TAMPEREEN
AMMATTIKORKEAKOULU

TIEDONHAKULOGIIKAN TOTEUTTAMINEN SELAINPOHJAISEEN SOVELLUKSEEN

Case: Ei huano -generaattori

Tuukka Ojala

Opinnäytetyö
Toukokuu 2017
Tietojenkäsittelyn koulutusohjelma
Ohjelmistotuotanto

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma
Ohjelmistotuotanto

OJALA, TUUKKA:
Tiedonhakulogiikan toteuttaminen selainpohjaiseen sovellukseen
Case: Ei huano -generaattori

Opinnäytetyö 38 sivua, joista liitteitä 3 sivua
Toukokuu 2017

Opinnäytteessä dokumentoidaan niitä suunnitteluvaiheita, toimintamalleja ja haasteita, joita ilmeni toteutettaessa tiedonhakulogiikkaa sovellukseen, joka toimii kokonaisuudessaan käyttäjän selaimessa. Sovellus ei hyödynnä minkäänlaista ulkopuolista palvelinta oman tilansa tallentamiseen. Työn toimeksiantajana toimi Vincit, ohjelmistotuotantoon, ict-palveluihin ja palvelumuotoiluun erikoistunut yritys.

Vincitin työntekijä voi saada hyvin suoritetusta työtehtävästä erityisen ”ei huano” –diplomin, jonka aiheen ja saajan päättävät muut työntekijät. Diplomiprosessi tapahtuu kokonaisuudessaan tiimiviestintäpalvelu Slackissa, joka on Vincitin sisäisen viestinnän pääasiallinen kanava. Tilaaja halusi sovelluksen, jolla aiemmin käsityönä tapahtunut diplomien valmistus automatisoitaisiin mahdollisimman pitkälle. Tavoitteena oli tehdä sovellus, jolla voisi luoda ”ei huano” –diplomeja Slack-viestien pohjalta. Tämän opinnäytetyön tarkoituksena on kuvata toimeksiannon suunnittelua ja toteutusta sekä dokumentoida sovelluksen kehittämistä tiedonhaun ja –käsittelyn osalta.

Tiedonhakulogiikan suunnitteluun vaikuttivat huomattavasti Slackin rajapinnan tarjoama data, jota jouduttiin mekaanisesti yhdistelemään useasta eri sijainnista ja sovelluksen käyttöympäristö, jossa ei voitu hyödyntää minkäänlaista tiedon esikäsittelyä. Niinpä haun optimointiin jouduttiin kiinnittämään erityistä huomiota. Valmis sovellus täytti sille asetetut vaatimukset, ja se otettiin käyttöön heti valmistuttuaan.

Asiasanat: selainsovellukset, tiedonkäsittely, Slack, Javascript

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Business Information Systems
Option of Software Development
OJALA, TUUKKA:
Implementing Data-Fetching Logic to a Browser-Based Application
Case: Ei huano -generaattori

Bachelor's thesis 38 pages, appendices 3 pages
May 2017

The purpose of this thesis was to document, design and implement data-fetching logic to an application that retrieves and processes messages from the team messaging service Slack. The application was made at Vincit, a company primarily specializing in software development and service design.

An employee at Vincit may get a so-called “not bad diploma” for a particularly well done job. The process of deciding the recipients of the diplomas is carried out entirely within Slack. There was a need for an app that would automate the diploma creation process as much as possible.

Implementing data fetching for this application posed some challenges. Some of the data provided by Slack was inconsistent and it had to be pieced together from various locations. None of the needed data could be processed in advance so particular care needed to be taken on optimizing the searching to be as efficient as possible. The application met the requirements it was given and it was successfully taken into use.

Keywords: web applications, data processing, JavaScript, Slack

SISÄLLYS

1 JOHDANTO

Tässä opinnäytteessä esittelemäni sovellus on tehty ohjelmistotalo Vincitin sisäiseen käyttöön. Vincit on ohjelmistotuotantoon ja ICT-palveluihin erikoistunut keskisuuri yritys, jonka toimenkuvaan kuuluvat verkkopalveluiden, mobiilisovellusten ja sulautettujen järjestelmien kehitys, palvelumuotoilu sekä ICT-palveluiden suunnittelu ja ylläpito.

Vincitin henkilöstön keskuudessa on vuosien saatossa syntynyt useita eri tapoja, joilla palkita kollegoja hyvin tehdystä työstä. Yksi näistä tavoista on niin kutsuttu Ei huano -diplomi. Esimerkki Ei huano –diplomista on nähtävillä kuvassa 1.

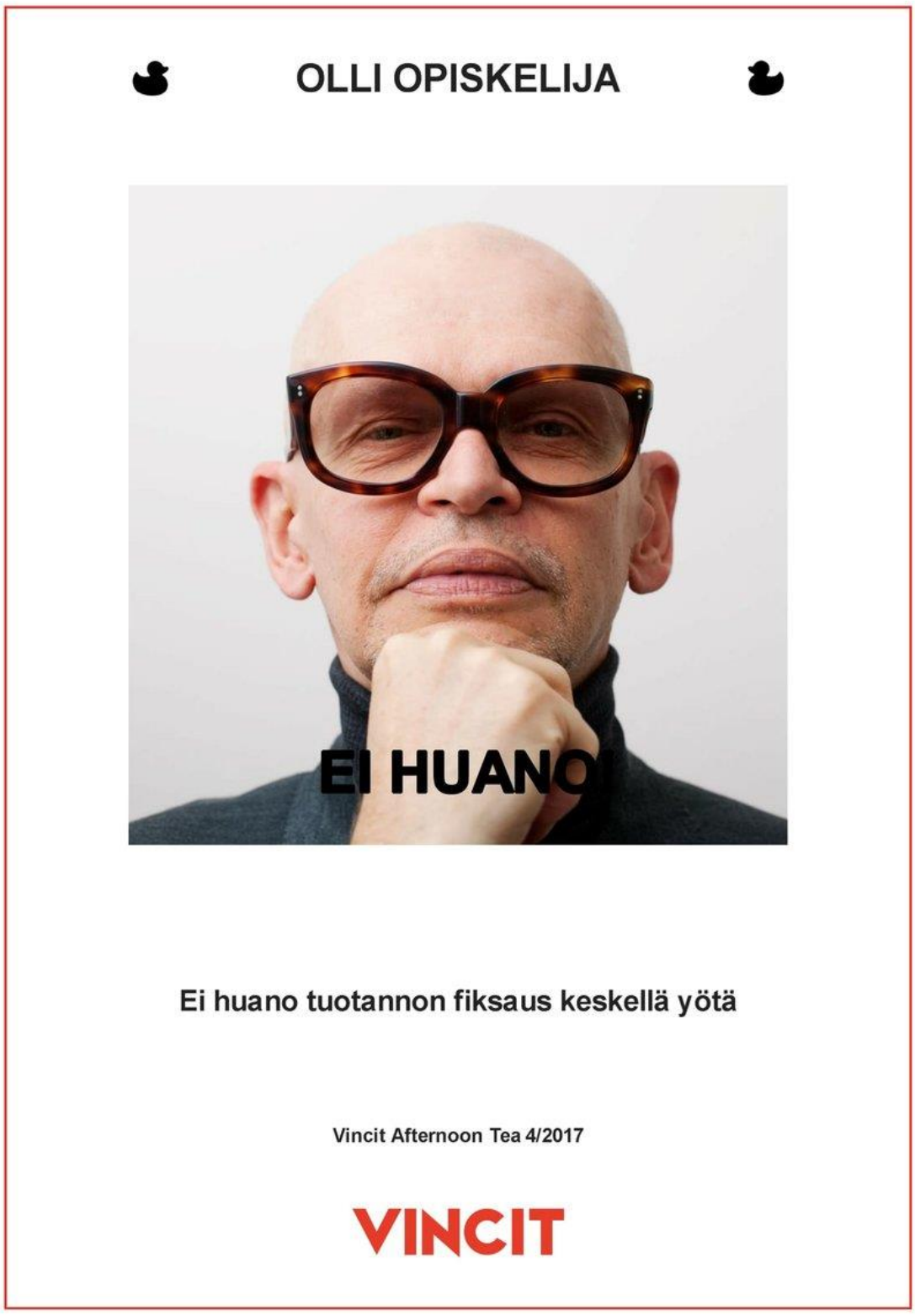
Jos yksi tai useampi työntekijä on tehnyt jotakin erityisen ansiokkaasti, voi hänet ilmiantaa millä tahansa Vincitin Slack-tiimin julkisella kanavalla. "Ilmiantoviesti" on vapaamuotoinen, mutta tyypillinen viesti voisi kuulua seuraavasti:

#eihuano tuotannon fiksaus keskellä yötä @olli.opiskelija

Tässä yhteydessä @olli.opiskelija on sen työntekijän käyttäjätunnus, jolle mahdollinen diplomi on osoitettu. Tunnuksia voi luonnollisesti mainita useamman, mikäli diplomi koskee useampaa henkilöä. Jos esimerkiksi asiakas antaa projektin kulumisesta erityisen hyvää palautetta, voi tällöin ei huano -diplomin osoittaa koko projektitiimille.

Kun viesti on lähetetty, jää lopun henkilökunnan päätettäväksi miten sille käy. Jos viesti saa vähintään kolme reaktiota, on diplomin saannin edellytykset täytetty. Ei huano -diplomit jaetaan kuukausittain niin kutsutussa Afternoon tea -tilaisuudessa, jonka pääasiallisena tarkoituksena on tiedottaa yrityksen toiminnasta keskitetysti koko henkilökunnalle.

Kuva 1. Esimerkki Ei huano -diplomista



Ei huano –diplomien luonnista vastaa Vincitin henkilöstötiimi. He olivat rakentaneet diplomit alusta pitäen käsityönä. Nyt haluttiin sovellus, Ei huano –generaattori, joka automatisoisi diplomien tekoprosessin mahdollisimman pitkälle.

Tavoitteena oli tehdä sovellus, joka hakee Vincitin Slack-tiimin `#eihuano`-kanavalta kaikki halutulla aikavälillä lähetetyt viestit, ja mahdollistaa diplomien teko ja muokkaus viestien sisällön pohjalta. Sovelluksen tuli myös osata hakea viestin mahdollisia kopioita eri Slack-kanavilta, sekä näyttää oikein sekä unicode- että slack-tiimin omat emoji-merkit.

Tarkoituksenani oli suunnitella, toteuttaa ja dokumentoida Ei huano –generaattori –sovelluksen tiedonhakulogiikka, joka hakee ja yhdistelee tietoa useasta Slackin rajapinnasta halutulla tavalla, täydentää saatua dataa tarvittaessa paikallisilla tiedoilla, ja on helposti yhdistettävissä erilliseen käyttöliittymään.

