Web-sovellus ohjelmistokehityksen jatkuvan integroinnin statistiikan visualisointiin

Juha Pitkänen

Opinnäytetyö
Tietojenkäsittelyn koulutusohjelma
2019

Tiivistelmä

Tekijä(t)
Juha Pitkänen

Koulutusohjelma
Tietojenkäsittelyn koulutusohjelma

Opinnäytetyön otsikko	**Sivu- ja liitesivumäärä**
Web-sovellus ohjelmistokehityksen jatkuvan integroinnin statistiikan visualisointiin	34 + 11

Tämän opinnäytetyön tavoitteena oli luoda web-sovellus, joka visualisoi ohjelmistokehityksen jatkuvan integroinnin statistiikkaa. Opinnäytetyön tavoitteena oli myös tutustua Node.js-ympäristön ja React-sovelluskehyksen käyttöön. Toimeksiantajana oli Accountor Finago Oy -ohjelmistoyritys.

Jatkuva integrointi on ohjelmistokehityksessä yleinen käytäntö, jonka tarkoituksena on yhdistää kehittäjien kirjoittamaa ohjelmistokoodia jaettuun tietolähteeseen mahdollisimman usein ja automatisoitujen työvaiheiden avulla varmistaa lähdekoodin eheys. Tämän mahdollistamiseksi tarvitaan automatisoituja työvaiheita ja testejä, joiden toiminnasta saadaan statistiikkaa.

Työ on toteutettu ensin suunnittelemalla sovelluksen toiminta, ulkoasu ja tietokantaratkaisu. Produkti koostuu Node.js-ympäristöllä kehitetystä rajapinnasta, Jenkins-ohjelman rajapinnan hyödyntämisestä, tietokannasta ja React-sovelluskehyksellä kehitetystä käyttöliittymästä. Lisäksi sovellus on paketoitu Docker-ohjelmaa hyödyntäen niin, että sen ajoympäristöä on mahdollisimman helppo vaihtaa.

Tein opinnäytetyöni samanaikaisesti käydessäni päivätyössä. Sovellus on toteutettu työajalla, opinnäytetyö taas työajan ulkopuolella. Kehitin sovellusta aina kahden viikon sprinteissä kerrallaan, kun muut työtehtävät sen mahdollistivat.

Käytännön toteutus onnistui hyvin ja vastasi odotuksiani, sillä lopputuloksena syntyi toimiva ja julkaisukelpoinen ohjelmisto jatkuvan integroinnin statistiikan visualisointiin. Sovelluksen vaatimusmäärittely tarkentui hieman sovellusta kehitettäessä. Suurimman osan uusista ideoista ja vaatimuksista ehdin sisällyttää sovellukseen, mutta ne vaatimukset joita ei sisällytetty opinnäytetyöhön toteutetaan jatkokehityksessä. Opinnäytetyön puitteissa syntynyt sovellus tarjoaa erinomaisen pohjan jatkokehitystä ajatellen.

Asiasanat
jatkuva integrointi, ohjelmistokooste, rajapinta, analytiikka, Javascript

Sisällys

1 Johdanto ... 1
2 Käsitteet ... 2
2.1 Versionhallinta ... 2
2.2 Ohjelmistokooste ... 3
2.3 Jatkuva integrointi ... 3
2.4 Julkaisuputki ... 4
3 Sovelluskehykset ja teknologiat ... 5
3.1 Node.js ... 5
3.1.1 Callback-funktiot ... 5
3.1.2 Paketinhallinta ... 6
3.2 React ... 6
3.2.1 Komponentit ... 6
3.2.2 Tila ... 7
3.2.3 Muuttujat ... 7
3.2.4 JSX ... 7
3.2.5 Virtuaalinen DOM ... 8
3.3 Webpack ... 9
3.4 Babel ... 10
3.5 Docker ... 11
4 Sovelluksen suunnittelu ... 12
4.1 Sovelluslogiikan suunnittelu ... 13
4.2 Tietokantamalli ... 14
5 Web-sovelluksen toteutus ... 16
5.1 Tiedonhaku ... 16
5.1.1 Tietojen käsittely ... 16
5.1.2 Tietojen tallennus ... 20
5.2 Rajapinta ... 23
5.3 Tietokantakyselyt ... 25
5.4 Käyttöliittymä ... 26
5.4.1 Näkymät ... 27
5.4.2 Komponentit ... 28
6 Julkaisu ... 29
7 Pohdinta ... 31
Lähteet ... 33
Liitteet ... 35

1 Johdanto

Tämä opinnäytetyö on tehty toimeksiantona Accountor Finago Oy -ohjelmistoyritykselle. Yritys tarjoaa taloushallinnon ohjelmistoja, Procountoria ja Tikonia, yrityksille ja tilitoimistoille pääsääntöisesti pilvipohjaisina SAAS-palveluina. Finagon pääkonttori sijaitsee Espoon Keilaniemessä.

Opinnäytetyön tavoitteena oli luoda web-sovellus ohjelmistokehityksen jatkuvan integroinnin statistiikan visualisointiin. Jatkuvalla integroinnilla tarkoitetaan ohjelmistokehityksessä yleisesti käytettyä käytäntöä, jonka mukaan kehittäjien kirjoittamaa ohjelmistokoodia pyritään liittämään osaksi jaettua tietolähdettä mahdollisimman usein, jopa useita kertoja päivässä. Jaetulla tietolähteellä tarkoitetaan ohjelmistokehityksessä usein versionhallinnan päähaaraa. Jatkuvaan integrointiin liittyy myös olennaisesti automatisoidut työvaiheet kuten lähdekoodin kääntäminen, testaaminen ja julkaisu kehityspalvelimelle. Projektin alkaessa toimeksiantajalla ei ollut olemassa tapaa nopeasti nähdä jatkuvan integroinnin automatisoitujen työvaiheiden keston ja häiriöajan kehitystä eri aikaväleiltä. Juuri tätä ongelmaa pyrin opinnäytetyöni produktilla helpottamaan. Produktin lisäksi opinnäytetyön tavoitteena oli myös tutustuttaa itseäni Node.js-ympäristön ja React-sovelluskehyksen käyttöön ja toimintaan.

Opinnäytetyön sovelluksen tehtävänä on kuvata jatkuvan integroinnin työvaiheiden keston ja häiriöajan kehitystä taulukolla ja viivakuvaajilla. Kehitystä täytyy olla mahdollista seurata eri mittaisilta jaksoilta ja useista eri julkaisuputkista. Julkaisuputkella tarkoitetaan jatkuvan integroinnin automatisoitujen työtehtävien ryhmää, jotka suoritetaan vaiheittain.

Raportissani käsitellään ensin opinnäytetyön kannalta keskeiset käsitteet ja käytännön toteutuksessa käytetyt teknologiat. Produktin suunnitteluun tutustutaan niin käyttöliittymän visuaalisen ulkoasun, tietokannan kuin sovelluslogiikankin osalta, jonka jälkeen syvennytään itse toteutuksen keskeisimpiin piirteisiin. Lopuksi esitellään sovelluksen julkaisukuntoon saattaminen ja yhteenveto saavutetuista tuloksista.

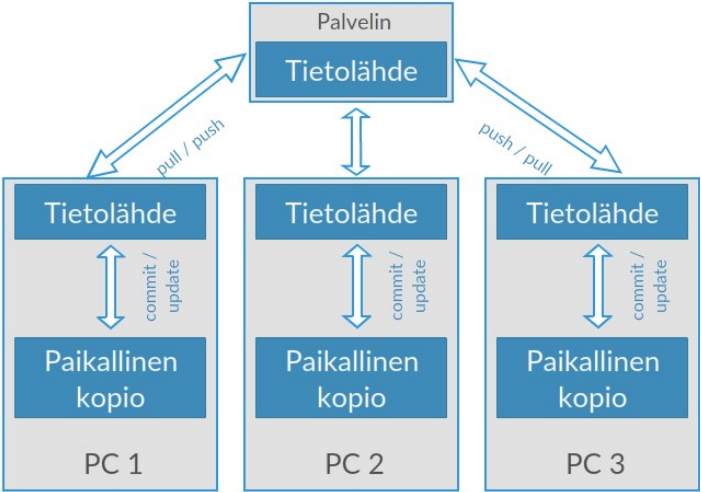
2 Käsitteet

Opinnäytetyössäni keskeisimpiä käsitteitä ovat versionhallinta, ohjelmistokooste, jatkuva integrointi ja julkaisuputki. Seuraavissa luvuissa avaan näiden käsitteiden merkitystä tarkemmin.

2.1 Versionhallinta

Ohjelmistokehityksessä versionhallinnalla tarkoitetaan järjestelmää, joka pitää kirjaa tiedostoihin tehdyistä muutoksista niin, että eri versiot ovat tunnistettavissa myöhemmin. Nämä tiedostot voivat olla mitä vain ohjelmistoissa tavanomaisesti käytettyjä tiedostoja, kuten koodi- ja asetustiedostoja (Git, 2019). Olennaista versionhallinnan toiminnassa on, että muutoksia voidaan seurata pitkälle menneisyyteen tarkasti, kuten esimerkiksi milloin jokin tiedosto on muuttunut ja kenen toimesta. Ohjelmisto voidaan palauttaa versionhallinnan avulla aikaisempaan tilaan, esimerkiksi virheen sattuessa. Tunnetuimpia ohjelmistokehityksen versionhallintajärjestelmiä ovat mm. Git, Subversion ja CVS. Käytin opinnäytetyössäni Git-versionhallintajärjestelmää, sillä se on jo yleisesti käytössä toimeksiantajalla.

Aikaisemman kuvauksen perusteella voisi ymmärtää, että versionhallinta on rinnastettavissa kehitettävän työn tallentamiseen eri tiedostoon jokaisella tallennuskerralla, jolloin vanhempi versio on edelleen erikseen avattavissa. Ohjelmistokehityksessä laajasti käytetyt hajautettu ja keskitetty versionhallinta kuitenkin palvelevat toistakin tarkoitusta: monen samanaikaisen työntekijän yhteistä työskentelyä. Hajautetussa versionhallinnassa versionhallintajärjestelmä säilyttää versioidut tiedostot keskitetysti palvelimella, josta käyttäjät voivat hakea itselleen aina viimeisimmän version kaikista tiedostoista historioineen oman työnsä pohjaksi (Michael Ernst, 2018). Tietolähdettä, joka sisältää viimeisimmän yhteisen version tiedostoista kutsutaan versionhallinnan päähaaraksi.



Kuva 1: Hajautetun versionhallinnan flow-kaavio (mukaillen Michael Ernst, 2018).

Versionhallinnalla on isoja etuja, sillä se mahdollistaa monen samanaikaisen henkilön työskentelyn samojen tiedostojen parissa. Samanaikainen työskentely tapahtuu usein haaroittamalla (engl. branching) päähaarasta oma erillinen haara. Tällöin kaikista tiedostojen viimeisimmistä versioista luodaan uusi haara palvelimelle, jonka käyttäjä voi hakea omalle koneelleen paikalliseksi kopioksi työnsä pohjaksi (Dan Radigan, 2019). Tähän kopioon tehdyt muutokset voidaan esimerkiksi tallentaa palvelimelle, josta ne voidaan yhdistää osaksi päähaaraa. Hajautetun versionhallinnan toimintalogiikkaa avataan kuvassa 1.

2.2 Ohjelmistokooste

Ohjelmistokoosteella (engl. build) tarkoitetaan kehitettävän ohjelmiston käännettyä, itsenäistä ja muista sovelluksen kehitystyössä vaadituista työkaluista riippumatonta versiota (Techopedia, 2019). Kääntämisellä tarkoitetaan ohjelmiston lähdekoodin kääntämistä tietokoneen prosessorille ymmärrettävään muotoon, useimmiten binäärikoodiksi. Tähän käytetään ohjelmointikielen omaa kääntäjää, eikä koostamisprosessi onnistu, jos kääntäjä havaitsee virheen lähdekoodin syntaksissa. Koostamisprosessiin voi liittää lisäksi myös mm. testausautomaatiota ja tiedostojen pakkaamista toiseen formaattiin.

Koostamisprosessi itsessään voidaan myös automatisoida, jolloin koostamisprosessin vaiheet ovat toistettavissa ilman ihmisen käsittelyä ja kaikki koostamiseen vaadittavat konfiguraatiot löytyvät ohjelmiston lähdekoodista (Agile Alliance, 2019). Automatisoitu koostamisprosessi on vaatimus jatkuvan integroinnin tehokkaalle hyödyntämiselle, mutta sen lisäksi se vähentää virhemarginaalia huomattavasti verrattuna manuaalisten vaiheiden suorittamiseen koostamisprosessissa (Martin Fowler, 2006).

