Jussi Latvaniemi

SaaS-palvelun paikallinen versio

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikan tutkinto-ohjelma

Insinöörityö

12.4.2018

Metropolia

Tekijä Otsikko Sivumäärä Aika	Jussi Latvaniemi SaaS-palvelun paikallinen versio 28 sivua 12.4.2018
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	tieto- ja viestintätekniikka
Ammatillinen pääaine	mediatekniikka
Ohjaajat	projektipäällikkö Guido Kohl yliopettaja Petri Vesikivi

Insinöörityö tehtiin suomalaiselle pienyritykselle, joka kehittää verkossa tehtäviä koulutuksia muille yrityksille. Yritys halusi tehdä SaaS-palvelun (Software as a Service) kehittämisestä helpompaa, nopeampaa ja vähemmän virhealtista. Ratkaisun täytyi myös sopia yrityksen SaaS-palveluun tulevaan kehitysideaan, joka on palvelimella toimivan SaaS-palvelun integroiminen GitHubiin, jotta siitä tulisi tehtyjen projektien ainoa lähde.

Aluksi työssä selvitettiin, miten yrityksen SaaS-palvelun voi asentaa sen kehittäjien paikallisille työasemille, jotta palvelun moduulipaketteja voidaan kehittää häiritsemättä muiden ohjelmoijien työtä. SaaS-palvelun asentamisesta kirjoitettiin mahdollisimman selvät opasteet yrityksen muille kehittäjille, jotta he voivat itsenäisesti asentaa ja käyttää paikallista SaaS-palvelua.

SaaS-palvelun paikallisen asentamisen jälkeen aloitettiin kehitystyössä pienten tehtävien automatisointi. Tehtävien automatisoinnissa käytettiin Node.js-sovellusta. Node.js valittiin automatisoinnin työkaluksi, koska se on ollut jo aiemmin käytössä yrityksen muissa projekteissa ja se on alustasta riippumaton. Tehdylle Node.js-sovellukselle voidaan antaa komentorivin kautta erilaisia käskyjä, jotka valmistelevat uuden moduulipaketin uutta projektia varten tai seuraavat jo olevan moduulipaketin tiedostoihin tehtyjä muutoksia ja lähettävät tehtyjen muutosten tiedot paikalliseen tietokantaan.

Node.js-sovelluksen kehittäminen ei ollut aivan suoraviivaista, vaan siihen jouduttiin kehityksen aikana lisäämään uusia ominaisuuksia. Uusien ominaisuuksien ideat olivat lähtöisin yrityksen muilta kehittäjiltä, jotka tekivät silloin SaaS-palvelun moduulipakettien kehitystyötä.

Valmis Node.js-sovellus ja paikallinen SaaS-palvelu otetaan osaksi SaaS-palvelun ja sen moduulipakettien kehitysprosessia, koska sen todettiin helpottavan moduulipakettien kehitystyötä automatisoimalla eri tehtäviä. Jos Node.js-sovelluksessa havaitaan käytön aikana parannusideoita, sitä voidaan tulevaisuudessa kehittää turvallisesti versionhallinnan kautta.

Avainsanat	Node.js, SaaS, tehtävien automatisointi, verkkosovelluskehitys

Author Title Number of Pages Date	Jussi Latvaniemi SaaS local version 28 pages 12 April 2018
Degree	Bachelor of Engineering
Degree Programme	Information and Communication Technology
Professional Major	Media Technology
Instructors	Guido Kohl, Project Manager Petri Vesikivi, Principal Lecturer

This final year project was done for a small Finnish company, which does e-learning courses for other the companies. The company wanted to make development of their SaaS cloud service easier, faster and more robust. The solution also had to fit in the company's SaaS future development plan, which was to integrate their SaaS (Software as a Service) with GitHub, so it would be the only source of the done projects.

In the thesis, the first thing was to find out how to install the SaaS to its developer's workstations, so the service's module packages could be developed without bothering other developers. Easily understandable instructions were written for the local SaaS-version's installation process for the company's developers, so they can install and use the local SaaS by themselves.

After the installation of the local SaaS began the development of automating small tasks when developing the SaaS module packages. Node.js was used to create the application for the automation of the tasks. Node.js was chosen as the automation tool because it was already in use in the company's other projects and it is platform independent. To the created Node.js application you can give command-line different arguments through the command-line to prepare a new module package for a new project or watch already existing project file changes and send the done changes to the local database.

The Node.js application's development was not quite straight forward, but during its development there was a need to add new features. Ideas for the new features came from the colleagues who were developing the SaaS module packages.

The finished Node.js application and the local SaaS will be added as a part of the current SaaS and its module package development process, because it made the development of module packages easier by automating tasks. If there are ideas for the improvement of the Node.js application during its use, those can be added safely using the version control system.

Keywords	Node.js, Saas, task automation, web-application development

Sisällys

Lyhenteet

CVS	*Cenralized Version Control System.* Keskitetty versionhallintajärjestelmä.
DVCS	*Distributed Version Control System.* Hajautettu versionhallintajärjestelmä, kuten esimerkiksi Git ja Mercurial.
HTML	*Hyper Text Markup Language.* Ohjelmointikieli, jolla tehdään verkkosivujen rakenne.
IaaS	*Infasructure as a Service.* Pilvipalvelu, jossa annetaan virtuaalinen tai fyysinen palvelin asiakkaan käytettäväksi.
NPM	*Node Package Manager.* Node.js-pakettien hallintaohjelma, jota käytetään komentorivin kautta.
PaaS	*Platform as a Service.* Pilvipalvelu, jossa tarjotaan asiakkaille palvelimelta tilaa asiakkaan omien sovellusten asentamista ja kehittämistä varten.
PHP	*Hypertext Preprocessor.* Apache-palvelimissa käytetty ohjelmointikieli.
SaaS	*Software as a Service.* Pilvipalvelu, jossa yritys tarjoaa asiakkaille pilvessä toimivaa sovellusta.
SFTP	*SSH File Transfer Protocol.* Mahdollistaa tiedostoihin pääsyn ja muokkaamisen SSH-yhteyden kautta.

1 Johdanto

Insinöörityö toteutettiin suomalaiselle Apprix Oy -pienyritykselle, joka tarjoaa koulutusohjelmia muille yrityksille. Erillisten koulutusohjelmien lisäksi Apprixilla on oma SaaS-palvelualusta, jossa asiakasyritykset pääsevät itse tekemään koulutuksia niille räätälöidyillä paketeilla. Apprixilla on kymmenisen työntekijää, joista yli puolet on ohjelmoijia. Yritys on tehnyt 826 000 euron liikevaihdon vuonna 2017 ja yrityksen liikevaihdon muutos oli 47 % edelliseen vuoteen verrattuna [1].

Apprix Oy:n SaaS-palvelusta on tullut asiakkaiden keskuudessa hyvin suosittu, ja SaaS-palvelua pyritään koko ajan kehittämään paremmaksi niin asiakkaille kuin sitä kehittäville Apprixin ohjelmoijillekin. Ohjelmoijat havaitsivat puutteita SaaS-palvelun pakettien kehitysprosessissa ja toivoivat siihen parannuksia. Suurimmat ongelmat kehitysprosessissa ovat versionhallinnan monimutkaisuus, koodien kopioiminen tiedostoista SaaS-palvelun tietokantaan ja uusien ominaisuuksien testaamisen hankaluus. Ongelmien ratkaisu ei pelkästään nopeuta ohjelmoijien tekemään kehitystyötä vähentämällä virheiden mahdollisuuksia ja turhaa työtä, se myös parantaa kehitystyön mielekkyyttä ohjelmoijien kesken.

Apprixille tehdyn insinöörityön tarkoituksena oli suunnitella yrityksen SaaS-palvelulle työkalu, jolla mahdollistettaisiin SaaS-palvelun asiakkaille tarkoitettujen moduulipakettien kehitys paikallisesti yrityksen ohjelmoijien omilla työasemilla. Työhön annettiin täysi vapaus päättää, mitä tekniikoita ja työkaluja käyttää, kunhan kaikki toimivat yrityksen ohjelmoijien työasemilla ongelmitta.

Aluksi täytyi selvittää, miten SaaS-palvelu ja sen tietokanta asennetaan paikallisesti ohjelmoijien omille työasemille. Asentamisen jälkeen pystyttiin paremmin suunnittelemaan tarvittavat työkalut työtehtävien automatisointiin. Työkaluksi päätettiin kehittää Node.js-sovelluksia, jotka käyttävät muun muassa tehtävien automatisointityökalulla Gulp.js:a. Gulp.js:llä pystytään automatisoimaan erilaisia projektien työvaiheita, esimerkiksi koodin ja kuvien optimoiminen ja testaus. Gulp.js valittiin projektissa käytettäväksi työkaluksi, koska nämä tekniikat ovat olleet yrityksessä jo aikaisemmin käytössä ja ne on todettu toimiviksi ja niillä oli mahdollista toteuttaa kaikki paikallisen versioon tarvittavat ominaisuudet.

2 Yrityksen lähtötilanne

Insinöörityön lähtötilanteessa yrityksen SaaS-palvelun pakettien kehittämiseen käytetyt koodit oli tallennettu useaan eri paikkaan ja niitä täytyi kopioida aika ajoin paikasta toiseen. Ensimmäisenä koodit olivat SaaS-palvelun paketin kehittäjän omalla tietokoneella ja paikalliset tiedostot ovat lisätty GitHub-versionhallintapalveluun. Paikallisesti pystyttiin kehittämään paketin perusrakenteet ja toiminnallisuudet, mutta ulkonäön viimeistely ja tietyt SaaS-palvelusta riippuvat toiminnallisuudet piti testata SaaS-palvelun testipuolella, joten koodit oli kopioitava paikallisista tiedostoista SaaS-palvelun tietokantaan, minkä jälkeen tarvittavat testaukset voitiin toteuttaa. SaaS-palvelun testipuolella tehtyjen testausten jälkeen koodit oli siirrettävä SaaS-palvelun tuotantopuolelle. Koodit voitiin siirtää tuotantopuolelle joko paikallisista tiedostoista tai SaaS-palvelun testipuolelta kopioimalla. Kopiointien jälkeen asiakkaat pääsivät ensimmäistä kertaa kokeilemaan SaaS-palvelun pakettiin tehtyjä muutoksia.

SaaS-palvelun koodien kopioimiseen ohjelmoijat toivoivat jonkinlaista ratkaisua, joka mahdollistaisi SaaS-palveluun tehtävien pakettien koodien kehittämisen pelkästään ohjelmoijien omilla työasemilla. Toiveena oli, että pakettien kehitystä voisi tehdä paikallisesti omalla työasemalla, jolloin vältytään erilaisilta ristiriidoilta, jos usea kehittäjä kehittää samaa toiminnallisuutta. Toiveena oli myös parempi yhteensopivuus GitHub-versionhallintapalvelun kanssa, koska se mahdollistaisi saman toiminnallisuuden kehittämisen ja jakamisen usean kehittäjän välillä.

Usein sovelluskehityksessä käytettävät koodit ovat vain paikallisesti kehittäjien työasemilla ja hajautetussa versionhallintajärjestelmässä kuten Gitissä. Nykyaikaiset verkkosovellukset osaavat jo itse hakea uudet päivitykset suoraan versionhallintajärjestelmästä, jolloin sovelluksen kehitykseen käytettävät tiedostot ovat vain kehittäjien omilla työasemilla ja versionhallintajärjestelmässä. Tämä oli myös tavoitteena Apprixin SaaS-palvelun moduulipakettien kehitysprosessissa. SaaS-palvelun moduulipakettien kehittäminen halutiin yksinkertaistaa niin, että moduulipakettien tiedostot ovat kehittäjien työasemalla ja ne siirretään komentorivin kautta versionhallintaan, minkä jälkeen voitaisiin SaaS-palvelussa yksinkertaisesti pyytää päivittämään moduulipaketti uusimpaan versioon.

Yrityksen SaaS-palvelun kehitysprosessi oli ohjelmoijien mielestä vanhanaikainen ja epäjohdonmukainen sekä altis monenlaisille virheille. Projekteja tehdessä oltiin huomattu, miten paljon erilaiset pienet virheet saattavat viedä paljon aikaa niitä korjattaessa.

