Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Lauri Koskinen

Scala.js-skriptituen lisääminen chatbot-moottoriin

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

13.11.2020

Tekijä Otsikko Sivumäärä Aika	Lauri Koskinen Scala.js-skriptituen lisääminen chatbot-moottoriin 21 sivua 13.11.2020
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Ohjelmistotuotanto
Ohjaajat	Lehtori Simo Silander

Insinöörityöni aiheena oli luoda Scala.js-skriptituki StarChatiin, joka on avoimen lähdekoodin chatbot-projekti. StarChatiä kehittää GetJenny Oy, jolle käytännön osuus tehtiin.

StarChat tarjoaa pelkän chatbot-moottorin, jota käytetään REST-rajapintojen kautta. StarChatissä bottia opetetaan vastaamaan kysymyksiin luomalla sinne tiloja. Vastausta haettaessa botti palauttaa tilan, joka sisältää vastauksen. Tilaa luotaessa siihen kirjoitetaan analysaattori, joka määrittää, milloin tila kuuluu palauttaa. Analysaattori kirjoitetaan StarChatin omalla skriptikielellä (domain specific language, DSL).

Työn tarkoituksena oli implementoida StarChatiin uusi skriptikieli analysaattoreiden luomiseen ja mitata sekä verrata sen suorituskykyä DSL-analysaattoreihin.

Skriptikieleksi valittiin Scala.js, koska StarChat on kirjoitettu Scala-ohjelmointikielellä. Scala ja Scala.js ovat hyvin samanlaisia, mutta Scala käännetään tavukoodiksi, joka suoritetaan JVM:ssä. Scala.js taas käännetään JavaScriptiksi, jota muun muassa selaimet suorittavat. StarChatin DSL on kehitetty analysaattoreiden tekemiseen, mutta se rajoittuu StarChatssä määritettyihin toimintoihin. Scala.js tarjoaisi tavan kirjoittaa aivan uudenlaisia analysaattoreita ilman, että se vaatisi StarChatin uudelleenkääntämistä.

Insinöörityössä käydään läpi ajon aikaista Scala.js:n kääntämistä JavaScriptiksi ja sen suorittamista JVM:ssä Javan ScriptEngine-rajapintaa käyttäen.

Scala.js-analysaattoreita verrattaessa olivat ne paljon hitaampia suorittaa kuin DSL-analysaattorit.

Avainsanat	Scala, Scala.js, JavaScript, chatbot, StarChat, ScriptEngine

Author Title Number of Pages Date	Lauri Koskinen Creating Scala.js Scripting Support for Chatbot Engine 21 pages 13 November 2020
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Professional Major	Software Engineering
Instructor	Simo Silander, Senior Lecturer

The objective of the study was to create Scala.js scripting support for StarChat. StarChat is an open source chatbot project developed by GetJenny Oy, for whom the practical part of the study was made.

StarChat provides only the chatbot engine that is accessed through REST interfaces. In StarChat, the bot is taught to answer questions by creating states. When retrieving a response, the bot returns a state that contains the response. When creating a state, an analyzer is written to it that determines when the state should be returned. The analyzer is written in StarChat's own domain specific language (DSL).

The purpose of the study was to implement a new scripting language in StarChat for creating analyzers and to measure and compare their performance.

Scala.js was chosen as the scripting language because StarChat is written in the Scala programming language. They are very similar, but Scala is compiled into byte code that is executed in JVM, whereas Scala.js is compiled into JavaScript, which e.g. browsers run. StarChat's DSL has been developed for creating analyzers but is limited to the function defined in StarChat. Scala.js provides a way to write completely new types of analyzers without requiring a recompilation of StarChat.

The thesis covers Scala.js compilation into JavaScript during runtime and its execution in the JVM using the Java ScriptEngine interface.

When comparing Scala.js analyzers, they were much slower to perform than DSL analyzers.

Keywords	Scala, Scala.js, JavaScript, chatbot, StarChat, ScriptEngine

Sisällys

1 Johdanto 1

2 StarChat 3

2.1 StarChat:in integraatio chat-alustaan 4
2.2 StarChatin opettaminen 5
2.3 StarChat analysaattorit (Analyzers) 9

3 Scala 10

4 Scala.js-integraation taustatutkimus 11

4.1 Scala.js:n kääntäminen 11
4.2 JavaScript suorittaminen 13

5 Toteutus 15

5.1 Scala.js:n ajonaikainen kääntäminen JavaScriptiksi 17
5.2 Scala.js-analysaattoreiden ajonaikainen suoritus ja muodostaminen 17
5.3 Scala.js-analysaattoreiden nopeus 19

6 Yhteenveto 20

Lähteet 21

1 Johdanto

Opinnäytetyö käsittelee StarChat-chatbot-moottoria. StarChat on GetJenny Oy:n kehittämä avoimen lähdekoodin projekti, joka löytyy Githubista [1]. Opinnäytetyön tekninen osuus on tehty GetJenny Oy:lle.

Chatbotit ovat chatteihin liitettyjä botteja, jotka kuuntelevat keskustelua ja voivat suorittaa erilaisia toimintoja riippuen siitä, mitä chatiin kirjoitetaan.

Chatboteihin törmää yleensä asiakaspalvelu-chateissä. Ne soveltuvat asiakaspalveluun erittäin hyvin, koska ne voivat palvella ympäri vuorokauden ja palvella samanaikaisesti useaa asiakasta kerralla. Ne ovat nopeita hakemaan tietoa ja vastaamaan kysymyksiin. Toisaalta botit osaavat vastata vain, mitä niille on opetettu, ja joskus ne saattavat antaa jopa aivan vääränlaisen vastauksen (kuva 1).

Kuva 1. Kuva vakuutusyhtiö Turvan Facebook-sivulta [2]. Chatbotti vastaa koomisesti.

Asiakaspalveluboteilla ei korvata asiakaspalvelijoita, vaan ne tekevät yhteistyötä. Uudet keskustelut ohjataan ensiksi botille, joka pyrkii vastaamaan kysymyksiin, tai vaikka kysymään taustatietoja ennen kuin siirretään ihmiselle. Esimerkiksi kuvitteellinen Terveysasemabotti voisi kerätä asiakkaan tietoja ja oireita ennen hoitajalle siirtämistä. Yksinkertaiset kysymykset botti pystyy hoitamaan kokonaan itse. Jos botti ei osaa vastata asiakkaan kysymykseen, se voi siirtää keskustelun ihmiselle, jonka jälkeen vastaus voidaan opettaa botille. LocalTapiola pystyi automatisoimaan yli 80 % keskusteluista heidän GetJenny-chatbotilla [3].

