
const selectEntryToUpdate = (onCallEntries: OnCallCalendarEntry[], date: DateTime) => {
  const selectedEntry = onCallEntries.find((item) => item.date === date.toISODate());
  setSelectedOnCallEntry(selectedEntry);
  setOpen(true);
};
```

Kuva 39. OnCallCalendarScreen.tsx. selectEntryToUpdate-funktion määritykset

Lisäksi boolean-tyypin open-tilamuuttujalle annetaan ”true”-arvo. Kyseisellä arvolla kontrolloidaan OnCalHandler-komponentin, tai tarkemmin tämän lapsielementtinä olevaa Dialog-komponentin, näkyvyyttä (kuva 40).

```
return (
  <Box sx={{ display: "flex", flexDirection: "column", justifyContent: "center" }}>
    <FormControl sx={{ width: "50%", textAlign: "center", margin: "auto" }}>
      <InputLabel id="calendarSelect">Select calendar view</InputLabel>
      <Select
        labelId="calendarSelect"
        id="calendarSelect"
        label="Select calendar view"
        value={isCalendarView ? "Calendar" : "List"}
      >
        <MenuItem value={"Calendar"} onClick={() => handleCalendarViewChange(true)}>
          Calendar
        </MenuItem>
        <MenuItem value={"List"} onClick={() => handleCalendarViewChange(false)}>
          List
        </MenuItem>
      </Select>
    </FormControl>
    <OnCallHandler
      open={open}
      setOpen={setOpen}
      onCallEntry={selectedOnCallEntry}
      updatePaidStatus={updatePaidStatus}
    />
    {renderCalendarOrList()}
    {renderCurrentOnCall()}
    <Button onClick={() => setOpen(true)}>TEST</Button>
  </Box>
);
};
```

Kuva 40. OnCallCalendarScreen.tsx. OnCallHandler-komponentti kalenterinäkymän palautusarvossa

Dialog-komponentti renderöidään kalenterinäkymään ponnahdusikkunana. Ikkunassa on kalenterinäkymässä klikattua päivää vastaavan päivystysviikon tiedot, eli päivystävä henkilö ja palkanmaksun tila. Tilan päivittämistä varmistetaan vielä sisäkkäisellä, pienemmällä Dialog-ikkunalla (kuva 41).

Kuva 41. OnCallHandler-komponentin dialog-ikkuna, jossa päivitetään valitun päivystysviikon palkanmaksun tila

Myöntyvän varmistusvastauksen kohdalla painike kutsuu updatePaidStatus-funktiota, jolla varsinainen POST-pyyntö lähetetään AWS:ään. Parametreina käytetään Dialog-komponenttiin syötettyjä päivystystietoja. (Kuva 42.)

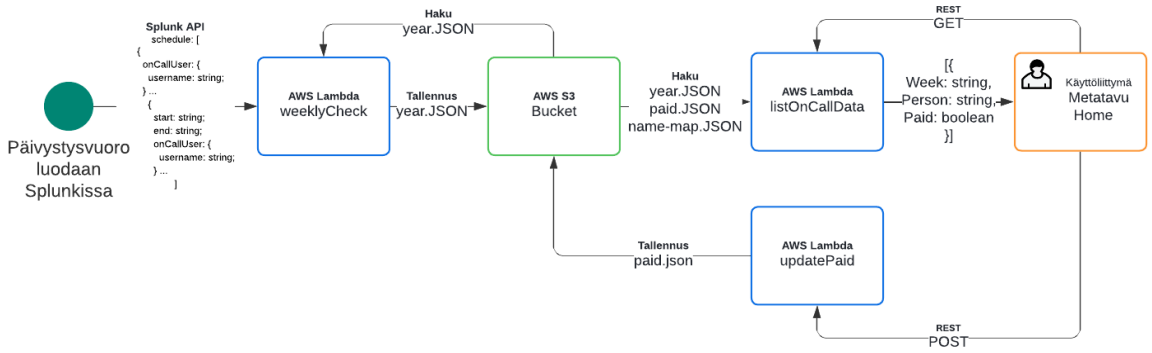
```
const updatePaidStatus = async (entry: OnCallCalendarEntry) => {
  if (entry.date) {
    const weekNumber = DateTime.fromISO(entry.date).weekNumber;
    const year = DateTime.fromISO(entry.date).year;
    const updateParameters: OnCallPaid = {
      paid: !entry.paid,
      year: year,
      week: weekNumber
    };
    await onCallApi.updatePaid({ onCall: updateParameters });
    getOnCallData(year);
  }
};
```

Kuva 42. OnCallCalendarScreen.tsx. updatePaidStatus-funktion määritykset

Koska päivystystiedot ovat nyt päivitetty, funktion viimeisellä rivillä haetaan tiedot uudelleen, joka laukaisee koko kalenterinäkymän uudelleen renderöinnin uusilla tiedoilla. Tässä tapauksessa joko päivitetyn kalenteriviikon päivänumerot muuttuvat vihreiksi maksetun palkan merkiksi tai väritys poistuu palkanmaksua peruttaessa.

