Aukusti Palomäki

SERVER-SENT EVENTS JAVA-PALVE-LINKÄYTTÄJÄVIESTINNÄSSÄ

Opinnäytetyö

Liiketalouden ammattikorkeakoulututkinto

Tietojenkäsittelyn koulutus

2024

Kaakkois-Suomen ammattikorkeakoulu

Tutkintonimike	Tradenomi (AMK)
Tekijä	Aukusti Palomäki
Työn nimi	Server-Sent Events Java-palvelinkäyttäjäviestinnässä
Toimeksiantaja	Disec Oy
Vuosi	2024
Sivut	28 sivua
Työn ohjaaja(t)	Miia Liukkonen, Jani Haiko

TIIVISTELMÄ

Tämän opinnäytetyön tavoitteena oli tutkia Server-Sent Events- (SSE) ja WebSocket-tekniikoiden välisiä eroja ja vertailla niiden käytännön soveltamista Disecin Yksa-järjestelmän viestinnän kehittämiseksi. Ensisijaisena tavoitteena on kehittää prototyyppi käyttäen SSE:tä reaaliaikaisten käyttäjäilmoitusten parantamiseksi Yksa-järjestelmässä ja laajentaa tätä prototyyppiä edelleen eri viestintätilanteita varten. Esimerkiksi käyttäjä saisi ilmoituksen, jos samaa asiakirjaa muokkaa joku muukin samanaikaisesti Yksa-järjestelmässä, mikä vähentäisi ristiriitoja ja parantaisi yhteistyötä. Lisäksi prototyyppi toimisi viestintäkanavana, jonka kautta käyttäjille ilmoitettaisiin järjestelmän päivityksistä ja huoltokatkoista.

Opinnäytetyön tavoitteena on myös vertailla Websocket-tekniikkaa ja SSE:tä. Vertailu antaa Disecille perustan, jonka pohjalta se voi tulevaisuudessa harkita WebSocket-tekniikan mahdollista käyttöönottoa SSE:n rinnalla. Prototyyppi on kehitetty SSE-tekniikalla ja WebSocket on toiminnallisena vertailukohteena kuin käytännön toteutuksena.

Opinnäytetyö rajoittuu näiden kahden viestintämenetelmän vertailuun, ja keskittyy enemmän SSE:n toiminnallisuuteen ja toteutukseen. WebSocket, joka on SSE:n ohella yleisimmin käytetty menetelmä, toimii sopivana vertailukohteena. Opinnäytetyö vertailee suorituskykyä, skaalautuvuutta, toteutuksen helppoutta ja soveltuvuutta reaaliaikaisen viestintään Yksa-järjestelmässä.

Opinnäytetyön tavoitteena oli luoda prototyyppi, jolla on mahdollista lähettää ilmoituksia käyttäjille tai kaikille hyödyntäen palvelintyöntö- eli server-push-tekniikkaa. Samalla vertaillaan eroja Server-Sent Events- ja WebSocket-tekniikan välillä, jotka ovat reaaliaikaisia web-tekniikoita tiedon lähettämistä varten.

Lopputuloksena saatiin toimiva prototyyppi, jolla voidaan luoda ilmoituksia, jotka ovat joko kaikille tai tietylle käyttäjälle sekä poistamaan niitä. Prototyyppi osoitti, että Disecin Yksa-järjestelmään voi soveltaa hyvin Server-Sent Events-tekniikkaa ja laajentaa sen käyttämistä muihinkin sovelluksen ominaisuuksiin kuin pelkästään ilmoituksien luomiseen.

Asiasanat: Server-Sent Events, WebSocket, reaaliaikaisuus, web-tekniikat

South-Eastern Finland
University of Applied Sciences

Degree title	Bachelor of Business Administration
Author	Aukusti Palomäki
Thesis title	Server-Sent Events in Java server-client communication
Commissioned by	Disec Oy
Time	2024
Pages	28 pages
Supervisor	Miia Liukkonen, Jani Haiko

ABSTRACT

The objective of this thesis was to investigate the differences between Server-Sent Events (SSE) and WebSocket technologies and to compare their practical application for enhancing communication in Disec Yksa. The primary objective was to develop a prototype using SSE to improve real-time user notifications in Yksa and to further extend this prototype for different communication situations. For example, a user would be notified if the same document is being edited by someone else in Yksa at the same time, thus reducing conflicts and improving collaboration. In addition, the prototype would serve as a communication channel to inform users about system updates and maintenance downtime.

The thesis also aimed to make a comparison between WebSocket and SSE techniques. This comparison would provide a basis for Disec to consider the possible future deployment of WebSocket alongside SSE. The prototype has been developed using SSE technology and WebSocket servers as a functional benchmark based on comparison rather than practical implementation.

The thesis was limited to a comparison of the two communication methods, focusing more on the functionality and implementation of SSE. WebSocket, which is the most used method alongside SSE, served as a suitable comparison. The thesis compared performance, scalability, ease of implementation and suitability for real-time communication in the Yksa application.

The aim of the thesis was to create a prototype that could be used to send notifications to users or everyone using server push technology. At the same time, the differences between Server-Sent Events and WebSocket, as real time web techniques for sending information, were compared.

The result was a working prototype for creating notifications that were either for everyone or for a specific user, as well as for deleting them. The prototype showed that the Server-Sent Events technology could be well applied to Yksa and extended to other applications beyond just creating notifications.

Keywords: Server-Sent Events, Web-Socket, real time, web techniques

SISÄLLYS

1 JOHDANTO

Tämän opinnäytetyön aiheena on Server-Sent Events (SSE) -tekniikan soveltaminen Yksa-järjestelmään, jotta voidaan kehittää käyttäjien ja järjestelmän välistä reaaliaikaista viestintää. Server-Sent Events termille ei ole suomennosta, joten opinnäytetyössä puhumme lyhenteellä SSE. Tavoitteena on kehittää prototyyppi, jossa hyödynnetään SSE:tä reaaliaikaisten käyttäjäilmoitusten tehostamiseksi Yksa-järjestelmässä. Prototyyppi toimii viestintäkanavana, joka tarjoaa välittömiä ilmoituksia, kuten järjestelmäpäivityksistä, huoltokatkoista tai muista ilmoitusta vaativista asioista. Tällä hetkellä on mahdollista luoda vain yksi viesti kaikille asiakkaille ja prototyypin tarkoitus on mahdollistaa yksilöllinen viestintä asiakasryhmille, käyttäjille tai kaikille.

Toissijaisena tavoitteena vertailla WebSocket-tekniikkaa ja SSE:tä. Vertailu antaa Disecille tietoa, jonka avulla se voi harkita WebSocket-tekniikan mahdollista käyttöönottoa SSE:n rinnalla. Prototyyppi kehitetään yksinomaan SSE-tekniikkaa käyttäen, ja WebSocket-tekniikka toimii toiminnallisena vertailukohtana. Vertailu mahdollistaa suorituskyvyn, skaalautuvuuden ja toteutuksen helppouden arvioinnin reaaliaikaisen viestinnän näkökulmasta. Esimerkiksi tekniikoiden integrointi olemassa oleviin järjestelmiin ja sen tarittavat infrastruktuurimuutokset. Prototyyppiä testataan mahdollisen laajasti, jotta saadaan selville mahdollisia jatkokehitys ideoita.

Tämän opinnäytetyön tavoitteena on tunnistaa sekä SSE:n että WebSocket-tekniikan vahvuudet sekä rajoitukset antaen perustetta SSE:n käyttöönotosta reaaliaikaisena viestintäratkaisuna Disecin Yksa-järjestelmään. Prototyyppi antaa mahdolliset kehityssuunnat ja jatkotoimenpiteet reaaliaikaisen viestintäominaisuuksien toteuttamisesta Yksa-järjstelmässä.

2 REAALIAIKAINEN WEB-VIESTINTÄ

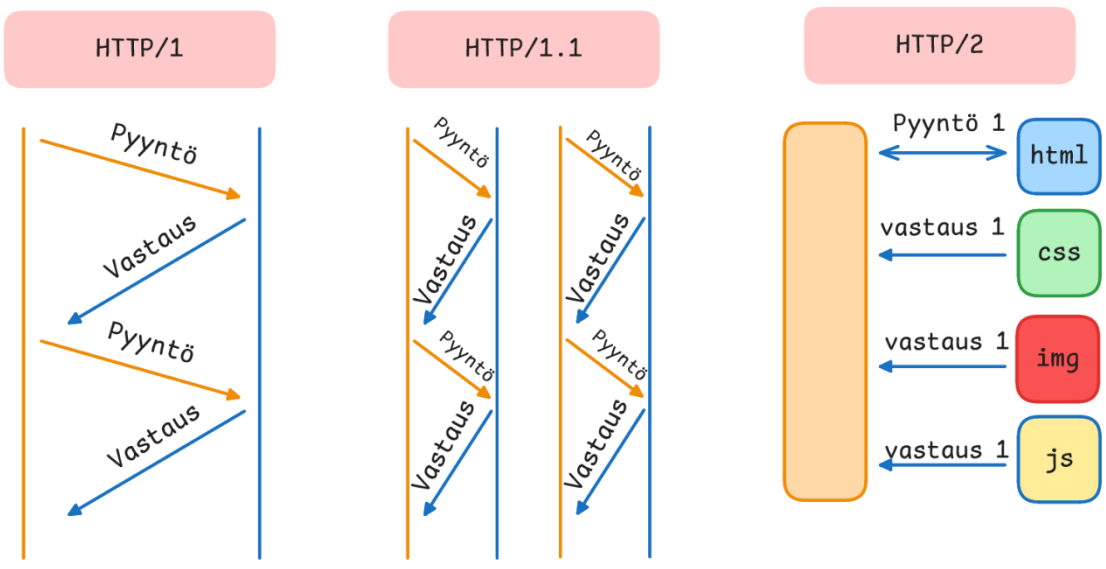
Reaaliaikaiset viestintätekniikat mahdollistavat välittömän tiedonsiirron palvelimen ja sen käyttäjien välillä. Perinteinen HTTP, Hypertext Transfer Protocol, eli hypertekstin siirtoprotokollan pyyntövastauksesta poiketen reaaliaikaiset viestintätekniikat mahdollistavat jatkuvan tiedon vaihdon. Tämä antaa dynaamisemman ja vuorovaikutteisemman käyttäjäkokemuksen. SSE:n (Server-

