Ville Myllykangas

Jatkuva integrointi ja pilvipalvelut Metropolian opetuksessa

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Insinöörityö

30.4.2017

Metropolia

Tekijä Otsikko Sivumäärä Aika	Ville Myllykangas Jatkuva integrointi ja pilvipalvelut Metropolian opetuksessa 33 sivua 30.4.2017
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikan Koulutusohjelma
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja	Yliopettaja Erja Nikunen

Tämän insinöörityön tavoitteena oli selvittää, voisiko pilvipalvelupohjaisia jatkuvan integroinnin työkaluja hyödyntää Metropolia Ammattikorkeakoulun jatkuvan integroinnin opetuksessa. Nykyisellään Metropolian jatkuvan integroinnin opetuksessa käytetään pelkästään Jenkins – jatkuvan integroinnin työkalua.

Tässä työssä tutkittiin Jenkinsiä ja kahta pilvipalvelupohjaista työkalua: GitLab CI:tä ja Travis CI:tä. Työssä käydään läpi näiden työkalujen yleisten ominaisuuksien ja käytön lisäksi niiden hyviä ja huonoja puolia. Työssä tutkittiin myös kojelautojen käyttömahdollisuuksia näiden jatkuvan integroinnin työkalujen kanssa.

Lopuksi tuloksista tehtiin yhteenveto, jossa kerättiin yhteen jatkuvan työkalujen hyvät ja huonot puolet. Näiden tulosten pohjalta koostettiin jatkoehdotuksia Metropolian jatkuvan integroinnin opetusta varten.

Avainsanat	Jatkuva integrointi, pilvipalvelut

Author	Ville Myllykangas
Title	Continuous Integration and Cloud Services in Teaching at Metropolia UAS
Number of Pages	33 pages
Date	30 April 2017
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Specialisation option	Software Engineering
Instructor	Erja Nikunen, Principal Teacher

The goal of this final project in engineering iwass to research if cloud based continuous integration tools could be used in Metropolia University of Applied Sciences' continuous integration teaching. Currently only the Jenkins continuous integration tool is in use in teaching.

In this project, Jenkins and two cloud based continuous integration tools were inspected. The other two cloud based tools were GitLab CI and Travis CI. The study gives general information about the three tools and how they are used in practice. In addition, the study goes through the pros and cons of the tools. Also possible usage of dashboards with the tools was researched.

Suggestions for further study and proposals on further improvement on Metropolia's continuous integration teaching are then made based on the results of the study.

Keywords	Continuous integration, cloud services

Sisällys

Lyhenteet

CI	Continous intergation. Jatkuva intergointi. Ohjelmistotuotannon käytäntö, jossa ohjelman toimivuus pyritään varmistamaan mahdollisimman usein.
TDD	Test driven development. Ohjelmistotuotannon käytäntö, jossa testit tehdään ennen varsinaista koodia.
JAR	Java Archive. Paketti, joka sisältää Java-ohjelman koodin.
JDK	Java Development Kit. Ohjelmistokehityspaketti Java-ohjelmia varten.
MIT	Massachusetts Institute of Technology. Vapaaohjelmistolisenssi, joka antaa käyttäjälle muokkaus-, kopiointi- ja käyttöoikeudet sillä ehdolla, että lisenssi säilyy lähdekoodissa.
REST	Representational state transfer. Internetin yli käytettävä rajapinta.

1 Johdanto

Tämän työn tavoite on selvittää, miten pilvipalveluihin pohjautuvia jatkuvan integroinnin työkaluja voisi hyödyntää Metropolia Ammattikorkeakoulun jatkuvan integroinnin opetuksessa ja opiskelijoiden projektien toteutuksessa. Tällä hetkellä Metropolian jatkuvan integroinnin opetuksessa käytetään pelkästään Jenkinsiä.

Tässä dokumentissa käyn ensin läpi yleisesti sen, mitä jatkuvalla integroinnilla tarkoitetaan ja miten se tapahtuu (luku 2). Seuraavaksi dokumentissa kerrotaan Jenkinsistä, jatkuvan integroinnin työkalusta, joka on tällä hetkellä käytössä Metropolian opetuksessa. Luvuissa 4 ja 5 esitellään vaihtoehtoisia jatkuvan integroinnin työkaluja, GitLab CI:tä ja Travis CI:tä. Lyhenne CI tulee jatkuvan integroinnin englanninkielisestä nimestä continous integration. Seuraavaksi tässä dokumentissa käsitellään vielä kojelautoja (kappale 6). Lopuksi luvussa 7 on yhteenveto.

Työn lähteinä on käytetty lähinnä verkkodokumentteja, mutta joitain kirjallisuuslähteitä löytyy myös. Näitä kirjallisuuslähteitä ovat Peter Smithin kirja Software Build Systems: Principles and Experience, ja Metropolian opiskelija-assistentin, Topi Kasarin, insinöörityö Jatkuvan integroinnin koontityökalut.

Menetelmä, jolla työ on tehty, on internet painotteinen tutkimustyö. Työ on kasattu yhdistämällä omaa tietämystäni edellä mainituista lähteistä saatuihin tietoihin.

2 Jatkuva integrointi

Jatkuva integrointi on ohjelmistotuotannon käytäntö, jossa sovelluksen koonti pyritään suorittamaan mahdollisimman usein, esimerkiksi tietyin aikavälein tai silloin, kun versionhallintaan tehdään muutoksia. Tämä prosessi on automaattinen, ja se tehdään jatkuvan integroinnin työkalujen avulla. Nämä työkalut ovat ohjelmia, jotka pystyvät hakemaan ohjelmien lähdekoodeja versionhallinnasta ja tekemään niiden koonteja.

Ohjelmien koonteihin liittyy yleensä koodin kääntäminen, testien ajo ja mahdollisesti staattisen analyysin työkalujen ajo. Koodin kääntäminen on tarpeen vain perinteisten

käännettävien ohjelmointikielten, kuten Javan, C#:n tai C++:n, kanssa. Se ei ole tarpeen skriptikielten kuten JavaScriptin, Php:n tai Rubyn kanssa. [1.] Kaikille ohjelmointikielelle kuitenkin löytyy testityökaluja.

Kuva 1. Jatkuva integrointi

Kuva 1 esittää jatkuvan integroinnin toimintaa. Kehittäjät vievät versionhallintaan muutoksia (development -> commit -> source control). Nämä muutokset saavat aikaan sovelluksen koonnin (initiate ci process -> build) ja testien ajon (test -> testing). Lopuksi kehittäjät käyvät testitulosraportin läpi (report) ja tekevät mahdollisesti korjauksia, jolloin kierros alkaa alusta.

2.1 Versionhallinta

Versionhallinnalla tarkoitetaan projektin tiedostojen tallentamista johonkin kaikkien projektin jäsenten käytettävissä olevaan palveluun. Tiedostoista pidetään useampia versioita tallessa, jolloin muutoksia on mahdollista peruuttaa.

Versionhallinta on tärkeä osa jatkuvan integroinnin automatisointia, koska jatkuvan integroinnin työkalujen on päästävä käsiksi ohjelman koodiin. Käytännössä se onnistuu antamalla työkalulle oikeudet päästä käsiksi projektin versionhallinnan tietovarastoon.

Tämä jatkuvan integroinnin ja versionhallinnan liittämien sallii myös joidenkin versionhallintatyökalujen kanssa sen, että versionhallinta ilmoittaa jatkuvan integroinnin työkalulle,

kun muutoksia tapahtuu tietovarastossa. Koonti voidaan silloin ajaa aina kun ohjelman koodiin on tehty muutoksia.

2.2 Kääntäminen

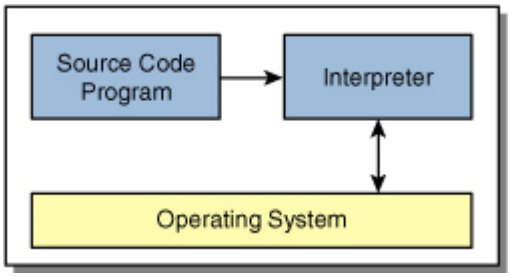
Monet ohjelmointikielet vaativat koodin kääntämisen, ennen kuin niitä on mahdollista ajaa. Kääntäminen suoritetaan koontityökalujen (esim. Ant, Maven) avulla. Kääntämisessä ohjelman koodi käännetään johonkin toiseen ohjelmointikieleen. Tämä käännös tapahtuu yleensä jostain ihmisluettavasta, korkeamman tason ohjelmointikielestä, kuten Javasta tai C#:sta, johonkin matalamman tason ohjelmointikieleen, kuten esimerkiksi konekieleen.

Jotta koontityökalut voivat kääntää ohjelman koodin, ne tarvitsevat ohjeet siitä, miten käännös on tehtävä. Tällaisia ohjeita kutsutaan koontiskripteiksi. Esimerkiksi Ant-koontia varten tarvitaan xml-muotoinen koontiskripti. Ohjelmointiympäristöt, kuten esim. NetBeans, pystyvät luomaan tällaisia scriptejä automaattisesti, mutta niiden tekeminen manuaalisesti on myös mahdollista.

Esimerkiksi C++-sovellusten lähdekoodi koostuu C++-kooditiedostoista (.cpp) ja header-tiedostoista (.h). Kääntäjä luo tämän koodin avulla ajettavissa olevan .exe-tiedoston.

Javan tapauksessa lähdekoodi koostuu Java-tiedostoista (.java). Kääntäjä luo näiden tiedostojen avulla luokka tiedostoja (.class). Nämä tiedostot paketoidaan sitten tarvittavan metadatan kanssa JAR-pakettiin (Java ARchive). Java-tulkki luo tämän paketin avulla suorittavan konekielisen ohjelman.

Kaikki ohjelmointikielet eivät vaadi ohjelman kääntämistä ennen kuin ne voidaan ajaa. Näitä kieliä kutsutaan skriptikieliksi. Skriptikielet ovat yleensä tulkittavia kieliä eivätkä käännettäviä kieliä. Tulkittavilla kielillä kirjoitettujen ohjelmien suoritus tapahtuu tulkin avulla. Tulkki on ohjelma, joka lukee ja suorittaa ohjelman koodin sen sijaan, että se kääntäisi sen. Kuvassa 2 on kaavio, joka kuvaa tulkin toimintaa. Siinä nähdään, kuinka tulkki (Interpreter) suorittaa ohjelman lähdekoodin (Source Code Program) käyttöjärjestelmässä (Operating System).

