Sami Lybeck

Asynkronisen REST-ohjelmointirajapinnan käyttö pitkään kestävässä tiedonhaussa

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

9.1.2024

Tiivistelmä

Tekijä:	Sami Lybeck
Otsikko:	Asynkronisen REST-ohjelmointirajapinnan käyttö pitkään kestävässä tiedonhaussa
Sivumäärä:	54 sivua + 1 liitettä
Aika:	9.1.2024
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Ohjelmistotuotanto
Ohjaajat:	Lehtori Simo Silander Tiimipäällikkö Markus Saukkonen

---

Insinöörityössä selvitettiin mahdollisuutta hyödyntää asynkronista API:a Visma Fivaldi -taloushallinto-ohjelmistossa.

Tavoitteena oli toteuttaa reaaliaikaista edistymistä visualisoiva latauspalkki Fivaldin Kiinteistönhallinta-sovelluksen taloyhtiön alustusvelhon käyttöön. Alustusvelhon noutaman tiedon hakuun kului paljon aikaa, mikä aiheutti käyttäjän turhautumista. Latauspalkin oli tarkoitus korvata aiemmin tiedonhaun aikana pyörinyt spinneri, joka ei kuvastanut latauksen edistymistä mitenkään.

Työ alkoi selvittämällä erilaisia asynkronisia tekniikoita, joilla latauspalkki voitaisiin toteuttaa. Selvityksen jälkeen kirjoitettiin palvelinpuolen sekä asiakaspuolen ohjelmakoodi.

Tuloksena saatiin Kiinteistönhallinta-sovelluksen alustusvelhon tiedonhaun ajaksi reaaliaikaista haun etenemistä visualisoiva latauspalkki. Latauspalkilla pystyttiin korvaamaan aiemmin tiedon haun aikana pyörinyt spinneri. Asynkronisten rajapintojen toteuttamisen lisäksi insinöörityössä todettiin yhtäaikaisesti ajettavan koodin nopeuttavan aikaa vieviä prosesseja.

Latauspalkin lisäksi asynkronisella API:lla ja yhtäaikaisesti ajettavalla koodilla toteutettava ohjelma todettiin hyödylliseksi myös muissa aikaa vievissä prosesseissa. Tällaisia prosesseja voivat olla Fivaldiin kehitteillä olevien widgettien tarvitseman tiedon haku. Näin käyttäjällä voisi olla pääsy widgetissä olevan ensimmäisen sivun dataan nopeasti sillä välin, kun muut sivut latautuvat taustalla.

Avainsanat:	asynkronisuus, REST-API, rajapinta, latauspalkki, edistymispalkki, moniajo, monisäikeisyys

Abstract

Author:	Sami Lybeck
Title:	Using Asynchronous REST API for Long-Lasting Data Retrieval
Number of Pages:	54 pages + 1 appendices
Date:	9 January 2024
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Software Engineering
Supervisors:	Simo Silander, Senior Lecturer Markus Saukkonen, Team Leader

---

The engineering thesis investigates, the possibility of utilizing an asynchronous API in the Visma Fivaldi financial management software.

The goal was to implement a real-time progress visualizing loading bar for the initialization wizard of Fivaldi's Property Management application. The data retrieval performed by the initialization wizard took a considerable amount of time, leading to user frustration. The loading bar was intended to replace the previously used spinner during data retrieval, which did not effectively reflect the progress of the loading process.

The study began by exploring various asynchronous techniques for implementing the loading bar. After the investigation, both server-side and client-side code were written.

As a result, a real-time loading bar visualizing the progress of data retrieval during the initialization wizard of the Property Management application was obtained. The loading bar successfully replaced the spinner used during data retrieval. In addition to implementing asynchronous API, the study provided insights into how concurrently executed code can expedite time-consuming processes.

Beyond the loading bar, the program implemented with asynchronous API and concurrently executed code was found to be useful in other time-consuming processes. Such processes could include retrieving data required for widgets under development in Fivaldi. This way, users could quickly access the data on the first page of a widget while other pages load in the background.

Keywords: asynchrony, REST API, interface, loading bar, progress bar, concurrent execution, multithreading

Sisällys

Lyhenteet

1 Johdanto....1

2 Fivaldi-taloushallinto-ohjelma....4

2.1 Käyttäjäryhmät ja ominaisuudet....4
2.2 Huoneistotietojärjestelmä ja taloyhtiön alustusvelho....5
2.3 Tekninen kuvaus....7
2.4 Ohjelmistokehys....8
2.5 Angular ja TypeScript....10
2.6 Java ja servletit....11
2.7 Spring Framework -ohjelmistokehys....13
2.8 Ohjelmointiympäristö ja työkalut....15

3 Teknologioiden vertailu....17

3.1 Teknologian valinta....17
3.2 Synkroninen vs asynkroninen API-kutsu....22

4 Palvelinpuolen ohjelman kirjoitus....24

4.1 Haasteet....24
4.2 Monisäikeistetty koodi ja säikeiden turvallisuus....25
4.3 Producer-Consumer-suunnittelumalli....27
4.4 Tilalliset luokat ja HtjWizardContextFactory....33
4.5 Kontrollerikerroksen koodi....35
4.6 Palvelukerroksen koodi....38

5 Asiakaspuolen ohjelman kirjoitus....43

5.1 Tietovirran vastaanottaminen ja muunnos asiakaspuolella....43
5.2 Tiedon käsittely ja visualisointi....45

6 Yhteenveto....48

Lähteet..... 50

Liitteet

Liite 1: Yksinkertaistettu sekvenssikaavio tiedon käsittelystä ja suoratoistosta asynkronisesti käyttöliittymään latauspalkkia varten

Lyhenteet

SPA: *Single Page Application.* Web-sovellustyyli, joka lataa vain yhden HTML-sivun ja päivittää dynaamisesti sisältöä ilman sivun uudelleenlatausta.

JSON: *JavaScript Object Notation.* Kevyt tiedonvaihtoformaatti, joka käyttää inhimillisesti luettavaa tekstiä.

URL: *Uniform Resource Locator.* Tietyn resurssin sijainti verkkotunnisteella, esimerkiksi verkkosivun osoite.

URN: *Uniform Resource Name.* Yksilöi resurssin pysyvällä tavalla, ei välttämättä osoita sen sijaintia verkossa.

URI: *Uniform Resource Identifier.* Merkkijono, joka yksilöi resurssin ainutlaatuisesti verkossa, voi olla URL tai URN.

HTTP: *Hypertext Transfer Protoco*l on protokolla web-sivujen tietojen siirtämiseen selaimen ja palvelimen välillä.

REST: *Representational State Transfer.* Verkkopalveluiden suunnittelutyyli, joka perustuu resurssien yksilöimiseen URI:lla ja standardoituihin operaatioihin.

API: *Application Programming Interface*. Rajapinta, joka mahdollistaa eri ohjelmistojen tai sovellusten vuorovaikutuksen ja integroinnin toistensa kanssa.

FIFO: *First In, First Out.* Jononhallinnan periaate, jossa ensimmäisenä saapunut elementti käsitellään ensimmäisenä.

MVC: *Model-View-Controller.* Ohjelmistoarkkitehtuuri, jossa sovellus jaetaan kolmeen osaan. Nämä osat ovat malli (model), näkymä (view) ja käsittelijä tai kontrolleri (controller).

1 Johdanto

Tässä opinnäytetyössä käsitellään asynkronisen REST-ohjelmointirajapinnan hyödyntämistä Visma Software Oy:n Fivaldi-taloushallinto-ohjelman käyttötarpeisiin.

Työ alkoi tutkimalla, pystyttäisiinkö Visma Fivaldi -taloushallinto-ohjelmaan toteuttamaan asynkronista REST-ohjelmointirajapintaa hyödyntäen edistymispalkki (determinate progress bar), joka indikoisi palvelimen prosessin etenemistä reaaliaikaisesti graafisessa web-käyttöliittymässä (selain). Tämäntyyppistä tiedon lähetystä palvelimelta selaimeen kutsutaan suoratoistoksi (streaming).

Monelle tuttu esimerkki suoratoistosta on esimerkiksi suoratoistopalvelu Netflixin tarjoamat elokuvat ja sarjat. Toinen esimerkki suoratoistosta on tiedoston lataamisessa selaimeen ilmaantuva latauspalkki, joka tarjoaa reaaliaikaista tilannekuvaa siitä, kuinka suuri tiedosto on kyseessä, paljonko siitä on milläkin ajanhetkellä ladattu laitteelle ja arvio siitä, kauanko latauksen saattaminen loppuun kestää.

Työ koostui myös sopivien teknologioiden valinnoista, palvelinpuolen ohjelmoinnista käsitellä ja lähettää dataa asynkronisesti asiakaspuolelle sekä ohjelmoida asiakaspuoli vastaanottamaan ja visualisoimaan prosessin edistymistä. Edistymispalkin tarkoituksena oli visualisoida prosessia, joka liittyy Kiinteistöjenhallinta-sovelluksen taloyhtiön alustusvelhon käyttöön.

Kiinteistöjenhallinta-sovellus on yksi monista sovelluksista Fivaldi-taloudenhallintaohjelmassa. Sitä käyttävät isännöitsijät ylläpitävät taloyhtiöiden perustietoja, kuten asukkaiden nimiä, osoitteita ja hallintaoikeuksia osakeryhmien muodossa.

Taloyhtiön alustusvelho on luotu helpottamaan isännöitsijän työtä. Sen avulla isännöitsijä saa hallinnoimansa taloyhtiön tiedot osakeryhmittäin Maanmittaus-

laitoksen huoneistotietojärjestelmästä (HTJ), mikä poistaa tarpeen niiden manuaaliselle kirjaamiselle.

Taloyhtiön tietojen hakeminen huoneistotietojärjestelmästä saattaa kuitenkin kestää useita minuutteja, mikä riippuu haettavien osakeryhmien määrästä. Tietojen haun aikana käyttöliittymässä pyörii kuvan 1 esittelemä spinneri (indeterminate progress spinner). Spinneri ei ilmaise prosessin etenemistä tai haettavien tietojen kokonaismäärää, minkä uskotaan aiheuttavan käyttäjälle turhautumista ja mahdollisesti sivulta poistumista.

Kuva 1: Kiinteistönhallinta-sovelluksen alustusvelhossa osakeryhmien tietojen haun aikana pyörivä spinneri, joka ei indikoi latauksen etenemistä.

Työn tavoitteena oli luoda uusi edistymispalkki, joka näyttää käyttäjälle tiedon huoneistotietojärjestelmästä osakeryhmittäin haettavien tietojen kokonaismäärän sekä monenko osakeryhmän tiedot on kullakin ajanhetkellä haettu. Tällaisen prosessin visualisointi edistymispalkilla pitkään kestävässä tiedonhaussa tarjoaa käyttäjälle konkreettisen todisteen ohjelman toiminnasta.

Pitkällä aikavälillä opinnäytetyön tutkimustuloksia ja koodia voidaan hyödyntää myös muissa Fivaldin edistymispalkkia kaipaavissa järjestelmissä korvaamaan

spinnerit. Asynkronista REST-ohjelmointirajapintateknologiaa tullaan myös hyödyntämään mahdollisesti tulevissa projekteissa, joissa suurien tietomäärien lähettäminen palvelimelta selaimeen pienissä erissä on järkevämpää kuin yhden aikaa vievän ja kerralla suuren tietomäärän. Eräs tällainen mahdollinen projekti on käyttöliittymään lisättävien widgettien tarvitsema tiedonhaku.

2 Fivaldi-taloushallinto-ohjelma

Visma Fivaldi on pilvipohjainen ohjelmisto, mikä tarkoittaa, että se on saatavilla verkkoselaimen kautta ilman tarvetta asentaa ohjelmaa paikallisesti. Käyttäjät voivat kirjautua järjestelmään omilla tunnuksillaan ja käyttää tarvitsemiaan ominaisuuksia. Ohjelma tarjoaa käyttäjilleen helppokäyttöisen käyttöliittymän ja tarvittavat työkalut taloushallinnon tehtävien suorittamiseen. Visma Fivaldi mahdollistaa myös yhteistyön tilitoimistojen ja asiakkaiden välillä, mikä tekee taloushallinnon hoitamisesta vaivatonta ja tehokasta.

2.1 Käyttäjäryhmät ja ominaisuudet

Visma Fivaldi on kattava taloushallinnon ohjelmisto, joka tarjoaa monipuolisen valikoiman ominaisuuksia ja ratkaisuja pk-yrityksille, tilitoimistoille, vuokrataloyhtiöille ja isännöitsijöille. Se on suunniteltu palvelemaan useita eri käyttäjäryhmiä, mukaan lukien:

Pienet ja keskisuuret yritykset (pk-yritykset): Pk-yritykset voivat hyödyntää Visma Fivaldia kirjanpitoon, myyntiin, ostoihin, palkanlaskentaan ja muihin taloushallinnon tarpeisiinsa.

Tilitoimistot: Tilitoimistot voivat käyttää ohjelmaa asiakkaidensa taloushallinnon hallintaan ja tarjota heille ammattimaista kirjanpitoa ja neuvontaa.

Vuokrataloyhtiöt ja isännöitsijät: Kiinteistönhallintaan ja taloushallintoon erikoistuneet organisaatiot voivat hyödyntää ohjelman kiinteistönhallinta- ja maksuliikenneominaisuuksia.

Visma Fivaldi tarjoaa laajan valikoiman taloushallinnon ominaisuuksia, kuten:

Kirjanpito: Ohjelma mahdollistaa tarkan kirjanpidon ylläpidon ja raportoinnin.

Myynti ja ostot: Ohjelmassa voit hallita myynti- ja ostolaskuja sekä seurata maksuja ja velkoja.

Palkanlaskenta: Palkanlaskentaominaisuus helpottaa palkanmaksun hallintaa.

Maksuliikenne: Ohjelmassa voi myös hoitaa maksuliikennettä sujuvasti ja varmistaa maksujen ajantasaisuuden.

Logistiikka ja varastohallinta: Ohjelma sisältää työkaluja logistiikkaan ja varastonhallintaan.

Kiinteistönhallinta: Kiinteistönhallintaan tarkoitetut työkalut auttavat vuokrataloyhtiöitä ja isännöitsijöitä hallinnoimaan kiinteistöjä ja vuokrasopimuksia. Eräs sovelluksen työkaluista on taloyhtiön alustusvelho.

Rekisterit: Asiakas- ja toimittajarekistereiden ylläpito sekä muita tärkeitä tietoja.

2.2 Huoneistotietojärjestelmä ja taloyhtiön alustusvelho

Huoneistotietojärjestelmä on Maanmittauslaitoksen ylläpitämä sähköinen rekisteri, joka on otettu käyttöön vuonna 2019. Sinne on siirretty Suomen noin 90 000 taloyhtiötä ja 1,5 miljoonaa osakehuoneistoa. Huoneistotietojärjestelmästä on säädetty laissa ja asunto-osakeyhtiöiden tulee siirtää osakeluettelonsa sinne tietyn siirtymäajan sisällä.

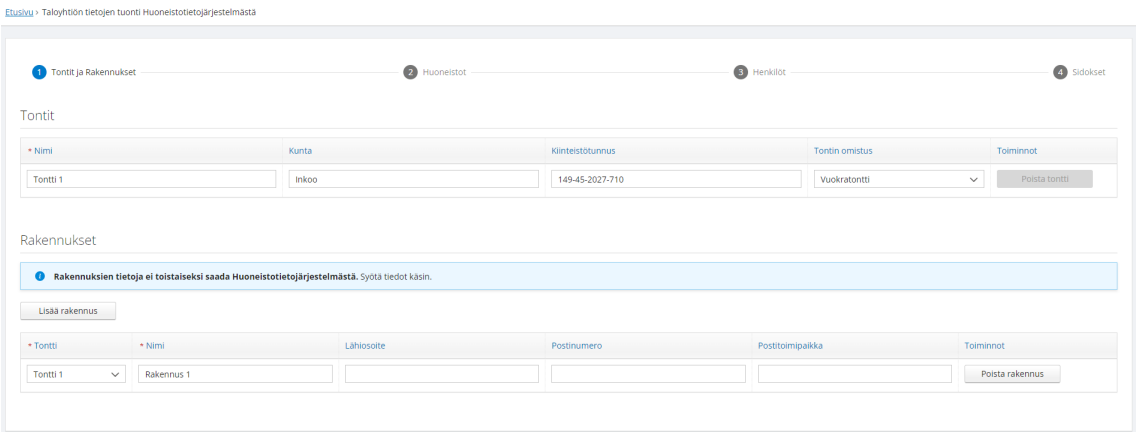
Järjestelmään kerätään tiedot osakehuoneistojen, kuten asuntojen ja autopaikkojen, omistuksista, panttauksista ja rajoituksista. Sen tarkoituksena on korvata paperiset osakekirjat ja näin ollen helpottaa monien eri toimijoiden, kuten taloyhtiön isännöitsijöiden, työtä. (1.)

