TAMPEREEN TEKNILLINEN YLIOPISTO

MIKKO AHLROTH

ELIXIR-OHJELMOINTIKIELEN SOVELTUVUUS VERKKOPALVELUIDEN TOTEUTUKSEEN

Diplomityö

Tarkastaja: professori Tommi Mikkonen
Tarkastaja ja aihe hyväksytty Tieto- ja sähkötekniikan tiedekunta-neuvoston kokouksessa 4. touko-kuuta 2016

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
AHLROTH, MIKKO: Elixir-ohjelmointikielen soveltuvuus verkkopalveluiden toteutukseen
Diplomityö, 58 sivua, 2 liitesivua
Joulukuu 2016
Pääaine: Hajautetut ohjelmistot
Tarkastaja: professori Tommi Mikkonen
Avainsanat: Elixir, Erlang, BEAM, Phoenix Framework, hajautus, verkkopalvelut, Internet

Internetin käyttäjämäärien kasvaessa myös verkkopalveluille asetetaan uusia haasteita. Mobiililaitteiden yleistyminen ja asioiden Internet kiihdyttävät kehitystä entisestään ja palveluiden tulee voida skaalautua sen mukaisesti. Palveluissa ovat yleistyneet myös WebSocket-teknologian mahdollistamat tiedon reaaliaikaiset päivitysominaisuudet. Verkkopalvelun toteutukseen valittavalla teknologialla tulee voida vastata näihin kehityksiin sekä muihin ohjelmistojen laatuvaatimuksiin ilman, että toteuttamisesta tulee liian monimutkaista.

Elixir on uusi Erlangiin pohjautuva ja sen virtuaalikoneella BEAMilla suoritettava funktionaalinen kieli, jonka tavoitteena on hyödyntää virtuaalikoneen tarjoamia ominaisuuksia, mutta parantaa niitä uudella syntaksilla, metaohjelmoinnilla ja kehitystyötä helpottavilla työkaluilla. Elixirille on toteutettu muun muassa sovelluskehys Phoenix Framework, joka on suunniteltu verkkopalveluiden pohjaksi. Itse kielen soveltuvuudesta verkkopalveluiden toteutukseen ei ole kuitenkaan vielä saatavilla tietoa.

Tässä diplomityössä toteutettiin Elixiriä ja sovelluskehys Phoenix Frameworkiä käyttäen esimerkkiverkkopalvelu Code::Stats. Toteutetulle palvelulle tehtiin suorituskykymittauksia, jotta voitiin verrata Elixirin suorituskykyä PHP:llä toteutettuun vastaavaan vertailupalveluun. Palvelun laatua arvioitiin myös manuaalisesti ohjelmistojen laatuvaatimuksia käyttäen. Näiden lisäksi kielen yleiseen arviointiin käytettiin toteuttamisen tuomaa ohjelmointikokemusta.

Toteutetun arvioinnin perusteella Elixir on ohjelmoijalle miellyttävä kieli käyttää. Kielen funktionaalisuus ja ominaisuudet kuten datan muuttumattomuus ja hahmonsovitus auttavat toteuttajaa tekemään ohjelmakoodista luotettavaa, kun niitä on oppinut hyödyntämään. Syntaksi on yksinkertainen oppia ja sen erityisominaisuudet, kuten putket, auttavat selkeän ja luettavan lopputuloksen aikaansaamisessa. Suorituskykymittaukset osoittavat, että Elixirillä toteutettu palvelu on PHP:llä toteutettua vastaavaa nopeampi kaikilla testatuilla osa-alueilla. Kielen mukana tulevien työkalujen avulla projektityö kuten riippuvuuksien hallinta on helppoa. Ongelmana on kuitenkin kielen uutuus ja ekosysteemin pienuus verrattuna muihin verkkopalveluiden toteutukseen käytettyihin kieliin.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
AHLROTH, MIKKO: Suitability of the Elixir programming language for web development, an evaluation
Master of Science Thesis, 58 pages, 2 appendix pages
December 2016
Major: Distributed software
Examiner: Professor Tommi Mikkonen
Keywords: Elixir, Erlang, BEAM, Phoenix Framework, distribution, web services, Internet

As the Internet gets ever more popular, web services are faced with new challenges. The so-called Internet of Things and the proliferation of mobile devices are only accelerating the progress and web services must be able to scale accordingly. New techniques, such as live updates, have also become possible with the use of the WebSocket protocol. The technology used for implementing modern web services must be able to provide an answer to both these new developments and software quality requirements without making implementation too complex.

Elixir is a new functional programming language that is based on Erlang and runs on its virtual machine, BEAM. Its purpose is to make use of the virtual machine's features, while improving on them with a new syntax, meta-programming features, and tools to ease the development process. A framework for implementing web services called Phoenix Framework has been implemented for the language, but the suitability of the language itself for web development has yet to be evaluated.

In this Master of Science Thesis, an example web service called Code::Stats was implemented, using Elixir and Phoenix Framework. The performance of the resulting service was measured and compared to an identical service implemented in the PHP programming language. The quality of the service was manually assessed using the software quality requirements. In addition, the experience gained in the implementation process was used for the general evaluation of Elixir.

Based on the evaluation, Elixir is a pleasant language to use for the programmer. The functional paradigm and properties of the language like immutability of data and pattern matching help the developer write reliable code, once their proper use has been learned. Elixir's syntax is simple to learn and its special features, like pipes, help create a clear and readable result. Performance measurements show, that the service implemented in Elixir is faster than a comparable PHP implementation in all areas tested. The tools provided with the language ease regular project management work, such as handling dependencies. The main issue is the newness of the language and the small size of the ecosystem, when compared to other languages.

ALKUSANAT

Tämä diplomityö ei syntynyt tyhjiössä, vaan olen saanut prosessin aikana apua useilta eri tahoilta. Haluan kiittää erityisesti seuraavia henkilöitä heidän tuestaan diplomityöprosessin aikana:

professori Tommi Mikkosta erinomaisesta ohjauksesta, loppumattomasta kannustuksesta, positiivisesta asenteesta, tarkoista huomioista ja liikojen rönsyjen katkomisesta,

työtoveriani Antti Pulakkaa lukemattomista neuvoista, viiltävästä analyysistä, rohkaisun sanoista, sopivasta painostamisesta ja oman ajan käyttämisestä työn hyväksi,

sekä rakasta vaimoani Jenniä punakynäntäyteisistä sivuista, vahvoista mielipiteistä, työnteon esteiden poistamisesta ja ehtymättömästä rakkaudesta.

Ohjelmointi on jättiläisten olkapäillä seisomista. Myös tämän diplomityön ohjelmointiosuus pohjautuu minua viisaampien ohjelmoijien kovan työn tuloksiin. Tämän vuoksi haluan osoittaa kiitokseni seuraaville henkilöille: José Valim, Chris McCord, Eric Meadows-Jönsson, Joe Armstrong, Robert Virding, Mike Williams, David Whitlock, James Fish, Loïc Hoguin sekä Lau Taarnskov. Lisäksi haluan kiittää koko Elixir-yhteisöä ja erityisesti IRC-verkko Freenoden kanavaa #elixir-lang heidän antamastaan avusta niin minulle kuin muillekin aloitteleville Elixir-ohjelmoijille.

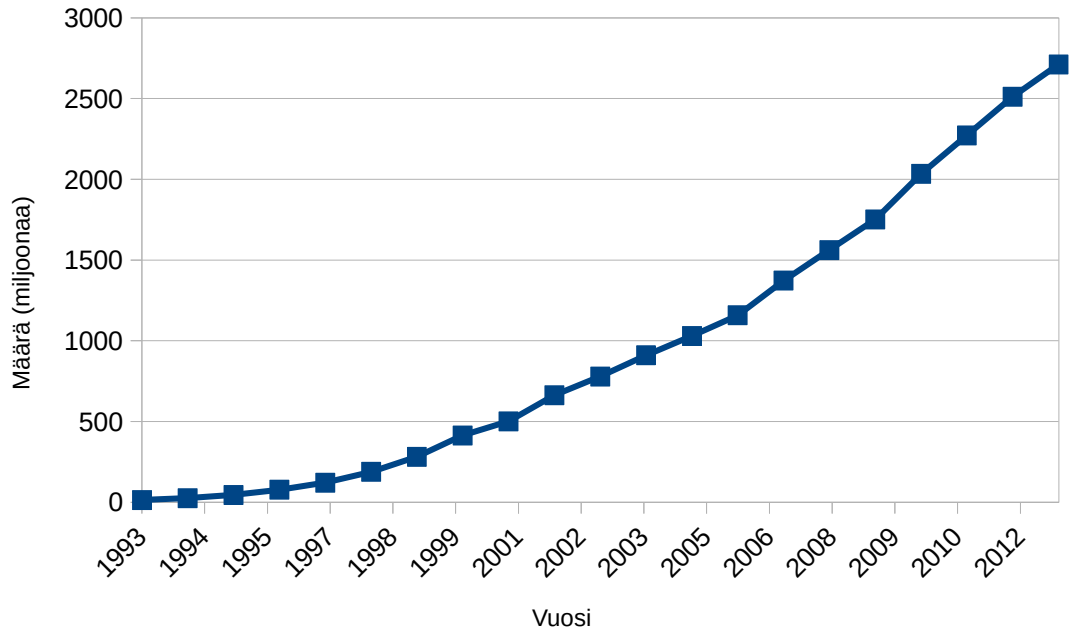
Tampereella 21.11.2016

Mikko Ahlroth

SISÄLLYS

1 JOHDANTO

Verkkopalvelulla tarkoitetaan Internetissä ja yleensä myös WWW:ssä käytettävää järjestelmää, joka tarjoaa käyttäjälleen jonkin palvelun. Näitä voivat olla esimerkiksi perinteiset verkkosivustot, pikaviestimet, kuva-arkistot ja sosiaalisen median palvelut. Verkkopalveluiden käyttäjämäärät kasvavat jatkuvasti. Internetin käyttäjien määrä maailmassa on kasvanut noin kahdella miljardilla kymmenen viime vuoden aikana ja kasvaa edelleen noin 200 miljoonan käyttäjän vuosivauhdilla [1]. Kuvassa 1 on havainnollistettu tätä kasvua.



Kuva 1: Internet-käyttäjien määrän kehitys kymmenen vuoden ajalta. Perustuu lähteeseen [1].

Vuosikymmenen aikana myös mobiililaitteilla verkkopalveluita käyttävien ihmisten määrä on moninkertaistunut, ja eräiden lähteiden mukaan jopa ylittänyt perinteisten työpöytäkoneiden käyttäjämäärän [2]. Vaikka kasvu onkin hidastunut perinteisillä työpöytäalustoilla, avaa esimerkiksi edullisten älypuhelinten myynti kehittyvillä markkinoilla verkkopalvelut täysin uusille käyttäjäryhmille. Osassa verkkopalveluista käyttäjien määrää kasvattaa entisestään myös niin kutsuttu *asioiden internet* (*Internet of Things – IoT*), joka tarkoittaa erilaisten koneiden ja laitteiden kytkemistä Internetiin [3].

Kasvavat käyttäjämäärät asettavat verkkopalveluille erityisiä vaatimuksia, ja ne tulee huomioida jo palveluiden suunnitteluvaiheessa. Samalla toteutustekniikoiden on

kyettävä skaalautumaan tarpeen mukaisesti. Tämä ei saa kuitenkaan aiheuttaa toteuttamisen liiallista monimutkaistumista, sillä se voi johtaa tehottomampaan työskentelyyn ja altistaa virheille. On siis olennaista löytää toteutustekniikoita, joilla voidaan saavuttaa vaadittu suorituskyky ja skaalautuminen säilyttäen kuitenkin mahdollisuudet yksinkertaiseen ja tehokkaaseen toteuttamiseen.

Viime vuosina tällaisia teknologioita – ohjelmointikieliä ja sovelluskehyksiä – on noussut esiin useita, muun muassa Ruby on Rails, Go, Scala, Node.js sekä Clojure. Kukin niistä pyrkii vastaamaan haasteisiin omilla painotuksillaan, mutta yhä useammissa on yhtenä keskeisenä tekijänä hajautus. Vuonna 2014 joukkoon liittyi Elixir, joka on Erlangin päälle rakennettu funktionaalinen ohjelmointikieli skaalautuvien ja ylläpidettävien ohjelmien toteuttamiseen. Kieli hyödyntää Erlangin virtuaalikonetta, mahdollistaen yksinkertaisen ja kevyen hajautuksen, skaalautuvuuden ja virhesietoisuuden. [4]

Tässä diplomityössä tutkitaan Elixirin soveltuvuutta verkkopalveluiden toteutukseen toteuttamalla esimerkkipalvelu, joka käyttää hyväkseen kielen tarjoamia ominaisuuksia. Soveltuvuutta arvioidaan mittaamalla toteutettua palvelua verkkopalvelujen laatuvaatimusten kautta. Tämän lisäksi pohditaan sitä, millaista toteutustyö oli kehittäjän näkökulmasta, ottaen huomioon toteuttamisessa vastaan tulleet ongelmat, valmiiden työkalujen saatavuus ja muita toteuttamistyön tehokkuuteen vaikuttavia asioita.

Työ rajataan soveltuvuuden arviointiin toteutetun palvelun ja siihen käytettyjen työkalujen osalta, ja esimerkiksi muiden kielellä toteutettujen ohjelmistojen ja sovelluskehysten arviointi jätetään tarkastelun ulkopuolelle. Koska palvelimella suoritettavien ohjelmistojen toteuttamiseen tarkoitetulla kielellä ei ole suoraa vaikutusta käyttöliittymien sisältöihin, työssä ei myöskään kiinnitetä huomiota palvelun käyttöliittymien toteuttamiseen tai arviointiin,vaan tarkastelu rajataan Elixirillä toteutettavaan taustajärjestelmään. Koska kieli on uusi, aiheesta ei ole vielä saatavilla tieteellistä tutkimustietoa, vaan arviot rajoittuvat lähinnä koulutusmateriaaleihin ja blogikirjoituksiin. Tämän vuoksi mahdollisuudet tulosten vertailemiseen aiempien tutkimusten kanssa ovat rajalliset.

Luvussa 2 käydään läpi verkkopalveluiden toteuttamisen erityispiirteitä ja laatuvaatimuksia, joita ne asettavat toteutustekniikoille. Samalla esitellään muutamia yleisimpiä tekniikoita ja sitä, kuinka ne vastaavat vaatimuksiin. Luvussa 3 syvennytään työn kohteena olevaan Elixir-ohjelmointikieleen, siihen läheisesti liittyviin muihin teknologioihin ja tyypilliseen Elixirillä toteutetun verkkopalvelun rakenteeseen. Näitä tekniikoita hyödyntäen toteutettava verkkopalvelu ja sen teknologiavalinnat esitellään luvussa 4. Luvussa 5 kerrotaan tarkemmin mittareista, joilla toteutuksen onnistuneisuutta ja samalla Elixirin soveltumista arvioidaan. Tehdyn arvioinnin tulokset ja tarkat mittausarvot käydään läpi luvussa 6. Lopuksi luvussa 7 vedetään yhteen työn keskeisimmät tulokset, esitellään päätelmät ja pohditaan mahdollisia jatkokehitysvaihtoehtoja.

2 VERKKOPALVELUT JA NIIDEN TOTEUTTAMINEN

Ohjelmistoalalla on muihin teollisuuden aloihin verrattuna lyhyt historia. Ensimmäiset korkean tason ohjelmointikielet kehitettiin 1940–1950-lukujen aikana. Verkkopalveluiden osalta kehitys käynnistyi laajemmin vasta 1990-luvulla, WWW:n yleistyessä. Siitä lähtien palveluiden käyttäjämäärät ovat kasvaneet räjähdysmäisesti, muuttaen samalla toteutustekniikoiden vaatimuksia. [1, 5]

Käytössä olevat tekniikat ovat siis kaikki nuoria ja syntyneet alati muuttuvaan ympäristöön. Ajan myötä esiin on kuitenkin noussut tiettyjä suunnittelumalleja ja rakenteita ratkaisemaan verkkopalveluiden yleisimpiä ongelmia. Haasteena on vertailla toteutustekniikoita toisiinsa ja löytää niiden vahvuusalueet, jotta niitä voidaan soveltaa oikeissa tilanteissa. Vertailun apuna voidaan käyttää järjestelmälle asetettuja laatuvaatimuksia.

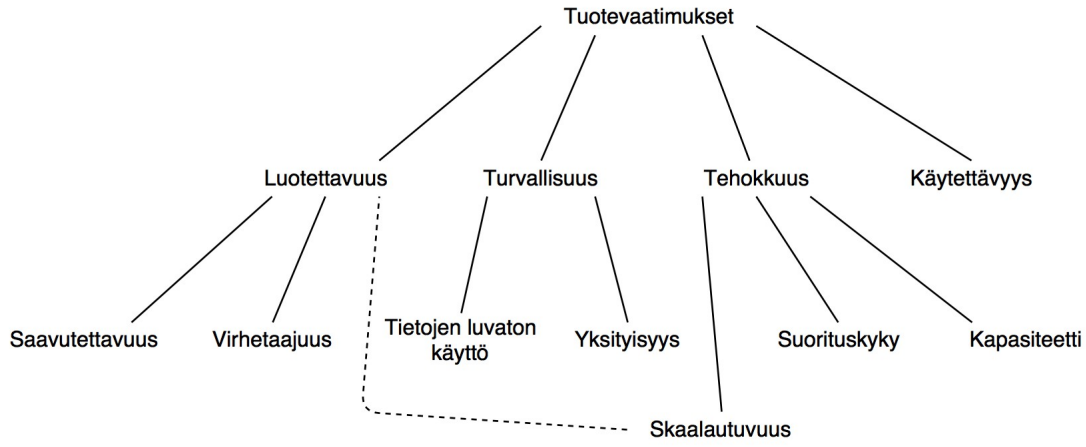
Tässä luvussa käydään kohdassa 2.1 läpi verkkopalveluiden toteuttamiseen liittyviä laatuvaatimuksia ja erityispiirteitä. Kohdassa 2.2 syvennytään tehokkuusvaatimukseen liittyviin reaaliaikaisiin päivityksiin ja niiden toteuttamiseen verkkopalvelussa. Lopuksi kohdassa 2.3 käydään läpi yleisiä verkkopalveluiden toteutukseen liittyviä tekniikoita ja sitä, miten ne vastaavat mainittuihin vaatimuksiin.

2.1 Verkkopalveluiden laatuvaatimukset

Ohjelmistoja kehitettäessä ja arvioitaessa niille asetetaan tiettyjä laatuvaatimuksia. Vaatimusmäärittely on yksi prosessi, jolla voidaan selvittää vaatimukset, joihin tietyn järjestelmän tulee vastata. Tällaiset vaatimukset voidaan jakaa van Lamsweerden mukaan *toiminnallisiin* ja *ei-toiminnallisiin* laatuvaatimuksiin. Toiminnalliset vaatimukset määrittävät sen, mitä järjestelmän tulee tehdä ja ne ovat järjestelmästä riippuvaisia, kun taas ei-toiminnalliset vaatimukset määrittävät järjestelmän yleispiirteet ja esittävät rajoituksia sille, miten toiminnalliset vaatimukset tulee toteuttaa. [6]

Ei-toiminnallisten laatuvaatimusten luokittelusta on useita näkemyksiä, mutta Sommervillen jaottelun mukaan ne jaetaan *prosessivaatimuksiin*, *ulkoisiin vaatimuksiin* j a *tuotevaatimuksiin* [7]. Prosessivaatimuksiin kuuluu ohjelmiston tuotantoprosessiin liittyviä asioita, kuten aikataulut ja käytettävät standardit. Ulkoiset vaatimukset sisältävät ohjelmiston viitekehykseen liittyviä seikkoja, kuten yhteensopivuus muiden järjestelmien kanssa ja lakitekniset vaatimukset [7]. Koska molemmat edellä mainitut liittyvät itse ohjelmiston rakenteen ulkopuolisiin asioihin, keskitytään tässä työssä tuotevaati-

muksiin, jotka määrittävät järjestelmän toimintaa ja rakennetta eri näkökulmista. Näitä ovat *luotettavuus*, *turvallisuus, tehokkuus* sekä *käytettävyys*. Ne alakohtineen ja suhteineen on esitetty kuvassa 2.



Kuva 2: Ohjelmistojen ei-toiminnalliset tuotevaatimukset Sommervillen jaottelun mukaan. Perustuu lähteeseen [7].

Ensimmäinen vaatimuksista, luotettavuus, tarkoittaa järjestelmän kykyä suorittaa vaaditut tehtävänsä. Toiminnot tulee voida suorittaa määriteltyjen olosuhteiden vallitessa ja tietyn ajan kuluessa. Luotettavuuteen kuuluvat alakohtina *saavutettavuus*, eli tässä tapauksessa se, että palvelu on asiakkaiden saatavilla ja vastaa pyyntöihin, sekä *virhetaajuus*, joka määrittää kuinka usein palvelu saa epäonnistua pyyntöjen käsittelemisessä loppukäyttäjän näkökulmasta. [7]

Verkkopalvelun taustajärjestelmän luotettavuus liittyy läheisesti myös tehokkuuteen. Mikäli järjestelmä ei pysty käsittelemään saapuvia pyyntöjä tarpeeksi nopeasti, alkaa niitä kertyä palvelimen pyyntijonoon odottamaan käsittelyä, minkä vuoksi käyttäjien kokemat vasteajat kasvavat. Tämä johtaa lopulta tilanteeseen, jossa viiveet kasvavat liian suuriksi ja pyyntöjä täytyy jättää käsittelemättä. Tällöin tietyt toiminnot jäävät suorittamatta, kuten tiedot tallentamatta tai viestit välittämättä, mikä johtaa käyttäjän näkökulmasta huonoon luotettavuuteen. Luotettavuuteen verkkopalvelussa liittyy myös virhetilanteiden käsittely siten, että jos niitä ei voida korjata, aiheutuisi niistä haittaa mahdollisimman pienelle käyttäjämäärälle. Esimerkiksi yhden pyynnön käsittelyssä tapahtuva virhetilanne ei ihannetapauksessa vaikuta muihin käynnissä oleviin pyyntöihin.

Turvallisuuden tulee olla jokaisen Internetissä toimivan palvelun suunnittelun yhtenä lähtökohtana. Turvallisuus vaatimuksena tarkoittaa järjestelmän ja sen tietojen luvattoman käytön estämistä, sekä järjestelmän toiminnan turvaamista silloinkin, kun siihen kohdistuu tahallisia tai tahattomia haittoja [7]. Verkkopalvelun omien tietojen suojaamisen lisäksi usein tulee kysymykseen käyttäjien lisäämien tietojen yksityisyys. Tiedot tulee suojata ulkoisilta urkkijoilta, esimerkiksi salaamalla kommunikaatio asiakkai-

den ja palvelinten välillä sekä salaamalla pysyvästi tietokantaan tallennetut tiedot, mutta ne tulee myös rajoittaa järjestelmän sisäisesti oikeille henkilöille.

Viime vuosina ovat tietoisuuteen nousseet myös valtiollisten tahojen asettamat tietoturvauhat, erityisesti tietovuotaja Edward Snowdenin paljastusten myötä. Paljastukset ovat johtaneet tietyissä palveluissa salauksen käyttöön myös palvelun sisäisesti, eli siten, etteivät myöskään palvelun ylläpitäjät voi lukea käyttäjien tuottamia sisältöjä. Tiedossa olevien valtiollisten tahojen suorittaman urkinnan vuoksi palveluiden ylläpitäjät ovat joutuneet lisäksi pohtimaan, missä maissa verkkopalvelimia on turvallista pitää. [8]

Tuotevaatimuksista mitattavin ja verkkopalvelulle haastavin on tehokkuusvaatimus. Tehokkuus tarkoittaa järjestelmän toiminnan nopeutta tietyissä tilanteissa, eli *suorituskykyä* – yksittäisten pyyntöjen käsittelyn nopeutta – sekä *kapasiteettia* – käsiteltävien pyyntöjen suurinta mahdollista määrää aikayksikössä yhteensä. Tehokkuuteen liittyy osaltaan myös *skaalautuvuus:* järjestelmän tulee voida kasvattaa kapasiteettiaan lisäämällä suoritusresursseja vastaamaan kasvanutta kuormaa. [7]

Verkkopalveluille on ominaista yhtäaikaisten käyttäjien verrattain suuret määrät. Koska palveluiden käyttö on usein välitöntä – verrattuna esimerkiksi sähköpostiin, jossa vastausta ei odoteta heti – odottavat käyttäjät palvelun vastaavan käyttäjämääristä huolimatta. Suosittu WhatsApp-pikaviestin on raportoinut palvelleensa parhaimmillaan 147 miljoonaa yhtäaikaista käyttäjää, jotka ovat lähettäneet jopa yli 700 000 viestiä sekunnissa [9]. Perinteisessä asiakas–palvelin-mallissa tällainen kuorma johtaa suureen määrään yhteyksiä palvelimen ja sen asiakkaiden välille: WhatsAppin tapauksessa yhtäaikaisia yhteyksiä on ollut yksittäisellä palvelimella parhaimmillaan 2,8 miljoonaa [10].

Suurten yhtäaikaisten käyttäjämäärien tukemisen lisäksi verkkopalvelun tulee pystyä käsittelemään pyynnöt nopeasti. Verkkopalveluiden käyttäjät eivät ole valmiita odottamaan vastauksia pitkään, vaan tutkimusten mukaan käyttäjät luovuttavat useissa tapauksissa esimerkiksi verkkosivun ensimmäisen latausyrityksen alle 10 sekunnin odottamisen jälkeen [11]. Liian pitkään kestäneiden sivulatausten toistuessa tämä kärsivällisyys laskee jopa alle 2 sekuntiin [11]. Järjestelmä ei siis voi jäädä odottamaan tietyn käyttäjän toimia tai pysähtyä kommunikaatioviiveiden takia, vaan sen tulee voida käsitellä sillä aikaa keskeytyksettä muita pyyntöjä.