StarChatissä bottia opetetaan luomalla sinne tiloja, jotka sisältävät vastauksen kysymykseen. Tilaan liitetään myös analysaattori, jonka avulla botti osaa valita oikean vastauksen kysymykseen.

Analysaattorit kirjoitetaan StarChatin omalla skriptikielellä (DSL). Opinnäytetyön tavoitteena on implementoida Scala.js-skriptituki StarChatiin, minkä avulla voisi vaihtoehtoisesti kirjoittaa analysaattoreita Scala.js-ohlelmointikielellä. Tarkoituksena on siis pystyä käyttämään molempia skriptikieliä analysaattoreiden luomiseen. Scala.js valittiin skriptikieleksi, koska StarChat on kirjoitettu Scala-ohjelmointikielellä.

DSL on rajoittunut StarChatissä määriteltyihin toimintoihin. Uusien analysaattoritoimintojen luominen vaatisi StarChatin kääntämisen uudelleen. Vapaampi Scala.js-ohjelmointikieli voisi olla ratkaisu tähän ongelmaan.

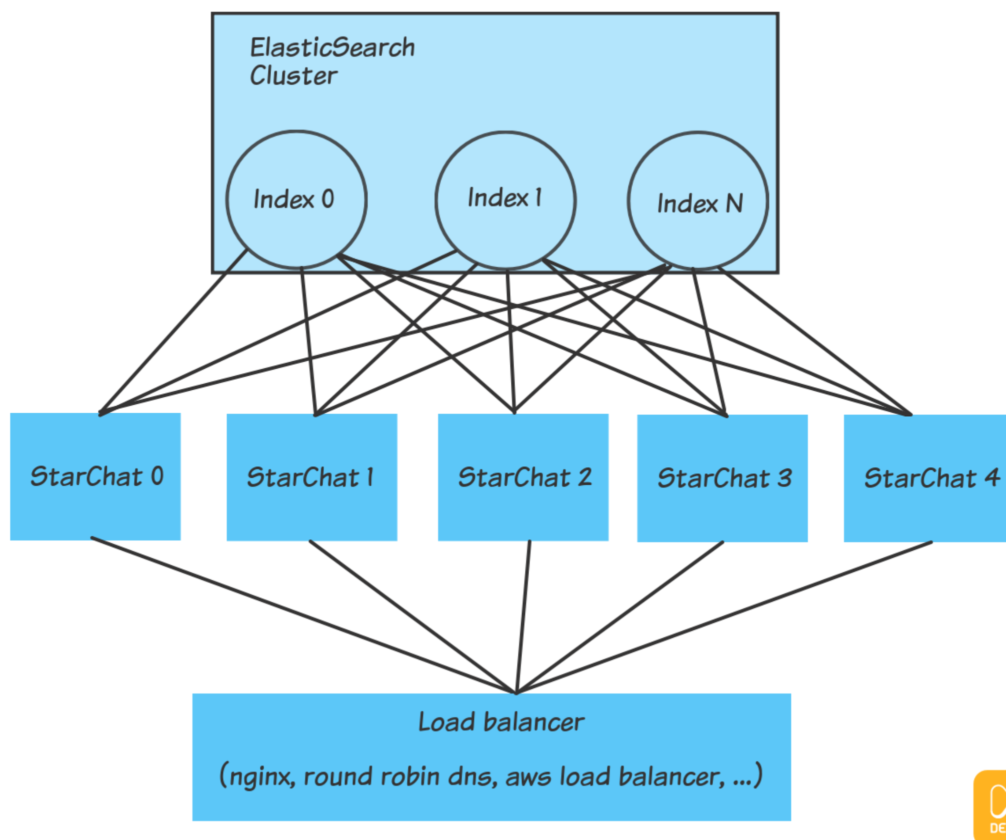
Ensimmäisessä luvussa esitellään StarChatiä, sen toimintaa ja DSL-analysaattoreita. Toisessa luvussa esitellään lyhyesti Scala-ohjelmointikieltä ja verrataan Java-ohjelmointikieleen. Kolmannessa luvussa tutkitaan tapoja Scala.js-koodin kääntämiseen JavaScript-koodiksi ja JavaScriptin suorittamista Java-virtuaalikoneessa. Neljännessä luvussa kerrotaan Scala.js-skriptituen toteutuksesta ja integraatiosta StarChatiin sekä verrataan DSL- ja Scala.js-analysaattoreiden nopeuksia. Lopuksi pohditaan lopputulosta ja parannuksia työhön.

2 StarChat

GetJenny on start up -yhtiö, joka on kehittänyt StarChatin. StarChatin kautta he tarjoavat backend-moottorin keskusteluiden hallinnoimiseen tai vastausten ehdottamiseen perustuen edellisiin keskusteluihin ja integroimaan tämän asiakkaiden kommunikaatioalustoille [4].

Kaikki StarChat-integraatiosovellukset ovat suljettua koodia, mutta backend on avointa lähdekoodia, jonka voi ladata, muokata ja käyttää vapaasti.

StarChat on horisontaalisesti skaalautuva sovellus. Tämä tarkoittaa sitä, että palvelupyyntöjen lisääntyessä pystytään luomaan useampi StarChat-node rinnakkain (kuva 2), jolloin saadaan lisää palvelukapasiteettia.