Isännöintitoimistot voivat suorittaa kyselyjä Maanmittauslaitoksen rajapintapalveluihin (REST), jolloin he saavat sovellukseensa reaaliaikaisia huoneistotietoja. Kuvassa 2 on esitetty Fivaldin Kiinteistönhallinta-sovelluksen taloyhtiön alus-

tusvelho, jossa huoneistotiedot on haettu Maanmittauslaitoksen rajapinnasta osakeryhmittäin.

Osakeryhmällä tarkoitetaan yksittäistä tai useampaa osaketta, joissa osakkeet on yksilöity järjestysnumeroin, esimerkiksi 1-100. Yhdessä nämä osakkeet muodostavat kokonaisuuden, jotka tuottavat hallintaoikeuden tiettyyn osakehuoneistoon. Osakeryhmät tunnistetaan osakeryhmätunnuksilla. Osakeryhmätunnuksilla voidaan hakea huoneistotietojärjestelmän rajapinnasta osakeryhmien tietoja, kuten huoneiston omistajien henkilötietoja ja huoneistonumeroita. (2.) Rajapinnan palauttaman osakeryhmän tiedot kartoitetaan (mapping) Fivaldissa sopivaan muotoon, jolloin Kiinteistönhallinnan taloyhtiön alustusvelhon on valmis otettavaksi käyttöön.

Taloyhtiön alustusvelho on työkalu, joka auttaa isännöitsijää siirtämään taloyhtiön tiedot Fivaldiin. Kun isännöitsijä käynnistää alustusvelhon, se aloittaa huoneistotietojen haun Maanmittauslaitoksen huoneistotietojärjestelmästä.



Kuva 2: Kiinteistönhallinnan taloyhtiön alustusvelho käyttöliittymässä huoneistotietojärjestelmästä saatujen tietojen jälkeen.

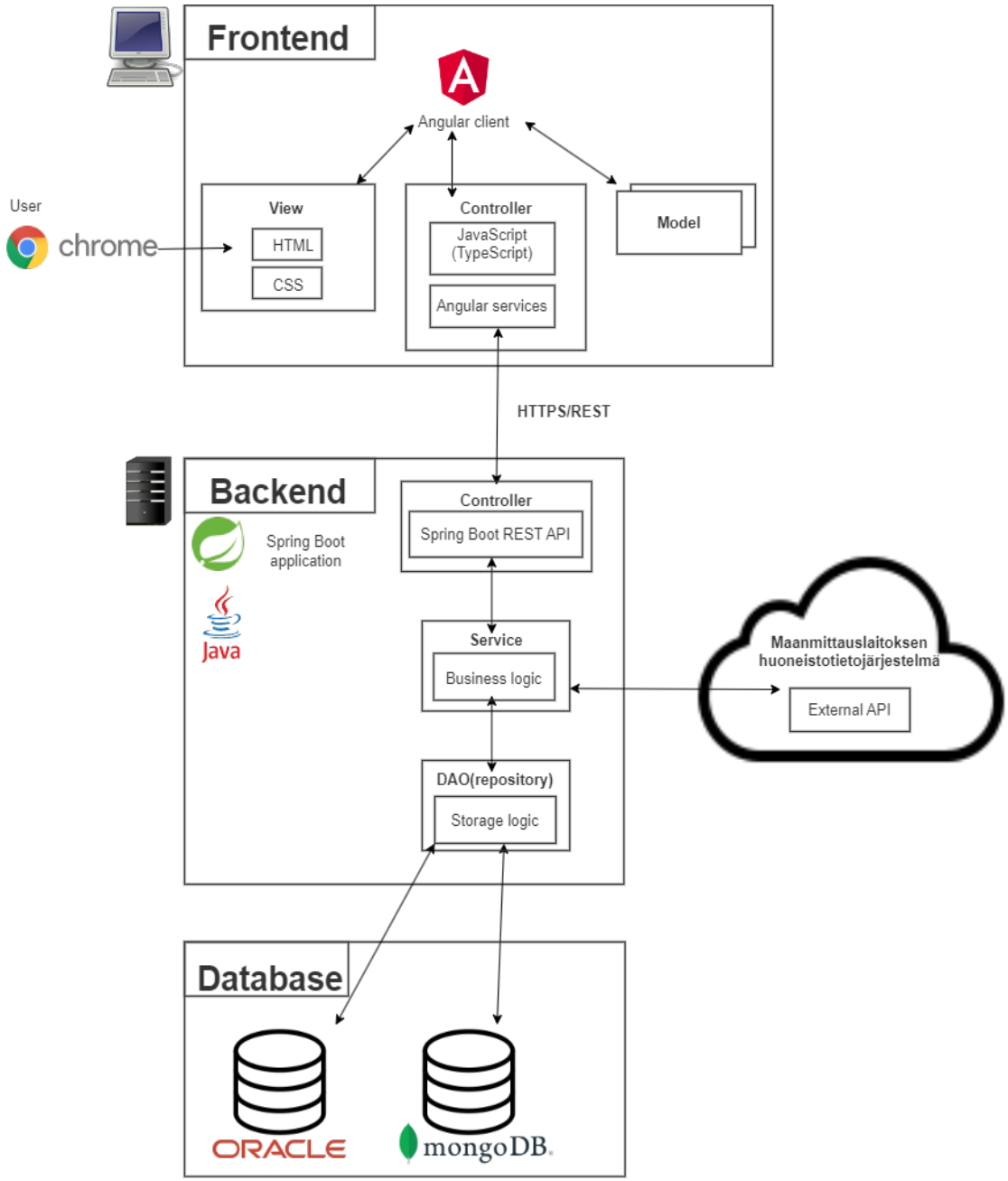
Kuvassa 2 nähdään käyttöliittymään avautuva alustusvelho tietojen haun jälkeen. Alustusvelhossa käyttäjä voi tarkastella ja muokata kiinteistöjen osakeryhmiin liittyviä tietoja askel askeleelta. Kun käyttäjä on edennyt alustusvelhon loppuun, taloyhtiön tiedot voidaan tallentaa Fivaldiin.

2.3 Tekninen kuvaus

Visma Fivaldi on varttuneempi ohjelmisto, joka on ollut käytössä yli 20 vuoden ajan ja siinä onkin nähtävissä usean eri sukupolven teknologioita. Alkuperäiset sovellukset rakennettiin Oracle Forms -teknologialla, jota voidaan nykyään pitää jo melko vanhentuneena. Osaa näistä Oracle Forms -perustaisista sovelluksista on myöhemmin päivitetty käyttämään Javaa. Se säilyttää kuitenkin samankaltaisen syntaksin kuin Oracle Forms -sovellukset. Näitä Oracle Formsin Java-alustalle siirrettyjä sovelluksia kutsutaan nimellä Re_Forms 21.

Valtaosa Visma Fivaldin sovelluksista on kuitenkin refaktoroitu hyödyntämään uudempia teknologioita. Kaikki uudet ominaisuudet kehitetäänkin käyttäen moderneja teknologioita. Tarkemmin määriteltynä palvelinpuolen toteutuksissa käytetään Spring Boot -työkalua yhdessä Javan kanssa, kun taas asiakaspuolen toteutuksissa hyödynnetään Angular-ohjelmistokehystä sekä TypeScript-ohjelmointikieltä. Näihin ohjelmistokehyksiin ja kieliin perehdytään tarkemmin seuraavissa luvuissa.

Kuvassa 3 on esitys insinöörityöhön liittyvistä Visma Fivaldi -taloushallinto-ohjelman osista korkean tason järjestelmäarkkitehtuurikaavion muodossa.



Kuva 3: Järjestelmän arkkitehtuurikaavio

2.4 Ohjelmistokehys

Fivaldin teknisten ratkaisujen taustalla hyödynnetään useita eri kirjastoja, työkalupakkeja ja ohjelmistokehyksiä. Tässä luvussa selvennetään näitä käsitteitä, jotta lukija saa käsityksen siitä, mitä nämä teknologiat ovat ja kuinka ne muodostavat Fivaldin arkkitehtuurin ytimen vaikuttaen järjestelmän toimintaan.

Termit "kirjasto", "ohjelmistokehys" ja "työkalupakki" tai "työkalu" voivat joskus sekoittua, ja niiden määrittely ei ole aina täysin yksiselitteistä. Seuraavaksi käsitellään, miten kirjastot, työkalut ja ohjelmistokehykset eroavat toisistaan, ja tarkastellaan niitä, jotka soveltuvat erityisen hyvin asynkronisten API:en kehittämiseen.

Kirjasto (library) on kokoelma valmiiksi kirjoitettuja luokkia ja funktioita, joita kehittäjät voivat käyttää helpottamaan tehtävien tai toimintojen suorittamista omissa ohjelmistoissaan. Ohjelmistotuotteessa voi olla useita riippuvuuksia eri kirjastoihin. Yksi tällainen kirjasto, joka helpottaa asynkronisten API:en rakentamista palvelinpuolella on Akka Streams. Se on reaktiivinen tietovirtojen käsittelyyn tehty kirjasto. Akka Streamsin lisäksi Akka HTTP on monipuolinen työkalu palvelin- ja asiakaspuolelle. Se on rakennettu tiedon suoratoistoa varten. Sen avulla HTTP-pyyntöjen ja -vastauksien rungot (body) voidaan suoratoistaa palvelimen kautta, jolloin saavutetaan jatkuvaa muistin käyttöä myös erittäin suurille pyynnöille ja vastauksille. Akka HTTP yhdessä Akka Streams -kirjaston kanssa mahdollistavat HTTP-pyyntöjen ja -vastausten käsittelyn reaktiivisella ja suoratoistopainotteisella tavalla. (3.)

Työkalupakille (toolkit) ei ole olemassa yhtä virallista määritelmää, vaan se riippuu asian kontekstista. Yleensä työkalupakilla tarkoitetaan kuitenkin joukkoa erilaisia työkaluja. Työkalu (tool) on apuväline, joka auttaa kehittäjää tiettyjen tehtävien suorittamisessa, ja ne voivat olla esimerkiksi ohjelmistoja, komentosarjoja, dokumentaatiota tai graafisen käyttöliittymän komponentteja. Työkalut voivat liittyä esimerkiksi koodin rakentamiseen (build) testaukseen tai versionhallintaan. (4.) Esimerkiksi Maven ja Gradle ovat rakennustyökaluja, jotka auttavat projektin kirjastojen riippuvuuksien hallinnassa ja sovelluksen kääntämisessä. Fivaldi-ohjelmistossa käytetään Gradlea. Yleensä yhdistetään useita työkaluja ohjelman kirjoitukseen. Esimerkki työkalupakista, joka helpottaa asynkronisten API:en luontia, on AsyncAPI. Se tarjoaa avoimen lähdekoodin työkaluja asynkronisten API:en määrittämiseen sekä dokumenttien ja koodin generointiin.

Ohjelmistokehys tai sovelluskehys (Software Framework), on ohjelmistotuote, joka muodostaa perustan tietokoneohjelman kehittämiselle. Kehys tarjoaa valmiiksi rakennettuja osia ja työkaluja, jotka helpottavat uuden ohjelmistotuotteen luomista. Sen tavoitteena on nopeuttaa ohjelmistokehitystä ja vähentää boilerplatea eli samoja koodin paloja, jotka toistuvat eri puolilla ohjelmaa.

Ohjelmistokehys ei yleensä toimi itsenäisenä suoritettavana ohjelmana, vaan se tarjoaa perustan, jonka päälle kehittäjät rakentavat lopullisen sovelluksen. Kehys sisältää valmiiksi määriteltyjä osia, kuten toiminnallisuuksia, komponentteja ja arkkitehtuurisia piirteitä, jotka voivat olla yleisesti käytettyjä tai standardoituja.

Kun kehittäjät käyttävät ohjelmistokehystä, voidaan keskittyä enemmän sovelluslogiikan kehittämiseen, sillä osa perusinfrastruktuurista ja toiminnallisuuksista on jo käytettävissä. Lopullinen toimiva tuote saadaan rakentamalla uusi ohjelma kehyksen päälle, mukauttamalla tarvittaessa ja laajentamalla kehyksen tarjoamia valmiita ratkaisuja. (5.)

Yksi tärkeimpiä eroja ohjelmistokehyksen ja kirjaston välillä on Inversion of Control (IoC). IoC on suunnittelumalli, jossa perinteinen ohjauksen suunta vaihtuu. Sen sijaan, että sovellus ohjaisi itse komponenttiensa toimintaa, IoC antaa ohjauksen kontrollerille tai säiliölle (IoC-container). Yksinkertaistettuna, ohjelmoijan kirjoittama koodi kutsuu kirjastoja, mutta ohjauksen käännöksessä kehys kutsuu ohjelmoijan kirjoittamaa koodia. (6.)

2.5 Angular ja TypeScript

Uudet asiakaspuolen sovellukset Fivaldiin on kehitetty hyödyntäen Angular-sovelluskehystä. Se on avoimen lähdekoodin web-sovelluskehys, joka tarjoaa selkeän ja organisoituneen kehitysympäristön. Tämä helpottaa sovellusten suunnittelua ja toteutusta. Angular on suunniteltu erityisesti yksisivuisten sovellusten (SPA) kehittämiseen.

TypeScript on Angularin virallinen ohjelmointikieli, joka tuo mukanaan staattisen tyypityksen ja modernin JavaScriptin ominaisuudet, mikä mahdollistaa virheiden havaitsemisen jo kehitysvaiheessa ja parantaa siten koodin turvallisuutta.

Angular-kääntäjä, jota kutsutaan myös termillä "ngc", on työkalu, jota käytetään Angular-sovellusten ja kirjastojen kääntämiseen. Ngc perustuu TypeScript-kääntäjään (tsc) ja laajentaa TypeScript-koodin käännösprosessia lisäämällä Angularin ominaisuuksiin liittyvää koodin generointia. Koska selaimet tukevat vain JavaScriptiä, TypeScript on käännettävä JavaScriptiksi tsc-kääntäjällä, jotta se voidaan suorittaa suoraan selaimessa. (7.)

2.6 Java ja servletit

Java on teknologiaperhe ja ohjelmistoalusta, joka kehitettiin alun perin Sun Microsystemsin toimesta. Tähän teknologiaan kuuluu korkean tason ohjelmointikieli, joka on luokkapohjainen ja oliopohjainen. Yksi sen keskeisistä tavoitteista on minimoida toteutusriippuvuus, mikä tarkoittaa, että Java-sovelluksia voidaan suorittaa eri tietokonejärjestelmissä. Lisäksi siihen kuuluu ajoaikainen ympäristö virtuaalikoneineen ja luokkakirjastoineen.

Java-ohjelmointikieli on monipuolinen. Sen avulla ohjelmoijat voivat kirjoittaa koodia, joka on käännettävissä ja toimii kaikilla Javaa tukevilla alustoilla ilman uudelleenkääntämistä. Java-sovellukset käännetään tavukoodiksi, mikä mahdollistaa niiden suorittamisen Java-virtuaalikoneessa (JVM) riippumatta tietokoneen taustalla olevasta arkkitehtuurista. (8.)

Javan ajonaikainen ympäristö (Java Runtime Environment, JRE) tarjoaa dynaamisia ominaisuuksia, kuten reflektiota ja ajonaikaista koodin muokkausta, mikä erottaa sen perinteisistä käännetyistä kielistä. (9.)

Java on yksi suosituimmista ohjelmointikielistä, ja sen avulla on luotu lukuisia tunnettuja ja menestyneitä sovelluksia. Sitä käytetään laajasti graafisissa käyttöliittymissä, yrityssovelluksissa, tieteellisissä sovelluksissa ja sulautetuissa jär-

jestelmissä. Javasta on julkaistu lukuisia eri versioita, joista Fivaldissa on kirjoitushetkellä käytössä versio 17.

Ennen kuin siirrytään asynkronisten teknologioiden tarkempaan tarkasteluun, käsitellään lyhyesti, mitä Servletit ja Servlet Container tarkoittavat web-sovelluksen kontekstissa, sillä ne ovat tärkeitä käsitteitä, kun puhutaan Java-pohjaisista web-sovelluksista. Tässä insinöörityössä toteutettava edistymispalkki toimii asynkronisen pyyntökäsittelyn yhteydessä ja ymmärtämällä, miten servletit ja servlet-säiliö liittyvät tähän, voidaan paremmin hahmottaa, miten asynkroninen pyyntökäsittely toimii Java-web-sovelluksissa.