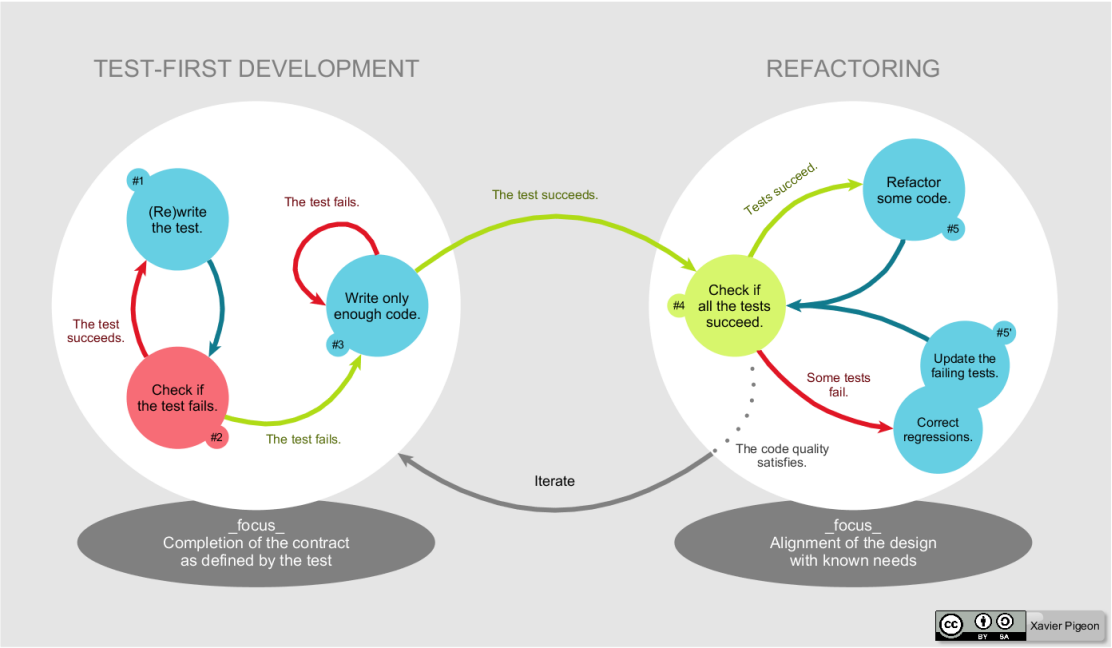


Kuva 2. Tulkin toiminta [2]

2.3 Testaus

Jotta mahdolliset virheet löytyvät ohjelmista, on niitä hyvä testata mahdollisimman huolellisesti kehityksen aikana. Pienten ohjelmien kanssa tällainen testaus on toki mahdollista tehdä manuaalisesti, mutta suurempien projektien kanssa testaus on yleensä tehtävä automaattisesti.

Suurempia ohjelmia suunniteltaessa testit on hyvä tehdä ennen varsinaisen toiminnallisuuden koodausta. Tällaista työtapaa kutsutaan testivetoiseksi kehitykseksi (eng. test driven developmentiksi). [3.]



Kuva 3. Testivetoinen kehitys

Testivetoisessa kehityksessä koodaus aloitetaan koodaamalla testit ensin. Jos testit eivät mene läpi, koodia on tehtävä vain sen verran, että ne menevät läpi. Kun kaikki testit menevät läpi, uusi koodi on hyvä refaktoroida. Refaktoroinnilla tarkoitetaan koodin siistimistä siten, että koodin toiminnallisuus pysyy samana, mutta sen sisäistä rakennetta parannetaan. Nämä parannukset voivat olla esimerkiksi toisteisen koodin poistamista, funktioiden ja luokkien uudelleen nimeämistä tai koodin sijaintipaikan muuttamista. Kuvasta 3 löytyy testivetoista kehitystä kuvaava kaavio.

Ohjelmien automaattiseen testaukseen löytyy monenlaisia valmiita kirjastoja. Esimerkiksi Java-ohjelmien testaus sujuu JUnit-kirjaston avulla. JUnit on hyvin suosittu kirjasto, mutta muita Java-testauskirjastoja ovat esimerkiksi Arquillian ja Jtest.

2.4 Staattinen analyysi

Staattisella analyysilla tarkoitetaan koodin analysointia ilman, että ohjelmaa ajetaan. Tällaisessa analysoinnissa tarkastellaan yleensä sitä, onko koodi kirjoitettu siististi ja noudattaen hyväksi todettuja koodauskäytäntöjä. Staattisella analyysilla pyritään myös löytämään koodista ns. haisuja (eng. code smell). Tällaisilla haisuilla tarkoitetaan ongelmia koodissa, jotka eivät ole varsinaisia ohjelmointivirheitä, vaan heikkouksia koodin rakenteessa.

Esimerkiksi staattisen analyysin työkalua checkstylea voidaan käyttää Java-koodin analysointiin. Se tarkastelee muun muassa koodin siisteyttä (esimerkiksi onko import-lauseiden jälkeen laitettu tyhjä rivi) ja nimeämiskäytäntöjä. [4.] Muita staattisen analyysin työkaluista ovat esimerkiksi PMD ja FindBugs.

2.5 Dynaaminen analyysi

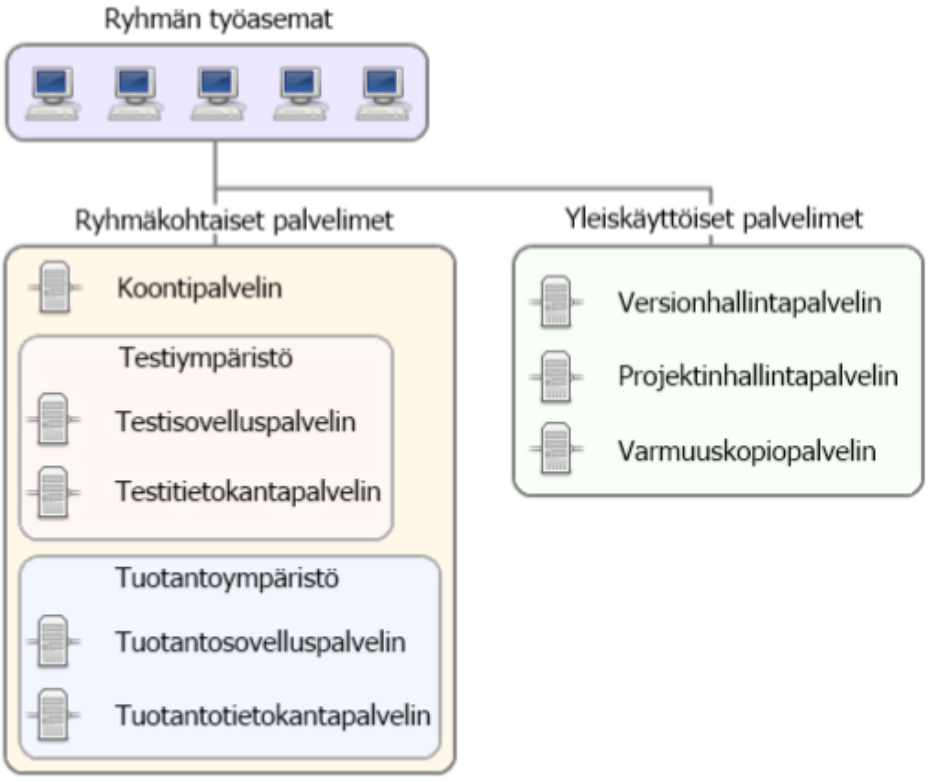
Vastakohta staattiselle analyysille on dynaaminen analyysi. Dynaamisessa analyysissa ohjelmaa tutkitaan ajon aikana. Asioita, joihin dynaamisessa analyysissa kiinnitetään huomiota, ovat esimerkiksi resurssien käyttö, ja se, missä funktioissa ajon aikana on tullut käytyä. Esimerkiksi AddressSanitizer on avoimen lähdekoodin sovellus, joka etsii muistin käyttöön liittyviä bugeja.

3 Jenkins

Tässä luvussa käsitellään tällä hetkellä (kevät 2017) Metropolian jatkuvan integroinnin opetuksessa käytettävää Jenkins-ohjelmaa. Luvussa kerrotaan ensin Jenkinsin opetuskäytöstä (luku 3.1). Sen jälkeen käydään läpi yleisiä tietoja Jenkinsistä, kuten historiaa ja teknisiä tietoja (luku 3.2). Lopuksi käydään läpi Jenkinsin hyvät ja huonot puolet (luvut 3.3 ja 3.4).

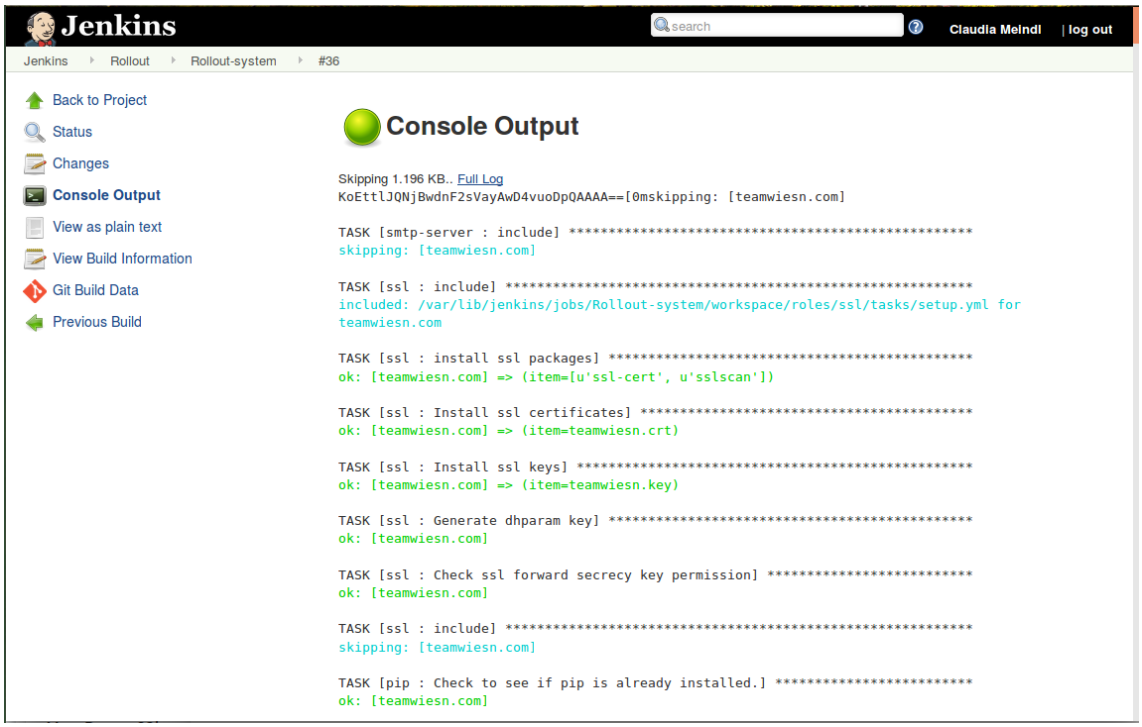
3.1 Käyttö opetuksessa

Opiskelijoiden projektien, kuten innovaatioprojektit ja ohjelmistotuotantoprojektit, koonnit tehdään Jenkinsiä käyttäen. Nämä Jenkins-sovellukset pyörivät Metropolian omissa palvelimissa, ja niiden ylläpidosta vastaa opiskelija-assistentti.



