Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Aleksi Kärkkäinen

# Ohjelmistojulkaisun automatisointi Jenkinsillä ja Ansiblella

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

19.4.2021

| | |
|---|---|
| Tekijä<br>Otsikko<br><br>Sivumäärä<br>Aika | Aleksi Kärkkäinen<br>Ohjelmistojulkaisun automatisointi Jenkinsillä ja Ansiblella<br><br>30 sivua<br>19.04.2021 |
| Tutkinto | Insinööri (AMK) |
| Tutkinto-ohjelma | Tieto- ja viestintätekniikka |
| Ammatillinen pääaine | Ohjelmistotuotanto |
| Ohjaajat | Yliopettaja Auvo Häkkinen<br>Manager Juha Kurppa<br>Team & Customer Lead Joonas Fager |


Uuden ohjelmistoversion saattaminen lähdekoodista kohdeympäristössä suorittavaksi ohjelmaksi on prosessi, jossa luotettavuus ja toistettavuus ovat avainasemassa. Prosessin pitää olla varma ja päivityksen jälkeen ohjelman tila tulee voida palauttaa virhetilanteessa takaisin toimivaksi. Ohjelmistojulkaisuprosessin vaiheista lähdekoodi, koonti, testaus ja julkaisu ovat laajalti automatisoitavissa. Automatisoinnin toteutukseen on tarjolla monia työkaluja, joista Jenkins ja Ansible ovat muodostuneet yhdeksi alan standardeista.

Jenkins ja Ansible ovat ohjelmistoja, jotka mahdollistavat ohjelmistojulkaisun luomiseen ja vientiin tarvittavat työkalut ja yksinkertaistavat prosessin käyttäjälle. Jenkinsissä voi suorittaa koontiin ja testaukseen tarvittavia työkaluja ja hallita koko ohjelmistojulkaisuprosessia. Ansible suorittaa alatason operaatiot kuten tiedostojen kopioinnin, varmuuskopioinnin ja asetustiedostojen luonnin.

Insinöörityössä esitellään tapa toteuttaa julkaisu- ja vientiprosessi Jenkinsin ja Ansiblen avulla Java-pohjaiselle sovellukselle ja pohditaan testaustapoja jatkuvan integraation saavuttamiseksi. Työn tuloksena luotiin ohjelmistojulkaisuprosessi, joka on lähes täysin automatisoitu.

Työtä voi käyttää perustana julkaisuprosessin automatisointiin mille tahansa alustalle. Annettuja esimerkkejä ja toteutustapoja voi suoraan soveltaa paikallisilla palvelimilla ajettaviin ohjelmistoihin.



| | |
|---|---|
| Avainsanat | ohjelmistojulkaisun automatisointi, Ansible, Jenkins |

| | |
|---|---|
| Author | Aleksi Kärkkäinen |
| Title | Automating software release process with Ansible and Jenkins |
| Number of Pages | 30 pages |
| Date | 19 April 2021 |
| Degree | Bachelor of Engineering |
| Degree Programme | Information and Communication Technology |
| Professional Major | Software Engineering |
| Instructors | Auvo Häkkinen, Head Teacher<br>Juha Kurppa, Manager<br>Joonas Fager, Team and Customer Lead |

The process from source code to a running program in the target environment requires reliability and repeatability. Process requires to be rigid and software state should be possible to roll back to previous working state in case of error. Software release processes, including source code, building, testing and deployment, can be highly automated. Multiply tools exist to implement the automation, of which Jenkins and Ansible have become one of the industry standards.

Jenkins and Ansible are programs which provide the tools needed to build and deploy a software release and simplify the process for the user. With Jenkins, one can run the tools needed for assembly and testing, and manage the entire software publishing process. Ansible performs low-level operations such as copying files, backing up, and creating configuration files.

The engineering thesis presents a way to implement a build and deploying with the help of Jenkins and Ansible for a Java-based application and discusses testing methods to achieve continuous integration. The work resulted in a software publishing process that is almost completely automated.

The work can be used as a basis for automating the publishing process on any platform. The given examples and implementations can be directly applied to software running on on-premise servers.

| | |
|---|---|
| Keywords | deployment automation, Ansible, Jenkins |

## Sisällys

## Lyhenteet

| | |
|---|---|
| B2B | Business to Business. Kaupankäynti yritysten välillä. |
| B2C | Business to Customer. Kaupankäynti henkilöasiakkaille. |
| CI | Continuous Integration. Jatkuva integraatio. Ohjelmistotuotannon menetelmä ohjelmistokoodin laadun varmistamiseksi. |
| GPG | Gnu Privacy Guard. OpenPGP-standardista tehty salausmenetelmä. |
| JSP | Java Servlet Pages. Palvelinpuolen ohjelmointiprotokolla. |
| SSH | Secure Shell. Salatun tietoliikenteen protokolla. |
| SVN | Subversion. Lähdekoodin versiohallintajärjestelmä. |

# 1 Johdanto

Verkkokaupat ovat kasvaneet suurimmaksi myyntikanavaksi yrityksille, jolloin vaatimukset sekä standardit uusien ominaisuuksien ja virheiden korjausten tuotantoon saattamiseksi ovat nousseet. Jatkuva integraatio on ollut yleistä frontend-puolella jo pitkään, mutta samoille toimintatavoille on tarvetta myös raskaassa Javan backend -maailmassa. Suuret verkkokauppa-alustat monine riippuvuuksineen tekevät ohjelmistojulkaisusta tarkan prosessin, jossa toistettavuus ja luotettavuus ovat avainasemassa. Liiketoiminnan kannalta elintärkeän järjestelmän päivitys tulee olla hyvin suunniteltu ja sisältää monia vaiheita, joista onnistunut jatkuva integraatio syntyy: versiohallinnasta integraatiopalvelimeen sekä automaattisesta testauksesta ympäristönhallintaan.

Työn tarkoituksena on esitellä tehty toteutus suomalaisen kansainvälisillä markkinoilla toimivan yrityksen Business to Business (B2B) -verkkokaupan tuotantoon vientiprosessiin. Asiakkaan huolena olivat aikaa vievät ohjelmistojulkaisut, jotka eivät olleet helposti toistettavissa eikä niitä tehty hallitussa ympäristössä. Tämä aiheutti epävarmuutta ja toimittajakohtaisen riippuvuuden asiakkaalle johtuen henkilöityneestä prosessista: Muutama työntekijä toimittajayrityksestä vei uusia versioita kauppoihin eikä tarkkaa prosessia oltu määritelty. Asiakas päätyi tilaamaan ohjelmistojulkaisuratkaisun, joka vähentäisi yksittäisiä versionpäivityskustannuksia pitkällä aikavälillä ja loisi selkeän polun koodimuutoksista verkkokaupan uuden version käyttöönottoon.

Insinöörityö kuvaa ensin ohjelmistojulkaisuprosessin kohteena olevan järjestelmän ja ohjelmistoarkkitehtuurin sekä käytetyt suoritusympäristöt. Luku 3 kuvaa ohjelmistojulkaisuprosessin ylätason päävaiheet ja avaa niiden sisältöä ja merkitystä. Toteutusvaiheessa esitetään ohjelmistojulkaisunprosessin toteutus käyttäen Jenkins-ohjelmistoa. Alatason operaatiot on toteutettu Ansible-ohjelmistolla. Sen käyttöä esitellään luvussa 5. Lopuksi luvussa 6 pohditaan mahdollisia kehitystarpeita, jotka mahdollistaisivat jatkuvan integraation julkaisustrategian.

## 2 Ohjelmistoarkkitehtuuri ja ympäristöt

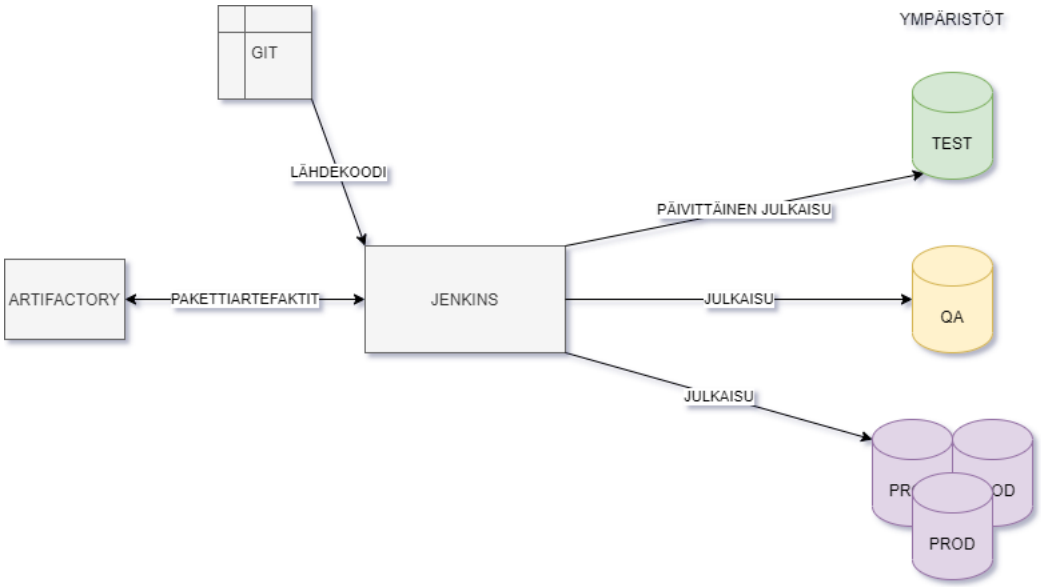
Versiopäivityksien kohteena oleva järjestelmä on Java-ohjelmointikielellä toteutettu verkkokauppa-alusta. Alusta on toteutettu käyttäen Spring-ohjelmistokehystä. Se tarjoaa laajat perusteet yritystason verkkokauppatoimintojen toteutukseen sekä mahdollistaa alustan ominaisuuksien monipuolisen muokkauksen asiakkaan tarpeiden mukaisesti. Asiakkaan toteutus pohjaa vanhempaan versioon alustasta, joka on koottu Java-versiolla 8. Ohjelmiston backend- ja frontend-arkkitehtuurit ovat sidoksissa toisiinsa siten, että ne suoritetaan samassa prosessissa. Frontend-teknologiana käytetään Java Servlet -sivuja (engl. Java Servlet Pages, JSP), jotka renderoidaan palvelimella. Tämän vuoksi frontend-osuuden päivitystä ei voi tehdä erikseen. Uudemmissa ohjelmistoissa backend- ja frontend -arkkitehtuurit on usein eriytetty niin, että backend tarjoaa rajapinnan. Rajapinnan kautta frontend-ohjelmisto kommunikoi backendin kanssa ja hakee, tallentaa, päivittää ja poistaa tietoa.