Käytettävyysvaatimus liittyy läheisesti useisiin muihin laatuvaatimuksiin. Esimerkiksi hidas tai epäluotettava järjestelmä ei ole käytettävä, vaikka sen käyttöliittymä olisikin erityisen helppokäyttöinen. Pääasiassa käytettävyys on kuitenkin käyttöliittymätekninen asia. Tässä diplomityössä keskitytään käytettävyyden arviointiin taustajärjestelmää koskevien vaatimusten kautta, eikä itse käyttöliittymän käytettävyyteen kiinnitetä huomiota. [7]

Yhteenvetona verkkopalvelun toteuttamiseen valittavien tekniikoiden tulee siis mahdollistaa järjestelmän skaalautuvuus suurien yhtäaikaisten käyttäjämäärien hallintaan. Tämän vuoksi tekniikkavalinnan yhtenä tärkeänä kriteerinä on mahdollisuus toteuttaa tehtävien rinnakkainen suoritus ohjelmoijan kannalta yksinkertaisesti ja palveli-

men resursseja säästäen. Rinnakkaisuudella haetaan samalla sekä tukea suurille yhteysmäärille että pyyntöjen mahdollisimman tehokasta käsittelyä. Eduksi on myös se, jos rinnakkaistetut tehtävät saa erotettua toisistaan siten, etteivät mahdolliset virhetilanteet vaikuta toisten tehtävien suoritukseen; tässä tapauksessa pyritään välttymään virheiden leviäminen pyynnöstä toiseen. Tietoturvan varmistamiseksi toteutustekniikasta tulisi löytyä tuki salatun viestintäväylän käyttämiseksi asiakkaan ja palvelimen välillä.

2.2 Reaaliaikaiset päivitykset ja WebSocket-protokolla

Modernit verkkopalvelut sisältävät usein erilaisia lyhytaikaisia dynaamisia sisältöjä, joiden arvo laskee ajan kuluessa [12]. Esimerkiksi pikaviestimien välittämien viestien odotetaan saapuvan perille pisimmillään sekunneissa, eikä vanhentunut viesti ole vastaanottajalleen välttämättä hyödyllinen. Lisäksi koska käyttäjien sietokyky viiveille vähenee palvelua käytettäessä [11], tulee palvelun pystyä päivittämään käyttäjän haluama tietosisältö nopeasti. Tämän toteuttamiseen käytetään reaaliaikaisia päivityksiä, joilla tietoa päivitetään käyttäjälle välittömästi kun sitä on palvelussa saatavilla.

Reaaliaikaisia päivityksiä on yksinkertaista kuunnella pitämällä auki verkkoyhteys palveluun esimerkiksi TCP:n päälle toteutetulla omalla protokollalla, jolla tietoa lähetetään. Tämä toimii esimerkiksi mobiilialustoille toteutetuissa natiivisovelluksissa, mutta verkkoselaimet eivät tietoturvasyistä johtuen tue tällaisten yhteyksien muodostamista. Aiemmin päivityksiin onkin käytetty asiakkaalta palvelimelle lähetettyjä HTTP-kyselyitä, joita toistetaan uusien tietojen hakemiseksi. Tämä johtaa kuitenkin viiveeseen tiedon saamisessa, sillä asiakas ei voi tietää, milloin uutta tietoa on saatavilla, vaan pyytää sitä tietyin väliajoin. Tapa on myös tehoton, sillä siinä suoritetaan pyyntöjä, vaikka uutta tietoa ei olisikaan, kuluttaen palvelimen resursseja ja verkon kapasiteettia. Pyynnöt sisältävät vähimmillään HTTP:n otsikkotiedot, jotka voivat viedä lähetettävään tietoon verrattuna valtaosan liikenteestä. [12]

Toistuvien pyyntöjen lähettämisen tehottomuuden vuoksi on kehitetty myös muita tapoja taivuttaa HTTP:tä palvelimelta asiakkaalle lähtevien viestien tukemiseksi. Näistä käytetyin on niin kutsuttu *long polling*. Siinä asiakas lähettää HTTP-pyynnön palvelimelle, mutta palvelin jättää vastaamatta ja pitää yhteyden auki niin kauan kunnes sillä on lähetettävää tietoa. Tällöin palvelin vastaa pyyntöön uudella tiedolla ja sulkee yhteyden. Asiakas käsittelee saamansa tiedot ja lähettää uuden pyynnön, joka jätetään jälleen auki. Tällä tavoin asiakkaan kokema viive pysyy pienenä, koska palvelin voi vastata välittömästi. Kuitenkin myös tässä tavassa on ongelmia. Kuten aiemmin mainituissa ajastetuissa kyselyissä, kuluttavat HTTP-pyyntöjen otsikkotiedot turhaan verkon kapasiteettia. Odottava HTTP-pyyntö saatetaan aikakatkaista jos uutta tietoa ei saavu tarpeeksi nopeasti. Myös asiakkaan ja palvelimen välissä olevat välityspalvelimet ja välimuistit voivat aiheuttaa ongelmia, sillä HTTP ei tarjoa tapaa kertoa niille, että kyseessä on long polling -pyyntö. [13]

Edellä mainittujen tekniikoiden ongelmien vuoksi verkkoselaimille kehitettiin WebSocket-protokolla. Se on TCP:n päällä toimiva kommunikaatioväylä, jonka yli sekä asiakas että palvelin voivat lähettää viestejä ilman yhteyden sulkemista ja uudelleen avaamista. Yhteys avataan käyttäen HTTP:tä, mutta se päivitetään WebSocket-protokollaan HTTP:n päivitystoiminnallisuudella. Näin WebSocket-palvelinta voidaan ajaa samassa portissa HTTP-palvelimen kanssa, ja myös välityspalvelimet voivat tunnistaa yhteyden tyypin ja käsitellä sitä oikein. Tietoturvallisuuden osalta WebSocket toteuttaa TCP:n päälle verkkoselainten vaatiman *saman alkuperän käytännön* (*same-origin policy*), joka määrää, että yhteyden voi muodostaa vain samalle palvelimelle, jolla WebSocket-yhteyden avaava sivusto on. [14]

HTTP:n päälle rakennettuihin päivitysmekanismeihin verrattuna WebSocket on tehokas. Sen viestit sisältävät vain hyvin vähäisen määrän metatietoa, joten ne kuluttavat vähemmän resursseja ja verkon kapasiteettia. Viestien vastaanottamisessa ei ole viivettä, sillä palvelin voi lähettää viestin milloin vain, eikä sen tarvitse odottaa asiakkaan avaavan uutta yhteyttä. Aikakatkaisun estämiseksi WebSocketissa on tuki niin kutsutuille *Ping*-viesteille, joihin vastapuoli vastaa *Pong*-viestillä. Niiden avulla yhteys voidaan pitää aktiivisena hyvin pienellä liikennemäärällä. Lisäksi, toisin kuin aiemmin mainitut HTTP-tekniikat, WebSocket on kaksisuuntainen; asiakas voi lähettää palvelimelle viestejä milloin tahansa, riippumatta siitä onko palvelin lähettänyt viestejä asiakkaalle. [14]

2.3 Sovelluspalvelinarkkitehtuurit

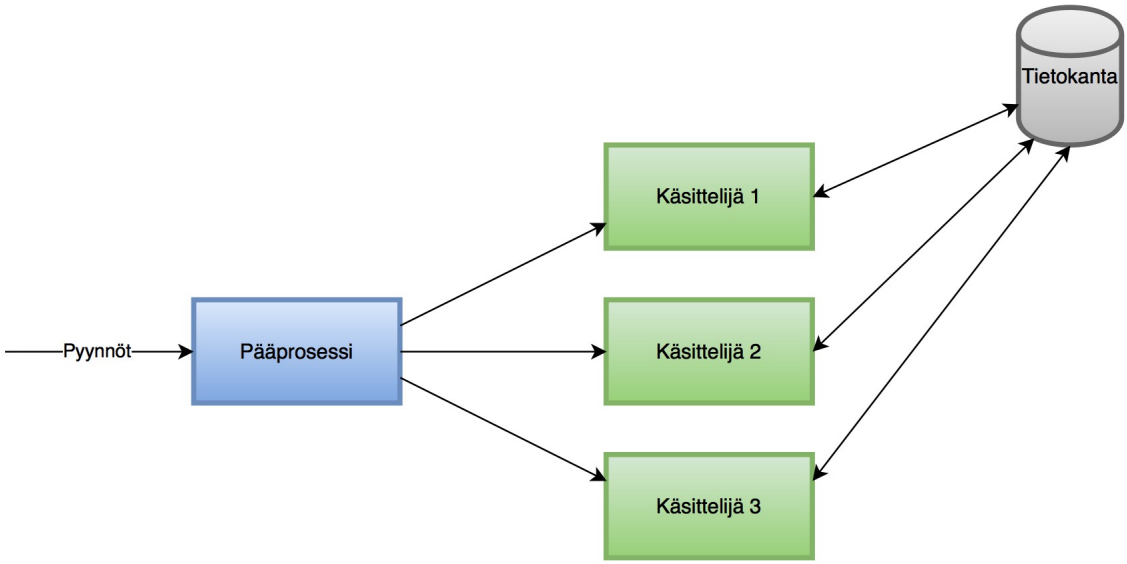
Verkkopalvelun taustajärjestelmä suoritetaan *sovelluspalvelimella*. Sovelluspalvelimen valinta riippuu sekä käytetystä ohjelmointikielestä että palvelimen tarjoamista ominaisuuksista. Eräs tärkeä suorituskykyyn vaikuttava ominaisuus on sovelluspalvelimen hajautusmalli, eli se, miten palvelin käsittelee useita eri pyyntöjä rinnakkain. Sovelluspalvelimet voidaan jakaa niiden hajautusmallien perusteella eri arkkitehtuureihin.

Ajan myötä sovelluspalvelimille on muodostunut kaksi pääarkkitehtuuria: *moniajava* (tyypistä riippuen *multi-threading* tai *multi-processing*) ja *tapahtumapohjainen* (*event-based*) arkkitehtuuri. Moniajavassa arkkitehtuurissa pyyntöjä käsittelemään käynnistetään useita suoritusyksiköitä eli säikeitä tai prosesseja, joita ajetaan rinnakkain. Tapahtumapohjaisessa arkkitehtuurissa pyyntöjä käsittelee yksi prosessi, joka vaihtaa käsiteltävää pyyntöä aina kun sovelluksen koodi jää odottamaan jotakin, esimerkiksi tietokantaa, tiedoston lukua tai verkkosisältöä.

Seuraavissa alakohdissa käsitellään näitä arkkitehtuureita tarkemmin ja pohditaan, kuinka ne voivat vastata aiemmissa kohdissa mainittuihin tehokkuus- ja luotettavuusvaatimuksiin. Koska turvallisuusvaatimus on itse suoritettavan sovelluksen eikä sovelluspalvelimen käsissä, sitä ei käsitellä tässä kohtaa. [15]

2.3.1 Moniajavat palvelimet

Moniajavassa palvelinarkkitehtuurissa kutakin vastaanotettua pyyntöä käsittelee yksi erillinen käyttöjärjestelmätason prosessi (multi-processing) tai pyyntöjen kesken jaetussa prosessissa oleva säie (multi-threading). Esimerkki arkkitehtuurista on esitettynä kuvassa 3. Arkkitehtuurin etuna on sovelluskoodin toteuttamisen yksinkertaisuus. Kun verkkopalvelun koodia ajetaan yhdessä suoritusyksikössä, voi kommunikaation ulkopuolisten järjestelmien kuten tietokannan tai muiden verkkopalvelinten kanssa hoitaa niin kutsutusti *blokkaavasti* (*blocking*). Tällöin sovelluksen suoritus keskeytyy tietokantakyselyn tai verkkopyynnön ajaksi ja jatkuu kun vastaus on saapunut. Ohjelmoijan ei siis tarvitse järjestää ohjelmaa takaisinkutsufunktioiksi, joita kutsutaan kun tietyt pyynnöt valmistuvat, vaan ohjelmassa säilyy lineaarinen suorituspolku. [15]



Kuva 3: Moniajava palvelin. Perustuu lähteeseen [15].

Perinteinen moniajava palvelin käyttää pyyntöjen käsittelyyn käyttöjärjestelmätason prosesseja. Pyyntöjä vastaanottamaan käynnistetään yksi pääprosessi, joka välittää pyynnöt eteenpäin käsittelijöille. Prosessien käytön etuna on automaattinen suoja sovelluskoodissa olevien virheiden leviämistä vastaan. Jos yksi käsittelijäprosessi kaatuu hallitsemattomaan virheeseen, se ei vaikuta muihin käsittelijöihin, vaan kyseiselle pyynnölle voidaan palauttaa virhetuloste ja kaatunut prosessi voidaan käynnistää uudelleen. Prosessien haittapuolena taas on niiden raskaus: kullakin prosessilla on oma muistialueensa, joten kaikkien pyyntöjen kesken jaetut tiedot täytyy joko kopioida jokaiseen prosessiin tai säilyttää jossain ulkoisessa järjestelmässä, josta prosessit voivat ne pyytää. Joissakin tapauksissa myös ohjelmakoodi tulee kopioida erikseen kunkin prosessin muistiin. Esimerkkeinä prosesseja käyttävistä moniajavista sovelluspalvelimista ovat PHP-FPM (PHP), uWSGI (Perl, Python, Ruby) sekä Unicorn (Ruby). [15]

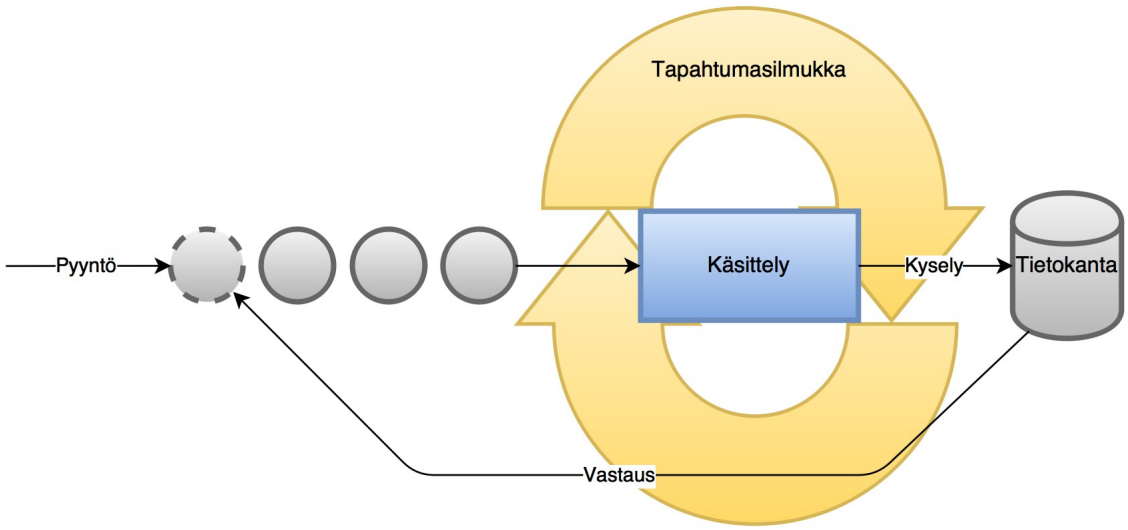
Koska käyttöjärjestelmätason prosessien käyttäminen on raskasta, voidaan niiden sijasta käyttää säikeitä. Säikeet ovat kevyempiä käynnistää ja sammuttaa, ja koska nii-

den muistialue on jaettu, ne voivat jakaa keskenään tietoa. Niistä voidaan esimerkiksi tallentaa tietoa kaikkien säikeiden yhteiseen välimuistiin. Yhteisten tietojen käsittelyssä tulee kuitenkin olla varovainen, sillä monen säikeen yhtäaikainen samojen muistialueiden käyttö voi johtaa tietojen korruptoitumiseen. Säikeiden haittapuolena on myös vaikeampi ohjelmakoodin virheiltä suojautuminen, sillä yhden säikeen kaatuminen voi kaataa myös kaikki muut säikeet. Säikeiden käyttö aiheuttaa rajoitteita myös käytettäville sovelluskehyksille ja kirjastoille, sillä niiden tulee tukea samanaikaista käyttöä useasta säikeestä (*thread-safe*). Mikäli käytettävät kirjastot eivät tue sitä itse, tulee sovelluskoodin toteuttajan rajoittaa niiden yhtäaikaista käyttöä lukkojen avulla. Esimerkkinä säikeitä käyttävästä moniajavasta sovelluspalvelimesta on Apache Tomcat (Java). [15]

Käytettiin sitten prosesseja tai säikeitä, pyyntöjen käsittelijöiden käynnistäminen ja sammuttaminen kunkin pyynnön kohdalla on usein palvelimen kuormaan nähden liian raskasta. Tämän vuoksi palvelimet käynnistävät käsittelijät yleensä etukäteen odottamaan pyyntöjä. Käsittelijät jätetään myös käyntiin pyynnön käsittelyn jälkeen, jotta niitä ei tarvitse käynnistää uudelleen seuraavaa pyyntöä varten. Tätä kutsutaan nimellä *preforking*. Palvelimille voidaan konfiguroida odottavien käsittelijöiden vähimmäismäärä ja ne voivat käynnistää uusia käsittelijöitä pyyntökuorman mukaan. Vaikka näin tehdään, voivat erillisten käsittelijöiden yhteensä viemä kokonaismuistimäärä ja niiden välillä suoritettavan käyttöjärjestelmän kontekstivaihdon viemä aika muodostua ongelmiksi, etenkin kun palvelinta skaalataan suurempien pyyntömäärien käsittelemiseksi. [15]

2.3.2 Tapahtumapohjaiset palvelimet

Tapahtumapohjaisessa palvelinarkkitehtuurissa pyynnölle ei käynnistetä omaa käsittelijää, vaan kaikki pyynnöt käsitellään samassa säikeessä, *tapahtumasilmukassa* (*event loop*). Tapahtumasilmukka ottaa käsittelyyn *tapahtuman* (*event*) ja suorittaa sen vaatimat toimenpiteet, kunnes tapahtuman käsittely on valmis tai tapahtuma jää odottamaan ulkoista ärsykettä. Käsittelyn jälkeen silmukka siirtyy seuraavaan tapahtumaan. Tapahtuma voi olla uusi pyyntö tai jonkin vanhemman pyynnön vaatiman tiedon saapuminen. Tietokantojen ja muiden ulkoisten sisältöjen kanssa kommunikoimiseen käytetään niin kutsutta *ei-blokkaavaa* (*non-blocking*) viestintää, jossa esimerkiksi tietokantakyselyn käynnistäminen ei pysäytä ohjelman suoritusta, vaan kyselykutsussa annetaan takaisinkutsufunktio, joka suoritetaan kun kyselyn tulos on saatavilla. Esimerkkinä tapahtumapohjaisesta sovelluspalvelimesta on Node.js. Kuvassa 4 on havainnollistus tapahtumapohjaisesta palvelimesta. [15]



Kuva 4: Tapahtumapohjainen palvelin ja sen tapahtumasilmukka. Perustuu lähteeseen [15].

Tapahtumapohjaisen palvelimen etuna on sen pieni suoritusresurssien kulutus verrattuna moniajavaan palvelimeen. Kun koodia ajaa vain yksi säie, ei sitä tarvitse kopioida tarpeettomasti moneen eri paikkaan. Yksittäisen säikeen tapauksessa ei myöskään tapahdu turhaa kontekstivaihtoa, vaan suoritin voi käyttää kaiken ajan koodin suorittamiseen. Palvelimella ei myöskään ole ylimääräisiä, ulkoista ärsykettä odottavia prosesseja tai säikeitä kuluttamassa turhaan muistia. Etenkin tapauksissa, joissa useimmat pyynnöt viestivät ulkoisten järjestelmien kuten tietokannan kanssa, tapahtumapohjainen palvelin on kevyt ja tehokas. [15]

Heikkoudet tapahtumapohjaisessa arkkitehtuurissa liittyvät sen suoritusmalliin. Koska kaikkien ulkoisesti kommunikoivien operaatioiden vastaukset käsitellään eri tapahtumana takaisinkutsufunktiossa, ei ohjelman suorituspolku ole enää lineaarinen, kuten moniajavassa palvelimessa. Tämä tekee sovelluksen suorituksen seuraamisesta ja virhetilanteiden tutkimisesta vaikeampaa, sillä suorituksen kulkemaa tarkkaa reittiä ei välttämättä ole saatavilla. Epälineaarisen sovelluksen toteuttaminen voi olla tottumattomalle ohjelmoijalle hankalampaa. Tapahtumapohjainen palvelin ei myöskään tarjoa suojaa sovelluskoodin virheitä vastaan, vaan suorittavan säikeen kaatuessa katoavat pahimmillaan koko tapahtumajono ja kaikkien suorituksessa olleiden pyyntöjen tila. Yhden säikeen käyttö kaikkien pyyntöjen suorittamiseen vaatii myös sen, ettei pyyntöä käsitellessä suoriteta raskasta laskentaa tai muuta suoritusaikaa merkittävästi vievää, sillä tapahtumasilmukan hidastuminen johtaa kaikkien pyyntöjen käsittelyn hidastumiseen. Sen sijaan raskaat operaatiot tulisi siirtää muihin säikeisiin, mikä tosin monimutkaistaa sovelluskoodia. [15]

3 ELIXIR VERKKOPALVELUISSA

Elixir on uusi Erlangin ja sen virtuaalikoneen päälle kehitetty funktionaalinen ohjelmointikieli. Kun tutkitaan sen käytettävyyttä verkkopalveluiden toteutukseen, on hyödyllistä tietää sen sisäisestä toteutuksesta. Suuri osa Elixirin ominaisuuksista tulee suoraan Erlangista ja sen virtuaalikoneesta, joten myös niitä tullaan käsittelemään.

Tässä luvussa käydään aluksi kohdissa 3.1 ja 3.2 läpi Erlangin ja sen virtuaalikoneen ominaisuuksia, joista suurin osa pätee suoraan myös Elixiriin. Kohdassa 3.3 esitellään Erlangin OTP-sovelluskehys ja sen avulla toteutettu Erlang-ohjelman tyypillinen rakenne, valvontapuu. Tämän jälkeen kohdassa 3.4 tutkitaan Elixirin erityispiirteitä ja sitä, mitä eroja ja yhtäläisyyksiä sillä on Erlangin kanssa. Lopuksi kohdassa 3.5 tutustutaan Phoenix Framework -sovelluskehykseen ja sillä toteutetun tyypillisen verkkopalvelun rakenteeseen.