Servlet on dynaamisen sisällön tuottamiseen käytetty Java-luokka, jota käytetään laajentamaan palvelimen ominaisuuksia. Se käsittelee HTTP-pyyntöjä ja -vastauksia, mikä mahdollistaa web-sovellusten palvelinpuolen logiikan toteuttamisen. Vaikka servletit voivat vastata kaikentyyppisiin pyyntöihin, niitä käytetään yleisesti laajentamaan verkkopalvelimien ylläpitämiä sovelluksia.

Servlet Container, tai servlet-säiliö, on osa web-palvelinta, joka hallinnoi ja suorittaa servlettejä. Se vastaanottaa HTTP-pyyntöjä, ohjaa ne oikeille servleteille ja huolehtii niiden elinkaaresta, kuten luomisesta ja tuhoamisesta. Servlet Container tarjoaa myös muita tärkeitä palveluita, kuten säikeiden hallintaa ja tietojen jakamista servletien välillä.

Perinteisesti servletit käsittelevät HTTP-pyyntöjä synkronisesti, mikä tarkoittaa, että servlet-säie odottaa, kunnes pyyntö on käsitelty loppuun ennen kuin se siirtyy käsittelemään seuraavaa pyyntöä. Tämä on tyypillinen synkroninen lähestymistapa. (10.)

Kuitenkin Java Servlet 3.1 -version myötä otettiin käyttöön asynkroninen pyyntökäsittely, joka mahdollistaa epäestävien (non-blocking) asynkronisten servletien luomisen. Asynkroniset servletit voivat vapauttaa säikeitä odottamaan pitkäkestoisten operaatioiden suorittamista, jolloin Servlet Container voi käsitellä

muita pyyntöjä samanaikaisesti. Tämä parantaa sovelluksen skaalautuvuutta. (11.)

2.7 Spring Framework -ohjelmistokehys

Spring Framework (usein pelkkä Spring) on ilmainen ja avoimen lähdekoodin ohjelmistokehys, joka tekee Java-sovellusten kehittämisestä ja ylläpidosta helpompaa, sillä se vapauttaa kehittäjät perusinfrastruktuurin, kuten tietokantayhteyksien hallinnan ja konfiguraation, rakentamisen vaivasta. Spring Framework tekee myös RESTful-web-palveluiden kehittämisestä helpompaa tarjoamalla laajasti valmiita ominaisuuksia. Tämä mahdollistaa kehittäjille keskittymisen olennaiseen, eli liiketoiminnan logiikkaan.

Spring-kehyksen alla on useita moduuleja ja ohjelmistokehyksiä. Tällaisia ovat esimerkiksi web-ohjelmointikehykset Spring Web MVC (usein pelkkä Spring MVC) ja Spring WebFlux.

Spring Web MVC on alkuperäinen web-ohjelmointikehys Spring Frameworkissa, ja se on suunniteltu erityisesti Servlet API:lle ja Servlet-säiliöille, joista mainitaan luvussa 2.6.

Spring WebFlux, puolestaan, on reaktiivinen web-ohjelmointikehys, joka lisättiin myöhemmin Spring Frameworkin versiossa 5.0. Se mahdollistaa reaktiivisen ohjelmoinnin, joissa tarvitaan joustavaa ja reagoivaa käyttäytymistä. Se toimii täysin epäestävästi ja tukee erilaisia palvelimia, kuten Nettyä, Undertowia ja Servlet-säiliöitä.

Molemmat web-ohjelmointikehykset toimivat rinnakkain Spring Frameworkissa. Kumpikin web-kehys on valinnainen, mikä mahdollistaa sovellusten valita joko toisen tai tarvittaessa molemmat. Esimerkiksi Spring MVC -kontrollereita voi käyttää yhdessä reaktiivisen WebClientin kanssa. (12.)

Spring Boot, joka on myös Fivaldissa käytössä, on Spring Frameworkin helppokäyttöinen ja tehokas laajennus, joka on suunniteltu nopeuttamaan ja helpottamaan verkkosovellusten ja mikropalveluiden kehittämistä Spring Frameworkilla. Kun Spring Framework vaatii usein manuaalista konfigurointia, Spring Boot tarjoaa käyttövalmiin ratkaisun, joka perustuu automaattisesti Spring Frameworkin oletusominaisuuksiin. Tämä mahdollistaa nopean käynnistyksen ja vähentää kehittäjälle jäävää konfigurointia. (13.)

Spring Boot noudattaa kerrostettua arkkitehtuuria, joka tässä yhteydessä tarkoittaa yleensä seuraavaa jakoa:

Käyttöliittymäkerros (view): Tämä kerros on nähtävillä käyttäjän selaimessa. Käyttäjän selaimessa suoritettava koodi tekee palvelimelle pyyntöjä käyttöliittymässä tehtyjen toimenpiteiden, kuten nappien painallusten ja sivulta toiselle navigoimisen pohjalta.

Kontrollerikerros (controller): Kerros on vastuussa esimerkiksi selaimelta palvelimelle tulevien pyyntöjen kuuntelusta. Pyynnön vastaanottaessa kontrollerikerros on vastuussa pyynnön ohjaamisesta oikealle palvelulle, sekä tuotetun tiedon ohjaamisesta oikealle näkymälle käyttöliittymäkerrokseen. Tämän kerroksen vastuulla on myös pyynnössä olevien parametrien validointi.

Palvelukerros (service): Tämän kerroksen vastuulla on liiketoimintalogiikan toteutuminen. Se vastaanottaa tietoa kontrollerikerrokselta, kutsuu tarvittaessa muita palveluita ja käsittelee saatua tietoa esimerkiksi kartoittamalla (mapping) tietoa toiseen muotoon.

Tietovarastokerros (repository): Tässä kerroksessa on ohjelman tallennuslogiikka ja se vastaa tietokantayhteyksistä ja tietokantakutsujen luomisesta. Kerros vastaanottaa tietoa palvelukerrokselta ja palauttaa sille tietoa. Tietovarastokerros on vastuussa CRUD (Create, Read, Update, Delete)-toimenpiteiden suorittamisesta tietokantaan eli tiedon tallentamiseen, hakemiseen, päivittämiseen ja poistamiseen liittyvästä logiikasta.

Näiden eri kerrosten välissä tieto liikkuu Java-olioiden avulla. Esimerkiksi kontrollerikerros voi lähettää palvelukerrokselle niin kutsutun DTO:n, joka tulee sanoista *Data Transfer Object*. Nämä DTO:t ovat tiedon siirtoon tarkoitettuja olioita, jotka pitävät sisällään esimerkiksi käyttöliittymästä saatua tietoa. Palvelukerros voi esimerkiksi lisätä tai poistaa jotakin oleellista tietoa DTO:sta ja muuntaa sen tietokantaentiteeteiksi.

Jos kerrosarkkitehtuuria noudatetaan täsmällisesti, kaikkien kerroksien tulisi olla olemassa. Lisäksi tieto ei saa kulkea kerrosten yli niin, että esimerkiksi kontrollerikerros kutsuu suoraan tietovarastokerrosta. Kutsut kulkevat myös vain ylhäältä alaspäin, joten esimerkiksi tietovarastokerros ei saa kutsua palvelukerrosta. (14.)

Kaiken kaikkiaan Spring Bootin arkkitehtuuri on suunniteltu helppokäyttöiseksi ja ymmärrettäväksi, ja sen modulaarinen rakenne mahdollistaa joustavien ja mukautettujen sovellusratkaisujen luomisen.

Javan ja Spring Bootin avulla voidaan rakentaa tehokkaita ja joustavia palvelinpuolen ratkaisuja. Yhdessä Angularin ja TypeScriptin kanssa tämä teknologiayhdistelmä mahdollistaa kokonaisvaltaisten ja skaalautuvien web-sovellusten toteuttamisen Fivaldi-ympäristössä.

2.8 Ohjelmointiympäristö ja työkalut

Ohjelmoijilla on vapaus valita oma ohjelmointiympäristö (IDE), mutta ohjeet ja konfiguraatiot on suunniteltu noudattamaan MicroSoftin Visual Studio Codea asiakaspuolen kehitystyössä ja JetBrainsin IntelliJ IDEAa palvelinpuolen kehityksessä. Vanhoja sovelluksia päivitellään edelleen Oraclen omilla kehitystyökaluilla, kuten Form Builderilla.

Projektissa käytetään Git-versionhallintaa, ja tähän tarkoitukseen hyödynnetään GitHub-palvelua. Git on hajautettu versionhallintajärjestelmä, joka mahdollistaa kehittäjien koodimuutosten seurannan ja koodin versiohistorian ylläpidon. Git-

Hub tarjoaa kätevän verkkopalvelun, johon voidaan tallentaa Git-versionhallinnan projekteja, tehdä yhteistyötä muiden kehittäjien kanssa, tarkastella koodimuutosten historiaa, luoda haaroja (branches) ja paljon muuta.

3 Teknologioiden vertailu

Insinöörityön luvussa 3 tarkastellaan erilaisia teknologioita, jotka mahdollistavat asynkronisen API:n käytön. Eri teknologioiden määrä on suuri, joten niihin kaikkiin ei perehdytä syvällisesti. Luvussa käydään läpi myös lopulliseen valintaan johtanutta päätöstä. Lisäksi käsitellään asynkronisen ja synkronisen API:n eroavaisuuksia ja asynkronisuuden tuomia etuja.

3.1 Teknologian valinta

Käytössämme oleva Spring Framework -ohjelmointikehys tukee useita erilaisia tekniikoita asynkronisen API:n toteuttamiseen. Näistä tekniikoista valittiin StreamingResponseBody-rajapinta kontrollerikerroksen metodin palautustyypiksi.

StreamingResponseBody-rajapinta (interface) mahdollistaa tiedon kirjoittamisen asynkronisesti HTTP-vastauksen runkoon. Sitä käytetään yleisesti, kun halutaan suoratoistaa suuria tietomääriä asiakkaalle lataamatta koko tietomäärää muistiin. Se on kontrollerikerroksen metodin palautustyyppi asynkroniselle pyyntökäsittelylle. Sitä käyttämällä sovellus voi kirjoittaa suoraan vastauksen OutputStreamiin ilman, että se estää Servlet-säiliön säiettä. Myöhemmässä kappaleessa käydään yksityiskohtaisemmin läpi OutputStreamin toiminnallisuuksia.

Kun käytetään StreamingResponseBodya suurten tietomäärien suoratoistoon, on suositeltavaa asettaa aikakatkaisun (timeout) raja. Muutoin käytetään taustalla olevan toteutuksen oletusaikakatkaisua, ja mikäli aikaraja ylittyy, sovellus aiheuttaa timeout-poikkeuksen. Aikakatkaisun rajaa voidaan muuttaa Springin application.properties -tiedostossa, ja sama määritys voidaan tehdä myös YAML-tiedostossa tai konfiguraatioluokassa. Esimerkkikoodi 1 näyttää, miten aikakatkaisu voidaan määrittää millisekunteina application.properties -tiedostossa.

```
spring.mvc.async.request-timeout = 600000
```

Esimerkkikoodi 1: application.properties-tiedoston määrittelemä raja pyynnön aikakatkaisuun millisekunteina.

StreaminResponseBody on funktionaalinen rajapinta, joka määrittelee yhden abstraktin metodin (15.), joka nähdään esimerkkikoodi 2:ssa.

```
void writeTo(OutputStream outputStream) throws IOException;
```

Esimerkkikoodi 2: StreamingResponseBodyn funktionaalisen rajapinnan määrittelemä metodi.

Funktionaalinen rajapinta (functional interface) on Java-ohjelmoinnin käsite, joka mahdollistaa lambda-lausekkeiden ja anonyymien funktioiden käytön. Funktionaalisessa rajapinnassa on tarkalleen yksi abstrakti metodi. (16.)

```
StreamingResponseBody responseBody = response -> {
    response.write("Tämä on esimerkki".getBytes());
  };
```

Esimerkkikoodi 3: StreamingResponseBody toteutettuna lambda-lausekkeella.

Käyttäessä lambda-lauseketta esimerkkikoodi 3:n esittelemällä tavalla StreaminResponseBodyn writeTo-metodia ei tarvitse erikseen toteuttaa. Tämä tekee koodista selkeämpää ja lyhyempää samalla, kun se mahdollistaa suoritettavan koodin helpomman hallinnan. Tämä lähestymistapa auttaa myös vähentämään ohjelmointivirheitä, koska se tarjoaa selkeät määritelmät toiminnallisuuksista ja vähentää monimutkaisen tilanhallinnan tarvetta.

Tiedon suoratoistoon palvelimelta asiakkaalle asynkronisesti on paljon erilaisia teknologioita, joita Spring Framework tukee.

Seuraavana on esimerkkejä:

Servletin OutputStream: Tämä on vanhin käytetty lähestymistapa tiedon suoratoistoon servlet-pohjaisissa sovelluksissa, kuten Spring Web MVC:ssä. OutputStream on abstrakti Java-luokka ja toimii yliluokkana kaikille luokille, jotka edustavat tavujen lähtövirtaa. Sen tarjoaa Java Standard Library (java.io-paketti). (17.)

Tässä lähestymistavassa OutputStream saadaan HttpServletResponsesta ja sitten sisältö kirjoitetaan OutputStream-olioon.

HttpServletResponse on Java Servlet -teknologian rajapinta.

Servlet, joka toteuttaa tämän rajapinnan, käsittelee saapuvia HTTP-pyyntöjä ja muodostaa niihin vastauksen HttpServletResponse-rajapinnan avulla. Tämä rajapinta tarjoaa metodeja, joilla servlet voi määrittää vastauksen ominaisuuksia, kuten HTTP-tilakoodin (status code), otsikot (headers) ja rungon (body).

Uudemmat tekniikat eivät kuitenkaan korvaa suoraan OutputStream-luokan käyttöä. Ne, kuten StreamingResponseBody, tuovat mukanaan korkeamman tason rajapintoja tai abstraktioita datavirtojen käsittelyyn. Tällaiset abstraktiot tarjoavat kätevän tavan hallita suoratoistoa vähentäen samalla siihen liittyvää manuaalista käsittelyä. (18.)

ResponseBodyEmitter: Springin versio 4.2 esitteli ResponseBodyEmitter- luokan. Se on useiden alaluokkien yläluokka, kuten SseEmitterin, jota tarkastelemme tarkemmin myöhemmin.

ResponseBodyEmitter mahdollistaa tapahtumien lähettämisen kirjoittamalla oliot vastaukseen sopivan HttpMessageConverterin kautta. Tämä on yleisesti käytetty tekniikka, esimerkiksi kirjoitettaessa JSON-tietoja. (19.)

HttpMessageConverter on rajapinta, jonka toteuttamat luokat toimivat HTTP-pyyntöjen ja vastauksien viestimuuntimina. Spring osaa asettaa joitakin sopivia viestimuuntimia automaattisesti. Esimerkiksi Java-oliot voidaan muuntaa automaattisesti JSON-muotoon (käyttäen Jackson-kirjastoa), mutta myös kustomoitujen viestimuuntimien käyttö on mahdollista. (20.)

Joskus voi olla hyödyllistä ohittaa viestimuunnos ja kirjoittaa suoraan vastauksen OutputStreamiin. Esimerkiksi kun suoritetaan tiedostojen lataamista tai

suurten tietomäärien suoratoistoa viestimuunnoksen ohittaminen voi parantaa suorituskykyä.

Viestimuunnoksen ohittaminen ja suoraan vastauksen OutputStreamiin kirjoittamisen voi toteuttaa esimerkiksi jo aikaisemmin mainitun StreamingResponseBody-palautusarvotyypin avulla.

SseEmitter: SseEmitter on ResponseBodyEmitterin alaluokka, joka tarjoaa lisänä tuen Server-Sent Event (SSE) -tekniikalle.

SSE on useimmissa selaimissa tuettu tekniikka, joka mahdollistaa tapahtumien yksisuuntaisen (uni-directional) suoratoiston milloin tahansa. Tapahtumat ovat UTF-8-koodattua tekstidataa ja noudattavat määriteltyä muotoa, joka koostuu avainarvoelementeistä (kuten tunnus, uudelleenyritys, data ja tapahtuman nimi) eroteltuina rivinvaihdoilla. Tekstidata voi olla yksinkertainen merkkijono tai monimutkaisempi JSON- tai XML-rakenne.