Kuva 4. Innovaatioprojektin kehitysympäristö

Kuvassa 4 on esitetty innovaatioprojektin kehitysympäristöön liittyvät koneet ja virtuaalikoneet. Kullekin ryhmälle annetaan käyttöön viisi kannettavaa tietokonetta varsinaista kehitystyötä varten. Kullekin ryhmälle on myös varattu viisi virtuaalipalvelinta: kaksi palvelinta sovelluksen ajoa varten (testi ja tuotanto), kaksi palvelinta tietokantoja varten (testi ja tuotanto) ja koontipalvelin. Näille koontipalvelimille on asennettu valmiiksi Jenkins. Lisäksi kehitysympäristöön kuuluu kolme yhteiskäyttöistä palvelinta: versionhallintapalvelin (GitLab ja Svn), projektinhallintapalvelin (Agilefant) ja varmuuskopiopalvelin. [5.]



Kuva 5. Jenkins

3.2 Yleistä

Jenkins on Javalla toteutettu jatkuvan integroinnin työkalu. Se on palvelinpohjainen ohjelma, jota on mahdollista ajaa esimerkiksi Apache Tomcatissa. Kuvassa 5 nähdään Jenkinsin konsolitulostenäkymä.

Jenkins on MIT-lisenssin alainen vapaa ohjelmisto. MIT-lisenssi on Massachusetts Institute of Technology -yliopiston mukaan nimetty vapaaohjelmisto lisenssi. [6.] Se on sisällöltään hyvin yksinkertainen, ja se antaa käyttäjälle muokkaus-, kopiointi- ja käyttöoikudet sillä ehdolla, että lisenssi säilyy lähdekoodissa.

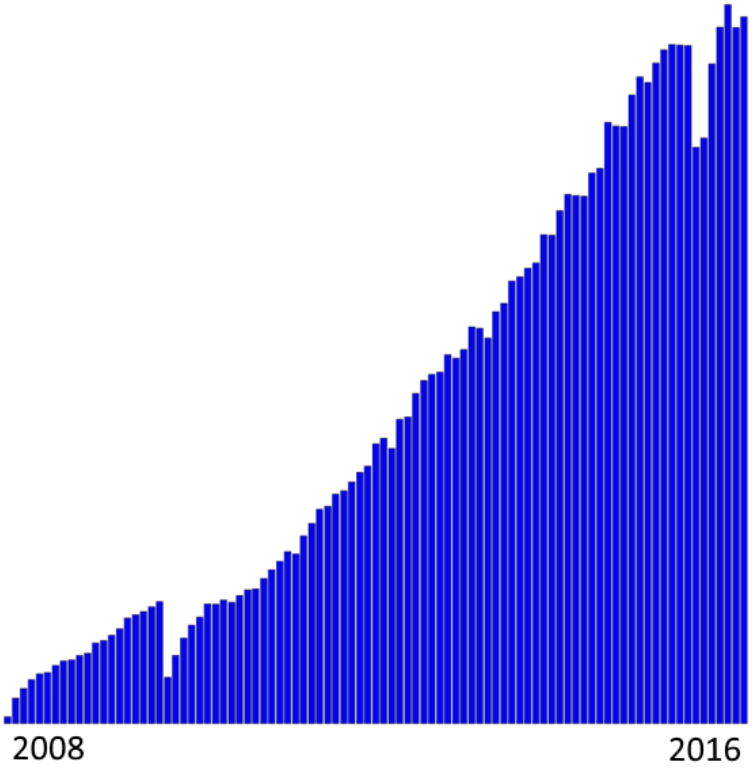
Jenkins syntyi Hudson-ohjelman kehityksen yhteydessä. Hudson on myös Jenkinsin tapaan jatkuvan integroinnin työkalu ja sen on kehittänyt Sun Microsystems. Hudsonin kehitys alkoi kesällä 2004, ja sen ensimmäinen versio julkaistiin helmikuussa vuonna 2005.

Hudsonin kehityksessä tuli kuitenkin vastaan ongelmia liittyen siihen, kuinka paljon Oracle pystyi hallitsemaan ohjelman kehitystä. Näistä projektin sisäisistä ongelmista seurasi se, että Jenkins-projekti haarautettiin Hudsonista. Tämä tapahtui tammikuussa vuonna 2011.

Jenkins tukee monia versionhallintaohjelmia, kuten Subversion, Git ja Mercurial, ja se voi ajaa esimerkiksi Apache Ant-, Maven- ja Gradle-koonteja. Jenkinsin ominaisuuksia on myös mahdollista laajentaa suhteellisen helposti lataamalla plugineja.

3.3 Hyviä puolia

Jenkins on hyvin suosittu jatkuvan integroinnin työkalu, joten sen tunteminen voi osoittautua opiskelijoille työelämässä hyvin hyödylliseksi. Jenkinsillä oli elokuussa 2016 yli 130 000 uutta asennusta. Kuvassa 6 nähdään pylväskaavio, joka kuvaa Jenkinssin asennusmääriä. Jenkinsin suosio ei myöskään osoita laskemisen merkkejä, vaan on sen sijaan yhä kasvussa. [7.]



Kuva 6. Jenkinsin asennusmäärät [8.]

Jenkisin ominaisuuksia on myös helppo laajentaa laajan plugin-valikoiman avulla. Jenkissä on saatavilla yli 1200 erilaista pluginia.

Toisin kuin tämän dokumentin luvuissa 5 ja 6 esiteltävät GitLab CI ja Travis CI Jenkins tukevat monia eri versionhallintatyökaluja kuten esimerkiksi gittiä, subversionia ja CVS:ää (= Concurrent Versioning System). GitLab CI ja Travis CI sen sijaan ovat sidottuja GitLabiin ja GitHubiin. Jenkinsin käyttöliittymä on myös melko helppokäyttöinen.

3.4 Huonoja puolia

Jenkinsin ylläpito aiheuttaa huomattavasti töitä opiskelija-assistentille ja mahdollisesti Helpdeskille. Tämä johtuu siitä, että ohjelmien koonti tapahtuu Metropolian omilla Jenkins-palvelimilla. Jos palvelimelta puuttuu ohjelmia tai työkaluja, joita Jenkisillä koottavat ohjelmat vaatisivat, on opiskelija-assistentin käytävä asentamassa ne kyseisille palvelimille.

Esimerkiksi php-sovelluksen koonnin suoritus vaatii php:n ja phpunit:n asennuksen palvelimille. Pilvipalveluihin pohjautuvissa jatkuvan integroinnin työkaluissa tällaista kehitysympäristön pystytystä ei yleensä tarvitse tehdä, koska pilvipalvelutyökalut tarjoavat yleensä valmiita virtuaalisia koontiympäristöjä ohjelmille.

Joillain kursseilla on ollut tapana, että kaikki Jenkins-projektit ajetaan samassa ympäristössä, joten projektien ajojen on mahdollista häiritä toinen toisiaan. Lisäksi erilaisten tulkkiohjelmien versioiden käyttö ei ole mahdollista. Jokin ajettava sovellus voi vaatia esimerkiksi php:n version 5 toimiakseen, kun taas jokin toinen voi vaatia version 7. Tällaisessa tilanteessa tarvittaisiin kaksi erillistä Jenkins-palvelinta. Pilvipalveluihin pohjautuvissa jatkuvan integroinnin työkaluissa kukin projekti saa yleensä oman eristetyn ajoympäristönsä. Tämä ongelma on toki mahdollista kiertää yksinkertaisesti järjestämällä kaikille kurssin ryhmille oma palvelin. Näin tehdään jo Innovaatioprojekti-kurssilla.

Koska Jenkins-palvelimet sijaitsevat koulun verkossa, voi yhteyden ottaminen esim. tietokantapalvelimiin Jenkinsistä käsin aiheuttaa palomuurien takia ongelmia. Tämä myös tarkoittaa sitä, että joihinkin näistä palvelimista (Innovaatioprojektin palvelimet) ei voi ottaa suoraan yhteyttä koulun verkon ulkopuolelta ilman SSH-tunnelia. Tämä aiheuttaa opiskelijoille vaivaa, jos he haluavat työskennellä kotoa käsin.

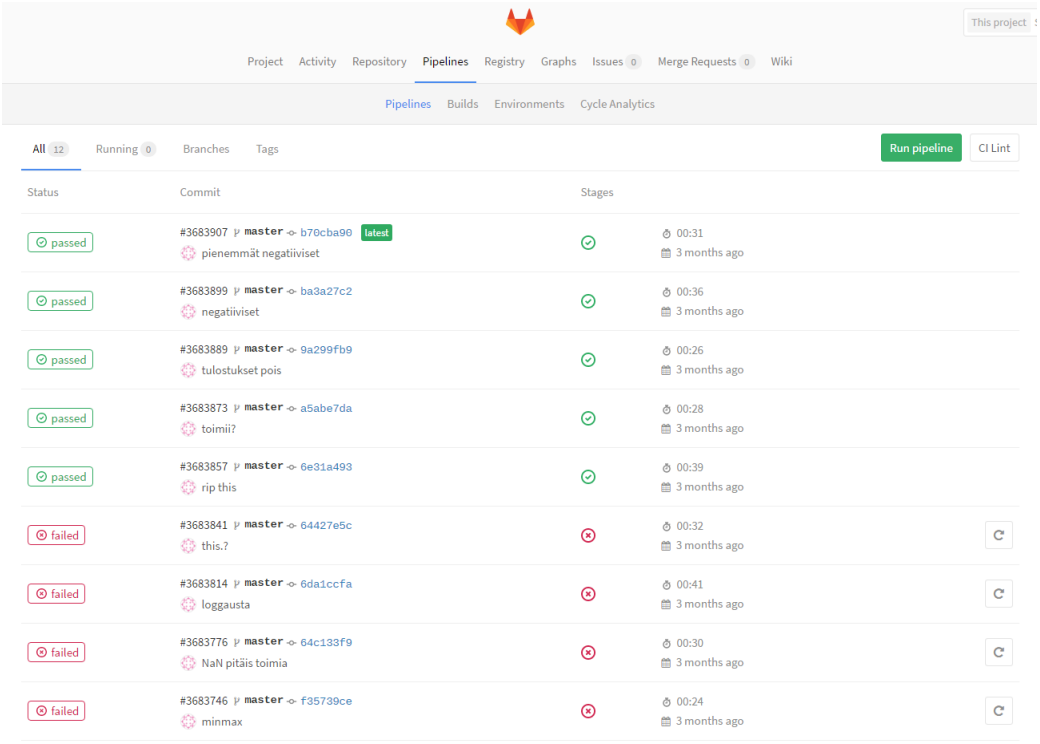
4 GitLab CI

Tässä luvussa esitellään GitLab CI:tä, joka on ensimmäinen kahdesta tässä dokumentissa esiteltävästä pilvipalvelupohjaisesta jatkuvan integroinnin työkalusta. Luvussa kerrotaan ensin yleisiä tietoja GitLab CI:stä (luku 4.1). Tämän jälkeen käydään läpi sitä, miten jatkuva integrointi GitLab CI:llä tapahtuu käytännössä (luku 4.2). Lopuksi käydään läpi sen hyvät ja huonot puolet (luvut 4.3 ja 4.4).