Tällaiset pienet virheet, kuten toisten muutosten päälle kopioiminen, on moderneilla käytännöillä saatu lähes kokonaan estettyä käyttämällä versionhallintaa. Seuraavaksi selvitettiin, mitä kaikkia ongelmia ohjelmoijat olivat havainneet SaaS-palvelun kehitysprosessissa.

Koodien lisääminen SaaS-palveluun piti ohjelmoijien mielestä tehdä helpommaksi. Lähtötilanteessa kaikki koodit piti kopioida paikallisesti työasemilla kehitetyistä tiedostoista SaaS-palvelun omaan tietokantaan, jotta ne saataisiin tallennettua testattavaksi tai asiakkaiden käytettäväksi. Koodien kopiointi aiheutti turhan ajankäytön lisäksi myös ongelman, jossa tehtyjä töitä katosi SaaS-palvelusta. Jos kehittäjiä olisi vain yksi jokaisessa projektissa, tätä ongelmaa ei olisi, mutta jokainen projektiin lisätty kehittäjä teki tehtävien jaosta yhä hankalampaa, ja lopulta kehittäjien määrää piti rajoittaa, jotta vältyttiin kaaokselta. Projekteissa, joissa oli useampi kehittäjä, jouduttiin tehtävät jakamaan SaaS-palvelun paketin eri osiin. Esimerkiksi kaksi ohjelmoijaa tekivät samaan aikaan kahta eri HTML-sivua (Hyper Text Markup Language). Samaa HTML-sivua ei pysty järkevästi kehittämään kaksi ohjelmoijaa ja testaamaan omia muutoksia yhtä aikaa, koska jos ensimmäinen ohjelmoija tekee muutoksia ja tallentaa ne testausta varten SaaS-palveluun, seuraavan ohjelmoijan tekemät muutokset tallentuvat ensimmäisen ohjelmoijan muutosten päälle, jolloin hänen tekemänsä muutokset katoavat. Ongelma ei ollut kuitenkaan niin paha, että tehty työ olisi kokonaan kadonnut, koska kummankin ohjelmoijan koodit ovat kuitenkin heidän omilla tietokoneillaan ja kummankin tekemät muutokset voitiin yhdistää jälkeenpäin versionhallinnassa, mutta toisten työn päälle tallentaminen teki saman paketin HTML-sivun testaamisesta hankalaa.

Koodien kopioinnin aiheuttamien töiden menetyksen lisäksi se kuluttaa turhaa aikaa. Ohjelmoijat halusivat, että koodit päivittyisivät automaattisesti SaaS-palveluun, jotta vältyttäisiin koodien kopioimiselta ja kehitettäviä paketteja pystyttäisiin testaamaan nopeammin SaaS-palvelussa. Automatisointi voisi myös samalla hoitaa koodien kokoamisen, pienentämisen ja muuttamisen vanhoille selaimille sopiviksi. Koodin lisäksi SaaS-palvelun paketissa käytettävät kuvat voisi samalla tavalla automaattisesti pakata pienempään kokoon.

SaaS-palvelun moduulipakettien versionhallinta ei toiminut toivotulla tavalla. SaaS-palvelun moduulipakettien koodit olivat GitHub-nimisessä versionhallintajärjestelmässä, mutta SaaS-palvelussa olevat koodit eivät päivittyneet sitä mukaan, kun versionhallinnassa oleviin koodeihin tehtiin muutoksia. Ohjelmoijat halusivat helpotusta tähän ongel-

maan. He halusivat, että GitHub-versionhallintajärjestelmä saataisiin paremmin yhdistettyä SaaS-palveluun. Tätä kehitystyötä ei kuulunut omaan tehtävänantooni muuten kuin siinä mielessä, että kehittämäni työkalun kuuluisi myös toimia kätevästi GitHub-versionhallinnan kanssa. Paikallisesti kehitettyjen ja testattujen pakettien pitäisi olla GitHub-versiohallinnassa, josta muutetut paketit voisi suoraan päivittää myös palvelimella olevaan SaaS-palvelun tuotanto- tai testipuolelle.

3 Verkkosovellukset ja niiden kehitys

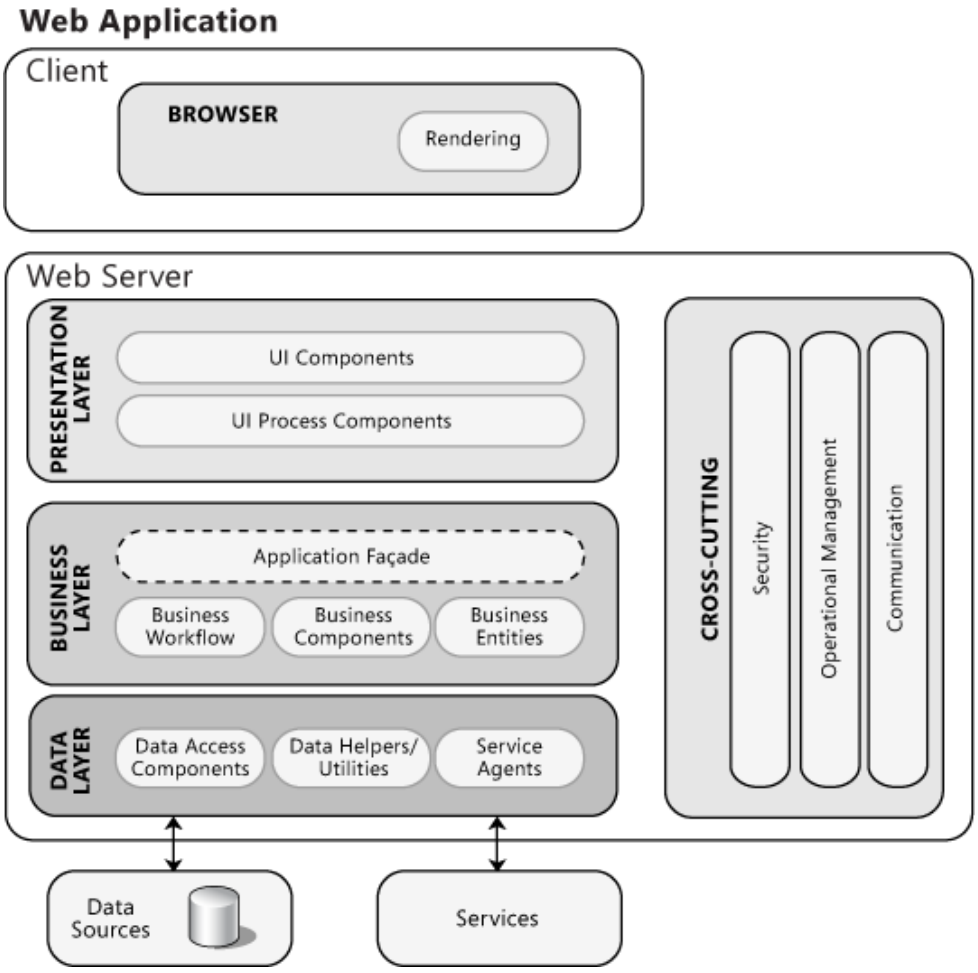
Ensin käydään läpi, millaisia nykyaikaiset verkkosovellukset ovat ja miten niitä kehitetään. Samalla vertaillaan hieman, miten verkkosovellukset ja niiden kehitystyö eroavat perinteisten sovellusten kehittämisestä. Lopuksi tarkennetaan verkkosovellusten kehitykseen käytettyjen työkalujen tarkempia tietoja ja sitä, mitkä niistä valittiin tämän insinöörityön suorittamista varten.

Perinteisten sovellusten kehityksessä aluksi tehtiin suuria sovelluksia, joiden lähdekoodin ylläpito kävi hankalaksi, mitä pidemmälle projektissa edettiin [2, s. 4]. Verkkosovellusten kehityksessä oli aluksi samanlainen ongelma. Esimerkiksi verkkosovelluskehityksen alkuvaiheissa sovelluksen toiminnallisuuksiin oli käytetty PHP-ohjelmointikieltä palvelimella, ja se oli kirjoitettu HTML-koodin sekaan, jolloin koodi oli jakautunut moneen tiedostoon ja sitä oli hankalaa ylläpitää. Tästä käytäntötavasta siirryttiin pois, kun keksittiin laittaa esimerkiksi tietokantaan vaikuttavat koodit omiin tiedostoihinsa ja laitettiin vain verkkosivun ulkonäköön vaikuttavat koodit HTML-koodin sekaan [2, s. 4]. Näin verkkosovelluksen kehityksestä tuli johdonmukaisempaa.

Modernit verkkosivut ovat muuttuneet yhä lähemmin vastaamaan perinteisiä sovelluksia. Useista staattisista verkkosivuista on tullut niin sanottuja verkkosovelluksia, joilla on samoja ominaisuuksia kuin perinteisillä sovelluksilla, mutta ne käyttävät käyttöliittymänään selainta. Verkkosovellukset ovat kehittyneet jo niin kattaviksi, että perinteisten sovellusten pohjalta on kehitetty vastaava verkkosovellus. Esimerkiksi kalenterit, sähköpostit, muistiot, kuvat ja albumit ovat saaneet omat verkkosovellusversiot [2, s. 6]. Yritykset ovat huomanneet verkkosovellusten kehityksen hyödyt verrattuna perinteisiin sovelluksiin. Verkkosovellukset voidaan julkaista vain yhdelle keskitetylle palvelimelle, jolloin vältytään sovelluksen asiakkaille jakamisen ongelmalta. Verkkosovellusten päivittäminen ja uudistaminen on myös perinteisiä sovelluksia paljon helpompaa ja halvempaa, koska

kaikki muutokset tehdään suoraan palvelimella olevaan sovellukseen ja muutokset näkyvät heti kaikille sovelluksen käyttäjille. [2, s. 6.]

Verkkosovellusten toiminnallisuus on jaettu useaan eri osaan. Verkkosovellusten käyttäjälle näkyvä toiminnallisuus tapahtuu käyttäjän omassa selaimessa [2, s. 4]. Nämä toiminnallisuudet ovat usein melko keveitä, koska selainympäristö käyttää JavaScript-ohjelmointikieltä toiminnallisuuksien lisäämiseen. JavaScript tukee selaimessa vain yhtä prosessia kerrallaan, jolloin raskaiden prosessien läpikäyminen selaimessa ei kannata. Lisäksi prosessin selaimessa prosessin suorittamiseen vaikuttaa käyttäjän oma päätelaite. Käyttäjillä ei välttämättä ole kovinkaan tehokasta päätelaitetta, jolloin raskaiden prosessien suorittaminen palvelimella on parempi vaihtoehto. Verkkosovellusten prosessointi siis suoritetaan pääasiassa palvelimella [2, s. 4]. Palvelin käsittelee tietokannasta saadut tiedot ja palauttaa ne käsittelyn jälkeen selaimelle. Kuvassa 1 näytetään tarkemmin, miten verkkosovelluksen toiminta on tyypillisesti jaoteltu.



Kuva 1. Tyypillinen verkkosovelluksen arkkitehtuuri [3].

Verkkosovellusten kehittäminen on myös saanut samoja piirteitä kuin perinteisten sovellusten ohjelmoiminen [4, s. 391]. Verkkosovellusten koodit usein testataan, kootaan ja pienennetään ennen tuotantoon siirtämistä, mikä vastaa hieman perinteisessä sovelluskehityksessä sovelluksen kokoamista. Usein lopputuloksena on hyvin hankalasti luettavaa tietokoneen automaattisesti kokoamaa koodia. Koodien kokoamisen tarkoitus on pienentää koodien kokoa, koska sivuille tulevat käyttäjät joutuvat lataamaan ne, mikä voi olla ajoittain hidasta.

Kuvien ja koodien pakkaaminen suoritetaan käyttäen paikallista Node.js-sovellusta. Node.js-sovellus pystyy automaattisesti suorittamaan tarvittavat toimenpiteet tiedostoja lisättäessä tai muutettaessa [4, s. 392.]. Kootut tiedostot kopioidaan kansioon, jossa ovat kaikki verkkosovelluksen tuotantoon valmiit tiedostot. Nämä tiedostot voidaan laittaa palvelimelle joko käsin tai palvelin voi ladata päivitetyn version suoraan versionhallintajärjestelmään, kun uusi versio on julkaistu. Parhaiten tunnettu ja käytetty versionhallintajärjestelmä on GitHub, jonne voi lisätä julkisia tai yksityisiä projekteja versionhallintaan.