Yksisuuntainen viestintä tarkoittaa, että palvelin voi lähettää tietoa asiakkaalle ilman, että asiakas voi lähettää tietoa takaisin palvelimelle. Tämä tekee SSE:stä sopivan ratkaisun tilanteisiin, joissa tarvitaan yksisuuntaista tiedonsiirtoa, kuten päivitysten tai tapahtumien ilmoittamiseen asiakkaalle.

SSE:n viestimuoto on määritelty mediatyypillä text/event-stream.

Spring WebFlux, tarjoaa sisäänrakennetun tuen SSE-tekniikalle SseEmitter-luokan kautta. SseEmitter-luokkaa voidaan käyttää Spring WebFluxissa SSE-palvelimen puolen toteutukseen. Tämä luokka mahdollistaa viestien lähettämisen asiakkaille asynkronisesti. Myös Spring MVC tukee SSE-spesifikaatiota Springing versiosta 4.2 lähtien. (21.)

WebSocket: WebSocketit mahdollistavat kaksisuuntaisen (bi-directional) viestinnän asiakkaan ja palvelimen välillä, mikä tekee niistä sopivia reaaliaikaisiin

interaktiivisiin sovelluksiin. Spring Framework tarjoaa WebSocket-tuen ja Spring WebFlux tukee reaktiivista WebSocket-ohjelmointia. (22.)

WebSocketit eroavat aiemmin mainitusta SSE:stä siinä, että WebSocketit tarjoavat kaksisuuntaista viestintää palvelimen ja asiakkaan välillä, kun taas SSE käyttää yksisuuntaista viestintää.

Reactive Streams API: Reactive Streams API on spesifikaatio asynkroniselle tietovirran käsittelylle ei-estävällä tavalla. Siitä on olemassa erilaisia toteutuksia, kuten esimerkiksi RxJava tai Akka Streams. Se määrittelee rajapinnat, metodit ja protokollat asynkronisten datasekvenssien käsittelemiseksi.

Spring WebFlux on rakennettu Reactive Streams API:n päälle ja tarjoaa reaktiivista ohjelmointitukea datavirtojen käsittelyyn. (23)

Eri tekniikoiden määrä tiedon suoratoistoon asynkronisella API:lla on kuitenkin valtava ja jokaiseen perehtyminen syvällisesti vie aikaa.

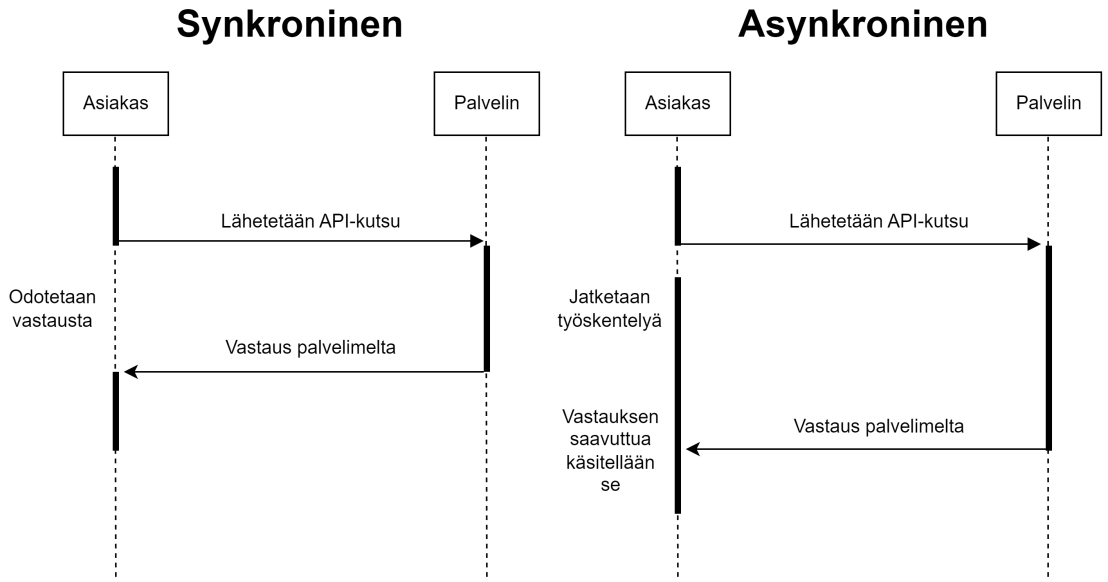
Tässä insinöörityössä päädyttiin StreamingResponseBodyn käyttöön, joka pohjautui pääasiassa siihen, että projektiin ei tarvinnut luoda uusia riippuvuuksia eri kirjastoihin, sillä Fivaldi-ohjelma on perinteinen verkkosovellus, jossa on tähän asti ollut käytössä synkroninen pyyntö-vastaus-vuorovaikutus. StreamingResponseBodyn valinta latauspalkin hyödyntämiseen pohjautui myös tulevaisuutta ajatellen, sillä tarkoituksena on jatkossa käyttää sitä myös suurien tietomäärien lähettämiseen asiakkaalle tavumuodossa. Näin ohjelman rakenne pysyisi johdonmukaisena, eikä sisältäisi useita erityyppisiä teknologioita tiedon suoratoistoon.

3.2 Synkroninen vs asynkroninen API-kutsu

Synkroninen REST API käsittelee asiakkaalta (selain) tulevat pyynnöt estävästi (blocking) eli asiakas joutuu odottamaan, että palvelin on käsitellyt pyynnön ja lähettänyt vastauksen. Jos palvelin on ruuhkautunut tai pyynnön käsittelyssä

kestää muusta syystä kauan, voi tästä aiheutua asiakkaalle viivettä, sillä ohjelman toiminnon suorittaminen on yleensä estynyt vastausta odotellessa.

Asynkroninen REST API puolestaan käsittelee pyynnöt ei-estävästi. Kun asiakas lähettää pyynnön API:lle, API kuittaa pyynnön ja antaa välittömän vastauksen merkiksi siitä, että pyyntö on vastaanotettu ja sitä käsitellään. Pyynnön varsinainen käsittely tapahtuu kuitenkin taustalla palvelimen puolella eikä asiakkaan tarvitse odottaa käsittelyn valmistumista. Asiakas voi jatkaa ohjelman käyttöä muilta osin ja käydä tarkistamassa tehtävän tilaa aika-ajoin esimerkiksi latauspalkin edistymistä seuraamalla. Tämä antaa asiakkaalle myös reaaliaikaisen kuvan ohjelman suoriutumisesta. Asynkroninen tapa toteuttaa REST API on erityisen hyödyllinen juuri pitkään kestävissä tehtävien suorituksissa. (24.)



Kuva 4: Synkronisen ja asynkronisen API-kutsujen erot

Kuten kuva 4 esittää, ero asynkronisessa ja synkronisessa API:ssa on siinä, kuinka ne käsittelevät pyyntö-vastaussykliä. Synkroninen API kutsu edellyttää asiakkaan odottavan vastausta, ja koko vastaus täytyy pitää muistissa palvelimella ennen lähettämistä asiakkaalle. Asynkronisessa kutsussa yhteys pidetään auki palvelimen lähettäessä tietoja. Tietoa lähetetään asiakkaalle osissa

(chunk), eikä asiakkaan tarvitse odottaa pyynnön valmistumista vaan voi jatkaa toimintaansa ja tarkistaa pyynnön etenemistä.

4 Palvelinpuolen ohjelman kirjoitus

Tässä luvussa syvennytään palvelinpuolen ohjelman kirjoittamiseen ja käsitellään sen eri näkökulmia. Tutustutaan palvelimen suorittaman koodin luomiseen ja käydään läpi työn mukanaan tuomat haasteet. Keskeisenä osana perehdytään monisäikeistetyn koodin etuihin ja riskeihin sekä tutkitaan Producer-Consumer-suunnittelumallia ja sen käytännön toteutusta tässä projektissa.

Lisäksi tutustutaan myös Spring Frameworkin beaneihin ja niiden vaikutukseen säieturvallisuuteen, erityisesti tarkastellen tilallisten ja tilattomien beanien ominaisuuksia.

4.1 Haasteet

Insinöörityön ensimmäinen haaste oli, ettei vastaavaa asynkronista API:a olla aiemmin Fivaldiin kehitetty. Työ alkoi tutustumalla internetin laajaan valikoimaan erilaisia teknologioita, joilla tehtävä mahdollisesti voitaisiin suorittaa.

Toisena haasteena nousi esiin eri tekniikoiden laaja valikoima. Luku 3.1 esittelee joitakin näistä tekniikoista. Haastavaa tässä oli valita sopivin vaihtoehto ja ylipäätään mahdollinen tapa toteuttaa reaaliaikaista edistymistä kuvaava latauspalkki.

Tehtävän kolmas haaste syntyi itse koodin kirjoittamisessa. Ensin oli refaktoroitava huomattava määrä vanhaa koodia selkeyttämään ohjelman toimintaa ja rakennetta. Refaktoroinnin yhteydessä syntyi ajatus, että taloyhtiön tietojen hakua ja prosessointia voitaisiin nopeuttaa hyödyntäen yhtäaikaista ohjelmointia (concurrent) käyttämällä monisäikeisyyttä (multi-threading). Tämä oli myös uusi lisäys Fivaldin arkkitehtuuriin, sillä tämänkaltaista monisäikeistettyä koodia ei ole Fivaldissa ollut aiemmin käytössä.

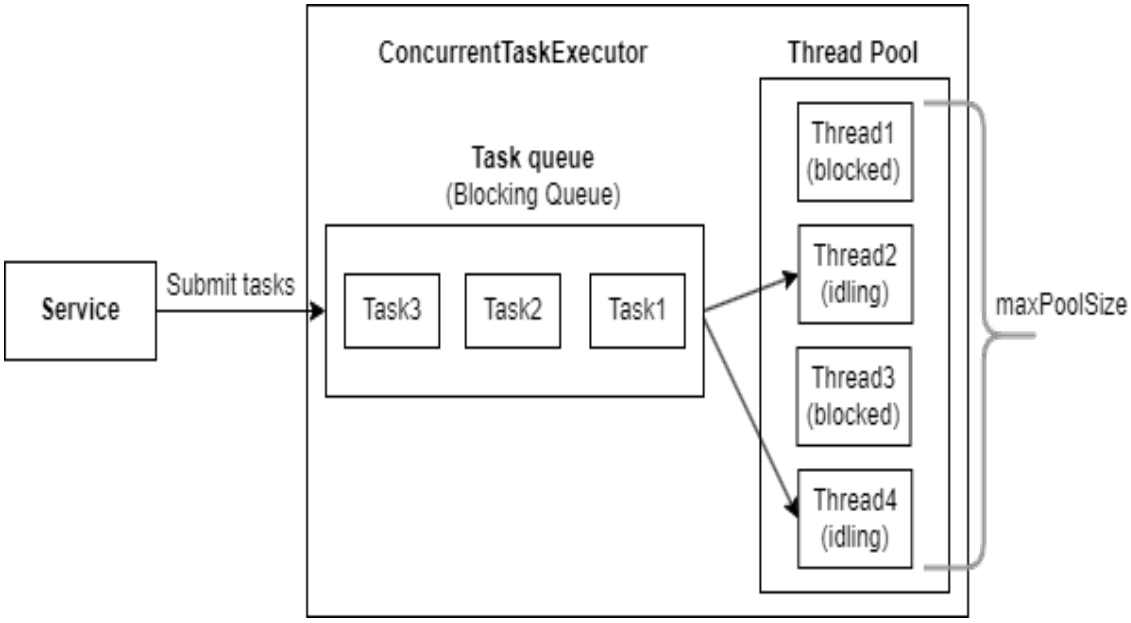
4.2 Monisäikeistetty koodi ja säikeiden turvallisuus

Tietojenkäsittelytieteessä säikeellä (thread) tarkoitetaan ohjelman suorituksen perusyksikköä, joka voi suorittaa tehtäviä samanaikaisesti muiden säikeiden kanssa.

Monisäikeistetyssä koodissa useampi säie suorittaa yhtäaikaisesti eri tehtäviä, mikä parantaa tehokkuutta ja vähentää ohjelman suoritukseen kuluvaa aikaa. Tämä lisää ohjelman suorituskykyä ja nopeuttaa monimutkaisten tai pitkään kestävien tehtävien käsittelyä.

Säikeiden turvallisuudesta (thread-safety) on kuitenkin syytä huolehtia, jotta vältetään odottamattomia ja vaikeasti diagnosoitavia ongelmia. Säieturvallisuus tarkoittaa käytännössä sitä, että kun useat tehtävät suoritetaan samanaikaisesti ohjelmassa, ne eivät häiritse toistensa toimintaa tai aiheuta ongelmia. Se varmistaa, että ohjelma toimii ennustettavasti ja välttyy virheiltä, kuten kilpailutilanteilta (race condition), joissa säikeet yrittävät käsitellä samaa resurssia samanaikaisesti. (25.)

Monisäikeistetyn koodin käyttö Fivaldissa herätti kokeneemmissa ohjelmistokehittäjissä huolta siitä, riittääkö sovellukselle annettujen säikeiden määrä, etenkin jos säikeiden avulla yhtäaikaisesti suoritettavan koodin käytöstä tulee yleinen tapa. Tämän ongelmaan ratkaisuun käytettiin Spring Frameworkin ConcurrentTaskExecutor-luokkaa, joka helpottaa useiden säikeiden hallintaa sekä kykenee hakemaan ThreadPoolista joutilaana (idling) olevia säikeitä. Idling-tilassa olevien säikeiden käyttö ei vaadi raskasta toimenpidettä uusien säikeiden luomiseksi, eikä ohjelman käyttöön tarjottujen säikeiden määrää jouduta kasvattamaan.



Kuva 5: Tehtävien lähettäminen ConcurrentTaskExecutorin avulla tehtäväjonosta Thread Pooliin

Kuvassa 5 esitellään yksinkertaistettuna ThreadPool-rakenne ja sen yhteistoiminta ConcurretTaskExecutor-luokan kanssa. ThreadPool pitää hallinnassa ja uudelleenkäyttää säikeitä tehtävien suorittamiseen. Se auttaa varmistamaan, että riittävä määrä säikeitä on käytettävissä jonossa oleville tehtäville. Kun tehtävä saapuu, valmiina oleva säie voidaan ottaa käyttöön sen sijaan, että luotaisiin uusi säie alusta asti. Tämä optimoi resurssien käyttöä ja nopeuttaa tehtävien suorittamista monisäikeisissä sovelluksissa.

Käytettäessä ConcurretTaskExecutoria asynkroniseen tiedon prosessointiin, on erittäin suositeltavaa määritellä konfiguraatiotiedosto sitä varten. Tuotantoympäristössä on järkevää harkita ConcurretTaskExecutorin rajoittamista säikeiden osalta asynkronisen toiminnallisuuden hallitsemiseksi. Ilman rajoituksia saattaa syntyä riski ylikuormittaa palvelimen resursseja. (26.) Esimerkkikoodi 4 esittelee konfiguraatiotiedostoa, jota voi muokata vastaamaan sovelluksen tarpeita ja palvelimen resursseja.

```
@Configuration
@EnableAsync
@EnableScheduling
public class AsyncConfig implements AsyncConfigurer {
```

```
    @Override
    @Bean(name = "asyncTaskExecutor")
    public Executor getAsyncExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(1);
        executor.setMaxPoolSize(2);
        executor.setQueueCapacity(25);
        return executor;
    }
    @Bean
    protected ConcurrentTaskExecutor getTaskExecutor() {
        return new ConcurrentTaskExecutor(this.getAsyncExecutor());
    }
    @Bean
    protected WebMvcConfigurer webMvcConfigurer() {
        return new WebMvcConfigurer() {
            @Override
            public void configureAsyncSupport(AsyncSupportConfigurer
configurer) {
                configurer.setTaskExecutor(getTaskExecutor());
            }
        };
    }
}
```

Esimerkkikoodi 4: ConcurretTaskExecutorin konfiguraatiotiedosto

Jos kaikki säikeet ovat varattuina ja ThreadPoolissa olevien säikeiden määrä ei ylitä esimerkkikoodissa 4 esillä olevan setMaxPoolSize-funktion määrittelemää säikeiden maksimi määrää, ThreadPooliin voidaan käynnistää uusi säie. Muussa tapauksessa tehtävät odottavat tehtäväjonossa, kunnes jokin säie vapautuu.