Vaikka toimeksiantona olikin tehdä kokonainen sovellus, ei vain osaa siitä, keskityn tässä raportissa ainoastaan tiedonhakulogiikkaan, sillä ainoastaan se osa sovelluksesta on itse tekemäni ja suunnittelemani. Käyttöliittymää ja muita tiedonhakulogiikkaan suoranaisesti liittymättömiä asioita käsitellään vain yleisellä tasolla, jotta lukijalle muodostuisi kokonaiskuva sovelluksen toiminnasta.

Koska Ei huano –generaattori toteutettiin Vincitin sisäisenä kehitysprojektina, voitiin sen teknologiavalinnoissa ottaa asiakasprojektia isompia riskejä. Niinpä projektissa päätettiin käyttää projektin aloitushetkellä mahdollisimman uusia teknologioita, jotka olisivat tekijöilleenkin tuoreita. Sovellus kirjoitettiin JavaScript-kielen ES2015-versiolla, joka oltiin standardoitu vajaata vuotta aiemmin, ja käyttöliittymäkirjastoksi valittiin niin ikään tuore React. Uusien tekniikoiden ohella tutuksi tuli myös tiedonhaun optimointi, sillä sovelluksen ensimmäiset versiot käsittelivät tietoa huomattavan hitaasti. Arkkitehtuurissa tehdyillä muutoksilla hakua saatiin nopeutettua merkittävästi.

Tämä opinnäyte on jaettu rakenteeltaan karkeasti työn tekovaiheiden mukaan. Toisessa luvussa paneudutaan ympäristön ja tilatun työn taustoihin sekä erityisesti siihen, miten tekemällämme sovelluksella pyrittiin muuttamaan olemassa olevia, tehottomaksi koettuja käytäntöjä. Kolmannessa luvussa keskitytään tarkemmin sovelluksen kehittämisessä käytettyihin tekniikoihin, sekä itse valitsemiimme että ennalta määrättyihin. Neljännessä ja viidennessä luvussa käsitellään tiedonhakukerroksen arkkitehtuuria ja teknistä toteutusta. Neljäs luku esittelee arkkitehtuurin sekä siinä käytettyjä tekniikoita ja tekotapoja, kun taas viidennessä luvussa käsitellään niitä

tekijöitä, joiden myötä arkkitehtuuri ja toiminta muodostui sellaiseksi kuin se muodostui.

2 SOVELLUKSEN TAUSTA JA KÄYTTÖYMPÄRISTÖ

2.1 Viestintäpalvelu Slack

Vincit käyttää sisäisessä viestinnässään Slack-viestintäpalvelua. Slack on verkkopalvelu, jota yritys tai jokin muu ryhmä voi käyttää sisäisen viestintänsä hallintaan. Tällaista ryhmää kutsutaan Slackin termein tiimiksi. Tiimin jäsenet voivat keskustella joko julkisilla kanavilla, vain kutsutuille jäsenille käytettävissä olevilla yksityisillä kanavilla tai 2-8 hengen välisissä yksityiskeskusteluissa. Slackin keskeisimpiä ominaisuuksia ovat kaiken julkisen viestinnän kattava viestiarkisto, tuki liitetiedostoille sekä työpöytä- ja mobiilisovellukset, joihin on mahdollista saada ilmoituksia haluamiltaan kanavilta. (Slack: Team communication for the 21st century)

Slackissa on myös koko joukko muita, kokonaisuuden kannalta hieman merkityksettömämpiä ominaisuuksia, joista nostetaan tässä yhteydessä esille tilatun sovelluksen kannalta kolme keskeisintä:

- Useita eri rajapintoja, joiden avulla voidaan luoda Slackia käyttäviä sovelluksia. Tällaisia sovelluksia voivat olla Ei huano -generaattorin kaltaiset Slack-tiimin sisältöä lukevat sovellukset, mutta on myös mahdollista tehdä niin kutsuttuja botteja, jotka saavat reaaliaikaista tietoa Slackin tapahtumista kuten lähetetyistä viesteistä ja voivat myös itse lähettää viestejä. (Slack: Slack API)
- Mahdollisuus lisätä tiimin sisälle omia emojeita eli hymiöitä, jotka voivat olla joko aliaksia olemassa oleville emojeille tai kokonaan uusia hymiökuvia. (Slack: Creating custom emoji)
- Mahdollisuus reagoida viestiin käyttäen mitä tahansa emojia. Reaktioiden lukumäärä sekä käytetyt reaktioemojit näytetään viestin yhteydessä. (Slack: Emoji reactions)

2.2 Ei huano –diplomien teko ennen sovellusta

Vaikka johdannossa kuvatun ”ei huano –viestin” voikin lähettää mille tahansa julkiselle Slack-kanavalle, on ne ollut tapana lähettää erityisesti tätä aihetta varten luodulle

`#eihuano`-kanavalle. Sinne myös kerätään kaikilta muilta julkisilta kanavilta sellaiset viestit, jotka alkavat jollakin seuraavista termeistä:

- `#eihuano`
- `eihuano`
- `:ela:` (Vincitin slack-tiimin oma emoji.)

”Ei huano –viestien” keräämisen apuna ollaan käytetty tarkoitukseen varta vasten tehtyä slack-bottia eli ohjelmaa, joka toimii reaaliaikaisesti Slack-tiimissä. Kyseinen botti tarkkailee kaikkea tiimin julkista viestiliikennettä ja huomattuaan jonkin edellä esiteltyjen kriteerien mukaisen viestin lähettää siitä kopion `#eihuano`-kanavalle. Näin diplomien aines saadaan koottua yhteen paikkaan.

Ennen Ei huano -generaattoria edellä esitelty Slack-botti oli ainoa diplomien tekoprosessia automatisoinut osa. Muuten diplomien rakentaminen tapahtui kokonaan käsityönä seuraavan kaavan mukaan:

- Etsittiin `#eihuano`-kanavalta sellaiset viestit, jotka vastasivat yllä esitellyn botin käyttämiä hakukriteerejä ja joissa oli vähintään kolme reaktiota.
- Etsittiin alkuperäinen, botin löytämä viesti ja laskettiin sen sekä botin luoman kopion saamat reaktiot yhteen.
- Kopioitiin viestin sisältö tekstinkäsittelyohjelmassa olevaan diplomipohjaan.
- Tulostettiin näin syntyneet diplomit yksi kerrallaan.

2.3 Uuden sovelluksen toimintavaatimukset

Ei huano -generaattori -sovellukselle ei oltu asetettu alun perin tarkkoja toimintavaatimuksia. Tehtäväksi oltiin ainoastaan annettu rakentaa sovellus, joka automatisoisi ei huano -diplomien teon ja suunnittelun mahdollisimman pitkälle kuitenkin siten, että luotuja diplomeja tuli olla mahdollista muokata ja esikatsella ennen tulostamista. Projektille oltiin nimetty yhteyshenkilö Vincitin henkilöstötiimistä, jossa sovellusta tultaisiin käyttämään, joten vaatimuksia tarkennettiin heidän toiveidensa pohjalta esitettyämme ensin oman ehdotuksemme.

Ehdotimme tehtäväksi sovellusta, joka hakisi halutulta aikaväliltä Slackin `#eihuano` – kanavan kaikki viestit. Kustakin löytyneestä viestistä olisi mahdollista luoda diplomi, johon on esitäytetty viestistä kerätyt diplomin vastaanottajien nimet sekä diplomin teksti, joka on itse viesti. Kun diplomit on luotu, voi ne esikatsella ja tulostaa yhdellä kertaa. Tulostus tapahtuu suoraan sovelluksesta.

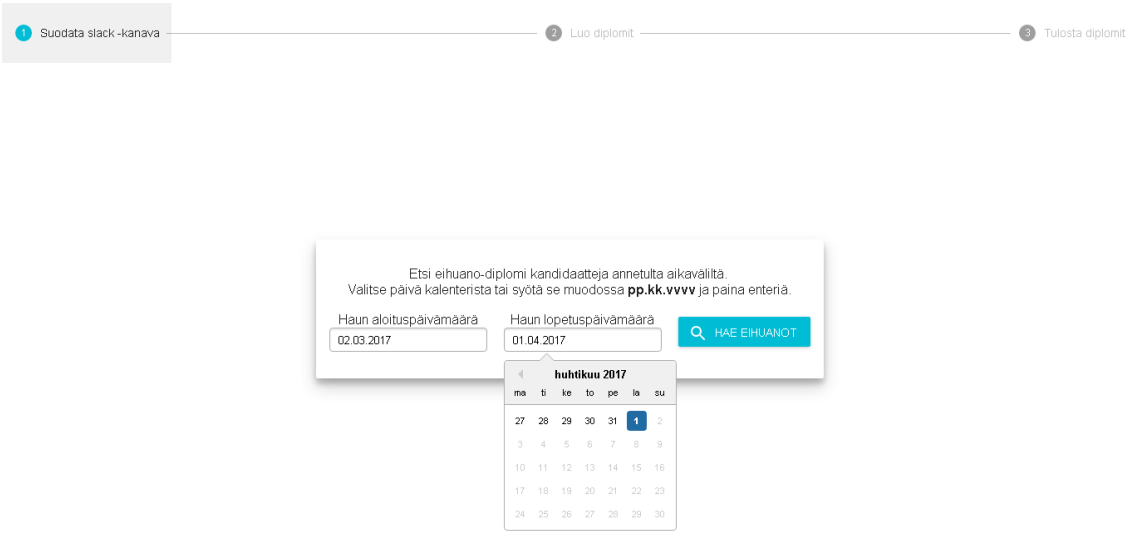
Ehdotus sovelluksen toimintavaatimuksista ja ominaisuuksista hyväksyttiin sellaisenaan. Kehitystyön edetessä vaatimuksia tosin tarkennettiin. Merkittävimpinä muutoksina viestien hakua muutettiin siten, että mukaan otettiin vain sellaiset viestit, joissa oli yhteensä kolme tai useampi reaktio. Lisäksi alkuperäisestä suunnitelmasta poiketen `#eihuano`-kanavan viestin reaktioihin laskettiin yhteen saman viestin jollakin toisella kanavalla mahdollisesti olevan viestikopion saamat reaktiot. Muut tarkennukset olivat luonteeltaan kosmeettisia, eivätkä ne vaikuttaneet tiedonhakulogiikan suunnitteluun ja toteutukseen.

2.4 Esimerkki valmiin sovelluksen käytöstä

Tämä esimerkki vastaa sovelluksen toimintaa tämän tekstin kirjoitushetkellä. Sovellus on jaettu kolmeksi eri sivuksi. Ensimmäisellä sivulla käynnistetään haku, toisella luodaan diplomit ja kolmannella ne tulostetaan.