Seuraavaksi selvitetään muutamia tärkeimpiä verkkokehityksessä käytettyjä työkaluja. Verkkokehitystyöhön on tehty lukuisia työkaluja ja joihinkin tehtäviin on tehty useampia eri versioita, jotka pyrkivät tekemään saman asian, mutta edellistä työkalua paremmin. Seuraavaksi myös kerrotaan, miten samanlaisia tehtäviä tekevät työkalut eroavat ja onko jokin työkaluista selvästi muita parempi.

3.1 Verkkosovelluskehityksessä käytetyt työkalut

Node.js

Node.js on alustasta riippumaton JavaScript-ohjelmointikielen tulkitsija. Se on kehitetty Googlen kehittämän V8-JavaScript-virtuaalikoneen ympärille. Node.js muokkaa V8-virtuaalikonetta sopimaan paremmin selaimen ulkopuolisiin ympäristöihin. Se mahdollistaa JavaScriptin suorittamiseen selainympäristön ulkopuolella, jolloin erilaisten sovellusten, kuten verkkokehitystyön tehtävien automatisointisovelluksia ja varsinkin palvelinten tekeminen on mahdollista. [5, s. 3.]

Node.js:n arkkitehtuuri on niin sanottu tapahtumasilmukka (Event-Loop) [5, s. 3]. Node.js tukee tapahtumia ohjelmointikielen tasolla, jolloin tapahtumia varten ei tarvitse ladata

erillistä kirjastoa kuten useimmilla muilla ohjelmointikielillä. Tapahtumasilmukka on perusosa JavaScript-ohjelmointikieltä, koska se on suunniteltu reagoimaan käyttäjien antaman vuorovaikutuksen kanssa [5, s. 33]. Kun käyttäjä antaa jonkin komennon, kuten klikkaa napista, siitä lisätään tapahtuma tapahtumasilmukkaan, josta se käsitellään seuraavalla silmukan läpikäynnillä.

Vaikka Node.js voi käyttää vain yhtä säiettä, se ei kuitenkaan tarkoita sitä, että sillä ei voisi suorittaa samanaikaisesti tehtäviä. Node.js käyttää niin sanottuja ei-sulkevia kirjastoja, eli näiden kirjastojen toiminnot eivät jää odottamaan esimerkiksi kiintolevyltä hitaasti saatavia tietoja, vaan ne suorittavat toisia tehtäviä samaan aikaa, kun järjestelmän kiintolevy palauttaa haluttuja tietoja. [5, s. 4.] Näin Node.js:n käyttämä säie ei jää toimettomaksi vaan käyttää aikansa mahdollisimman tehokkaasti.

Node.js:n kasvava suosio perustuu sen tapahtumasilmukka-arkkitehtuuriin, koska sillä saa tehtyä helposti tehokkaita palvelinsovelluksia [6, s. 1]. Perinteisissä sovelluksissa käytetään samanaikaisten tehtävien suorittamiseen useita säikeitä. Säikeillä ohjelmointi mielletään usein hankalaksi, koska säikeiden mukana tulee omat ongelmansa. Suurimmat haasteet ovat muistin jakaminen eri säikeiden välillä ja niin sanotut lukkiutumatilanteet (deadlock), jolloin prosessit odottavat toinen toisensa valmistumista, jotta suoritusta voitaisi jatkaa, mutta kumpikaan prosessi ei koskaan valmistu. [7, s. 1.]

Node.js-moduulien hallintatyökalut

Node.js on helposti laajennettavissa NPM:n (Node Package Manager) avulla. NPM on Node.js-moduulien hallintaohjelma. Se on yhteydessä NPM-rekisteriin, johon on listattu useita Node.js-kirjastoja. [8, s. 43.] Julkiset kirjastot ovat kaikkien ladattavissa ja käytettävissä, mikä tekee uusien ominaisuuksien lisäämisestä Node.js-sovelluksiin helppoa. Moduulipakettien lataaminen onnistuu asentamalla Node.js, jonka mukana tulee myös NPM. Asennuksen jälkeen voidaan komentoriviin antaa komento

NPM-kirjastot ovat NPM-yhteisön lisäämiä ja ylläpitämiä Node.js-moduulipaketteja [5, s. 4]. Pakettien koko saattaa vaihdella hyvinkin yksinkertaisen toiminnallisuuden lisäämisestä kokonaisiin kirjastoihin. Ongelmana yhteisön luomissa paketeissa on se, että niitä ei välttämättä yllä pidetä tarpeeksi, jolloin pakettiin saattaa muodostua tietoturvariskejä. Pakettien yksi huonoista puolista on myös se, että ne ovat jonkun muun tekemää koodia. NPM-pakettien toimintaa saattaa olla vaikea selvittää, jos ne on pienennetty ja sotkettu lukukelvottomaksi.

Yhteisön lisäämät paketit mahdollistavat haitallisten pakettien jakamisen ja käyttämisen. Projektissa tuleekin miettiä tarkkaan, tarvitaanko siinä juuri NPM-pakettia, jos se on jokseenkin tuntematon vai voiko saman toiminnon kehittää itse sopivassa ajassa. Ainoa tapa selvittää, ovatko paketit turvallisia vai ei, on käymällä niiden koodit itse läpi ja selvittämällä, tekevätkö ne vain niitä asioita, joita niiden on luvattu tekevän. NPM-pakettien koodeja ei kannata lisätä mihinkään sovelluksen kohtaan, jossa käsitellään arkaluontoista tietoa [9]. Näin varmistutaan siitä, että ohjelma toimii oikein eikä kukaan pääse tärkeisiin tietoihin käsiksi.

NPM-pakettien hallintaohjelman tilalle on myös Google, Facebook, Exponent ja Tilde yhdessä kehittäneet vastaavan komentorivikäyttöliittymän Yarnin. Se on kehitetty korjaamaan NPM-pakettien hallintaohjelman vikoja. Se ei kuitenkaan ole kokonaan oma sovelluksensa, koska se edelleen käyttää NPM-rekisteriä pakettien hakuun. Yarn asentaa paketit NPM:ää nopeammin käyttämällä välimuistia (cache) pakettien asentamisessa, jotta käyttäjän ei tarvitse turhaan ladata samaa pakettia useampaan kertaan, vaan Yarn kopioi saman paketin suoraan välimuistista uusiin projekteihin. [10.]

Tehtävien automatisointityökalut

Pitkän aikaa verkkokehitystyötä on tehty hyvin manuaalisesti. Verkkosivuissa käytettyjä koodeja ja kirjastoja on kopioitu projektista toiseen ja verkkosivujen testaukset on suoritettu käsin. Joitain työkaluja on ollut, jotka ovat tehneet koodin tarkastuksia ja muita tehtäviä, mutta nämä työkalut on pitänyt suorittaa aina käsin. Verkkosivujen kehittäjät saattavat kuitenkin unohtaa käyttää näitä työkaluja, jolloin tulee lisää ongelmia. [4, s. 391.]

Verkkokehityksessä on otettu mallia perinteisten sovellusten kehityksestä. Perinteisissä sovelluksissa koodit on ensin käännettävä tietokoneen ymmärtämäksi konekieleksi, jotta ne voidaan suorittaa. Tämän kääntöprosessin aikana koodin laatu tarkistetaan ja koodikannan toiminnoille suoritetaan yksikkötestejä [4, s. 391]. Projektin muillekin tiedostoille saatetaan tehdä joitain tehtäviä, kuten kuvien pakkaamista tai koodien yhdistelyä ja pienennystä. Samantyyppistä kokoamisprosessia on pyritty myös tuomaan verkkokehityksen puolelle. Verkkokehityksessä on pyritty automatisoimaan kaikki tehtävät, jotka tarvitaan tuotantoon kelpaavan sovelluksen luomiseen.

Tällaisia tehtävien automatisointityökaluja on tehty jo muille ohjelmointikielille. Esimerkiksi Ant, Gradle, Rake, Make ja Maven automatisoivat projektin rakennusprosessin eri

vaiheita. Ne voivat testata ja kääntää koodia sekä automaattisesti kirjoittaa dokumentaatiot koodista. [4, s. 391.] JavaScriptillekin on tullut vastaavia työkaluja, kuten Grunt ja Gulp. Ne tekevät samaan tyyliin tehtävien automatisointia kuin muiden ohjelmointikielten automatisointityökalut, ja niillä on hyvin aktiiviset yhteisöt kehittämässä uusia ominaisuuksia. Seuraavaksi tarkastellaan tarkemmin, minkälaisia tehtäviä Grunt ja Gulp automatisoivat verkkokehityksessä ja miten ne eroavat toisistaan.

Gruntilla pystytään tekemään samanlaiset asiat kuin perinteisten sovellusten rakennustyökaluilla, eli sillä voi tarkastaa koodin laadun, yksikkötestata sovelluksen toiminnallisuudet, koota ja pienentää JavaScript-koodit, pakata kaikki projektissa käytetyt kuvat ja eri kehyskirjastojen tarvitsemia tehtäviä. Kokoamisprosessin jälkeen verkkosovellus on tuotantoon valmis, ja sen jälkeen se usein julkaistaan palvelimelle, josta asiakkaat tai käyttäjät pääsevät käyttämään sitä [4, s. 392].

Gulp on nopeasti saanut suosiota verkkokehittäjien joukossa sen tehokkuuden vuoksi. Gulp tarjoaa samoja ominaisuuksia kuin Grunt, mutta se pystyy suorittamaan tehtäviä asynkronisesti, mikä nopeuttaa useiden tehtävien suorittamista. Gulp käyttää Node.js:n ”stream”- eli tietovirta- ja ”pipe”- eli putkitusominaisuuksia tehtävien suorittamisessa [4, s. 401]. Näin Gulpin ei tarvitse kirjoittaa tilapäisiä tiedostoja, vaan tehtävissä voidaan käyttää samaa tietovirtaa useassa eri pienessä tehtävässä. Koodin suorittamisesta tulee tehokkaampaa ja usein myös nopeampaa. Gulpin tehtävät on suunniteltu olemaan hyvin pieniä. Ne tekevät korkeintaan yhden asian hyvin [4, s. 401]. Isompia tehtäviä saadaan tehtyä yhdistelemällä näitä pieniä tehtäviä toisiinsa. Esimerkkikoodissa 1 kuvataan, miten ”return gulp” jälkeen rakennetaan yksi isompi kuvan optimointitehtävä aivan pienistä osista. Tämän Gulptehtävän tarkoitus on tehdä projektiin lisätyistä kuvista tiedostokooltaan pienempiä ja tallentaa pienennetyt kuvat tuotantoon siirrettävään kansioon.

```
gulp.task('images', () => {
  if (argv.noimg) {
    return null;
  }
  return gulp
    .src(IMG_ENTRY)
    .pipe(plumber())
    .pipe(gulpif(!argvproduction, changed(IMG_DEST)))
    .pipe(gulpif(
      argv.production,
      imagemin({
        verbose: true,
      })))
    .pipe(plumber.stop())
    .pipe(gulp.dest(IMG_DEST))
    .on('error', gutil.log);
  });
```

Esimerkkikoodi 1. Gulpilla tehty kuvan tiedostokoon pienennystehtävä. Koodista nähdään, miten Gulpin yksittäinen tehtävä koostetaan useasta pienestä tehtävästä.

Verkkosovellusten testaus

Perinteisten sovellusten testauksessa käydään läpi kaikki sovelluksen osa-alueet kaikista pienimmästä yksiköstä aina kokonaisuuden testaukseen asti [11, s. 7]. Näitä testauskäytäntöjä on pyritty ottamaan käyttöön myös verkkosovellusten kehitykseen. Varsinkin suurten verkkosovellusten testaamisesta on tullut tärkeää, ja monet ohjelmoijat väittävät sen olevan pakollista, eikä verkkosovelluksen kehittäjällä pitäisi olla mitään tekosyytä jättää testauksia pois. Hyvin tehdyt testit helpottavat suurten sovellusten kehittämistä, koska testien avulla pystytään heti toimintoja muuttaessa huomaamaan, mihin kaikkiin muihin toimintoihin muutokset voivat vaikuttaa. Näin ei tule kehittäjille yllätyksenä, että jokin vähän käytetty toiminnallisuus on rikkoutunut muutoksia tehtäessä. Testeillä myös varmistutaan siitä, että asiakkaalle lähetetty sovellus tekee asiat kuten pitääkin. Manuaalisesti testaamalla saattaa tulla erilaisia inhimillisiä virheitä ja testaajat saattavat unohtaa testata joitain ominaisuuksia, jolloin tuotantoon viety sovellus ei välttämättä toimikaan niin kuin pitäisi.

