

Henrik Heikkilä

Verkkokauppojen käyttäjästatistiikan reaaliaikainen visualisointi

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikan koulutusohjelma

Insinöörityö

1.5.2016

Tekijä(t) Otsikko Sivumäärä Aika	Henrik Heikkilä Verkkokauppojen käyttäjästatistiikan reaaliaikainen visualisointi 32 sivua 1.5.2016
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Lehtori Simo Silander Team leader Jaakko Kourula
<p>Insinöörityön tavoitteena oli parantaa Smilehouse Oy:n sisäisten työkalujen valikoimaa luomalla sovelluskokonaisuus, jolla voidaan reaaliaikaisesti visualisoida eri verkkokauppojen käyttäjien toimintaa eri puolilla maapalloa.</p> <p>Insinöörityössä määriteltiin kävijätiedon visualisointiin tarkoitettu sovellus, valittiin sopivat ohjelmistokehykset ja muut toteutuksessa käytettävät työkalut ja ohjelmistot. Sovellus päätettiin jakaa useampaan eri komponenttiin ja toteuttaa suurimmaksi osaksi Spring-ohjelmistokehyksen päälle.</p> <p>Insinöörityössä kerrottiin sovelluksen toteuttamisesta, suorituskykyoptimoinnista, testauksesta ja siitä, kuinka liian tiukat aikataulut vaikuttavat ohjelmistokehityksessä. Insinöörityössä käytiin läpi myös valittujen ohjelmistokehysten, työkalujen ja rajapintojen käyttötapoja, hyviä ja huonoja puolia.</p>	
Avainsanat	Visualisointi, reaaliaikaisuus, WebSocket

Author(s) Title	Henrik Heikkilä Real-time visualization of web shop user statistics
Number of Pages Date	32 pages 1 May 2016
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Simo Silander, Senior Lecturer Jaakko Kourula, Team leader
<p>Objective of this thesis was to improve the collection of internal tools used by Smilehouse Oy by creating an application that can be used to visualize the behavior of different web shops customers across the globe.</p> <p>This thesis defined an application for web shop visitor information visualization, choosing suitable frameworks and other tools and software for implementation. The application was decided to be divided into several different components and to be implement mostly on top of the Spring framework.</p> <p>This thesis explained the implementation of the application, performance optimization, testing and how too tight schedules affect software development. This thesis also reviewed the selected software frameworks, tools and interfaces, their uses, pros and cons.</p>	
Keywords	Visualization, real-time, WebSocket

Sisällys

Lyhenteet

1	Johdanto	1
2	Toiminnallisten vaatimusten määrittely	2
3	Sovelluksen tekninen määrittely	3
3.1	Sovelluksen käyttöympäristöt	3
3.2	Sovelluksen jako komponentteihin	4
3.2.1	Ydinkomponentti	5
3.2.2	Datankerääjäkomponentti	6
3.2.3	Visualisointikomponentti	6
3.3	Komponenttien välinen kommunikointi	6
3.3.1	REST	7
3.3.2	WebSocket	7
3.3.3	STOMP	7
3.4	Rajapintakuvaukset	8
3.4.1	WebSocket-yhteys	8
3.4.2	REST-rajapinta	8
3.5	Ympäristöt	9
3.6	Kerättävä data	9
4	Sovelluksen toteutus	10
4.1	Kehitystyökalut	10
4.2	Projektin rakenne	10
4.3	Sovelluksen tekninen rakenne	11
4.4	Datan lukeminen palvelinten lokitiedostoista	12
4.5	Datan tallentaminen tietokantaan	13
4.6	Komponenttien välinen kommunikointi	15
4.6.1	Datankerääjäkomponentin ja ydinkomponentin välinen yhteys	15
4.6.2	Ydinkomponentin WebSocket-yhteys	16
4.6.3	Ydinkomponentin REST-rajapinta	17
4.7	Suorituskykyoptimointi	19
4.7.1	Suorituskykyoptimoinnin lähtökohta	19

4.7.2	Monisäikeistäminen	20
4.7.3	Suorituskykyoptimoinnin lopputulos	22
4.8	Visuaalinen toteutus	23
4.8.1	Reaaliaikainen kartta	23
4.8.2	Tiheyskartta	24
4.8.3	Kartan interaktiivisuus	26
4.9	Sovelluksen testaus	27
4.9.1	Datankerääjäkomponentin testaus	27
4.9.2	Ydinkomponentin testaus	28
4.9.3	Visualisointikomponenttien testaus	28
4.9.4	Havaintoja ja huomioita testauksesta	29
5	Sovelluksen käyttöönotto	29
5.1	Sovelluksen tuotantoon siirto	29
5.2	Tulevat tuotantoon siirrot	30
5.3	Sovelluksen tulevaisuus	30
6	Yhteenveto	31
	Lähteet	32

Lyhenteet

HTML	HyperText Markup Language. Erityisesti internetsivuissa käytetty merkin- täkieli.
HTTP	Hypertext Transfer Protocol. Hypertekstin siirtoprotokolla.
JPA	Java Persistence API. Relaatiodatan hallintaa määrittelevä ohjelmointira- japinta.
JSON	JavaScript Object Notation. Yksinkertainen tiedostomuoto tietorakenteiden säilytykseen.
POM	Project Object Model. Maven-projekteissa käytetty XML-esitysmuoto pro- jektista.
REST	Representational State Transfer. Arkkitehtuurimalli ohjelmointirajapintojen toteuttamiseen.
STOMP	Simple Text-orientated Messaging Protocol. Tekstipohjainen viestiproto- kolla.
TCP	Transmission Control Protocol. Tietoliikenneprotokolla, jolla voidaan luoda yhteyksiä verkossa olevien tietokoneiden välille.
XML	Extensible Markup Language. Tietynlaisen merkinmäkielen yläkäsite.

1 Johdanto

Tässä insinöörityössä määritellään ja toteutetaan sovelluskokonaisuus, jolla voidaan visualisoida samanaikaisesti useiden eri verkkokauppojen käyttäjätilastoja. Sovellus toteutetaan ensisijaisesti Java-ohjelmointikielellä ja Spring-ohjelmistokehyksellä.

Tarve tälle sovellukselle syntyi, kun verkkokauppoja ja muita eCommerce-ratkaisuja tuottava Smilehouse Oy halusi kehittää sisäisiä työkalujaan, joita voidaan hyödyntää asiakkaiden lähinnä kansainvälisten tarpeiden selvittämisessä ja hyödyllisen käyttäjädatan keräämisessä.

Yksi näistä tarpeista oli sisäinen työkalu, jolla voidaan visuaalisesti havainnoida eri verkkokauppojen käyttäjien toimintaa, ajankohtia ja maantieteellisiä sijainteja. Erityisesti kansainvälisten asiakkuuksien kohdalla käyttäjien visuaalinen havainnollistaminen maailmankartalla on hyödyksi.

Alkuperäiset ideat ja suunnitelmat tälle sovellukselle olivat paljon laajemmat, joten tässä työssä käsitellään sovelluksen ensimmäistä vaihetta: eri verkkokauppojen kävijätilastojen visualisointia eri tavoin. Koska sovelluksen jatkokehitystarpeet ja ideat ovat jo tässä vaiheessa tiedossa, käsitellään tässä työssä myös eri mahdollisuuksia ja tapoja jatkokehitykselle.

Muita ideoita sovelluksen toiminnollisuuksille oli kerätä dataa ja visualisoida myös muista asioista kuin vierailuista, esimerkiksi sivustokäyttäytymiseen liittyen, kuinka monella eri sivulla yksittäinen sivuston käyttäjä vierailee vierailunsa aikana ja miten erilaisia nämä ovat eripuolilla maailmaa.

Tämä aihe valittiin insinöörityöksi, koska työn tekijää kiinnosti erilaisen datan kerääminen, käsittely ja visualisointi. Smilehouse Oy:llä oli ajatus työkalusta, joka vastasi näitä. Insinöörityön tavoitteena on toteuttaa sovelluskokonaisuus, joka täyttää nämä kriteerit ja parantaa työn tekijän ammattitaitoa ohjelmistotalalla.

2 Toiminnallisten vaatimusten määrittely

Tässä luvussa käydään läpi sovellukselle asetetut toiminnallisten vaatimusten määrittelyt. Sovelluksessa on seuraavia perusvaatimuksia:

- Sovelluksen on tuettava monia eri samanaikaisia tietolähteitä/verkkokauppoja.
- Sovellus tarjoaa käyttäjälle visuaalisesti tietoa verkkokauppojen käyttäjistä.
- Loppukäyttäjälle näkyvä data on oltava riittävän anonymiä.
- Sovelluksen on tarjottava selkeät rajapinnat helpolle jatkokehitykselle.
- Sovellus ei saa vaarantaa verkkokauppojen toimivuutta tai tietoturvaa.
- Sovellus tulee ensisijaisesti vain yrityksen sisäiseen käyttöön.

Smilehousen toimittamia verkkokauppoja on useita, ja jokainen verkkokauppa on erilainen: on erilaisia aihealueita, kokoja, maakohtaisia ja kansainvälisiä kauppia. Tämän vuoksi on hyödyllistä kerätä dataa mahdollisimman monesta eri verkkokaupasta samanaikaisesti, jotta erilaisten kauppiaiden asiakkaiden käyttäytymiseroja voidaan verrata sekä verkkokauppojen että maantieteellisten sijaintien suhteen.

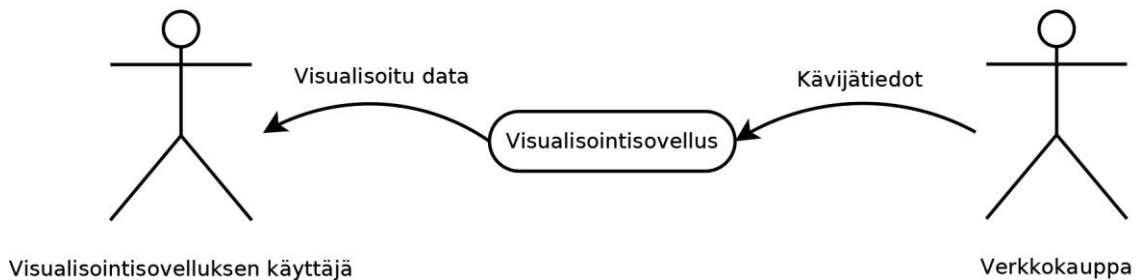
Sovelluksen loppukäyttäjä on tässä tapauksessa ihminen, ja ihmisen on helpompi ymmärtää tietoa, joka on visualisoitu ymmärrettävään muotoon. Tämän vuoksi on tärkeää, että kerätty data on myös visualisoitu selkeästi ja helposti ymmärrettävästi.

