



Oliver Martikainen

Integroitu palvelinten valvontanäkymä

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintäteknikan tutkinto-ohjelma

Insinöörityö

5.5.2021

Tiivistelmä

Tekijä: Oliver Martikainen
Otsikko: Integroitu palvelinten valvontanäkymä
Sivumäärä: 25 sivua
Aika: 5.5.2021

Tutkinto: Insinööri (AMK)
Tutkinto-ohjelma: Tieto- ja viestintätekniikan tutkinto-ohjelma
Ammatillinen pääaine: Ohjelmistotuotanto
Ohjaajat: Lehtori Simo Silander
Tiiminvetäjä Tommi Turunen

Insinööriyön tarkoituksena on luoda yhdistetty valvontanäkymä useasta eri palvelimesta tulevalle palvelinten valvontahälytysdatalle. Näkymän tarkoituksena on tarjota tehokas ja käyttäjäystävällinen tapa hallinnoida siihen kerättyä palvelinten valvontahälytyksiä. Työssä on tarkoitus hyödyntää moderneja web-ohjelmoinnin työkaluja, kuten Node.js ja React.js. Tavoitteena on kokonaisuus, jota on helppo muokata ja ylläpitää, mutta tarjoaa samalla tehokkaan työkalun asiakastiimin käyttöön.

Insinööriyössä käydään läpi projektin arkkitehtuurisuunnitelmat ja suurimmat haasteet. Arkkitehtuuriosiossa käydään läpi eri ratkaisuvaihtoehtoja havaittuihin ongelma-kohtiin sekä eri vaihtoehtojen heikkouksia ja vahvuuksia. Projektin palvelinohjelmisto toteutettiin Node.js-prosessina, joka kerää kaikista palvelinten valvontajärjestelmistä hälytykset, prosessoi ne yhteen listaan ja välittää ne REST-rajapinnan avulla eteenpäin. React.js-pohjainen käyttöliittymä huolehtii palvelinten valvontahälytyksien suodattamisesta ja tarjoaa käyttäjälle näkymän, josta niitä voi hallinnoida.

Lopputuloksena on web-sovellus, joka täyttää projektille asetetut vaatimukset ja saa voimakasta positiivista palautetta asiakastiimin edustajalta. Projektin kehitysehdotuksissa todetaan, että jos palvelinten valvontahälytysmäärät moninkertaistuvat, saattavat tietyt arkkitehtuuriratkaisut vaatia uudelleensuunnittelua.

Avainsanat: JavaScript, Node.js, React.js, Full-Stack, web-ohjelmointi

Abstract

Author: Oliver Martikainen
Title: Integrated Server Monitoring Dashboard
Number of Pages: 25 pages
Date: 5 May 2021

Degree: Bachelor of Engineering
Degree Programme: Information and Communication Technology
Professional Major: Software Engineering
Supervisors: Simo Silander, Senior Lecturer
Tommi Turunen, Manager

The goal of the study was to create a modern full-stack web-application for server monitoring. The application needs to be integrated to several servers that gather server monitoring events, and to gather them into one place, where they can be distributed. For the applications backend, the plan is to use Node.js, and the frontend was to be done with React.js, since both are popular tools with plenty of available developers.

The thesis explains the architectural plan of the project and goes into detail on the most pressing problems of each stage in the plan. Each discussed problem is examined, and potential solutions are presented, together with the strengths and weaknesses of the solution. The main purpose of the backend is to gather all the server monitoring events from the integrated monitoring services and host a REST web interface that can be used to fetch the events. The React.js frontend focuses on filtering the server monitoring events and offering an efficient and responsive dashboard that can be used to manage the events.

The result of the development project is an efficient web-application that fulfils its requirements. In the event of a major increase in server monitoring events some of the architectural decisions made for the web-application may need revision, or the applications user responsiveness might suffer.

Keywords: JavaScript, Node.js, React.js, Full-Stack, web-application

Sisällys

Lyhenteet

1	Johdanto	1
2	Projektin määrittely	2
2.1	Tekniset vaatimukset	2
2.2	Projektiorganisaatio	3
3	Arkkitehtuuri	3
3.1	TrueSight-integraatio	4
3.2	Palvelinohjelmisto	5
3.3	Käyttöliittymä	5
4	Käytetyt työkalut	8
4.1	Versionhallinta	8
4.2	Palvelinohjelmisto	8
4.3	Käyttöliittymä	8
4.4	Kehitysympäristö	9
5	Toteutus	9
5.1	TrueSight-integraatio	9
5.1.1	Autentikointi	9
5.1.2	Valvontahälytyksien nouto	10
5.2	Palvelinohjelmisto	11
5.3	Käyttöliittymä	13
5.3.1	Tietojen nouto	14
5.3.2	Tiedonsiirto ja käsittely	17
5.3.3	Renderöinti	20
6	Tulevaisuuden kehitysehdotukset	22
7	Yhteenveto	23
	Lähteet	25

Lyhenteet

- GZIP: *GNU-zip*. Http-kutsujen yleinen kompressointimetodi. Sallii datan siirtämisen tehokkaammassa muodossa.
- HTTP: *Hypertext Transfer Protocol*. Web-ohjelmoinnin yleisin tiedonsiirto-protokolla.
- JSON: *JavaScript object notation*. Web-ohjelmoinnissa yleisesti käytetty datan välitysmuoto.
- JWT: *JSON Web Token*. Web-ohjelmoinnissa käytetty autentikointimetodi.
- REST: *Representational State Transfer*. Web-ohjelmoinnissa käytetty rajapinta-arkkitehtuurimalli.
- SSE: *Server-Sent-Events*. Web-ohjelmointi tiedonvälitystekniikka, jossa palvelinohjelmisto lähettää käyttöliittymälle viestin, kun uutta tietoa on tarjolla.
- TSIM: *TrueSight Infrastructure Management Server*. Palvelimien valvonta-ohjelmistoja kontrolloiva palvelin.
- TSPRES: *TrueSight Presentation Server*. TSIM-palvelinten hallinnointinäkyvä.

1 Johdanto

Kaikki maailman verkkosovellukset pyörivät jossain päin maailmaa jonkun hallinnoimassa palvelimessa. Ilman verkkosovelluksia pyörittävien palvelinten valvontaa niiden isäntäpalvelimet saattavat kohdata useita erilaisia ongelmaita ja aiheuttaa verkkosovelluksen kaatumisen. Isojen yritysten verkkokaupoille jokainen pois päältä vietetty minuutti voi maksaa miljoonia menetettyinä tuotemyynteinä. Tästä syystä verkkosovelluksia ja muita palveluita pyörittävien palvelinten kriittisten parametrien valvonta on tärkeää. Palvelinten valvonta tuottaa jatkuvalla tahdilla dataa tilanpäivityksien muodossa, ja kun valvonnassa on tuhansia palvelimia, alkaa yhden minuutin aikana kertyä dataa yhä enemmän. Aktiivisesti käytössä olevat palvelimet tuottavat tilanpäivityksen aina prosessorin käyttöasteen muuttuessa levytila tai lokitiedostot saavuttavat maksimikokoaan tai jos yksikin tärkeistä prosesseista sammuu.

Useiden eri asiakkaiden palvelininfrastruktuurien hallinnointi ja valvonta tarkoittavat, että yksinkertaiset ratkaisut eivät aina toimi ja valvonta joudutaan toteuttamaan usean eri järjestelmän avulla. Tuhansien palvelinten valvonta tarkoittaa, että valvontatapahtumia tulee merkittäviä määriä, ja niiden hallinnointi vaatii tehokasta ja riittävän skaalautuvaa hallinnointinäkömää.

Tämän insinööriyön tavoitteena on integroida asiakkaiden palvelimia valvovat järjestelmät niiden rajapintojen kautta yhteen keskitettyyn näkymään, joka pysyy käsittelemään sisään tulevat datamäärät riittävän tehokkaasti, ettei käyttäjäkokemus kärsi. Tekniseltä tasolta projektin tavoitteet käydään läpi tarkemmin projektin määrittelyosuudessa.

Raportissa tarkastellaan myös projektin vaatimusten täyttymisen kannalta olennaisia arkkitehtuurihaasteita, toteutuksessa hyödynnettyjä teknologioita sekä projektin toteutuksessa tehdyt arkkitehtuuripäätökset. Pyrkimyksenä on selkeyttää, miksi eri toteutusvaihtoehtoihin on päädytty.

Lopputuloksena on tuotannossa oleva web-sovellus, joka yhdistää kaikkien eri asiakkuuksien palvelinten valvontahälytykset yhteen valvontanäkymään, joka tarjoaa tehokkaan käyttöliittymän valvontaa suorittaville henkilöille.