4.3 Producer-Consumer-suunnittelumalli

Producer-Consumer-malli on suunnittelumalli, joka käsittelee tietojen jakamista ja synkronointia tuottajien (producers) ja kuluttajien (consumers) välillä. Tämä malli on hyödyllinen tilanteissa, joissa tietoa tuotetaan nopeammin kuin se voidaan käsitellä tai kuluttaa.

Tässä mallissa on kaksi päätoimijaa, tuottajat ja kuluttajat, jotka toimivat omissa säikeissään. Kuluttajia ja tuottajia voidaan käynnistää myös useisiin säikeisiin. Näiden kahden toimijan välillä toimii jaettu resurssi.

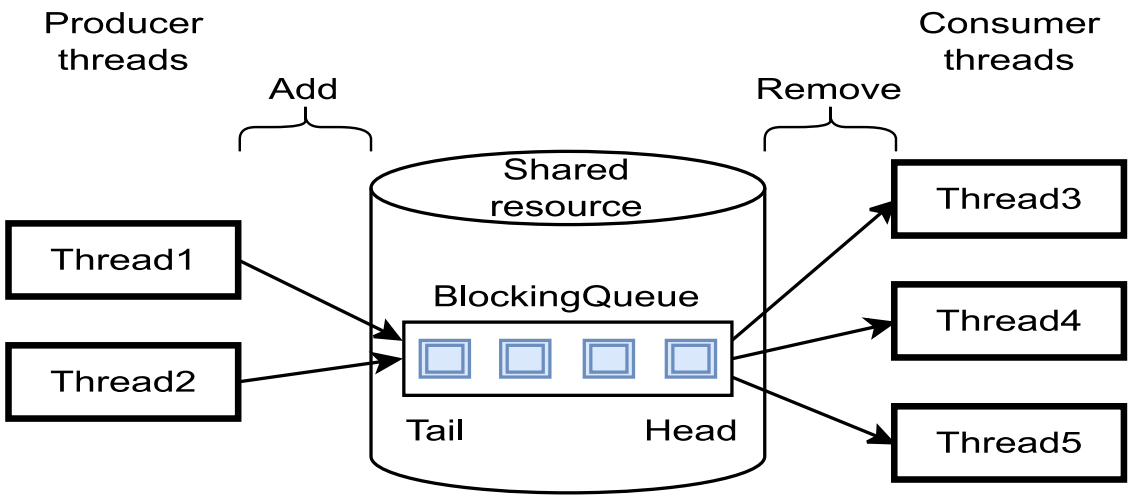
Tuottajat ovat nimensä mukaisesti vastuussa tietojen tuottamisesta tai hakemisesta. Ne lisäävät tietoa jaetulle resurssille, josta kuluttajat voivat sen noutaa.

Kuluttajat ovat vastuussa tuotetun tiedon kuluttamisesta tai käsittelemisestä. Ne poistavat tietoa jaetulta resurssilta ja käsittelevät sitä.

Jaettu resurssi toimii tuottajien ja kuluttajien välisenä välimuistina tai puskurina. Tämä välimuisti voi olla esimerkiksi jono tai jokin muu tietorakenne, jossa tuotettu tieto sijoitetaan odottamaan kuluttamista.

Tärkeää tässä mallissa on synkronoinnin ja kilpailutilanteiden hallinta. Koska useissa säikeissä operoivat tuottajat ja kuluttajat voivat toimia jaetun resurssin parissa samanaikaisesti, on varmistettava, että tieto käsitellään turvallisesti ja oikeassa järjestyksessä. (27.)

Tämän insinöörityön tavoitteena olevan reaaliaikaisen latauspalkin toteuttamiseksi päädyttiin käyttämään Producer-Consumer-mallia, jotta tietoa latauksen edistymisestä saataisiin suoratoistettua jaetun resurssin kautta käyttöliittymään. Lisäksi tämän monisäikeisen mallin käyttäminen nopeutti tiedon hakua ja prosessointia verrattuna aikaisempaan toteutukseen.



Kuva 6: Producer-Consumer mallissa voidaan käyttää myös useita säikeitä. BlockingQueue-tyyppinen FIFO-jono on eräs tapa käyttää jaettua resurssia.

Kuten kuva 6 esittää, Producer-Consumer mallissa on mahdollista käynnistää tuottajia ja kuluttajia useisiin säikeisiin toimimaan yhtäaikaisesti. Tässä opinnäytetyössä esiteltävää latauspalkkia varten jokaiselle API-kutsulle määriteltiin ai-

noastaan yksi säie tuottajalle ja yksi säie kuluttajalle, mutta optio useamman tuottajan tai kuluttajan luontiin säilytettiin. Tämä päätös perustui resurssipulan pelkoon sekä siihen, että isännöitsijä, joka käyttää Kiinteistönhallinta-sovelluksen taloyhtiöiden alustusvelhoa, hakee taloyhtiöiden tiedot yleensä kertaluontoisesti. Näin ollen äärimmäisen suorituskyvyn tarvetta ei ollut. Useamman producerin käyttö olisi vaatinut myös lisäselvityksiä huoneistotietojärjestelmän kyvystä käsitellä useita samanaikaisia pyyntöjä.

Lautauspalkin toteutuksessa tuottajia ja kuluttajia edustavista luokista luodaan aina uusi ilmentymä jokaista API-kutsua kohden säieturvallisuuden takaamiseksi. Spring Framework luo yleensä luokista vain yhden ilmentymän (singleton), jonka takia näille luokille tuli antaa annotaatio @Scope(”prototype”), josta on lisää luvussa 4.4. Tuottajan ja kuluttajan ilmentymät käynnistetään omiin säikeisiinsä, joissa tuottaja hakee taloyhtiön tietoja Maanmittauslaitoksen huoneistotietojärjestelmästä iteroiden läpi sille konstruktorin parametrina annettua osakeryhmätunnuksista koostuvaa listaa ja kuluttaja kartoittaa (mapping) tuottajan tuottamaa dataa Kiinteistönhallinta sovelluksen alustusvelhon tarvitsemaan muotoon.

Näiden kahden luokan välistä kommunikointia ja monitorointia varten kirjoitettiin luokka nimeltä Broker, joka pitää sisällään kaksi BlockingQueue-rajapinnan toteuttamaa jonoa.

BlockingQueue on Java-ohjelmointikielen java.util.concurrent-paketin sisältämä rajapinta (interface). Se edustaa tietorakennetta, joka toimii kuten tavallinen jono (queue), mutta tarjoaa myös tiettyjä estäviä (blocking) ominaisuuksia. Tällaisia estäviä ominaisuuksia ovat esimerkiksi se, että jos jono on täynnä, alkioiden lisääminen siihen estyy. Samoin, jos jono on tyhjä, poisto-operaatio joutuu odottamaan, kunnes siihen on lisätty uusi alkio. BlockingQueue tarjoaa myös tehokkaan ja turvallisen tavan kommunikoida ja synkronoida säikeiden välillä, erityisesti tilanteissa, joissa yksi säie yrittää lisätä tietoa jonoon, samalla kun toinen yrittää ottaa tietoa jonosta. (28.)

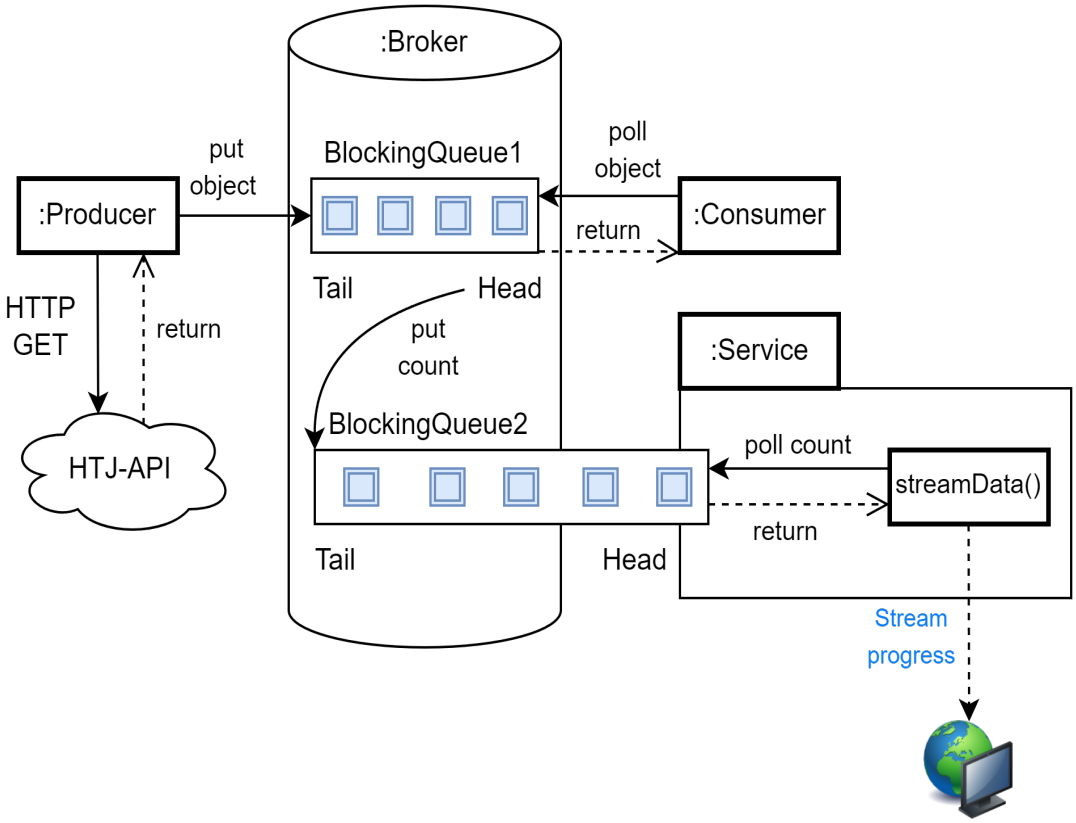
Latauspalkin toteuttaminen Producer-Consumer-suunnittelumallin avulla, vaati jonkinlaisen jaetun resurssin tuottajien ja kuluttajien välille. Toiseksi tuottajan hakemaa tietomäärää haluttiin ilmaista reaaliaikaisesti käyttöliittymässä.

Osana ratkaisua näihin ongelmiin kirjoitettiin monitoroijana ja väylänä toimiva Broker-luokka, jonka toimintaa havainnollistetaan sivun 31 kuvassa 7. Tuottajien ja kuluttajien tapaan, myös Broker-luokasta luodaan aina uusi ilmentymä API-kutsua kohden. Erona näillä on se, että Broker on tilallinen Springin hallinnoimaton Java-luokka, kun taas tuottajaa ja kuluttajaa edustavat luokat ovat Springin hallinnoimia tilallisia luokkia.

Broker-luokka pitää sisällään erilaisia instanssimuuttujia, mukaan lukien kaksi BlockingQueue-rajanpinnan toteuttavaa jonoa.

Broker luo ja alustaa ensimmäisen jonon omassa konstruktorissaan. Tämä jono toimii väylänä tietojen siirtämiseksi tuottajalta kuluttajalle. Tuottajat voivat lisätä tietoa tähän jonoon, ja kuluttajat voivat noutaa tietoa siitä FIFO-periaatteella.

Toinen jono annetaan Brokerille konstruktorin parametrina palvelukerroksen metodilta. Tämä mahdollistaa tiedon siirron Brokerin ja palvelukerroksen metodin välillä. Broker kirjoittaa tähän jonoon tietoa siitä, kuinka monta kohdetta ensimmäiseksi mainitun jonon kautta on kulkenut. Palvelukerroksen metodi noutaa jonosta tätä lukumäärää ja vertaa sitä jo tietämäänsä kokonaislukumäärään. Tätä tietoa palvelukerroksen metodi kirjoittaa luvussa 3.1 mainitun StreamingResponseBodyn writeTo-metodia hyödyntäen reaaliaikaisesti käyttöliittymään, mikä mahdollistaa latauksen tilan visuaalisen seurannan.

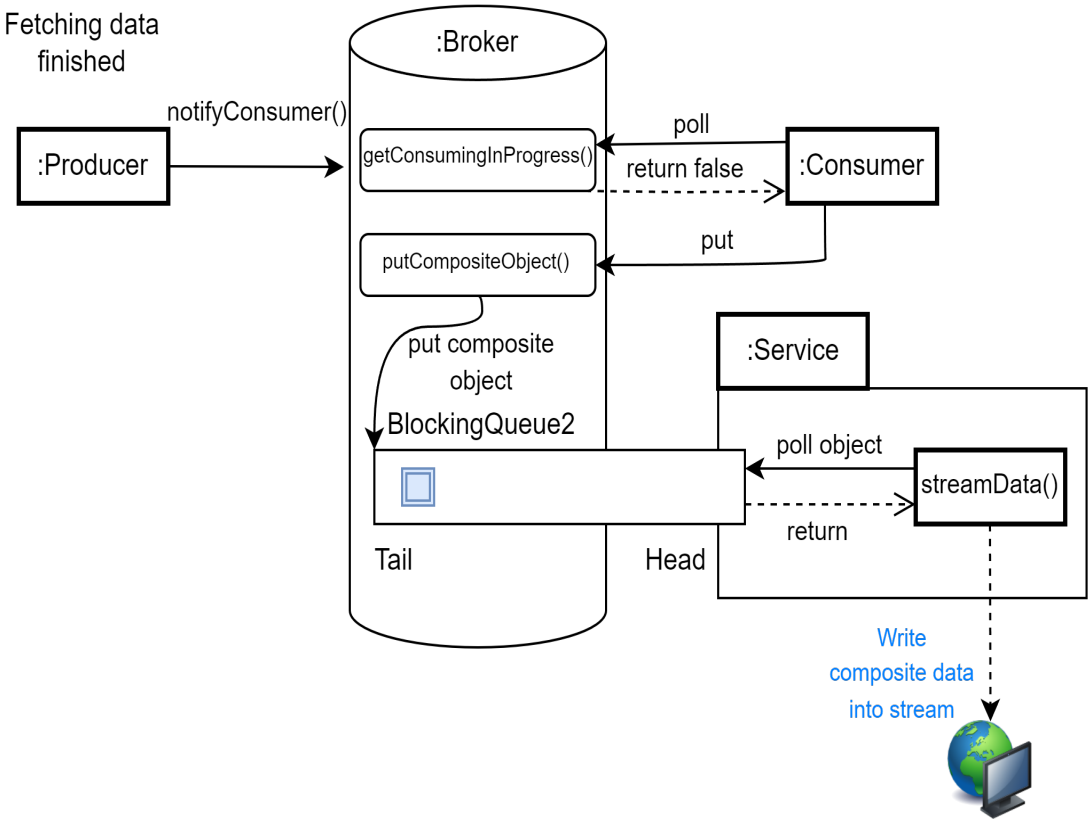


Kuva 7: Yksinkertaistettu esimerkki Producer-Consumer suunnittelumallin toteuttamisesta latauspalkin käyttötapaukseen

Näiden kahden jonorakenteen hallinnoinnin lisäksi Broker on vastuussa virheiden hallinnasta. Jos tuottajasäie kohtaa virheen, se ilmoittaa siitä Brokerille. Broker tallentaa virhetiedon instanssimuuttujaansa, jota kuluttajasäie tarkistaa jokaisen iteraation alussa. Jos virhe on ilmennyt tuottajasäikeessä, myös kuluttajasäie lopettaa toimintansa. Näin vältytään tilanteelta, että säikeet jäävät taustalle pyörimään ja varaamaan resursseja.

Kuva 8 havainnollistaa tilannetta, jossa tuottaja on lopettanut tiedonhaun huoneistotietojärjestelmästä. Tuottaja ilmoittaa haun päättymisestä Brokerille, jonka kautta kuluttaja saa tiedon. Kun kuluttaja on saanut käsiteltyä kaikki jonossa olleet elementit se antaa näistä elementeistä koostetun olion Brokerille. Broker kirjoittaa koosteolion palvelukerroksen metodin kanssa jaettuun jonoon, jonka

palvelukerroksen metodi kirjoittaa asiakkaalle johtavaan tietovirtaan. Tätä tietovirran prosessointia asiakaspuolella tarkastellaan luvussa 5.1.



Kuva 8: Tilanne, jossa Producer-luokan tiedon haku on päättynyt

Kuvan 8 esittämässä tilanteessa selaimessa nähtävä edistymispalkki on edennyt loppuun, ja taloyhtiön tietoja sisältävä koosteolio on kirjoitettu tietovirtaan. Latauksen päätteeksi taloyhtiön alustusvelho, joka vastaa sivun 6 kuvan 2 esitystä, ilmestyy asiakkaan ruudulle. Luvussa 5.2 keskitytään tiedon visualisointiin asiakaspuolella, ja sivun 46 kuvassa 11 näkyy edistymispalkin lopullinen ilme.