2.3 Jatkuva integrointi

Jatkuva integrointi (engl. Continuous Integration, CI) on ohjelmistokehityksen yleinen käytäntö, joka tarkoittaa kehittäjän oman työn liittämistä osaksi jaettua tietolähdettä usein, mahdollisesti useita kertoja päivässä. Versionhallinta ja jatkuva integrointi toimivat hyvin usein yhdessä ja tällöin tietolähteellä tarkoitetaan versionhallinnan päähaaraa (Thinksys, 2017). Jatkuvan integroinnin käytännön mukaisesti kehittäjän haaran lähdekoodi koostetaan ja testataan ennen kuin se voidaan yhdistää osaksi päähaaraa. Mikäli tämä onnistuu, haarat voidaan yhdistää toisiinsa.

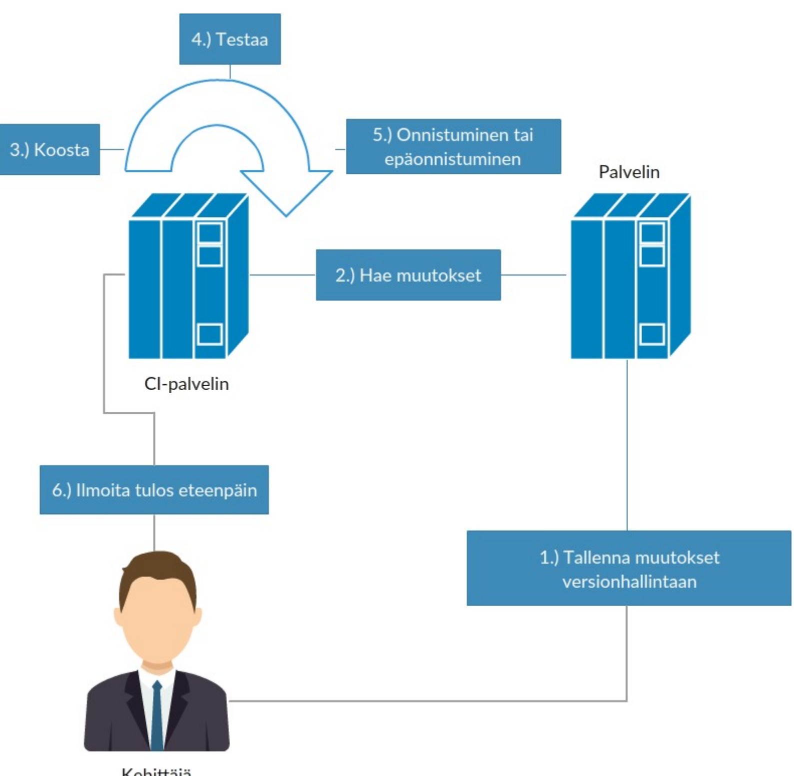
Nimensä mukaisesti jatkuva integrointi pyrkii integroimaan yksittäisten kehittäjien työtä osaksi päähaaraa. Jatkuvan integroinnin avulla pyritään saavuttamaan vakaa ohjelmisto, jonka mahdolliset virheet tulevat mahdollisimman aikaisessa vaiheessa esiin. Tämän varmistamiseksi jatkuva integrointi tarvitseekin tuekseen testausautomaatiota (Martin

Fowler, 2006). Testausautomaatioon voi kuulua esimerkiksi yksikkö-, integraatio- ja regressiotestejä, jotka ovat kaikki ohjelmoituja ajettavaksi automaattisesti ja itsenäisesti (Juhana Huotarinen, 2016).

Jatkuvan integroinnin toiminta perustuukin näin ollen erillisiin työvaiheisiin. Koostaminen, testaaminen ja julkaisu palvelimelle ovat kaikki erillisiä työvaiheita, joista voidaan muodostaa CI-palvelimelle julkaisuputki. Tämän opinnäytetyön puitteissa CI-palvelimella tarkoitetaan palvelinta, jossa ajetaan Jenkins-ohjelmaa.

2.4 Julkaisuputki

Julkaisuputki (engl. pipeline) on jatkuvan integroinnin työtehtävien ryhmä, jotka suoritetaan vaiheittain. Tehtävän onnistuessa julkaisuputki siirtyy seuraavaan, kunnes kaikki vaiheet ovat onnistuneesti suoritettu (Jenkins, 2019). Mikäli jokin tehtävä epäonnistuu, seuraavaa vaihetta ei useimmiten suoriteta ja tehtävien ajo keskeytyy.



Kuva 2: Jatkuvan integroinnin periaatteen flow-kaavio (mukaillen Thinksys, 2017).

Kuvan 2 kaaviossa julkaisuputkena nähdään palvelimen suorittamat työvaiheet ”Koosta”, ”Testaa” ja ”Onnistuminen tai epäonnistuminen”. Kaikkien vaiheiden on onnistuttava, jotta julkaisuputki määrittelee ajon onnistuneeksi. Kuvan mukaisesti, jos julkaisuputki epäonnistuu joko koostamis- tai testausvaiheessa, julkaisuputki keskeytyy ja tieto epäonnistuneesta ajosta voidaan lähettää eteenpäin.

3 Sovelluskehykset ja teknologiat

Tämän luvun tarkoituksena on esitellä opinnäytetyön käytännön toteutuksessa käytettyjä teknologioita.

3.1 Node.js

Node on Ryan Dahlin vuonna 2009 kehittämä avoimeen lähdekoodiin perustuva runtime-ympäristö koodin suorittamiseen palvelimella, joka käyttää Javascript-ohjelmointikieltä. Perinteisesti Javascript on mielletty front end -kehityksen ohjelmointikieleksi, mutta Node mahdollistaa sen laajamittaisen käytön palvelinohjelmoinnissa. Node on toimintalogiikaltaan oletusarvoisesti asynkroninen, jolla tarkoitetaan sitä, että ohjelmisto kykenee jatkamaan koodin suorittamista, vaikka aikaisempi vaihe ei ole vielä valmistunut (Node.js, 2019). Esimerkki alla Noden asynkronisesta toiminnasta:

- Node-sovellus tekee kyselyn tietokantaan.
- Sovellus käsittelee seuraavaa koodiriviä.
- Kun tietokantakysely on valmis, haetut tiedot palautetaan käyttäjälle.

Useimmista ohjelmointikielistä poiketen Node toimii yhdellä esteettömällä säikeellä (engl. thread). Tällöin tehtäviä, joiden valmistuminen voi kestää jonkin aikaa, ei jäädä odottamaan. Vertailuksi alla perinteisemmän synkronisen ohjelmointikielen logiikka vastaaville työvaiheille.

- Sovellus tekee kyselyn tietokantaan.
- Sovellus odottaa kyselyn valmistumista.
- Kun tietokantakysely on valmis, haetut tiedot palautetaan käyttäjälle.
- Sovellus käsittelee seuraavaa koodiriviä.

3.1.1 Callback-funktiot

Noden asynkroninen luonne vaatii suoritettujen tehtävien tietojen palauttamisen käyttäjälle tai takaisin ohjelmiston muuttujiin callback-funktioina. Callback-funktio useimmiten annetaan toiselle funktiolle parametrinä, jolloin callback-funktio suoritetaan välittömästi, kun sitä edeltävä funktio on valmistunut (Sebastian Lindström, 2017). Esimerkiksi tietokantakyselyn tapauksessa funktiolle, joka tekee kyselyn tietokantaan, voidaan antaa parametrinä callback-funktio, joka lähettää kyselyn tulokset käyttäjälle. Node hyödyntää callback-funktioita jatkuvasti, sillä suunnitellusti sen arkkitehtuuri ei jää odottamaan yksittäisen

funktion valmistumista, vaan suorittaa callback-funktion, kun edeltävä funktio on valmistunut.

3.1.2 Paketinhallinta

Node itsessään ei ole sovelluskehys, vaan ympäristö, johon on luotu vuosien saatossa lukuisia erilaisia sovelluskehyksiä. Näitä sovelluskehyksiä ja muita paketteja voidaan ladata Noden virallisen paketinhallintajärjestelmän, Node Package Managerin (NPM), kautta (W3Schools, 2019). Avoimen lähdekoodinsa ja paketinhallintajärjestelmän helppokäyttöisyyden ansiosta Node sovelluskehyksineen ovat tänä päivänä äärimmäisen suosittuja. Useimmat käyttämäni sovelluskehykset ovat ladattu nimenomaan Noden paketinhallintajärjestelmän kautta.

3.2 React

React on Facebookin kehittämä, alun perin vuonna 2013 julkaistu Javascript-kirjasto, joka on suunnattu käyttöliittymien kehittämiseen. Reactin toiminta perustuu komponentteihin, niiden tilaan (engl. state) ja niiden muuttujiin (engl. props) (React, 2019a).

3.2.1 Komponentit

React-komponentit määritellään luokkina, jotka periytetään Reactin sisäänrakennetusta Component-luokasta. Komponenttien avulla käyttöliittymä voidaan jakaa pieniin ja uudelleenkäytettäviin osiin, joita on helpompi hallita. Näillä komponenteilla on tietynlainen elinkaari. Alustusvaiheessa yleisimmät suoritetut metodit ovat:

- constructor()
  Komponentille alustetaan tila ja mahdolliset tapahtumankäsittelijät
- componentDidMount()
  Metodi ajetaan välittömästi, kun komponentti on asetettu DOM-puuhun. Mahdolliset verkkokutsut on hyvä tehdä tässä.
- render()

React-komponentin on aina suoritettava render() metodi, joka yksinkertaisuudessaan käsittelee komponentin käsittelemät muuttujat ja palauttaa komponentin sen näytettävässä muodossa. Tämä metodi ajetaan myös, jos komponentti päivittyy, eli sen tilaa tai muuttujia päivitetään. Kun komponenttia päivitetään, elinkaaren mukaisesti suoritetaan componentDidUpdate() metodi (React, 2019b).

Kun komponentti poistetaan DOM-puusta, eikä sitä enää tarvita, komponentin elinkaari päättyy. Tällöin suoritetaan componentWillUnmount() metodi, jossa voidaan muun muassa poistaa ajastuksia ja peruuttaa verkkokutsuja.

3.2.2 Tila

React-komponenttien tilalla tarkoitetaan komponentin yksityisiä, vain itsensä hallinnoimia muuttujia. Reactissa on tilan hallinnalle oma ”state” objekti, jota voidaan päivittää setState() metodilla. Tila ja muuttujat ovat molemmat hyvin samankaltaisia, sillä ne molemmat ovat tavallisia Javascript-objekteja, jotka sisältävät tietoja vaikuttaen renderöinnin lopputulokseen, mutta erona on muuttujien objektin siirtäminen parametrinä komponentille, kun taas tilan arvoja hallitaan komponentin sisällä (React, 2019c). Alla esimerkki komponentin konstruktorista, jossa määritetään sekä muuttujat että tila:

```
constructor(props) {
   super(props)
   this.state = {date: new Date()}
}
```

Kuva 3: React-komponentin konstruktori

Esimerkin konstruktorissa määritetään muuttujat, joka hyödyntää periytetyn React.Component luokan konstruktoria. Sen lisäksi alustetaan ”state” objekti, joka saa tässä tapauksessa yhden attribuutin: attribuutin ”date” arvo on nykyinen päivämäärä. Muuttujat saatiin parametrinä, tila asetettiin komponentin sisällä.

3.2.3 Muuttujat

Muuttujat ovat tallennettuna tilan tapaan tavalliseen Javascript-objektiin, mutta niiden arvot ovat muuttumattomia. Muuttujien arvoa ei voi siis suoraan vaihtaa komponentin sisällä, mutta sitä varten attribuutteja voidaan tallentaa komponentin tilaan.

3.2.4 JSX

JSX on Reactin käyttämä XML-tyyppinen syntaksi Javascript-ohjelmointikielelle. Se ei ole vaadittu syntaksi Reactin käyttämiselle, mutta se selkeyttää Reactin käyttöä ja mahdollistaa tarkempien virheviestien antamisen virhetilanteissa. Alla JSX-syntaksilla kirjoitettu vakio, joka tulostaisi renderöinti vaiheessa sivulle otsikon tekstillä ”Hello, world!”:

```
const element = (
  <h1 className="header">
    Hello, world!
  </h1>
)
```

Kuva 4: JSX-syntaksilla kirjoitettu vakio

Selaimelle tämä kääntyy tavalliseksi Javascriptiksi, esimerkiksi Babel-kääntäjän avulla. Tavallisella Javascriptillä kyseinen vakio näyttäisi tältä:

```
const element = React.createElement(
  'h1',
  {className: 'header'},
  'Hello, world!'
)
```

Kuva 5: Javascriptillä kirjoitettu vakio

React lukee luodut komponentit ja vakiot ja luo niiden perusteella DOM-puun päivittäen sitä tarpeen mukaan.