4.1 Yleistä

GitLab CI on GitLab-versionhallintatyökaluun integroitu jatkuvan kehityksen työkalu. GitLab on ilmainen git-pohjainen versionhallintatyökalu, josta on maksullinen versio yrityskäyttöön. GitLab julkaistiin vuonna 2011 ja jako ilmaisversioon (Community Edition) ja

maksullisen versioon (Enterprise Edition) tehtiin vuonna 2013. Kuvassa 7 nähdään Pipelines-näkymä GitLabissa.



Kuva 7. GitLab CI

4.2 Käyttö

GitLab CI:n käyttöönotto tapahtuu lisäämällä .gitlab-ci.yml-niminen tiedosto GitLab-projektin juureen. Tämä tiedosto on YAML (YAML Ain't Markup Language) -muotoinen tiedosto, joka kertoo GitLab CI:lle, kuinka projektin koonti on tehtävä. [9.]

```
image: frekele/ant
distribute:
  stage: build
  script:
    - ant dist

test:
  stage: test
  script:
    - ant test
```

Koodi 1. Esimerkki gitlab-ci.yml-tiedosto

Yml-tiedosto kertoo GitLab CI:lle, mistä Docker-imagesta on tehtävä Docker-kontti sovelluksen koontia varten. GitLab CI hakee nämä imaget Docker Hubista. Dockeria käydään enemmän läpi luvussa 4.2.2. Esimerkkikoodissa 1 tämä löytyy ensimmäiseltä riviltä. Tässä esimerkissä koonti tehdään “frekele/ant”-imagessa. [10.]

Lisäksi yml-tiedosto pitää sisällään GitLab CI:n jobeja. Esimerkissä näitä jobeja ovat distribute ja test. Kunkin jobin alta on löydyttävä vähintään script-elementti. Kukin job ajetaan omassa imagessaan, joten niiden ajot eivät vaikuta toisien jobien ajoihin. Tämä elementti on lista, jossa on ranskalaisin viivoin annettu komentorivikäskyt, joilla koonti tehdään. Esimerkissä kukin job sisältää vain yhden käskyn, distribute sisältää “ant dist” -komentorivikäskyn ja test sisältää “ant test” -komentorivikäskyn.

Komentirivikäskyjen lisäksi job voi sisältää stage-elementin. Stage-elementti kertoo GitLab CI:lle, kuuluuko job build, test vai deploy vaiheeseen. Jos stage-elementti puuttuu jobista, GitLab CI käyttää oletusarvona testiä. Stage-määrittely kertoo GitLab CI:lle, missä järjestyksessä koonnit on tehtävä. Jos edellisen stagen koonnit eivät mene läpi, seuraavien stagejen koonteja ei ajeta. Lisäksi saman stagen jobit ajetaan yhtä aikaa. Koontijärjestys on build, test, deploy.

Lopullinen koonti prosessi on siis:

1. Kaikki build-jobit ajetaan yhtä aikaa.
2. Jos build-jobit onnistuvat, ajetaan kaikki test-jobit yhtä aikaa.
3. Jos kaikki test-jobit onnistuvat, ajetaan kaikki deploy-jobit yhtä aikaa.
4. Jos kaikki deploy-jobit onnistuvat, commitin koonti merkitään onnistuneeksi.
5. Jos mikään jobeista epäonnistui, commitin koonti merkitään epäonnistuneeksi.

Jobien ja imagen määrittelyn lisäksi yml-tiedostossa voidaan tehdä monia muita vaihtoehtoisia määrittelyjä. Näitä ovat esimerkiksi:

- services
  - Määrittelee mitä Docker palveluja käytetään.
- before_script
  - Sisältää listan komentorivikäskyjä, jotka ajetaan ennen kunkin script-elementin käskyjä.
  - Tätä listaa voidaa käyttää esimerkiksi imagen alustukseen.
- after_script
  - Sisältää listan komentorivikäskyjä, jotka ajetaan kunkin script-elementin käskyjen jälkeen.
- variables
  - Tämän elementin avulla on mahdollista asettaa ympäristömuuttujia Docker-imagea varten.
  - Voidaan käyttää job kohtaisten ympäristömuuttujien määrittelyyn tai sellaisten ympösristömuuttujien määrittelyyn joita kaikki jobit voivat käyttää.
- cache
  - Tämän elementin avulla on mahdollista säilyttää kansioita ja tiedostoja koontien välillä.
  - Määrittelyn voi tehdä yksittäiselle jobille tai kaikkille jobeille riipuen siitä, missä kohtaa määrittely tehdään.

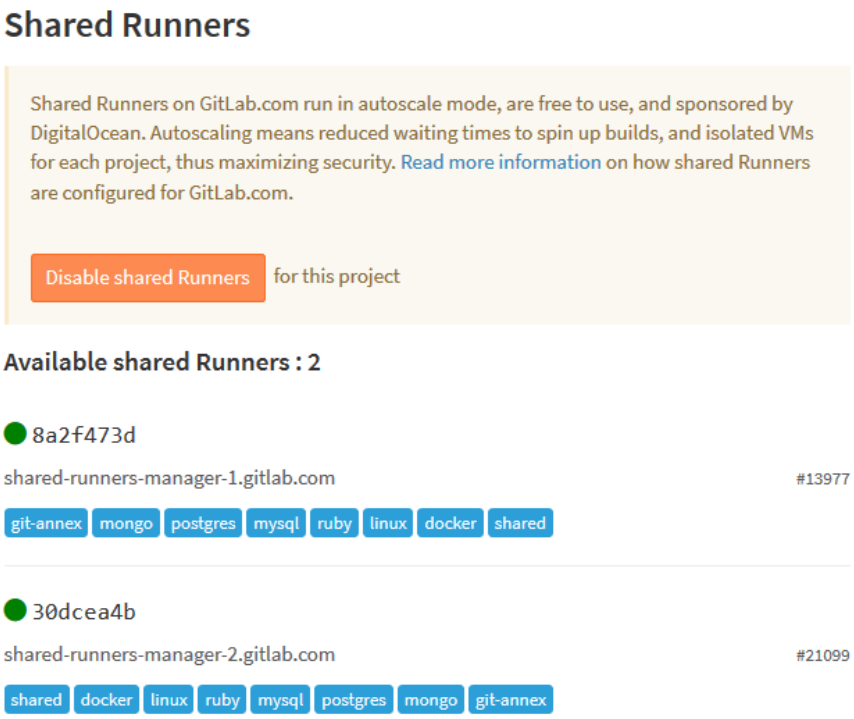
Kun yml-tiedosto on oikealla paikalla projektin juuressa ja sen sisältö on kunnossa, GitLab-suorittaa projektin koonnin jokaisen tietovarastoon tehtävän commitin ja mergen yhteydessä. Koonnin suorittaa Runner-virtuaalikone.

4.2.1 Runner

Runner on kone, joka suorittaa sovelluksen koonnin. Se voi olla virtuaalikone, VPS (virtual private server), tavallinen kone, Docker-säiliö tai säilörypäs. Runnereita on kahden tyyppisiä, jaettuja (englanniksi shared) ja erityisiä (englanniksi specific). [11.]

Jaetut runnerit ovat ilmaisia runneroita, joiden käyttöä varten käyttäjien ei tarvitse asentaa mitään omille koneilleen. Ne on sponsoroinut DigitalOcean. Jaetut runnerit ovat automaattisesti skaalautuvia. Automaattinen skaalaus vähentää odotusaikoja ennen kuin koonti voidaan aloittaa ja sallii jokaisen projektin käyttää eristettyä virtuaalikonetta.

Jaetut runnerit käyttävät Docker-kontteja, jotka GitLab CI kopioi Docker Hub palvelusta saatavista imageista. Kuvassa 8 nähdään, kuinka sovelluksen koontiin on liitetty kaksi runneria.



Kuva 8. Koontiin liitettyjä jaettuja runnereita

4.2.2 Docker-kontit

Docker on avoimen lähdekoodin ohjelma, jonka avulla on mahdollista luoda Linux-sovellussäiliöitä eli Docker-kontteja. Dockerin avulla ohjelmia voi ajaa turvallisesti näissä eristetyissä säiliöissä. [12.]

Docker Hub on pilvipalvelu, jonka avulla käyttäjät voivat jakaa Docker imagejaan. Virtuaalikoneista puhuttaessa imageilla tarkoitetaan kopiota käyttöjärjestelmän tilasta jollakin tietyllä hetkellä. Docker Hubin avulla on myös mahdollista automatisoida uusien imagien tekemistä.

```
1  gitlab-ci-multi-runner 1.3.2 (0323456)
2  Using Docker executor with image monostream/apache-ant ...
3  Pulling docker image monostream/apache-ant ...
4  Running on runner-30dcea4b-project-1380897-concurrent-0 via runner-30dcea4b-machine-1468215266-598a2f3e-digital-ocean-4gb...
5  Cloning repository...
6  Cloning into '/builds/Ville1/Java-CI-Example-2'...
7  Checking out 12f4d628 as master...
8  $ ant dist
9  Buildfile: /builds/Ville1/Java-CI-Example-2/build.xml
10
11 compile:
12     [mkdir] Created dir: /builds/Ville1/Java-CI-Example-2/build/main
13     [javac] Compiling 2 source files to /builds/Ville1/Java-CI-Example-2/build/main
14
15 dist:
16     [mkdir] Created dir: /builds/Ville1/Java-CI-Example-2/dist
17       [jar] Building jar: /builds/Ville1/Java-CI-Example-2/dist/SimpleProject.jar
18
19 BUILD SUCCESSFUL
20 Total time: 1 second
21
22 Build succeeded
```

Kuva 9. Docker komentorivillä

Kuvassa 9 nähdään komentorivinäkymässä, kuinka jaettu runner (gitlab-ci-multi-runner) hakee Docker Hubista imagen monostream/apache-ant (rivit 2 - 4), kloonaa tietovaraston koodit sinne (rivit 5 - 7) ja ajaa varsinaisen koonnin (rivit 8+).