3.1 Erlang

Erlang on funktionaalinen ohjelmointikieli, joka suunniteltiin rinnakkaisten, vikasietoisten ja pitkäkestoisten ohjelmien toteuttamiseen. Se kehitettiin vuonna 1986 Ericssonin ohjelmistolaboratoriossa uuden sukupolven AXE-N-puhelinkeskusta varten. Tavoitteena oli kehittää kieli korvaamaan Ericssonin aiemmissa puhelinkeskuksissa käytetty vanhentunut PLEX-ohjelmointikieli. [16]

Puhelinkeskukset olivat 1980-luvulla vaatimuksiltaan epätyypillisiä järjestelmiä. Niiden tuli hallita laajamittainen rinnakkaisuus, sillä yhden puhelinkeskuksen tuli voida palvella jopa satoja tuhansia käyttäjiä. Vikasietoisuuden tuli olla erityisen hyvällä tasolla, koska ongelmat puheluiden välittymisessä saattoivat johtaa suuriin ongelmiin ihmisten ja yritysten normaalielämässä. Puhelinkeskusta ei myöskään voinut sammuttaa ohjelmistopäivitysten suorittamisen ajaksi, vaan niissäkin tilanteissa puheluiden tuli toimia normaalisti. Näiden ongelmien ratkaisemiseksi ei tuohon aikaan löytynyt sopivia vaihtoehtoja, joten Ericsson päätti toteuttaa oman kielen, Erlangin, joka keskittyisi juuri näihin asioihin. [16]

Erlang otettiin tuotantokäyttöön AXD-puhelinkeskuksessa vuonna 1998. AXD sisälsi noin 2,6 miljoonaa riviä Erlang-koodia ja oli Ericssonille suuri menestys. Ericsson kielsi kuitenkin Erlangin käyttämisen uusissa projekteissa saman vuoden helmikuussa, viitaten muun muassa oman ohjelmointikielen ylläpidon aiheuttamiin kustannuksiin. Tämän seurauksena Erlang julkaistiin avoimena lähdekoodina kyseisen vuoden lopussa, mikä antoi kielelle mahdollisuuden yleistyä Ericssonin ulkopuolella. Tämän jälkeen Er-

lang alkoi löytää puhelinkeskusten sijaan paikkaansa esimerkiksi verkkopalveluista, joilla on usein samanlaisia rinnakkaisuus- ja luotettavuusvaatimuksia. [16]

Rinnakkaisuus- ja luotettavuusvaatimusten täyttämiseksi Erlangiin suunniteltiin erityisen kevyt hajautusmalli. Erlang-ohjelmistoa suoritetaan *prosesseissa*, joita ajetaan rinnakkain. Kyseiset prosessit eivät ole käyttöjärjestelmätason prosesseja, vaan niitä ajetaan Erlangin virtuaalikoneessa. Kullakin prosessilla on oma muistinsa, eikä prosessi voi koskea toisten prosessien muistiin. Kommunikaatio prosessien välillä tapahtuu aina asynkronisella viestinvälityksellä. Erlangissa ei ole manuaalista muistinhallintaa, vaan prosessien ja muistin hallinta on siirretty virtuaalikoneelle, jossa suoritus tapahtuu. Virtuaalikone voi optimoida prosessien käyttämän muistin siten, ettei prosessien muistialueilla ole suurta vähimmäiskokoa eikä näin ollen kulu turhaa tilaa prosessien muistien erotteluun. Tämän ansiosta Erlangin prosessit vievät vain vähän muistia verrattuna esimerkiksi käyttöjärjestelmätason prosesseihin. [16]

3.2 BEAM

BEAM (*Bogdan's Erlang Abstract Machine*) on Erlangin virtuaalikone, jossa ohjelmat suoritetaan. Se toteuttaa edellisessä kohdassa mainitut Erlangin vaatimat järjestelmätason ominaisuudet, kuten kevyen rinnakkaistuksen, erilliset prosessit, asynkronisen viestinvälityksen sekä suoritettavan ohjelmakoodin ajonaikaisen päivittämisen. Tämän lisäksi se tarjoaa Erlangin kielelliset ominaisuudet, kuten datan muuttumattomuuden (*immutability*) ja niin kutsutun hahmonsovituksen (*pattern matching*). BEAM suunniteltiin alun perin Erlangin käyttöön, mutta sittemmin on toteutettu myös muita sitä käyttäviä kieliä, kuten Elixir ja LFE (*Lisp Flavoured Erlang*). [17]

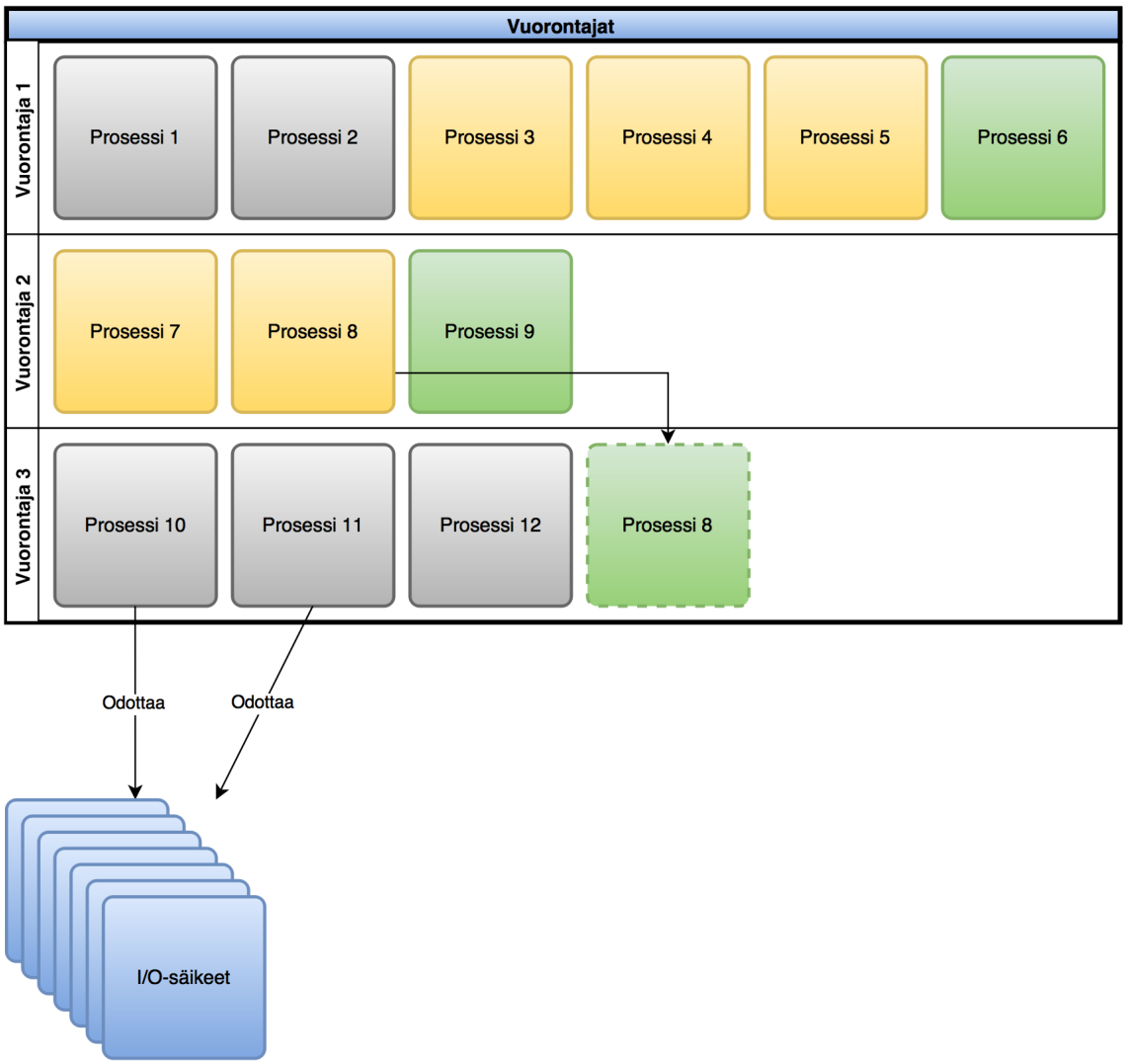
Erlangin vaatimusten mukaisesti BEAM suorittaa ohjelman prosesseissa, jotka ovat erotettuja toisistaan. Nämä prosessit ovat virtuaalikoneen sisäisiä, eivätkä näy käyttöjärjestelmän tasolla; käyttöjärjestelmä näkee vain yhden BEAM-prosessin. Prosesseilla on oma muistialueensa, joka alustetaan muutaman sadan tavun kokoiseksi ja joka kasvaa tarpeen mukaan. Kaikki prosessin data on tällä muistialueella, eikä prosessi pääse käsiksi muiden prosessien muistialueisiin. Jos prosessi tarvitsee jotain tietoa oman muistialueensa ulkopuolelta, se kopioidaan ensin prosessin omalle muistialueelle ennen kuin sitä voi käsitellä. Tästä poikkeuksena suurilla binäärimuuttujilla on suorituskyvyn parantamiseksi oma muistialueensa, ja prosessilla on vain osoitin kyseiseen kohtaan muistialuetta. Tämä on kuitenkin ohjelmoijan kannalta näkymätöntä ja kyseiset muuttujat käyttäytyvät samalla tavalla kuin muutkin. [17]

Koska Erlangin prosessit ovat kevyitä, niitä voidaan käynnistää laitteistosta riippuen satoja tuhansia tai jopa miljoonia. Virtuaalikoneen tulee siis hallita suurten prosessimäärien pitäminen ajossa niin, ettei aikaa tuhlata liikaa kontekstivaihtoihin. BEAM käyttää tähän sisäisiä vuorontajia (*scheduler*), joita käynnistetään oletusarvoisesti yksi kutakin laitteiston suoritinta kohden. Vuorontajat ovat käyttöjärjestelmätasolla yksittäi-

siä säikeitä. Kullakin vuorontajalla on oma suoritusjono, joka sisältää suoritusta odottavat prosessit, minkä lisäksi niillä on joukko eri syistä nukkuvia prosesseja, jotka odottavat jotain ulkoista ärsykettä voidakseen jatkaa suoritusta. Ajossa oleva prosessi vaihdetaan seuraavaan, jos se jää odottamaan viestiä toiselta prosessilta, tai viimeistään tietyn suoritusajan jälkeen. Jos vuorontajilla ei ole tarpeeksi suoritettavia prosesseja, ne voivat ottaa prosesseja toisten vuorontajien suoritusjonoista. Tietyin väliajoin vuorontajista valitaan yksi päävuorontaja, joka tarkistaa, onko kaikilla vuorontajilla tasainen määrä kuormaa, ja siirtää tarvittaessa prosesseja tilanteen korjaamiseksi. Sisäistä vuoronnusta käyttämällä voidaan välttyä kalliilta käyttöjärjestelmätason kontekstivaihdoilta. [17]

Vuorontajien lisäksi BEAM pitää myös varalla I/O-säikeitä, joihin siirretään suoritettavaksi säikeen pysäyttävät luku- ja kirjoitustehtävät. Näiden säikeiden tarkoituksena on estää vuorontajia pysähtymästä jonkin suoritusta estävän I/O-operaation suorittamiseen. Operaation ollessa käynnissä sitä vastaava BEAM-prosessi siirretään vuorontajassa odottavaan tilaan. BEAMin sisäinen rakenne vuorontajien ja I/O-säikeiden osalta on esitetty kuvassa 5. Kuvasta voidaan nähdä, kuinka vuorontaja 3 ottaa vuorontajalta 2 yhden suoritusta odottavan prosessin itselleen, jotta ajossa olisi mahdollisimman monta prosessia. Kuvassa näkyy myös, kuinka osa prosesseista odottaa I/O-säikeissä suoritettavia levyoperaatiota. [17]

Erlangin virheenkäsittely on suunniteltu vikasietoisten hajautettujen ohjelmistojen toteuttamista silmällä pitäen. Virheenkäsittelyssä painotetaan sitä, että tapahtuneen virheen käsittelee ulkopuolinen prosessi, eikä se, jolle virhe sattui. Näin voidaan tehdä, koska BEAMin prosessit ovat eriytettyjä, eikä yhden kaatuminen vaikuta muihin. Kyseinen virheenkäsittelytapa on valittu sen vuoksi, ettei prosessi, jossa virhe tapahtui, toimi välttämättä enää oikein. BEAM tarjoaa sen toteuttamista varten prosessien linkityksen. Kun prosessi kohtaa virheen, josta se ei voi toipua, se kaatuu ja lähettää tiedon asiasta niille prosesseille, jotka on linkitetty siihen. Näiden prosessien tehtävänä on käsitellä prosessin kaatuminen ja varmistaa, että sen suorittamat tehtävät siirretään jollekin toiselle prosessille. Tavoitteena on, että jos jokin ohjelman osa kaatuu, se ei vaikuta muihin osiin. [16]



Kuva 5: Prosesseja suoritettavana BEAMissa. Harmaat prosessit odottavat ulkoista ärsykettä jatkaakseen, keltaiset ovat valmiina suoritettavaksi ja vihreät ovat kyseisellä hetkellä suoritettavana.

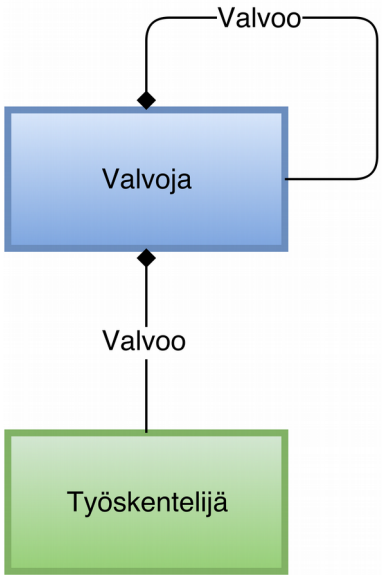
3.3 Erlang-ohjelmiston rakenne

Ohjelmistokehityksessä päädytään usein ratkaisemaan toistuvasti samantapaisia ongelmia. Ratkaisuja näihin ongelmiin koostetaan kirjastoiksi, sovelluskehyksiksi ja suunnittelumalleiksi. Erlangissa ongelmat liittyvät yleensä hajautukseen, virheenkäsittelyyn ja koodin ajonaikaiseen päivittämiseen. Nämä ovat vaikeita ongelmia, joiden ratkaiseminen vie aikaa ja aiheuttaa helposti virheitä. Tämän vuoksi Erlang-jakelun mukana tulee sovelluskehys OTP (*Open Telecom Platform*), joka on kokoelma sekä yleiskäyttöisiä kirjastoja että suunnittelumalleja Erlang-ohjelmistokehityksen helpottamiseksi. Tyypillisen Erlang-ohjelman rakenne koostuukin suureksi osaksi juuri OTP:n eri tilanteisiin tar-

joamista suunnittelumalleista. OTP tarjoaa toteutukset esimerkiksi tilakoneelle, tapahtumankäsittelijälle ja TCP-palvelimelle. [18]

Yleensä yksi työtä tekevä prosessi on linkitetty yhteen prosessiin, joka valvoo sen tilaa. Tämän jälkimmäisen prosessin nimi on *valvoja* (*supervisor*), ja sen tehtävä on seurata prosessin tilaa ja tarvittaessa käynnistää se uudelleen tai ryhtyä muihin määriteltyihin toimiin. Valvoja ei itse tee työtä, vaan kaikki työn tekeminen jaetaan *työntekijöille* (*workers*), joita se valvoo. Vastaavasti työntekijäprosessit eivät valvo muita prosesseja, vaan suorittavat varsinaisen työn tekevää ohjelmakoodia. Kun tällaisia valvojia asetetaan lisäksi valvomaan toisia valvojia, jotka puolestaan valvovat edelleen valvojia ja työtä tekeviä prosesseja, saadaan puumainen rakenne, *valvontapuu* (*supervisor tree*). [18]

Kuvassa 6 on esitetty valvontapuun luokkakaavio. Kukin ohjelman prosessi liittyy valvontapuun johonkin osaan, jolloin kaikilla prosesseilla on oma valvoja, joka huolehtii siitä. Näin mikä tahansa ohjelman osa voidaan virheen sattuessa sulkea ja käynnistää uudelleen. Valvontapuusta on hyötyä myös ohjelmaa suljettaessa: kukin valvoja kertoo valvomilleen prosesseille ohjelman sulkeutuvan, jolloin prosessit tekevät tarpeelliset sulkeutumiseen liittyvät toimenpiteet ja lopulta sulkevat itsensä. Kun kaikki ylimmän tason valvojan lapsiprosessit ovat sulkeutuneet, on ohjelma suljettu hallitusti. [18]



Kuva 6: Valvontapuun luokkakaavio. Valvoja voi valvoa sekä muita valvojia että työskentelijäprosesseja.

Valvontapuun tarkoituksena on kasvattaa ohjelman vikasietoisuutta [18]. Mikäli prosesseja ei valvota, niiden tilaa ei voida tietää. Tällöin ei voida olla varmoja, onko esimerkiksi jonkin tehtävän suorittaminen edelleen kesken vai onko sitä suorittava prosessi kaatunut. Kun kaikkia prosesseja valvotaan, saadaan välittömästi tieto jonkin osan toimimattomuudesta, ja voidaan ryhtyä tarvittaviin toimenpiteisiin.

3.4 Elixir

Elixir on funktionaalinen, dynaamisesti tyypitetty ohjelmointikieli, joka on toteutettu suoritettavaksi BEAM-virtuaalikoneella. Se tarjoaa Erlangin ominaisuuksien lisäksi joukon omia lisäyksiään. Kielellä oli kehitysvaiheessa kolme tavoitetta: hyödyntää mahdollisimman paljon Erlangin olemassa olevaa ekosysteemiä ja pysyä sille yhteensopivana, nostaa kehittäjien tuottavuutta muun muassa metaohjelmoinnin ja kehitystyökalujen avulla, sekä olla yksinkertaisesti laajennettavissa. [20]

Elixir on täysin yhteensopiva Erlangin ja OTP:n kanssa. Esimerkiksi Erlang-koodia voi kutsua Elixiristä ilman ylimääräistä ajonaikaista kustannusta. Erlangin ekosysteemin hyödyntämisen tarkoituksena on saada kieleen Erlangin ja sen virtuaalikoneen tarjoamat ainutlaatuiset ominaisuudet, mutta täydentää samalla niiden puutteita. Kielen pääkehittäjä José Valim on sanonut ”rakastin kaikkea [Erlangissa] näkemääni, mutta inhosin asioita, joita en nähnyt” [21]. BEAM on vakaa, testattu pohja ja Erlangin kirjastoja käyttämällä säästytään keksimästä uudelleen monia asioita. Samalla uusi kieli voi kuitenkin paikata niiden heikompia osa-alueita, joita ovat Valimin mukaan muun muassa metaohjelmointi ja polymorfismin toteuttaminen. [20, 21]

Turhan ja toisteisen ohjelmakoodin vähentämiseksi Elixiriin lisättiin mainittu metaohjelmointijärjestelmä. Metaohjelmointi toteutetaan kielessä *makrojen* avulla. Makro on käännösaikaisesti suoritettavaa koodia, joka saa argumenttinaan osan ohjelman lähdekoodista ja tuottaa sen perusteella lopulta ajonaikaisesti suoritettavan koodin. Tämän avulla kieleen voidaan toteuttaa uusia rakenteita, jotka toimivat samalla tavoin kuin kieleen sisäänrakennetut vastaavat. Itse asiassa useat kielen perusrakenteet ovatkin sisältä samanlaisia makroja, jotka vain tulevat automaattisesti kielen mukana. [20]

Esimerkki makroilla toteutetusta uudesta avainsanasta on koodilistauksessa 1. Esimerkissä luodaan avainsana `pipe_through`, jota voidaan käyttää rivin 12 esimerkin mukaisesti antaen sille argumenttina listan. Makro tarkistaa käännösaikaisesti rivillä 3 käytetäänkö sitä oikeassa paikassa ohjelmakoodia, ja jos ei, nostaa virheen rivillä 4. Tässä makrolla voidaan siis sekä yksinkertaistaa rivillä 6 olevaa lopullista koodia rivin 12 muotoon että lisätä koodiin käännösaikaisia oikeellisuustarkistuksia, tehostaen näin ohjelmoijan tuottavuutta kahdella eri tavalla. [20, 22]

```
1       defmacro pipe_through(pipes) do
2         quote do
3           if pipeline = @phoenix_pipeline do
4             raise "cannot pipe_through inside a pipeline"
5           else
6             Scope.pipe_through(__MODULE__, unquote(pipes))
7           end
8         end
9       end
10
11      # Ylläolevan makron käyttöesimerkki
12      pipe_through [:browser, :admin]
```

Koodilistaus 1: Esimerkki Elixir-makrosta ja sen käytöstä, sovellettu lähteestä [22].

Ohjelmointityön tuottavuutta nostetaan Elixirissä myös tehokkaiden työkalujen kautta. Tätä varten kielen mukana tulee kaksi hyödyllistä työkalua projektien hallintaan, *Mix* ja *Hex*. Mix on yleiskäyttöinen *make*n kaltainen työkalu, joka suorittaa sille määriteltyjä *tehtäviä* (*task*). Oletuksena tehtäviä on muun muassa riippuvuuksien hallinnalle, ohjelman kääntämiselle ja suorittamiselle, testien ajamiselle ja uusien projektien luomiselle. Ohjelmoija voi lisätä helposti omia tehtäviään, esimerkiksi usein kyseisessä projektissa käytettävien komentojen suorittamiseen. Mixin riippuvuuksien hallintaominaisuuksien avulla projektille voi määrittää tietyt riippuvuudet, jotta niistä voidaan asentaa samat versiot kaikkiin kehitysympäristöihin. Tämä vähentää riippuvuuksien eroista aiheutuviin virheisiin kuluvaa aikaa. Hex taasen on Mixin kanssa integroitu paketinhallintajärjestelmä. Siihen kuuluu pakettivarasto *hex.pm*, johon ohjelmoijat voivat lisätä omia kirjastojaan, sekä komentorivityökalu, jolla paketteja voi ladata, päivittää ja asentaa projektiin. Mix käyttää Hexiä riippuvuuksien lataamiseen silloin, kun riippuvuudet ovat saatavissa hex.pm:stä. Kyseisten työkalujen tarkoitus on vähentää ohjelmoijalta tyypillisiin projektinhallintaan liittyviin toimenpiteisiin kuluvaa aikaa. [23]

Kolmantena suunnittelun päämääränä Elixirissä on laajennettavuus. Yllä kuvattujen makrojen lisäksi tätä tavoitetta täyttämään toteutettiin kieleen tietotyyppien polymorfismi. Polymorfismi kielessä toteutetaan *protokollien* avulla. Protokolla sisältää tietyt säännöt ja rajapinnan, jotka toimivat sopimuksena tietotyypin ja sen käyttäjän välillä. Tietotyyppi voi toteuttaa protokollan, jolloin tietotyypin käyttäjä voi käyttää sitä kuin mitä tahansa muuta saman protokollan toteuttavaa tietotyyppiä. Näin esimerkiksi JSON-muotoon tietorakennetta muuttavan koodin ei tarvitse tietää, miten kukin tyyppi serialisoidaan merkkijonoksi, vaan tyypit voivat toteuttaa hypoteettisen JSON-protokollan, määrittäen itse millaisen muotoon ne muutetaan. Protokollien avulla kielen standardikirjaston moduuleja voi laajentaa käytettäväksi omien tietotyyppien kanssa, jolloin vältytään toteuttamasta samoja algoritmeja uudestaan ja uudestaan. [20]

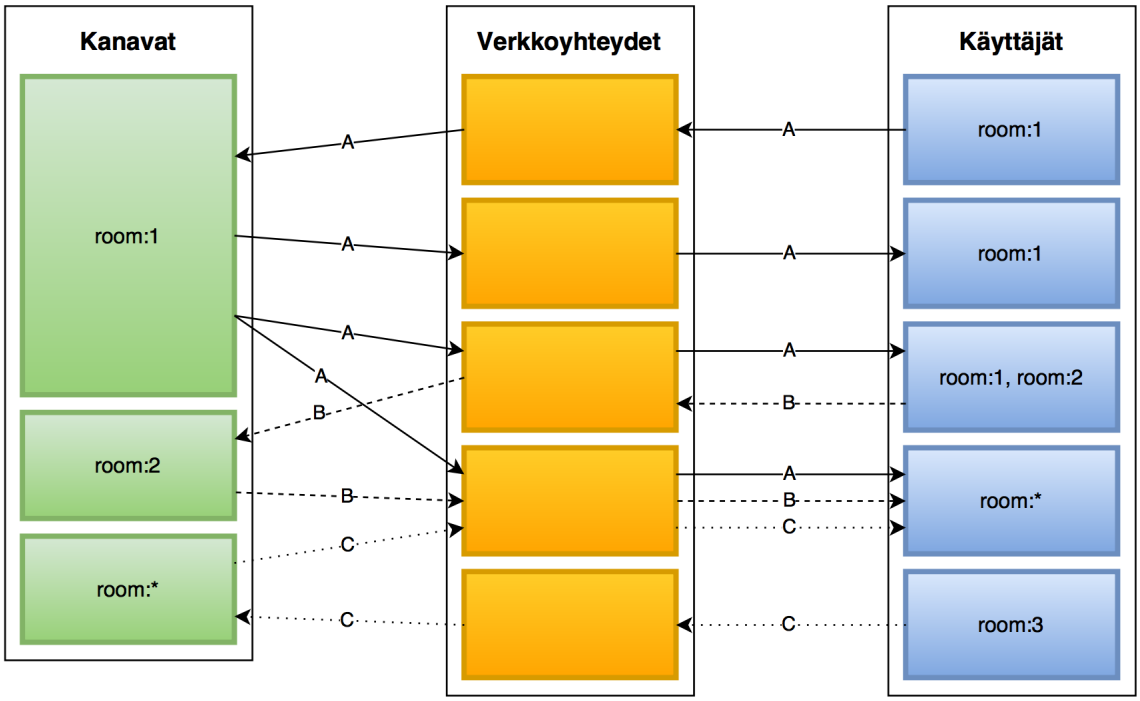
3.5 Phoenix Framework

Phoenix Framework on Elixirillä tehty sovelluskehys verkkopalveluiden toteuttamiseen. Se käyttää MVC-mallia (*Model–View–Controller – malli–näkymä–kontrolleri*) ja on suunniteltu sekä kehittäjän tehokkuutta että ohjelmiston suorituskykyä silmällä pitäen. Sovelluskehys sisältää valmiit toteutukset useimmille tyypillisen verkkopalvelun osille, kuten muun muassa pyyntöjen ohjauksen oikeille moduuleille, tuen HTML-sivupohjille, joihin voi syöttää dynaamista sisältöä, sekä kommunikaatioon tietokannan kanssa. [24]

Phoenix sisältää perinteisten verkkosivujen toteuttamiseen liittyvien ominaisuuksiensa lisäksi asiakkaan ja palvelimen keskeiseen reaaliaikaiseen kommunikaatioon tarkoitetun viestintäjärjestelmän, *kanavat* (*channels*). Kanavat abstrahoivat alemmalla tasolla varsinaisesti käytetyn viestintämenetelmän, joka on tyypillisesti WebSocket. Phoenixin kanavat toimivat julkaisija–tilaaja-mallin (*publisher–subscriber model*) mukaises-

ti. Mallissa julkaisijat luokittelevat viestit *aiheisiin* (*topic*), ja tilaajat voivat valita minkä aiheiden viestit ne haluavat. Kukin viesti välitetään automaattisesti kaikille, jotka ovat tilanneet kyseisen aiheen. Aiheilla voi olla myös aliaiheita, joista tilaaja voi valita yhden, useamman tai kaikki. Kanavissa julkaisijat ja tilaajat voivat vaihtaa rooleja dynaamisesti aina tilanteen niin vaatiessa. [24]

Kuvassa 7 on esitetty esimerkki Phoenixin kanavien käyttötilanteesta. Tilanteessa asiakkaat – kuvattuna sinisellä – ovat yhteydessä palvelimeen ja sen eri aiheisiin. Palvelimen päässä kullakin asiakkaalla on oma verkkoyhteys (*socket*) – kuvattuna oranssilla – jonka kautta asiakkaan viestit välitetään. Kukin asiakas on tilannut itselleen viestit yhdeltä tai useammalta kanavalta, jotka ovat kuvattuna vihreällä. Kullakin kanavalla on sama aihe, mutta eri aliaiheet. Alimman kanavan aliaiheella ”*” on erityismerkitys: se sisältää kaikki aiheen ”room” aliaiheet, jotka eivät kuulu jollekin toiselle kanavalle. Asiakaspäässä samalla merkillä ”*” on hieman eri merkitys: se tilaa asiakkaalle kaikkien aiheen ”room” aliaiheiden viestit. Tämän vuoksi toiseksi alin asiakas saa sekä viestit A, B, että C. Vaikka asiakas tilaisikin usean kanavan viestit, on hänellä silti vain yksi verkkoyhteys, jonka yli kaikki viestit välitetään. Näin vältytään ylimääräisiltä yhteyksiltä ja minimoidaan yhteyden viestiä kohden aiheuttamat kustannukset. [24]



Kuva 7: Phoenixin kanavien käyttötilanne. Siniset laatikot ovat käyttäjiä, oranssit palvelimen verkkoyhteyksiä ja vihreät kanavia.