1. Valitaan ajankohta jolta viestit halutaan hakea. Oletuksena sovellus ehdottaa haettavaksi viestejä viimeisimmän kuukauden ajalta. Esimerkki tästä vaiheesta on nähtävillä kuvassa 2.

Kuva 2. Päivämäärän valinta1



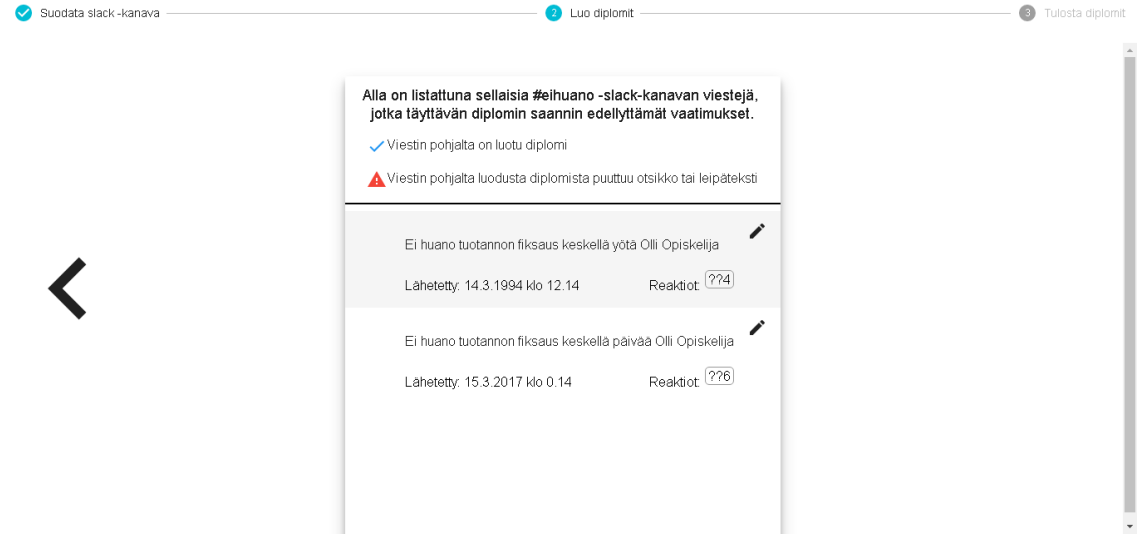
2. Painetaan "hae". Sovellus siirtyy tulossivulle. Tästä poiketen näytetään virheviesti, jos:

a. Viestejä ei löydy

b. Viestien haku ei onnistunut

3. Hakutulossivulla näytetään kaikki em. kriteerejä vastaavat `#eihuano`-slack-kanavan viestit. Kustakin viestistä näytetään viestin lähettäjä, teksti sekä kaikki sen saaneet reaktiot. Viestin yhteydessä on myös painike diplomin luomiselle tai muokkaamiselle, mikäli diplomi on jo olemassa. Esimerkki tästä vaiheesta on nähtävillä kuvassa 3.

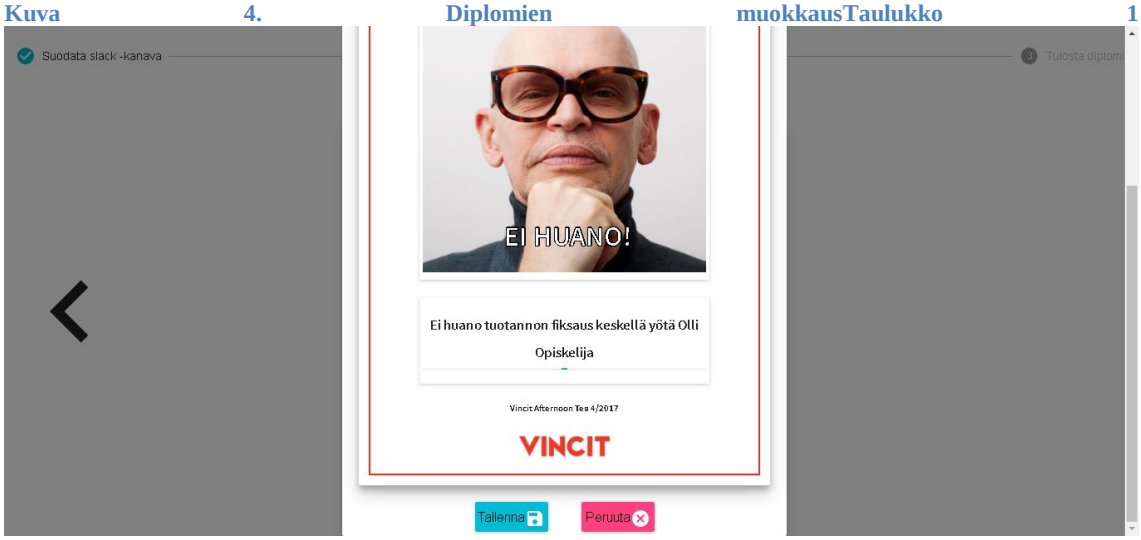
Kuva 3. Lista diplomiksi kelpaavista viesteistä



4. Luo- tai muokkaa -painiketta painettaessa avautuu modaali-ikkuna, johon diplomin tiedot syötetään. Ikkunan kenttiin on täytetty valmiiksi diplomin vastaanottajan nimi tai nimet sekä diplomin teksti, joka on oletuksena sama kuin

valittu Slack-viesti. Mikäli painettiin muokkaa-painiketta, näytetään ikkunassa aiemman tallennuksen mukainen diplomi. Tässä yhteydessä diplomin voi myös poistaa. Esimerkki tästä vaiheesta on nähtävillä kuvassa 4.

Kuva 4. Diplomien muokkausTaulukko 1



5. Kun Diplomi on valmis, painetaan "tallenna". Palataan takaisin hakutulossivulle.
6. Diplomit luodaan kohtien 4 ja 5 mukaisesti. Kun diplomeja on luotu vähintään yksi, voi käyttäjä siirtyä esikatselu- ja tulostussivulle painamalla "seuraava".
7. Tulostussivulla näytetään kaikki luodut diplomit siinä muodossa kuin ne paperille tulostuvat. Sivulla on painike "tulosta", joka käynnistää kaikkien diplomien tulostuksen, sekä painike valmiiden diplomien listaamiseen. Esimerkki tästä vaiheesta on nähtävillä kuvassa 5.

Kuva 5. Diplomien tulostaminenKuva 2

3 KÄYTETYT TEKNOLOGIAT

3.1 Johdanto

Ei huano -generaattori päädyttiin kirjoittamaan selaintekniikoilla ns. single-page application (SPA) -mallin mukaisesti, jossa sovelluksen logiikka sijaitsee yhdellä html-sivulla, jonka sisältöä päivitetään dynaamisesti sen sijaan, että näkymää vaihdettaessa ladattaisiin kokonaan uusi sivu (Takada: Modern web applications: an overview). Ohjelmointikielenä käytettiin JavaScriptiä ja sen ES2015-versiota. Käyttöliittymän teossa käytettiin React-kirjastoa, ja valmis sovellus julkaistiin Vincitin sisäisellä Dokku-palvelimella.

Sovelluksen tilaaja ja pääasiallinen käyttäjä, Vincitin henkilöstötiimi, ei asettanut erityisiä vaatimuksia sovelluksen teko- tai käyttötapaan liittyen. Niinpä tekemämme SPA-toteutus ei olisi suinkaan ollut ainoa mahdollinen tekotapa. Muita varteenotettavia toteutustapoja olisi ollut ainakin kaksi: toteuttaa sovellus käyttäjän tietokoneelle asennettavana työpöytäsovelluksena tai selainsovelluksena, jossa kaikki laskenta tapahtuu palvelimella.

Ei huano -generaattorin kirjoittaminen työpöytäsovelluksen muotoon olisi ollut ongelmallista ennen kaikkea kahdesta syystä. Työpöytäsovellus olisi pitänyt asentaa tai vähintään kopioida jokaiselle tietokoneelle jolla sitä tultaisiin käyttämään. Vincitillä käytettävien järjestelmien kirjo on laaja, sillä käytössä on niin Windowsin, Mac OS X:n / macOS:n kuin Linuxinkin eri versioita. Niinpä työpöytäkäytössä tarvittava binääripaketti olisi pitänyt kääntää erikseen joka alustalle sopivaan muotoon tai toteuttaa sovellus niin sanottuna universaalina binäärinä, joka toimisi mahdollisesti oikein kaikissa edellä mainituissa järjestelmissä. Lisäksi työpöytäsovellus olisi käytännössä sulkenut pois Ei huano -diplomien teon kaikilla muilla paitsi x86/x64-arkkitehtuuria käyttävillä laitteilla. Sovelluksen käyttö ei siis olisi ollut mahdollista esimerkiksi iOs- ja Android-tableteilla ja -puhelimilla.

Toinen syy oli tulostusasettelun toteuttaminen. Yksi harvoista sovellukselle asetetuista vaatimuksista oli voida esikatsella diplomeja ennen niiden tulostamista. Toisin sanoen diplomin tuli näkyä esikatselussa tarkalleen siten kuin se tulisi paperille tulostumaan.

Tämän ominaisuuden toteuttaminen koettiin työpöytäsovelluksena tarpeettoman hankalaksi, sillä selainpohjaisessa toteutuksessa tulostusasettelu voitaisiin määritellä suoraan CSs-tyyleillä(Mozilla Developer Network: Paged media). Näin luotu asettelu voitaisiin näyttää samalla tavalla sekä ruudulla että paperilla.

Työpöytätoteutus ei olisi ollut Ei huano -generaattorin tapauksessa järkevä vaihtoehto, sillä siitä ei olisi ollut mitään etua selainpohjaiseen toteutukseen nähden. Sillä oltaisiin ainoastaan tehty sovelluksen kehitys, käyttöönotto ja mahdollisesti myös käyttö tarpeettoman mutkikkaaksi.

Toinen vaihtoehto sovelluksen toteuttamiseen olisi ollut toteuttaa se siten, että palvelimella oleva ohjelma olisi käsitellyt kunkin pyydetyn sivun valmiiksi, ja antanut selaimelle valmiin html-sivun. Tämä tekotapa vastaa mallia, jolla web-sovelluksia on tehty ennen SPA-toteutuksia sekä niiden ohella. Tällöin mahdollisia ohjelmointikieliä tiedonhaun toteuttamiseen olisi ollut useita, Javascriptin lisäksi esimerkiksi Php, Python, Java tai Ruby.