4.3 Hyviä puolia

GitLab tarjoaa REST-rajapintoja (Representational state transfer), joiden avulla tietovaraston ja koontien tietoihin on mahdollista päästä suhteellisen helposti käsiksi muista ohjelmista. Docker-konttien takia kehitysympäristöjen teko ja hallinta on helpompaa kuin Jenkinsissä.

4.4 Huonoja puolia

GitLab CI on tarkoitettu vain GitLabissa sijaitsevia projekteja varten. Jos ohjelman koodi on jo jossain toisessa versionhallinta työkalussa, on se siirrettävä GitLabiin.

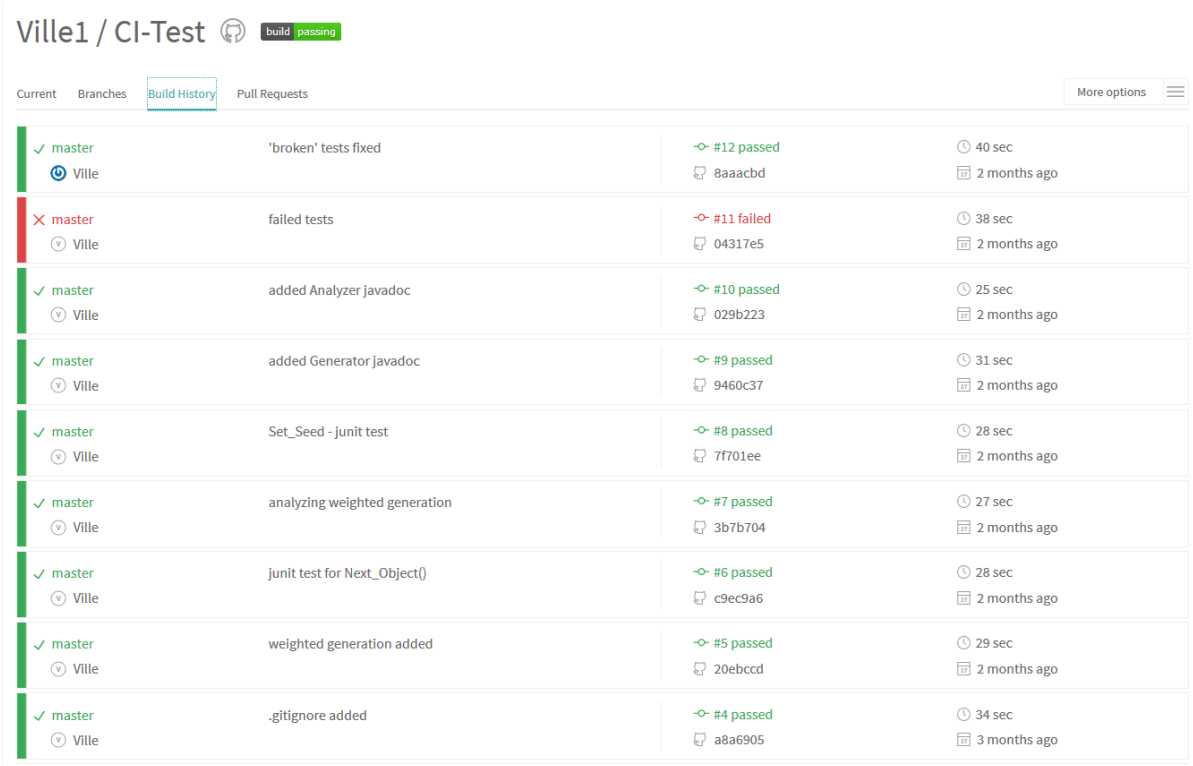
Testituloksista ja staattisen analyysin raporteista ei ole mahdollista saada graafisia esityksiä GitLab CI:n sisällä. GitLab CI:n Runner- ja Docker – image järjestelmän käyttö voi myös olla vaikeaa uudelle käyttäjälle.

5 Travis CI

Tässä luvussa esitellään Travis CI, joka on GitLab CI:n ohella toinen dokumentissa esiteltävistä jatkuvan integroinnin työkaluista. Luku alkaa Travis CI:n yleisillä tiedoilla (luku 5.1). Tämä kappale sisältää Travis CI:n teknisiä tietoja, liiketoimintamallin yms. Tämän jälkeen kerrotaan lyhyesti GitHubista, joka on Travis CI:n käyttämä versionhallintasovellus (luku 5.2). GitLab CI:n luvun tavoin myös tämä luku pitää sisällään kappaleen, jossa esitellään tarkemmin sitä, miten jatkuva integrointi työkalun avulla tapahtuu käytännössä (luku 5.3). Tämä luku sisältää myös Travis CI:n hyvät ja huonot puolet (luvut 5.4 ja 5.5).

5.1 Yleistä

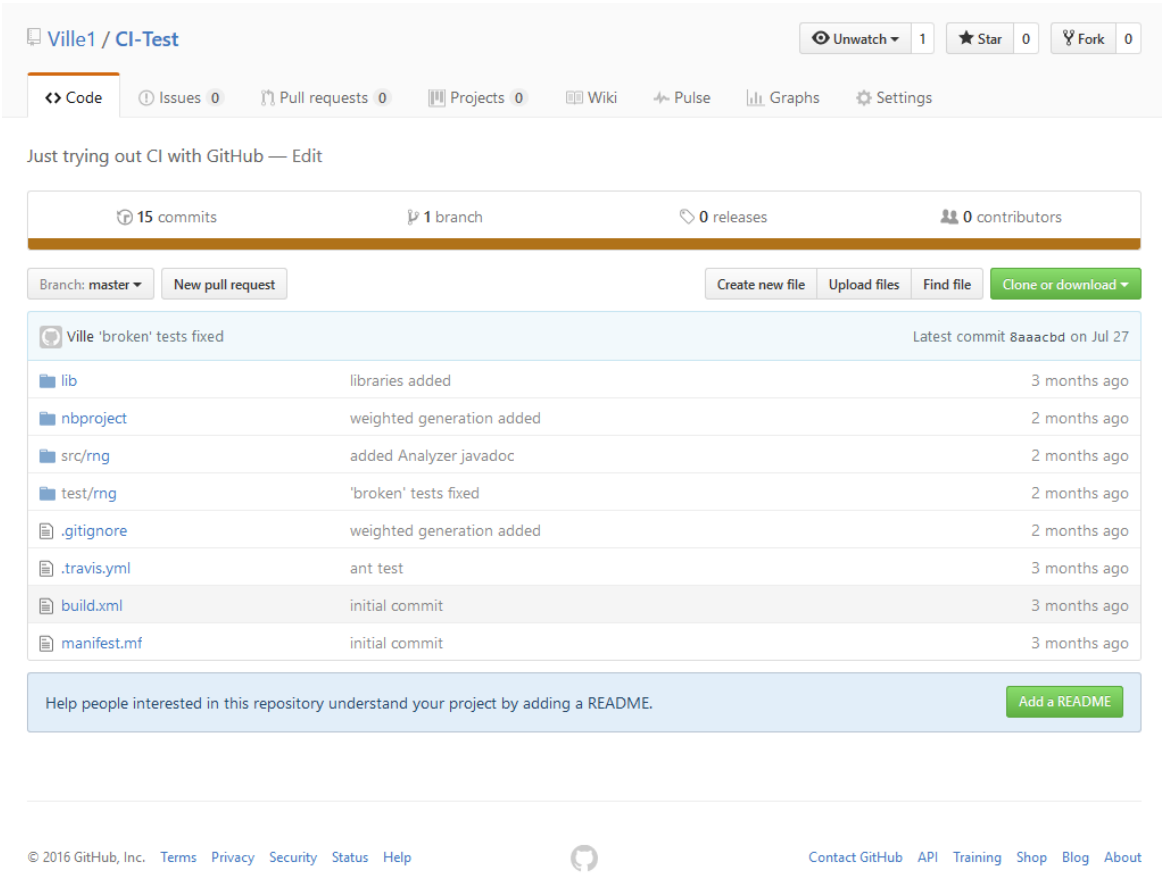
Travis CI on Ruby-ohjelmointikielellä kirjoitettu avoimen lähdekoodin jatkuvan integroinnin työkalu. GitLab CI:in tavoin Travis CI:n jakelu tapahtuu MIT-lisenssin alla. Travis CI toimii GitHub-versionhallintatyökalun kanssa. Avoimen lähdekoodin projektien koonti Travisin verkkosivulla (travis-ci.org) on ilmaista, mutta yksityisten projektien koontien järjestäminen on maksullista. Kirjautuminen Travisin verkkosivuille tapahtuu GitHub-tilin avulla. Koska Travis on avoimen lähdekoodin projekti, käyttäjien on mahdollista integroida se omille alustoilleen, mutta yritys ei suosittele sen yrittämistä tavallisille käyttäjille. Kuvassa 10 nähdään koontihistoria Travis CI:ssä. [13; 14.]



Kuva 10. Travis CI

5.2 GitHub

Travis CI hakee koottavien ohjelmien lähdekoodit GitHub-versionhallintatyökalusta. GitHub on git-pohjainen versionhallintatyökalu. Se on Travisin tavoin kirjoitettu Ruby-kielellä. [15.] Kuvassa 11 nähdään projektin etusivu GitHubissa.