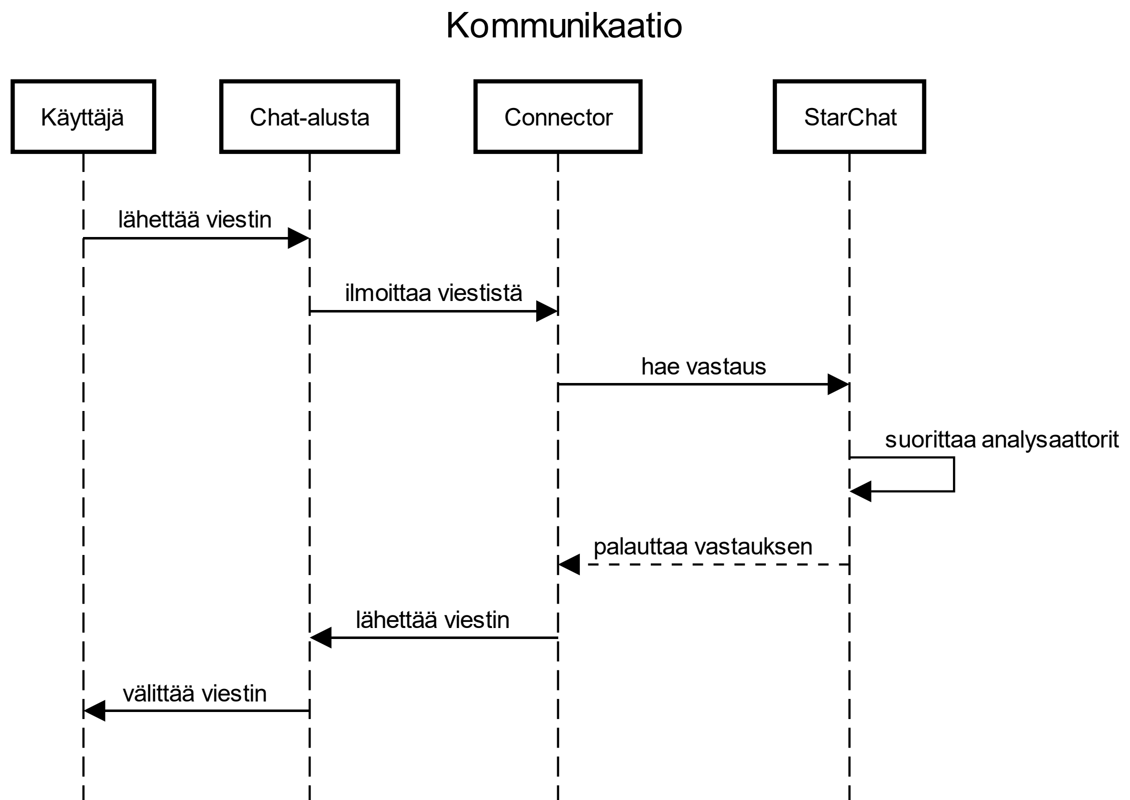


Kuva 2. StarChat-skaalautuvuus [5].

Horisontaalinen skaalautuvuus saavutetaan siten, että “Load balancer” jakaa kuormituksen StarChat-nodeille tasaisesti. StarChat-nodet käyttävät yhteistä tietokantaa (ElasticSearch Cluster).

2.1 StarChatin integraatio chat-alustaan

Jotta StarChatin kanssa voi käydä keskustelua, täytyy StarChat kytkeä johonkin chat-alustaan. Chat-alustoja ovat esimerkiksi Whatsapp tai Slack. Chat-alustalta vaaditaan avointa rajapintaa keskustelun käymiseksi. StarChatin liittämiseksi haluamaansa chat-alustaan täytyy luoda integraatiosovellus ns. “connector”, joka yhteensovittaa chat-alustan ja StarChatin rajapinnat (kuva 3).



Kuva 3. Chat-alustan, connectorin ja StarChatin kommunikaatio.

Chat-alustan ja StarChatin integraatio tapahtuu siten, että chat-alusta ilmoittaa uusista keskusteluista ja viesteistä connectorille esim. webhookin avulla. Webhook on tapa saada uutta tietoa silloin, kun sitä tulee, jolloin tiedon perään ei tarvitse kysellä. Connector hakee vastauksen viestiin StarChatiltä ja lähettää sen Chat-alustalle. Käynnissä olevat keskustelut ja keskusteluhistoriaa tulee käsitellä connectorissa, koska keskustelun tila tulee liittää ”hae vastaus” -pyynnön mukaan.

2.2 StarChatin opettaminen

Jotta StarChat osaisi vastata kysymyksiin, sille täytyy opettaa vastauksia. Opettaminen tarkoittaa sitä, että StarChatiin talletetaan valmiita vastauksia, joihin liitetään analysaattori, joka arvioi vastauksen sopivuutta kysymykseen pisteyttämällä.

Opettaminen tapahtuu siten, että StarChatiin luodaan ensiksi indeksi. Indeksit ovat käytännössä eri botteja, joita voi olla useita ja niitä voidaan opettaa eri tarkoituksiin. Esimerkiksi eri kielille eri botti. Indeksi luodaan tekemällä post-kutsu “/<index_name>/index_management/create" päätepisteeseen, jossa “<index_name>” on botin nimi. Nimi tulee olla muotoa “index_getjenny_italian_usecase0”.

Vastauksia opetetaan botille luomalla tiloja (state). Tila sisältää muun muassa tilan nimen, halutun vastauksen ja analysaattorin (analyzer). Analysaattori on skripti, jolla määritellään, milloin botin tulee antaa tilan vastaus. Tila luodaan tekemällä post-kutsu “/<index_name>/decisiontable” päätepisteeseen.

```
{
  "state": "forgot_password",
  "executionOrder": 0,
  "maxStateCount": 0,
  "analyzer":         "booleanAnd(keyword(\"password\"),booleanOr(keyword(\"re-
set\"),keyword(\"forgot\")))",
  "queries": [
    "I forgot my password",
    "my password is wrong",
    "don't remember the password"
  ],
  "bubble": "Hello %name%, click the button below to start password recovery
process.",
  "action": "show_button",
  "actionInput": {
    "text to be shown on button": "password_recovery"
  },
  "stateData": {
    "url": "www.getjenny.com"
  },
  "successValue": "eval(show_buttons)",
  "failureValue": "dont_understand",
  "evaluationClass": "default",
  "version": 12
}
```

Esimerkkikoodi 1. Decisiontable-päätepisteen pyynnön body.

Esimerkkikoodi 1:stä oleelliset osiot ovat state, analyzer ja bubble. State-kenttä kertoo tilan nimen. Analyzer-kentässä on analysattori-skripti, jonka StarChat suorittaa, kun hakee vastausta viestiin. Analysaattoreista kerrotaan myöhemmin. Bubble sisältää botin vastauksen.

StarChatiltä haetaan vastaus Post “/<index_name>/get_next_response” päätepisteestä.

```
{
  "conversationId": "conv_12131",
  "traversedStates": [
    "state_0",
    "state_1",
    "state_3"
  ],
  "userInput": {
    "text": "Hello, I would like to reset my password, is it possible?",
    "img": "aGVsbG8K="
  },
  "state": [],
  "data": {
    "name": "Donald Duck",
    "job": "idle"
  },
  "threshold": 0,
  "evaluationClass": "default",
  "maxResults": 1,
  "searchAlgorithm": "NGRAM3"
}
```

Esimerkkikoodi 2. Keskustelupäätepisteen pyynnön body

Kun botilta kysytään vastausta, traversedStates-kentän listaan tulee laittaa kaikki tilat, missä keskustelu on käynyt. UserInputin sisällä on objekti, jossa text-kentässä on käyttäjän viesti.

