Haaga-Helia
ammattikorkeakoulu Oy

Luonnollista kieltä (NLP) käyttävä chatbot-sovellus

Sarianna Silvonen

Opinnäytetyö
Tietojenkäsittelyn koulutusohjelma
2019

Haaga-Helia
ammattikorkeakoulu Oy

Tiivistelmä

14.5.2019

Tekijä(t) Silvonen Sarianna	
Koulutusohjelma Tietojenkäsittelyn koulutusohjelma	
Opinnäytetyön otsikko Luonnollista kieltä (NLP) käyttävä chatbot-sovellus	**Sivu- ja liitesivumäärä** 43 + 23
Opinnäytetyön otsikko englanniksi A chatbot application using Natural Language Processing	

Chatbotit ovat tietokoneohjelmia, jotka keskustelevat ihmiskäyttäjien kanssa tekstin tai puheen välityksellä ja suorittavat käyttäjien pyytämiä tehtäviä ja toimintoja. Tekstipohjaisia chatbotteja käytetään nykyään yleisesti yritysten asiakaspalvelutehtävissä, kun taas puhekäyttöiset ohjelmat ovat Sirin ja Alexan kaltaisten suosittujen digitaalisten assistenttien taustalla.

Tein opinnäytetyöni keväällä 2019. Kyseessä oli toiminnallinen opinnäytetyö, jonka tavoitteena oli rakentaa tekstipohjainen, luonnollista kieltä käyttävä chatbotti, joka poimii käyttäjän vapaasti kirjoittamasta syötteestä tarvitsemansa hakuparametrit, hakee niiden avulla tietoa ulkoisesta lähteestä ja palauttaa vastauksen käyttäjälle.

Luonnollisen kielen käsittely eli NLP (Natural Language Processing) on tekoälyn osa-alue, jonka avulla tietokoneohjelmat tulkitsevat ja käyttävät ihmiskieltä. NLP-tekniikoita voidaan soveltaa monelle eri tieteenalalle ja liiketoiminnan alueelle, ja ne ovat keskeisiä chatbottien toiminnassa.

Rakensin chatbotin Googlen omistamalle Dialogflow-alustalle, joka hyödyntää tekoälyä ja koneoppimista chatbottien toiminnassa. Botti hakee käyttäjän pyynnöstä juna-aikatauluja rautatieliikenteen avoimesta rajapinnasta, jota ylläpitää Traffic Management Finland. Dialogflow'n ja rajapinnan yhdistäminen tapahtuu palvelimettomassa Google Cloud Functions -ympäristössä, jonne koodasin Node.js:n avulla funktion hakemaan dataa rajapinnasta ja muotoilemaan sitä. Työssäni käsittelin siis pääasiallisesti näitä kolmea eri komponenttia, jotka olivat kaikki minulle uusia, sekä niiden välisiä yhteyksiä. Lisäksi integroin chatbotin Slackin ja verkkodemoympäristön kautta muidenkin käyttäjien tavoitettavaksi.

Työn lopputuloksena sain aikaan toimivan chatbotin, joka hakee rajapinnan kautta juna-aikatauluja ja palauttaa niitä käyttäjälle. Lisäksi dokumentoin ohjeet, joiden avulla voi yhdistää Dialogflow'n ja Google Cloud Functionsin chatbotin kehitysympäristöksi. Työn tuloksena opin paljon uutta asiaa käyttämistäni ohjelmista, tekniikoista ja työkaluista sekä hahmotin, mitä kaikkea täytyy ottaa huomioon hyvän chatbotin rakentamisessa.

Asiasanat

Ohjelmistokehitys, NLP, chatbot, pilvipalvelut, Dialogflow, Google Cloud Functions, Node.js

Sisällys

1 Johdanto 1
2 Luonnollisen kielen käsittely tietokoneella 3
2.1 Mikä on luonnollinen kieli? 3
2.2 Miten luonnollista kieltä käsitellään tietokoneessa? 4
2.1 Miten luonnollisen kielen käsittelyä hyödynnetään? 5
3 Chatbotit 8
3.1 Chatbottien historiaa 8
3.1 Chatbottien luokittelua 10
4 Kehitysympäristön rakentaminen 12
4.1 Dialogflow 12
4.2 Google Cloud Functions 17
4.3 Rautatieliikenteen avoin rajapinta 18
4.4 Yhdistetty kehitysympäristö 19
5 Chatbotin kehittäminen 23
5.1 Webhookin rakenne 23
5.2 Dialogflow-chatbotin rakenne 24
5.3 Juna-aikataulutietojen hakeminen 29
5.4 Chatbotin kouluttaminen ja analytiikka 31
6 Pohdinta 34
Lähteet 37
Liitteet 44
Liite 1. Kuvakaappaukset: Google Cloud Functions 44
Liite 2. Kuvakaappaukset: Dialogflow 47
Liite 3. Lähdekoodi: index.js 54
Liite 4: Lähdekoodi: package.json 64
Liite 5: Rajapinnan palauttama JSON-data 65
Liite 6: Vuokaavio chatbotin toiminnasta 66

1 Johdanto

Chatbotit ovat tekstin tai puheen välityksellä ihmiskäyttäjien kanssa keskustelevia tietokoneohjelmia, jotka on yleensä suunniteltu suorittamaan joku tietty tehtävä tai toiminto. Nykyään tekstipohjaisia chatbotteja käytetään enenevissä määrin yritysten nettisivuilla avustamassa sivuston käyttäjiä ja toimimassa perustason asiakaspalvelijoina ympäri vuorokauden, kun taas monipuoliset henkilökohtaiset digitaaliset apulaiset toimivat pääasiassa puheohjauksella.

Opinnäytetyöni käsittelee luonnollista kieltä käyttävää chatbot-sovellusta, jonka olen rakentanut käyttäen Googlen Dialogflow-alustaa ja Google Cloud Functions -palvelua. Luonnollisen kielen käyttö tarkoittaa sitä, että chatbotti ei tarjoa käyttäjälle valmiita vastausvaihtoehtoja, vaan ottaa vastaan käyttäjän vapaasti kirjoittaman syötteen ja poimii siitä tarvitsemansa hakuparametrit tekoälyä ja koneoppimista hyödyntämällä.

Tekemäni demosovellus hakee käyttäjän pyytämiä juna-aikatauluja Traffic Management Finlandin ylläpitämän rautatieliikenteen avoimen rajapinnan kautta (Traffic Management Finland 2019a). Chatbotin toimintalogiikka on toteutettu Dialogflow'ssa, ja Google Cloud Functions puolestaan ajaa koodin, joka hakee käyttäjän pyytämät tiedot rajapinnasta. Käyttäjä voi siis kysyä botilta (englanniksi) juna-aikatauluja esimerkiksi Helsingistä Tampereelle tiettynä päivänä, ja botti palauttaa avoimen rajapinnan kautta haetut aikataulutiedot.

Opinnäytetyön lähtökohta oli toimeksiantajayrityksen tarve saada demosovellus integroitavaksi yrityksen kehittämään RCS-ympäristöön (Rich Communications Services). Minusta riippumattomista syistä RCS-ympäristö ei kuitenkaan tullut käyttövalmiiksi sellaisessa aikataulussa, että olisin ehtinyt ottaa sen mukaan opinnäytetyöhön. Jouduin siis rajaamaan RCS:n pois työn aiheesta ja keskittymään pelkästään Dialogflow'n, Google Cloud Functionsin ja rautatieliikenteen avoimen rajapinnan yhteistyöhön, jossa siinäkin on paljon pohdittavaa ja yhteen sovitettavaa.

Opinnäytetyön toiminnallisena tavoitteena oli siis saada aikaan toimiva demochatbotti, jota voi jatkojalostettuna käyttää RCS-alustan toiminnallisuuden esittelemiseen. Varsinainen haettava data ei tässä tarkoituksessa ole oleellinen, vaan rautatieliikenteen avoimen rajapinnan sijaan olisin voinut yhtä hyvin käyttää jotain toista avointa ulkoista rajapintaa ja hakea jotain muuta tietoa kuin juna-aikatauluja. Toivonkin, että jatkossa tätä chatbotin ja webhookin pohjaa voi soveltaa myös uusiin botteihin, jotka hakevat tietoa toisista ulkoisista rajapinnoista.

Tietojenkäsittelyn monimuoto-opinnoissani olen suuntautunut pääasiassa ohjelmistosuunnitteluun. Kun aloin tehdä opinnäytetyötä, lähtötasoni oli sellainen, että olin suorittanut kaikki monimuotopuolella tarjolla olleet ohjelmointikurssit. Näillä kursseilla käytettiin ohjelmointikielenä lähinnä Javaa, mutta niillä hankittu yleinen ymmärrys ohjelmoinnista oli hyödyksi opinnäytetyötä suorittaessani. Osasin myös jonkin verran perustason JavaScriptiä ja JSONia sekä HTML- ja CSS-kieliä, joita tässä työssä ei kuitenkaan juurikaan tarvittu.

Oma oppimistavoitteeni oli perehtyä tarkemmin luonnollisen kielen käsittelyyn ja chatbotteihin yleisellä tasolla sekä erityisesti Dialogflow'n käyttöön chatbottien kehittämisessä. Halusin myös opetella ohjelmoimaan paremmin hyödyntämällä Node.js-ympäristöä ja JavaScriptin uudempia ominaisuuksia, ja chatbotin ulkoisen webhookin koodaaminen tarjosikin sopivasti ohjelmointihaastetta. Google Cloud Functions oli myös minulle uusi tuttavuus, jonka käyttö vaati opettelua, mutta toivottavasti se on myös jatkossa käyttökelpoinen työkalu. Avoimen rajapinnan käyttö antoi mahdollisuuden perehtyä tarkemmin JSONin kautta palautettavan datan rakenteeseen, joka tässä tapauksessa oli varsin monitasoista, ja sen käsittelyyn koodissa.

2 Luonnollisen kielen käsittely tietokoneella

Luonnollisen kielen käsittely eli NLP[1] (Natural Language Processing) on tekoälyn osa-alue, jota hyödyntämällä tietokoneohjelmat oppivat ymmärtämään, tulkitsemaan ja käyttämään ihmiskieltä. NLP pohjautuu useaan eri tieteenalaan, kuten tietojenkäsittelyyn ja tietokonelingvistiikkaan. (SAS 2019.) Luonnollisen kielen käsittely tietokoneella kuuluu koneoppimisen alaan, jonka perinteisen määritelmän mukaan tietokoneohjelman voidaan katsoa oppivan, jos se suoriutuu sille annetusta tehtävästä kerta toisensa jälkeen paremmin siten, että edistymistä voidaan mitata numeerisella tavalla (Brink et al. 2017: 5).

2.1 Mikä on luonnollinen kieli?

”Luonnollinen kieli” on kielitieteen tutkimuksessa monimerkityksinen termi, jota on vaikea määritellä täsmällisesti (mm. Lyons 1991: 53-57). Tieteen termipankin (2019a) lyhyen määritelmän mukaan luonnollinen kieli on ”jonkin ihmisryhmän äidinkielenään käyttämä kieli, joka on luonnollisen kehityksen tulos”. Termipankin pidempi määritelmä selventää, että luonnollisella tarkoitetaan kolmea asiaa. Ensinnäkin luonnollisena pidetään kieltä, joka on syntynyt ja kehittynyt tietyssä ihmisyhteisössä hyvin pitkän ajan kuluessa muotoutuen soveliaaksi juuri puhujayhteisönsä käyttötarkoituksiin. Toiseksi luonnollinen kieli on sellainen, jonka lapsi omaksuu äidinkielekseen automaattisesti kasvaessaan kieltä puhuvien ihmisten joukossa. Kolmanneksi tätä luonnollisesti omaksuttua äidinkieltä käytetään ihmisten arjessa yhteydenpitoon ja hahmottamaan ympäristöä verbaalisesti.

Luonnollinen kieli käsittää joukon sanoja merkityksineen sekä kieliopillisia keinoja, joilla ilmaistaan esimerkiksi suhteita, aikamuotoja ja kysymyksiä. Ihmisille luonnollinen kieli on menetelmä kuvata maailmaa ja sen tapahtumia tiivistetyssä muodossa. Kielen ymmärtäminen edellyttää tietoa kielen kuvaamasta maailmasta. Tämä muodostaakin merkittävän haasteen tietokoneille, jotka eivät oikeasti voi ymmärtää maailmasta mitään, vaikka voivatkin koneoppimisen menetelmien avulla harjaantua käsittelemään luonnollisella kielellä annettuja syötteitä ja tuottamaan ymmärrettäviä ja järkeviltä vaikuttavia vastauksia. (Haikonen 2017: 47-51.)

Tässä opinnäytetyössä käytän termiä ”luonnollinen kieli” tarkoittamaan ihmiskieliä siinä mielessä kuin ne Tieteen termipankin määritelmän mukaan ymmärretään. Käytännössä

---

[1] Tätä ei pidä sekoittaa toiseen samaa lyhennettä käyttävään käsitteeseen Neuro-Linguistic Programming eli neurolingvistinen ohjelmointi, joka nimestään huolimatta on psykologian eikä tietojenkäsittelyn osa-alue ja koskee aivan eri asiaa.

työssäni käsitelty luonnollinen kieli on englanti, jota rakentamani chatbot-sovellus käyttää (kielen valinnasta tarkemmin kohdassa 4.1).

2.2 Miten luonnollista kieltä käsitellään tietokoneessa?

NLP kattaa useita erilaisia tekniikoita, joiden avulla ihmiskieltä tulkitaan tietokoneessa. Luonnollisella kielellä tuotettu data on hyvin monimuotoista, ja sitä halutaan käsitellä käytettäväksi monissa eri käytännön sovellutuksissa, joten sitä on lähestyttävä useasta eri kulmasta. Työssä hyödynnetään niin tilastotieteen ja koneoppimisen menetelmiä kuin sääntöihin ja algoritmeihin perustuvia lähestymistapoja. (SAS 2019.)

Perustoimenpiteitä ovat esimerkiksi saneistus (tokenization) ja lauseenjäsennys (parsing), lemmatisointi (lemmatization) tai typistäminen (stemming), sanaluokkaleimaus (part-of-speech tagging), kielen tunnistus ja semanttisten suhteiden tunnistus (SAS 2019).

Saneistus tarkoittaa juoksevan tekstin jakamista saneiksi eli yksiköiksi, joita voidaan käyttää jatkoanalyysissä. Saneistuksessa poistetaan välimerkit tai ne eristetään omiksi yksiköikseen. (Tieteen termipankki 2019b.) Lauseenjäsennys puolestaan on virkkeiden tai sananmuotojen tunnistusta niiden rakenteen perusteella (Tieteen termipankki 2019c). Lemmatisointi tai typistäminen tarkoittaa sanojen palauttamista taivutetuista muodoista perusmuotoon (Eranti & Ylä-Anttila 2017: 28). Sanaluokkaleimauksesa saneelle annetaan leima tai koodi, joka ilmaisee saneen sanaluokkaa (Tieteen termipankki 2019d).

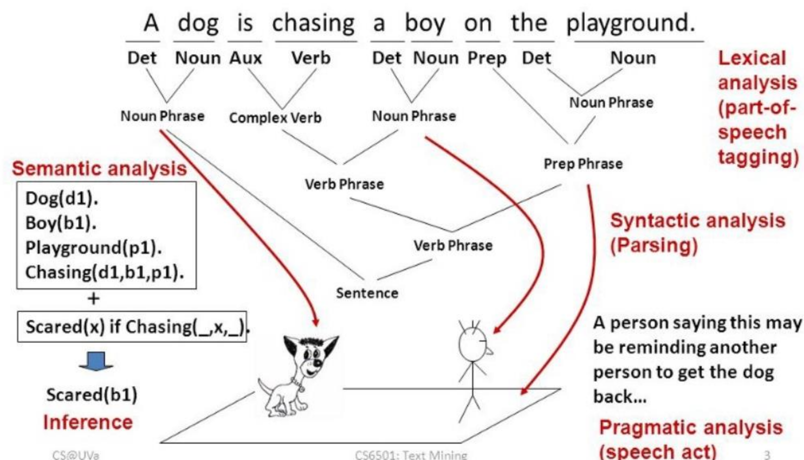
Kielen tunnistuksessa kone tunnistaa, millä luonnollisella kielellä käsiteltävänä oleva teksti on kirjoitettu – tämä on tietenkin ensimmäinen ehto, jonka on täytyttävä, jotta datan mielekäs jatkokäsittely on mahdollista. Semanttisten suhteiden tunnistus puolestaan viittaa siihen, miten käsiteltävän tekstin sanat suhtautuvat merkitykseltään toisiinsa. Semanttisten suhteiden rajaus ja määrittely on itse kielitieteen sisälläkin monimutkainen kysymys (mm. Murphy 2003: 8-12) ja entistä haastavampi, kun ihmisten sijaan tekstien tulkitsijoina ovatkin tietokoneohjelmat (mm. Auger ja Barrière 2010: 2-8).

Tiivistettynä siis NLP-työkalut pilkkovat kielen lyhyiksi palasiksi, yrittävät ymmärtää palasten välisiä suhteita ja tulkita sitä, miten palaset yhdessä luovat merkityksen. Nämä perustehtävät ovat aina lähtökohtana, kun kehittyneemmät NLP-ohjelmat esimerkiksi luokittelevat tekstin sisältöä, etsivät sen pääkohdat, hakevat rakenteista tietoa tai analysoivat suurista tekstimassoista välittyvää tunnetilaa (sentiment analysis). NLP:n avulla voidaan myös kääntää puhetta tekstiksi ja tekstiä puheeksi, laatia tiivistelmiä pitkistä teksteistä sekä tuottaa teksti- tai puhemuotoisia konekäännöksiä kahden eri kielen välillä. (SAS 2019.)

Kuva 1 esittää pääpiirteissään luonnollisen kielen käsittelyn vaiheet. Alkutilanteessa on käyttäjän antama (teksti)syöte, jolle ohjelma suorittaa eritasoisia kielitieteellisiä analyysejä ja tuottaa lopputuloksena olettamansa käyttäjän tarkoittaman merkityksen. Mitä paremmin NLP onnistuu tässä, sitä lähempänä lopputulos on ihmiskäyttäjän oikeasti tarkoittamaa merkitystä.

Kuva 1. Luonnollisen kielen käsittelyn vaiheet (kuvan mallina Dale 2000: 3).

Kuva 2 on havainnollistava esimerkki erään lauseen käsittelystä näiden analyysimenetelmien avulla. Lause jaetaan ensin saneiksi, joille määritellään sanaluokat, ja sille suoritetaan syntaktinen analyysi (lauseenjäsennys) ja semanttinen analyysi (merkityksen päätteleminen). Pragmaattinen analyysi eli sen päätteleminen, mitä puhuja haluaa repliikillään saada aikaan, vaatinee tässä tapauksessa niin edistyneen tason tietoa ulkomaailmasta, että tietokoneohjelman voi olla vaikea saavuttaa sellaista.



Kuva 2. Esimerkki luonnollisen kielen käsittelystä (Chandrayan 2017; alkuperäinen lähde Wang 2016).

2.1 Miten luonnollisen kielen käsittelyä hyödynnetään?

NLP:n avulla tietokoneet pystyvät viestimään ihmisten kanssa ihmisten ymmärtämällä kielellä ja suorittamaan kielen ymmärrystä vaativia työtehtäviä. Ne voivat esimerkiksi lukea tekstiä tai kuunnella puhetta, tulkita lukemaansa ja kuulemaansa ja poimia siitä tärkeiksi määritellyt kohdat. Koneet pystyvät analysoimaan kielipohjaista datamassaa paljon

nopeammin ja tasalaatuisemmin kuin ihmiset. Nykypäivän yhteiskunta tuottaa jatkuvasti valtavia määriä strukturoimatonta (teksti)dataa, kuten sosiaalisen median päivityksiä, jonka tehokas analysointi on mahdotonta ilman tietokoneiden ja NLP:n käyttöä. (SAS 2019.)

Parikymmentä vuotta sitten markkinoille tullut Microsoft Word 97 -tekstinkäsittelyohjelman kieliopin tarkistustyökalu oli todennäköisesti ensimmäinen laajalle kuluttajien käyttöön levinnyt NLP-sovellus (Dale 2000: 6; Heidorn 2000: 181-207). Nykyään kuluttajilla on käytössään runsaasti arkipäiväisiä tekniikoita, joissa hyödynnetään tekstipohjaisen NLP:n tehokkuutta. Tällaisia ovat esimerkiksi automaattisen täydennyksen ja ennakoivan tekstinsyötön kaltaiset ominaisuudet, erilaisten tekstinkäsittelyohjelmien oikeinkirjoituksen ja kieliopin tarkistustyökalut sekä useat plagiaatintunnistusjärjestelmät ja roskapostin suodatustyökalut (Medelyan 2016).

NLP-työkaluja käytetään myös laajalti liiketoiminnassa eri aloilla. Esimerkiksi ruotsalainen pankki Swedbank otti käyttöön virtuaalisen assistentin, joka auttoi asiakkaita etsimään itse vastauksia tavallisimpiin kysymyksiinsä. Tämä vapautti ihmistyöntekijät keskittymään tuottavampaan myyntiin pelkän perustason asiakaspalvelun sijaan. Kolmen ensimmäisen käyttökuukauden aikana NLP-assistentti pystyi hoitamaan 78 % asiakkaiden kyselyistä itsenäisesti siirtämättä keskustelua eteenpäin ihmistyöntekijälle. (Faggella 2018.)