2 Projektin määrittely

Projektin asiakkaana toimii palvelinten valvontaa suorittava tiimi, eli kyse on CGI:n sisäisestä asiakkaasta. Tarkoituksena on luoda selaimessa pyörivä näkymä, josta voi hallinnoida palvelinhälytyksiä ja joka korvaisi vanhan sovelluspohjaisen hitaan hallinnointityökalun. Modernin selainpohjaisen hallinnointinäkymän pitäisi nopeuttaa valvontatapahtumien käsittelyä sekä helpottaa merkittävästi työkalun käyttöönottoa ja päivittämistä verrattuna vanhaan sovelluspohjaiseen työkaluun.

2.1 Tekniset vaatimukset

Lopputuloksena syntyvän selainpohjaisen applikaation olisi tarkoitus pyöriä moderneilla ja suosituimmilla selaimilla eli Firefoxilla ja Chromella. Tämä rajaus tarkoittaa, ettei tarvitse käyttää kehitysaikaa vanhempien ja hankalampien selaimien, kuten Internet Explorer 11 -version tukemiseen. Selaimella tarjottavalla käyttöliittymällä on tarkoitus pystyä hallinnoimaan kaikkia TSIM-palvelimilla olevia palvelinten valvontahälytyksiä, ja datan päivitys pitäisi tapahtua noin kerran minuutissa. Käyttöliittymän on tarkoitus tehostaa hälytysten hallinnointia, eli sen tulisi käynnistyä vanhaa työkalua nopeammin ja pystyä etsimään hälytysdatasta haluttujen kriteerien perusteella sopivia hälytyksiä tehokkaasti sekä yleisesti vastaamaan käyttäjän toimintoihin ilman huomattavia viiveitä. Lopputuloksena olevan web-sovelluksen pitää kyetä käsittelemään myös poikkeustilanteissa tapahtuvaa hälytystulvaa, jolloin voi olla kaikilla TSIM-palvelimilla yhteensä maksimissaan 400 000 samanaikaista hälytystä, kun keskimäärin niiden kokonaismäärä on noin 50 000 kappaletta.

Vaatimusten täyttäminen vaatii siis, että projektin web-sovellus tarjoaa modernin selainkäyttöliittymän, jossa on isompien datamäärien hallintaa mietitty, sekä palvelinohjelman, joka on integroitu kaikkiin TSIM-palvelimiin ja kerää niistä minuutin välein palvelinhälytystietoja.

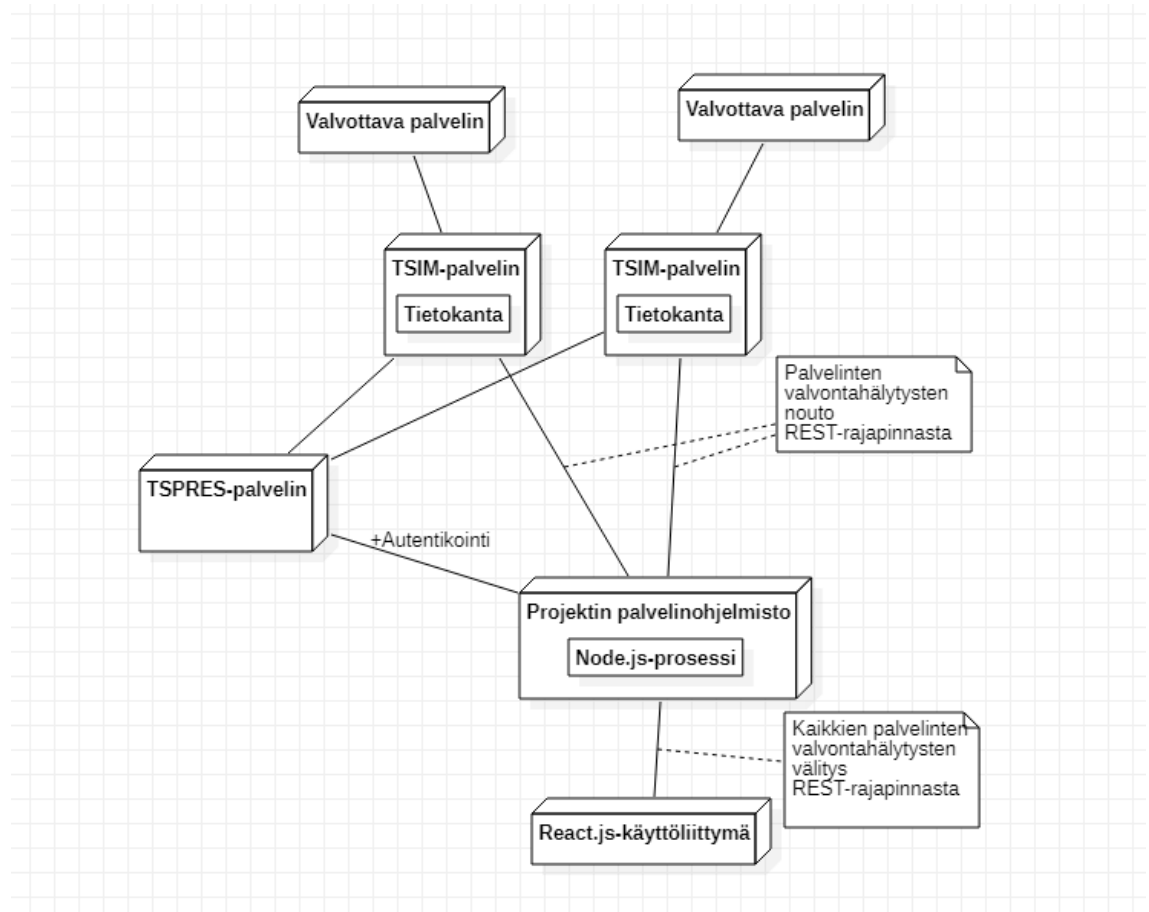
2.2 Projektiorganisaatio

Projektin ohjelmistokehittämisestä vastaa yksi ohjelmoija, jonka tukena toimii kaksi palvelinten valvontajärjestelmien asiantuntijaa sekä käyttöliittymäsuunnittelussa auttava asiakastiimin edustaja. Ohjelmoinnin organisoinnissa käytettiin Agile-inspiroitua toimintamallia eli ohjelma paloiteltiin pienempiin kokonaisuuksiin. Aina uuden palan valmistuttua asiakastiimin edustajalle esitellään lopputulos ja saadun palautteen perusteella tehdään muutoksia kyseiseen osaan sekä olemassa oleviin suunnitelmiin.

3 Arkkitehtuuri

Projekti on rakenteeltaan kolmiosainen, jonka pohjana toimivat TrueSight-palvelinvalvontaan tarkoitetut palvelimet. TrueSight on valvontajärjestelmien tuoteperhe, johon kuuluu TrueSight Infrastructure Management Servers- (TSIM) sekä TrueSight Presentation Server (TSPRES) -tuotteet. TSIM-palvelimet keräävät valvonnassa olevien palvelinten valvontahälytyksiä, ja TSPRES-palvelimet tarjoavat TSIM-palvelinten hallinnointisovelluksen. TrueSight-palvelimet tarjoavat REST-rajapinnan, jonka avulla palvelinten valvontahälytykset haetaan. TrueSight-palvelimet toimivat käytännössä hälytysten tietokantana ja projektin oma palvelinohjelmiston on tarkoitus toimia tietokantoja yhdistävänä keskusyksikkönä, joka välittää niitä eteenpäin käyttöliittymälle. Projektin palvelinohjelmisto pyörii sille tarkoitetulla paikallisella Windows-palvelimella Node.js-ohjelman avulla. Palvelinohjelmistoon yhdistetty käyttöliittymän tärkeimpänä tehtä-

vänä on huolehtia kerätyn palvelinten valvontahälytysten esittämisestä käyttäjälle selkeässä ja tehokkaasti käsiteltävässä muodossa. Kuva 1 näyttää suunnitellun infrastruktuurin eri osat.



Kuva 1. Arkkitehtuurisuunnitelman lopputuloksena oleva infrastruktuurikaavio.

3.1 TrueSight-integraatio

TrueSight-palvelimet toimivat projektin runkona, TSIM-palvelimet toimivat tietokantana ja TSPRES-rajapinta autentikointitapana. Kyseisten järjestelmien integroiminen sovellukseen onnistuu niiden tarjoamien REST-rajapintojen kautta.