Sent Events) ja WebSocket kaltaiset tekniikat ovat keskeisiä reaaliaikaisessa viestinnässä. SSE helpottaa yksisuuntaisessa viestinnässä palvelimelta asiakkaalle, kun taas WebSocket tukee kaksisuuntaisen viestinnän yhden yhteyden kautta. (Cloudflare s.a.)

2.1 HTTP:n kehitys

HTTP:n kehitys on ollut kesteistä reaaliaikaisen viestintätekniikan kehitykselle. Yksinkertaiseen tiedostojen vaihtoon suunniteltu HTTP on kehittynyt tukemaan monimutkaisia verkkosovelluksia. HTTP/1.0 toi mukanaan tilakoodit, versiohallinnan ja otsaketiedot, joka teki kutsuprotokollasta monipuolisemman ja laajennettavan. Merkittävä kehitysaskel tapahtui, kun HTTP/1.1 standardisoitiin, ottaen käyttöön jatkuvat yhteydet ja tiedon putkittamisen. Lisäksi se mahdollisti datan sisällön neuvottelun. Uudistuksen myötä HTTP/1.1 monimutkaisempia sovelluksia, kuten WebDAV. WebDAV mahdollistaa tiedostojen muokkaamisen, siirtämisen ja luomisen palvelimella. (Mozilla 2024b.)

Vaikka HTTP/1.1 mahdollisti useiden pyyntöjen ja vastausten saamisen yhden yhteyden kautta ja tuki reaaliaikaista viestintää, sillä on kuitenkin tiettyjä rajoituksia. Keskeinen rajoitus on useimpien selainten asettama kuuden samanaikaisen yhteyden rajoitus per verkkosivu (Mozilla 2024c). Tämä rajoitus voi aiheuttaa ongelmia sovelluksissa, jotka vaativat reaaliaikaisia päivityksiä, etenkin kun käyttäjä aukaisee useita välilehtiä. Ongelma koskee erityisesti palvelimen lähettämiä tapahtumia SSE:n osalta, sillä kukin avoin välilehti yrittää muodostaa oman yhteyden palvelimeen, joka johtaa yhteyden katkeamiseen, kun yli kuusi välilehteä on aktiivisena.



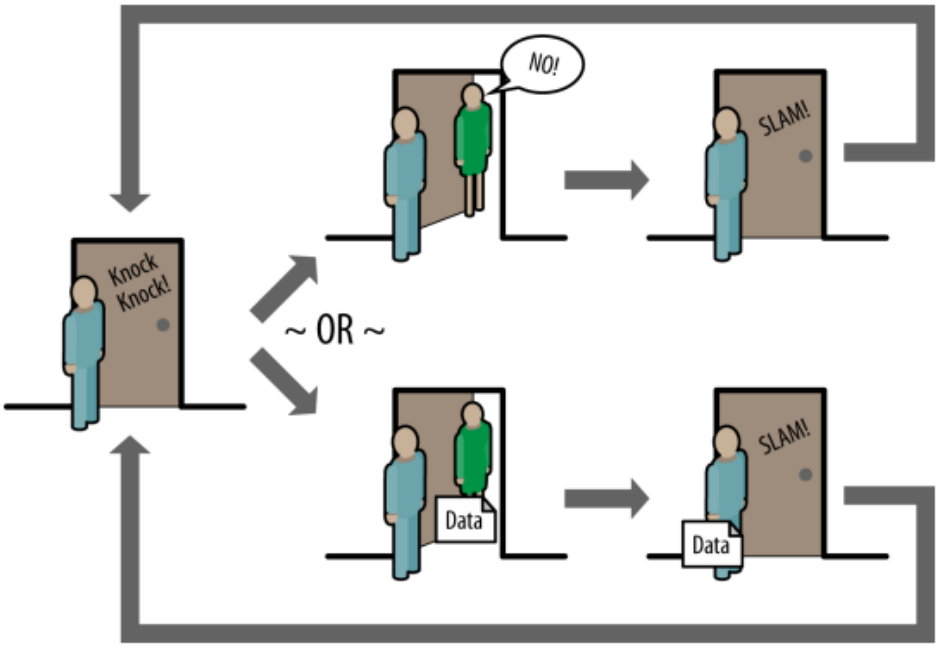
Kuva 1. Malli HTTP pyynnöistä eri versioissa

Vuonna 2015 julkaistussa HTTP/2-standardissa ratkaistiin monia HTTP/1.1:ssä havaittuja suorituskykyongelmia ottamalla käyttöön multipleksointi-tekniikka, multiplexing, joka mahdollistaa useiden pyyntöjen lähettämisen ja vastausten vastaanottamisen yhden yhteyden kautta, kuten kuvassa yksi esitetään. Tämä parannus nosti yhteysrajan mainitusta kuudesta samanaikaisesta yhteydestä sataan samanaikaiseen yhteyteen sivua kohden (Mozilla 2024c.) HTTP/2 sisältää myös push-palvelimen, server push ja otsaketietojen, headers, pakkaamisen, jotka parantavat verkkosuorituskykyä. HTTP/3, joka käyttää QUIC-protokollaa TCP:n sijaan, paransi suorituskykyä ja turvallisuutta entisestään vähentämällä viivettä ja tehostaen virheenkäsittelyä. (Mozilla 2024b.)

Reaaliaikaiset viestintätekniikat ovat olennainen osa nykyaikaista web-kehitystä ja ne mahdollistavat verkkokeskustelut, suorat päivitykset ja yhteistyövälineet. Esimerkiksi yksisuuntaisella kommunikoinnilla toimiva SSE toimii hyvin ohjelmissa, jotka vaativat mahdollisen pientä viivettä tiedon lähettämisessä. Esimerkiksi lentojen aikataulun esittämisessä, jossa saapumis- ja lähtöaikajoen ajantasaisuus on ratkaisevaa, SSE voi tarjota reaaliaikaisia päivityksiä ilman, että käyttäjän täytyisi jatkuvasti päivitellä sivua tai lähettää toistuvia pyyntöjä.

2.2 Polling-tekniikat

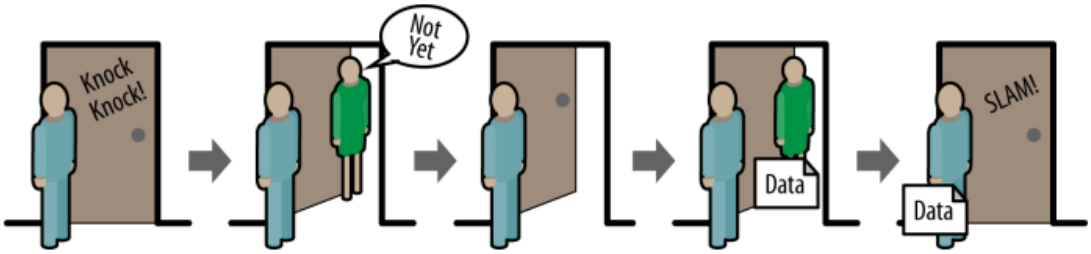
Polling on peräkkäistä kyselyä asiakkaalta palvelimelle kysymys-vastaus-menetelmällä. Verkkosovelluksissa tätä tekniikkaa varten hyödynnetään AJAX-kutsuja, jotka voivat olla esimerkiksi 0,5 sekunnin välein tehtäviä palvelinkutsuja tai yhteyden ylläpitämistä Keep-Alive-asetuksella otsakkeessa. Verkkosovellus avaa yhteyden palvelimeen HTTP kutsun avulla, kysyy tietoa, vastaanottaa tiedon ja yhteys katkeaa. Tätä kolmea askelta toistetaan jatkuvasti taustalla luoden dynaamisen käyttäjäkokemuksen, mahdollistaen reaaliajassa sivun päivittämisen (Javascript.info 12.11.2022).



