Jatkuvan kehityksen toteuttaminen Node.js-rajapintasovelluksessa

Janne Fagerström

Haaga-Helia ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma
Opinnäytetyö
2023

Tiivistelmä

Tekijä
Janne Fagerström

Tutkinto
Tietojenkäsittelyn koulutusohjelma

Opinnäytetyön nimi
Jatkuvan kehityksen toteuttaminen Node.js-rajapintasovelluksessa

Sivu- ja liitesivumäärä
42 + 1

Tässä toiminnallisessa opinnäytetyössä laadittiin esimerkkirajapintasovellus, jonka ympärillä toimii jatkuvan kehityksen mukainen automaattinen kehitysputki. Tämän esimerkkisovelluksen tarkoituksena on toimia lähtökohtana uuden rajapintasovelluksen kehityksessä. Se sisältää yksinkertaisen Node.js-sovelluksen toteutuksen sekä automaattisen koodin analyysin, testauksen ja sovelluksen käyttöönoton käyttäen GitHub Actions -toimintaprosesseja.

Jatkuva kehitysputki on järjestelmä automatisoituja prosesseja, joiden tarkoitus on tehostaa ohjelmistokehitystä. Jatkuvat kehitysmenetelmät sisältävät tarkistusvaiheita, jotka takaavat kehitetyn ohjelmiston toimivuuden ja tekevät kehityksestä luotettavan ja vakaan prosessin. Kaikkia jatkuvan kehityksen prosesseja ei voida automatisoida täysin. Joissain tilanteissa täysin automaattiset ratkaisut eivät ole aina paras ratkaisu, kuten esimerkiksi arviointivaiheissa.

Ohjelmiston testaaminen on iso osa jatkuvaa kehitysprosessia. Erilaisia testausmenetelmiä ovat muun muassa yksikkötestaus, integraatiotestaus ja hyväksyntätestaus, ja niitä voidaan suorittaa kehitysputken eri vaiheissa.

Jatkuvan toimituksen ja käyttöönoton prosessista löytyy monenlaisia ohjelmiston julkaisumalleja. Erilaisten julkaisumallien tarkoituksena on luoda mahdollisimman luotettava, ja käyttäjille näkymätön prosessi uuden ohjelmistoversion käyttöönottoon. Oikean julkaisumallin käyttö riippuu ohjelmistosta ja asiakontekstista.

Opinnäytetyössä kehitetyn sovelluksen jokainen kehitysaskel kuvailtiin työssä. Tuotoksena syntyi esimerkkiprojekti, joka toimii hyvänä lähtökohtana uuden rajapintasovelluksen kehityksessä.

Asiasanat
Jatkuva kehitys, jatkuva integraatio, jatkuva testaus, jatkuva toimitus, jatkuva käyttöönotto

Sisällys

1 Johdanto

1.1 Opinnäytetyön tarkoitus ja tavoitteet

Jatkuva kehitys (tunnetaan myös nimellä CI/CD) on muodostunut ohjelmistokehityksessä keskeiseksi toimintamalliksi, jonka tarkoituksena on luoda ohjelmistokehityksestä automatisoitujen prosessien kautta luotettavampaa ja tehokkaampaa.

Opinnäytetyön tavoitteena oli luoda Node.js-rajapintasovellus, ja sen ympärille automatisoitu kehitysputki jatkuvan kehityksen käytänteitä mukaillen. Tämän esimerkkisovelluksen tarkoituksena on olla lähtökohta uuden rajapintasovelluksen kehityksessä, jota ohjelmistokehittäjät voivat käyttää uuden projektin aloituspisteenä.

Opinnäytetyön teoriaosuuteen on tarkoituksena kerätä ajankohtaista tietoa kirjallisuudesta jatkuvasta kehityksestä ja sen käytänteistä, jotka luovat tietoperustan esimerkkisovelluksen automatisoidun kehitysputken laatimiseksi. Sovellukseen laaditun kehitysputken tulisi kattaa jatkuvan kehityksen kaikki eri vaiheet, lähdekoodin muutoksista alkaen ohjelmiston käyttöönottoon asti.

Opinnäytetyön esimerkkirajapintasovellus on julkisesti saatavilla GitHub-sivustolta käyttöoikeutettuna MIT-lisenssillä (liite 1). Se sisältää esimerkit rajapintatoteutuksesta, lähdekoodin tarkistus- ja testausvaiheista sekä skripteistä, joilla automatisoituja prosesseja luodaan. Automatisoitu kehitysputki ja sen vaiheet kehitettiin käyttämällä Github Actions -prosesseja. Työssä esitellään myös, miten sovellus voidaan julkaista automaattisesti Heroku-palvelimelle.

Vaikka tämän työn sovelluspohja on toteutettu JavaScript-pohjaisilla työkaluilla, käsiteltyjä jatkuvan kehityksen konsepteja pystytään helposti mukailemaan muihin kehitysympäristöihin ja ohjelmointikieliin.

1.2 Opinnäytetyön rajaukset

Sovelluspohja on toteutettu Node.js-ajoalustalle. Tämä tarkoittaa sitä, että sovelluskehitykseen käytetyt työkalut ovat suurimmaksi osaksi JavaScript-pohjaisia, kuten koodin lintteri ESLint ja testiviitekehys Jest. Tässä opinnäytetyössä ei keskitytä siihen, miten toteutetaan todelliseen tuotantokäyttöön sopivia Node.js-sovelluksia.

JavaScript-kielen vastikkeena käytetään yhä useammin TypeScript-kieltä, joka on erinomainen vastike tavalliselle JavaScript-kielelle. Tämän opinnäytetyön sovellusta ei ole kehitetty TypeScript-kielellä, koska työn pääpaino kohdistuu jatkuvaan kehitykseen. TypeScript-kielen käyttöä kuitenkin pohditaan viimeisessä luvussa.

Lähdekoodin hallintaohjelma Git on yleinen työkalu kehittäjien keskuudessa. Git-ohjelmasta puhutaan teoriaosuudessa ja käytännön toteutuksen osuudessa, mutta sen oikeaoppista käyttöä ei käsitellä sen syvemmin.

1.3 Keskeiset käsitteet

Blue-green-julkaisu on ohjelmiston julkaisustrategia, missä ohjelmiston tuotantoversiota ja testiversiota ylläpidetään kahdessa identtisessä ympäristössä samanaikaisesti.

Canary-julkaisu on ohjelmiston julkaisustrategia, missä käyttäjiä siirretään pikkuhiljaa käyttämään ohjelmiston uutta versiota.

Dark launch -julkaisu on ohjelmiston julkaisustrategia, missä ohjelmiston käyttäjien tekemiä pyyntöjä testataan samanaikaisesti ja näkymättömästi myös ohjelmiston seuraavassa versiossa.

Hyväksyntätestaus (engl. acceptance testing) on ohjelmiston käyttöönottoa edeltävä vaihe, jossa testataan ohjelmiston toimivuutta tyypillisesti asiakasvaatimusten mukaan.

Integraatiotestaus (engl. integration testing) on automatisoitu ohjelmiston testaus, jossa testataan ohjelmiston eri komponentteja ja niiden yhteistoimintaa.

Jatkuva integraatio (engl. continuous integration) on jatkuvan kehityksen ensimmäinen vaihe, jonka tarkoituksena on validoida ohjelmiston toimivuus, kun sovellukseen tehdään uusia muutoksia.

Jatkuva kehitys (engl. continuous development) on ohjelmistokehityksessä käytettyjen automatisoitujen vaiheiden kokonaisuus, jonka tarkoituksena on tehostaa kehitysprosessia.

Jatkuva käyttöönotto (engl. continuous deployment) on jatkuvan kehityksen viimeinen vaihe, jossa ohjelmisto julkaistaan käyttöönotettavaksi halutulla tavalla.

Jatkuva testaus (engl. continuous testing) on jatkuvaan kehitykseen sisältyvät testausvaiheet, joita kehitysputki sisältää.

Jatkuva toimitus (engl. continuous delivery) on jatkuvan kehityksen vaihe, jossa ohjelmiston lähdekoodi kootaan yhdeksi kokonaisuudeksi, josta se saadaan valmiiksi käyttöönottoa varten.

Linttaus (engl. linting) on automatisoitu työkalu lähdekoodin tarkistamiseen, jonka tarkoituksena on löytää koodista mahdollisia ongelmakohtia.

Ominaisuusliputus (engl. feature flagging) on ohjelmiston kooditasolla toimiva asetus, joka määrittää mitkä ohjelmiston ominaisuudet ovat käytössä missäkin ympäristöissä.

Yksikkötestaus (engl. unit testing) on automatisoitu ohjelmiston testaus, jossa testataan ohjelmiston pienimpiä mahdollisia yksittäisiä funktioita.

Vetopyyntö (engl. pull request) on toiminto, jossa Git-haara yhdistetään toiseen haaraan. Vetopyyntöön liitetään tyypillisesti erilaisia esitarkistusvaiheita ennen kuin kyseiset haarat yhdistetään.

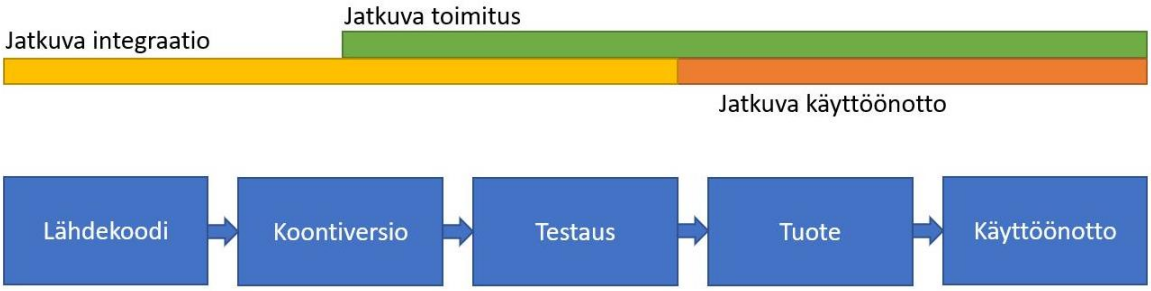
2 Jatkuvan kehitysputken vaiheet

Ohjelmistokehityksessä olisi suotavaa pystyä luomaan julkaisuja lähdekoodista nopeasti, automatisoidusti ja luotettavasti. Jatkuvan kehityksen menetelmillä (englanniksi CI/CD, eli Continuous Integration ja Continuous Delivery) pyritään edistämään näitä ominaisuuksia. Jatkuvan kehityksen tavoitteena on saada lähdekoodiin tehdyt muutokset etenemään kehitysputken eri vaiheista läpi mahdollisimman pienellä käsityöllä. Lähdekoodin koontiversion rakentaminen, testaaminen, paketointi ja julkaiseminen muodostavat ohjelmiston jatkuvan kehitysputken. (Laster 2020.)