Kuva 11. GitHub-versionhallintatyökalu

5.3 Käyttö

GitLab CI:n tapaan Travis CI:n käyttöönotto GitHub-tietovarastoon viedyssä projektissa tapahtuu lisäämällä projektin hakemistorakenteen juureen yaml-konfiguraatio tiedosto. Travis CI:n tapauksessa tämän tiedoston nimi on oltava .travis.yml, jotta Travis tunnistaa sen konfiguraatiotiedostoksi.

```
language: java

script:
  - jdk_switcher use oraclejdk8
  - ant test
```

Koodiesimerkki 2. .travis.yml-tiedoston sisältö

GitLab CI:n konfiguraatiossa jatkuvan integroinnin työkalulle kerrottiin, missä virtuaalikone-imagessa koonti halutaan ajaa. Travis CI:n tapauksessa jatkuvan intergroinnin työkalulle kerrotaan vain se, mitä ohjelmointikieltä koottavassa ohjelmassa on käytetty. Esimerkkikoodissa 2 tämä lukee ensimmäisellä rivillä, jossa ohjelmointikieleksi on valittu Java.

Ohjelmointikielimäärittelyn avulla Travis CI osaa valita sopivan virtuaalikone imagen koonnin ajoympäristöksi. Nämä imaget ovat Travisin tarjoamia, ja ne sisältävät useita versioita halutusta ohjelmointikielestä. GitLab CI:n tavoin imageina käytetään Docker imageja. Ne myös sisältävät useita koontityökaluvaihtoehtoja.

Tämän lisäksi esimerkkikoodissa on määritelty script-elementti. Tämä elementti sisältää komentorivikäskyt, joilla koonti tehdään. Esimerkkikoodissa valitaan ensin Javan JDK-versio 8 ja ajetaan sen jälkeen Antin test target. JDK (Java Development Kit) on ohjelmistokehityspaketti Java-ohjelmien koodaamista varten.

```
1  Using worker: worker-linux-docker-68cde390.prod.travis-ci.org:travis-linux-3
2
3  Build system information
4
5  $ export DEBIAN_FRONTEND=noninteractive
6  $ git clone --depth=50 --branch=master https://github.com/Ville1/CI-Test-2.git Ville1/CI-Test-2
7
8  This job is running on container-based infrastructure, which does not allow use of 'sudo', setuid and setguid executables.
9  If you require sudo, add 'sudo: required' to your .travis.yml
10 See https://docs.travis-ci.com/user/workers/container-based-infrastructure/ for details.
11 $ phpenv global 5.4 2>/dev/null
12
13 $ phpenv global 5.4
14
15 $ composer self-update
16
17 Updating to version 8cf9733001d2bc6f063c09de3cd9371f29268911.
18     Downloading: 100%
19 Use composer self-update --rollback to return to version 1d8f05f1dd0e390f253f79ea86cd505178360019
20
21 $ php --version
22 PHP 5.4.37 (cli) (built: Feb 12 2015 01:06:06)
23 Copyright (c) 1997-2014 The PHP Group
24 Zend Engine v2.4.0, Copyright (c) 1998-2014 Zend Technologies
25     with Xdebug v2.2.7, Copyright (c) 2002-2015, by Derick Rethans
26 $ composer --version
27 Composer version 1.3-dev (8cf9733001d2bc6f063c09de3cd9371f29268911) 2016-10-24 11:10:31
28
29 $ cd test
30
31
32 The command "cd test" exited with 0.
33 $ phpunit
34 PHPUnit 4.5.0 by Sebastian Bergmann and contributors.
35
36 Configuration read from /home/travis/build/Ville1/CI-Test-2/test/phpunit.xml
37
38 .....
39
40 Time: 68 ms, Memory: 4.25Mb
41
42 OK (5 tests, 20 assertions)
43
44
45 The command "phpunit" exited with 0.
46
47 Done. Your build exited with 0.
```

Kuva 12. Travis CI:n loki

Kuvassa 12 nähdään Travis CI:n luoma loki esimerkkisovelluksen koonnista. Kyseessä on yksinkertainen php-sovellus. Tätä kuvaa ei ole suoraan otettu Travis CI:stä, vaan se on otettu Notepad++-tekstinkäsittelyohjelmaan kopioidusta lokista. Tämä on tehty luettavuuden parantamiseksi, koska Travis CI tulostaa lokin mustalle taustalle. Osa riveistä on piilotettu: ajoympäristön tietojen tulostus (rivi 3), Debian Frontend -asennus (rivi 5) ja koodin haku versionhallinasta (rivi 6).

Esimerkkikoonnin komentorivinäkymän sisältö:

- Rivi 1: Travis CI hakee imagen ajoa varten.
- Rivi 3: Ajoympäristön tietojen tulostus (piilotettu).
- Rivi 6: Koodin haku version hallinasta (piilotettu).
- Rivit 13 - 15: Php-version valinta.
- Rivit 15 – 19: Composer-työkalun päivitys.
- Rivit 21 - 27: Php:n ja Composerin versiotietojen tulostus.
- Rivit 29+: Sovelluksen koonnin ajo.

5.4 Hyviä puolia

Travis CI on valmiiden virtuaalikone-imagejen takia helppokäyttöinen. Toisin kuin GitLab CI:tä käyttäessä Travis CI:n kanssa ei tarvitse etsiä tai tehdä omaa Docker imagea sovelluksen koontia varten. GitLab CI:n tavoin Travis CI tarjoaa ohjelmoijien käyttöön REST-rajapintoja, joilla on mahdollista esimerkiksi hakea tietoja Travisissa tapahtuneista koonnesta.

5.5 Huonoja puolia

Testituloksista ja staattisen analyysin raporteista ei ole mahdollista saada graafisia esityksiä Travis CI:n sisällä.

6 Kojelaudat

Jatkuvan integroinnin tilanne voi muuttua tiheään tahtiin sitä mukaan, kun versionhallintaan tehdään muutoksia. Jollain hetkellä koonti ei ehkä mene läpi ollenkaan ja ehkä jollain toisella hetkellä se menee läpi, mutta testit eivät mene läpi. Tilanteen muuttumistahti riippuu toki luonnollisesti kehittäjien määrästä ja työajasta.

Tämän muutostahdin takia sovelluskehitysprojektin yleistilanteesta voi olla vaikea saada selkeää kuvaa, varsinkin jos halutaan tarkkailla useampia projekteja yhtä aikaa. Tämä ongelma on ehkä mahdollista ratkaista kojelautasovelluksilla, joita esitellään tässä luvussa.

Ensin luvussa kerrotaan yleisesti siitä, mitä kojelaudalla tarkoitetaan (luku 6.1), ja sen jälkeen siirrytään käymään läpi kojelautojen käyttöä ohjelmistoprojekteissa ja opetuksen tukena (luvut 6.2 ja 6.3). Seuraavaksi kerrotaan siitä, minkälaisia kojelautoja on saatavilla dokumentissa esitellyille jatkuvan integroinnin työkaluille (Jenkins, GitLab CI, Travis CI) (luvut 6.4 ja 6.5). Näissä luvuissa esiteltävät kojelaudat ovat suurimmaksi osaksi ilmaisia, joten luvun lopusta löytyy vielä kappale kaupallisista kojelaudoista (luku 6.6).

6.1 Yleistä

Kojelaudalla (eng. dashboard) tarkoitetaan projektin hallinnan yhteydessä graafisia käyttöliittymiä, joihin on järjestetty tietoa helposti luettavassa muodossa. Ne ovat yleensä verkkosivuja. Kojelautoja käytetään esim. myynnissä ja markkinoinnissa. [16.]

Kuva 13. Kojelautoja

Ensimmäisen kojelaudan kehitti Hewlett Packard. Sen nimi oli yksinkertaisisesti Dashboard, ja se oli työkalu Windowsin kustomointia varten.

Termi kojelauta juontaa juurensa auton kojelaudasta. Auton kojelaudan tapaan projektin kojelauta tarjoaa yhdellä vilkaisulla tiedot projektin nykytilanteesta, kun taas auton kojelauta tarjoaa tiedot auton nykytilanteesta. Kuvassa 13 nähdään joitain esimerkkikojelautoja.

6.2 Kojelaudat ohjelmistoprojekteissa

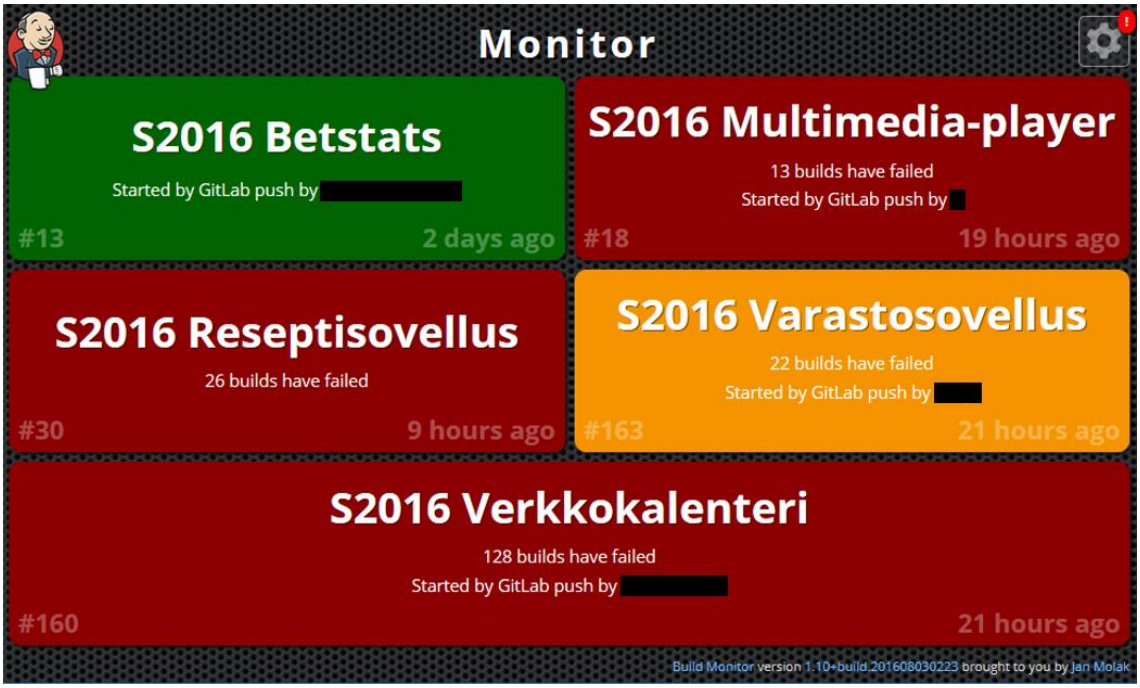
Myös ohjelmistoprojekteissa voidaan hyödyntää kojelautoja. Kojelaudassa näytettävät tiedot voidaan monesti saadaan suoraan jatkuvan integroinnin työkalusta. Ohjelmistoprojektien tapauksessa niistä löytyy yleensä tietoja esimerkiksi viimeisimpien koontien tiloista ja testituloksista.