Terveydenhoitoalalla puolestaan NLP-sovelluksia käytetään esimerkiksi parantamaan sähköisten terveystietojen tarkkuutta ja täsmällisyyttä siten, että vapaamuotoisesta tekstistä muokataan standardoidun mallin mukaista tietoa. Puheohjattu NLP antaa terveydenhoidon ammattilaisille mahdollisuuden sanella muistiinpanoja niiden kirjoittamisen sijaan tai tuottaa potilaille henkilökohtaisesti räätälöityjä hoito-ohjeita. Tämän hetken kiinnostavin sovellusala lienee kuitenkin NLP-työkalujen, kuten IBM:n kuuluisan Watsonin, käyttö kliinisessä työssä, jossa ne voivat tukea lääkäreitä hoitoratkaisujen valinnassa. Työkalujen avulla voidaan käydä läpi lääkäreiden vapaamuotoisia muistiinpanoja ja poimia niistä automaattisesti potilasta koskevat riskitekijät tai muut oleelliset seikat, kuten esimerkiksi potilaan asumisolosuhteisiin tai mielenterveyteen liittyvät havainnot, jotka saattavat jäädä puuttumaan strukturoidun mallin mukaan täytetyistä sähköisistä potilastiedoista. NLP:n soveltamista terveydenhoitoalalle hankaloittavat kuitenkin lääkärien käyttämä vaikeaselkoinen ammattikieli ja useiden termien tai lyhenteiden monimerkityksisyys. (Bresnick 2016.)

Kielipalvelujen alalla NLP, koneoppiminen ja konekääntäminen (machine translation) ovat tärkeitä kehitysalueita. Tämän vuoden puolella OpenAI on esitellyt kehittämänsä oppivan ja ennustavan mallin nimeltä GPT-2, joka osaa tuottaa koherenttia, luonnollisen tuntuista

tekstiä (annetussa esimerkissä englanniksi) sekä vastata ”lukemaansa” tekstiä koskeviin kysymyksiin, vaikka sitä ei ennakkoon olisikaan koulutettu juuri vastaavan kaltaisilla teksteillä. Tällaista yleispätevää mallia voisi käyttää muun muassa parantamaan keskusteluagenttien (chatbottien) suoritusta sekä kääntämään tekstejä kielestä toiseen. (Radford et al. 2019.)

Erilaiset kielialan NLP-sovellukset ovat erityisen tärkeitä maailman toiseksi väkirikkaimmassa maassa Intiassa, jonka talous kasvaa kohisten. Intiassa puhutaan yhteensä lähes 1600 kieltä tai murretta, ja maassa on peräti 22 virallista kieltä ja useita erilaisia kirjoitusjärjestelmiä. Vaikka englantia puhutaankin laajalti, noin 80-85 % väestöstä ei kuitenkaan osaa sitä. Viime vuosina älypuhelinten käyttö maaseudulla on lisääntynyt huomattavasti, ja tämä on lisännyt paikallisilla kielillä tarjottujen palvelujen kysyntää. (Sheth 2017.) Tähän liittyen, kun etsin materiaalia opinnäytetyön tietoperustaosiota varten, panin itsekin merkille, että huomattava osa NLP:tä ja chatbotteja käsittelevistä artikkeleista, kirjoista ja blogeista on intialaisten asiantuntijoiden kirjoittamia.

Markkinatutkimusyritys Tractican raportin mukaan (Tractica 2017) NLP-teknologia on voimakkaasti kasvava ala, ja siihen liittyvien ohjelmistojen maailmanlaajuisen markkina-arvon arvioidaan lähes 40-kertaistuvan vuodesta 2016 (136 miljoonaa dollaria) vuoteen 2025 (5,4 miljardia dollaria) mennessä. Laitteet ja palvelut mukaan luettuina koko NLP-kentän arvon ennustetaan saavuttavan jopa 22,3 miljardia dollaria vuoteen 2025 mennessä.

Kovasta kasvusta ja huikeista tuottoennusteista huolimatta NLP-alan pitkän linjan asiantuntija, MIT:n huippututkija Boris Katz suhtautuu skeptisesti tietokoneiden kielenoppimiskyvyn rajoihin. Katzin mukaan merkittävin tieteellinen tutkimus koneoppimisen alalla on peräisin jo parinkymmenen vuoden takaa, vaikka silloin kehitettyjä ideoita onkin pystytty ottamaan laajalti käyttöön vasta viime vuosina teknologian kehityksen myötä. Nykytekniikka osaa tehokkaasti havaita kielellistä säännöllisyyttä ja toistoa, mikä toimii tiettyyn rajaan asti käytännön sovelluksissa, kuten ennakoivassa tekstinsyötössä. Laskennallinen toiston ennakointi ei kuitenkaan osoita todellista älykkyyttä. Katz vertaa koneoppimista ihmislapsen kielenopetteluun. Koneoppimisessa tietokonetta koulutetaan valtavien datamassojen avulla, kun taas ihmiset elävät kolmiulotteisessa, moniaistisessa maailmassa ja oppivat kieltä samalla kun oppivat tuntemaan maailmaa. Katzin mukaan tekoälytutkimuksessa tulisikin keskittyä kouluttamaan tietokoneita samanlaisella monimuotoisella tavalla soveltaen muun muassa kehityspsykologian, kognitiotieteen ja neurotieteen menetelmiä. (Knight 2019.)

3 Chatbotit

Chatbotin voidaan määritellä olevan tietokoneohjelma, joka suorittaa automatisoituja tehtäviä keskustelualustoilla simuloiden keskustelua ihmisten välillä. Yrityksen oma chatbotti voi toimia virtuaalisena asiakaspalvelijana, joka on aina tavoitettavissa. (Hupli 2018.) Nimitys ”chatterbot” eli ”jutteleva (ro)botti” esiintyi ensimmäisen kerran vuonna 1994, ja sen keksi Michael Maudlin (Khan & Das 2018). Suppean internet-haun perusteella suomenkielisissä kirjoituksissa chatbotista näkyy käytettävän myös termejä (asiakas)palvelurobotti tai keskustelu(ro)botti. Koska mitään yksiselitteistä suomenkielistä termiä ei ole vielä vakiintunut käyttöön, käytän tässä työssä termiä chatbotti.

3.1 Chatbottien historiaa

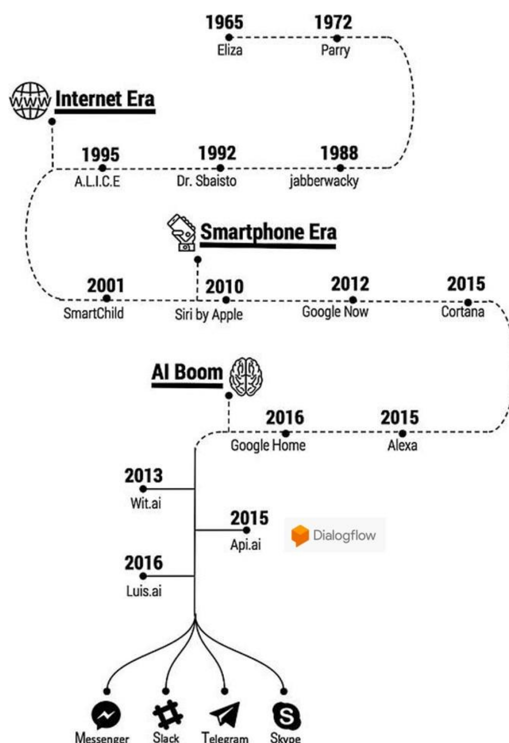
Keskustelupohjaisista käyttöliittymistä voidaan käyttää useita nimiä, ja niiden tekninen rakenne (tietokannat, moduulit) ja käytännön esitystapa (teksti, puhe, visuaalinen esitys) voidaan toteuttaa monella eri tavalla. Yhdistävä tekijä on kuitenkin kyky kommunikoida käyttäjien kanssa luonnollista kieltä käyttäen. (Janarthanam 2017.)

Nykyään chatbotteja on kaikkialla, mutta ajatuksena tekstipohjainen keskusteleva käyttöliittymä on jo lähes 70 vuotta vanha. Brittiläinen matemaatikko Alan M. Turing kehitti jo vuonna 1950 kuuluisan Turingin testin, jonka tarkoituksena oli määritellä, osasiko tietokone ”ajatella”. Turing päätteli, että jos tietokone käyttäytyy ja reagoi tietoisen olennon tavoin, sitä on pidettävä tietoisena olentona. Testissä ihmiskäyttäjä keskustelee etäyhteyden välityksellä sekä toisen ihmisen että testattavan koneen kanssa, esittää molemmille kysymyksiä ja päättelee vastausten perusteella, onko vastaaja kone vai ihminen. Koe toistetaan useita kertoja eri ihmisten kanssa, ja mitä useampi keskustelukumppani erehtyy luulemaan konetta ihmiseksi, sitä paremmin koneen katsotaan menestyneen testissä. (Britannica 2019.) Virallisen määritelmän mukaan Turingin testin läpäisyyn vaaditaan se, että kone onnistuu huijaamaan 30 % keskustelukumppaneista, eikä tähän kykeneviä käyttöliittymiä osattu rakentaa vielä kymmeniin vuosiin Turingin ajatuskokeen jälkeen (Janarthanam 2017). Turingin testin läpäisi vasta vuonna 2014 venäläisten kehittäjien luoma botti nimeltä Eugene, joka esitti 13-vuotiasta ukrainalaispoikaa (Schofield 2014).

Modernit chatbotit saivat alkusysäyksen vuonna 1964, jolloin Massachusetts Institute of Technologyn tutkija Joseph Weizenbaum kehitti tekstipohjaisen botin nimeltä Eliza. Se noudatti hyvin yksinkertaisia sääntöjä ja enimmäkseen toisti käyttäjien repliikit heille takaisin simuloiden tiettyä psykologista terapiasuuntausta. (Weizenbaum 1966.) Tekstikäyttöliittymät painuivat taka-alalle kolmen viimeisen vuosikymmenen aikana, kun tietokoneita

on käytetty pääasiassa graafisten käyttöliittymien kautta. Nykyään tekoälyn kehitys ja laitteiden koon pieneneminen vaikuttavat yhdessä siihen, että keskustelupohjaiset käyttöliittymät ovat taas nousussa. Etenkin älykkäiden kodinkoneiden ja muiden vastaavien laitteiden kanssa kätevin vuorovaikutustapa on puhekäyttöliittymä. (Janarthanam 2017.) Tosin puheohjauksellakin toimivat järjestelmät kääntävät puheen sisäisesti ensin tekstimuotoon jatkokäsittelyä varten (Khan & Das 2018).

Luontevasti jutteleva chatbotti ei sinänsä vielä merkitse suurta edistysaskelta tekoälyn saralla. Kriitikoiden mukaan botit, jotka on varta vasten kehitetty läpäisemään Turingin testi, eivät varsinaisesti edistä tekoälyn kehitystä tai toimi hyödyllisinä työkaluina missään käytännön tarkoituksessa. Tekoälytutkimuksen ja etenkin koneoppimisen ja luonnollisen kielen käsittelyn tutkimuksen piirissä onkin kehitetty erilaisia keskustelevia käyttöliittymiä, joilla oli selkeä käyttötarkoitus. Ne saattavat esimerkiksi vastata kysymyksiin ja tarjota rajapinnan tietokantaan tai puheohjattavaan järjestelmään. 2010-luvun kehitysaskelia keskustelevien käyttöliittymien alalla edustavat Applen Siri, IBM:n Watson, Microsoftin Cortana, Amazonin Alexa sekä Google Assistant. (Janarthanam 2017, Khan & Das 2018.) Lisäksi kehittäjät ovat muutaman viime vuoden aikana voineet lisätä omia chatbottejaan mm. Facebook Messengerin ja Slackin alustoille, mikä mahdollistaa laajan käyttäjäkunnan saavuttamisen (Khan & Das 2018). Kuva 3 esittää chatbottien historian tiivistetyssä muodossa aina Elizasta nykyaikaisiin monialustaisiin sovelluksiin. Kaaviossa mainittu Api.ai oli Dialogflow'n alkuperäinen nimi ennen kuin tuote liitettiin Googlen ekosysteemiin.



Kuva 3. Chatbottien historiaa (Khan & Das 2018).

3.1 Chatbottien luokittelua

Hupli (2018) jakaa chatbotit kolmeen kategoriaan. Niistä yksinkertaisimpien eli käsikirjoitettujen bottien kanssa voi keskustella vain etukäteen ohjelmoitujen dialogien kautta, joissa kysymykset ja vastausvaihtoehdot on valmiiksi annettu. Älykkäässä chatbotissa taas käyttäjä voi kirjoittaa vapaasti haluamansa kysymyksen, jonka botti tulkitsee ja johon se pyrkii vastaamaan, tosin usein valmiiksi kirjoitetulla vaihtoehdolla. Hybridibotit puolestaan ovat vielä älykkäämpiä, sillä ennalta ohjelmoitujen polkujen lisäksi ne tulkitsevat myös vapaata tekstiä, mikä tekee vuorovaikutuksesta sujuvampaa. Kahden viimeisen kategorian chatboteissa käytetään apuna tekoälyä ja koneoppimista sekä luonnollista kieltä käsitteleviä työkaluja.

Toinen tapa luokitella botteja on sen mukaan, minkälaista lisäarvoa ne tarjoavat käyttäjälle. Elharrar (2017) jakaa botit tällä perusteella seitsemään eri tyyppiin. Yleisin tyyppi on ”optimoija”, jonka tarkoitus on suorittaa joku käytännön tehtävä paremmin tai sujuvammin kuin käyttäjä voisi itse sen tehdä sovelluksen tai verkkosivun kautta. Elharrarin mukaan suurin ongelma on kuitenkin siinä, että valtaosa tällaisista boteista ei itse asiassa ole lainkaan kätevämpi käyttää kuin vaihtoehtoiset menetelmät, vaan jopa päinvastoin.

Muita bottityyppejä ovat muun muassa ”yhden hitin ihme” eli minibotti, jolla voi esimerkiksi tehdä kuvasta meemin, sekä ”proaktiivinen botti”, joka toimittaa oikeaa tietoa oikeaan aikaan, esimerkiksi ajankohtaisia säätietoja. ”Sosiaalinen botti” suorittaa myös tietyn tehtävän, mutta toimii viestialustalla usean ihmisen käytössä samanaikaisesti, kuten vaikkapa Slackin botit. ”Bottikilpiä” puolestaan voidaan käyttää asiakastuessa tai vastaavassa palvelussa käsittelemässä ikäviä tilanteita neutraalilla tavalla: ”The idea is to exchange a talk to a cold-hearted person with a chat with a friendly robot.” Botti, joka keskustelee käyttäjän kanssa silkan keskustelun vuoksi, on tyypiltään ”juttelija”, mutta tällainen botti voi myös toimia mielipidevaikuttajana, etenkin jos se on profiloitu jonkun suositun julkisuuden henkilön mukaan. Viimeinen kategoria on ”superbotti”, jota nimeä Elharrar käyttää Alexan, Sirin tai Cortanan kaltaisista monipuolisista, älykkäistä henkilökohtaisista avustajista.

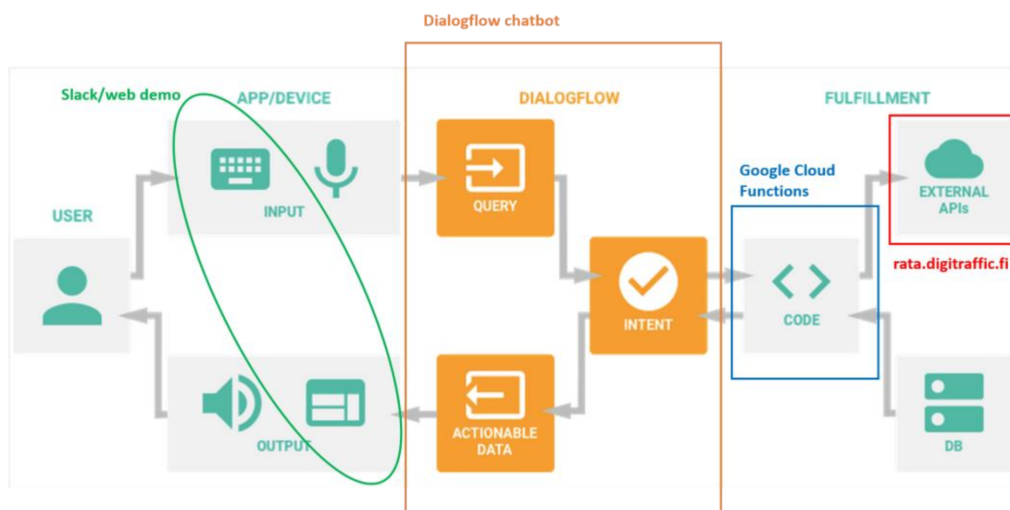
Tässä opinnäytetyössä käyttämäni Dialogflow-alustalla rakennetun chatbotin arvioisin kuuluvan Huplin määritelmän mukaan jonnekin ensimmäisen ja toisen kategorian väliin. Dialogflow tarjoaa kyllä käyttöön ”älyä” NLP:n ja koneoppimisen muodossa, mikä mahdollistaa esimerkiksi käyttäjän antamien aika- ja päivämäärätietojen tulkinnan ja käsittelyn joustavasti. Bottini on kuitenkin rakennettu varsin suppeaa käyttötarkoitusta varten, eli se

hakee vain tietynmallisia tietoja vain yhdestä lähteestä ja tarjoaa vastaukset valmiiksi käsikirjoitetussa muodossa.

Elharrarin kategorioista bottini sijoittuisi lähinnä yleisimpään optimoijaluokkaan. Käyttömukavuus tosin kärsii siitä, että koska rautatieliikenteen avoin rajapinta rajoittaa palautetut tiedot vain suoriin junayhteyksiin annettujen asemien välillä, botin toimittamat aikataulutulokset eivät kata läheskään kaikkia VR:n nettisivuhaun kautta saatavia aikataulutietoja, joissa huomioidaan myös junan vaihdot.

4 Kehitysympäristön rakentaminen

Ennen kuin saatoin ryhtyä kehittämään itse chatbottia, minun piti rakentaa kehitysympäristö ja varmistaa yhteyden toimiminen Dialogflow'sta Google Cloud Functionsin kautta avoimeen rajapintaan ja takaisin. Kuva 4 on yleisesitys siitä, miten Dialogflow-chatbotin ympäristön eri osat ovat yhteydessä toisiinsa. Keskiössä on Dialogflow, jossa rakennettu chatbotti hakee backendissä (tässä Google Cloud Functionsissa) määritellyn koodin välityksellä dataa ulkoisesta rajapinnasta (tässä tapauksessa *rata.digitraffic.fi*), joka puolestaan on tietenkin omalla tahollaan yhteydessä datan sisältävään tietokantaan, ja palauttaa sen käyttäjälle jonkinlaisen käyttöliittymän kautta, jotka tässä tapauksessa ovat Slack ja verkkodemotyökalu. Näitä komponentteja ja niiden yhteistyötä käsitellään tarkemmin tässä luvussa.



Kuva 4. Chatbotin kehitys- ja toimintaympäristö. (Pohjana oleva kuva: Rizvi 2018.)

4.1 Dialogflow

Dialogflow[2] on Googlen tarjoama alusta, jolla käyttäjä voi luoda oman teksti- tai puheohjattavan chatbotin tekoälyä ja koneoppimista hyödyntäen. Dialogflow pyörii Google Cloud Platform -alustalla ja on optimoitu yhteistyöhön Google Assistantin kanssa. Dialogflow käyttää koneoppimisen tekniikoita opetellakseen käyttäjien antamia syötteitä ja tarjoaa niiden perusteella mahdollisimman hyödyllisen vastauksen. Sen kautta voi integroida saman botin toimimaan monella eri alustalla, Google Assistantin lisäksi myös esimerkiksi

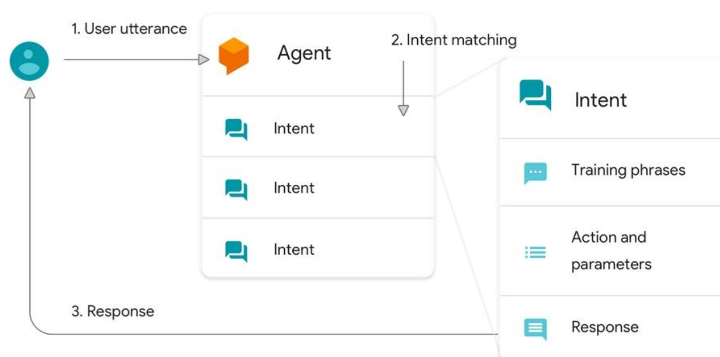
[2] Alusta oli aiemmalta nimeltään API.AI, mutta ostettuaan sen kehittäneen yrityksen Google antoi sille vuonna 2017 uuden nimen Dialogflow (Google 2018). Netistä löytyy silti vielä vanhallakin nimellä tuotteeseen liittyvää materiaalia, mm. chatbottien kehitykseen liittyviä kysymyksiä ja koodiesimerkkejä (Stack Overflow, Github).

Alexan, Cortanan, Facebook Messengerin tai Slackin alustalla, samoin kuin useiden eri laitteiden kautta. (Dialogflow 2019a.)

Valmiita komponentteja käyttämällä yksinkertaisen chatbotin voi periaatteessa luoda myös ilman ohjelmointitaitoa. Tällainen botti rajoittuu kuitenkin käytännössä toimimaan yksinomaan Dialogflow'n sisällä ilman yhteyksiä ulkoisiin rajapintoihin. Se voi esimerkiksi antaa määrämuotoisia, ennalta ohjelmoituja vastauksia käyttäjän kysymyksiin. Ulkoisten yhteyksien määrittäminen vaatii kykyä ymmärtää Dialogflow'n API-määrityksiä sekä todennäköisesti taitoa ohjelmoida webhook-toiminnallisuus. (Mohanoor 2018a.)

Dialogflow'ssa chatbottia kutsutaan nimellä **agent**, josta käytän tässä suomennosta agentti. Agentti koostuu useasta toiminnosta nimeltä **intent** (aikomus), joista jokaiselle on määritelty selvä tehtävä keskustelussa käyttäjän ja agentin välillä. Käytännössä jokainen aikomus vastaa yhtä keskusteluvuoroa (**conversation turn**): käyttäjä antaa syötteen, ja agentti jäsentää sen ja palauttaa vastauksen käyttäjälle. (Dialogflow 2019b.) Esimerkiksi omassa botissani Welcome Intent käynnistyy, kun käyttäjä syöttää jonkun kyseisen aikomuksen tunnistaman tervehdyksen ("hello", "hi", "good morning" jne. – näitä voi opettaa Dialogflow'lle niin paljon kuin tarvitsee), ja botti vastaa käyttäjälle lyhyellä tekstillä, jossa kerrotaan, mitä botti osaa tehdä.