TSIM-palvelinten REST-rajapintaan on tarkoitus minuutin välein tehdä http-kutsuja, joilla haetaan uudet palvelinvalvontahälytykset. Sitä ennen palvelinohjel-

miston tarvitsee hakea autentikointikoodi TSPRES-palvelimen REST-rajapinnasta http-kutsulla. TSPRES-palvelimen antama autentikointikoodi on voimassa 24 tuntia, eli se pitää päivittää ainakin kerran päivässä.

3.2 Palvelinohjelmisto

Sovelluksen oman palvelinohjelman tärkeimpänä tehtävänä on kerätä palvelinvalvontahälytykset TSIM-palvelimilta, käsitellä ne haluttuun muotoon ja välittää ne käyttöliittymälle.

Palvelinvalvontahälytyksien noutaminen on suhteellisen suoraviivainen tehtävä. Palvelin voi suorittaa minuutin välein toistuvaa noutoprosessia, jossa se lähettää http-kutsun uusista palvelinten valvontatapahtumista kaikkiin TSIM-palvelinten rajapintoihin ja kerätä ne yhteen.

Ongelmaksi muodostuu kerätyn datan tallennus palvelimen uudelleenkäynnistystä ajatellen. Tarkoituksena on hyödyntää TSIM-palvelinten tietokantoja ja minimoida sovelluksen oman palvelimen tekemää työtä, eli palvelinten valvontahälytykset voidaan säilyttää suoraan palvelinohjelmiston muistissa. Tämä nopeuttaa merkittävästi datan käsittelyä ja jakamista, mutta jos palvelin tai palvelinohjelmisto käynnistetään uudelleen, kaikki kerätty data katoaa ja pitää hakea TSIM-palvelimilta uudestaan. Tuotantopalvelimen uudelleenkäynnistys tapahtuu kuitenkin vain muutaman kuukauden välein päivityksiä varten, mutta kehitysvaiheessa palvelinohjelmisto saatetaan uudelleen käynnistää useita kertoja tunnissa.

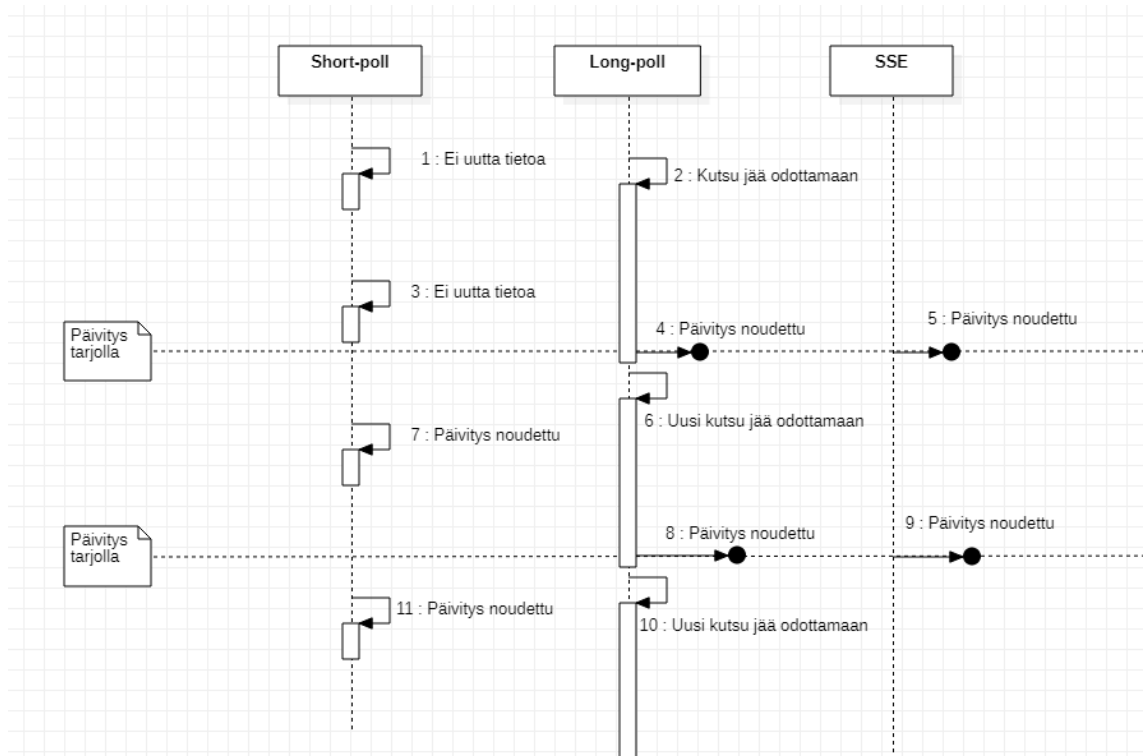
3.3 Käyttöliittymä

Käyttöliittymän tehtävänä on tarjota selkeä ja käyttäjäystävällinen näkymä, jossa palvelinvalvontahälytyksiä voidaan käsitellä. Käyttöliittymän ohjelmoinnissa pitää tehdä päätös, millä tavalla päivitettyjä tietoja noudetaan, talletetaan ja renderöidään. Datapäivityksien noutamiseen on muutama yleinen tapa: short-poll, long-poll sekä SSE eli Server-Sent Events.

Short-poll tarkoittaa käyttöliittymältä palvelimelle tasaisin väliajoin lähetettäviä kyselyitä. Kyseinen tapa on yksinkertainen, mutta voi aiheuttaa viiveitä päivityksissä, jos palvelimen tiedot päivittyvät juuri sille tehdyn kyselyn jälkeen.

Long-poll-menetelmässä [1] käyttöliittymä lähettää palvelimelle kyselyn, joka jätetään odottamaan palvelimen vastausta, eli kunnes palvelimen tiedot päivittyvät ja palvelin antaa odottavalle kyselylle päivitettyt tiedot. Kyselyn palautumisen jälkeen lähetetään uusi kysely odottamaan vastausta. Näin vältetään viiveet päivityksissä, mutta kyseinen menetelmä vie enemmän resursseja palvelimelta.

SSE-menetelmässä [2] palvelimen ja käyttöliittymän välillä ylläpidetään jatkuvaa yhteyttä. Heti kun palvelimen omat tiedot päivittyvät se välittää kyseisen yhteyden kautta tiedon tästä käyttöliittymälle, joka voi hakea päivitykset omaan käyttöön ilman viivettä. Kaikista tavoista tämä on teknisesti haastavin, mutta poistaa kaikki viiveet ja on moderneilla teknologioilla hyvinkin tehokas resurssien osalta. Kuva 2 näyttää kolmen aikaisemmin esitellyn tekniikan viiveet.



Kuva 2. Datapäivystekniikoiden viiveet. Short-poll-metodilla käyttäliittymä lähettää palvelimelle tasaisin väliajoin kyselyitä, ja jos palvelimen tietojen päivitystahtia ei ole synkronoitu käyttäliittymän poll-intervallien kanssa, voi käyttäliittymän tietojen päivittymisessä tulla ylimääräisiä viiveitä. Long-poll-kyselyissä käyttäliittymä lähettää palvelimelle kyselyn, joka jää odottamaan vastausta, eli seuraavaa tietopäivitystä. SSE-metodissa palvelimen ja käyttäliittymän välillä on koko ajan avonainen yhteys, jota pitkin lähetetään tietopäivitykset heti, kun ne ovat tarjolla.[3]

Käyttäliittymän datankäsittelyyn on muutamia erilaisia yleisiä tapoja. Yksinkertaisin tapa on lähettää kaikki data käyttäliittymälle istunnon alussa ja päivittää sitä tarpeen mukaan. Näin saadaan vähennettyä palvelimelle aiheutuvaa datankäsittely kuormitusta ja hyödynnettyä käyttäjien omia resursseja, mutta käyttäliittymän käytettävyys ja vasteajat voivat kärsiä, jos dataa on liikaa tai internetyhteys on hidas. Toinen tapa on hakea vain renderointiin tarvittava data ja antaa palvelimen huolehtia datan käsittelystä, mutta se voi merkittävästi lisätä palvelimelta vaadittuja resursseja käyttäjämäärien kasvaessa.

4 Käytetyt työkalut

4.1 Versionhallinta

Projektissa käytetään Git-versionhallintatyökalua ja GitHub-desktop-käyttöliittymää sille. GitHub-desktop tarjoaa kätevän tavan hallinnoida eri Git-haaroja ja projektin versioita sekä näkymän, josta voi helposti tarkistaa koodiin tehdyt muutokset.