4.4 Tilalliset luokat ja HtjWizardContextFactory

Springissä termi ”bean” viittaa komponenttiin tai luokkaan, joka on hallinnoitu Springin Inversion of Control (IoC) -kontainerin avulla. Näitä beaneja voidaan in-

jektoida muihin sovelluksen osiin, kuten palveluihin, kontrollereihin ja muihin komponentteihin, mikä helpottaa eri osien yhteistyötä. Springissä bean voi olla mikä tahansa Spring-sovelluksen hallinnoima komponentti, ja ne voidaan luoda ja konfiguroida eri tavoin. Yleisesti Spring Bootissa luokat merkitään annotaatioilla kuten @Component, @Controller, @Service, @Repository, jolloin ne tulkitaan Spring Beaneiksi. Nämä beanit voivat sitten olla osa sovelluksen komponenttipuuta, josta ne ovat helposti saatavilla ja käytettävissä muiden sovelluksen osien kesken. Spring Boot tekee Spring-komponenttien luomisesta ja hallinnasta helppoa ja tehokasta. (29.)

Oletuksena Spring luo jokaisesta beanista vain yhden ilmentymän (instance), ja useimmiten nämä beanit ovat tilattomia (stateless), mikä vähentää yhtäaikaisesti ajettavaan koodiin liittyviä ongelmia.

Tässä insinöörityössä toteutetun edistymispalkin toimintatavan vuoksi jouduttiin kuitenkin luomaan myös tilallisia beaneja ja Springin hallinnoimattomia Java-luokkia. Tällaisia tilallisia beaneja ovat luvussa 4.3 esitellyt tuottajia ja kuluttajia edustavat luokat, sekä tilallinen, Springin hallinnoimaton Broker-luokka.

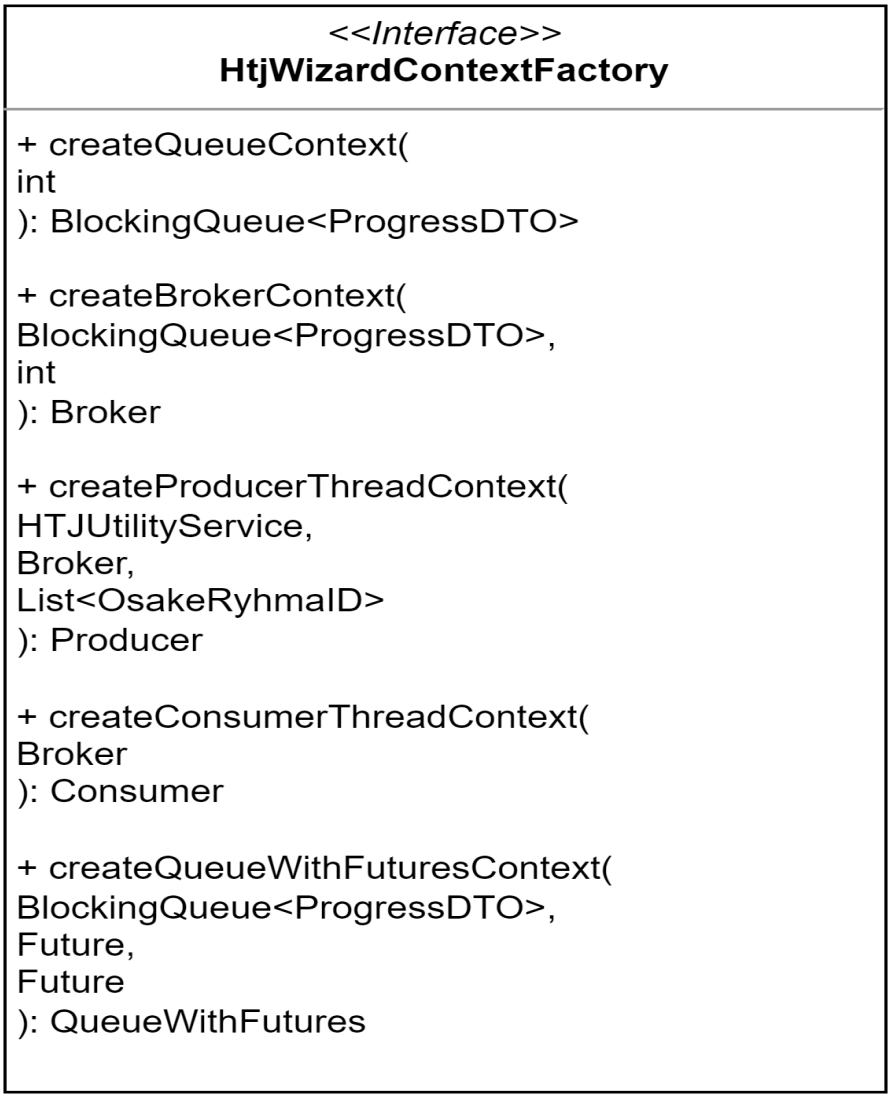
Yleensä tilalliset beanit merkitään luokan alussa käyttämällä @Scope("prototype") -annotaatiota. Tämä annotaatio varmistaa sen, että jokaista pyyntöä kohti luodaan uusi beanin ilmentymä. (30.) Tämä taas takaa, että kullakin käyttäjällä on oma erillinen ilmentymä, joka ei jaa tilaa muiden käyttäjien kanssa. Tilallisten beanien yhteydessä on tärkeää kiinnittää huomiota säieturvallisuuteen. Säieturvallisuuden varmistamiseksi luotiin uusi rajanpinta HtjWizardContextFactory, joka määrittelee metodit, jotka ovat vastuussa näiden tilallisten olioiden luomisesta.

HtjWizardContextFactoryImpl-luokka toteuttaa HTJWizardContextFactory-rajapinnan, ja tämä konkreettinen luokka vastaa tilallisten beanien ja Java-luokkien ilmentymien luomisesta aina, kun uusi asiakas käynnistää Kiinteistönhallinnan asetusvelhon päivittääkseen huoneistotietoja. Se varmistaa, että jokainen säie, joka käyttää tilallisia beaneja, saa uuden ilmentymän new-operaation avulla.

Uusien ilmentymien luominen jokaiselle säikeelle HTJWizardContextFactory-rajapinnan kautta varmistaa, että eri säikeet eivät kilpaile samoista ilmentymistä. Tämä edistää säieturvallisuutta ja välttää mahdolliset konfliktit ja kilpailutilanteet resursseista. Tämä on erityisen tärkeää tilallisten beanien tapauksessa.

HTJWizardContextFactory-rajapinta helpottaa myös palvelukerroksen yksikkötestien laatimista, kun uusien olioiden luonti on kapseloitu erilliseen toteutukseen. Tämän ansiosta testiluokka voi luoda oliot yksinkertaisesti käyttämällä rajapintaa, mikä tekee testaamisesta vaivatonta ja ylläpidosta selkeämpää. Rajapinta tarjoaa abstraktion, joka eristää testiluokan konkreettisen toteutuksen yksityiskohdista. Näin ollen testiluokka voi keskittyä testattavan toiminnallisuuden varmistamiseen ilman tarvetta huolehtia siitä, miten oliot luodaan. Tämä parantaa koodin joustavuutta ja mahdollistaa toteutuksen vaihtamisen ilman suoraa vaikutusta testien toimivuuteen.

Kuvassa 9 esitellään luokkakaavion muodossa HTJWizardContextFactory-rajapinnan sisältämiä metodeita.



Kuva 9: Luokkakaavio HtjWizardContextFactory-rajapinnan sisältämistä metodeista

4.5 Kontrollerikerroksen koodi

Spring Bootissa kontrolleriluokalle kirjoitetaan @Controller- tai @RestController-merkintä, jonka perusteella Spring osaa luoda automaattisesti luokasta olion, joka sitten injektoidaan sovelluksen käyttöön. (31.)

Kontrollerikerroksen metodit tulisi useimmiten merkitä Springin @Get-, @Post-, @Put-, @Patch- tai @DeleteMapping -annotaatioilla riippuen siitä, millaisia

HTTP-pyyntöjä kyseiset metodit käsittelevät. (32.) Edistymispalkkia varten oli tarpeen kuitenkin luoda vain @GetMapping-merkitty metodi.

Asiakkaan lähettäessä palvelimelle pyynnön tiedon hakemiseksi kontrolleriker-roksen luokan metodi vastaanottaa pyynnön ensimmäisenä palvelimen päässä. Jotta uuden toiminnallisuuden testaaminen olisi mahdollista jo työn alkuvaiheessa ohjelmoiminen aloitettiin kirjoittamalla kontrolleriluokalle uusi metodi. Metodille annettiin nimeksi: getHtjDataForCompanyWizard, kuvastamaan sen toimintaa, eli hakea huoneistojen tietoja Kiinteistönhallinnan käyttöliittymän alustusvelhon käyttöön. Metodin alkuun lisättiin @GetMapping- merkintä, jotta Spring osaa kuunnella polkuun lähetettyjä GET-tyyppisiä HTTP-pyyntöjä ja näin ollen on vastuussa tiedon lähettämisestä asiakkaalle.

```
@GetMapping(value =
"/companies/{id}/htj/initialization-wizard/getData", produces =
MediaType.APPLICATION_NDJSON_VALUE)
public ResponseEntity<StreamingResponseBody>
getHTJDataForCompanyWizard(@PathVariable("id") String id) {
    String username = PermissionContext.getUsername();
    StreamingResponseBody responseBody = response ->
htjWizardService.streamHTJData(response, id, username);

    return new ResponseEntity<>(responseBody, HttpStatus.OK);
}
```

Esimerkkikoodi 5: Kontrollerikerroksen metodi getHTJDataForCompanyWizard, joka palauttaa StreamingResponseBodyn asiakkaalle.

Esimerkkikoodi 5 esittelee StreamingResponseBodyn käyttöä kontrollerikerroksen metodissa, joka vastaanottaa polkumuuttujana (path variable) asiakasta yksilöivän id:n. StreamingResponseBody käyttää lambda-lauseketta, joka tekee koodista lyhyempää ja selkeämpää. StreamingResponseBody-rajapintaa käytetään suoratoistetun datan luomiseen ja lähettämiseen vastausvirtaan. Tämä mahdollistaa sen, että dataa ei tarvitse ensin ladata kokonaan muistiin, vaan se lähetetään ja kirjoitetaan vastausvirtaan palvelukerroksen luokan HTJWizardService:n kautta sitä mukaa, kun se on käsitelty. Käyttämällä StreamingResponseBody-rajapintaa saavutetaan tehokkuutta ja parempaa skaalautuvuutta, erityisesti suurten datamäärien käsittelyssä tai kun on tarpeen lähettää tietoa ripotel-

len asiakkaalle, kuten juuri latauksen edistymistä seuraavassa toiminnallisuudessa.

Metodin palauttama tietotyyppi on ResponseEntity<StreamingResponseBody>. StreamingResponseBody on kääritty ResponseEntity-luokkaan, jotta vastauksen mukana voidaan lähettää lisätietoa, kuten esimerkiksi HTTP-tilakoodi.

Vastauksen formaatiksi on tässä metodissa ilmoitettu MediaType.APPLICATION_NDJSON_VALUE (Newline Delimited JSON), eli rivinvaihdolla erotettu formaatti, jossa jokaisella rivillä on oma JSON-olio. Vaikka jokainen rivi on itsekseen kelvollista JSON-formaattia, koko tiedosto kokonaisuutena ei ole enää teknisesti kelvollinen JSON, koska se sisältää useita JSON-tekstejä. Perinteiseen synkronisen API-kutsun tuottamaan vastaukseen tottuneen asiakkaan on otettava tämä huomioon vastaanottaessaan asynkronista tietovirtaa.

Seuraavaksi esimerkkikoodi 6 ja esimerkkikoodi 7 havainnollistavat JSON- ja NDJSON-formaattien eroja.

```
[
    {”totalCount”:3,”fetchedCount”:1, ”wizardHtjData”:null},
    {”totalCount”:3,”fetchedCount”:2, ”wizardHtjData”:null},
    {”totalCount”:3,”fetchedCount”:3,
     ”wizardHtjData”:{
          ”huoneisto”:”A1”,
          ”osoite”:”Testikuja 1 A”
          ”osakeryhmätunnus”:”ABCDEFGHJ”,
          ”nimi”:”Erkki Esimerkki”
                     }
     }
]
```

Esimerkkikoodi 6: Perinteinen JSON-muotoinen taulukko (array), jossa useampi JSON-rivi on eroteltu pilkuin ja koko sisältö on kääritty hakasulkeisiin.

```
    {”totalCount”:3,”fetchedCount”:1, ”wizardHtjData”:null}
    {”totalCount”:3,”fetchedCount”:2, ”wizardHtjData”:null}
    {”totalCount”:3,”fetchedCount”:3,
     ”wizardHtjData”:{
          ”huoneisto”:”A1”,
          ”osoite”:”Testikuja 1 A”
          ”osakeryhmätunnus”:”ABCDEFGHJ”,
          ”nimi”:”Erkki Esimerkki”
```

```
                    }
        }
```

Esimerkkikoodi 7: NDJSON-muotoinen tietovirta, jossa yksittäisiä JSON-rivejä ei eroteta pilkuin, eikä niitä ole kääritty hakasulkeisiin.

Kuten esimerkkikoodi 6 esittää, useat JSON-rivit erotellaan pilkuin ja koko sisältö on kääritty hakasulkeisiin. Sen sijaan esimerkkikoodi 7:ssä nähtävässä NJDSON-formaatissa jokainen rivi sisältää yhden JSON-olion, ja rivinvaihtomerkki erottaa ne toisistaan. Tämä helpottaa myös vastauksen käsittelyä, kun kukin olio on selkeästi eroteltu omalle rivilleen. NDJSON on erityisen hyödyllinen, kun halutaan suoratoistaa suurta määrää JSON-olioita. (33.)

4.6 Palvelukerroksen koodi

Luvussa 2.7 esiteltyä Springin kerrosarkkitehtuuria noudattaen tämän insinöörityön liiketoimintalogiikan toteutuksesta vastuussa olevat metodit ja metodi kutsut kirjoitettiin palvelukerroksella jo olemassa olevalle HTJWizardService-luokalle.

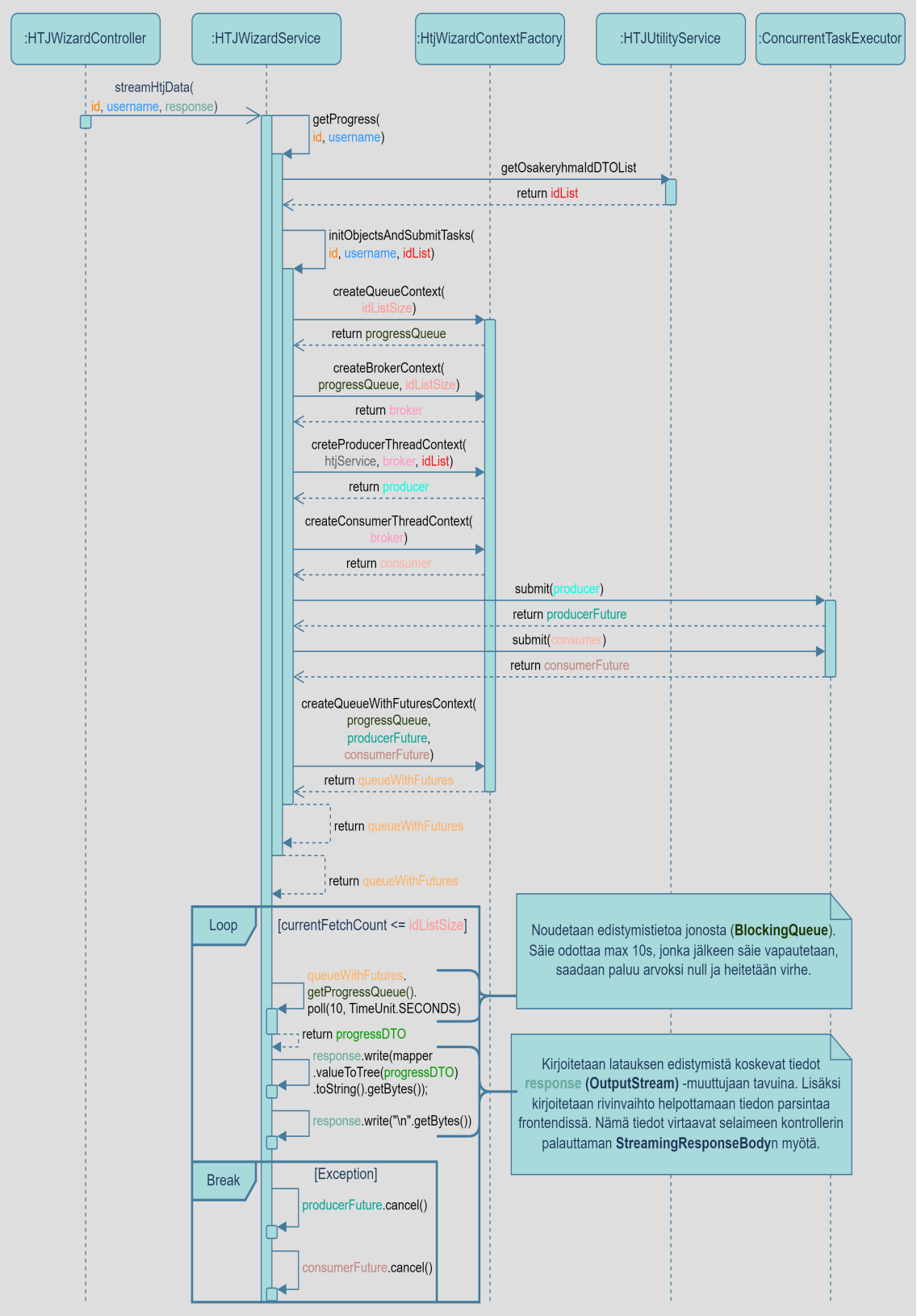
Ennen insinöörityön aloitusta HTJWizardService-luokka piti sisällään useita funktioita, jotka kutsuivat huoneistotietojärjestelmän rajapintoja taloyhtiön huoneistotietojen hakemiseksi ja kartoittamiseksi oikeaan muotoon.