Tietokantojen käsittelyyn Phoenixin mukana tulee riippuvuutena *Ecto*, joka on Elixirillä toteutettu tietokanta-abstraktiokirjasto. Se tarjoaa työkalut kyselyjen abstrahointiin erilaisten tietokantojen välillä sekä apureita monimutkaisten kyselyjen tekemiseen ja tiedon mallintamiseen sisäisestä tietorakenteesta tietokantatauluihin. Ecto täyttää Phoenixin MVC:ssä mallin tehtävän. Se ei kuitenkaan ole pakollinen riippuvuus, vaan

mallit ja tietokantaintegraation voidaan toteuttaa myös itse tai käyttäen jotain muuta kirjastoa. [24]

Phoenix on rakennettu muillakin tavoin modulaariseksi. Suuri osa toiminnoista on rakennettu pienten, uudelleenkäytettävien moduulien, *plugien* (*Plug*), päälle. Plugit ovat funktioita tai Elixirin moduuleja, jotka saavat parametrinaan tietorakenteessa asiakkaan lähettämän pyynnön ja siihen liittyvät tiedot, kuten vastauksen. Plugi tekee pyynnölle oman prosessointinsa, esimerkiksi lisää vastaukseen otsikkokenttiä tai tulostaa pyynnön tiedot lokitiedostoon. Lopuksi se palauttaa muutetun pyynnön, joka voidaan antaa seuraavalle plugille, tai tietyissä tilanteissa pudottaa kokonaan käsittelystä ja palauttaa vastaus asiakkaalle. Näin plugeista muodostuu *plugijono* (*pipeline*), johon tulevat pyynnöt syötetään, ja jonka toisesta päästä pyynnöt tulevat käsiteltyinä. Koodilistauksessa 2 on esimerkki plugijonosta, jossa pyynnöt kulkevat järjestyksessä ylimmästä plugista alimpaan. [24]

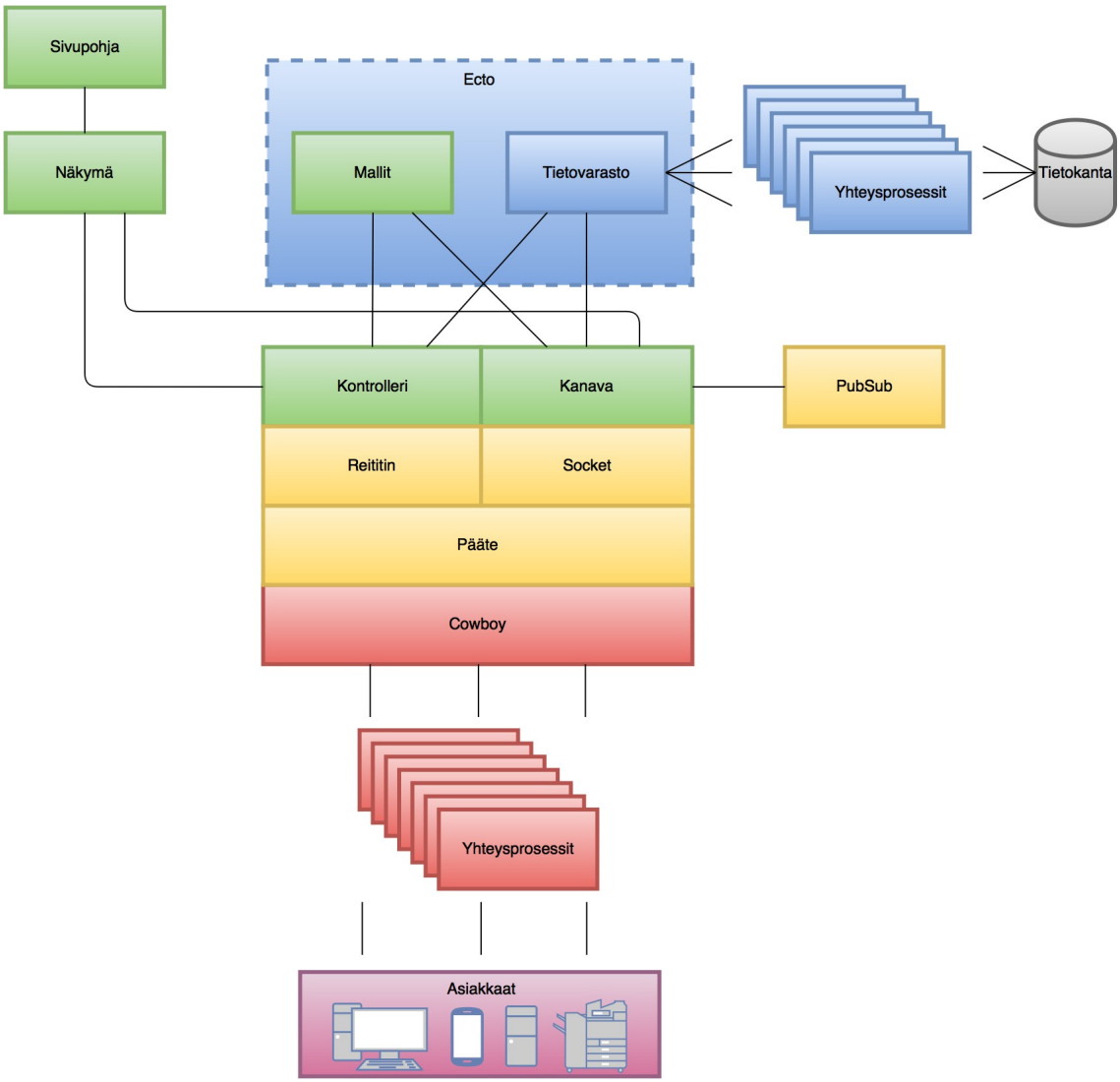
```
1       pipeline :browser do
2         plug :accepts, ["html", "text"]
3         plug :fetch_session
4         plug :protect_from_forgery
5         plug :put_secure_browser_headers
6       end
```

Koodilistaus 2: Phoenixin reitittimessä (router) käytettävä plugijono. [24]

Phoenixin plugeista koostuva rakenne kannustaa ohjelmoijaa eriyttämään toimintoja yksinkertaisiin plugeihin. Näin niistä voidaan tehdä yleiskäyttöisiä ja niitä voidaan käyttää hyödyksi myös muissa projekteissa. Myös koodista voidaan saada luettavampaa, kun isot funktiot ja syvät ehtolausetasot voidaan korvata kutsuttavien funktioiden jonolla. [24]

Alemmalla tasolla Phoenix käyttää Erlangilla toteutettua Cowboyta, joka on palvelin sekä HTTP- että WebSocket-protokollille [25]. Cowboy on tyypiltään moniajava palvelin ja siinä hyödynnetään BEAMin tarjoamaa kevyttä hajautusta avaamalla jokaiselle HTTP-pyynnölle ja WebSocket-yhteydelle oma prosessi. Tällä pyritään maksimoimaan palvelimen suorituskyky suorittamalla asioita mahdollisimman paljon rinnakkain, jolloin esimerkiksi raskaat tietokantaoperaatiot eivät estä uusien pyyntöjen käsittelyä samaan aikaan. Samalla eri pyynnöt voidaan eriyttää toisistaan niin, ettei pyynnön kaatuminen vaikuta muihin pyyntöihin, eikä pyynnöstä jää jäljelle tietoa, joka pitäisi siivota ennen seuraavan pyynnön käsittelyä. [26, 27]

Kuvassa 8 on havainnollistettu Phoenixillä toteutetun verkkopalvelun tyypillinen rakenne. Kuvan palvelu tarjoaa asiakkaille sekä perinteisen HTTP-rajapinnan, että reaaliaikaisen WebSocket-protokollaa käyttävän viestintäväylän, minkä lisäksi se tallentaa tietoja taustalla olevaan tietokantaan. Asiakkaiden pyynnöt ja WebSocket-yhteyksien kautta tulevat viestit vastaanottaa kuvassa punaisella merkitty Cowboy, joka käsittelee ne omassa prosessissaan: kukin HTTP-pyyntö saa oman prosessinsa, kun taas WebSocket-yhteyden kaikki viestit käsittelee sama prosessi. Prosessi aloittaa pyynnön tai yhteyden kautta tulevan viestin käsittelyn kuvassa keltaisella olevasta Phoenixin *päätteestä* (*Endpoint*), jossa on määritelty sekä tiettyjä sääntöjä yleisimpien pyyntöjen käsittelyyn, kuten esimerkiksi tiedostojen palvelemiseen HTTP:n yli, että WebSocket-viestien käsittely. Tästä eteenpäin HTTP-pyyntöjen ja WebSocket-viestien käsittely eriytyy: pyynnöt siirtyvät *reitittimeen* (*Router*), josta ne ohjataan oikealle kontrollerille, kun taas viestit ohjataan välivaiheen kautta oikeaan kanavaan. [24, 25, 26]



Kuva 8: Phoenixillä toteutetun verkkopalvelun tyypillinen rakenne. Punaisella Cowboy, keltaisella Phoenix, sinisellä Ecto ja vihreällä käyttäjän toteuttamat moduulit. Perustuu lähteisiin [24, 25, 26, 27, 28, 29].

Mikäli kontrolleri tai kanava haluaa käyttää tietokantaan tallennettua tietoa, se muodostaa mallien avulla kyselyn, joka annetaan Ecton *tietovarastolle* (*Repo*). Ecto pitää auki useita tietokantayhteyksiä eri prosesseissa, jotta yhteyksiä ei tarvitse avata ja sulkea jokaisen pyynnön yhteydessä. Tietovarasto antaa pyynnön yhdelle näistä prosesseista, joka hoitaa tiedon haun tietokannasta. Tietokannasta ja muista lähteistä saatu, käyttäjälle lähetettävä tieto muokataan näytettävään muotoon näkymässä. Näkymät voivat käyttää myös sivupohjia, usein HTML:ää, joihin voi sisällyttää tietokannan tietoa. Kontrollereista poiketen kanavat voivat myös lähettää reaktiona viestejä muille kanavan tilaajille tai kokonaan muihin kanaviin; viestien välittämisestä vastaa Phoenixin *Pub-Sub*-järjestelmä. [24, 25, 26, 28]

4 TOTEUTETTAVA VERKKOPALVELU

Elixirin soveltuvuuden arvioimiseksi tässä diplomityössä toteutetaan esimerkkisovelluksena ohjelmointikielten käytön tilastointipalvelu *Code::Stats*. Palvelun tarkoituksena on kerätä tietoa käyttäjän ohjelmointimääristä sekä -tavoista ja tuottaa niistä hänelle merkityksellistä tilastotietoa. Palveluun toteutetaan taustajärjestelmä sekä yksinkertainen käyttöliittymä, joka päivittyy automaattisesti palvelua käytettäessä.

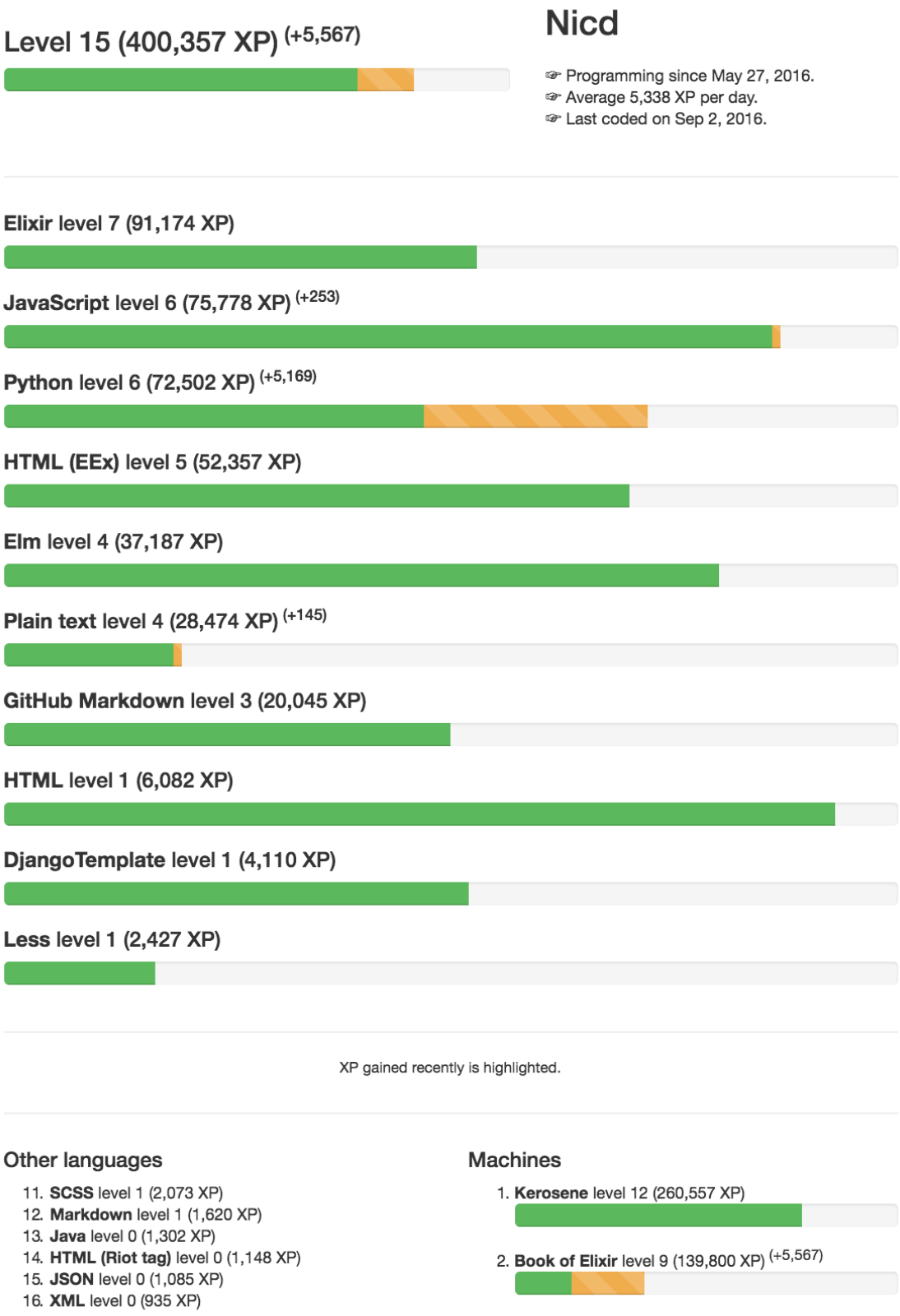
Tässä luvussa kuvataan toteutettavan palvelun yksityiskohdat. Kohdassa 4.1 kuvataan palvelun toiminnallisuutta yleisellä tasolla. Tämän jälkeen kohdassa 4.2 käydään läpi palvelun teknologiavalinnat, eli mitä ohjelmistoja ja niiden versioita toteutuksessa käytetään. Lopuksi kohdassa 4.3 syvennytään toteuttamiseen liittyviin teknisiin valintoihin sekä palvelun tarkempaan toimintaan ja rakenteeseen.

4.1 Palvelun kuvaus

Code::Stats on ohjelmoijille suunnattu tilastointipalvelu. Käyttäjän ohjelmoidessa tämän tekstieditoriin asennettava lisäosa lähettää palveluun tietoa käytetyistä ohjelmointikielistä ja niiden käyttömääristä. Palvelu koostaa tiedoista käyttäjälle tilastot, joiden avulla tämä voi seurata edistymistään eri ohjelmointikielten käytössä. Käytettyjen ohjelmointikielten määrät lasketaan kirjoitettujen merkkien määrästä: yhdestä merkistä saa yhden *kokemuspisteen* (*experience point, XP*). Tietyllä määrällä kokemuspisteitä pääsee tietylle *tasolle* (*level*), joka auttaa hahmottamaan kehitystä.

Palveluun toteutetaan yksinkertainen rekisteröitymistoiminnallisuus, jossa käyttäjä saa valita itselleen käyttäjätunnuksen ja salasanan sekä antaa halutessaan sähköpostiosoitteensa. Mikäli käyttäjä unohtaa salasanansa, mutta on antanut sähköpostiosoitteensa, hän voi luoda uuden salasanan palvelusta sähköpostiosoitteeseen lähetettävän resetointilinkin avulla. Käyttäjällä on oma asetussivu, jossa hän voi vaihtaa salasanansa tai sähköpostiosoitteensa, sekä halutessaan poistaa tunnuksensa kokonaan.

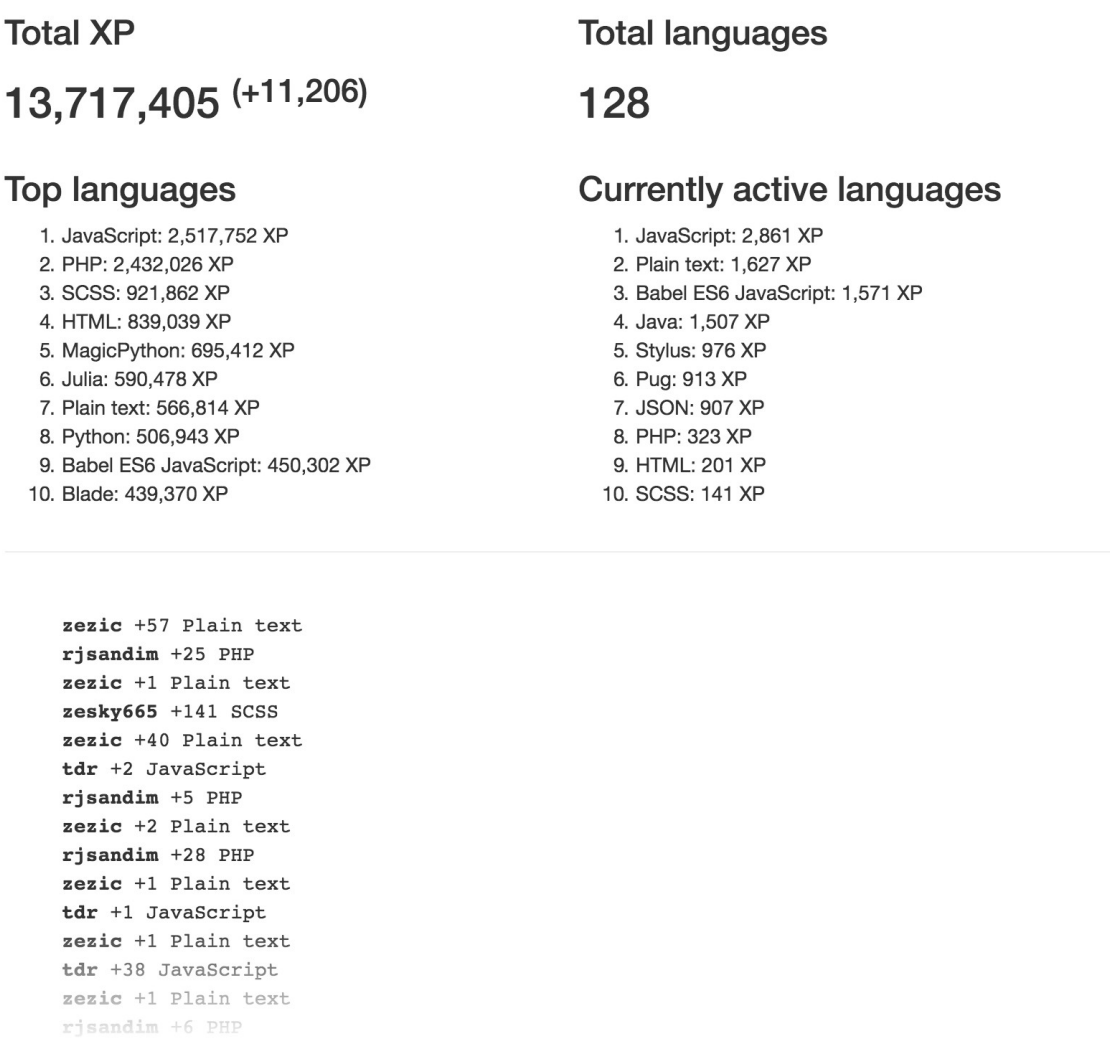
Jokaisella käyttäjällä on kuvassa 9 esitetyn kaltainen profiilisivu, jossa käyttäjän edistyminen on esitettynä kokonaisuutena ja ohjelmointikielikohtaisesti. Kullekin kielelle näytetään kokonaiskokemuspisteet sekä hiljattaiset kokemuspisteet, joiksi lasketaan pisteet viimeisen 12 tunnin ajalta sekä siltä ajalta kun käyttäjän profiilisivu on ollut auki selaimessa. Kokemuspisteet esitetään myös eroteltuna *koneen* (*machine*) perusteella. Kone kuvastaa yhtä ohjelmointiin käytettyä tietokonetta, jotta käyttäjä voi tunnistaa millä tietokoneella tämä tekee eniten töitä. Kun käyttäjä pitää profiilisivun auki selaimessa, päivittyvät pistelukemat ja saavutetut tasot automaattisesti tämän ohjelmoidessa.



Kuva 9: Esimerkki käyttäjän profiilisivusta.

Oletuksena profiilisivu on julkinen, mutta käyttäjä voi asettaa oman profiilisivunsa yksityiseksi, jolloin muille käyttäjille ja sisäänkirjautumattomille vieraille näytetään tieto, ettei profiilia löytynyt.

Palvelun etusivulla on esiteltynä kaikkien järjestelmän käyttäjien yhdistetyt tilastot. Sivulla näytetään suosituimmat kielet ja lista viimeisimmistä järjestelmään saapuneista kokemuspisteistä. Listassa esitetään sisääntulleiden kokemuspisteiden saajan käyttäjätunnus, käytetty kieli ja pisteiden määrä. Yksityistunnusten kohdalla näytetään käyttäjätunnuksen sijasta teksti ”Private user”. Myös etusivun tiedot päivittyvät automaattisesti, kun kuka tahansa järjestelmän käyttäjistä saa pisteitä. Etusivusta on esimerkki kuvassa 10.



Kuva 10: Esimerkki palvelun etusivusta.

4.2 Teknologiavalinnat

Toteutettavassa palvelussa käytetään asiakas–palvelin-mallia. Käyttäjät ottavat yhteyttä yhteen palvelimeen, joka käsittelee kutsut, välittää tietoja muille käyttäjille ja keskustelee tietokannan kanssa. Diplomityön aikarajoitteiden vuoksi tässä työssä rajoitutaan toteuttamaan palvelu niin, että se toimii yhdellä palvelimella, syventymättä tarkemmin sen hajautusmahdollisuuksiin.

Taustajärjestelmä (*backend*) toteutetaan käyttäen Elixiriä. Diplomityön kirjoitushetkellä Elixirillä verkkopalveluiden toteutukseen on vain yksi varteenotettava ja aktiivisessa kehityksessä oleva sovelluskehys: Phoenix Framework. Tämän vuoksi palvelu toteutetaan käyttäen sitä ja sen oletusriippuvuuksia: HTTP-palvelimena Cowboy, sekä tietokannan käsittelyyn Ecto. Tietokantana on PostgreSQL, sillä se tarjoaa kattavat relaatiotietokannan ominaisuudet ja sille on hyvä tuki Phoenixissä ja Ectossa. Ecton ja tietokannan välissä on käytössä Postgrex-kirjasto, joka hoitaa tietokannan kanssa kommunikoinnin. Toteutuksessa käytettävien eri ohjelmistojen tarkat versiot löytyvät taulukosta 1.

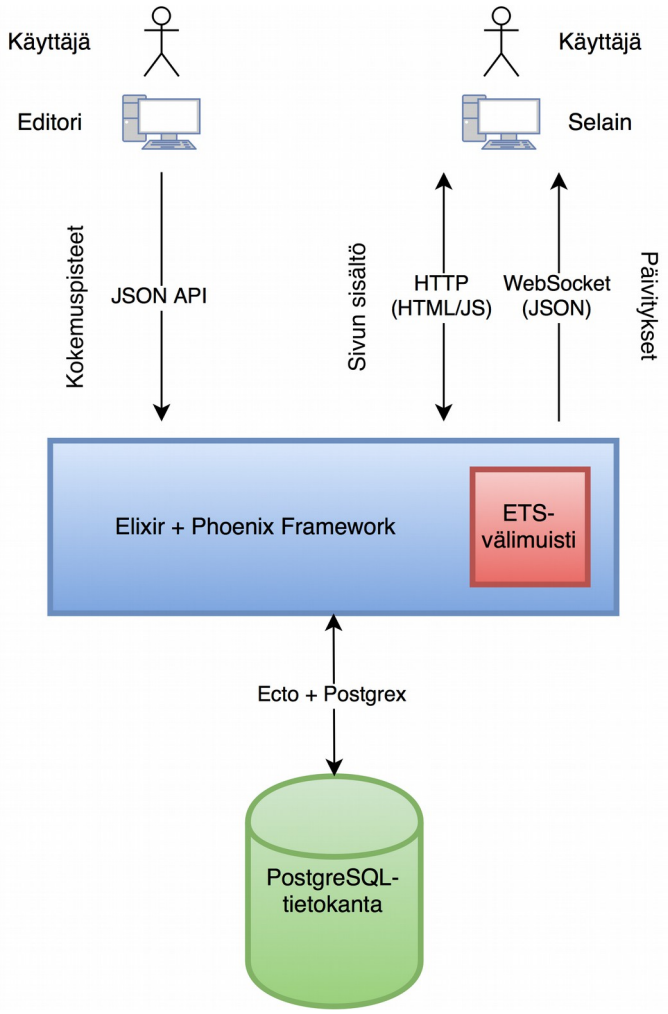
Koska tässä diplomityössä keskitytään Elixirin arviointiin, ei asiakkaan *käyttöliittymän* (*frontend*) toteutukseen kiinnitetä enempää huomiota. Periaatteessa taustajärjestelmän käyttämiseksi riittäisi mikä tahansa HTTP:tä ja WebSocket-yhteyksiä tukeva käyttöliittymä, oli se sitten verkkosivu tai suoraan käyttäjän alustalle tehty natiivisovellus. Käyttöliittymän dynaamiset ominaisuudet toteutetaan JavaScriptillä sekä Elmillä ja viestintään palvelimen kanssa käytetään Phoenixin tarjoamaa viestintäkirjastoa (*phoenix.js*).