Yhteenveto

Nyt, kun kaikki päivystyskalenterin osat on esitelty, havainnollistetaan kuvassa 43 näiden osien välinen toiminta ja keskinäiset suhteet kaaviomuodossa.



Kuva 43. UML-kaavio päivystyskalenterin toiminnan kiertokulusta

Sykli käynnistyy, kun päivystysvuoro luodaan Splunkissa. weeklyCheck -Lambda-funktio tarkistaa automaattisesti viikon välein Splunkin sovellusrajapinnasta seuraavan viikon päivystysvuoron tiedot, muotoilee ja tallentaa nämä kuluvaa vuotta vastaavaan JSON-tiedostoon. Tiedosto tallennetaan AWS S3-bucketiin.

Kun käyttäjä navigoi Metatavu Homen päivystyskalenteriin, lähettää käyttöliittymä näkymän renderöityessä GET-pyynnön AWS API Gateway -URL-URL-osoitteeseen, joka aktivoi listOnCallData-lambda-funktion. Funktio hakee AWS S3 -bucketista päivystystietojen esittämiseen tarvittavat JSON-tiedostot ja lähettää nämä yhtenä JSON-muodossa.

Kun palkanmaksun tilaa päivitetään, lähettää käyttäjä käyttöliittymän kautta POST-pyynnön AWS API Gateway -URL-osoitteeseen, joka aktivoi updatePaid-lambda-funktion. Funktio vertaa pyynnön rungosta saatuja päivystystietoja tämän vuotta vastaavaan JSON-tiedostoon ja tallentaa muutokset, mikäli päivystysviikko löytyy tiedostosta. Lopulta tiedosto tallennetaan AWS S3 -bucketiin.

4 PÄÄTÄNTÖ

Päivystyskalenterin uudelleentoteutus on mielestäni onnistunut käyttötarkoituksensa saavuttamisessa, eli päivystysvuorojen tarkastelun klassisessa ja helposti ymmärrettävässä kalenterinäkymässä. Sovellus on sisällytetty osana Metatavu Homea, jolloin se on helposti käytettävissä samassa alustassa muiden työajan seurantaan tarkoitettujen työkalujen kanssa. Sovelluksen ydintoiminnallisuudet toimivat, kuten projektin alussa on suunniteltu.

Mahdolliset jatkokehitysideat voisivat keskittyä käyttömukavuuden ja saavutettavuuden kasvattamiseen. Ehdotuksia ovat esimerkiksi päivystysvuorojen luontimahdollisuuden siirtäminen osaksi päivystyskalenterin käyttöliittymää erillisen Splunk-sovelluksen sijaan, kalenterinäkymän mukauttaminen käyttäjän tarpeiden mukaiseksi esimerkiksi elementti- ja tekstikokojen tai sijaintien muokkaamisominaisuudella.

Päivystysvuorojen seuranta ja suunnittelu ovat olleet vakiintuneita rutiineja osana Metatavun arkea jo pitkän aikaa, joten näen työkalun tarpeen pitkäikäisenä. Lisäksi päivystyskalenteri on luotu käyttäen alalla standardisoituneita työkaluja ja teknologioita, jotka ovat todennettujen tahojen, kuten suurten yritysten, ylläpitämiä, taaten sovelluksen teknisen eheyden ja tuen pitkäksi aikaa. Uusi päivystyskalenteri hyödyntää paljon valmiiksi tehtyjä osia tämän edellisestä inkarnaatiosta. Etenkin sen palvelinpuolen toteutuksesta, josta oma osuuteni oli siirtää ja muokata tämän koodit uudelle sovelluskehykselle sopivaksi. Tässä yhteydessä koodia olisi voinut hioa tietyistä kohdin hieman perusteellisemmin, esimerkiksi yhdenmukaistamalla lambda-funktioiden tapoja käsitellä niille tulevia pyyntöjä joko kyselymerkkijonoa tai pyynnön runkoa käyttämällä kaikissa määrityksissä.

Olen tyytyväinen lopputulokseen. Koin projektin sopivan haastavaksi ja osaamistani kasvattavaksi. Projektin aikana sain käsitellä itselleni tuttuja teknologioita ja osa-alueita, mutta samalla myös poistua mukavuusalueeltani omaksumalla uusia konsepteja, joista en ollut aikaisemmin tiennyt mitään. Tällaisia olivat esimerkiksi pilvialustojen palvelujen sisällyttäminen sovelluksiin.

