jamk.fi

Verkkosovellus Huginn-instanssien hallinnoinnille

Veeti Karttunen

Opinnäytetyö
Syyskuu 2018
Tekniikan ja liikenteen ala
Insinööri (AMK), Ohjelmistotekniikan tutkinto-ohjelma

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Karttunen, Veeti	Opinnäytetyö, AMK	maaliskuu 2018
	Sivumäärä 62	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty: x

Työn nimi
Verkkosovellus Huginn-instanssien hallinnoinnille

Tutkinto-ohjelma
Tieto- ja viestintätekniikka

Työn ohjaaja(t)
Rintamäki Marko, Huotari Jouni

Toimeksiantaja(t)
Sparta Consulting Oy

Tiivistelmä

Opinnäytetyön tavoitteena oli automatisoida Huginn-ohjelmiston käyttöönottoprosessi. Tämän käyttöönottoprosessin automatisointi päätettiin toteuttaa verkkosovelluksena. Verkkosovelluksen tarkoituksena oli mahdollistaa Huginn-instanssin luominen, poistaminen, sammuttaminen ja käynnistäminen automatisoidusti Azuren pilvipalveluun.

Sovelluksen toteutus tehtiin projektina Sparta Consulting Oy:lle. Projektissa hyödynnettiin Sparta Consulting Oy:n resursseja, kuten versionhallintaa ja Azuren pilvipalvelua. Toteutuksessa keskeisiä komponentteja olivat Elixir ohjelmointikieli, Phoenix Framework, Terraform ja Azuren pilvipalvelu.

Tuloksena saatiin toiminnallinen verkkosovellus, jonka avulla voitiin luoda ja poistaa Huginn-instansseja automatisoidusti Azuren pilvipalvelussa. Huginn-instansseja pystyi myös sammuttamaan ja uudelleenkäynnistämään. Verkkosovellus oli rajoitettu toimimaan ainoastaan Sparta Consulting Oy:n omassa sisäverkossa.

Lopputulos oli toiminnallinen ja Sparta Consulting Oy:n mukaan tarpeen täyttävä. Seuraavia kehitysaskeleita olisi lisätä automatisointia, kuten automaattinen Huginn-instanssin sammuttaminen tietyllä ajanhetkellä. Jos sovellukseen halutaan päästä käsiksi muualtakin kuin vain Sparta Consulting Oy:n omasta sisäverkosta, tulisi sovellukseen lisätä myös autentikaatio.

Avainsanat (asiasanat)
Verkkosovellus, Elixir, Terraform, Azure, Käyttöönotto

Muut tiedot (salassa pidettävät liitteet)

jamk.fi

Description

Author(s)	Type of publication	Date
Karttunen, Veeti	Bachelor's thesis	march 2018
		Language of publication: Finnish
	Number of pages: 62	Permission for web publication: x

Title of publication
Web application for management of Huginn instances

Degree programme
Software Engineering

Supervisor(s)
Rintamäki Marko, Huotari Jouni

Assigned by
Sparta Consulting Oy

Abstract

The project was assigned by Sparta Consulting Oy with the goal to create an automated deployment process for Huginn application. It was decided to implement the automation of the deployment process as a web application, the purpose of which was to create, delete, turn on and turn off the Huginn instance in Azure cloud service.

The application was implemented as a project to Sparta Consulting Oy. Sparta Consulting Oy offered for this project their resources, such as version control and Azure cloud service. The key components for the application were Elixir, Phoenix Framework, Terraform, and Azure.

The result was a functional web application that was able to create and delete Huginn instances through an automated process in Azure cloud service. The application was also able to turn on and turn off the Huginn instances. The web application was limited for use in the internal network of Sparta Consulting Oy only.

The result was functional and met the requirements set by Sparta Consulting Oy. The next development steps would be to add more automation; for example, turning off the Huginn instance at a certain date. If access to the application from public network is needed, authentication will be necessary.

Keywords/tags (subjects)
Web application, Elixir, Terraform, Azure, Deployment

Miscellaneous (Confidential information)

Sisältö

Kuviot

Taulukot

1 Työn lähtökohdat

1.1 Johdanto

Työn tarkoituksena oli tuottaa verkkopalvelu, joka tarjoaa automatisoidun prosessin Huginn järjestelmän tekniselle käyttöönotolle Azuren pilvipalveluun. Järjestelmien tekninen käyttöönotto voi olla hyvin työläs prosessi, jos järjestelmä koostuu useasta eri riippuvuudesta kuten esimerkiksi tietokannasta, verkkosovelluksesta ja kolmannen osapuolen ohjelmasta. Usein järjestelmä käyttää myös ympäristömuuttujia, jotka tulee asettaa järjestelmää ajavaan ympäristöön. Nämä kaikki osa-alueet teknisessä käyttöönotossa ovat itsessään pieniä tehtäviä, mutta yhdessä niistä muodostuu usein suuri kokonaisuus, josta on helppo unohtaa vaiheita.

Automatisoitu prosessi, jonka tehtävänä on ottaa vastaan tarvittavat tiedot järjestelmän teknistä käyttöönottoa varten, helpottaa käyttöönottoa huomattavasti. Sen sijaan, että henkilö, jonka tehtävänä on toteuttaa järjestelmän tekninen käyttöönotto, joutuisi manuaalisesti käymään jokaisen eri kohdan läpi, täytyy hänen vain täyttää lomake, joka käsitellään ohjelmallisesti, ja tekninen käyttöönotto on täten tehty.

1.2 Huginn

Huginn on Sparta Consulting Oy:n tuottama järjestelmä, joka valvoo yrityksen liiketoimintakriittisen datan integriteettiä. Huginn on verkkosovellus, josta ajetaan aina yhtä erillistä instanssia asiakasta kohden. Aikaisemmin nämä instanssit toimivat asiakkaan omalla palvelimella, mutta on sittemmin päätetty siirtää pilvipalveluun ajettavaksi. Pilvipalvelun käyttö mahdollistaa yhtenäisen ympäristön Huginnille, mikä vähentää epävarmuustekijöitä palvelun käyttöönotossa.

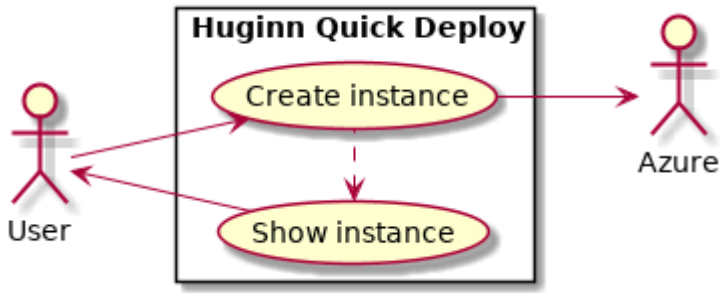
Huginn perustuu sääntöjen määrittämiseen, joiden toteutumista valvotaan yrityksen järjestelmissä. Valvonta toteutetaan lähettämällä yrityksen data eräajona Huginnille, jonka yhteydessä data tarkastetaan sääntöjen avulla. Sääntö on koodia, joka tarkkailee Huginnille lähetettyä dataa. Sääntö voi tarkkailla muun muassa datassa olevan

jonkun tietyn sarakkeen arvoa tai jonkun arvon muuttumista tietyn ajan kuluessa. Jos Huginnille lähetetty data ei läpäise jonkin säännön vaatimuksia, Huginn luo tästä havainnon. Havainto on tapahtuma Huginnissa, joka sisältää havaintoviestin. Havainnon luoma viesti määritetään säännössä, joka on luonut havainnon. Havainnon avulla yritys voi tarkastaa datan, josta Huginn on luonut havainnon, ja vahvistaa, että kyseessä on esimerkiksi vain näppäilyvirhe. Havainnossa ilmenevä datan poikkeama voi kuitenkin olla myös haitallista toimintaa, kuten tietomanipulaation yritys. Tämänkaltaisten virheiden tapahtuminen yrityksen järjestelmissä voi jäädä jopa kokonaan huomaamatta, tai sen tunnistaminen voi kestää useita kuukausia, mutta Huginnin avulla virhe voidaan tunnistaa minuuteissa.

1.3 Käyttötapaukset

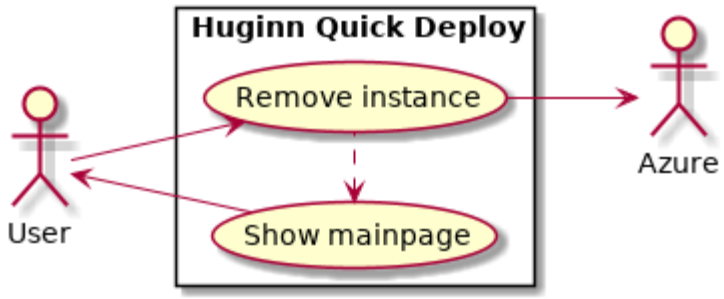
Ohjelmiston suunnittelu on tärkeä osa onnistuneen ohjelmiston luomisprosessia, joten suunnittelun helpottamiseksi on olemassa useita menetelmiä. Yksi yleinen suunnittelua avustava menetelmä on luoda ohjelmistosta käyttötapauksia. Käyttötapaukset auttavat ymmärtämään ohjelmiston eri toimintoja, joka auttaa luomaan käyttötapauksista vaatimuksia. Huginn Quick Deployn suunnittelussa luotiin yleisiä käyttötapauksia.

Instanssin luominen on Huginn Quick Deployn yksi tärkeimmistä käyttötapauksista. Instanssin luominen tapahtuu sekä Huginn Quick Deployssa että Azuressa, sillä instanssi tulee pyörimään Azuren pilvipalvelussa kun se on luotu. Kun käyttäjä on pyytänyt Huginn Quick Deployta luomaan instanssin, Huginn Quick Deploy näyttää luodun instanssin tiedot käyttäjälle. (ks. kuvio 1).



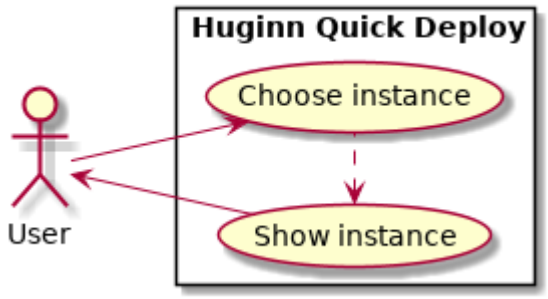
Kuvio 1. Käyttötapaus instanssin luonnista

Instanssin poistaminen on toinen keskeinen käyttötapaus. Kun käyttäjä haluaa poistaa instanssin, poisto toteutetaan sekä Huginn Quick Deployssa että Azuressa. Kun poistamistoiminto on tehty, Huginn Quick Deploy näyttää ohjelmiston päänäkymän. (ks. kuvio 2).



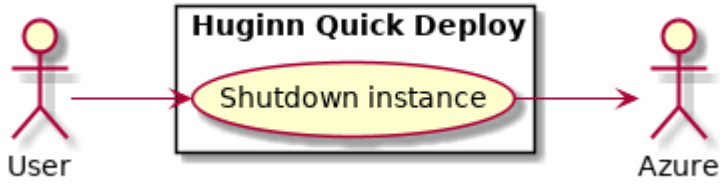
Kuvio 2. Käyttötapaus instanssin poistamisesta

Jotta Huginn Quick Deploy olisi käytettävissä käyttäjän täytyy voida selata instanssien välillä. Instanssien selaamista varten luotiin myös käyttötapaus. Kun käyttäjä selaa instansseja ja valitsee niistä jonkun, Huginn Quick Deploy näyttää käyttäjälle kyseisen instanssin. (ks. kuvio 3).

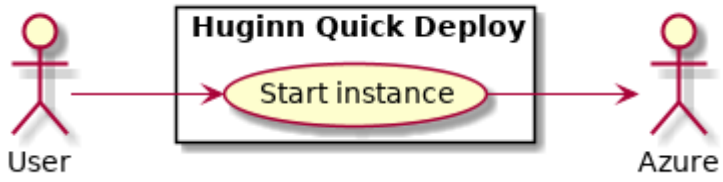


Kuvio 3. Käyttötapaus instanssien selaamisesta

Huginn Quick Deployhyn haluttiin toiminto, jolla oli mahdollista sammuttaa ja käynnistää Huginn-instansseja, joten sitä varten luotiin myös käyttötapauksia. Instanssin sammuttamisen käyttötapaus on hyvin yksinkertainen. Kun käyttäjä haluaa sammuttaa instanssin, instanssin sammutuksen pyyntö viedään eteenpäin Azurelle, joka sammuttaa virtuaalikoneen, jossa Huginnia ajetaan. (ks. kuvio 4). Samankaltainen käyttötapaus luotiin myös instanssin käynnistämisestä (ks. kuvio 5).



Kuvio 4. Käyttötapaus instanssin sammuttamisesta



Kuvio 5. Käyttötapaus instanssin käynnistämisestä

1.4 Vaatimukset

Kun aloitetaan ohjelmiston suunnittelu, tulee kartoittaa toimeksiantajan vaatimukset. Vaatimus on käsite, joka kuvaa jotain, mitä ohjelma voi tehdä (Haikala; Mikkonen, 2011, 61). Vaatimuksia on yleisesti kolmenlaisia: toiminnallisia, ei-toiminnallisia ja reunaehtoja. Tässä työssä ei asetettu yhtäkään reunaehtoa, mutta toiminnallisia ja ei-toiminnallisia vaatimuksia asetettiin kylläkin (ks. taulukko 1-2). Vaatimukset kartoitettiin yhdessä toimeksiantajan kanssa palavereissa sekä niitä muodostettiin käyttötapauksista.

Toiminnalliset vaatimukset (ks. taulukko 1) ovat vaatimuksia, jotka vaikuttavat ohjelmiston toimintaan. Tässä työssä toiminnalliset vaatimukset keskittyivät pääasiassa käyttöliittymän piirteisiin. Lomake, jolla voidaan luoda uusi Huginn-instanssi, on vaatimus, jonka mukaan ohjelmiston tulee sisältää sähköinen lomake, joka aloittaa automatisoidun luomisprosessin Azuren pilvipalveluun. Lista Huginn-instansseista on hyvin yksinkertaisesti näkymä, jonka avulla voidaan nähdä kaikki instanssit, jotka ovat olemassa. Näkymä Huginn-instanssin tarkemmista tiedoista on oma näkymä, jossa voidaan tarkastella yksittäistä instanssia tarkemmin ja saada tietoa kyseisen instanssin tilasta. Huginn-instanssi tulee voida sammuttaa, jotta asiakkaalta voidaan evätä pääsy kyseiseen Huginn-instanssiin, mutta instanssin tiedot ovat vielä tallessa ja se voidaan tarvittaessa käynnistää uudelleen. Instanssi tulee voida poistaa, eli kun instanssia ei enää tarvita, voidaan instanssi poistaa eikä siitä jää minkäänlaisia yksityistietoja talteen.

Ei-toiminnalliset vaatimukset (ks. taulukko 2) eivät vaikuta ohjelmiston toimintoihin, vaan kuvastavat laadullista näkökulmaa ohjelmistossa. Verkkosovellus on käytettävissä yrityksen omassa sisäverkossa, eli työssä valmistuva sovellus tulee käyttöön vain yrityksen omaan sisäverkkoon. Ohjelmisto käyttää Azuren pilvipalvelua, koska toimeksiantajalla on Azuren pilvipalvelu jo omassa käytössään.

Taulukko 1. Toiminnalliset vaatimukset

Toiminnalliset vaatimukset
Huginn-ohjelmisto tulee asentaa automaattisesti instanssin luonnin yhteydessä
Instanssin tiedot tulee voida yksilöidä
Instanssin IP-osoite tulee hakea automaattisesti instanssia luodessa
Instanssille kriittiset salasanat tulee generoida luonnin yhteydessä
Lista Huginn-instansseista
Näkymä Huginn-instanssin tarkemmista tiedoista
Huginn-instanssi tulee voida sammuttaa ja käynnistää
Instanssi tulee voida poistaa

Taulukko 2. Ei-toiminnalliset vaatimukset

Ei-toiminnalliset vaatimukset
Ohjelmisto on verkkosovellus
Verkkosovellus on käytettävissä vain yrityksen omassa sisäverkossa
Huginn-instannseja ajetaan Azuren pilvipalvelussa
Generoituja salasanoja ei tule säilöä
Poistetusta instanssista ei saa jäädä jälkiä

2 Ohjelmiston komponentit

2.1 Yleistä

Kun luodaan uutta ohjelmistoa, se koostuu usein erilaisista komponenteista. Näitä komponentteja ovat muun muassa ohjelmointikieli, tietokanta ja erilaiset kirjastot. Komponenttien tarkoituksena on tarjota toiminnallisuutta, jota ei tarvitse tehdä uudelleen ja joka on usein hyvin keskittynyttä johonkin tiettyyn osa-alueeseen. Tällä tavoin ohjelmiston luomisessa voidaan keskittyä päätoiminnallisuuksiin.