Verkkosovelluksen yksikkötestaus on niin hyvä kuin testien tekijä on. Jos testien tekijä ei käy kaikkia mahdollisuuksia testien aikana läpi tai unohtaa päivittää testit, testien hyödyt jäävät pieniksi. Huonojen testien lisääminen saattaa jopa viedä enemmän aikaa kuin testien tekemättä jättäminen, koska testien kirjoittamiseen käytetty aika ei poista sitä mahdollisuutta, että kehittäjä joutuu etsimään ja korjaamaan ongelman puutteellisten testien takia.

Versionhallinta

Versionhallintaa on tehty jo pitkään erilaisia tekniikoita käyttäen. Yleisin tiedostojen ja projektien versionhallintatekniikka on ollut yksinkertaisesti kopioida tiedosto tai projekti toiseen kansioon. Tällaisessa versionhallinnassa on paljon ongelmia, ja siksi versionhallintaan on kehitetty ajan myötä aina parempia ratkaisuja. Aluksi versionhallintajärjestelmät toimivat pelkästään paikallisesti. Tämän järjestelmän heikkoutena oli sen taipumattomuus usean ohjelmoijan kesken tehtyyn yhteistyöhön. [12.] Jos jokaisella on oma paikallinen versio, jossa ei ole muiden tekemiä muutoksia, on mahdollista, että tietoa häviää tai tiedostoissa syntyy ristiriitaisuuksia.

Paikallisista versionhallintajärjestelmistä siirryttiin palvelimella toimiviin keskitettyihin versionhallintajärjestelmiin (CVCS). Keskitetyssä versionhallinnassa oli paljon hyviä puolia, ja se oli usean vuoden standardi versionhallinnan käytäntö. Keskitetty versionhallinta mahdollisti seurata, mitä kukin projektin jäsen oli tekemässä, ja järjestelmänvalvoja saattoi myös määrätä, mihin kenelläkin oli oikeus tehdä muutoksia. Keskitetyn versionhallinnan huonona puolena oli sen haavoittuvaisuus, jos palvelimella jotain meni vikaan. [12.] Palvelimelta kadonneen projektin historiaa on vaikeaa palauttaa ilman varmuuskopiota, jotka saattavat muutenkin olla jopa useita päiviä vanhoja.

Palvelimelle keskitetystä versionhallinnasta siirryttiin hajautettuihin versionhallintajärjestelmiin (DVCS), joiden avulla pyrittiin korjaamaan keskitetyn järjestelmän heikkouksia. Hajautetussa versionhallinnassa palvelimella oleva versiohistoria on hajautettu kaikkien käyttäjien kesken. Jos palvelimella olevalle versiohistorialle tapahtuu jotain, se voidaan palauttaa helposti peilaamalla kenen tahansa käyttäjän versiohistoria. Tällaisia hajautettuja versionhallintajärjestelmiä ovat esimerkiksi Git, Mercurial, Bazaar ja Darcs. [12.] Git on yksi suosituimmista versionhallintajärjestelmistä nykyään, mutta esimerkiksi Mozilla Firefox-selainkehityksessä käytetään Mercurialia [13].

3.2 Pilvipalvelumuodot

IaaS-palvelut (Infrastructure as a Service) ovat eräänlainen itsepalvelumalli, jossa käyttäjille tarjotaan kaikki pilvipalvelun luomiseen tarvittavat palvelimet, tallennustila ja verkkoyhteydet. Palvelimet voivat olla joko virtuaalisia tai kokonaan oikeita palvelimia. Käyttäjät saavat siis IaaS-palveluista ulkoisen palvelintilan, johon he voivat asentaa oman palvelimensa ja siihen mahdolliset pilvipalvelusovellukset. IaaS-palveluista asiakkaat

saavat kaikki perinteisten palvelinten ominaisuudet, mutta ne ovat tarpeen mukaan skaalattavia. Esimerkiksi juhlapyhien aikaan asiakas voi tarvita palvelimelleen enemmän resursseja, jotta palveluiden toiminta on taattu myös ruuhka-aikoina. IaaS-palvelut ovat myös edullisia asiakkaille, koska heidän ei tarvitse sijoittaa palvelimiin ja palvelinkeskuksen vaatimiin tiloihin. Asiakkaat voivat vain vuokrata tarvitsemansa infrastruktuurin ja rakentaa oman palvelun sen päälle.

PaaS-palvelut (Platform as a Service) antavat asiakkaille mahdollisuuden asentaa omia sovelluksia pilvipalvelualustalle. PaaS-palvelut eroavat IaaS-palveluista sen verran, että asiakkaiden ei tarvitse välittää sovelluksensa käyttämän alustan ylläpidosta. Palvelun asiakkaat voivat siis keskittyä omien sovellustensa kehittämiseen, testaamiseen ja julkaisemiseen. PaaS-palvelussa sovellusten kehittäminen on nopeaa, yksinkertaista ja edullista asiakkaalle, koska kaikki sovelluksen tarvitsemat palvelimet, tallennustila ja käyttöjärjestelmät tarjoaa ja ylläpitää PaaS-palvelun tarjoava yritys.

SaaS-palvelut ovat yritysten tarjoamia pilvipalveluita, joissa palveluina on sovelluksia. ”SaaS” on lyhenne sanoista ”Software as a Service” eli sovellukset palveluna. Niiden käyttöliittymä toimii selaimeen tehdyn verkkosivukäyttöliittymän kautta. SaaS-sovelluksen käyttäjien ei tarvitse asentaa sovellusta omille laitteilleen, ja SaaS-palvelut ovat pilvessä kaikkien saatavilla. SaaS-palvelua voidaan siis vuokrata sen käyttöön tarvituksi ajaksi, jolloin käyttäjällä on oikeus käyttää verkossa olevaa sovellusta. [14.] Vuokra-ajan päätteeksi oikeudet sovellukseen puretaan ja käyttäjiltä evätään pääsy sovellukseen ja sovellukseen käyttämään tietokantaan. Käyttäjä voi myös määrittää, mitä osaa SaaS-palvelusta hän haluaa vuokrata käyttöönsä. Käyttäjien ei tarvitse ostaa koko sovelluksen kaikkia ominaisuuksia.

SaaS-palvelut voivat olla edullisia niin käyttäjille kuin palvelun tarjoajille. Käyttäjille halpuus usein ilmenee SaaS-palvelun vuokrauksessa. Palvelua voi vuokrata vain siksi aikaa, kuin sitä tarvitaan, eikä käyttäjän tarvitse ostaa koko sovellusta omille työasemilleen. Toinen hyöty käyttäjille on SaaS-palveluiden helppo saatavuus. Käyttäjät menevät vain SaaS-palvelun verkkosivuille ja aloittavat sovelluksen käytön. Itse palveluntarjoajille SaaS-palvelun ylläpito on perinteistä sovellusta helpompaa, koska SaaS-palvelu toimii keskitetysti, jollain palvelimella eikä käyttäjien omilla koneilla. Näin ollen palveluntarjoajien tarvitsee vain varmistaa, että keskitetty sovellus toimii niin kuin pitääkin.

SaaS-palveluiden huono puoli käyttäjille on luottamuksen tarve, että palveluun pääsyä ei lakkauteta tai palveluun ei tule mitään huomattavaa muutosta [14]. Jos palvelun tarjoaja estää pääsyn sovellukseen yllättäen, saattaa käyttäjä menettää tehdyt työt, jotka on tallennettu palveluntarjoajan palvelimelle eikä käyttäjä voi jatkaa sovelluksen käyttöä. Toinen luottamusta tarvitseva seikka on SaaS-palvelun tietoturva. Koska SaaS-palveluun pääsee käsiksi suoraan verkon kautta, on hyvin tärkeää, että sovellus on tietoturvallinen.

Useat isot yritykset tarjoavat omia SaaS-palveluita, kuten Microsoft, Adobe, Amazon ja Google. Microsoft ja Adobe tarjoavat omien perinteisten sovellustensa lisäksi myös niiden pilvipalveluita. [15.] Microsoft, Google ja Dropbox esimerkiksi kilpailevat pilvitallennustilan tarjoamisella ja Google kilpailee osittain Microsoftin kanssa pilvessä toimivien toimistotyökalujen tarjoamisesta.

Apprixin SaaS-palvelun idea on tarjota sovelluksen käyttäjille koulutusten tekoa varten sovellus, jolla asiakkaat pystyvät kehittämään omia näyttäviä koulutuksia helposti. Käyttäjät voivat samalla työkalulla luoda useaan eri aiheeseen liittyvän koulutuksen, jotka on kehitetty vastaamaan koulutuksen kävijöiden oppimistarpeita. Yksi hyvä puoli verkossa olevissa koulutuksissa on se, että koulutukset voidaan suorittaa mistä vain, kunhan käyttäjällä on pääsy verkkoon. Verkossa olevien koulutusten toinen hyvä puoli on, että niiden käyttöliittymänä toimivat selaimet, joten koulutuksen tekoon käytettyjen laitteiden käyttöjärjestelmästä ei tule ongelmaa, vaan koulutukset voidaan suorittaa esimerkiksi puhelimella.

4 Projektin vaiheet

4.1 Projektin alkuvalmistelut

Ensimmäiseksi insinöörityönä tehdyssä projektissa varmistettiin, että kaikki ymmärsivät, mitä ollaan tekemässä, joten tein palaveriin liittyen selonteon projektiin liittyvistä asioista. Selonteossa listattiin esimerkiksi, mikä oli senhetkinen tilanne, mitä ongelmia siinä tilanteessa oli, millä tavoilla ne voisi ratkaista ja mahdolliset ongelmakohdat. Kun palaverin selonteko oli hyväksytty, pystyttiin insinöörityön tekeminen aloittamaan.

Ennen projektin aloittamista kehitettiin erilaisia käyttäjätarinoita SaaS-palvelun moduulipakettien kehityksestä. Alussa kerättyjen käyttäjätarinoiden pohjalta pystyi paremmin miettimään, mitä ominaisuuksia sovelluksessa pitäisi olla ja minkälaisia ongelmia sen tulisi ratkaista. Suunnittelussa käytetyt käyttäjätarinat olivat seuraavat:

- SaaS-palvelun kehittäjä haluaa muokata jo olemassa olevaa elementtiä, mutta hän ei ole varma, onko versionhallinnassa kaikista uusin versio vai onko joku tehnyt suoraan palvelimelle niin sanottuja pikakorjauksia, joten SaaS-palvelun kehittäjä kopioi palvelimella olevan ohjelman ja tarkistaa, onko siinä jotain eroavaisuuksia verrattuna versionhallinnassa olevaan tiedostoon. Kopioinnin jälkeen SaaS-palvelun kehittäjä varmistuu siitä, että versionhallinnassa on uusin versio. Hän tekee ohjelmaan muokkaukset, kopioi sen palvelimelle testattavaksi ja tallentaa sen versionhallintaan.
- SaaS-palvelun kehittäjä tekee omia muutoksia SaaS-palvelussa olevaan ohjelmaan. Hieman muutosten tallentamisen jälkeen SaaS-palvelun kehittäjä huomaakin, että hänen tekemänsä muutokset ovat hävinneet palvelimella olevasta versiosta, koska joku toinen kehittäjä on tehnyt omia muutoksia ja tallentanut ensimmäisen kehittäjän muutosten päälle.
- SaaS-palvelun kehittäjä aikoo tehdä ohjelmaan uusia muutoksia. Ensin hän tarkistaa, mitä hänen tiimissään oleva kehittäjä on juuri silloin tekemässä ja vaikuttaako se hänen kehitystyöhönsä ja pitääkö hänen varoa jotain, ettei sotke toisen kehittäjän työntekoa.
- SaaS-palvelun kehittäjä aikoo luoda uuden moduulin palveluun. Hän kopioi vastaavan projektin tai alkaa tehdä projektia moduulipohjaa käyttäen. Hän sitten kopioi vastaavia tiedostoja muista projekteista uuteen projektiin.
- SaaS-palvelun kehittäjä aikoo luoda uuden projektin. Hän voi joko ladata Githubista vastaavan projektin ja viedä sen suoraan paikallisen SaaS-palvelun tietokantaan kirjoittamalla yhden komennon komentoriviin, minkä jälkeen SaaS-palvelun kehittäjä voi käynnistää tiedostojen seurannan, joka tallentaa tehdyt muutokset SaaS-palvelun tietokantaan tai siirtää staattiset tiedostot SaaS-palvelun käytettäväksi. Muutosten teon ja testausten jälkeen muutokset lisätään versionhallintaan ja SaaS-palvelun palvelimella olevaan versioon tuodaan tehdyt muutokset GitHubin kautta.