Loppukäyttäjälle näkyvä data on oltava riittävän anonymiä, jotta loppukäyttäjä ei pysty saamastaan datasta yksilöimään tai tunnistamaan verkkokauppojen yksittäisiä käyttäjiä, eli loppukäyttäjä ei esimerkiksi näe kävijöiden IP-osoitteita, eikä tietyn tietämänsä IP-osoitteen käyntitietoja.

Jo ennen sovelluksen suunnittelun aloitusta oli selvää, että sovellusta tullaan jatkokehittämään myöhemmin sitä mukaa, kun uusia tarpeita ja ideoita tulee esille. Tämän johdosta on sovelluksen suunnittelussa huomioitava myös jatkokehityksen helppous.

Hyvä tietoturva on tärkeä internetissä olevien asioiden kanssa erityisesti silloin, kun on kyse rahasta ja henkilötiedoista, joita kumpiakin käsitellään paljon verkkokaupoissa. Sovelluksen suunnittelussa on otettava huomioon tämä, ja sovellus on suunniteltava siten, ettei se heikennä kauppiaiden tietoturvaa.

Sovellus on ensisijaisesti tarkoitettu vain yrityksen sisäiseen käyttöön, sillä julkisen käytön salliminen vaatisi mahdollisesti ylimääräisiä sopimuksia. Lisäksi sovelluksen julkinen käyttö korostaisi sovellukseen liittyviä tietoturvariskejä huomattavasti.



Kuva 1. Sovelluksen käyttäjät ja osapuolet

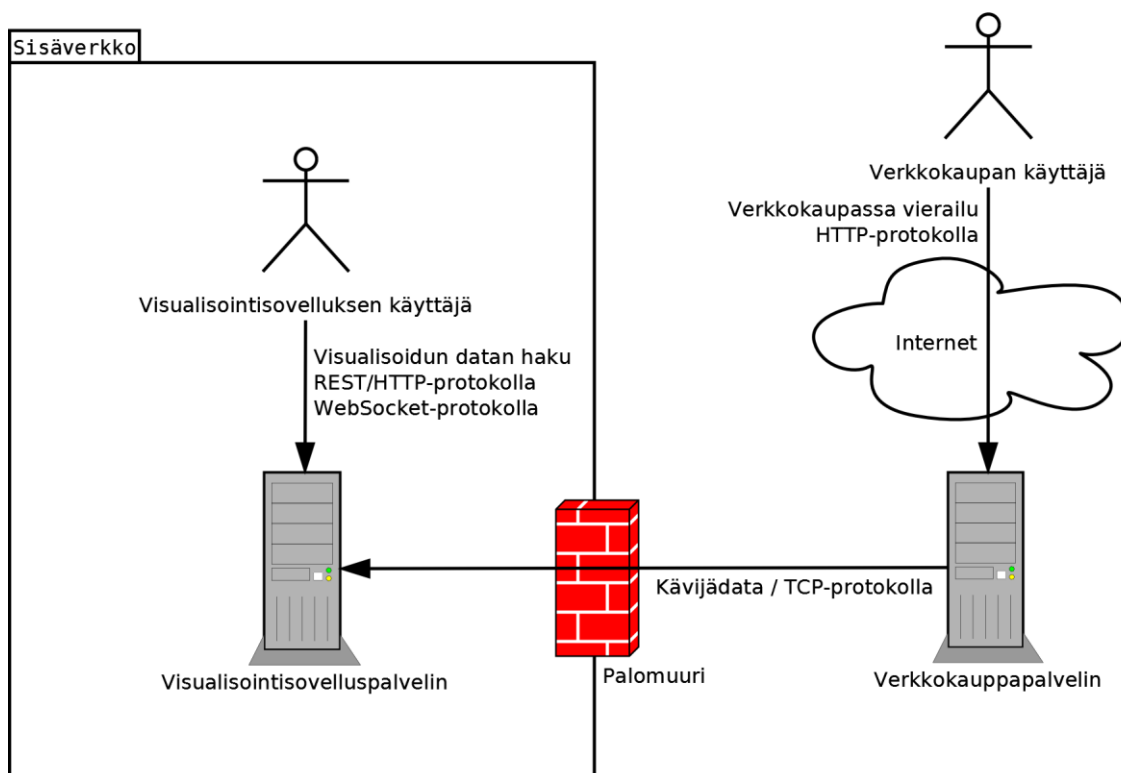
Sovelluksen käyttäjä pystyy katselemaan sovelluksesta visualisoitua dataa, joka on peräisin verkkokauppojen kävijätiedoista (kuva 1).

3 Sovelluksen tekninen määrittely

Tämä luku sisältää toteutettavan visualisointisovelluksen määrittelyn. Tähän kuuluvat toteutettavan sovelluskokonaisuuden eri komponenttien tekniset määrittelyt. Tässä luvussa käydään läpi myös komponenttien välisten rajapintojen ja käsiteltävän datan määrittelyt.

3.1 Sovelluksen käyttöympäristöt

Sovellusta ajetaan sisäverkossa sijaitsevalla sovelluspalvelimella, johon vain sisäverkossa olevat käyttäjät pääsevät. Sovellus vastaanottaa käyttäjädataa eri verkkokauppa-palvelimilta, jotka kuitenkin voivat sijaita eri paikoissa, myös sisäverkon ulkopuolella.



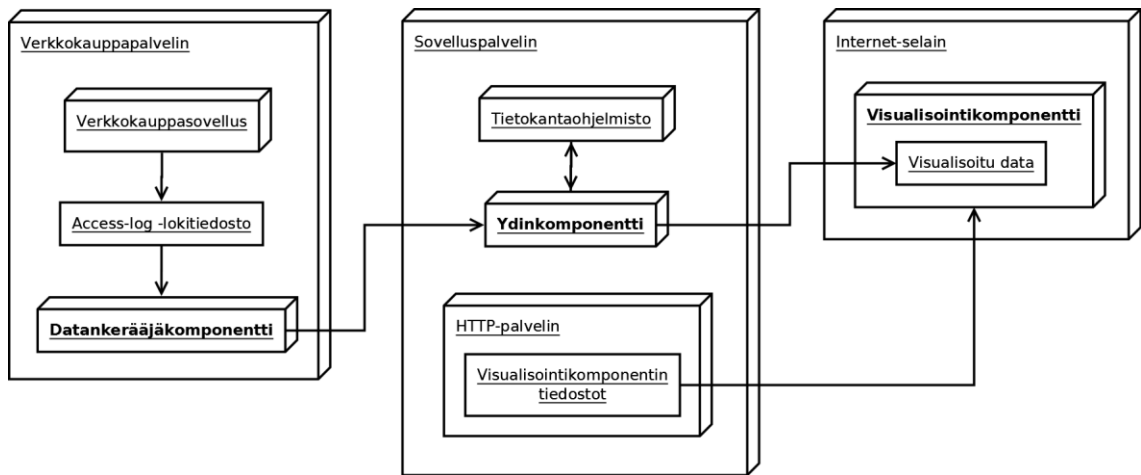
Kuva 2. Ympäristöjen kokonaisarkkitehtuuri ja käytetyt protokollat

Visualisointisovelluspalvelimeen pääsee suoraan käsiksi vain sisäverkosta ja palomuurin läpi ulkoverkosta ainoastaan sallituista osoitteista, jotka ovat verkkokauppalvelimien osoitteet (kuva 2).

Visualisointisovellus on vain sisäiseen käyttöön, joten sitä ei tarvitse voida käyttää muualta kuin sisäverkosta. Osasta verkkokauppalvelimista yhteys visualisointisovelluspalvelimeen menee internetin läpi, jos ne sijaitsevat eri palveluntarjoajien tiloissa, ja osasta palvelimista yhteys menee sisäverkon kautta, jos palvelimet sijaitsevat samassa tilassa.

3.2 Sovelluksen jako komponentteihin

Sovellus jaetaan eri palvelimilla ajettaviin komponentteihin, jotta monesta eri verkkokauppasovelluksesta voidaan kerätä samanaikaisesti tietoa vaarantamatta verkkokauppojen ja niiden palvelinten tietoturvaa ja jotta sovelluksen ominaisuuksia on mahdollisimman helppo laajentaa jatkossa.



Kuva 3. Korkean tason rakennearkkitehtuuri

Sovellus jaetaan kolmeen erityyppiseen komponenttiin: yksi ydinkomponentti, yksi tai useampi datankerääjäkomponentti sekä yksi tai useampi visualisointikomponentti (kuva 3). Kuvassa korostettuna sovelluksen varsinaiset komponentit. Nuolet kertovat osien välisistä yhteyksistä ja nuolen suunta kertoo, kumpaan suuntaan tieto liikkuu tässä yhteydessä, mikä ei välttämättä ole sama kuin suunta, josta itse yhteys muodostetaan.

Ideana on, että komponenttien väliset rajapinnat ovat mahdollisimman selkeitä ja yksinkertaisia, jotta sovellusta on helppo jatkokehittää lisäämällä erilaisia toteutuksia komponenteista.

3.2.1 Ydinkomponentti

Sovelluksen ydinkomponentti on nimensä mukaan sovelluksen ydin, ja käytännössä se sisältää suurimman osan sovelluksen toiminnollisuuksista, eikä ilman ydinkomponenttia millään muulla komponentilla tee käytännössä mitään.

Ydinkomponentti yhdistää kaikki muut komponentit, ja tallentaa datankerääjäkomponenttien keräämän datan myöhempää käyttöä varten. Ydinkomponentti tallentaa datankerääjäkomponenttien keräämän datan relaatiotietokantaan, ja tarjoaa WebSocket-yhteyden ja REST-rajapinnan, joita visualisointikomponentit käyttävät.