Tämä toteutustapa olisi jo ratkaissut työpöytäsovelluksen merkittävimmät ongelmat. Sovellus olisi verkossa, joten sitä voitaisiin käyttää periaatteessa millä tahansa internetiin pääsevällä päätelaitteella ilman lisäasennuksia, ja tulostusasettelukin voitaisiin määritellä CSS-tyyleillä.

Keskeisin syy tämän toteutustavan hylkäämiseen oli SPA-toteutukseen verrattuna "verkkosivumaisempi" käyttökokemus. Jokainen näkymän vaihtaminen, kuten hakutulosten lataus sekä diplomien luonti ja esikatselu, olisi edellyttänyt uutta sivulatausta, jolloin koko käyttöliittymä olisi latautunut uudestaan. SPA-tyyppisessä toteutuksessa koettiin olevan helpompi luoda käyttökokemuksesta ”sovellusmaisempi”. Tällainen vaikutelma saadaan aikaan lataamalla käyttöliittymässä uudelleen vain ne elementit, joiden sisältöä on muutettu sen sijaan, että ladattaisiin yhden muutoksen takia koko sivu uudelleen (Takada: Modern web applications: an overview).

Kehitystyö olisi ollut myös marginaalisesti haastavampaa, sillä palvelinpohjainen toteutus olisi edellyttänyt erillisen kehitysympäristön käyttöä. Sen sijaan SPA-toteutuksen kehitys ei tarvinnut node.js-ajoympäristön asennuksen lisäksi muita lisätyökaluja. Itse sovellus ei SPA-toteutuksena tarvitse välttämättä toimiakseen

verkkopalvelinta lainkaan; sitä voi ajaa esimerkiksi siirrettävältä muistivälineeltä niin halutessaan.

On sanottava, että valinta SPA- ja palvelinpohjaisen web-toteutuksen välillä oli ennen kaikkea niin sanottu makukysymys. Kummallakin tavalla oltaisiin päästy tyydyttävään lopputulokseen, mutta SPA-toteutuksella koettiin olevan helpompi tehdä käyttökokemuksesta edellä kuvatulla tavalla sovellusmainen. Lisäksi tällä projektilla haluttiin testata tekniikoita, jotka olisivat paitsi yleisesti niin myös tiimin jäsenille uusia, ja tällä tekotavalla päästiin testaamaan tekohetkellä web-kehityksen tuoreita trendejä.

3.2 Javascriptin ominaisuuksia ja ominaispiirteitä

3.2.1 Johdanto

Javascript on ohjelmointikieli, jota käytetään yleisimmin dynaamisen toiminnallisuuden lisäämiseen verkkosivuille. Tämän lisäksi sitä käytetään nykyisin myös palvelinympäristöissä. Kielen ensimmäinen versio julkaistiin Netscape Navigator –selaimessa vuonna 1995. Sittemmin kielen kehityksestä on siirtynyt vastaamaan Ecma-standardointiorganisaatio. Javascript-kielen spesifikaatiosta käytetäänkin nimitystä EcmaScript; JavaScript-nimi otettiin käyttöön lähinnä markkinointisyistä, ja sitä käytetään ainoastaan puhuttaessa juuri tästä EcmaScript-standardin toteutuksesta. (Rauschmayer: How JavaScript was created; Rauschmayer: Standardization: EcmaScript)

EcmaScript-kieli on saanut vaikutteita useasta ohjelmointikielestä, kuten Scheme, Python, Javaja Self. EcmaScript ei edusta voimakkaasti mitään erityistä ohjelmointiparadigmaa; niin olio- kuin funktionaalinen ohjelmointi on mahdollista. Esimerkiksi EcmaScriptin syntaksi on saanut vaikutteita Java-kielestä, mutta funktioita on mahdollista välittää toisten funktioiden parametreinä Pythonin ja Schemen tapaan. C++- ja Java-kielistä poiketen EcmaScriptin oliot ovat prototyyppi-, eivät luokkapohjaisia. (Rauschmayer: Basic JavaScript)

3.2.2 ES2015-version mukanaan tuomat muutokset

EcmaScript-kielestä on julkaistu useita eri versioita kielen standardoinnin jälkeen vuonna 1997. Viimeisin standardoitu versio, EcmaScript 2016, julkaistiin kesäkuussa 2016, juuri Ei huano -generaattorin valmistumisen jälkeen. Projektin aloitushetkellä EcmaScript-kielen uusin versio oli EcmaScript 6, uudistetun nimeämiskäytännön mukaan EcmaScript 2015, joka oltiin julkaistu vuoden 2015 kesäkuussa. Tällöin standardi ei ollut vielä täysin tuettu yhdessäkään selaimessa. Standardia haluttiin kuitenkin käyttää tässä projektissa vielä puutteellisesti tuettunakin, sillä se toi kieleen useita kehittämistä helpottavia ominaisuuksia, minkä lisäksi se saatettiin helposti muuttaa useimpien selainten tukemaan EcmaScript 5 –version mukaiseen muotoon (ks. 3.4). (Rauschmayer: About EcmaScript 6 (ES6))

EcmaScriptin versio 2015 toi kieleen useita sellaisia toimintoja, joiden puutetta oltiin ennen paikattu erilaisilla vakiintuneilla toteutusmalleilla. EcmaScript-kielen vanhemmissa versioissa ei esimerkiksi ollut mahdollista määritellä muuttujien näkyvyyttä lohkotasolla, vaan funktion sisällä määritellyt muuttujat olivat käytettävissä koko funktiossa. Niinpä oli syntynyt käytäntö, jossa kapeampaa muuttujanäkyvyyttä edellyttänyt koodi kirjoitettiin anonyymiin funktioon, joka suoritettiin välittömästi. Tästä IIFE (Immediately invoced function execution) –mallista tuli tarpeeton EcmaScript 2015 –version myötä, sillä siinä tuli mahdolliseksi määritellä muuttujat lohkotasolla. (Rauschmayer: Core ES6 features)

Muita EcmaScript 2015 –versiossa tulleita uusia syntaksiominaisuuksia ovat mm.:

- Virallinen tuki moduuleille
- Taulukoiden ja objektien rakenteen purku muuttujiin (destructuring)
- Oletusarvot funktioparametreille
- Luokkia matkiva syntaksi korvaamaan rakentajafunktioiden käyttö
  (Rauschmayer: Core ES6 features)

  Sovellusta suunniteltaessa koettiin, että tämänkaltaisten ominaisuuksien käyttö tekisi ohjelmakoodista luettavampaa ja helpompaa kirjoittaa. Uusimman mahdollisen kieliversion käyttö nähtiin järkevänä myös sovelluksen elinkaaren

kannalta; ylläpito pitkänkin ajan jälkeen voisi olla helpompaa, mikäli ohjelman koodi on tekohetkellä tuoreinta mahdollista.

3.2.3 Asynkronisten operaatioiden toteutus ja hallinta

EcmaScriptin tapa toteuttaa rinnakkaisten operaatioiden suorittaminen poikkeaa esimerkiksi Pythonista ja Javasta. Edellä mainitut kielet ovat ns. monisäikeisiä. Ne voivat siis suorittaa useita operaatioita rinnakkain toisistaan riippumatta. Jos esimerkiksi käyttöliittymä ja tiedonhaku sijaitsisivat omissa säikeissään, voisi käyttöliittymää käyttää samaan aikaan kun tietoa haetaan taustalla. (Mozilla Developer Network: Concurrency Model and Event Loop)

EcmaScript on yksisäikeinen kieli. Ohjelmien suorittaminen perustuu ns. tapahtumajonon ja -silmukan (engl. event queue, event loop) käyttöön. Kun koodissa kutsutaan funktiota, lisätään se tapahtumajonoon. Tapahtumasilmukka hakee joka kierroksella tapahtumajonosta uuden funktiokutsun ja suorittaa sen. EcmaScriptissä ei siis ole valmista tapaa rinnakkaisuuden toteuttamiseen, vaan kaikki operaatiot on suoritettava peräkkäin. Poikkeuksen tähän muodostavat mahdollisesti käytetyt, EcmaScript-tulkin ulkopuolella suoritettavat lisäosat, jotka voivat tukea säikeistystä. (Mozilla Developer Network: Concurrency Model and Event Loop)

Koska EcmaScriptissä kaikki toiminnot jakavat saman tapahtumasilmukan, tarkoittaa tämä sitä, että myös käyttöliittymän tapahtumakäsittelyfunktiot suoritettaisiin vasta ajallaan muiden suorittumassa olevien funktioiden jälkeen. Jos esimerkiksi viestien haku Ei huano –generaattorissa tapahtuisi kokonaisuudessaan yhden ja saman funktion sisällä, ei viestien hakua olisi mahdollista peruuttaa ennen hakufunktion palautumista. Tämä johtuu siitä, että peruuta-napin painaminen suorittaa funktion, joka suoritettaisiin vasta hakufunktion palauduttua.

Ratkaisu tähän ongelmaan on kirjoittaa ohjelmakoodi siten, että yksittäinen funktio käyttää mahdollisimman vähän aikaa. Näin tapahtumasilmukka voi suorittaa muita funktioita sillä välin, kun suorittaminen vaihtuu funktiosta toiseen.

Tästä syystä EcmaScriptissä on erityisen merkityksellistä erotella synkroniset ja asynkroniset operaatiot toisistaan. Ei huano -generaattorin tiedonhakulogiikka on pitkälti asynkroninen, sillä tietoa haetaan Slackin rajapinnasta, ja verkkohaun kaltaiset operaatiot ovat asynkronisia, sillä niiden käyttämää aikaa ei voida arvioida ennakkoon.

Asynkronisuuden toteuttamiseen EcmaScriptissä on kaksi yleistä tapaa: callback-funktiot ja Promise-objektit. Callback-funktioissa asynkronisen operaation suorittavalle funktiokutsulle annetaan parametriksi funktio, joka suoritetaan vasta operaation päätyttyä. Tämä callback-funktio saa parametreinaan suoritetun operaation lopputuloksen, usein virheobjektin ja funktion suorittamisen tuloksena saatua dataa.

Callback-funktioiden käytössä on kaksi ongelmaa, jotka voivat tehdä niiden käytöstä työlästä: Jos koodissa joudutaan suorittamaan peräkkäin useita asynkronisia operaatioita, esimerkiksi tietokantakyselyjä, joutuu koodin kirjoittamaan useisiin sisäkkäisiin funktioihin tähän tapaan:

```
performQuery('query1', (err, data) => {
    performAnotherQuery('query', (err, data) => {
        ...
    });
});
```

Tämä voi tehdä koodista hankalalukuista. Lisäksi joka callback-funktiossa täytyy erikseen tarkistaa, onko funktion mukana tullut virhettä, ja palauttaa se mikäli sellainen on saatu. Sisäkkäisten callback-funktioiden ongelmaa on ratkottu erilaisilla ulkopuolisilla kirjastoilla, kuten esimerkiksi Async (Async: Projektin esittely). Näillä kirjastoilla voidaan mm. ”niputtaa” useita asynkronisia operaatioita yhteen siten, että suorittaminen jatkuu joko kaikkien operaatioiden valmistuttua, tai heti kun yksikin operaatioista on valmistunut.