Jatkuva integraatio, toimitus ja käyttöönotto ovat suhteellisen uusia kehityskäytäntöjä, jotka ovat nousseet suosioon viime vuosina (Rossel 2017). Jatkuvan kehityksen vaihetta, jossa ohjelmisto laaditaan yhdeksi toimivaksi kokonaisuudeksi lähdekoodista, voidaan kutsua jatkuvaksi integraatioksi. Vaihetta, joka varmistaa sovelluksen laadun, voidaan kutsua jatkuvaksi testaukseksi. Vaihetta, joka tuo lopputuotteen käyttäjien saataville, voidaan kutsua jatkuvaksi käyttöönotoksi. (Laster 2020.)

Jatkuva integraatio, jatkuva toimitus ja jatkuva käyttöönotto muodostavat kokoelman käytänteitä, joilla ohjelmiston muutokset saadaan lähdekoodista alkaen käyttöönottoon asti tehokkaasti ja luotettavasti. Jatkuvan kehitysmenetelmän käytännöt hyödyntävät tuotteen kehittäjiä ja sen käyttäjiä. (Rossel 2017.)

Rossel (2017) kuvailee, että jatkuvan integraation, toimituksen ja käyttöönoton kokonaisuutta voidaan ajatella yhtenäiseksi automaattiseksi kehitysputkistoksi tai -liukuhihnaksi. Kaikki kolme vaihetta liittyvät testaus- ja käyttöönottoprosessin automatisointiin (kuva 2), jolloin kehittäjän manuaalinen puuttuminen prosessiin on minimaalista, virheiden riski pienenee ja ohjelmistojen rakentaminen ja käyttöönotto helpottuu niin paljon, että jokainen tiimin kehittäjä pystyy tekemään sen.



Kuva 1. Jatkuvan integraation, -toimituksen ja -käyttöönoton vaiheet jatkuvassa kehitysputkessa (mukaillen Rossel 2017).

Jatkuvan integraation, toimituksen ja käyttöönoton asennus ei ole aina helppoa ja voi viedä paljon aikaa, varsinkin jos sitä integroidaan olemassa olevaan projektin. Oikein tehtynä jatkuviin

kehitysmenetelmiin käytetty työaika voi kuitenkin hyvittää itsensä takaisin pidemmällä aikavälillä, parantaen ohjelmistokehityksen laatua. (Rossel 2017.)

2.1 Jatkuva integraatio

Ensimmäinen vaihe jatkuvan kehityksen liukuhihnassa on jatkuva integraatio (Rossel 2017; Laster 2020). Jatkuvan integraation vaiheen tarkoituksena on validoida ohjelmisto, kun se on merkitty lähdehallintajärjestelmään. Sillä pyritään takaamaan ohjelmiston toimivuus edelleen uusien muutosten jälkeen. Lähteenhallintajärjestelmien avulla koodia voidaan säilyttää yhdessä paikassa. Sieltä kehittäjien on helppo tarkastella lähdekoodia ja tehdä siihen muutoksia. (Rossel 2017.)

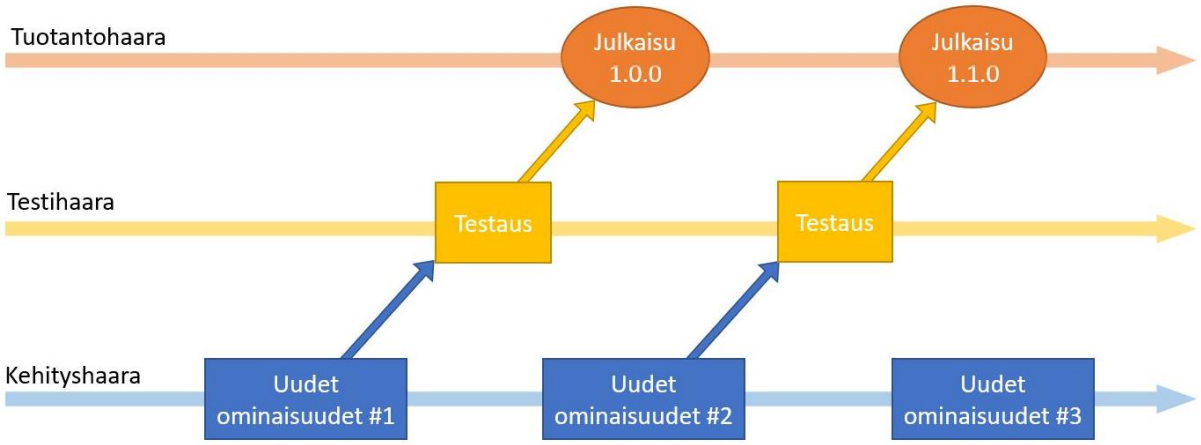
Tyypillisessä ohjelmistokehityksen mallissa kehittäjä haaroittaa ohjelmiston lähdekoodin omaan paikalliseen kehitysympäristöön (Dingare 2022). Koodin haaroittamisella tarkoitetaan Git-järjestelmässä tehtyä koodin kopiointia omaan ympäristöön, jossa ei tule välittömästi yhteentörmäyksiä projektin yhteisen ‘päähaaran’ kanssa. Kun koodimuutokset ovat valmiina, niitä halutaan työntää yhteiseen lähdekoodin tietovarastoon. Jos yhteisessä lähdekoodivarastossa oleva koodi on ristiriidassa toisen kehittäjän laatimien muutosten kanssa, ristiriidat täytyy ensiksi ratkoa. (Laster 2020.)

Jatkuvan integraation kehitysputki sisältää usein esitarkistusvaiheen, joka käynnistyy joka kerta, kun koodimuutoksia toimitetaan yhteiseen koodipohjaan. Git-hallintajärjestelmää ylläpitävät erilaiset julkiset ja kaupalliset sivustot, kuten GitHub, GitLab ja Bitbucket. Näistä sivustoista löytyy monia ominaisuuksia koodimuutosten esitarkistusvaiheeseen. Useimmat esitarkistusvaiheet sisältävät tapoja tarkastella ja kommentoida koodimuutoksia. (Laster 2020.)

Ohjelmistoprojekteissa kehittäjät voivat haaroittaa koodiston omaan kehitysympäristöönsä ja liittää muutoksensa yhteiseen jaettuun koodipohjaan tekemällä Git-järjestelmässä vetopyynnön (Dingare 2022). Vetopyynnön prosessissa kohdehaaran omistajan tulee vähintäänkin hyväksyä muutokset, mutta siihen voidaan liittää paljon enemmänkin prosesseja. Vetopyyntöihin tyypillisesti liitetään automaattisia tarkistuksia, kuten koontiversion rakentaminen ja automaattitestien ajo. Vetopyyntöön liitetyt prosessit takaavat paremmin sen, että uudet koodimuutokset toimivat halutulla tavalla, eivätkä riko olemassa olevaa koodipohjaa. (Laster 2020.)

Lähdekoodin hallintajärjestelmissä, kuten Git-ohjelmassa, koodipohjasta voi olla olemassa useita eri kehityshaaroja samanaikaisesti. Eri kehityshaaroissa voidaan työskennellä omassa eristetyssä ympäristössä, joka ei häiritse ohjelmiston muita vaiheita. Kehityksessä voisi olla laadittuna esimerkiksi omat haarat kehitykseen, testaukseen ja tuotantoon (kuva 3). Ensiksi kaikki uusi koodi liitetään kehityshaaraan, ja kun se on hyväksytty, se siirtyisi testihaaraan. Kun koodi on läpäissyt testihaaran hyväksyntävaiheen, se voidaan siirtää tuotantohaaraan. Toinen mahdollisuus kehityshaarojen järjestykseen olisi muodostaa yksi päähaara ja luoda siitä tuotantohaara jokaista uutta julkaisua

kohti. Julkaisuhaaroihin voisi vielä lisätä korjauspaketteja jälkikäteen, mutta ei uusia ominaisuuksia. (Rossel 2017.)



Kuva 2. Esimerkki Git-haarojen käytöstä. Uusia ominaisuuksia kehitetään samanaikaisesti kehityshaarassa, kun edellisten ominaisuuksien testaus ja julkaisu on käynnissä.

Kun koodimuutokset on työnnetty yhteiseen tietovarastoon, jatkuvan integraation työkalut voivat vastaanottaa siitä tiedon automaattisesti. Jatkuvan integraation tavoitteena on saada vahvistus koodimuutosten validiteetista mahdollisimman nopealla aikavälillä. Tarkistusvaiheessa voidaan kääntää lähdekoodi, ajaa yksikkötestit, tarkistaa testikattavuus, analysoida koodin kirjoituksen suuntaviivauksia ja paljon muuta. Ongelmatilanteet tulisi saada tunnistettua automaattisella tarkistuksella mahdollisimman pian, jotta ohjelmiston koodikanta ei olisi rikkinäisessä tilassa. (Rossel 2017; Laster 2020; Dingare 2022.)

Automatisoitu tarkistusvaihe voi epäonnistua, jos koodiin on jäänyt syntaktinen virhe tai jos testit eivät läpäise vaatimuksia. Epäonnistumisen määritteleminen on kehittäjien päätettävissä. Koodin syntaktiset virheet ja testien epäonnistumiset ovat selviä integraatioprosessin virhetiloja. Rossel (2017) toteaa, että sen sijaan vaikeammin määriteltäviä epäonnistumisen rajoja voivat olla koodiston testikattavuusprosentti tai teknisen velan määrä. Teknisellä velalla tarkoitetaan ohjelmistokehitysratkaisuja, jotka toimivat kriteereiden mukaisesti, mutta eivät ole pidemmällä aikavälillä kestäviä ratkaisuja. (Rossel 2017; Kruchten, Nord & Ozkaya 2019.)