Taulukko 1: Palvelussa käytettävien eri ohjelmistojen versiot.

Ohjelmisto	**Versio**
Cowboy	1.0.4
Ecto	2.0.4
Elixir	1.3.4
Erlang	19.1.1
Phoenix Framework	1.2.1
PostgreSQL	9.5
Postgrex	0.11.2

4.3 Taustajärjestelmän rakenne ja toiminta

Toteutettavan palvelun taustajärjestelmä on Phoenix-sovelluskehyksellä toteutettu ohjelma. Kuvassa 11 on yleistasoinen kuvaus järjestelmän rakenteesta. Järjestelmä voidaan jakaa kolmeen pääosaan niiden tarjoamien rajapintojen kautta: HTML-muotoiset verkkosivut, JSON-muodossa (*JavaScript Object Notation*) lähetetyt reaaliaikaiset päivitykset sivujen tilastoihin sekä JSON-muotoa käyttävä rajapinta kokemuspisteiden lisäämiseen. Kahta ensin mainittua käytetään tavallisen selaimen avulla, kun taas rajapintaa käytetään vain epäsuorasti, tekstieditoriin liitetyn lisäosan kautta. Sekä rajapinta että verkkosivut palvellaan käyttäen HTTP:tä, mutta koska reaaliaikapäivitykset vaativat jatkuvan yhteyden palvelimeen, käytetään niihin WebSocket-protokollaa.



Kuva 11: Palvelun rakenne yleistasolla.

Kaikki järjestelmän pysyvät tiedot talletetaan tietokantaan, josta niitä myös haetaan tarpeen mukaan asiakkaiden käyttäessä järjestelmää. Poikkeuksen tästä muodostavat järjestelmän kokonaistilastot, jotka näytetään etusivulla. Niiden laskeminen jokaisen pyynnön kohdalla olisi liian raskasta, joten ne ladataan järjestelmän käynnistyessä *ETS-*

tauluun (*Erlang Term Storage*). ETS on Erlangin sisäänrakennettu muistinvarainen avain–arvo-tietokanta. Se mahdollistaa tietojen tehokkaan tallennuksen ja myöhemmän lukemisen, sekä toimii tapana jakaa tietoja prosessien välillä [19]. Kokonaistilastot päivitetään ETS:ssä aina, kun järjestelmään saapuu rajapinnan kautta lisää kokemuspisteitä, sekä ladataan uusiksi tietokannasta 15 minuutin välein. Näin etusivulla näytettävät tiedot voidaan palvella suoraan muistista, mikä on tietokantaan verrattuna erittäin tehokasta.

4.3.1 Reaaliaikaiset päivitykset

Verkkosivuilla vieraileville asiakkaille lähetetään WebSocketin yli reaaliaikaisia päivityksiä järjestelmään saapuvista kokemuspisteistä sekä etusivulla että käyttäjän profiilisivulla. Päivitykset toteutetaan käyttäen Phoenixin kanavatoiminnallisuutta: etusivun päivityksille on kanava *frontpage*, kun taas kullakin käyttäjällä on oma kanava muotoa *users:<käyttäjätunnus>*, esimerkiksi *users:Nicd*. Kun käyttäjä vierailee etusivulla tai profiilisivulla, selain ottaa yhteyden oikeaan kanavaan. Taustajärjestelmä tarkistaa, onko käyttäjällä oikeutta seurata kanavan päivityksiä; esimerkiksi profiilisivua voi seurata vain jos se on julkinen tai jos on kirjautuneena sisään kyseisenä käyttäjänä.

Selain saa kanavalle liittyessään järjestelmältä alkutiedot, joiden perusteella käyttöliittymään piirretään tilastot. Tämän jälkeen niin kauan kuin käyttäjä pysyy sivulla, lähettää taustajärjestelmä tiedot uusista saapuneista kokemuspisteistä, joiden avulla käyttöliittymässä suoritettava Elm-kielellä toteutettu sovellus muuttaa sivun sisältöjä. Päivitystiedot lähetetään käyttäjälle JSON-muodossa. Mikäli käyttäjän verkkoyhteys palveluun katkeaa, ottaa käyttöliittymä automaattisesti yhteyden uudelleen kun verkkoyhteys palaa toimivaksi. Tällöin taustajärjestelmä lähettää jälleen kaikki tiedot, jotta käyttöliittymä voidaan päivittää, vaikka välissä jäisikin saamatta muutostietoja.

Verkkosivujen autentikointi toteutetaan *istuntoautentikaatiolla* (*session authentication*). Siinä käyttäjä, lähetettyään kirjautumislomakkeella oikean tunnuksen ja salasanan, saa evästeenä *istuntotunnisteen* (*session ID*), jonka avulla käyttäjän selain tunnistetaan sisäänkirjautuneeksi. Taustajärjestelmässä tallennetaan vastaavasti Phoenixin sisäänrakennettuun istuntomuistiin istuntotunniste ja sillä sisäänkirjautuneen käyttäjän tunnistenumero. Kun selain lähettää pyynnön sivulle, joka vaatii tunnistautumisen, tarkistaa *AuthRequired*-plugi onko selain lähettänyt evästeen, jossa on oikea istuntotunniste. Jos istuntotunnistetta ei ole tai se on vanhentunut, käyttäjälle näytetään virhesivu. Vastaavasti sivuilla, jonne sisäänkirjautunut käyttäjä ei saa päästä – kuten sisäänkirjaus- ja rekisteröitymissivut – tarkastaa *AuthNotAllowed*-plugi istunnon tilan ja ohjaa kirjautuneen käyttäjän omaan profiiliinsa.

4.3.2 Rajapinta

Järjestelmän rajapinta toteutetaan samanlaisena kontrollerina kuin verkkosivuja palvelevat kontrollerit, se vain ottaa sisään ja lähettää JSON-muotoista tietoa HTML:n sijasta.

Rajapinnassa on vain yksi toiminto: kokemuspisteiden lisääminen. Käyttäjän editori lähettää rajapintaan tiedon siitä, millä ohjelmointikielillä käyttäjä on ohjelmoinut ja kuinka monta merkkiä. Lisäksi lähetetään tieto siitä, milloin ohjelmointi tapahtui. Tämä sen vuoksi, että verkkoyhteyksien ongelmat saattavat estää palvelun saatavuuden ajoittain, jolloin editori voi tallettaa kokemuspisteet muistiin ja lähettää ne myöhemmin. Koodilistauksessa 3 on esimerkki rajapinnassa lähetettävästä tiedosta.

```
1       {
2         "coded_at": "2016-04-24T01:43:56+12:00",
3         "xps": [
4           {"language": "C++",    "xp": 15},
5           {"language": "Elixir", "xp": 30},
6           {"language": "EEx",    "xp": 3}
7         ]
8       }
```

Koodilistaus 3: Rajapinnassa lähetettävä kokemuspistetieto.

Rajapinnan autentikaatio toteutetaan *tunnisteautentikaatiolla* (*token authentication*). Siinä rajapinnan käyttäjä lähettää avaimena toimivan tunnisteen HTTP-pyynnön otsikkokentässä *X-API-Token*. Sopivan tunnisteen käyttäjä saa palvelun sivulta, jossa tämä voi lisätä ja poistaa koneita. Kullakin koneella on oma tunnisteensa. Tunniste on muodostettu käyttäjätunnuksesta ja koneen tunnistenumerosta, jotka on allekirjoitettu käyttäjäkohtaisella avaimella niin, ettei ulkopuolinen voi väärentää sitä. Taustajärjestelmässä rajapintapyynnön saapuessa *ApiAuthRequired*-plugi tarkistaa, että tunnisteen sisältämä käyttäjätunnus, koneen tunnistenumero ja tunnisteen allekirjoitus ovat oikeat. Jos pyyntö ei täytä ehtoja, palautetaan virheviesti ja lopetetaan pyynnön käsittely.

5 SOVELTUVUUDEN MITTARIT

Elixirin soveltuvuutta mitataan sekä kielen että toteutetun palvelun tasolla. Kielen arvioinnissa käytetään versiota 1.3.3 sekä diplomityön kirjoitushetkellä olemassa olevaa kielen ekosysteemiä. Toteutetulle palvelulle suoritettavissa mittauksissa käytetään Code::Statsin versiota 1.7.2. Tarkka git-versiotiiviste on `702e 308d 6718 1e48 dfba 37b6 193e be60 7c6b 5616`. Käytetyn ohjelmistoversion voi ladata osoitteesta <https://github.com/Nicd/code-stats/releases/tag/1.7.2>.

Soveltuvuutta mitataan useasta eri näkökulmasta, käyttäen pohjana toteutetun esimerkkipalvelun tuomaa ohjelmointikokemusta. Tässä luvussa kerrotaan kielen ja toteutetun palvelun mittaustavat. Kohdassa 5.1 kerrotaan kielen arvioimisesta ohjelmoijan omasta näkökulmasta tarkasteltuna. Toteutetun esimerkkipalvelun rakenteen arvioinnin kriteerit esitellään kohdassa 5.2. Lopuksi kohdassa 5.3 selostetaan esimerkkipalvelun suorituskyvyn mittaustekniikat.

5.1 Ohjelmoijan näkökulma

Elixirin soveltuvuutta arvioidaan ensimmäisenä ohjelmoijan näkökulmasta. Ohjelmoijana toimii diplomityön kirjoittaja, joka perustaa kokemuksensa esimerkkipalvelun toteuttamiseen. Ohjelmoijalla on aiempaa kokemusta useista korkean ja matalan tason ohjelmointikielistä, niin staattisesti kuin dynaamisesti tyypitetyistä. Verkkopalveluita ohjelmoija on toteuttanut käyttäen Javaa, PHP:tä ja Pythonia. Elixir on ensimmäinen ohjelmoijan käyttämä funktionaalinen ohjelmointikieli.

Arviointia suoritetaan aloittaen kielen syntaksista, edeten sen ominaisuuksiin ja lopulta kielen työkaluihin ja ekosysteemiin. Kussakin osiossa pohditaan, kuinka kyseinen kielen ominaisuus vaikuttaa verkkopalveluiden toteuttamiseen. Code::Statsin toteuttamiskokemusta käytetään apuna ominaisuuksien hyötyjen ja haittojen sekä käytön helppouden arvioinnissa.

5.2 Esimerkkipalvelun rakenne

Toteutetun esimerkkipalvelun rakenteesta voidaan arvioida Elixiriä pohtimalla, mitä rajoituksia ja painotuksia sen käyttäminen asettaa palvelulle. Tavoitteena on löytää kielen vahvuudet ja heikkoudet, eli tarkemmin ne asiat, joiden toteuttaminen on kielellä luontevaa ja vastaavasti ne, jotka aiheuttavat ongelmia ja lisätyötä. Rakennetta mitataan manuaalisella arvioinnilla.

Palvelun manuaalisessa arvioinnissa kiinnitetään huomiota kohdassa 2.1 mainittuihin ohjelmiston laatuvaatimuksiin: luotettavuuteen ja turvallisuuteen. Tehokkuusvaatimusta käsitellään erikseen palvelun suorituskyvyn kautta kohdassa 5.3. Koska palvelun käyttöliittymän arviointi on rajattu pois tästä diplomityöstä, käytettävyyttä arvioidaan vain tehokkuuden ja luotettavuuden kannalta: liian tehoton tai epäluotettava palvelu ei ole käytettävä.

5.3 Esimerkkipalvelun suorituskyky

Elixirillä toteutetun palvelun suorituskyvyn arvioimiseksi toteutetaan mittauksia testialustalla. Suorituskykymittauksilla pyritään arvioimaan, miten Elixirillä toteutettu verkkopalvelu skaalautuu palvelemaan suurta asiakasmäärää. Mittaustuloksia verrataan vastaavaan, toisella kielellä toteutetun vertailuohjelman samalla alustalla saavutettuihin tuloksiin.

Palvelun suorituskykyä mitataan rasitustesteillä, joissa palvelulle simuloidaan suuri määrä yhtäaikaisia käyttäjiä. Testeillä pyritään löytämään piste, jossa palvelu ei enää pysty käsittelemään uusia asiakkaita, eli suurin mahdollinen samanaikaisten käyttäjien määrä. Suorituskykymittauksiin käytetään kahta rasitustestausohjelmaa: pelkkien HTTP-pyyntöjä käyttäviin testeihin *Siegeä* sen yksinkertaisuuden vuoksi ja WebSocket-tuen vaativaan viimeiseen testiin taas *Tsungia*.

Seuraavassa alakohdassa 5.3.1 kerrotaan tarkemmin käytettävästä mittausalustasta, jossa palvelua ajetaan. Tulosten vertailuun käytettyjen vertailuohjelmien tiedot käydään läpi alakohdassa 5.3.2. Alakohdasta 5.3.3 alkaen kerrotaan tarkemmin eri mittauskohteista ja niiden mittausmetodeista.

5.3.1 Käytettävä alusta

Suorituskykymittauksiin käytetään Raspberry Pi 2 Model B -minitietokonetta, jossa on Linux-järjestelmä. Laitteen Broadcom BCM2836 -sirussa on 900 megahertsin neliytiminen ARM Cortex-A7 -suoritin, yhteensä 64 kilotavun L1-välimuistilla, 512 kilotavun L2-välimuistilla ja 1 gigatavun keskusmuistilla. Laitteessa on massamuistina MicroSD-kortti, josta peräkkäisten tietojen lukunopeus on mitattuna noin 22 megatavua sekunnissa. Käyttöjärjestelmänä on Arch Linux ARM, jossa Linux-ytimen versio 4.4.28. Palveluun liittyvien ohjelmistojen versiot on kuvattu aiemmin taulukossa 1.

Mittausten kuorma luodaan testaustyökalulla eri koneella kuin missä mitattava palvelu on, jotta palvelun suorituskyky ei vaihtele muiden ohjelmistojen vuoksi. Pyyntöjä lähettävänä koneena on MacBook Pro 2,5 gigahertsin Intel Core i7 -suorittimella ja 16 gigatavun muistilla. Koneet kytketään toisiinsa 100 Mb/s nopeudella toimivalla langallisella lähiverkolla.

Alustalle asennetaan Code::Statsista luvun alussa mainittu versio ja konfiguroidaan se tuotantotilaan, jossa ylimääräiset infotulosteet ovat pois päältä. Palvelu asete-

taan sisäisesti porttiin 1337, johon liikenne ohjataan *Nginx*-verkkopalvelimen kautta portista 8080. Palvelu voisi toimia myös itse suoraan portissa 8080, mutta se asetetaan välityspalvelimen taakse mittausten tasavertaisuuden vuoksi, koska toteutettava vertailupalvelin ei toimi ilman välityspalvelinta. Palvelussa tai vertailuohjelmassa ei käytetä TLS-salausta. Palvelun tarkka konfiguraatio on liitteessä A.

Code::Statsin käytettävään versioon tehdään pieniä muutoksia testien suorittamiseksi. Palvelusta poistetaan käytöstä gzip-pakkaus muuttamalla tiedoston *lib/code_stats/endpoint.ex* rivillä 11 gzip-avaimen arvoksi `false`. Vertailuohjelman toteuttamiseen käytettävän ajan rajallisuuden vuoksi työtä yksinkertaistetaan poistamalla käytöstä palvelun profiilisivujen päivityksiin käytettävästä kanavasta käyttäjän alkutietojen haku. Tämä toteutetaan muuttamalla tiedoston *web/channels/profile_channel.ex* sisältämän *join*-funktion sisältö koodilistauksen 4 mukaiseksi.

```
21      with \
22        %User{} <- AuthUtils.get_user(username)
23      do
24        {:ok, %{}, socket}
25      else
26        _ -> {:error, %{reason: "Unauthorized."}}
27      end
```

Koodilistaus 4: Profiilikanavan liittymisfunktion testeissä käytetty versio.

Mittaustilannetta varten otetaan PostgreSQL-tietokantaan tuotantojärjestelmästä anonymisoitu kopio, jossa on tarvittavat taulut ja niissä valmiiksi 283 eri käyttäjää. Tietokannan tila alustetaan samaksi jokaisen mittauksen alussa, jotta se ei vaikuta tulokseen. Samaa tietokantaa käytetään myös vertailuohjelmalle. Tietokanta aiheuttaa välttämättömästi hidastuksia palvelujen käsittelyyn ja voi muodostua pullonkaulaksi, mutta se aiheuttaa samanlaisen viiveen myös vertailtavalle ohjelmalle. PostgreSQL:ää käytetään sen Arch Linux ARM -paketin oletuskonfiguraatiolla.

5.3.2 Vertailuohjelmat

Suorituskykymittauksia verrataan muihin vastaaviin ohjelmiin, jotta saadaan vertailutietoa siitä, millä tasolla toteutetun palvelun tehokkuus on. Eri testeissä käytetään eri ohjelmia sen mukaan, mitä ollaan mittaamassa.

Staattisten sisältöjen kuten tiedostojen ja yksinkertaisten sivupohjien palvelemisen vertailuun käytetään Nginx-verkkopalvelinta. Nginx on yleisesti tunnettu korkeasta suorituskyvystään erityisesti staattisten sisältöjen, kuten kuvien sekä JavaScript- ja CSS-tiedostojen palvelemisessa. Sillä voi siis vertailla, onko toteutettu palvelu näissä pahasti jäljessä. Nginxin testeissä käytetty konfiguraatio on annettu liitteessä B. Käytössä on Nginxin versio 1.10.2.

Palvelun muiden ominaisuuksien vertailua varten toteutetaan itse yksinkertainen vertailuohjelma käyttäen PHP-ohjelmointikieltä (versio 7.0.12). Ohjelmaan toteutetaan vain mittauksen alla olevat sivut, eli esimerkiksi rekisteröitymistoiminnallisuuksia ei

niihin rakenneta. Mittauksen kohteena olevat sivut tehdään kuitenkin vastaamaan läheisesti esimerkkipalvelun toteutusta, eli tekemään samat tietokantakyselyt ja lähettämään samat reaaliaikaiset päivitykset. Vertailupalvelimen lähdekoodi on saatavilla osoitteesta <https://github.com/Nicd/code-stats-comparisons>.

Vertailuun käytettävänä moniajavana sovelluspalvelimena käytetään PHP-kielisiä ohjelmia suorittavaa *PHP-FPM*:ää (versio 7.0.12). Sille rakennetaan PHP-sovellus, jossa tietokannan kanssa kommunikointiin käytetään PHP:n sisäänrakennettua *PDO*-kirjastoa ja WebSocket-yhteyksiin tapahtumapohjaista *Ratchet*-palvelinta (versio 0.3.5). Koska PHP-FPM:ssä ajettavat prosessit eivät voi kommunikoida suoraan Ratchetille, tarvitaan niiden väliin *ØMQ*-viestijono (versio 4.1.5). Lisäksi, koska PHP-FPM ei tarjoa suoraan HTTP-rajapintaa, käytetään Nginxiä sen välityspalvelimena. Myös Ratchet asetetaan Nginxin välittämäksi, jotta se on saatavilla samassa portissa kuin PHP-sovellus.

5.3.3 Testien suorittaminen

Kussakin testissä testattavalta ohjelmalta mitataan sen kykyä vastata suureen määrään pyyntöjä. Mitattavina arvoina ovat sekunnissa käsiteltyjen pyyntöjen määrä (*pyyntötaajuus, p/s*), virheiden prosentuaalinen osuus sekä pyyntöjen käsittelyyn kuluneet suurin, pienin sekä keskimääräinen aika. Viimeisessä, Tsungilla suoritettavassa testissä arvoina ovat rajapintapyyntöjen pyyntö- ja virhetaajuus sekä WebSocket-yhteyksien virhetaajuus. Virheiksi tulkitaan virheellisen HTTP-tilakoodin palauttaneet pyynnöt, aikakatkaistut pyynnöt ja epäonnistuneet tai aikakatkaistut WebSocket-yhteydet.

Mittauksissa käytettävä kuorma luodaan viimeistä testiä lukuun ottamatta rasitustestaustyökalulla siten, että yhtäaikaisten asiakkaiden määrää nostetaan tyhjästä *kohdemäärään*, jota ylläpidetään niin että kukin asiakas suorittaa testikuvauksen kuvaamat pyynnöt 200 kertaa. Kohdemäärä on aluksi 10 asiakasta, ja sitä nostetaan jokaisella kerralla 10 asiakkaalla, kunnes löydetään taso, jossa palvelu ei enää pysty käsittelemään saapuvia pyyntöjä, vaan sekunnissa käsiteltyjen pyyntöjen määrä laskee tai virheiden määrä nousee huomattavasti. Kun optimaalisin taso on löydetty, suoritetaan sitä käyttäen lopulliset mittaukset.

Käytettävän alustan ja verkon kuorman aiheuttamia mittaustulosten pieniä vaihteluja ei voida täysin estää. Tämän vuoksi kaikki lopulliset mittaukset toistetaan viisi kertaa ja tuloksina käytetään näiden mittausten keskiarvoja.

5.3.4 Staattisten sisältöjen palveleminen

Ensimmäisessä testissä mitataan staattisten sisältöjen palvelemisen tehokkuutta. Staattisia sisältöjä on jokaisella sivulla, minkä vuoksi niiden palvelemisen tehokkuus on tärkeää. Muista testeistä poiketen tässä Nginxiä ei käytetä palvelun välityspalvelimena, sillä kyseisen tilanteen mittaamisesta ei olisi hyötyä. Sen sijaan palvelua ajetaan suoraan portissa 8080.

Testaus suoritetaan siten, että kukin yhdistävä asiakas lähettää GET-pyynnön tiedostoille *static/js/app.js*, *static/css/app.css* sekä *images/Logo_crushed.png*, tässä järjestyksessä. Nämä tiedostot lähetetään normaalisti kaikille palvelun asiakkaille, joilla niitä ei ole vielä välimuistissa, joten ne vastaavat hyvin palvelun normaalia käyttötapausta. Vertailuohjelmana käytetään Nginxiä.

5.3.5 Etusivun palveleminen vierastunnuksille

Tässä testissä mitataan palvelun kykyä palvella yksittäinen sivu sisäänkirjautumattomille käyttäjille. Koska etusivun tiedot löytyvät palvelun välimuistista eivätkä sisäänkirjautumattomat käyttäjät aiheuta pyyntöjä tietokantaan, antaa testi tiedon pelkkien sivupohjien palvelemisen suorituskyvystä.

Testissä sisäänkirjautumattomat asiakkaat – eli sellaiset, joilla ei ole auki olevaa istuntoa tai evästeitä – lähettävät GET-pyynnön palvelun etusivulle. Asiakkaat eivät avaa etusivun reaaliaikaisten päivitysten vaatimaa WebSocket-yhteyttä. Vertailuun käytetään sekä Nginxiä että toteutettua vertailupalvelinta.

5.3.6 Profiilisivun palveleminen

Tässä testissä mitataan sellaisia yksittäisiä pyyntöjä profiilisivulle, jotka johtavat palvelun sisäisesti sivupohjan prosessointiin ja tietokantakyselyihin. Tällaiset pyynnöt muodostavat suurimman osan pyyntömäärästä ja niissä tulee kattavasti testattua kaikki palvelimen perustoiminnallisuudet.

Testissä kukin asiakas on sisäänkirjautunut. Yksinkertaistuksen vuoksi kaikissa pyynnöissä käytetään samaa käyttäjätunnusta. Kukin asiakas lähettää GET-pyynnön käyttäjän *Nicd* profiilisivulle. Asiakkaat eivät avaa profiilisivujen reaaliaikaisten päivitysten vaatimaa WebSocket-yhteyttä. Vertailuun käytetään toteutettua vertailupalvelinta.

5.3.7 Tietokantaa käyttävien ja käyttämättömien pyyntöjen yhtäaikainen palveleminen

Tässä testissä mitataan, kuinka palvelu pärjää sekä tietokantaa käyttävien että käyttämättömien pyyntöjen samanaikaisesta käsittelystä. Koska alustan massamuistin lukunopeus on suhteellisen hidas, tulevat tietokantakyselyt pidentämään pyyntöjen kestoja. Tällöin palvelun tulisi voida palvella muita pyyntöjä ongelmitta.

Testissä kukin asiakas lähettää sisäänkirjautumattomana käyttäjänä GET-pyynnön ensin etusivulle ja sitten käyttäjän *Nicd* profiilisivulle. Asiakkaat eivät avaa sivujen reaaliaikaisten päivitysten vaatimaa WebSocket-yhteyttä. Vertailuun käytetään toteutettua vertailupalvelinta.

5.3.8 Reaaliaikapäivitysten palveleminen

Viimeisenä testataan reaaliaikapäivitysten palvelemisen suorituskykyä. Testissä mallinnetaan tilannetta, jossa palvelulla on suuri määrä kävijöitä kuuntelemassa päivityksiä ja pienempi ydinjoukko tuottamassa niihin uutta sisältöä. Pyrkimyksenä on päästä lähelle tyypillisen verkkopalvelun käyttötilannetta, jossa palvellaan sekä HTTP- että WebSocket-asiakkaita ja kommunikoidaan samaan aikaan tietokannan kanssa.