Reaaliaikaisen tiedon suoratoisto palvelimelta selaimeen tietojen haun edistymisestä ei ollut järkevästi toteutettavissa olemassa olevaan ratkaisuun. Tämän vuoksi päädyttiin refaktoroimaan HTJWizardService-luokka lähes kokonaan uudelleen. Tietojen hakua ja kartoitusta varten luotiin erilliset luokat. Refaktorointi toteutettiin osin luvussa 4.3 esiteltyä Producer-Consumer-mallia käyttäen, jossa tietojen hausta vastaa tuottaja eli Producer-luokka ja tiedon kartoituksesta sopivaan muotoon kuluttaja eli Consumer-luokka. Broker-luokka vastaa tietojen haun edistymisen laittamisesta jonoon, joka on jaettu HTJWizardService-luokan kanssa.

Insinöörityössä HTJWizardService-luokkaa refaktoroitiin siten, että sen pääasiallinen vastuu liittyy HTJWizardContextFactory-luokan metodien kutsumiseen, jotka on esitelty luvussa 4.4. Tämä käsittää tarvittavien olioiden luonnin ja Producer- sekä Consumer-luokkien käynnistämisen omiin säikeisiinsä.

HTJWizardServicen toinen merkittävä tehtävä on suoratoistaa tietoa palvelimelta asiakkaalle. Tätä suoratoistoa varten luokalle kirjoitettiin uusi metodi nimeltä streamHTJData. Kuten esimerkkikoodi 5 sivulla 36 esittelee, kontrolleri kutsuu HTJWizardServicen streamHTJData-metodia ja antaa tälle parametrina OutputStream-tyyppiseen response-muuttujan, asiakasta yksilöivän id:n sekä käyttäjätunnuksen.

Seuraavaksi kuva 10 esittelee HTJWizardServicen toimintaa sekvenssikaavion muodossa.



Kuva 10: Sekvenssikaavio palvelukerroksen HTJWizardService-luokan toiminnasta.

Kuten kuva 10 esittää, palvelukerroksen prosessi tiedon suoratoistoon alkaa kontrollerin kutsumalla streamHTJData-metodia. Tämän jälkeen streamHTJData-metodi kutsuu getProgress-nimistä metodia. GetProgress-metodi kutsuu HTJUtilityServicen metodia getOsakeryhmaIdDTOList, jonka paluu arvona saadaan lista osakeryhmätunnuksista. Tämän jälkeen getProgress-metodi kutsuu initObjectsAndSubmitTasks-metodia. Kyseinen metodi kutsuu HtjWizardContextFactory-luokan olioiden luontimetodeita.

Ensin luodaan BlockingQueue-rajapinnan toteuttama jono, jonka koko määräytyy osakeryhmätunnuksien lukumäärän perusteella.

Seuraavaksi luodaan Broker-luokan ilmentymä. Brokerille annetaan luonnin yhteydessä aikaisemmin luotu jono sekä tieto osakeryhmätunnuksien lukumäärästä.

Kolmanneksi luodaan Producer-Consumer-mallin mukainen tuottaja, jota edustaa Producer-luokka. Luonnin yhteydessä Producerille annetaan kolme parametria. Ensimmäinen parametri on htjService, jonka avulla Producer voi noutaa huoneistotietoja huoneistotietojärjestelmästä. Toisena parametrina Producer saa aiemmin luodun Broker-luokan ilmentymän, jonka välityksellä se voi laittaa hakemiaan tietoja jonoon. Kolmas parametri on osakeryhmätunnuksista koostuva lista, jota iteroimalla Producer hakee huoneistotietoja htjServicen metodeita kutsumalla.

Neljäntenä luodaan kuluttajaa edustava Consumer-luokka. Tälle annetaan parametrina aiemmin luotu Broker-luokan ilmentymä, jonka välityksellä Consumer voi hakea Producerin tuottamaa dataa jonosta, tarkastella mahdollisia virheitä Producerissa ja tiedon haun päätteeksi antaa koostamansa olion Brokerin metodille, jonka kautta koosteolio huoneistotiedoista päätyy alustusvelhon käyttöön.

Näiden luontioperaatioiden jälkeen Producer-ja Consumer-luokat käynnistetään ConcurretTaskExecutorin avulla omiin säikeisiinsä. ConcurretTaskExecutor palauttaa Future-oliot molemmista säikeistä.

Viidentenä luontioperaationa luodaan QueueWithFutures-luokan ilmentymä. Se saa parametrinaan ensimmäisessä vaiheessa luodun jonon, sekä Producerille ja Consumerille käynnistettyjen säikeiden Futuret. InitObjectsAndSubmitTasks-metodi palauttaa QueueWithFutures-olion getProgress-metodille, joka puolestaan palauttaa sen streamHTJData-metodille.

Saatuaan QueueWithFutures-olion StreamHTJData-metodi aloittaa iteroinnin ja aina kun QueueWithFutures-olion sisältämään jonoon saapuu elementti, se kirjoitetaan kontrollerilta saatuun response-muuttujaan. Elementin kirjoituksen jälkeen responseen kirjoitetaan rivinvaihto helpottamaan tiedon parsintaa asiakaspuolella. Näin responseen kirjoitettua tietoa suoratoistetaan asiakaspuolelle.

QueueWithFutures-olion sisältämiä Futureita käytetään mahdollisessa virhetilanteessa Consumerin ja Producerin säikeiden sulkemiseen, jotta ne eivät jää taustalle varaamaan resursseja.

5 Asiakaspuolen ohjelman kirjoitus

Kiinteistönhallinta-sovelluksen asiakkaalle selaimessa näkyvä osa on tärkeässä roolissa käyttäjäystävällisyyden kannalta. Alustusvelhon tarkoituksena on aloittaa alustusprosessi, joka mahdollistaa käyttäjälle helpon pääsyn ja hallinnan huoneistotietoihin. Käyttäjän käynnistäessä alustusvelhon käyttöliittymästä käsin selaimen suorittama koodi lähettää palvelimelle pyynnön, joka aloittaa huoneistojen haun ja kartoituksen. Pyynnön käsittely palvelimella voi viedä kuitenkin paljon aikaa ja käyttäjä joutuu odottamaan pitkään vastauksen saapumista nähtäväkseen selaimeen, joten reaaliaikaisen latauspalkin visuaalinen toteutus tuli kriittiseksi osaksi käyttäjäystävällisyyttä.

Kiinteistönhallinnan sovelluksen asiakaspuolen koodi toteutettiin luvussa 2.5 esiteltyjä teknologioita eli Angular-ohjelmistokehystä sekä TypeScript-ohjelmointikieltä käyttäen. Asiakaspuolen koodi, joka vastasi huoneistojen tietojen hausta ja näyttämisestä alustusvelhon käyttöön, oli jo olemassa. Kuitenkin alkuperäinen toteutus teki kutsuja palvelimelle perinteiseen synkroniseen rajapintaan. Tämän vuoksi asiakaspuolella olevaa tiedon vastaanottavaa osaa täytyi uudelleenkirjoittaa, jotta se olisi sopiva vastaanottamaan asynkronista tietovirtaa.

5.1 Tietovirran vastaanottaminen ja muunnos asiakaspuolella

Asiakaspuolella tiedon vastaanottaminen alkaa perinteisellä HTTP GET -kutsulla rajapintaan huoneistotietojen hakua varten. Tämän jälkeen ohjelma tilaa (subscribe) palvelimelta tulevaa tietovirtaa.

Palvelimelta selaimeen saapuva tieto on esimerkkikoodi 7:ssä esiteltyä NDJSON-formaattia, joka on käännetty tavukoodiksi. Koska tieto saapuu asynkronisesti pieninä paloina (chunk), ei voida olettaa yhden tavukoodin palan edustavan palvelinpuolen vastaavaa kokonaista Java-olioita. Jokainen pala sisältää olion tietoja, mutta niitä ei voida kääntää kokonaiseksi validiksi JavaScriptin

JSON-olioksi, ennen kuin ne on kokonaan vastaanotettu. Toisaalta yksi selaimeen saapunut pala voi myös sisältää useampia tavumuotoisia rivinvaihdolla erotettuja JSON-tyyppisiä paloja.

Prosessin monimutkaisuuden vuoksi palvelimelta saapuvan tavukoodin täytyy kulkea esimerkkikoodi 8:ssa esitellyn kolmen erilaisen putken (pipe) läpi. Tämän jälkeen vastaanotettua tietoa voidaan helposti käsitellä ja visualisoida selaimen suorittamassa koodissa.

```
this.htjWizardService.getAPIDataStream().subscribe(async (result) => {
  // Muunna tavumuoto tekstiksi:
  const textData = result.body.pipeThrough(new TextDecoderStream());

  // Puskuroi dataa rivinvaihdolla erotetuiksi paloiksi:
  const bufferedData = textData.pipeThrough(this.splitStream('\n'));

  // Parsi palat JSON-muotoisiksi:
  const jsonResult = bufferedData.pipeThrough(this.parseJSON());

  // Luo lukija käsittelemään tuloksia:
  const reader = jsonResult.getReader();
  // Koodi jatkuu…
```

Esimerkkikoodi 8: Palvelimelta saapuva tietovirta kulkee kolmen putken läpi

Ensimmäinen putki muuntaa tietovirtaan saapuvat tavukoodin palat merkkijonomuotoon. Tämän jälkeen kyseinen merkkijonomuotoinen virta annetaan seuraavalle putkelle jatkokäsittelyyn.

Toinen putki vastaanottaa ensimmäiseltä putkelta merkkijonoksi muunnettua tietovirtaa ja toimii puskurina. Se suodattaa saapuvia tiedon palasia ja kerää niitä, kunnes tietovirtaan saapuu uusi rivi. Rivinerotin ilmoittaa, että siihen asti kerätyt merkkijonot muodostavat sisällöltään yhden kokonaisen palvelinpuolen Java-olion. Tämä putki pilkkoo rivinvaihdolla erotetut merkkijonot ja lähettää ne yksitellen kolmannelle putkelle.

Kolmas putki muuntaa toiselta putkelta saadun merkkijonon validiksi JavaScriptin JSON-olioksi. Tällä tavoin palvelimelta vastaanotettu tietovirta käy läpi sarjan muunnoksia, ennen kuin se saavuttaa sovelluksen käyttölogiikan.

5.2 Tiedon käsittely ja visualisointi

Kun palvelimelta vastaanotettua NDJSON-muotoista tavuvirtaa on pilkottu JSON-muotoon luvussa 5.1 esiteltyjen putkien kautta voidaan keskittyä sovelluksen käyttölogiikkaan ja visualisointiin käyttöliittymässä. Käsiteltävät JSON-oliot vastaavat sisällöltään sivun 37 esimerkkikoodi 7:ssä esiteltyä NDJSON-muotoista dataa.

Sivulla 44 esimerkkikoodi 8:n lopussa näkyvä reader-muuttuja toimii lukijana. Se vastaa tietovirtaan saapuneiden elementtien lukemisesta. Esimerkkikoodi 9 esittelee, kuinka readerin read()-metodia kutsutaan while-iteraatiossa, mikä käynnistää JSON-olioiden käsittelyn.

```
//...jatkoa
while (true) {

// Odotetaan että lukijalle saapuu elementtejä
  const { done, value } = await reader.read();

// Jos valmista annetaan käyttöliittymään huoneistotiedot
  if (done) {
    this.ut.toastSuccess('Valmis! Huoneistotiedot on tuotu
onnistuneesti järjestelmään. Kiitos että jaksoit odottaa!');

    let wizardHTJData = this.progressJSON.wizardHTJData;
    this.htjWizardData =
this.htjWizardService.getHTJForm(wizardHTJData);

    break;
  }

// Käsitellään latauspalkin päivittämistä
  this.processStream(value);
}
```

Esimerkkikoodi 9: Luetaan tietovirtaan tulevia elementtejä

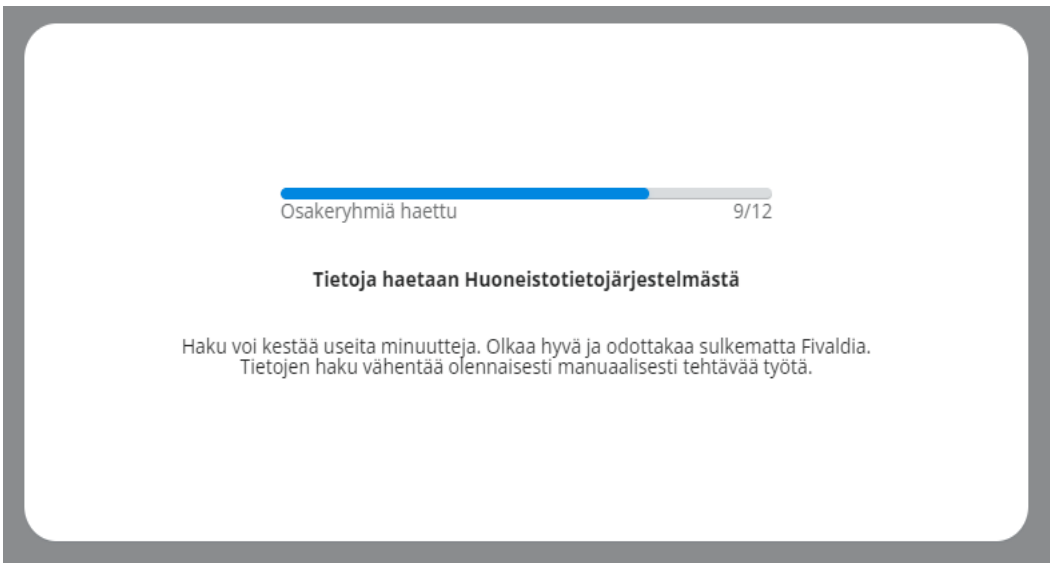
Esimerkkikoodi 9 esittää, kuinka reader-lukija lukee tietovirtaan saapuvia elementtejä. Metodikutsun reader.read() eteen on lisätty await-avainsana, mikä tarkoittaa, että lukija odottaa, kunnes JSON-oliot ovat käytettävissä. ProcessStream-metodi vastaanottaa nämä JSON-oliot latauspalkin päivittämistä varten.

```
 processStream(progressJSON) {
    this.totalCount = progressJSON.totalCount;
```

```
    this.currentCount = progressJSON.fetchedCount;
    this.progressCount = (this.fetchedCount * 100) / this.totalCount;
  }
```

Esimerkkikoodi 10: progressCount-muuttuja ilmaisee latauspalkin edistymistä

Esimerkkikoodi 10:n esittelemä metodi parsii JSONista latauksen edistymistä koskevat tiedot ja vastaa latauspalkin päivittämisestä näkymään. Angular seuraa this.progressCount-muuttujaa, ja aina kun sen arvo muuttuu, Angular automaattisesti päivittää näkymän. Tämä dynaaminen päivitys latauspalkissa toteutetaan Angularin taustalla olevan Change Detection -mekanismin avulla. Mekanismi havaitsee muutokset seurattaviin muuttujiin ja päivittää näkymän välittömästi, jotta käyttöliittymä vastaisi aina ajankohtaista tilannetta. (34.)