3.2.5 Virtuaalinen DOM

DOM, eli Document Object Model on mallinnus jäsennellystä tekstistä HTML-sivua varten. HTML-sivu itsessään on vain tekstiä, kun taas DOM-puu on muistiin tallennettu mallinnus kyseisestä tekstistä. Tutkitaan esimerkkinä seuraavaa muuttujan alustusta:

```
var user = document.getElementById("user")
```

Kuva 6: Muuttujan alustus Javascriptissä

Rivillä luodaan muuttuja, jonka arvoksi tallennetaan HTML-elementti, jonka ID:ksi on määritetty ”user”. Tätä elementtiä ei haeta varsinaisen HTML-sivun tekstistä, vaan muistiin tallennetusta DOM-puusta. Esimerkissä ”document” on DOM-puun mallinnus HTML-sivun juuresta ja ”getElementById” on HTML DOM API:n sisäinen metodi.
DOM-puun ongelma on sen hitaus internet-sivujen kasvaessa kokoa ja tullen koko ajan monimutkaisemmiksi. Kun tapahtumankäsittelijä laukaistaan sivulla, täytyy ensin etsiä

jokainen HTML-elementti, jonka arvo voisi muuttua tapahtumasta ja sen jälkeen päivittää kaikki elementit tarpeen mukaan.

Tätä ongelmaa React pyrkii ratkaisemaan virtuaalisen DOM-puun avulla. Koska varsinainen DOM-puu on jo mallinnus HTML-sivusta, virtuaalinen DOM-puu on itse asiassa mallinnus mallinnuksesta. Kun tapahtumankäsittelijä laukaistaan sivulla joka on kehitetty Reactilla, React renderöi sivun uudelleen render() metodilla. Tällöin renderöitävä komponentti asetetaan virtuaaliseen DOM-puuhun ja muutokset tarkistetaan varsinaista DOM-puuta vastaan. Virtuaalisen DOM-puun päivittäminen ja muutosten tarkistaminen varsinaisesta DOM-puusta on huomattavasti nopeampaa kuin vain varsinaisen DOM-puun käsittely (Codecademy, 2019). Muutosten löytymisen jälkeen React päivittää varsinaisen DOM-puun vastamaan virtuaalisen DOM-puun tietoja. Lopputulos on sama kuin varsinaista DOM-puuta käytettäessä, mutta tietojen käsittely ja päivittäminen tapahtuu nopeammin.

3.3 Webpack

Webpack on kehitystyökalu, jonka avulla voi paketoida Javascript-tiedostoja ja muita moduuleja yhteen tiedostoon. Kun sovellus paketoidaan Webpackin avulla, se tarkistaa tiedostojen riippuvuudet (importit Javascript-tiedostoissa) ja luo riippuvuuskaavion näiden pohjalta. Riippuvuuskaavion ja määriteltyjen asetusten pohjalta Webpack luo yhden tai useamman optimoidun tiedoston projektiin, joka sisältää useiden tiedostojen tiedot (Webpack, 2019).

Käytännössä tämä tapahtuu niin, että Webpackille määritetään projektista aloituspiste (engl. entrypoint). Kun sovelluksen paketoiminen aloitetaan, Webpack aloittaa aloituspisteestä ja tarkistaa, mistä kaikista tiedostoista tämä tiedosto on riippuvainen. Tämä jatkuu tiedostosta tiedostoon, kunnes Webpack on selvittänyt tarkalleen mitä tiedostoja tarvitaan ja missä järjestyksessä. Tästä syntyy riippuvuuskaavio ja optimoitu bundle-tiedosto.

Tämä helpottaa kehittäjän työtä huomattavasti, kun työskennellään Javascriptin parissa. Alla esimerkkinä yhden skriptitiedoston tuonti HTML-sivulle:

```
<head>
  <script src="scripts.js"></script>
</head>
```

Kuva 7: Skriptitiedoston tuonti HTML-sivulle

Tämä lähestymistapa toimii edelleen, mutta internet-sivujen kasvaessa ja web-kehityksen laajentuessa yksittäisistä tuonneista muodostuu hankalasti ylläpidettävä kokonaisuus. Jos skriptitiedostoja on kymmeniä, on jokainen tuotava erikseen ja sen lisäksi pidettävä huolta, että ne tuodaan oikeassa järjestyksessä.

Kun sovelluksen tiedostot on pakattu Webpackin avulla, tuo yksi pakattu ja optimoitu tiedosto voidaan tuoda sivulle kuvan 7 mukaisesti, mutta se sisältää itsessään kaiken mitä sivusto tarvitsee toimiakseen, automaattisesti oikeassa järjestyksessä.

3.4 Babel

Babel on Javascript-kääntäjä jota useimmiten käytetään uudemmalla Javascript-versiolla kirjoitetun koodin kääntämiseen taaksepäin yhteensopivaan versioon selaimia varten. Babel kääntää syntaksia ja paikkaa puuttuvia ominaisuuksia kohdeympäristön kyvyssä käsitellä koodirivejä (Babel, 2019). Alla esimerkki uudemmalla Javascript-versiolla (ES2015) kirjoitetusta map-funktiosta:

```
arrayOfInts.map((value) => value + 1)
```

Kuva 8: ES2015-syntaksi

Yllä olevan esimerkin mukaan arrayOfInts (joka on lista kokonaislukuja) käydään läpi yksitellen ja palautetaan jokaisen listan arvon kohdalla kyseinen arvo, johon on lisätty yksi. Alla sama funktio vanhemmalla Javascript-versiolla (ES5):

```
arrayOfInts.map(function(value) {
   return value + 1
})
```

Kuva 9: ES5-syntaksi

Babelin avulla web-kehittäjät voivat kirjoittaa kuvan 8 mukaista uudemman Javascript-version syntaksia, mutta selaimelle se käännetään kuvan 9 mukaiseksi vanhemman version syntaksiksi.

3.5 Docker

Docker on työkalu, jonka avulla voidaan tehokkaasti virtualisoida sovelluksia ja niiden ajoympäristöjä. Dockerin pääkäsitteet jakautuvat kuvaan (engl. image) ja konttiin (engl. container). Kuvalla tarkoitetaan virtuaalikoneen asetuksia, kuten käytettävää käyttöjärjestelmää tai tietokantaa, kun taas kontti on yksittäinen ajettava instanssi kuvasta. Yhdestä kuvasta voi olla samaan aikaan monta konttia.

Dockerin hyödyt syntyvät juurikin siitä, että ajettavan sovelluksen lisäksi voidaan määrittää tarkasti ajoympäristö. Käyttöjärjestelmän versio, käytettävä tietokanta ja muiden osa-alueiden versiot ovat määritetty kuvassa. Tällöin jokainen instanssi käyttäytyy identtisesti niiltä osin, mitä kuvan asetuksissa on määritetty (Ekaterina Novoseltseva, 2017).

Tämän lisäksi Dockerin kontit ovat helposti siirrettävissä koneelta toiselle, sillä ne toimivat omassa virtuaalikoneessaan. Kehittäjä voi kehittää sovellusta ja ajaa sitä omalla tietokoneellaan kontissa, jonka voi lopulta siirtää sellaisenaan vaikkapa tehokkaammalle koneelle. Kontti ja sen sisältämä sovellus käyttäytyy täysin samalla tavalla molemmissa tilanteissa. Samasta syystä Docker konttien jakaminen toisten ihmisten kesken on helppoa.

Tämän opinnäytetyön toimeksiannon produkti on automatisoitu niin, että koodimuutokset käynnistävät automaattisesti prosessin, jossa sovellus asennetaan uuteen instanssiin määritetystä kuvasta ja ladataan palvelimelle. Lopputuloksena syntyy julkaisuvalmis kontti, joka sisältää kaiken mitä sovellus tarvitsee toimiakseen.

4 Sovelluksen suunnittelu

Sovelluksen suunnittelu alkoi toukokuun alussa, kun istuimme alas Finagon kehitysjohtajien kanssa tekemään kevyttä vaatimusmäärittelyä ja projektisuunnitelmaa. Tulevan websovelluksen olisi tarkoitus hakea Jenkins API:n kautta ohjelmistokoosteiden tietoja eri julkaisuputkista niin, että niistä voidaan laskea koostamisprosessien keskimääräistä suorituskestoa ja virhetilanteiden keskimääräistä kestoa. Virhetilanteiden kestolla (jäljemmin häiriöaika) tarkoitetaan sitä aikaväliä, kun julkaisuputki epäonnistuu ja sitten onnistuu seuraavan kerran.

Suunnittelun pohjalta ensimmäinen tekemäni käyttöliittymän mockup on nähtävissä liitteessä 1.

Liitteen 1 mockupin vasemmalla puoliskolla on kaksi paneelia, joista ylempi näyttää julkaisuputkia, joiden onnistumisprosentti on korkea, kun taas alempi vastaavasti niitä, joiden onnistumisprosentti on alhainen. Valmiissa työssä tämä puolisko esittää vain yhtä paneelia, ilman erityistä järjestystä. Listarakenteisen paneelin näyttämät tiedot sovittiin seuraavanlaisiksi:

- Julkaisuputken nimi
- Keskimääräinen häiriöaika
- Onnistumisprosentti
- Keskimääräinen suorituskesto

Näkymän oikea puolisko tarjoaisi kaksi erilaista viivakuvaajaa. Ylempi näyttäisi keskimääräisen häiriöajan kehityksen valitulta aikaväliltä. Alempi taas näyttäisi vastaavan kuvaajan keskimääräisen suorituskeston kehityksestä.

Näkymiä tulisi olla useampia tukemaan eripituisia ajanjaksoja tarkkailulle, ainakin tiedot kuluvan kuukauden ja kuluvan vuoden ajalta. Monet julkaisuputket on myös haaroitettu ja näistä päätimme tarkkailla viimeisintä kehityshaaraa ja versiojulkaisujen kehityshaaroja.

Jenkins API:n kautta saatavat tiedot julkaisuputkien tuloksista tulisi olla mahdollisimman ajantasaisia ja myös itse näkymä tulisi päivittää tasaisin väliajoin automaattisesti. Näin sovelluksen voisi jättää päälle esimerkiksi yrityksen seinänäyttöihin kiinnitettyihin tietokoneisiin niin, että data päivittyy jatkuvasti ilman manuaalista päivittämistä. Keskustelussa myös asetettiin tavoitteeksi, että projektin puitteissa valmistuu sovelluksen käyttöliittymä, backend-logiikka ja tietokantaratkaisu.

4.1 Sovelluslogiikan suunnittelu

Ennen sovelluslogiikan suunnittelun aloitusta aloin tutkimaan mahdollisia sovelluskehyksiä ja teknologioita, joita käyttäisin työssäni. Käyttöliittymän kehittämiseen valikoitui React, sillä se toimii alustana React-vis visualisointikirjastolle, jota suunnittelin käyttäväni viivakuvaajien muodostamiseen. Lisäetuna Reactin hyödyntämä virtuaalinen DOM-puu nopeuttaisi sovelluksen jatkuvaa renderöintiä uusien tietojen hakemisen yhteydessä. Reactia on käytetty yrityksessä jo muihinkin projekteihin, joten mahdollista ylläpitoa ja jatkokehitystäkin ajatellen se oli luonteva valinta.

Node valikoitui palvelinohjelmointiympäristöksi sen paketinhallintatyökalun vuoksi, sillä se mahdollisti muiden tarvittavien moduulien lataamisen ja hyödyntämisen sujuvasti. Node on myös ollut yleisesti käytössä monissa React-sovelluksissa, joten tiedon haun ja mahdollisten ongelmien selvittämisen helpottamiseksi se sopi erinomaisesti projektini tarpeisiin.