Promise-objekteilla on pyritty ratkaisemaan callback-funktioiden käytössä havaittuja ongelmia. Promise-objekteja käytettäessä asynkronisen operaation suorittava funktio palauttaa objektin, jonka then()-funktiossa annettuna parametrina oleva funktio suoritetaan operaation päätyttyä, ja palauttaa uuden promise-objektin. Tämä mahdollistaa promise-objektien ketjuttamisen sekä sen, että suorituksessa tulleet virheet

voidaan käsitellä keskitetysti suoritusketjun päätyttyä. Lisäksi Promise-rajapinta sisältää edellä kuvaillut, mm. Async-kirjaston tarjoamat ominaisuudet useiden asynkronisten operaatioiden suorittamiseen ja hallintaan. (Mozilla Developer Network: Promise)

Ei huano -generaattorin tiedonhakulogiikan kaikki asynkroniset operaatiot on toteutettu promise-pohjaisesti. Alun perin käytetyn Slack-kirjaston takia api-kutsujen tulokset jouduttiin hakemaan callback-funktioilla, mutta myöhemmin niissäkin voitiin siirtyä promise-pohjaiseen toteutukseen, kun Slackin tekemä rajapintakirjasto vaihdettiin omatekoiseen.

3.3 Express-sovellus tiedonhakulogiikkaa varten

Tekniikkavalintoja tehtäessä oli tiedonhakulogiikka tarkoitus sijoittaa kokonaan omalle palvelimelleen. Käytännössä tälle palvelimelle tehtäisiin rajapinta, jonka avulla käyttöliittymä voisi hakea tarvitsemansa Slack-viestit. Näin käyttöliittymä ja tiedonhaku saataisiin toisistaan riippumattomiksi kokonaisuuksiksi, ja samalla mahdollistettaisiin myös keskeneräisten diplomien tallennus, mikäli sellaiselle ominaisuudelle ilmenisi myöhemmin tarvetta.

Palvelimen koodia oli tarkoitus suorittaa Node.js-ajoympäristössä. Node.js on itsenäinen versio Googlen V8-Javascript-tulkista. Rajapinnan tekoon valittiin käytettäväksi Express, sillä se on eräs käytetyimpiä palvelinkehyksiä Node.js:lle, ja lisäksi niin kutsutun MEAN stackin (MEAN = MongoDB, Express, AngularJS, Node) palvelinkehys (NPM: Most starred packages; Mean.io: projektin esittely). Lisäksi Slackin rajapinnan kanssa kommunikointiin suunniteltiin käytettäväksi Slackin omaa Javascript-kirjastoa. Peruste ylipäätään minkään http-kehyksen käyttöön oli mahdollinen tuleva laajennustarve. Alkuperäisessä suunnitelmassa rajapintaan olisi tullut vain yksi kutsu, ja se oltaisiin voitu hyvin toteuttaa Node.js:n mukana tulevalla HTTP-moduulilla. (Node.js-projektin kotisivu; Node.js: HTTP)

3.4 JavaScript-koodin jälkikäsittelytyökalut

Vaikka JavaScript oltiinkin valittu käytettäväksi ohjelmointikieleksi, jouduttiin kehityksessä käyttämään muutamia työkaluja ennen kuin sovellusta saattoi käyttää selaimessa. Näitä työkaluja olivat Babel, jolla kirjoittamamme uusi, vielä osittain ei-tuettu Javascript käännettiin laajemmin tuettuun muotoon, ja Webpack, joka paketoi automaattisesti kaiken sovelluksessa olleen koodin yhdeksi tiedostoksi.

JavaScriptin ES2015-versiolla kirjoitettujen sovellusten keskeisinpiä ongelmia on se, ettei standardi ole edelleenkään täysin tuettu kaikissa selaimissa ja selainversioissa, jotka on julkaistu version standardoimisen jälkeen (Kangax: ECMAScript 6 compatibility table). Jotta koodi olisi mahdollisimman yhteensopivaa eri selainten välillä, täytyy se kääntää varmuuden vuoksi JavaScriptin ES5-version mukaiseksi koodiksi. Tämä standardi julkaistiin vuonna 2009, joten ikänsä puolesta sitä voi pitää tähän tarkoitukseen riittävän tuettuna (Rauschmayer: About ECMAScript 6 (ES6)).

Laajemman yhteensopivuuden takaamiseksi kaikki lähdekoodi ajettiin käännösvaiheessa Babel-ohjelman läpi. Babel on niin kutsuttu transpiler, joka kääntää saamastaan lähdekoodista ES5-standardin mukaista koodia. Babel koostuu useista osista, joilla määritellään sen tukemat ominaisuudet. Näin Babel ei rajoitu ainoastaan ES2015-koodin kääntämiseen, vaan sitä voi käyttää myös ES2015-standardia uudempien, vielä kokeellisten Javascriptin ominaisuuksien käyttämiseen. (Babel: Projektin esittely)

Ei huano -generaattorin lähdekoodi koostui useista kymmenistä JavaScript-tiedostoista. Kun mukaan lisäksi lasketaan kaikki käytetyt paketit, nousee tiedostojen määrä useisiin satoihin. Niinpä käsittelyn helpottamiseksi sovelluksesta tehdään kaikkine riippuvuuksineen yksi paketti, joka ladataan muistiin kerralla sovelluksen avaamisen yhteydessä. Paketointiin käytettiin Webpackia, sillä se oli parhaiten yhteensopiva käytettyjen kirjastojen kanssa. Ennen Webpackia samaan tarkoitukseen kokeiltiin Browserifya, mutta sitä ei voitu enää käyttää, sillä tiedonhakulogiikan toimintatapa osoittautui yhteensopimattomaksi sen kanssa.

3.5 Valmiin sovelluksen julkaiseminen Dokku-ympäristössä

Ei huano -generaattorin julkaisussa käytettiin Vincitin sisäisessä verkossa olevaa Dokku-palvelinta. Dokku on Docker-ohjelmiston päälle rakennettu, avoimen lähdekoodin PaaS-toteutus. PaaS (Platform as a Service) on palvelumuoto, jossa valmis ohjelmakoodi syötetään järjestelmään, joka pystyttää automaattisesti koko sovelluksen tarvitseman infrastruktuurin, kuten esimerkiksi http- ja tietokantapalvelimen. (Dokku: projektin esittely; Butler: PaaS Primer)

Eräs esimerkki PaaS-toteutuksesta on Heroku, jonka käyttö perustuu niin kutsuttuihin buildpackeihin. Buildpack on kokoelma ohjelmia, joilla tutkitaan automaattisesti minkälaisilla tekniikoilla sovellus on kirjoitettu, ja asennetaan kaikki sen tarvitsemat ohjelmat ja palvelut. Herokuun luodaan sovellukselle profiili, jossa ilmoitetaan millaisia palveluita (web-palvelin, tietokanta...) se tarvitsee. Sovellus julkaistaan viemällä koodi Herokuun git-versionhallintajärjestelmää käyttäen. Kun Heroku on vastaanottanut koodin, etsii se käytettäväksi projektin ohjelmointikielen ja -kehyksien mukaisen buildpackin, ja ottaa sen avulla sovelluksen käyttöön. (Heroku: tuotteen kotisivu; Heroku: Creating a Buildpack)

Dokku on toiminnallisilta osiltaan Heroku-kopio, jonka voi asentaa omalle palvelimelleen. Se on myös suoraan yhteensopiva Herokun buildpackien kanssa. Dokkua käytetään Vincitillä yleisesti sisäisten sovelluksien sekä kokeiluversioiden julkaisuun, joten sen katsottiin olevan luonteva vaihtoehto myös Ei huano –generaattorin julkaisemiseen.

4 TIEDONHAKULOGIIKAN SUUNNITTELU JA RAKENNE

4.1 Johdanto

Ei huano -generaattorin tiedonhakulogiikka on rakennettu käyttöliittymästä erilliseksi osaksi. Se käännetään tällä hetkellä osaksi käyttöliittymän koodia, mutta tarvittaessa se on mahdollista irrottaa omaksi ohjelmakseen. Tässä luvussa käsitellään tiedonhakulogiikan arkkitehtuuria ja toimintaa moduuli kerrallaan, sekä esitellään kunkin moduulin kannalta olennainen osuus Slackin rajapinnasta. Arkkitehtuuriin johtaneita syitä käydään tarkemmin läpi luvussa 5.

Tiedonhakulogiikka koostuu seuraavista moduuleista:

- BotMessageDecoder: hakee slack-botin lähettämien viestikopioiden alkuperäiset viestit
- EmojiManager: Luokka kaikkien käytettävissä olevien emoji-merkkien, niin Unicode-standardin kuin Slack-tiimin omien, hakemiseen tekstimuotoisten aliasten perusteella.
- EntityStore: Luokka Slack-tiimin käyttäjien ja kanavien säilytykseen ja hakemiseen.
- MessageDecoder: Luokka yksittäisen viestin sisällön muuntamiseen käyttöliittymään sopivaksi.
- SlackController: Luokka, jossa haetaan `#eihuano`-kanavan viestit, ja muodostetaan niistä edellä mainittuja moduuleja käyttäen viestilista käyttöliittymää varten.

Lisäksi mukana on useita apufunktioita, moduuli toimintalokin ylläpitämiseen sekä rajapintakirjasto Slackin kanssa kommunikointiin, mutta ne eivät ole toteutuksen kuvauksen kannalta olennaisia eikä niitä käsitellä erikseen tässä yhteydessä.

Tiedonhakulogiikan arkkitehtuuri on rakennettu pitkälti olio-ohjelmoinnin periaatteiden mukaisesti. Jokainen tiedonhakulogiikan osa, kuten viestien käsittely tai emoji-listan ylläpitäminen, on oma, itsenäinen luokkansa, jota käytetään paitsi toisiinsa liittyvien funktioiden ryhmittelyyn niin myös tilan tallentamiseen. Nykyisen kaltainen

olioarkkitehtuuri ei olisi ollut ainoa mahdollinen vaihtoehto, mutta projektin aloitushetkellä se oli tekijälleen tutuin ja luontevin.