Lähdekoodin lisäksi samassa tietovarastossa kannattaisi säilyttää siihen liittyvät testi-, rakennus-, ja tietokantakomentosarjat, sekä ohjelmiston kokoonpanon asetukset ja muut vastaavat tiedostot. Automaatiopalvelimen tulisi tarkistaa jokainen muutos, mitä tiedostoihin tehdään. Muutoksia olisi suotavaa tehdä pienissä osissa. Jos isokokoisia muutoksia tehdään samanaikaisesti, niistä on vaikeampi tunnistaa mahdollisia uusia virhetilanteita (Rossel 2017). Kun vetopyynnön muutokset on

hyväksytty ja uusi koodi on liitetty yhteiseen koodipohjaan, viimeistään silloin jatkuvan toimitusprosessin vaiheet lähtevät käyntiin (Laster 2020).

2.2 Jatkuva toimitus

Jatkuvan toimituksen vaihe tekee ohjelmistosta käyttöönottovalmiin tuotteen (Rossel 2017). Sillä tarkoitetaan vaihetta, jossa koodimuutoksista luodaan niihin liittyvät koonti-, testaus- ja pakkaustoiminnot, joiden kautta tuotetaan uusi käyttöön otettava versio ohjelmistosta. Jatkuvan toimituksen prosessi jatkuu integraatiovaiheen loputtua ja suorittaa kehitysliukuhihnan vaiheet lopputuotteeseen asti. (Laster 2020.)

Jatkuvan toimitus koostuu peräkkäisesti toteutettavista vaiheista. Vuorostaan jokainen toimituksen vaihe voi koostua useammasta alitehtävästä. Toimitusvaiheeseen liittyy yleensä lähdekoodin kokoaminen, yksikkötestien suorittaminen, integraatiotestien ajo, suoritusmetriikan kerääminen ja artefaktien julkaisu. (Laster 2020.)

Jatkuvan integraatioprosessin lopussa ohjelmistosta luodaan artefakti (Rossel 2017). Artefaktit ovat kehitysputken aikana muodostettuja tiedostoja, jotka liittyvät tai sisältyvät lopputuotteeseen ja käyttöönottoon, kuten pakatut ZIP-tiedostot tai EXE-ajotiedosto. Näiden vaiheiden jälkeen voi tapahtua siirto hyväksyntävaiheeseen, missä artefaktit kootaan yhteen ja niiden toiminnallinen testaus suoritetaan. (Laster 2020; Dingare 2022.)

Artefaktin tyypistä riippumatta se kannattaisi versioida jollakin järjestelmällä, kuten semanttisella versioinnilla: ”major.minor.patch” (Laster 2020). Major-, minor- ja patch-julkaisut auttavat erottamaan eritasoiset julkaisut toisistaan (Gough, Auburn & Bryant 2022). Artefaktin versioinnin avulla pitäisi pystyä selvittämään niiden tuottamiseen käytetty lähdekoodi, jotta ilmeentyviä ongelmia pystytään jäljittämään (Laster 2020).

Versioituja artefakteja voidaan hallita tietovarastotyökalulla. Tietovarastot voivat toimia erillään kehitys-, testaus- ja tuotetasoissa järjestettyinä tietueina. Silloin samanaikaisesti käynnissä olevien julkaisuprosessien tasot voidaan erotella. (Laster 2020.)

Automatisoitujen prosessien takia ohjelmiston päivittäminen uusimpaan versioon ei ole aina toivottavaa. Jotkut jatkuvan kehityksen vaiheet, kuten testaus ja virheiden korjaus, voivat vaatia artefakteista vakaita versioita, jotka eivät muutu joka kerta, kun automaattista kehitysputkea suoritetaan. (Laster 2020.)

Kun ohjelmiston koontiversio läpäisee kaikki tiimin hyväksyntäkriteerit, artefakti on valmis toimitettavaksi käyttäjille. Jatkuvan kehityksen tuottamat artefaktit otetaan lopuksi käyttöön

tuotantopalvelimella jatkuvan käyttöönoton yhteydessä (Rossel 2017). Jatkuva toimitus voi käynnistää automaattisesti jatkuvan käyttöönoton vaiheen (Laster 2020).

2.3 Jatkuva testaus

Kaikkia jatkuvan kehitysprosessin tuotoksia ei haluta aina julkaista välittömästi. Ensiksi olisi tarpeellista todentaa, että julkaistavat muutokset toimivat oikein käyttämällä esimerkiksi jatkuvan testauksen menetelmiä. Testitulokset määrittävät jatkuvan toimituksen etenemisen. Jos liukuhihnaprosessissa käytetään väärää versiota artefaktista tai jokin toiminnallisuus ei enää toimi oikein, automaatiotestien pitää havaita nämä ongelmat ja nostattaa siitä varoitus. Tätä jatkuvan kehityksen prosessia voidaan kutsua jatkuvaksi testaamiseksi. (Laster 2020.)

Jatkuva testaus tarkoittaa automatisoitujen testien ja muiden analyysien suorittamista, joiden laajuus voi kasvaa jatkuvasti koodin kulkiessa kehitysputken läpi. Erityyppisiä testausmuotoja ovat muun muassa:

- Yksikkötestaus, joka keskittyy yksittäisten koodialueiden testaamiseen eristettynä muusta sen kanssa vuorovaikutuksessa olevasta koodipohjasta.
- Integraatiotestaus, joka varmistaa, että ohjelmiston komponentit ja niiden toiminnot toimivat yhdessä.
- Hyväksyntätestaus, joka mittaa ennalta määritettyjen kriteerien perusteella järjestelmän ominaisuuksia, kuten suorituskykyä, skaalautuvuutta, kapasiteettia ja toimintaa kuormituksen alla.

Testausmuotoja on monenlaisia, kuten tietoturva-, käytettävyys- ja suorituskykytestaus (Scott, MacDonald & Powers 2021). Jatkuvan testauksen tavoitteena on taata peräkkäisillä testaus- ja analyysitasoilla, että ohjelmiston koodi on riittävän laadukas käytettäväksi seuraavaan julkaisuun (Laster 2020).

Fernandes da Costa (2021) tuo esille hyvän huomion, että hyvin kirjoitetut testit voivat olla parasta dokumentaatiota, mitä kehittäjä voi saada koodista. Koska testien on läpäistävä onnistuneesti, niiden täytyy pysyä ajan tasalla. Testit kuvailevat eri rajapintojen käyttöä ja auttavat sitä kautta kehittäjiä ymmärtämään, miten koodikanta toimii.

Testeihin kirjoitetaan odotukset ohjelmiston toimivuudesta. Mitä enemmän testejä on ja mitä enemmän ne käyttäytyvät kuin todellisten käyttäjien tekemät toiminnot, sitä parempia takeita ne antavat kehittäjälle ohjelmiston toimivuudesta. Huomioitavaa on, että testien tarkoituksena on raportoida epäonnistumisista, kun ohjelmisto toimii väärin. Testit, jotka eivät voi koskaan epäonnistua, ovat käyttökelvottomia. (Fernandes da Costa 2021.)

Testit eivät voi todistaa toimiiko ohjelma oikein. Ne voivat todistaa vain tapauksia, missä ohjelma ei toimi ennalta määritetyillä virhetiloilla. Testit eivät voi koskaan todistaa, että ohjelmistossa ei ole ainuttakaan virhettä. Sovelluksiin voidaan antaa lähes loputon määrä eri syötteitä, eikä niitä kaikkia voi testata. Testit kattavat yleensä aikaisemmin havaitut virheet tai tietyt tilat, joiden toimivuus halutaan aina varmistaa. (Fernandes da Costa 2021.)

Koko ohjelman testaaminen manuaalisesti on aikaa vievä ja virhealtis prosessi. Automaattiset testit tehostavat testaamisen prosessia. Testien avulla saadaan koodimuutoksia tehdessä nopeasti vastaus takaisin koodin toimivuudesta ja ne nopeuttavat siten virheiden korjaamista. Mitä pienempi viive saadaan koodin kirjoittamisen ja sen palautteen välille, sitä vakaampaa kehityksestä tulee. (Fernandes da Costa 2021.)

Kun kehittäjä lisää automaatiotestejä kehittämiinsä ominaisuuksiin, kaikki tiimin jäsenet hyötyvät siitä. Testien kirjoittaminen voi olla aluksi aikaa vievää, mutta pitkällä aikavälillä automaattitestien hyödyt kasvavat manuaaliseen testaamiseen verrattuna. Automaatiotesti voi suorittaa testaamisen hyvin nopeasti tai melkein välittömästi. Näin ollen mitä pidempi elinkaari ohjelmistolla on, sitä paremman hyödyn automaattitestaamisesta saa manuaaliseen testaamiseen verrattuna. (Fernandes da Costa 2021.)

Automaattiset testit auttavat kehittäjiä tuottamaan laadukkaita ohjelmistoja. Kun testit ovat automatisoituja ja hyvin määriteltyjä, ohjelmistokehittämisen kustannukset vähenevät. Testien avulla kehittäjät pystyvät varmistamaan muiden tekemän työn laatua, ja sen etteivät ne riko sovelluksen muita osia. (Fernandes da Costa 2021.)

Automaattiset testit eivät kuitenkaan poista manuaalisen testaamisen tarvetta kokonaan. Toiminnan todentaminen manuaalisella testaamisella, kuten ohjelman käyttäminen asiakkaan näkökulmasta, on edelleen tarpeellista. (Fernandes da Costa 2021.)

2.3.1 Yksikkötestaus

Kaikista yksinkertaisin testausmuoto on yksikkötestaus (Scott, MacDonald & Powers 2021). Yksikkötestien kirjoittamisella varmistetaan, että ohjelmiston tietyt osat tulostavat oikeita ja haluttuja arvoja. Yksikkötestit ovat koodia, joka kutsuu koodiston muita funktioita ennalta määritetyllä syötteellä ja tarkistaa, onko tulos se, mitä haluttiin. Yksikkötestit ilmoittavat, tulostivatko testattavat koodit haluttuja arvoja vai ei. Ne testaavat nimensä mukaisesti pieniä ja eristettyjä koodilohkoja kerrallaan. (Rossel 2017; Laster 2020; Scott, MacDonald & Powers 2021.)