Kuva 2. Esimerkkikuvaus mitä kiertokysely on

Tämä ei kuitenkaan ole alun perin tarkoitettu tapa hyödyntää HTTP:n kutsua, vaan ratkaisu ennen kuin WebSocket sekä SSE kehitettiin ja vähitellen yleistyivät. Kiertokysely aiheuttaa seuraavan ongelman, resurssien käytön. Esimerkiksi jatkuvan GET-pyynnön tekeminen ja vastauksen odottaminen luo tilanteen, jossa luodaan tyhjiä vastauksia, tuhlataan myös palvelimen resursseja. Pienessä skaalassa tämä saattaa tuntua pieneltä ongelmalta, mutta kun aletaan puhumaan sadoista tuhansista tai miljoonista pyynnöistä minuutin sisällä, vaatii se palvelimelta paljon enemmän tehoa (Javascript.info 2022).

Eri sovelluksille on erilaisia kommunikointiratkaisuja, jotka ovat soveltuvaisempia kuin muut. Esimerkiksi sääsovellus ei vaadi sekunnin tarkkaa päivittämistä, siihen voi riittää esimerkiksi 5 minuutin välein oleva päivittäminen, jolloin kiertokyselytekniikka soveltuu tähän käyttötapaan hyvin. Uutiset ovat myös toinen yhteydenlähde, joka ei vaadi millisekunnin sisällä olevaa reagoimista. Edestakaisin kommunikoimisessa vaaditaan kuitenkin parempi tapa välittää tietoa vähentäen viivettä sekä mahdollista palvelintehontarvetta. (Cook 2014.)



Kuva 3. Esimerkkikuvaus mitä pitkä kiertokysely, long-polling on (Cook 2014.)

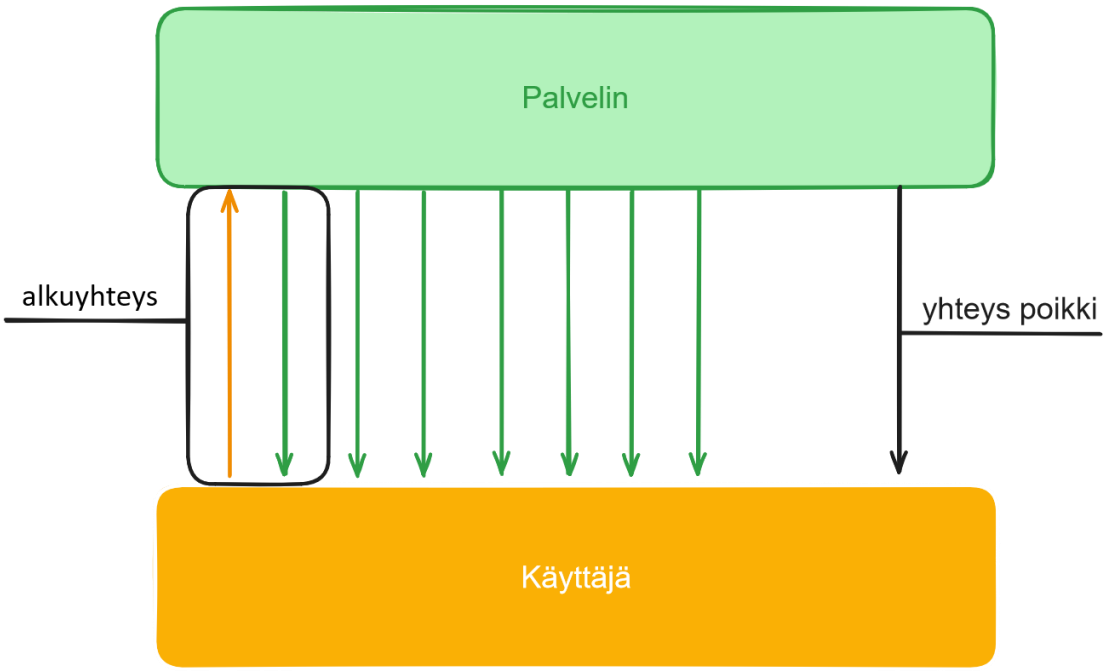
Polling-tekniikan lisäksi on kiertokysely (long-polling). Kiertokysely-tekniikassa luodaan GET-pyyntö, jota palvelin käsittelee niin pitkään, kunnes saa uutta tietoa, jonka se lähettää käyttäjälle. Esimerkiksi saman ongelman voisi ratkaista luomalla aikataulutettu GET-pyyntö, mutta tämä aiheuttaa tilanteen, jossa tiedon voisi saada myös nopeammin. Mitä jos tieto olisikin mahdollista saada 13 sekunnin päästä? Käyttäjä joutuisi odottamaan ylimääräisen seitsemän sekuntia, jos pyyntö olisi ajoitettua 20 sekunnin välein. Kiertokyselyn GET-pyyntö sen sijaan voisi kestää 30 sekuntia, jonka aikana, jos tulee uutta informaatiota, se lähetetään asiakkaalle heti. (Bidelman 2010.)

Kiertokysely kuulostaa todella samalta kuin SSE, mutta suurin ero näiden välillä on yhteystapa. SSE on pysyvä yhteys, joka katkeaa vain käskystä tai virheestä, kun taas kiertokysely on lyhytaikainen yhteys, jota toistetaan tietyin ajoin. SSE pysyvän yhteyden ansiosta syntyy vähemmän rasitusta palvelimenpuolelta kuin kiertokyselyn jatkuvan yhteyden luonti ja sen todentaminen. Tämä myös lisää viivettä verrattuna SSE.

3 SERVER SENT EVENTS- JA WEBSOCKET-TEKNOLOGIAT

3.1 Server-Sent Events

Server-Sent Events (SSE) on verkkoprotokolla, joka mahdollistaa yksisuuntaisen yhteyden palvelimelta käyttäjälle (Ger 2024). SSE on osana HTML5-teknologiaa, joka tarjoaa yksinkertaisen tavan työntää dataa käyttäjälle, server push, hyödyntäen vain HTTP-yhteyttä. SSE:n ensisijainen käyttötarkoitus on ylläpitää jatkuvaa yhteyttä datan välittämiseen reaaliajassa palvelimelta käyttäjälle. Koska SSE on yksisuuntainen yhteys, nopeuden ja turvallisuuden näkökulmasta se on usein paras valinta, jos tavoitteena on lähettää pelkästään tietoa teksti muodossa käyttäjille.



Kuva 4. Malli SSE yhteydestä

SSE-yhteyttä kutsutaan data push -tekniikaksi, eli datan työntämiseksi (Cook 2014). Jos SSE ei täytä vaatimuksia käyttäjän ja palvelimen kommunikoimiseen, voidaan harkita vaihtoehtoisia menetelmiä, kuten WebSocket-tekniikka tai kiertokysely (polling).

3.2 Server-sent Events HTTP 1.1:n käytössä oleva rajoitus

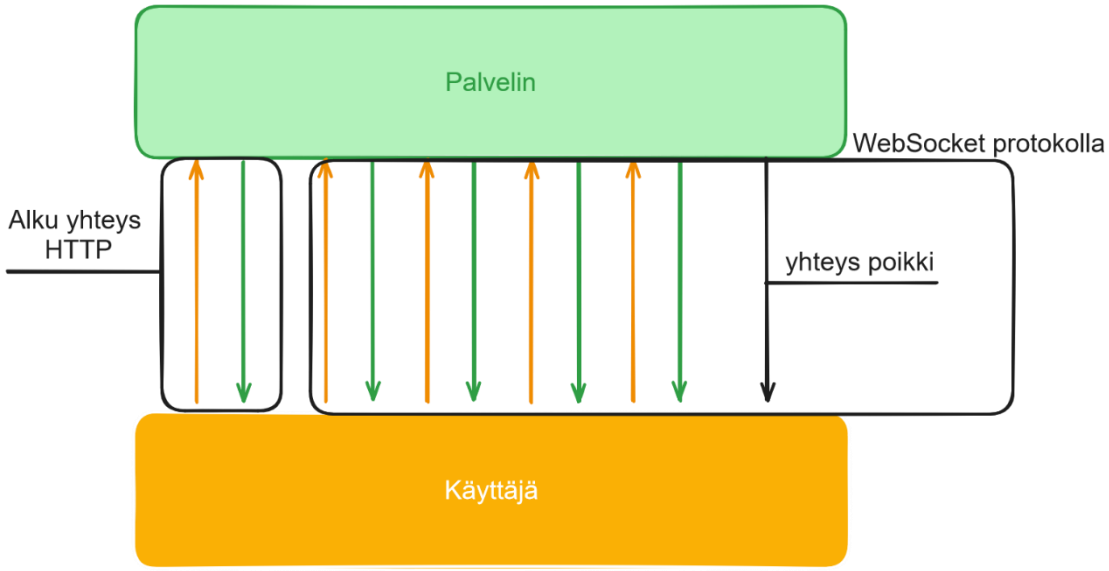
Yksi suurimmista haasteista käyttäessä SSE:tä HTTP/1.1:n välityksellä on selainten asettama kuuden samanaikaisen yhteyden rajoitus per verkkosivusto. Jokainen välilehti tai ikkuna, joka avautuu samalta verkkosivustolta, yrittää luoda oman SSE-yhteyden, jos verkkosivusto ei ole erikseen ohjattu toimimaan muuten. Kun välilehtiä on avattu yli kuusi, uudet välilehdet eivät välttämättä saa yhteyttä. Tämä voi häiritä reaaliaikaista tiedonkulkua erityisesti silloin, kun käyttäjät avaavat useita välilehtiä. (Mozilla 2024c.) Yksi ratkaisutapa tähän on käyttää BroadcastChannel-ohjelmointirajapintaa.