Botin vastaus voi olla seuraavanlainen:

```
[
  {
    "conversationId": "conv_12131",
```

```
    "state": "forgot_password",
    "traversedStates": [
      "state_0",
      "state_1",
      "state_3",
      "forgot_password"
    ],
    "maxStateCount": 0,
    "analyzer": "keyword(\"password\")",
    "data": {
      "name": "Donald Duck",
      "job": "idle"
    },
    "bubble": "Hello Donald Duck, click the button below to start password
recovery process",
    "action": "show_button",
    "actionInput": {
      "text to be shown on button": "password_recovery"
    },
    "stateData": {
      "url": "www.getjenny.com"
    },
    "successValue": "eval(show_buttons)",
    "failureValue": "dont_understand",
    "score": 0.83
  }
]
```

Esimerkkikoodi 3. Keskustelupäätepisteen vastaus Json-muodossa

Esimerkkikoodi 3:ssa botti palauttaa listan tiloja. Tällä kertaa listassa on vain yksi forgot_password-tila. Botin vastaus löytyy bubblesta, joka vastaa botin puhekuplaa chatissä. Actionissä voi olla joku muu toiminto kuin puhekuplan näyttäminen. Tässä action sisältää “show_button”-toiminnon, joka kertoo, että chatissä kuuluu näyttää actionInput:ssa olevat napit. ActionInput:n arvon avain on napin teksti ja arvona on botin tilan nimi. Actioneiden toteutus tulee implementoida chat-alustalla ja connectorissa. TraversedStates- ja data-osiot tulee liittää seuraavaan get_next_response-pyyntöön.

2.3 StarChat-analysaattorit (Analyzers)

Analysaattori on tilaan liitetty skripti, joka määrittää, milloin tila pitää palauttaa (esimerkkikoodi 1). Määritys tapahtuu pisteytyksellä.

Analysaattorit kirjoitetaan StarChatin omalla skriptikielellä, joka koostuu "rakennuspalikoista" [6]. Niitä ovat operaattorit, esim. booleanAnd ja booleanOr, ja atomit, esimerkiksi keyword, lastTravStateIs ja hasTravState. Keyword-atomi katsoo, esiintyykö määritelty sana käyttäjän viestissä. lastTravStateIs-atomilla voidaan tarkistaa käyttäjän viimeisintä tilaa, jossa on käynyt. hasTravState:lla voidaan katsoa, onko käyttäjä käynyt ollenkaan tietyssä tilassa. Atomeita voidaan yhdistää toisiinsa seuraavasti:

```
booleanAnd(keyword("password"),booleanOr(keyword("reset"),keyword("forgot")))
```

Esimerkkikoodi 4. StarChat-analysaattori

Esimerkkikoodi 4:ssä analysaattori katsoo, esiintyykö sana “password” sanojen “reset” tai “forgot” kanssa.

Kun StarChatiltä haetaan vastausta, se suorittaa botin tilojen analysaattorit ja palauttaa eniten pisteitä saaneet ja vähimmäispistemäärän ylittäneet tilat. Vähimmäispistemäärä ilmoitetaan threshold-kentässä (esimerkkikoodi 2).

Analysoijien on tärkeä olla nopeasti suoritettavissa, jotta hakuaika pysyy mahdollisimman pienenä. StarChat rakentaa ja tallettaa tilojen analysoijat välimuistiin optimoidakseen suorituskyvyn.

StarChatin skriptikieli on rajattu siellä määriteltyihin toimintoihin (keyword, hasTravState, jne.), ja uusien toimintojen lisääminen vaatisi StarChatin kääntämisen uudelleen. Tämän takia selvitetään, onko Scala.js:llä kirjoitettu skripti yhtä nopeasti suoritettavissa.

3 Scala

Scala on funktionaalinen, oliomallinen ja imperatiivinen ohjelmointikieli [7]. Scalalla kirjoitetut ohjelmat ajetaan Java-virtuaalikoneessa. Verrattuna Javaan Scala on vähemmän jaaritteleva. Scalan älykkään kääntäjän ansiosta ohjelmoijan ei tarvitse tarkentaa, mitä kääntäjä voi jo johtaa koodista, kuten tyypit.

Esimerkkikoodi 5:ssä määritellään Person-luokka Javalla. Scalassa vastaava tehdään esimerkkikoodi 6:ssa, jossa käytetään case-luokkaa mallintamaan muuttumatonta dataa.

```
public class Person implements Serializable {
    private final String firstName;
    private final String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }
}
```

Esimerkkikoodi 5. Person-luokka tehty Javalla

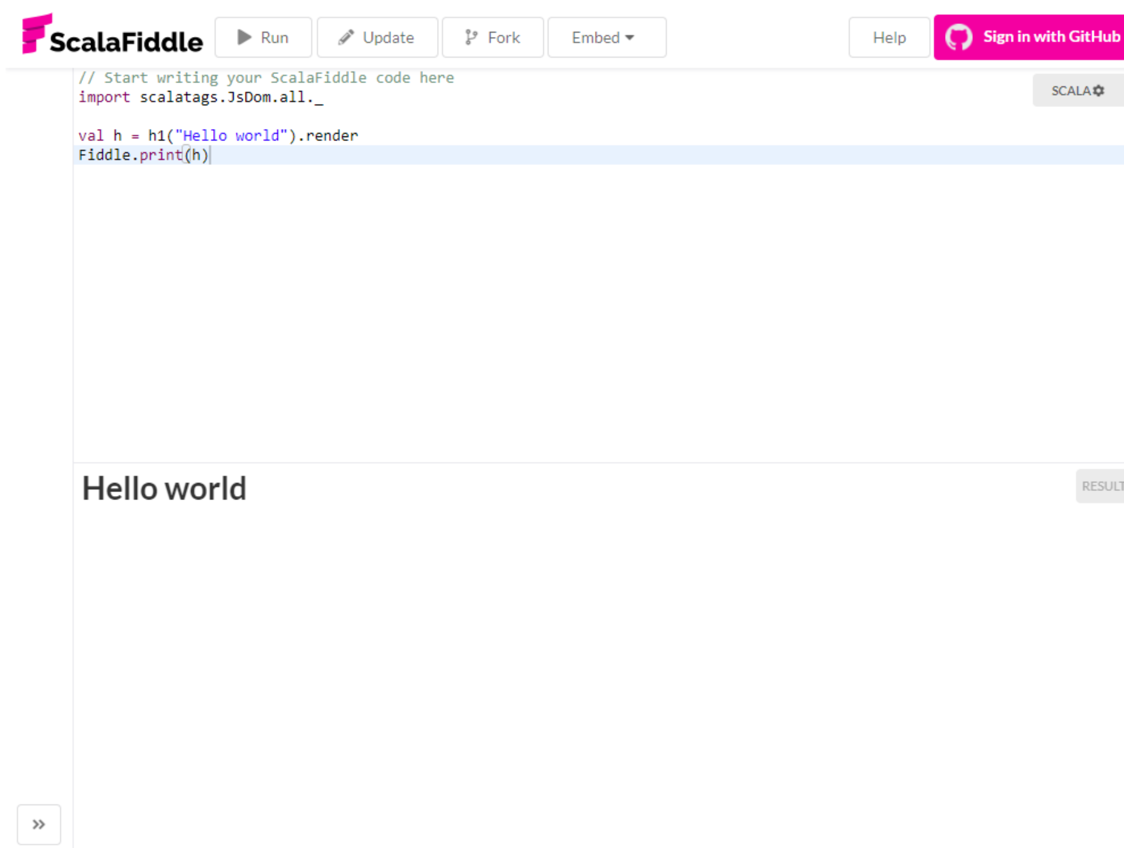
```
case class Person(firstName: String, lastName: String)
```