Ohjelmistokehittäjät pystyvät itse määrittelemään yksikkötestejä projektiinsa. Kehittäjät tietävät itse parhaiten, miten eri toiminnot käyttäytyvät ja voivat sitä kautta määrittelemään niille sopivia

testitapauksia. Kehityslähestymistapaa, jossa suunnitellaan ensiksi yksikkötestit valmiiksi ja sitten toteutetaan muutokset testejä noudattaen, kutsutaan testivetoiseksi kehitykseksi. (Laster 2020.)

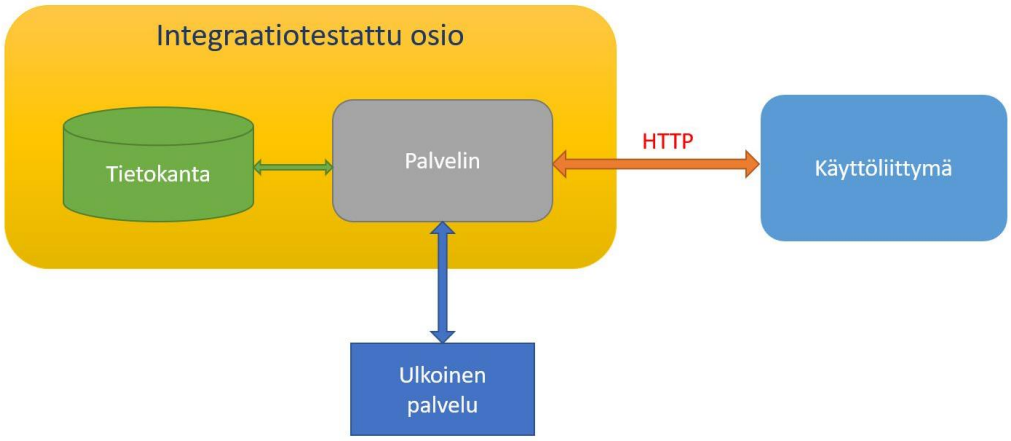
Testitapausten syötteiden ja tulosten on oltava ennalta määritettyjä ja arvattavia. Koodimuutoksia tapahtuu jatkuvasti kehityksen alkuvaiheissa ja hyvin usein, joten yksikkötestien suoritusaikojen pitäisi olla lyhytkestoisia. Yksikkötestit eivät saisi olla riippuvaisia ulkoisista tietolähteistä tai moduuleista, kuten tietokantakomponenteista. Jos koodista löytyy ulkoisia riippuvuuksia, yksikkötestauksessa nämä riippuvuudet tyypillisesti tekaistaan ’mock’-toiminnolla. Mock-funktio näyttäytyy tietokoneohjelmalle edelleen samanlaisena resurssina tai toimintona, mutta ei oikeasti suorita sisällään alkuperäistä funktiotaan. (Rossel 2017; Laster 2020.)

Jos yksikkötestit ilmoittavat ‘onnistumisesta’, se ei automaattisesti tarkoita, että koko ohjelmisto toimii halutulla tavalla. Yksittäistä funktiota vasten tarvitaan yleensä enemmän kuin yksi testi sen testaamiseen. Testien kuuluisi kattaa suuren osuuden koko koodia ja käsitellä vähintäänkin kaikista yleisimmät käyttötapaukset. Testien suhteen niiden laatu ja määrä ovat molemmat tärkeitä asioita. Testikattavuuden tarkistamiseen löytyy työkaluja, mutta työkalut eivät osaa analysoida testaavatko yksikkötestit oikeasti hyödyllisiä tai järkeviä asioita. (Rossel 2017.)

2.3.2 Integraatiotestaus

Yksikkötestien sijaan integraatiotesteillä saadaan parempaa osviittaa siitä, toimiiko ohjelmisto kokonaisuudessaan odotetulla tavalla. Integraatiotestien kautta voidaan testata, tallentuvatko tietokantamuutokset, saako sovellus vastaanotettua tietoja ulkopuolisesta palvelusta, kirjoitetaanko lokitietoja oikeasti tiedostoon tai kulkeeko data oikein käyttöliittymältä asti palvelimelle (kuva 4). Integraatiotestien kattavuus on laajempi kuin yksikkötesteissä. (Rossel 2017; Fernandes da Costa 2021.)

Rajapintasovelluksessa integraatiotesteillä saadaan testattua, kuinka rajapintapalvelut toimivat tai miten tietokannan tila muuttuu. Vaikka integraatiotestaamisella saadaan testattua tietokannan toimintaa, tuotannossa käytettävää tietokantaa ei saisi sekoittaa testidataan. Testaamista varten tulisi käyttää siihen eristettyä testitietokantaa. (Fernandes da Costa 2021.)



Kuva 3. Esimerkki integraatiotestien laajuudesta verkkosovelluksessa (mukaillen Fernandes da Costa 2021).

Integraatiotestejä olisi tarpeellista kirjoittaa aina niissä tilanteissa, joissa eri ohjelmiston eri osien tai ulkoisten rajapintapalveluiden yhteistoiminta on kriittistä. On huomioitava, että integraatiotestien ei ole tarkoitus testata ulkoisia komponentteja, vaan sitä, että ohjelmiston omat toiminnot toimivat ulkoisten komponenttien kanssa oikein. (Fernandes da Costa 2021.)

2.3.3 Hyväksyntätestaus

Kaikkia testaustoimintoja ei aina pystytä automatisoimaan (Laster 2020). Hyväksymistestaus on käytäntö, millä varmistetaan, toimiiko sovellus liiketoiminnallisesta näkökulmasta, ja sopiiko ohjelmisto suunnatuille loppukäyttäjille. (Fernandes da Costa 2021.)

Hyväksyntätestauksessa on tärkeä tarkistaa ohjelman oikean toimivuuden lisäksi myös, että tyydyttävätkö toiminnot pääkäyttäjien vaatimukset ja osaavatko käyttäjät käyttää ohjelmaa. Vaikka hyväksyntätestejä pystytään kirjoittamaan ainakin osittain yksikkö- tai integraatiotestitasolla, tämän laatuista testausta on lähes mahdotonta automatisoida tietokoneohjelmalla. (Fernandes da Costa 2021.)

2.3.4 Koodin analyysi

Koodausmetriikka ja -analyysi eivät validoi ohjelmistoa samalla kaksivaihtoehtoisella tavalla kuin hyväksytty/hylätty -tyyppinen testaus. Niitä voisi kuitenkin sisällyttää testausprosessiin, sillä ne arvioivat koodia ja sitä testaavan koodin laatua. Niitä voi myös käyttää kehitysliukuhihnan eri kohdissa estämään tai sallimaan prosessien etenemistä. Koodausmetriikkaan ja -analyysiin voidaan sisällyttää esimerkiksi:

- Testitapausten kattaman koodin määrän arviointi, jota voidaan kutsua koodikattavuudeksi.

- Koodirivien määrän, monimutkaisuuden, rakenteen tai tyylin analysointia. Tähän tarkoitetut työkalut mittaavat tuloksia etukäteen määriteltyihin kynnysarvoihin ja tuottavat raportin tuloksista.

Testikoodin kattavuuden määrittäminen vaatii monitahoista lähestymistapaa eri näkökulmista. Siihen tulisi sisältyä koodin esittelyä ja tarkastelua tiimin muiden jäsenien kanssa. Testaustyökalut sisältävät myös automaattisia koodintarkasteluominaisuuksia, jotka auttavat arvioimaan, kuinka hyvin testit toimivat koodipohjassa. (Scott, MacDonald & Powers 2021.)

2.4 Jatkuva käyttöönotto

Jatkuvan kehityksen viimeinen vaihe on jatkuva käyttöönotto. Jatkuva käyttöönotto automatisoi ohjelmiston käyttöönottoprosessin ja julkaisee tuotteesta uuden version asiakkaille käytettäväksi (Rossel 2017; Laster 2020). Asennustavasta riippuen, uuden julkaisun käyttöönotto voi esimerkiksi tarkoittaa asennusta pilveen, ohjelmisto- tai verkkopäivitystä tai saatavilla olevien julkaisujen listausta (Laster 2020).

Ohjelmiston pitäisi olla saatavilla käyttöönotettavaksi tuotantoympäristöön milloin tahansa. Se tarkoittaa, että koontiversion rakennus toimii, koodi kääntyy ja testit onnistuvat. Tätä kautta lisätyt uudet ominaisuudet ja virhekorjaukset voivat olla tuotantoympäristössä käytössä erittäin nopeasti. (Rossel 2017.)

Käyttöönoton jälkeinen julkaisun peruutus voi käydä kalliiksi kehittäjille sekä käyttäjille, joten uuden julkaisun valmius tulisi arvioida etukäteen. Arviointiin voidaan sisällyttää menetelmiä, joilla uuden julkaisun toimintaa pystytään ensiksi kokeilemaan ennen sen todellista käyttöönottoa (Laster 2020). Olisi tärkeää suunnitella julkaisustrategia, jossa riskit olisivat pienet tuotantoympäristössä. Julkaisustrategiat vaihtelevat ohjelmiston infrastruktuurin ja käytössä olevien ulkoisten palveluiden mukaan (Gough, Auburn & Bryant 2022).