BroadcastChannel-rajapinnan avulla useat saman alkuperän välilehdet tai ikkunat voivat jakaa tietoja ilman, että jokainen välilehti avaa oman yhteyden palvelimeen. Tämä sovellusrajapinta mahdollistaa eri selainkontekstien, kuten välilehtien välisen reaaliaikaisen viestinnän. (Mozilla 2024a). Ensimmäinen välilehti avaa SSE-yhteyden palvelimeen ja luo BroadcastChannel-kanavan.

Seuraavat välilehdet eivät avaa uusia SSE-yhteyksiä, sen sijaan ne muodostavan yhteyden samaan BroadcastChannel-kanavaan ja vastaanottavat päivityksiä, jotka SSE-yhteyttä ylläpitämä välilehti lähettää. Tämän avulla voi välttää HTTP/1.1:n kuuden yhteyden rajan ylittyminen.

3.3 WebSocket

WebSocket-tekniikka on viestintäprotokolla, joka helpottaa reaaliaikaista vuorovaikutusta palvelimen ja asiakkaan, client, välillä. Toisin kuin SSE, WebSocket tukee kaksisuuntaista viestintää TCP-protokollan avulla (Mozilla).



Kuva 5. Malli WebSocket yhteydestä

Vaikka Websocket mahdollistaa kaksisuuntaisen viestinnän, se lisää monimutkaisuutta ja mahdollisia turvallisuusriskejä, koska se on riippuvainen TCP-protokollasta, joka toimii erillään HTTP-protokollasta (Mozilla s.a). WebSocket-tekniikka helpottaa reaaliaikaista viestinnän mahdollistamista käyttäjien ja palvelimien välillä, esimerkiksi viestintäsovelluksissa tai yhteistyövälineissä.

3.4 Tietoturvallisuus ja WebSocket-yhteyksien haasteet

WebSocket-protokolla suunniteltiin alun perin käytettäväksi verkkoselaimissa toimivien skriptien kanssa, mutta sitä voidaan käyttää myös muissa kuin selainpohjaisissa ohjelmissa. Tämä tuo mukanaan tietoturvariskejä erityisesti origin-otsakkeen mahdollisen väärinkäytön vuoksi. Origin-otsake toimii turvana luotetuissa selainympäristöissä toimivaa haitallista JavaScriptiä vastaan ja varmistaa, että vain valtuutetut skriptit voivat aloittaa WebSocket-yhteyksiä.

Kyseistä otsaketta on kuitenkin mahdollista manipuloida tai väärentää, jolloin voidaan esittää harhaanjohtava alkuperä, origin. (Fette & Melnikov 2011.)

Pelkästään origin-otsakkeeseen luottaminen turvallisuuden takaamiseksi on riittämätöntä. Palvelimen täytyy tarkistaa WebSocketin kautta saapuva tieto hyvin tarkasti, välttääkseen tietoturvahyökkäyksiä. Esimerkiksi SQL-tietokantahyökkäyksessä yritetään syöttää SQL-kyselyitä muokkaamaan tietokannan tietoja, tätä kutsutaan SQL-injektiohyökkäykseksi. Jotta tältä voidaan välttyä, täytyy kaikki tieto, jota WebSocketin kautta lähetetään muuntaa sellaiseksi, ettei SQL kykene tulkitsemaan sitä. Tätä kutsutaan escapettamiseksi, jolloin tietyt merkit muutetaan eri muotoon, jottei niitä kyetä tulkitsemaan. (Fette & Melnikov 2011.)

WebSocketin yhtenä yleisenä riskinä on cross-origin-hyökkäykset. Kun haitallinen verkkosivusto tai skripti yrittää luoda WebSocket-yhteyden palvelimeen, joka odottaa yhteyttä vain tietyltä alkuperältä, originin, tulee kyseinen alkuperäotsake tarkistaa. Tarpeen vaatiessa yhteys tulee hylätä, ja tämä on yleensä HTTP 403 (virhe)-tilakoodilla. Ilman tätä suojausta hyökkääjä voisi käyttää luotetussa selaimessa toimivaa skriptiä luomaan luvattoman WebSocket-yhteyden, joka antaa mahdollisuuden esiintyä luotettavana käyttäjänä ja voi johtaa tietojen manipulointiin tai varastamiseen. (Fette & Melnikov 2011.)

WebSocketilla ei ole tiettyä tapaa asiakkaiden todennukseen ensimmäisen yhteyden, kättelyprosessin aikana. Palvelin voi kuitenkin hyödyntää perinteisiä HTTP-pohjaisia todennusmenetelmiä, kuten todennustietoja, evästeitä tai transport layer security todennusta, kuljetustason turvallisuussalausprotokolla (TLS). TLS salaa asiakkaan ja palvelimen välillä vaihdetut tiedot suojaamaan salakuuntelulta ja manipuloinnilta. (Fette & Melnikov 2011.)

Koska SSE (Server-Sent Events) on osa HTML5-teknologioita, toimii se HTTP:n kautta tehden siitä valmiiksi turvallisemman kuin WebSocket ja kevyemmän (Mozilla 2024c). Tämä myös tarkoittaa sitä, että SSE ei tuota mahdollisia ongelmia palomuurin kanssa, joka voi mahdollisesti voi käydä WebSocketin kanssa, joka ei omista tiettyä natiivi tapaa todentaa yhteyttään ja tietoja palvelimelle. Yksinkertaisen arkkitehtuurin, vahva TLS-salaus, vähäisempi

hyökkäyspinta, tehokas resurssien hallinta tekee SSE:n valitsemisesta luotettavuuden ja tietoturvan näkökulmassa varmemmalta valinnalta kuin WebSocket.

4 ILMOITUKSIEN LÄHETTÄMINEN

Opinnäytetyön toimeksiantajana toimii Disec Oy, joka tarjoaa eri yrityksille Yksa-arkiston- ja kokoelmahallintajärjestelmän. Käyttäjämäärän kasvaessa tarve parantaa palvelimen ja asiakkaiden välistä viestintää on korostunut. Yksi keskeinen parannuskohde on ilmoitusjärjestelmä, joka mahdollistaa ennakkoilmoitusten tekemisen. Esimerkiksi tuotantopäivityksestä seuraama käyttökatkos tai tiedotus mahdollisista järjestelmävioista. Tätä varten kehitetään prototyyppi, jossa käytetään SSE-protokollaa, joka mahdollistaa reaaliaikaiset ilmoitukset asiakkaille mahdollisen nopeasti.

Yksa käyttää Spring- ja Vue.js-ohjelmistokehyksiä, framework, joissa Spring toimii palvelinpuolella ja Vue.js selainpuolella. Spring on Java-kielellä toteutettu kehys, joka tukee keskeisiä ominaisuuksia, kuten rajapintojen kehitystä. Vue.Js puolestaan on JavaScript-pohjainen käyttöliittymäkehys, joka mahdollistaa komponenttipohjaisen rakenteen ja tilanhallinnan. Ilmoitusten hallintalogiikka on kirjoitettu Spring-kehykseen ja ilmoitusten ulkoasu on toteutettu uudelleenkäytettävänä Vue-komponenttina, joka helpottaa prototyypin jatkokehitystä ja ylläpitoa.

4.1 Ilmoituksien logiikka

Prototyypin kehittämisessä hyödynnetään kahta eri polkua erottamaan käyttäjien ja ylläpitäjien toimintoja. Tämä mahdollistaa tietoturvallisen osoitteen, johon ei pääse kuin järjestelmänvalvojat luomaan ilmoituksia käyttäjille. Koska molemmat polut käyttävät samaa tietoa, rakennetaan keskitetty palveluluokka, service class, joka hallitsee ilmoituslogiikkaa ja jakaa tarvittavia toimintoja muille. Palveluluokan nimi tulee olemaan SSEService. Koska molemmat polut tarvitsevat ilmoitukseen liittyviä tietoja, SSEService käytetään myös ilmoituksien tallentamiseen (kuva 6). Prototyypissä tallennetaan ilmoituksien tiedot väliaikaisesti muistiin.

```
@Service  18 usages
public class SSEService {

    private static final Logger log = LogManager.getLogger(SSEService.class);  4 usages

    private final Set<MessageData> globalMessages = ConcurrentHashMap.newKeySet();  8 usages
    private final Map<String, SseEmitter> emitters = new ConcurrentHashMap<>();  4 usages
    private final Map<String, Set<MessageData>> sessionMessages = new ConcurrentHashMap<>();  8 usages
```

Kuva 6. Ilmoituksien tiedon tallentaminen muistissa

Koska käsitellään ilmoituksia, jotka noudattavat yhdenmukaista rakennetta, voimme hyödyntää Javan sisäänrakennettua record-tyyppiä. Record mahdollistaa tietueiden kätevän määrittelyn, mikä osaltaan vähentää toistuvaa koodia ja tekee sovelluksesta helpommin ylläpidettävän ja selkeämmän.

```
public record MessageData(String id,
                          String title,
                          String message,
                          String color,
                          boolean countDown,
                          String endDateTime) {
}
```