Testissä asiakkaat jaetaan kahteen ryhmään, *kuuntelijoihin* ja *tuottajiin*. Kuuntelijat jaetaan edelleen puoliksi etusivun ja käyttäjätunnuksen *Nicd* profiilisivun kuuntelijoihin. Kuuntelijat avaavat WebSocket-yhteyden ja liittyvät oman sivunsa kanavalle jäädden odottamaan sen viestejä. Tuottajat luovat toistuvasti kokemuspisteitä Nicd-käyttäjätunnukselle lähettämällä POST-pyyntöjä rajapinnan osoitteeseen */api/my/pulses*. Koodilistauksessa 5 on kuvattuna rajapintaan lähetettävä tieto.

```
1       {
2         "coded_at": "2016-11-06T01:00:00+0200",
3         "xps": [
4           {"language": "C++",    "xp": 15},
5           {"language": "Elixir", "xp": 30},
6           {"language": "EEx",    "xp": 3}
7         ]
8       }
```

Koodilistaus 5: Rajapintaan lähetettävä kokemuspistetieto.

Testin kuorma luodaan siten, että ensimmäisen 3 sekunnin aikana palvelua käyttämään käynnistetään 10 tuottajaa sekunnissa, eli yhteensä 30. Kukin tuottaja lähettää rajapintaan 200 kokemuspistepyyntöä. Kymmenen sekunnin kuluttua testin alusta, järjestelmään yhdistetään kuuntelijoita aiemmin mainitussa suhteessa 100 kuuntelijan sekuntivauhdilla. Palvelulle etsitään suurinta kuuntelijamäärää, jolla se toimii vielä luotettavasti. Vertailuun käytetään toteutettua vertailupalvelinta.

6 SOVELTUVUUSARVIOINTI

Esimerkkipalvelu Code::Stats toteutettiin kevään ja kesän 2016 aikana. Suorituskykymittaukset ja muu arviointi tehtiin saman vuoden syksynä. Toteuttamisen aikana toukokuussa palvelu siirtyi tuotantokäyttöön osoitteeseen <https://codestats.net/>. Tuotantokäytöstä saatuja kokemuksia käytetään apuna järjestelmää arvioitaessa.

Tässä luvussa esitellään soveltuvuusarvioinnin tulokset sekä suorituskykymittauksiin käytetyt lopulliset metodit. Kohdassa 6.1 käydään läpi Elixirin vahvuuksia ja heikkouksia ohjelmoijan subjektiivisesta näkökulmasta. Tämän jälkeen kohdissa 6.2 ja 6.3 keskitytään toteutettuun esimerkkipalveluun: ensin sen rakenteeseen ja siihen miten Elixir siihen vaikutti ja lopuksi palvelun suorituskykyyn.

6.1 Ohjelmoijan näkökulma

Elixirin ohjelmoijalle tarjoama rajapinta koostuu kielen syntaksista, sen ominaisuuksista sekä ekosysteemistä. Ekosysteemillä tarkoitetaan tässä tapauksessa sekä kielen tarjoamia ohjelmistotuotannollisia työkaluja että kielelle tarjolla olevia valmiita kirjastoja ja paketteja. On huomattava, että tässä kohdassa läpikäytävät asiat ovat ohjelmoijan henkilökohtaisia ja siten subjektiivisia mielipiteitä.

Tässä kohdassa käydään ensin läpi kielen syntaksia yleisesti alakohdassa 6.1.1. Seuraavana alakohdassa 6.1.2 käsitellään Elixirin hahmonsovituksen käyttöä ja hyötyjä. Alakohdan 6.1.3 aiheena ovat kielen syntaksiin kuuluvat *putket* (*pipe*) ja `with`-lohko. Tämän jälkeen alakohdassa 6.1.4 siirrytään käsittelemään metaohjelmointiin liittyviä makroja. Alakohdissa 6.1.5 ja 6.1.6 edetään syntaksista kielen ominaisuuksiin, käyden läpi funktionaalisen ohjelmoinnin ja datan muuttumattomuuden sekä Elixirin prosessimallin vaikutuksia ohjelmointiin. Kielen mukana tulevia työkaluja ja ekosysteemiä käsitellään lopuksi alakohdassa 6.1.7.

6.1.1 Syntaksi

Elixirin syntaksi on päällisin puolin hyvin erilainen Erlangiin verrattuna. Siinä missä Erlangin syntaksi perustuu Prologiin [16], Elixir on saanut inspiraationsa paljolti Rubysta [23]. Erlangin syntaksi käyttää koodilohkojen erotteluun Prologin tyyliin useita eri välimerkkejä ja on yleisesti melko tiivistä. Elixir sen sijaan luottaa vähemmän välimerkkeihin ja käyttää lohkojen erotteluun useimmissa tapauksissa avainsanoja `do` ja `end`. Myös funktioiden ja moduulien määrittelyyn käytetään avainsanoja ja moduulin sisältö suljetaan koodilohkoon, kun taas Erlangissa koko tiedosto kuuluu samaan moduuliin.

Syntaksien erojen vaikutuksista ei ole tieteellisiä tutkimuksia joten ne ovat käytännössä makuasia, mutta Elixirin syntaksi vaikutti yksinkertaisemmalta lukea ja muokata.

Esimerkkinä välimerkkien käytöstä Erlangissa useat samassa koodilohkossa olevat peräkkäiset lausekkeet tulee erotella toisistaan pilkuilla ja viimeinen lauseke tulee päättää tilanteesta riippuen joko pisteeseen tai puolipisteeseen. Koodia muokatessa tulee muistaa korjata välimerkit vastaamaan uutta tilannetta. Elixirissä lausekkeet erotellaan rivinvaihdoilla ja lohkon päättyminen aina avainsanalla `end`, joten rivejä muokatessa niiden päissä ei ole välimerkkejä, joista huolehtia.

Elixirin syntaksissa on joitain mielipiteitä jakavia erikoisuuksia, jotka johtuvat kielen toteutuksesta. Ensimmäiset niistä ovat implisiittiset sulkeet funktiokutsuissa. Koska Elixirissä on vain vähän avainsanoja ja useat kielen rakenteet, kuten `if`, ovat pohjimmiltaan makrojen avulla toteutettuja funktiokutsuja, tulisi niissä käyttää sulkeita, esimerkiksi `if(x, do: 1, else: 0)`. Implisiittisten sulkeiden avulla tämä voidaan kirjoittaa siistimmin: `if x, do: 1, else: 0`. Sulkeiden jättäminen pois tavallisista funktiokutsuista johtaa kuitenkin epäselvään koodiin ja pahimmillaan väärään toiminnallisuuteen. Esimerkiksi putki `f x y |> g z` tulkitaan muotoon `f(x, y |> g(z))` eikä muotoon `f(x, y) |> g(z)`, jota ohjelmoija on todennäköisesti tarkoittanut. Ongelmasta päästään kuitenkin kirjoittamalla sulkeet kuuliaisesti jokaiseen funktiokutsuun, mikä onkin tyylioppaiden suositus [30].

Toinen mainittava syntaksin erikoisuus on nimettömien ja nimettyjen funktioiden erilainen käyttö. Nimetön funktio luodaan avainsanalla `fn`, joka palauttaa viittauksen luotuun funktioon. Funktiota `f = fn x -> :bar end` ei kutsuta kuitenkaan `f(x)` niin kuin mitä tahansa nimettyä funktiota, vaan `f.(x)`. Yksi syy eroon juontuu edellisessä kappaleessa mainituista implisiittisistä sulkeista. Koska sulkeet lisätään implisiittisesti myös sellaiseen funktioon, jolla ei ole yhtään argumentteja, on esimerkiksi muuttujaa *g* kutsussa `f(g)` oikeasti funktiokutsu: `f(g())`. Kun muuttujaan tallennetulla nimettömällä funktiolla on oma kutsusyntaksi, voi sen antaa argumenttina funktiolle kutsumatta sitä. Käytännössä nimettömiä funktioita tuli esimerkkipalvelua toteuttaessa käytettyä kuitenkin vain hyvin harvoin, joten kutsutapojen ero ei muodostunut ongelmaksi.

Yhteenvetona Elixirin syntaksi oli pääosin yksinkertainen ja nopea oppia. Seuraavissa alakohdissa käsiteltävät putket, `with` ja makrot vaativat enemmän ajattelua, mutta kahden ensin mainitun oppiminen onnistui ja niitä tuli käytettyä palvelun toteuttamisessa runsaasti. Kokonaisuudessaan syntaksi ei tullut missään vaiheessa toteuttamisen esteeksi, vaan sillä sai esitettyä haluamansa asiat yleensä selkeästi ja tehokkaasti.

6.1.2 Hahmonsovitus

Hahmonsovitus tarkoittaa tietyn arvon sovittamista annettuun *hahmoon* (*pattern*). Sillä voidaan sekä tarkistaa annetun arvon vastaavan hahmoa että purkaa tietorakenteesta osia uusiin muuttujiin. Esimerkiksi lause `{:ok, result} = {:ok, 13}` tarkistaa että oikealla puolella oleva arvo vastaa vasemman puolen hahmoa ja asettaa muuttujan *result*

arvoksi oikealla puolella vastaavassa kohdassa olevan arvon 13. Hahmonsovitusta käytetään Elixirissä muuttujien sijoitusoperaatioissa, ehtolauseissa sekä funktiomäärittelyissä. [31]

Elixiriä käyttäessä hahmonsovitus muuttui nopeasti yhdeksi käytetyimmistä työkaluista. Sillä on helppo ottaa funktion palauttamasta tietorakenteesta olennainen osa ja samalla paluuarvon oikeellisuus tulee tarkastettua: muuttujan sijoituksessa suoritus kaatuu virheeseen jos hahmonsovitus epäonnistuu. Hahmonsovituksen avulla voi myös luoda samasta funktiosta eri versioita, joiden välillä valitaan kutsussa käytetyn arvon perusteella. Esimerkiksi funktio `def f(" " <> rest), do: f(rest); def f(text), do: text` kutsuu itseään rekursiivisesti uudestaan ilman syötteenä saadun merkkijonon alussa ollutta välilyöntiä. Jos merkkijonon alussa ei ole välilyöntiä, kutsutaan automaattisesti funktion jälkimmäistä versiota, sillä ensimmäisen funktion kohdalla hahmonsovitus ei onnistu. Lopputuloksena funktio palauttaa uuden merkkijonon, joka on syötteenä annettu merkkijono alussa olevat välilyönnit poistettuna.

Hahmonsovituksen puuttumisen huomasi toteutuksen aikana muita kieliä käyttäessä. Samassa lauseessa voi tarkistaa jonkin arvon tyypin sekä sisällön ja purkaa arvon yhteen tai useampaan muuttujaan. Tähän menee muissa kielissä helposti useampi rivi. Hahmonsovituksella koodista voi tehdä myös idiomaattisempaa, esimerkiksi korvaamalla funktiolle annettujen argumenttien tarkasteluun tarvittava ehtolausekkeiden yhdistelmä useammalla eri funktiolla, joista kutsuttava valitaan arvon perusteella. BEAMiin toteutettujen optimointien ansiosta hahmonsovitusta käyttävä tapa on myös huomattavasti suorituskykyisempi.

6.1.3 Putket ja with

Putket ja `with` ovat Elixirin kaksi vastausta hyvin yleisiin kielen käyttötapauksiin. Ne ovat *syntaktista sokeria* (*syntactic sugar*), eli vaihtoehtoisia ja helpompia kirjoitusmuotoja usein käytetylle, kömpelömmin kirjoitettavalle koodille. Putket ovat nimensä mukaisesti putkia funktiosta toiseen, kun taas `with` on tarkoitettu yksinkertaistamaan syviä ehtolausekeketjuja.

Ohjelmoinnissa tulee usein vastaan tilanteita, joissa jotain tiettyä dataa syötetään funktiosta toiseen, kunnes funktioketjun suorittamisen jälkeen saadaan haluttu lopputulos. Otetaan esimerkkinä Code::Statsin plugista *RequestTime* seuraava laskutoimitus: `format_output(get_unit(trunc(Float.round(diff)), time_units))`. Koodi muodostaa kasvavan pinon, jonka sulkeita on vaikea seurata. Myös väliin tulevista parametreista on vaikea huomata, mille funktiolle ne kuuluvat. Rivin kontrollivuo luetaan sisältä ulospäin, eikä suoraan näe, mikä kohta evaluoidaan ensimmäisenä. Koodin voi pilkkoa useammalle riville, mutta tällöin tuloksia tulee asettaa tilapäisiin muuttujiin.

Elixirin putket kääntävät kontrollivuon jälleen normaaliksi, kuten voidaan nähdä koodilistauksesta 6, jossa *RequestTime*-plugin laskutoimitus on toteutettu putkea käyt-

täen. Putki alkaa syötteestä joka voi olla muuttuja, literaali tai funktiokutsu ja etenee siten, että kunkin vaiheen tulos syötetään seuraavassa vaiheessa kutsuttavan funktion ensimmäiseksi argumentiksi. Funktioiden suoritusjärjestys on helppo nähdä, eivätkä niiden saamat lisäargumentit katoa funktionimen yhteydestä. Kuten aiemmassa alakohdassa mainittiin, putken yhteydessä tulee muistaa käyttää sulkeita, että koodi toimii kuten ohjelmoija tarkoittaa sen toimivan. Rajoituksena on myös se, että putkessa siirtyvää dataa ei voi saada funktioon muulle paikalle kuin ensimmäiseksi argumentiksi.

```
1       diff
2       |> Float.round()
3       |> trunc()
4       |> get_unit(@time_units)
5       |> format_output()
```

Koodilistaus 6: Esimerkki Elixirin putkisyntaksista.

Putki toimii hyvin lineaarisesti etenevään kontrollivuohon, mutta osa funktioista voi palauttaa joskus virheitä. Esimerkiksi verkkopalvelun MVC-mallin kontrollereissa haetaan usein tietoa tietokannasta, joka voi palauttaa tilanteesta riippuen eri virhestatuksia. Koska saatu virhestatus syötetään seuraavan funktion argumentiksi, tulee putken kaikkien funktioiden käsitellä myös edellisiltä vaiheilta mahdollisesti tulevat virheet tai suoritus kaatuu. Tämä ei ole käytännöllistä, sillä funktioihin tulee lisätä turhaa koodia hoitamaan putkisyntaksin puutteita. Tällaisissa tilanteissa putken käyttöä tulisikin välttää kokonaan.

Edellä kuvattu tilanne, jossa suoritus etenee kutsusta toiseen mikäli ne eivät palauta virheitä, on kontrollereissa hyvin yleinen. Koodilistauksessa 7 on mukaelma Code::Statsin *PulseController*-moduulissa käytetystä koodista uusien kokemuspisteiden lisäämiseksi järjestelmään. Koodissa kutsutaan peräkkäin viittä eri funktiota, joista kukin voi palauttaa virhestatuksen. Jos jokin virhe palautuu, se tulee käsitellä, tässä tapauksessa palauttamalla käyttäjälle virheilmoitus. Koodista huomaa, että tarkisteluista ja ehdoista muodostuu ”pyramidi”, joka kasvaa nopeasti ja jossa virheentarkistelukoodia joudutaan toistamaan jokaisella tasolla. Kontrollerin tehtävien kasvaessa koodista tulee nopeasti lukukelvotonta.

Elixirin avainsana `with` on suunniteltu tätä tilannetta varten. Sitä käytettäessä koodi järjestetään eri tavalla. Avainsanan jälkeen listataan suoritettavat funktiot ja niiden sallitut paluuarvot. Kunkin funktion paluuarvoa verrataan hahmonsovituksella sallittuun paluuarvoon ja jos hahmonsovitus onnistuu, siirrytään seuraavan funktion suorittamiseen. Jos kaikkien funktioiden suoritukset palauttavat halutut arvot, suoritetaan `do` ja `end` -avainsanojen välissä oleva koodi, jonka paluuarvosta tulee koko `with`-lohkon paluuarvo. Mikäli mikä tahansa funktio palauttaa arvon, jonka hahmonsovitus ei onnistu, siirrytään `else`-lohkoon, jossa funktion palauttama arvo voidaan käsitellä.

```
1       case parse_timestamp(datetime) do
2         {:ok, %DateTime{} = datetime} ->
3           case check_datetime_diff(datetime) do
4             {:ok, datetime} ->
5               case create_pulse(user, machine, datetime) do
6                 {:ok, pulse} ->
7                   case create_xps(pulse, xps) do
8                     {:ok, inserted_xps} ->
9                       case update_caches(inserted_xps) do
10                        :ok -> put_status(conn, 201)
11                        {:error, status} -> check_error(conn,
        status)
12                      end
13
14                    {:error, status} -> check_error(conn, status)
15                  end
16
17                {:error, status} -> check_error(conn, status)
18              end
19
20            {:error, status} -> check_error(conn, status)
21          end
22
23        {:error, status} -> check_error(conn, status)
24      end
25
26      def check_error(conn, :not_found), do: put_status(conn,
        404)
27      def check_error(conn, :generic), do: put_status(conn, 400)
28      def check_error(conn, :internal), do: put_status(conn, 500)
```

Koodilistaus 7: Esimerkki funktioiden ehdollisesta suorittamisesta ilman with-syntaksia.

Koodilistauksessa 8 on esitetty koodilistauksen 7 koodi uudestaan, tällä kertaa käyttäen `with`iä. Listauksesta voi huomata, että koodi mahtuu paljon pienempään tilaan kun virheentarkisteluita ei toisteta jokaisen ehdon jälkeen. `with`-lohkossa kutsuttavat funktiot pysyvät tiiviissä listassa, eikä koodi siirry jatkuvasti uudelle sisennystasolle. Koodista näkee paljon helpommin, mitä funktioita siinä kutsutaan ja missä järjestyksessä. Tässä versiossa ei olisi samaa ongelmaa uusien funktiokutsujen lisäämisessä listaan, koska koodin kompleksisuus pysyy samana funktioiden määrästä huolimatta. Sekä putket että `with` vastaavat hyvin yleisiin käyttötapauksiin ja niistä oli palvelun toteuttamisessa suurta hyötyä. Etenkin `with`-rakennetta käytettiin lähes jokaisessa kontrollerissa, sillä sen avulla koodi saatiin pidettyä tiiviinä, mutta silti selkeänä.

```
with \
  {:ok, %DateTime{} = datetime} <- parse_timestamp(time),
  {:ok, datetime} <- check_datetime_diff(datetime),
  {:ok, pulse} <- create_pulse(user, machine, datetime),
  {:ok, inserted_xps} <- create_xps(pulse, xps),
  :ok <- update_caches(inserted_xps)
do
  put_status(conn, 201)
else
  {:error, :not_found} ->
    put_status(conn, 404)

  {:error, :generic} ->
    put_status(conn, 400)

  {:error, :internal} ->
    put_status(conn, 500)
end
```

Koodilistaus 8: Esimerkki funktioiden ehdollisesta suorittamisesta with-syntaksilla.

6.1.4 Makrot

Kuten luvussa 3 esitettiin, Elixiriä voi laajentaa makrojen avulla. Makroilla voi luoda uutta syntaksia, joka muutetaan käännösaikaisesti varsinaiseksi Elixiriksi. Niiden käyttö ei ole kuitenkaan ongelmatonta. Elixirin aloitusopas varoittaa makrojen olevan vaikeampia kirjoittaa kuin tavallisten funktioiden, eikä niitä tulisi käyttää ellei niistä saa merkittävää hyötyä tilanteessa. Oppaan mukaan ”eksplisiittinen on parempi kuin implisiittinen”, sillä makroilla on helppoa vahingossa koodauksen helpottamisen sijasta piilottaa se, mitä koodi oikeasti tekee. [32]

Esimerkkipalvelun toteuttamisessa ei tullut esiin sellaisia tilanteita, joissa olisi tarvinnut toteuttaa jotain käyttämällä makroja. Sen sijaan muun muassa Phoenix Frameworkiin ja Ectoon on toteutettu useita ohjelmointia helpottavia makroja, joiden avulla koodista saa helpommin luettavaa ja ymmärrettävää. Esimerkiksi palvelun reititys tapahtuu seuraavanlaisilla koodiriveillä, joissa on HTTP-metodi, sivulle haluttu osoite sekä pyynnön käsittelevän kontrollerin moduuli ja funktio: `get "/machines/:id", MachineController, :view_single`. Phoenixiin toteutettu *get*-makro muokkaa riviä niin, että lopullinen käännetty reititinkoodi on sarja funktioita, jotka valitsevat pyynnölle oikean käsittelyfunktion käyttämällä merkkijonojen hahmonsovitusta. Hahmonsovitus on tähän erittäin tehokasta, makrojen avulla koodista saadaan samalla helposti kirjoitettavaa ja ymmärrettävää.

6.1.5 Funktionaalisuus ja datan muuttumattomuus

Elixir on funktionaalinen kieli, jossa kaikki data on muuttumatonta. Tämä tarkoittaa sitä, että tiettyä muistissa olevaa dataa ei ole mahdollista muuttaa. Nämä ominaisuudet

on peritty Erlangilta; toisin kuin Erlangissa, Elixirissä muuttujan voi kuitenkin osoittaa uudelleen johonkin toiseen muistipaikkaan. Näin vältytään muun muassa Erlangissa välillä käytettävältä muuttujien numeroinnilta, jolla pyritään kiertämään muuttujien yhden asetuskerran rajoitus [33].

Kielen funktionaalisuus aiheutti suunnitteluvaiheessa ohjelman rakenteeseen suuria eroja aiemmin opittuun oliokeskeiseen suunnitteluun verrattuna. Ohjelmaa suunniteltaessa mietittiin olioiden sijaan uudelleenkäytettäviä funktioita ja tiedon kulkua niiden läpi. Sen sijaan, että ohjelman tilaa säilytetään olioissa, jotka piilottavat sen sisälleen, tila on jatkuvasti eksplisiittisesti näkyvillä. Jos funktiot pidetään niin kutsutusti *puhtaina* (*pure*) – eli niiden kutsuminen ei aiheuta sivuvaikutuksia – on ohjelmoijan helppo päätellä ohjelman tila tietyn funktiokutsun jälkeen. Elixir ei kuitenkaan pakota funktioita olemaan puhtaita, joten ohjelmoijan on huolehdittava itse oikeasta tasapainosta. Esimerkiksi tietokannan kanssa kommunikoidessa tilaan ei voi luottaa samalla tavalla, sillä se riippuu tietokannan silloisesta tilasta. Funktionaalisuus tuntui kuitenkin selkiyttävän koodin seuraamista suurimmassa osassa tilanteista.

Datan muuttumattomuus aiheuttaa aluksi työtä koodin suunnittelussa, koska olio-ohjelmoinnista tutut tekniikat eivät enää toimi. Esimerkiksi listan iteroivassa `Enum.each`-funktiossa ei voi muuttaa funktion ulkopuolista muuttujaa, vaan sen sijaan pitäisi käyttää `Enum.reduce`-funktiota, jolla lista redusoidaan elementti kerrallaan haluttuun arvoon. Vastapainona muuttumattomuus helpottaa muuttujien tilan päättelyä. Funktiota kutsuessa voi olla varma, ettei se muuta minkään ylemmän tason muuttujan tilaa. Kielissä, joissa esimerkiksi olioita voi siirtää funktiolta toiselle viitteinä, on mahdollista muuttaa vahingossa samaa oliota, joka on käytössä jossain muualla. Tällaisten ongelmien lähdettä on vaikea löytää, mutta Elixirissä niitä ei voi tapahtua. BEAM voi siis välittää kaikki muuttujat funktioille viitteinä, eikä niistä tarvitse tehdä kopioita jos funktio ei tee niistä omaa muokkaustaan. Näin voidaan säästää muistia ja suoritusaikaa.

Elixirin opettelu ensimmäisenä funktionaalisena kielenä oli alussa melko vaivaloista. Olio-ohjelmointiin tottuneena funktionaalisten rakenteiden suunnittelu vaati omien ajatusprosessien muuttamista ja useasti vastaus löytyi vasta pitkän etsinnän jälkeen. Sen sijaan muiden funktionaalisten kielten omaksuminen Elixirin jälkeen on ollut helpompaa, joten myös Elixirin oppiminen onnistunee nopeammin, jos on jo aiempaa kokemusta funktionaalisesta ohjelmoinnista. Code::Statsin toteuttamisen jälkeen eri ratkaisumallit alkavat syntyä jo luonnostaan. Funktionaalisuus ja datan muuttumattomuus ovat auttaneet luomaan luotettavampaa koodia jossa tietynlaisia virheitä ei voi tapahtua ja samoja ominaisuuksia on jäänyt kaipaamaan muista ohjelmointikielistä.

6.1.6 Prosessit ja hajautus

BEAMin kevyt prosessimalli mahdollistaa asioita, joita perinteisillä käyttöjärjestelmän prosesseilla tai säikeillä on vaikeampi tehdä. Yhden BEAMin prosessin vaatima muistialue on sen käynnistyessä hyvin pieni, vähimmillään vain 309 sanaa, joka on tyypilli-

sessä 64-bittisessä järjestelmässä hieman alle 2,5 kilotavua [34]. Prosessien välinen kontekstivaihto on myös huomattavasti käyttöjärjestelmän kontekstivaihtoa kevyempi, koska se tapahtuu virtuaalikoneen sisällä ja täysin käyttöjärjestelmän ytimen ulkopuolella. Keveyden vuoksi BEAMin prosesseja voi käynnistää ongelmitta satoja tuhansia tai jopa miljoonia samalla koneella.