4.2 Palvelinohjelmisto

Projektin palvelinohjelmistona hyödynnetään Node.js-ohjelmistoa, joka on avoimen lähdekoodin palvelinpuolen JavaScript-ajoympäristö. Näin voidaan käyttää palvelinohjelmiston ja käyttöliittymän ohjelmoinnissa samaa ohjelmointikieltä ja ympäristöä, minkä pitäisi nopeuttaa kehitystyötä. Lisäksi Node.js:n yhteydessä voidaan hyödyntää valmiita ohjelmamoduuleita, jotka tarjoavat valmiita kokonaisuuksia ohjelmistoon. Tärkeimpinä esimerkkeinä on Express.js, joka on suosittu REST-rajapintojen luontiin tarkoitettu moduuli ja Dotenv, joka sallii Node.js-konfigurointitiedostojen käyttämisen ja siten helpottaa kehitys- ja tuotantoympäristön välillä siirtymistä.

Tuotantoympäristössä palvelinohjelmiston pyörittämiseen halutaan tekniseltä tasolta yksinkertainen ratkaisu, joka käynnistyy automaattisesti palvelimen kanssa. NSSM (Not-Sucking-Service-Manager) tarjoaa tähän hyvän ratkaisun, joka muuntaa Node.js-prosessin Windows Serviceksi.

4.3 Käyttöliittymä

Käyttöliittymän luontiin hyödynnetään suosittua avoimen lähdekoodin React.js-web-sovelluskehystä, joka helpottaa monipuolisempien selainkäyttöliittymien ohjelmointia. React.js-sovellusten ohjelmointi tapahtuu pääsääntöisesti Ja-

vaScript-ohjelmointikielellä. Lisäksi hyödynnämme React.js:lle tarkoitettua Material.ui-käyttöliittymäkomponenttikirjastoa visuaalisten toimintojen kehityksessä, sillä se tarjoaa valmiita käyttäjäystävällisiä ratkaisuja.

4.4 Kehitysympäristö

Visual Studio Code on ohjelmointiin tarkoitettu avoimen lähdekoodin tekstieditori, joka on yleisesti käytetty JavaScript-ohjelmoinnissa. Se tarjoaa selkeän käyttöliittymän ja useita käteviä lisäosia, joilla voi tehostaa kehitystyötä. Tärkeimpänä näistä on ESLint, joka toimii JavaScript-projekteissa koodistandardien tarkistajana sekä helpottaa syntaksivirheiden tunnistamista.

5 Toteutus

Tämän luvun tarkoituksena on käydä läpi arkkitehtuuriluvussa läpikäytyjä ongelmakohtia ja selittää, miten ne ratkaistiin. Tarkoituksena on tarkastella syvällisemmin projektin ohjelman kannalta kriittisimpiä osuuksia ja perustella toteutuksissa tehtyjä päätöksiä.

5.1 TrueSight-integraatio

TrueSight-integraation tärkeimmät tehtävät ovat huolehtia TSIM-palvelinten autentikoinnista TSPRES-palvelimen tarjoaman REST-rajapinnan kautta sekä TSIM-palvelinten REST-rajapinnan hyödyntäminen palvelinten valvontahälytyksien keräämisessä.

5.1.1 Autentikointi

Palvelimen valvontahälytyksien hakeminen TSIM REST -rajapinnoista vaatii TSPRES-palvelimelta noudetun autentikointikoodin, eli ensimmäisenä pitää tehdä autentikointi-http-kutsu. Esimerkkikoodi 1 näyttää, kuinka TSPRES REST

-rajapinnasta noudetaan autentikointikoodi. Haku toteutetaan yksinkertaisella mutta tehokkaalla logiikalla, jolla ajetaan päivitysfunktio toistuvasti halutuun aika-
välein.

```
const refreshToken = async () => {
  //fetch new authentication token from TSPRES server
  const response = await axios.post(LOGIN_URI,
    {
      'username': username,
      'password': password,
      'tenantName': TRUESIGHT_TENANT
    }
  )
  setToken(data.response.authToken)
}

//JavaScript natiivi funktiolla voidaan ajaa koodi 24h välin
setInterval(() => {
  refreshToken()
}, 24 * 60 * 60 * 1000) //Repeat every 24h
```

Esimerkkikoodi 1. Autentikointikoodin päivitys tekemällä http-kutsu TSPRES REST -rajapintaan.

Palvelinohjelmiston käynnistyksen yhteydessä ja 24 tunnin välein toteutettavan päivityksen lisäksi palvelinohjelmisto seuraa TSIM-rajapintaan lähetettyjen http-kutsujen vastauskoodeja siltä varalta, että jokin niistä palauttaa 401-tilannekoodin. Tämä tarkoittaa, että jokin palvelin ei hyväksynyt käytettyä autentikointikoodia ja palvelinohjelmisto hakee TSPRES-rajapinnan kautta uuden koodin.

5.1.2 Valvontahälytyksien nouto

Palvelimen valvontahälytysten hakeminen TSIM-palvelimilta vaatii sen aikavälin määrittämistä, jonka välein hälytykset halutaan. TSIM-palvelimet käyttävät UNIX-aikaa, joka alkaa nollasta ja mittaa kuluneiden sekuntien määrää sitten 1. tammikuuta 1970 [4]. Palvelinohjelmiston käynnistyksen yhteydessä haluamme hakea kaikki hälytyksen 0-hetken ja http-kyselyn muodostamisen välissä, eli kaikki hälytykset, jotka palvelimella on olemassa sillä hetkellä. Ylimääräisen datansiirron välttämiseksi seuraavat haut tehtäisiin edellisen http-kyselyn ja tulevan kyselyn väliselle ajalle, eli haettaisiin vain uusia hälytyksiä.

Esimerkkikoodi 2 näyttää, miten yksittäiseltä TSIM-palvelimen REST-rajapinnasta voidaan http-kutsulla hakea halutun aikavälin palvelimen valvontahälytykset. Kyselyyn pitää lisätä TSPRES-palvelimelta saatu autentikointikoodi.

```
const fetchEvents = async (startTime, endTime, serverURI) => {
  const TOKEN = getToken()
  //Authorization token
  const requestConfig = {
    'headers': { 'Authorization': `authtoken ${TOKEN}` },
  }
  //hälytysten hakuparametrit
  const searchParameters = {
    'groupingOperator': 'AND',
    'leftExpression': {
      'value': startTime, //epochSeconds
      'identifier': dateAttribute,
      'operator': 'GREATER'
    },
    'rightExpression': {
      'value': endTime, //epochSeconds
      'identifier': dateAttribute,
      'operator': 'SMALLER_OR_EQUALS'
    }
  }
  const response = await axios.post(
    serverURI,
    searchParameters,
    requestConfig
  )
  return response
}
```

Esimerkkikoodi 2. Palvelimen valvontahälytysten parametrit, joiden avulla tehdään http-kysely TSIM REST -rajapintaan.

5.2 Palvelinohjelmisto

Palvelinohjelmisto on Node.js-prosessi, joka on NSSM-ohjelman avulla muutettu Windows Service -muotoon. Näin saamme ohjelmiston automaattisesti käynnistymään aina isäntäpalvelimen uudelleenkäynnistyksen yhteydessä ja sisäisesti käytössä olevat etähallintatyökalut pystyvät hallinnoimaan Servicen tilaa.

Tärkeimpänä tehtävänä palvelinohjelmistolla on integroida TrueSight-rajapinnat, kerätä niiden palvelinten valvontahälytykset yhteen ja tarjota ne käyttöliittymälle sopivassa muodossa.

TSIM-palvelimien hälytystietojen kerääminen vaatii http-kyselyn lähettämistä kaikkiin valvontatietoja kerääviin TSIM-palvelimiin sekä niiden prosessointia käyttöliittymälle lähetettävään yhtenäiseen listaan. Palvelinohjelmistolla ylläpidetään manuaalisesti listaa kaikista käytössä olevista TSIM-palvelimista ja niiden REST-rajapintojen osoitteista. Ohjelmiston käynnistyessä haetaan kaikki TSIM-palvelimilla olevat hälytykset, jonka jälkeen talletetaan suoritettun kyselyn ajankohta. Talletettua ajankohtaa hyödynnetään seuraavassa päivityskyselyssä, jotta vältetään hakemasta uudelleen aikaisemmin vastaanotettuja tietoja ja vähennetään ylimääräisen prosessointia.