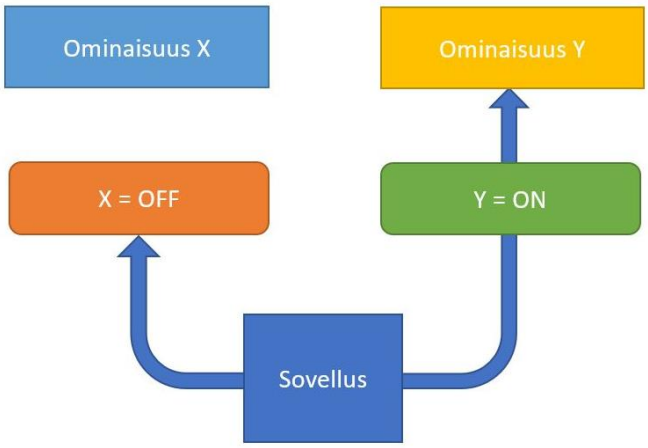
Rajapintasovelluspäivityksessä suuret (major) muutokset vaikuttavat eniten sen käyttäjiin. Muutoksien tapahtuessa käyttäjältä edellytetään joitain vastatoimia, jotta rajapinnan käyttö toimisi edelleen. Rajapintasovelluksesta tulisi olla voimassa samanaikaisesti uusin versio sekä vanhempi versio riittävän pitkäksi ajaksi. Silloin käyttäjillä olisi riittävästi aikaa tehdä päivityksiä ja migraatioita uuden rajanpintaversion käyttöön, eikä vanhemmalla rajapintaversiolla laadittu toiminto yllättäen lakkaa toimimasta. Yksi metodi rajapintasovellusten versiointiin on versioida URL-osoitteet otsikolla, joka merkitsee rajapinnan major-version: ‘GET /attendees Version: v1’. (Gough, Auburn & Bryant 2022.)

Pienet (minor) ohjelmiston muutokset eivät aseta samoja rajoitteita kuin suuret muutokset. Pienet muutokset eivät vaadi käyttäjien puolelta lisätoimia, joten niitä voidaan julkaista helpommin ja useammin. (Gough, Auburn & Bryant 2022.)

Jos rajapintasovelluksen käyttäjä ja kehittäjä toimivat tiiviissä yhteistyössä, ohjelmiston versioinnin ja elinkaaren suunnittelu ei ole yhtä merkittävää. Tämä voi olla tilanne, jos rajapintaa kehitetään ja käytetään saman tiimin sisällä. Silloin tärkeintä olisi suunnitella julkaisustrategia, jossa molemmat osapuolet toimivat yhdessä niin julkaisussa kuin käyttöönotossa. (Gough, Auburn & Bryant 2022.)

2.4.1 Ominaisuusliputus

Ominaisuusliputus (engl. feature flag) on kooditasolla yksi tapa määrittää ja ottaa käyttöön uusia ominaisuuksia voimassa olevassa sovelluksessa (kuva 5). Ominaisuusliputus asetetaan tyypillisesti konfiguraatioasetuksissa käynnissä olevan sovelluksen ulkopuolella. Ne mahdollistavat sovelluksen uuden julkaisun käyttöönoton uusien ominaisuuksien ollessa ensiksi pois käytöstä. Kun tiimi tai tuotteen omistaja asettaa ominaisuuden käytettäväksi, sovellus alkaa suorittamaan eri koodihaaraa. Käyttöönoton laajuus voi olla käyttäjäkohtaisesti asteittaista, tai ominaisuus voidaan asettaa voimaan globaalisti saman tien. Tällä tavalla käyttäjiä voidaan siirtää osajoukko kerrallaan käyttämään uutta järjestelmää ja ohjelman uudet ominaisuudet saadaan testattua. (Gough, Auburn & Bryant 2022.)



Kuva 4. Ominaisuusliputuksessa poissa päältä olevat toiminnot eivät ole sovelluksen käytössä.

Ominaisuuslippujen avulla virheiden ilmaantuessa uuden julkaisun ominaisuuksia voidaan kääntää vielä takaisin alkuperäiseen tilaan kohtuullisen helposti. Ominaisuuslipun ollessa pois käytöstä siihen liittyvä koodi edelleen sisältyy tuotteeseen, mutta ei ole enää aktiivisessa käytössä (Laster 2020). Jos ongelmia ei ilmene, käyttäjien siirtoa uuteen versioon voidaan lisätä asteittain, kunnes kaikki käyttäjät ovat uusimmassa versiossa. Jos käyttäjien siirtoa uuteen julkaisuun ei tehdä

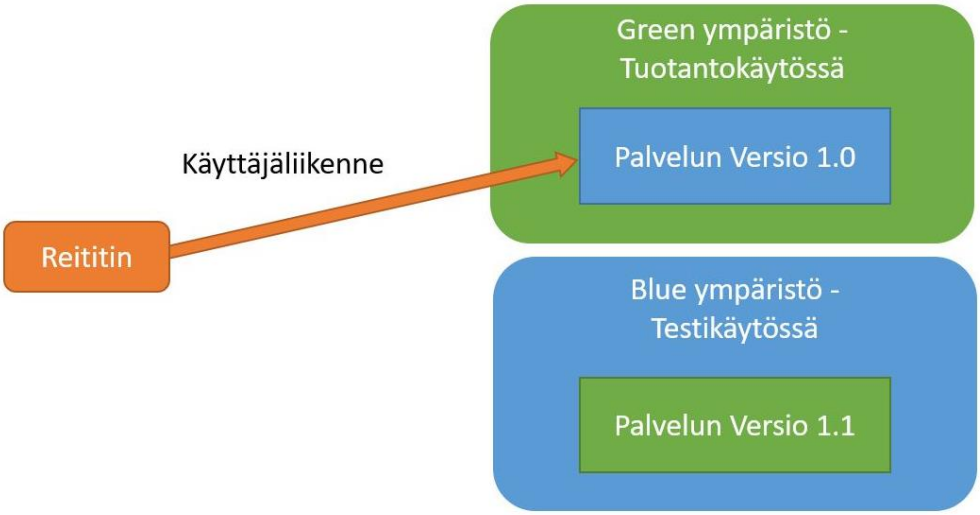
asteittain tai jollain muulla tavalla muuten hallitusti, virhetilanteiden ilmentyessä seuraukset voivat olla vakavia (Gough, Auburn & Bryant 2022).

2.4.2 Blue-green-julkaisu

Käyttäjäliikenteen hallinta mahdollistaa sellaisten julkaisumallien laatimisen, joissa liikennettä voidaan reitittää hallitusti (Gough, Auburn & Bryant 2022). Blue-green-julkaisussa ohjelmistojen käyttöönottoa varten ylläpidetään kahta identtistä tuotantoympäristöä. Ne voivat olla värikoodattuna esimerkiksi ‘sininen’ ja ‘vihreä’. Väreillä ei ole suurta merkitystä, sillä ne toimivat vain tunnisteina. Blue-green-julkaisussa toinen ympäristöistä on aina tuotantokäytössä ja toista käytetään testausympäristönä, jossa sitä seuraavaa julkaisua valmistellaan (kuva 6). Ympäristöjen edessä kuuluu olla reititin, joka toimii käyttäjien porttikäytävänä sovellukseen. Käyttäjäliikenteen olisi tarkoitus ohjautua aina sillä hetkellä toiminnassa olevaan tuotantoversioon. Tällä tavalla käyttäjäliikenteen vaihtaminen kahden tuotantoympäristön välillä olisi nopeaa, helppoa ja käyttäjälle näkymätöntä toimintaa. (Laster 2020.)

Kun ohjelmiston uusi julkaisu on valmis viimeiseen hyväksyntätestausvaiheeseen, se voidaan ottaa käyttöön testausympäristössä. Kun se on testattu ja hyväksytty, asiakasliikennettä ohjaava reititin voidaan vaihtaa ohjaamaan saapuva liikenne tähän testausympäristöön, jolloin siitä tulee uusi tuotantoympäristö. Sitten taas edellinen tuotantoympäristö on seuraavaksi käytettävissä testiympäristönä. Jos uudesta tuotantoympäristöstä löytyvästä julkaisusta löytyy ongelmia ja edellinen tuotantoinstanssi on edelleen toisessa ympäristössä toiminnassa, asiakasliikenteen voi vielä ohjata takaisin edelliseen versioon. Tällä tavalla ongelmat saadaan nopeasti poistettua käyttäjien näkyvistä ja niitä pystytään korjaamaan samanaikaisesti toisessa ympäristössä. (Laster 2020)

Blue-green-julkaisu toimii hyvin sen yksinkertaisuutensa vuoksi. Tosin sen käyttö vaatii kaksinkertaisen määrän resursseja toimiakseen rinnakkain tuotanto- ja testiympäristön kanssa. (Gough, Auburn & Bryant 2022.)

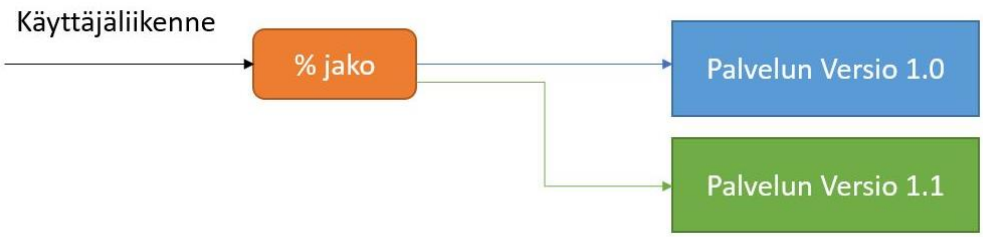


Kuva 5. Palvelun julkaisu blue-green-strategialla (mukaillen Gough, Auburn & Bryant 2022).

2.4.3 Canary-julkaisu

Canary-julkaisun yhteydessä aluksi vain pieni osajoukko käyttäjistä reititetään uuteen tuotantoversioon ja loput pidetään vanhassa (kuva 7) (Laster 2020, Gough, Auburn & Bryant 2022). Jos uusi tuotantoversio toimii ongelmitta nykyisellä käyttäjämäärällä, ohjatun käyttäjäliikenteen määrää voidaan kasvattaa vähitellen, kunnes kaikki käyttäjät käyttävät uutta julkaisua. Sen jälkeen edellinen tuotantojulkaisu voidaan poistaa käytöstä ja vaihtaa testiympäristöksi. (Laster 2020.)