Esimerkkinä kevyiden prosessien tarjoamasta mahdollisuudesta Cowboy käyttää sisäisesti pyyntöjen palvelemiseen kirjastoa nimeltä *Ranch*, joka käynnistää jokaista pyyntöä varten oman prosessin, joka suljetaan pyynnön käsittelyn jälkeen [27]. Tapa yhdistää moniajavien ja tapahtumapohjaisten palvelinten hyvät puolet. Yhden prosessin jäädessä odottamaan ulkoista syötettä voidaan ajoon ottaa toinen ja maksimoida näin suorittimen käyttö. Samalla prosessien ohjelmakoodia ei tarvitse kuitenkaan kirjoittaa tapahtumapohjaisen palvelimen vaatimaan takaisinkutsutyyliin. Prosessit tarjoavat myös suojaa, sillä muiden prosessien muistiin ei voi koskea, muistiin ei jää tietoja seuraavalle pyyntöprosessille eikä kaatuva prosessi vaikuta muihin käsittelijöihin.

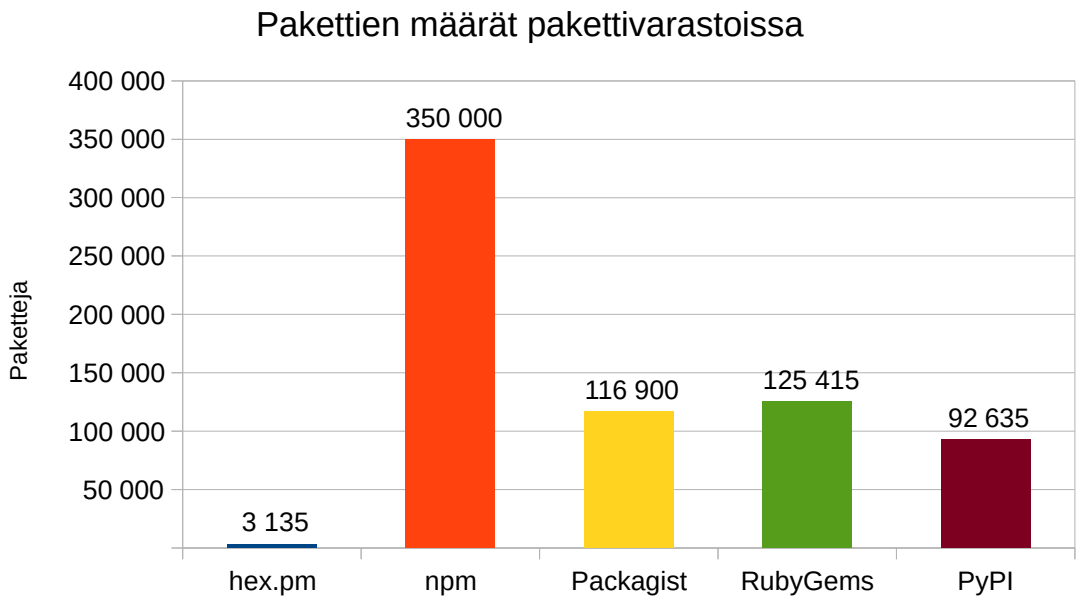
Hajautus virtuaalikoneen sisällä tarjoaa myös muita hyötyjä. Uuden prosessin käynnistäminen ja valvominen on yksinkertaista ja BEAM tarjoaa siihen valmiit työkalut. Prosesseilla on automaattisesti viestijonot ja niiden välillä voi siirtää mitä tahansa kielen tietotyyppejä, jopa funktioviittauksia [35]. Code::Statsissa tätä käytettiin hyödyksi esimerkiksi tausta-ajojen suorittamisessa. Kun kontrollerissa tarvitsee esimerkiksi käynnistää tausta-ajona käyttäjän tietokantaan tallennetun välimuistin päivittäminen, voidaan se käynnistää kutsulla `Task.start(User, :update_cached_xps, [user, true])`. Argumentteina käynnistettävälle prosessille voidaan antaa lista mitä tahansa tietotyyppejä, eikä niitä tarvitse serialisoida välissä esimerkiksi merkkijonoksi. Välissä ei tarvita myöskään erillistä viestijonoa, kuten vertailupalvelimen toteutuksessa, koska kaikki on saman virtuaalikoneen sisällä.

6.1.7 Ekosysteemi ja työkalut

Elixir on verrattain uusi kieli, sillä sen ensimmäinen vakaa versio julkaistiin vasta kaksi vuotta sitten. Tästä johtuen myös kielen ekosysteemi on toistaiseksi hyvin pieni. Erlang on ohjelmointikielten joukossa vanha, mutta myös sen käyttö on rajoittunut pienelle joukolle. Kuvassa 12 olevasta kaaviosta nähdään eri ohjelmointikielten yhteisöjen suhteellisia kokoja. Vaikka paketinhallintajärjestelmien pakettien määrät eivät vastaakaan yksi yhteen kielen käyttäjien kanssa, voidaan niistä vetää johtopäätöksiä siitä, miten suosittuja tietyt kielet ovat. Ylivoimaisena johtajana on JavaScript-paketteja säilyttävä *npm* yli 350 000 paketin varastollaan. PHP:n *Packagist*, Rubyn *RubyGems* sekä Pythonin *PyPI* ovat kaikki noin 100 000 paketin tuntumassa, kun taas Elixirin hex.pm sisältää vain vähän yli 3 000 pakettia.

Pakettien määrällä on suora vaikutus kielellä työskentelyyn. Ohjelmoidessa ratkotaan usein samoja ongelmia ja integroidutaan samoihin järjestelmiin kuin muutkin. Suositulla kielellä on valmiina suuri pakettivarasto kirjastoja ja kehitystyökaluja. Näiden

käyttäminen mahdollistaa tuottavuuden suuren kasvun, kun ajan voi käyttää itse toteuttamisen sijaan valmiin kirjaston integroimiseen. Valmiilla kirjastoilla on aina omat painopisteensä ja heikot kohtansa; laajasta pakettivalikoimasta löytyy helpommin toteutus, jossa on projektille juuri sopivat ominaisuudet. Ekosysteemin koko vaikuttaa myös suoraan siihen, kuinka paljon ohjelmoijan käytettävissä on työntekoa auttavia resursseja, kuten ohjeita ja tukisivustoja.



Kuva 12: Eri ohjelmointikielten paketinhallintajärjestelmien pakettien määrät syksyllä 2016. Perustuu lähteisiin [36, 37, 38, 39, 40].

Elixirin kohdalla valinnanvaraa ei ole paljoa. Verkkopalveluiden tekemiseen tarkoitettuja sovelluskehyksiä on aktiivisessa kehityksessä vain kourallinen ja niistä Phoenix Framework on ainoa, joka tähtää kaikenkattavaksi ratkaisuksi verkkokehitykseen. Kielelle löytyy matalan tason kirjastot kaikille yleisimmille tietokannoille, mutta Ecto on ainoa tietokantariippumaton korkeamman tason kirjasto. Elixir-koodista voi käyttää Erlang-kirjastoja ilman ajonaikaista kustannusta, mutta myöskään Erlangin tarjonta ei ole kovin suurta verrattuna yleisempiin kieliin. Suurin osa peruskäyttötapauksista on kuitenkin katettu vähintään yhdellä kirjastolla ja kaikille yleisimmille tekstieditoreille ja kehitysympäristöille on olemassa editorituki automaattisella täydennyksellä ja syntaksin värityksellä. Harrastelijalle pakettien vähyys ei ole ongelma, mutta yritysten tulee harkita mahdollisesti itse toteutettavien osioiden kustannusta. [41]

Elixirillä on kuitenkin hyvät mahdollisuudet kasvattaa suosiotaan. Kielellä alkuun pääseminen oli yksinkertaista sen tarjoamien työkalujen ansiosta. Elixirin monipuolisella projektityökalu Mixillä voi luoda yhdellä komennolla uuden projektipohjan, joka sisältää projektin perusrakenteen, konfiguraatiotiedoston, yksikkötestipohjan ja valinnaisesti myös valvontapuun, jonka projekti käynnistää. Mix konfiguroidaan projektikohtaisesti *mix.exs*-tiedostolla, jossa määritellään muun muassa projektin riippuvuudet. Riip-

puvuudet voidaan hakea komennolla `mix deps.get`, joka käyttää oletuksena hex.pm-pakettivarastoa. Riippuvuuksien tarkat versiot tallennetaan *mix.lock*-tiedostoon, jonka jakamalla kukin projektin kehittäjä saa käyttöönsä samat versiot. Mixin avulla voi lopulta myös julkaista projektinsa hex.pm-pakettivarastoon ilman, että tarvitsee poistua komentoriviltä.

6.2 Esimerkkipalvelun rakenne

Code::Stats noudattaa Phoenix Frameworkin MVC-rakennetta, joka on esitetty aiemmin kuvassa 8. Rakenteella pyritään hajottamaan sovellus kokonaisuuksiin, joilla on omat selkeät vastuualueet. Tässä onnistuttiin pääosin hyvin. Alakohdassa 6.1.3 kuvatun `with`-syntaksin avulla kontrollerien kontrollivuot ja virhetilanteiden käsittelykoodit pystyttiin pitämään selkeinä ja vapaina turhasta toistosta. Elixirin sisäänrakennettu sivupohjamoottori *EEx* teki HTML-sivupohjien kirjoittamisesta erittäin helppoa, sillä niissä pystyi käyttämään suoraan Elixirin omaa syntaksia. Phoenix sisältää lisäksi useita EEx:ssä käytettäviä apufunktioita muun muassa HTML-lomakkeiden, linkkien ja resurssien osoitteiden generointiin.

BEAMin prosessimalli tuo palvelulle luotettavuutta. Jos HTTP-pyyntöä tai WebSocket-yhteyttä käsittelevä prosessi kaatuu käsittelemättömään virheeseen, muut käsittelijäprosessit jatkavat normaalisti. Tämä takuu saadaan automaattisesti, kun ohjelma järjestetään valvontapuumallin mukaisesti. Code::Statsista löytyi muutamia virheitä tuotantoon oton jälkeen, mutta vaikka ne johtivat käsittelijän kaatumiseen ja asiakkaalle ikävään virheilmoitukseen, yksikään niistä ei vaikuttanut järjestelmän vakauteen.

Elixir on dynaamisesti ja vahvasti tyypitetty kieli, eikä tyyppejä voi sen vuoksi tarkastaa etukäteen käännösvaiheessa. Koska tyypit voi tarkastaa vasta ajonaikaisesti, on ohjelmakoodiin mahdollista kirjoittaa ohjelmointivirheitä, jotka paljastuvat vasta kun koodi käyttää vääräntyyppistä arvoa. Pahimmillaan koodi ei kaadu, vaan jatkaa vääräntyyppisen arvon käyttämistä, ennalta-arvaamattomin seurauksin. Elixirissä tämän vaaraa voi vähentää käyttämällä hahmonsovitusta kattavasti. Kirjoittamalla funktiot, ehtolauseet ja sijoitukset niin, että ne hyväksyvät vain oikeanlaisia syötteitä, saadaan koodiin samalla sekä tyyppi- että sisältötarkistukset. Näin virheen sattuessa prosessi kaatuu ja tekee sen mahdollisimman lähellä virheen syntypaikkaa. Haluttaessa suurempaa varmuutta voidaan käyttää Erlangille toteutettua staattista koodianalysaattoria *Dialyzeria*, joka pystyy analysoimaan myös Elixiristä käännettyjä BEAM-tavukooditiedostoja [42].

Mikäli ohjelmointikielessä ei ole suuria puutteita, verkkopalvelun turvallisuus ei ole suoraan kielen asia. Kielen ekosysteemi vaikuttaa kuitenkin tehtäviin ratkaisuihin. Projektin ohjelmointiosuutta aloitettaessa kielelle ei ollut saatavilla sopivan yksinkertaista käyttäjän autentikointi- ja autorisointikirjastoa, joten nämä toiminnot toteutettiin itse. Tärkeiden turvallisuusominaisuuksien toteuttaminen itse suositun ja testatun kirjaston käyttämisen sijasta on aina riski jota pitää harkita tarkkaan erityisesti, jos kyseessä

ei ole harrastusprojekti. Sittemmin kielelle on tullut saataville useita eri vaihtoehtoja autentikointiin ja autorisointiin [41], mutta vaihtoehtoja on edelleen vähän verrattuna suositumpiin ohjelmointikieliin.

6.3 Suorituskykytestien tulokset

Suorituskykytestien tulokset löytyvät seuraavista alakohdista. Yhteisenä kaikille tuloksille oli se, että mittausalustan pienestä suoritustehosta ja hitaasta massamuistista johtuen saadut tehokkuusarvot ovat melko pieniä. Niitä voidaan kuitenkin vertailla keskenään ja nähdä, mikä tekniikka saa käytettyä alustaa tehokkaimmin. Viimeisessä alakohdassa 6.3.6 käydään läpi käytetyistä menetelmistä johtuvat mittaustulosten rajoitteet.

Viimeistä lukuun ottamatta kussakin testissä käytettiin Siegen vipuja `-b -H 'Cache-control: no-cache'`, joilla testiohjelma asetetaan rasitustestitilaan ja palvelinta ohjeistetaan kiertämään mahdolliset välimuistit. Kun välimuisteja ei käytetä, voidaan mitata tarkemmin palvelinten pyyntöjen käsittelyn suorituskykyä.

PHP-toteutusta vertailtaessa käynnistettiin aina myös Ratchet, vaikka testissä ei mitattaisikaan WebSocket-yhteyksiä, sillä Ratchetin katsotaan olevan olennainen osa palvelua. PHP:tä ajettiin PHP-FPM-sovelluspalvelimella, jonka käsittelijäprosessien määräksi asetettiin kiinteä 20. Tämän suuremmilla käsittelijämäärillä ei havaittu olevan juurikaan positiivista vaikutusta testituloksiin.

6.3.1 Staattisten sisältöjen palveleminen

Staattisten sisältöjen palveleminen on verkkopalvelimille erittäin yksinkertaista ja sen vuoksi myös tehokasta. Molemmat vertaillut palvelimet pystyivät käsittelemään pyyntöjä niin tehokkaasti, ettei kuormaa lähettänyt kone pystynyt luomaan niitä enempää, minkä vuoksi palvelinten murtumispistettä ei voitu löytää. Asiakasmäärää kasvattamalla kuitenkin havaittiin, että sadan yhtäaikaisen asiakkaan jälkeen asiakasmäärän kasvulla oli lievä tuloksia heikentävä vaikutus, joten testit päätettiin suorittaa sadalla yhtäaikaisella asiakkaalla.

Testiin käytettiin Siegen vipuja `-c 100 -r 200`, joilla 100 asiakasta suorittaa kukin 200 pyyntöä eli yhteensä 20 000 pyyntöä. Mittaustulokset näkyvät taulukoissa 2 ja 3. Niistä voidaan havaita, että molemmat vertaillut palvelimet pystyvät pitämään suorituskykynsä hyvin tasaisina eikä niillä ole ongelmia käsitellä kaikkia saapuneita pyyntöjä. Pisimmän ja lyhimmän pyynnön kestojen ero selittyy tiedostojen kokoeroilla: pyydetty JavaScript-tiedosto *app.js* on 230 kilotavun koollaan lähes 10 kertaa suurempi kuin 26 kilotavun kokoinen PNG-muotoinen logokuva. Kummallakaan palvelimella yksikään pyyntö ei epäonnistunut.

Tuloksista voidaan siis nähdä, että Code::Stats on vain noin prosentin verran hitaampi kuin Nginx palvellessaan samoja tiedostoja. Sekä Nginx että Phoenix Frameworkin käyttämä Plug hyödyntävät alustalla Linux-käyttöjärjestelmän *sendfile*-järjestelmä-

kutsua, joka mahdollistaa tiedostojen lähettämisen tekemättä niiden sisällöstä ylimääräistä kopiota kutsuvan ohjelman muistiin. Tämän tekniikan avulla suorituskykyä rajoittaa eniten käyttöjärjestelmän tehokkuus.

Taulukko 2: Nginx-palvelimen mittaustulokset.

	Keskimäär. kesto (s)	**Pyyntö-taajuus (p/s)**	**Pisin kesto (s)**	**Lyhin kesto (s)**	**Virheet (%)**
	1,14	87,73	3,35	0,03	0,00
	1,13	87,89	3,23	0,05	0,00
	1,13	87,99	3,20	0,01	0,00
	1,13	87,97	3,24	0,01	0,00
	1,14	87,84	3,24	0,04	0,00
Keskiarvo	1,13	87,88	3,25	0,03	0,00

Taulukko 3: Code::Stats-palvelimen mittaustulokset.

	Keskimäär. kesto (s)	**Pyyntö-taajuus (p/s)**	**Pisin kesto (s)**	**Lyhin kesto (s)**	**Virheet (%)**
	1,14	87,26	3,38	0,04	0,00
	1,14	87,10	3,40	0,03	0,00
	1,15	86,83	3,39	0,04	0,00
	1,14	87,11	3,36	0,04	0,00
	1,14	87,13	3,41	0,04	0,00
Keskiarvo	1,14	87,09	3,39	0,04	0,00

6.3.2 Etusivun palveleminen vierastunnuksille

Tässä testissä päätekijänä on ohjelmien kyky renderöidä HTML-sivupohja nopeasti. Koska asiakkaat eivät olleet sisäänkirjautuneita, ei pyynnöistä aiheutunut tietokantakyselyitä. Näin ollen ohjelman tuli vain laskea muutama kokemuspistearvo välimuistista löytyvien tietojen perusteella ja muodostaa niistä valmis sivu lähetettäväksi. PHP-ohjelmaan ei yksinkertaistuksen vuoksi toteutettu välimuistia, vaan arvot kirjoitettiin suoraan lähdekoodiin. Vertailun vuoksi palveltiin Nginxillä valmiiksi renderöityä HTML-sivua.

Testit suoritettiin Siegen vivuilla `-c 100 -r 200`, joilla 100 yhtäaikaista asiakasta lähettää kukin 200 pyyntöä, eli yhteensä 20 000 pyyntöä palveluun. Asiakasmäärä valittiin samaksi kuin edellisessä testissä sen vuoksi, että rajoittavana tekijänä oli jälleen kuorman luoneen tietokoneen suorituskyky. Mittaustulokset ovat nähtävissä taulukoissa 4, 5 ja 6. Nginxin tulokset osoittavat, että testin kohteena olevan HTML-sisällön palve-

leminen on erittäin tehokasta. Muilla toteutuksilla ylivoimaisesti suurin osa ajasta menee siis kyseisen sisällön tuottamiseen järjestelmän tiedoista.

Mittaustuloksista voi päätellä, että Code::Stats on PHP-vertailuohjelmaa huomattavasti tehokkaampi palvelemaan etusivua. Pyyntöjen keskimääräinen kesto on Code::Statsilla noin 4,5 kertaa lyhyempi ja vastaavasti palvelu voi käsitellä noin 4,5 kertaa enemmän pyyntöjä sekunnissa. Myös pyyntöjen kestojen vaihtelu on pienempi, pisimmän ja keskimääräisen keston väli on noin 0,4 sekuntia pienempi kuin PHP-toteutuksella.

Taulukko 4: Nginx-palvelimen mittaustulokset.

	Keskimäär. kesto (s)	**Pyyntö-taajuus (p/s)**	**Pisin kesto (s)**	**Lyhin kesto (s)**	**Virheet (%)**
	0,10	998,00	0,53	0,00	0,00
	0,10	963,39	0,49	0,00	0,00
	0,10	1 013,68	0,50	0,00	0,00
	0,10	1 021,45	0,38	0,00	0,00
	0,10	994,04	0,35	0,00	0,00
Keskiarvo	0,10	998,11	0,45	0,00	0,00

Taulukko 5: Code::Stats-palvelimen mittaustulokset.

	Keskimäär. kesto (s)	**Pyyntö-taajuus (p/s)**	**Pisin kesto (s)**	**Lyhin kesto (s)**	**Virheet (%)**
	0,23	430,66	0,48	0,01	0,00
	0,23	432,62	0,45	0,01	0,00
	0,23	432,99	0,40	0,01	0,00
	0,23	431,78	0,37	0,01	0,00
	0,23	431,03	0,38	0,01	0,00
Keskiarvo	0,23	431,82	0,42	0,01	0,00

Taulukko 6: PHP-palvelimen mittaustulokset.

	Keskimäär. kesto (s)	**Pyyntö-taajuus (p/s)**	**Pisin kesto (s)**	**Lyhin kesto (s)**	**Virheet (%)**
	1,05	95,00	1,62	0,04	0,00
	1,05	95,03	1,62	0,02	0,00
	1,05	95,10	1,62	0,03	0,00
	1,05	94,98	1,69	0,04	0,00
	1,05	94,94	1,65	0,04	0,00
Keskiarvo	1,05	95,01	1,64	0,03	0,00

6.3.3 Profiilisivun palveleminen

Verkkopalvelut ovat yleensä tekemisissä jonkinlaisen tietokannan kanssa. Code::Statsin profiilisivu on esimerkki sivusta, joka tekee useita raskaita tietokantakyselyjä ja yhdistelee niistä saatuja tietoja. Tietokannasta haetaan muun muassa käyttäjän kokemuspisteet kokonaisuudessaan sekä viimeisen 12 tunnin ajalta. Viimeisimmät kokemuspisteet lisätään kokonaiskokemuspisteisiin ja kaikki tallennetaan takaisin tietokantaan, mikä vie aikaa. Tietokantaoperaatioiden hitautta pahentaa mittausalustan erityisen hidas massamuisti.

Tietokannan hitauden vuoksi suurimmaksi mahdolliseksi asiakasmääräksi jäi vain 30, joten testit ajettiin Siegen vivuilla `-c 30 -r 200`, pyyntöjen kokonaismääräksi tullen 6 000. Mittaustulokset ovat taulukoissa 7 ja 8. Code::Statsin ensimmäisen mittauksen lyhimmän keston tieto jätettiin pois epäluotettavana, sillä Siege laski sen epäonnistuneesta pyynnöstä.

Tuloksista huomaa, että molemmat toteutukset hidastuivat tietokannan johdosta merkittävästi. Code::Stats oli jopa 14 kertaa hitaampi kuin edellisessä testissä PHP:llä muutos ei ollut niin suuri. Samoin molemmille toteutuksille alkoi tulla yksittäisiä epäonnistuneita pyyntöjä. Epäonnistuneiden pyyntöjen määrä vaihteli mittausalustan kuormituksen mukaan, mutta oli kuitenkin hieman suurempaa PHP-palvelimella. Epäonnistumisille ei voi kuitenkaan asettaa kovin suurta painoarvoa niiden esiintymistiheyden suuren vaihtelun vuoksi. PHP-toteutus oli Code::Statsia keskimäärin noin 1,5 kertaa hitaampi pyyntöjen käsittelyssä ja sen pisimmät pyynnöt kestivät hieman pidempään.

Taulukko 7: Code::Stats-palvelimen mittaustulokset. Viivalla merkitty tulos on jätetty huomiotta epäluotettavana.

	Keskimäär. kesto (s)	**Pyyntö-taajuus (p/s)**	**Pisin kesto (s)**	**Lyhin kesto (s)**	**Virheet (%)**
	0,98	30,44	3,38	-	0,05
	0,97	30,62	3,43	0,11	0,00
	0,97	30,59	4,73	0,12	0,00
	0,97	30,64	4,20	0,11	0,00
	0,96	31,00	1,90	0,12	0,00
Keskiarvo	0,97	30,66	3,53	0,12	0,01

Taulukko 8: PHP-palvelimen mittaustulokset.

	Keskimäär. kesto (s)	**Pyyntö-taajuus (p/s)**	**Pisin kesto (s)**	**Lyhin kesto (s)**	**Virheet (%)**
	1,35	22,01	2,90	0,15	0,00
	1,46	20,41	5,16	0,13	0,68
	1,52	19,52	5,24	0,13	1,27
	1,36	21,91	4,28	0,15	0,00
	1,36	21,97	3,66	0,14	0,00
Keskiarvo	1,41	21,16	4,25	0,14	0,39

6.3.4 Tietokantaa käyttävien ja käyttämättömien pyyntöjen yhtäaikainen palveleminen

Kun pyyntöjä käsittelevä palvelin odottaa vastausta tietokannalta, on tehokkainta, jos se voi käsitellä samaan aikaan muita pyyntöjä. Tämän testin tarkoituksena oli mitata tätä yleistä tilannetta. Testi ajettiin samoilla arvoilla kuin edellinen testi, eli `-c 30 -r 200`, joka johti 6 000 pyyntöön. Myös tällä kertaa jouduttiin poistamaan Code::Statsin arvoista yksi datapiste, jonka Siege ilmoitti väärin.

Mittaustulokset löytyvät taulukoista 9 ja 10. Tässä testissä Phoenix Frameworkin käyttämä Cowboy-palvelin ja BEAM-virtuaalikone näyttävät etunsa. Koska jokaiselle pyynnölle avataan oma prosessi, ei toisten prosessien odottava tila vaikuta uusiin pyyntöihin, vaan ne voidaan käsitellä välittömästi. Käyttöjärjestelmän prosesseja käyttävän PHP-FPM:n tapauksessa 20 prosessin raja tulee nopeasti vastaan, jolloin kaikki prosessit odottavat tietokantaa eikä uusia pyyntöjä voi käsitellä. PHP-FPM:n voi asettaa myös dynaamisen tilan jossa prosesseja voi käynnistää lisää tarpeen mukaan, mutta käyttöjär-

jestelmätason prosesseina niiden muistinkäyttö on huomattavasti suurempi kuin Elixirin prosessien. Dynaamisuudesta saatava skaalautuvuushyöty on siis rajallinen.

Taulukko 9: Code::Stats-palvelimen mittaustulokset. Viivalla merkitty tulos on jätetty huomiotta epäluotettavana.