Esimerkkikoodi 3 näyttää, miten saamme JavaScriptin natiivi-Date()-objektin avulla laskettua kyselyn muodostamisen ajan UNIX-aikamuodossa. Async-await-rakenteen ja Promise.all()-toimintojen avulla pystymme lähettämään kaikki kyselyt käytännössä samaan aikaan, mutta jatkamaan koodin prosessointia vasta kaikkien kyselyiden vastauksien saavuttua. JavaScriptin setInterval()-funktio toimii yksinkertaisena tapana toistaa palvelinten valvontahälytyspäivitykset aina minuutin välein.

```
const fetchAllServers = async (tsimServersList) => {
  //kyselyn UNIX-aika, JavaScript natiivi arvot millisekunneissa
  const endTime = Math.round(new Date().valueOf() / 1000)

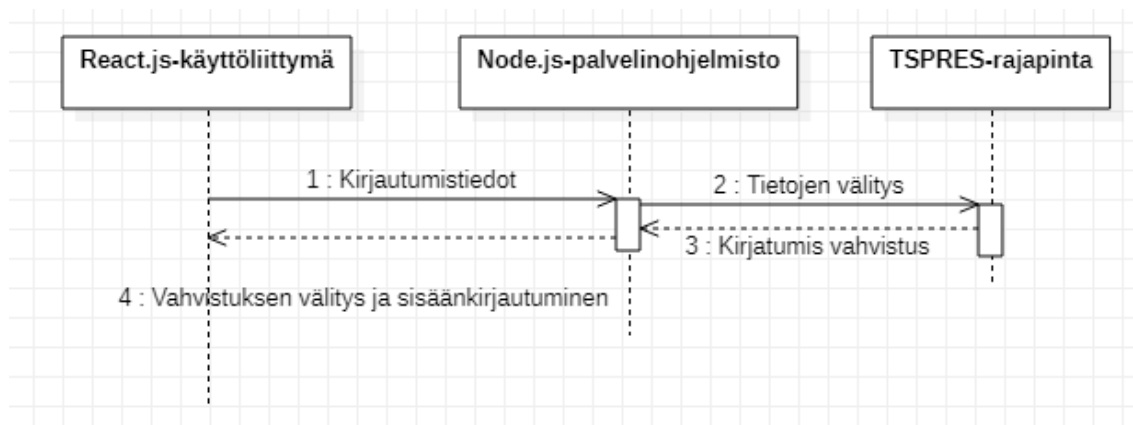
  //lähetetään kaikki kyselyt ja kerätään niiden Promise listaan
  const promiseList = tsimServersList.map(server => {
    const { serverIp, nextStartTime } = server
    return fetchEvents(nextStartTime, endTime, serverIp)
  })
  //odotetaan kaikki kyselyt valmistuu
  const responseList = await Promise.all(promiseList)
  //palautetaan kaikkien kyselyiden vastaukset listana
  return responseList
}

setInterval(async () => {
  const serverUpdatesList = await fetchAllServers(TSIM_SERVER_LIST)
  //päivitetään seuraavan haun aloitusajat
  //ja prosessoidaan vastaanotetut palvelin valvontahälytykset
  processServerUpdates(serverUpdatesList)
}, 60 * 1000) //toistetaan minuutin välein
```

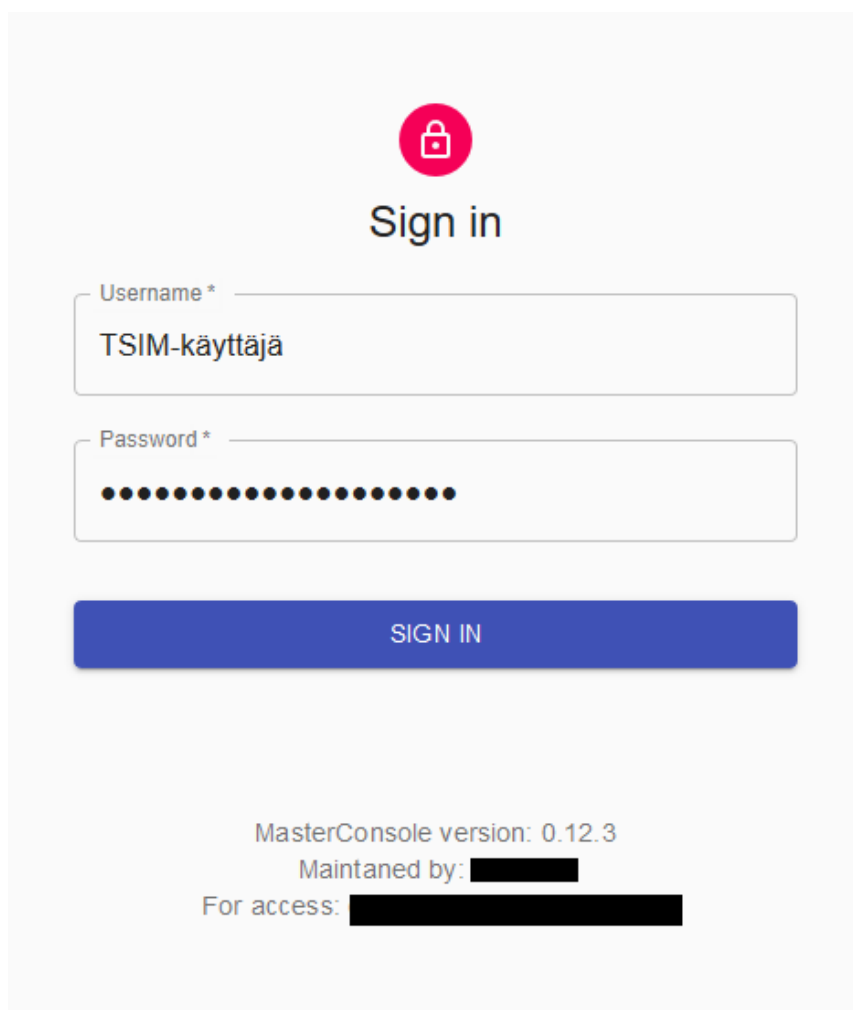
Esimerkkikoodi 3. Kaikkien TSIM-palvelinten valvontahälytysten jatkuva päivittäminen.

5.3 Käyttöliittymä

Selaimessa pyörivä käyttöliittymä ohjelmoidaan React.js-ohjelmointikehyksessä, joka helpottaa monipuolisten websovellusten kehittämistä. Käyttöliittymän komponenttien ulkoasun luonnissa hyödynnämme Material.ui-kirjaston valmiita komponentteja, jotka ovat ulkonäöltään myös tyylikkäitä. Kirjautumiseen voimme hyödyntää muokattua Material.ui:n tarjoamaa avoimen lähdekoodin mallisivua, josta lähetetään tiedot palvelinohjelmistolle. Kuva 3 näyttää käyttäjän kirjautumiseen käytettyä prosessia. Kirjautumistiedot vahvistetaan TSPRES-palvelimen käyttäjätietokannasta sen tarjoaman REST-rajapinnan kautta. Vastauksena ohjelmistopalvelin lähettää takaisin JWT-koodin, jolla käyttöliittymä autentikoidaan sen ja ohjelmistopalvelimen välisessä kommunikaatiossa. Kuva 4 näyttää Material.ui-komponenteista rakennetun kirjautumissivun.



Kuva 3. Kirjautumisen autentikointiprosessi.



The image shows a sign-in form with a red lock icon at the top. Below the icon is the text "Sign in". There are two input fields: "Username*" containing "TSIM-käyttäjä" and "Password*" with masked characters. A blue "SIGN IN" button is below the fields. At the bottom, there is version information: "MasterConsole version: 0.12.3", "Maintaned by: [redacted]", and "For access: [redacted]".

Kuva 4. Material.ui-komponenteista rakennettu kirjautumissivu.

5.3.1 Tietojen nouto

Arkkitehtuuri-luvussa käsiteltiin eri tietojen noudon toteutusvaihtoehtoja, joista päädyimme käyttämään SSE- eli Server-Sent-Events-mallia. Short-poll- ja long-poll-malleissa palvelinohjelmiston rajapinnan tarvitsee kyetä vastaamaan tietonoutokyselyihin, ja käyttöliittymä huolehtii kyselyiden ajoittamiseen liittyvästä logiikasta. SSE-mallissa palvelinohjelmisto joutuu toteuttamaan logiikan, jolla käyttöliittymälle ilmoitetaan, kun päivitettyä tietoa on tarjolla.