Verkkokauppa-alusta sisältää kaksi komponenttia: verkkokaupan ydintoiminnot (engl. core) sekä integraatio-ohjelmiston (engl. integration). Ydinohjelmisto sisältää kaikki tarvittavat toiminnallisuudet ostamiseen, sisällönhallintaan, valuuttalaskentaan sekä tuotehallintaan, ja integraatio-ohjelma toimii integraatiorajapintana ulos- ja sisäänpäin. Integraatioissa muutetaan tietoa eri muotoihin riippuen vastaanottavasta järjestelmästä. Kummatkin komponentit sijaitsevat omissa git-säilöissään, niistä tehdään erilliset versiopaketit ja ne sijaitsevat omilla palvelimillaan. Verkkokaupan tarvitsemat palvelut on jaettu kolmeen klusteriin: app-palvelimia on kolme instanssia, integraatio- ja tietokantapalvelimia kaksi instanssia.

Useat palvelimet eivät kuitenkaan vaikeuta julkaisun vientiä ympäristöihin, sillä Ansiblella toimintojen tekeminen usealle palvelimelle samaan aikaan on helppoa eikä vaadi muutamaa koodiriviä enempää verrattuna tilanteeseen, jossa palvelimia olisi vain yksi. Käytössä olevat ympäristöt ovat TEST, QA ja useampi PROD-ympäristö eri myyntialueiden mukaan.

Jenkins on keskeisessä osassa julkaisuprosessia. Jenkins hakee ja tuottaa lähdekoodista paketinhallintajärjestelmä Artifactoryyn pakettiartefakteja, jotka sisältävät koottua ohjelmistokoodia. JFrog Artifactory on universaali DevOps-ratkaisu end-to-end-automaatioihin ja binääripakettien ja -artefaktien hallintaan [1]. Jenkins suorittaa myös

ohjelmistojulkaisujen viennin ympäristöihin ja hallitsee kaikkien järjestelmien kanssa kommunikoinnin (kuva 1).




Kuva 1. Julkaisuprosessin arkkitehtuuri.

Ympäristöpalvelimet, Jenkins ja Artifactory ovat asiakkaan itse ylläpitämiä CentOS-käyttöjärjestelmää käyttäviä palvelimia. Palvelimet sijaitsevat samassa sisäverkossa, mikä antaa mahdollisuuden nopeaan tiedonsiirtoon ja viiveettömään kommunikointiin. Artifactory ja Jenkins sijaitsevat samalla palvelimella niiden resurssien käytön luonteen vuoksi: Artifactory tarvitsee paljon levytilaa, mutta ei vie juuri muita resursseja. Jenkinsille jää siis suurin osa prosessori- ja muistiresursseista eikä järjestely hidasta kummankaan palvelun toimintaa merkittävästi. Jenkins tallentaa ja lataa pakettiartefakteja Artifactorysta. Lähdekoodin se hakee Git-versiohallinnasta ja hallitsee julkaisuprosessit kaikkiin suoritusympäristöihin. Jenkins on avainasemassa julkaisuprosessissa.

# 3 Ohjelmistojulkaisuprosessin vaiheet

Ohjelmistojulkaisuprosessi kuvaa vaiheet, jotka vaaditaan ohjelmistoon tehdyn muutoksen viemiseksi käyttöympäristöön (kuva 2) [2].


Kuva 2. Ohjelmistojulkaisuprosessin vaiheet.

Prosessi alkaa lähdekoodijärjestelmästä (engl. source code management), joka mahdollistaa koodin koonnin, testauksen ja lopulta julkaisun. Jokainen näistä vaiheista on jossain muodossa käytössä kaikissa maailman ohjelmistotuotantoprosesseissa. Organisaatio säilöö lähdekoodinsa keskitettyyn säilytysjärjestelmään, joka mahdollistaa versioinnin, muutosten seurannan, kehittäjien yhteistoiminnan ja laadunvarmistuksen. Koontivaiheessa ohjelmistokoodi käännetään ja paketoidaan. Täysin automatisoituna se mahdollistaa useiden pakettien tekemisen päivittäin, jopa samanaikaisesti. Paketti tallennetaan ulkopuoliseen järjestelmään, jossa pakettiversioita säilytetään. Luodulle koodipaketille ajetaan automatisoidut testit kuten yksikkötestit, integraatiotestit ja toiminnalliset testit. Ohjelmisto julkaistaan kohdeympäristöön, esimerkiksi testi-, esituotanto- tai tuotantoympäristöön. Julkaisuprosessin kehittyneisyydestä riippuen julkaisun voi tehdä täysin automaattisesti tai kehittäjän avustuksella.

## 3.1 Lähdekoodi

Organisaatio säilöö lähdekoodinsa keskitettyyn säilytysjärjestelmään, joka mahdollistaa versioinnin, muutosten seurannan, kehittäjien yhteistoiminnan ja laadunvarmistuksen. Tärkeintä julkaisuprosessin parantamisen kannalta on automatisoinnin mahdollisuus. Lähdekoodin muutoksia tarkastellaan koneellisesti ja koontivaihe käynnistetään niitä havaittaessa [2]. Lähdekoodin säilytys ja mahdollisuus automatisointiin ovat

ohjelmistojulkaisuprosessin perusta. Yleisin käytössä oleva versionhallintajärjestelmä Git on aktiivisesti ylläpidetty Linus Torvaldsin vuonna 2005 kehittämä ohjelmisto [3]. Muita versionhallinta-työkaluja on esimerkiksi Subversion (SVN), jonka käyttö on kuitenkin vähenemässä Gitin monipuolisemman toimintopaletin myötä. Git on hajautettu ja Subversion on keskitetty versionhallintajärjestelmä. Hajautetussa versionhallintajärjestelmässä kehittäjä kopio koko lähdekoodisäilön koneelleen ja tekee muutoksia paikalliseen säilöönsä. Paikallisen säilöön tehdyt muutokset voidaan yhdistää alkuperäiseen säilöön. Keskitetyssä versionhallintajärjestelmässä lähdekoodisäilö on verkkoyhteyden päässä ja kehittäjä tekee muutokset suoraan alkuperäiseen säilöön. Keskitetty säilö saattaa olla tietoturvan kannalta helpompi hallita ja käyttöönotto nopeampaa.

Lähdekoodin on oltava kaikkien siihen pääsyn tarvitsevien saatavilla. Git-versiohallintajärjestelmässä kehittäjät työntävät (engl. push) tekemänsä muutokset versiohallintaan haaroihin, joita yhdistetään toisiin haaroihin (engl. merge). Usein käytetään master- tai production-haaraa tuotannossa olevan koodin perustana ja development-haaraa kehitysympäristön koodipohjana. Kehittäjät yhdistävät ominaisuuskohtaiset haaransa ympäristökohtaisiin haaroihin vetoehdotuksilla (engl. pull request). Yhdistämisten jälkeen voidaan rakentaa julkaisupaketti. Julkaisupaketin rakentavalla järjestelmällä on oltava myös pääsy koodiin, ja useammat git-säilytysjärjestelmät tarjoavat autentikaatiotoimintoja koneelliseen käyttöön. Asiakkaalla on käytössä BitBucket lähdekoodin säilytysjärjestelmänä, joka on yksi suosituimmista git-palveluista. Lähdekoodijärjestelmään on mahdollista rakentaa erilaisia herätteitä (engl. webhooks), joiden avulla BitBucket voi ilmoittaa toiselle järjestelmälle tapahtumista lähdekoodisäilössä. Näiden tapahtumien avulla voidaan laukaista ohjelmistopaketin koonti, testaus tai julkaisun vienti.

## 3.2 Koonti

Lähdekoodi ei useimmiten ole muodossa, jossa sitä voitaisiin suorittaa. Koontivaiheessa lähdekoodi muunnetaan muotoon, jossa kohdejärjestelmä voi ymmärtää sitä. Projektissa käytetään Java-kieltä, joka käännetään koontivaiheessa Javan tavukoodiksi. Tavukoodia luetaan Java-tulkkien avulla (esimerkiksi Java Runtime Environment), jotka muuntavat sen lopulta ajonaikaisesti konekielelle.

Lähdekoodisäilössä säilytetään koodia, jota kehittäjät voivat muokata. Koodisäilössä ei ole järkevää säilyttää ulkoisia riippuvuuksia ja koodikirjastoja, koska ne voi ladata koontivaiheessa ennen kääntämistä. Ongelmana ovat kuitenkin mahdolliset muutokset ja tuen päättyminen riippuvuuskirjastoille. Järjestelmä, josta kirjastot ladataan, voi sulkeutua tai kohdeosoitteet voivat muuttaa muotoaan. Tällöin koontiprosessi ajautuu virhetilaan käyttäjästä riippumattomasta syystä. Riippuvuudet ja koodikirjastot voi kuitenkin säilyttää samassa ulkoisessa järjestelmässä, johon koottava paketti tallennetaan.

Projektissa pakettien ja kirjastojen säilömiseen käytetään JFrogin Artifactorya. Artifactorya voi käyttää esimerkiksi välikätenä virallisten Maven-säilöjen ja koontijärjestelmän välissä. Tällöin muutokset virallisessa Maven-säilössä eivät vaikuta koontivaiheeseen, koska riippuvuuskirjastot ovat saatavilla kehittäjien hallinnassa olevassa Artifactoryssa. Riippuvuuksien lataamisen käytetään projektissa Gradlea, joka on avoimen lähdekoodin koonnin automatisointityökalu [4]. Gradlella on mahdollista myös kääntää Java-koodia, mutta projektissa käytetty verkkokauppa-alusta sisältää oman kääntöprosessin, joka on toteutettu Apache Antilla. Apache Ant on Java-kirjasto ja komentoriviohjelma, jolla voidaan luoda prosesseja ja tavoitteita, jotka ovat riippuvaisia toisistaan. Useimmiten sitä käytetään Java-ohjelmien kääntämiseen ja julkaisupakettien luomiseen [5].