Kuva 5 esittää Dialogflow-agentin rakenteen yleisellä tasolla. Kuvan kohdassa 1 käyttäjä antaa syötteen (User utterance). Kohdassa 2 Dialogflow poimii NLP:n avulla syötteestä oleelliset parametrit ja etsii niitä vastaavan aikomuksen (Intent matching). Tässä käytetään apuna agentin koulutusta aiempien samaa aikomusta koskevien syötteiden avulla (Training phrases). Aikomukseen on Dialogflow'ssa määritelty myös parametrit, joita syötteestä etsitään, sekä toiminto, joka käynnistetään (Action and parameters). Lopulta kohdassa 3 Dialogflow palauttaa käyttäjälle asianmukaisen vastauksen (Response) joko suoraan ohjelman sisäisesti tai käyttämällä ulkoista webhookia, jonka toiminta kuvataan tarkemmin alempana.



Kuva 5. Dialogflow-agentin toiminta yleisellä tasolla (Dialogflow 2019c).

Toinen tärkeä käsite on **entity** (entiteetti[3]). Entiteetit määrittelevät, mitä spesifisiä tietoja käyttäjien luonnollisella kielellä antamista syötteistä poimitaan. Dialogflow tarjoaa käyttöön laajan valikoiman valmiita järjestelmäentiteettejä (system entities), ja lisäksi kehittäjä voi laatia tarpeen mukaan omia entiteettejä. (Dialogflow 2019d.) Esimerkiksi tässä botissa käytetään *@sys.date-time*-nimistä järjestelmäentiteettiä (Google 2019a) käsittelemään käyttäjän antamaa aika- ja päivämäärätietoa sekä itse määrittelemiäni entiteettejä nimeltä *@from* ja *@to*, jotka määrittävät lähtö- ja saapumisasemat.[4] Entiteettien avulla kerätyt tiedot välitetään parametreinä webhookille.

Muita tärkeitä käsitteitä ovat **contexts** eli kontekstit, joiden avulla voidaan siirtää tietoa yhdestä aikomuksesta toiseen ja hallinnoida keskustelun kulkua (Dialogflow 2019e). Omassa botissani käytän kontekstia ainoastaan muistamaan annetut *from*- ja *to*-parametrit yhden keskusteluvuoron ajan, kuten tarkemmin kuvattu kohdassa 5.2.

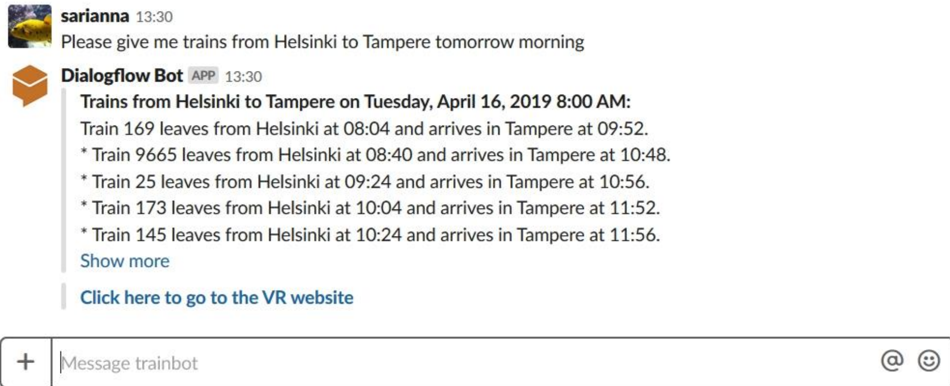
Lisäksi Dialogflow'ssa on myös mahdollisuus kutsua aikomuksia silloin, kun jotain tapahtuu ulkoisella alustalla (**events**), vaikka käyttäjä ei olisikaan antanut syötettä (Dialogflow 2019f). Tätä toimintoa en ole hyödyntänyt botissani.

Tässä opinnäytetyössä ohjelmoitu chatbotti toimii ainoastaan tekstipohjaisesti ja vain englanniksi, vaikka hakeekin käyttämänsä datan suomalaisesta avoimesta rajapinnasta. Dialogflow tukee useita muita luonnollisia kieliä englannin lisäksi, mutta toistaiseksi ei vielä suomea (Dialogflow 2019g). Opinnäytetyötä varten laatimaani demosovellusta on tarkoitus esitellä kansainväliselle yleisölle ja mahdollisesti käyttää pohjana muita sovelluksia varten, joten englannin käyttö on perusteltua.

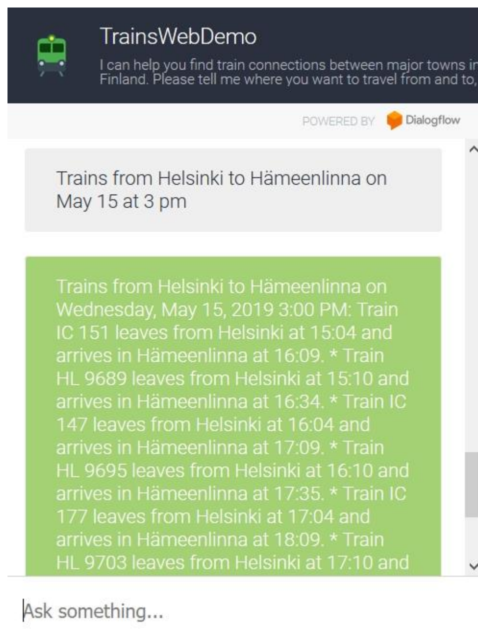
Dialogflow tukee useita eri integraatiomahdollisuuksia sosiaalisen median ja viestinnän alustoihin (Dialogflow 2019h), joista tässä havainnollistamassa kaksi kuvaa eri alustoilta. Kehittämäni chatbotti toimii toistaiseksi toimeksiantajayrityksen sisäisessä Slack-ympäristössä (Kuva 6) ja Dialogflow'n omassa, visuaalisesti vaatimattomassa verkkodemointegraatiossa (Kuva 7).

---

[3] Entity-sanan voisi suomentaa myös olioksi, mutta käytän tässä mieluummin entiteettiä, jotta termi ei sekaannu olioon eli objektiin ohjelmointiterminä.
[4] Dialogflow'ssa on kyllä olemassa valmiita järjestelmäentiteettejä maantieteellisiä paikkoja varten, mutta useimpia suomalaisia paikannimiä se ei tunnista, joten ne täytyy määritellä erikseen omana entiteettinään.



Kuva 6. Esimerkki chatbotin toiminnasta Slack-ympäristössä.



Kuva 7. Esimerkki chatbotin toiminnasta verkkodemoympäristössä.

Jatkossa on tarkoitus integroida botti toimimaan myös toimeksiantajayritykseni kehittämällä RCS-alustalla, jossa sitä hyödynnetään demotarkoituksiin esittelemään alustan käyttöä. Chatbotin ulkoasu kohenee huomattavasti, kun käyttöön saadaan RCS-alustan monipuoliset multimediaominaisuudet.

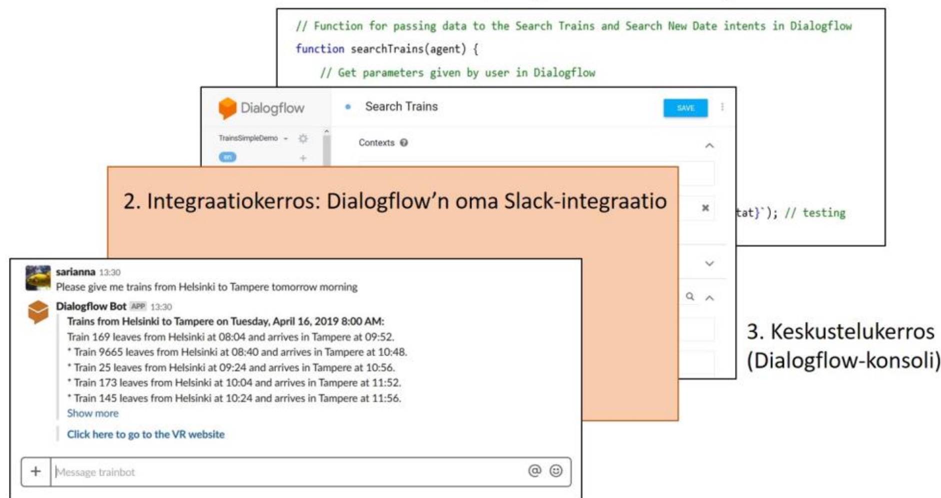
Chatbottini hakee käyttäjän haluaman tiedon eli tässä tapauksessa juna-aikataulut avoimesta ulkoisesta rajapinnasta (katso kohta 4.3). Tässä käytetään hyväksi Dialogflow'n tukemaa ulkoista webhook-toimintoa. En ole löytänyt webhook-termille hyvää suomenkielistä käännöstä, joten käytän tässä työssä englanninkielistä termiä.

Webhook on tapa, jolla sovellus voi toimittaa muille sovelluksille ajantasaista tietoa. Tavanomaista ohjelmointirajapintaa (API) käytettäessä tietoa tarvitsevan sovelluksen täytyy säännöllisin väliajoin kysellä tietoa saadakseen sen mahdollisimman tuoreena. Webhookia käytettäessä tieto haetaan juuri silloin kun sitä tarvitaan, joten se on aina ajanmukaista. Webhook tekee sovellukseen HTTP-kutsun, joka täytyy käsitellä sovelluksessa. (Quinlan 2014.)

Dialogflow-sovelluksessa käytetään termiä **fulfillment** (täyttäminen, toteutuminen) kuvaamaan sitä, kun käyttäjän pyyntö täytetään eli käyttäjä saa haluamansa tiedon chatbotista. Fulfillment voidaan ohjelmoida tapahtumaan suoraan Dialogflow-sovelluksen sisällä tai ulkoisesti webhookin kautta, kuten tässä käsitellyssä esimerkkitapauksessa tehdään. Kun Dialogflow'n logiikka päättelee, että käyttäjän viestin sisältö täyttää fulfillmentin kutsumisen ehdot, Dialogflow tekee HTTP POST -kutsun webhookille ja lähettää sille JSON-objektin, jossa on tietoa kyseessä olevasta aikomuksesta (intent). Webhook tekee tiedon perusteella siihen ohjelmoidut toiminnot ja lähettää Dialogflow'lle vastauksen. (Dialogflow 2019i.)

Tämän chatbotin kohdalla näin tapahtuu Search Train -nimisessä intentissä (Kuva 13), jossa tyypillisimmässä tapauksessa on käyttäjän antamat lähtö- ja saapumispaikat, eli itse määrittämäni entiteettien *@from* ja *@to* mukaiset parametrit, sekä jossain muodossa oleva päivämäärä- ja/tai ajankohtatieto, jonka käsittelee Dialogflow'n oma järjestelmäentiteetti *@sys.date-time*. Näiden tietojen perusteella Google Cloud Functionsiin ohjelmoitu webhook hakee *rata.digitraffic.fi*-rajapinnasta halutut juna-aikataulut ja lähettää ne eteenpäin Dialogflow'hun käyttäjän nähtäviksi. Tämä on tarkemmin selitetty luvussa 5.

Tiivistettynä Dialogflow'lla rakennettua chatbottia voisi kuvata neljästä kerroksesta koostuvana kokonaisuutena. Ylimpänä ja lähimpänä loppukäyttäjää on käyttöliittymäkerros (UI Layer), joka on yleensä visuaalinen. Sen alla on integraatiokerros (Middleware/Integration Layer), joka yhdistää käyttöliittymän Dialogflow-agenttiin. Seuraavaksi on keskustelukerros (Conversation Layer), joka kattaa kaikki Dialogflow-alustan sisällä määritellyt komponentit, kuten aikomukset ja entiteetit. Alimpana ja kauimpana käyttäjästä on webhook-kerros (Webhook/Fulfillment Layer), joka on tarpeen liiketoimintalogiikan toteuttamiseksi, jos botin on tarkoitus hakea dataa ulkoisesta lähteestä. Kooditasolla kerrosten toiminnallisuus voi mennä osittain päällekkäin, ja kaikissa boteissa ei välttämättä ole kaikkia kerroksia. (Mohanoor 2019a.) Kuva 8 havainnollistaa, miten nämä kerrokset yhdistyvät toisiinsa kehittämässäni chatbotissa.



Kuva 8. Dialogflow-chatbotin kerrokset (kuvan mallina Mohanoor 2019a).

4.2 Google Cloud Functions

Palvelimettomassa Google Cloud Functions -ympäristössä käyttäjät voivat rakentaa omia pilvitoimintoja (cloud functions). Ympäristö tukee yksinkertaisia toimintoja, jotka yhdistyvät käyttäjän pilvi-infrastruktuurin tapahtumiin. Toiminto käynnistyy, kun sen tarkkailema tapahtuma (event) käynnistyy. Tapahtuma on mikä tahansa asia, joka tapahtuu pilviympäristössä, esimerkiksi tietokannan päivitys tai uusien tiedostojen lisääminen järjestelmään. Tapahtumiin voi vastata luomalla käynnistysehdon (trigger), jossa määritellään mihin tapahtumaan ehto reagoi. Kun toiminto sidotaan käynnistysehtoon, tapahtumiin voidaan tarttua ja niitä voidaan käsitellä. (Google 2019b.) Kuva 9 havainnollistaa, miten pilvitoiminto vastaa tapahtumiin, kutsuu ulkoisia rajapintoja ja välittää vastauksen takaisin sitä kutsuneelle taholle.

Kuva 9. Google Cloud Functions (Google 2019c).

Tässä käsitellyssä chatbot-esimerkissä tapahtuma on chatbotin käyttäjältä saatu viesti, johon lähetetään vastaus. Viestin sisällöstä riippuen vastaus voi olla määrämuotoinen, jos kyseessä on esimerkiksi tervetulotoivotus tai keskustelun lopetus, tai muodoltaan vaihte-

leva silloin kun pilvitoiminto kutsuu ulkoista rajapintaa eli *rata.digitraffic.fi*-rajapintaa ja hakee sieltä käyttäjän pyytämät aikataulutiedot.

Google Cloud Functionsissa käyttäjän laatima koodi suoritetaan Googlen valmiissa ympäristössä, jossa käyttäjän ei tarvitse huolehtia infrastruktuurista eikä ylläpitää, konfiguroida tai päivittää omia palvelimia. Google hoitaa ympäristön sekä ohjelmiston että laitteiston osalta, joten käyttäjän täytyy vain kirjoittaa haluamansa koodi pilvitoimintoon. Resursseja on käytössä niin paljon kuin kulloinkin tarvitaan aina muutamasta päivittäisestä kutsukerrasta miljooniin kertoihin. Omaa pilvitoimintoa on helppo testata paikallisesti, koska sen voi siirtää mihin tahansa standardien mukaiseen ympäristöön, joka toimii Google Cloud Functionsin tukemilla kielillä ja versioilla. Näitä ovat Node.js 6 (tuetaan 4/2020 asti), 8 tai 10 (beta), Python 3.7 tai Go 1.11. Cloud Functions toimii loogisena kerroksena, joka yhdistää eri pilvipalveluita toisiinsa ja täydentää olemassa olevia palveluita, ja sen kautta on helppo hyödyntää myös Googlen muuta laajaa, kehittäjille suunnattua palvelutarjontaa. (Google 2019b.)

4.3 Rautatieliikenteen avoin rajapinta

Jotta chatbot olisi oikeasti hyödyllinen apuväline eikä vain hauska lelu, sen täytyy pystyä toimittamaan käyttäjälle tämän pyytämät tiedot. Monimuotoinen ja jatkuvasti päivittyvä data haetaan yleensä jostain chatbotin ulkopuolisesta lähteestä. Rakentamani chatbotti hakee juna-aikatauluja Traffic Management Finlandin tarjoaman rautatieliikenteen avoimen rajapinnan kautta (*rata.digitraffic.fi*). Rajapinta tarjoaa monenlaista tietoa Suomen rataverkolla kulkevista junista, tässä käytettyjen aikataulutietojen lisäksi myös junien sijainti- ja kokoonpanotietoja. Tietojen lähteinä toimivat Liike-sovellukset ja matkustajainformaatiojärjestelmä MIKU. (Traffic Management Finland 2019a.)

Kehittämäni chatbotti käyttää rajapinnan reittiperusteista hakua, joka toimii seuraavan mallin mukaisen URL-osoitteen kautta:

https://rata.digitraffic.fi/api/v1/live-trains/station/
<departure_station_code>/<arrival_station_code>
?departure_date=<departure_date>&startDate=<startDate>
&endDate=<endDate>&limit=<limit>

URLissa on kaksi pakollista hakuparametriä, *departure_station_code* ja *arrival_station_code* eli lähtöaseman ja saapumisaseman asemakoodit. Haku palauttaa vain

suorat yhteydet annettujen asemien välillä. Ilman muita hakuparametrejä palautetaan hakuhetkestä 24 tunnin ajan kulkevat junat, jotka pysähtyvät annetuilla asemilla.

Vapaaehtoisia parametrejä ovat *departure_date* (yyyy-mm-dd) eli lähtöpäivä, *startDate* ja *endDate* (ISO 8601) eli haetun aikavälin tarkempi alku ja loppu, *limit* eli haettujen junien lukumäärä sekä *include_nonstopping*, jolla voidaan sisällyttää hakuun myös ne junat, jotka ajavat pysähtymättä annettujen asemien läpi. (Traffic Management Finland 2019b.)

Esimerkiksi junat Helsingistä Tampereelle 23.4.2019 klo 17-19 haetaan seuraavan URL-osoitteen kautta:

https://rata.digitraffic.fi/api/v1/live-trains/station/HKI/TPE?startDate=2019-04-23T17:00:00.000Z&endDate=2019-04-23T19:00:00.000Z

Haku palauttaa *junat*-tyyppisen vastauksen JSON-muodossa (Traffic Management Finland 2019c). Liite 5 sisältää esimerkin palautetusta datasta. Vastaustyyppi sisältää useita tietokenttiä, joista chatbotti poimii tarvitsemansa tiedot (näistä tarkemmin luvussa 5). On syytä huomata, että rajapinta palauttaa annettujen asemien läpi kulkevien junien tiedot koko niiden kulkemalta matkalta, ei ainoastaan annetulta väliltä. Yllä annetun URLin mukainen Helsinki-Tampere-haku palauttaa siis Vaasaan asti kulkevan junan kaikki asemat, ei ainoastaan Helsinki-Tampere-välille sijoittuvia asemia.

4.4 Yhdistetty kehitysympäristö

Kehittämäni chatbotin toimintaympäristö koostuu useasta palasesta. Chatbotin keskustelulogiikan ja koneoppimisen tarjoava Dialogflow on vahvasti optimoitu toimimaan yhdessä niin ikään Googlen omistamien Google Assistant -työkalun ja Firebase-kehitysympäristön kanssa. Lopullisena tarkoituksenani on kuitenkin saada aikaan itsenäinen botti, joka yhdistetään lähitulevaisuudessa toimeksiantajayrityksen kehittämään RCS-ympäristössä toimivaan ratkaisuun, joten puhtaasti Googlen ekosysteemiin optimointi ei ollut tarkoituksenmukaista.

Dialogflow on periaatteessa ilmainen ja vapaasti käytettävissä, mutta ilmaisversio ei salli yhteyksiä Googlen ekosysteemin ulkopuolelle ulkoisiin rajapintoihin (Google 2019d), joten yksinomaan sillä laaditun botin käytännön hyöty jää vähäiseksi. Kun Dialogflow’n keskusteluagentti (eli chatbotti) ohjataan toimimaan ulkoisen webhookin kautta, joka puolestaan rakennetaan (maksullisessa) Google Cloud Functions -ympäristössä, avautuu mahdolli-

suus hakea chatbotin kautta tietoa mistä tahansa ulkoisesta rajapinnasta, jota voidaan kutsua pilvitoiminnolla.

Kun lähdin viemään eteenpäin tätä yhdistelmää, jonka jokainen komponentti oli ensi alkuun minulle uusi ja vieras, törmäsin pian vaikeuksiin löytää näiden työkalujen käyttöön soveltuvaa tietoa ja ohjeita. Google tarjoaa kyllä runsaasti omaan ekosysteemiinsä upotettuja esimerkkejä ja demosovelluksia, mutta avun löytäminen epätavanomaisempia ratkaisuja varten vaati pitkällistä Stack Overflow'n kaltaisten palstojen ja itsenäisten blogien läpi kahlaamista, kyselemistä ja testailua. Tämän opinnäytetyön sivutuotteena syntyikin varsinaisen chatbotin kehittämisen lisäksi alla annetut käytännön ohjeet siihen, miten tällä yhdistelmällä voidaan luoda chatbotin kehitysympäristö.

Alkuoletuksena on, että käyttäjällä on olemassa oleva maksullinen tili Google Cloud Platforms -ympäristössä. Itse sain opinnäytettä tehdessäni käyttää toimeksiantajayrityksen kautta tarjottua tiliä, joten en joutunut luomaan omaa tiliä alusta asti. Mikäli tiliä ei vielä ole, sellaisen voi helposti luoda annettujen ohjeiden avulla (Google 2019e). Lisäksi ainakin tämän kirjoitushetkenä keväällä 2019 Google mainostaa tarjoavansa uusille käyttäjille 300 USD edestä ilmaista käyttöoikeutta ennen kuin palveluista laskutetaan käyttäjän luottokorttia.