3.2.2 Datankerääjäkomponentti

Sovelluksen datankerääjäkomponentti kerää eri lähteistä kaiken datan, jota sovellus käyttää. Datankerääjäkomponentteja voi olla saman aikaisesti käynnissä useita erilaisia, erilaisissa paikoissa riippumattomina toisistaan, ja jokainen datankerääjäkomponentti on yhteydessä ainoastaan ydinkomponenttiin.

Datankerääjäkomponentteja ajetaan jokaisella verkkokauppal palvelimella, jonka tietoa halutaan kerätä. Datankerääjäkomponentti käyttää käytännössä yksisuuntaista yhteyttä jolla se siirtää kerätyn datan ydinkomponentille prosessoitavaksi ja säilytettäväksi.

3.2.3 Visualisointikomponentti

Sovelluksen visualisointikomponentit tarjoavat visualisoitua dataa loppukäyttäjälle. Alustavasti visualisointikomponentteja on vain kaksi erilaista: yksi joka näyttää reaaliaikaisia tilastoja maailmankartalla, ja toinen, joka näyttää historiatietoja maailmankartalla.

Visualisointikomponentit ovat HTML- ja JavaScript-pohjaisia toteutuksia, joita käytetään internetselaimella. Visualisointikomponentit saavat visualisointiin tarvittavan datan suoraan ydinkomponentilta. Tämä mahdollistaa helpon käytön lukuisilla eri laitteilla, eikä ole sidottu tiettyyn laitekohtaiseen asiakasohjelmistoon.

3.3 Komponenttien välinen kommunikointi

Datankerääjäkomponentit kommunikoivat ydinkomponentin kanssa käytännössä yksisuuntaisesti matalan tason TCP-protokollaa käyttäen lähettämällä JSON-formaatissa olevia viestejä, jotka ovat rivinvaihtomerkillä eroteltuna.

Ydinkomponentti tarjoaa sekä REST-rajapinnan että WebSocket-yhteyden visualisointikomponenteille. Ydinkomponentti lähettää kaikki sen saamat tilapäivitykset reaaliaikaisesti STOMP-protokollan mukaisina viesteinä kaikille visualisointikomponenteille, jotka ovat sillä hetkellä WebSocket-yhteydessä ydinkomponenttiin. REST-rajapinnalla visualisointikomponentit voivat hakea dataa aikaisemmin tallennetuista tapahtumista.

3.3.1 REST

REST (Lyhenne sanoista Representational State Transfer) on ohjelmistoarkkitehtuurimalli, joka perustuu HTTP-protokollaan ja jota käytetään ohjelmointirajapintojen toteutukseen. REST ei ota kantaa komponenttien toteutuksen yksityiskohtiin tai protokollan syntaksiin, vaan keskittyy enemmän komponenttien roolituksiin.

REST-rajapintaa käyttävät järjestelmät tyypillisesti kommunikoivat HTTP-protokollan yli käyttäen samoja pyyntömetodeja, jota internetselaimet käyttävät web-sivujen hakuun ja tiedon lähetykseen palvelimille. [1.]

3.3.2 WebSocket

WebSocket-protokolla mahdollistaa kaksisuuntaisen kommunikaation yhden TCP-yhteyden läpi. WebSocket on suunniteltu tuettavaksi sekä internetselaimilla että WWW-palvelimilla. WebSocket-protokolla on itsenäinen TCP-pohjainen protokolla, joka alustetaan normaalin HTTP-kutsun yhteydessä käyttäen Upgrade-ylätunnistetta. [2.]

WebSocket-protokolla mahdollistaa asynkronisten viestien lähettämisen palvelimelta selaimelle, vaikka selain olisi palomuurin takana. Etu tässä on se, että viestit välittyvät reaaliaikaisesti selaimelle, eikä selaimen tarvitse erikseen hakea viestejä.

3.3.3 STOMP

STOMP (lyhenne sanoista Simple Text-Orientated Messaging Protocol) on yksinkertainen tekstipohjainen viestiprotokolla. STOMP on alun perin tarkoitettu asynkroniseen viestien lähettämiseen eri asiakasohjelmien välillä välittäjäpalvelinten kautta. [3.]

STOMP on suunniteltu lähtökohtaisesti olemaan mahdollisimman kevyt, yksinkertainen ja helposti käytettävä riippumatta ympäristöstä tai muista vaikuttavista tekijöistä. Tästä johtuen STOMP oli ideaali valinta tähän sovellukseen käytettäväksi.

3.4 Rajapintakuvaukset

Ydinkomponentti tarjoaa kaksi erilaista rajapintaa: WebSocket-yhteyden sekä REST-rajapinnan. WebSocket-yhteys on reaaliaikaisille päivityksille, ja REST-rajapinta on historiatietojen hakua varten.

3.4.1 WebSocket-yhteys

Visualisointikomponentit voivat tilata yhden tai useamman viestipolun, joita pitkin ne vastaanottavat erilaisia reaaliaikapäivityksiä. Polut alkavat aina `"/queue/realtime"`, ja polkujen perään voidaan valinnaisesti lisätä vielä haluttu viestilähde, esimerkiksi `"/queue/realtime/kauppa1"`.

Ydinkomponentti lähettää jokaisen viestin kahteen polkuun: `"/queue/realtime"`, johon kaikki viestit päätyvät, ja lisäksi viestin lähteen mukaiseen viestipolkuun, eli jos viesti on peräisin `"kauppa1"`-nimiseltä palvelimelta, lähetetään kyseinen viesti myös `"/queue/realtime/kauppa1"`-polkuun.

3.4.2 REST-rajapinta

Visualisointikomponentit voivat hakea historiadataa tietyin rajauksin ydinkomponentilta käyttäen REST-rajapintaa. Historiadata voidaan hakea joko HTTP GET- tai HTTP POST -pyynnöillä. Polku historiadatapyynnöille on `"/viz/heatmap"`, ja mahdolliset rajaukset määritetään pyynnön parametreina.

Historiadatassa voidaan rajata haettavaksi vain tietyltä palvelimelta peräisin oleva data käyttämällä `"nodeDescription"`-parametria, jonka arvona on halutun palvelimen nimi. Historiadata voidaan myös rajata tietylle aikavälille käyttäen `"startDate"`- ja `"endDate"`-parametreja, joiden arvo on haluttu aikaleima ISO 8601 -formaattissa.

Kaikki parametrit ovat valinnaisia, ja puuttuva parametri tarkoittaa, ettei tuloksia rajata lainkaan kyseisen parametrin mukaan. Ilman mitään parametreja tehty haku palauttaa kaiken mahdollisen historiadan kaikista lähteistä koko ajalta, mistä historiatietoja on saatavissa.

3.5 Ympäristöt

Sovelluksen kaikki komponentit on ensisijaisesti suunniteltu käytettäväksi Linux-palvelimilla mahdollisimman vähäisillä riippuvuuksilla. Java-ohjelmointikielen tuomista hyödyistä johtuen voi komponentteja käyttää muun tyyppisillä palvelimilla myös.

Taulukko 1. Eri ympäristöjen ympäristövaatimukset. Minimiversio sulkeissa.

Vaatus	Kääntöympäristö	Datankerääjäkomponentti (Verkkokauppapalvelin)	Ydinkomponentti (Sovelluspalvelin)
Java	X (JDK 8)	X (JRE/JDK 6)	X (JRE/JDK 8)
MySQL/MariaDB			X
Apache Tomcat			X (Versio 8.0)
Apache Maven	X		

Kääntöympäristön, ja myös kehitysympäristön, vaatimukset ovat käytännössä vain Java 8 JDK itse kääntöä varten, ja Apache Maven kääntöprosessin automatisointiin ja riippuvuuksien hallintaan.

Datankerääjäkomponentti on suunniteltu kaikkein vähäisimpien vaatimusten mukaan, eikä se siten vaadi ympäristöltä mitään muuta kuin Java-ajoympäristön. Käytännössä jokainen Smilehousen tuottama verkkokauppasovellus on Java-pohjainen, jolloin jokaiselta verkkokauppapalvelimelta löytyy Java jo ennestään.

Ydinkomponentti on suunniteltu ajettavaksi Apache Tomcat -palvelimella, ja käyttää MySQL-yhteensopivaa relaatiotietokantaa datan tallentamiseen. Visualisointikomponentti sisältää vain staattisia HTML- ja JavaScript-tiedostoja, joten sitä voidaan ajaa miltä tahansa http-palvelimelta.

3.6 Kerättävä data

Datankerääjäkomponentti kerää eri verkkokaupoilta kävijätietoja: kävijän IP-osoite, käyntiajankohta ja käynnin kohde. Kerättävä data tallennetaan MariaDB-relaatiotietokantaan myöhempää käyttöä varten. Data prosessoidaan ja anonymisoidaan ennen sen tarjoamista visualisointikomponenteille. Kävijän maantieteellinen sijainti paikallistetaan kävijän IP-osoitteen perusteella käyttäen MaxMind:n GeoIP-tietokantaa ja IP-osoite piilotetaan visualisointikomponenteilta.

4 Sovelluksen toteutus

Tässä luvussa käsitellään sovelluksen toteutus. Toteutus aloitettiin asentamalla tarpeelliset kehitystyökalut kehittämiseen käytettävälle tietokoneelle.

4.1 Kehitystyökalut

Paikallisena kehitysympäristönä oli kannettava tietokone, johon oli asennettuna Xubuntu Linux. Linux oli ideaali vaihtoehto paikalliselle kehitysympäristölle, koska tällöin kaikki tarpeelliset työkalut, palvelimet ja sovellukset voitiin helposti asentaa paikallisesti.

Kehitysympäristöön asennettiin JDK8, Apache Tomcat, Apache Maven, Git sekä Eclipse IDE -ohjelmistot. Ohjelmiston lähdekoodia kirjoitetaan Eclipse IDE -ohjelmointiympäristöllä ja pidetään Git-versiohallintajärjestelmässä. Apache Maven -ohjelmistoa käytetään riippuvuuksien hallintaan ja kääntöprosessin automatisointiin. Sovelluksen kaikkia komponentteja ajettiin paikallisesti kehityskoneella kehitysvaiheessa.