LÄHTEET

AWS Credentials s.a. Serverless. WWW-dokumentti. Saatavissa: https://www.serverless.com/framework/docs/providers/aws/guide/credentials [viitattu 1.4.2024]

AWS Lambda s.a. Amazon Web Services. WWW-dokumentti. Saatavissa: https://aws.amazon.com/lambda/ [viitattu 1.4.2024]

Built-in React Hooks s.a. Meta. WWW-dokumentti. Saatavissa: https://react.dev/reference/react/hooks [viitattu 29.3.2024]

Chris, M. 2023. Difference between Library and Framework. LinkedIn. WWW-dokumentti. Saatavissa: https://www.linkedin.com/pulse/difference-between-library-framework-myra-chris/ [viitattu 30.1.2024]

Daigle, K. 2023. Octoverse: The state of open source and rise of AI in 2023. Blogi. Saatavissa: https://github.blog/2023-11-08-the-state-of-open-source-and-ai/ [viitattu 30.1.2024]

Document Object Model s.a. MDN Web Docs. WWW-dokumentti. Päivitetty 17.12.2023. Saatavissa: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model [viitattu 28.12.2023]

Golden, B. 2013. Amazon Web Services For Dummies. New Jersey: John Wiley & Sons, Inc. E-kirja. Saatavissa: https://kaakkuri.finna.fi/Record/nelli29_mamk.2670000000421149?sid=3360668092&imgid=1 [viitattu 20.11.2023]

ingydotnet. 2013. If YAML ain't markup language, what is it? StackOverflow. Keskustelufoorumiviesti. Saatavissa: https://stackoverflow.com/questions/6968366/if-yaml-aint-markup-language-what-is-it [viitattu 1.4.2024]

JavaScript s.a. MDN Web Docs. WWW-dokumentti. Päivitetty 25.9.2023. Saatavissa: https://developer.mozilla.org/en-US/docs/Web/JavaScript [viitattu 28.12.2023]

JavaScript Guide s.a. MDN Web Docs. WWW-dokumentti. Päivitetty 2.5.2023. Saatavissa: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide [viitattu 28.12.2023]

Klems, M. 2018. AWS Lambda Quick Start Guide. Birmingham: Packt Publishing Ltd. E-kirja. Saatavissa: https://kaakkuri.finna.fi/Record/nelli29_mamk.4100000005116208?sid=3623403686 [viitattu 2.1.2024]

MDN Web Docs Glossary: Definitions of Web-related terms s.a. MDN Web Docs. WWW-dokumentti. Päivitetty: 8.6.2023 Saatavissa: https://developer.mozilla.org/en-US/docs/Glossary [viitattu 30.1.2024]

Replicating objects s.a. Amazon Web Services. WWW-dokumentti. Saatavissa: https://docs.aws.amazon.com/AmazonS3/latest/userguide/replication.html [viitattu 1.4.2024]

Serverless Framework Concepts s.a. Serverless. WWW-dokumentti. Saatavissa: https://www.serverless.com/framework/docs/providers/aws/guide/intro [viitattu 20.11.2023]

Serverless Infrastructure Providers s.a. Serverless. WWW-dokumentti. Saatavissa: https://www.serverless.com/framework/docs/providers [viitattu 1.4.2024]

State: A Component's Memory s.a. Meta. WWW-dokumentti. Saatavissa: https://react.dev/reference/react/hooks [viitattu 29.3.2024]

Synchronizing with Effects s.a. Meta. WWW-dokumentti. Saatavissa: https://react.dev/learn/synchronizing-with-effects [viitattu 29.4.2024]

The Basics s.a. TypeScript. WWW-dokumentti. Päivitetty 28.11.2023. Saatavissa: https://www.typescriptlang.org/docs/handbook/2/basic-types.html [viitattu 20.11.2023]

Thorgersen, S. 2020. Keycloak Intro. Youtube. Videoleike. Julkaistu 2.4.2020. Saatavissa: https://www.youtube.com/watch?v=duawSV69LDI [viitattu 1.4.2024]

Type coercion s.a. MDN Web Docs. WWW-dokumentti. Päivitetty: 6.6.2023 Saatavissa: https://developer.mozilla.org/en-US/docs/Glossary/Type_coercion [viitattu 30.1.2024]

TypeError s.a. MDN Web Docs. WWW-dokumentti. Päivitetty: 26.5.2023 Saatavissa: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypeError [viitattu 28.3.2024]

Unary Plus (+) s.a. MDN Web Docs. WWW-dokumentti. Päivitetty: 15.8.2023 Saatavissa: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Unary_plus [viitattu 30.1.2024]

Uploading objects s.a. Amazon Web Services. WWW-dokumentti. Saatavissa: https://docs.aws.amazon.com/AmazonS3/latest/userguide/upload-objects.html [viitattu 1.4.2024]

Virtual DOM and Internals s.a React. WWW-dokumentti. Saatavissa: https://legacy.reactjs.org/docs/faq-internals.html [viitattu 4.4.2024]

W3Techs s.a. Usage statistics of JavaScript as client-side programming language on websites. WWW-dokumentti. Saatavissa: https://w3techs.com/technologies/details/cp-javascript/ [viitattu 22.1.2024]

What is Amazon S3? s.a. Amazon Web Services. WWW-dokumentti. Saatavissa: https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html [viitattu 1.4.2024]

What is AWS Lambda? s.a. Amazon Web Services. WWW-dokumentti. Saatavissa: https://docs.aws.amazon.com/lambda/latest/dg/welcome.html [viitattu 20.11.2023]

Writing Markup with JSX s.a. Meta. WWW-dokumentti. Saatavissa: https://react.dev/reference/react/hooks [viitattu 29.3.2024]