2.2 Elixir

2.2.1 Yleistä

Aina kun aloitetaan uuden ohjelmiston kehittäminen, tulee eteen ohjelmointikielen valinta. Ohjelmointikieli kannattaa valita usein käyttötarkoituksen mukaan. Tällä tavoin voi säästää aikaa työltä, joka voi toisen kaltaisella kielellä viedä paljon ylimääräistä aikaa. Ohjelmointikieli voidaan valita myös muilla perusteilla, kuten aikaisemman kokemuksen perusteella. Jos kielellä on toteutettu aiemmin samankaltainen projekti, se voi olla kannattava valinta myös nykyiseen projektiin. Kielen valinnalla on myös suuri vaikutus kehitystyössä, sillä kieli, joka ei sovellu käyttötarkoitukseen tai on muuten vain ikävä käyttää, voi heikentää kehitystyön tahtia ja tuloksia. Kun valitaan ohjelmointikieli, tulee ottaa huomioon kielen syntaksi, ominaisuudet ja käyttökohteet. (Christensen 2015.)

Elixir on funktionaalinen ohjelmointikieli, joka on luotu Erlang Virtual Machinen päälle. Sen vahvuuksia on usean prosessoriytimen käyttö ja Erlang ohjelmointikieleen verrattuna helpommin ymmärrettävissä oleva syntaksi. (Elixir n.d.)

Elixir valittiin Huginn Quick Deployment projektiin, koska toimeksiantaja käyttää Elixiriä pääasiallisena ohjelmointikielenä muissakin projekteissa. Elixir tarjoaa myös hy-

vän ohjelmistokehyksen verkkosovellusten kehittämiseen, nimeltä Phoenix Framework. Sillä Elixir on jo käytössä toimeksiantajalla ja se sopii käyttötarkoitukseen, ei ollut mitään syytä etsiä toista kieltä projektia varten.

2.2.2 Rakenne

Elixir syntaksi on saanut vaikutteita Ruby ohjelmointikielestä, sillä se sisältää muun muassa Rubysta peräisin olevan def end rakenteen (Official Ruby FAQ). Elixir syntaksi eroaa jossain määrin yleisistä ohjelmointikielistä, sillä siinä ei juurikaan käytetä if-else rakennetta, eikä Elixir sisällä silmukointia mahdollistavia funktioita. Silmukointi toteutetaan sen sijaan rekursiolla. (Recursion n.d.)

Ollessaan funktionaalinen kieli, Elixir syntaksi perustuu funktioiden kirjoittamiseen. Funktio on kokoelma operaatioita, jonka tarkoitus on suorittaa jokin tietty tehtävä. Funktio ottaa vastaan dataa, suorittaa annetulle datalle määrätyt operaatiot ja tulostaa lopuksi lopputuloksen. Funktioiden vahva puoli on se, että niistä ei muodostu sivuvaikutuksia, kun taas olio ohjelmoinnin metodeista voi muodostua suuriakin sivuvaikutuksia. (Functional programming – introduction n.d.)

Elixir keskittyy voimakkaasti yhdenaikaisuuteen, joka mahdollistaa tehokkaan tavan hyödyntää ohjelmaa ajavan tietokoneen laskentatehoa. Tämä yhdenaikaisuus on toteutettu jakamalla Elixir ohjelma useaan prosessiin. Nämä prosessit ovat eristetty toisistaan ja keskustelevat keskenään hyödyntämällä prosessien välisiä viestejä. (Elixir n.d.)

2.2.3 Mix

Elixirin kehityksessä on otettu huomioon myös helppo aloittaminen sekä kielen oppimisessa, että projektin aloittamisessa, sillä vaikea alkuun pääseminen voi laskea käyttömukavuutta. Elixirin kehittäjä José Valim pitää tärkeänä tarjota hyvät työkalut ohjelmointikielen ohelle, jonka takia Elixirin mukana tulee Mix niminen kehitystyökalu. (Schuster 2014.)

Mix on työkalu, joka tarjoaa toiminnallisuuden ohjelman luomiseen, kääntämiseen, rakentamiseen, ajamiseen ja testaamiseen. Mix toiminnot ovat niin sanottuja taskeja, joita ajetaan komentolinjalta. Mix helpottaa ohjelmien kehittämistä huomattavasti, sillä minimissään kehittämistä varten ei tarvitse pystyttää minkäänlaista kehitysympäristöä. (Introduction to Mix n.d.)

Mixillä voidaan luoda uusi mix projekti käyttämällä *mix new* komentoa. Komennolle tulee antaa argumentiksi projektin nimi ja sille voidaan myös antaa muita asetuksia. Mix luo kansion, jonka nimeksi asetetaan projektin nimi. Tähän kansioon luodaan tiedostoja, jotka luovat hyvän alustan aloittaa kehittämään uutta ohjelmaa. (Introduction to Mix n.d.)

Mixin avulla voidaan käynnistää luotu elixir projekti käyttämällä komentoa *iex -S mix*. Tämä komento käynnistää projektin elixirin interaktiivisessa konsolissa ja on täten helposti käytettävissä. Kun projektiin tehdään muutoksia, ne voidaan sisällyttää käynnissä olevaan konsoliin ajamalla komento *recompile*. Tämä komento kääntää tiedostot, jotka ovat muuttuneet projektin ollessa käynnissä. (Introduction to Mix n.d.)

Mix sisältää mahdollisuuden ajaa ohjelma erilaisissa ympäristöissä, joita on tavallisesti kolme: dev, test ja prod. Dev on ympäristö, joka on tarkoitettu käytettäväksi ohjelman kehitysvaiheessa, kun taas test on tarkoitettu ohjelman testaukseen, ja viimeisenä prod joka on tarkoitettu tuotantoon otettavan ohjelman ympäristöksi. Nämä ympäristöt vaikuttavat ohjelman konfiguraatioon, jolloin esimerkiksi ohjelmaa kehittäessä ei tarvitse käyttää HTTPS protokollaa, mutta tuotantokäytössä voidaan vaatia kaikkien evästeiden tulevan vain HTTPS protokollan kautta. (Introduction to Mix n.d.)

Mix kehitystyökalua hyödynnettiin myöhemmin ohjelmiston kehitysvaiheessa sen käyttötarkoituksiin. Tämä sisälsi muun muassa lähdekoodin kääntämisen sekä kehitysympäristön ajamisen.

2.2.4 GenServer

GenServer on Elixir behaviour, joka tuo palvelimen kaltaisia toimintoja Elixiriin. GenServer tarjoaa Elixiriin helpon keinon luoda uusia prosesseja, jotka ajavat ohjelman pääprosessin kanssa samanaikaisesti. GenServer on helposti ohjelmoitavissa kehittäjän tarpeiden mukaisesti. (GenServer – Elixir n.d.)

GenServer käyttää kutsuja, joilla voidaan pyytää tietynlaista toiminnallisuutta GenServeriltä. Kutsuja on kahdenlaisia, synkronisia ja asynkronisia. Synkroniset kutsut voivat saada GenServeriltä vastauksen, kun taas asynkroniset kutsut eivät koskaan saa GenServeriltä vastausta. Vaikka synkroninen kutsu voi saada vastauksen GenServeriltä, GenServerin ei ole pakko vastata kutsuun. (GenServer – Elixir n.d.)

GenServerillä voidaan muun muassa säilöä tilaa, jota hyödynnetään koko ohjelmassa, tai sillä voidaan ajaa funktioita, joiden suorittamiseen menee huomattava määrä aikaa. Sitä ei tule kuitenkaan käyttää turhan takia, esimerkiksi koodin organisointiin. GenServeriä on hyvä käyttää silloin kun toiminnallisuus vaatii oman prosessin, mutta jos tilanne ei sitä vaadi on täysin turhaa monimutkaistaa ohjelmaa lisäämällä siihen tarpeettomia palasia. (GenServer – Elixir n.d.)

Työn toteutuksessa GenServerillä toteutettiin Terraform-ohjelman ja Azure-pilvipalvelun välinen keskustelu. Koska Terraform-ohjelman luomis- ja poistamiskutsujen ajamiseen kului noin 10-20 minuuttia, GenServerillä oli mahdollista ohjata kutsu toiseen Erlang prosessiin. GenServeriä hyödynnettiin myös Azuren autentikaatiotokenin säilömiseen.

2.2.5 Phoenix Framework

Phoenix Framework on Elixir ohjelmistokehys, joka tarjoaa toiminnallisuuksia verkkosovelluksen luomiseksi. Phoenix on toteutettu toimimaan MVC-arkkitehtuurin mukaisesti, ja muistuttaa rakenteeltaan Ruby on Rails ja Django Framework ohjelmistokehyksiä. (Overview – Phoenix n.d.)

Phoenix koostuu seitsemästä pääosiosta, jotka ovat: Endpoint, Router, Controller, View, Template, Channel ja PubSub. Näistä keskeisimpiä ohjelman toimivuuden kannalta ovat Endpoint, Router, Controller, View ja Template. Channel ja PubSub keskittyvät pääasiassa reaaliaikaisen kommunikaation luontiin, joka ei ole vaatimus ohjelman toiminnalle. (Overview – Phoenix n.d.)

Endpoint kuvastaa HTTP pyynnön elinkaaren päätepisteitä. Jokainen HTTP pyyntö lähetetään tai vastaanotetaan Endpointissa. Endpointin tarkoitus on käsitellä HTTP pyynnöt ennen kuin ne lähetetään routerille. (Endpoint – Phoenix n.d.)

Router on nimensä veroisesti reititin, mutta kyseessä oleva reititin ohjaa HTTP pyyntöjä oikealle controllerille. Router sisältää polkuja, joiden avulla http pyynnöt osataan ohjata oikealle controllerin actionille. Nämä polut ovat kehittäjän määritettävissä ja niille voidaan antaa muuttujia, jotta polkuihin voidaan lisätä dynaamisuutta. (Routing – Phoenix n.d.)

Controller tarjoaa funktioita ohjelmalle, jotka käsittelevät routerilta vastaanotettuja pyyntöjä. Controllerin tilanteessa näitä funktioita kutsutaan actioneiksi, vaikkakin kyseessä on vain normaali elixir funktio. Actionit valmistelevat tietoa viewejä varten, pyytää renderöintiä sekä uudelleenohjaa pyyntöjä. (Controllers – Phoenix n.d.)

View renderöi templateja controllerin pyyntöjen mukaan. Viewiin voidaan myös luoda uusia funktioita, joita on helppo kutsua templatessa. Nämä funktiot ovat samankaltaisia decoratoreiden ja facade kuvion kanssa. (View – Phoenix n.d.)

Template on sapluuna, johon asetetaan dataa. Verkkosovelluksessa template on tyypillisesti HTML:ää sisältävä tiedosto. Vaikka template on usein pääasiassa HTML tiedosto, voidaan templateen myös sisällyttää elixir koodia Embedded Elixir teknologialla. (Templates – Phoenix n.d.)

Huginn Quick Deployn käyttöliittymä, backend ja tietokanta toteutettiin Phoenix Frameworkilla. Phoenix Framework oli Huginn Quick Deployssa tärkein yksittäinen Elixir kirjasto.

2.2.6 Ecto

Ecto on Elixir kirjasto, jonka avulla voidaan ottaa yhteys tietokantaan ja käsitellä tietokannassa olevaa dataa. Ecto sisältää adaptereja, joilla voidaan lisätä tuki eri tietokantamoottoreille kuten esimerkiksi Postgresql. (Ecto – Ecto n.d.)

Ecto on jaettu neljään eri komponenttiin, jotka ovat: Repo, Schema, Changeset ja Query. Nämä kaikki eri komponentit luovat kokonaisuuden, joka tarjoaa helpon ohjelmallisen lähestymistavan tietokannan informaation käsittelyyn. (Ecto – Ecto n.d.)

Repo on moduuli, joka keskustelee tietokannan kanssa. Sen avulla voidaan luoda, päivittää ja poistaa rivejä tietokannan taulusta, mutta Repo tarjoaa myös mahdollisuuden tehdä tietokantakyselyjä SQL-kielellä. Jotta tietokantaan voidaan ottaa yhteys, Repolle annetaan tietokantayhteys konfiguraatio, joka sisältää tietokannan, käyttäjätiedot ja osoitteen. (Ecto.Repo – Ecto n.d.)

Schema, eli kaavio ottaa dataa tietolähteestä ja asettaa ne Elixir structiin. Schema voidaan määritellä kahden APIn avulla, schema ja embedded_schema. Schema APIa käytetään tavallisesti pysyvän tietolähteen kanssa, kuten tietokantataulu. Embedded_schema APIa käytetään, kun halutaan luoda schema jonkin toisen scheman sisään tai jos halutaan säilöä schema vain sisäisessä muistissa. (Ecto.Schema – Ecto n.d.)

Changeset on keino validoida ja seurata muutoksia dataan ennen kuin ne asetetaan tietokantatauluun (Ecto - Ecto). Changesetin avulla voidaan myös tehdä muutoksia dataan, jolloin ei tarvitse tehdä suoria pyyntöjä tietokantaan Repon kautta. Nämä keinot tulevat funktioista nimeltä cast ja change. Cast on tarkoitettu käytettäväksi, kun changeset saa dataa ulkopuoliselta lähteeltä ja change on tarkoitettu käytettäväksi ohjelman sisällä, kun changeset saa dataa ohjelman sisäisesti. (Ecto.Changeset – Ecto n.d.)

Query tuo Ectoon formaatin, jonka avulla voidaan luoda ohjelmallisia tietokantakyselyjä. Sitä käytetään datan hakemiseen ja sen manipulointiin. Query perustuu pääosin avainsanapohjaiseen rakenteeseen. Näiden avainsanojen avulla voidaan hakea ja suodattaa tietokannasta dataa. (Ecto.Query – Ecto n.d.)

Ectoa käytettiin Huginn Quick Deployssa instanssien tietueiden luontiin, päivittämiseen sekä poistamiseen. Ecton hyödyntäminen kehitysvaiheessa vähensi huomattavan määrän työtä, joka olisi vaadittu, jos kehityksessä ei olisi hyödynnetty valmista kirjastoa tietokannan kanssa keskusteluun.

2.2.7 HTTPoison

HTTPoison on HTTP client Elixirille. Sillä voidaan lähettää dataa verkkoon ja vastaanottaa dataa verkosta. HTTPoisonia voidaan käyttää esimerkiksi APIen käyttöön. HTTPoison tukee sekä GET että POST pyyntöjä, joiden avulla voidaan yleisesti käyttää eri sivustojen APEja. HTTPoison palauttaa verkosta saadun vastauksen Elixir structina. (HTTPoison n.d.)

HTTPoisonilla lähetettiin kaikki kutsut, jotka oli luotu manuaalisesti, Azurelle. HTTPoisonin avulla luotiin muun muassa ominaisuudet, joilla oli mahdollista sammuttaa ja käynnistää Huginn-instansseja.

2.2.8 Embedded Elixir

Embedded Elixir on Elixirin osio, joka mahdollistaa Elixir koodin sulauttamisen tekstiin tai tiedostoon. Embedded Elixir sisältää kolme API:a joiden avulla Embedded Elixir toimii ja ne ovat: eval_string tai eval_file, function_from_string tai function_from_file, ja compile_string tai compile_file. Eval_string ajaa tekstiin sulautetun elixir koodin ja vaikuttaa täten tekstin sisältöön, myös eval_file toimii samalla periaatteella. Function_from_string toimii käytännössä samalla tavalla kuin eval_string, mutta tässä tilanteessa tekstin sisältämä Elixir koodi ajetaan, kun ohjelma käänne-

tään. Compile_string kääntää kyseessä olevan tekstin, jotta Elixir voi lukea tekstin sisältämän koodin. Sekä eval_string, että function_from_string käyttävät tätä funktiota. (EEx n.d.)

Embedded Elixiriä hyödynnettiin Terraformin konfiguraatiotiedostojen luontiin. Konfiguraatiotiedostojen oli äärimmäisen tärkeää olla ohjelmallisesti luotuja, sillä konfiguraatiotiedostossa asetettiin muun muassa Huginn-instanssille asetettavat salasanat.

2.3 PostgreSQL

PostgreSQL on relaatiotietokantamoottori, joka perustuu vuonna 1986 kehitettyyn POSTGRES järjestelmään. Se on tehokas avoimen lähdekoodin järjestelmä, joka pyrkii tuomaan SQL standardin mukaisen toiminnallisuuden käyttäjille. PostgreSQL tukee SQL:ää lähes täysin, vaikkakin jotkut osiot ovat toteutettu eri tavalla mitä SQL standardissa.(PostgreSQL: About n.d.)

PostgreSQL valittiin Huginn Quick Deployn tietokantamoottoriksi. Päätös tehtiin pääasiassa aikaisemman kokemuksen perusteella.

2.4 Azure

Azure on Microsoftin tuottama pilvipalvelukokonaisuus, joka tarjoaa useita erilaisia palveluita liiketoimintatarkoituksiin. Azurea voidaan käyttää verkkosovelluksella tai API rajapinnan avulla.

Azure pitää sisällään useita erilaisia tunnuksia, joilla voidaan rajata kyseisen identiteetin käyttöoikeuksia. Nämä tunnukset koostuvat subscription tunnuksesta, client tunnuksesta, client sala-avaimesta ja tenant tunnuksesta. Kriittisimmät tunnukset ovat client tunnus, ja client sala-avain, sillä nämä kaksi antavat pääsyn Azuren API rajapintaan. Subsciption tunnus määrittää, että mitä laskutusta kyseinen identiteetti käyttää. Viimeisenä on tenant tunnus, joka määrittää identiteetin hakemiston. Hakemisto sisältää kaikki kyseiseen hakemistoon pääsevät käyttäjät, sekä kaikki kyseiseen

hakemistoon asetetut laskutussuunnitelmat. Toisin sanoen hakemisto on alhaisin tunniste Azuressa. (Create identity for Azure app in portal 2018.)