Kuva 7. Ilmoitus viestin rakenne

Ilmoitusviestin tallentamisesta varten kirjoitetaan apumetodi, jonka avulla viesti voidaan muuttaa standardoituun muotoon. Apumetodia käytetään sekä yksityisten että kaikille käyttäjille tarkoitettujen ilmoitusten luontiin. Tämä myös helpottaa sovelluksen ylläpitoa, kun on vain yksi paikka, jota tarpeen vaatiessa täytyy muokata.

```
/**
 * Helper method to create a JSON message from MessageData.
 *
 * @param messageData the MessageData object to convert to JSON.
 * @return a JSON message data.
 */
private static String createJsonMessage(final MessageData messageData) {
    return String.format(
        "{\"id\":\"%s\",\"title\":\"%s\",\"message\":\"%s\", \"endDateTime\":\"%s\", \"countDown\":%s, \"color\":\"%s\"}",
        messageData.id(),
        messageData.title(),
        messageData.message(),
        messageData.endDateTime(),
        messageData.countDown(),
        messageData.color()
    );
}
```

Kuva 8. Apumetodi-Ilmoitus viestin rakentamiselle

Seuraava vaihe on itse ilmoituksen tallennus muistiin. Sovellus ylläpitää kahta erilaista viestikokonaisuutta: kaikille käyttäjille tarkoitettuja ilmoituksia (kuva 9), ja käyttäjäkohtaisia ilmoituksia (kuva 10). Näiden kahden ilmoituskategorian hallintaan kirjoitetaan kaksi metodia, jotka vastaavat viestien tallentamisesta.

```java
/**
 * Add a global message and notify all connected users.
 *
 * @param messageData the data of the global message to be updated.
 */
public void addGlobalMessage(final MessageData messageData) {  1 usage
    // Create a new MessageData instance for the global message
    final MessageData globalMessage = new MessageData(
        UUID.randomUUID().toString(),
        messageData.title(),
        messageData.message(),
        messageData.color(),
        messageData.countDown(),
        messageData.endDateTime()
    );
    globalMessages.add(globalMessage);
    for (final SseEmitter emitter : emitters.values()) {
        try {
            emitter.send(SseEmitter.event().data(globalMessage, MediaType.APPLICATION_JSON));
        } catch (final IOException e) {
            log.error( message: "Error sending global message to session: {}", e.getMessage());
            emitter.completeWithError(e);
        }
    }
}
```

Kuva 9. Metodi, jolla lisätään kaikille saatavilla oleva ilmoitus

```java
/**
 * Add a session message for a specific user session.
 *
 * @param sessionId    identifier for the user session.
 * @param messageData  message data to be stored.
 */
public void addSessionMessage(final String sessionId, final MessageData messageData) {  1 usage
    final Set<MessageData> messages = sessionMessages.computeIfAbsent(sessionId, k → ConcurrentHashMap.newKeySet());
    messages.add(new MessageData(UUID.randomUUID().toString(),
                                 messageData.title(),
                                 messageData.message(),
                                 messageData.color(),
                                 messageData.countDown(),
                                 messageData.endDateTime()));
}
```

Kuva 10. Metodi, jolla lisätään käyttäjäkohtainen ilmoitus

Käyttäjäkohtaisen ilmoituksen käsittely eroaa yleisistä ilmoituksista siten, että viesti tallennetaan käyttäjän sessiotunniste, sessionId. Sessiotunniste on uniikki jokaiselle käyttäjälle kunkin kirjautumiskerran aikana, mikä takaa viestin yksityisyyden ja turvallisuuden. Kun käyttäjä kirjautuu ulos, viestejä ei enää

uudessa kirjautumisessa näe, sillä sessiotunniste on aina uusi. Tämä mekanismi suojaa viestien päättymistä väärille käyttäjille.

```java
/**
 * Send stored session messages and global messages to a user session.
 *
 * @param sessionId   the identifier for the user session.
 * @param emitter     the SseEmitter to send messages to.
 */
public void sendSessionMessages(final String sessionId, final SseEmitter emitter) {  1 usage
    // Send session-specific messages
    final Set<MessageData> sessionMessagesSet = sessionMessages.get(sessionId);
    if (sessionMessagesSet != null && !sessionMessagesSet.isEmpty()) {
        sessionMessagesSet.forEach(message -> {
            try {
                emitter.send(SseEmitter.event().data(message, MediaType.APPLICATION_JSON));
            } catch (final IOException e) {
                log.error( message: "Error sending session message to session [{}]: {}", sessionId, e.getMessage());
            }
        });
    }

    // Send global messages
    if (!globalMessages.isEmpty()) {
        globalMessages.forEach(message -> {
            try {
                emitter.send(SseEmitter.event().data(message, MediaType.APPLICATION_JSON));
            } catch (final IOException e) {
                log.error( message: "Error sending global message to session [{}]: {}", sessionId, e.getMessage());
            }
        });
    }
}
```

Kuva 11. Metodi, sendSessionMessages, jolla lähetetään viestit käyttäjille

Kirjautumishetkellä rakennetaan lähetettävät ilmoitukset hakemalla muistista sessiotunnisteen perusteella. Tällöin ilmoitus lähetetään käyttäjälle heti. Sama pätee käyttäjän päivittäessä sivun. Kirjoitetaan metodi, joka lähettää sekä yksityiset että kaikille tarkoitetut ilmoitukset (kuva 11).

4.2 Ilmoitusten julkinen puoli

Kun tarvittavat metodit ilmoitusten luomiseen ja lähettämiseen ovat valmiita, seuraava vaihe on rakentaa julkinen polku, jonka kautta käyttäjät muodostavat yhteyden ja vastaanottavat ilmoituksia. Tätä varten luodaan SSEController-Public-luokka, jonka vastuulla on viestien lähettäminen käyttäjille.

```
@RestController
@RequestMapping(value = "/sse")
public class SSEControllerPublic {

    private final SSEService sseService;  6 usages

    public SSEControllerPublic(final SSEService sseService) { this.sseService = sseService; }


    @GetMapping(path = "/stream", produces = "text/event-stream; charset=UTF-8")
    @Authorization(requiredLevel = AccessRight.READ_DOC)
    public SseEmitter streamEvents(final HttpServletRequest request) {
        final String sessionId = request.getSession().getId();
        final SseEmitter emitter = sseService.addEmitter(sessionId, TimeUnit.MINUTES.toMillis( duration: 10));

        emitter.onCompletion(() -> {
            sseService.removeEmitter(sessionId);
        });

        emitter.onTimeout(() -> {
            try {
                emitter.complete();
            } finally {
                sseService.removeEmitter(sessionId);
            }
        });

        emitter.onError((throwable) -> {
            try {
                emitter.completeWithError(throwable);
            } finally {
                sseService.removeEmitter(sessionId);
            }
        });

        // Send stored messages to sessions
        sseService.sendSessionMessages(sessionId, emitter);
        return emitter;
    }
}
```

Kuva 12. SSEControllerPublic-luokka

Julkinen polku määritellään osoitteeseen /sse. Polun käyttöä kuitenkin rajoitetaan vain niille kirjautuneille käyttäjille, joilla on oikeus lukea metatietoja. Tämän avulla ilmoitukset rajoittuvat tiettyihin käyttäjiin, kuten arkistonhoitajille, eikä palvelun vierailijoille. @GetMapping avulla luomme /sse polkua varten stream-osoitteen (kuva 12). Kun käyttäjä muodostaa yhteyden /sse/stream-osoitteeseen, tarkistetaan sessiotietojen kautta käyttäjän oikeudet ja sessiotunnisteen, jonka perusteella ilmoituksia lähetetään. SSE yhteyttä ylläpidetään kymmenen minuuttia, jonka jälkeen yhteys katkaistaan, jos käyttäjä on yhä sivulla, luo se uuden SSE-yhteyden. Aikarajoittimella vältämme turhien yhteyksien ylläpidon, joka vie muistia palvelimelta.