Canary-julkaisu voi olla joissain tapauksissa erinomainen vaihtoehto, koska käyttäjäjoukon reititys on tarkasti hallittua. Järjestelmässä kuuluu olla hyvä monitorointi, jotta ongelmat voidaan tunnistaa nopeasti ja tarvittaessa pystytään peruuttamaan vanhaan versioon. Canary-julkaisussa on se hyvä puoli, että kokonaista toista järjestelmäinstanssia ei tarvitse ylläpitää samanaikaisesti, niin kuin blue-green-julkaisussa. Sen sijan canary-julkaisussa riittää, että vain uusista komponenteista pidetään yllä kahta versiota samanaikaisesti. Tämä voi säästää kustannuksissa ja yksinkertaistaa ympäristöjen hallintaa. (Gough, Auburn & Bryant 2022.)

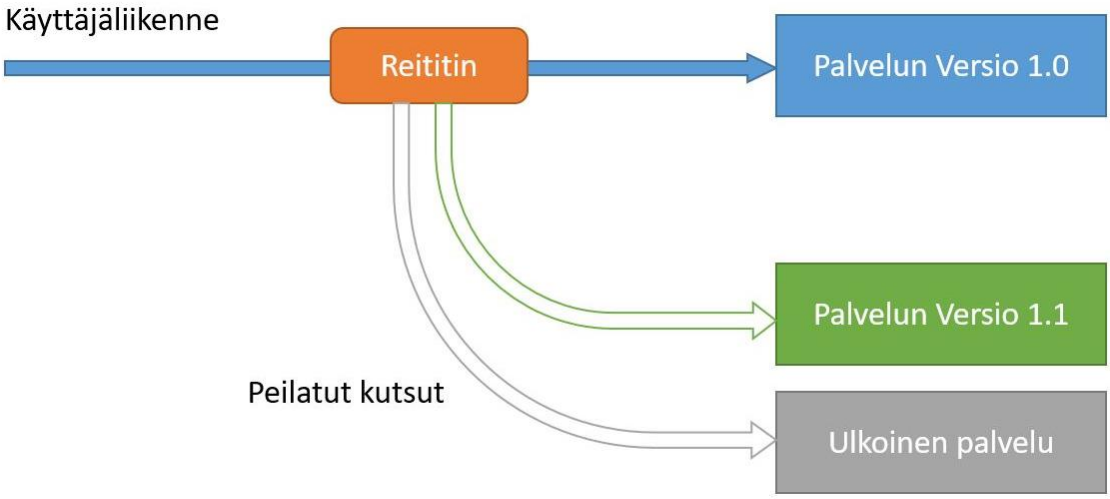


Kuva 6. Palvelun julkaisu canary-strategialla. Osajoukko käyttäjistä reititetään uuteen versioon (mukaillen Gough, Auburn & Bryant 2022).

2.4.4 Dark launch -julkaisu

Käyttäjäliikennettä ja -pyyntöjä voidaan myös peilata, eli luoda niistä kopio ja lähettää toiseen osoitteeseen, kuten uuteen rajapintaversioon (kuva 8). Käyttäjien peilatuista pyynnöistä ei palauteta suoraa vastausta takaisin. Sen sijaan vastaukset käsitellään ja arvioidaan kaistan ulkopuolella, esimerkiksi vertaamalla sitä alkuperäisen pyyntökohteessa olevan palvelun tuottamiin tuloksiin. (Gough, Auburn & Bryant 2022.)

Verkkoliikenteen peilaaminen muihin palveluihin mahdollistaa ohjelmistojen pimeän julkaisemisen (dark launch). Silloin käyttäjä ei ole itse tietoinen uudesta julkaisusta, mutta käyttäjän tekemiä pyyntöjä voidaan tarkkailla sisäisessä kehityksessä. Pimeän julkaisun suurin ero canary-julkaisuun on se, että käyttöönoton kokeiluvaiheessa käyttäjäliikennettä peilataan samanaikaisesti myös palvelun toiseen versioon. (Gough, Auburn & Bryant 2022.)



Kuva 7. Palvelun julkaisu dark-launch-strategialla (mukaillen Gough, Auburn & Bryant 2022).

Käyttäjäliikenteen peilaaminen ulkoiseen palveluun on yleistynyt viime vuosina. Dark-launch-käyttöönotolla voidaan arvioida julkaisun toiminnallista suorituskykyä, ilman liiketoiminnallisia vaikutuksia. (Gough, Auburn & Bryant 2022.)

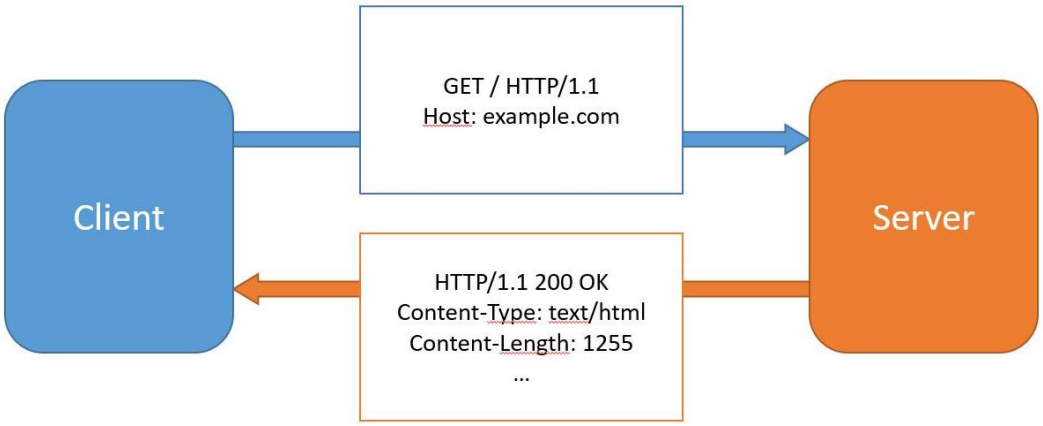
Dark-launch-julkaisussa koodia testataan tai otetaan käyttöön tuotantoon asteittain. Tästä voidaan kerätä seurantatietoja analysointia varten, paljastamatta käyttäjille tietoja tapahtumasta tai sen tuloksista. Tarkoituksena on saada tietoa siitä, kuinka muutos toimisi tuotantokuormituksessa vaikuttamatta nykyiseen käyttäjäkokemukseen. Ajan myötä kuormitusta voidaan lisätä, kunnes toiminnon katsotaan olevan valmis käytettäväksi. Ominaisuusliputusta voidaan käyttää pimeän julkaisun mekanismin toteuttamiseen. (Laster 2020.)

3 Rajapintasovelluksessa käytetyt teknologiat

3.1 Http-protokolla

HTTP (Hypertext Transfer Protocol) on yksi Internetissä laajimmin käytetyistä sovellusprotokollista, jolla World Wide Web (WWW) -palvelut toimivat. HTTP on tekstipohjainen asiakas-palvelinprotokolla, joka toimii TCP:n päällä. TCP (Transmission Control Protocol) on verkkoprotokolla, jossa dataa lähetetään kahden päätteen välillä luotettavasti, ja siihen on liitettynä tarkistusvaiheita datan eheyden varmistamiseksi (MDN Web Docs 2023c). Tavallinen HTTP toimii TCP-portin 80 kautta. Yksinkertaisin HTTP-protokolla on tänä päivänä vanhentunut turvallisuussyistä. Nykyään sivustojen tulisi käyttää suojattua versiota, eli HTTPS (Hypertext Transfer Protocol Secure). (Van Winkle 2019.)

HTTP-protokollassa verkkoasiakas, kuten verkkoselain, lähettää HTTP-pyynnön verkkopalvelimelle, minkä seurauksena verkkopalvelin palauttaa HTTP-vastauksen takaisin (kuva 9). Yleensä HTTP-pyynnössä osoitetaan, mistä resurssista pyytävä osapuoli on kiinnostunut ja HTTP-vastaus toimittaa pyydetyn resurssin. GET-pyyntöä käytetään, kun verkkoasiakas pyytää verkkopalvelimelta asiakirjaa, kuvaa, verkkosivua tai muuta tiedostoa. GET-pyynnöt ovat kaikista yleisimpiä HTTP-tyyppejä. (Van Winkle 2019.)



Kuva 8. HTTP GET-pyyntö asiakkaan (Client) ja palvelimen (Server) välillä (mukaillen Van Winkle 2019).

GET-pyyntöjen lisäksi HEAD ja POST ovat yleisimpiä pyyntötyyppejä. HEAD toimii samalla tavalla kuin GET, paitsi että HEAD-pyynnöllä kysellään vain teknistä tietoa resurssista. Jos verkkoasiakas haluaa vain tiedustella verkkosivulla ylläpidetyn tiedoston kokoa, se voi lähettää sitä kohden HEAD-pyynnön. POST-tyyppiä käytetään silloin, kun verkkoasiakas haluaa lähettää tietoja palvelimelle, kuten verkkolomakkeen tallentaminen. POST-pyyntöön tyypillisesti liitetään

verkkopalvelimen tilan muutos. Verkkopalvelin voi reagoida POST-pyyntöön esimerkiksi lähettämällä sähköpostin tai muuttamalla tietokannan tietoja. (Van Winkle 2019; MDN Web Docs 2023b.)

GET-, HEAD- ja POST-pyyntöjen lisäksi on olemassa muitakin HTTP-tyyppejä, jotka eivät ole yhtä paljon käytössä:

- PUT-tyyppiä käytetään asiakirjojen lähettämiseen web-palvelimelle. POST-pyyntöä käytetään myös verkkopalvelimen tilan muuttamiseen, joten PUT ei ole yhtä yleisesti käytössä.
- DELETE-tyypillä pyydetään poistamaan dokumenttia tai muuta resurssia. POST-pyyntöä käytetään myös verkkopalvelimen tilan muuttamiseen, joten DELETE ei ole yhtä yleisesti käytössä.
- CONNECT-tyyppiä käytetään HTTP-yhteyden muodostamiseen välityspalvelimen kautta.
- TRACE-tyypillä kysellään diagnostisia tietoja verkon välityspalvelimelta. Verkkopyynnöt eivät aina kulje välityspalvelimen kautta, eivätkä kaikki verkkovälityspalvelimet tue TRACE-tyyppiä.
- OPTIONS-tyypillä voidaan kysyä, mitä HTTP-pyyntötyyppejä palvelin tukee tietylle resurssille. Verkkopalvelin voi vastata takaisin tekstillä, kuten ‘Allow: OPTIONS, GET, HEAD, POST’. Kaikki verkkopalvelimet eivät tue OPTIONS-toimintoa.