Esimerkkikoodi 6. Person-luokka tehty Scalalla

4 Scala.js-integraation taustatutkimus

4.1 Scala.js:n kääntäminen

Scala.js-koodin ajonaikaiseen kääntämiseen liittyviä tutkimuksia ja artikkeleita etsiessä selvisi, ettei sitä ole tutkittu kovin paljoa. Yksi projekti kuitenkin löytyi, ScalaFiddle [8], joka tarjoaa graafisen toimintaympäristön Scala-koodin kirjoittamiseen (kuva 4). Se kääntää käyttäjän kirjoittaman Scala-koodin serverillä JavaScriptiksi, jonka se palauttaa käyttäjän selaimeen suoritettavaksi.

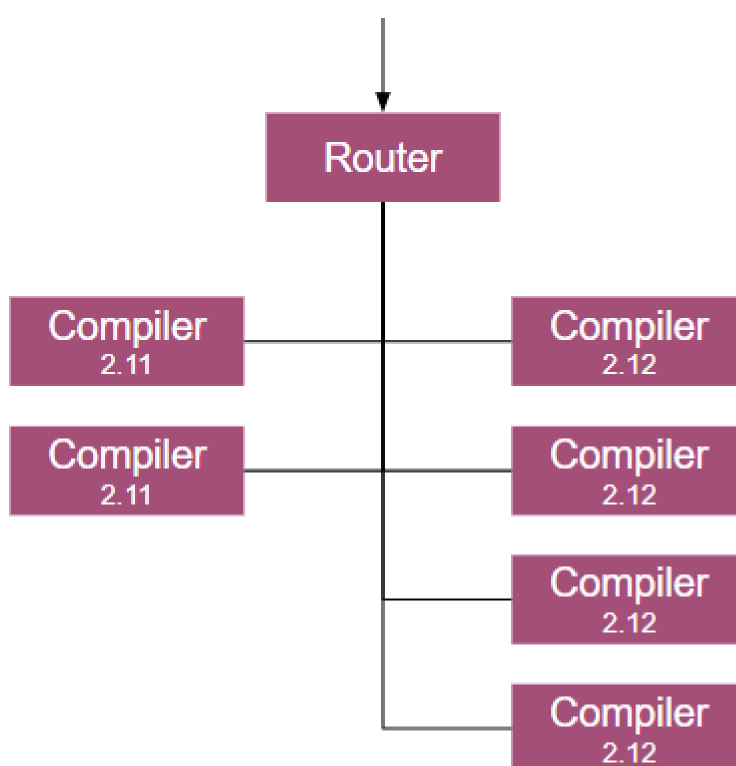


Kuva 4. Kuvakaappaus ScalaFiddle-verkkosovelluksesta.

ScalaFiddle on avoimen lähdekoodin projekti ja löytyy kokonaisuudessaan Githubista [9]. ScalaFiddle-projektia tutkimalla näki, miten Scala-kääntäjä saadaan tuottamaan JavaScriptiä Scala.js-lisäosan avulla.

ScalaFiddlen tekijä, Otto Chrons, kertoo medium-sivustoon tehdyssä julkaisussaan [10], miten ScalaFiddle saavutti nopean Scala-koodin kääntämisen tallettamalla kaiken välimuistiin.

ScalaFiddle jaetaan sisäisesti reitittäjään (router) ja useaan kääntäjään (compiler) (kuva 5).



Kuva 5. ScalaFiddle on jaettu reitittäjään ja useampaan kääntäjään [10].

Reitittäjä saa kaikki kääntämispyynnöt ja valitsee optimaalisimman saatavilla olevan kääntäjän Scala-version, kirjastoriippuvuuksien ja aktiivisuuden perusteella.

Normaalisti kääntäjä hakee kirjastot JAR-paketeista (Java Archive). ScalaFiddle taas irroittaa niistä .class ja .sjsir-tiedostot, jotka se tiivistää ja tallettaa mukautettuun FlatFileSystemiin. FlatFileSystem on iso tiedosto muistissa, joka sisältää kaikki .class ja .sjsir-tiedostot kaikkien kirjastojen JAR-paketeista. Kääntäjät voivat viitata tähän isoon tiedostoon, jolloin ne voivat käyttää samoja resursseja.

4.2 JavaScriptin suorittaminen

JavaScriptin suorittamiseksi löytyi muutama tapa [11]:

- ScriptEngine ajaa JavaScript -koodia Java-virtuaalikoneessa (JVM).
- ProcessBuilder käyttää ulkopuolista prosessia kuten Node.js.
- Scala.js:n eval-metodi suorittaa nykyisessä JS-prosessissa.

Scala.js eval voidaan sulkea heti pois, koska StarChat pyörii JVM-prosessissa eikä JS-prosessissa. ProcessBuilder voi olla toimiva, mutta ScriptEngine tarjoaa paljon monipuolisemman ja paremman rajapinnan JavaScriptin suorittamiseksi. Lisäksi StarChat ei tule riippuvaiseksi ulkopuolisesta JS-prosessista, ja siksi ScriptEngine tuli käyttöön. Javan ScriptEngine käyttää Nashorn-moottoria JavaScriptin suorittamiseen.