Azure tarjoaa erilaisia palveluita, joilla kaikilla on oma tarkoituksensa. Azureen voidaan luoda esimerkiksi virtuaalikoneita, joita voidaan hyödyntää muun muassa palvelimena. Virtuaalikone on kuitenkin vain murto-osa Azuren tarjoamista palveluista, sillä se sisältää muun muassa tietokantoja, analytiikkaa, mobiilipalveluja, verkkopalveluja ynnä muuta vastaavaa. (What is Azure? n.d.)

Azure sisältää käyttäjien lisäksi usein myös sovelluksia, jotka tarvitsevat oikeuksia Azuren palveluihin. Tätä varten Azuressa on identiteetit, jotka ovat samankaltaisia käyttäjien kanssa, mutta identiteetillä ei voi kirjautua Azuren käyttöliittymään. (Create identity for Azure app in portal 2018.)

Azure oli Huginn Quick Deployn yksi pääkomponenteista. Huginn-instanssit luotiin Azuren pilvipalveluun, josta ne olivat saatavilla.

2.5 Terraform

Terraform on sovellus, jonka tarkoitus on automatisoida pilvipalvelun infrastruktuurin asentaminen. Terraform on sovelluksena helppokäyttöinen, sillä se vaatii toimiakseen vain yhden tiedoston, josta voidaan lukea kaikki tarvittavat toimenpiteet.

Terraform perustuu konfiguraatiotiedoston lukemiseen. Tämä konfiguraatiotiedosto kuvaa kaikki resurssit, jotka tulee asentaa pilvipalveluun. Kuvaaminen tapahtuu koodin paloilla, joissa on kerrottu, mikä resurssi on kyseessä, sekä kyseiseen resurssiin vaadittavat tiedot. Konfiguraatiotiedostoon voidaan asettaa myös dynaamisia muuttujia, joiden arvo asetetaan, kun Terraform käynnistetään. Nämä arvot voidaan asettaa joko ympäristömuuttujan, tiedoston tai komentolinjan kautta. (Input Variables n.d.)

Koska Terraform toimii yhdessä tietyssä kansiossa, ja infrastruktuurien tila on tallennettuna terraformiin, on tärkeää voida vaihtaa ohjelman ympäristöä jollain keinolla. Siksi Terraform sisältää käsitteen workspace, jonka avulla voidaan vaihtaa ympäristöä

ja luoda lisää infrastruktuureita. Kun Terraformissa vaihdetaan workspacea, se luo uuden eristetyn ympäristön muokkaamatta kansiorakennetta. (Workspaces n.d.)

Terraformin neljä yleisintä komentoa ovat init, plan, apply ja destroy. Näillä komennoilla voidaan initialisoida, suunnitella, luoda ja poistaa infrastruktuuri. Init-komennon tarkoitus on alustaa paikallisia asetuksia sekä dataa, jota käytetään muiden komentojen ohella. Plan komento luo luetusta konfiguraatiotiedostosta suunnitelman. Tämä suunnitelma näyttää, että mitä apply-komento tekisi kyseisellä konfiguraatiolla. Apply-komento on komento, jolla konfiguraatiotiedoston sisältö käsitellään ja toteutetaan määriteltyyn pilvipalveluun API-kutsujen avulla. Apply komento varastoi myös kyseisen infrastruktuurin tilan terraformiin. Destroy-komento taas poistaa kyseisen infrastruktuurin määritellystä pilvipalvelusta. (Terraform Commands n.d.)

Terraformia käytettiin vähentämään Huginn Quick Deployn ja Azuren välisiä suoria kutsuja. Jos Huginn-instanssien luominen olisi tehty täysin Azuren API-rajapintaa vasten, olisi vaaditun työn määrä ollut huomattavasti suurempi. Terraformin avulla oli mahdollista jättää suurin osa API-rajapinnan kutsuista pois.

2.6 Kehitysympäristö

Kehitysympäristö koostuu useista eri osa-alueista. Tähän kuuluvat käyttöjärjestelmä, jossa ohjelmistoa kehitetään, tekstinkäsittelyohjelma, jolla itse ohjelmisto kirjoitetaan, sekä versionhallintakokonaisuus, jonka avulla tarkkaillaan ohjelmiston muutoksia sekä säilötään lähdekoodia.

Ohjelmiston kehityksessä käytettiin Ubuntua. Ubuntu-koneeseen asennettiin kaikki vaadittavat ohjelmat ohjelmiston kehitystä varten. Nämä ohjelmat ovat Erlang, Elixir, Visual Studio Code, Terraform, Phoenix Framework, Git ja PostgreSQL.

Erlangin ja Elixirin asennuksessa käytettiin hyödyksi ohjelmaa nimeltä asdf. Asdf on versionhallintaohjelma, joka on tarkoitettu erityisesti ohjelmointikielten versionhallintaan.

Versionhallintaohjelmana toimii Git, jonka tarkoitus on tarkkailla lähdekoodissa tapahtuvia muutoksia. Nämä muutokset voidaan lähettää Bitbucket-palveluun, joka on versionhallintapalvelu. Bitbucket sisältää git repositorion sekä muita versionhallintaan liittyviä työkaluja.

3 Työn toteutus

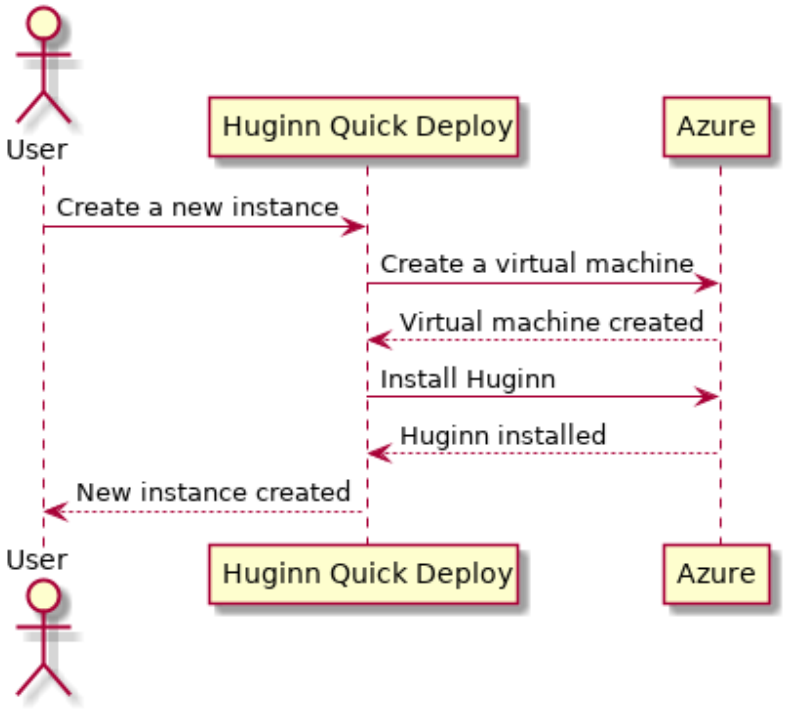
3.1 Ohjelmiston suunnittelu

Kun ohjelmistoa aletaan suunnitella, tulee käydä läpi ohjelmiston vaatimukset ja johtaa näistä vaatimuksista sitten ominaisuuksia. Näiden ominaisuuksien tekninen toteutus ei ole suunnitteluvaiheessa vielä kovinkaan tärkeää, mutta teknologioita valitessa tulee kuitenkin varmistaa, että nämä ominaisuudet on mahdollista toteuttaa kyseisillä teknologioilla.

Kun ohjelmiston suunnittelu aloitettiin, ensimmäisenä aiheesta keskusteltiin toimeksiantajan kanssa palaverissa ja selvennettiin, mitä ollaan tekemässä. On tärkeää voida keskustella toimeksiantajan kanssa projektista, sillä se voi auttaa ymmärtämään paljon projektista. Ohjelmistosuunnittelun tärkein yksittäinen työkalu on kommunikointi (Haikala & Mikkonen 2011, 69).

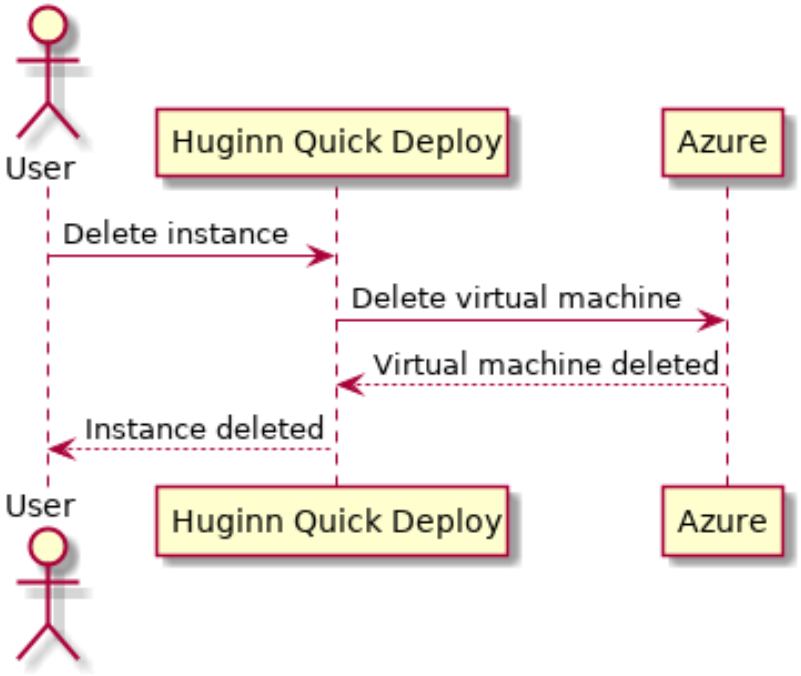
Kun projekti oli avattu ja vaatimukset olivat selvillä, ohjelmistosta tehtiin sekvenssikaavioita kuvaamaan Huginn Quick Deployn tärkeimpiä prosesseja. Nämä kaaviot kuvaavat hyvin, minkälaisia osia ohjelmisto vaatii.

Kuvio 6 kuvaa tapahtumaketjua, joka tapahtuu, kun käyttäjä luo uuden instanssin Huginn Quick Deploy -ohjelmistolla. Käyttäjä luo uuden instanssin Huginn Quick Deploy -käyttöliittymällä, joka pyytää Azurea luomaan uuden virtuaalikoneen. Kun Azure on luonut uuden virtuaalikoneen, Huginn Quick Deploy asentaa Huginn-ohjelmiston kyseiselle virtuaalikoneelle. Kun kaikki nämä vaiheet ovat käyty läpi, on uusi instanssi luotu ja valmiina käytettäväksi.



Kuvio 6. Sekvenssikaavio instanssin luomiseksi Huginn Quick Deployyn

Huginn-instanssin poistaminen (ks. Kuvio 3) tapahtuu pyytämällä Huginn Quick Deployta poistamaan instanssin. Huginn Quick Deploy lähettää Azurelle poistopyynnön, joka poistaa virtuaalikoneen. Kun Azure on toteuttanut instanssin poistamisen, instanssi on poistettu.



Kuvio 7. Sekvenssikaavio instanssin poistamiseksi Huginn Quick Deploysta

3.2 Kehitysympäristön pystyttäminen

3.2.1 Ohjelmien asentaminen

Jotta itse ohjelmiston kehitys voitiin aloittaa, tuli asentaa Elixir ja Erlang Ubuntu-koneelle. Tämä tapahtui käyttämällä asdf-nimistä ohjelmaa. Ensimmäinen vaihe oli asentaa asdf ubuntu-koneelle, jonka jälkeen tuli asentaa sekä Erlang että Elixir tällä asdf-ohjelmalla Ubuntu-koneelle. Ensimmäisenä tuli ladata asdf github-palvelusta *git clone* -komennolla.

Kuviossa 8 näkyy git clone -komento, ja sen argumentiksi on annettu asdf:n repositorion osoite, latauskansio sekä versionumero. Komento luo kansion nimeltä .asdf ja lataa asdf:n repositorion sisällön viimeisimmästä julkaistusta versiosta kyseiseen kansioon. Asdf on nyt olemassa kyseisellä tietokoneella, mutta se ei ole vielä toiminnallinen. Seuraava vaihe oli siis ajaa asdf-ohjelman asennuskomennot. (ks. kuvio 8).

```
veeti@veeti:~$ git clone https://github.com/asdf-vm/asdf.git ~/.asdf --branch v0
.6.1
Cloning into '/home/veeti/.asdf'...
remote: Enumerating objects: 53, done.
remote: Counting objects: 100% (53/53), done.
remote: Compressing objects: 100% (32/32), done.
remote: Total 3487 (delta 22), reused 41 (delta 20), pack-reused 3434
Receiving objects: 100% (3487/3487), 587.61 KiB | 612.00 KiB/s, done.
Resolving deltas: 100% (1849/1849), done.
Note: checking out '909a0e9b4b2fc37adcf3f3227c1354759be9921e'.
```

Kuvio 8. Asdf-sovelluksen lataaminen

Komennot kuviossa 9 asettavat Ubuntussa toimivan bash-komentokehotteen ajamaan asdf:n initialisointi tiedostot joka kerta kun bash käynnistyy. Useat ohjelmointikielet vaativat asennuksen yhteydessä useita ohjelmia toimiakseen. Asdf:n sivuilla on listattu kyseiset ohjelmat sen tukemille kielille, joten nämä ohjelmat tuli asentaa myös, jotta asdf toimii oikein. Asdf:n avulla voitiin asentaa Elixir ja Erlang. Ohjelmointikielen asentaminen asdf:lla sisältää kolme vaihetta: asdf pluginin lisäämisen, pluginin sisältämän kielen asentamisen ja kielen asettamisen joko globaaliksi tai lokaaliksi. Ensimmäisenä täytyi asentaa Erlang, sillä Elixir on riippuvainen Erlangista. (ks. kuvio 9).

```
veeti@veeti:~$ echo -e '\n. $HOME/.asdf/asdf.sh' >> ~/.bashrc
veeti@veeti:~$ echo -e '\n. $HOME/.asdf/completions/asdf.bash' >> ~/.bashrc
```

Kuvio 9. Asdf-sovelluksen asentaminen

Asdf-pluginin lisääminen on erittäin helppoa kielelle, jolle on olemassa tuki valmiina asdf:ssa (ks. kuvio 10). Kun plugin oli lisätty, tuli asentaa Erlangista uusin versio eli Erlang 21.1.

```
veeti@veeti:~$ asdf plugin-add erlang
initializing plugin repository...
```

Kuvio 10. Erlang-kielen lisääminen asdf-sovellukseen

Asennuskomento vaatii asennettavan kielen nimen sekä kielen versionumeron, joka halutaan asentaa (ks. kuvio 11). Viimeinen vaihe Erlang-kielen asentamisessa oli asettaa kieli globaaliksi, tarkoittaen, että asdf asettaa kyseisen version aktiiviseksi.

```
veeti@veeti:~$ asdf install erlang 21.1
Extracting source code
Building Erlang/OTP 21.1 (asdf_21.1), please wait...
```

Kuvio 11. Erlang-version 21.1 asentaminen asdf-sovelluksella

Kun asetetaan jokin kieli globaaliksi, komennolle tulee antaa asetettavan kielen nimi sekä versio, joka halutaan asettaa globaaliksi (ks. kuvio 12). Erlang-kielen asennuksen jälkeen sama toistettiin Elixir-kielelle. Kun Erlang ja Elixir oli asennettu, jäljelle jäi asentaa PostgreSQL, Terraform, Phoenix Framework ja Visual Studio Code. PostgreSQL:n asentaminen oli hyvin yksinkertainen prosessi, sillä sen asennus vaati vain yhden komennon ajamista Ubuntun bashilla.

```
veeti@veeti:~$ asdf global erlang 21.1
```

Kuvio 12. Erlang-kielen asettaminen globaaliksi asdf-sovelluksella

Kuviossa 13 oleva komento kutsuu APT-paketinhallintaohjelmaa asentamaan PostgreSQL ja postgresql-contrib -ohjelmien asennusta. Tämä komento asentaa kaikki tarvittavat riippuvuudet sekä PostgreSQL:n. Kun asennusprosessi oli viety loppuun, PostgreSQL oli asennettuna ja ajossa. Terraformin asennus vaati binääripaketin latauksen ja sen asettamisen PATH-ympäristömuuttujaan. Binääripaketti ladattiin Terraform sivustolta.

```
veeti@veeti:~$ sudo apt install postgresql postgresql-contrib -y
```

Kuvio 13. PostgreSQL:n asentaminen APT-ohjelmalla

Kuviossa 14 oleva rivi lisättiin .bashrc tiedostoon, jonka bash ohjelma lukee käynnistyessään. Edellä mainittu komento asettaa terraform-binääritiedoston PATH-ympäristömuuttujaan tehden Terraformista bash-ympäristössä ajettavan ohjelman.

```
export PATH=$PATH:$HOME/terraform
```

Kuvio 14. Terraformin suorituspolun asettaminen

Phoenix Frameworkin asentaminen vaatii olemassa olevan Erlang ja Elixir asennuksen sekä staattisten resurssien hallintaan nodejs-ohjelman. Phoenix Framework asennettiin käyttämällä Elixirin mix-työkalua.