4.2 Projektin rakenne

Sovellusta kehitetään Maven-projektina, mikä mahdollistaa sovelluksen eri komponenttien jaon omiksi moduuleikseen. Koko projektilla on yksi pää POM-tiedosto, jolla on määritetty alimoduuleiksi sovelluksen muut komponentit.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.smilehouse.viz</groupId>
  <artifactId>viz</artifactId>
  <version>0.1</version>
  <packaging>pom</packaging>
  <description>Aggregator/Parent project for Viz. Useful for cleaning everything at once.</description>

  <modules>
    <module>viz-main</module>
    <module>viz-collector</module>
    <module>viz-map</module>
  </modules>
</project>
```

Koodiesimerkki 1. Pää POM-tiedosto XML-muodossa

Koko projektin pää POM-tiedosto sisältää vain alimoduulimäärytykset (koodiesimerkki 1), ja kaikki muut määrytykset on jaettu alimoduuleille, eli sovelluksen eri komponenttien projekteille. Tällainen pääprojekti mahdollistaa toimintojen tekemisen kaikille alimoduuleille yhdellä kertaa, esimerkiksi projektien väliaikaistiedostojen poisto.

Sovelluksen jokaisella komponentilla on oma Maven-aliprojekti, jonka POM-tiedostossa on määriteltynä kaikki kyseiseen komponenttiin liittyvät riippuvuudet ja muut konfiguraatiomäärytykset.

4.3 Sovelluksen tekninen rakenne

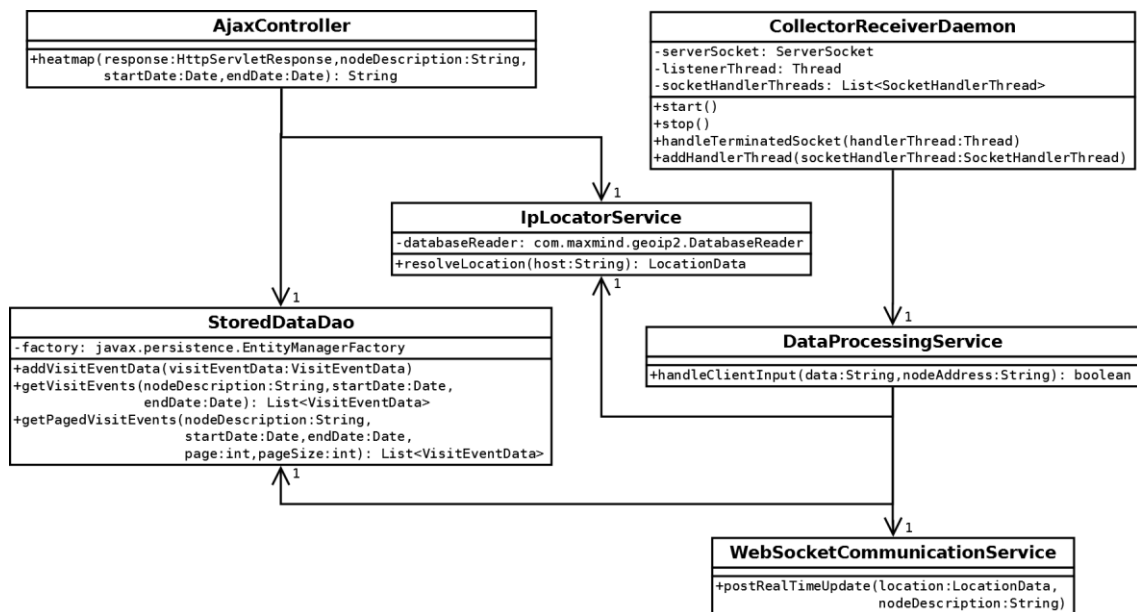
Sovelluksen rakenteessa tärkeimmässä osassa on ydinkomponentti, sillä se pitää sisällään suurimman osuuden sovelluksen toiminnollisuuksista ja toimii kaikkia komponentteja yhdistävänä tekijänä.

Visualisointikomponentti vastaanottaa ja/tai hakee anonymisoituja ja prosessoituja tietoja verkkokauppasivustojen vierailuista ja visualisoi ne internetsivulle. Visualisointikomponentit vastaanottavat reaaliaikaisia päivityksiä, ja hakevat erikseen historiatietoja ydinkomponentilta.

Datankerääjäkomponentti yksinkertaisuudessaan vain tarkkailee verkkokauppal palvelimen access-log-lokitiedoston päivittymistä, ja lukee sieltä tiedot sivustolle tehdyistä vierailuista ja lähettää ne ydinkomponentille käsiteltäväksi.

Access-log on lokitiedosto, johon WWW-palvelimet tallentavat tiedot jokaisesta sivulle tehdystä kutsusta. Tyypillisesti lokitiedosto sisältää jokaisen kutsun aikaleiman, IP-osoitteen, josta kutsu tuli, kutsun koko polun sekä käyttäjän selaimen tietoja. Sovellus toteutetaan lukemaan ja ymmärtämään Apache httpd- ja Apache Tomcat -ohjelmistojen oletus lokitusformaattia.

Toteutuksen rakenteellinen suunnittelu aloitettiin ydinkomponentin luokkakaaviosta, jossa havainnollistetaan ydinkomponentin eri toiminnallisten luokkien toiminnallisuudet ja suhteet toisiinsa (kuva 4).



Kuva 4. Ydinkomponentin luokkakaavio.

CollectorReceiverDaemon-luokka vastaanottaa datankerääjäkomponentilta tulevat sanomat ja siirtää ne prosessoitavaksi DataProcessingService-luokalle. Sanomat tallennetaan tietokantaan ja prosessoidaan sekä anonymisoidaan ja siirretään edelleen WebSocketCommunicationService-luokalle, joka lähettää datan reaaliajassa kaikille aktiivisille visualisointikomponenteille.

Visualisointikomponentit hakevat historiadataa AjaxController-luokalta, joka hakee tietokannasta tarvittavan datan, prosessoi sen ja palauttaa prosessoidun datan kutsujalle. AjaxController-luokan tekemä prosessointi sisältää tulosten suodatuksen, IP-osoitteiden paikallistamisen, datan anonymisoinnin ja lopullisen datan muotoilun JSON-muotoiseksi viestiksi.

4.4 Datan lukeminen palvelinten lokitiedostoista

Datankerääjäkomponentti tarkkailee verkkokauppapalvelinten access-log-lokitiedostojen päivittymistä reaaliaikaisesti, ja lukee kaikki lokeihin kirjoitetut uudet rivit, käsittelee ne ja lähettää ydinkomponentille. Uusien lokitiedostojen luontia ja mahdollista poistoa seurataan käyttämällä WatchService-tyypin luokkaa, joka reagoi kaikkiin tiedostojen luonteihin ja poistoihin tietyssä hakemistossa [4.]. Lokitiedostojen päivittymistä seurataan Apache Commons -kokoelman Tailer-luokalla, joka reagoi kaikkiin muutoksiin seurattavan tiedoston sisällä [5.].

```

if (kind == StandardWatchEventKinds.ENTRY_CREATE) {

    LOG.info("New log detected: " + file);
    LogTailerListener listener = new LogTailerListener();
    Tailer tailer = Tailer.create(file,
        listener, 1000, true);
    tailers.put(file, tailer);

    if (tailer != null) {
        LOG.warn("Duplicate Tailer added for file \""
            + file
            + "\". Stopping and removing old tailer...");
        tailer.stop();
    }

} else if (kind == StandardWatchEventKinds.ENTRY_DELETE) {

    LOG.info("Log file deleted: " + file);
    Tailer t = tailers.remove(file);
    if (t != null) {
        t.stop();
    }

}

```

Koodiesimerkki 2. Tailer-olion luonti ja poisto lokitiedostojen hakemiston sisällön muuttuessa

WatchService:ltä voidaan kysyä ja jäädä odottamaan tarkkailtavan hakemiston muutoksia. Tässä yhteydessä on kiinnostuttu ainoastaan uuden tiedoston luonnista tai vanhan poistamisesta, joiden yhteydessä joko luodaan uusi Tailer-olio tai poistetaan vanha vastaavasti (koodiesimerkki 2). Luotavalle Tailer-oliolle annetaan parametrina Listener-olio, joka käsittelee Tailer-olion tarkkaileman tiedoston muutokset, eli parsii tiedostoon kirjoitetut uudet lokirivit ja lähettää ne edelleen ydinkomponentille.

4.5 Datan tallentaminen tietokantaan

Datankerääjäkomponentilta tuleva data prosessoidaan ydinkomponentissa ja tallennetaan MySQL-yhteensopivaan SQL-relaatiotietokantaan myöhempää käyttöä varten. Jokaiselle viestille luodaan juoksevilla numeroinnilla oma, viestin yksilöivä ID-numero. Kantaan tallennetaan viestin mukana tulleiden tietojen lisäksi palvelimen IP-osoite, josta viesti tuli. Tätä tietoa ei tässä vaiheessa suoraan käytetä mihinkään, mutta se voi osoittautua hyödylliseksi myöhemmässä vaiheessa, kun sovellusta jatkokehitetään.

Taulukko 2. Tietokantataulun rakenne

Sarakkeen nimi	Tyyppi	Sarakkeen selite
id	bigint	Merkinnän yksilöivä, juokseva numero
sourceAddress	varchar	IP-osoite, josta vierailu sivustolle tehtiin
nodeAddress	varchar	IP-osoite, josta datankerääjäkomponentin viesti tuli
nodeDescription	varchar	Viestin lähdepalvelimen nimi tai selite
timestamp	datetime	Aikaleima jolloin sivustovierailu tehtiin

Tietokantaan tarvitaan vain yksi taulu, johon tiedot tallennetaan viiteen eri sarakkeeseen. Tietokantasovellus luo id-numeron automaattisesti sarakkeelle asetetun `auto_increment`-määreen avulla.

Tietokannan käsittelyssä hyödynnetään EclipseLink JPA -toteutusta, joka hoitaa tiedon muuntamisen ja siirtämisen Java koodissa Bean-luokkien ja itse tietokannan välillä kumpainkin suuntaan ilman, että tarvitsee kirjoittaa varsinaisia SQL-lauseita itse.