Kun Google Cloud Platforms on käytössä, Cloud Functions -alustalla luodaan toiminto napsauttamalla **Create Function** -nappia (Liite 1, Kuva 17). Avautuu **Create function** -näkymä, johon annetaan toiminnon tiedot. Esimerkkikuvassa (Liite 1, Kuva 18) näkyy oletuksena annetun **function-1**-nimisen toiminnon tiedot. Kuvassa peitettynä näkyvä URL on sama kuin Dialogflow'n **Fulfillment**-näkymässä annettu webhookin URL.

URL on muotoa *https://valittu-palvelin-kayttaja.cloudfunctions.net/toiminto,*
jossa *valittu-palvelin* on käytetty Googlen palvelin (esimerkiksi *europe-west-1*), *kayttaja* on käyttäjän nimi ja *toiminto* on toiminnolle annettu nimi (kuten esimerkissä *function-1*).

Demosovelluksessani en ole säätänyt muita kuvassa näkyviä asetuksia, vaan **Memory allocated** -kohtaan muistin määräksi sai jäädä 256 MB, **Trigger** eli toiminnon käynnistysehto on HTTP ja **Source code** -lähteenä on **Inline editor** eli kuvassa näkyvä editori. Tämä editori ei kuitenkaan sovellu pitkäjänteiseen kehitystyöhön, joten käytännössä käytin monipuolisempaa ulkoista koodieditoria (Visual Studio Code) ja kopioin sisäiseen editoriin aina kulloisenkin testattavan koodin. Laajamittaisemmassa kehitystyössä toki olisi varmasti tarpeen käyttää jonkinlaista repository-ympäristöä koodille ihan versionhallinnan-

kin vuoksi, mutta koska projektini oli pieni ja tein sitä yksin, selvisin manuaalisella versionhallinnalla.

Runtime eli käytettävä ohjelmointiympäristö on demosovelluksessani Node.js 8, joka on tällä hetkellä (huhtikuussa 2019) Google Cloud Functionsissa annettu oletusvaihtoehto. Node.js oli valintani myös siksi, että osasin jo valmiiksi jonkin verran JavaScriptiä, vaikka Node-ympäristö ja siihen liittyvä toiminnallisuus moduuleineen ja riippuvuuksineen vaatikin paljon uuden asian oppimista. Muita tuettuja kieliä ovat myös Noden versiot 6 (4/2020 asti) ja 10 (beta) sekä Python 3.7 ja Go 1.11.

Cloud Function -toiminnon asetuksissa **Function to execute** -kenttään tulee sama nimi kuin *index.js*-koodin *exports*-kohdassa (Liite 1, Kuva 19). **Service account** -valikosta valitaan **Dialogflow Integrations** ja napsautetaan **Create**-nappia. Luotu toiminto tulee näkyviin **Cloud Functions** -listaan (Liite 1, Kuva 17). Tämän jälkeen voidaan ryhtyä kehittämään toimintoa sisäisessä editorissa olevien *index.js*- ja *package.json*-tiedostojen avulla (Liitteet 3 ja 4). Käyn kehittämäni chatbotin rakenteen tarkemmin läpi kohdassa 5.

Kun Google Cloud Functions on otettu käyttöön ja pilvitoiminto luotu, voidaan luoda Dialogflow'ssa uusi agentti, joka hyödyntää kyseistä pilvitoimintoa ulkoisena webhookina. Dialogflow'n oma dokumentaatio tarjoaa kattavan ohjeistuksen siihen, miten alusta otetaan käyttöön ja ensimmäinen agentti (eli chatbotti) luodaan (Dialogflow 2019j). Alustaa käytetään Google-tilin kautta. On syytä huomioida, että tilin on oltava sama kuin Google Cloud Platforms -ympäristössä käytettävä tili, jotta Google Cloud Functionsin pilvitoiminto voidaan yhdistää Dialogflow'n keskusteluagentin webhookiksi. Esimerkiksi omassa tapauksessani tein ensi alkuun sen virheen, että olin kirjautunut Dialogflow'hun henkilökohtaisella Google-tililläni, kun taas Google Cloud Functions toimi toimeksiantajayrityksen Google-tilin kautta, eikä Dialogflow näin ollen saanut yhteyttä pilvitoimintoon.

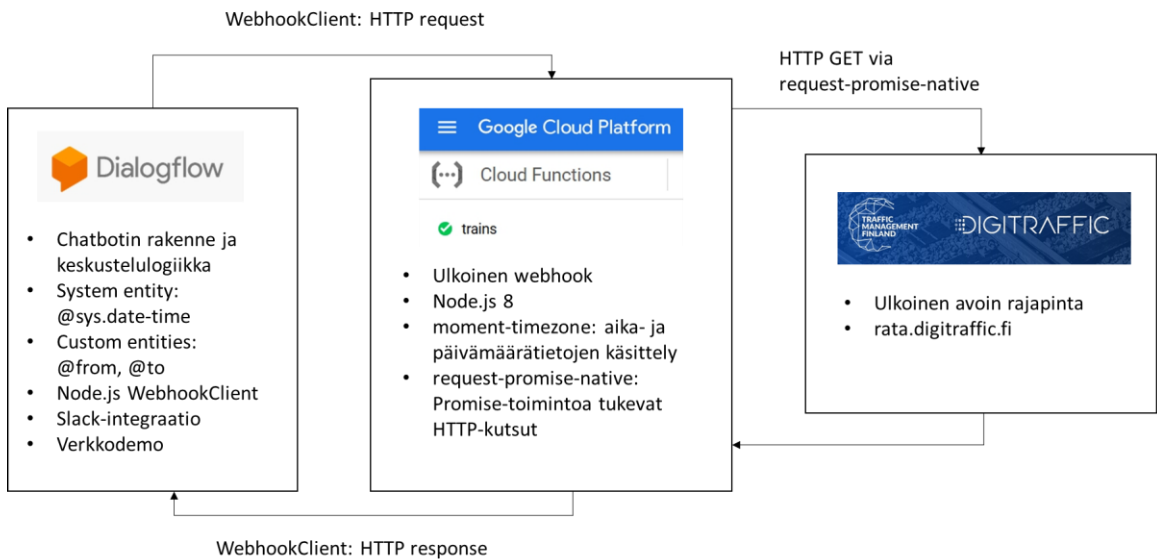
Uudelle agentille määritellään asetukset, kuten nimi ja kuvaus (Liite 2, Kuva 21). Asetuksissa **Service account** on yllä mainittu Google Cloud Platformsin tili, jonka avulla käyttäjä autentikoidaan. Autentikointiin on annettu tarkat ohjeet (Dialogflow 2019k). **API version** on uudelle Dialogflow-agentille aina V2. Asetussivun alareunassa kannattaa vielä panna lokiasetuksista päälle **Log interactions to Dialogflow** ja **Log interactions to Google Cloud**, jolloin lokitietoja voi käyttää monipuolisesti käytön seuraamiseen, testaukseen ja bugien etsintään (esimerkiksi Liite 1, Kuva 20).

Dialogflow'n **Fulfillment**-sivulla määritellään, miten agentti hakee käyttäjän haluamaa tietoa (Liite 2, Kuva 27). Tässä tapauksessa Dialogflow'n sisäinen koodieditori (**Inline Edi-**

tor) on otettava pois käytöstä (**Disabled**), koska käytössä on ulkoinen webhook (**Enabled**). Tässä webhookin URL on muuten sama kuin Google Cloud Functions -toiminnon URL (Liite 1, Kuva 18), mutta sen perään lisätään vielä */webhook*: *https://valittu-palvelin-kayttaja.cloudfunctions.net/toiminto/webhook*

5 Chatbotin kehittäminen

Kun yllä määritellyt toimenpiteet on tehty, ympäristö on valmis ulkoista webhookia käyttävän chatbotin kehittämistä varten. Kuva 10 esittää tiivistetysti valmiin chatbotin osat ja niiden keskinäiset yhteydet.



Kuva 10. Chatbot-sovelluksen rakenne.

5.1 Webhookin rakenne

Chatbotin käyttämä webhook pyörii siis Google Cloud Functionsissa. Ensin laadin botille alustavan rakenteen Dialogflow'n tarjoaman esimerkkipohjan perusteella (Skachov 2019). Pohja on laadittu Node.js-alustalla toimivaa webhookia varten ja sisältää *index.js*- ja *package.json* -tiedostot, joiden avulla Dialogflow saa yhteyden webhookiin. Jouduin kuitenkin muokkaamaan valmista pohjaa, koska se on rakennettu mobiilioptimoitua Firebase Cloud Functions -ympäristöä varten, kun taas minulla oli käytössä Google Cloud Functions. Lisäksi oli tietenkin tarpeen ohjelmoida kaikki ne toiminnot, jotka hakevat haluamani datan rautatieliikenteen ulkoisesta rajapinnasta. Tämä toiminnallisuus muodostaakin suurimman osan koodista.

Tämän opinnäytetyön Liite 3 sisältää kokonaan oman ratkaisuni *index.js*-tiedoston lähdekoodin selventävine kommentteineen ja Liite 4 sisältää *package.json*-tiedoston lähdekoodin. Oman JavaScript-koodini lisäksi hyödynsin webhookin toiminnassa tässä esiteltyjä ulkoisia ja kirjastoja ja komponentteja.

Request on Node.js-ympäristöön kehitetty yksinkertainen HTTP-kutsuja tekevä apuohjelma (Request 2019). Koodissani käytän kirjastoa nimeltä *request-promise-native*, joka tukee ES6-standardin mukaisia Promise-objekteja, jotka mahdollistavat asynkronisia toimintoja JavaScriptissä (Mozilla 2019). Asynkroninen toiminnallisuus on välttämätöntä, jotta voin varmistaa, että kaikki haluttu tieto on haettu ulkoisesta rajapinnasta ennen kuin webhook palauttaa vastauksen Dialogflow'lle.

Moment (Moment 2019) on JavaScriptille kehitetty kirjasto, joka helpottaa huomattavasti aika- ja päivämäärätiedon käsittelyä. Käytän koodissani kirjaston *moment-timezone*-versiota, joka tukee eri aikavyöhykkeitä.

WebhookClient (Dialogflow 2019l) on Dialogflow'n tarjoama Node.js-luokka, joka hoitaa yhteydet Dialogflow'n ja ulkoisen webhookin välillä. Koodissani on muuttuja nimeltä *agent*, joka on WebhookClient-luokan objekti. Jokaista Dialogflow'n aikomusta (intent) kohti on laadittu *index.js*-tiedostoon oma funktio, joka palauttaa Dialogflow'lle vastauksen käyttäjän repliikkiin. Merkittävin ja raskasrakenteisin funktio on *searchTrains(agent)*, joka ottaa vastaan Dialogflow'lta tulevat hakuparametrit, muokkaa niitä tarpeen mukaan, suorittaa varsinaisen tiedonhaun ulkoisesta rajapinnasta, muotoilee sopivan vastauksen ja lähettää sen takaisin Dialogflow'lle.

Koko lähdekoodi on liitetty mukaan opinnäytetyöhön (Liite 3), ja koodin kommenteista ilmenee, mitä kussakin vaiheessa tapahtuu ja miksi olen päätynyt tekemään tällaisia ratkaisuja. Kattavasta kommentoinnista on itsellenikin apua, kun lähden jatkossa kehittämään eteenpäin sekä tätä chatbottia että muita vastaavia botteja, jotka käyttävät hakevat tietoa eri rajapinnoista.

5.2 Dialogflow-chatbotin rakenne

Kuten Dialogflow'n yleisen esittelyn yhteydessä (kohta 4.1) käy ilmi, chatbotin tärkeimmät elementit ovat aikomukset (intents) ja entiteetit (entities). Rakentamassani junabotissa on käytössä Taulukossa 1 esitetyt aikomukset (Liite 2, Kuva 22) ja webhookin *index.js*-koodissa (Liite 3) niitä vastaavat funktiot. Liitteenä oleva vuokaavio (Liite 6,Kuva 35) puolestaan havainnollistaa chatbotin toimintaa ja aikomusten välisiä yhteyksiä.

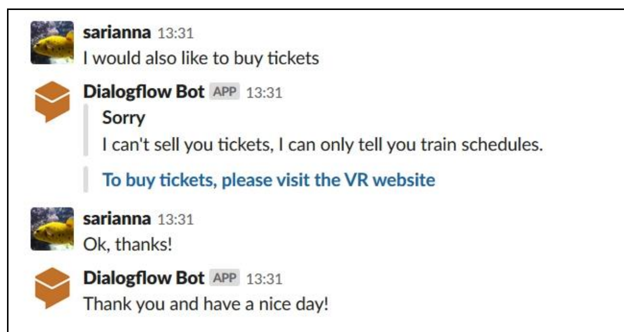
Taulukko 1. Chatbotin rakenne.

Aikomus Dialogflow'ssa	**Funktio webhookissa**	**Toiminta**
Buy Tickets (Kuva 12)	buyTickets(agent)	Jos käyttäjä haluaa ostaa junalippuja, hänet ohjataan tämän vas-

		tauksen kautta VR:n nettisivuille. Chatbotin kautta voi ainoastaan kysellä aikatauluja. Junatietojen avoin rajapinta ei salli minkäänlaista tietojen siirtoa siihen suuntaan, ainoastaan tiedon hakua, eli lippujen ostaminen rajapinnan kautta ei onnistu.
Default Welcome Intent (Kuva 11)	welcome(agent)	Toivottaa käyttäjän tervetulleeksi ja kertoo lyhyesti, mitä chatbotti osaa tehdä.
Default Fallback Intent	fallback(agent)	Oletusvastaus, joka ilmaisee, että botti ei ymmärrä käyttäjän antamaa syötettä. Vastauksessa kerrataan, mitä botti osaa tehdä.
End Conversation (Kuva 12)	endConversation(agent)	Määrämuotoinen lopetus, joka annetaan keskustelun päätteeksi, kun käyttäjä esimerkiksi kiittää tai hyvästelee.
List Stations	listStations(agent)	Antaa listan kaikista botin tunnistamista asemista.
Search New Date (Kuva 13; Liite 2: Kuva 25)	searchTrains(agent)	Käyttää samaa funktiota kuin Search Trains, mutta muistaa kontekstina edellisen haun *from*- ja *to*-parametrit yhden keskusteluvuoron ajan ja hakee aikataulut uudelleen vain uudella päivämäärätiedolla.
Search Trains (Kuva 13, Kuva 14, Kuva 15, Kuva 16; Liite 2: Kuva 23, Kuva 24)	searchTrains(agent)	Tärkein funktio, jossa tapahtuu chatbotin varsinainen toiminnallisuus. Funktio ottaa vastaan käyttäjän antamat hakuparametrit *from*, *to* ja *datetime* sekä muotoilee niistä *rata.digitraffic.fi*-APIn hyväksymän URL-osoitteen, hakee pyydetyt tiedot ja muotoilee ne käyttäjälle esitettävään muotoon.

Määrämuotoisten tekstien tapauksessa (esim. Kuva 11 ja Kuva 12) vastausta ei välttämättä tarvitse hakea webhookin kautta, vaan sen voi halutessaan määrittää tulemaan suoraan Dialogflow'n omasta käyttöliittymästä. Tässä chatbotissa olen kuitenkin ohjelmoinut kaikki vastaukset webhookiin, jotta kokonaisuus on selkeämpi hahmottaa samasta lähdekooditiedostosta. Webhookin saa halutessaan käyttöön tai pois käytöstä kunkin aikomuksen kohdalla asettamalla **Intent**-sivun alareunasta päälle **Enable webhook call for this intent** -vaihtoehdon (Liite 2, Kuva 24). Lisäksi **Action and parameters** -kenttään annetaan ulkoisessa webhookissa suoritettavan toiminnon nimi (Liite 2, Kuva 23).

Kuva 11. Welcome intent, määrämuotoinen vastaus.

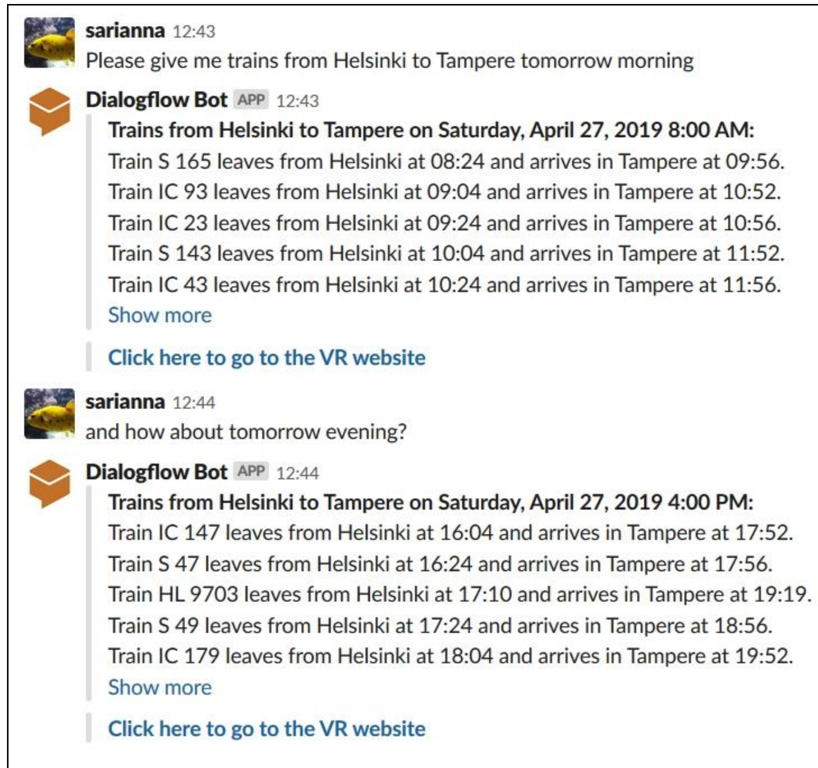


Kuva 12. Buy Tickets intent ja End Conversation intent, määrämuotoiset vastaukset.

Aikatauluhaussa on mahdollisuus hakea samojen asemien välisiä yhteyksiä kahtena eri ajankohtana siten, että chatbotti muistaa annetut hakuparametrit yhden keskusteluvuoron ajan. Tässä hyödynnetään Dialogflow'n konteksteja. Olisi toki mahdollista määrittää botti muistamaan parametrit useammankin keskusteluvuoron ajaksi, mutta tässä on vaarana se, että Dialogflow sekoittaa muistetut parametrit käyttäjän seuraavan haun uusiin parametreihin ja antaa vääriä vastauksia.

Esimerkiksi aikatauluja Helsingistä Tampereelle voi hakea aamulla ja illalla kuvan osoittamalla tavalla (Kuva 13). Ensimmäinen haku menee Search Trains -aikomuksen kautta ja

toinen Search New Daten kautta. Botti muistaa annetut parametrit ”Helsinki” (*from*) ja ”Tampere” (*to*) ja suorittaa toisen haun uuden ajankohdan mukaan (Liite 2, Kuva 25). Kuvassa näkyvät ajat antavat hyvän esimerkin luonnollisen kielen käsittelystä ja Dialogflow'n sisäisen *@sys.date-time*-entiteetin toiminnasta: ”tomorrow morning” tulkitaan automaattisesti hakuhetkeä seuraavan päivän aamuksi kello 8 ja ”tomorrow evening” iltapäiväksi kello 16.



Kuva 13. Search Train intent ja Search New Date intent.

Aikomusten lisäksi entiteetit ovat toinen tärkeä Dialogflow-chatbotin elementti. Bottini hyödyntää kolmea entiteettiä, jotka toimivat hakuparametreinä (Taulukko 2).

Taulukko 2. Chatbotin entiteetit ja parametrit.

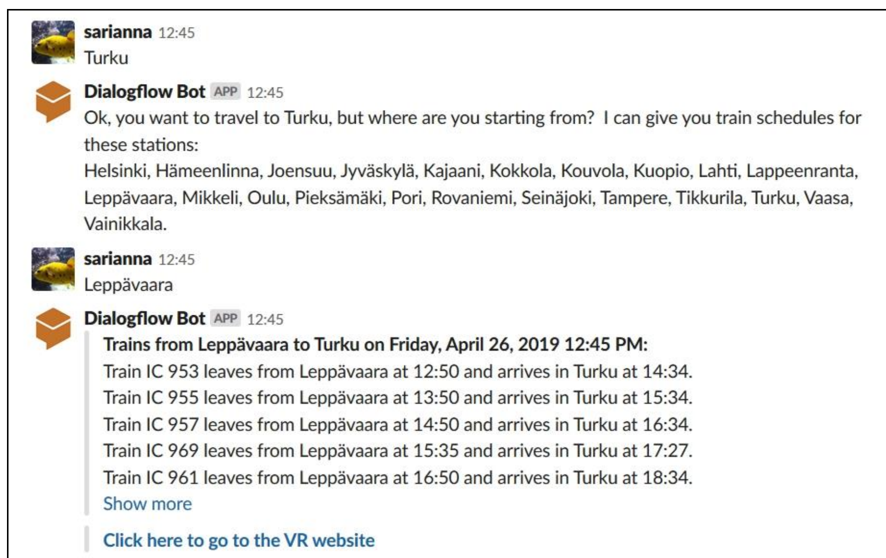
Entiteetti	**Parametri**	**Kuvaus**
@from	from	Lähtöasema
@to	to	Saapumisasema
@sys.date-time	datetime	Päivämäärä- ja/tai aikatieto

Dialogflow'n sisäänrakennettu *@sys.date-time*-entiteetti (Google 2019a) käsittelee käyttäjän antamaa päivämäärä- ja aikatietoa monipuolisesti joko merkkijonona (vain päivämäärä tai vain kellonaika) tai objektina (yhdistetty päivämäärä ja kellonaika). Dialogflow ymmär-

tää myös tavallisimmat luonnollisella kielellä (tässä siis englanniksi) esitetyt aikaan liittyvät ilmaukset. Esimerkiksi jos käyttäjä kirjoittaa ”Friday morning”, Dialogflow rakentaa tiedon perusteella objektin, jossa on hakuhetkeä seuraavan perjantain päivämäärä ja aikaväli 8:00-12:00.