Tiedon ajantasaisuuden takaamiseksi aloin pohtimaan erilaisia ajastusratkaisuja. Suunnittelin ajastavani uusien tietojen haun node-cron moduulilla. Moduuli perustuu GNU:n crontabiin ja mahdollistaa toimintojen ajastamisen käyttäen crontabin syntaksia ilman komentoriviä, suoraan lähdekoodissa. Käyttöliittymän päivitykseen niin, että uudet tiedot tulevat myös näkyviin, olin suunnitellut käyttäväni Ajaxia, joka tekniikkana on yhdistelmä selaimen sisäänrakennetun XMLHttpRequest-objektin käsittelyä ja DOM-puun päivittämistä. Lopullisessa versiossa tämä vaihtui Javascriptin sisäänrakennetun fetch-rajapinnan käyttöön, koska se palveli ajastuksen tarkoitusta yhtä hyvin ja oli minulle tutumpi. Suunnittelin tiedot päivitettäväksi kymmenen minuutin välein, sekä käyttöliittymälle että tietokantaan.

Tietokannaksi valitsin MySQL:n, sillä se on jo laajasti käytössä yrityksessä. Tietokannan osalta ei projektin puitteissa ollut kummempia vaatimuksia, joten en käyttänyt aikaa erilaisten tietokantaratkaisujen vertailemiseen.

Sovelluslogiikkaa suunnitellessani jaoin ensin sovelluksen ajatuksen pienempiin osiin. Alusta asti oli selkeää, että sovellus koostuisi käyttöliittymästä, REST-rajapinnasta, tietokannasta ja ajastetuista toiminnoista niin Jenkins API:n suuntaan kuin käyttöliittymälläkin. Suunnitteluni pohjalta muodostin vuokaavion, joka kuvaa sovelluslogiikkaa. Kaavio on nähtävissä liitteessä 2.

Liitteen 2 kaaviolla on kuvattu, kuinka käyttöliittymä hakee Ajax-kutsun avulla jatkuvasti viimeisintä dataa palvelimelta. Palvelin hakee tällöin tietokannasta viimeisimmät tiedot ja palauttaa ne vastauksena käyttöliittymälle näytettäväksi. Samaan aikaan palvelimella on erillinen ajastettu node-cron funktio, joka hakee tasaisin väliajoin Jenkins API:sta viimeisimmät tiedot, laskee tiedoista tallennettavat arvot ja tallentaa ne tietokantaan.

4.2 Tietokantamalli

Suunniteltu tietokanta koostui neljästä taulusta. Tietokantahakujen selkeyttämiseksi päätin tallentaa erillisiin tauluihin päivä-, kuukausi ja vuosikohtaisen tiedon julkaisuputkien tilasta. Näin tekemällä tietokantakyselyt olivat huomattavasti yksinkertaisempia, sekä tämä mahdollisti valmiiksi laskettujen sarakkeitten tallentamisen aikakohtaisesti eri tauluihin. Neljännen taulun tehtävä on pitää kirjaa jokaisen mitatun julkaisuputken edellisestä käsitellystä ohjelmistokoostenumerosta ja sen statuksesta. Tietokantani luokkamallit näyttävät seuraavilta:

Sarake	Tietotyyppi	Kuvaus
id	INT	Rivin yksilöllinen tunniste.
job	VARCHAR	Julkaisuputken nimi.
pipeline	VARCHAR	Julkaisuputken haaran nimi.
totalduration	INT	Kaikkien tämän julkaisuputken haaran ohjelmistokoosteiden yhteenlaskettu suorituskesto millisekunteina.
avgduration	TIME	Keskimääräinen suorituskesto esitettävässä muodossa, hh:MM:ss.
totalbuilds	INT	Tämän julkaisuputken kaikkien ohjelmistokoosteiden yhteenlaskettu summa.
successfulbuilds	INT	Onnistuneiden ohjelmistokoosteiden yhteenlaskettu summa.
ratio	INT	Ohjelmistokoosteiden onnistumisprosentti.
cycles	INT	Lukumäärä siitä, kuinka monesti julkaisuputki ollaan saatu häiriötilasta takaisin onnistuneeksi.
downtime	INT	Julkaisuputken yhteenlaskettu häiriöaika millisekunteina.
avgfixtime		Keskimääräinen häiriöaika esitettävässä muodossa, hh:MM:ss.
date	DATE	Päivämäärä, jonka ajalta tiedot ovat.

Kuva 10: jenkins_daily_data, jenkins_monthly_data ja jenkins_yearly_data -tietokantataulut

Kolme ensimmäistä taulua ovat rakenteeltaan täysin samanlaiset, mutta niihin tallennetaan eri aikaväleiltä tietoa.

Sarake	Tietotyyppi	Kuvaus
id	INT	Rivin yksilöllinen tunniste.
job	VARCHAR	Julkaisuputken nimi
pipeline	VARCHAR	Julkaisuputken haaran nimi.
lastsavedbuild	INT	Tämän julkaisuputken haaran edellisen tallennetun ohjelmistokoosteen järjestysnumero.
failed	ENUM	Tieto siitä, onko edellisen tallennetun ohjelmistokoosteen status onnistunut vai ei. Mahdolliset arvot true ja false.
failedtimestamp	BIGINT	Edellisen epäonnistuneen ohjelmistokoosteen suorituspäivä ja kellonaika millisekunteina, laskettu epochista.

Kuva 11: jenkins_last_saved_builds -tietokantataulu

5 Web-sovelluksen toteutus

Tämän luvun tarkoitus on kuvata opinnäytetyöni produktin toteutusta, eri osa-alueita ja niiden toimintaa.

5.1 Tiedonhaku

Tiedonhaku Jenkins API:n kautta tapahtuu ajastetulla node-cron funktiolla, joka käynnistetään server.js-tiedostossa. Sovelluksen käynnistysskripti ”npm start” ajaa tämän kyseisen tiedoston, joten ajastin on aina päällä, kun sovelluskin on käynnistetty.

```
cron.schedule('*/10 * * * *', function () {
  request(JENKINS_API_URL, function (error, response, body) {
    if (!error && response.statusCode === 200) {
      responseHandler.handleResponse(body)
    } else {
      errorHandler.printRequestError(response.statusCode, body)
    }
  })
})
```

Kuva 12: Node-cron funktio

Ajastin on ajastettu ajamaan joka kymmenes minuutti. Funktio tekee kutsun Accountor Finagon Jenkins-ohjelman rajapintaa vastaan ja jos kutsu onnistuu, annetaan saadut tiedot parametrinä responsehandler.js-tiedoston handleResponse funktiolle. Tässä vaiheessa tiedot sisältävät JSON-muodossa kaikki Jenkinsistä löytyneet julkaisuputket ja linkit päätepisteisiin, josta niiden tarkemmat tiedot löytyvät. Responsehandler on vastuussa tietojen parsimisesta ja käsittelystä tallennettavaan muotoon.

5.1.1 Tietojen käsittely

Responsehandler-tiedosto vastaa tietojen käsittelystä. Handleresponse funktio parsii ensin JSON-parametrin muuttujaan tavalliseksi Javascript-listaksi. Tästä listasta otetaan erilliseen muuttujaan ”jobs”-objekti, joka sisältää kaikki löytyneet julkaisuputket, niiden nimet ja linkit päätepisteisiin. Funktiossa aloitetaan foreach-silmukka, joka käy yksitellen läpi projektiin tallennetun listan monitoroitavista julkaisuputkista. Tämä lista sisältää jokaisen julkaisuputken nimen ja sen haarat, joista haluan tallentaa tietoa tietokantaan. Foreach-silmukassa kutsun filterMonitoredJobs funktiota, jolle annan parametrinä käsiteltävän monitoroitavan julkaisuputken tiedot ja aikaisemmin mainitun listan kaikista Jenkins-

ohjelmasta löytyneistä julkaisuputkista. Tämä toistuu siis jokaiselle monitoroitavien julkaisuputkien listan objekteille.

```
/**
 * Parses the JSON containing every job found from Jenkins.
 * Starts a loop iterating over every specified job in the config file.
 * @param body
 */
function handleResponse (body) {
   var json = JSON.parse(body)
   var allJobs = json.jobs
   monitoredJobs.forEach(function (monitoredJob) {
     filterMonitoredJobs(monitoredJob, allJobs)
   })
 }
```

Kuva 13: handleResponse-funktio

Kuvan 14 filterMonitoredJobs-funktiossa tarkastan, löytyykö monitoroitavaksi asetettu julkaisuputki kaikkien löytyneiden julkaisuputkien joukosta. Jos julkaisuputki löytyy, kutsun uudelleen Jenkins-ohjelman API:a päätepisteestä, joka löytyy julkaisuputken objektin ”url” muuttujasta. Tästä päätepisteestä saan tiedot julkaisuputken eri haaroista ja siirrän tiedot parametrinä seuraavalle funktiolle, joka on filterPipelineData. Tämä funktio saa lisäksi parametrinä myös monitoroitavan julkaisuputken tiedot, jotka ovat niitä tietoja, jotka on tallennettu projektiin.

```
* @param monitoredJob
 * @param allJobs
 */
function filterMonitoredJobs (monitoredJob, allJobs) {
   var foundJob = allJobs.find(job => job.name === monitoredJob.name)
   if (foundJob == null) {
     console.error('Monitored job ' + monitoredJob.name + ' was not found!')
     return
   }
   request(asJsonUrl(foundJob.url), function (error, response, body) {
     if (isValid(error, response)) {
       filterPipelineData(body, monitoredJob)
     } else {
       errorHandler.printRequestError(response.statusCode, body)
     }
   })
 }
```

Kuva 14: filterMonitoredJobs-funktio

Seuraavaksi filterPipelineData-funktiossa parsin taas julkaisuputken tiedot JSON-muodosta Javascript listaksi. Tästä listasta tallennan erilliseen muuttujaan kaikki löytyneet julkaisuputken haarat, jotka löytyvät hieman hämäävästi myös "jobs"-nimisestä muuttujasta. Funktiossa käynnistetään myös uusi foreach-silmukka, joka käy läpi kaikki monitoroitavan julkaisuputken seurattavat haarat. Tässä silmukassa annetaan vuorollaan jokainen määritetty seurattava haara ja kaikkien löytyneiden julkaisuputkien lista filterJobPipelines funktiolle.

```
/**
 * Parse the JSON containing all the pipelines of a single job.
 * Start a loop iterating over every specified pipeline for this job
 * in the config file.
 * @param body
 * @param monitoredJob
 */
function filterPipelineData (body, monitoredJob) {
   var jobData = JSON.parse(body)
   var allPipelines = jobData.jobs
   var pipelines = monitoredJob.pipelines
   pipelines.forEach(function (pipeline) {
     filterJobPipelines(pipeline, allPipelines)
   })
 }
```

Kuva 15: filterPipelineData-funktio

Kuvan 16 filterJobPipelines-funktiossa pyrin etsimään määritetyn seurattavan haaran kaikkien löytyneiden julkaisuputken haarojen listasta. Jos määritetty seurattava haara on "release", etsin löytyneiden haarojen listasta viimeisimmän versiojulkaisuhaaran. Muussa tapauksessa etsin kaikkien löytyneiden haarojen listasta monitoroitavaa haaraa suoraan nimiä vertaamalla. Jos etsittävä haara löytyy, kutsun julkaisuputken haaran "url"-objektiin tallennettua päätepistettä muutamalla lisäehdolla: haluan tiedon haaran nimestä, nimestä, johon on yhdistettynä julkaisuputken nimi ja haaran nimi ja listan kaikista julkaisuputken ohjelmistokoosteista. Tähän listaan haluan sisällyttää koosteiden järjestysnumeron, keston millisekunteina, lopputuloksen ja päivämäärän millisekunteina laskettuna epochista. Tämän kutsun tiedot annan parametrinä readBuildData-funktiolle.

```
/**
 * Checks that the specified pipeline is found for the job in Jenkins.
 * If the pipeline is found, proceed to read build data from it.
 * In the case of release pipelines, only read build data from the latest one.
 * @param monitoredPipeline
 * @param allPipelines
 */
function filterJobPipelines (monitoredPipeline, allPipelines) {
  var foundPipeline
  switch (monitoredPipeline) {
    case 'release':
      foundPipeline = findLatestReleasePipeline(allPipelines)
      break
    default:
      foundPipeline = allPipelines.find(
        pipeline => pipeline.name === monitoredPipeline
      )
      break
  }
  if (foundPipeline == null) {
    console.error('Monitored pipeline ' + monitoredPipeline + ' was not found!')
    return
  }
  request(asJsonTreeUrl(foundPipeline.url), function (error, response, body) {
    if (isValid(error, response)) {
      readBuildData(body)
    } else {
      errorHandler.printRequestError(response.statusCode, body)
    }
  })
}
```

Kuva 16: filterJobPipelines-funktio

ReadBuildData-funktio on vastuussa muutamasta eri toiminnosta ja on luettavissa liitteessä 4. Funktiossa haetaan tietokannasta tämän julkaisuputken haaran edellinen tallennettu ohjelmistokooste. Listasta, joka sisältää haaran kaikkien koosteiden tiedot, tiputetaan pois kaikki koosteet, joiden järjestysnumero on pienempi kuin kannasta löytyneen edellisen koosteen järjestysnumero. Näin yhdenkään koosteen tietoja ei käsitellä kahdesti. Lisäksi tietokannasta haetaan tämän edellisen tallennetun koosteen status, eli oliko se onnistunut vai ei. Tässä vaiheessa tallennan tietokantaan myös käsiteltävän listan viimeisimmän koosteen järjestysnumeron viimeisimmän käsitellyn koosteen tietokannan kolumniin seuraavaa ajoa varten. Lopuksi annan parametrinä findDistinctDates funktiolle julkaisuputken ja sen haaran nimen, listan tallennettavia koosteita tietoineen ja tiedon edellisen tallennetun koosteen statuksesta.