Käännetystä lähdekoodista muodostetaan usein paketti eli yksi tiedosto, joka sisältää kaiken tarvittavan ohjelmiston suorittamiseen. Paketti voi olla esimerkiksi zip-tiedosto, joka siirretään ulkoiseen järjestelmään Artifactoryyn säilytykseen. Toteutuksesta riippuen pakettiin voidaan sisällyttää järjestelmän vaatimat käyttäjätunnukset kuten tietokannan tunnukset, rajapintojen tunnukset tai salausavaimet tai ne voivat olla määritelty suoraan kohdejärjestelmään. Tarvittavat tunnukset voidaan myös viedä vasta julkaisuvaiheessa.

### 3.3 Testaus

Onnistuneen lähdekoodin kääntämisen jälkeen ohjelmistoa pitää suorittaa ja toimintoja voidaan testata. Ohjelmistojen testaustapoja ovat

- yksikkötestit
- integraatiotestit
- järjestelmätestaus
- hyväksyttämistestaus
- regressiotestaus.

Automaattisen testauksen näkökulmasta helpoin tapa on yksikkötestit. Yksikkötestaaminen sisältää ohjelmiston yksittäisten komponenttien testauksen ja sen päätarkoituksena on tarkistaa, että ohjelmiston itsenäiset osat toimivat suunnitellulla tavalla [6]. Valitettavasti yksikkötestaus jää helposti tekemättä huonosti suunnitelluissa ohjelmistoprojekteissa, ja niiden puute aiheuttaa kumulatiivista korjausvelkaa ohjelmistolle. Vaikka toiminnallisuus on nopeampaa toteuttaa ilman testausta, testien puuttuminen vaikeuttaa kehitystä ohjelmiston laajentuessa. Yksikkötestit luovat perustan ohjelmiston toiminnan testaukselle ja laajasti käytettynä antavat turvaa kehittäjille muuttaa olemassa olevaa lähdekoodia. Yksikkötestien kattavuuden ja tehokkuuden mittaavia työkaluja on saatavilla kuten mutaatiotestaus, jossa lähdekoodia muutetaan ja tarkastetaan epäonnistuvatko yksikkötestit. Yksikkötestien tulee epäonnistua, jos koodia mutatoidaan eikä se enää toimi niin kuin on tarkoitettu. Asiakkaalla on välttävästi käytössä yksikkötestejä eikä niitä ajeta säännöllisesti. Testaus on keskitetty hyväksyttämistestaukseen: kehittäjä testaa toiminnot omassa kehitysympäristössään. Asiakas testaa toteutetut toiminnot testiympäristössä, jonka jälkeen muutokset siirretään QA-ympäristöön. QA-ympäristössä testataan tarvittaessa integraatiomuutokset ja muut muutokset, joita ei pystytä testamaan testiympäristössä.

### 3.4 Julkaisu

Testauksen hyväksymisen jälkeen uusi ohjelmistoversio julkaistaan suoritusympäristöön. Julkaisuvaiheessa voidaan luoda tarvittavat asetustiedostot ohjelmistolle, jos niitä ei ole tehty koontivaiheessa. Asetustiedostojen luominen julkaisuvaiheessa antaa mahdollisuuden käyttää samaa julkaisupakettia kaikkiin ympäristöihin, koska paketit eivät ole ympäristökohtaisia. Asetustiedostot sisältävät usein arkaluontoisia tietoja, joita ei haluta päästää ulkopuolisten käsiin, kuten

tietokannan käyttäjätunnuksia ja salausavaimia. Nämä tiedot ovat kuitenkin säilytettävissä versionhallinnasta, kunhan tarpeellisesta suojauksesta huolehditaan. Versionhallinnassa muutoksia pystytään seuraamaan ja paluu aiempiin versioihin on mahdollista myös asetustiedostojen osalta.

Git Crypt on eräs työkalu, jolla suojauksen voi toteuttaa git-versionhallinnassa. Git Cryptillä on mahdollista salata läpinäkyvästi tiedostoja käyttäessä git-ohjelmistoa koodisäilönä [7]. Ohjelma käyttää salaamisessa ja sen purkamisessa apuna GNU Privacy Guardia (GPG). Git Crypt mahdollistaa myös käyttäjien, joilla ei ole oikeutta avata salattuja tiedostoja, tehdä muutoksia salaamattomaan koodiin ilman pääsyä salattuun sisältöön. GPG on OpenPGP-standardista tehty toteutus. GNU Privacy Guard antaa mahdollisuuden salata ja allekirjoittaa dataa ja viestejä [8]. GPG tarjoaa esimerkiksi salausavainparien hallinnan, komentorivityökalun ja tuen secure shellille (ssh).

Julkaisuprosessin tulee olla kokonaisuutena vakaa ja ennustettava. Prosessin tärkeimpiä huomioitavia ominaisuuksia ovat

- toistettavuus
- palautettavuus
- vaikuttavuus.

Julkaisuviennin prosessin tulee toimia jokaisella suorituskerralla samalla tavoin ja ennustettavasti. Suoritusympäristön ohjelmistojen riippuvuuksien versioiden tulee pysyä samana (ja niiden päivitys tulee tehdä hallitusti), joten tarkkoja versiointeja on käytettävä tai varmistettava mahdollisten versiovarianttien yhteensopivuus. Prosessin on myös selvittävä virhetilanteista: virhetilanteesta ilmoitetaan tietoa tarvitseville tahoille ja ongelma korjataan tai järjestelmä palautetaan viimeisimpään toimivaan tilaan.

Palautettavuus on ominaisuus, joka mahdollistaa järjestelmän palautuksen edelliseen tilaan (engl. rollback). Julkaisuviennin prosessissa tämä ominaisuus on elintärkeä, koska järjestelmää ei haluta jättää vialliseen tai keskeneräiseen tilaan virhetilanteen sattuessa. Tuotantoympäristöjen suhteen rollback-tilanteet tulisi olla harjoiteltu ja testattu toimivaksi tarpeeksi usein, jotta katastrofin sattuessa palautuksen tiedetään olevan mahdollista ja se osataan tehdä. Varmuuskopioiden säilyttäminen ja ottaminen on entistä halvempaa,

mutta valitettavasti alalla on silti esiintynyt vajaita ratkaisua, jotka ovat hyvin riskialttiita virheille ja saattavat luottaa liikaa palveluntarjoajan varmuuskopiointiratkaisuihin.

Vaikuttavuudella tarkoitetaan prosessin vaikutuksia muihin järjestelmiin ja käyttäjiin. Kuinka pitkä käyttökatko asiakkaille syntyy vai pystytäänkö julkaisu tekemään katkotta? Tuleeko monitorointijärjestelmille asettaa huoltokatko, jotta turhia hälytysilmoituksia ei lähde sidosryhmille? Ovatko integraatiot toteutettu niin, että katko ei aiheuta virheitä tuotuun dataan? Ohjelmiston rajapinnat muihin järjestelmiin on määriteltävä huolellisesti ja otettava huomioon julkaisuprosessissa, jos niitä ei ole toteutettu huomioimaan katkotilanteita.

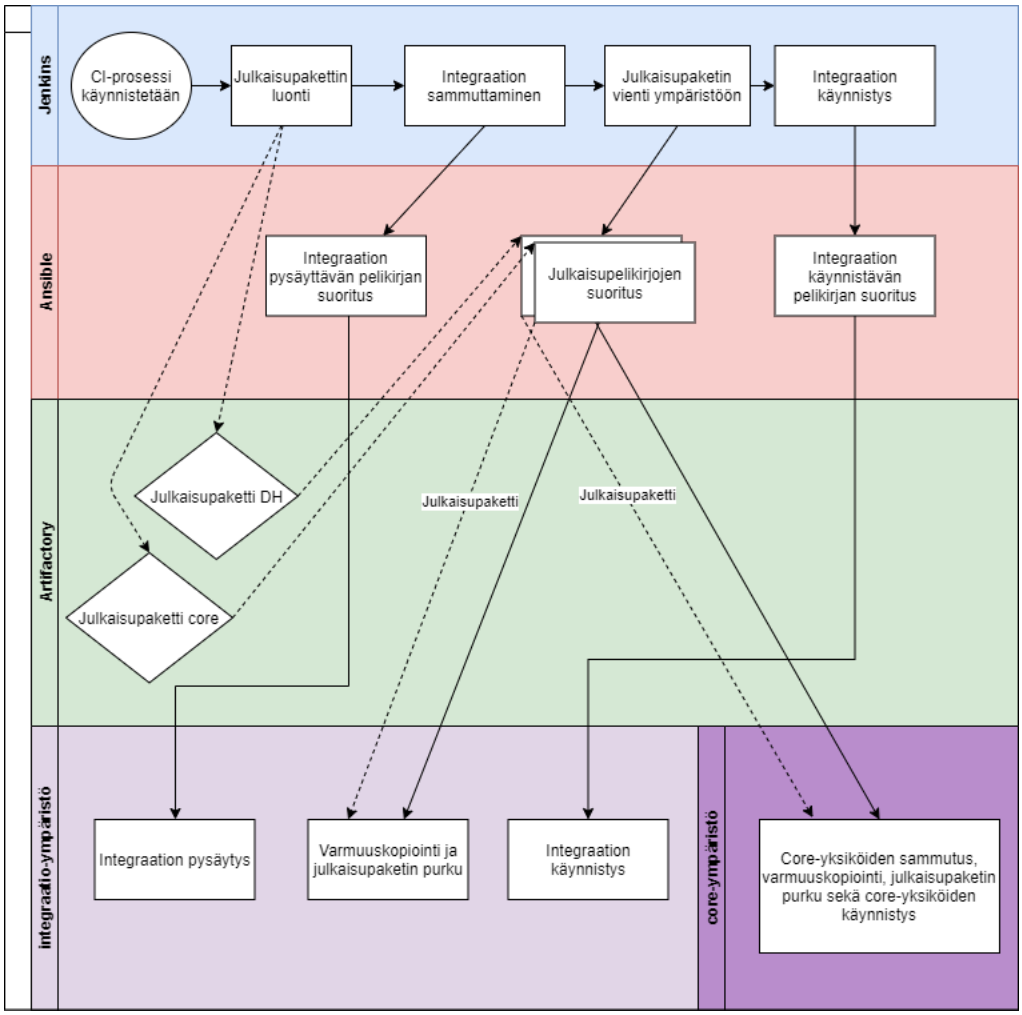
# 4 Jenkins-ohjelmistojulkaisun luominen ja vienti ympäristöihin