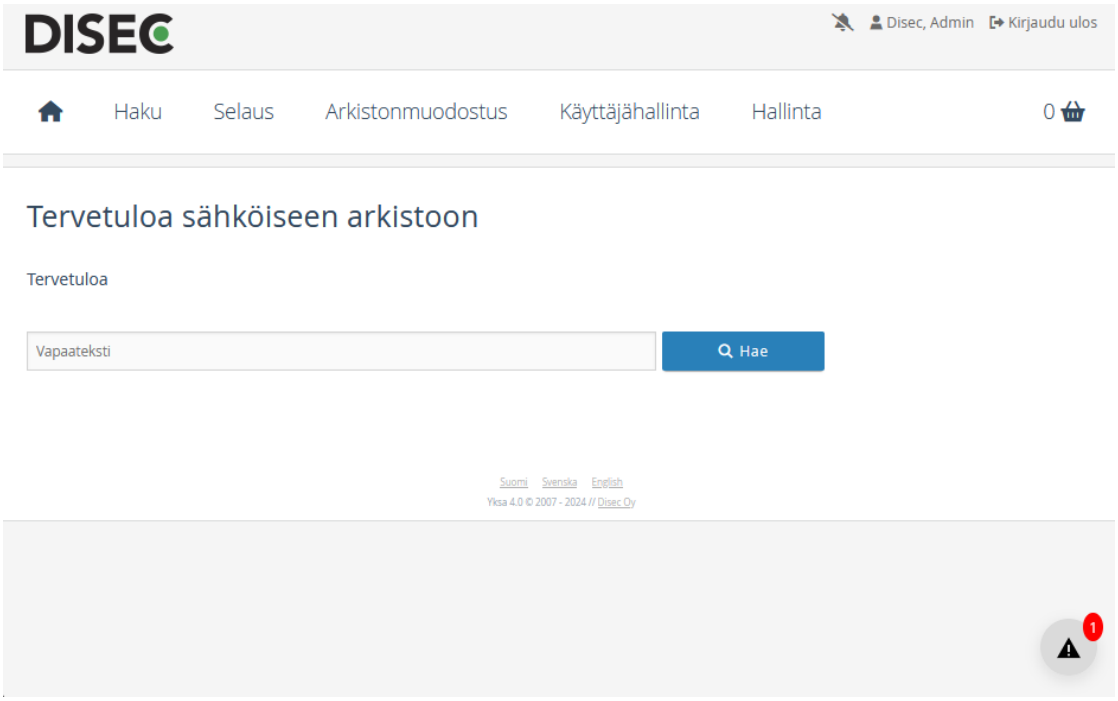
4.3 Ilmoitusten näyttäminen käyttäjille

Yksa-järjestelmässä hyödynnetään Vue.js-ohjelmistokehystä, jolla voidaan luoda komponentteja selaimen käyttöä varten. Ilmoitusten näyttämistä varten kirjoitetaan SseNotification.vue-komponentti.

```
<template>  Show component usages
    <div>
        <!-- Show button logic -->
        <transition name="fade">
            <button v-if="messages.length > 0" class="show-btn" @click="showAlerts">
                <FaIcon icon="triangle-exclamation"/>
                <span class="count-badge" v-if="unreadCount">{{ unreadCount }}</span>
            </button>
        </transition>

        <!-- Transition for messages -->
        <transition-group name="slide-fade" tag="div" class="alert-list">
            <div
                v-for="msg in visibleMessages"
                :key="msg.id"
                :style="{ backgroundColor: msg.color, color: 'black' }"
                class="alert-container"
            >
                <!-- Alert content -->
                <div class="alert-content">
                    <!-- Updated title styling -->
                    <span class="alert-title">{{ msg.title }}</span>
                    <p class="alert-message">{{ msg.message }}</p>
                    <!-- Show countdown if message is visible -->
                    <p v-if="msg.visible && msg.countDown" class="countdown">{{ msg.countdown }}</p>
                </div>
                <!-- Close button in the top-right -->
                <button class="close-btn" @click="minimize(msg.id)">
                    <FaIcon icon="times"/>
                </button>
            </div>
        </transition-group>
    </div>
</template>
```

Kuva 13. SseNotification-komponentin ulkoasu

Komponentti tarkistaa ulkoasun perusteella ja esittää ilmoitusten määrän pienenä lukuna ilmoitusnapin vieressä. Jos uusia ilmoituksia ei ole, ilmoitusnappi pidetään piilossa käyttäjältä. Kun ilmoitus lähetetään käyttäjälle, hän voi painaa ilmoitusnappia tuodakseen ilmoitukset esille.



Kuva 14. Ilmoitusnappi sivuston oikeassa alakulmassa

Ilmoitusten vastaanottamista varten komponentti muodostaa yhteyden stream-osoitteeseen EventSource-objektin avulla, sillä SSE ei ole pelkkä pyyntökutsu vaan pysyvä yhteys. EventSource-objektin käsittelyn helpottamiseksi käytetään UseVue-kirjastoa, joka tarjoaa apuja esimerkiksi yhteyden uudelleenyrityksen hallintaan, yhteyden sulkemiseen ja sen avaamisen useEventSource avulla.

```
// Connect to SSE using useEventSource
const {data, open, close} = useEventSource( url: `${App.getAppContext()}/sse/stream`, events: [], options: {
    autoReconnect: {
        retries: 5,
        delay: 10000
    }
});

// Watch the incoming `data` from SSE
watch(data, cb: eventData => {
    if (eventData) {
        try {
            const parsedData = JSON.parse(eventData);

            // Check if it's a deletion event
            if (parsedData.event === 'delete' && parsedData.id) {
                // Remove the message by its ID
                const index = messages.value.findIndex(msg => msg.id === parsedData.id);
                if (index !== -1) {
                    messages.value.splice(index, deleteCount: 1); // Remove the message from the array

                    cleanUpLocalStorage();
                }
                return;
            }

            if (parsedData.id && parsedData.message && parsedData.color !== undefined) {
                if (receivedMessageIds.value.has(parsedData.id)) return;

                receivedMessageIds.value.add(parsedData.id);
                const newMessage: Message = {
                    id: parsedData.id,
                    title: _.escape(parsedData.title),
                    message: _.escape(parsedData.message),
                    color: _.escape(parsedData.color),
                    date: _.escape(parsedData.endDateTime) || "",
                    countDown: parsedData.countDown,
                    countdown: '',
                    visible: false,
                    isRead: false // Mark as unread initially
                };

                messages.value.push(newMessage);

                // Automatically handle countdown if applicable
                if (newMessage.countDown) {
                    newMessage.countdown = calculateCountdown(newMessage.date);
                }
            } else {
                console.error("Received message does not contain necessary properties.");
            }
        } catch (error) {
            console.error("Error parsing SSE message:", error);
        }
    }
});
```

Kuva 15. Käyttäjän yhteyden luonti ilmoituksia varten

VueUse-kirjasto tuo mukanaan useita etuja, joista käytetään useEventsource-funtiota luomaan SSE-yhteyden. Toisena ominaisuutena käytämme useDocumentVisibility, joka mahdollistaa tavan seurata, onko käyttäjä aktiivisesti sivulla. Tämä on tärkeä erityisesti tilanteissa, joissa on rajattu yhteyksien

määrä, joka ulkoisen pääsynhallintajärjestelmän rajoitusten takia käyttää vain HTTP/1.1 yhteyttä. Tämän rajoitteen takia voidaan avata vain kuusi välilehteä.

Tämän ongelman ratkaisemiseksi useDocumentVisibility-toimintoa käytetään seuraamaan, onko sivu aktiivisesti käytössä. Kun sivu ei ole näkyvissä, voidaan yhteys katkaista ja avata uudelleen, kun käyttäjä palaa sivulle. Tämä vähentää ylimääräisten resurssien käyttöä ja pitää yhteyden määrän hallinnassa.

```
// Use VueUse to track document visibility state
const documentVisibility = useDocumentVisibility();

// Watch for visibility changes
watch(documentVisibility, cb: () => {
    if (documentVisibility.value === 'visible') {
        startCountdownUpdates(); // Start updates if the tab is active
        open();
    } else {
        stopCountdownUpdates(); // Stop updates if the tab is inactive
        close(); // Close SSE connection when tab is inactive
    }
});

onMounted( hook: () => {
    if (documentVisibility.value === 'visible') {
        startCountdownUpdates(); // Start countdown updates initially
    }
```

Kuva 16. Seurataan onko sivu aktiivisesti käytössä vai ei

Toinen VueUse-kirjaston merkittävä hyöty on sen automaattinen muistinhallinta. Ilman VueUse-kirjastoa kehittäjän tulisi itse kirjoittaa koodia, joka varmistaa esimerkiksi document.hidden-tilan seurannan lopettamisen, kun komponenttia ei enää tarvita. Automaattinen muistinhallinta auttaa estämään ylimääräisten prosessien jäämisen taustalle, mikä voisi kuormittaa selainta ja vaikuttaa sovelluksen suorituskykyyn. VueUse käyttö yksinkertaistaa tätä hallintaprosessia ja tarjoaa valmiita ratkaisuja yleisiin ongelmiin, mikä puolestaan parantaa sovelluksen tehokkuutta ja tekee siitä helpommin ylläpidettävän.

4.4 Ilmoitusten luominen

Nyt, kun ilmoitusten vaatimat polut ja logiikka on tehty, tehdään järjestelmänvalvojien puoli, joka on vastuussa ilmoitusten hallinnasta. Tätä varten käytämme eri polkua (kuva 17).

```java
public class SSEController {

    private final SSEService sseService;  10 usages

    public SSEController(final SSEService sseService) {
        this.sseService = sseService;
    }

    /**
     * GET all global messages.
     *
     * @return ResponseEntity containing a list of global messages.
     */
    @GetMapping(value = "/messages/global")
    public ResponseEntity<List<SSEService.MessageData>> getGlobalMessages() {
        final List<SSEService.MessageData> globalMessages = sseService.getGlobalMessages();
        return ResponseEntity.ok(globalMessages);
    }

    /**
     * GET messages by sessionId.
     *
     * @param sessionId the unique identifier for the user session.
     * @return ResponseEntity containing a list of messages associated with the session.
     */
    @GetMapping(value = "/messages/{sessionId}")
    public ResponseEntity<List<SSEService.MessageData>> getMessagesForSession(@PathVariable final String sessionId) {
        final List<SSEService.MessageData> messages = sseService.getMessagesForSession(sessionId);
        return ResponseEntity.ok(messages);
    }
```

Kuva 17. GET-pyynnöt palvelimen ylläpito näkymään

Kuvan yhdeksän mukaan esitetään kaikille tarkoitettujen ilmoitusten luominen ja lähettäminen, täytyy olla myös käyttäjäkohtainen tapa lähettää ilmoituksia. Tätä varten kirjoitetaan sendToUser-metodi, joka hakee sessiolistasta tunnisteen ja lähettää käyttäjälle tarkoitetut ilmoitukset. Kuvassa 18 esitetään kyseisen metodi.