	Keskimäär. kesto (s)	**Pyyntö-taajuus (p/s)**	**Pisin kesto (s)**	**Lyhin kesto (s)**	**Virheet (%)**
	0,53	56,07	5,25	-	0,73
	0,45	66,37	1,77	0,01	0,00
	0,45	66,29	2,25	0,01	0,00
	0,45	65,84	2,64	0,01	0,00
	0,45	66,45	1,59	0,01	0,00
Keskiarvo	0,47	64,20	2,70	0,01	0,15

Taulukko 10: PHP-palvelimen mittaustulokset.

	Keskimäär. kesto (s)	**Pyyntö-taajuus (p/s)**	**Pisin kesto (s)**	**Lyhin kesto (s)**	**Virheet (%)**
	0,83	35,89	3,25	0,03	0,00
	0,83	35,88	3,77	0,03	0,00
	0,82	36,17	2,02	0,04	0,00
	0,82	36,10	2,54	0,05	0,00
	0,82	36,12	2,03	0,03	0,00
Keskiarvo	0,82	36,03	2,72	0,04	0,00

6.3.5 Reaaliaikapäivitysten palveleminen

Reaaliaikapäivitysten testissä mitataan, kuinka paljon kuuntelijoita palvelu pystyy tukemaan. Testiä toistettiin nostamalla kuuntelijoiden määrää sadalla kerrallaan, kunnes virheiden määrä rajapintapyynnöissä tai WebSocket-yhteyksissä nousi liian suureksi. Code::Stats pärjäsi vielä 700 yhtäaikaisen kuuntelijan kanssa ennen kuin rajapintapyyntöjen virheet lähtivät kasvamaan suuremmin. PHP-palvelin sen sijaan pystyi palvelemaan 600 käyttäjää, jonka jälkeen sen WebSocket-yhteydet alkoivat epäonnistua. Testeissä käytetyt lopulliset Tsungin asetukset löytyvät samasta git-repositoriosta kuin PHP-toteutuksen koodi.

Taulukoissa 11 ja 12 ovat saadut mittaustulokset. Tulokset ovat kahtiajakoiset. Code::Stats pystyi palvelemaan samanaikaisesti 100 kuuntelijaa enemmän kuin PHP-toteutus ilman yhtään WebSocket-yhteyksien virhetilannetta, mutta rajapintapyyntöjen palveleminen ei toiminut yhtä luotettavasti. PHP-palvelin oli keskimäärin hieman hi-

taampi palvelemaan rajapintapyyntöjä ja pärjäsi Code::Statsia pienemmällä kuuntelijamäärällä WebSocket-yhteyksien kanssa ajoittain erittäin huonosti, epäonnistuen pahimmillaan vastaanottamaan lähes viidesosan yhteyksistä. Sen sijaan sen rajapintapyynnöistä epäonnistuivat vain hyvin harvat.

Taulukko 11: Code::Stats-palvelimen mittaustulokset 700 kuuntelijalla.

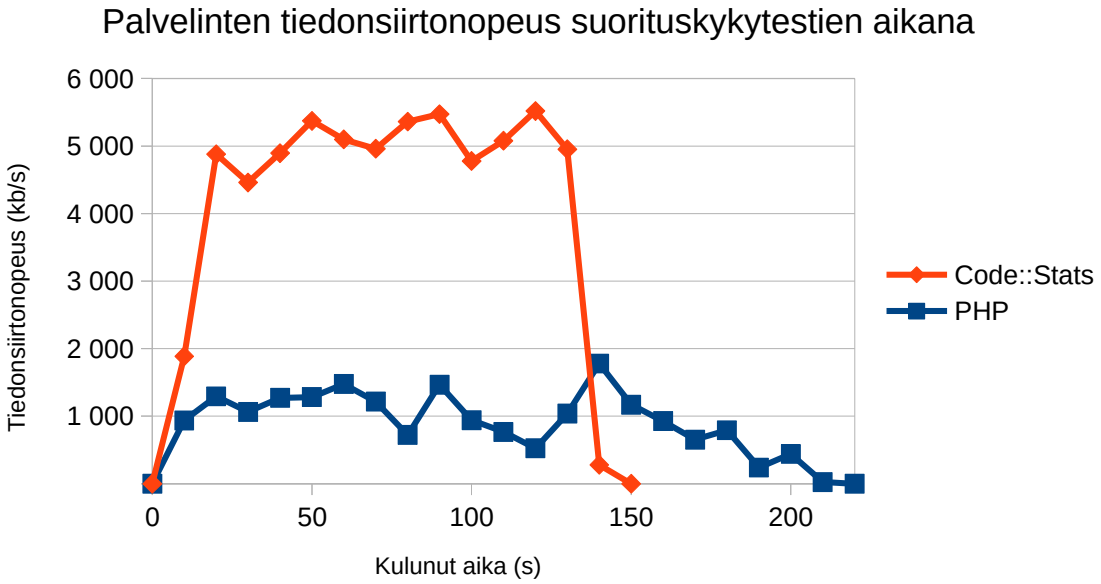
	Rajapintapyyntöjen pyyntötaajuus (p/s)	**Rajapintapyyntöjen virheet (%)**	**WebSocket-yhteyksien virheet (%)**
	26,54	1,65	0,00
	22,90	1,35	0,00
	24,84	2,55	0,00
	25,14	1,33	0,00
	25,41	3,80	0,00
Keskiarvo	24,97	2,14	0,00

Taulukko 12: PHP-palvelimen mittaustulokset 600 kuuntelijalla.

	Rajapintapyyntöjen pyyntötaajuus (p/s)	**Rajapintapyyntöjen virheet (%)**	**WebSocket-yhteyksien virheet (%)**
	22,85	0,50	0,33
	22,29	0,22	0,17
	23,05	0,73	11,00
	22,89	0,23	18,17
	21,28	0,42	2,33
Keskiarvo	22,47	0,42	6,4

Yllä mainitut luvut eivät kuitenkaan kerro koko totuutta. Tarkastelemalla kuvassa 13 havainnollistettavia palvelinten lähettämien rajapintapyyntöjen vastausten sekä reaaliaikapäivitysten yhteenlaskettuja tiedonsiirtonopeuksia voidaan saada toisenlainen kuva suorituskyvystä. Kuvaajasta voidaan huomata kaksi eri asiaa. Ensinnäkin Code::Statsin tiedonsiirtonopeus on keskimäärin huomattavasti suurempi kuin PHP-toteutuksen. Osa erosta liittyy siihen, että Code::Statsilla oli enemmän kuuntelijoita, mutta se ei selitä kaikkea, sillä kuuntelijamäärien ero oli vain noin 1,2-kertainen. Tämä viittaa siihen, että joko Code::Stats lähettää viestit nopeammin kuin PHP-toteutus tai PHP-palvelin jättää osan viesteistä lähettämättä kokonaan.

Toisena erona kuvaajasta voi huomata, että keskimäärin PHP-palvelimen testeissä on mennyt enemmän aikaa kuin Code::Statsilla. Jälkimmäinen käyttikin pisimpään kestäneeseen testiin alle 150 sekuntia, kun PHP:llä suoritetut testit kestivät 180 sekunnista 210 sekuntiin.



Kuva 13: Code::Statsin ja vertailupalvelimen tiedonsiirtonopeudet testien aikana. Arvot ovat testien keskiarvoja.

6.3.6 Mittaustulosten rajoitteet

Saaduilla mittaustuloksilla on tiettyjä rajoitteita, jotka johtuvat käytetyistä menetelmistä. Rajoitteet tulee tunnistaa, kun tulosten perusteella tehdään johtopäätöksiä. Suurimpana rajoitteena on se, että vertailua varten tehtiin vain yksi vaihtoehtoinen toteutus. Tulokset olisivat luotettavampia, jos niitä voisi verrata useampaan eri tekniikalla tehtyyn palvelimeen. Valinta yhden palvelimen toteuttamisesta tehtiin diplomityön aikarajoitteiden vuoksi. Tekniikaksi valittiin PHP, jotta tulokset olisivat kaikesta huolimatta sen yleisyyden vuoksi mahdollisimman käyttökelpoisia.

Toisena rajoitteena on itse alustan suorituskyky. Osassa testejä alustan suorituskyky vaihteli testistä toiseen niin, että epäonnistuneiden pyyntöjen määrät muuttuivat merkittävästi. Rajoitteen vakavuutta vähennettiin suorittamalla kaikki testit viisi kertaa ja arvioimalla yksittäisten tulosten lisäksi kaikkien tulosten keskiarvoja. Mittaukset, joiden tulokset olivat selvästi heikompia kuin muiden samassa testissä, poistettiin ja suoritettiin uudelleen oikeamman tuloksen saamiseksi.

7 PÄÄTELMÄT JA JATKOKEHITYSEHDOTUKSET

Internetin alati kasvavat käyttäjämäärät ja uudet käyttäjäryhmät kuten mobiili ja asioiden Internet vaativat verkkopalveluilta uudenlaista skaalautuvuutta. Samalla toteutukseen käytettävillä teknologioilla tulee voida luoda muihinkin ohjelmistojen laatuvaatimuksiin vastaava palvelu ilman, että ohjelmakoodin tuottaminen muuttuu liian vaikeaksi. Elixir on uusi ohjelmointikieli, joka pyrkii vastaamaan näihin haasteisiin. Tässä diplomityössä toteutettiin esimerkkiverkkopalvelu Code::Stats. Toteuttamistyössä saatujen kokemusten ja lopullisen palvelun laadun perusteella voidaan arvioida kielen soveltuvuutta.

Elixir pohjautuu Erlangiin, jota on kehitetty 1980-luvulta lähtien. Erlangia on käytetty massiivista skaalautumista ja vikasietoisuutta vaativissa ohjelmistoissa, muun muassa puhelinkeskuksissa ja WhatsApp-pikaviestinjärjestelmässä. Sen virtuaalikone BEAM on siis testattu ja vakaa alusta, jonka päällä Elixir toimii. BEAM tarjoaa yksinkertaiset hajautusominaisuudet ja työkalut, joiden avulla voi mallintaa ohjelmaa vikasietoisilla valvontapuilla.

Ohjelmointikokemuksen perusteella Elixir on erittäin miellyttävä kieli käyttää. Funktionaalisuudesta, datan muuttumattomuudesta ja hahmonsovituksesta on ohjelmoidessa suurta hyötyä. Niiden avulla ohjelmakoodista saa tehtyä helposti selkeää ja luotettavaa. Syntaksiltaan kieli on selkeä ja siinä on yleisesti käytettyjä rakenteita yksinkertaistavia työkaluja kuten putket ja `with`. Kyseisten ominaisuuksien kattava hyödyntäminen vaatii kuitenkin opettelua, etenkin jos ohjelmoija ei ole entuudestaan tuttu funktionaalisen paradigman kanssa.

Myös makroilla voi päästä eroon ylimääräisestä toisteisesta koodista ja tehdä lopputuloksesta yksinkertaisemman ymmärtää. Toisaalta makrot piilottavat toteutusyksityiskohtia, joten niillä voi saada aikaan myös sekavaa koodia, jonka ymmärtämiseen vaaditaan paljon työtä. Phoenixiä käytettäessä ei juuri tule vastaan tilanteita, joissa jonkin asian toteuttamiseen pitäisi tehdä oma makro, mutta kielen ja sovelluskehyksen valmiita makroja tulee käytettyä runsaasti. Makrot ovatkin oikeissa käsissä hyödyllisiä ja voimakkaita työkaluja.

Elixirin ekosysteemi on yleisesti käytettyihin kieliin verrattuna vielä pieni. Saatavilla on vain pieni määrä erilaisia kirjastoja sekä työkaluja, ja esimerkiksi kokonaisvaltaisia verkkopalveluiden sovelluskehyksiä on aktiivisessa kehityksessä vain yksi: Phoenix Framework. Ekosysteemin pienuuden vuoksi myös apuresursseja on saatavilla vä-

hemmän. Harrastelijoille asia ei ole välttämättä ongelma, mutta Elixiriä kaupalliseen käyttöön harkitsevien tulee punnita asiaa tarkkaan.

Toteutetun palvelun perusteella Elixirin suorituskyky on perinteistä PHP-verkkopalvelinta parempi. Palvelu on PHP-versiota nopeampi kaikilla testatuilla osa-alueilla, eron ollen osassa huomattava. Tuloksia tulisi kuitenkin vertailla myös muihin tyypillisiin verkkoteknologioihin, jotta suorituskyvystä saadaan kattavampi kuva. Testejä olisi myös hyvä suorittaa erilaisilla alustoilla, jotta alustan aiheuttamat testitulosten hajonnat saadaan kitkettyä.

Code::Statsin jatkokehityksenä Elixirin tarjoamia hajautusominaisuuksia voitaisiin tutkia enemmän jakamalla toteutettu palvelu usealle erilliselle fyysiselle palvelimelle. Palvelun rakenteessa olisi myös mahdollista hyödyntää paremmin Elixirin tukea ohjelman jakamiseksi erillisiksi applikaatioikseen. Tällöin Phoenix Frameworkillä toteutettu verkkorajapinta olisi erillinen bisneslogiikan ja tietomallit sisältävästä applikaatiosta ja applikaatiot keskustelisivat toistensa kanssa tarjoamillaan rajapinnoilla.

LÄHTEET

[1] Internet Live Stats – Internet Users, verkkosivu. Saatavissa: http://www.internetlivestats.com/internet-users/ (viitattu 22.9.2015).

[2] D. Bosomworth, Mobile Marketing Statistics 2015, verkkosivu. 2015. Saatavissa: http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/ (viitattu 22.9.2015).

[3] M. Pervilä, Asioiden internet paisuu lähivuosina, Tivi. 2014. Saatavissa: http://www.tivi.fi/CIO/2014-03-19/Asioiden-internet-paisuu-l%C3%A4hivuosina-3208875.html (viitattu 22.9.2015).

[4] Plataformatec, Elixir, verkkosivu. 2015. Saatavissa: http://elixir-lang.org/ (viitattu 1.10.2015).

[5] A. Vesanen, II Ohjelmointikielten kehityshistoriaa, Oulun yliopisto, Tietojenkäsittelytieteiden laitos. 2015. Saatavissa: https://noppa.oulu.fi/noppa/kurssi/815338a/luennot/815338A_luentokalvot_-_kielten_historiaa.pdf (viitattu 3.10.2015).

[6] T. Ruuska, Vaatimusmäärittely ketterässä ohjelmistokehityksessä, pro gradu, Jyväskylän yliopisto, 78 s. 2012. Saatavissa: https://jyx.jyu.fi/dspace/bitstream/handle/123456789/38590/URN:NBN:fi:jyu-201209202463.pdf?sequence=1 (viitattu 15.10.2015).

[7] A. Cremers, S. Alda, Organizational Requirements Engineering – Chapter 9, Non-functional Requirements, University of Bonn, Institute of Computer Science III. 2006. Saatavissa: http://www.academia.edu/7725732/Chapter_9_Non-functional_Requirements_Organizational_Requirements_Engineering (viitattu 15.10.2015).

[8] Who Is Hosting This?, Snowden's Global Impact, verkkosivu. 2014. Saatavissa: http://www.whoishostingthis.com/blog/2015/05/20/snowdens-global-impact/ (viitattu 15.10.2015).

[9] R. Reed, That's Billion with a B: Scaling to the next level at WhatsApp. 2014. Saatavissa: http://www.erlang-factory.com/static/upload/media/1394350183453526efsf2014whatsappscaling.pdf (viitattu 17.10.2015).

[10] R. Reed, Scaling to Millions of Simultaneous Connections. 2012. Saatavissa: http://www.erlang-factory.com/upload/presentations/558/efsf2012-whatsapp-scaling.pdf (viitattu 17.10.2015).

[11] F. Nah, A Study on Tolerable Waiting Time: How Long Are Web Users Willing to Wait? 2003. AMCIS 2003 proceedings. Paper 285. Saatavissa: http://aisel.aisnet.org/cgi/viewcontent.cgi?article=1751&context=amcis2003 (viitattu 21.9.2016).

[12] L. Brenna, D. Johansen, Engineering Push-Based Web Services, University of Tromsø. 2005. Saatavissa: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.61.649&rep=rep1&type=pdf (viitattu 25.9.2016).

[13] S. Loreto et al, Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP, Internet Engineering Task Force, verkkosivu. 2011. Saatavissa: https://tools.ietf.org/html/rfc6202 (viitattu 1.10.2016).

[14] I. Fette, A. Melnikov et al, RFC 6455 – The WebSocket Protocol, Internet Engineering Task Force, verkkosivu. 2011. Saatavissa: https://tools.ietf.org/html/rfc6455 (viitattu 24.9.2015).

[15] B. Erb, Concurrent Programming for Scalable Web Architectures, Ulm University, Institute of Distributed Systems. 2012. Saatavissa: https://oparu.uni-ulm.de/xmlui/bitstream/handle/123456789/2450/vts_8082_11772.pdf?sequence=1&isAllowed=y (viitattu 2.10.2016).

[16] J. Armstrong, A History of Erlang, Ericsson AB. 2007. Saatavissa: http://www.cse.chalmers.se/edu/year/2009/course/TDA381_Concurrent_Programming/ARCHIVE/VT2009/general/languages/armstrong-erlang_history.pdf (viitattu 18.10.2015).

[17] R. Virding, Hitchhiker's Tour of the BEAM, Erlang User Conference. 2014. Saatavissa: https://erlangcentral.org/euc-2014-robert-virding-hitchhikers-tour-of-the-beam/ (viitattu 4.11.2015).

[18] F. Hébert, Learn You Some Erlang for Great Good! No Starch Press, San Francisco, California, USA. 2013, 624 p. Saatavissa: http://learnyousomeerlang.com/ (viitattu 4.11.2015).

[19] Erlang STDLIB Reference Manual – ets, Ericsson AB. 2015. Saatavissa: http://erlang.org/doc/man/ets.html (viitattu (7.2.2016).

[20] J. Valim, Elixir Design Goals. 2013. Saatavissa: http://elixir-lang.org/blog/2013/08/08/elixir-design-goals/ (viitattu 9.11.2015).

[21] Ruby Rogues, RR Elixir with José Valim. 2013. Saatavissa: https://devchat.tv/ruby-rogues/114-rr-elixir-with-jose-valim (viitattu 9.11.2015).

[22] C. McCord, Phoenix Framework. 2015. Saatavissa: https://github.com/phoenixframework/phoenix/blob/873d3b9cdf43aff74e77f2df261c07ba2f19a302/lib/phoenix/router.ex (viitattu 17.11.2015).

[23] E. Meadows-Jönsson, Hex Package Manager, Erlang Factory SF Bay 2015. Saatavissa: http://www.erlang-factory.com/sfbay2015/eric-meadowsjonsson (viitattu 18.11.2015).

[24] Phoenix Framework, Guides, verkkosivu. 2015. Saatavissa: http://www.phoenixframework.org/docs/overview (viitattu 21.11.2015).

[25] Z. Rauf, Guts of Phoenix websocket channels, verkkosivu. 2015. Saatavissa: http://www.zohaib.me/guts-of-phoenix-channels/ (viitattu 8.12.2015).

[26] C. McCord, Phoenix – a framework for the modern web, NDC Oslo 2015. Saatavissa: http://www.chrismccord.com/blog/2015/06/26/ndc-oslo-2015-phoenix-a-framework-for-the-modern-web/ (viitattu 8.12.2015).

[27] J. Kain, Ranch Application Design, verkkosivu. 2015. Saatavissa: http://learningelixir.joekain.com/ranch-application-design/ (viitattu 8.12.2015).

[28] J. Valim, Ecto: A language integrated query for Elixir, Code Mesh London. 2013. Saatavissa: http://www.catchtalk.tv/events/codemesh/videos/jose-valim-ecto-a-language-integrated-query-for-elixir (viitattu 8.12.2015).

[29] J. Kain, Cowboy Application Design, verkkosivu. 2015. Saatavissa: http://learningelixir.joekain.com/cowboy-application-design/ (viitattu 8.12.2015).

[30] N. Marin et al. The Elixir Style Guide, verkkosivu. 2016. Saatavissa: https://github.com/levionessa/elixir_style_guide/blob/c1ab80fcddf1f65038b65738c4077065e13d3e5e/README.md (viitattu 11.11.2016).

[31] Plataformatec, Elixir Getting Started Guide: Pattern matching, verkkosivu. 2016. Saatavissa: http://elixir-lang.org/getting-started/pattern-matching.html (viitattu 12.11.2016).

[32] Plataformatec, Elixir Getting Started Guide: Macros, verkkosivu. 2016. Saatavissa: http://elixir-lang.org/getting-started/meta/macros.html (viitattu 12.11.2016).

[33] J. Valim, Comparing Elixir and Erlang variables. 2016. Saatavissa: http://blog.plataformatec.com.br/2016/01/comparing-elixir-and-erlang-variables/ (viitattu 13.11.2016).

[34] Erlang User's Guide: Efficiency Guide – Processes, Ericsson AB. 2016. Saatavissa: http://erlang.org/doc/efficiency_guide/processes.html (viitattu 13.11.2016).

[35] Erlang User's Guide: Getting Started with Erlang – Concurrent Programming, Ericsson AB. 2016. Saatavissa: http://erlang.org/doc/getting_started/conc_prog.html (viitattu 13.11.2016).

[36] Hex: The package manager for the Erlang ecosystem, verkkosivu. 2016. Saatavissa: https://hex.pm/ (viitattu 13.11.2016).

[37] Npm, Npm Inc., verkkosivu. 2016. Saatavissa: https://www.npmjs.com/ (viitattu 13.11.2016).

[38] Packagist Statistics, verkkosivu. 2016. Saatavissa: https://packagist.org/statistics (viitattu 13.11.2016).

[39] RubyGems.org Stats, verkkosivu. 2016. Saatavissa: https://rubygems.org/stats (viitattu 13.11.2016).

[40] PyPI – the Python Package Index, Python Software Foundation, verkkosivu. 2016. Saatavissa: https://pypi.python.org/pypi (viitattu 13.11.2016).

[41] J. Beckmann, Awesome Elixir, verkkosivu. 2016. Saatavissa: https://github.com/h4cc/awesome-elixir/tree/d45fbdd67a69ee0a272d76e1db3b65e61165b4a4 (viitattu 13.11.2016).

[42] Erlang User's Guide: Reference Manual – Dialyzer, Ericsson AB. 2016. Saatavissa: http://erlang.org/doc/man/dialyzer.html (viitattu 14.11.2016).

LIITE A: CODE::STATS-KONFIGURAATIO SUORITUSKYKYMITTAUKSIA VARTEN

```
1       use Mix.Config
2       config :code_stats, CodeStats.Endpoint,
3         http: [host: "192.168.10.59", port: {:system, "PORT"}],
4         url: [host: "192.168.10.59", port: 8080, path:
5       "/elixirimpl"],
6         cache_static_manifest: "priv/static/manifest.json",
7         secret_key_base:
        "AHAHwRHJAETJOHAegkohaworhnoawnrhAWHMnr--"
8
9       # Configure your database
10      config :code_stats, CodeStats.Repo,
11        adapter: Ecto.Adapters.Postgres,
12        username: "code-stats-test",
13        password: "1234",
14        database: "code-stats-test",
15        pool_size: 40,
16        idle_timeout: 30_000
17
18      # Email config, override in your env.secret.exs
19      config :code_stats, CodeStats.Mailer,
20        adapter: Bamboo.MailgunAdapter,
21        api_key: "blabla",
22        domain: "blabla"
23
24      config :code_stats,
25
26        # If the site is proxied, the URL helpers may end up with
        the wrong URL.
27        # This value is used as absolute URL instead. No trailing
        slash!
28        absolute_url: "http://192.168.10.59:8080/elixirimpl",
29
30        site_name: "Code::Stats",
31
32        # Address to send email from in the form of {"Name",
        "address@domain.example"}
33        email_from: {"Code::Stats", "noreply@mg.codestats.net"},
34
35        # Twitter account username to insert link in footer, nil
        to disable
36        twitter_account: "code_stats",
37
38        # Extra HTML that is injected to every page, right before
        </body>. Useful for analytics scripts.
39        analytics_code: ""
```

LIITE B: NGINX-KONFIGURAATIO SUORITUSKYKYMITTAUKSIA VARTEN

```
1       worker_processes 4;
2       worker_cpu_affinity auto;
3       worker_rlimit_nofile 4096;
4
5       events {
6         worker_connections 2048;
7       }
8
9
10      http {
11        error_log /var/log/nginx/error.log;
12        include mime.types;
13        default_type application/octet-stream;
14
15        sendfile on;
16
17        # Set tight timeouts since siege doesn't support them
18        send_timeout 8;
19
20        keepalive_timeout 65;
21
22        server {
23          listen 8080;
24          server_name localhost 192.168.2.2;
25          access_log off;
26
27          # Pass any urls beginning with /elixirimpl to Elixir
        implementation
28          # Fake Phoenix into thinking it's at root url
29          location /elixirimpl/ {
30            proxy_pass http://127.0.0.1:1337/;
31            proxy_read_timeout 5;
32            proxy_send_timeout 5;
33          }
34
35          location /live_update_socket/ {
36            proxy_pass http://127.0.0.1:1337/live_update_socket/;
37
38            # Required for websockets
39            proxy_http_version 1.1;
40            proxy_set_header Upgrade $http_upgrade;
41            proxy_set_header Connection "upgrade";
42            proxy_set_header Host "192.168.10.59";
43          }
44
45          # All PHP files are served by PHP implementation
46          location ~ \.php {
47            root /home/alarm/code-stats-comparisons/php;
```

```
48            fastcgi_pass unix:/run/php-fpm/php-fpm.sock;
49            fastcgi_read_timeout 5;
50            fastcgi_send_timeout 5;
51            fastcgi_split_path_info ^(.+\.php)(/.+)$;
52            include fastcgi.conf;
53          }
54
55          # Serve static files for static tests
56          location / {
57            root /home/alarm/statictest;
58          }
59        }
60      }
```