Järjestelmän antaman entiteetin lisäksi tein Dialogflow'hun kaksi omaa entiteettiä nimeltä *@from* ja *@to* eli lähtö- ja saapumisasemien luettelot (Liite 2, Kuva 26). Ne ovat sisällöltään identtiset, mutta entiteettejä täytyy kuitenkin olla kaksi, jotta käyttäjän kyselyissä erotetaan kumpaan suuntaan junayhteyksiä haetaan (on eri asia hakea ”from Tampere to Helsinki” kuin ”from Helsinki to Tampere”). Asemien ja paikkatietojen valikointia ja nimeämistä selvitän tarkemmin alempana (kohta 5.3).

Kuva 14 esittää tapausta, jossa käyttäjä on antanut vain yhden hakuparametrin. Tällaisessa tapauksessa botti on koodattu ymmärtämään annettu arvo *to*-parametriksi sillä oletuksella, että käyttäjä antaa tiedoksi ensin sen paikan, jonne haluaa matkustaa. Tällöin Search Trains -aikomus ohjeistaa käyttäjää antamaan myös lähtöpaikan eli *from*-parametrin. Tämä tapahtuu määrittämällä **Intent**-sivulla **Action and parameters** -kohdassa kyseisen parametrin kohdalle kehote (prompt), joka näytetään käyttäjälle, jos parametri puuttuu syötteestä (Kuva 23). Koska lähtöaika on valinnainen parametri, sille ei voi määrittää kehotetta. Alla olevan kuvan tapauksessa haku suoritetaan siis kyselyhetken ajankohdan mukaan. Kuten ylempänä selvitetty, botti kuitenkin muistaa asemaparametrit yhden vuoron ajan, joten käyttäjä voi halutessaan suorittaa saman haun vielä toiselle ajankohdalle.

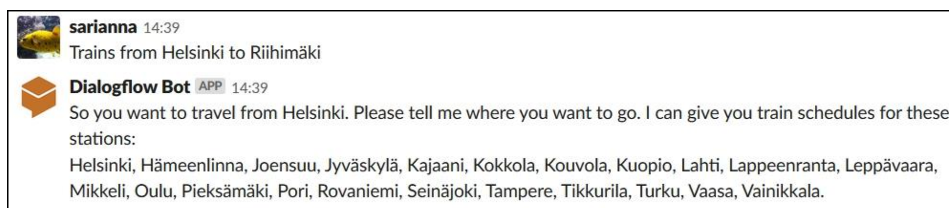


Kuva 14. Search Trains ja kehotteen käyttö hakuparametrien kyselyyn.

Kuva 15 esittää tilannetta, jossa käyttäjän antamien asemien välillä ei ole suoraa junayhteyttä. Määrämuotoinen vastaus on peräisin lähdekoodin (Liite 3) kohdasta ”Error handling 1”, eli *rata.digitraffic.fi*-rajapintaan on suoritettu haku annetuilla asemakoodeilla ja saatu vastaukseksi ”TRAIN_NOT_FOUND”.

Kuva 15. Search Trains intent: pyydettyä junayhteyttä ei löydy.

Kuva 16 puolestaan esittää tilannetta, jossa käyttäjän antamaa asemaa ei löydy botille määritellystä asemalistasta eli botti ei tässä tapauksessa ymmärrä syötettä, vaikka Riihimäki onkin aivan validi asema. Botti on siis kerännyt syötteestä *from*-parametrin (Helsinki) ja kehottaa nyt käyttäjää antamaan myös *to*-parametrin. Botin antama kehote on määritelty *to*-parametrin prompt-kentässä (Liite 2, Kuva 23).



Kuva 16. Search Trains intent, kehotteen käyttö, kun botti ei ymmärrä syötettä.

5.3 Juna-aikataulutietojen hakeminen

Rautatieliikenteen avoin rajapinta (*rata.digitraffic.fi*) palauttaa reittiperusteissa haussa ainoastaan suorat junayhteydet haettujen asemien välillä (Traffic Management Finland 2019b). Tästä syystä päädyin supistamaan demochatbotin käytössä olevien asemien määrän pariin kymmeneen isoimpaan tai merkittävimpään asemaan, jotka valitsin VR:n kaukoliikenteen rataverkkokartan perusteella (VR 2019). Koska rajapinta ymmärtää hakuehtona vain VR:n viralliset asemakoodilyhenteet eikä asemien tai kaupunkien koko nimiä, päädyin selkeyden vuoksi kovakoodaamaan asemat ja niiden koodit webhookin *stationsObj*-nimiseen muuttujaan, josta haetaan käyttäjän antama lähtö- tai saapumispaikan nimi (eli Dialogflow’sta tullut *from*- tai *to*-parametri) ja korvataan se vastaavalla asemakoodilla (Liite 3).

Muuttujien kovakoodaaminen ei tietenkään ole paras mahdollinen käytäntö, mutta mielestäni se on perusteltua tässä tapauksessa käyttäjäystävällisyyden vuoksi, kun kyseessä on demokäyttöön tarkoitettu chatbotti. Olisi toki mahdollista hakea rajapinnan kautta kaikki Suomen asemat, mutta suurimmalla osalla niistä ei ole suoria junayhteyksiä keskenään, joten käyttäjän haku palauttaisi useimmiten vain turhauttavan virheilmoituksen. Lisäksi etenkin suurimpien kaupunkien kohdalla on ongelma siinä, että käyttäjän oletettavasti antama kaupungin nimi ja aseman nimi eivät ole yksiselitteisiä. Esimerkiksi ”Turku” ei ole rajapinnan tunnistama aseman nimi, vaan asemia ovat ”Turku asema”, ”Turku satama” ja ”Turku tavara”, joilla jokaisella on omat koodinsa (Traffic Management Finland 2019d). Espoon tapauksessa puolestaan se asema, jolla kaukojunat pysähtyvät, onkin nimeltään ”Leppävaara” eikä ”Espoo”. Tällaiset tapaukset vaatisivat joka tapauksessa kustomoitua kovakoodausta, jotta botti ymmärtää palauttaa oikeat tiedot.

Seuraavaksi käsitellään Dialogflow’n lähettämää *datetime*-parametria eli päivämäärä- ja/tai aikatietoa, joka on siis järjestelmän oman *@sys.date-time*-entiteetin määrittämässä muodossa. Objektimuotoisista tiedoista poimitaan haluttu haun aloitushetki eli aikavälin alku. Esimerkiksi käyttäjän antama sana ”morning” on Dialogflow’n mielestä objektimuotoinen aikaväli 8:00-12:00, josta poimitaan hakuehtoon kellonaika 8:00. Rautatieliikenteen rajapinta antaa aikataulutiedot UTC0-vyöhykkeen mukaan, joten Dialogflow’sta tuleva tieto siirretään samalle vyöhykkeelle aikataulujen hakua varten.

Chatbottia testattaessa kävi ilmi, että Dialogflow’n Slack- ja web demo -integraatiot eivät osaa käsitellä Suomen kesäaikaa, vaan palauttavat kesäajan vallitessakin aikatiedon vyöhykkeellä +02:00 eikä +03:00, mistä aiheutuu tunnin heitto aikatauluhaussa. Tästä syystä jouduin lisäämään koodiin manuaalisesti kesäajan käsittelyn Dialogflow’n kautta tulevien päivämäärien osalta.

Jos käyttäjä ei ole antanut päivämäärää tai aikaa (se ei ole pakollinen hakuehto), oletushakuhetkenä käytetään nykyhetkeä, jota varten generoidaan uusi Date-objekti, joka muutetaan ISO-8601-muotoiseksi stringiksi eli muotoon "YYYY-MM-DDTHH:mm:ss.sssZ" (International Organization for Standardization 2019, w3schools.com 2019). Lopuksi vielä muokataan hakuajankohdan ilmaiseva ISO-stringi rajapinnan ymmärtämään muotoon.

Lähtö- ja saapumisasemien asemakoodit ja hakuaikastringi lisätään rajapinnan URL-osoitteeseen hakuparametreiksi, ja näin saadaan aikaan hakustringi, joka on rautatieliikenteen rajapinnan hyväksymää muotoa (ks. kohta 4.3). Demobotissa rajoitetaan palautettavien junien määrä kymmeneen (URL-hakuparametri *limit=10*), jotta tiedot mahtuvat jotenkin järkevästi chatbotin puhekuplaan.

Varsinainen HTTP-kutsu suoritetaan *request-promise-native*-kirjaston avulla (Request 2019). Rajapinta palauttaa JSON-muotoisen objektin, joka sisältää kokonaisuudessaan kaikki ne junat, jotka kulkevat haettujen asemien väliltä haettuna ajankohtana. Ensin varmistetaan, onko objektissa todella aikataulusisältöä vai eikö haettujen asemien välillä ole suoria junayhteyksiä. Jos rajapinta palauttaa "TRAIN_NOT_FOUND"-virheilmoituksen, Dialogflow'lle palautetaan virheilmoitus käyttäjälle näytettäväksi.

Seuraavaksi suodatetaan varmuuden vuoksi aikataulusta pois vielä muut kuin matkustajajunat. Tällaisia tapauksia ei luultavasti ole kovin paljoa, mutta testitilanteessa onnistuin sattumalta löytämään ainakin yhden tapauksen, jossa haettujen kaupunkien (Turku ja Kuopio) välillä oli ainoastaan suora tavarajunayhteys, mutta ei matkustajajunaa. Eli jos tämän suodatuksen jälkeen junia ei jää jäljelle, käyttäjälle palautetaan samoin virheilmoitus.

Lopuksi haetaan jäljelle jääneistä matkustajajunista junan tyyppi ja numero (*trainType* ja *trainNumber*). Jokaisen junan osalta käydään läpi *timeTableRows*-rivit ja etsitään tieto siitä, onko *stationShortCode* haettu asema, onko *type* "DEPARTURE" vai "ARRIVAL" ja mikä on *scheduledTime*, joka on siis junan lähtö- tai saapumisaika kyseiseltä asemalta (esim. Kuva 34). Ajat muutetaan *moment-timezonen* (Moment 2019) avulla Suomen aikavyöhykkeelle, jolloin ne täsmäävät oikeiden juna-aikataulujen kanssa (tämän voi tarkistaa tekemällä saman haun VR:n julkiselta nettisivulta, www.vr.fi).

Testikäytössä kävi ilmi, että rajapinta palauttaa joissain tapauksissa väärässä järjestyksessä olevia lähtöaikoja[5], vaikka sen palauttamien junien pitäisikin olla järjestetty JSONiin lähtöajan mukaan. Lisäsin koodiin vielä sorttausfunktion (*sortedArray*) joka järjestää tulokset lähtöajan mukaan oikeaan järjestykseen ennen Dialogflow'lle palauttamista.

5.4 Chatbotin kouluttaminen ja analytiikka

Dialogflow'n konsolin **Training**-osassa chatbottia voi kouluttaa käyttäjän antamien syötteiden perusteella (Liite 2, Kuva 28). Dialogflow käsittelee luonnollista kieltä koneoppimisprosessien kautta, joten chatbotin kanssa keskustelevien käyttäjien syöttämää relevanttia dataa voidaan käyttää opettamaan bottia toimimaan paremmin ja luotettavammin. Dia-

---

[5] Ainakin Tampere-Helsinki-välillä Tampereelta kello xx:02 ja xx:07 junat olivat taulukossa välillä jostain syystä keskenään väärässä järjestyksessä, eli esim. klo 12.07 lähtevä juna näytettiin ennen klo 12:02 lähtevää junaa.

logflow antaa myös mahdollisuuden ladata uuden botin käytettäväksi aiemmin kerättyä ulkoista dataa, esimerkiksi todellisia asiakkaiden repliikkejä yritysten asiakaspalvelulokeihin tallennetuista keskusteluista. Tämä on hyödyllinen ominaisuus esimerkiksi sellaisessa tapauksessa, jossa olemassa oleva ihmisvoimin hoidettu chattipalvelu halutaan siirtää ainakin osittain botin hoidettavaksi. (Dialogflow 2019m.)

Training-osiossa näkyy **Conversation**-sarakkeessa kunkin keskustelun ensimmäinen repliikki ja **Requests**-sarakkeessa käyttäjän repliikkien koko lukumäärä kyseisessä keskustelussa. **No match** -sarake ilmaisee, montako repliikkiä Dialogflow ei ole onnistunut yhdistämään sopivaan aikomukseen. Kun yhden keskustelun avaa, saa näkyviin kaikki repliikit, jotka voi joko hyväksyä (**Approve**) kerralla tai käydä läpi yksi kerrallaan ja hyväksyä, hylätä tai poistaa. Repliikkiä napsauttamalla avautuu näkymä, jossa voi manuaalisesti opettaa chatbottia yhdistämään tietyn käyttäjän antaman syötteen tiettyyn entiteettiin (Liite 2, Kuva 29) ja määrittää, mihin aikomukseen repliikki yhdistetään. (Dialogflow 2019m.)

Junabotin tapauksessa botti ei vaadi laajaa koulutusta, koska halutut parametrit ovat melko yksiselitteiset, vain asemien nimet ja aika/päivämäärätieto, jota Dialogflow osaa käsitellä. Käytin kyllä **Training**-osiota opettaakseni botille eroa ilmausten ”from Tampere” ja ”to Tampere” välillä, eli kumpaan parametriin kukin ilmaus tulee kytkeä, sekä täsmentämään moniosaisia ajan ilmauksia, jotka tulee ymmärtää *@sys.date-time*-entiteetteinä. Lisäksi opetin botille tavallisimpia englanninkielisiä tervehdys- ja hyvästelyfraaseja, jotta se osaa antaa oikealla hetkellä vastaukseksi Welcome- tai End Conversation -aikomuksen.

Chatbotin kouluttaminen tapahtuu Dialogflow’n koneoppimisalgoritmien kautta. Botti hyödyntää sekä siihen itseensä syötettyä koulutusdataa että Dialogflow’n omia sisäisiä mallinnuksia ja laatii näiden tietojen perusteella yksilöllisen algoritmin. Tämä algoritmi päättää, mikä aikomus yhdistetään mihinkin syötteeseen, ja sitä päivitetään aina, kun kehittäjä tekee chatbottiin muutoksia tai kun bottia koulutetaan uudella datalla. (Dialogflow 2019n.)

Kuva 30 esittää Dialogflow-konsolissa kehittäjän säädettävissä olevia koneoppimiseen liittyviä asetuksia. **Match mode** määrittää, mitä algoritmeja käytetään kaikkia niitä aikomuksia varten, joissa koneoppiminen on käytössä. Hybridimoodi käyttää sekä kieliopillisia sääntöjä että koneoppimista, kun taas ML-moodi käyttää pelkkää koneoppimista. **ML classification threshold** määrittää numeerisen arvon, joka mittaa millä varmuustasolla chatbot yhdistää syötteet aikomuksiin. Ääriarvot ovat 0.0 eli täysin epävarma ja 1.0 eli täysin varma. Varmuuskynnyksen alle jääviin syötteisiin annetaan vakiona vastaukseksi Fallback Intent eli oletusvirheilmoitus. **Automatic spell correction** antaa mahdollisuuden

hyväksyä myös kirjoitusvirheellisiä syötteitä, jotka Dialogflow pyrkii tulkitsemaan virheettömiksi. (Dialogflow 2019n.)

Dialogflow'n lähdekoodi ei ole avointa, eikä algoritmin rakentamisen tarkempaa toimintaa ei ole paljastettu, vaan se toimii ns. black box -tyyppisesti. Suoraan konsolissa testattaessa Dialogflow palauttaa kyllä jokaista syöterepliikkiä kohden JSON-muotoisen metatietopaketin (**Diagnostic info**; Liite 2, Kuva 31). Näistä tiedoista on poimittavissa numeerinen arvo nimeltä *intentDetectionConfidence* (Liite 2, Kuva 32), joka mittaa sitä, miten varma Dialogflow on siitä, että on valinnut juuri tämän syötteen kohdalla oikean aikomuksen. Tämä liittyy siis yllä mainittuun **ML classification threshold** -kynnysarvoon. Esimerkkikuvien tapauksessa varmuusarvo 0,87524074 on reilusti yli kynnysarvon 0,3. Oman botin toimintaa voi testailla erilaisilla syötteillä ja seurata tämän palautusarvon muutoksia, joista voi jossain määrin päätellä, mitä algoritmissa tapahtuu. (Mohanoor 2018b.)

Analytics-osiossa Dialogflow esittää dataa chatbotin käytöstä sekä numeerisesti että visuaalisesti vuokaavion muodossa (Liite 2, Kuva 33). Visuaalisen esityksen saa laajennettua yksityiskohtaisemmaksi, ja se antaa jokaisen aikomuksen kohdalla myös tiedon siitä, montako prosenttia käyttäjistä ja viesteistä päätyi kyseiseen aikomukseen asti tai lopetti botin käytön siinä. Näin voi helposti nähdä, jäävätkö käyttäjät jumiin tiettyihin kohtiin, missä on parantamisen varaa ja missä määrin botti toimii niin kuin on suunniteltu. **Analytics**-osiossa voi myös vertailla esimerkiksi käyttäjämääriä ja kyselyjen tiheyttä nykyhetken ja jonkun aiemman ajankohdan välillä. (Dialogflow 2019o.)

Tässä kehitetyn chatbotin tapauksessa analytiikka ei ole vielä kovin hyödyllistä, koska botti on ollut enimmäkseen vain omassa testikäytössäni, joten kerätty data ei edusta todellisten, ulkopuolisten käyttäjien repliikkejä. Toiminto on kuitenkin mielenkiintoinen ja varmasti erittäin käyttökelpoinen silloin, kun halutaan esimerkiksi siirtää yrityksen asiakaspalvelua osittain chatbotin hoidettavaksi ja seurata, miten asiakkaat ottavat muutoksen vastaan.

6 Pohdinta

Tämän opinnäytetyön tekeminen oli minulle kaiken kaikkiaan melkoinen oppimisprosessi, koska JavaScriptin perusteita lukuun ottamatta lähes kaikki käytetyt tekniikat ja työkalut (Dialogflow, Google Cloud Functions, Node.js ja sen moduulit) olivat minulle uusia, samoin kuin chatbottien tarkempi toimintalogiikka. Toimeksiantajayritykseltä sain tukea toimintaympäristön rakentamisessa eli Dialogflow'n ja Google Cloud Platformsin yhdistämisessä. Varsinaisen chatbotin kehitystyön tein kuitenkin itsenäisesti. Pystyin käyttämään jonkin verran tukena Dialogflow'n tarjoamia esimerkkipohjia (kuten mainittu kohdassa 5.1), mutta chatbotin varsinainen bisneslogiikka pitää jokaisen kehittäjän kuitenkin laatia itse, koska se on vahvasti riippuvainen käytetystä rajapinnasta.

Rakentamani chatbotti toimii annettujen rajoitusten puitteissa siten kuin sen kuuluukin eli hakee juna-aikataulutietoja ja palauttaa ne käyttäjälle. Siinä mielessä opinnäytetyön tavoite siis toteutui. Toimintaa voisi tietenkin vielä hienosäätää, ja ehkä vielä aikataulun salliessa teenkin niin, mutta opinnäytetyötä varten valmistui nyt tässä esitelty versio. Mitä enemmän perehdyin työn kuluessa chatbottien toimintaan, sitä paremmin hahmottui kokonaiskäsitykseni siitä, miten rakentaa selkeä ja loogisesti toimiva botti. Tässä olivat suurena apuna Dialogflow'n omat tutoriaalit ja opetusvideot, Stack Overflow'n keskustelut sekä asiantuntevat blogit (mm. Mohanoor 2019b). Myös Noden omatoimiseen opetteluun löytyy paljon hyödyllistä materiaalia (mm. Hamedani 2018; Kiessling 2017).

Jos lähtisin nyt rakentamaan bottia alusta alkaen uudestaan, voisi olla helpompi hakea käyttäjältä tiedot yksi kerrallaan erillisinä keskusteluvuoroina, esimerkiksi kysyä ensin lähtöasema, sitten saapumisasema ja viimeiseksi mahdollinen lähtöajankohta, jos se on eri kuin nykyhetki. Näin olisi helpompi käsitellä käyttäjän antamaa syötettä, jos joka vaiheessa tulisi vain yksi parametri kerrallaan. Toisaalta tällöin täytyy huolehtia kontekstien avulla siitä, että kaikki pyydetyt parametrit pysyvät ”muistissa” siihen asti, kunnes pyyntö kokonaisuudessaan saadaan lähetettyä ulkoiselle rajapinnalle. Voisin myös opettaa botin tunnistamaan kaikki suomalaiset rautatieasemat (joita on 541 kappaletta; Traffic Management Finland 2019e). Vaikka kaikkia asemia ei käytettäisikään hakuparametreinä, chatbotin olisi silti hyvä tuntea ne, jottei käyttäjä ihmettele, miksei botti tunnista jotain sellaista aseman nimeä, jonka käyttäjä tietää validiksi (esim. Riihimäki, Kuva 16).

Ulkoisen rajapinnan ominaisuudet vaikuttavat suuresti siihen, miten monivaiheista käsittelyä ulkoisen webhookin kautta haettu vastaus vaatii ennen Dialogflow'hun palauttamista. Rautatieliikenteen rajapinnan suhteen työn edetessä kävi ilmi, että vaikka alkuvaatimukset vaikuttivatkin yksinkertaisilta (lähtöasema, saapumisasema ja lähtöaika), haun toteuttami-

nen Dialogflow'n ja rajapinnan yhteistyönä ei ollutkaan ihan niin yksinkertaista. Ensinnäkin käyttäjän antaman kaupungin nimen yhdistäminen oikeaan asemakoodiin oli sen verran hankalasti toteutettavissa, että päädyin lopulta kovakoodaamaan käytetyt asemakoodit (katso kohta 5.3). Tosin nyt kun olen perehtynyt Dialogflow'hun tarkemmin, voisin toteuttaa haun myös niin, että varsinainen entiteetti ja siitä johdettu parametri olisikin suoraan asemakoodi ja aseman nimi toimisi vain synonyyminä. Tällöin asemakoodin kovakoodaus webhookiin ei välttämättä olisi tarpeen. Monimerkityksisten asemanimien ongelma tosin pysyy tällöinkin ennallaan.