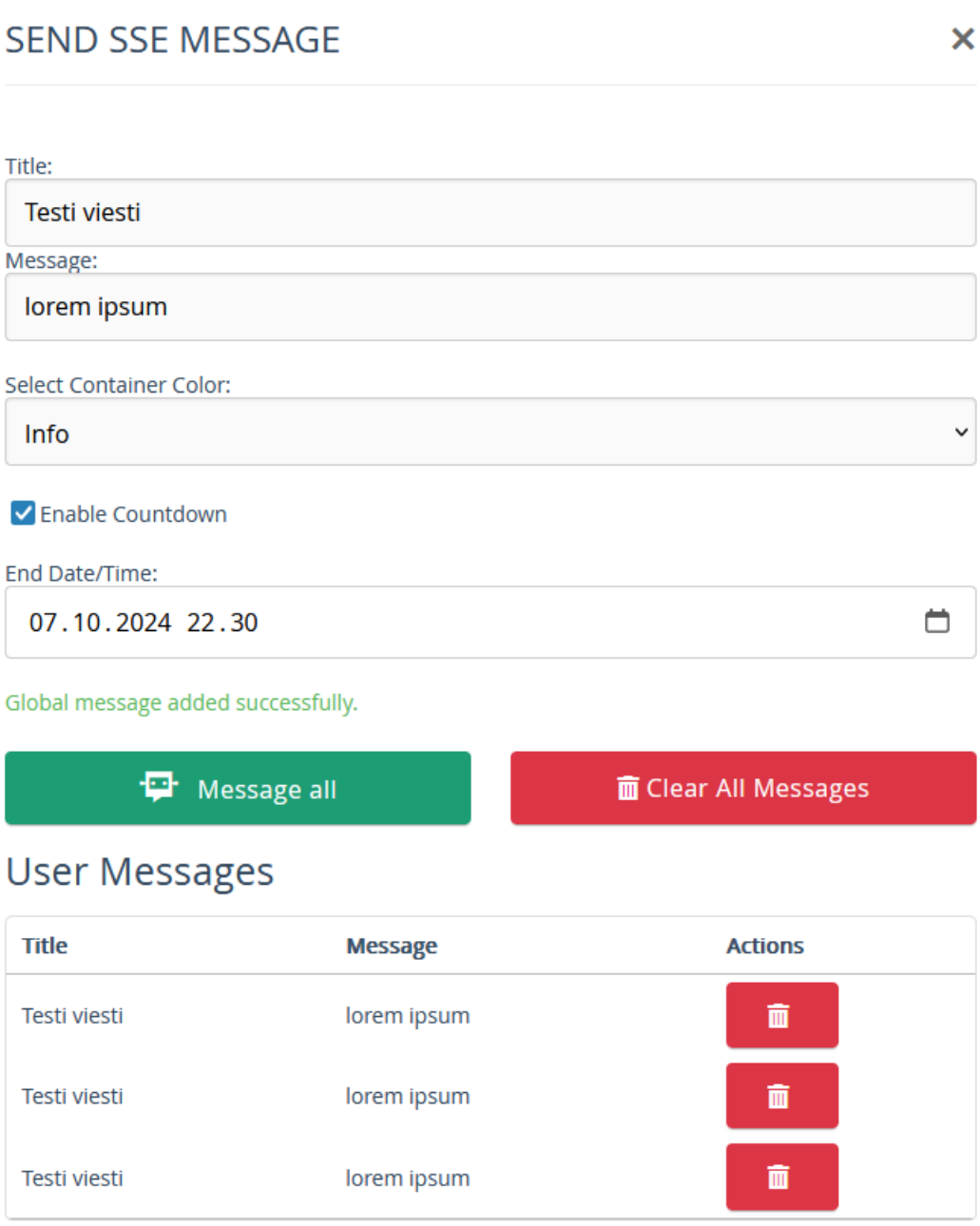
```java
/**
 * Send a message to a specific user session via SSE.
 *
 * @param sessionId    the user session.
 * @param messageData  the message data to be sent to the user.
 */
public void sendToUser(final String sessionId, final MessageData messageData) {  1 usage
    final SseEmitter emitter = emitters.get(sessionId);
    if (emitter != null) {
        try {
            final String newId = UUID.randomUUID().toString();
            // Create a new MessageData with the updated ID
            final MessageData updatedMessageData = new MessageData(
                newId,
                messageData.title(),
                messageData.message(),
                messageData.color(),
                messageData.countDown(),
                messageData.endDateTime()
            );

            final String userSseMessage = createJsonMessage(updatedMessageData);
            emitter.send(SseEmitter.event().data(userSseMessage));
            Thread.sleep( millis: 100);
        } catch (final IOException | InterruptedException e) {
            log.error( message: "Error sending to session [{}]: {}", sessionId, e.getMessage());
            emitter.completeWithError(e);
        }
    }
}
```

Kuva 18. Metodi, jolla lähetetään yksityinen viesti

Hyödyntämällä kuvassa 18 esitettyä sendToUser-metodia yhdessä muiden ilmoituslogiikan komponenttien kanssa, kusen SSEService-luokan kanssa, voidaan vastaa, että viestit toimitetaan käyttäjille nopeasti ja tehokkaasti. Erillinen metodi mahdollistaa ilmoituksien lähettämisen integroimisen helpommin, esimerkiksi kehittämään eri käyttäjärooleille mahdollisuuden ilmoitusten luomiseen ja lähettämiseen julkisella puolella.

4.5 Prototyypin lopputulos

Lopputuloksena on toteutettu prototyyppi, joka mahdollistaa ilmoitusten turvallisen luomisen ja lähettämisen käyttäjille hyödyntäen Spring- ja Vue.js-ohjelmistokehyksiä (framework). Prototyyppi erottaa käyttäjien ja ylläpitäjien toiminnot omiin polkuihinsa (kuva 12 ja kuva 17), mikä turvaa vain valtuutetuille ilmoitusten luomisen, kun taas käyttäjät voivat vastaanottaa ilmoituksia julkisen polun kautta.



Kuva 19. Ylläpitäjien näkymä ilmoituksen luomisesta

Prototyypin keskeiset ominaisuudet muodostuvat keskitetystä SSEService-luokasta, joka toimii ilmoitusten logiikan ytimenä, sekä muistiin perustuvasta tallennusmekanismista. Tallennus erottelee globaalit- ja sessiopohjaiset ilmoitukset, varmistaen samalla viestien yksityisyyden ja turvallisuuden.

Käyttäjäkohtaiset ilmoitukset sekä globaalit ilmoitukset näytetään saman komponentin avulla, joka ilmoittaa käyttäjälle ilmoitusnapin tavoin (kuva 14). Ilmoitusnappi reagoi saapuvaan tietoon esittäen ilmoitusten määrän. Jotta ilmoitus-

ten määrä pysyy muistissa, täytyy tieto tallentaa paikallisesti muistiin hyödyntäen viestin identiteettiä. Tällöin käytetään VueUse-kirjaston useStorage tallentamaan viestin id-arvot, jonka avulla tarkistetaan, onko luettuja ilmoituksia.

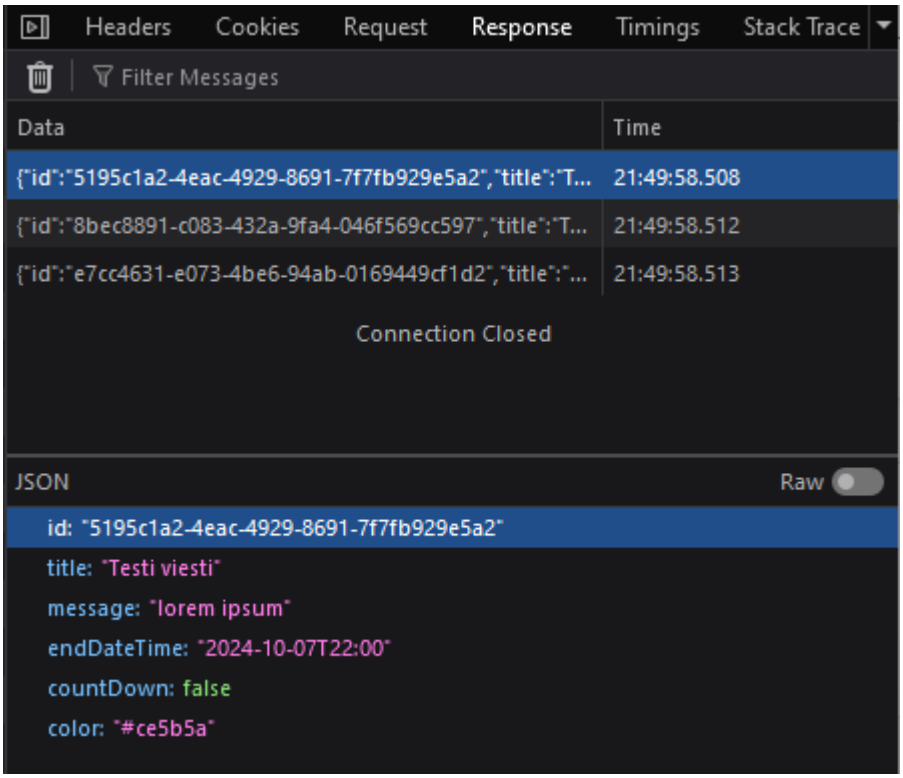
```
// Computed property for unread count
const unreadCount = computed( getter: () =>
    messages.value.filter(msg => !readMessageIds.value.includes(msg.id)).length);
```

Kuva 20. Ei luettujen viestien laskenta määrän esittämistä varten

Tämän jälkeen, kun käyttäjä avaa ilmoitukset osion, tausta-ajona tarkistetaan tallennetut id-arvot verraten vastaanotettuihin viesteihin ja poistetaan ne id:t jotka eivät ole ilmoituksissa.