Node.js:n natiiviratkaisut SSE-toteutukseen vaativat jonkin verran työstämistä, mutta Express.js-moduuli, jota hyödynnämme REST-rajapinnan luomisessa,

antaa meille helpon tavan toteuttaa SSE-ratkaisu. Esimerkkikoodi 4 näyttää Express.js-ratkaisun SSE-toteutukseen, jonka avulla palvelinohjelmisto voi lähettää haluamansa viestin aktiivisille käyttäliittymille aina, kun se on saanut päivityksiä TSIM-palvelinten valvontahälytyksiin.

```
eventRouter.get('/eventSubscription', async (request, response) => {
  response.status(200).set({
    'connection': 'keep-alive', //pitää yhteyttä jatkuvasti elossa
    'cache-control': 'no-cache',
    'content-Type': 'text/event-stream' //kertoo käyttäliittymälle
    että yhteys on jatkuvaa tekstijono
  })

  //lähetetään kyseinen viesti aina kun SSE tapahtuma triggeröidään
  const eventUpdateListener = (time) => response.write(`data: 'EVENT
  UPDATES AVAILABLE, ${time}'\n\n`)

  response.app.on(EVENT_EMITTER_NAME, eventUpdateListener)

  //kun käyttäliittymä sulkee yhteyden poistetaan se tapahtuman
  kuuntelijoiden listasta
  request.on('close', () => {
    response.app.removeListener(EVENT_EMITTER_NAME, eventUpdateLis-
    tener)
  })
})
```

Esimerkkikoodi 4. Palvelinohjelmiston SSE-toteutuslogiikka Express.js-moduulin toimintoja hyödyntäen.

Käyttäliittymän puolella pystymme hyödyntämään selainten JavaScript-natiivitoimintaa EventSource [5], joka on tuettu kaikissa moderneissa selaimissa. Esimerkkikoodi 5 implementoi SSE-viestien kuuntelulogiikan, joka noutaa päivitykset palvelinten valvontahälytysdataan. Toteutuksen lopputuloksena on viiveetön päivitys, joka pitäisi tapahtua yhtä aikaa kaikkien aktiivisten käyttäliittymien kanssa. Lisäksi toteutus on suhteellisen selkeä ja helppo ylläpitää potentiaalisia muutoksia ajatellen.

```

const eventSubscription = (eventsDispatch) => {
  const eventSub = new EventSource(`/api/events/eventSubscription`)

  eventSub.onopen = () => {
    console.log('EVENT SUBSCRIPTION ACTIVE')
  }
  //jos yhteys sulkeutuu jostain syystä
  eventSub.onerror = () => {
    console.log('EVENT SUBSCRIPTION CONNECTION LOST')
    eventSub.close()
    //yritetään yhdistää uudelleen 30 sekunnin välein
    setTimeout(() => eventSubscription(eventsDispatch), (30 *
1000))
  }
  //kun backend viestittää päivityksistä lähetetään päivityksien
noutu http-kutsu
  eventSub.onmessage = async () => {
    const eventUpdateData = await eventService.getEventUpdates()
    eventsDispatch({ type: 'EVENT_UPDATE', newEvents: even-
tUpdateData.eventList })
  }

  return eventSub
}

```

Esimerkkikoodi 5. SSE-viestien seuraaminen käyttöliittymässä EventSource-natiivitoiminnalla. Palvelinohjelmiston SSE-viestin saapuessa eventsub.onmessage()-funktio ajetaan ja käyttöliittymä lähettää http-kutsun, jolla noudetaan uudet valvontatapahtumat.

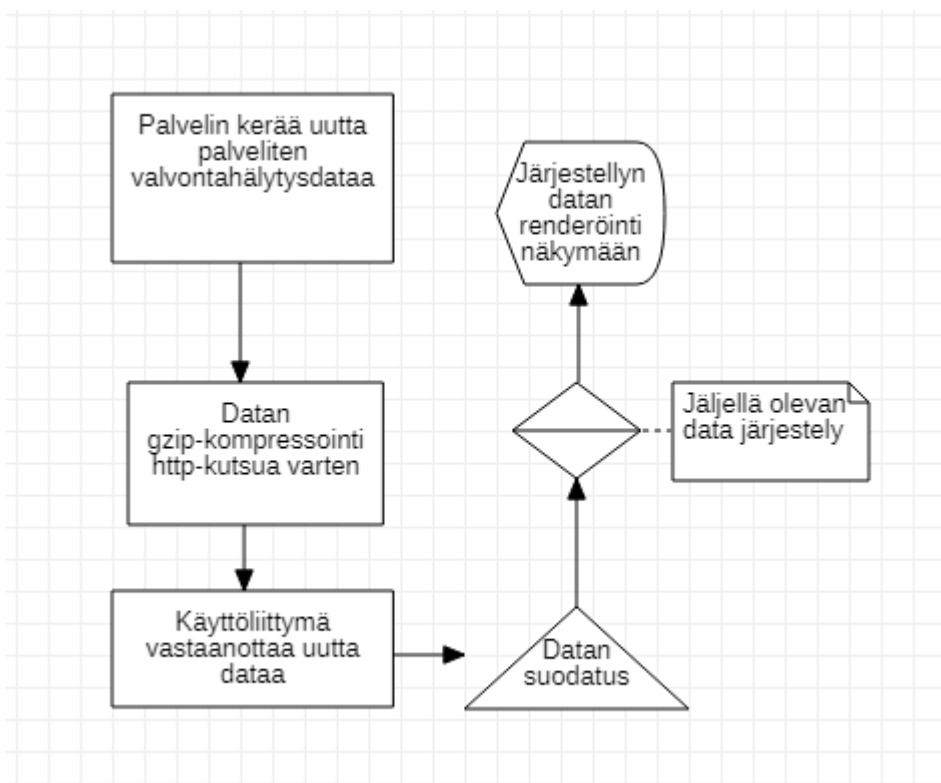
Eli nykyisessä toteutuksessa palvelinohjelmisto ilmoittaa käyttöliittymille, milloin on päivityksiä palvelinten valvontahälytysdataan tarjolla. Viestin saatuaan käyttöliittymät lähettävät välittömästi http-kutsun, jolla ne noutavat päivitysdatan. SSE-tekniikalla voisi teoriassa lähettää suoraan päivitystiedot käyttöliittymille, mutta kyseinen tekniikka ei salli salattuja https-kutsuja ilman monimutkaisia muutoksia eikä tietojen kompressoitua gzip-tekniikalla. Toteutettua mallia voisi parantaa hyödyntämällä esimerkiksi WebSocket-tekniikkaa, joka sallisi palvelinohjelmiston lähettää päivitysdatan suoraan käyttöliittymille https-kutsuissa sekä gzip-kompressoitua käyttäen [6]. Tämä lisäisi merkittävästi ohjelman monimutkaisuutta ilman, että saavutettaisiin mitään huomattavia parannuksia ohjelman suorituskyvyssä, eli tässä vaiheessa tyydymme yksinkertaisempaan, mutta hieman epätehokkaampaan malliin.

5.3.2 Tiedonsiirto ja käsittely

Web-ohjelmoinnissa on tärkeää löytää tasapaino laskentaprosessien tekemisestä käyttöliittymässä ja palvelinohjelmistossa. Mitä enemmän laskettavaa siirtää käyttöliittymälle, sitä enemmän voi säästää palvelinohjelmiston tehoissa, mutta käyttäjävasteajat voivat kärsiä. Projektin käyttöliittymän kannalta eniten laskentatehoa vie palvelinten valvontahälytysten suodattaminen, järjesteleminen ja renderöinti. Tästä syystä käyttöliittymälle lähetettyjen hälytysten määrä on tärkein kriteeri sillä tehtyjen laskentatoimien arvioinnissa.

Kaikilta TSIM-palvelimilta pitäisi yhteensä olla keskimäärin noin 50 000 hälytystä talletettuna, mutta poikkeuksellisissa tilanteissa TSIM-palvelimille voi jäädä merkittävästi enemmän hälytyksiä talteen. Kyseisissä poikkeustilanteissa talletettuja hälytyksiä voi olla maksimissaan 400 000 kappaletta. Eli jos halutaan, että käyttöliittymä vasteajat ja mukavuus ovat määritysten mukaisia, pitää sen pystyä käsittelemään 400 000 talletettua hälytystä.

Yksinkertaisin toteutus on antaa käyttöliittymän huolehtia tietojen prosessoinnista, eli ensin sille siirretään data, se suodattaa sen suodatuskriteereiden läpi, jonka jälkeen jäljelle jäävä data järjestellään esimerkiksi päivämäärän mukaan. Viimeisenä siitä muodostettaisiin renderöintiin tarkoitettuja komponentteja. Kyseisen mallin toteuttaminen vaatii kuitenkin riittävien tehojen varmistamisen selaimilta. Kuva 5 havainnollistaa käyttöliittymän kannalta olennaiset vaiheet palvelinten valvontahälytysdatan prosessoinnista.