Seuraavaksi tallennettavien koosteiden listasta tehdään findDistinctDates funktiossa kolme uutta listaa: listat, jossa on koosteet, jotka kuuluvat samalle päivälle, kuukaudelle ja vuodelle. Nämä listat iteroidaan foreach-silmukassa niin, että rivi kerrallaan kutsutaan save-funktiota, joka tallentaa tiedot oikeaan tietokantatauluun. Esimerkiksi jos alkuperäinen lista sisältää koosteiden tietoja viikon ajalta saman kuukauden sisältä, tulisi päiväkohtaiseen tietokantatauluun 7 riviä, yksi kutakin päivää kohden. Kuukausi- ja vuosikohtaiseen tauluun tulisi vain yksi rivi, olettaen ettei ajankohdille olisi jo riviä tietokannassa.

5.1.2 Tietojen tallennus

Save-funktio ensin käsittelee ohjelmistokoosteiden tiedot prepareDataForSaving-funktion avulla, joka on nähtävissä liitteessä 5. Funktiossa alustetaan ensin objekti, jossa alussa kaikki mitattavat arvot ovat asetettu nollaan. Tieto siitä, onko tämän julkaisuputken haaran edellinen tallennettu kooste epäonnistunut vai ei, tallennetaan myös objektin muuttujaan päivämäärineen ja kellonaikoineen. Tämän jälkeen koosteita aletaan käymään läpi yksi kerrallaan seuraavalla logiikalla:

1.) Jos kooste on onnistunut, kuvasta 17 nähtävässä funktiossa lisätään alustetun objektin muuttujaan koosteen kesto millisekunteina ja kerrytetään onnistuneiden koosteiden lukumäärää yhdellä. Jos onnistunutta koostetta edelsi epäonnistunut kooste, tallennetaan objektin muistiin yksi sykli (tapahtumaketju, jossa epäonnistunut kooste ollaan saatu seuraavan kerran onnistuneeksi), kerrytetään häiriöajan pituutta vähentämällä käsiteltävän koosteen aikaleimasta edellisen epäonnistuneen koosteen aikaleima ja asetetaan tieto siitä muistiin, että edellinen kooste ei ollut enää epäonnistunut. Tieto edellisen koosteen tilasta tallennetaan tässä välissä myös tietokantaan, seuraavia suorituskertoja varten.

```
async function handleSuccessfulBuild (build, data) {
  if (data.lastFailed) {
    await saveFailedToSuccessCycle(build.timestamp, data)
  }
  data.totalDuration += build.duration
  data.successfulBuilds++
}

/**
 * When a previously failing build is run successfully,
 * count the downtime between the first failed attempt and
 * the successful one.
 * @param buildTimestamp
 * @param data
 */
function saveFailedToSuccessCycle (buildTimestamp, data) {
  data.cycles++
  data.downTime += buildTimestamp - data.lastFailedBuildTimestamp
  data.lastFailed = false
  data.lastFailedBuildTimestamp = null
  DBAccessor.updateFailedBuildInfo(
    data.job,
    data.pipeline,
    data.lastFailed,
    data.lastFailedBuildTimestamp
  )
}
```

Kuva 17: Onnistuneen koosteen käsittelyyn liittyvät funktiot

2.) Jos kooste on epäonnistunut, tallennetaan objektin muistiin tieto siitä, että edellinen kooste on epäonnistunut ja jos edellisen epäonnistuneen koosteen aikaleimaa ei ole saatavilla, asetetaan siihen tämän koosteen aikaleima. Näin tekemällä varmistun siitä, että jos epäonnistuneita koosteita on ollut peräkkäin useampia, muistissa on aina sarjan ensimmäisen epäonnistuneen koosteen aikaleima, jota lopulta käytetään häiriöajan laskemiseen. Nämä toimenpiteet suorittava funktio on nähtävissä kuvassa 18.

```
async function handleFailedBuild (build, data) {
  data.lastFailed = true
  if (data.lastFailedBuildTimestamp === null) {
    data.lastFailedBuildTimestamp = build.timestamp
    await DBAccessor.updateFailedBuildInfo(
      data.job,
      data.pipeline,
      data.lastFailed,
      data.lastFailedBuildTimestamp
    )
  }
}
```

Kuva 18: handleFailedBuild-funktio

Tiedot epäonnistuneesta koosteesta tallennetaan myös tietokantaan, seuraavia suorituskertoja varten.

Tämän jälkeen koosteista lasketut tiedot akkumuloidaan tietokannassa jo olevien tietojen kanssa. Saman päivän aikana samasta julkaisuputken haarasta voi tulla tietoa monta kertaa, joten haluan vain päivittää tietokannassa jo olemassa olevaa päiväkohtaisen tietokantataulun riviä. Kuukausikohtaiseen tietokantatauluun tulee harvemmin uusia rivejä ja vuosikohtaiseen sitäkin harvemmin. Kun mahdolliset tiedot on akkumuloitu, tietoja aletaan tallentamaan tietokantaan.

Ennen tietokantaan tallentamista lasken tietokannan sarakkeisiin valmiiksi näytettävässä muodossa olevia arvoja, joita ovat:

Arvo	Laskentalogiikka
Keskimääräinen kesto	Yhteenlaskettu kesto / onnistuneet koosteet
Onnistumisprosentti	Onnistuneet koosteet / kaikki koosteet x 100
Keskimääräinen häiriöaika	Yhteenlaskettu häiriöaika / syklit

Kuva 19: Näytettävässä muodossa tallennetut arvot

Seuraavaksi lasketut tiedot tallennetaan tietokantaan. Jos ajankohtaa vastaavaa rivi löytyy jo tietokantataulusta, vanhaa riviä vain päivitetään uusilla lasketuilla tiedoilla.

5.2 Rajapinta

Rajapinta on kytketty sovellukseen server.js-tiedoston määrityksessä niin, että kaikki kutsut sovellukseen polun "/api" kautta käsitellään rajapinnan kontrollerissa. Rajapinnan päätepisteet listattuna alla:

- /api
  - /month
    - /:version (etsittävän julkaisuputken haara)
    - /graphdata
      - /:version
  - /year
    - /:version
    - /graphdata
      - /:version
  - /releases

Tiedon haku viimeisen kuukauden sisältä tehdään siis polusta /api/month/:version. Tällöin kontrolleri ensin tarkistaa viimeisen kuukauden, jolta dataa löytyy vielä valitulle julkaisuputken haaralle ja palauttaa tietokannasta tämän kuukauden osalta kaikkien julkaisuputkien tiedot, joilta löytyy kysytty haara. Täysin samalla periaatteella toimii myös vuosikohtaisen tiedon hakeminen, polusta /api/year/:version, mutta tarkistaen tietenkin kuukauden sijasta vuoden. Näistä päätepisteestä haetut tiedot ovat lista julkaisuputkia, joilla on seuraavat tiedot: julkaisuputken nimi, keskimääräinen häiriöaika, onnistumisprosentti, keskimääräinen kesto ja päivämäärä/ajanjakso jolle tieto kuuluu. Näitä tietoja käytetään taulukkonäkymän muodostamiseen käyttöliittymälle.

Polun /api/releases kautta voidaan hakea listaus kaikista tietokantaan tallennetuista versiojulkaisujen haaroista. Kun tietoja haetaan Jenkins API:sta, haetaan aina viimeisimmän versiojulkaisuhaaran tietoja, jos sellainen on määritetty. Tämä tarkoittaa sitä, että tietokantaan kerrytetään ajan kuluessa monien eri julkaisuversioiden tietoja.

Viivakuvaajille voidaan hakea tiedot /api/month/graphdata/:version ja /api/year/graphdata/:version päätepisteiden kautta. Nämäkin päätepisteet ensin tarkistavat viimeisimmän kuukauden tai vuoden jolta dataa on näytettäväksi. Kuukausikohtaisten tietojen kohdalla kannasta haetaan päiväkohtaisten tietojen taulusta tiedot viimeisen 30 päivän ajalta, vuosikohtaisten tietojen kohdalla taas kuukausikohtaisten tietojen taulusta viimeisen 12 kuukauden ajalta. Molemmissa tapauksissa tietokantahaku palauttaa listan

julkaisuputkia, jossa on seuraavat tiedot: päivämäärä, yhteenlaskettu suorituskesto millisekunteina, onnistuneiden koosteiden lukumäärä, yhteenlaskettu häiriöaika millisekunteina ja syklien lukumäärä.

Tämä lista parsitaan kontrollerissa niin, että ensin listan perusteella tehdään uusi lista päivämääriä, joka sisältää kaikki alkuperäisestä listasta löytyneet päivämäärät. Tämä päivien lista iteroidaan foreach-silmukassa, joka kerta hakien alkuperäisen listan kaikki julkaisuputket, joiden päivämäärä on sama kuin silmukan tietyllä hetkellä käsittelemä päivämäärä. Kuvan 20 countGraphValues-funktiossa akkumuloidaan tämän listan julkaisuputkien tiedot yhteen niin, että yhdelle päivälle saadaan laskettua kaikkien julkaisuputkien koosteiden keskimääräinen kesto ja keskimääräinen häiriöaika.

```
/**
 * Accumulates the data belonging to a same date together,
 * returning the accumulated average duration and fix time for a given date.
 * @param data
 * @param day
 */
function countGraphValues (data, day) {
   var duration = 0
   var downTime = 0
   var successfulBuilds = 0
   var cycles = 0
   data.forEach(row => {
     duration += row.totalduration
     downTime += row.downtime
     successfulBuilds += row.successfulbuilds
     cycles += row.cycles
   })
   return {
     date: new Date(day).getTime(),
     avgDuration: toFixed(duration / successfulBuilds),
     avgFixTime: toFixed(downTime / cycles)
   }
}
```

Kuva 20: countGraphValues-funktio

Jos keskimääräinen kesto ja häiriöaika olivat laskettavissa, objekti sisältäen yhden päivän akkumuloidut tiedot lisätään funktiosta palautettavaan listaan. Foreach-silmukka toistaa tämän toimenpiteen jokaiselle löytyneelle päivämäärälle niin, että lopuksi funktio palauttaa listan kaikista päivistä akkumuloituine tietoineen.

5.3 Tietokantakyselyt

Sovelluksen tietokantakyselyt, joita rajapinnan kontrolleri kutsuu, on toteutettu Node-QueryBuilder moduulilla. Moduulin tarkoitus on tarjota kehittäjille universaali tapa yhdistää ja tehdä kyselyitä mihin tahansa tietokantaan. Käytännössä tämä tapahtuu niin, että Node-QueryBuilder noudattaa omaa syntaksiaan, jonka se automaattisesti kääntää valitulle tietokannalle sopivaksi syntaksiksi.