4.2.1 ScriptEngine

JavaScriptiä voidaan suorittaa Java virtuaalikoneella käyttämällä Nashorn JavaScript -moottoria [11]. Sitä käytetään Javan ScriptEngine-rajapinnan avulla. Moottori alustetaan kutsumalla ScriptEngineManager:in getEngineByName-metodia seuraavalla tavalla (esimerkkikoodi 7):

```
import javax.script._

val manager = new ScriptEngineManager(getClass.getClassLoader)
val engine: ScriptEngine with Compilable = manager
  .getEngineByName("nashorn") match {
  case engine: ScriptEngine with Compilable => engine
  case _ => throw new Exception("Engine is not compilable")
}
```

```
val script: String = "1+x"
val compiledScript: CompiledScript = engine.compile(script)
```

Esimerkkikoodi 7. JavaScriptin kääntäminen ScriptEnginellä

Edellä on käytetty myös vapaavalintaista Compilable-rajapintaa, jonka avulla voidaan kääntää skripti kerran ja suorittaa se monta kertaa ilman uudelleenkääntämistä. Käännös tapahtuu kutsumalla compile-metodia, joka ottaa skriptin tavallisena merkkijonona.

Seuraavassa esimerkkikoodi 8:ssa esitetään, kuinka skripti suoritetaan kutsumalla käännetyn skriptin eval-metodia. Metodiin voidaan liittää bindings-olio, joka asettaa JavaScript-moottorin globaaliin kontekstiin muuttujia, joihin voidaan viitata skriptissä. eval-metodi palauttaa sen, mitä skripti palauttaa.

```
val bindings: Bindings = engine.createBindings()
bindings.put("x", 4)
val result: Double = compiledScript.eval(bindings)
  .asInstanceOf[Double]
```

Esimerkkikoodi 8. Bindings-esimerkki

Result-muuttujan arvoksi tulee ”5” skriptillä ”x+1”, kun asetetaan bindings muuttujassa ”x” arvoksi ”4”.

Scala-koodista voidaan myös kutsua skriptissä määriteltyjä funktioita käyttämällä valinnaista Invocable-rajapintaa. Rajapinta tarjoaa invokeFunction-metodin, joka ottaa funktion-nimen merkkijonona ja parametrit objekteina.

ScriptEngine mahdollistaa myös Scalan luokkien ja objektien käytön skriptissä. Luokkien tai objektien hakemiseen käytetään Java.type-metodia, joka ottaa luokkaa vastaavan polun merkkijonona. Esimerkkikoodi 10:ssä esitetään, kuinka JavaScript-koodissa luodaan Scala-luokan (esimerkkikoodi 9) ilmentymä.

```
package my.package

case class MyScalaClass(str: String)
```

Esimerkkikoodi 9. Yksinkertainen Scala-luokka

```
val compiledScript = compilerEngine.compile(
  """
    |var MyScalaClass = Java.type("my.package.MyScalaClass")
    |MyScalaClass.apply("my own string")
    |""".stripMargin
)
val res: MyScalaClass = compiledScript.eval().asInstanceOf[MyScalaClass]
```

Esimerkkikoodi 10. Scala-luokan ilmentymän luonti JavaScript-koodissa

5 Toteutus

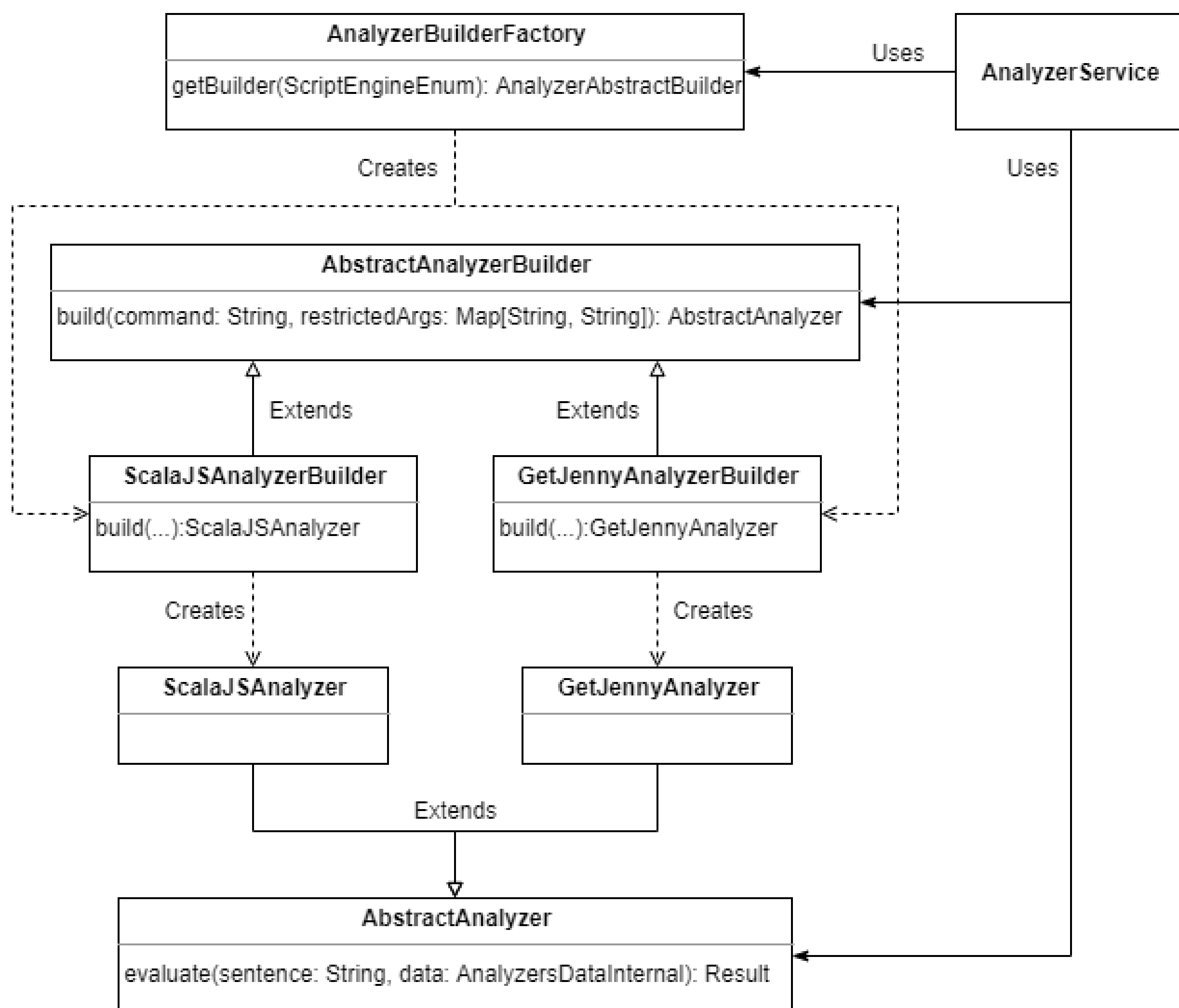
Scala.js-tuen toteutuksessa säilytin vanhat rajapinnat analysaattoreiden luomiseen ja evaluoimiseen, mitä DSL:ssä on käytetty. Ainoastaan lisättiin abstraktit kerrokset rajapinnoille.