Myös aika- ja päivämäärätiedon käsittely Dialogflow'sta rajapintaan ja takaisin oli monivaiheinen prosessi, jossa oli huomioitava Dialogflow'sta tulevan *datetime*-tiedon muoto, aikavyöhykkeen muutos, kesä- ja talviajan vaihtelu sekä käsiteltävien stringien ja objektien muotoilu. Dialogflow'n integraatiot Slackiin ja verkkodemoympäristöön asettivat omat haasteensa parametrien käsittelyn suhteen.

Lisäksi testaillessani koodia kävi ilmi, että vikatilanteissa koodin Promise-rakenteen virheiden hallinnassa on puutteita, jos esimerkiksi rajapintaan ei saada yhteyttä. Käytännön toiminnassa nämä vikatilanteet peittyvät kuitenkin sen alle, että Dialogflow katkaisee yhteyden ulkoiseen webhookiin automaattisesti viiden sekunnin kuluttua, jos vastausta ei siihen mennessä tule. Tällöin käyttäjälle palautetaan kyseiselle aikomukselle Dialogflow-konsolissa määritelty virheilmoitus (Liite 2, Kuva 24, Text response).

Dialogflow on määritellyt viiden sekunnin rajoituksen (request timeout) kaikille ulkoisille webhookeille, eikä chatbotin kehittäjä voi itse vaikuttaa sen pituuteen (Dialogflow 2019i). Tämä aikarajoitus vaikuttaa myös siihen, miten monivaiheista koodia voi suorittaa webhookissa ennen vastauksen palauttamista Dialogflow'lle. Jos rajapinnasta palautettu JSON vaatii pitkällistä läpikäyntiä ja muokkaamista ennen kuin siitä saadaan irti käyttäjän haluama tieto, voi olla vaarana, että Dialogflow ehtii katkaista yhteyden ennen kuin tieto on saatu kaivettua esiin. Esimerkiksi Stack Overflow'ssa löytyy erilaisten chatbottien kehittäjien kysymyksiä ja ratkaisupyyntöjä, jotka liittyvät tähän timeout-ongelmaan.

Kokonaisuudessaan opinnäytetyöprojekti oli mielenkiintoinen ja haastava, ja koen oppineeni sen aikana paljon uutta, jota voin toivoakseni käyttää hyödykseni myös jatkossa työssäni. Etenkin perehtyminen JavaScriptin uusimpiin ominaisuuksiin ja Node.js-ympäristöön on varmasti hyödyksi monenlaisissa tehtävissä, ei pelkästään chatbottien kehittämisessä. Samoin myös Google Cloud Functions -toimintoihin tutustuminen chatbotin webhookin rakentamisen yhteydessä helpottaa vastaavien pilvitoimintojen käyttöönottoa myös muihin tarkoituksiin. Toivon myös, että tämän yhden chatbotin pohjalta on hel-

pompi kehittää muita vastaavia työkaluja, ja että tässä opinnäytetyössä esitetyistä ohjeista Google Cloud Platformsin ja Dialogflow'n yhteensovittamiseen on ehkä apua jatkossakin. Tältäkin osin koen siis työn tavoitteiden toteutuneen, vaikka kuten ohjelmistokehityksessä yleensäkin, tuntuu siltä, että vähän kun pääsee pintaa syvemmälle tutustumaan johonkin aihealueeseen, avautuukin valtavan ulapan täydeltä uutta ja läheltä liippaavaa asiaa, johon olisi vielä kiinnostavaa perehtyä.

Lähteet

Auger, A. & Barrière, C. 2010. Probing semantic relations: Exploration and identification in specialized texts. John Benjamins Publishing Company. Amsterdam/Philadelphia.

Bresnick, J. 2016. What Is the Role of Natural Language Processing in Healthcare? Luettavissa: https://healthitanalytics.com/features/what-is-the-role-of-natural-language-processing-in-healthcare. Luettu: 6.5.2019.

Brink, H., Richards, J.W. & Fetherolf, M. 2017. Real-World Machine Learning. Manning. Shelter Island.

Britannica 2019. Turing test. Luettavissa: https://www.britannica.com/technology/Turing-test. Luettu: 17.4.2019.

Chandrayan, P. 2017. Artificial Intelligence: Natural Language Processing Fundamentals. Luettavissa: http://www.techprenuer.com/entrepreneur/artificial-intelligence-natural-language-processing-fundamentals/. Luettu: 5.5.2019. Kuvan alkuperäinen lähde: Wang. H. 2016: CS5601-Text-Mining. https://github.com/wang296/CS6501-Text-Mining/blob/master/lectures.md.

Dale, R. 2000. Symbolic Approaches to Natural Language processing. Sivut 1-9 teoksessa Dale, R., Moisl, H. & Somers, H (eds.) 2000: Handbook of Natural Language Processing. Marcel Dekker, Inc. New York.

Dialogflow 2019a. Overview. Luettavissa: https://dialogflow.com/. Luettu: 16.4.2019.

Dialogflow 2019b. Intents overview. Luettavissa: https://dialogflow.com/docs/intents. Luettu: 16.4.2019.

Dialogflow 2019c: Agents overview. Luettavissa: https://dialogflow.com/docs/agents/. Luettu: 4.5.2019.

Dialogflow 2019d. Entities overview. Luettavissa: https://dialogflow.com/docs/entities. Luettu: 24.4.2019.

Dialogflow 2019e. Contexts overview. Luettavissa: https://dialogflow.com/docs/contexts. Luettu: 24.4.2019.

Dialogflow 2019f. Events overview. Luettavissa: https://dialogflow.com/docs/events. Luettu: 24.4.2019.

Dialogflow 2019g. Languages. Luettavissa: https://cloud.google.com/dialogflow-enterprise/docs/reference/language. Luettu: 16.4.2019.

Dialogflow 2019h. Integrations. Luettavissa: https://dialogflow.com/docs/integrations. Luettu: 23.4.2019.

Dialogflow 2019i. How fulfillment works. Luettavissa: https://dialogflow.com/docs/fulfillment/how-it-works. Luettu: 23.4.2019.

Dialogflow 2019j. Getting Started. Luettavissa: https://dialogflow.com/docs/getting-started. Luettu: 16.4.2019.

Dialogflow 2019k. Setting up authentication. Luettavissa: https://dialogflow.com/docs/reference/v2-auth-setup. Luettu: 16.4.2019.

Dialogflow 2019l. WebhookClient. Luettavissa: https://dialogflow.com/docs/reference/fulfillment-library/webhook-client. Luettu: 24.4.2019.

Dialogflow 2019m. Training. Luettavissa: https://dialogflow.com/docs/training-analytics/training. Luettu: 25.4.2019.

Dialogflow 2019n. Machine learning. Luettavissa: https://dialogflow.com/docs/agents/machine-learning. Luettu: 5.5.2019.

Dialogflow 2019o. Analytics. Luettavissa: https://dialogflow.com/docs/training-analytics/analytics. Luettu: 25.4.2019.

Dialogflow 2019p: Create fulfillment using webhook. Luettavissa: https://dialogflow.com/docs/tutorial-build-an-agent/create-fulfillment-using-webhook. Luettu: 4.5.2019.

Elharrar, D. 2017. 7 Types of Bots. Different ways to deliver value. Luettavissa: https://chatbotsmagazine.com/7-types-of-bots-8e1846535698. Luettu: 17.4.2019.

Eranti, V. & Ylä-Anttila, T. 2017. Yhteiskunnan mittaaminen: Big data ja tiedonlouhinta. Julkaistu SlideSharessa 13.12.2017. Luettavissa: https://www.slideshare.net/tuylaant/yhteiskunnan-mittaaminen-big-data-ja-tiedonlouhinta. Luettu: 17.4.2019.

Faggella, D. 2018. Swedish Bank Uses Natural Language Processing for Virtual Customer Assistance. Luettavissa: https://emerj.com/ai-case-studies/swedish-bank-uses-natural-language-processing-virtual-customer-assistance/. Luettu: 6.5.2019.

Google 2018. Building a Chatbot with Dialogflow and Google Cloud Platform. Video: https://www.youtube.com/watch?v=5r4AAIfe4Rw. Katsottu: 12.3.2019.

Google 2019a. System entities. Luettavissa: https://cloud.google.com/dialogflow-enterprise/docs/reference/system-entities. Luettu: 24.4.2019.

Google 2019b. Cloud Functions Overview. Luettavissa: https://cloud.google.com/functions/docs/concepts/overview. Luettu: 23.4.2019.

Google 2019c. Google Cloud Functions: Event-driven serverless compute platform. Luettavissa: https://cloud.google.com/functions/. Luettu: 5.5.2019.

Google 2019d. Pricing. Luettavissa: https://cloud.google.com/functions/pricing. Luettu: 16.4.2019.

Google 2019e. Getting Started with Google Cloud Platform. Luettavissa: https://cloud.google.com/gcp/getting-started/. Luettu: 16.4.2019.

Haikonen, P.O.A. 2017. Tietoisuus, tekoäly ja robotit. Art House. Helsinki.

Hamedani, M. 2018. Node.js Tutorial for Beginners. Learn Node in 1 Hour. Video. Katsottavissa: https://www.youtube.com/watch?v=TlB_eWDSMt4. Katsottu: 12.3.2019.

Heidorn, G. E. 2000: Intelligent Writing Assistance. Sivut 181-207 teoksessa Dale, R., Moisl, H. & Somers, H (eds.) 2000: Handbook of Natural Language Processing. Marcel Dekker, Inc. New York.

Hupli, M. 2018. Chatbot FAQ – kaikki mitä chatboteista on syytä tietää juuri nyt. Luettavissa: https://www.salesforce.com/fi/blog/2018/chatbot-usein-kysytyt-kysymykset.html. Luettu: 17.4.2019.

International Organization for Standardization 2019. Date and time format - ISO 8601. Luettavissa: https://www.iso.org/iso-8601-date-and-time-format.html. Luettu: 5.5.2019.

Janarthanam, S. 2017. Hands-on chatbots and conversational UI development: build chatbots and voice user interfaces with Chatfuel, Dialogflow, Microsoft Bot Framework, Twilio, and Alexa Skills. Packt Publishing. E-kirja.

Khan, R. & Das, A. 2018. Build Better Chatbots. Apress. E-kirja.

Kiessling, M. 2017. The Node Beginner Book. Luettavissa: https://www.nodebeginner.org/. Luettu: 10.3.2019.

Knight, W. 2019. The man who helped invent virtual assistants thinks they're doomed without a new AI approach. Luettavissa: https://www.technologyreview.com/s/612826/virtual-assistants-thinks-theyre-doomed-without-a-new-ai-approach/. Luettu: 6.5.2019.

Lyons, J. 1991. Natural Language and Universal Grammar. Cambridge University Press. New York.

Medelyan, A. 2016. 8 natural language processing (NLP) examples you use every day. Luettavissa: https://getthematic.com/insights/8-natural-language-processing-nlp-examples-you-use-every-day-without-noticing/. Luettu: 17.4.2019.

Mohanoor, A. 2018a. Can you use Dialogflow without writing code? Luettavissa: https://miningbusinessdata.com/can-use-dialogflow-without-writing-code/. Luettu: 5.5.2019.

Mohanoor, A. 2018b. Dialogflow Machine Learning Algorithm. Luettavissa: https://miningbusinessdata.com/dialogflow-machine-learning-algorithm/. Luettu: 5.5.2019.

Mohanoor, A. 2019a. The 4 layers of a Dialogflow bot. Luettavissa: https://miningbusinessdata.com/the-4-layers-of-a-dialogflow-bot/. Luettu: 4.5.2019.

Mohanoor, A. 2019b. Mining Business Data. Blogi. Luettavissa: https://miningbusinessdata.com/. Luettu: 2.5.2019.

Moment 2019. Moment.js. Parse, validate, manipulate, and display dates and times in JavaScript. Luettavissa: https://momentjs.com/. Luettu: 24.4.2019.

Mozilla 2019. Using promises. Luettavissa: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises. Luettu: 24.4.2019.

Murphy, M.L. 2003. Semantic Relations and the Lexicon: Antonymy, Synonymy, and Other Paradigms. Cambridge University Press. New York.

Quinlan, N. 2014. Whats' a Webhook? Luettavissa: https://sendgrid.com/blog/whats-webhook/. Luettu: 23.4.2019.

Radford, A., Wu, J., Amodei, D., Amodei, D. Clark, J., Brundage, M. & Sutskever, I. 2019. Better Language Models and Their Implications. Luettavissa: https://openai.com/blog/better-language-models/. Luettu: 6.5.2019.

Request 2019. Request-Promise. Luettavissa: https://www.npmjs.com/package/request-promise. Luettu: 24.4.2019.

Rizvi, M. 2018. A Guide to Building an Intelligent Chatbot for Slack using Dialogflow API. Luettavissa: https://www.analyticsvidhya.com/blog/2018/03/how-to-build-an-intelligent-chatbot-for-slack-using-dialogflow-api/. Luettu: 5.5.2019.

SAS 2019. Natural Language Processing: What it is and why it matters. Luettavissa: https://www.sas.com/en_us/insights/analytics/what-is-natural-language-processing-nlp.html. Luettu: 17.4.2019.

Schofield, J. 2014. Computer chatbot 'Eugene Goostman' passes the Turing test. Luettavissa: https://www.zdnet.com/article/computer-chatbot-eugene-goostman-passes-the-turing-test/. Luettu: 17.4.2019.

Sheth, R. 2017. India: the Next Frontier in Language Services. Luettavissa: https://slator.com/features/india-next-frontier-language-services/. Luettu: 6.5.2019.

Skachov, D. 2019. Dialogflow Fulfillment Webhook Template for Node.js and Cloud Functions for Firebase. Luettavissa: https://github.com/dialogflow/fulfillment-webhook-nodejs/blob/master/functions. Luettu: 24.4.2019.

Tieteen termipankki 2019a. Kielitiede:luonnollinen kieli. Luettavissa: https://tieteentermipankki.fi/wiki/Kielitiede:luonnollinen kieli. Luettu: 17.4.2019.

Tieteen termipankki 2019b. Language Technology:tokenise. Luettavissa: https://tieteentermipankki.fi/wiki/Language Technology:tokenise. Luettu: 17.4.2019.

Tieteen termipankki 2019c. Nimitys:parsing. Luettavissa: https://www.tieteentermipankki.fi/wiki/Nimitys:parsing. Luettu: 17.4.2019.

Tieteen termipankki 2019d. Nimitys:part of speech tag. Luettavissa: https://www.tieteentermipankki.fi/wiki/Nimitys:part of speech tag. Luettu: 17.4.2019.

Tractica 2017. Natural Language Processing Market to Reach $22.3 Billion by 2025. Luettavissa: https://www.tractica.com/newsroom/press-releases/natural-language-processing-market-to-reach-22-3-billion-by-2025/. Luettu: 6.5.2019.

Traffic Management Finland 2019a. Rautatieliikenne: Junien aikataulut, toteumatiedot, sijainnit ja kokoonpanot. Luettavissa: https://www.digitraffic.fi/rautatieliikenne/. Luettu: 15.4.2019.

Traffic Management Finland 2019b. Reittiperusteinen haku. Luettavissa: https://www.digitraffic.fi/rautatieliikenne/#reittiperusteinen-haku. Luettu: 24.4.2019.

Traffic Management Finland 2019c. Junat. Luettavissa: https://www.digitraffic.fi/rautatieliikenne/#junat. Luettu: 24.4.2019.

Traffic Management Finland 2019d. Metatiedot. Luettavissa: https://www.digitraffic.fi/rautatieliikenne/#metatiedot-metadata. Luettu: 24.4.2019.

Traffic Management Finland 2019e. Stations. Luettavissa: https://rata.digitraffic.fi/api/v1/metadata/stations. Luettu: 5.5.2019.

VR 2019. Kaukoliikenteen rataverkko ja tärkeimmät rautatieasemat. Luettavissa: https://www.vr.fi/cs/vr/fi/kaukoliikenteen-reittikartta. Luettu: 24.4.2019.

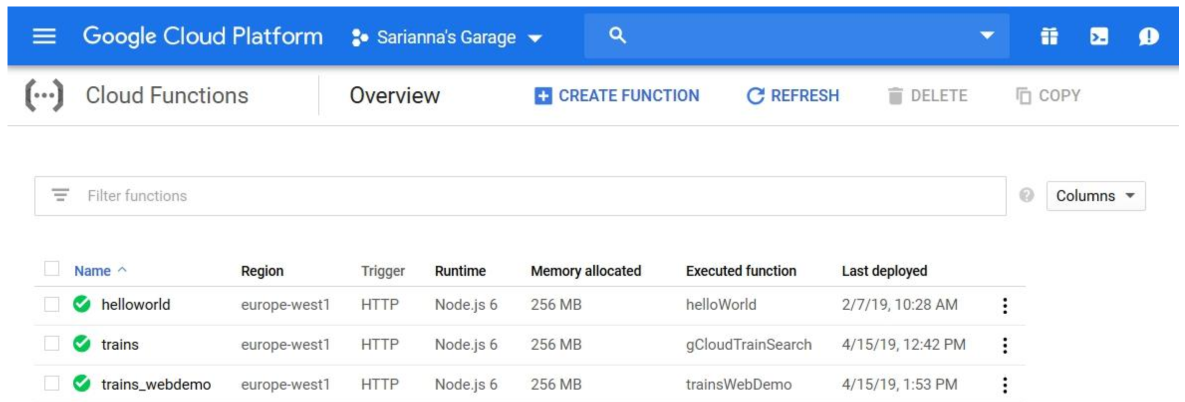
W3schools.com 2019. JavaScript toISOString() Method. Luettavissa: https://www.w3schools.com/jsref/jsref_toisostring.asp. Luettu: 5.5.2019.

Weizenbaum, J. 1966. ELIZA--A Computer Program For the Study of Natural Language Communication Between Man and Machine. Communications of the ACM. Volume 9, Number 1 (January 1966): 36-35.

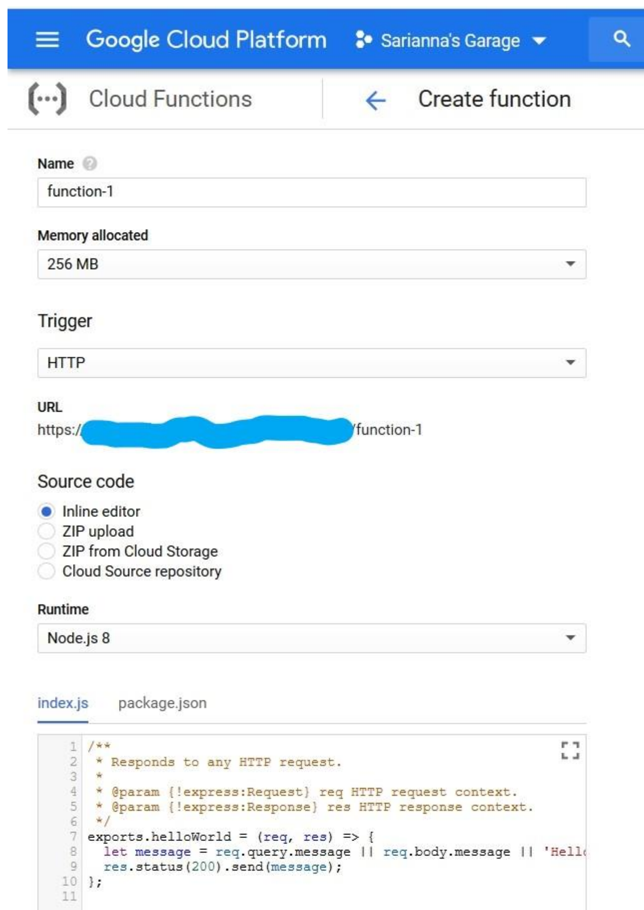
Liitteet

Liite 1. Kuvakaappaukset: Google Cloud Functions

Liitteen kuvakaappaukset selventävät pilvitoiminnon käyttöönottoa Google Cloud Functions -toimintaympäristössä.



Kuva 17. Luotujen toimintojen lista Google Cloud Functions -alustalla.



Kuva 18. Cloud Function -toiminnon asetukset, näytön yläosa.

Google Cloud Platform   Sarianna's Garage

Cloud Functions   ← Create function

```
/**
 * Responds to any HTTP request.
 *
 * @param {!express:Request} req HTTP request context.
 * @param {!express:Response} res HTTP response context.
 */
exports.helloWorld = (req, res) => {
  let message = req.query.message || req.body.message || 'Hell
  res.status(200).send(message);
};
```

Function to execute

helloWorld

Advanced options

Region

us-central1

VPC connector

Fully-qualified resource name

Timeout

60 seconds

Maximum function instances

Service account

Dialogflow Integrations

Environment variables

+ Add variable

Hide

Create   Cancel

Kuva 19. Cloud Function -toiminnon asetukset, näytön alaosa.