Jokainen tiedonhakulogiikan luokka ottaa vastaan rakentajassaan tarvitsemansa riippuvuudet muihin luokkiin. Näin mahdollistetaan riippuvuuksien korvaus toisilla versioilla ajonaikaisesti esimerkiksi testaustilanteessa, jossa ei välttämättä haluta hakea oikeaa dataa verkosta. Lisäksi Slackin rajapinnasta saatavaa tietoa varastoivat luokat pitävät sisällään update-metodin, joka hakee rajapinnasta tarvittavan datan.

Ei huano -generaattorissa käytettiin Slackin rest-rajapintaa viestien hakemiseen. Rajapintakutsujen autentikointiin käytetään avainta, joka on mahdollista luoda Slack-tiimin hallintasivuilta. Rajapinta palauttaa ainoastaan JSON-muotoista dataa.

4.2 SlackController

SlackController on luokka, jota kautta haetaan kaikki `#eihuano`-Slack-kanavalla olevat, diplomin kriteerit täyttävät viestit halutulta aikaväliltä. Luokassa on yksi julkinen metodi, `getEiHuanoMessages(oldest, latest)`, jossa oldest on vanhimman halutun viestin aikaleima ja latest uusimman halutun viestin aikaleima ns. unix-muodossa, eli sekunteina 01.01.1970 00:00 jälkeen. Metodi palauttaa Promise-objektin, jonka suoritusarvossa ovat ei huano -viestit. Esimerkki tämän funktion tuottamasta datasta on nähtävillä liitteessä 3.

Viestien haku etenee seuraavassa järjestyksessä:

- Ladataan muistiin Slack-tiimin käyttäjät, kanavat ja emojit.
- Haetaan `#eihuano`-kanavan viestihistoria oldest- ja latest-arvojen mukaiselta aikaväliltä.
- Käydään viestit läpi. Jos viesti osoittautuu subtype-kentän mukaisesti botin lähettämäksi (ks. liite 2), laitetaan se niin kutsuttuun bottiviestien pinoon, jossa olevat viestit täydennetään ennen kanavan kaikkien viestien käsittelyä. Muussa tapauksessa viesti laitetaan kanavalle ensisijaisesti lähetettyjen viestien pinoon.
- Haetaan botin lähettämiä kopioviestejä vastaavat alkuperäisviestit BotMessageDecoderilla (ks. 4.5) ja lisätään ne ensisijaisesti kanavalle

lähetettyjen viestien pinoon. Näin viestikopiot on korvattu alkuperäisillä kuitenkin siten, että viestien reaktiomäärä on kummankin viestistä olemassa olevan kappaleen reaktioiden summa.

- Suodatetaan näin saadusta pinosta pois kaikki viestit, joilla on vähemmän kuin kolme reaktiota.
- Puretaan jäljelle jääneen pinon viestit käyttöliittymän käyttämään muotoon MessageDecoderilla (ks. 4.3).

4.3 MessageDecoder

MessageDecoder on kenties keskeisin tiedonhakulogiikan komponentti. Se sisältää luokkametodeja, joilla Slackin rajapinnasta saadusta viestiobjektista (ks. liite 1 ja liite 2) poimitaan vain Ei huano -generaattorin kannalta olennainen tieto, eli viestin aikaleima, lähettäjä, teksti, tekstissä mainitut käyttäjät sekä reaktiot. Lisäksi viestistä tehdään käyttöliittymää varten lukukelpoisempi korvaamalla viestissä olevat kanavien ja käyttäjien tunnisteet niiden oikeilla nimillä, ja muuttamalla jotkin koodatut erikoismerkit UniCode-merkeiksi.

Slackin rajapinnan palauttamissa kanavan viesteissä käyttäjät ja kanavat eivät esiinny niminä. Jos viestiin on esimerkiksi kirjoitettu `@olli.opiskelija`, muuttuu se Slackin rajapinnassa muotoon `<@u123456abc>`, jossa `123456abc` on `@olli.opiskelija`-käyttäjän yksilöllinen alfanumeerinen tunniste. Samoin käy kanavien nimille: ne muuttuvat muotoon `<#c123456abc>`, jossa `123456abc` on kanavan yksilöllinen alfanumeerinen tunniste. Tämä johtuu siitä, että niin kanavien kuin käyttäjienkin nimiä on mahdollista vaihtaa, joten oikeellisen tiedon varmistamiseksi on rajapinnan käyttäjän syytä hakea nimitiedot aina erikseen (Slack: Rename a channel; Slack: Change your username). MessageDecoder etsii viestistä kaikki tällä tavoin koodatut tiedot, ja korvaa ne EntityStorea käyttäen kunkin tiedon nimellä (ks. 4.4).

Kuten luvussa 2.1 mainittiin, on Slack-tiimin sisällä mahdollista asettaa omia emojeita. Tästä syystä niitä voi olla kahdenlaisia: joko Unicode-merkistön mukaisia tai linkkejä, jotka viittaavat Slack-tiimiin ladattuihin emojikuviin. Kummankin tyyppiset emojit kirjoitetaan viestiin samalla tavalla, `:tunnus:`, jossa tunnus on emojin yksilöllinen nimi, esimerkiksi thumbsup. MessageDecoder etsii EmojiManageria käyttäen viestistä

kaikki emojit (ks, 4.4). Jos löydetty emoji on Unicode-merkki, korvataan se vastaavalla merkillä. Jos löydetty emoji on taas tiimin oma emoji, jätetään emojin tunnus paikoilleen ja lisätään emojin kuvalinkki erilliseen objektiin, jonka kenttien niminä toimivat emojien tunnukset. Käyttöliittymä voi käyttää tätä objektia kuvien näyttämiseen. Myös reaktioiden emojeille tehdään vastaava korvaustoimenpide.

Näiden lisäksi MessageDecoder korvaa viestissä mahdollisesti esiintyvän "`#eihuano`" -sanan muodolla "Ei huano" sekä muuttaa html-koodatut <, > ja & -merkit ja hexamuotoiset merkit UniCode-merkeiksi. Luokan ainoa julkinen metodi `decodeMessages(message)` ottaa vastaan Slackin rajapinnan palauttaman viestiobjektin ja palauttaa Promise-objektin, jonka suoritusarvona on käsitelty viesti sekä customEmoji-objekti, johon on koottu kaikki viestissä ja reaktioissa olleet tiimin emojien kuvalinkit. Liitteessä 3 olevat messages-taulukon objektit ovat MessageDecoderin luomia.

4.4 EntityStore ja EmojiManager

EntityStore on eräänlainen välimuistiluokka, jossa pidetään Vincitin Slack-tiimin kanava- ja käyttäjäluettelo kokonaisuudessaan. Se sisältää metodit kanavan ja käyttäjän tietojen palauttamiseen nimen tai tunnisteen perusteella. Tiedot palautetaan Slackin rajapinnan mukaisina objekteina sellaisenaan.

EmojiManager on EntityStoren tapaan välimuistiluokka. Siinä säilötään kaikki kulloinkin käytettävissä olevat emojit. Näitä ovat lista kaikista Slackin tuntemista UniCode-standardin mukaisista emojeista sekä Vincitin Slack-tiimin omat emojit.

EmojiManager palauttaa emojin tunnusta vastaavan emoji-merkin, joka on yleensä unicode-merkki tai kuvalinkki. Toisinaan haettu emoji ei ole näistä kumpikaan, sillä kuten luvussa 2.1 mainittiin, voi tiimin sisällä määritellä myös emoji-aliaksia. Niinpä emoji-listaa haettaessa täytyy erikseen tarkistaa mitkä tiimin emojeista ovat aliaksia, ja korvata ne EmojiManagerin sisäisessä emojikokoelmassa aliasta vastaavalla emojilla, oli kyseessä sitten kuvalinkki tai unicode-merkki.

4.5 BotMessageDecoder

BotMessageDecoder pitää kirjaa niin kutsutuista bottiviesteistä eli viesteistä, jotka ovat botin luomia kopioita jollekin muulle kanavalle lähetetyistä viesteistä. Lisäksi se yrittää etsiä kunkin viestin alkuperäisen version.

Kun BotMessageDecoderilla on lista botin lähettämistä viesteistä, pyrkii se ensin lajittelemaan viestit niiden alkuperäisten kanavien mukaan. Kanavan nimen voi olettaa esiintyvän aina viestin lopussa, joten sitä voidaan käyttää alkuperäisen lähetyskanavan todentamiseen. Mikäli kanavaa ei syystä tai toisesta löydy joko itse viestistä tai Slack-kanavien joukosta, laitetaan kyseinen viesti ns. orpojen viestien pinoon. Koska näiden viestien alkuperää ei voida jäljittää, täytyy ne käsitellä erikseen. Lajittelun yhteydessä otetaan myös ylös vanhimman ja uusimman bottiviestin aikaleima kullakin kanavalla.

Kun lajittelu on tehty, hakee BotMessageDecoder kunkin kanavan viestit vanhimman ja uusimman viestin mukaiselta aikaväliltä. Kun kopioviestiä vastaava alkuperäinen viesti on löytynyt, korvataan kopioviestissä olevat puutteelliset lähettäjätiedot alkuperäisviestin tiedoilla, ja lasketaan kummankin viestin saamat reaktiot yhteen. Lisäksi kopioviestistä poistetaan alkuperäisen kanavan nimi. Orpojen viestien pinoa käsiteltäessä ei voida hyödyntää alkuperäisiä viestejä, joten niiden kohdalla haetaan täydelliset lähettäjätiedot EntityStoresta (ks. 4.4).

5 Tiedonhakulogiikan toteutuksessa ilmenneet haasteet

5.1 Siirtyminen pois erillisen tiedonhakupalvelimen käytöstä

Ei huano -generaattorin käyttöliittymää ja tiedonhakulogiikkaa oltiin ehditty kehittää jo pitkään erillisinä komponentteina. Kun perustoiminnallisuus oltiin saatu valmiiksi kummassakin, tuli aika kokeilla niiden toimimista yhteen. Silloin huomattiin, että erillinen palvelin tiedonhaulle on käytännössä turha, sillä rajapinnan ainoa tehtävä oli kutsua metodia, joka suoritti varsinaisen haun ja käsittelyn. Niin päätettiin poistaa erillinen palvelin kokonaan ja siirtää hakukoodi osaksi käyttöliittymää kuitenkin siten, että käyttöliittymä ja hakulogiikka olisivat jatkossakin kaksi täysin erillistä kokonaisuutta, jotka eivät olisi riippuvaisia toisistaan.