Kaikki sovelluksen tietokantakyselyt tehdään dbaccessor.js-tiedostossa. Tämän tiedoston alussa määritetään käytettävät tietokanta-asetukset (kuva 21). Produktissani on määritetty erilaiset tietokanta-asetukset riippuen siitä, ajetaanko sovellusta lokaalisti kehittäjän koneella vai sisäisellä palvelimella. Tämä ero tunnistetaan Noden ympäristömuuttujasta. Lisäksi tiedostossa määritetään käytettävä tietokanta ja yhteystapa.

```
import settings from '../config/db.config.json'
import production from '../config/db.config.production.json'
import db from 'node-querybuilder'

let usedSettings = process.env.NODE_ENV !== 'production' ? settings : production
const pool = db.QueryBuilder(usedSettings, 'mysql', 'pool')
```

Kuva 21: dbaccessor-tiedoston asettamat tietokanta-asetukset

Kaikki tietokantakyselyt noudattavat pitkälti samaa kaavaa. Otetaan esimerkiksi rajapinnan kontrollerin käyttämä kysely, jossa haetaan tietyn julkaisuputken haaran kuukausikohtaiset tiedot (kuva 22):

```
function fetchMonthlyData (pipeline, date, callback) {
    pool.get_connection(qb => {
      qb.select(['job', 'avgfixtime', 'ratio', 'avgduration', 'date'])
        .where('pipeline', pipeline)
        .where('date', date)
        .order_by('job')
        .get('jenkins_monthly_data', (error, response) => {
          qb.release()
          return callback(error, response)
        })
    })
}
```

Kuva 22: fetchMonthlyData-funktio

Kyselyt alkavat yhteyden muodostamisella. SELECT-lause aloitetaan määrittämällä lista haettavien kolumnien nimiä QueryBuilderin select-funktiolle. WHERE-ehtolauseet luodaan määrittämällä kaksi parametriä, joista ensimmäinen on tietokannan kolumni ja toinen vertailtava arvo. Tulosten järjestelemiseen voi käyttää order_by -funktiota. QueryBuilderin get -funktio vastaa FROM määritystä, jossa määritellään tietokantataulu josta tiedot haetaan. Samassa funktio annetaan myös callback-funktio, joka voi palauttaa joko virheen sellaisen sattuessa tai onnistuneen kyselyn tuloksen. Callback-funktiossa ensin vapautetaan luotu yhteys niin, ettei se jää auki. Lopuksi palautetaan rajapinnan kontrollerille toinen callback-funktio, joka sisältää joko virheen tai onnistuneen kyselyn tapauksessa tietokantakyselyn tulokset.

5.4 Käyttöliittymä

Sovelluksen valmis käyttöliittymä on nähtävissä liitteessä 3. Sovelluksen käyttöliittymä koostuu kuukausittaisen ja vuosittaisen tiedon näkymistä sekä julkaisuputkien päähaaralle (develop) että versiojulkaisujen haaroille (release/xx.x.x). Server.js-tiedostossa on asetettu kaikki osoitteet palauttamaan index.html sivu (kuva 23), pois lukien rajapinnan käyttämät päätepisteet.

```
<!DOCTYPE html>
<html lang="en">

<head>
   <title>CI Statistics</title>
   <!-- Latest compiled and minified CSS -->
   <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
integrity="sha384-
BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
       crossorigin="anonymous">
</head>

<body>
   <div id="root"></div>
   <script type="text/javascript" src="/bundle.js"></script>
</body>

</html>
```

Kuva 23: index.html

Webpackin luoman bundle.js tiedoston aloituspiste on index.js tiedosto, joka on nähtävissä kokonaisuudessaan liitteessä 6. Vaikka kaikki sivut avautuvat index.html -sivua käyttämällä, Webpackin aloituspisteessä määritetään mitä tietoja eri osoitteiden kautta näytetään.

Eri osoitteiden perusteella index.js renderöi oikean komponentin. Esimerkiksi liitteen 6 mukaisesti, osoite /monthly/develop renderöi MonthlyDevelop-komponentin.

5.4.1 Näkymät

Näkymäkomponentteja on neljä: näkymät kuukausittaiselle ja vuosittaiselle tiedolle julkaisuputkien päähaarasta, sekä julkaisuputkien versiojulkaisujen haaroista. Ne noudattavat pitkälti samaa logiikkaa, mutta näytettävät tiedot luonnollisesti vaihtelevat. Kuukausittaisen datan näyttäminen julkaisuputkien päähaarasta, eli MonthlyDevelop-komponentti, on nähtävissä liitteessä 7.

Liitteen 7 komponentin konstruktorissa alustetaan komponentin tilaan tyhjät listat, ”jobs” ja ”graphData”. Näistä ensimmäistä käytetään käyttöliittymän vasemman puoliskon taulukkonäkymän täydentämiseen, kun taas jälkimmäistä nimensä mukaisesti kaavioiden piirtämiseen. Metodissa componentDidMount() tehdään kutsut rajapintaan ja asetetaan sivu uusimaan nämä kutsut 10 minuutin välein. Kutsuja tehdään kaksi, toinen taulukkomuotoiselle datalle ja toinen kuvaajille. Kuvassa JOB_DATA ja GRAPH_DATA ovat komponentin lokaaleja muuttujia, jotka sisältävät kutsuttavan rajapinnan päätepisteen. Kutsun tulos asetetaan konstruktorissa määritellylle oikealle muuttujalle setState() metodilla. Jos rajapintakutsu epäonnistuu, näytetään sovelluksen sivulla virheviesti avautuvassa dialogi-ikkunassa.

Metodi componentWillUnmount() huolehtii siitä, että kun sivulta poistutaan, poistetaan myös rajapintakutsujen ajastus. Jos ajastusta ei erikseen poisteta, kutsujen ajastus jäisi päälle jokaiselle avatulle sivulle. Tämä aiheuttaisi sovellukseen muistivuotoja.

Kaikkien näkymien render() metodi renderöi Page-komponentin, johon annetaan muuttujina näytettävän sivun otsikko, taulukon otsikko, viivakuvaajien otsikko, taulukon näytettävä sisältö sekä viivakuvaajien näytettävä sisältö. Näin samaa Page-komponenttia voidaan käyttää kaikille näkymille, kun näkymäkohtaiset erot ovat muunneltavissa muuttujien kautta.

5.4.2 Komponentit

Näkymien käyttämä Page-komponentti koostuu itsessään useasta uudelleen käytettävästä komponentista. Page-komponentti on nähtävissä liitteessä 8.

NavBar-komponentti luo sivulle navigointipalkin. Komponentin muuttujissa määritetään palkissa näytettävä auki olevan sivun nimi. Itse navigointipalkin komponentissa luodaan alasvetovalikkojen sisältö, jota varten tehdään rajapintakutsu /api/releases päätepisteeseen. Tämä palauttaa listan kaikista versiojulkaisuista, joiden tietoja löytyy tietokannasta. Tämän listan avulla luodaan alasvetovalikon sisältö, linkki kullekin julkaisuputkien versiojulkaisuhaaralle. Release-komponentti, joka on nähtävissä render() metodissa, luo linkityksen rajapinnan käyttämien päätepisteiden ja niitä vastaavien komponenttien välille. Komponentin render() metodi renderöi navigointipalkin sivun nimellä ja alasvetovalikoilla varustettuna. NavBar-komponentti on nähtävissä liitteessä 9.

Page-komponentin näkymä jakautuu kahteen yhtä suureen kolumniin. Vasemmanpuoleiseen renderöidään JobList-komponentti (liite 10), joka sisältää taulukkomuotoisen datan. JobList-komponentille annetaan muuttujina näytettävä data ja julkaisuputken haaran nimi.

JobList-komponentissa luodaan paneeli, jonka sisälle luodaan taulukko. Näytettävä data muunnetaan rivi riviltä Job-komponenteiksi. Job-komponentissa määritetään taulukon solujen arvot annetun rivin perusteella. Tämä lista Job-komponentteja annetaan näytettävälle taulukolle renderöitäväksi sisällöksi.

Page-komponentin oikealle puoliskolle luodaan kaksi Chart-komponenttia (liite 11) allekkain viivakuvaajia varten. Chart-komponentille annetaan muuttujina näytettävä data, kuvaajan nimi ja näytettävien julkaisuputkien haarojen nimi. Näytettävä data sisältää listan objekteja, jotka koostuvat päivämäärästä, yhteenlasketusta kestosta ja yhteenlasketusta häiriöajasta. Kuvaajille annetaan muuttujana muunnettu lista, jossa on määritetty päivämäärät muuttujaan x ja kuvaajasta riippuen joko kesto tai häiriöaika muuttujaan y. Chart-komponentti käyttää näitä arvoja suoraan x- ja y-akselien pisteiden laskemiseen.

Kuvaajassa päivämäärät muunnetaan näytettävään muotoon displayableDate-funktiolla ja kestot millisekunneista näytettävään muotoon millisToDuration-funktiolla. Kuvaaja piirretään vain, jos datapisteitä on enemmän kuin yksi. Mikäli datapisteitä ei ole tai niitä on vain yksi, kuvaajan tilalla näytetään createNotification-metodin luoma teksti ”No data to display.”

6 Julkaisu

Sovelluksen julkaisukuntoon saattaminen lähti liikkeelle niin, että tein lähdekoodille oman tietolähteen (engl. repository) toimeksiantajan versionhallintajärjestelmään. Sovelluksen lähdekoodiin tallensin myös oman tiedoston CI-palvelinta varten, Jenkinsfilen (kuva 24).

```
pipeline {
    agent {
        kubernetes {
            label "docker-${UUID.randomUUID().toString()}"
            defaultContainer 'jnlp'
            yamlFile 'Pod.yaml'
        }
    }

    stages {
        stage('Build') {
            steps {
                container('docker') {
                    sh 'docker build -t procountor/ci-statistics:kubernetes .'
                }
            }
        }

        stage('Push to registry') {
            steps {
                container('docker') {
                    sh 'docker push procountor/ci-statistics:kubernetes'
                }
            }
        }

        stage('Clean') {
            steps {
                container('docker') {
                    sh 'docker rmi -f procountor/ci-statistics:kubernetes'
                }
            }
        }
  }
}
```

Kuva 24: Jenkinsfile

Tämän tiedoston avulla määritetään sovelluksen oman julkaisuputken työvaiheet. Työvaiheessa ”Build” sovelluksesta luodaan Docker-kuva, joka noudattaa Dockerfile-tiedostossa (kuva 25) määritettyjä ohjeita.

```
FROM node:8
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD [ "npm", "start", "--production" ]
```

Kuva 25: Dockerfile

Docker-kuvaa luodessa määritetään ensin sovelluksen ajoympäristö. Tässä tapauksessa käytössä on Dockerin tarjoama valmis ympäristö, jossa on asennettuna Noden viimeisin LTS-versio. Tämän jälkeen määritetään projektin työkansio, eli /usr/src/app. Package.json ja package-lock.json ovat molemmat tiedostoja jotka säilyttävät tiedon niistä moduuleista, joita sovellus tarvitsee toimiakseen. Ne kopioidaan molemmat paikoilleen työkansioon. Komento ”npm install” asentaa kaikki moduulit NPM-paketinhallintaohjelman kautta, joista sovellus on riippuvainen. Tämän jälkeen kopioidaan koko projekti juuresta lähtien paikoilleen työkansioon ja paljastetaan portti 3000, jotta sovellukseen saa yhteyden. Viimeinen rivi määrittää komennon, joka ajetaan, kun Docker-kuvasta luotu kontti käynnistetään. Tässä tapauksessa käynnistetään sovellus komennolla ”npm start”, lisäehdolla ”production”, jotta kontti käyttää oikeaa tietokantaa.

Työvaiheessa ”Push to registry” luotu kuva viedään toimeksiantajan Docker Hubiin, josta se on julkaistavissa haluttuun paikkaan. Työvaihe ”Clean” poistaa julkaisuputkessa luodun Docker-kuvan, sillä kun edellinen työvaihe on onnistunut, kuva on jo tallessa Docker Hubissa.

Tämän jälkeen toimeksiantajan CI-palvelimelle luotiin uusi julkaisuputki, joka käynnistyy silloin kun sovelluksen tietolähteessä havaitaan koodimuutos. Käynnistyessään julkaisuputki hakee sovelluksen viimeisimmät koodit versionhallinnasta ja ajaa Jenkinsfilessä määritetyt työvaiheet. Tällöin havaittu koodimuutos johtaa aina automaattisesti uuteen Docker-kuvaan toimeksiantajan Docker Hubissa, josta on julkaistavissa sovelluksen viimeisin versio.

7 Pohdinta

Kaiken kaikkiaan olen lopputulokseen tyytyväinen. Sovelluksesta tuli eheä ja julkaisukuntoinen kokonaisuus sisältäen käyttöliittymän, rajapinnan ja tietokannan. Produktin käytettävyys nousi myös odotusten tasolle, sillä sain sen toimimaan niin kuin alun perin suunniteltiin toimeksiantajan kehitysjohtajien kanssa. Toimeksiantajankin mielestä työ täytti asetetut vaatimukset ja oli lopputulokseen tyytyväinen.