Kuva 5. Yksinkertaisen datakäsittelyprosessin toimintajärjestys.

400 000 talletetun hälytyksen maksimimäärä tarkoittaa noin 100 MB dataa JSON-muodossa. Kyseisen tietomäärän siirtäminen asiakastiimin internetnopeuksilla menisi noin 20 sekuntia, joka aiheuttaisi liian suuren odotuksen. Data voidaan merkittävästi pienentää gzip-kompressoinnilla noin 5,5 MB:n kokoon ja lyhentää tiedonsiirtoon menevää aikaa 1–2 sekuntiin, joka on täysin hyväksyttävä sisäiseen käyttöön tarkoitetussa työkalussa. Eli käyttöliittymän avautuessa voidaan lähettää kaikki ohjelmistosovelluksessa olevat hälytykset yhdessä http-kutsussa, jossa kestää noin 1–2 sekuntia. Sen jälkeen kaikki päivitykset voidaan siirtää omissa pienemmissä erissä. Minuutin välein tehdyt päivitykset valvontahälytyksiin ovat keskimäärin noin 50 hälytyksen kokoisia ja gzip-kompressoinnilla noin 1 KB:n kokoisia, eli siirtyvät alle 100 millisekunnissa.

Seuraavana tärkeänä kriteerinä on käyttöliittymän kyky suodattaa ja järjestellä 400 000 hälytyksen tietomassa valittujen kriteereiden mukaan. Hälytyksien suodattimia voi rakentaa useista eri hälytysten attribuuteista ja rajattomasta mää-

rästä eri kriteeriyhdistelmiä, mutta käytännössä kaikki käyttötapaukset hyödyntävät maksimissaan kolmea eri suodatuskriteeriä. Kyseiset kriteerit suodattavat pois noin 95 % kaikista hälytyksistä ja vie noin 200 millisekuntia käsitellä 400 000 hälytyksen massa, joka on käyttöliittymän avauksen yhteydessä hyväksyttävä viive.

Valvontahälytysten järjesteleminen voidaan yksinkertaisimmillaan toteuttaa JavaScript'in natiivilla `Array.sort()`-metodilla [7], jonka algoritmi on selaimen omasta toteutustavasta riippuvainen, mutta voimme tarkastella sitä yleisimmän V8 JavaScript -ajoympäristön toteutuksella. V8:n `sort`-funktio käyttää TimSort-algoritmia [8], jonka asymptoottinen suoritus aika (kaava 1) [9] kasvaa merkittävästi järjestettävän listan kasvaessa.

$$O(n \log n) \quad (1)$$

Lähes kaikki käyttötapaukset järjestelevät tiedot päivämäärän mukaan ja 400 000 satunnaisesti sekoitetun hälytyksen listan järjestäminen kestää noin 2 sekuntia. Varmistamalla, että hälytykset on jo valmiiksi järjestelty päivämäärän mukaan palvelinohjelmistossa säästää meiltä tuon 2 sekunnin järjestelytarpeen käyttöliittymän avauksen yhteydessä. Yleisesti ottaen ennen järjestelyä suodatetaan pois 95 % hälytyksistä, jolloin jäljelle jää noin 2 500–20 000 järjestettävää hälytystä, jotka voidaan käsitellä alle 50 millisekunnissa.

Valvontahälytyksiin tulevien päivityksien kohdalla voimme välttää kaikkien 400 000 hälytyksen uudelleen suodattamista käsittelemällä ensin ainoastaan uudet hälytykset, jotka ovat keskimäärin 50 hälytyksen kokoisia listoja. Eli ensin tarkistamme, mitkä niistä pääsevät suodatuskriteereistä läpi ja järjestelemme ne. Molemmissa menee yhteensä alle 10 millisekuntia, jos käsiteltävänä on alle 100 hälytystä. Tämän jälkeen lisäämme ne aikaisemmin suodatettuun ja järjestellyn hälytyslistan perään, jonka jälkeen teemme lopullisen järjestelyn listalle. Tilanteessa, jossa on 400 000 valmiiksi järjesteltyä hälytystä ja niiden perään lisätään 50 keskenään järjesteltyä hälytystä, 'sort'-algoritmi pystyy suoriutumaan merkittävästi nopeammin tehtävästään, sillä sen tarvitsee löytää oikea paikka

vain uusille tapahtumille. Tässä tilanteessa aikaa kuluu keskimäärin 50 millisekuntia, ja jos suodatuksen jälkeen tapahtumia on vain 2 500–20 000, aikaa kuluu alle 30 millisekuntia.

Eli lopputuloksena pystymme siirtämään kaiken tietokannoissa olevan datan suoraan käyttöliittymään käsiteltäväksi ilman, että käyttäjäkokemus kärsii. Käyttöliittymän käynnistyminen hidastuu keskimäärin 0,5 sekunnilla ja pahimmillaan 2 sekunnilla hitaallakin internetyhteydellä, ja päivitykset pystytään prosessoimaan helposti alle 100 millisekunnissa.

Käyttöliittymän puolella tehtävien laskentaprosessien ansiosta ei tarvitse implementoida monimutkaisempaa logiikkaa käyttöliittymän ja palvelinohjelmiston välillä. Samalla säästämme merkittävästi palvelinohjelmistolle aiheutuvaa kuormitusta ja pystymme helpommin käsittelemään kasvavia käyttäjämääriä. Lopputulos on myös yleisissä käyttötilanteissa merkittävästi nopeampi myös käyttäjän näkökulmasta, sillä suodatuskriteerien vaihtaminen ei vaadi käyttöliittymältä http-kyselyiden lähettämistä palvelinohjelmistolle, joka palauttaisi niihin sopivat tapahtumat.

Toteutustapa saattaa tarvita muutoksia siinä vaiheessa, kun valvontahälytysten määrät alkavat merkittävästi kasvaa, mutta tämänhetkisten suunnitelmien mukaan niitä olisi tarkoitus suodattaa ja vähentää kasvavissa määrin jo TSIM-palvelimien puolella, eli toteutetun ratkaisun pitäisi olla kestävä. Lisähuomiona prosessoinnin testauksessa on tärkeää asettaa React.js 'production'-tilaan, sillä se nopeuttaa merkittävästi prosessointia poistamalla käytöstä toimintoja, jotka ovat kehitysvaiheessa hyödyllisiä, mutta tuotannossa turhia [10]. Esimerkiksi debug- ja SourceMap-toiminnallisuudet ovat pois päältä 'production'-tilassa, jotka helpottavat ongelmien selvittämistä, mutta hidastavat koodin prosessointia.

5.3.3 Renderöinti

Renderöintilogiikka vaikuttaa eniten tilanteisiin, joissa näytettävää tietoa on enemmän kuin käyttöliittymän näkymään mahtuu, ja osa tiedoista piilotetaan

näkymältä vierityspalkin taakse. Tietojen renderöiminen näytölle vaatii niiden muodostamista React.js-komponenteiksi, mikä vaatii selaimelta prosessointia ja muistia. Pienille datamäärille prosessointi tapahtuu käytännössä välittömästi, mutta jos muutetaan kerralla 400 000 hälytystä renderöintivalmiiksi komponentiksi, vie se yli 5 sekuntia ja merkittävän määrän muistia. Tämä aiheuttaisi myös käyttöliittymän jäätymisen prosessoinnin ajaksi, mikä ei todellakaan olisi käyttäjävälittävää. Eli tarvitaan logiikka, jolla määritellään, mitkä kaikki hälytykset ovat näkymän kannalta tärkeitä ja tarvitsevat omat renderöintivalmiit komponentit.

Selaimessa pyörivälle käyttöliittymälle siirretään kaikki valvontahälytysdata sitä mukaan, kun palvelinohjelmisto niitä kerää. Tämän ansiosta meidän ei tarvitse renderöintivaiheessa miettiä, kuinka paljon käyttäjä saa selata tietoa ennen kuin uutta tietoa pitää hakea http-kutsulla palvelimelta. Eli voimme keskittyä suoraan näkymän kannalta oleellisiin komponentteihin ja saamme tiedot suoraan selaimessa pyörivän käyttöliittymän muistista.