Siirtyminen ei kuitenkaan sujunut täysin ongelmitta. Käyttämämme Slack-rajapintakirjaston huomattiin olevan soveltumaton selainkäyttöön, sillä se ei käyttänyt omassa verkkoliikenteessään selainten tukemia ominaisuuksia. Slackin oma rajapintakirjasto päädyttiin korvaamaan omatekoisella kirjastolla, jonka rajapinta tehtiin vastaamaan Slackin kirjastoa mahdollisimman pitkälle, jotta käytetty kirjasto voitaisiin vaihtaa mahdollisimman pienin muutostöin tiedonhakulogiikan lähdekoodissa. Myöhemmin tekemäämme kirjastoa tosin optimoitiin muuttamalla asynkroniset toiminnot callback-pohjaisesta toteutuksesta promise-pohjaiseksi, kuten muussakin koodissa oltiin toimittu.

Siirtyminen pois erillisestä tiedonhakupalvelimesta sulki myös pois mahdollisuuden sovelluksen tilan tallentamiseen ajojen välillä. Yksinkertaisimmillaan tämä tarkoittaa sitä, ettei esimerkiksi keskeneräisten diplomien tallennusta käyttäjän selaimen ulkopuolelle voitu enää toteuttaa. Tiedonhakupalvelimesta olisi kuitenkin ollut myös se etu, että harvoin muuttuva tieto, kuten käyttäjä- ja kanavalista, oltaisiin voitu tallentaa välimuistiin, joka oltaisiin päivitetty vain silloin kun Slackista olisi näihin listoihin uutta sisältöä saatavilla. Tämä olisi nopeuttanut toistuvia hakuja huomattavasti, sillä tällä hetkellä jokainen uusi haku hakee uudestaan kaikki Vincitin Slack-tiimin käyttäjät, kanavat ja omat emojit; tietoja, jotka muuttuvat vain harvoin hakujen välillä.

Koska mahdollisuutta ulkopuolisen välimuistin käytölle ei enää ollut, jouduttiin tiedonhaun toteuttamisessa tekemään useita hakua nopeuttavia ratkaisuja. Kehityksen alkuvaiheessa oltiin toimittu siten, että kunkin käyttäjän tiedot oltiin haettu Slackin rajapinnasta vasta silloin kun niitä tarvittiin. Käytännössä kun viestiä käsiteltäessä vastaan tuli käyttäjätunniste, haettiin sen tiedot rajapinnasta. Tämä teki hausta paitsi hidasta niin myös epätasaista, sillä haetussa viestijoukossa olleiden käyttäjien määrä vaihteli. Toiset haut saattoivat kestää muutamia sekunteja toisten mennessä yli kymmenen sekunnin. Niinpä päädyttiin ratkaisuun, jossa kaikki Vincitin Slack-tiimin käyttäjät ja kanavat haettiin muistiin ennen `#eihuano`-kanavan viestihistorian läpikäymistä. Näin haun alkuun tuli pieni viive, mutta vastaavasti viestien käsittely nopeutui ja viestijoukkojen käsittelyaikojen väliset erot tasaantuivat. Ainoastaan tällä toimenpiteellä kuukauden viestihistorian haku- ja käsittelyaika putosi keskimäärin neljästätoista sekunnista 3-4 sekuntiin.

Mainittakoon, että syy haun hitauteen ei ollut tiedonhakulogiikan siirto osaksi selaimessa suoritettavaa koodia. Käsittelynopeudessa erillisen palvelimen ja selaimen välillä ei havaittu merkittäviä eroja. Erillistä palvelinta käytettäessä oltaisiin kuitenkin voitu päätyä käyttämään toisenlaisia optimointiratkaisuja. Esimerkiksi välimuistin sisältöä ei olisi tarvinnut hakea ennen jokaista hakua, vaan se oltaisiin voitu muodostaa uudelleen säännöllisin väliajoin hakutapahtumasta erillään.

5.2 Useissa eri muodoissa ja sijainneissa ollut data

Ei huano -generaattoria testattaessa huomattiin, ettei sen näyttämään diplomilistaukseen tullut kaikkia niitä viestejä, joiden reaktiomäärä olisi oikeuttanut ei huano -diplomiin. Tämä johtui siitä, että suuri osa `#eihuano`-kanavalle olleista viesteistä ei tullut sinne suoraan kenenkään ihmisen lähettämänä, vaan erillinen Slack-botti tarkkaili kaikkia Vincitin Slack-kanavia, ja kopioi kaikki löytämänsä `#eihuano`-, `eihuano`- tai `:ela:`-sanalla alkaneet viestit `#eihuano`-kanavalle. Tämä viesti oli kuitenkin täysin erillinen kopio, ei siis viite alkuperäiseen, joten viestistä oli olemassa kaksi erillistä versiota, joissa oli myös itsenäiset reaktiot. Lisäksi kopioviestissä olleet lähettäjätiedot olivat epätäydelliset. Liitteessä 2 on esimerkki tyypillisestä botin luomasta viestistä, jonka tiedot ovat epätäydelliset liitteen 1 esimerkkiin verrattuna.

Tiedonhakulogiikan toimintaa tuli tarve muuttaa siten, että kopioviestin löytäessään se etsii alkuperäisen viestin ja laskee kummankin viestin saamat reaktiot yhteen. Alkuperäinen kanava oli mahdollista löytää helposti, sillä lähes kaikissa kopioviesteissä luki kanava, jolta viesti oli kopioitu. Jos viesti oli esimerkiksi alun perin #tampere-kanavalta, luki alkuperäisen viestin perässä [#tampere]. Ongelmallista oli ennen kaikkea se, että toisinaan alkuperäistä viestiä tai koko kanavaa ei olut enää olemassa, eikä alkuperäisen kanavan nimikään ollut lisätty viesteihin aina samalla tavalla. Botin kehityksen alkuvaiheessa merkintätapa näkyy vaihdelleen <#kanava> ja (#kanava) -muotojen välillä, ja aivanvarhaisimmista viesteistä kanava puuttuu kokonaan. Kyseisten viestien ei huano -diplomit oltiin luonnollisesti tehty jo vuosia sitten, mutta nämä erilaiset merkintätavat haluttiin myös ottaa huomioon, jotta sovellus osaisi tarvittaessa käsitellä ne oikein.

5.3 Singleton-moduuleista itsenäisiin riippuvuuksiin

EcmaScript on tukenut moduuleja virallisesti kielen ES2015-versiosta lähtien. Moduuli on tavallinen EcmaScript-tiedosto, joka voi sisältää mitä tahansa koodia, mutta johon on erikseen merkitty ne viittaukset (muuttujat, funktiot, luokat...), joista halutaan tehdä julkisia. Näitä julkisia viittauksia on mahdollista käyttää muusta koodista käsin. Eräs EcmaScript-moduulien erikoispiirre on se, että ne ovat ns. singletoneja. Tämä tarkoittaa sitä, että moduulien sisältämä koodi suoritetaan ja ladataan muistiin vain kerran. Näin kaikki tiettyä moduulia käsittelevät koodin osat käyttävät samaa kopiota. (Rauschmayer: Modules)

Tiedonhakulogiikan tehokkaan toiminnan kannalta on olennaista, että sen luokista käytetään vain yhtä versiota kerrallaan. Esimerkiksi EntityStore-luokassa olevia käyttäjä- ja kanavalistoja tarvitaan MessageDecoder- ja BotMessageDecoder-luokissa. Suorituskyvyn optimoinnin kannalta on viisainta pitää muistissa vain yksi EntityStore-objekti, jota kumpikin näistä luokista käyttää, sillä tällöin luokan sisältämät tiedot tarvitsee hakea Slackin rajapinnasta vain kerran.

Tästä syystä tiedonhakulogiikkaa suunniteltaessa päädyttiin ratkaisuun, jossa kustakin luokasta olisi muistissa vain yksi kopio, jolloin kunkin luokan tila olisi automaattisesti kaikkien muiden luokkien käytettävissä. Tämän mallin toteuttamisessa käytettiin

hyväksi edellä esiteltyä moduulien singleton-piirrettä. Kukin luokista oli omassa moduulissaan. Sen sijaan että luokkamäärittelystä oltaisiin tehty julkinen, luotiin luokasta uusi objekti, ja tehtiin tästä objektiviitteestä julkinen. Näin saatiin aikaan tilanne, jossa moduulin käyttöön ottanut koodi sai käyttöönsä valmiin objektin, jota voitiin käyttää sellaisenaan.

Töiden edetessä alettiin tiedonhakulogiikan luokille kirjoittaa myös yksikkötestejä. Silloin havaittiin ongelma. Koska testien lopputulosten tulee olla ennustettavia, ei niissä voi hakea verkosta tietoa, jonka muuttumattomuutta ei voida taata. Käytännössä Slack-rajapintakirjastoa tuli muokata testejä varten siten, että se palauttaisi pyydettäessä ennalta määriteltyä testidataa. Nyt moduulien singletoniuteen perustuva suunnittelumalli oli kääntynyt itseään vastaan, sillä jokaisessa luokassa oli kiinteä viittaus aitoon, rajapinnan kanssa kommunikoivaan Slack-kirjastoon, jota ei ollut mahdollista muuttaa sovelluksen ajon aikana.

Tästä syystä singletoneihin perustuvasta suunnittelumallista päätettiin luopua. Jokaista luokkaa muutettiin siten, että ne saisivat rakentajissaan tarvitsemansa riippuvuudet muihin luokkiin. Esimerkiksi MessageDecoder tarvitsee toimiakseen EntityStore- ja EmojiManager-luokkien objekteja, joten ne toimitetaan sille luokan rakentajassa. Nyt testeissä saatettiin toimia siten, että kaikki Slackin rajapintakirjastoa käyttävät luokat saivat riippuvuutenaan testidataa lähettävän kirjaston.

6 POHDINTA

Ei huano -generaattori valmistui kesäkuussa 2016, ja se otettiin Vincitin henkilöstötiimissä käyttöön saman tien. Sovelluksen suunnittelussa ei tänä aikana olla havaittu merkittäviä puutteita. Yksittäisiä virhetilanteita on ollut, mutta ne on saatu korjattua. Tilaaja sai haluamansa kaltaisen sovelluksen ja vielä toimivana, joten tehdyn työn voi siltä osin katsoa olleen onnistunut.