Jatkokehitystä ajatellen ensimmäisenä tulisi kirjoittaa projektille yksikkö- ja integraatiotestit. Olen aloittanut näiden testien kirjoittamisen sovellukselle, mutta opinnäytetyön laajuuden puitteissa jätin ne raportistani pois. Testit helpottaisivat mahdollista laajempaa jatkokehitystä, josta on jo useita ideoita niin itseni kuin toimeksiantajankin puolesta, kuten näkymä julkaisuputkien automaatiotestien tuloksista ja näkymä taulukosta, josta kävisi ilmi ketkä kehittäjistä on korjannut julkaisuputkien häiriötilanteita.

Aloitin opinnäytetyöni käytännössä ilman kokemusta Noden ja Reactin käytöstä. Myös Javascriptin tuntemukseni oli aloitettaessani vähäistä. Työ avasi erinomaisesti näiden kahden teknologian käyttöä ja sitä, mihin nämä ovat kykeneviä. Aluksi oli haastavaa lähteä liikkeelle nollasta, mutta ohjelmointikokemukseni muiden ohjelmointikielien parissa helpotti uuden omaksumista. Monissa tapauksissa hahmottelin ensin editoriin aluksi koodinpätkän Java-ohjelmointikielellä, jonka jälkeen aloin tutkimaan, kuinka tämä kirjoitettaisiin muita teknologioita tehokkaasti hyödyntämällä. Sovelluksessani ei todennäköisesti ole kaikkialla käytetty parhaita mahdollisia käytäntöjä, mutta työn teko oli joka tapauksessa erittäin miellyttävää ja kehittävää, sillä koko ajan sai oppia uutta.

Suurimmat haasteet käytettyjen teknologioiden kanssa olivat Noden asynkroninen luonne ja Docker-ohjelman hyödyntäminen. Noden asynkronisuus oli jotakin, johon en ollut tottunut aikaisemmin ja aluksi se tuntuikin jopa huonolta arkkitehtuurilta. Kuitenkin työtä kehittäessäni aloin ymmärtää, kuinka hyödyntää tätä mahdollisimman tehokkaasti. Javascriptin syntaksi on edelleen mielestäni epäselvä, eikä ohjelmointikielen tyypittömyys edesauta asiaa. Docker-ohjelman käyttöön taas sain paljon apua toimeksiantajan testausautomaatio ja infrastruktuuri tiimiltä, ilman heitä olisi sovellus jäänyt todennäköisesti julkaisematta.

Teknologiahaasteiden ulkopuolelta haasteita loi myös projektin vaatimusten ja toiminnallisuuksien muuttuminen muutamaan otteeseen. Näitä loi sattuneet väärinkäsitykset ja vasta kehittäessä eteen nousseet ongelmat. Näistä kuitenkin selvittiin kohtuullisen helposti, sillä istuimme alas toimeksiantajan kehitysjohtajan kanssa kerran kuukaudessa tutkailemaan

työn nykyistä tilaa ja seuraavia kehitysvaiheita. Haastamalla toisiamme toistemme ideoilla saimme luotua ongelmiin ratkaisut, joihin molemmat olivat tyytyväisiä.
Uusien teknologioiden oppimisen lisäksi opin tätä työtä tehdessäni myös paljon lisää yleisesti ohjelmistokehityksestä. Kun työ tehtiin alusta loppuun asti vain itseni toimesta, pääsin näkemään ohjelmistotuotannon elinkaaren jokaisen vaiheen aina suunnittelusta julkaisuun asti ja kaikki näihin vaiheisiin liittyvistä huomioitavista asioista.

Opinnäytetyön tekoprosessi oli mielestäni kaiken kaikkiaan opettava ja havahduttava kokonaisuus: itse produktin luonti muistutti läheisesti varsinaista päivätyötäni toimeksiantajalla, mutta työn sanoiksi pukemisen vaikeus yllätti. Eri osa-alueiden kartoittaminen ja toiminnallisuuksien kuvailu oli yllättävän haastavaa, vaikka sovelluslogiikka oli peräisin omalta näppäimistöltä ja kirkkaana mielessä. Tätä olisi helpottanut aktiivisempi kirjoittaminen aina yhden työvaiheen valmistuttua, näin jälkikäteen ajatellen. Kaikista opinnäytetyössäni saaduista opeista viisastuneena on hyvä lähteä taas kohtia uusia haasteita niin työpaikalla kuin henkilökohtaisten ohjelmointiprojektienkin parissa.

Lähteet

Agile Alliance, 2019. Automated build. Luettavissa:
https://www.agilealliance.org/glossary/automated-build Luettu: 4.1.2019.

Babel, 2019. What is Babel? Luettavissa:
https://babeljs.io/docs/en/ Luettu: 11.2.2019

Codecademy, 2019. React: The Virtual DOM. Luettavissa:
https://www.codecademy.com/articles/react-virtual-dom Luettu: 10.10.2018

Dan Radigan, 2019. Feature branching your way to greatness. Luettavissa:
https://www.atlassian.com/agile/software-development/branching Luettu: 2.1.2019.

Ekaterina Novoseltseva, 2017. Top 10 Benefits of Docker. Luettavissa:
https://dzone.com/articles/top-10-benefits-of-using-docker Luettu: 10.1.2019

Git, 2018. 1.1 Getting Started - About Version Control. Luettavissa:
https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control Luettu: 11.10.2018.

Jenkins, 2019. Pipeline. Luettavissa:
https://jenkins.io/doc/book/pipeline/ Luettu: 4.1.2019

Juhana Huotarinen, 2016. Tiesitkö jatkuvan julkaisun olevan jo arkipäivää? Luettavissa:
https://gofore.com/tiesitko-jatkuvan-julkaisun-olevan-jo-arkipaivaa/ Luettu: 7.11.2018

Martin Fowler, 2006. Continuous Integration. Luettavissa:
https://martinfowler.com/articles/continuousIntegration.html Luettu: 11.11.2018.

Michael Ernst, 2018. Version control concepts and best practices. Luettavissa:
https://homes.cs.washington.edu/~mernst/advice/version-control.html Luettu: 16.10.2018.

Node.js, 2019. About Node.js. Luettavissa:
https://nodejs.org/en/about/ Luettu: 11.12.2018

React, 2019a. Tutorial: Intro to React. Luettavissa:
https://reactjs.org/tutorial/tutorial.html Luettu: 5.10.2018.

React, 2019b. Components and Props. Luettavissa: https://reactjs.org/docs/components-and-props.html Luettu: 6.10.2018.

React, 2019c. State and Lifecycle. Luettavissa: https://reactjs.org/docs/state-and-lifecycle.html Luettu: 7.10.2018.

Sebastian Lindström, 2017. Getting to know asynchronous JavaScript: Callbacks, Promises and Async/Await. Luettavissa: https://medium.com/codebuddies/getting-to-know-asynchronous-javascript-callbacks-promises-and-async-await-17e0673281ee Luettu: 5.1.2019

Techopedia, 2019. Build. Luettavissa: https://www.techopedia.com/definition/3759/build Luettu: 4.1.2019.

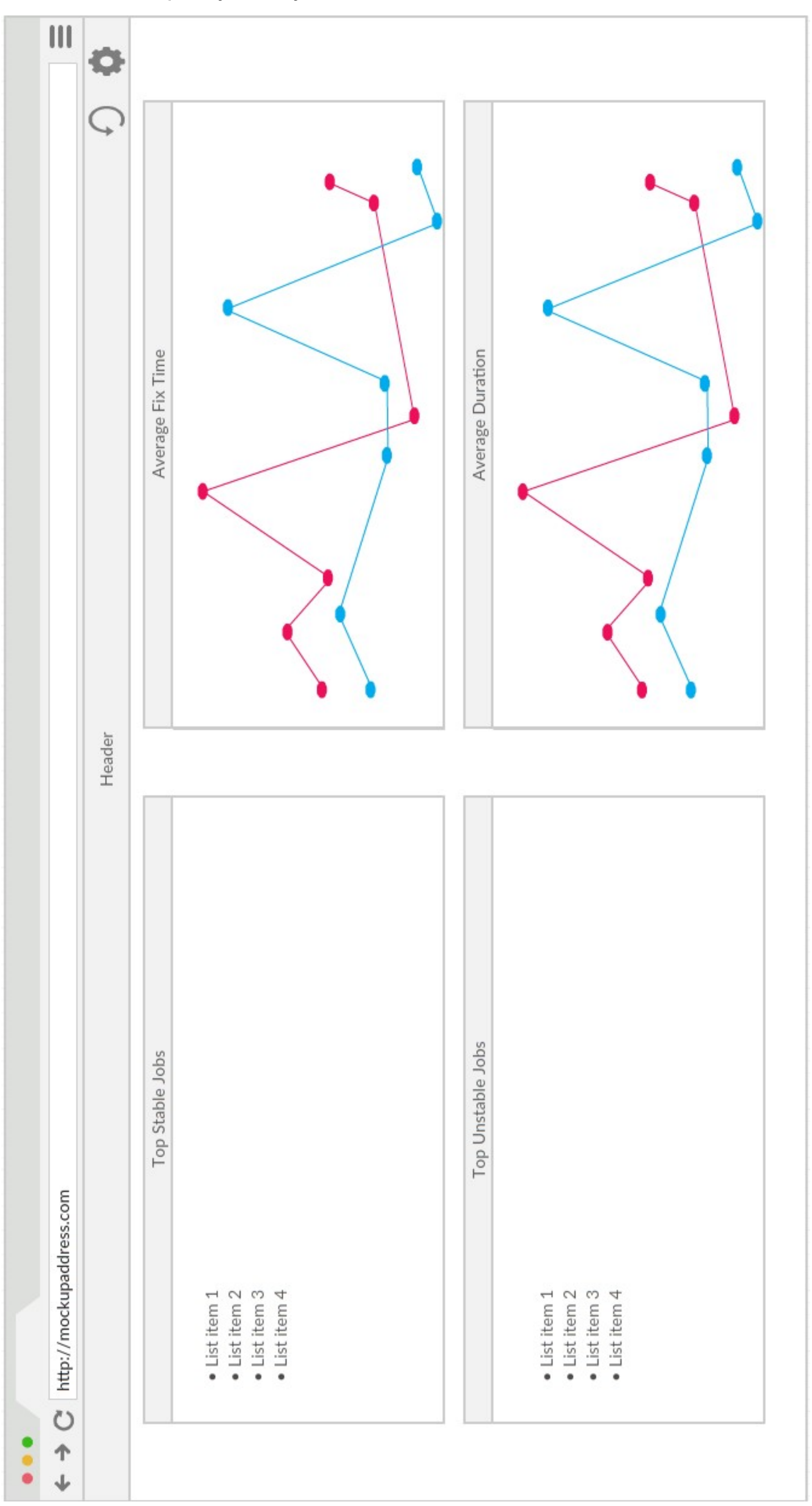
Thinksys, 2017. The Fundamentals of Continuous Integration Testing. Luettavissa: https://www.thinksys.com/qa-testing/fundamentals-continuous-integration-testing/ Luettu: 16.10.2018.

Webpack, 2019. Concepts. Luettavissa: https://webpack.js.org/concepts/ Luettu: 3.1.2019.