Jenkins on avoimen lähdekoodin automaatio-ohjelmisto, jota käytetään hallinnoimaan jatkuvan integraation prosesseja [9]. Palvelin tarkastelee lähdekoodimuutoksia versiohallinnasta. Jenkins aloittaa toimintoja havaitessaan muutoksia tai lähdekoodijärjestelmän herätteestä (engl. webhook). Toiminnot kuten lähdekoodin kääntäminen, testien ajaminen tai ohjelmistojulkaisun tekeminen voidaan määritellä yksityiskohtaisesti. Ne ovat suoritettavissa halutussa järjestyksessä niin, että suoritus keskeytyy virhetilanteessa. Keskeytyksestä ilmoitetaan asiasta vastuussa oleville tahoille, jos virheitä esiintyy tai määriteltyjä testejä ei läpäistä. Toimintoja voi suorittaa myös käyttäjän aloitteesta.

Projektissa Jenkinsillä toteutettiin koonti (engl. build) ja ohjelmistojulkaisu (engl. deployment). Jokaiselle vaiheelle toteutettiin oma Jenkins-putki (engl. pipeline), joiden määrittelyt ovat versionhallinnassa perinteisemmän Jenkins-töiden (engl. jobs) ollessa Jenkinsin tietokannassa. Näin määrittelyt löytyvät samasta lähteestä, muutokset niihin ovat helposti seurattavissa ja Jenkins-putket voi tarvittaessa luoda uudestaan vain muutamalla painalluksella sen sijaan, että määrittelyt jouduttaisiin kirjoittamaan uudestaan Jenkinsin käyttöliittymän kautta.

Ohjelmistojulkaisun luominen ja koonti aloitetaan Jenkinsistä käynnistämällä Jenkins-orkestrointiputki, joka hallinnoi muiden Jenkins-putkien suoritusta. Putkella on vaiheita,

joita suoritetaan ja niistä syntyy prosessi ohjelmistojulkaisun luomiseksi ja viemiseksi ympäristöihin (kuva 3).




Kuva 3. Ohjelmistojulkaisun tuonti- ja vientiprosessin vuokaavio

Ohjelmistojulkaisuprosessi aloitetaan käynnistämällä Jenkins-orkestrointiputki, joka käynnistää toisia Jenkins-putkia. Ensin käynnistetään julkaisupaketteja luovat putket, jotka tuottavat julkaisupaketit ja lataavat ne Artifactoryyn. Tämän jälkeen sammutetaan integraatiojärjestelmä suorittamalla Ansible-pelikirja. Seuraavaksi julkaisupaketit viedään kohdeympäristöihin Ansible-pelikirjat suorittamalla ja lopuksi käynnistetään integraatiojärjestelmä omalla pelikirjallaan.

## 4.1 Lähdekoodin hallinta

Asiakkaalla on käytössä BitBucket lähdekoodin hallintajärjestelmänä. Asiakkaalla on kaksi lähdekoodiarkistoa (engl. repository), joihin on jaettu kaupan ydintoiminnot (engl. core) ja integraatiodatan käsittely. Kauppaa koskeva lähdekoodi tallennetaan core-arkistoon ja kehitys tehdään development-haaraan. Ominaisuudet ja korjaukset tehdään haaroihin, jotka ovat merkitty tikettijärjestelmän yksikäsitteisillä tunnuksilla. Haarat liitetään myöhemmin development- ja master-haaroihin. Development-haara on ensimmäinen sijainti, josta julkaisupaketteja tehdään ja viedään testipalvelimelle. Kun ensimmäinen testaus on suoritettu, liitetään development-haaraan aiemmin liitetyt ominaisuuskohtaiset haarat master-haaraan. Master-haarasta tehdään julkaisupaketit ensin laadunvalvontaympäristöön QA ja QA:lla testauksen hyväksymisen jälkeen tuotantoympäristöihin. Vastaavasti integraatioarkistossa käytetään pelkästään master-haaraa julkaisujen luonnin pohjana, koska integraatiojärjestelmää ei suoriteta testiympäristössä kuin poikkeustilanteissa. Sama logiikka kuitenkin pätee ominaisuuksien liittämisessä master-haaraan: jokainen ominaisuus on merkitty tikettijärjestelmän tunnuksella. Tikettijärjestelmänä käytetään Atlassianin Jiraa, joka on saman toimittajan tuote kuin BitBucket. Toimittaja tarjoaa integraatiot järjestelmien välille ja antaa mahdollisuuden yhdistää koodimuutokset tiketteihin. Merkityt ominaisuudet liitetään master-haaraan julkaisupakettia varten. Ennen julkaisupaketin tekemistä säilöjen lähdekooditila kirjataan ympäristökohtaisella versionumerolla (tag), jonka avulla paketin koontijärjestelmä tietää, minkä tilainen lähdekoodi halutaan rakentaa.

## 4.2 Julkaisupaketin luonti

Automaatiopalvelin Jenkins toimii rajapintana lähdekoodin ja palvelinympäristöjen välissä. Jenkins tuottaa lähdekoodista julkaisupaketteja Artifactoryyn, josta Jenkins hakee ne julkaisuvaiheessa.

Julkaisupaketit luodaan Jenkins-komentoputkilla. Komentoputket voi määritellä lähdekoodissa Groovy-kieleen pohjaavilla asetustiedostoilla, joissa asiakkaalla on käytetty kuvaavaa syntaksia (engl. declarative syntax). Kuvaavaa syntaksia validoidaan ennen suoritusta, mikä antaa yhden varmistusaskeleen toimintojen ennustettavuudelle. Toinen vaihtoehto olisi käyttää skriptipohjaista syntaksia, jossa validointia ei tehdä.

Skriptipohjainen syntaksi antaa enemmän vapauksia, mutta jättää vastuun kirjoittajalle komentojen oikeellisuudesta, jolloin alttius virheille on suurempi.

Kuvaavan syntaksin Jenkins-komentoputki koostuu suoritusympäristöä määrittelevästä osasta ja suoritusaskelista. Ympäristöä määrittelevässä osassa asetetaan mahdolliset ympäristömuuttujat, listataan käytetyt työkalut, putken suorittava agentti, mahdolliset lisäasetukset ja suoritusparametrit (esimerkkikoodi 1).

```
agent any
tools {
   jdk 'Java 1.8'
   ant 'Ant 1.9.1'
}
environment {
   ANT_HOME = "$WORKSPACE/bin/platform/apache-ant-1.9.1"
   ANT_OPTS = "-Xms1G -Dfile.encoding=UTF-8"
}
options {
   timestamps()
   buildDiscarder(logRotator(numToKeepStr: '50', artifactNumToKeepStr: '1'))
   timeout(time: 300, unit: 'MINUTES')
   disableConcurrentBuilds()
}
```

Esimerkkikoodi 1. Julkaisuputken työkalujen, ympäristömuuttujien ja valintojen määrittely.

Agentti-osio määrittää suorittavan agentin. Määritteeksi voi antaa 'any', jos kaikki käytettävät agentit ovat samanlaisia eikä agentin tyypillä ole merkitystä. Tools-osio määrittää putken käytettävissä olevat työkalut. Java ja Apache Ant tarvitaan koodin kääntämiseen. Environment-osio asettaa tarvittavat ympäristömuuttujat kuten Ant-ohjelman suorituksen kotihakemisto ja mahdolliset lisäasetukset. Options-osio sisältää putken tuloksen käsittelyä ja lokitusta koskevat määreet. Timestamps antaa lokille aikaleimat, buildDiscarder määrittää kuinka monta suorituksen lokia pidetään muistissa ja tallessa pidettävien artefaktien määrä. Timeout määrittää putken suorituksen aikakatkaisurajan. DisableConcurrentBuilds estää useamman saman tyyppisen putken suorittamisen samaan aikaan. Ympäristömäärityksillä pystytään luomaan toistettava ja yksinkertainen ympäristö, jossa koodin kääntämisen voi suorittaa ilman ulkopuolisia häiriötekijöitä. Vaihtoehtoisesti voitaisiin käyttää konttiteknologiaa suoritusympäristön luomiseen, mutta sitä ei koettu tarpeelliseksi tämän projektin osalta.

Parameters-osiota käytetään putkessa ehdollisten suoritusten määrittämiseen ja niille voidaan antaa nimi, oletusarvo (defaultValue) ja kuvaus (esimerkkikoodi 2).

```
parameters {
   string(name: 'GIT_TAG',
      defaultValue: '1.0.0',
      description: 'Version of the package')
   choice(name: 'TARGET_ENVIRONMENT',
      choices: 'test\nqa\nprod1\nprod2\nprod3',
      description: 'Target environment to build package for')
   booleanParam(name: 'PUBLISH_BUILD',
      defaultValue: true,
      description: 'Publish built package to Artifactory')
}
```

Esimerkkikoodi 2. Suoritusparametrien määrittely.

Suoritettavat vaiheet (stages) sisältävät putken suorittamat askeleet. Valmistaututumisvaihe (prepare) lokittaa tarvittavat parametritiedot ja muuttaa Artifactoryn URL-osoitteen Gradlessa asiakkaan oman Artifactoryn osoitteeksi, koska oletuksena koodissa käytetään toimittajan omaa Artifactory-osoitetta (esimerkkikoodi 3).

```
stage("Prepare") {
   steps {
      echo "Building version ${GIT_TAG} to environment ${TARGET_ENVIRONMENT}"
      echo "Releasing package to artifactory: ${PUBLISH_BUILD}"
   sh "git clean -xfd"
   //Set artifactory URL to custom Artifactory
   sh "sed -i -e 's/http:\\/\\/artifactory.regular.com\\/libs-
release/http:\\/\\/artifactory.custom.com:8081\\/artifactory\\/libs-release/g'
build.gradle"
   }
}
```

Esimerkkikoodi 3. Valmistautumisvaiheen määrittely.