Analysaattoreilla on kaksi vaihetta: rakennus ja evaluointi. Rakennusvaiheessa skripti luetaan merkkijonosta, josta rakennetaan analyzer-olio, jolla on evaluate-metodi. Evaluate-metodi ottaa parametrikseen käyttäjän viestin sekä AnalyzersDataInternal-lukan ilmentymän dataolion, joka sisältää keskusteluun liittyvää dataa, kuten

- konteksti, jossa on botin indeksin nimi, sekä tilan nimi, johon analysaattori on kytketty
- listan tiloista, jossa keskustelu on käynyt
- taulukon, jossa on poimittua dataa keskustelusta, esimerkiksi käyttäjän nimi tai sähköposti.

Analyzer-oliot talletetaan välimuistiin evaluointia varten. Analyzer-oliot evaluoidaan aina, kun botilta kysytään uutta vastausta. Analyzerit palauttavat Result-luokan ilmentymän, joka sisältää numeropisteytyksen sekä dataolion.

StarChatissä analysaattoreista vastaa AnalyzerService, joka rakentaa analysoijat ja tallettaa ne muistiin evaluointia varten. Jotta se saadaan käsittelemään myös Scala.js-analyaattoreita, analysaattoreiden käsittely muutettiin Gamman abstract factory -suunnittelumallin kaltaiseksi. [12, s. 78]



Kuva 6. Luokkakaavio Scala.js-integraatiosta.

Uusien skriptikielien lisäämistä varten luotiin abstraktit kerrokset rakentajille (builder) ja analysaattoreille (analyzer). Rakentajilla on build-metodi, joka ottaa merkkijonoparametrina skriptin, josta se luo uuden analysaattorin.

AnalyzerServiceä muutettiin lukemaan skriptin tyyppi skriptin ensimmäiseltä riviltä ja hakemaan tyypin perusteella sitä vastaava rakentaja käyttäen AnalyzerBuilderFactory-tehdasta (kuva 6). Builderit tuottavat AbstractAnalyzer-rajapintaa käyttäviä analysaattoreita, joiden evaluate-metodia AnalyzerService kutsuu vastauksen hakemisen yhteydessä.

5.1 Scala.js:n ajonaikainen kääntäminen JavaScriptiksi

Scala.js-koodin kääntäminen JavaScriptiksi oli hankalaa eikä ohjeita sen tekemiseen löytynyt. Toteutus on suurimaksi osaksi otettu luvusta 4.1 mainitusta ScalaFiddle-projektista, josta poistettiin tähän projektiin tarpeettomia ominaisuuksia. ScalaFiddlen Scala.js-kääntäjän toteutuksesta ei löytynyt dokumentaatiota, joten kääntäjän luomisessa katsottiin mallia lähdekoodista [13].

Scala.js:n kääntäminen tapahtuu luomalla ”scala.tools.nsc.Global”-luokka, joka tarvitsee kääntämiseen tarvitut peruskirjastot, kuten boot-tiedostot, scala-js- ja scala-kirjaston sekä Scala.js-lisäosan ja asetukset kohdekansion määrittämiseksi, johon käännetyt tiedostot tulevat. Kääntäjästä luodaan uusi sisäluokka Run, jolla käännetään Scala.js-tiedostot compileFiles-metodin avulla.

Kääntäjä tuottaa nyt sjsir-päätteisiä tiedostoja, jotka linkitetään JavaScript-muotoon Scala.js-linkerillä. Linker ottaa kääntäjän tuottamat tiedostot sekä peruskirjastot. Linkerille pitää määritellä myös main-metodi, ja minkä objektin sisällä main-metodi on, jotta sen tuottaman JavaScriptin voi suorittaa.

Jos käännöksen tai linkityksen aikana tulee virheitä, samaa linkerin tai kääntäjän ilmentymää ei voida käyttää toista kertaa, jolloin se pitää alustaa uudelleen.

5.2 Scala.js-analysaattoreiden ajonaikainen suoritus ja muodostaminen

Scala.js-analysaattoreiden suorittamiseksi skriptissä tulee määrittää main-metodi, jota kutsutaan ensimmäisenä, kun skriptiä evaluoidaan. Main-metodi pitää olla ennalta määritetyn nimisessä objektissa, joka määritellään linkerille ModuleInitializerilla.

Analysoijan tulee palauttaa Result-olio, joka on määritelty StarChatissä. Nashorn sallii Java/Scala-luokkien käytön, jolloin pääohjelman Result-olio voidaan luoda ja palauttaa suoraan skriptistä. Scala.js-koodissa tämä tapahtuu seuraavalla tavalla:

```
import scala.scalajs.js.Dynamic.{global => g}

val AnalyzersDataInternal = g.Java.applyDynamic("type")("com.getjenny.ana-
lyzer.expressions.AnalyzersDataInternal")
val Result = g.Java.applyDynamic("type")("com.getjenny.analyzer.expres-
sions.Result")
Result.applyDynamic("apply")(1.0, AnalyzersDataInternal.applyDynamic("ap-
ply")())
```

Esimerkkikoodi 11. Pääohjelman luokkien käyttö Scala.js-koodissa

JavaScriptissä pääohjelmaluokkien hakemiseen käytetään Java.type(”polku luokkaan”) -tyyliä. Scala.js-koodissa (esimerkkikoodi 11) Java-objekti haetaan global-objektista, jonka type-metodia kutsutaan suorittamalla applyDynamic-metodi, jolle annetaan parametriksi kutsuttavan metodin nimi. Sitten currying-menetelmällä voidaan antaa luokan polku parametriksi. Tätä applyDynamic-metodia käytetään aina, kun käsitellään pääohjelman Scala.js:n Dynamic-luokan ilmentymiä.