Jos verkkopalvelimelle lähetetään pyyntötyyppi, mitä se ei tue, palvelimen pitäisi palauttaa vastaus takaisin ‘400 Bad Request’ -koodilla. (Van Winkle 2019; MDN Web Docs 2023b.)

3.2 Node.js

Node.js on alusta verkkosovellusten, verkkopalvelimien ja yleiskäyttöisten ohjelmien kehittämiseen. Se on suunniteltu skaalautuvaksi ratkaisuksi verkkosovelluskehitykseen hyödyntäen palvelimessa toimivaa JavaScript-koodikieltä ja asynkronista syöte/tuloste -tietoliikennettä datan siirrossa. Node.js:n suosio on kasvanut jatkuvasti viime vuosien aikana ja se on käytössä myös isoissa yrityksissä ja projekteissa. (Herron 2020.)

Node.js-arkkitehtuuri poikkeaa muista tyypillisistä sovellusalustoista. Tyypillisesti sovelluksen skaalautumisessa käytetään tietokoneen suorittimen monisäikeisyyttä, tosin Node.js karttaa monisäikeisyyttä pysyäkseen yksinkertaisempana ratkaisuna. Node.js:n mukaan, yksisäikeisissä tapahtumajohtoisissa ohjelmistorakenteissa muistin käyttö ja latenssiprofiili on kuormituksessa pienempi, suoritusteho on nopeampaa ja ohjelmointimalli on yksinkertaisempaa. (Herron 2020; Node.js 2023.)

Node.js on pohjimmiltaan yksittäinen moottori JavaScript-ohjelmien suorittamiseen. Se soveltuu myös yleiskäyttöiseen ohjelmointiin, vaikka onkin keskittynyt sovelluspalvelinkehitykseen. Node.js:llä kehitetään verkkosovelluskehyksiä ja sovelluspalvelimia, mutta se ei ole itsessään

sovelluspalvelinalusta, vaan Python- tai Go-kielen tapainen ohjelmien suoritusalusta. Node.js perustuu verkkoselain Chromen V8 JavaScript-moottoriin. Tästä syystä Chrome-selaimeen toteutetut parannukset sen suorituskyyn ja ominaisuuksiin hyödyntävät myös Node.js-alustaa. (Herron 2020.)

Jotta Node.js-sovellukset voivat ylläpitää tehokasta suoritustasoa, saapuvat tapahtumat pitää käsitellä nopeasti. Jos Node.js:n ainoa suoritussäie joutuu suuren laskukuormituksen alle, se ei voi enää käsitellä tapahtumia yhtä hyvin, ja silloin suoritusteho kärsii. Tämä ei suoranaisesti tarkoita sitä, että Node.js -arkkitehtuuri on virheellinen. Sen sijaan kehittäjän täytyy tunnistaa ohjelmiston osat, joissa laskentakuormitus on suurimmillaan. Kuormitettuihin kohtiin tulee laatia ratkaisuja, esimerkiksi hyödyntämällä tapahtumasilmukkaa paremmin, kehittämällä suorittavaa algoritmia tai yhdistämällä toteutukseen koodikirjaston käyttöä. (Herron 2020; Node.js 2023.)

Node.js:n sisäänrakennetut moduulit mahdollistavat JavaScript-kielen suorittamisen lisäksi muun muassa:

- Komentorivityökalut
- Binääritietojen käsittelyä puskuriobjekteilla
- Takaisinkutsufunktioiset ja tapahtumapohjaiset TCP- ja UDP- protokollien suoritinkannat
- DNS-hakutoiminnot
- HTTP-, HTTPS- ja HTTP/2-solvelluspalvelimet
- Sisäänrakennetun tuen yksikkötestaukseen.

Node.js:n verkkokerros on matalatasoinen ja helppokäyttöinen ohjelmistokokonaisuus. HTTP-palvelimen laatiminen onnistuu kirjoittamalla vain muutaman rivin koodia. Tosin tämä pakottaa kehittäjän noudattamaan valmiiksi määriteltyjä menetelmiä, kuten miten HTTP-pyyntöjä ja -otsikoita käsitellään. Tänä päivänä verkkosovelluskehittäjien ei tyypillisesti tarvitse työskennellä alhaisella HTTP-protokollien ohjelmatasolla. Jotkut ohjelmistoalustat tarjoavat HTTP-palvelimen suoraan, mutta Node.js-kehittäjän täytyy itse määrittää HTTP-palvelin. (Herron 2020.)

3.2.1 Express

Verkkopalvelimien laatimisen yksinkertaistamiseksi Node.js-yhteisöstä löytyy useita verkkosovelluskehyksiä, jotka tarjoavat yleisesti vaadittuja korkeamman tason rajapintoja. Näillä kehyksillä on myös helppo lisätä HTTP-palvelimeen lisäominaisuuksia, kuten sessioiden hallintaa, evästeitä, staattisten tiedostojen lähetystä ja lokikirjoitusta. Kun viitekehykset sisältävät valmiiksi toteutettuja lisäominaisuuksia, kehittäjät pystyvät keskittymään paremmin myös sovelluksen muihin tärkeisiin osa-alueisiin, kuten liiketoimintalogiikkaan. (Herron 2020.)

Express on suosituin Node.js-verkkokehys ja se toimii monien suosittujen Node.js-kehysten taustalla olevana kirjastona. Expressin tarjoamia ominaisuuksia ovat:

- Pyyntöjen käsittely erilaisille HTTP-toiminnoille, eri URL-reiteillä
- Yleiset verkkosovellusasetukset, kuten yhteyden muodostamiseen käytettävä portti tai vastausten laatiminen
- Väliohjelmistojen lisääminen mihin tahansa kohtaan pyyntöjen käsittelyputkessa.

Vaikka Express on itsessään minimalistinen kokonaisuus, siihen on kehitetty yhteensopivia väliohjelmistopaketteja lähes kaikkiin verkkokehitysongelmiin. (MDN Web Docs 2023a.)

Yleensä verkkokehykset määrittelevät olevansa joko mielipiteellisiä tai mielipiteettömiä viitekehyksiä, eli ne ovat toteutustavaltaan joko joustamattomia tai joustavia. Express on mielipiteetön, eli joustava viitekehys. Siinä voi lisätä lähes mitä tahansa yhteensopivia väliohjelmistoja käsittelyketjuun. Express-sovelluksen kokoonpanon voi jäsentää omalla tavalla vaikka yhteen tiedostoon tai useisiin tiedostoihin, käyttämällä mitä tahansa hakemistorakennetta. (MDN Web Docs 2023a.)

3.2.2 Jest

JavaScript-kieli tukee testiviitekehyksiä, jotka helpottavat testien kirjoittamiseen vaadittavaa työtä. Monet testityökalut on kirjoitettu JavaScriptille modulaarisesti siten, että niistä on mahdollista valita yksi kirjasto testien ajamiseen, toinen kirjasto testiväitteille ja kolmas tekaisemaan vastauksia. (Flanagan 2020.)

Vaikka testauskehyksiä on monia, kuten Jest, Mocha, Jasmine tai Karma, monet niistä käyttävät samanlaista syntaksia. Testauskehykset mahdollistavat useiden erilaisten testiväitteiden käytön. Testiväitteillä voidaan esimerkiksi tarkistaa, asettautuuko tulostettu arvo tietylle välille, tai että vastaako merkkijono tiettyä mallia tai onko tulostettu arvo validi. (Scott, MacDonald; Powers 2021.)

Jest on suosittu testikehys JS-kielelle, ja se sisältää kaiken testaukseen tarvittava toiminnan (Flanagan 2020). Jest tukee tietueiden tekaisemisen, asynkronisten tulosten käsittelyn, ajastimien simuloinnin ja tilannekuvatestauksen, jolla voi tarkistaa käyttöliittymässä tapahtuneita muutoksia. (Scott, MacDonald; Powers 2021.)

Jest-viitekehyksessä kaikki pyörii test()-funktion sisällä. Test()-funktioon syötetään kaksi argumenttia. Kuten koodilohkosta 1 nähdään, ensimmäinen argumentti on testin nimike, jota käytetään testiraportissa. Toinen argumentti on JS-funktio, joka sisältää yhden tai useamman testiväitteen. Väitteet todentavat niihin verrattavat arvot joko todeksi (hyväksytty testi) tai epätodeksi (hylätty testi).

Testiväitteiden luomiseen käytetään expect()-funktiota. Se toimii tarkistusfunktioiden kanssa, kuten toBe(), jotka arvioi testikutsun tulokset. (Scott, MacDonald; Powers 2021.)

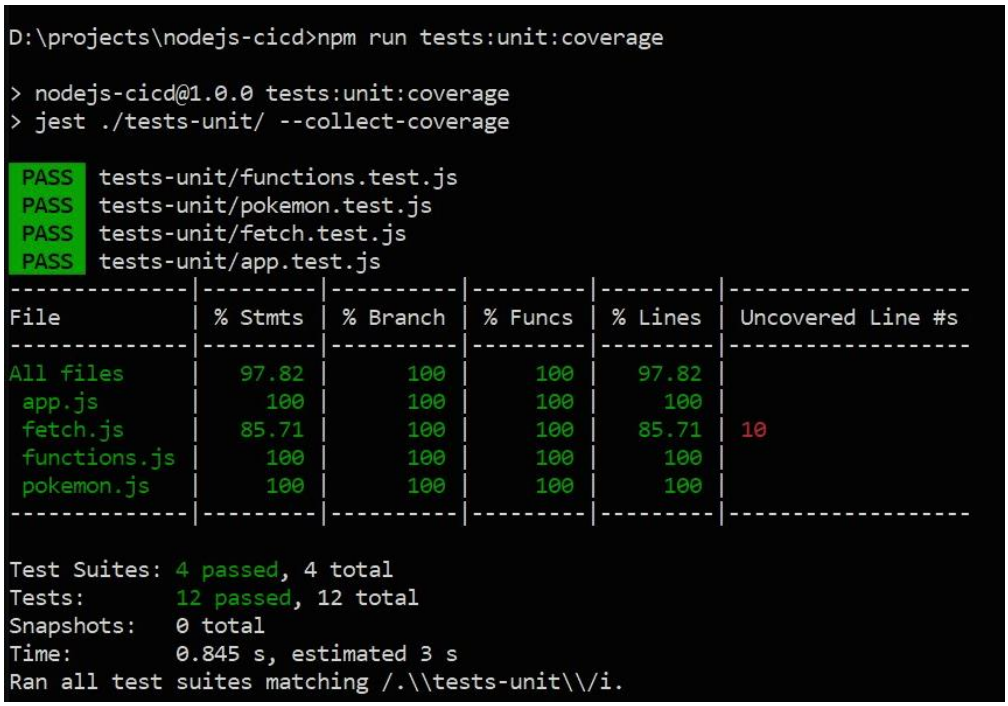
```
function AddOne(number) {
  return number + 1;
}
test("AddOne() returns incremented number", function () {
  const result = AddOne(10);
  expect(result).toEqual(11);
});
```

Koodilohko 1. Esimerkki Jest-yksikkötestistä.