CREATE METRIC CREATE EXPORT

Filter by label or text search

Cloud Function, trains, europe-west1 | All logs | Any log level | Last hour | Jump to now

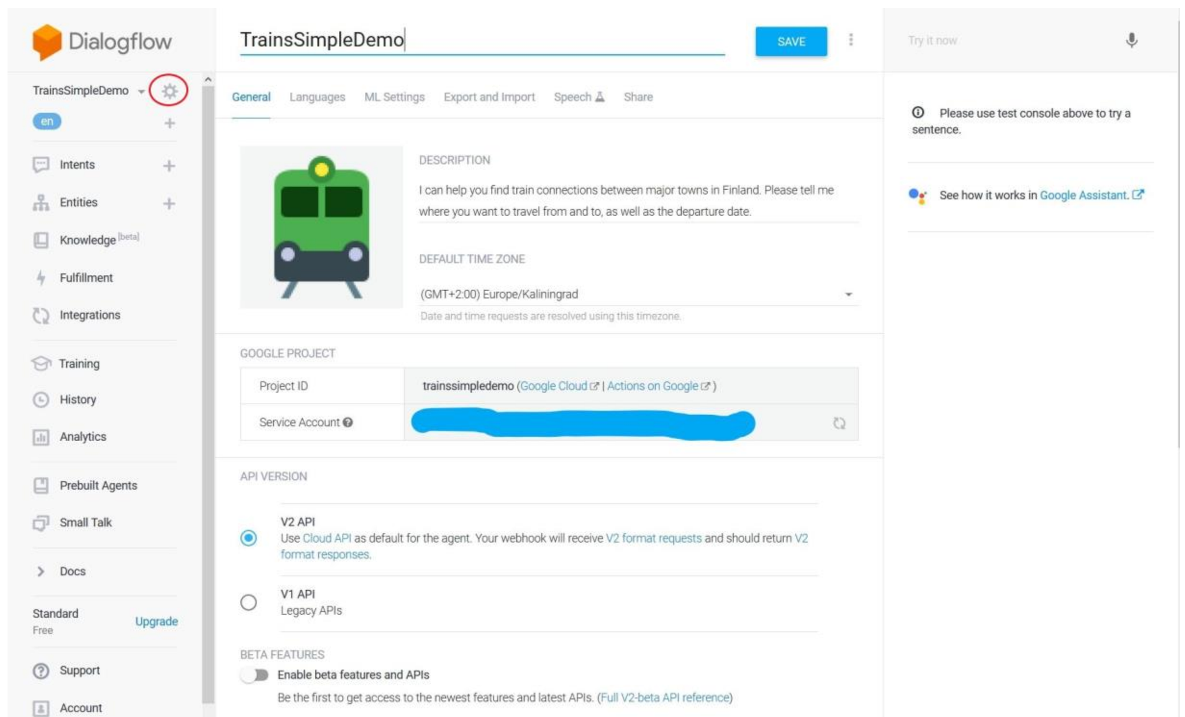
Showing logs from **all time** (EEST) **Download logs** **View Options**

```
2019-04-26 12:52:12.730 EEST trains 3z88sz6h0sw0 Train IC 144 leaves from Tampere at 15:07
2019-04-26 12:52:12.730 EEST trains 3z88sz6h0sw0 and arrives in Helsinki at 16:19.
2019-04-26 12:52:12.730 EEST trains 3z88sz6h0sw0 Train HL 9682 leaves from Tampere at 14:07
2019-04-26 12:52:12.730 EEST trains 3z88sz6h0sw0 and arrives in Helsinki at 15:35.
2019-04-26 12:52:12.730 EEST trains 3z88sz6h0sw0 Train S 178 leaves from Tampere at 14:02
2019-04-26 12:52:12.730 EEST trains 3z88sz6h0sw0 and arrives in Helsinki at 14:54.
2019-04-26 12:52:12.729 EEST trains 3z88sz6h0sw0 Train IC 174 leaves from Tampere at 13:07
2019-04-26 12:52:12.729 EEST trains 3z88sz6h0sw0 and arrives in Helsinki at 14:35.
2019-04-26 12:52:12.729 EEST trains 3z88sz6h0sw0 Train IC 22 leaves from Tampere at 13:02
2019-04-26 12:52:12.651 EEST trains 3z88sz6h0sw0 These trains were returned: [{"trainNumber":22,"departureD…
2019-04-26 12:52:12.058 EEST trains 3z88sz6h0sw0 The URL is https://rata.digitraffic.fi/api/v1/live-trains/…
2019-04-26 12:52:12.058 EEST trains 3z88sz6h0sw0 depDateTime in UTC is 2019-04-26T09:52:12Z
2019-04-26 12:52:12.053 EEST trains 3z88sz6h0sw0 depDateTime as ISOString is 2019-04-26T09:52:12.053Z
2019-04-26 12:52:12.053 EEST trains 3z88sz6h0sw0 Given From and To parameters: from Tampere to Helsinki
2019-04-26 12:52:12.053 EEST trains 3z88sz6h0sw0 Dialogflow Request body: {"responseId":"10314d5f-f276-4f66…
2019-04-26 12:52:12.051 EEST trains 3z88sz6h0sw0 Dialogflow Request headers: {"host":"europe-west1-sarianna…
2019-04-26 12:52:12.039 EEST trains 3z88sz6h0sw0 Function execution started
```

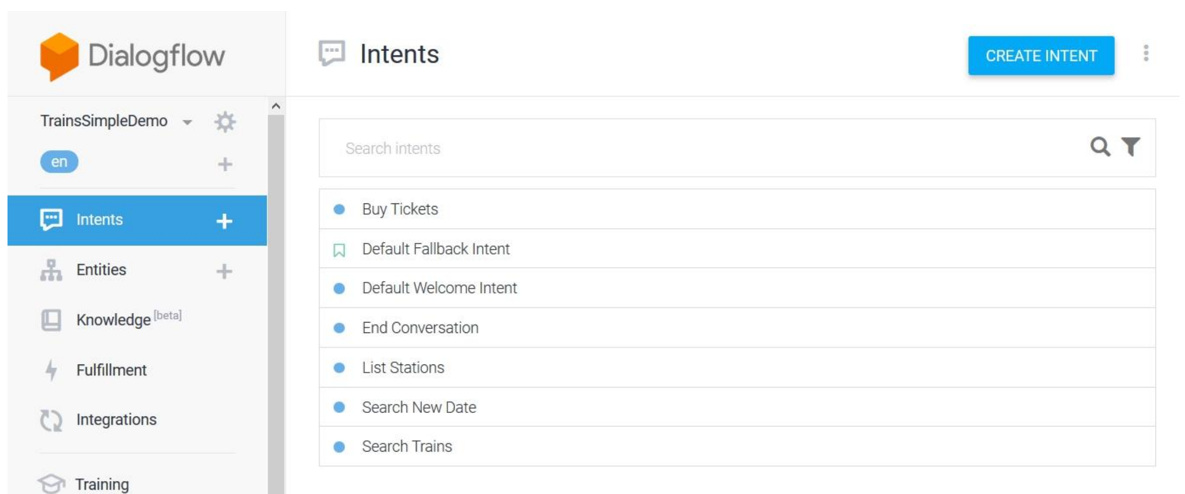
Kuva 20. Google Cloud Platform: Ote pilvitoiminnon Stackdriver-lokitiedoista.

Liite 2. Kuvakaappaukset: Dialogflow

Liitteen kuvakaappaukset selventävät Dialogflow-agentin eri komponenttien rakennetta ja asetuksia.

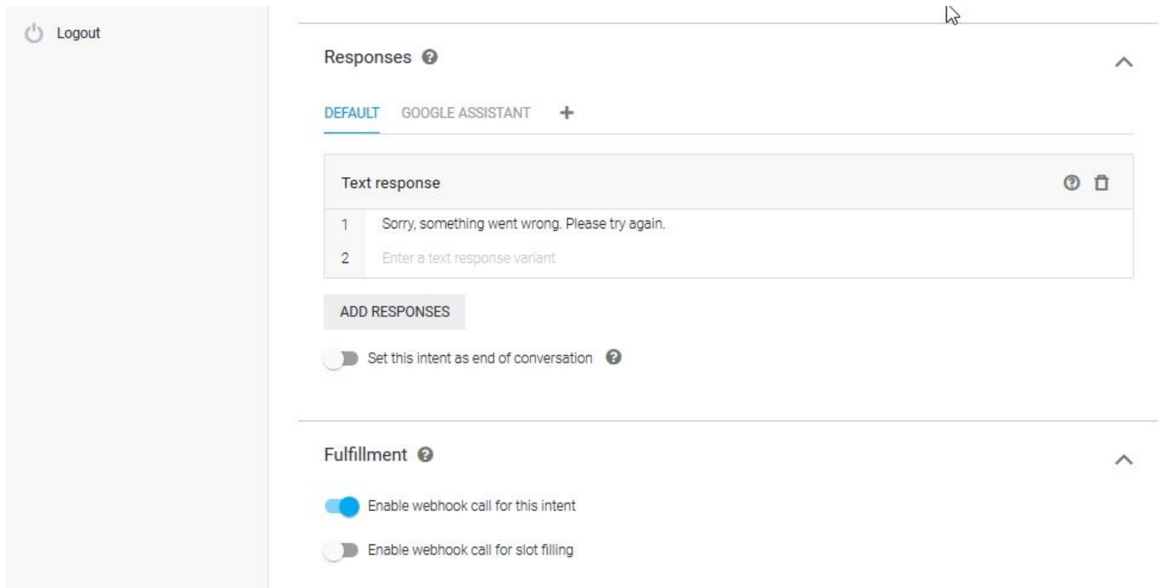


Kuva 21. Koko agentin asetukset, joihin pääsee napsauttamalla rattaan kuvaa ylävasemmalla.

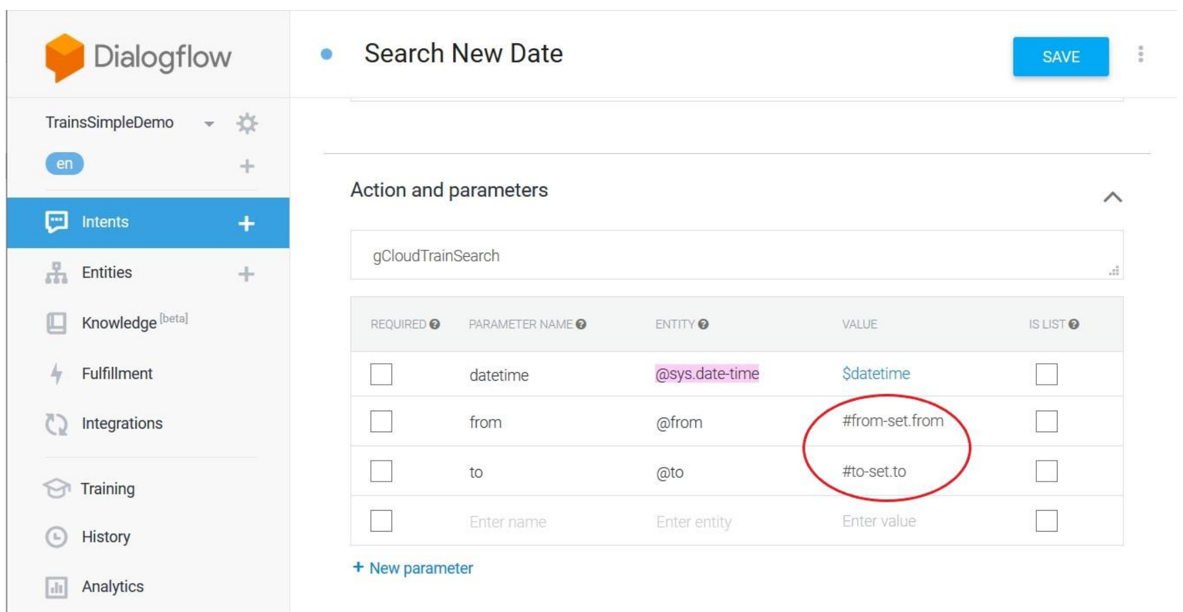


Kuva 22. Intents: Lista agenttiin määritellyistä aikomuksista.

Kuva 23. Intents: Search Trains, näytön yläosa. Esimerkki demosovelluksen tärkeimmän aikomuksen määrittelystä. **Action and parameters** -kentässä gCloudTrainSearch on ulkoisessa webhookissa suoritettavan toiminnon nimi.



Kuva 24. Intents: Search Trains, näytön alaosa. **Fulfillment**-kohdassa on valittu **Enable webhook call for this intent**, joten käyttäjän pyytämä tieto haetaan webhookin kautta.



Kuva 25. Intents: Search New Date, Action and parameters. *From*- ja *to*-parametrit periytyvät kontekstin kautta edellisestä hausta.

Kuva 26. Entities: Esimerkki itse luodusta entiteetistä nimeltä *from* eli lähtöasema. Entiteetin arvoille voi määritellä myös synonyymejä, kuten tässä asemakoodit.

Kuva 27. Fulfillment: Ulkoisen webhookin yhdistäminen Dialogflow'hun.

Kuva 28. Training: Agentin kouluttaminen käyttäjän syöttämän datan avulla.

Kuva 29. Training: Esimerkki syötetystä datasta ja siitä, miten agentti on sen tulkinnut. Tässä tapauksessa tulkinta on oikea.

General Languages ML Settings Export and Import Speech Share

MATCH MODE
Select the match mode that suits your agent best.

- Use the **Hybrid (Rule-based and ML)** mode for agents with a small number of examples/templates in intents, especially the ones using composite entities.
- Use **ML only** mode for agents with a large number of examples in intents, especially the ones using @sys.any

Hybrid (Rule-based and ML)

ML CLASSIFICATION THRESHOLD
Define the threshold value for the confidence score. If the returned value is less than the threshold value, then a fallback intent will be triggered, or if there is no fallback intents defined, no intent will be triggered.

0.3

AUTOMATIC SPELL CORRECTION
Allow ML to correct spelling of query during request processing.

TRAIN

Kuva 30. ML Settings: Chatbotin koneoppimisasetukset.

Intents CREATE INTENT

Search intents

- Buy Tickets
- Default Fallback Intent
- Default Welcome Intent
- End Conversation
- List Stations
- Search New Date
- Search Trains

Try it now

USER SAYS COPY CURL
Trains from Turku to Tampere on 10 May at 10 am

DEFAULT RESPONSE

Trains from Turku to Tampere on Friday, May 10, 2019 10:00 AM:

CLICK HERE TO GO TO THE VR WEBSITE

CONTEXTS RESET CONTEXTS
from-set to-set

INTENT
Search Trains

ACTION
gCloudTrainSearch

PARAMETER	VALUE
to	Tampere
datetime	{"date_time":"2019-05-10T10:00:00+03:00"}
from	Turku

DIAGNOSTIC INFO

Kuva 31. Diagnostic info: Chatbotin testaaminen Dialogflow’n konsolissa.

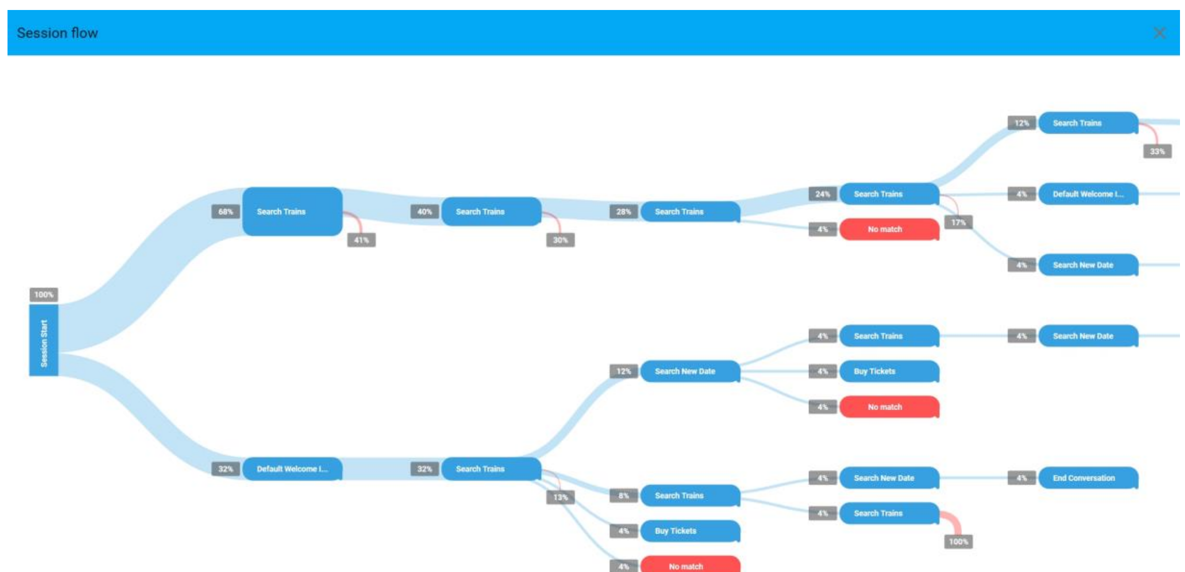
Diagnostic info

RAW API RESPONSE FULFILLMENT REQUEST FULFILLMENT RESPONSE FULFILLMENT STATUS


```
53             "date_time": "2019-05-10T10:00:00+03:00"
54           },
55           "to.original": "Tampere"
56         }
57       }
58     ],
59     "intent": {
60       "name": "projects/trainssimpledemo/agent/intents/
       ",
61       "displayName": "Search Trains"
62     },
63     "intentDetectionConfidence": 0.87524074,
64     "diagnosticInfo": {
65       "webhook_latency_ms": 372
66     },
67     "languageCode": "en"
68   },
69   "webhookStatus": {
70     "message": "Webhook execution successful"
71   }
72 }
```


CLOSE COPY FULFILLMENT REQUEST AS CURL COPY RAW RESPONSE

Kuva 32. Diagnostic info: *intentDetectionConfidence*-arvo vastauksen metadatassa.



Kuva 33. Analytics: Dialogflow’n analytiikkaa chatbotin (testi)käytöstä.

Liite 3. Lähdekoodi: index.js

Tämä koodi on Google Cloud Functions -toiminnon *index.js*-tiedoston sisältö. Tämän lisäksi tarvitaan *package.json*-tiedosto, jonka sisältö on Liitteessä 4.