Riippuvuuksien muokkausvaihe poistaa tietyt riippuvuudet ohjelmistolta, jos pakettia tehdään ympäristöön TEST, PROD2 tai PROD3 (esimerkkikoodi 4).

```
stage("Remove Translation.com extensions for TEST, PROD2 and PROD3") {
   when {
      anyOf {
         environment name: 'TARGET_ENVIRONMENT', value: 'test'
         environment name: 'TARGET_ENVIRONMENT', value: 'prod2'
         environment name: 'TARGET_ENVIRONMENT', value: 'prod3'
      }
      steps {
         sh "sh jenkins/removeTranslationsFromLocalExtensions.sh"
      }
   }
}
```

Esimerkkikoodi 4. Riippuvuuksien muokkausvaiheen määrittely.

Riippuvuuksien latausvaihe lataa Gradle-työkalua apuna käyttäen ohjelmiston vaatimat riippuvuudet: verkkokauppa-alustan ja mahdolliset ylimääräiset riippuvuudet (esimerkkikoodi 5).

```
stage('Dependencies') {
      steps {
      sh "./gradlew setupdeps cleantemp"
   }
}
```

Esimerkkikoodi 5. Riippuvuuksien latausvaiheen määrittely.

Asetusvaihe luo verkkokauppa-alustan käyttämälle tietokanta-ajurille viimeisestä päivityksestä kertova merkkitiedosto (esimerkkikoodi 6).

```
stage('Configuration') {
   steps {
      sh "touch bin/platform/lib/dbdriver/.lastupdate"
   }
}
```

Esimerkkikoodi 6. Tietokanta-ajurin päivitystiedoston asetusvaiheen määrittely.

Paketin koontivaihe suorittaa paketin koonnin käyttäen Apache Ant -työkalua (esimerkkikoodi 7).

```
stage('Build') {
   steps {
      sh "ant -f \
         bin/app/build.xml clean updateMavenDependencies production"
   }
}
```

Esimerkkikoodi 7. Julkaisun koodipaketin kansiorakenteen muodostusvaihe.

Paketointivaihe yhdistää käännetyn ohjelmiston kansiorakenteet ja lisää versionumeron kertovan merkkitiedoston sekä arkistoi pakettiartefaktin Jenkinsiin (esimerkkikoodi 8).

```
stage('Package') {
   steps {
      sh "mkdir temp/package"
      sh "unzip -qq \
         temp/app/server/server-extensions.zip \
         -d $WORKSPACE/temp/package"
      sh "unzip -qq \
         temp/app/server/server-basic.zip \
         -d $WORKSPACE/temp/package"
      // Append version information to a file
      dir("temp/package/app/") {
```

```
            sh "echo 'Version information' > version-info.txt"
            sh "echo 'Built from repository: ${GIT_URL}' >> version-info.txt"
            sh "id=\$(git log -1 --pretty=format:\"%h\") && echo \"Built from
commit id: \$id\" >> version-info.txt"
            sh "echo 'Version tag: ${GIT_TAG}' >> version-info.txt"
        }
        sh "cd temp/package && \
              mv app app6 && \
              zip -q -r ../customer-release-${GIT_TAG}.zip app6"
        archiveArtifacts artifacts: "temp/customer-release-${GIT_TAG}.zip",
              fingerprint: true
    }
}
```

Esimerkkikoodi 8. Paketointivaiheen määritys.

Julkaisuvaihe tallentaa pakettiartefaktin Artifactory-säilöön. Testipaketit viedään omaan säilöönsä, jolle on vilkkaampi poistosykli (esimerkkikoodi 9).

```
stage('Publish') {
   when {
      environment name: 'PUBLISH_BUILD', value: 'true'
   }
steps {
   script {
      def server = Artifactory.server 'artifactory.custom.com'
      def releasePath = 'project-release-local/customer-release'
      def uploadTarget = "$releasePath/customer-release-${GIT_TAG}.zip"
      if (env.TARGET_ENVIRONMENT == 'test') {
         uploadTarget = "$releasePath/customer-release-${GIT_TAG}.zip"
      }
      def uploadSpec = """{"files":[{
         "pattern": "temp/customer-release-${GIT_TAG}.zip",
         "target": "${uploadTarget}"
      }]}"""
      server.upload(uploadSpec)
      }
   }
}
```

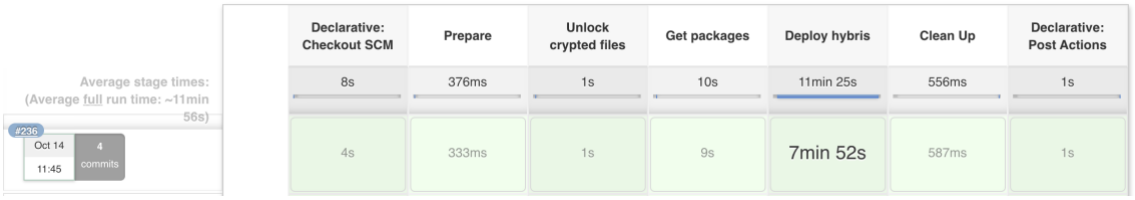
Esimerkkikoodi 9. Julkaisuvaiheen määritys.

Viimeinen vaihe poistaa lopuksi kaiken versionhallintaan kuulomattoman sisällön (esimerkkikoodi 10).

```
stages {
   stage('Clean Up') {
      steps {
         sh "git clean -xfd"
      }
   }
}
```

Esimerkkikoodi 10. Siivousvaiheen määritys.

Jenkins-putken suoritusta seurataan vaihekohtaisesti Jenkinsin graafisesta käyttöliittymästä tai konsolista suorituksen lokista (kuva 4).




Kuva 4. Jenkins-putken graafinen käyttöliittymä etenemisen seurantaan.

Mahdolliset virhetilanteet näkyvät graafisessa käyttöliittymässä vaiheen kohdalla punaisella ja onnistuminen puolestaan vihreällä. Komentoputken vaiheet näytetään sarakkeina, joiden solujen sisällä on vaiheen nimi ja suoritusaika. Suoritusaika ja lisätyt kommitoinnit verrattuna edelliseen ajoon näkyvät ajon järjestysnumeron alapuolella. Muuta informaatiota ei ole saatavilla näkymästä oletusasetuksilla, mutta tarkemmat virhetiedot löytyvät putken konsolilokista. Graafisen käyttöliittymään on mahdollista tuoda monenlaista tietoa lisäosien (engl. plugin) avulla.

## 4.3 Ohjelmistojulkaisun Jenkins-putki

Ohjelmistojulkaissussa käytetään kahta työkalua: Ansiblea ja Jenkinsiä. Jenkins-komentoputki suorittaa Ansible-pelikirjan (playbook), johon on määritelty prosessi ja askeleet julkaisupaketin viemiseksi ympäristöön. Lisäksi komentoputki suorittaa muita askeleita: esimerkiksi joissain vaiheissa pyydetään käyttäjän syötettä varmistukseksi, että tarvittavat manuaaliset toimenpiteet ovat suoritettu.

Julkaisunputken rakenne on pohjimmiltaan sama kuin julkaisun koontiputkessa: ympäristömuuttujat, parametrit, agenttimäärite ja vaiheet. Ensimmäisenä määritellään jälleen koko putkea koskevat määritellyt (esimerkkikoodi 11).

```
pipeline {
   agent any
   options
   timestamps
   buildDiscarder(logRotator(numToKeepStr: '50', artifactNumToKeepStr: '1
      timeout(time: 300, unit: 'MINUTES')
      disableConcurrentBuilds()
```

```
   parameters
      string(name: 'GIT_TAG',
         defaultValue: '1.0.0',
         description: 'Version of the package')
      choice(name: 'TARGET_ENVIRONMENT',
         choices: 'test\nqa\nprod1\nprod2\nprod3',
         description: 'Target environment to deploy to')
      booleanParam(name: 'SKIP_BACKUP',
          defaultValue: false,
          description: 'Do NOT take backups')
}
```

Esimerkkikoodi 11. Julkaisunputken ympäristömuuttujien ja työkalujen määrittely.

Valmistautumisvaihe lokittaa valitut muuttujat ja ajaa ”git clean”. Se poistaa kaikki koodisäilön ulkopuoliset tiedostot kansiorakenteesta (esimerkkikoodi 12).

```
stage('Prepare') {
   steps {
      echo "Deploying version $GIT_TAG to $TARGET_ENVIRONMENT"
      echo "Skipping backups: $SKIP_BACKUP"
      sh "git clean -xfd"
   }
}
```

Esimerkkikoodi 12. Valmistautumisvaiheen määrittely.

Salauksen purkuvaihe avaa Git-cryptillä salatut arkaluontoiset tiedostot. Julkaisuvaihe purkaa salasanatiedoston. Salasanan avulla puretaan Ansible Vaultin salaus, joka sisältää tarvittavien käyttäjätunnusten salasanat (esimerkkikoodi 13).

```
stage('Unlock crypted files') {
   steps {
      sh "git-crypt unlock"
   }
}
```

Esimerkkikoodi 13. Salausvaiheen määrittely.

Pakettien latausvaihe lataa julkaistavat ohjelmistokoodipaketit (esimerkkikoodi 14).

```
stage('Get packages') {
   steps {
      script {
         def server = Artifactory.server 'artifactory.custom.com'
         def releasePath = 'project-release/customer-release'
          def downloadSpec = """
            {"files": [{
          "pattern": "$releasePath/customer-release-${GIT_TAG}.zip"
            }]}"""
            server.download(downloadSpec)
      }
      sh "mv customer-release/customer-release-${GIT_TAG}.zip ./ansible"
   }
}
```

Esimerkkikoodi 14. Pakettien latausvaiheen määrittely.