```

@Entity(name = "visitEventData")
@Table(name = "visitEventData")
public class VisitEventData implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String sourceAddress;
    private String nodeAddress;
    private String nodeDescription;
    @Temporal(javax.persistence.TemporalType.TIMESTAMP)
    private Date timestamp;
    @Transient
    private LocationData location;

```

Koodiesimerkki 3. Osa tietokantaan tallennettavaa vierailutietoa kuvaavasta luokasta

Tietokannan taulun rakennetta vastaavaan luokkaan määritetään kaikki taulussa olevat sarakkeet, sekä niiden tyyppi (koodiesimerkki 3). Määrittelyssä käytetään `@`-merkillä alkavia merkintöjä JPA-toteutuksen vaatimia lisätietoja varten. Luokassa määritetty, `"Transient"`-merkinnällä merkitty `"location"`-muuttuja sisältää ohjelman ajonaikana sijain-tidataa, eikä sitä tallenneta tietokantaan. `"Transient"`-merkinnällä merkityt muuttujat au-tomaattisesti jätetään käsittelemättä datan tallentamisessa ja hakemisessa [6.].

```

/**
 * Adds a new visit event to the database
 *
 * @param visitEventData
 *      visit event to add
 */
public void addVisitEventData(VisitEventData visitEventData) {
    if (visitEventData == null) {
        LOG.warn("Attempted to persist a null VisitEventData");
        return;
    }
    EntityManager em = factory.createEntityManager();
    em.getTransaction().begin();
    em.persist(visitEventData);
    em.getTransaction().commit();
    em.close();
}

```

Koodiesimerkki 4. Vierailudataa sisältävän olion datan tallennus tietokantaan

Olioissa olevan tallentaminen tietokantaan onnistuu helposti vain muutamalla rivillä koodia, kun käytetään JPA-rajapinnan mukaista toteutusta (koodiesimerkki 4). Datan hakeminen tietokannasta olioihin onnistuu lähes yhtä helposti ja vaivattomasti.

4.6 Komponenttien välinen kommunikointi

Datankerääjäkomponentti on yhteydessä ainoastaan ydinkomponenttiin, ja visualisointikomponentti on ainoastaan yhteydessä ydinkomponenttiin. Ydinkomponentti on käytännössä kaikki komponentit yhdistävä tekijä.

4.6.1 Datankerääjäkomponentin ja ydinkomponentin välinen yhteys

Datankerääjäkomponentti avaa TCP-yhteyden ydinkomponenttiin käynnistyessään, ja lähettää tällä samalla yhteydellä kaiken datan. Jokainen sanoma, jonka datankerääjäkomponentti lähettää, on selkokielineen, JSON-formaatissa oleva viesti. Sanomat on eroteltu toisistaan rivinvaihtomerkillä.

Jokaisen sanoman JSON- viesti sisältää vain yhden olion, jonka nimi on viestin tyyppi ja sisältö on viestin tyyppille ominainen. Tässä vaiheessa viestejä on vain yhden tyyppisiä,

nimeltään "visitEventData", joka sisältää yhden vierailutiedon. Vierailutieto sisältää vierailijan IP-osoitteen, vierailukohteen sekä aikaleiman, jolloin vierailu tapahtui (koodiesimerkki 5).

```
{
  "visitEventData": {
    "sourceAddress": "77.240.19.181",
    "nodeDescription": "Test shop app1",
    "timestamp": "2015-12-11T07:39:16.344Z"
  }
}
```

Koodiesimerkki 5. Esimerkki datankerääjäkomponentin lähettämästä viestistä

Ydinkomponentti vastaa jokaiseen vastaanotettuun sanomaan lähettämällä takaisin saman sanoman joko "OK:"- tai "FAIL:"-etuliitteillä, sen mukaan ymmärsikö ydinkomponentti sanoman vai ei. Vastaus on vain ongelmatilanteiden selvittämistä helpottava asia, ja datankerääjäkomponentti ei tee vastauksella mitään.

4.6.2 Ydinkomponentin WebSocket-yhteys

Ydinkomponentti anonymisoi ja prosessoi jokaisen datankerääjäkomponentilta vastaanotetun sanoman heti, ja lähettää sen edelleen jokaiselle yhdistyneelle visualisointikomponentille WebSocket-yhteyden yli. WebSocket-yhteyden yli lähetetään STOMP-protokollan mukaisia viestejä.

```

/**
 * Posts a real-time update through websocket
 *
 * @param Location
 *         the new location to send
 * @param nodeDescription
 *         description of the node this update originated from
 */
public void postRealTimeUpdate(LocationData location, String nodeDescription) {
    if (location == null) {
        LOG.warn("Attempted to send a null update");
        return;
    }
    template.convertAndSend("/queue/realtime", location);
    if (nodeDescription != null && !nodeDescription.isEmpty()) {
        template.convertAndSend("/queue/realtime/" + nodeDescription,
                                location);
    }
}
}

```

Koodiesimerkki 6. Viestin lähetys WebSocket-yhteyden yli

Koodiesimerkki 6:ssa nähdään, kuinka sama viesti lähetetään kahteen eri polkuun, jos virhetilanteita ei ole. Jos viesti on tyhjä, ei sitä lähetetä ollenkaan ja jos viestistä puuttuu viestin lähde, ei sitä lähetetä minkään viestilähteen omaan polkuun.

```

{
    "lat":9.85,
    "lon":76.9667
}

```

Koodiesimerkki 7. Esimerkki ydinkomponentin lähettämästä reaaliaikaisesta päivitysviestistä.

Jokainen ydinkomponentin lähettämä JSON-muotoinen viesti sisältää koordinaatin maantieteelliseen sijaintiin, josta kutsu alun perin tuli (koodiesimerkki 7). Koska nämä viestit ovat reaaliaikaisia, ei erillistä aikaleimaa tarvita.

4.6.3 Ydinkomponentin REST-rajapinta

Visualisointikomponentit voivat myös hakea historiatietoja tietyin hakuehdoin asynkronisesti ydinkomponentilta. Ydinkomponentti koostaa JSON-muotoisen vastauksen pyynnön parametreina annettujen ehtojen mukaisesti erillisessä käsittelymetodissa (Koodiesimerkki 8).

```

@RequestMapping(value = "/heatmap", produces = "application/json")
public @ResponseBody String heatmap(HttpServletResponse response,
    @RequestParam(required = false) String nodeDescription,
    @RequestParam(required = false) @DateTimeFormat(pattern = "yyyy-MM-dd'T'HH:mm:ss.SSSX") Date startDate,
    @RequestParam(required = false) @DateTimeFormat(pattern = "yyyy-MM-dd'T'HH:mm:ss.SSSX") Date endDate) {
    LOG.info("/heatmap called with nodeDescription=" + nodeDescription
        + " startDate=" + startDate + " endDate=" + endDate);

    response.setHeader("Access-Control-Allow-Origin", "*");
    return constructHeatmapJson(nodeDescription, startDate, endDate);
}

```

Koodiesimerkki 8. REST rajapinnan heatmap-kutsun käsittelymetodi

Ydinkomponentin vastausviesti näihin pyyntöihin on JSON-muotoinen viesti, joka sisältää "data"-nimisen taulukon, jonka alkiot sisältävät maantieteellisen koordinaatin sekä osumalukumäärän kyseisessä koordinaatissa (koodiesimerkki 9).

```

{
  "data": [
    {
      "lat": 29.8782,
      "lon": 121.5495,
      "weight": 1
    },
    {
      "lat": 9.85,
      "lon": 76.9667,
      "weight": 4
    },
    {
      "lat": 42.8333,
      "lon": 12.8333,
      "weight": 2
    }
  ]
}

```

Koodiesimerkki 9. Esimerkki ydinkomponentin antamasta historiadatasta.

Palautettu historiatieto on satunnaisessa järjestyksessä, joten jos visualisointikomponentissa halutaan tehdä jonkinlainen histogrammi tai vastaava, on jokainen aikahaarukka haettava erillisillä kutsuilla.

4.7 Suorituskykyoptimointi

Koska suurimmilla verkkokaupoilla on hyvin suuret määrät kävijöitä, on suorituskykyoptimointi tarpeen erityisesti ydinkomponentin kohdalla tilanteissa, joissa käsitellään tallennettua historiadataa. Historiadata muodostetaan hakemalla tietokannasta tallennetut tiedot halutuilla rajauksilla, esimerkiksi tietyltä aikaväliltä. Testidatana käytettiin 3,6 miljoonaa satunnaisesti luotua IP-osoitetta.

4.7.1 Suorituskykyoptimoinnin lähtökohta

Lähtökohtana toimi yksinkertainen perustoteutus, jossa yhdellä säikeellä ensin haetaan tietokannasta tarvittava data. Tämän jälkeen jokaisen tuloksen kohdalla haetaan tallennetun IP-osoitteen perusteella maantieteellinen sijainti. Tulokset joiden maantieteellistä sijaintia ei voitu selvittää, jätetään pois. Lopuksi yhdistetään samasta maantieteellisestä sijainnista tulleet kutsut yhteen laskien niiden lukumäärä, ja muodostetaan JSON-viesti. Tällä toteutuksella suoritus aika hakukutsun lähettämisestä tulosten vastaanottamiseen oli noin viisi minuuttia, mikä tarkoitti noin 12 000 IP-osoitetta sekunnissa.

Ensisijainen tavoite suorituskykyoptimoinnissa oli saada koko prosessi tapahtumaan monella samanaikaisella säikeellä, jotta voidaan paremmin hyödyntää nykyaikaisten tietokoneiden moniydinprosessoreita.

Seuraavat muutokset suunniteltiin tähän prosessiin:

- Tietokannasta tulevat tulokset jaetaan joukkoihin, joissa jokaisessa oli korkeintaan tuhat tulosta.
- Tulosjoukot käydään läpi kahdeksalla säikeellä, kunnes kaikki tulosjoukot on käyty läpi.
- Läpikäydyt tulosjoukot jaetaan kahteen erilliseen jonoon maantieteellisen sijainnin leveysasteen lukuarvon toisen bitin mukaan, jotta kaikki duplikaatit päätyvät samaan jonoon.
- Samaan aikaan kaksi muuta säiettä käyvät jonoja läpi, ja laskevat yhteen samojen maantieteellisten sijaintien määrät.
- Kun kaikki tulokset on käyty läpi, niistä muodostetaan yksi JSON-muotoinen viesti, joka sisältää maantieteelliset koordinaatit, sekä lukumäärän montako tulosta kullakin koordinaatilla on.

4.7.2 Monisäikeistäminen

Ennen optimointia suoritettujen testien perusteella todettiin, että IP-osoitteen paikallistaminen on kuormittavin osa prosessia, joten tämä päätettiin jakaa kahdeksaan säikeeseen. Suoritettujen testien perusteella tulosten lukumäärien laskenta vei noin kolmanneksen siitä ajasta, mitä meni IP-osoitteiden paikallistamiseen. Jotta tulosten lukumäärien laskenta ei jäisi prosessin pullonkaulaksi, ajetaan tulosten laskentaa kahdella säikeellä.