Kuviossa 15 olevalla komennolla pyydetään mixiä asentamaan Phoenix Frameworkin versio 1.4.0 hex-paketinhallintajärjestelmästä, joka on tarkoitettu sekä Erlangille, että Elixirille. Jotta Phoenix Framework voi toimia halutulla tavalla tuli asentaa vielä nodejs-ohjelma. Nodejs asennettiin käyttämällä APT-ohjelmaa, ja sen asennuskomentoa.

```
veeti@veeti:~$ mix archive.install hex phx_new 1.4.0
```

Kuvio 15. Phoenix Frameworkin asentaminen Elixiriin

Komento kuviossa 16 on sama mitä käytettiin myös PostgreSQL:n asentamiseen, mutta eroavana tekijänä on tietenkin asennettavan ohjelman nimi. Viimeisenä oli Visual Studio Coden asentaminen, joka toteutettiin lataamalla ohjelman asennuspaketti ja ajamalla se. Asennuksen jälkeen tuli asentaa vaadittavat riippuvuudet käyttämällä APT-ohjelmaa.

```
veeti@veeti:~$ sudo apt install nodejs-legacy
```

Kuvio 16. Nodejs:n asentaminen

Riippuvuudet voidaan asentaa kuvion 17 komennolla, joka pakottaa APT-ohjelman tarkistamaan mahdollisten riippuvuuksien puuttumisen ja asentamaan puuttuvat riippuvuudet Ubuntulle.

```
veeti@veeti:~$ sudo apt install -f
```

Kuvio 17. Riippuvuuksien asentaminen APT:ssa

3.2.2 Projektin luominen ja kytkeminen versionhallintaan

Projektin luominen ja sen kytkeminen versionhallintaan sisälsi kolme selkeää vaihetta: uuden projektin luominen, sen kytkeminen git-versionhallintaan sekä projektin konfigurointi. Ensimmäisenä luotiin uusi Phoenix Framework -projekti.

Uuden projektin luominen Phoenix Frameworkilla on yksinkertaista. Sitä varten on luotu mix-komentoja, jotka auttavat sekä uusien projektien luonnissa että olemassa olevan projektin hallinnoinnissa.

Komento kuviossa 18 luo uuden Phoenix Framework -projektin, joka sisältää jo valmiin pohjan projektin kehittämistä varten. Kun uusi projekti oli luotu, tuli alustaa uusi git-repositorio tälle projektille. Git-repositorio voidaan alustaa yksinkertaisesti kutsumalla *git init* -komentoa uuden projektin kansion sisällä. Kun uusi repositorio oli alustettu, sille tuli asettaa osoite, jonne kaikki muutokset tallennetaan. Tämä toteutettiin komennolla *git remote add origin*, joka lisää etärepositorion. Tälle komennolle annettiin parametriksi Bitbucket-palvelun repositorion osoite.

```
veeti@veeti:~$ mix phx.new huginn_quick_deploy
```

Kuvio 18. Uuden Phoenix Framework -projektin luonti

Kun uusi projekti oli luotu, tuli luoda uusi tietokanta PostgreSQL-järjestelmään. Ensimmäinen vaihe oli asettaa salasana PostgreSQL-tietokannan oletuskäyttäjälle. Salasana asetettiin PostgreSQL:n omasta komentokehotteesta komennolla *\password*.

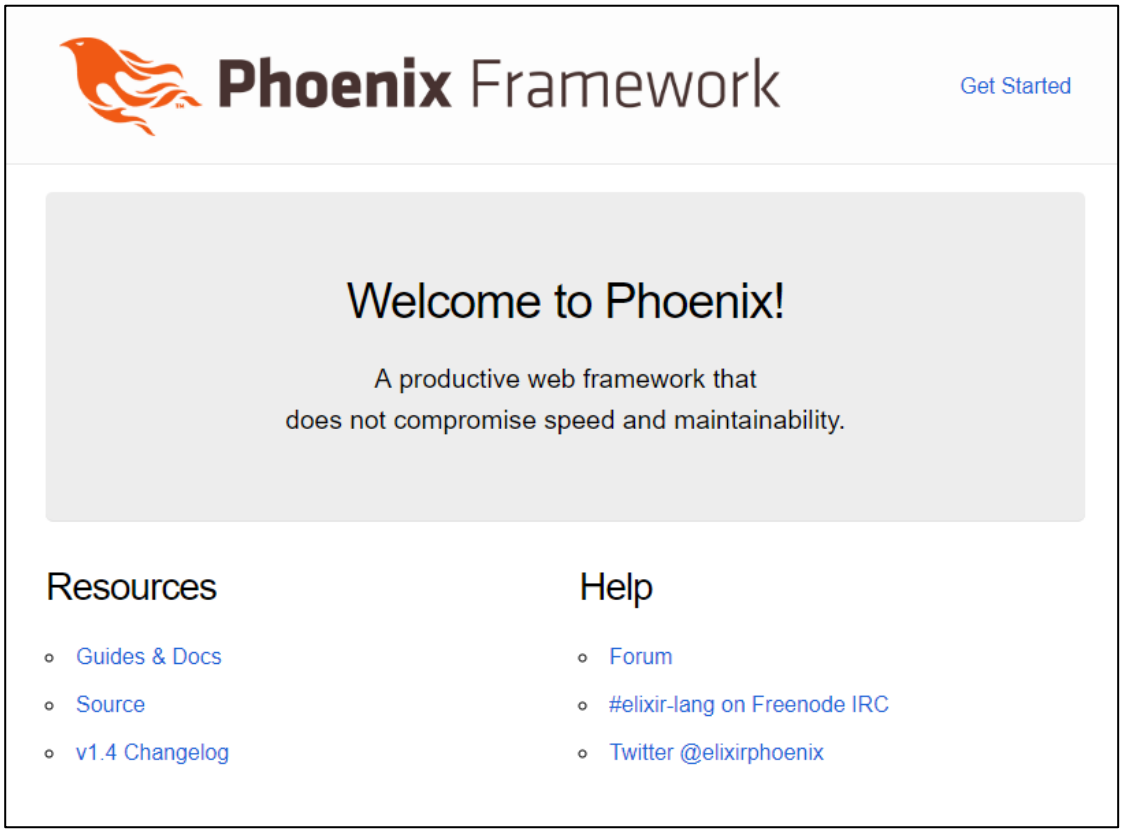
Normaalisti PostgreSQL:ään luodaan uusi tietokanta sen omasta komentokehotteesta, mutta Phoenix Frameworkia käytettäessä uusi tietokanta tulee luoda mix-komentojen avulla. Ennen kuin uutta tietokantaa voi luoda, tuli asettaa postgreSQL-käyttäjän tunnukset Phoenix projektin dev.exs-konfiguraatiotiedostoon.

Konfiguraatiotiedostoon (ks. kuvio 19) asetetaan valmiiksi oletuskohta postgreSQL-tunnuksille, kun Phoenix Framework -projekti luodaan. Username-kohtaan asetettiin PostgreSQL-käyttäjän tunnus, password-kohtaan asetettiin kyseisen käyttäjän salasana, database on tietokannan nimi, joka halutaan luoda, hostname on tietokannan osoite, ja viimeisenä pool_size on samanaikaisten yhteyksien määrä kyseiseen tietokantaan. Kun tietokannan tiedot oli asetettu dev.exs-konfiguraatiotiedostoon, oli mahdollista luoda uusi tietokanta. Tämä tietokanta luotiin mixin komennolla *mix ecto.create*, joka luo uuden tietokannan.

```
69  # Configure your database
70  config :huginn_quick_deploy, HuginnQuickDeploy.Repo,
71    username: "postgres",
72    password: "postgres",
73    database: "huginn_quick_deploy_dev",
74    hostname: "localhost",
75    pool_size: 10
76
```

Kuvio 19. PostgreSQL:n tietokannan tietojen asettaminen Phoenix Frameworkissa

Kun kaikki nämä vaiheet oli saatu suoritettua, kehitysympäristö oli valmis kehityksen aloittamiseksi. Phoenix-projekti voitiin käynnistää käyttämällä komentoa *iex -S mix phx.server.* Edellä mainittu komento käynnistää Elixirin interaktiivisen komentokehotteen, ja käynnistää Phoenix-serverin. Phoenix Frameworkin oletussivuun pääsi käsiksi verkkoselaimella osoitteella localhost:4000. (ks. kuvio 20).



Kuvio 20. Phoenix Frameworkin oletusnäkymä ensimmäisellä käynnistyskerralla

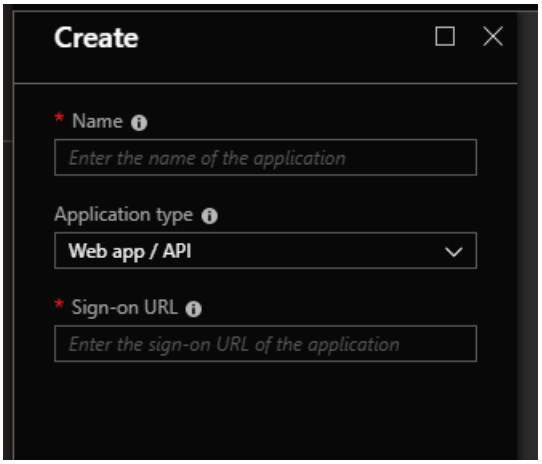
3.3 Terraform sovelluksen konfigurointi

3.3.1 Azure identiteetin ja roolin luonti

Ensimmäinen ohjelmiston konkreettinen kehitysvaihe oli Terraform-sovelluksen konfigurointi Azure-pilvipalvelua varten. Konfigurointiin kuului Azuren identiteetin luominen, terraformin yhteysavainten asettaminen, resurssien luominen ja Huginn-ohjelman automaattinen asentaminen. Jotta terraform voi luoda Azureen resursseja tulee Azureen luoda uusi identiteetti Huginn Quick Deploy -ohjelmistolle. Identiteetti sisältää client-avaimen, jolla Azure tunnistaa kyseessä olevan identiteetin, sekä client secret -avaimen, jolla Azure vahvistaa tunnistautumisen.

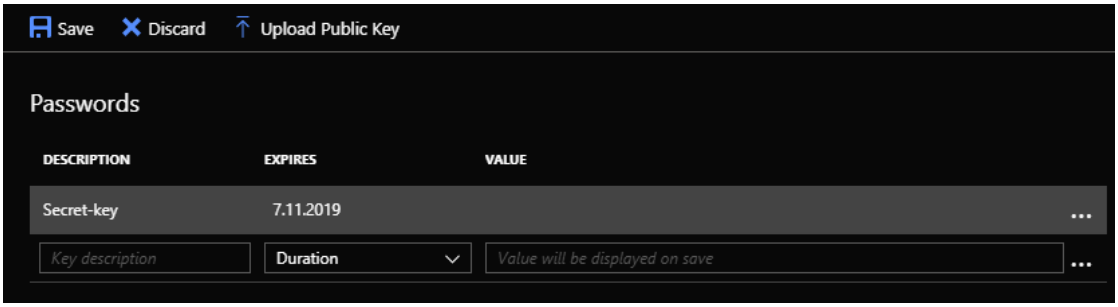
Jotta Terraform voi yhdistää Azureen ja luoda resursseja, sitä varten oli luotava Azureen oma identiteetti. Identiteetti luotiin Azure Active Directory -näkymässä, App

Registrations -välilehdellä. Tämä näkymä sisältää painikkeen nimeltä New application registration, josta painamalla aukeaa lomake, jolla rekisteröidään uusi sovellus Azureen (ks. kuvio 21). Lomakkeeseen tuli asettaa sovelluksen nimi, sovellustyyppi, sekä osoite.



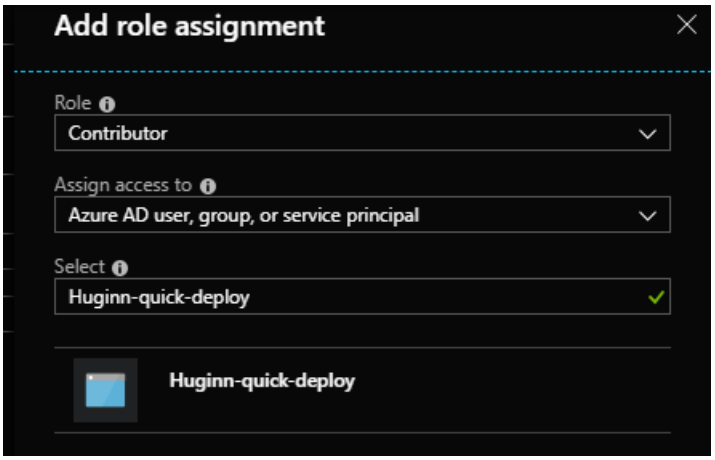
Kuvio 21. Sovelluksen rekisteröinti Azureen

Kun identiteetti oli luotu, tuli sille asettaa salainen avain, jotta Terraform voi kirjautua Azureen sisälle. Tämä avain asetettiin identiteetin asetuksista, keys-välilehdeltä. Kuviossa 22 on asetettu avain, jonka kuvaus on *Secret-key* ja se vanhenee vuoden kuluttua. Kun *Save*-painiketta painetaan, salainen avain näkyy *Value* kohdassa kerran, jonka jälkeen siihen ei pääse enää Azuressa käsiksi.



Kuvio 22. Salaisen avaimen luominen Huginn Quick Deploy -identiteetille

Identiteetti sisältää Terraformille vaadittavat tiedot, jotta terraform voi yhdistää Azureen, mutta identiteetille tuli asettaa vielä rooli subscriptioniin. Tämä rooli antaa identiteetille tarvittavat oikeudet toimia Azuressa. Rooli asetettiin *Subscriptions*-näkymässä. Tässä näkymässä valittiin subscription, johon rooli haluttiin asettaa. Valitun subscriptionin *Access control* -välilehdellä lisättiin rooli nimeltä Contributor (ks. kuvio 23). Contributorilla on oikeus vaikuttaa Azuren virtuaalikoneisiin. Roolin jäseneksi valittiin äskettäin luotu identiteetti.



Kuvio 23. Roolin asettaminen Huginn Quick Deploy -identiteetille

3.3.2 Azure virtuaalikoneen luominen Terraformilla

Jotta Huginn-instanssi voidaan asentaa Azureen ajettavaksi, tulee Huginnilla olla virtuaalikone, jossa Huginnia voidaan ajaa. Koska Azure vaatii virtuaalikonetta varten useita erilaisia Azuren resursseja, tulee Terraformin konfiguraatioon asettaa useita resursseja asennettavaksi. Nämä resurssit ovat: resource group, virtual network, subnet, public ip, network security group, network interface ja virtual machine. Ensimmäisenä tulee kuitenkin luoda terraform.tf niminen tiedosto, joka toimii terraformin konfiguraationa.

Ennen kuin Azureen voidaan alkaa luomaan mitään resursseja, tulee asettaa Terraformille yhteystiedot. Terraformille asetetaan provider nimeltä azurerm, jolle tulee

antaa subscription_id, client_id, client_secret ja tenant_id. Subscription avain saadaan Azuren subscriptions näkymästä, Client_id on Azuressa aiemmin luodun identiteetin Application_id, Client_secret on kyseiselle identiteetille luotu salainen avain, ja viimeisenä, tenant_id on Azuren directory_id-tunnus. (ks. kuvio 24).

```
provider "azurerm" {
    subscription_id = "123A-123A-123A"
    client_id       = "321B-321B-321B"
    client_secret   = "ABCD-4321-ABCD"
    tenant_id       = "A123-B321-D456"
}
```

Kuvio 24. Azuren yhteystietojen asettaminen terraformin konfiguraatioon

Kun yhteystiedot oli asetettu tuli luoda Azure resource groupia varten osio terraform.tf-tiedostoon. Kun terraform ajetaan tämä osio luo Azureen resource groupin. Ensimmäisellä rivillä kerrotaan, että kyseessä on resurssi, joka on Azuren resource group, ja tämän resurssin nimi on terraformgroup. Resource groupille tulee asettaa nimi, paikka, ja mahdolliset tagit. (ks.kuvio 25).

```
resource "azurerm_resource_group" "terraformgroup" {
    name     = "Deployer_test"
    location = "northeurope"

    tags {
        deployer = "test"
        uuid = "test"
    }
}
```

Kuvio 25. Resource groupin luominen

Tässä vaiheessa oli hyvä hetki kokeilla, että Azuren yhteystiedot olivat oikein asetettu ja kaikki toimi niin kuin pitääkin. Aluksi Terraform alustettiin *terraform init* -komennolla linux-komentokehotteessa, jonka jälkeen ajettiin *terraform apply*, joka ajaa terraformin luomisprosessin. Tämän prosessin jälkeen tarkastettiin, että itse komento-

kehotteessa ei ollut virheitä, ja Azureen oli ilmestynyt Deployer_test niminen resource group. Kun Terraform-asetukset oli vahvistettu toimiviksi, tuli resource group poistaa, joka toteutettiin käyttämällä *terraform destroy* -komentoa.