Ohjelmistojulkaisuvaiheessa Ansible-pelikirja suorittaa julkaisuoperaation. Pelikirjalle annetaan ympäristö (engl. environment), johon paketti viedään pelikirjan nimi (engl. playbook), lisämuuttujia (julkaisun Git-tagi ja mahdollisuus ohittaa varmuuskopioiden ottaminen) sekä lisävalinnat Ansible-komentorivityökalulle (Vaultin salasanan sisältävän tiedoston polku) (esimerkkikoodi 15).

```
stage('Deploy core') {
   steps {
      dir("$WORKSPACE/ansible") {
         ansiblePlaybook(
            inventory: 'environments/$TARGET_ENVIRONMENT/inventory',
            playbook: 'core-release.yml',
            extraVars: [release_tag: '$GIT_TAG',
                  skip_backup: '$SKIP_BACKUP'],
                  extras: '--vault-password-file files/vault-pass')
      }
   }
}
```

Esimerkkikoodi 15. Julkaisunvaiheen määrittely.

Siivousvaihe poistaa jälleen kaikki versionhallintaan kuulumattoman sisällön (esimerkkikoodi 16).

```
stage('Clean Up') {
   steps {
      sh "git clean -xfd"
}
```

Esimerkkikoodi 16. Siivousvaiheen määrittely.

Jälkitoimintovaihe lukitsee Git Cryptillä suojatut tiedostot. Tämä askel suoritetaan aina, kun Jenkins putken suoritus loppuu, myös virhetilanteessa (esimerkkikoodi 17).

```
post {
   always {
      sh "git-crypt lock"
   }
}
```

Esimerkkikoodi 17. Jälkitoimintovaiheen määrittely.

## 4.4 Orkestroivat Jenkins-komentoputket

Julkaisu- ja tuotantoputkia voi ajaa yksittäin, mutta on hyödyllistä luoda Jenkins-komentoputkia, jotka suorittavat muita Jenkins-komentoputkia. Näitä ohjaavia ja suorittavia yksiköitä kutsutaan orkestroiviksi Jenkins-komentoputkiksi. Orkesteroivaa putkea voi käyttää tekemään julkaisuprosessista mahdollisimman vaivaton ja kokoamaan kaikki askeleet yhden putken alle. Ilman orkestrointiputkia käyttäjän pitää käynnistää erikseen useampia julkaisupaketin luonti- ja julkaisupaketin vientiputkia ja pitää huolta, että järjestelmät sammutetaan oikeassa järjestyksessä. Orkestrointiputki hoitaa kaiken tämän käyttäjän puolesta, ja suorituksen kokonaiskestosta jää merkintä, jonka avulla voi paremmin arvioida päivityksen aiheuttaman katkon kesto. Kuvassa 2 kuvattu ohjelmistojulkaisun luontiprosessi ja ympäristöön vientiprosessi voidaan kuvata kokonaisuudessaan orkesteroivalla putkella, kun prosessin askeleet on jaettu erillisiin Jenkins-putkiin ja saadaan eheä, yksi tapa suorittaa prosessi.

Putkia on mahdollista käynnistää rinnakkain niin, että orkesteroiva putki hallinnoi suoritusta ja yhdenkin suorituksen päättyessä virheeseen voidaan kaikki muut työt keskeyttää välittömästi. Rinnakkaista suoritusta on järkevää käyttää esimerkiksi julkaisupakettien luonnissa ja julkaisussa core- ja integraatioympäristöön (esimerkkikoodi 18).

```
stage('Build packages') {
   failFast true
      parallel {
         stage('Build core') {
           when {
              environment name: 'BUILD_CORE', value: 'true'
           }
           steps {
              build (
                 job: 'Core_RELEASE',
                 propagate: true,
                    parameters: [
                       string(name: 'GIT_TAG', value: "${GIT_TAG}"),
                       string(name: 'TARGET_ENVIRONMENT',
                          value: "${TARGET_ENVIRONMENT}"),
```

```
                        booleanParam(name: 'PUBLISH_BUILD', value: true)
                    ]
                )
            }
        }
        stage('Build integration) {
            when {
                environment name: 'BUILD_INTEGRATION', value: 'true'
            }
            steps {
                build (
                    job: 'integration_RELEASE',
                    propagate: true,
                    parameters: [
                        string(name: 'GIT_TAG', value: "${GIT_TAG}"),
                        booleanParam(name: 'PUBLISH_BUILD', value: true)
                    ]
                )
            }
    }
  }
}
```

Esimerkkikoodi 18. Orkestroivan Jenkins-komentoputken rinnakkainen ohjelmistopaketin luontiprosessin määrittely.

Ohjelmistojulkaisuprosessin jakaminen mahdollisimman pieniin osiin mahdollistaa resurssien tehokkaan jakamisen: Enemmän prosessointia ja levynnopeutta vaativa julkaisupaketin luonnin voi ohjata Jenkins-agentille, jolla on enemmän resursseja ja julkaisupaketin viennin voi tehdä pienemmillä resursseilla. Orkestroiva komentoputki taas vaatii vain minimaalisen määrän resursseja sen antaessa vain käynnistyskäskyjä muille putkille tai tekemälle muita nopeita operaatioita.

Kun julkaisupaketin luonti ja vienti ovat erillisiä putkia, niitä voidaan käyttää helposti orkestroivassa putkessa ja lisätä tarvittaessa manuaalisia tai ehdollisia askeleita tarpeen tullen. Ennen core-paketin vientiä tarvitsee katkaista integraatioiden lähetys integraatiojärjestelmälle. Manuaalisen vahvistuksen vaativan vaiheen lisääminen putkeen on hyödyllistä ja antaa turvaa putken suorittajalle siitä, että kaikkien käyttäjän syötettä tarvitsevien askeleiden suorittamisesta ilmoitetaan erikseen (esimerkkikoodi 19).

```
stage('MANUAL STEP: Integration Maintenance Mode) {
   when {
      anyOf {
         environment name: 'DEPLOY_CORE, value: 'true'
         environment name: 'DEPLOY_INTEGRATION', value: 'true'
      }
   }
   steps {
      input "Set Integration Maintenance Mode on ${TARGET_ENVIRONMENT}. Click
proceed when Maintenance Mode has been set"
   }
}
```

Esimerkkikoodi 19. Manuaalinen askel Jenkins-komentoputkessaputkessa. Käyttäjän tulee vahvistaa putken suorituksen jatkaminen graafisessa käyttöliittymässä.

Yhden orkestroivan putken suorittaminen myös yksinkertaistaa vientiprosessin ja vähentää luettavan dokumentaation määrää putkea suorittavalle osapuolelle.

## 4.5 Ajastettu julkaisu

Jenkinsillä on mahdollista luoda ajastettuja julkaisuja. Se voi olla hyödyllistä ympäristöille, joille voi julkaista ilman käyttökatkoa tai jos käyttökatkosta ei ole suurta haittaa. Jatkuvan viennin toteuttaminen on myös mahdollista. Useimmiten frontend-sovellukset ovat vaivattomasti julkaistavissa jatkuvasti ilman ajastusta. Backend-sovelluksissa julkaisu saattaa vaatia enemmän suunnitelmallisuutta riippuen ohjelmiston arkkitehtuurista ja sidonnaisuuksista. Projektissa käytettiin ajastettua julkaisua julkaisuprosessin pitkän keston ja backend- ja frontend-sovelluksen vahvan keskinäisen sidonnaisuuden vuoksi.

Ympäristöön TEST on toteutettu ajastettu julkaisu. Jenkins tarkastelee git-säilön tilaa ja vie uuden version ohjelmistosta palvelimelle joka päivä kello 11.00, jos kehityshaarasta löydetään muutoksia verrattuna viime tarkastelukertaan. Aiemmin mainitulle orkestrointiputkelle tulee antaa parametrit suoritusta varten, mikä ei ole mahdollista ajastuksella. Varsinkin kun käytetyt parametrit ovat esimerkiksi git-tagi, joka tulee lisätä ensin git-säilöön, jotta Jenkins osaa hakea oikean koodin. Ratkaisuksi TEST-ympäristöllä on oma orkestroiva putki. Sitä voi ajaa automaattisesti, koska putki luo ohjelmallisesti tarvittavat muuttujat paketin luonti- ja julkaisuputkien suoritukseen. Putken toteutus on suurilta osin sama kuin orkestroivan putken. Erona on vaihe, jossa parametrit määritellään: GIT_TAG-muuttujaan merkitään päivämäärä ja aika sekä asetetaan muut tarvittavat muuttujat (esimerkkikoodi 20).

```
stage('Configure') {
   steps {
      script {
      java.time.LocalDateTime dateTime = java.time.LocalDateTime.now();
      env.GIT_TAG =
dateTime.format(java.time.format.DateTimeFormatter.ofPattern("uuuuMMdd-
HHmmss"));
      env.TARGET_ENVIRONMENT = 'test';
      env.BUILD_CORE = Boolean.parseBoolean('true');
      env.DEPLOY_CORE = Boolean.parseBoolean('true');
      env.SKIP_BACKUP = Boolean.parseBoolean('true');
      }
   }
}
```

Esimerkkikoodi 20. Putkien ajoon vaadittavien parametrien asetus.

TEST-ympäristölle on myös oma paketin luontiputki. Siihen ei ladata koodia git-tagin mukaan vaan suorituksen aikainen koodi kehityshaarasta. Tällöin GIT_TAG-muuttujaa käytetään vain versiopaketin nimeämiseen sekä versiotiedoston merkintään.

# 5 Ansible

Ansible on avoimen lähdekoodin ohjelmistojen ja asetusten hallintaan sekä ohjelmistojulkaisujen tekemiseen tarkoitettu työkalu UNIX-tyyppisille käyttöjärjestelmille [10]. Ansible sopii vakaiden ja toistuvien toimintojen suorittamiseen niin, että lopputulos on aina sama riippumatta toistokerroista (idempotenttisuus). Toiminnot voivat olla esimerkiksi tiedostojen kopiointia palvelimille, palveluiden hallintaa tai varmuuskopioiden tekemistä. Ohjelmisto on kirjoitettu Pythonilla, joten sen asennus vaaditaan käytettävässä järjestelmässä. Tärkeitä osia Ansiblelle ovat pelikirjat (engl. playbook), isännät (engl. hosts), roolit (engl. roles) ja tehtävät (engl. tasks). Niiden avulla voi rakentaa prosessin, jolla suoritetaan esimerkiksi ohjelmistojulkaisun tekeminen tai ympäristön luonti.