Viimeinen käyttäjätarina ei ole senhetkiseen tilanteeseen liittyvä. Se on tavoite, johon yrityksen kehitysprojektit pyrkivät. Tarkoituksena on tehdä sovellusten tuotantoprosessista tehokkaampaa. Tässä viimeisessä käyttäjätarinassa on yrityksen tulevien projektien vaatimuksia, jotka täytyi ottaa huomioon insinöörityöprojektini suunnitelmia tehdessä. Näin ollen sovelluksesta tuli tehdä mahdollisimman helposti muokattava ilman, että sovelluksen arkkitehtuuria tulisi suuremmin muuttaa, koska se mahdollistaa helpomman yhdistämisen muihin sovelluksiin tulevaisuudessa.

Kun kaikki käyttäjätarinat oli selvitetty ja niistä kerätty ongelmat, pystyttiin niiden perusteella suunnittelemaan paikallista kehitysympäristöä paremmin, koska niiden pohjalta voitiin ottaa huomioon myös SaaS-palveluun tulevia uusia ominaisuuksia. Ennen kuin

ongelmiin pystyttiin esittämään mitään varmoja ratkaisuvaihtoehtoja, täytyi SaaS-palvelun toimintaa tutkia. SaaS-palvelun toiminnasta selvitettiin, miten sen saisi toimimaan SaaS-palvelun kehittäjien paikallisessa ympäristössä ja miten SaaS-palvelun pakettien kehittämisen prosessia voisi optimoida. Lisäksi ratkaisuissa otettiin huomioon, mitä SaaS-palvelun GitHubiin yhdistäminen vaatisi tämän projektin osalta.

Paikallisen SaaS-palvelun asennus

Yksi tehtävänannon pääkohdista oli selvittää ja suunnitella, miten SaaS-palvelun paikallisen version voisi toteuttaa. Ennen suunnittelun aloittamista selvitettiin tarkemmin, miten kyseinen SaaS-palvelu toimii ja mitä sen paikallisen version asentaminen vaatii.

SaaS-palvelu toimii Apache-palvelimella ja käyttää SQL-relaatiotietokantaa. Paikallista versiota varten asennettiin XAMPP-sovellus, jolla on mahdollista luoda paikallinen Apache-palvelin ja SQL-relaatiotietokanta. XAMPP-sovellus on myös yhteensopiva Windows-, Linux- ja MacOS-käyttöjärjestelmiin, joten se vastasi myös insinöörityön alustasta riippumattomuuden vaatimusta.

Apache-palvelimen perusasetukset XAMPP-sovelluksessa eivät suoraan sopineet SaaS-palvelulle, vaan Apache-palvelimen asetuksissa piti erikseen laittaa eri lisäosia päälle. SQL-tietokannan asentaminen onnistui tuotannossa olevan SaaS-palvelun tietokannan vedoksen viemisellä paikalliseen tietokantaan. Ongelmana oli kuitenkin se, että vedoksen koko oli suurempi kuin paikallisen tietokannan suurimman sallitun vietävän vedoksen koko. Ongelmaan oli yksinkertainen ratkaisu, kun tietokannan vedoksen enimmäiskoon rajaa nostettiin sen verran, että vedoksen tietokantarakenne saatiin vietyä paikalliseen tietokantaan.

Viennin jälkeen tietokanta oli lähes sama kuin tuotannossa olevan SaaS-palvelun tietokanta. Se sisälsi paljon ylimääräistä tietoa, jota ei ollut etukäteen poistettu. Nämä tiedot olivat turhia paikallisessa versiossa, koska se oli tarkoitettu lähinnä SaaS-palvelun moduulipakettien testaamiseen ja kehittämiseen, eikä siellä ollut tarkoitus säilyttää mitään tietoa pysyvästi. Tästä syystä annetusta tietokannan vedoksesta karsittiin kaikki ylimääräinen tieto pois, jotta tietokannan enimmäistuontikokoa ei tarvitsisi muuttaa tietokannan asetustiedostoista, kun tietokannan rakennetta asennetaan, ja tämä tekisi paikallisen SaaS-palvelun asentamisesta helpompaa. Näin myös vältytään tietoturvariskeiltä, koska annetussa tietokantavedoksessa oli joitain aikaisempien projektien tietoja. Näiden tietojen mukana oli myös joitain käyttäjien suoritustietoja. Karsitussa tietokantavedoksessa

näitä tietoja ei jaeta kaikille työntekijöille, vaan vedoksen mukana tulevissa tiedoissa on vain ne minimitiedot, joita tarvitaan paikallisen SaaS-palvelun asentamiseen.

Karsitussa tietokannan vedoksessa on myös yleinen käyttäjä, jolla on kaikki projektien tekoon ja testaamiseen tarvitut oikeudet SaaS-palvelussa. Näin SaaS-palvelua asentaessa ei tarvitse paikalliseen tietokantaan luoda aina uutta käyttäjää, kun SaaS-palvelu asennetaan ohjelmoijan tietokoneelle tai paikallinen tietokanta halutaan puhdistaa kaikesta turhasta tiedosta. Näin tietokannan vedoksen viennin jälkeen kehittäjät pääsevät suoraan testaamaan paikallisen SaaS-palvelun toimivuutta.

Toimivan tietokannan asentamisen jälkeen SaaS-palvelussa oli vieläkin ongelmia: palveluun ei päässyt kirjautumaan sisään, eikä palvelu antanut virheilmoitusta siitä, miksi palveluun kirjautuminen oli evätty. Ongelman selvitykseen käytettiin aluksi yksinkertaisesti erilaisia konsoliin kirjoitettuja viestejä ja koodin pysäytyskohtia (breakpoint), joiden avulla pyrittiin selvittämään, mistä virhe voisi johtua. Nämä menetelmät eivät auttaneet ongelman selvittämisessä, koska ne eivät olleet sellaisissa paikoissa, jotka olisivat näyttäneet, miksi sisäänkirjautuminen on estynyt. Niitä ei saatu laitettua sopiviin kohtiin, koska SaaS-palvelun koodikanta oli vielä näin projektin alkuvaiheessa hyvin tuntematonta ja koko koodikannan läpikäymiseen olisi mennyt todella paljon aikaa, joten virheen selvitys-tapaa oli muutettava. Ongelma selvisi, kun tutkittiin selaimen tallentamaa suoritusraporttia. Raportista nähtiin, mitkä toiminnot käynnistyivät juuri ennen virhetilannetta. Selvityksessä löytynyt toiminnallisuus seurasi, oliko selaimessa oikeita kirjautumiseen liittyviä evästeitä. Koska evästeitä ei ollut, toiminto käski selaimen lataamaan sivun uudestaan, mikä tarkoitti palaamista kirjautumissivulle.

Kun kirjautumisongelman aiheuttava toiminto oli saatu selvitettyä, jouduttiin ratkaisemaan puuttuvat evästeet. Evästeiden puuttuminen johtui Apache-palvelimen asetuksista. Asetuksista ei ollut aktivoitu kaikkia tarvittavia moduuleja, ja yksi niistä oli tarvittava evästemoduuli, joka mahdollisti istuntoevästeiden luomisen palvelimella. Niiden aktivoimisen jälkeen SaaS-palveluun pystyttiin kirjautumaan tavalliseen tapaan ja sen ominaisuuksia selvittämään ja testaamaan. Paikallisesta SaaS-palvelusta testattiin kaikki sen ominaisuudet ja varmistettiin, että se oli asennettu oikein paikalliselle työasemalle. SaaS-palvelun tarkastuksessa selvitettiin, tallentuivatko kaikki tiedot tietokantaan ja linkittyivätkö staattiset tiedostot oikein. Kun kaikkien ominaisuuksien toiminta oli varmistettu niin, että ne toimivat oikein, pystyttiin seuraavaksi keskittymään SaaS-palvelun toimintojen tarkempaan selvittämiseen ja Node.js-sovelluksen suunnitteluun.

4.2 Työtehtävien automatisointi Node.js:llä

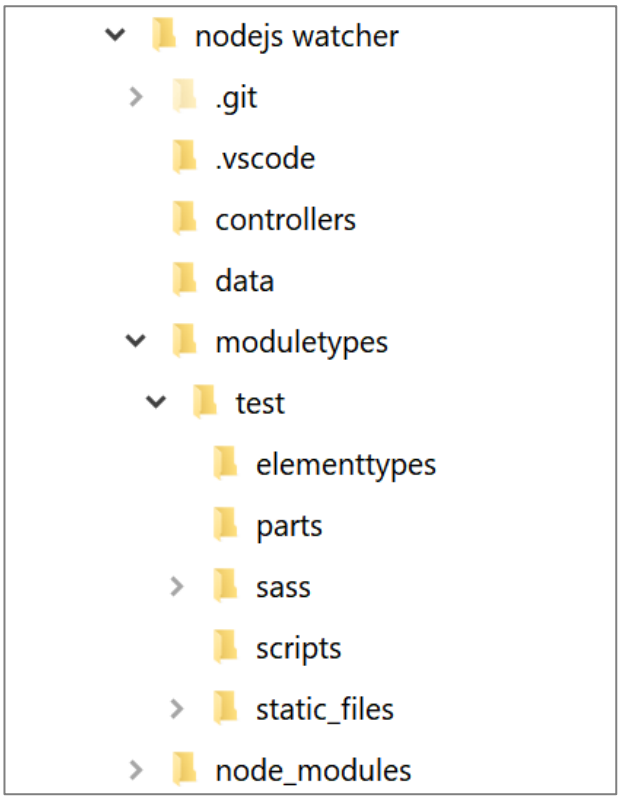
Sovelluksen suunnittelu

Paikallisen SaaS-palvelun asennuksen jälkeen seurasi Node.js-sovelluksen suunnittelu. Suunnittelussa tuli ottaa huomioon työntekijöiden mielipiteet. Sovelluksen tarkoitus on helpottaa paikallista kehitystyötä ja nopeuttaa palveluiden kehitystä. Sovellus ei saa aiheuttaa mitään ylimääräisiä ongelmia. Kaikki minkä voi automatisoida, automatisoidaan. Mitä vähemmän sovelluksen käyttäjä joutuu tekemään sovelluksen toimintaan liittyviä muutoksia tai korjauksia, sitä parempi.

Sovellus ei saa kaatua ilmoittamatta selkeästi ongelmista tai luoda ongelmia, joita pitää korjata aina jälkeenpäin. Sovelluksen pitää kaatua mahdollisimman aikaisin, jotta virheet eivät mene liian pitkälle huomaamatta ja aiheuta korjauksia vaativia ongelmia. Virheen ilmoituksen tulee olla selkeä käyttäjälle, jotta virheen selvittämiseen ei kulu aikaa, vaan se voidaan nopeasti korjata ja suorittaa ohjelma uudestaan ilman lisävirheitä.

Työkalun tulee toimia kaikkien työntekijöiden työasemilla. Se ei siis saa olla käyttöjärjestelmästä riippuvainen. Sovelluksen tulee olla helposti käytettävissä ja ymmärrettävissä. Sovelluksen tulee neuvoa käyttäjää kaikissa tilanteissa, joissa käyttäjä ei ole antanut tarvittavia tietoja. Sovellus varmistaa käyttäjältä, onko hän varmasti syöttänyt oikeat tiedot ennen toiminnallisuuden aloittamista.