Esimerkkikoodi on tarkoitettu Slack-integraatiota varten, eli siinä käytetään Rich Cards -toiminnallisuutta, joka ei toimi Dialogflow'n verkkodemoympäristössä. Verkkodemoa varten koodissa on kommentteina stringimuotoiset palautusarvot.

```
/**

* Copyright 2017 Google Inc. All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *    http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

/*
Train traffic data provided by Traffic Management Finland (https://www.digitraffic.fi/)
and used under the following license:
Creative Commons Attribution 4.0 International (CC BY 4.0)
*/

'use strict';

// Include nodejs request-promise-native package as dependency
// because async API calls require the use of Promises
const rpn = require('request-promise-native');
const moment = require('moment-timezone'); // for datetime formatting
const hostAPI = 'https://rata.digitraffic.fi/api/v1/live-trains/station/'; // root URL of the API
const {WebhookClient} = require('dialogflow-fulfillment');
const {Card} = require('dialogflow-fulfillment'); // for Rich Cards

// gCloudTrainSearch is the "Function to execute" in Google Cloud Functions
exports.gCloudTrainSearch = (req, res) => {
    const agent = new WebhookClient({request: req, response: res}); // Dialogflow agent
    console.log('Dialogflow Request headers: ' + JSON.stringify(req.headers)); // testing
    console.log('Dialogflow Request body: ' + JSON.stringify(req.body)); // testing
```

```
    // Default welcome intent
    function welcome(agent) {
        // Looks like line breaks don't work in Cards
        var welcomeCard = new Card({
            title: `Hello!`,
            text: `I can give you train schedules for direct connections between the following sta-
tions:\n
            Helsinki, Hämeenlinna, Joensuu, Jyväskylä, Kajaani, Kokkola, Kouvola, Kuopio, Lahti,
Lappeenranta, Leppävaara (Espoo), Mikkeli, Oulu, Pieksämäki, Pori, Rovaniemi, Seinäjoki, Tampere,
Tikkurila (Vantaa), Turku, Vaasa, Vainikkala`,
        });
      // Dialogflow's Web demo does not support Cards
      //agent.add(`Hello! I can give you train schedules for direct connections between the follow-
ing stations: Helsinki, Hämeenlinna, Joensuu, Jyväskylä, Kajaani, Kokkola, Kouvola, Kuopio, Lahti,
Lappeenranta, Leppävaara (Espoo), Mikkeli, Oulu, Pieksämäki, Pori, Rovaniemi, Seinäjoki, Tampere,
Tikkurila (Vantaa), Turku, Vaasa, Vainikkala`);
      agent.add(welcomeCard);
    }

    // Default fallback intent
    function fallback(agent) {
      agent.add(`Sorry, I don\'t understand. Please tell me where you want to travel from and where
to, and on what date. \nI can give you schedules for the following stations:\n
      Helsinki, Hämeenlinna, Joensuu, Jyväskylä, Kajaani, Kokkola, Kouvola, Kuopio, Lahti, Lappeen-
ranta, Leppävaara (Espoo), Mikkeli, Oulu, Pieksämäki, Pori, Rovaniemi, Seinäjoki, Tampere, Tikkuri-
la (Vantaa), Turku, Vaasa, Vainikkala`);
    }

    // List Stations intent
    function listStations(agent) {
        agent.add(`I can give you schedules for direct connections between the following sta-
tions:\n
        Helsinki, Hämeenlinna, Joensuu, Jyväskylä, Kajaani, Kokkola, Kouvola, Kuopio, Lahti, Lap-
peenranta, Leppävaara (Espoo), Mikkeli, Oulu, Pieksämäki, Pori, Rovaniemi, Seinäjoki, Tampere, Tik-
kurila (Vantaa), Turku, Vaasa, Vainikkala`);
      }

    // End Conversation intent
    function endConversation(agent) {
        agent.add('Thank you and have a nice day!');
      }

    // Buy Tickets intent
    function buyTickets(agent) {
        var ticketsCard = new Card({
            title: `Sorry`,
            text: `I can\'t sell you tickets, I can only tell you train schedules.`,
            buttonText: `To buy tickets, please visit the VR website`,
            buttonUrl: `https://www.vr.fi/cs/vr/fi/etusivu`
        });
```

```
        agent.add(ticketsCard);
        //agent.add('Sorry, I can\'t sell you tickets, I can only tell you train schedules. To buy
tickets, please visit the VR website: https://www.vr.fi/cs/vr/fi/etusivu');
      }

    // Function for passing data to the Search Trains and Search New Date intents in Dialogflow
    function searchTrains(agent) {
        // Get parameters given by user in Dialogflow

        // 1. Handle departure and arrival stations (required parameters)

        const depStat = agent.parameters.from; // departure station
        const arrStat = agent.parameters.to;    // arrival station
        console.log(`Given From and To parameters: from ${depStat} to ${arrStat}`); // testing

        // Station codes for the biggest cities are hard-coded here for demo purposes.
        // The API only uses station codes, not whole city names.
        // The API returns only direct train connections,
        // so no point in including every possible combination of all existing stations.
        // Note: Espoo = Leppävaara and Vantaa = Tikkurila
        var stationsObj =
            {'stations':
            [{  'name': 'Helsinki',
                'abb': 'HKI'},
            {   'name': 'Turku',
                'abb': 'TKU'},
            {   'name': 'Tampere',
                'abb': 'TPE'},
            {   'name': 'Leppävaara',
                'abb': 'LPV'},
            {   'name': 'Espoo',
                'abb': 'LPV'},
            {   'name': 'Hämeenlinna',
                'abb': 'HL'},
            {   'name': 'Lahti',
                'abb': 'LH'},
            {   'name': 'Pori',
                'abb': 'PRI'},
            {   'name': 'Jyväskylä',
                'abb': 'JY'},
            {   'name': 'Vaasa',
                'abb': 'VS'},
            {   'name': 'Seinäjoki',
                'abb': 'SK'},
            {   'name': 'Kuopio',
                'abb': 'KUO'},
            {   'name': 'Pieksämäki',
                'abb': 'PM'},
            {   'name': 'Joensuu',
                'abb': 'JNS'},
```

```
    {   'name': 'Lappeenranta',
        'abb': 'LR'},
    {   'name': 'Kouvola',
        'abb': 'KV'},
    {   'name': 'Vainikkala',
        'abb': 'VNA'},
    {   'name': 'Kokkola',
        'abb': 'KOK'},
    {   'name': 'Oulu',
        'abb': 'OL'},
    {   'name': 'Rovaniemi',
        'abb': 'ROI'},
    {   'name': 'Mikkeli',
        'abb': 'MI'},
    {   'name': 'Kajaani',
        'abb': 'KAJ'},
    {   'name': 'Tikkurila',
        'abb': 'TKL'},
    {   'name': 'Vantaa',
        'abb': 'TKL'}
    ]};

// Convert departure and arrival stations into station codes dep and arr
// to be used as URL query parameters
var dep, arr, i = '';
for (i in stationsObj.stations) {
    if (depStat == stationsObj.stations[i].name) {
        dep = stationsObj.stations[i].abb;
    }
    if (arrStat == stationsObj.stations[i].name) {
        arr = stationsObj.stations[i].abb;
    }
}

// 2. Handle the departure date and time (optional parameter)

var depDateTime = ''; // placeholder for datetime string

// If datetime is given by the user

if(agent.parameters.datetime) {
    var datetimeFromDialogflow = agent.parameters.datetime;
    var type = typeof datetimeFromDialogflow; // testing
    console.log(`typeof datetimeFromDialogflow is ${type}`); // testing

    // If Dialogflow returns datetime as a string (only date or only time given)

    if (type == "string") {
        depDateTime = datetimeFromDialogflow;
        console.log(`datetime string from Dialogflow is: ${depDateTime}`); // testing
```

```
    }

    // If Dialogflow returns datetime as an object (both date and time given)

    else {
        // Parse all possible @sys.date-time object formats to get datetime string
        // https://cloud.google.com/dialogflow-enterprise/docs/reference/system-entities
        // {"date_time":"2018-04-06T16:30:00-06:00"}
        // {"startDateTime":"2018-04-06T12:00:00-06:00","endDateTime":"2018-04-06T16:00:00-
06:00"}
        // {"endDateTime":"2018-04-06T16:00:00-06:00","startDateTime":"2018-04-06T12:00:00-
06:00"}
        // {"startDate":"2018-04-01T12:00:00-06:00","endDate":"2018-04-30T12:00:00-06:00"}
        // {"endDate":"2018-04-30T12:00:00-06:00","startDate":"2018-04-01T12:00:00-06:00"}
        // {"startTime":"2018-04-06T08:00:00-06:00","endTime":"2018-04-06T12:00:00-06:00"}
        // {"endTime":"2018-04-06T12:00:00-06:00","startTime":"2018-04-06T08:00:00-06:00"}
        if (datetimeFromDialogflow.hasOwnProperty('date_time')) {
            depDateTime = datetimeFromDialogflow.date_time;
            console.log(`date_time from Dialogflow is: ${depDateTime}`);
        }
        else if (datetimeFromDialogflow.hasOwnProperty('startDateTime')) {
            depDateTime = datetimeFromDialogflow.startDateTime;
            console.log(`startDateTime from Dialogflow is: ${depDateTime}`);
        }
        else if (datetimeFromDialogflow.hasOwnProperty('startDate')) {
            depDateTime = datetimeFromDialogflow.startDate;
            console.log(`startDate from Dialogflow is: ${depDateTime}`);
        }
        else if (datetimeFromDialogflow.hasOwnProperty('startTime')) {
            depDateTime = datetimeFromDialogflow.startTime;
            console.log(`startTime from Dialogflow is: ${depDateTime}`);
        }
    }
    // Manual fix for time zone offset since the DF integrations
    // do not understand Daylight Saving Time and return date-time
    // from Dialogflow as +02:00 throughout the year
    // 31.3.-27.10.2019
    // 29.3.-25.10.2020
    // 28.3.-31.10.2021
    var tzOffset = depDateTime.slice(19, 25);
    var shortDateTime = depDateTime.slice(0, 10);
    console.log(`tzOffset is ${tzOffset}`);

    if (moment(shortDateTime).isBetween('2019-03-31', '2019-10-27') ||
    moment(shortDateTime).isBetween('2020-03-29', '2020-10-25') ||
    moment(shortDateTime).isBetween('2021-03-28', '2021-10-31'))
    {
        if(tzOffset === '+02:00') {
            tzOffset = '+03:00'; // Daylight Saving Time
            depDateTime = depDateTime.slice(0,19) + tzOffset;
```

```
                console.log(`depDateTime with fixed offset is ${depDateTime}`);
            }
        }
    }
    // If datetime is not given by the user, default is the moment of query
    else {
        depDateTime = new Date(); // automatically in UTC
        depDateTime = depDateTime.toISOString();
        console.log(`depDateTime as ISOString is ${depDateTime}`);
    }

    // Then convert depDateTime to UTC since the API times are in UTC
    // Fetched train schedules are converted to Finnish time later
    depDateTime = moment.utc(depDateTime).format();
    console.log(`depDateTime in UTC is ${depDateTime}`);

    // Construct query strings according to the parameters given

    // Edit datetime formatting for API startDate query parameter
    // Datetime comes from Dialogflow with time zone offset (2019-04-03T10:00:00+03:00),
    // but the API does not accept that format
    var sliced = depDateTime.slice(0,19);
    depDateTime =  sliced + '.000Z';

    // Query string for the API
    // Limit set to 10 trains returned
    var trainUrl = `${hostAPI}${dep}/${arr}?startDate=${depDateTime}&limit=10`;
    console.log('The URL is ' + trainUrl); // testing

    // Asynchronous operations require Promises
    return new Promise((resolve, reject) => {
        // Make the HTTP request with request-promise-native
        // https://www.npmjs.com/package/request-promise

        var options = {
            uri: trainUrl,
            headers: {
                'User-Agent': 'Request-Promise-Native'
            },
            json: true
        };

        // All handling of returned JSON data goes under .then and before .catch
        // The Promise can be resolved in one of three possible ways or it can be rejected
        rpn(options)
            .then((json) => { // all this between {} is the first argument for .then
                // If a correctly formatted JSON is returned: Promise resolved
                if (json) {
                    var result = ''; // the answer goes here
```

```
                        var trains = JSON.stringify(json); // make a string out of the returned 
JSON object
                        console.log(`These trains were returned: ${trains}`); // testing

                        // Parse the returned train info for the desired data
                        var trainsArray = JSON.parse(trains); // Make an array out of the stringi-
fied JSON

                        // Error handling 1:
                        // Train not found (station not found or no direct connections at all)
                        // .code and TRAIN_NOT_FOUND come from the API
                        if (trainsArray.code == 'TRAIN_NOT_FOUND') {
                            console.log('API call returned TRAIN_NOT_FOUND');
                            result = agent.add('Station not found or no direct connections availa-
ble. Please try another search.'); // error msg to Dialogflow
                            resolve(result); // Promise resolved
                        } // Promise resolved, option 1
                        else {
                            // Filter total trainsArray for passenger trains only
                            // Field names come from the API 
(https://www.digitraffic.fi/rautatieliikenne/#junat)
                            var passengerTrains = []; // new array for passenger trains
                            for (var i = 0; i < trainsArray.length; i++) {
                                if (trainsArray[i].trainCategory == 'Long-distance' || train-
sArray[i].trainCategory == 'Commuter') {
                                    passengerTrains.push(trainsArray[i]);
                                }
                            }
                            // Error handling 2:
                            // If the passengerTrains array remains empty after filtering,
                            // i.e. the JSON only returned cargo trains and no passenger trains.
                            // (For some connections, e.g. Turku-Kuopio, there may be direct cargo 
trains but no direct passenger trains.)
                            if (passengerTrains.length < 1) {
                                console.log('passengerTrains array empty');
                                result = agent.add('No passenger train connections available. 
Please try another search.'); // error msg to Dialogflow
                                resolve(result); // Promise resolved
                            } // Promise resolved, option 2
                            else {
                                // If passenger trains are found, parse them for the desired data:
                                // Train number, departure and arrival stations and times

                                // Placeholders for collecting all parsed data
                                var output = '';
                                var outputArray = [];

                                // Loop through array of train objects
                                // passengerTrains is the top-level array
                                for (var i = 0; i < passengerTrains.length; i++) {
```

```
                    var train = passengerTrains[i]; // each train is an object
                    var trainNumber = train.trainNumber;
                    var trainType = train.trainType;
                    // Loop through array of timeTableRows inside each train object
                    for (var j = 0; j < train.timeTableRows.length; j++) {
                        var row = train.timeTableRows[j]; // each row is an object
                        if (row.stationShortCode == dep && row.type == 'DEPARTURE')
{
                            var depInfo = `Train ${trainType} ${trainNumber} leaves
from ${depStat} at `;
                            var depTime = row.scheduledTime;
                            // Set departure time to local time
                            depTime = moment(depTime).tz('Europe/Helsinki');
                            depTime = moment(depTime).format(); // back to ISO
string
                            // Slice ISO-8601 format to show just time: 2019-03-
18T15:04:00.000Z to 15:04
                            depTime = depTime.slice(11, 16);
                            var trainObj = {
                                depInfo: depInfo,
                                depTime: depTime
                            }
                            trainObj.depInfo = depInfo;
                            trainObj.depTime = depTime;
                            outputArray.push(trainObj);
                            console.log(trainObj); // testing
                        }
                        if (row.stationShortCode == arr && row.type == 'ARRIVAL') {
                            var arrTime = row.scheduledTime;
                            arrTime = moment(arrTime).tz('Europe/Helsinki');
                            arrTime = moment(arrTime).format();
                            arrTime = arrTime.slice(11, 16);
                            var arrInfo = ` and arrives in ${arrStat} at
${arrTime}.`;
                            trainObj.arrInfo = arrInfo;
                            console.log(arrInfo); // testing
                        }
                    } // for loop end

                    // For loop results in array of objects with 3 properties:
[depInfo, depTime, arrInfo]
                    // Sort outputArray by depTime to display results in correct
order
                    // https://stackoverflow.com/questions/6937173/how-to-sort-an-
array-of-objects-using-their-object-properties
                    var sortedArray = outputArray.sort(function(a, b) {
                        if (a.depTime < b.depTime) {
                            return -1;
                        }
                        else if (a.depTime > b.depTime) {
```

```
                        return 1;
                    }
                    return 0;
                });
                var txt = '';
                for (var k = 0; k < sortedArray.length; k++) {
                    depInfo = sortedArray[k].depInfo;
                    depTime = sortedArray[k].depTime;
                    arrInfo = sortedArray[k].arrInfo;
                    // Line breaks not supported in output to Dialogflow con-
sole or web demo

                    txt = `${txt} ${depInfo} ${depTime} ${arrInfo} \n`;
                }
            } // for loop end

            // Set final iteration of sortedArray as output
            output = txt;
            console.log(output);

            // Format response as a Rich Card
             // Web demo does not support Cards

             // Format depDateTime from UTC to the correct time zone
             depDateTime = moment(depDateTime).tz('Europe/Helsinki');
             console.log(`depDateTime with time zone set: ${depDateTime}`); //
testing

             // Format depDateTime from ISO format back into natural language
             // moment().format('LLLL'); --> Friday, April 5, 2019 4:45 PM
             var outputDateTime = moment(depDateTime).format('LLLL');
             console.log(`outputDateTime is ${outputDateTime}`);

             var outputCard = new Card({
                 title: `Trains from ${depStat} to ${arrStat} on ${output-
DateTime}: `,

                 text: `${output}`,
                 buttonText: 'Click here to go to the VR website',
                 buttonUrl: 'https://www.vr.fi/cs/vr/fi/etusivu'
             });

             result = agent.add(outputCard); // send the collected output to
Dialogflow
             //result = agent.add(`Trains from ${depStat} to ${arrStat} on
${outputDateTime}: ${output}`);
             resolve(result); // Promise resolved
         } // Promise resolved, option 3, else end
     } // first else end
 } // if(json) end
 // In case none of the "if" conditions apply: Promise rejected
 // This should really never come up unless the API generates
```

```
                    // a JSON file in the wrong format or something.
                    // In this case, the default text response from DF is shown to the user.
                    else {
                        console.log('Promise rejected');
                        reject(Error('Promise rejected'));
                    } // else end (Promise rejected)
                }, (error) => { // and this is the second argument for .then
                    // If the API can't be reached, the URL is wrong etc.
                    // In this case the webhook request from Dialogflow usually times out (5 sec-
onds)
                    console.log('Error message from .then: ' + error);
                    }) // rpn(options).then end
                .catch((error) => { // if .then fails
                    throw new Error('Error.message from .catch: '+ error.message);
                }); // rpn(options).catch end
        }); // Promise end
    } // searchTrains end

    // Mapping functions to Dialogflow intents
    let intentMap = new Map();
    intentMap.set('Default Welcome Intent', welcome);
    intentMap.set('Default Fallback Intent', fallback);
    intentMap.set('End Conversation', endConversation);
    intentMap.set('List Stations', listStations);
    intentMap.set('Buy Tickets', buyTickets);
    // Two intents (Search Trains and Search New Date) call the same searchTrains function in the
backend.
    // The only difference is that Search New Date retains the "from" and "to" parameters from the
first search.
    intentMap.set('Search Trains', searchTrains);
    intentMap.set('Search New Date', searchTrains);
    agent.handleRequest(intentMap);

}; // exports end
```

Liite 4: Lähdekoodi: package.json

Tämä koodi on Google Cloud Functions -toiminnon *package.json*-tiedoston sisältö. Tämän lisäksi tarvitaan *index.js*-tiedosto, jonka sisältö on Liitteessä 3.

```
{
  "name": "gCloudTrainSearch",
  "description": "Dialogflow agent for searching train schedules via the VR API",
  "version": "0.0.1",
  "dependencies": {
    "dialogflow-fulfillment": "^0.5.0",
    "actions-on-google": "^2.2.0",
    "request-promise-native": "^1.0",
    "request": "^2.88.0",
    "moment" : "^2.24.0",
    "moment-timezone": "^0.5.23"
    }
}
```

Liite 5: Rajapinnan palauttama JSON-data

Kuva 34 on esimerkki rautatieliikenteen avoimen rajapinnan palauttamasta *junat*-muotoisesta JSON-datasta. Esimerkissä näkyy ensin koko junan tiedot ja *timeTableRows*-taulukossa kunkin asemapysähdyksen tiedot omina objekteinaan. Juna lähtee (*type: "DEPARTURE"*) Helsingistä (*stationShortCode: "HKI"*), saapuu (*type: "ARRIVAL"*) Pasilaan (*stationShortCode: "PSL"*) ja lähtee eteenpäin Pasilasta *scheduledTime*-kentässä määritellyn ajan mukaisesti. Jokaisen aseman osalta saapumista ja lähtöä varten annetaan oma *timeTableRows*-objekti, vaikka juna ohittaisikin aseman pysähtymättä. Esimerkiksi Helsingistä Tampereelle kulkevassa junassa on 89 *timeTableRows*-objektia.

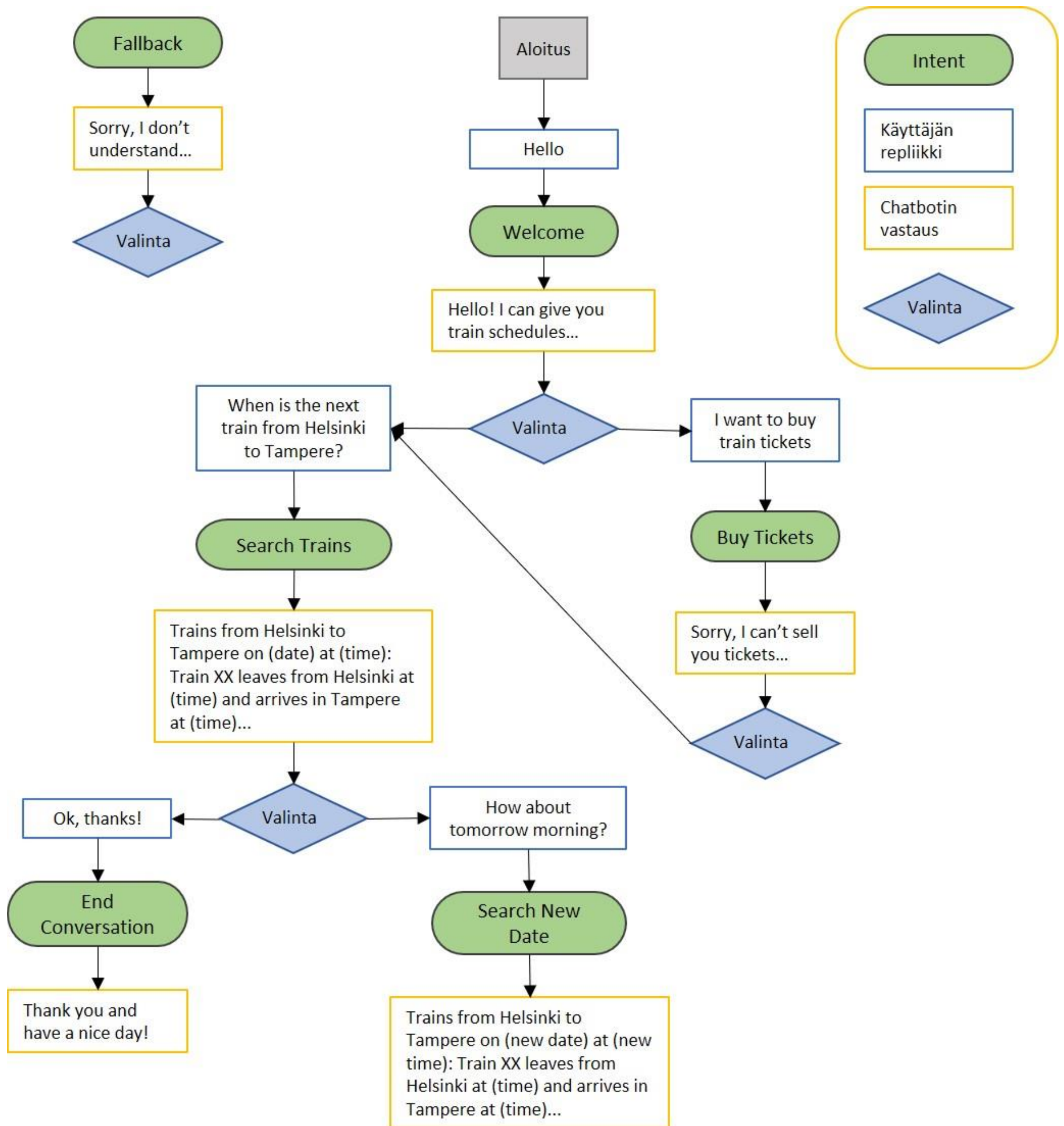
JSON   Raw Data   Headers

Save   Copy   Collapse All   Expand All


```
▼ 0:
    trainNumber:                53
    departureDate:              "2019-04-23"
    operatorUICCode:            10
    operatorShortCode:          "vr"
    trainType:                  "S"
    trainCategory:              "Long-distance"
    commuterLineID:             ""
    runningCurrently:           false
    cancelled:                  false
    version:                    253020078899
    timetableType:              "REGULAR"
    timetableAcceptanceDate:    "2019-02-22T11:06:53.000Z"
  ▼ timeTableRows:
    ▼ 0:
        stationShortCode:       "HKI"
        stationUICCode:         1
        countryCode:            "FI"
        type:                   "DEPARTURE"
        trainStopping:          true
        commercialStop:         true
        commercialTrack:        "8"
        cancelled:              false
        scheduledTime:          "2019-04-23T17:24:00.000Z"
    ▼ 1:
        stationShortCode:       "PSL"
        stationUICCode:         10
        countryCode:            "FI"
        type:                   "ARRIVAL"
        trainStopping:          true
        commercialStop:         true
        commercialTrack:        "3"
        cancelled:              false
        scheduledTime:          "2019-04-23T17:29:00.000Z"
    ▼ 2:
        stationShortCode:       "PSL"
        stationUICCode:         10
        countryCode:            "FI"
        type:                   "DEPARTURE"
        trainStopping:          true
        commercialStop:         true
        commercialTrack:        "3"
        cancelled:              false
        scheduledTime:          "2019-04-23T17:30:00.000Z"
```


Kuva 34. Esimerkki *rata.digitraffic.fi*-rajapinnan palauttamasta JSON-datasta.

Liite 6: Vuokaavio chatbotin toiminnasta



Kuva 35. Vuokaavio chatbotin toiminnasta (mallina käytetty kuvaa Dialogflow 2019p).