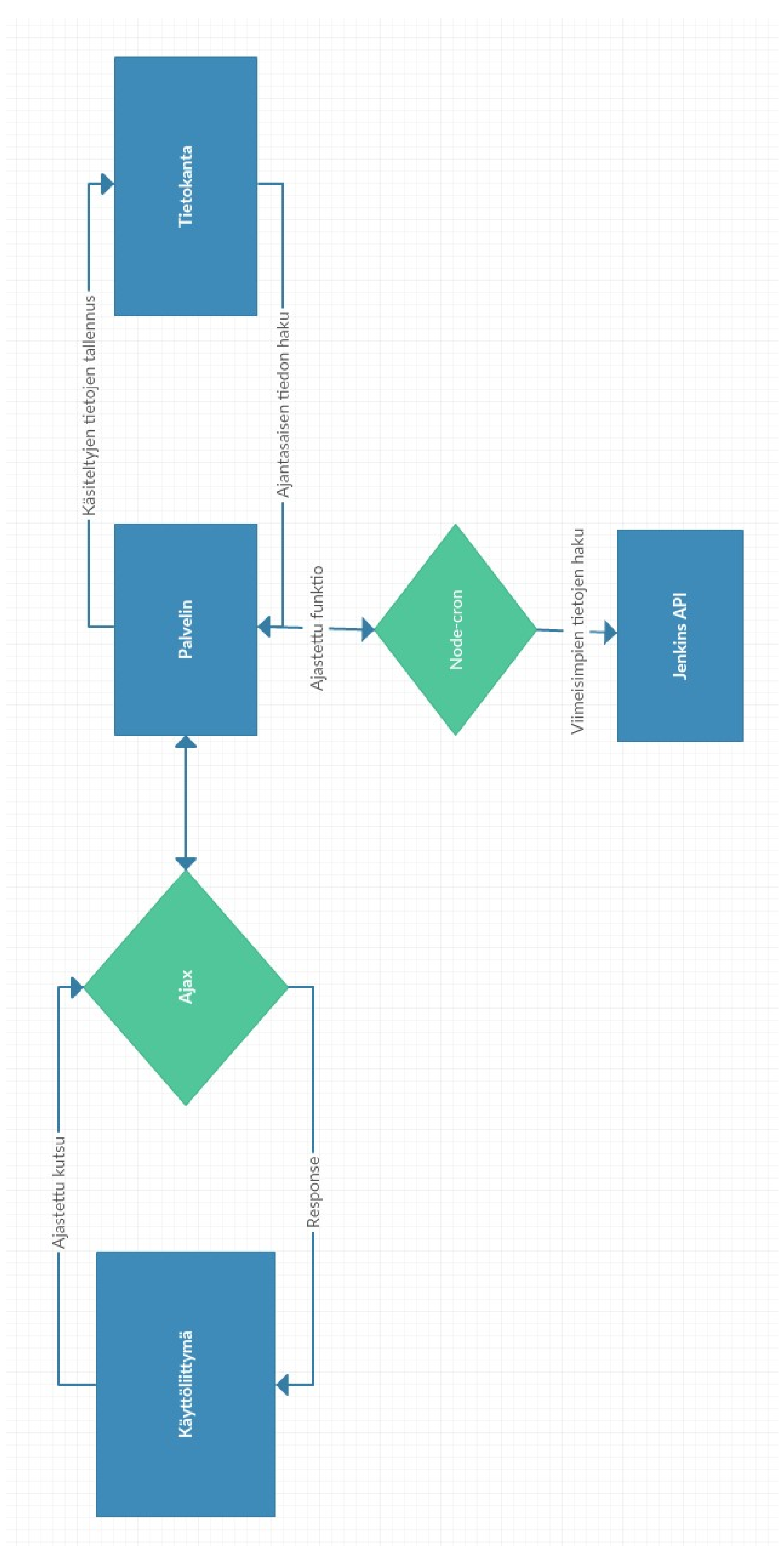
W3Schools, 2019. Node.js NPM. Luettavissa: https://www.w3schools.com/nodejs/nodejs_npm.asp Luettu: 18.1.2019.

Liitteet

Liite 1: Mockup käyttöliittymästä



Liite 2: Sovelluslogiikan vuokaavio



Liite 3: Sovelluksen valmis käyttöliittymä

Name	Average Downtime	Success Ratio	Average Duration
build-procountor	15:44:21	86%	01:02:24
build-procountor-api	N/A	100%	01:01:33
build-timers	01:17:09	96%	01:05:12
docker-timers-configs	N/A	100%	00:36:09
docker-tomcat-configs	03:03:42	80%	00:04:17
jooq-generate-and-deploy-to-nexus	N/A	100%	00:05:12
mysql-bdd-docker	N/A	100%	00:26:21
regression-integration-tests	14:43:35	10%	00:35:43
regression-test-procountor	22:07:37	0%	N/A
regression-test-procountor-api	20:20:03	0%	N/A
SchemaUpdate	N/A	100%	00:54:44

Liite 4: readBuildData-funktio

```
/**
 * Check the builds found from a single pipeline.
 * Update the database table regarding the last saved build in this
 * pipeline and proceed to handle unsaved build data.
 * @param body
 */
function readBuildData (body) {
  var json = JSON.parse(body)
  var fullName = json.fullName
  var nameSplitIndex = fullName.indexOf('/')
  var job = fullName.substring(0, nameSplitIndex)
  var pipeline = json.name
  DBAccessor.getLastSavedBuild(job, pipeline, function (error, previouslySaved) {
    if (error) console.error(error)
    var unsavedBuilds = json.builds.filter(
      build =>
        build.number > emptyToZero(previouslySaved) && buildIsFinished(build)
    )
    if (unsavedBuilds.length === 0) {
      return
    }
    var lastSavedBuildData = getStatusFromLastBuild(previouslySaved)
    var newSavedInfo = {
      job: job,
      pipeline: pipeline,
      lastsavedbuild: unsavedBuilds[0].number
    }
    if (previouslySaved.length === 0) {
      DBAccessor.insertLastSavedBuild(newSavedInfo)
    } else {
      DBAccessor.updateLastSavedBuild(newSavedInfo, previouslySaved[0].id)
    }
    findDistinctDates(job, pipeline, unsavedBuilds, lastSavedBuildData)
  })
}
```

Liite 5: prepareDataForSaving-funktio

```
/**
 * Loops through each build of a pipeline, handling both successful and failed builds.
 * Returns an object with values saveable to the database.
 * @param builds
 * @param job
 * @param pipeline
 * @param time
 */
function prepareDataForSaving (builds, job, pipeline, time, lastSavedData) {
  var data = {
    job: job,
    pipeline: pipeline,
    totalDuration: 0,
    totalBuilds: builds.length,
    successfulBuilds: 0,
    cycles: 0,
    downTime: 0,
    lastFailed: lastSavedData.failed,
    lastFailedBuildTimestamp: lastSavedData.failedtimestamp,
    date: time
  }
  builds.forEach(function (build) {
    switch (build.result) {
      case 'SUCCESS':
        handleSuccessfulBuild(build, data)
        break
      case 'FAILURE':
        handleFailedBuild(build, data)
        break
      default:
        break
    }
  })
  return data
}
```

Liite 6: index.js

```
/**
 * Start point of the application. Sets up routes for all navigation links.
 */
const Main = () => (
    <main>
      <Switch>
        <Route exact path="/" component={MonthlyDevelop} />
        <Route exact path="/monthly/develop" component={MonthlyDevelop} />
        <Route exact path="/yearly/develop" component={YearlyDevelop} />
        <Route
          exact
          path="/monthly/release/:version"
          component={MonthlyReleasePath}
        />
        <Route
          exact
          path="/yearly/release/:version"
          component={YearlyReleasePath}
        />
      </Switch>
    </main>
  )

  const MonthlyReleasePath = props => {
    return (
      <MonthlyRelease
        fullName={'release/' + props.match.params.version}
        key={props.match.params.version}
      />
    )
  }

  const YearlyReleasePath = props => {
    return (
      <YearlyRelease
        fullName={'release/' + props.match.params.version}
        key={props.match.params.version}
      />
    )
  }

  ReactDOM.render(
    <BrowserRouter>
      <Main />
    </BrowserRouter>,
    document.getElementById('root')
  )
```

Liite 7: MonthlyDevelop-komponentti

```
/**
 * Class for fetching and updating the data of MONTHLY -> develop view.
 * Creates the viewable page by the use of the Page class.
 */
export default class MonthlyDevelop extends React.Component {
  constructor (props) {
    super(props)
    this.state = {
      jobs: [],
      graphData: []
    }
  }
  componentDidMount () {
    this.fetchData()
    this.interval = setInterval(() => this.fetchData(), 600000)
  }
  fetchData () {
    fetch(JOB_DATA)
      .then(response => response.json())
      .then(responseData => {
        this.setState({ jobs: responseData })
      })
      .catch(error => alert('Error!\n' + error))

    fetch(GRAPH_DATA)
      .then(response => response.json())
      .then(responseData => {
        this.setState({ graphData: responseData })
      })
      .catch(error => alert('Error!\n' + error))
  }
  componentWillUnmount () {
    clearInterval(this.interval)
  }
  render () {
    return (
      <Page
        title={DISPLAY_NAME}
        listLabel={LIST_LABEL}
        graphLabel={GRAPH_LABEL}
        jobs={this.state.jobs}
        graphData={this.state.graphData}
      />
    )
  }
}
```

Liite 8: Page-komponentti

```
* Displayable page base class. Uses props defined in separate page views
 * and uses common components to generate a view.
 */
export default class Page extends React.Component {
  render () {
    let navBarTitle = this.props.title
    let jobData = this.props.jobs
    let graphData = this.props.graphData
    let listLabel = this.props.listLabel
    let graphLabel = this.props.graphLabel
    return (
      <div>
        <NavBar title={navBarTitle} />
        <Grid fluid>
          <Row className="show-grid">
            <Col xs={6}>
              <JobList jobs={jobData} label={listLabel} />
            </Col>
            <Col xs={6}>
              <Chart
                data={graphData.map(point => {
                  return {
                    x: point.date,
                    y: point.avgDuration
                  }
                })}
                title={TITLE_DURATION}
                label={graphLabel}
              />
              <Chart
                data={graphData.map(point => {
                  return {
                    x: point.date,
                    y: point.avgFixTime
                  }
                })}
                title={TITLE_FIXTIME}
                label={graphLabel}
              />
            </Col>
          </Row>
        </Grid>
      </div>
    )
  }
}
```

Liite 9: NavBar-komponentti

```
export default class NavBar extends React.Component {
  constructor (props) {
    super(props)
    this.state = {
      releases: []
    }
  }
  componentDidMount () {
    fetch(RELEASES)
      .then(response => response.json())
      .then(responseData => {
        this.setState({ releases: responseData })
      })
      .catch(error => alert('Error!\n' + error))
  }
  render () {
    var releaseItemsForMonthly = this.state.releases.map(release => (
      <Release key={release.pipeline} release={release} scope="/monthly/" />
    ))
    var releaseItemsForYearly = this.state.releases.map(release => (
      <Release key={release.pipeline} release={release} scope="/yearly/" />
    ))
    return (
      <Navbar fluid>
        <Navbar.Header>
          <Navbar.Brand>{this.props.title}</Navbar.Brand>
        </Navbar.Header>
        <Nav>
          <NavDropdown title="MONTHLY" id="basic-nav-dropdown">
            <LinkContainer to="/monthly/develop">
              <MenuItem>develop</MenuItem>
            </LinkContainer>
            {releaseItemsForMonthly}
          </NavDropdown>
          <NavDropdown title="YEARLY" id="basic-nav-dropdown">
            <LinkContainer to="/yearly/develop">
              <MenuItem>develop</MenuItem>
            </LinkContainer>
            {releaseItemsForYearly}
          </NavDropdown>
        </Nav>
      </Navbar>
    )
  }
}
```

Liite 10: JobList-komponentti

```
/**
 * Creates a Bootstrap table for job list and populates
 * it with Job objects mapped from JSON response.
 */
export default class JobList extends React.Component {
  render () {
    var jobs = this.props.jobs.map(job => <Job key={job.job} job={job} />)
    var label = this.props.label
    var date = new Object(this.props.jobs[0]).date
    return (
      <Panel>
        <Panel.Heading>
          <Panel.Title componentClass="h3">
            {dateUtils.displayableLongDate(date)}
            <div style={{ float: 'right' }}>
              <Label bsStyle="primary">{label}</Label>
            </div>
          </Panel.Title>
        </Panel.Heading>
        <Panel.Body>
          <Table striped>
            <thead>
              <tr>
                <th>Name</th>
                <th>Average Downtime</th>
                <th>Success Ratio</th>
                <th>Average Duration</th>
              </tr>
            </thead>
            <tbody>{jobs}</tbody>
          </Table>
        </Panel.Body>
      </Panel>
    )
  }
}
```

Liite 11: Chart-komponentti

```
export default class Chart extends React.Component {
  createNotification (dataPoints) {
    if (dataPoints <= 1) {
      return (
        <p style={{ color: 'gray', position: 'absolute' }}>
          No data to display
        </p>
      )
    }
  }
  render () {
    return (
      <Panel>
        <Panel.Heading>
          <Panel.Title componentClass="h3">
            {this.props.title}
            <div style={{ float: 'right' }}>
              <Label bsStyle="primary">{this.props.label}</Label>
            </div>
          </Panel.Title>
        </Panel.Heading>
        <Panel.Body>
          {this.createNotification(this.props.data.length)}
          <FlexibleXYPlot
            margin={{ left: 50, right: 20 }}
            height={245}
            yDomain={graphSettings.getYDomainRange(this.props.data)}
          >
            <VerticalGridLines />
            <XAxis
              tickFormat={point => dateUtils.displayableDate(point)}
              tickTotal={8}
            />
            <YAxis
              tickFormat={point => dateUtils.millisToDuration(point, false)}
              tickValues={graphSettings.getTickValues(this.props.data)}
            />
            <LineSeries
              data={this.props.data.length > 1 ? this.props.data : []}
            />
          </FlexibleXYPlot>
        </Panel.Body>
      </Panel>
    )
  }
}
```