6.3 Kojelaudat opetuksen tukena

Opiskelijoiden ohjelmistoprojektien jatkuvan integroinnin tiloista on toki myös mahdollista luoda kojelautoja. Tällaisien kojelautojen avulla opettajien ja opiskelija-assistentin olisi mahdollista saada yhdellä silmäyksellä kuva siitä, kuinka kurssin projektit edistyvät ja siitä, mitkä opiskelijaryhmät mahdollisesti tarvitsevat apua.

6.4 Kojelaudat Jenkinsissä

Jenkinssiin löytyy plugineja, joilla jobeista on mahdollista luoda kojelautoja. Esimerkkejä tällaisista plugineista ovat Build Monitor View (kuva 14) ja Dashboard View.

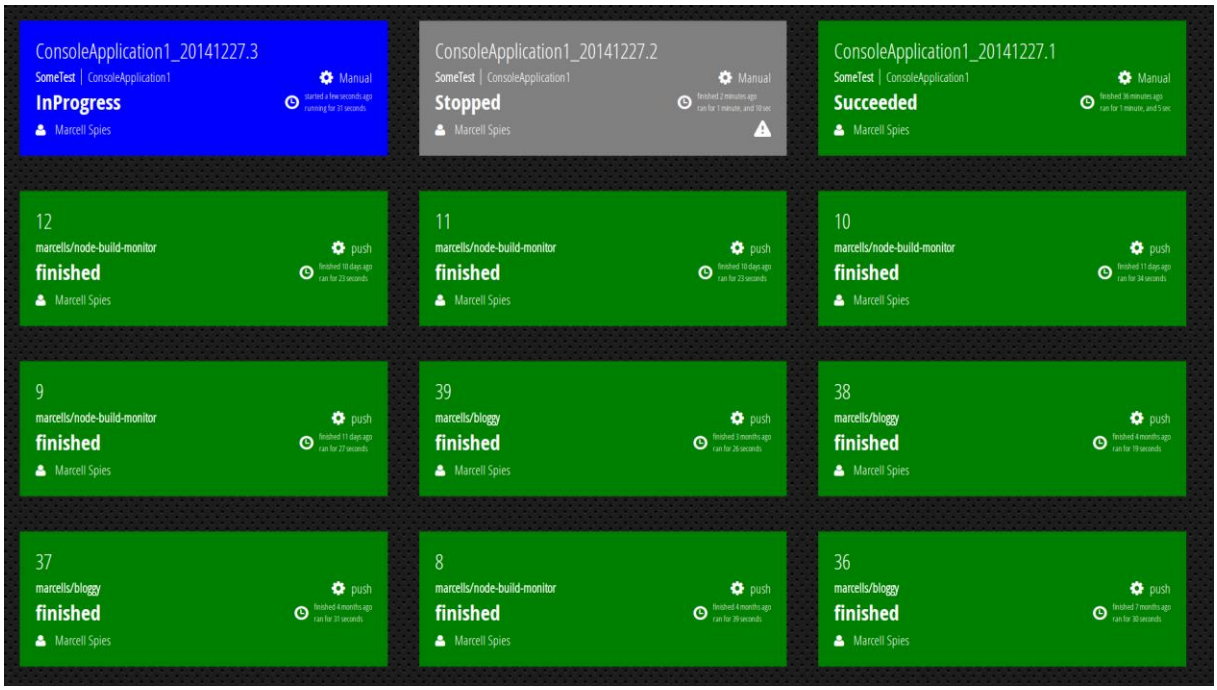


Kuva 14. Jenkinsin Build Monitor View -pluginin muodostama kojelauta

6.5 Kojelaudat GitLab CI:ssä ja Travis CI:ssä

Koska GitLab CI ja Travis CI ovat pilvipalvelupohjaisia jatkuvan integroinnin työkaluja, toisin kuin Jenkins, jota varten on käyttäjällä oltava oma palvelin tai kone, ei niihin ole mahdollista asentaa plugineja.

Jos näiden pilvipalvelupohjaisten jatkuvan integroinnin projektien koonneista haluaa kojelautoja, on ne tehtävä erillisellä sovelluksella. Tällaisia kojelautasovelluksia löytyy esimerkiksi GitHub:ista avoimen lähdekoodin lisenssin alla jaettuina. Yksi esimerkki näistä vapaaohjelmistoista on node-build-monitor (https://github.com/marcells/node-build-monitor, kuva 15). Tämä kojelauta on koodattu JavaScriptin Node.js-kirjaston avulla ja sen tukemiin jatkuvan integroinnin työkaluihin kuuluvat muun muassa kaikki kolme tässä dokumentissa käsiteltyä työkalua (Jenkins, GitLab CI ja Travis CI).

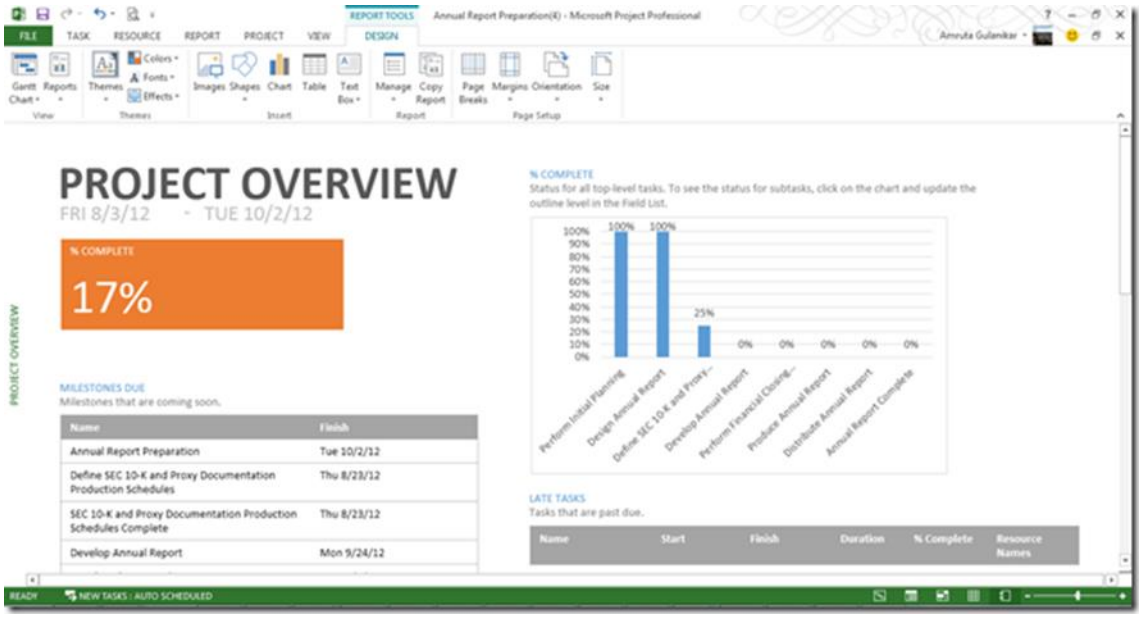


Kuva 15. node-build-monitor-sovelluksen luoma kojelauta

Koska tällaiset kojelauta sovellukset on jaettu avoimen lähdekoodin lisenssin alla, niiden lähdekoodia on mahdollista muokata omia tarpeita vastaaviksi.

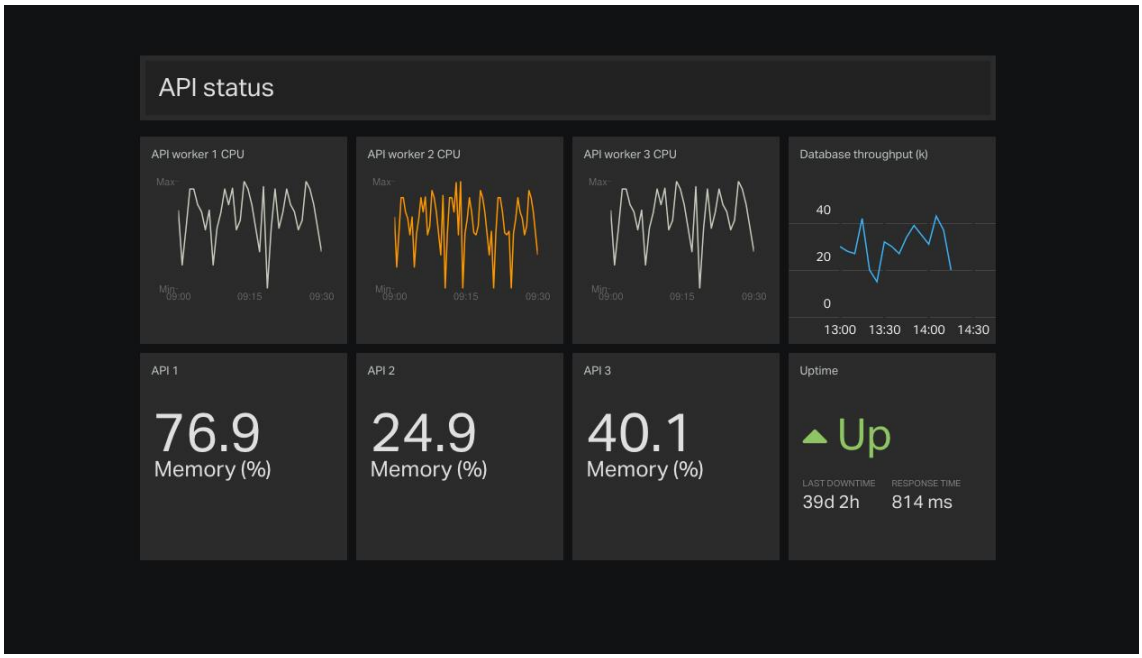
6.6 Kaupalliset kojelaudat

Edellisissä luvuissa puhuttiin ilmaisista avoimen lähdekoodin kojelaudoista, mutta myös kaupallisia kojelauta sovelluksia löytyy. Tällaisia kaupallisia kojelautoja löytyy esimerkiksi projektin hallintatarkoituksiin (esim. Microsoft Project, kuva 16). [17.]



Kuva 16. Microsoft Project

Jatkuvan integroinnin tarkkailuun soveltuvia kaupallisia kojelautoja on vaikeampi löytää kuin esim. projektin hallintaan soveltuvia kojelautoja. Yksi esimerkki kuitenkin kaupallisesta jatkuvaan integrointiin soveltuvasta kojelaudasta on Geckoboardin GitHub Dashboard (kuva 17). [18.]