Tiedonhakulogiikan toteutukseen jäi muutamia seikkoja, jotka olisi voinut toteuttaa huolellisemmin ja ennen kaikkea vikasietoisemmin. Monet käsittelyluokista ovat liian naiiveja oman tilansa suhteen. Jos esimerkiksi EmojiManagerilta haetaan tiettyä emojia vastaava Unicode-merkki tai URL, ei hakumetodi tarkista onko luokan omassa emoji-listassa ylipäänsä mitään, vaan saattaa ilmoittaa ettei emojia löydy, jos listaa ei olla ymmärretty ensin päivittää. Kaikista ulospäin näkyvistä luokkien metodeista olisivoinut tehdä asynkronisia, jolloin ne voisivat hakea verkosta tarvitsemansa datan mikäli sitä ei oltu jo haettu. Päivitys voitaisiin edelleen tehdä selkeyden vuoksi ennen yhtäkään hakua, mutta ohjelma ei toimisi odottamattomalla tavalla, jos dataa ei oltaisikaan haettu. Toinen vaihtoehto olisi ollut muokata kaikkia päivittämistä vaativia tietoja hakevia metodeja palauttamaan jonkinlainen virhe, jos päivitystä ei oltu tehty.

Viestien hakua olisi myös voinut nopeuttaa entisestään. W3C on määritellyt erillisen Web worker -rajapinnan, joka mahdollistaa koodin suorittamisen verkkosivulla toisessa säikeessä käyttöliittymän toiminnasta erillään (Hickson: Web workers). Näin tiedonhaun päivämääräriippumaton data, toisin sanoen Slackin kanava-, käyttäjä- ja emoji-listat, oltaisiin voitu hakea valmiiksi jo ennen kuin haku on ehtinyt alkaa. Tällä keinolla hakua oltaisiin voitu nopeuttaa entisestään käyttämällä se aika, joka käyttäjältä kuluu haun aloittamiseen, näiden tietojen esilataamiseen.

Tätä projektia aloittaessani olin työskennellyt Javascriptin parissa ennenkin, mutta olin käyttänyt toisenlaisia käännöstyökaluja ja kielen vanhempaa ES5-versiota. Projektissa tulivat siis tutuiksi niin EcmaScriptin ES2015-versio ja siihen liittyvät työkalut kuin myös Slackin rajapinta, sekä erityisesti asynkroninen ohjelmointi promiseja käyttäen. Yleisemmällä tasolla opin myös paljon siitä, minkälaiset tekijät vaikuttavat ohjelman nopeaan toimintaan, sekä millaisin ratkaisuin tiedon hakemista ja koostamista on mahdollista abstraktoida.

Lähteitä valitessani pyrin tukeutumaan riippumattomiin tietolähteisiin, jotka olisivat mahdollisimman lähellä alkuperäistä aihetta. Siksi lähteisiin on valikoitunut paljon käyttöohjeita ja kehittäjädokumentaatiota suoraan niiltä tahoilta, jotka ovat olleet mukana kehittämässä käytettyjä tekniikoita ja standardeja. Pidin lähteissä olennaisena myös niiden realistista käyttöä. Toisin sanoen kaikki lähteet ovat sellaisia, joita käyttäisin missä tahansa työssä, en ainoastaan raporttia laadittaessa. Tätä työtä tehdessäni lähes kaikki siinä käydyt tekniikat olivat hyvin tuoreita, eikä niistä ollut olemassa juurikaan kirjallisuutta. Lähes kaikki relevantiksi kokemani materiaali oli saatavilla ainoastaan verkosta, osa myös samat verkkotekstit sisältävinä kirjoina.

Vaikka edellä kuvailemani tiedonhakulogiikka onkin itse tekemäni, ei sen nykymuotoinen toteutus ole yksin omaa ansiotani. Jokainen tekemäni muutos ja uusi ominaisuus kävi läpi huolellisen koodikatselmoinnin, jossa sain korvaamatonta palautetta kirjoittamastani koodista. Näiden katselmointien yhteydessä syntyivät ratkaisut sovelluksen keskeisimpiin ongelmiin.

Koska kehitetty sovellus on tarkoitettu tilaajan sisäiseen käyttöön ja siinä käsitellään yksityisen Slack-tiimin luottamuksellisia viestejä, on työn eettisiin lähtökohtiin kiinnitettävä erityistä huomiota. Olen pyrkinyt takaamaan tilaajan yksityisyyden säilymisen tässä raportissa poistamalla esimerkeistä kaikki Slack-tiimin yksilöivät viitteet, kuten käyttäjätunnukset ja viestien tekstit. Data on näiltä osin korvattu mahdollisimman autenttisella keksityllä sisällöllä.

Tämän raportin tarkoitus on olla mahdollisimman tarkoin rakennettu dokumentti yhdestä mahdollisesta tavasta rakentaa tiedon hakemiseen ja käsittelyyn keskittyvä selainpohjainen sovellus. Tarkoituksenani ei ole ollut esittää aiheeseen useita vaihtoehtoisia lähestymistapoja tai tehdä muutakaan yleisluontoista tutkimusta, vaan kertoa siitä, minkälaisiin ratkaisuihin juuri tässä projektissa päädyttiin, ja mistä syistä ne valikoituivat käyttöön. Siksi tämä raportti soveltuu tutkittavaksi ennen kaikkea tilanteessa, jossa ollaan suunnittelemassa jonkin vastaavankaltaisen sovelluksen arkkitehtuuria ja muita teknisiä ratkaisuja. Jos sovellus tulee hakemaan ja käsittelemään paljon verkosta saatavaa dataa suoraan selaimessa ilman että tietoa on mahdollista säilöä etukäteen välimuistiin, voi tämän raportin avulla ainakin yrittää kiertää vastaavassa tehtävässä itse kohtaamani sudenkuopat.

LÄHTEET

Async. Projektin esittely. Luettu 15.04.2017. https://github.com/caolan/async

Babel. Projektin esittely. Luettu 10.04.2017. https://babeljs.io/

Butler, B., Network World. 2013. PaaS Primer: What is platform as a service and why does it matter? Julkaistu 11.02.2013. Luettu 10.04.2017. http://www.networkworld.com/article/2163430/cloud-computing/paas-primer--what-is-platform-as-a-service-and-why-does-it-matter-.html

Dokku. Projektin esittely. Luettu 15.03.2017. https://github.com/dokku/dokku

Heroku. Creating a Buildpack. Luettu 15.03.2017. https://devcenter.heroku.com/articles/buildpacks#creating-a-buildpack

Heroku. Tuotteen kotisivu. Luettu 15.03.2017. https://www.heroku.com/

Hickson, I. (toim.), The World Wide Web Consortium. 2015. Web Workers. Working Draft. Julkaistu 24.09.2015. Luettu 29.03.2017. http://www.w3.org/TR/2015/WD-workers-20150924/

Kangax. ECMAScript 6 compatibility table. Luettu 12.04.2017. http://kangax.github.io/compat-table/es6/

Mean.io. Projektin esittely. Luettu 10.04.2017. http://mean.io/

Mozilla Developer Network. Concurrency Model and Event Loop. Luettu 02.04.2017. https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop

Mozilla Developer Network. Paged media. Luettu 06.04.2017. https://developer.mozilla.org/en-US/docs/Web/CSS/Paged_Media

Mozilla Developer Network. Promise. Luettu 12.4.2017. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

Node.js. HTTP. Luettu 18.03.2017. https://nodejs.org/api/http.html

Node.js-projektin kotisivu. Luettu 18.03.2017. https://nodejs.org/en/

Npm. Most starred packages. Luettu 10.04.2017. https://www.npmjs.com/browse/star

Rauschmayer, A., Exploring ES6. About ECMAScript 6 (ES6). Luettu 23.03.2017. http://exploringjs.com/es6/ch_about-es6.html

Rauschmayer, A., Exploring ES6. Core ES6 features. Luettu 23.03.2017. http://exploringjs.com/es6/ch_core-features.html

Rauschmayer, A., Exploring ES6. Modules. Luettu 04.04.2017. http://exploringjs.com/es6/ch_modules.html#sec_overview-modules

Rauschmayer, A., Speaking JavaScript. How JavaScript was created. Luettu 18.05.2017. http://speakingjs.com/es5/ch04.html

Rauschmayer, A., Speaking JavaScript. Standardization: EcmaScript. Luettu 18.05.2017. http://speakingjs.com/es5/ch05.html

Slack. Change your username. Luettu 18.03.2017. https://get.slack.help/hc/en-us/articles/216360827-Change-your-username

Slack. Creating custom emoji. Luettu 18.03.2017. https://get.slack.help/hc/en-us/articles/206870177-Create-custom-emoji

Slack. Emoji reactions. Luettu 18.03.2017. https://get.slack.help/hc/en-us/articles/206870317-Emoji-reactions

Slack. Renaming a channel. Luettu 18.03.2017. https://get.slack.help/hc/en-us/articles/201654073-Rename-a-channel

Slack. Slack API. Luettu 18.03.2017. https://api.slack.com/

Slack. Team Communication for the 21st Century. Luettu 18.03.2017. https://slack.com/is

Takada, M., Single page apps in depth. Modern web applications: an overview. Luettu 06.04.2017. http://singlepageappbook.com/goal.html

LIITTEET

Liite 1. Esimerkki Slackin rajapinnan palauttamasta kanavan viestistä, jonka lähettäjä on Slack-käyttäjä


```
{
    "type": "message",
    "user": "U13Q54RX9",
    "text": "Viestin sis\u00e4lt\u00f6",
    "ts": "1377064800.13903",
    "reactions": [
        {
            "name": "thumbsup",
            "users": [
                "U0H20D0CK",
                "UG1FU9W5Z"
            ],
            "count": 2
        }
    ]
}
```

Liite 2. Esimerkki Slackin rajapinnan palauttamasta kanavan viestistä, jonka lähettäjä on Slack-botti


```
{
    "text": "Kopioi minut [#alkuperainen_kanava]",
    "username": "olli.opiskelija",
    "bot_id": "B824EOW0",
    "icons": {
        "image_48": "http:\/\/example.com\/bot.png"
    },
    "type": "message",
    "subtype": "bot_message",
    "ts": "1490868081.716184",
    "reactions": [
        {
            "name": "thumbsup",
            "users": [
                "U039DR3SS",
                "U1952F0RD",
                "U0148U2DU",
            ],
            "count": 3
        }
    ]
}
```

Liite 3. Esimerkki Ei huano -generaattorin tiedonhakukerroksen tuottamasta viestidatasta


```
{
    "messages": [
        {
            "text": "Ei huano tuotannon fiksaus keskellä yötä Olli Opiskelija",
            "reactions": [
                {
                    "name": "eihuano",
                    "count": 6,
                    "custom": true
                },
                {
                    "name": "medal",
                    "count": 4,
                    "custom": false,
                    "symbol": "??"
                }
            ],
            "mentioned_users": [
                {
                    "username": "olli.opiskelija",
                    "real_name": "Olli Opiskelija",
                    "image": "https://example.com/image_url.png"
                }
            ],
            "timestamp": "1496157600"
        }
    ],
    "customEmoji": {
        "eihuano": "https://example.com/emoji_url.png"
    }
```