## 5.1 Ansible-pelikirjat

Ansible-pelikirja on tapa tallentaa ja suorittaa Ansiblen konfiguraatio-, julkaisu- ja orkestrointitoimintoja [11]. Tärkeimmät määrittelyt ovat isännät ja roolit. Pelikirjat ovat toisin sanoen ylätason ohjeita Ansiblen funktioiden suorittamiseen (esimerkkikoodi 21).

```
---
```

```
- hosts: core-app
  remote_user: ssh_user
  roles:
   - role: notify-flowdock
     flowdock_message: "{{ deployment_begin_message }}"
   - core-prepare
   - core-shutdown
   - core-release
   - core-startup
   - backups-clean
   - role: notify-flowdock
     flowdock_message: "{{ deployment_end_message }}"
```

Esimerkkikoodi 21. Core-ympäristöön vientipelikirjan määritelmä.

Isännät ovat pelikirjan toiminnan kohde. Tämä parametri määrittää isännät, joille kyseessä oleva vaihe suoritetaan. Esimerkiksi määre ”all” ajaa toiminnon kaikille ja yksittäinen määre taas pelkästään yhdelle isännälle. Isäntäkonfiguraatiot ovat ympäristökohtaisia.

Roolit ovat toimintoja, joita suoritetaan. Roolit voivat olla isäntäkohtaisia tai yhteisiä. Roolit sisältävät tehtäviä, joita suoritetaan määritellyssä järjestyksessä. Roolit ovat järkevää jakaa toimintojen mukaan osiin, jolloin niitä on helppo käyttää uudelleen pelikirjoissa. Esimerkiksi julkaisupaketin vienti voi olla eräs rooli, tai sen voi jakaa pienempiin osiin kuten sammutus-, käynnistys- ja varmuuskopiointirooleihin.

## 5.2 Ansible-tehtävät

Ansible-tehtävät ovat toimintayksiköitä [12], jotka käyttävät Ansible-moduuleita toimintojen suorittamiseen. Tehtävien kirjoittamisessa suositellaan käytettäväksi idempotenttisuusperiaatetta ja tehtävän tulisi suorittaa vain yksi toiminto. Tehtävien syntaksi rajoittaa useamman toiminnon suorittamista, mutta usein käytetyimpiin toimintoihin saattaa olla jo luotuna esimerkiksi shell-skriptejä ennen Ansiblen käyttöönottoa. Nämä skriptit saatetaan vain siirtää suoritettavaksi tehtäviin, jolloin Ansiblen rooli jää vain järjestelmälliseksi skriptinsuorittajaksi. Tehtävissä on suositeltavampaa käyttää Ansible-moduuleita, jolloin syötettä, ulostuloa ja parametreja on helpompi hallita. Myös idempotenttisuutta pystytään helpommin noudattamaan, koska se on sisäänrakennettuna moduuleiden syntaksissa. Tätä periaatetta ei ole täysin noudatettu projektissa, koska osa tehtävistä on yleiskäyttöisiä vanhoja määrittelyitä, joita ei ole täysin päivitetty. Esimerkiksi seuraava komentoriviltä suoritettava komento:

```
- name: Add execute rights to scripts
  shell: find . -name "*.sh" -exec chmod +x {} \;
  when: "core_installed.stat.exists == True"
  tags: core shutdown
```

Esimerkkikoodi 22. Suoritusoikeuden lisääminen skripteille.

Suoritusoikeuden asettaminen kaikille skripteille on tietoturvariski. Tiedetään, ettei muille skripteille kuin ohjelman käynnistysskriptille pitäisi olla tarvetta muuttaa suoritusoikeutta. Ansible-moduuli ’file’ tarjoaa syntaksin, jolla saman toiminnon voi suorittaa pelkästään tiedostolle, jonka arvoja halutaan muuttaa (esimerkkikoodi 23).

```
- name: Add execute rights to scripts with Ansible file module
  file: {{core_root}}/start.sh
  mode: u+x
  when: core_installed.stat.exists|default(false)|bool
  tags: core shutdown
```

Esimerkkikoodi 23. File-moduulin käyttö tiedoston oikeuksien asettamiseksi.

Käynnistysskriptille voi lisätä suoritusoikeuden käyttäen Ansiblen file-moduulia. Aikaisemmin suorituslupa annettiin kaikille käyttäjille. Oikeuden voi rajata vain käyttäjälle, joka käyttää käynnistysskriptiä palvelimella. Kannattaa huomioida myös päivitetty syntaksi when-arvolle.

## 5.3 Pelikirjan suoritus

Projektissa on omat pelikirjansa ja tehtävänsä core- ja integraatio -ympäristöihin. Päävaiheet ovat kuitenkin samat: valmisteluvaihe, ohjelman sammutus, varmuuskopiointi, julkaisupaketin kopiointi, konfiguraatiotiedostojen määrittely, ohjelman käynnistys, käynnistymisen varmistustarkastukset ja siivous.

Valmisteluvaihe tarkastaa, että tarvittava kansiorakenne on palvelimella ja luo sen tarvittaessa sekä kopioi julkaisupaketin ympäristöihin. Valmisteluvaiheelle on oma roolinsa core-prepare. Rooli sisältää tehtävätiedoston, joka suoritetaan oletuksena roolia suorittaessa. Kansiorakenteen tarkastuksen voi tehdä helposti käyttäen Ansiblen file-moduulia (esimerkkikoodi 24)

```
- name: Verify core folder
  file: path={{core _root}}/ state=directory
  tags: prepare core
```

Esimerkkikoodi 24. File-moduulin käyttö kansion olemassaolon tarkastukseen ja luomiseen.

Julkaisupaketin kopiointi tehdään käyttäen copy-moduulia (esimerkkikoodi 25).

```
- name: Copy the release package to the server(s)
  copy: src={{package_name}}.zip dest={{core_home}}/{{package_name}}.zip
  when: no_copy is not defined
  tags: core
```

Esimerkkikoodi 25. Julkaisupaketin kopiointi palvelimille.

Sammutukselle on oma rooli core-shutdown. Ohjelmisto on määritelty CentOS-palveluna (service), jolla voi hallita ohjelman tilaa. Palvelut tarjoavat sammutuksen, käynnistyksen ja statuksen tarkastamisen lisäksi mahdollisuuden käynnistää ohjelma automaattisesti palvelimen käynnistyessä. Oikein toteutettujen CentOS-palveluiden avulla palvelinta uudelleen käynnistävän operaattorin ei tarvitse tietää, mitä ohjelmia palvelimella suoritetaan. Ansible tarjoaa service-moduulin palveluiden hallintaan (esimerkkikoodi 26).

```
- name: Shutdown core service
  become: true
  service:
    name: tomcat
    state: stopped
  when: tomcat_service.stat.exists and
        (not operation_target_server is defined)
```

Esimerkkikoodi 26. CentOS-palvelun pysäyttäminen service-moduulilla.

Kun Become-määre on totta, oikeudet ylennetään root-tasolle, jota tarvitaan palveluiden hallintaan.

Seuraavaksi Ansible-pelikirja suorittaa core-release-roolin. Siinä varmuuskopioidaan ohjelmisto sekä tietokanta ja viedään uusi julkaisupakettiversio palvelimille. Varmuuskopiointi tehdään ohjelman pysäytyksen jälkeen. Ohjelma tulee pystyä palauttamaan varmuuskopioiden avulla täysin pysäyttämistä edeltävään tilaan. Sen vuoksi koko ohjelman kansiorakenne varmuuskopioidaan tietokannan lisäksi. Kansiorakenteen varmuuskopioinnin voi tehdä archive-moduulilla (esimerkkikoodi 27).

```
- name: Backup the core directory
  archive:
    path: "{{core _root}}/*"
    dest: "{{backup_root}}/{{ansible_date_time.date}}/{{backup_core
_file}}.tar.gz"
    exclude_path: "{{core_root}}/log"
  when: skip_backup is undefined or not skip_backup|bool
  tags: backup
```

Esimerkkikoodi 27. Kansiorakenteen varmuuskopiointi archive-moduulilla.

Tietokanta varmuuskopioidaan mysqldump-työkalulla, joka on MySQL-tietokannoissa käytetty datan viemistyökalu (esimerkkikoodi 28).

```
- name: Backup the core database
  shell: mysqldump -u{{db_username}} -p{{db_password}} -h {{db_host}}
{{db_name}} | gzip >
{{backup_root}}/{{ansible_date_time.date}}/{{db_name}}_dbdump.sql.gz
  when: skip_backup is undefined or not skip_backup|bool
  tags: backup
```

Esimerkkikoodi 28. MySQL-tietokannan varmuuskopiointi mysqldump-työkalulla.

Varmuuskopioiden valmistuttua uusi ohjelmisto viedään ympäristöön file- ja unarchive-moduuleilla. Pakattu julkaisupaketti on siirretty palvelimille, joten ensin poistetaan vanha koodipohja, joka sijaitsee bin-kansiossa, ja puretaan sen jälkeen uusi koodi kansiorakenteeseen (esimerkkikoodi 29)

```
- name: Remove old core/bin
  file: path={{core_root}}/bin state=absent
  tags: core

- name: Unpack and install the new version
  unarchive:
    src: {{core_home}}/{{package_name}}.zip
    dest: {{core_home}}
    remote_src: True
  tags: core
```

Esimerkkikoodi 29. Vanhan ohjelmakoodin poisto ja julkaisupaketin purku unarchive-moduulilla.