Toiminnallisuuksien selvittämisen jälkeen pystyin aloittamaan itse Node.js-sovelluksen kehittämisen. Kuvassa 2 näytetään Node.js-sovelluksen kansiorakenne. ”nodejs watcher” on vain projektissa käyttämäni työnimi, jonka juuressa ovat kaikki sovelluksen tiedostot.



Kuva 2. Node.js-sovelluksen kansiorakenne. Kansiorakenteeseen on myös merkitty paikallisen SaaS-palvelun ”test”-niminen moduulipaketti ja sen kansiorakenne.

Projektikansion juuressa ovat myös itse Node.js-sovellukset ”createmoduletype.js” ja ”main.js”. Sovellusten lisäksi juuressa on sovellusten asetustiedosto ”configs.json”. Asetustiedosto sisältää Node.js-sovellusten tarvitsemat tietokantaan yhdistämistiedot ja paikallisen SaaS-palvelun kansion sijainnin. Projektikansion juuressa ovat myös kehitystyöhön liittyvät tiedostot, kuten ”.eslintrc.json”, joka sisältää ohjeet, mihin tyyliin sovellusten koodit tulisi kirjoittaa, ja ”.gitignore”, joka estää tiedostojen lisäämisen Git-versionhallintajärjestelmään.

Projektikansion tärkeimmät kansiot ovat ”controllers”-, ”data”- ja ”moduletypes”-kansiot. Loput kansiot ovat vain projektin tekoon ja versionhallintaan liittyviä. ”Controllers”-kansiossa ovat projektin Node.js-sovellusten yhteiset JavaScript-koodit, jotka käsittelevät tiedostojen muutokset ja tietokantaan lähetykset. Ne toimivat eräänlaisina kirjastoina pääsovellusten käytettäväksi, koska kumpikin sovellus tekee hyvin samanlaisia asioita, kuten luo, muokkaa tai poistaa tiedostoja ja lähettää ja vastaanottaa tietokannan tietoja. Nämä sovellusten käyttämät JavaScript-koodit on laitettu omaan kansioon selvyyden vuoksi. ”Controllers”-kansioon laitetut JavaScript-tiedostot ovat sovellusten yleisessä käytössä olevia Node.js-moduuleja, joita voidaan kutsua muiden sovellusten sisällä käyttämällä Node.js:n ”require”-komentoa.

Projektin ”data”-kansiossa on SaaS-palvelun pakettien metadata eli tiedostoihin liittyvät lisätiedot, joita ei muussa tapauksessa olisi. Metadata on kerätty tähän kansioon, jotta Node.js-sovellus tietäisi tallennettujen kooditiedostojen tarkemmat tiedot ja osaisi tallentaa ne tietokantaan oikean tunnisteen paikalle, jos ne ovat siellä jo olemassa. Metadatatiedosto sisältää seuraavat paketin tiedot:

- paketin metadata (tietokantatunniste, nimi, omistaja)
- HTML-elementtien metadata (tietokantatunniste, tyyppi, nimi)
- HTML-osien nimet.

Projektikansion ”moduletypes”-kansioon tallennetaan kaikki muokattavissa olevat SaaS-palvelun paketit. Niistä valitaan Node.js-sovellusta käynnistäessä kansion nimen perusteella se, mihin kansioon halutaan tehdä muutoksia ja minkä kansion tiedostojen muuttumista tulisi seurata. SaaS-pakettien kansion juuressa on myös paketin metadatatiedosto, johon on lisätty samat tiedot kuin projektin datakansioon, mutta kaikista tiedoista on poistettu tietokantamerkinnän id-tunniste. Tämän metadatatiedoston tarkoitus on mennä SaaS-paketin versionhallintaan mukaan, jolloin jokainen, joka paketin lataa versionhallinnasta, pystyisi asentamaan sen tietokantaan yhdellä komentorivin komennolla. Koska versionhallinnassa ovat SaaS-paketin kaikkien tiedostojen sisällöt ja niiden metadatatiedot, ne voidaan siirtää yksinkertaisesti tietokantaan ja aloittaa itse SaaS-paketin kehittäminen mahdollisimman nopeasti.

SaaS-pakettien sisällä olevista metadatatiedostosta on poistettu tietokantamerkintöjen tunnisteet, koska ne eivät vastaa kaikkien kehittäjien tietokantojen merkintöjen tunnisteita, koska jokaisella kehittäjällä on oma paikallinen tietokanta, johon SaaS-paketin tiedot on tallennettu ja tietojen id-numerot vaihtelevat eri kehittäjien välillä. Tästä syystä paketin sisäisen metadatasta on poistettu turhat id-tunnisteet. Kun komentorivillä käsketään lisäämään paketin tiedot tietokantaan, luo Node.js-sovellus samalla omaan ”data”-kansioon paketin metadatan, joissa on tietokantamerkintöjen tunnukset, joita voidaan käyttää tallennettaessa tietokantaan uudestaan.

Paikallinen työkalu koostuu kahdesta eri Node.js-sovelluksesta. Ensimmäinen sovellus luo SaaS-palveluun paketin luontiin tarvittavat kansiot ja tietokantamerkinnät, ja toinen Node.js-sovellus mahdollistaa uusien elementtien lisäämisen, poiston ja muokkaamisen tallentamisen automaattisesti. Paikallisen työkalun rakenne oli suunniteltu niin, että sovellus säilyttää jokaisesta asennetusta SaaS-palvelun paketista omat metadatatiedostot. Näiden tiedostojen avulla paikallinen Node.js-sovellus tietää SaaS-palvelun paketin

osista enemmän ja pystyy lähettämään tarvittavat tiedot paikallisen SaaS-palvelun tietokantaan.

SaaS-palvelun paketit ladataan paikallisen Node.js-sovelluksen moduulipakettikansioon, jossa ovat kaikki muutkin SaaS-palvelun asennetut paketit. Näitä paketteja pystytään yksitellen muokkaamaan niin, että kehittäjä kirjoittaa komentoriviin, mitä pakettia hän haluaa muokata, jolloin Node.js-sovellus alkaa seurata juuri sen paketin muutoksia. Kaikkia asennettuja SaaS-paketteja kyllä pystyisi seuraamaan, mutta seurattavien tiedostojen määrä saattaisi kasvaa jossain vaiheessa liian suureksi, jolloin seurantaprosessi hidastuu.

Paikallisten Node.js-sovellusten JavaScript-koodit eivät kaikki ole samassa ohjelman tiedostossa, vaan aina kun mahdollista JavaScript-koodit jaetaan oman aihealueensa tiedostoihin. Näin ollen sellaiset JavaScript-koodit, jotka käsittelevät tiedostoja tai tietokantaan liittyviä kutsuja, laitetaan omiin tiedostoihinsa. Koodeja pystytään näin jakamaan muiden Node.js-sovellusten kanssa eikä samanlaisia toimintoja tarvitse kirjoittaa moneen kertaan uudestaan tai Node.js-sovelluksia tarpeettomasti yhdistää keskenään. Näin sovelluksen toiminnallisuudet selkiytyvät ja koodista tulee helpommin luettavaa, koska koodia lukemalla heti huomataan, että seuraavaksi siirrytään käsittelemään tiedostoja tai haetaan tietokannasta tietoja.

Projektiin valitut tekniikat ja työkalut

Paikallisen SaaS-palvelun kehityksen automatisointiin valittiin käyttöön Node.js-sovellusympäristöksi, NPM valittiin Node.js-moduulienhallintaan, Gulp automaattisten tehtävien suorittamiseksi ja GitHub valittiin projektin versionhallintajärjestelmäksi. Automaattisia testejä tähän projektiin ei lisätty, koska se olisi vaatinut testityökalujen käytön opettelua, johon ei tässä projektissa saatu varattua aikaa. Lisäksi yrityksellä ei vielä tässä vaiheessa ollut kiinnostusta lisätä kehitysprosesseihin yksikkötestausmenetelmiä, koska se vaatisi projektien kehitysprosessin ja rakenteen muuttamista.

Projektiin valittiin Gulp automaattisten tehtävien suorittamiseen, koska sitä oli jo aikaisemmin yrityksessä käytetty ja se oli tuttu minun lisäkseni myös muille Apprixin työntekijöille, joten sen ympärille kehitetty projekti olisi kaikille kehittäjille helppo ottaa käyttöön ja mahdollisesti jatkokehittää. Gulp on muutenkin tehokkaampi kuin Grunt suorittamaan tehtäviä, koska se käyttää hyväkseen Node.js:n tapahtumavetoista arkkitehtuuria, jolloin se mahdollistaa asynkronisten eli samanaikaisten tehtävien suorittamisen.

Git-versionhallintajärjestelmä otettiin tähän projektiin käyttöön, koska se on Apprixilla jo käytössä, vaikka mikä tahansa muu hajautettu versionhallintajärjestelmä olisi myös sopinut tähän projektiin. GitHub mahdollistaa myös Node.js-sovelluksen yksinkertaisen lataamisen kehittäjien omille työasemille. Jos jollekin kehittäjälle tulee jotain parannusehdotuksia Node.js-sovellukseen, hän voi jatkaa sovelluksen kehitystä tekemällä GitHubissa olevaan versioon muutoksia vaarantamatta sovelluksen toimivuutta. Kun sovellukseen on kehitetty jokin uusi ominaisuus ja se on hyväksytty, voidaan ominaisuus siirtää pääsovellukseen ja jakaa uusi versio muille kehittäjille.

Sovelluksen toteutus

Itse projektin sovelluksen tekeminen alkoi sen tärkeimmistä toiminnallisuuksista, jotka olivat SaaS-palvelun pakettien sisällä olevien HTML-elementtien ja JavaScript-tiedostojen automaattinen tallennus paikalliseen SaaS-palvelun tietokantaan. Tarkoituksena oli käyttää Gulpin seurantatoimintoa (watch) tiedostojen muutosten seuraamiseen. Aina, kun tiedostoissa havaitaan muutos, sen sisältö tarkistetaan, minkä jälkeen tiedoston tiedot on tallennettava paikallisen SaaS-palvelun tietokantaan palvelun käytettäväksi.

Aluksi tiedostojen tietojen tallennus suunniteltiin toimimaan omien tietokantaan lähetettävien SQL-kyselyiden perusteella. SaaS-palvelun toimintoja selvittäessäni huomasin, että se käyttää erilaisia API-kutsuja koodien tallentamiseen tietokantaan. Ajattelin käytää samoja API-kutsuja myös omassa Node.js-sovelluksessa, koska silloin vältyttäisiin koodin kopioimiselta ja API-kutsujen toiminnallisuus olisi aina ajan tasalla senhetkisen SaaS-palvelun kanssa. Näin ollen minun ei tarvitsisi muuttaa oman ohjelmani koodeja aina, kun SaaS-palvelun API-pyyntöihin tulee muutoksia. Toinen vaihtoehto olisi ollut, että teen Node.js-sovelluksen niin, että se keskustelee suoraan SaaS-palvelun tietokannan kanssa, mutta luovuin tästä ideasta, koska omien kutsujen ylläpitäminen tuottaisi turhaa työtä ja tässä projektissa varta vasten pyrittiinkin vähentämään kaikkien tekemän työn määrää. Suunnittelin Node.js-sovelluksen hyödyntämään mahdollisimman paljon SaaS-palvelun omia API-kutsuja.

Node.js-sovellukseen jouduttiin tekemään muutama oma tietokantakysely, koska SaaS-palvelu ei itsessään sellaisia API-kutsuja sallinut, koska Node.js:lle tehdyt tietokantakyselyt olisivat olleet tietoturvariski SaaS-palvelun tuotantoversiossa. API-kutsu, joka palauttaisi muidenkin käyttäjien tietoja kuin oman, olisi tietoturvariski ja mahdollistaisi tieto-

jen päätymisen sellaisille henkilöille, joille ne eivät kuulu. Räätälöityjen tietokantakyselyiden tekeminen oli mahdollisia, koska paikallinen tietokanta oli itse luotu ja kaikki tietokantayhteyden luontiin tarvittavat tiedot olivat saatavilla.