Kuva 11: Käyttäjä saa reaaliaikaista tilannekuvaa haettujen osakeryhmien määrästä latauspalkin muodossa

Kuva 11 näyttää lopullisen ilmeen reaaliaikaiselle latauspalkille, joka ilmestyy taloyhtiön tietojen haun ajaksi korvaten aiemmin kuvassa 1 sivulla 2 esitellyn spinnerin. Latauspalkin sininen osa kasvaa jokaisen haetun osakeryhmän tietojen kohdalla samalla, kun latauspalkin alapuolella oikealla oleva numero kasvaa. Latauspalkin alapuolella oleva numero 9 viittaa senhetkiseen määrään ja numero 12 on haettavien osakeryhmien tietojen kokonaismäärä. Kun latauspalkki on täynnä, käyttäjä saa ilmoituksen onnistuneesta tietojen hausta. Tämän

jälkeen käyttöliittymässä avautuu asetusvelho, joka vastaa kuvan 2 esitystä sivulla 6.

Liite 1 sisältää yksinkertaistetun sekvenssikaavion, jossa tiedon hakuprosessi huoneistotietojärjestelmästä, sen käsittely ja asynkroninen suoratoisto käyttöliittymään tapahtuvat.

6 Yhteenveto

Insinöörityössä selvitettiin mahdollisuutta hyödyntää asynkronista REST-ohjelmointirajapintaa tiedon suoratoistoon pitkään kestävässä tiedonhaussa Fivaldi-taloudenhallintaohjelmistoon. Lisäksi selvitettiin, pystyisikö tämän suoratoiston avulla luomaan reaaliaikainen edistymispalkki. Edistymispalkin oli tarkoitus tulla Kiinteistönhallinta-sovelluksen taloyhtiön alustusvelhotyökaluun. Alustusvelho hakee taloyhtiön tietoja Maanmittauslaitoksen huoneistotietojärjestelmästä ja tässä prosessissa saattaa kulua useita minuutteja. Asynkronisen REST-ohjelmointirajapinnan hyödyntäminen tiedon suoratoistoon Fivaldissa onnistui ja tuloksena saatiin palvelimella tapahtuvan tiedon haun reaaliaikaista edistymistä visualisoiva latauspalkki käyttöliittymään. Edistymispalkilla onnistuttiin korvaamaan tietojen haun aikana aikaisemmin pyörinyt spinneri, joka ei indikoinut prosessin etenemistä tai siihen kuluvaa aikaa mitenkään, mikä aiheuttaa käyttäjien turhautumista. Lisäksi insinöörityössä havaittiin yhtäaikaisesti ajettavan monisäikeistetyn koodin tuottavan huomattavasti nopeampaa tiedonkäsittelyä ja hakua.

Toistaiseksi projektissa on käytössä Java 17 -versio, mutta projektin versiopäivityksen myötä tulevaisuudessa tutkimusta voisi jatkaa korvaamalla useiden säikeiden käyttöä Java 21 -versioon tulleilla virtuaalisäikeillä, jotka ovat paljon kevyempiä ja tehokkaampia kuin perinteiset käyttöjärjestelmäsäikeet.

Fivaldin sovelluksissa käytetään edelleen monin paikoin spinnereitä, jotka muistuttavat kuvan 1 esitystä sivulla 2. Insinöörityön tuloksen perusteella näiden vanhentuneiden latausindikaattorien korvaaminen moderneilla latauspalkkeilla voisi olla perusteltua parantamaan käyttökokemusta odotusaikojen hallinnassa ja tarjoamaan käyttäjälle välittömän käsityksen prosessin etenemisestä.

Tulevaisuuden suunnitelmiin kuuluu myös hyödyntää asynkronisia REST-rajapintoja Fivaldissa kehitettävien widgettien tietojen noutoon ja prosessointiin.

Web-sovelluksissa widgetit ovat pieniä, itsenäisiä osia käyttöliittymässä, jotka tarjoavat tiettyjä toiminnallisuuksia tai näyttävät tietoja. Niitä käytetään usein parantamaan käyttäjäkokemusta ja tarjoamaan nopeaa pääsyä tiettyihin toimintoihin tai tietoihin. Esimerkiksi kassavirtaennuste on yksi suunniteltu widget Fivaldiin. Kassavirtaennuste vaatii suuren määrän tietojen hakua tietokannasta ja sen prosessointia. Asynkronisen REST-rajapinnan käyttäminen tällaisessa käyttötapauksessa mahdollistaa nopeamman pääsyn osaan tiedoista, eikä käyttäjän tarvitse odottaa, että kaikki tiedot palautuvat kerralla palvelimelta.

Widgetissä voi olla useita sivuja, mikä mahdollistaa käyttäjälle nopean pääsyn ensimmäisen sivun tietoihin samalla, kun muut sivut latautuvat taustalla ajan kanssa. Tämä merkittävästi parantaa käyttökokemusta, kun käyttäjä voi välittömästi tarkastella osaa datasta ja samalla odottaa muiden sivujen sisältöjen latautumista.

Lisäksi Fivaldissa on monia muita pitkään kestäviä prosesseja, joita hoidetaan nykyisin yöajoilla. Näiden korvaaminen monisäikeistetyllä koodilla ja suoratoistolla, voisi tarjota reaktiivisemman ja reaaliaikaisemman tiedon tarjoamisen käyttäjille.

Lähteet

1 Huoneistotietojärjestelmä tuo taloyhtiöiden ja osakehuoneistojen tiedot yhteen. Verkkoaineisto. <https://www.maanmittauslaitos.fi/huoneistotietojarjestelma>. Luettu 2.1.2024.

2 Sanastoa. Verkkoaineisto. <https://osakehuoneistorekisteri.fi/sanastoa>. Luettu 2.1.2024.

3 1.Introduction. 2016. Verkkoaineisto. <https://doc.akka.io/docs/akka-http/current/introduction.html>. Päivitetty 26.11.2016. Luettu 6.1.2024.

4 Sapra, Puneet. 2018. Library — Framework and Toolkit. Verkkoaineisto. <https://medium.com/the-mighty-programmer/shared-code-framework-and-toolkit-f07b21d07ba9>. Päivitetty 25.1.2018. Luettu 6.1.2024.

5 Edwin, Njeru Mwendi. 2014. Journal of Software Engineering and Applications Vol.7 No.8. Verkkoaineisto. <https://www.scirp.org/journal/PaperInformation?PaperID=47999>. Päivitetty 22.7.2014. Luettu 2.1.2024.

6 Fowler, Martin. 2005. Inversion Of Control. Verkkoaineisto. <https://martinfowler.com/bliki/InversionOfControl.html>. Päivitetty 26.6.2005. Luettu 6.1.2024.

7 Introduction to Angular concepts. Verkkoaineisto. <https://angular.io/guide/architecture>. Luettu 2.1.2024.

8 What is Java technology and why do I need it?. Verkkoaineisto. <https://www.java.com/en/download/help/whatis_java.html>. Luettu 2.1.2024.

9 McCluskey, Glen. 1998. Using Java Reflection. Verkkoaineisto. <https://www.oracle.com/technical-resources/articles/java/javareflection.html>. Päivitetty 1 /1998. Luettu 2.1.2024.

10 Thakral, Kartik. 2022. Introduction to Java Servlets. Verkkoaineisto. <https://www.geeksforgeeks.org/introduction-java-servlets>. Päivitetty 12.9.2022. Luettu 2.1.2024.

11 Shenoy, Anjana. Lead Curriculum Developer. Moxley, Susan. Editor. Pillai, Anju Subbiah.QA. Java EE 7: Using Non-blocking I/O in the Servlet 3.1 API. Verkkoaineisto. <https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/HTML5andServlet31/HTML5andServlet%203.1.html>. Luettu 2.1.2024.

12 Spring WebFlux. Verkkoaineisto. <https://docs.spring.io/spring-framework/reference/web/webflux.html>. Luettu 2.1.2024.

13 Srivastava, Prashant. 2022. Difference between Spring MVC and Spring Boot. Verkkoaineisto. <https://www.geeksforgeeks.org/difference-between-spring-mvc-and-spring-boot/>. Päivitetty 31.5.2022. Luettu 2.1.2024.

14 Helsingin Yliopiston Agile Education Research -tutkimusryhmä. 2018. Sovelluksen rakenne ja pyynnön kulku sovelluksessa. Verkkoaineisto. <https://materiaalit.github.io/wepa-s17/part4/>. Päivitetty 28.5.2018. Luettu 2.1.2024.

15 Stoyanchev, Rossen. Interface StreamingResponseBody. Verkkoaineisto. <https://docs.spring.io/spring-framework/docs/5.0.10.RELEASE_to_5.0.11.RELEASE/Spring%20Framework%205.0.11.RELEASE/org/springframework/web/servlet/mvc/method/annotation/StreamingResponseBody.html>. Luettu 3.10.2023.

16 Package java.util.function Description. Verkkoaineisto. <https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html>. Luettu 2.1.2024.

17 Chandrakan, Kumar. 2021. Guide to Java OutputStream. Verkkoaineisto. <https://www.baeldung.com/java-outputstream>. Päivitetty 19.5.2021. Luettu 3.1.2023.

18 Streaming Data with Spring Boot RESTful Web Service. Verkkoaineisto. <https://technicalsand.com/streaming-data-spring-boot-restful-web-service/>. Päivitetty 5.10.2020. Luettu 2.1.2024.

19 Stoyanchev, Rossen. Hoeller, Juergen. Class ResponseBodyEmitter. Verkkoaineisto. <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/mvc/method/annotation/ResponseBodyEmitter.html>. Luettu 3.10.2023.

20 1.1.3. HttpMessageConverters. Verkkoaineisto. <https://docs.spring.io/spring-boot/docs/current/reference/html/web.html#web.servlet.spring-mvc.message-converters>. Luettu 3.10.2023.

21 Roza, Ger. 2023. Server-Sent-Events-in-Spring. Verkkoaineisto. <https://www.baeldung.com/spring-server-sent-events>. Päivitetty 28.9.2023. Luettu 3.10.2023.

22 WebSockets. 2023. Verkkoaineisto. <https://docs.spring.io/spring-framework/reference/web/webflux-websocket.html>. Luettu 3.10.2023.

23 Kocak, Burak. 2023. Spring Web vs Spring Webflux. Verkkoaineisto. <https://medium.com/@burakkocakeu/spring-web-vs-spring-webflux-9224260c47b5>. Päivitetty 8.3.2023. Luettu 3.10.2023.

24 Garai, Arpendu Kumar. 2022. Getting Started with Spring WebFlux. Verkkoaineisto. <https://reflectoring.io/getting-started-with-spring-webflux/>. Päivitetty 10.3.2022. Luettu 3.10.2023.

25 Oracle Corporation. 2010. Thread Safety. Verkkoaineisto. <https://docs.oracle.com/cd/E19455-01/806-5257/6je9h033e/index.html>. Luettu 2.1.2024.

26 Hoeller, Juergen. Class ConcurrentTaskExecutor. Verkkoaineisto. <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/concurrent/ConcurrentTaskExecutor.html>. Luettu 2.1.2024.

27 Jeyapal, Karthik. 2023. System Design Patterns: Producer Consumer Pattern. Verkkoaineisto. <https://medium.com/@karthik.jeyapal/system-design-patterns-producer-consumer-pattern-45edcb16d544>. Päivitetty 15.4.2023. Luettu 2.1.2024.

28 BlockingQueue. Verkkoaineisto. <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html>. Luettu 2.1.2024.

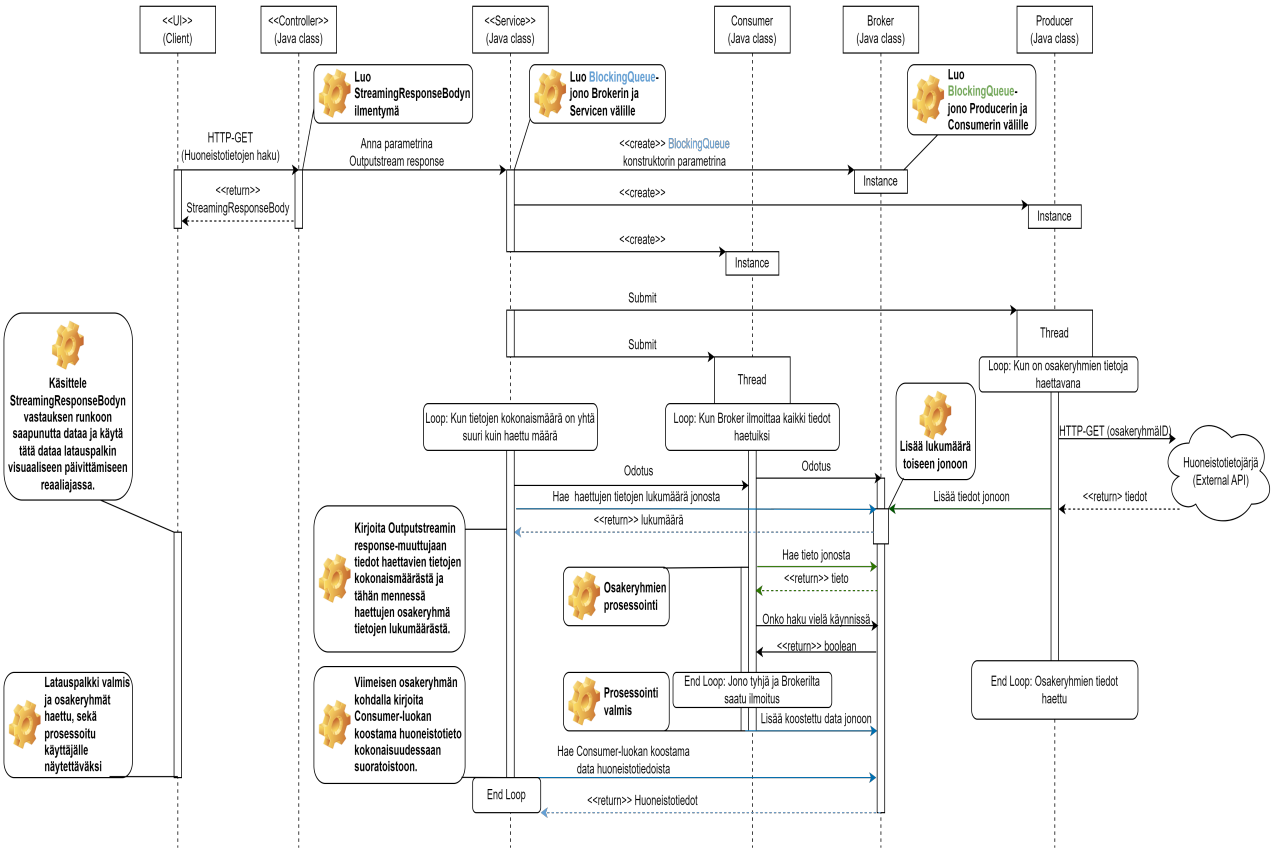
29 17.Spring Beans and Dependency Injection. Verkkoaineisto. <https://docs.spring.io/spring-boot/docs/2.0.x/reference/html/using-boot-spring-beans-and-dependency-injection.html>. Luettu 22.10.2023.

30 Bean scopes Verkkoaineisto. <https://docs.spring.io/spring-framework/docs/3.0.0.M3/reference/html/ch04s04.html>. Luettu 22.10.2023.

31 Baeldung. The Spring @Controller and @RestController Annotations. Verkkoaineisto. <https://www.baeldung.com/spring-controller-vs-restcontroller>. Päivitetty 28.12.2023. Luettu 2.1.2024.

32 Mapping Requests. Verkkoaineisto. <https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-requestmapping.html>. Luettu 2.1.2024.

33 Santagostino, Jaga. 2018. Newline delimited JSON is awesome. Verkkoaineisto. <https://medium.com/@kandros/newline-delimited-json-is-awesome-8f6259ed4b4b>. Päivitetty 20.1.2018. Luettu 3.10.2023.

34 Angular change detection and runtime optimization. Verkkoaineisto. <https://angular.io/guide/change-detection>. Päivitetty 4.5.2022. Luettu 2.1.2024.

Yksinkertaistettu sekvenssikaavio tiedon käsittelystä ja suoratoistosta asynkronisesti käyttöliittymään latauspalkkia varten