Valvontahälytysdatan käsittely, eli suodattaminen ja järjestely, tapahtuu heti, kun se on noudettu palvelinohjelmistolta, ja renderöintiä varten meillä on oikeassa järjestyksessä oleva lista, jossa on kaikki näytölle sallitut vaihtoehdot jäljellä. Pystymme laskemaan, kuinka monta hälytystä mahtuu kerralle hälytysten näkymään laskemalla sen korkeuden pikseleissä ja jakamalla yksittäisen hälytyksen viemän korkeustilan pikselimäärät. Riippuen selaimen zoomitasosta ja käytetyn näytön resoluutiosta kyseessä on noin 7–50 hälytystä. Kyseisen tiedon perusteella ja seuraamalla käyttäjän scroll-liikkeitä voimme määritellä, milloin renderöidä vain näkymän kannalta tarvittavat hälytykset ja muodostaa vain niistä tarvittavat React.js-komponentit. Näin vältämme turhien komponenttien prosessoinnin ja tallettamista selaimen muistiin. Esimerkkikoodi 6 toteuttaa renderöitävien komponenttien lukumäärän laskennan. Esimerkistä näemme myös handleScroll-funktion, jonka avulla seurataan käyttäjän scroll-liikettä.

```

const handleScroll = (event) => {
  //nykyinen y arvo näkymän ylälaidassa
  const posY = event.target.scrollTop
  //scroll aiheuttama muutos y akselissa
  const deltaPosY = viewportTopY - posY
  //tarkistetaan onko liikuttu 1 hälytyksen verran
  const bChangeNeeded = (Math.abs(deltaPosY) > EVENT_ROW_HEIGHT)
  if (bChangeNeeded) {
    //päivitetään react state objecti joka päivittää näkymän
    setLastPosY({ viewportTopY: posY, clientHeight: event.target.clientHeight })
  }
}

//kaikki numerot ovat pixeli-määriä
//hälytyksien lista välikkö joka renderöidään
const renderQuantity = Math.floor(clientHeight / EVENT_ROW_HEIGHT)
const startIndex = Math.floor(viewportTopY / EVENT_ROW_HEIGHT)
const endIndex = startIndex + renderQuantity

//luo uuden listan joka sisältää vain renderöitävät hälytykset
const splitList = eventsProcessed.slice(startIndex, endIndex)

```

Esimerkkikoodi 6. Näkymän kannalta olennaisten valvontahälytysten lukumäärän ja sijaintien laskenta. StartIndex kertoo, mistä kohtaa hälytyslistasta ensimmäinen renderöitävä komponentti muodostetaan, ja endIndex kertoo viimeisen renderöitävän hälytyksen.

6 Tulevaisuuden kehitysehdotukset

Nykyisen kokonaisuuden suurimpia kehityskohteita ovat manuaalisen konfiguroinnin minimointi sekä skaalautuvuuden parantaminen suurempia hälytysmassoja ajatellen. Palvelinohjelmistoon pitää tällä hetkellä manuaalisesti päivittää TSIM-palvelimien listaa ja sijaintitietoja, mutta ne ovat myös teoriassa mahdollista saada TSPRES-rajapinnasta. Kyseisellä muutoksella palvelinohjelmisto pystyisi automaattisesti päivittämään TSIM-palvelinten listaa ja hakemaan niistä kaikki valvontahälytykset, kun ne kytketään TSPRES-palvelimeen.

Hälytysmassan potentiaaliseen kasvuun varautuminen ei näillä näkymin ole tarpeellista, koska tarkoituksena on vähentää TSIM-palvelimille kerääntyvien turhien hälytysten määrää. Valvottujen palvelimien määrä on kuitenkin voimakkaassa kasvussa, ja TSIM-palvelimien verkosto kasvaa joka vuosi, eli on täysin

mahdollista, että muutaman vuoden päästä 400 000 ei ole enää poikkeustilanteessa tapahtuva maksimiluku, vaan palvelinohjelmiston ja käyttöliittymän pitää selviytyä merkittävästi suuremmista määristä.

Kyseiseen tilanteeseen varautuminen vaatii ensinnäkin palvelinohjelmiston puolella valvontahälytysten siirtämistä pois ohjelmiston muistista johonkin tietokantaan, jotta Node.js-prosessin muistirajoitukset eivät ylity ja kaada koko prosessia. Samalla käyttöliittymälle ei voitaisi enää lähettää kaikkea hälytysdataa kerralla. Sen sijaan käyttöliittymän pitäisi lähettää palvelinohjelmistolle kriteerit, jonka perusteella se palauttaisi käyttöliittymälle vain tarvittavat hälytykset suoraan tietokannasta. Kyseinen ratkaisu vaatii myös todennäköisesti useamman palvelinohjelmistoinstanssin yhtäaikaista pyörittämistä palvelimella, jotta on riittävästi laskentatehoa käytössä.

7 Yhteenveto

Projektin määrittelyssä tavoitteena oli luoda web-sovellus, joka pystyy näyttämään kaikki TSIM-palvelinten valvontahälytykset yhdessä näkymässä tehokkaasti käsiteltävässä muodossa. Kriteerien mukaan käyttöliittymän tarvitsee pyöriä vain moderneissa selaimissa ja kyetä käsittelemään lähitulevaisuudessa näköpiirissä olevia valvontahälytysmääriä, eli keskimäärin noin 50 000 hälytystä ja poikkeuksellisesti 400 000 hälytystä kerralla.

Lopputuloksena syntyneet Node.js-pohjainen palvelinohjelma ja React.js-pohjainen selainkäyttöliittymä täyttävät projektin kriteerit, mutta jättävät vielä tilaa jatkokokehtamiselle. Tärkeimpänä tavoitteena oli kuitenkin vanhaa olemassa olevaa työkalua parempi versio, jota asiakastiimin käyttäjät pystyvät hyödyntämään tehokkaammin valvontahälytysten käsittelyssä. Asiakastiimin edustajan kommenttien avulla pystyttiin keskittymään heidän työskentelynsä kannalta oleellisempiin tekijöihin, kuten hälytysten suodatuskriteereiden monipuolisuuteen ja päivitysnopeuteen sekä erityisesti suuren hälytysmäärän selaamiseen.

Projektin haasteellisin vaihe oli käyttöliittymän ja palvelinohjelman välisen laskentaprosessien tasapainon etsinnässä. Suuret ja yleisesti käytössä olevat internetsivut kuten Facebook ja Reddit siirtävät käyttöliittymälle vain pienen osan tarjolla olevasta datasta, sillä niiden käytössä olevat datamassat ovat liian suuria käyttöliittymän prosessoitavaksi. Kyseinen toimintamalli on kuitenkin merkittävästi hankalampi toteuttaa kuin tämän projektin malli. Järkevällä optimoinnilla ja algoritm ratkaisulla saatiin aikaiseksi huomattavasti nopeampi ja helpompi ratkaisu, kuin mitä palvelin pohjaiseen laskentaan perustuva ratkaisu olisi ollut.

Lähteet

- 1 Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP. 2011. Verkkoaineisto. Internet Engineering Task Force. <<https://tools.ietf.org/html/rfc6202#section-2.1>>. Luettu 4.5.2021.
- 2 Express.js API reference. 2019. Verkkoaineisto. OpenJS Foundation. <<http://expressjs.com/en/5x/api.html#app.onmount>>. Luettu 5.5.2021.
- 3 Long-Polling vs Websockets vs Server-Sent Events. 2020. Media. Amit2197kumar. <<https://systemdesignbasic.wordpress.com/2020/02/01/12-long-polling-vs-websockets-vs-server-sent-events>>. Luettu 5.5.2021.
- 4 The Open Group Base Specifications. 2018. Verkkoaineisto. IEEE and The Open Group. <https://pubs.opengroup.org/onlinepubs/9699919799/xrat/V4_xbd_chap04.html#tag_21_04_15>. Luettu 4.5.2021.
- 5 EventSource. 2021. Verkkoaineisto. Mozilla. <<https://developer.mozilla.org/en-US/docs/Web/API/EventSource>>. Päivitetty 16.4.2021. Luettu 2.5.2021.
- 6 Node.js v16.10 documentation - Zlib. 2021. Verkkoaineisto. OpenJS Foundation. <<https://nodejs.org/api/zlib.html>>. Luettu 4.5.2021.
- 7 Array.prototype.sort(). 2021. Verkkoaineisto. Mozilla. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort>. Päivitetty 28.4.2021. Luettu 2.5.2021.
- 8 Getting things sorted in V8. 2018. Verkkoaineisto. Zünd, Simon. <<https://v8.dev/blog/array-sort>>. Luettu 2.5.2021.
- 9 ListSort. 2020. Verkkoaineisto. Pochmann, Stefan. <<https://github.com/python/cpython/blob/main/Objects/listsort.txt>>. Päivitetty 3.2.2020. Luettu 2.5.2021.
- 10 Optimizing Performance. 2021. Verkkoaineisto. Facebook Inc. <<https://reactjs.org/docs/optimizing-performance.html>>. Luettu 05.05.2021.