Pääohjelman ja skriptin kommunikointi oli hankalaa. Koska main-metodi ottaa parametrikseen listan merkkijonoja, olioargumentteja ei voitu antaa suoraan main-metodille. Ensimmäisessä versiossa skriptin tarvitsemat olioargumentit ja vastaus talletettiin JavaScriptin globaaliin kontekstiin, jota käyttäen pääohjelma ja skripti pystyivät välittämään objekteja toisilleen. Globaalista kontekstista lukeminen ja tallettaminen osoittautui todella aikaa vieväksi ja analysaattori käytti suurimman osan ajasta siihen.

Päädyin lopulta ratkaisuun, jossa skripti evaluoidaan kerran rakentamisvaiheessa, jossa se tallettaa analyzer-metodin JavaScriptin globaaliin kontekstiin (esimerkkikoodi 12), josta pääohjelma hakee ja tallettaa tämän välimuistiin. Suorittamisvaiheessa pääohjelma voi kutsua tätä metodia suoraan invokeMethod-komentoa käyttämällä, johon voi liittää olioargumentit. Tämä ratkaisu osoittautui noin viisi kertaa nopeammaksi kuin aikaisempi toteutus.

```
type/SCALAJS
import scalajs.js
import scala.scalajs.js.Dynamic.{global => g}

object ScalaJSAnalyzer {
  def main() = {
```

```
    g.analyzer = (sentence: String,
                  analyzersDataInternal: js.Dynamic,
                  restrictedArgs: js.Dynamic) => {

      val Result = g.Java.applyDynamic("type")("com.getjenny.analyzer.expres-
sions.Result")
      val keyword = "password"
      val rx = {"""\b""" + keyword + """\b"""}.r
      val freq = rx.findAllIn(sentence).toList.length
      val queryLength = """\S+""".r.findAllIn(sentence).toList.length

      val score = if (queryLength > 0)
        freq.toDouble / queryLength.toDouble
      else
        0.0

      Result.applyDynamic("apply")(score, analyzersDataInternal)
    }
  }
}
```

Esimerkkikoodi 12. Scala.js-analysaattori

5.3 Scala.js-analysaattoreiden nopeus

Nopeustestissä verrataan DSL- ja Scala.js-analysaattoreita, jotka etsivät regexiä käyttäen avainsanoja tekstistä. Scala.js-analysaattoriskriptinä käytetään esimerkkikoodi 12 ja DSL-skripti oli vOneKeyword(”password”). Testauksessa analysaattorit suoritettiin 1000 kertaa. DSL-analysaattorin keskimääräinen nopeus oli noin 53,1 µs ja Scala.js 1041.3 µs. DSL oli siis keskimäärin noin 20 kertaa nopeampi kuin Scala.js-analysaattori.

6 Yhteenveto

Työn tavoitteena oli pystyä kirjoittamaan StarChatin analysaattoreita Scala.js-ohjelmointikielellä nykyisen DSL:n lisäksi.

Työssä käytettiin ScalaFiddle-projektia hyödyksi Scala.js-koodin kääntämiseen JavaScriptiksi. JavaScript suoritettiin Javan ScriptEngine-rajapintaa käyttäen Nashorn-moottorissa. Scala.js-analysaattorit saatiin toimimaan StarChatissä DSL:n rinnalla abstract factory -suunnittelumallia käyttäen.

Scala.js-analysaattoreiden kirjoittaminen ei ollut kovin intuitiivista. Pääohjelman Scala-luokkia käsiteltäessä Scala.js-koodissa täytyi käyttää applyDynamic-metodia näiden metodien kutsumiseen. Tämä teki koodista sekavaa ja vaikeasti luettavaa.

Opinnäytetyön toteutus osoittautui vain hitaammaksi ja hankalammaksi käyttää verrattuna DSL:ään. Parannuksena pitäisi enemmän tutkia tapoja käyttää Scala-luokkia Scala.js-analysaattoreissa. Ehkä jopa tehdä jonkinlainen kirjasto, jota voitaisiin käyttää Scala.js-skriptissä, jossa olisi hyödyllisiä toimintoja analysaattoreiden tekemiseen.

Toinen parannusidea olisi parantaa analysaattoreiden uudelleenkäytettävyyttä. Ensiksi tulee mieleen, että jos Scala.js-skriptillä kirjoitettaisiin vain uusia DSL-skriptikielen atomeita, niitä voitaisiin käyttää DSL-skripteissä. Tällöin Scala.js-atomeita voitaisiin uudelleen käyttää tilojen analysaattoreissa ja pystyttäisiin luomaan uusia toimintoja analysaattoreihin ilman StarChatin kääntämistä uudelleen.

Lähteet

1 GetJenny Oy. Verkkosivusto. <https://github.com/GetJenny/starchat>. Luettu 8.11.2020.

2 Turva. Verkkosivusto. <https://www.face-book.com/turva/posts/10156693351853597>. Luettu 13.11.2020.

3 LocalTapiola. Verkkosivusto. <https://www.getjenny.com/customers>. Luettu 8.11.2020.

4 GetJenny Oy. Verkkosivusto. <https://getjenny.github.io/starchat-doc/about/>. Luettu 12.11.2020.

5 GetJenny Oy, Verkkosivusto. <https://getjenny.github.io/starchat-doc/#scalability>. Luettu 13.11.2020.

6 GetJenny Oy, Verkkosivusto. <https://getjenny.github.io/starchat-doc/#analyzer>. Luetu 13.11.2020.

7 Wikipedia. Verkkosivu. <https://fi.wikipedia.org/wiki/Scala_(ohjelmointikieli)>. Luettu 9.11.2020

8 ScalaFiddle, Verkkosivusto. <https://scalafiddle.io/>. Luettu 9.11.2020.

9 ScalaFiddle, Verkkosivusto. <https://github.com/scalafiddle>. Luettu 9.11.2020.

10 Chrons, Otto. Artikkeli. <https://medium.com/@otto.chrons/what-makes-scalafiddle-so-fast-9a3edf33ed4d>. Luettu 10.11.2020.

11 Draper, Paul. Verkkosivusto. <https://stackoverflow.com/questions/36764639/how-to-run-javascript-code-from-within-scala-jvm/36783046#36783046>. Luettu 11.10.2020.

12 Gamma, Erich. 1995. Design Patterns: Elements of Reusable Object-Oriented Software.

13 ScalaFiddle. Verkkosivu. <https://github.com/scalafiddle/scalafiddle-core>. Luettu 10.11.2020.