```
/*
 * Splitting the results into batches and submitting to the pool
 */

Iterator<VisitEventData> it = events.iterator();
while (it.hasNext()) {
    List<VisitEventData> datas = new LinkedList<>();
    for (int i = 0; i < 1000 && it.hasNext(); i++) {
        datas.add(it.next());
    }
    resolverPool.submit(new ResolverTask(datas, queue1, queue2));
}
resolverPool.shutdown();
```

Koodiesimerkki 10. Tulosten jakaminen tuhannen tuloksen joukkoihin ja syöttäminen säiepoolille

IP-osoitteiden paikallistamisoperaatio jaettiin säikeisin käyttäen Javan omaa ExecutorService-rajapintaa, jossa kahdeksan säikeen säiepooli suoritti niille annettuja tehtäviä. Selvitettävät IP-osoitteet jaettiin tuhannen osoitteen joukkoihin, joista jokainen annettiin suoritettavaksi säiepoolille (koodiesimerkki 10).

ExecutorService-rajapinnan toteuttava olio saatiin kutsumalla Executors-luokan metodia "newFixedThreadPool" parametrilla 8. Kyseinen metodi palauttaa ExecutorService-rajapintaa toteuttavan olion, joka käyttää metodin parametrina annetun lukuarvon (8) verran säikeitä suorittaakseen tehtäviä, joita sille annetaan myöhemmin. [7.]

```

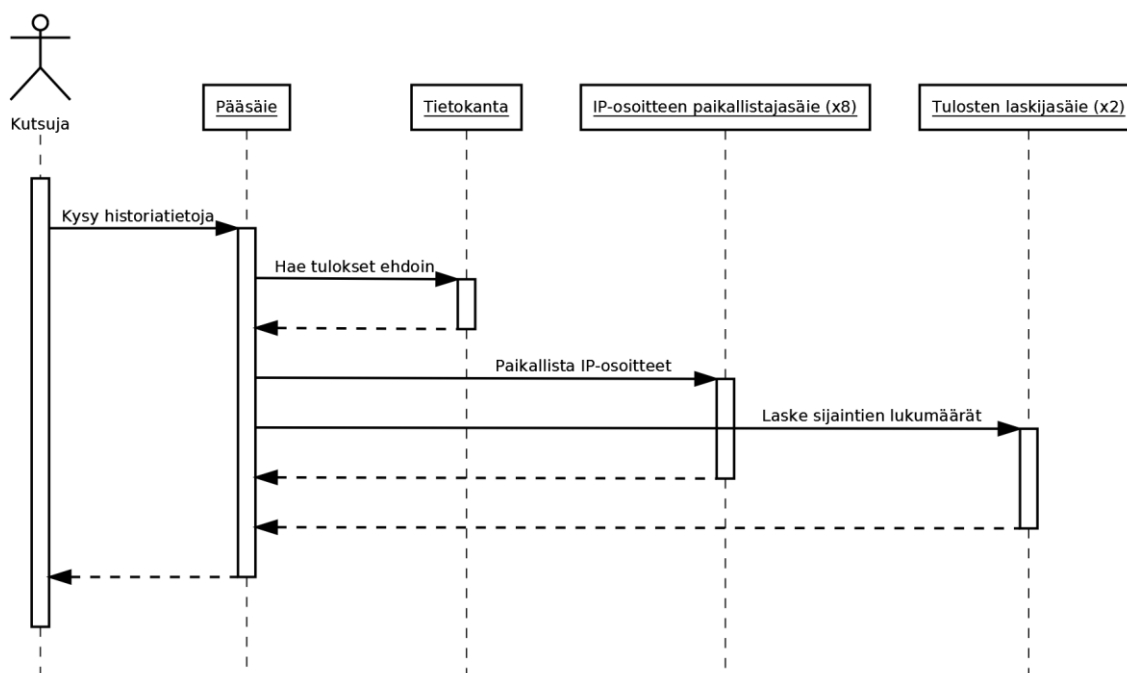
@Override
public void run() {
    LocationData location;
    for (VisitEventData e : datas) {
        location = ipLocatorService.resolveLocation(e
            .getSourceAddress());
        if (location != null) {
            if ((Double.doubleToRawLongBits(location.getLat()) & mask) == mask) {
                queue1.add(location);
            } else {
                queue2.add(location);
            }
        }
    }
}

```

Koodiesimerkki 11. IP-osoitteiden paikallistaminen ja jakaminen kahteen jonoon

Jokainen IP-osoitteen paikallistajasäie käy järjestyksessä läpi sille annetut IP-osoitteet ja paikallistaa ne, ja mikäli paikallistaminen onnistui, lisää tulos yhteen kahdesta josta, joita tulosten laskijasäikeet lukevat (koodiesimerkki 11). Jotta tulosten laskijasäikeet saavat laskettua jokaisen päällekkäisen tuloksen, on varmistettava, että jokainen sama koordinaatti päättyy samaan jonoon.

Paikallistettujen tulosten jakaminen jonoon määräytyy paikallistetun maantieteellisen koordinaatin leveysasteen liukulukuarvon raa'an binäärimuodon toisen bitin mukaan. Tämä jakamisperuste toimii paremmin kuin esim. maantieteellisen alueen mukaan jako, sillä tulokset eivät usein ole maantieteellisesti kovin tasaisesti jakautuneet.



Kuva 5. Sekvenssikaavio monella samanaikaisella säikeellä tapahtuvasta prosessoinnista.

Tulosten rinnakkaista prosessointia eri vaiheissa voidaan paremmin havainnollistaa sekvenssikaaviolla (kuva 5), missä ilmenee rinnakkain tapahtuvat toiminnot ja toimintoja tekevien säikeiden määrät.

4.7.3 Suorituskykyoptimoinnin lopputulos

Muutosten jälkeen samalla 3,6 miljoonan satunnaisen IP-osoitteen testidatalla koko suoritus aika oli noin 67 sekuntia, mikä tarkoitti noin 54 000 IP-osoitetta sekunnissa. Alkuperäinen tulos samalla datalla oli noin viisi minuuttia, eli noin 12 000 IP-osoitetta sekunnissa. Havaintojen perusteella suorituskykyä saatiin parannettua noin 350 %, eli yli neljä kertaa nopeammaksi.

Kyseiset tulokset saatiin kehitysympäristönä toimivalla kannettavalla tietokoneella, jossa on neliydinprosessori. Monella säikeellä ajaminen mahdollisti prosessorin kaikkien ytimien käytön samaan aikaan. Prosessointiin liittyvien säikeiden kokonaislukumäärän ollessa yhdeksän, pystyy sovellus hyödyntämään vieläkin useampaa prosessoriydintä oikeassa palvelinympäristössä.

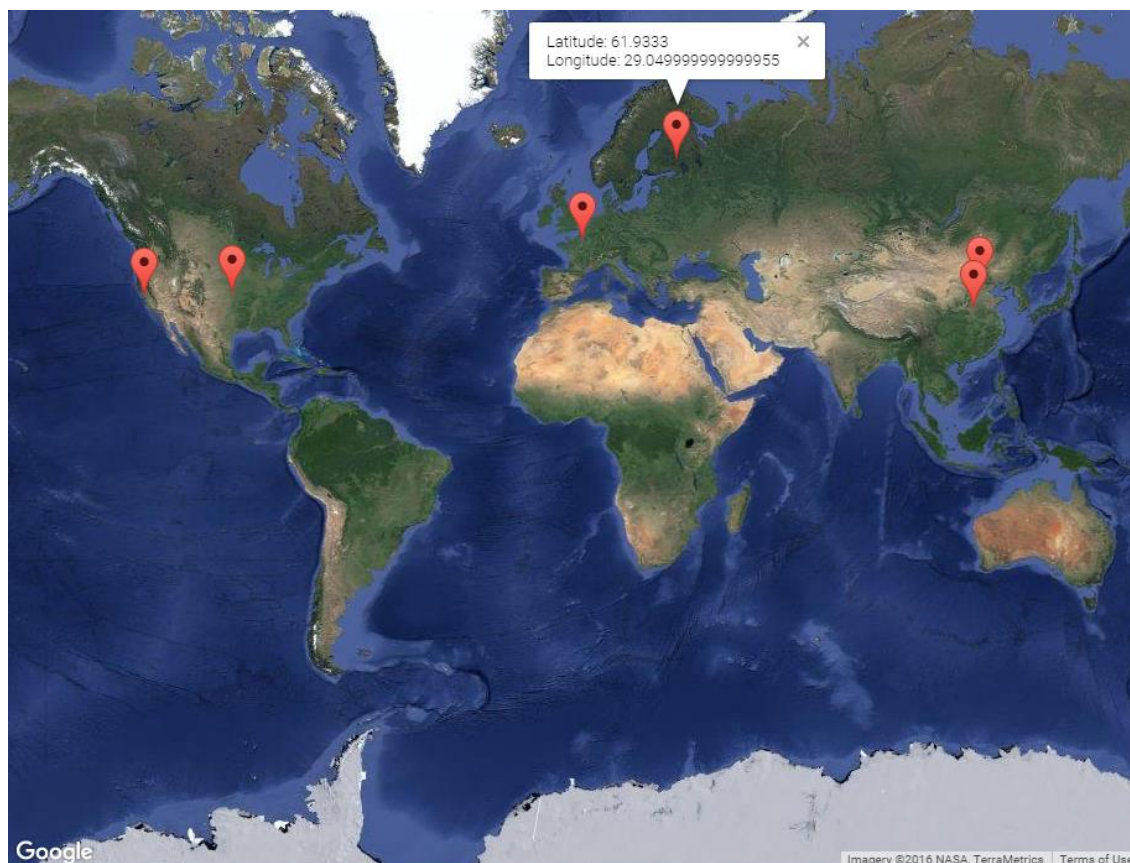
4.8 Visuaalinen toteutus

Sovellukseen toteutettiin tässä vaiheessa kaksi erilaista Visualisointikomponenttia: reaaliaikainen sekä historiallinen visualisointi. Reaaliaikainen komponentti käyttää WebSocket-yhteyttä vastaanottaakseen reaaliaikaisia päivityksiä. Historiadatakomponentti hakee REST-rajapinnalta halutut tiedot tiheyskartalla visualisointia varten.

Molemmat komponentit on toteutettu käyttäen Google Maps API -ohjelmointirajapintaa, jolla piirretään maailmankartta, kartalle tulevia "nuppineuloja" sekä tiheyskartta. Komponenttien muut osat ovat toteutettu käyttäen HTML5-, CSS- sekä JavaScript-tekniikoita.

4.8.1 Reaaliaikainen kartta

Reaaliaikaisessa komponentissa karttaan pudotetaan uusi "nuppineula" jokaista käyntiä kohden, ja näytetään viiden sekunnin ajan tarkemmat koordinaatit kävijästä (kuva 6). Nuppineulat katoavat kartalta minuutin kuluttua niiden ilmestymisestä, jotta kartta ei tule liian täydeksi.

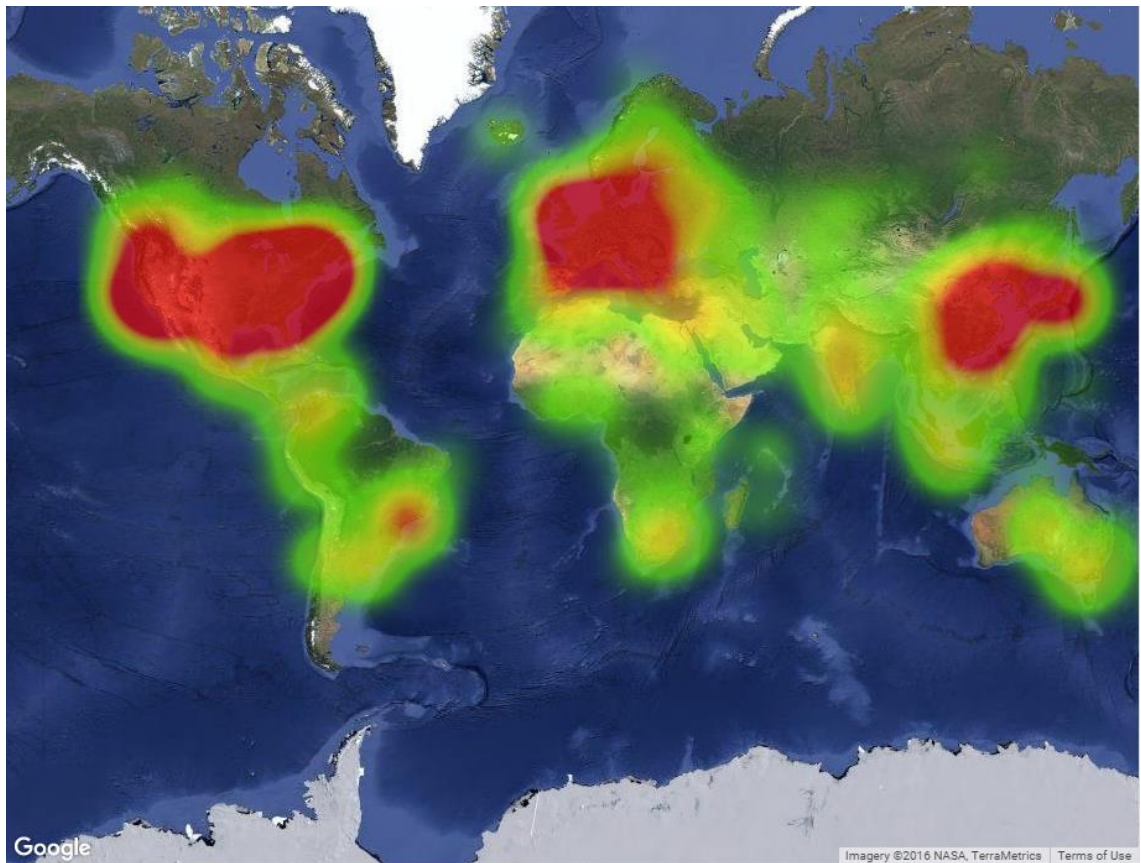


Kuva 6. Reaaliaikainen visualisointikomponentti

WebSocket-yhteydellä kommunikointia varten otettiin käyttöön SockJS JavaScript -kirjasto, jolla saadaan muodostettua WebSocket-yhteys selaimesta käyttäen JavaScriptiä. Viestien käsittelyä helpottamaan otettiin käyttöön STOMP-WebSocket JavaScript-kirjasto, joka helpottaa viestien vastaanottoa ja käsittelyä selaimen puolella.

4.8.2 Tiheyskartta

Tiheyskartassa maailmankartan päälle piirretään tiheyskartta, jossa punaiset alueet ovat aktiivisimpia, ja vihreät alueet vähintään aktiivisia (kuva 7).



Kuva 7. Tiheyskartta sivustolla käynneistä satunnaisesti luodulla esimerkkitiedolla

Historiadataa näyttävä visualisointikomponentti hakee ydinkomponentin REST-rajapinnalta halutut historiatiedot, esimerkiksi edellisen viikon käynnit kaikilla sivustoilla tai kaikki kerätty historiatieto tietyltä sivustolta. Haetut tiedot syötetään Google Maps API:n tarjoamalle tiheyskartta-funktiolle, joka piirtää tiheyskartan maailmankartan päälle (Kuva 5).