Kun tiedostojen seuranta oli saatu valmiiksi, oli aika kehittää sovellukseen ominaisuus, joka helpottaisi projektin aloittamista tai jo tehdyn projektin lisäämistä ja jatkamista. Projektin aloitusta varten kehitettiin oma Node.js-sovellus, koska se ei tarvinnut samoja tiedoston seuraamisominaisuuksia kuin tiedostojen seuranta ja automaattisesti tallentava ohjelma. Tässä projektinluontisovelluksessa oli otettava huomioon, mitä kaikkea käyttäjät tekevät ennen projektin aloitusta. Aluksi kaikkia projektin aloituksen aikana tehtyjä asioita ei saatu kerättyä, mutta luontisovelluksen kehityksen aikana pidetyissä kehityspalavereissa kävi ilmi muutamia SaaS-palvelun kehittäjien projektien alussa tekemiä asioita, jotka vaikuttivat esimerkiksi tiedostojen nimen merkityksen heikentämiseen ja metadatatiedoston tietojen eheyden tarkastamiseen ennen minkään toiminnon aloittamista.

Aluksi SaaS-palvelun pakettien elementtien tiedostojen nimet olivat uniikkeja eikä niitä ollut merkitty erilliseen metadatatiedostoon, koska tiedostojen nimet olivat samat kuin metadataan merkitty elementtityypin nimi. Mutta myöhemmin huomattiin, että SaaS-palvelussa saa olla saman elementtityypin omaavia elementtejä, joten elementtityyppien nimet eivät olleet uniikkeja eikä niitä voitu käyttää tiedostojen nimessä. Tästä syystä tiedostojen nimen merkitystä heikennettiin ja niistä tehtiin sellaisia, että ne voivat olla mitä tahansa.

Kun projektin aloitus oli lähes valmis, huomattiin, että usein projektia aloittaessa SaaS-palvelun kehittäjät ottavat jonkin SaaS-palvelun pakein pohjan, johon he joko lisäävät tai josta he poistavat paketin HTML-elementtejä. Tämä olisi sotkenut metadatatiedostossa olevat tiedot, koska mitään komentoa ei ole HTML-elementtien tiedostojen poiston aikana käytetty. Tätä varten sovellukseen kehitettiin metadatatiedoston eheyden tarkistus, ennen kuin tehdään mitään. Näin ollen, jos metadatatiedostossa on joitain elementin tietoja, joista ei ole omaa tiedostoa olemassa, niiden tiedot poistetaan saman tien metadatatiedostosta. Tällä tavalla estetään projektin luonnissa ylimääräisten tyhjien HTML-elementtien luonti tai ohjelman kaatuminen luomisvaiheessa.

Projektin alussa ei tiedetty aivan kaikkia tapahtumia, joita SaaS-palvelun kehittäjät tekevät, koska kysytyt kysymykset eivät olleet tarpeeksi kattavia tai vastaajat eivät vain muistaneet sillä hetkellä kertoa kaikkea. Projektin kehityksen aikana näistä uusista SaaS-palvelun kehittäjien antamista kommenteista pystyttiin sovellukseen lisäämään uusia

ominaisuuksia, joita muussa tapauksessa ei olisi lisätty. Jos uusien ominaisuuksien kehittäminen käyttäjien kommenttien pohjalta olisi jätetty tekemättä, olisi Node.js-sovelluksen käyttäjäkokemus ollut huonompi ja muutokset olisi pitänyt tehdä sovelluskehitystyön jälkeen, jolloin se on aina hankalampaa, koska joitain jo toimivia sovelluksen toimintoja mahdollisesti jouduttaisiin muuttamaan.

Node.js-sovelluksen testaukset

Node.js-sovelluksia kehittäessä uusien ominaisuuksien ja toimintojen koodit kehitettiin erillisessä tiedostossa. Näin testien aikana ei luotu turhia tietokantamerkintöjä ja vältettiin turhien tiedostojen luomista, jos se ei ollut juuri se, mitä testattiin. Erillisessä testitiedostossa oli nopeampaa selvittää kehitysvaiheessa tulleet ongelmat ja näin vähennettiin ylimääräistä työtä. Erillisessä testitiedostossa pystyttiin testaamaan sovelluksen yksittäisiä toimintoja. Toiminnolle annettiin sellaisia arvoja, joita sen haluttiin saada. Arvojen annon jälkeen nähtiin millaisia vastauksia toiminto palautti. Jos toiminto ei palauttanut haluttuja arvoja, voitiin syyt nopeasti selvittää ja korjata. Tämä testaustapa nopeutti sovelluksen kehitystä ja paransi sovelluksen yleistä toimivuutta, koska varmistuttiin siitä, että annetuilla arvoilla toiminto palauttaa haluttuja vastauksia. Toiminto lisättiin myöhemmin sovellukseen, kun se testien pohjalta toimi niin kuin pitikin. Lisäyksen jälkeen toiminnon yhteensopivuus testattiin heti, kun se oli lisätty osaksi muuta sovellusta. Näin varmistuttiin siitä, että muut toiminnot eivät rikkoutuneet uuden toiminnon lisäämisen jälkeen.

Kun projektin sovellukset lähestyivät valmistumista, suoritettiin sovellusten käyttäjätestausta. Käyttäjille annettiin tehtäväksi asentaa paikallinen SaaS-palvelu omalle työasemalleen, asentaa paikallisen SaaS-palvelun automatisointityökalu ja luoda uusi SaaS-palvelupaketti ja lisätä sinne uusi HTML-elementti. Testit suoritettiin kaikilla toimistossa löytyvillä käyttöjärjestelmillä, jotta varmistuttaisiin siitä, että sovellukset toimisivat käyttöjärjestelmästä riippumatta.

MacOS-käyttöjärjestelmää käyttävällä työasemalla oli hankaluuksia saada itse SaaS-palvelua ladattua pilvipalvelimelta, koska tiedostojen siirrossa käytetty sovellus katkaisi yhteyden palvelimeen satunnaisesti ja suuren tiedostomäärän lataaminen vei niin paljon aikaa, että palvelun lataaminen oli lähes mahdotonta. SaaS-palvelu saatiin siirrettyä MacOS-koneelle, kun SaaS-palvelun tiedostot siirrettiin lähiverkkolevylle, josta se oli huomattavasti nopeampi ja helpompi ladata. Ongelmana tällä sijoituspaikalla on kuitenkin

se, että siellä olevaa tiedostoja jouduttaisiin päivittämään manuaalisesti toisin kuin pilvipalvelimella olevaa tiedostoa, jota päivitetään sitä mukaa, kuin SaaS-palvelua päivitetään. Tämä oli vain väliaikainen ratkaisu, jotta testejä voitiin jatkaa.

MacOS-käyttöjärjestelmän testeissä ilmeni myös kansioiden oikeuksien kanssa ongelmia. MacOS-käyttöjärjestelmässä XAMPP:n Apache-palvelimella oli ongelmia jakaa sen kansioissa olleita tiedostoja, koska palvelimella ei ollut kaikkia sen tarvitsemia oikeuksia. Yksi idea tämän korjaukseen oli muuttaa kansioiden käyttäjäoikeudet niin, että XAMPP:n Apache-palvelin pystyy niitä paikallisesti jakamaan. Myöhemmin kuitenkin huomattiin, että XAMPP:n Apache-palvelimen asetuksista pystyi muuttamaan käytettyä käyttäjää sellaiseksi, jolla oli oikeudet kansoihin, minkä jälkeen SaaS-palvelu ja Node.js-sovellukset toimivat samalla tavalla kuin Windows-käyttöjärjestelmässä.

Linux -käyttöjärjestelmässä oli hieman ongelmia XAMPP:n toimintaan saamisessa, koska siihen ei saanut suoraan ladattua yksinkertaista pakettia, jolla sovellus olisi suoraan asentunut oikein. XAMPP:n asentamisen jälkeen SaaS-palvelussa ja Node.js-sovelluksessa ei ollut mitään eroa Windows- ja MacOS-käyttöjärjestelmään, joten sovellus oli valmis käyttöönottoa varten.

4.3 Lopputulos

Insinöörityöprojektin alussa tehdyt suunnitelmat eivät pysyneet samana projektin kehityksen aikana. Node.js -työkaluun tuli paljon uusia ominaisuuksia, joita ei aluksi osattu lisätä suunnitelmiin. Suunnitelmia muutti myös tulevaisuudessa lisättävä GitHub-integraatio. Usein nämä ideat uusista prosessia helpottavista ideoista tulivat muilta Apprixin ohjelmoijilta, kun he kertoivat työntekoprosesseistaan.

Yksi haasteellisimmista ongelmista oli keksiä ratkaisu siihen, miten paikallinen Node.js -sovellus tietää kaikki SaaS-palvelun paketeissa olevien HTML-sivujen metadatan. HTML-sivulla oli tunniste, nimi ja tyyppi. Kummatkaan arvot eivät olleet uniikkeja, joten Node.js-sovelluksessa piti ottaa myös huomioon, mihin tiedostoon viitataan, jotta muutokset voidaan joko hakea tai viedä tietokantaan. Koska projektin määrittelyssä oli sanottu, että kaiken pitää olla mahdollisimman automaattista, eivät käyttäjän itse kirjoittamat metadatat olisi ollut kelpaava ratkaisu. Ongelman ratkaisuksi keksin käyttää erillistä metadatatiedostoa Node.js-sovelluksen sisällä. Jokaisella SaaS-palvelun paketilla on

oma metadatatiedosto, johon tallentuisivat kaikki Node.js-sovelluksen ja SaaS-palvelun tarvitsemat oheistiedot.

Tämä ratkaisu ei toimisi GitHubissa olevan version kanssa. Jos jokainen ohjelmoija lataisi GitHubista paketin uusimman version, tulisi heidän tietokantojensa kanssa ongelmia, sillä metadatatiedoissa olisivat SaaS-palvelun osien tunnisteet erilaiset, kuin ne olivat heidän omassa paikallisessa tietokannassaan. Joillakin kehittäjillä voi jo olla tietokannassaan jokin muu elementti, jolla on sama tunniste tietokannassa kuin ladatussa metadatatiedostossa. Ainoana ratkaisuna tähän ongelmaan keksin käyttää kahta metadata-JSON-tiedostoa. Toiseen tallentuvat paikallisen tietokannan tarvitsemat tiedot ja toiseen on merkitty SaaS-palvelun paketin osien perustiedot, jotta Node.js-sovellus pystyy rakentamaan SaaS-palvelun paikalliseen tietokantaan paketin metadatatietojen pohjalta.

Nämä JSON-metadatatiedostot ovat myös melko virhealttiita. Node.js-sovelluksen käyttäjät voivat lisätä tai poistaa tiedostoja silloin, kun Node.js-sovellusta ei suoriteta. Näin metadatatiedostot eivät ole enää ajan tasalla ja aiheuttavat erilaisia virheitä, koska Node.js-sovellus luottaa siihen, että metadatan tiedot ovat oikeita. Tämä ongelma ratkaistiin lisäämällä Node.js-sovelluksen käynnistyksen yhteyteen automaattinen tarkistus. Tarkistus katsoo, onko metadatassa oleviin tietoihin vastaavia tiedostoja oikeissa paikoissa. Jos metadatasta ei tietoja löydy, poistetaan metadata JSON -tiedostoista.

Projektiin olisi voinut käyttää enemmänkin Node.js:n tukemia tapahtumia ja asynkronisuutta, koska Node.js perustuu tapahtumasilmukan eli ”Event-Loop”:n ympärille ja tukee asynkronisesti toimivia funktioita. Node.js-ohjelmat käsitelevät tapahtumasilmukassa olevia tapahtumia yksi kerrallaan ja suorittavat ne siinä järjestyksessä, kuin ne ovat silmukkaan tulleet. Omassa ohjelmassani oli tiedosto- ja kansiorakennetta tehtäessä käytetty synkronista tiedostojen ja kansioiden luontia. Valitsin synkronisten funktioiden käytön kansiorakenteen luomisessa, koska alikansioiden luominen kuitenkin joutuu odottamaan, että ylempi kansio on ensin olemassa. Kansioiden luominen olisi voinut olla asynkroninen, mutta asynkronisuudesta saatu hyöty olisi ollut käytännössä huomaamattoman pieni.