Kuva 17. Geckoboardin GitHub Dashboard

7 Yhteenveto

Tähän dokumentin viimeiseen lukuun on kerätty yhteenveto. Luvussa kerrataan ensin jatkuvan integroinnin työkalujen hyvät ja huonot puolet (luku 7.1). Tämän jälkeen niitä vertaillaan toisiinsa neljän eri kategorian perusteella (luku 7.2). Nämä kategoriat ovat helppokäyttöisyys, koontiympäristön pystytys, ylläpito ja laajennukset. Luvun lopuksi esitetään jatkoehdotuksia Metropolian jatkuvan integroinnin opetusta varten (luku 7.3).

7.1 Jatkuvan integroinnin työkalut

7.1.1 Jenkins

Jenkins on jatkuvan integroinnin työkalu, joka on tällä hetkellä käytössä Metropolian opetuksessa. Sen hyviin puoliin kuuluvat laaja käyttäjäkunta, helppokäyttöiset pluginit, laaja tuki erilaisille versionhallinta työkaluille ja helppokäyttöinen käyttöliittymä.

Sen huonot puolet syntyvät siitä, että Jenkinsiä varten on ylläpidettävä omaa palvelinta, jossa ohjelmien koonnit ja ajot suoritetaan. Tällaisen palvelimen ylläpidosta syntyviin töihin kuuluu esimerkiksi käyttäjien hallinta ja tulkkien ja laajennusten asennus.

7.1.2 GitLab CI

GitLab CI tarjoaa ohjelmoijien käyttöön laajaan REST-rajapinnan, jota vasten on mahdollista koodata ulkopuolisia ohjelmia (esim. kojelautoja) GitLab CI käyttää ohjelmien koontiympäristöinä Docker imageja. Niiden ansiosta käyttäjän on mahdollista tehdä omia koontiympäristöjä tai hakea valmiita, muiden käyttäjien tekemiä, koontiympäristöjä Docker Hubista.

GitLab CI:n huonoihin puoliin kuuluu se, että sen avulla ei ole mahdollista luoda graafisia esityksiä testitulosten ja staattisen analyysin tulosten trendeistä ilman ulkopuolisia sovelluksia. Docker-imagejen ja Runnereiden käyttö voi myös olla hämmentävää uudelle käyttäjälle.

7.1.3 Travis CI

Paljolti GitLab CI:n tapaan myös Travis CI tarjoaa ohjelmoijien käyttöön REST-rajapinnan. Tämän lisäksi Travis CI:n tarjoaa valmiit virtuaalikone-imaget sovellusten koontiympäristöiksi. Näiden imagejen käyttö on huomattavasti suoraviivaisempaa kuin GitLab CI:n Runner Docker-image järjestelmän käyttö.

Kuten GitLab CI:ssä, myöskään Travis CI:ssä ei ole mahdollista saada aikaan graafisia esityksiä testituloksille tai staattiselle analyysille ilman ulkopuolisia sovelluksia.

7.2 Vertailu

Perustuen näihin tuloksiin arvostelin nämä kolme jatkuvan integroinnin työkalua seuraavasti:

Taulukko 1. Jatkuvan integroinnin työkalujen arvostelu

Nimi	Helppokäyttöisyys	Koontiympäristön	Ylläpito	Laajennukset
Jenkins	1	2	2	1
GitLab CI	3	1	1	2
Travis CI	2	1	1	2

Taulukossa 1 jatkuvan integroinnin työkalut on arvosteltu neljän kategorian mukaan. Ne ovat helppokäyttöisyys, koontiympäristön pystytys, ylläpito ja laajennukset. Arvionumeroissa pienemmät luvut ovat parempia.

7.2.1 Helppokäyttöisyys

Tämä kategoria kuvaa sitä, kuinka helppoa koonnin konfigurointi on. Tähän ei lasketa mukaan koontiympäristön pystytystä esim. ohjelmien asentamista.

Tässä kategoriassa annoin ykkössijan Jenkinsille, suoraviivaisen ja hyvät ohjeet omaavan graafisen käyttöliittymän ansiosta. Sekä Travis CI:n ja GitLab CI:n konfiguraation teko tapahtuu konfiguraatiotiedoston avulla, joten siinä suhteessa ne ovat tasoissa. Travis CI kuitenkin hoitaa virtuaalikoneiden hallinnan automaattisesti, kun taas GitLab CI:ssä tämä on konfiguroitava itse.

7.2.2 Koontiympäristön pystytys

Tämä kategoria kuvaa sitä, kuinka helppoa jatkuvaan integrointiin tarvittavan ympäristön pystytys on. Tähän kategoriaan siis kuuluvat sellaiset toimet, jotka on tehtävä ennen kuin koontia itsessään voidaan alkaa konfiguroida.

Tässä kategoriassa laitoin Jenkinsin viimeiselle sijalle johtuen siitä, että sitä varten on järjestettävä oma palvelin, jolla sitä ajetaan. Se, että koontiympäristö on jaettu useiden ohjelmien välillä, voi myös aiheuttaa ongelmia pystytyksessä. Sekä GitLab CI että Travis CI tekevät koonnit Docker-imageissa, joten koontiympäristöä ei välttämättä tarvitse tehdä itse, koska valmiita imageja löytyy.

7.2.3 Ylläpito

Tämä kategoria kuvaa sitä, kuinka helppoa olemassa olevan jatkuvan integroinnin ympäristön ylläpitäminen on. Annoin ensimmäisensijan myös tässä kategoriassa pilvipalvelupohjaisille työkaluille. Perusteluni tähän on se, että Jenkinsissä koontiympäristö on jaettu useamman ohjelman välillä, joten esimerkiksi tulkkien (esim. Java) päivittäminen voi aiheuttaa ongelmia, jos kaikki ohjelmat eivät käytä samaa versiota.

7.2.4 Laajennukset

Tämä kategoria kuvaa sitä, kuinka helppoa laajennusten asentaminen työkaluun on. Ensinmäinen sija tässä kategoriassa meni Jenkinsille. Jenkinsiin löytyy laaja valikoima erilaisia plugineja, ja niiden haku ja asennus sujuu helposti Jenkinsin käyttöliittymän kautta. Pilvipalvelupohjaisille työkaluille löytyy joitain kolmannen osapuolen sovelluksia, kuten web clienttejä ja kojelautoja, mutta varsinaisia laajennuksia niihin ei ole mahdollista asentaa.

7.3 Jatkoehdotukset

Perustuen tämän insinöörityön tuloksiin jatkoehdotukseni on se, että pääpaino jatkuvan integroinnin opetuksessa on jatkossakin Jenkinsissä. Päädyin tähän tulokseen, koska Jenkins on yhä hyvin suosittu jatkuvan integroinnin työkalu ja sen huonot puolet (esim.

koontiympäristön pystytys) eivät loppujen lopuksi ole kovin suuria. Oikeastaan Jenkinsin tarve omalle palvelimelle on syy pitää se mukana opetuksessa, koska jos opiskelijat haluavat kokeilla Jenkinsiä omalla ajallaan voi palvelimen saaminen olla vaikeaa. Vastaavasti pilvipalvelupohjaisia jatkuvan integroinnin työkaluja on suhteellisen helppo lähteä kokeilemaan itse.

Pilvipalvelupohjaiset työkalut voivat toki tarjota kiinnostavan vaihtoehdon Jenkinsille. Voisi myös olla hyvä tarjota opiskelijoille mahdollisuus kokeilla niitä Jenkinsin sijasta. Tällaisessa tilanteessa suosittelen Travis CI:tä GitLab CI:n yli johtuen siitä, että koin Travis CI:n helppokäyttöisemmäksi ja ylipäätään selkeämmäksi kokonaisuudeksi.

Lähteet

1 Wikipedia. Scripting language. Verkkodokumentti. <https://en.wikipedia.org/wiki/Scripting_language> Luettu 23.9.2016.

2 Smith, Peter. 2011. Software Build Systems: Principles and Experience. Addison-Wesley Professional.

3 Wikipedia. Test driven development. Verkkodokumentti. <https://en.wikipedia.org/wiki/Test-driven_development> Luettu 23.9.2016.

4 Checkstyle. Verkkodokumentti. <http://checkstyle.sourceforge.net/> Luettu 28.10.2016.

5 Kasari, Topi. 2012. Jatkuvan integroinnin koontityökalut. Metropolia Ammattikorkeakoulu.

6 Wikipedia. MIT License. Verkkodokumentti. <https://en.wikipedia.org/wiki/MIT_License> Luettu 26.9.2016.

7 Denis Polkhovskiy. Comparison between continuous integration tools. Verkkodokumentti. <https://dspace.cc.tut.fi/dpub/bitstream/handle/123456789/24043/polkhovskiy.pdf?sequence=1&isAllowed=y> Luettu 6.9.2016.

8 Jenkins. Jenkins infra-statistics. Verkkodokumentti. <http://stats.jenkins.io/> Luettu 6.9.2016.

9 GitLab. Quick Start. Verkkodokumentti. <https://docs.gitlab.com/ce/ci/quick_start/README.html> Luettu 26.9.2016.

10 GitLab. Configuration of your builds with .gitlab-ci.yml. Verkkodokumentti. <https://docs.gitlab.com/ce/ci/yaml/README.html> Luettu 26.9.2016.

11 GitLab. Runners. Verkkodokumentti. <https://docs.gitlab.com/ce/ci/runners/README.html> Luettu 26.9.2016.

12 Docker dokumentaatio. Verkkodokumentti. <https://docs.docker.com/> Luettu 24.10.2016.

13 Wikipedia. Travis CI. Verkkodokumentti. <https://en.wikipedia.org/wiki/Travis_CI> Luettu 30.9.2016.

14 GitHub. Travis CI Readme. Verkkodokumentti. <https://github.com/travis-ci/travis-ci/blob/2ea7620f4be51a345632e355260b22511198ea64/README.textile> Luettu 7.10.2016.

15 Wikipedia. GitHub. Verkkodokumentti. <https://en.wikipedia.org/wiki/GitHub> Luettu 30.9.2016.

16 TechTarget. dashboard. Verkkodokumentti. <http://searchcio.techtarget.com/definition/dashboard> Luettu 10.1.2017.

17 Software Advice. Vendor Showdown: Comparing the 5 Best Project Tracking Software Dashboards. Verkkodokumentti. <http://www.softwareadvice.com/resources/best-project-tracking-software-dashboards/#microsoft> Luettu 29.1.2017.

18 Geckoboard. Github dashboard. Verkkodokumentti. <https://www.geckoboard.com/integrations/github/> Luettu 29.1.2017.