```
$.getJSON(heatmapURL).done(function(result) {
    $.each(result.data, function(i, entry) {
        heatmapData.push({
            location : new google.maps.LatLng(entry.lat, entry.lon),
            weight : Math.sqrt(entry.weight)
        });
    });
    heatmap = new google.maps.visualization.HeatmapLayer({
        data : heatmapData,
        dissipating : true,
        radius : 50,
        map : map
    });
    $("#overlay").html("");
});
```

Koodiesimerkki 12. Tiheyskartan muodostaminen

Tiheyskartta muodostetaan ydinkomponentilta saadun JSON-muotoisen viestin mukaan. Viestissä oleva taulukko käydään läpi alkio kerrallaan, ja jokaisesta alkioista muodostetaan Google Maps API tarjoamassa tiheyskartassa käytettävä tietopiste, jolla on painoarvo (koodiesimerkki 12). Ydinkomponentilta saadusta painoarvosta otetaan neliöjuuri, jota käytetään itse tiheyskartassa. Tämä siksi, että tiheyskartasta saadaan selkeämpi, kun pisteiden painoarvot saadaan tasattua paremmin näin.

4.8.3 Kartan interaktiivisuus

Kummallakin visualisointikomponentilla kartta on interaktiivinen: karttaa voi suurentaa tai pienentää, ja sitä voi raahata eri suuntaan. Oletuksena kartta on asemoitu keskelle ruutua ja suurennettu siten, että suurin piirtein koko maapallo mahtuu ruudulle (koodiesimerkki 13).