Seuraavaksi terraform.tf-tiedostoon lisättiin Azure virtual network. Resurssin rakenne konfiguraatiossa on resource groupin kanssa lähes sama, mutta virtual network sisältää myös address_space ja resource_group_name -attribuutit. Address_space-attribuutin tietotyyppi on lista, joka mahdollistaa usean eri osoiteavaruuden käytön. Resource_group_name-attribuuttiin voitaisiin asettaa suoraan resource groupin nimi, mutta Terraformilla on mahdollista käyttää muuttujia, joille asetetaan arvo, kun Terraform ajetaan. (ks. kuvio 26).

```
19  resource "azurerm_virtual_network" "terraformnetwork" {
20    name = "Deployer_test_Vnet"
21    address_space = ["10.0.0.0/16"]
22    location = "northeurope"
23    resource_group_name = "${azurerm_resource_group.terraformgroup.name}"
24
25    tags {
26      deployer = "test"
27      uuid = "test"
28    }
29  }
```

Kuvio 26. Virtual networkin luominen

Terraformin konfiguraatioon lisättiin loputkin resurssit, jotka olivat azurerm_subnet, azurerm_public_ip, azurerm_network_security_group, azurerm_network_interface ja azurerm_virtual_machine. Resursseihin vaadittavat attribuutit saatiin terraformin virallisesta dokumentaatiosta, joka löytyi Terraformin verkkosivuilta. Loput resurssit olivat muuten lähes samoja lisätä paitsi azurerm_network_security_group ja azurerm_virtual_machine.

Azure network security group sisälsi samat attribuutit, kun muissakin resursseissa, mutta security groupin sisään tuli lisätä myös sääntöjä. Sääntö sisältää useita attri-

buutteja, joilla voidaan hallita verkkoliikennettä. Nämä attribuutit asetettiin hyväksymään SSH-yhteys toimeksiantajan sisäverkosta. Network security groupiin asetettiin myös toinen sääntö hyväksymään HTTP-yhteys toimiston sisäverkosta. (ks. kuvio 27).

```
security_rule {
  name = "SSH"
  priority = 1001
  direction = "Inbound"
  access = "Allow"
  protocol = "Tcp"
  source_port_range = "*"
  destination_port_range = "22"
  source_address_prefix = "123.123.123.123"
  destination_address_prefix = "*"
}
```

Kuvio 27. SSH-yhteyden salliva turvallisuussääntö

Viimeisenä oli azurerm_virtual_machine, joka on edellä mainituista resursseista monimutkaisin. Virtual machine -resurssiin tuli asettaa yleisten tietojen lisäksi myös massamuistin tiedot, käyttöjärjestelmätiedot, käyttäjätiedot ja tunnistautumisasetukset. Merkittävimmät osiot virtuaalikoneen toiminnan kannalta olivat virtuaalikoneen tyyppi ja massamuistin tyyppi Azuressa. Nämä kaksi asetusta vaikuttivat sekä laskutukseen, että virtuaalikoneen suorituskykyyn. Virtuaalikoneen tyypiksi asetettiin Azuren B1s-virtuaalikonetyyppi, joka on yleiskäyttöön tarkoitettu virtuaalikonetyyppi. Massamuistiksi asetettiin Premium_LRS-levy, joka on tehokas SSH-levy. Massamuistiksi asetettiin SSH-levy, koska normaalityyppinen kovalevy todettiin liian hitaaksi. Käyttöjärjestelmäksi virtuaalikoneelle valittiin Ubuntu, sillä Huginn tukee Ubuntu-käyttöjärjestelmää. (ks. kuvio 28).

```
104  resource "azurerm_virtual_machine" "terraformvm" {
105    name = "Deployer_test_VM"
106    location = "northeurope"
107    resource_group_name = "${azurerm_resource_group.terraformgroup.name}"
108    network_interface_ids = ["${azurerm_network_interface.terraformnic.id}"]
109    vm_size = "Standard_B1s"
110
111    storage_os_disk {
112      name = "Deployer_test_os_disk"
113      caching = "ReadWrite"
114      create_option = "FromImage"
115      managed_disk_type = "Premium_LRS"
116    }
117
118    storage_image_reference {
119      publisher = "Canonical"
120      offer = "UbuntuServer"
121      sku = "16.04.0-LTS"
122      version = "latest"
123    }
124
125    os_profile {
126      computer_name = "test"
127      admin_username = "test"
128      admin_password = "testPasswd123"
129    }
130
131    os_profile_linux_config {
132      disable_password_authentication = false
133      ssh_keys {
134        path = "/home/test/.ssh/authorized_keys"
135        key_data = "ssh-rsa AAABBBCCCC test@test"
136      }
137    }
138
139    tags {
140      deployer = "test"
141      uuid = "test"
142    }
143  }
144
```

Kuvio 28. Virtuaalikoneen luominen

Kun Terraformin konfiguraatio oli asetettu, oli aika kokeilla konfiguraation toiminnallisuus. Toiminnallisuus kokeiltiin samalla tavalla, kuin aikaisemmin, eli ajamalla *terraform apply*. Kun Terraform oli ajanut komennon, komentokehote ilmoitti lopuksi vielä, että seitsemän resurssia oli lisätty Azureen. (ks. kuvio 29).

```
Apply complete! Resources: 7 added, 0 changed, 0 destroyed.
veeti@veeti:~/test$
```

Kuvio 29. Terraform-sovelluksen vahvistus resurssien luonnista

Vaikka Terraform ilmoittaa resurssien lisäyksestä, oli kuitenkin järkevää varmistaa, että nämä resurssit todella löytyivät Azuresta. Azuren resource groups -näkymästä löytyi Deployer_test niminen resource group. Deployer_test resource groupin sisältä löytyi kaikki Terraform-konfiguraatioon asetetut resurssit. (ks. kuvio 30).

NAME	TYPE	LOCATION
Deployer_test_IP	Public IP address	North Europe
Deployer_test_Net_Interface	Network interface	North Europe
Deployer_test_os_disk	Disk	North Europe
Deployer_test_Sec_Group	Network security group	North Europe
Deployer_test_VM	Virtual machine	North Europe
Deployer_test_Vnet	Virtual network	North Europe

Kuvio 30. Terraformilla luodut resurssit Azuressa

Kun resurssien olemassaolo oli vahvistettu, oli hyvä varmistaa, että kyseiseen virtuaalikoneeseen on mahdollista ottaa yhteys. Ensin otettiin SSH-yhteys virtuaalikoneen IP-osoitteeseen, joka vahvisti, että virtuaalikoneeseen oli mahdollista ottaa yhteys. Tämän jälkeen virtuaalikoneelle asennettiin nginx-palvelin, jotta HTTP-yhteydenotto voitiin myös vahvistaa. HTTP-yhteydenotto vahvistettiin ottamalla yhteys virtuaalikoneen IP-osoitteeseen verkkoselaimella. Verkkoselain avasi nginx-palvelimen oletussivun, joka vahvisti HTTP-yhteyden toimivuuden. (ks. kuvio 31).

Kuvio 31. Nginx oletusnäkymä verkkoselaimessa

3.3.3 Huginn-ohjelman asentaminen virtuaalikoneeseen

Kun Terraform-konfiguraation avulla voitiin luoda virtuaalikone Azureen, oli viimeinen vaihe Terraformin konfiguroinnissa asettaa se asentamaan Huginn. Terraformin virtuaalikone-resurssille voidaan antaa erikoiskäskyjä, joiden avulla voidaan lähettää tiedostoja virtuaalikoneeseen ja ajaa virtuaalikoneessa käskyjä.

Jotta Huginn voitaisiin asentaa virtuaalikoneelle, virtuaalikoneelle tulee lähettää Huginnin binääritiedostot. Nämä tiedostot ovat pakattuna .tar.gz-tiedostoon. Terraformilla voidaan lähettää virtuaalikoneelle tiedostoja provisioner ”file” -rakenteella. Tälle rakenteelle annetaan lähetettävän tiedoston polku ja polku johon tiedosto halutaan asettaa virtuaalikoneessa. Polkujen lisäksi rakenteelle annetaan yhteydenottoa varten connection-tiedot, jotka ovat virtuaalikoneen käyttäjätunnus ja salasana. (ks. kuvio 32).

```
139   provisioner "file" {
140     source = "/home/veeti/huginn.tar.gz"
141     destination = "/home/huginn/huginn.tar.gz"
142
143     connection {
144       user = "test"
145       password = "testPasswd123"
146     }
147   }
```

Kuvio 32. Tiedoston lähettäminen Terraformilla

Huginn-ohjelma täytyy asentaa virtuaalikoneelle, jota varten Terraform-konfiguraatioon tuli lisätä etäkomentoja ajava provisioner ”remote-exec” -rakenne. Rakenteen sisään asetettiin inline-lista, jonka sisään asetettiin komentoja yksittäisinä lista-elementteinä (ks. kuvio 33). Huginnin asentamisen vaiheet olivat: nginxin ja postgresql:n asentaminen, Huginn paketin purkaminen ja asettaminen oikeaan paikkaan, Huginn ympäristömuuttuja-tiedoston asetttaminen vaadittuun paikkaan, Huginn.service-tiedoston asettaminen systemd kansioon, Huginn-palvelun käynnistäminen ja nginx-palvelimen konfiguraation asettaminen.

```
149   provisioner "remote-exec" {
150     inline = [
151       "sudo apt update",
152       "echo \"hello\""
153     ]
154
155     connection {
156       user = "test"
157       password = "testPasswd123"
158     }
159   }
```

Kuvio 33. Remote-exec -rakenne

Kun kaikki asennusvaiheet oli tehty, tuli kokeilla ajaa Terraform ja varmistaa, että Huginn toimii virtuaalikoneella. Virtuaalikoneen IP-osoitteeseen otettiin yhteys verk-

koselaimella, johon aukesi Huginn-ohjelman sisäänkirjautumissivu (ks. kuvio 34). Huginn voidaan nyt asentaa Terraform-sovelluksella ja yhdellä komennolla, mutta tällä hetkellä Huginn asentuu virtuaalikoneelle samoilla tiedoilla.

Kuvio 34. Huginnin sisäänkirjautumissivu

3.4 Käyttöliittymän kehittäminen

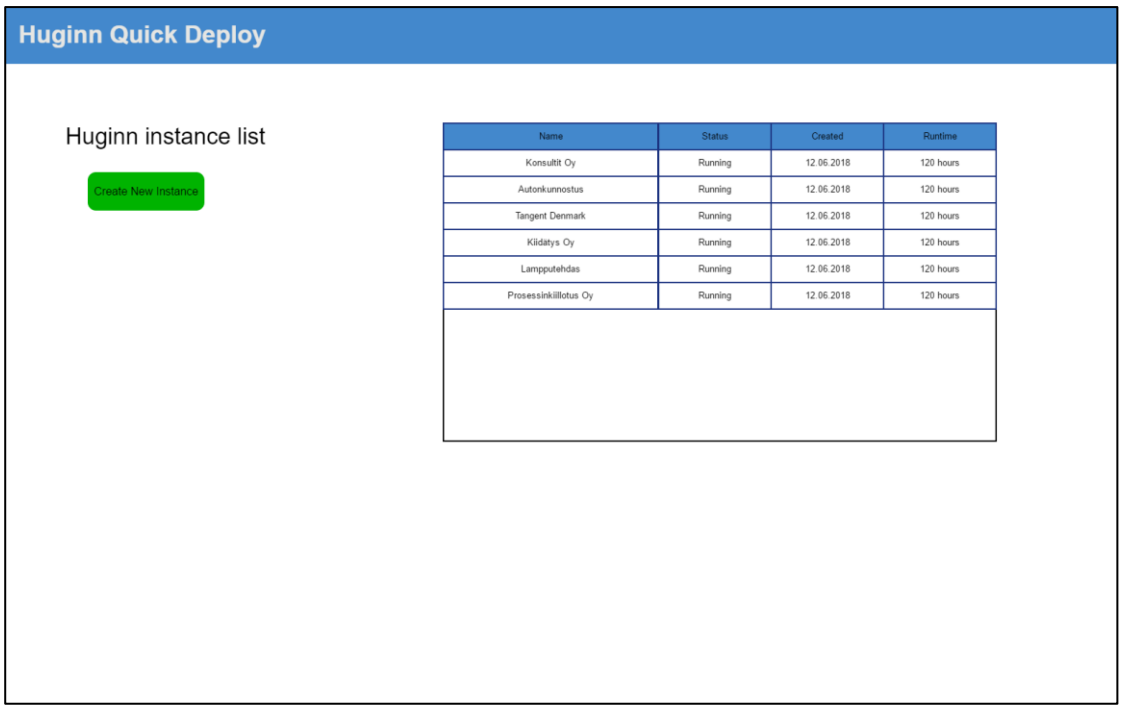
3.4.1 Yleistä

Käyttöliittymän kehittäminen koostui neljästä päävaiheesta: mockupin luonti, html-sivujen luonti, html sivujen muovaus phoenix yhteensopiviksi ja backend-toiminnallisuuden liittäminen käyttöliittymään. Käyttöliittymän kehityksessä ei vielä tuotu minkäänlaista todellista toiminnallisuutta ohjelmistolle, vaan pelkkä käyttöliittymä.

3.4.2 Mockup

Käyttöliittymän ensimmäinen vaihe oli mockupin luominen. Mockup tehtiin käyttämällä avoimen lähdekoodin ohjelmaa nimeltä Pencil. Vaatimusten mukaan käytöliittymässä tuli olla lista Huginn-instansseista. Tästä tehtiin ensimmäinen käyttöliittymämockup.

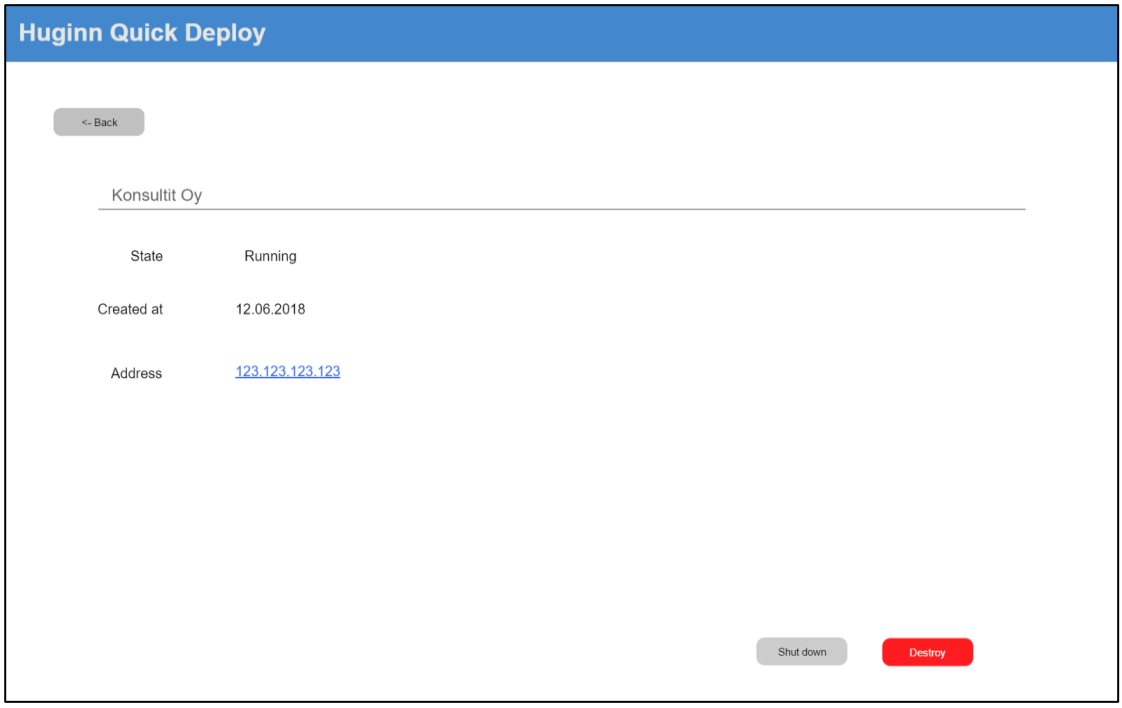
Kuvio 30 sisältää Huginn Quick Deploy -sovelluksen päänäkymän. Päänäkymä koostuu kahdesta ydinkomponentista: instanssilistasta ja uuden instanssin luomispainikkeesta. Instanssilista luetteloi kaikki eri Huginn-instanssit. Tässä luettelossa voidaan nähdä kyseisen instanssin nimi, tila, luomishetki ja ajoaika. Uuden instanssin luomispainikkeen tarkoitus on avata lomake, jolla voidaan luoda uusi instanssi. (ks. kuvio 35).