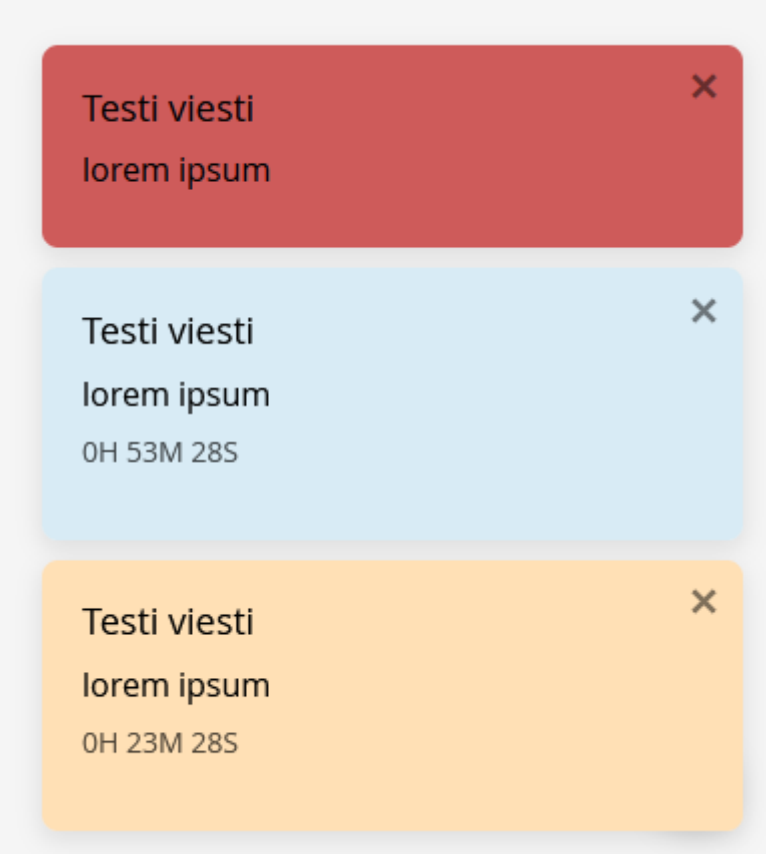
```
// Clean up local storage to remove IDs that don't match current messages
const cleanUpLocalStorage = () => {  Show usages
    const validMessageIds = messages.value.map(msg => msg.id);

    // Remove any IDs from readMessageIds that don't exist in the current messages
    readMessageIds.value = readMessageIds.value.filter(id => validMessageIds.includes(id));
};
```

Kuva 21. Luettujen viestien paikallisen muistin puhdistaminen olemattomista viesti identiteeteistä



Kuva 22. Miltä SSE:n lähettävät ilmoitukset näyttävät Firefox-selaimen kehittäjätyökalujen näkymässä

Painamalla ilmoitusnappia tulee esiin itse ilmoitus, joka esitetään kuvan 23 mukaisena. Ilmoitus sisältää otsikon, tekstin sekä mahdollisen ajastimen. Ilmoituksen viestin värin voi valita myös kolmesta eri määritellystä väristä.



Kuva 23. Ilmoitusten ulkoasu

Tämä värivalikoima auttaa käyttäjiä erottamaan viestin tärkeyden, parantaen käyttökokemusta. Jatkossa värejä voisi laajentaa tai mukauttaa käyttäjän mieltymysten mukaan. Lisäksi ajastintoiminnon avulla käyttäjä näkee, koska esimerkiksi tuotantopäivitys on alkamassa. Tekemällä simppelin prototyypin ulkoasun, on helppo jatkokehittää siitä soveltuvampi.

5 PÄÄTÄNTÖ

Kun tarkastelen valmiin prototyypin jälkeen asetettuja opinnäytetyön tavoitteita, koen, että saavutin niistä melkein kaikki. Opinnäytetyön ydin oli luoda prototyyppi, mikä onnistui hyvin ja avasi paljon jatkokehitysmahdollisuuksia, myös Disecin Yksa-järjestelmän muihin osiin. Opinnäytetyössä tuli myös hyvin raportoitua, mitä Server-Sent Events on ja selittää sen toiminta, mutta WebSocket-tekiikan osalta selvitys jäi omasta mielestä vähäiseksi.

Opinnäytetyön aloitusvaiheessa en ollut vielä perehtynyt syvällisesti Java-ohjelmointikieleen tai Spring-ohjelmistokehykseen (framework), mutta aikaisempi kokemus web-kehityksestä JavaScriptin kanssa auttoi paljon prototyypin ulkoasun kehittämisessä, sillä Vue on JavaScript-pohjainen. Koska Java on hyvin suosittu ja Spring-ohjelmistokehys on laajasti käytetty, löysin helposti avustavaa tietoa, joka tuki prototyypin kehitystä. Tämä oli minulle hyvin antoisa tilaisuus oppia uutta Javasta ja palvelimen ja käyttäjän välisestä yhteydestä.

Prototyypin kehitykseen aikataulullisesti riittävästi aikaa ja suurin osa aika kului ratkomaan Server-Sent Events toiminnallisuutta ja kirjoittaa helposti jatkokehitettävää ja ylläpidettävää koodia. Prototyypin aikataulutuksessa tuli haasteelliseksi pitää mielessä, että kyseessä on prototyyppi, ei valmis tuote. Tämä tarkoittaa, että saattoi olla hetkiä, jolloin keskityin liikaa ulkoasun kehittämiseen tai kuinka kirjoittaisin mahdollisimman hyvää koodia. Aikatauluun ei myöskään jäänyt prototyypin syvälliseen testaamiseen testausympäristössä, jossa olisi ollut muita käyttäjiä, vaan ennemmin paikallisessa ympäristössä.

Lopputuloksesta olen tyytyväinen ja prototyypin jatkokehityspotentiaali on iso.
Palvelin puoli:

- Ajastettujen ilmoitusten automaattinen poisto
- Ilmoitusten tallentaminen tietokantaan muistin sijaan
- Ilmoitusten muokkaaminen
- Ilmoitusten lähettäminen roolin tai palvelu asiakkaan perusteella

Käyttäjä puoli:

- Luettujen ilmoitusten muistaminen
- Ajastetun ilmoituksen piilottaminen
- yhdistää aikaisemman ilmoitus systeemiin tehden yhden ilmoitus komponentin

Kokonaisuudessaan prototyyppi on edistänyt minua paremmaksi Java-kehittäjäksi tarjoten riittävästi haastetta sekä kiinnostusta reaaliaikaista viestintää kohtaan ja sen jatkosoveltamiseen. Kiitos vielä Disecille mahdollisuudesta prototyypin kehittämisestä, jota tullaan käyttämään myös Yksa-järjestelmässä tulevaisuudessa.

LÄHTEET

Bidelman, E. 2010. Stream updates with server-sent events. WWW-dokumentti. Päivitetty 30.11.2010. Saatavissa: https://web.dev/articles/eventsource-basics [viitattu 8.7.2024]

Cloudflare s.a. What is HTTP? Why is HTTP/2 faster than HTTP/1.1? WWW-dokumentti. Saatavissa: https://www.cloudflare.com/learning/performance/http2-vs-http1.1/ [viitattu 14.7.2024]

Cook, D. 2014. Data Push Apps with HTML5 SSE. O'Reilly Media, Inc. E-kirja. Saatavissa: https://www.oreilly.com/library/view/data-push-apps/9781449371920 [viitattu 3.7.2024]

Fette, I. & Melnikov A. 2011. The WebSocket Protocol. WWW-dokumentti. Saatavissa: https://datatracker.ietf.org/doc/html/rfc6455 [viitattu 22.9.2024]

Ger, R. 2024. Server-Sent Events in Spring. WWW-dokumentti. Päivitetty 12.6.2024. Saatavissa: https://www.baeldung.com/spring-server-sent-events [Viitattu 3.7.2024]

Mozilla. s.a. WebSockets. WWW-dokumentti. Päivitetty 8.6.2023 Saatavissa: https://developer.mozilla.org/en-US/docs/Glossary/WebSockets [viitattu 3.7.2024]

Mozilla. 2024c. Using server-sent events. WWW-dokumentti. Päivitetty 26.7.2024. Saatavissa: https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events [viitattu 19.9.2024]

Mozilla. 2024a. BroadcastChannel. WWW-dokumentti. Päivitetty 3.6.2024. Saatavissa: https://developer.mozilla.org/en-US/docs/Web/API/BroadcastChannel [19.9.2024]

Mozilla. 2024b. Evolution of http. WWW-dokumentti. Päivitetty 8.10.2024. Saatavissa: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP [viitattu 22.7.2024]