```
var zoomScale;
if ($(window).width() > $(window).height()) {
    zoomScale = $(window).height() * 1.2; // 1.2 because we are not that interested in the north&south poles
} else {
    zoomScale = $(window).width();
}
var zoomValue = Math.round(Math.log(zoomScale / 256) / Math.LN2);
```

Koodiesimerkki 13. Suurennustason laskeminen

Google Maps API:n suurennustaso (zoomValue) on ruudulla olevien pikseleiden suhde maantieteelliseen pituuteen eikä ruudun suhde maantieteelliseen pituuteen, mikä teki sopivan suurennustason laskemisesta hankalampaa. Pienimmällä suurennustasolla

koko maapallon kartta on 256x256 pikselin kokoinen, ja jokainen sitä suurempi suurennustaso tuplaa aina edellisen [8.]. Tällöin sopiva suurennustaso voidaan laskea kaavalla:

$$\text{suurennustaso} = \frac{\log(\frac{x}{256})}{\log(2)} , \quad (1)$$

Jossa x on ruudun korkeuden tai leveyden arvoista pienempi, pikseleissä.

Tästä saatu arvo on vielä pyöristettävä, sillä kartassa sallitut suurennustasot ovat kokonaislukuja.

4.9 Sovelluksen testaus

Sovelluksen toimintaa testattiin useaan kertaan kehityksen aikana, ja lopuksi testattiin kaikki toiminnollisuudet kertaalleen, ettei sovellukseen ole jäänyt mitään pahoja virheitä ennen käyttöönottoa.

Suoritetut testaukset painottuivat enemmän normaalien olosuhteiden toiminnan testaamiseen, ja vähemmän poikkeustilanteiden testaamiseen johtuen rajallisesta aikataulusta toteutusvaiheessa.

4.9.1 Datankerääjäkomponentin testaus

Datankerääjäkomponentin testaus painottui kahteen osa-alueeseen: lokitiedostojen käsittely ja yhteyden toimivuus ydinkomponentille. Testauksessa käytettiin paljon Netcat-ohjelmaa, joka on työkalu, jolla voidaan lukea ja kirjoittaa TCP- ja UDP-yhteyksiin.

Lokitiedostojen käsittelyn testaus suoritettiin käynnistämällä Netcat-työkalu kuuntelemaan tiettyä porttia, johon datankerääjäkomponentti yhdisti ydinkomponentin sijaan. Tämän jälkeen lokitiedostoja luotiin uusia, poistettiin, ja niihin kirjoiteltiin uusia lokirivejä ja samaan aikaan tarkistettiin Netcat-työkalun tulostamia, avoimesta TCP-yhteydestä luetuja viestejä, jotka oli tarkoitettu ydinkomponentille.

Yhteyden toimivuuden testauksessa datankerääjäkomponentti yhdistettiin normaalisti ydinkomponenttiin. Tämän jälkeen yhteys katkaistiin terminoimalla ydinkomponentti ja

käynnistämällä se uudelleen, ja tarkistettiin, että datankerääjäkomponentti osasi avata yhteyden uudelleen, ja viestit kulkivat uuden yhteyden läpi.

4.9.2 Ydinkomponentin testaus

Ydinkomponentissa oli eniten testattavia asioita, mikä johtui ydinkomponentin suuresta eri toiminnollisuuksien määrästä. Testattavia asioita olivat datankerääjäkomponenttien yhteyksien käsittely, datan tallennus tietokantaan, datan prosessointi sekä rajapintojen testaus.

Yhteyksien testaaminen suoritettiin käynnistämällä ja sulkemalla eri datankerääjäkomponentteja samanaikaisesti, ja tarkkailemalla, että ydinkomponentti osasi siivota vanhat yhteydet ja niiden käsittelijät pois.

Datan tallennus tietokantaan testattiin yksinkertaisesti käynnistämällä yksi datankerääjäkomponentti, joka lähetti viestejä ydinkomponentille, joka talletti ne tietokantaan. Tietokantaa luettiin erillisellä työkalulla, ja sinne tallentunutta tietoa verrattiin alun perin ydinkomponentille lähetettyyn dataan.

Datan prosessointia testattiin tekemällä kyselyjä ydinkomponentin REST-rajapintaan erilaisilla parametreilla, ja vertaamalla takaisin saatua vastausta tietokannasta löytyvään tietoon. Kyselyitä oli helppo testata tavallisella internet-selaimella, johtuen REST-rajapinnan yhteensopivuudesta selainten käyttämän tavallisen HTTP-protokollan kanssa.

4.9.3 Visualisointikomponenttien testaus

Visualisointikomponentissa testattavaa oli reaaliaikaisessa kartassa sijaintipäivitysten riittävä reaaliaikaisuus ja sijainnin oikeellisuus. Tiheyskartan testaus oli suoritettava silmä määräisesti arvioimalla tiheyskartan oikeellisuus siihen syötettyyn dataan nähden.

Lisäksi reaaliaikaisessa kartassa testattiin WebSocket-yhteyden automaattinen uudelleenavaaminen yhteyden katketessa. Tämä testattiin sulkemalla ydinkomponentti ja käynnistämällä se hetken päästä uudelleen, jonka jälkeen katsottiin, osasiko reaaliaikainen kartta avata WebSocket-yhteyden uudelleen ja alkaa vastaanottaa päivityksiä sen yli.

4.9.4 Havaintoja ja huomioita testauksesta

Suurimman osan suoritetuista testeistä olisi voinut automatisoida tekemällä JUnit-testejä ja tekemällä muilla tavoin automatisoituja testejä. Rajallisen ajan vuoksi automatisoituja testejä ei tehty yhtään, mutta tarkoituksena olisi tehdä testejä sovelluksen jatkokehityksen yhteydessä.

Testauksessa olisi ollut myös hyödyllistä testata enemmän erilaisia poikkeustilanteita ja sovelluksen toimintaa tällaisissa tilanteissa. Muutamia poikkeustapauksia testattiin sovelluksen kehitysvaiheessa, kun itse tehtiin vahingossa jokin virhe, esimerkiksi vahingossa määritettiin väärät tunnukset tietokannalle tai väärä polku tarkkailtaville lokitiedostoille.

5 Sovelluksen käyttöönotto

Tässä luvussa käsitellään sovelluksen toteutusvaiheen jälkeisiä asioita. Sovelluksen toteutusvaiheen jälkeen sovellus siirrettiin tuotantoon varsinaista käyttöä varten ja pohdittiin jatkosuunnitelmista.

5.1 Sovelluksen tuotantoon siirto

Kun sovellus oli varsinaisen toteutuksen osalta lähes valmis, oli aika siirtää se oikeaan ympäristöön. Sovelluspalvelimeksi hankittiin palvelin, johon oli valmiiksi asennettuna Debian Linux. Ensimmäiseksi palvelimelle asennettiin Java JDK8, Apache HTTP Server, Apache Tomcat sekä MariaDB.

Apache HTTP Server -palvelinohjelmisto oli oletuskonfiguraatioilla valmiiksi sopivassa tilassa, joten sille ei tarvinnut tehdä mitään. Apache Tomcat -ohjelmistosta poistettiin asennuksen mukana tulevat esimerkkiwebsovellukset ja lisättiin uusi käyttäjä "tomcat-users.xml" -konfiguraatiotiedostoon manager-script-roolilla, jotta kyseisellä käyttäjällä voidaan helposti asentaa sovelluksen ydinkomponentti. MariaDB-tietokantasovellukseen luotiin uusi tietokanta sovellusta varten, sekä tehtiin uusi tietokantakäyttäjä, jolle annettiin oikeudet tähän tietokantaan.

Sovelluksen ydinkomponentti asennettiin Apache Tomcat-ohjelmistoon, ja sovelluksen visualisointikomponentin tiedostot kopioitiin Apache HTTP Server-palvelinohjelmiston webroot-hakemistoksi määritettyyn hakemistoon.

Tässä vaiheessa ei ollut vielä mahdollisuutta käyttää oikeita verkkokauppoja datan keräämiseen, mutta datankerääjäkomponentin asennus olisi hoitunut seuraavalla tavalla: datankerääjäkomponentti siirretään verkkokauppalvelimelle, jota halutaan tarkkailla. Verkkokauppalvelimen käynnistyskomentosarjaan lisätään rivi, joka käynnistää datankerääjäkomponentin palvelimen käynnistyessä. Lopuksi datankerääjäkomponentti käynnistetään kerran käsin.

5.2 Tulevat tuotantoon siirrot

Tulevaisuudessa seuraavat tuotantoon siirrot tulevat olemaan yksinkertaisempia, koska kaikki sovelluksen ulkopuoliset riippuvuudet on jo ennestään asennettu ja säädetty kuntoon.

Visualisointikomponenttien päivityksessä tai uusien lisäämisessä tarvitsee kyseiset komponentit vain kopioida Apache HTTP Server-palvelinohjelmiston webroot-hakemistoon. Ydinkomponentin päivityksessä tarvitsee uusi paketti asentaa Apache Tomcat-ohjelmistoon, joka onnistuu esimerkiksi joko käsin kopioimalla valmis paketti webapp-hakemistoon tai käyttäen Tomcatin manager-app-sovellusta paketin lataamiseen.

5.3 Sovelluksen tulevaisuus

Sovellusta on tarkoitus jatkokehittää sisäisesti sitä mukaan, kun laajennusideoita tulee esille. Tässä vaiheessa tiedossa olevia jatkokehitysideoita on erilaisten visualisointikomponenttien lisääminen, erityisesti historiadatan esittämiseen liittyen sekä eri datan kerääminen pelkän vierailutiedon lisäksi.

Sovelluksen toteutusta on dokumentoitu koodiin sekä Javadoc-kommenteina että tavallisina koodikommenteina. Sovelluksen toteutuksesta on myös kirjoitettu lyhyt toiminnollinen kuvaus, jotta sovellusta voidaan jatkokehittää helposti myös muiden ihmisten toimesta.

6 Yhteenveto

Sovellus saatiin suunniteltujen toiminnollisuuksien osalta valmiiksi määräajassa, ja se on valmis käyttöönottoon. Sovellus nykyisellä toteutuksella pystyy seuraamaan minkä tahansa Apache httpd- tai Apache Tomcat pohjaisen verkkokaupan tai muun sivuston käyttäjiä. Sovelluksessa on kaksi valmista ja toimivaa visualisointikomponenttia: reaaliaikainen kartta sekä tallennetusta datasta muodostettava tiheyskartta.

Ohjelmistokehykseksi valittu Spring osoittautui hyväksi valinnaksi, koska sillä on hyvä tuki sovelluksessa käytetyille rajapinnoille ja tekniikoille, ja lisäksi Spring-ohjelmistokehyksen API-dokumentaatio oli selkeä ja riittävä. Sovelluksen tämänhetkisille toiminnollisuuksille Spring on tosin hieman turhan laaja, mutta etu tässä on silti se, että sovelluksen jatkokehitys ei hidastu liian yksinkertaisen ohjelmistokehyksen rajoittuneisuuteen.

Toteutuksen aikana insinöörityön tekijä on oppinut datan keräämiseen ja prosessointiin liittyvistä haasteista ja virheenkäsittelyyn sekä suorituskykyyn liittyviä asioita. Insinöörityön tekijä oppi toteutuksen aikana myös uusina asioina Google Maps API:n ja WebSocket-teknologian käytön. Sovellusta ei ole vielä otettu sisäiseen käyttöön, joten sovelluksen varsinaiseen käyttöön liittyviä asioita ja mielipiteitä ei voitu vielä käsitellä muiden kuin sovelluksen toteuttajan näkökulmasta.

Insinöörityön tekijä oli tyytyväinen työn lopputulokseen ja piti työtä mielestään sopivan monimutkaisena ja haastavana sekä suunnittelun että toteutuksen osalta. Insinöörityön tekijä olisi toivonut saavansa enemmän käytettävää aikaa tehdä sovelluksesta monipuolisemman ja laajemman tässä vaiheessa.

Lähteet

- 1 Architectural Styles and the Design of Network-based Software Architectures. Verkkodokumentti.
http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm Luettu 16.12.2015.
- 2 RFC 6455: WebSocket-protokolla. Verkkodokumentti.
<https://tools.ietf.org/html/rfc6455> Luettu 16.12.2015.
- 3 STOMP Protocol Specification, Version 1.1. Verkkodokumentti.
<http://stomp.github.io/stomp-specification-1.1.html> Luettu 16.12.2015.
- 4 Java Platform SE 8 documentation: WatchService. Verkkodokumentti.
<https://docs.oracle.com/javase/8/docs/api/java/nio/file/WatchService.html> Luettu 31.3.2016.
- 5 Apache Commons IO documentation: Tailer. Verkkodokumentti.
<https://commons.apache.org/proper/commons-io/apidocs/org/apache/commons/io/input/Tailer.html> Luettu 31.3.2016.
- 6 EclipseLink JPA User Guide. Verkkodokumentti.
https://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Mapping/Basic_Mappings/Transient Luettu 31.3.2016.
- 7 Java Platform SE 8 documentation: ExecutorService.
<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html> Luettu 31.3.2016.
- 8 Google Maps JavaScript API. Verkkodokumentti.
<https://developers.google.com/maps/documentation/javascript/maptypes> Luettu 28.1.2016.