Kuvio 35. Huginn Quick Deployn päänäkymä

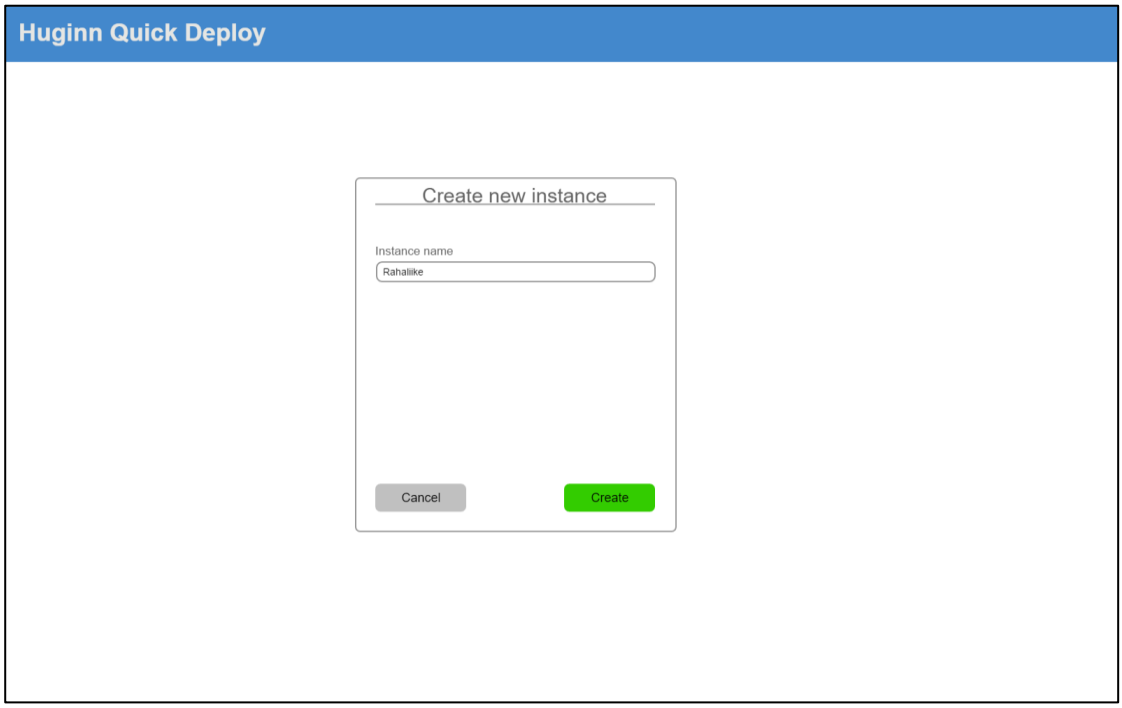
Kun ensimmäinen näkymä oli suunniteltu, tuli jatkaa mockupin kehitystä ja luoda näin loputkin näkymät. Näkymiä suunniteltiin yhteensä kolme: päänäkymä, instanssinäkymä ja instanssin luomisnäkymä.

Instanssinäkymä sisältää instanssin tiedot ja vaatimuksissa määritetyt toiminnot. Instanssin tiedot ovat instanssin nimi, tila, luomispäivä ja osoite. Vaaditut toiminnot instanssinäkymässä ovat instanssin sammuttaminen, käynnistäminen ja poistaminen. Sammuttamispainike on näkyvillä, kun instanssi on päällä, ja käynnistyspainike on näkyvillä, kun instanssi on sammutetussa tilassa. (ks. kuvio 36).



Kuvio 36. Huginn Quick Deployn instanssinäkymä

Instanssin luomisnäkymä on hyvin yksinkertainen, sillä se sisältää vain yhden lomakkeen. Tämän lomakkeen täyttäminen aloittaa uuden Huginn instanssin luomisprosessin. (ks. kuvio 37).

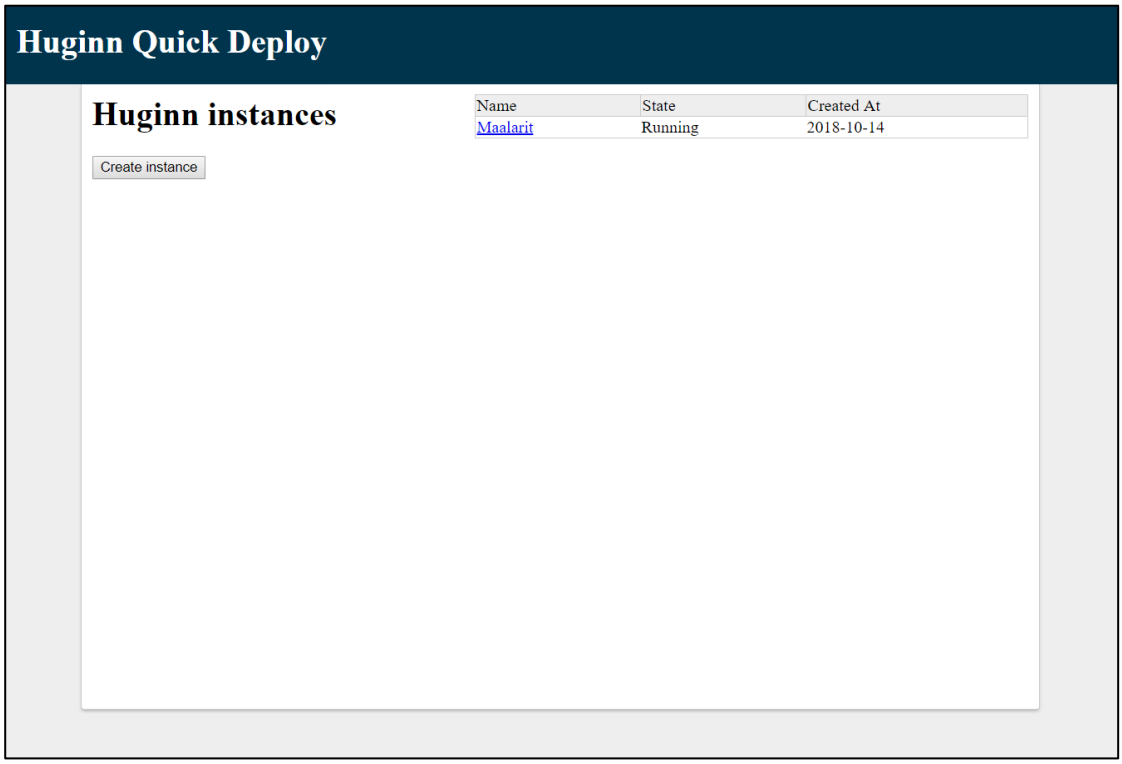


Kuvio 37. Uuden instanssin luomisnäkymä

3.4.3 HTML sivut

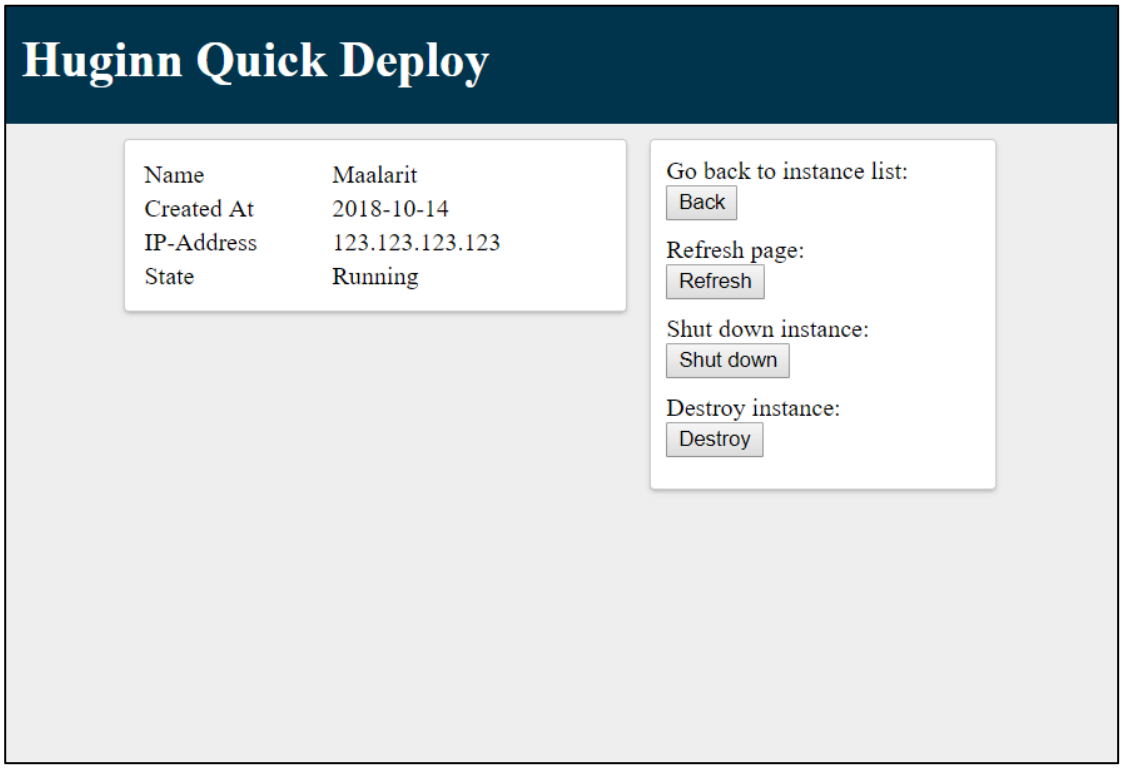
Kun mockup oli luotu, seuraava vaihe oli luoda HTML-sivut mockupin pohjalta. Nämä sivut eivät olleet vielä millään tavalla toiminnallisia, sillä ne olivat vain yhdistelmä HTML:ää ja CSS:ää.

HTML versio Huginn Quick Deployn päänäkymästä on hyvin samankaltainen mockup-kuvaan verrattuna. HTML-versiossa instanssilistasta jätettiin kuitenkin ajoaika pois, sillä se havaittiin tarpeettomaksi tiedoksi. (ks. kuvio 38).



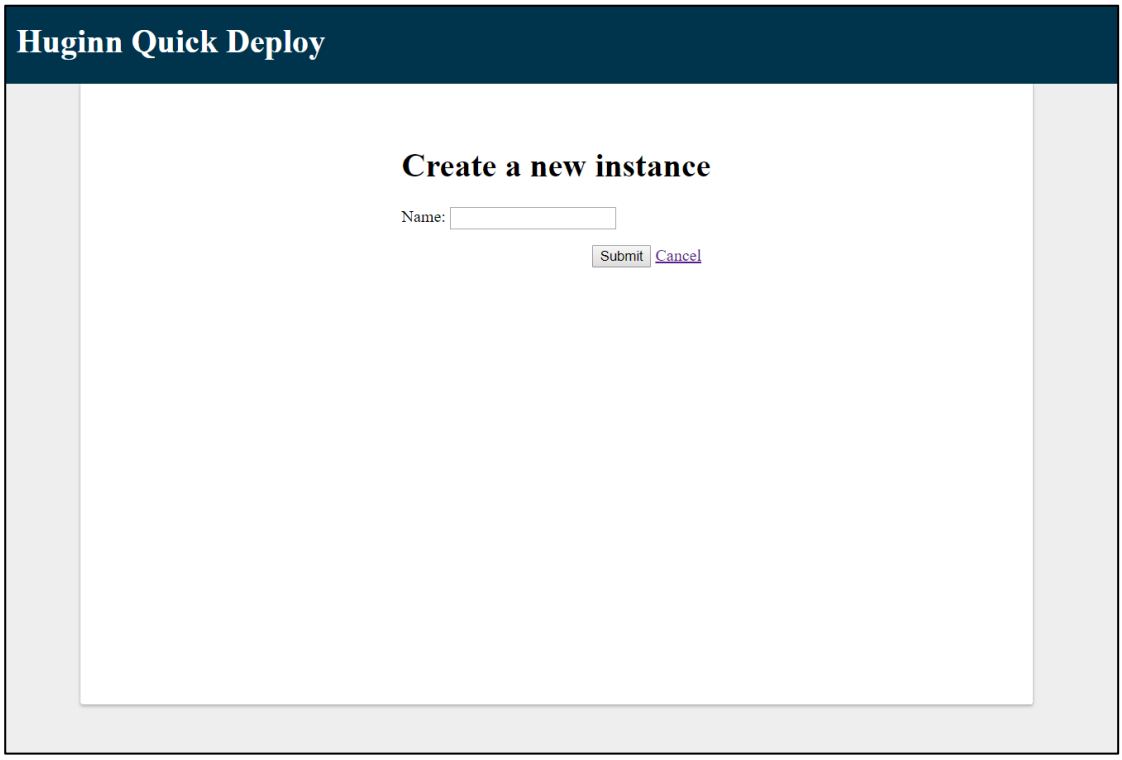
Kuvio 38. HTML-versio päänäkymästä

Instanssinäkymän ulkonäköä muutettiin hieman, jotta se vastaisi paremmin Huginn-sovelluksen kaltaista käyttöliittymää. Kaikki komponentit sisällytettiin kuitenkin HTML-sivuun. (ks. kuvio 39).



Kuvio 39. HTML-versio instanssinäkymästä

Uuden lomakkeen luomisnäkymä säilytettiin samana, eikä siihen tehty mitään muutoksia mockup-kuvaan verrattaen. (ks. kuvio 40).



Kuvio 40. HTML-versio instanssin luomisnäkymästä

3.4.4 Phoenix templaattien luonti

Kun HTML-sivut oli luotu, ne tuli muovata Phoenix Frameworkiin toimiviksi. Phoenix Framework käyttää HTML-sivujen renderöinnissä Embedded Elixiriä, eli HTML-sivuista tuli luoda EEx-tiedostoja.

Phoenix Frameworkiin voidaan asettaa HTML-sivujen pohja, joka sisältää sivujen toistuvan rakenteen. Pohjaan asetettiin linkki CSS-tyylitiedostoon sekä HTML-sivujen yläpalkki, joka toistuu jokaisessa näkymässä. Phoenix Frameworkissa staattiset tiedostot lisättiin käyttämällä *static_path()* -funktiota. Funktiolle asetetaan yhteyttä kuvaava *@conn*-muuttuja, sekä tiedostopolku, joka osoittaa CSS-tyylitiedostoon (ks.

kuvio 41). Yläpalkki muodostui normaalista HTML ja CSS -koodista.

```
9
10      <title>Huginn Quick Deploy</title>
11      <link rel="stylesheet" href="<%= static_path(@conn, "/css/app.css") %>">
12    </head>
```

Kuvio 41. CSS-tyylitiedoston linkitys

Jotta pohjaan voitiin sisällyttää uutta sisältöä, pohjaan täytyi asettaa renderöintifunktio, joka vastaanottaa renderöitävän näkymän, sekä renderöitävän sapluunan. Funktiolle annetaan myös kaikki renderöitävälle sapluunalle asetetut datat. Kun funktio ajetaan, se muuttuu halutun näkymän HTML-koodiksi. (ks. kuvio 42).

```
21        <main role="main">
22          <%= render(@view_module, @view_template, assigns) %>
23        </main>
```

Kuvio 42. Renderöintifunktio

Kaikkiin HTML-sivuihin lisättiin .html-tiedostopäätteen loppuun vielä .eex-pääte, joka antaa Phoenix Frameworkin ymmärtää, että kyseiset tiedostot ovat renderöitävissä. Tämän jälkeen tiedostot asetettiin Phoenix Frameworkin tarjoamaan templates-kansioon ja sen sisällä page-kansioon. Kaikki näkymät, jotka halutaan luoda, tulee asettaa templates-kansioon.

Tässä vaiheessa kaikki HTML-sivut olivat valmiita kokeiltavaksi, mutta niihin ei päässyt vielä käsiksi. Tämä johtui Phoenix Frameworkista, sillä siihen ei oltu asetettu reitityksiä eri näkymille.

3.5 Backendin kehittäminen

3.5.1 Yleistä

Backendin kehittäminen sisälsi seuraavat vaiheet: HTML-sivujen reititys, HTML-sivujen dynaaminen täyttäminen, Terraformin kytkeminen Huginn Quick Deployhyn ja Azuren API-kutsujen luominen. Kun nämä kehitysvaiheet oli tehty, tuotteen vaatimukset olivat täytetty.

3.5.2 HTML-sivujen reititys

Jotta aikaisemmin luotuihin HTML-sivuihin pääsi käsiksi Phoenix Frameworkin kautta, HTML-sivut tuli reitittää Phoenix Frameworkissa controllereihin. Controllerilla voidaan pyytää tiettyä HTML-sivua renderöitäväksi, joka lähetetään käsittelyn jälkeen käyttäjän verkkoselaimelle.

Phoenix Framework sisältää router.ex-tiedoston, johon asetettiin vaadittavat polut. Polku on funktio, joka sisältää itse polun, joka osoittaa näkymään selaimella, controllerin, josta näkymä halutaan renderöidä ja atomin, joka kutsuu controller-tiedostossa oikeaa funktiota (ks. kuvio 43). Kuviossa 43 on kaksi polkua, joista ensimmäinen osoittaa Huginn Quick Deployn pääsivulle ja toinen osoittaa uuden instanssin luomissivulle.

```
19      get("/", PageController, :index)
20      get("/new", PageController, :new)
```

Kuvio 43. Kaksi esimerkkipolkua router.ex-tiedoston sisällä

Router.ex-tiedosto ei ota kantaa, että mikä HTML-sivu tulee renderöidä vaan se ainoastaan ohjaa HTTP-kutsun oikealle controller-funktiolle. Controller-funktio, jolle rou-

ter.ex-tiedosto on ohjannut HTTP-kutsun, määrittää renderöitävän HTML-sivun. Kyseinen funktio voi myös asettaa lisää dataa renderöitävälle sivulle, mutta tässä vaiheessa funktio ainoastaan renderöi vaaditun sivun. (ks. kuvio 44).

```
9   def index(conn, _params) do
10    render(conn, "index.html")
11  end
```

Kuvio 44. Päänäkymän controller-funktio

Kun ensimmäiselle sivulle oli asetettu sekä reititys, että controller-funktio, oli hyvä hetki kokeilla, että sivu todella toimi. Huginn Quick Deploy -palvelin käynnistettiin ja siihen otettiin yhteys selaimella. Kun selaimen osoitekenttään asetettiin osoitteeksi *localhost:4000*, Huginn Quick Deployn päänäkymä aukesi selaimeen. Sivu ei vielä ollut toiminnallinen, eikä muita sivuja vielä ollut asetettu.