Synkronisesti kansioiden luonti on kätevä luotaessa useita alikansioita, jolloin varmistuttiin siitä, että ylemmät kansiot olivat valmiina, ennen kuin alikansioita aletaan luoda. Synkroniset toiminnot mahdollistivat myös kansiorakenteen luomisen ilman vastakutsuja (callback). Asynkroniset toiminnot sopivat parhaiten palvelin- ja selainympäristöön,

koska palvelimella tietojen haku saattaa kestää jonkin aikaa ja tulokset voidaan lähettää tai parsia heti, kuin ne on saatu. Komentorivisovelluksessa asynkronisuus ei välttämättä ole niin tärkeää, mutta se on usein suositeltavaa [5, s. 69].

Automaattinen testaus olisi voinut olla hyvä lisä tähän projektiin. Projektia tehdessä huomattiin, miten esimerkiksi yksikkötestausta olisi voitu suorittaa automaattisesti sovelluksen eri toiminnoille. Jos projektia olisi tehty testivetoisesti, olisin luultavasti välttynyt joidenkin muuttujien väärien tietojen aiheuttamien virheiden etsimiseltä. Olisin huomannut suoraan automaattisista testeistä, että funktiot eivät anna oikeita arvoja, ja olisin suoraan voinut korjata nämä ongelmat. Testausta ei lisätty tähän työhön, koska en ollut aiemmin koettanut tehdä automaattisia testejä. Testien tekeminen on turhaa ja ne vievät aikaa, jos ne eivät ole tarpeeksi hyviä ja kattavia.

5 Yhteenveto

Insinöörityöprojektissa tarkoituksena oli mahdollistaa SaaS-palvelun kehittäminen paikallisesti työn tilaajayrityksen ohjelmoijien omilla tietokoneilla, jotta SaaS-palvelun moduulipakettien kehittäminen olisi lähempänä modernia web-kehittämistä. Lopputuloksen piti tehdä SaaS-palvelun paikallisesta kehityksestä mahdollisimman helppoa, jotta palvelun kehitystyö olisi entistä nopeampaa ja vältyttäisiin virhetilanteilta.

Aluksi selvitettiin, mitä ongelmia SaaS-palvelun pakettien kehittämisessä on. Käyttäjätarinoiden teon jälkeen voitiin listata ratkaistavat ongelmat ja muut projektin vaatimukset. Listattujen ongelmien ja vaatimusten pohjalta voitiin SaaS-palvelun paikalliselle versiolle suunnitella Node.js-sovellus, joka vastaisi listattuihin asioihin.

Projektissa opastettiin Apprixin ohjelmoijia siitä, miten SaaS-palvelu saadaan asennettua heidän työasemilleen, jolloin paikallinen kehitystyö on mahdollista. Tämä ei kuitenkaan ole riittävän hyvä vastaus annettuihin projektin vaatimuksiin. Pelkkä paikallinen SaaS-palvelu ratkaisee vain tiimityöskentelyn ongelmat. Paikallinen SaaS-palvelu mahdollistaa SaaS-palvelun pakettien testaamisen paikallisesti kehittäjien omilla työasemilla, jolloin kukaan muu ei voi tallentaa muutosten päälle ja samaa paketin osaa voi kehittää moni yhtä aikaa. Pelkkä paikallinen versio kuitenkin lisäsi kopioinnin määrää paikallisesta testipalvelusta tuotantopalveluun, joten tähän oli saatava jokin ratkaisu.

SaaS-palvelulle kehitettiin Node.js-sovellus, joka hoitaa tietojen tallennuksen paikalliseen SaaS-palvelun tietokantaan ja paikallisen SaaS-palvelun staattisiin tiedostoihin automaattisesti. Node.js-sovellus myös mahdollistaa kooditiedostojen säilyttämisen erillisinä tiedostoina, jotka voidaan lisätä GitHub-versionhallintajärjestelmään. Node.js-sovelluksen käyttämät tiedostot sisältävät kaiken tiedon, jota paikallinen SaaS-palvelu tarvitsee pakettien luomiseen. Pakettien luominen on Node.js-sovelluksella tehty hyvin yksikertaiseksi loppukäyttäjälle. Node.js -sovellus luo uusia SaaS-palvelun paketteja yhdellä komentorivin käskyllä, ja luotua pakettia voidaan heti sen luonnin jälkeen muokata. Jos käyttäjällä on jo valmis SaaS-palvelun paketti, sen voi myös yhdellä komentorivin käskyllä lisätä paikalliseen SaaS-palvelun tietokantaan, jolloin kaikkien paketissa olevien tiedostojen tiedot siirretään SaaS-palvelun saataville.

Node.js-sovelluksen käyttö itse moduulipakettia muokatessa tehtiin mahdollisimman yksinkertaiseksi. Yhdellä komentorivin komennolla käyttäjä määrittelee, minkä paketin muutoksia tulisi seurata. Node.js-sovellus sitten tekee kaikki tarvittavat toimenpiteet, jotta käyttäjän moduulipakettiin tekemät muutokset päivittyvät myös paikallisessa SaaS-palvelun versiossa. Tehdyt muutokset voidaan myöhemmin tallentaa myös GitHub-versionhallintaan, kun ominaisuudet on tarpeeksi hyvin testattu. Näin muut ohjelmoijat pääsevät käsiksi paikallisesti tehtyihin muutoksiin ja päivitetty moduulipaketti voidaan myös päivittää palvelimella olevaan SaaS-palveluun, jossa asiakkaat pääsevät käyttämään uutta ominaisuutta.

Jatkossa saattaa tulla uusia vaatimuksia Node.js-sovellukselle. GitHub-integraatio on yrityksessä tekeillä, ja sitä varten voi Node.js-sovellus ehkä tarvita joitain muutoksia. Koodia on dokumentoitu käyttäen JSDoc-standardia, jolloin jokaiselle toiminnolle on kirjoitettu, mitä se tekee ja mitä parametreja toiminnot tarvitsevat toimiakseen oikein. Kommentteja on myös lisätty muutaman pidemmän funktion sisälle, jotta koodin lukijalle selviää, mitä funktion eri vaiheissa tapahtuu. Näin Node.js-sovellusta on helpompi muokata myös sellaisen ohjelmoijan, joka ei ole ollut aikaisemmin tekemisissä tämän projektin kanssa.

Insinöörityötä tehdessä huomattiin, miten tärkeää sovelluksen testausten tekeminen on ja miten ne olisivat helpottaneet sovelluksen kehitystyötä. Aluksi sovelluksen kehittäminen on helppoa ja nopeaa, kun koodikanta on hyvin pieni ja sovelluksen käsin testaaminen on yksinkertaista. Sovelluksen kasvaessa eri ominaisuudet ovat riippuvaisia toisistaan, jolloin vanhoja toiminnallisuuksia muuttaessa rikkoutuu huomaamatta jokin toinen

toiminnallisuus. Jos projektissa olisi automaattisia testauksia, huomattaisiin heti muutosten teon jälkeen, läpäiseekö sovellus testit vai ei, ja mahdolliset virheet pystytään heti korjamaan.

Toinen projektin aikana huomattu sovellustestaukseen liittyvä asia oli projektin vaatimusten muuttuminen kehitystyön aikana. Sovellusta kehitettäessä vaatimuksiin tuli erilaisia muutoksia, koska selvisi, miten käyttäjät saattavat toimia sovellusta käyttäessään. Vanhojen ominaisuuksien kehittäminen vastaamaan muuttuneita vaatimuksia oli hankalaa, koska ominaisuuksia muuttaessa piti ottaa huomioon koko sovelluksen toiminta. Jos projektissa olisi ollut kunnolliset testit, olisi niistä suoraan nähty, mihin tehdyt muutokset vaikuttivat, jolloin koko sovelluksen toimintojen riippuvuuksia ei olisi tarvinnut itse muistaa. Koska projektin koodikannan koko oli muutama tuhat riviä, muutosten tekeminen ja toimintojen riippuvuuden muistaminen ei ollut liian hankalaa, mutta jo tässä huomattiin, että testeistä olisi ollut hyötyä. Tätä suuremmissa projekteissa olisi ollut lähes mahdotonta tietää tehtyjen muutosten vaikutus koko sovelluksen toimintaan ilman, että sovelluksen toimintaa testattaisiin käsin. Testit myös helpottavat sellaisten kehittäjien työtä, jotka eivät projektia tehdessä ole olleet mukana. Testien avulla he voisivat olla varmoja, että heidän tekemänsä muutokset eivät vaikuta mihinkään muuhun sovelluksen toiminnallisuuteen.

Gulp-tehtävät olisi voinut tehdä Gulpin toimintaperiaatteen mukaan. Gulpin automaattiset tehtävät koostuvat pienistä osista, jotka tekevät yhden asian hyvin. Tässä insinöörityössä Gulpin tehtävät saattoivat tehdä isojakin toimintoja kerralla. Tämä kuitenkin mahdollistaa sen, että tehty Node.js-sovellus ei ole Gulpista riippuvainen. Sovelluksessa pystytään käyttämään mitä tahansa automaattista tehtävien suorittajaa, kuten Gruntia.

Tehty insinöörityö mahdollistaa paikallisesti SaaS-palvelun moduulipakettien kehittämisen, ja moduulipakettien kehittämisestä tehtiin paikallisesti mahdollisimman helppoa automaattisesti suoritettavien tehtävien avulla. Insinöörityössä tehty sovellus mahdollistaa myös tulevaisuudessa lisättävän vahvemman GitHub-integraation SaaS-palvelun ja GitHubin välillä. SaaS-palvelun moduulipakettien kehittämistyö ei muutu mitenkään tehdyn integraation jälkeen, mutta se vähentää tiedostojen paikasta toiseen kopiointia entisestään.

Lähteet

1 Taloustiedot. Verkkoaineisto. Finder. <https://www.finder.fi/IT-sovelluksia+IT-ohjelmistoja/Apprix+Oy/Helsinki/yhteystiedot/531677/>. Luettu 30.4.2018

2 Jazayeri, Mehdi. 2009. Some Trends in Web Application Development. Future of Software Engineering. IEEE.

3 Application Archetypes. 2018. Verkkoaineisto. Microsoft. <https://msdn.microsoft.com/en-us/library/ee658104.aspx#WebApplicationArchetype/>. Luettu 22.2.2018.

4 Odell, Den. 2014. Pro JavaScript Development. E-kirja. Apress.

5 Hughes-Croucher, Tom & Wilson, Mike. 2012. Node: Up and Running. E-kirja. O'Reilly Media.

6 Teixeira, Pedro. 2012. Hands-on Node.js. E-kirja. Leanpub.

7 Tilkov, Stefan & Vinoski, Steve. 2010. Node.js: Using JavaScript to Build High-Performance Network Programs. IEEE Internet Computing. Vol. 14, s. 80–83.

8 Adams, Chad R. 2015. Mastering JavaScript High Performance. E-kirja. Pact Publishing.

9 Gilbertson, David. 2018. I'm harvesting credit card numbers and passwords from your site. Here's how. Verkkoaineisto. Hacker Noon. < https://hackernoon.com/im-harvesting-credit-card-numbers-and-passwords-from-your-site-here-s-how-9a8cb347c5b5/>. Luettu 7.1.2018.

10 Severien, Tim. 2016. Yarn vs npm: Everything You Need to Know. Verkkoaineisto. Sitepoint. <https://www.sitepoint.com/yarn-vs-npm/>. Päivitetty 26.10.2016. Luettu 20.3.2018.

11 Hazem, Saleh. 2013. Javascript Unit Testing. E-kirja. Packt Publishing.

12 Chacon, Scott & Straub, Ben. 2014. Pro Git. 2nd Edition. E-kirja. Apress.

13 Getting Mozilla Source Code Using Mercurial. 2018. Verkkoaineisto. Mozilla. <https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Source_Code/Mercurial/>. Päivitetty 15.1.2018. Luettu 21.3.2018.

14 Cheang, Henry. 2014. The Arguments for Software-as-a-Service (SaaS). Verkkoaineisto. Cimpl. <http://blog.cimpl.com/bid/205831/The-Arguments-for-Software-as-a-Service-SaaS/>. Luettu 25.3.2018.

15 Patrizio, Andy. 2018. Application Archetypes. Verkkoaineisto. Datamation. <https://www.datamation.com/cloud-computing/50-leading-SaaS-companies.html/>. Luettu 25.3.2018.