Ohjelmakoodin purun jälkeinen vaihe asettaa useita ympäristökohtaisia asetustiedostoja: verkkokauppa-alustan asetustiedostoja, Spring-ohjelmistokehyksen turvallisuusasetuksia ja SSO-kirjautumisen vaativia sertifikaatteja. Asetustiedostojen luonnissa voi käyttää Ansible-sapluunoita, joiden avulla luodaan ehdollista sisältöä parametrikirjastosta. Esimerkiksi tietokantayhteyden käyttäjätunnuksen ja salasanan sisältämät asetusmuuttujat luodaan käyttäen Ansiblen-muuttujia (esimerkkikoodi 30).

```
db.url={{ db_url }}
db.driver=com.mysql.jdbc.Driver
db.username={{ db_username }}
db.password={{ db_password }}
```

Esimerkkikoodi 30. Ansible-sapluunan määrittely tietokannan asetuksille.

Ansible-sapluunat ovat roolikohtaisia ja niitä voi käyttää template-moduulin avulla (esimerkkikoodi 31).

```
- name: Copy properties template
  template: src=core.properties dest={{core_root}}/config/core.properties
owner=ssh_user group=ssh_user mode=644 backup=false
  tags: core
```

Esimerkkikoodi 31. Ansible-sapluunan käyttäminen asetustiedoston luomiseen.

Core-startup-rooli käynnistää ohjelmiston Ansiblen service-moduulilla. Käynnistymistä tarkkaillaan kahdella tavalla: varmistetaan, että ohjelma kuuntelee määriteltyä porttia ja palauttaa versiosivun käynnistyksen valmistuttua. Tarkastus tehdään kahden minuutin kuluttua, koska ohjelman käynnistäminen vie noin kolme minuuttia. Ansiblen wait_for-moduuli tarkastaa, että ohjelma käynnistyy porttiin 80 (esimerkkikoodi 32).

```
- name: Wait for core to start on port 80 after two minutes
  wait_for: port=80 delay=120
  when:
    (wait_for_startup is defined) and
    (wait_for_startup|bool) and
    (operation_target_server is not defined)
  tags: core startup
```

Esimerkkikoodi 32. Portin kuuntelun tarkistus wait_for-moduulilla.

Käynnistyksen jälkeinen tehtävä odottaa, että versiosivusto palauttaa onnistuneen http-tilakoodin. Tarkistus tehdään Ansiblen uri-moduulilla (esimerkkikoodi 33).

```
- name: Wait for version page and fail after six minutes
  uri:
   url: "http://localhost:80/version"
  register: result
  until: result.status == 200
  retries: 17
  delay: 20
  tags: core startup
```

Esimerkkikoodi 33. URL-osoitteen vastauksen http-tilakoodin tarkistus uri-moduulilla

Siivousvaihe suorittaa backups_clean-roolin ja poistaa määritettyä aikarajaa vanhemmat varmuuskopiot, jotta palvelimelta ei kuluteta turhaan levytilaa. Tiedostot etsitään käyttämällä find-moduulia ja poisto tehdään file-moduulilla (esimerkkikoodi 34).

```
- name: Listing the oldest backup folders
  find:
    paths: "{{backup_root}}"
    age: "{{backup_age_days}}d"
    recurse: yes
    file_type: directory
  register: cleanup

- name: Remove oldest backups
  file: path="{{item.path}}" state=absent
  with_items: "{{cleanup.files}}"
```

Esimerkkikoodi 34. Vanhojen varmuuskopioiden poisto find- ja file-moduuleilla.

# 6 Kohti jatkuvaa toimitusta

Jatkuva julkaisu (engl. continuous deployment) on julkaisustrategia, jossa mikä tahansa koodi, joka läpäisee automatisoidun testivaiheen, julkaistaan suoraan tuotantoympäristöön [13]. Jatkuva julkaisu poistaa ihmisen tekemät varmistukset laadunvalvonnasta ja luottaa lähes täysin automatisoituun laadunvarmistukseen (engl. quality assurance). Jatkuva toimitus (engl. continuous delivery) on löyhempi julkaisustrategia, jossa uutta koodia tuotetaan jatkuvasti testattavaksi ja laadun varmistuksen käyttöön. Tässä projektissa olisi mahdollista käyttää hyödyksi osittain jatkuvan julkaisun periaatteita.

Jatkuvan julkaisun periaatteista mielenkiintoisin on kirjoittajan mielestä automaattinen laadun varmistus. Sitä voi toteuttaa monilla tavoin: yksikkötestit, integraatiotestit, järjestelmätestaus, hyväksyttämistestaus ja regressiotestaus.

Asiakkaan mukautetussa koodipohjassa on tehty todella vähän yksikkötestejä, joten kattavan testimäärän toteuttaminen ei ole kustannusten takia mahdollista. Integraatio-testien toteuttaminen taas olisi mahdollista. Integraatioita on muun muassa identiteettipalveluun ja asiakkuudenhallintajärjestelmään. Kummatkin järjestelmät ovat välttämättömiä verkkokaupan käytettävyyden kannalta, ja niiden toimivuus voitaisiin varmistaa integraatiotestein. Eniten hyötyä olisi luultavasti regressiotestauksesta eli testiaineistosta, joka testaa ohjelmiston nykyistä toimintaa. Regressiotestauksella

varmistetaan, etteivät muutokset ohjelmistoon riko aiemmin toteutettuja, toimivia ominaisuuksia. Hyväksyttämistestausta tekee asiakas manuaalisesti.

Automatisoidut ja ohjelmistoarkkitehtuurin osia valvovat testit voisivat mahdollistaa jatkuvan julkaisun tulevaisuudessa, jos se koetaan hyödylliseksi. Ohjelmiston laadun kannalta muutos olisi tarpeen ja antaisi turvaa siitä, että tällä hetkellä henkilöityneet testiprosessit eivät häviä työntekijän lähtiessä yrityksestä. Varsinkin jos nykyisellä verkkokauppa-alustalla halutaan jatkaa ja alustapäivitys tulee ajankohtaiseksi. Tällöin kattava automatisoitu testiaineisto antaa merkkejä päivityksen aiheuttamista virheistä ja helpottaa virheiden löytämistä sekä korjaamista.

# 7 Yhteenveto

Asiakkaan ohjelmistojulkaisuprosessi oli vanhentunut ja riippuvainen yksittäisten kehittäjien koneiden ympäristöistä. Julkaisuprosessin kestoon, ongelmiin ja virhetilanteisiin ei ollut näkyvyyttä. Julkaisut olivat hankalia suorittaa ja vaativat kattavat ohjeet virheiden välttämiseksi. Julkaisun tekeminen vaati työkalujen asentamista kehittäjän koneelle, joten suoritusjärjestelmät eivät olleet välttämättä yhteneväisiä.

Työn tavoitteena oli luoda varma, helppokäyttöinen ja aiempaa nopeampi prosessi ohjelmistojulkaisun viemiseksi ympäristöihin. Prosessi tuli olla dokumentoitu ja mahdollisimman automatisoitu. Jenkinsin avulla toteutettiin järjestelmä, jossa koko prosessin voi suorittaa yhdestä paikasta, yhdellä hiiren painalluksella. Toistettavuus ja suorituksen keston ja laadun valvonta mahdollistettiin, kun suoritus siirrettiin yhteiskäytössä olevalle palvelimelle Jenkins-ohjelmistoon yksittäisen kehittäjän koneen sijasta.

Työn alkuvaiheessa esiteltiin työkalut ja prosessit, joiden puitteisiin ratkaisua toteutettiin. Seuraavaksi esiteltiin tapa toteuttaa automatisoitu julkaisuprosessi lähdekoodista käyttöönottoon Jenkinsin ja Ansiblen avulla. Työn lopussa pohdittiin mitä parannuksia ja lisäyksiä toteutukseen tulisi tehdä, jotta prosessi voitaisiin hyväksyä jatkuvan integraation tai jatkuvan ohjelmistojulkaisun periaatteiden mukaiseksi prosessiksi.

Raporttia voi käyttää ohjeena uuden julkaisuprosessin toteutukseen tai dokumentaationa tehdystä julkaisuprosessista. Se avaa perusteet automatisoidun julkaisun toteutukseen ja ymmärrystä prosessin eri vaiheista. Riippumatta toteutuksessa käytettävistä työkaluista, periaatteet pysyvät samoina ja automatisoinnin kasvaessa näiden periaatteiden käyttö ei tule vähenemään. Automatisoinnin mahdollisuuksia tullaan hyödyntämään tulevaisuudessa yhä enemmän ja laajemmin ohjelmistotuotannossa.

## Lähteet


1 Jfrog. Verkkoaineisto.
https://www.jfrog.com/confluence/display/JFROG/JFrog+Artifactory
Luettu: 19.09.2020

2 Hari's Technical Space. Verkkoaineisto.
https://haritibcoblog.com/2020/08/13/concepts-the-4-stages-of-the-ci-cd-pipeline
Luettu 20.03.2021

3 Atlassian. Verkkoaineisto.
https://www.atlassian.com/git/tutorials/what-is-git
Luettu 26.02.2020

4 Gradle. Verkkoaineisto.
https://docs.gradle.org/current/userguide/userguide.html
Luettu 21.03.2021

5 Apache. Verkkoaineisto.
https://ant.apache.org
Luettu 26.02.2020

6 PerformanceLab. Verkkoaineisto.
https://performancelabus.com/unit-testing-importance
Luettu 21.03.2021

7 Github. Verkkoaineisto.
https://github.com/AGWA/git-crypt
Luettu: 13.10.2020

8 The GNU Privacy Guard. Verkkoaineisto.
https://gnupg.org/
Luettu: 13.10.2020

9 Jenkins User Documentation. Verkkoaineisto.
https://www.jenkins.io/doc/
Luettu: 25.03.2021

10 Wikipedia. Verkkoaineisto.
https://en.wikipedia.org/wiki/Ansible_(software)
Luettu: 19.09.2020

11 Ansible Documentation. Verkkoaineisto.
https://docs.ansible.com/ansible/latest/user_guide/playbooks.html
Luettu: 16.10.2020

12 Ansible Documentation. Verkkoaineisto.
https://docs.ansible.com/ansible/latest/user_guide/basic_concepts.html#tasks
Luettu: 7.11.2020

13 What is continuous deployment? Verkkoaineisto.
https://searchitoperations.techtarget.com/definition/continuous-deployment
Luettu: 14.02.2021