Kaikille muillekin sivuille asetettiin polku ja controller-funktio, jotka eivät sisältäneet vielä mitään toiminnallisuutta, mutta tällä tavoin oli mahdollista linkittää sivut toisiinsa. Sivujen linkitykseen käytettiin normaalia HTML:n a-elementtiä. Instanssinäkymään asetettiin testidataa, jotta sivun toimivuus voitiin vahvistaa.

3.5.3 Tietokantamallin luominen

Vaikka Huginn Quick Deploylle luotiin tietokanta projektin luomisvaiheessa, siinä ei ollut vielä minkäänlaisia tauluja. Huginn Quick Deployta varten tuli luoda taulu nimeltä instances, jonka tarkoituksena oli säilöä Huginn-instanssien tiedot.

Phoenix Frameworkiin voidaan luoda uusi tietokantataulu käyttämällä *mix phx.gen.schema* -taskia. Kyseiselle taskille tulee asettaa scheman nimi, taulun nimi, kenttien nimet ja niiden tietotyypit. Taululle asetettiin kentiksi nimi, koodi, uuid, tila ja ip-osoite. Phoenix Framework lisää tauluun myös kolme automaattisesti luotua kenttää, jotka ovat id, inserted_at ja updated_at. (ks. kuvio 45).

```
veeti@veeti:~/huginn-quick-deploy$ mix phx.gen.schema Instance instances
name:string code:string uuid:string state:string ip_address:string
```

Kuvio 45. Uuden tietokantataulun luomiskomento

Nimi ja koodi ovat tietomuodoltaan tekstiä ja ne ovat käytännössä instanssin nimi. Nimi on arvo, joka asetetaan uuden instanssin luomislomakkeessa. Koodi on lähes sama kuin nimi, mutta se on muotoiltu sopimaan Terraformin konfiguraatioon, sekä Azuren resurssien nimikenttiin sopivaksi. Uuid on kenttä, jonka perusteella Huginn Quick Deploy näyttää kyseisen instanssin instanssinäkymän. Tilan avulla nähdään, että onko kyseinen instanssi luotu ja ajamassa vai ollaanko sitä luomassa vai ollaanko sitä poistamassa. IP-osoitteen avulla käyttäjä pääsee kyseisen instanssin sivustoon käsiksi.

Kun mix task on ajanut Huginn Quick Deployn lib-kansion sisälle oli luotu instance.ex-tiedosto. Instance.ex on Schema instansseille, johon voidaan asettaa asetuksia Ectoa varten. Instance.ex tiedoston kautta tietokannalle kerrottiin, että name, code ja uuid -kentät ovat uniikkeja. Tämä tieto asetettiin changeset funktion sisään. (ks. kuvio 46).

```
20  def changeset(instance, attrs \\ %{}) do
21    instance
22    |> cast(attrs, [:name, :code, :uuid, :state, :ip_address])
23    |> validate_required([:name, :code, :uuid, :state, :ip_address])
24    |> unique_constraint(:name)
25    |> unique_constraint(:code)
26    |> unique_constraint(:uuid)
27  end
```

Kuvio 46. Kenttien name, code ja uuid uniikkirajoitteet instance.ex-tiedostossa

3.5.4 HTML-sivujen dynaamisuus

Tällä hetkellä kaikki HTML-sivut olivat staattisia, joka tarkoittaa, että sivujen sisältö säilyi aina samana. Jotta Huginn Quick Deploy sovelluksella oli mahdollista tehdä sen vaatimat tehtävät, tuli siihen lisätä interaktiot tietokannan kanssa.

Ensimmäinen vaihe oli tehdä uuden instanssin luomislomakkeesta toiminnallinen siinä määrin, että lomakkeen lähettäminen palvelimelle loi tietokantaan uuden tietueen. Ensinnäkin tuli lisätä instanssin luomislomakkeen HTML-tiedostoon Phoenix Framework lomake, joka on lähestulkoon samanlainen lomake verrattuna HTML5 lomakkeeseen, mutta se sisältää muun muassa automaattisesti luodun CSRF-tokenin, joka lisää lomakkeen turvallisuutta. Lomaketta varten kutsutiin form_for-funktiota, jolle annettiin lomakkeen kontrollerin polku. Lomakkeelle annettiin vaadittavat kentät ja painikkeet, jotka olivat nimikenttä, lähetyspainike ja keskeytyspainike. (ks. kuvio 47).

```
4     <%= form_for @changeset, page_path(@conn, :create), fn f -> %>
5       <label>Name:</label>
6         <%= text_input f, :name %>
7         <div class="form-actions">
8           <%= submit "Submit" %>
9           <%= link "Cancel", to: "/" %>
10        </div>
11    <% end %>
```

Kuvio 47. Uuden instanssin luomislomake

Nyt kun uuden instanssin luomislomake sisälsi toiminnallisen lomakkeen, tuli sitä varten lisätä toiminnallisuus luomislomakkeen kontrolleriin. Luomislomakkeen kontrollerin nimi oli create ja sen sisälle asetettiin koodia, joka vastaanottaa lomakkeen datan ja asettaa sen tietokantaan. Kontrolleriin asetettiin myös instanssitaulun muiden tietojen alustavat arvot. Kun tiedot ovat asetettu tietokantaan, kontrolleri uudelleenohjaa käyttäjän kyseiseen instanssinäkymään. (ks. kuvio 48).

```
def create(conn, %{"instance" => params}) do
  params = params
  |> Map.put("code", String.replace(String.downcase(params["name"]), " ", "-"))
  |> Map.put("uuid", UUID.generate())
  |> Map.put("state", "Pending")
  |> Map.put("ip_address", "Waiting")

  %Instance{}
  |> Instance.changeset(params)
  |> Repo.insert()
  |> case do
    {:ok, instance} ->
      redirect(conn, to: page_path(conn, :show, instance.uuid))
    {:error, _err} ->
      changeset = Instance.changeset(%Instance{})
      conn
      |> assign(:changeset, changeset)
      |> render("new.html")
  end
end
```

Kuvio 48. Luomislomakkeen kontrolleri

Seuraava vaihe oli lisätä vaadittava toiminnallisuus Huginn Quick Deployn päänäkymään, eli instanssin listausnäkymään. Tällä hetkellä päänäkymän sisältämä lista oli staattinen. Päänäkymän kontrolleri itsessään päätyi olemaan hyvin yksinkertainen, sillä listan populointi tietokannasta ei vaatinut kovinkaan monimutkaista koodia. Kontrolleri hakee tietokannasta kaikki instanssitietueet *Repo.all()* -komennolla ja asettaa tietueet renderöitävälle sivulle. (ks.kuvio 49).

```
def index(conn, _params) do
  instances = Repo.all(from i in Instance, order_by: [desc: i.updated_at])

  conn
  |> assign(:instances, instances)
  |> render("index.html")
end
```

Kuvio 49. Päänäkymän kontrolleri

Kun päänäkymän kontrolleri oli valmis, jäljellä oli enää instanssinäkymän kontrolleri. Instanssinäkymän kontrollerin tarkoitus on hakea tietokannasta yksittäisen instanssin

tiedot instanssin uuid-kentän perusteella. Haetun instanssin tiedot asetetaan renderöitävälle sivulle, jotka näkyvät lopulta verkkoselaimella instanssinäkymässä. (ks. kuvio 50).

```
def show(conn, %{"instance" => uuid}) do
  case Repo.one(from i in Instance, where: i.uuid == ^uuid) do
    instance = %Deployer.Instance{} ->
      conn
      |> assign(:instance, instance)
      |> render("show.html")
    nil ->
      redirect(conn, to: page_path(conn, :index))
  end
end
```

Kuvio 50. Instanssinäkymän kontrolleri

Huginn Quick Deploylla pystyi nyt luomaan uusia instanssitietueita ohjelman tietokantaan, mutta tietueita ei voinut vielä poistaa. Poistamistoiminnallisuuden lisääminen vaati uuden kontrollerin. Poistamiskontrolleria kutsutaan HTTP POST -pyynnön avulla, joka tulee instanssinäkymäsivulta. Poistaminen tapahtuu hakemalla poistettava tietue tietokannasta ja antamalla ectolle poistopyyntö, jolloin ecto poistaa kyseisen tietueen tietokannasta. (ks. kuvio 51).

```
def remove(conn, %{"instance" => uuid}) do
  instance = Repo.one(from i in Instance, where: i.uuid == ^uuid)
  Repo.delete(instance)

  redirect(conn, to: page_path(conn, :index))
end
```

Kuvio 51. Instanssin poistamiskontrolleri

Kun kaikki kontrollerit oli tehty, Huginn Quick deployn UI oli lähes täysin toiminnallinen. Ainoat puuttuvat toiminnot UI:ssa olivat instanssin sammuttaminen ja käynnistys, mutta instanssin sammuttaminen ja käynnistäminen vaati Azuren API-rajapinnan ohjelmoinnin, joka toteutettiin myöhemmin.

3.5.5 Terraformin kytkeminen ohjelmistoon

Jotta Terraformia voitiin hyödyntää ohjelmallisesti, ensimmäinen vaihe oli muovata Terraformin konfiguraatio dynaamiseksi. Käytännössä tämä tarkoitti konfiguraation muokkaamista .eex-tiedostoksi, eli terraform.tf.eex, ja muuttamalla kaikki instanssille kriittiset kentät konfiguraatiossa dynaamisesti asetettaviksi. Dynaamisesti asetettavia kenttiä oli muun muassa instanssin nimi ja Azure REST rajapinnan autentikaatiotiedot. Jokaisen konfiguraatioon luodun rakenteen nimi tuli asettaa dynaamiseksi kuvio 52 mukaisesti. On äärimmäisen tärkeää voida asettaa luotavan instanssin nimi dynaamiseksi, sillä muutoin terraform pyrkisi luomaan Azureen duplikaatti-instansseja, joka ei ole mahdollista.

```
resource "azurerm_resource_group" "terraformgroup" {
    name     = "Deployer_<%= terraform.name %>"
    location = "northeurope"

    tags {
        deployer = "<%= terraform.name %>"
        uuid = "<%= terraform.uuid %>"
    }
}
```

Kuvio 52. Azure resource group -rakenne muutettuna eex muotoon

Kun konfiguraatio oli muovattu dynaamiseksi, tuli Huginn Quick Deployhyn luoda komento, joka luo terraform.tf.eex -tiedostosta terraform.tf -tiedoston. Terraform.tf -tiedoston luominen vaati dynaamisten kenttien täyttämistä jollakin arvolla, jonka toteutus tehtiin uuteen moduuliiin Elixirillä. Moduulin nimeksi asetettiin Terraform ja kyseiseen moduuliin luotiin lopulta kaikki Terraformiin liittyvät toiminnot. Elixir sisältää moduulin .eex-tiedostojen evaluointiin, jonka nimi on EEx.

Terraform.tf-tiedoston luontia varten tuli luoda uusi funktio nimeltä *eval_terraform_config()*. Funktion tarkoituksena on hoitaa kaikki terraform.tf-tiedoston luontiin

vaadittavat vaiheet. Vaiheita on kaksi, joista ensimmäinen on vahvistaa huginn binäärin olemassaolo ja sen tiedostopolun tallettaminen muuttujaan. Toinen vaihe on luoda terraform.tf-tiedosto, jonka yhteydessä myös varmistetaan, että kaikki vaadittavat arvot on asetettu. *Eval_terraform_config()* -funktio vastaanottaa parametriksi listan, joka sisältää kaikki luomiseen vaadittavat arvot. Funktion ensimmäiset kuusi riviä varmistavat Huginn binääripaketin olemassaolon. Riveillä 8-10 luodaan itse terraform.tf-tiedosto. (ks. kuvio 53).

```
1  def eval_terraform_config(bindings) do
2    {:ok, list} = File.ls()
3    huginn = Enum.filter(list, fn x ->
4      String.match?(x, ~r/^(huginn_)[0-9]\.[0-9]\.[0-9].+(ubuntu).+(\.amd64\.tar\.gz)$/)
5    end)
6    case huginn do
7      [filename] ->
8        content = EEx.eval_file("terraform.tf.eex",
9                                terraform: Map.put(bindings, :huginn, filename))
10       File.write("terraform.tf", content)
11     _ ->
12       {:error, "Could not find huginn package, cannot create terraform.tf"}
13   end
14 end
```

Kuvio 53. Terraform.tf-tiedoston luontifunktio

Terraformin konfiguraation luomisen jälkeen tuli lisätä myös konfiguraation poistaminen. Konfiguraation poistaminen toteutettiin funktiossa nimeltä *remove_terraform_config*. Funktio sisälsi yhden komennon, joka poisti terraformin konfiguraation. Komento oli *File.rm()* niminen ja sitä käytetään elixirissä tiedostojen poistamiseen.

Seuraava vaihe oli luoda komennot, joilla voidaan ajaa terraformille yleisiä komentoja ohjelmallisesti. Luodut komennot olivat: workspacen luonti, valitseminen ja poistaminen; terraformin initialisointi, luonti ja tuhoaminen. Nämä komennot toteutettiin hyödyntämällä *System.cmd()* -komentoa. *System.cmd()* -komennolla voidaan ajaa järjestelmätason komentoja. Komennolle annetaan kaksi parametria, joista ensimmäinen on järjestelmätason komento ja toinen parametri on järjestelmätason komennon argumentit. (ks. kuvio 54).

```
167   defp init_terraform do
168     System.cmd("terraform", ["init", "-input=false"])
169   end
```

Kuvio 54. Terraformin initialisointi-komento

Hyödyntämällä aiemmin luotuja komentoja, voitiin luoda komento, joka vastaanottaa instanssitietueen ja luo tietueen tietojen perusteella uuden Huginn-instanssin Azureen. Kun komento oli luotu, vastaan tuli ongelma, joka aiheutti projektissa paljon päänvaivaa. Terraformin luontikomennon ajaminen vei noin 15 minuuttia aikaa ja tämän 15 minuutin aikana Huginn Quick Deploy ei vastannut HTTP-kutsuihin. Käytännössä Huginn Quick Deploy jumittui niin pitkäksi ajaksi, kunnes terraform komento oli ajettu loppuun. Ratkaisu ongelmaan oli GenServer, joka on palvelimen kaltainen osio Elixirissä. GenServerillä voidaan luoda uusia prosesseja Erlang VM:ään, joille voidaan lähettää kutsuja. Nämä kutsut voivat pyytää prosessia ajamaan komentoja, jotta itse pääprosessi ei jumiudu ajamaan pitkään ajettavaa komentoa.

GenServerin implementointi vaati muutamia vaiheita, jotta sitä pystyi käyttämään hyödyksi. Ensimmäinen vaihe oli pyytää Elixiriä aktivoimaan GenServer Terraform-moduuliin, joka toteutettiin kutsumalla *use GenServer* -komentoa Terraform-moduulin alussa. Toinen vaihe on lisätä GenServerin vaatimat callbackit, jotka ovat: init, start_link ja handle_cast. Start_link on callback, jota kutsutaan, kun Huginn Quick Deploy käynnistetään. Start_link käynnistää GenServerin kyseisessä moduulissa ja kutsuu samalla init-callbackia. Huginn Quick Deployn tapauksessa init-callback oli lähes tarpeeton, joten se palautti vain ok-viestin. Handle_cast-callback on kaikista keskeisin toiminnallisuus, sillä se vastaanottaa GenServerille lähetetyt kutsut. Terraform-moduulin tilanteessa handle_cast-callback joko pyytää Terraform-moduulia luomaan uuden instanssin tai poistamaan aiemmin luodun instanssin. Handle_cast-callback sisältää Elixirissä yleisen case-rakenteen, jossa muuttujan sisältö vaikuttaa funktion lopputulokseen. Handle_cast-callbackin tilanteessa, jos action-muuttujan arvo on :add, silloin luodaan uusi instanssi, jos arvo on :remove, kyseessä oleva instanssi poistetaan. (ks. kuvio 55).

```
47  def handle_cast({:push, {action, instance}}, _queue) do
48    case action do
49      :add ->
50        Logger.info("Adding instance with name: #{inspect instance.name}")
51        create(instance)
52        {:noreply, "Created"}
53      :remove ->
54        Logger.info("Removing instance with name: #{inspect instance.name}")
55        delete(instance)
56        {:noreply, "Removed"}
57    end
58  end
```

Kuvio 55. Handle_cast-callback

Kun Terraform moduulin toiminnallisuus oli saatu valmiiksi, viimeisenä tuli asettaa kontrollereille kutsut, jotka luovat uuden instanssin tai poistaa olemassa olevan instanssin. Instanssin luomiskutsu asetettiin uuden instanssin luomislomakkeen kontrolleriin. Instanssin luomiseen vaaditut tiedot saatiin suoraan kontrollerilta, eikä luomiskutsua varten tarvittu tehdä muutoksia kontrolleriin. Instanssin poistaminen tapahtui instanssin poistokontrollerissa, mutta sitä tuli muokata siten että poistokontrolleri ei enää poistanut instanssitietuetta tietokannasta suoraan, vaan kutsui Terraform-moduulia poistamaan instanssin, jonka jälkeen tietue poistettiin tietokannasta.

Tässä kehityksen vaiheessa Huginn Quick Deploylla oli mahdollista luoda toiminnallisia Huginn-instansseja. Huginn-instansseihin oli mahdollista ottaa yhteys verkkoselaimella ja Huginn oli täysin käytettävissä.

3.5.6 Azuren API-kutsut

Huginn Quick Deployhyn tuli vielä lisätä Azure-moduuli, jonka päätarkoitus oli keskustella Azuren API-rajapinnan kanssa. Azure-moduulin toiminnot olivat IP-osoitteen automaattinen haku, instanssin sammuttaminen ja instanssin käynnistäminen. Azuren API-rajapintaa hyödyntävät toiminnot toteutettiin hyödyntämällä Elixirin HTTPoison-kirjastoa.

Jotta Azuren API-rajapinta toimisi halutulla tavalla, tuli Azure-moduuliin luoda autentikointifunktio. Autentikointifunktion tarkoitus oli pitää rajapinnan toiminnot saatavilla aina kun Azure-moduulia pyydettiin suorittamaan jokin toiminto. Azuren API-rajapintaan autentikoiduttiin lähettämällä rajapinnalle kirjautumistiedot, johon Azure vastasi lähettämällä autentikaatiotokenin. Autentikaatiotoken oli merkkijono, joka toimi yhden tunnin kerrallaan, jonka jälkeen autentikaatio tuli tehdä uudestaan. Autentikaatiotokenin säilyttämiseen hyödynnettiin GenServeriä, jotta autentikaatiotokenia ei tarvinnut tallentaa levylle, mutta se oli silti aina saatavilla, kun Huginn Quick Deploy oli käynnissä.

Kun Huginn-instansseja oli mahdollista luoda ja poistaa, Huginn Quick Deploy oli lähes valmis, mutta yhteydenotto selaimella Huginn-instanssiin vaati käyttäjää avaamaan Azuren käyttöliittymän ja käydä hakemassa Huginn-instanssin IP-osoite Azuresta käsin. Tämä lisäsi epämukavuustekijän Huginn Quick Deployn käyttöön, jonka takia Huginn Quick Deployhyn tuli lisätä IP-osoitteen automaattinen haku. IP-osoitteen haku toteutettiin HTTP GET-pyynnöllä Azuren API-rajapinnassa. HTTP GET-pyyntö vaati instanssin nimen, Azuren subscription id:n sekä autentikaatiotokenin. Kyseisten tietojen avulla Azure pystyi palauttamaan kyseessä olevan instanssin IP-osoitteen. IP-osoite päivitettiin kyseisen instanssin tietokanta-tietueeseen, jonka jälkeen se asetettiin linkiksi instanssinäkymään. Linkin avulla Huginn-instanssin IP-osoite oli heti saatavilla instanssin luomisen jälkeen.

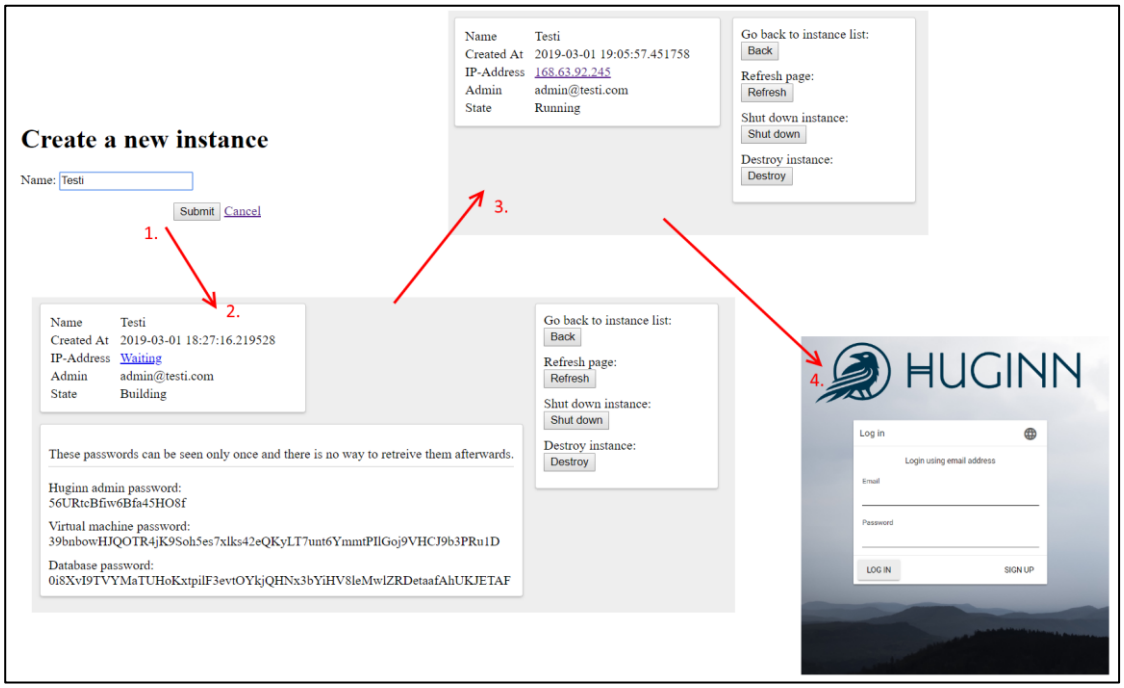
Huginn Quick Deployhyn haluttiin lisätä toiminto, jolla voidaan sammuttaa Huginn-instanssi ja käynnistää Huginn-instanssi. Tämä toiminto vaati myös Azuren API-rajapinnan hyödyntämistä, sillä instanssin sammuttaminen tai käynnistäminen tarkoittaa käytännössä Azuren virtuaalikoneen sammuttamista tai käynnistämistä. Käynnistämistä ja sammuttamista varten Huginn Quick Deployn instanssinäkymään oli lisätty tarvittavat painikkeet jo käyttöliittymän toteutusvaiheessa. Azuren API-rajapinta sisälsi valmiit pyynnöt virtuaalikoneen käynnistämiselle ja sammuttamiselle, joten toimintojen toteutusta varten Azuren API-rajapinnalle lähetettiin HTTPoisonilla joko käynnistys- tai sammutuspyyntö.

Kun nämä toiminnot oli luotu ja kytketty Huginn Quick Deployhyn, kaikki Huginn Quick Deployn vaatimukset olivat täytetty ja verkkosovellus oli valmis. Huginn Quick Deploy sisälsi kaikki toiminnot, jotka siltä vaadittiin.

4 Tulokset

Opinnäytetyön lopputuloksena saatiin aikaiseksi toiminnallinen verkkosovellus, jolla oli mahdollista luoda ja poistaa Huginn-instansseja yhdellä verkkolomakkeella. Sovelluksen kehitystyö kesti kokonaisuudessaan noin 2 kuukautta, jonka aikana kartoitettiin vaaditut ominaisuudet, suunniteltiin verkkosovelluksen prosessin kulku, valittiin vaaditut teknologiat ja kehitettiin toiminnallinen sovellus hyödyntämällä valittuja teknologioita.

Huginn Quick Deployn käyttö oli hyvin helppoa. Se sisälsi käyttäjälle vain yhden vaiheen, joka oli verkkolomakkeen täyttäminen. Tämä verkkolomake sisälsi pelkästään luotavan Huginn-instanssin nimikentän. Kun verkkolomake oli täytetty ja lähetetty takaisin palvelimelle, Huginn Quick Deploy aloitti uuden Huginn-instanssin luomisprosessin ja uudelleenohjasi käyttäjän instanssinäkymään. Instanssinäkymään tulostettiin myös automaattisesti luodut salasanat instanssia varten. Instanssin salasanat olivat näkyvillä vain kerran eikä niitä tallennettu minnekään tietoturva syistä. Kun Huginn-instanssin luomisprosessi oli valmis ja sivu ladattiin uudelleen, IP-osoitekenttä sisälsi linkin virtuaalikoneelle asetettuun IP-osoitteeseen. Kun linkkiä seurasi, se vei käyttäjän Huginn-verkkosovelluksen kirjautumissivulle. Kyseinen Huginn-verkkosovellus oli täysin käytettävissä ja toiminnallinen. (ks. kuvio 56).



Kuvio 56. Instanssin luomisprosessi

Huginn Quick Deployn vaatimusten ja käyttötapausten toteutuminen vahvistettiin hyväksyntätestauksella toimeksiantajan kanssa. Tämä tarkoitti käytännössä sitä että ohjelman toiminnot käytiin toimeksiantajan kanssa yhdessä läpi. Laadun varmistamiseksi ohjelmistoa testattiin myös ohjelmiston kehitysvaiheessa. Aina kun jokin toiminto oli valmis, se testattiin ja korjattiin jos toiminnosta löytyi virheitä.

Huginn Quick Deploylle oli ideoitu jatkokehitysideoita, joita ei sisällytetty opinnäytetyön laajuuteen. Jatkokehitysideoita oli muun muassa Huginn-instanssin ajastaminen siten että instanssi sammutetaan tietyn ajanjakson jälkeen, jonka jälkeen olisi mahdollista laskuttaa asiakasta uudesta ajanjaksosta ja jatkaa instanssin ajoa. Huginn Quick Deployta ajettiin kehityksen aikana paikallisesti, eikä siihen päässyt käsiksi, kun vain yhtiön sisäverkossa. Jos sovellus oltaisiin haluttu asettaa saataville muualtakin, olisi siihen tullut implementoida myös autentikaatio.

5 Pohdinta

Opinnäytetyön tavoitteena oli kehittää verkkosovellus, jonka avulla oli mahdollista luoda Huginn-ohjelmiston sisältäviä instansseja yhdellä verkkolomakkeella. Opinnäytetyön tavoitteet saavutettiin, sillä aiheen laajuus oli rajattu sopivan suuruiseksi. Valittujen teknologioiden hyödyntäminen opinnäytetyössä osoittautui hyödylliseksi, sillä ne vähensivät lopullista työn määrää huomattavasti.

Verkkosovelluksen toteutukseen käytetyt teknologiat olivat tiivistettynä Elixir, Terraform ja Azure. Elixir oli ohjelmointikielenä normaalista poikkeavaa, mutta sen dokumentaatio oli todella laadukasta, joka helpotti työn toteutusta. Verkkosovelluksen olisi voinut toteuttaa käytännössä lähes millä tahansa tunnetulla ohjelmointikielellä, mutta Elixir valittiin, koska sitä käytettiin toimeksiantajan muissakin projekteissa. Opinnäytetyön toteutus oli mahdollinen saavuttaa Terraformin avulla, sillä Terraform toi erittäin paljon automatisointia Azuren API-rajapintaan. Azure oli pilvipalveluna todella hyvä käyttää, sillä sen tarjoamat palvelut ja dokumentaatio oli hyvää. Azuren käyttöönotto opinnäytetyön alussa oli hieman työlästä, sillä Azure on niin laaja palvelu, että sen pääpiirteiden ymmärtäminen vaati aikaa. Azuren sijaan toteutuksessa olisi voinut käyttää myös Amazonin tarjoamaa AWS pilvipalvelua, mutta opinnäytetyössä päädyttiin käyttämään Azurea sillä se oli toimeksiantajalla jo käytössä.

Opinnäytetyön toteutuksessa ei tullut vastaan sen kummempia ongelmia, sillä siinä käytetty ohjelmointikieli oli tuttu. Suurin ongelma toteutuksessa oli Terraformin ajoon kuluvan ajan käsittely siten ettei muut osiot verkkosovelluksesta kärsisi, mutta ongelmaan löydettiin ratkaisu Elixirin sisältämän GenServer-osion avulla. Toinen huomattava vastaan tullut hidaste oli Terraform. Terraform oli täysin uusi teknologia, jota ei ollut aiemmin tullut vastaan, joten sen toiminta tuli opetella täysin opinnäytetyön toteutuksen aikana.

Lähteet

Christensen, W. 2015. How to Choose a Programming Language. treehouse. Blogiartikkeli. Viitattu: 11.10.2018. https://blog.teamtreehouse.com/choose-programming-language.

Controllers - Phoenix. N.d. Tekninen dokumentti hexdocsin sivustolla. Tekninen dokumentti Viitattu: 29.11.2018. https://hexdocs.pm/phoenix/controllers.html.

Create identity for Azure app in portal. 2018. Microsoft Docs. Tekninen dokumentti. Viitattu: 4.12.2018. https://docs.microsoft.com/fi-fi/azure/active-directory/develop/howto-create-service-principal-portal.

Ecto - Ecto. N.d. Tekninen dokumentti hexdocsin sivustolla. Viitattu: 4.12.2018. https://hexdocs.pm/ecto/Ecto.html.

Ecto.Changeset - Ecto. N.d. Tekninen dokumentti hexdocsin sivustolla. Viitattu: 4.12.2018. https://hexdocs.pm/ecto/Ecto.Changeset.html.

Ecto.Query - Ecto. N.d. Tekninen dokumentti hexdocsin sivustolla. Viitattu: 4.12.2018. https://hexdocs.pm/ecto/Ecto.Query.html.

Ecto.Repo - Ecto. N.d. Tekninen dokumentti hexdocsin sivustolla. Viitattu: 4.12.2018. https://hexdocs.pm/ecto/Ecto.Repo.html.

Ecto.Schema - Ecto. N.d. Tekninen dokumentti hexdocsin sivustolla. Viitattu: 4.12.2018. https://hexdocs.pm/ecto/Ecto.Schema.html.

EEx. N.d. Tekninen dokumentti hexdocsin sivustolla. Viitattu: 4.12.2018. https://hexdocs.pm/eex/EEx.html.

Elixir. N.d. Kehittäjän sivusto. Viitattu: 10.12.2018. https://elixir-lang.org/.

Endpoint - Phoenix. N.d. Tekninen dokumentti hexdocsin sivustolla. Viitattu: 29.11.2018. https://hexdocs.pm/phoenix/endpoint.html.

Functional programming - introduction. N.d. Opetusartikkeli tutorialspoint-sivustolla. Viitattu: 10.12.2018. https://www.tutorialspoint.com/functional_programming/functional_programming_introduction.htm.

GenServer - Elixir. N.d. Tekninen dokumentti hexdocsin sivustolla. Viitattu: 28.11.2018. https://hexdocs.pm/elixir/GenServer.html.

Getting Started. N.d. Tekninen dokumentti hexdocsin sivustolla. Viitattu: 25.10.2018. https://hexdocs.pm/ecto/getting-started.html.

Haikala, I ja Mikkonen, T. 2011. Ohjelmistotuotannon käytännöt. Helsinki : Talentum.

HTTPoison. N.d. Tekninen dokumentti hexdocsin sivustolla. Viitattu: 4.12.2018. https://hexdocs.pm/httpoison/HTTPoison.html.

Input Variables. N.d. Tekninen dokumentti terraformin sivustolla. Viitattu: 25.10.2018. https://www.terraform.io/intro/getting-started/variables.html.

Introduction to Mix. N.d. Elixir. Viitattu: 11.10.2018. https://elixir-lang.org/getting-started/mix-otp/introduction-to-mix.html.

Introduction to Terraform. N.d. Tekninen dokumentti terraformin sivustolla. Viitattu: 25.10.2018. https://www.terraform.io/intro/index.html.

Official Ruby FAQ. N.d. Ruby. Viitattu: 11.10.2018. https://www.ruby-lang.org/en/documentation/faq/7/.

Overview - Phoenix. N.d. Tekninen dokumentti hexdocsin sivustolla. Viitattu: 29.11.2018. https://hexdocs.pm/phoenix/overview.html.

PostgreSQL: About. N.d. PostgreSQL. Viitattu: 4.12.2018. https://www.postgresql.org/about/.

Recursion. N.d. Elixir. Viitattu: 10.12.2018. https://elixir-lang.org/getting-started/recursion.html.

Routing - Phoenix. N.d. Tekninen dokumentti hexdocsin sivustolla. Viitattu: 29.11.2018. https://hexdocs.pm/phoenix/routing.html.

Schuster, W. 2014. José Valim on the Elixir language, concurrency, iteration. InfoQ. 30.11.2014. Viitattu: 11.10.2018. https://www.infoq.com/interviews/valim-elixir.

Templates - Phoenix. N.d. Tekninen dokumentti hexdocsin sivustolla. Viitattu: 4.12.2018. https://hexdocs.pm/phoenix/templates.html.

Terraform Commands. N.d. Tekninen dokumentti terraformin sivustolla. Viitattu: 25.10.2018. https://www.terraform.io/docs/commands/index.html.

What is Azure? N.d. Microsoft Azure. Viitattu: 4.12.2018. https://azure.microsoft.com/en-us/overview/what-is-azure/.

Workspaces. N.d. Tekninen dokumentti terraformin sivustolla. Viitattu: 25.10.2018. https://www.terraform.io/docs/state/workspaces.html.