Jest komentoparametri ‘--collect-coverage' käynnistää testikattavuusanalyysin. Tätä parametria voi käyttää komentoriviltä tai sen voi lisätä sovelluksen package.json-testikomentoon. Jest-testikattavuusraportti arvioi prosenttilukuna, kuinka suuri osa lähdekoodista on testattuna (kuva 10). Prosenttiluvut ovat jaettuna eri sarakkeisiin.

- Functions-sarakkeessa näytetään, mikä osa JS-funktioista on testattu. Tämä on hyvä lähtökohta testikattavuuden arviointiin, mutta ei kerro koko totuutta. Vaikka kaikki funktiot testattaisiin, se ei tarkoita, että kaikki koodirivit käydään läpi.
- Statements-sarakkeessa näytetään, mikä osa kaikista koodilausekkeista on katettu.
- Branch-sarakkeessa näytetään, mikä osa koodin ehdollisella logiikalla haarautuvista reiteistä suoritetaan.

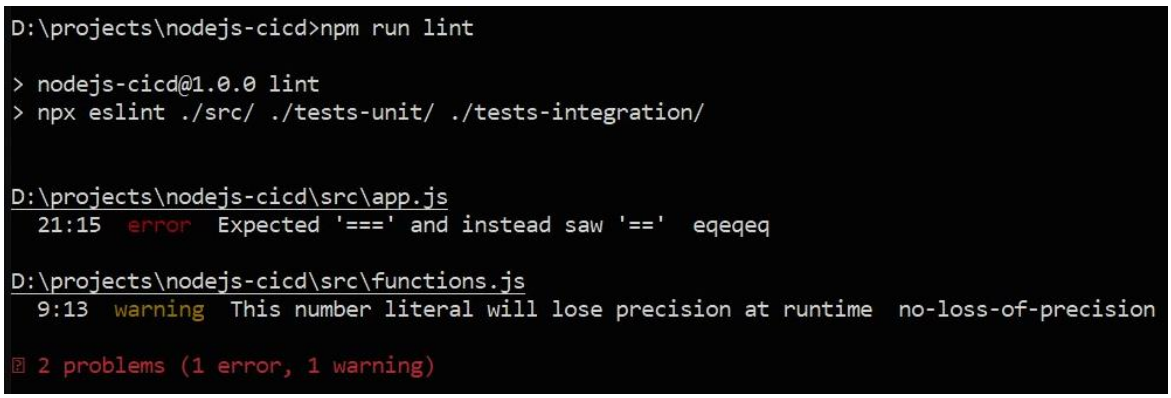
Raportti voi myös lisäksi viitata niihin koodiriveihin, joista puuttuu testikattavuus. Jest-komentoriviraportti palauttaa pikaisen katsauksen koodikattavuudesta. Siitä luodaan myös kattavampi raportti HTML-muodossa. Raportti kertoo testikattavuudesta lauseiden (Statements), haarojen (Branch) ja funktioiden (Functions) tarkan lukumäärän prosenttiosuuksien sijaan. (Scott, MacDonald & Powers 2021.)



Kuva 9. Jest-raportti testikattavuudesta ajamalla ’jest –collect-coverage’. Vihreät tekstit viittaavat hyvistä testituloksista.

3.2.3 ESLint

Lintteri on kirjoitetun koodin tarkistamiseen suunnattu työkalu. Lintterit etsivät koodista ongelmallisia kohtia, jotka voivat olla epäselvän näköisiä, epäoptimaalisia tai voivat tuottaa virheitä ajon aikana. Kuten kuvasta 11 nähdään, lintteri ilmoittaa konfiguroitavien asetusten perusteella koodiin liittyvistä suuntaviivausvirheistä. Kun lähdekoodi täyttää lintterin määritetyt asetukset, lintterin ajo läpäistään onnistuneesti. (Flanagan 2020.)



Kuva 10. Esimerkki ESLint-tuloksista ajamalla ’npm run lint’. Raportti antaa virheilmoituksen ’löysän’ vertailuoperaattorin käytöstä ja varoittaa liukuluvun tarkkuuden menetyksestä.

ESLint on konfiguroitava JavaScript-lintteri. ESLint-ohjelman ytimenä ovat säännöt. Säännöt vahvistavat, täyttääkö koodi tietyt vaatimukset ja mitä tehdään, jos vaatimukset eivät toteudu. ESLint sisältää satoja sisäänrakennettuja sääntöjä. ESLint-ekosysteemistä löytyy monia laajennuksia, jotka lisäävät sääntöjen määrää. (Flanagan 2020; ESLint 2023.)

ESLint-jäsentäjä muuntaa lähdekoodin abstraktiksi syntaksipuuksi, jota ESLint arvioi. ESLint käyttää oletusarvoisesti sisäänrakennettua Espree-jäsennintä, joka on yhteensopiva tavallisten JavaScript-ohjelmien kanssa. ESLint tarjoaa CLI-komentorivikäyttöliittymän, jonka avulla linttauksen voi suorittaa etäpäätteestä. CLI:ssä on useita asetuksia, joilla voi ohjata komentojen toimintaa. Node.js -ohjelmointirajapinnalla ESLintiä voi käyttää Node.js-koodissa. Rajapinta mahdollistaa laajennusten, integraatioiden ja muiden työkalujen kehittämisen ESLintiin. (ESLint 2023.)

ESLint-säännöt ovat kokonaan konfiguroitavissa. Asetustiedoston avulla voidaan asettaa juuri ne säännöt, mitä ESLint-ohjelmalla halutaan valvoa (Flanagan 2020).

3.3 GitHub Actions

GitHub tarjoaa lähdekoodin tietovaraston ja jatkuvan kehityksen alustan samassa paikassa GitHub Actions -työkaluilla. GitHub Actions -työkaluilla voidaan luoda työnkulkuprosesseja (workflow), jotka suorittavat automaattisesti työvaiheita sarjassa. Työvaiheet ovat prosessinkulkuja, joilla voidaan määrittää esimerkiksi lähdekoodin rakentamista, testausta, linkitystä tai julkaisua. Jokainen työvaihe sisältää yhden tai useamman alitehtävän. GitHub Actions -työnkulkuprosessit määritellään tietovaraston hakemistoon /.github/workflowsfolder, mistä GitHub tunnistaa ne automaattisesti. GitHub Actions -työnkulkuprosessit kirjoitetaan YAML-kielellä. (Halmagiu, Hiltunen, Osipova, & Rautio 2023.)

4 Rajapintasovelluksen kehitys

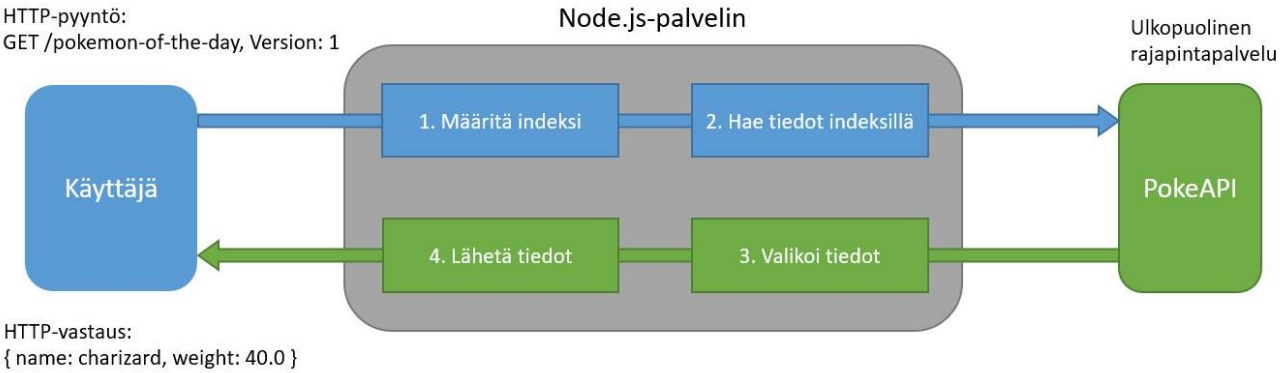
4.1 Esimerkkisovelluksen sisältö ja toiminta

Tämän opinnäytetyön tavoitteena on kehittää Node.js-projekti, joka sisältää valmiiksi toteutetun esimerkkitoteutuksen jatkuvan kehitysputken prosesseista. Sen tarkoituksena on toimia lähtöpisteenä uuden rajapintasovelluksen kehityksessä.

Esimerkkiprojekti toteutetaan luvussa 3 listatuilla teknologioilla, jotka pohjautuvat suurimmaksi osaksi JavaScript-kieleen. Jatkuvan kehitysputken prosessit muodostetaan käyttämällä GitHub Actions -toimintoja. Automatisoidun kehitysputken toiminnassa sovelletaan luvun 2 teoriatietoa. Jatkuvan kehitysputken lopussa ohjelmisto julkaistaan automaattisesti Heroku-palvelimelle.

Tähän esimerkkisovellukseen tulisi sisällyttää ainakin yksi toimiva rajapintaosoite, johon jatkuvan kehitysputken automaattitestien toiminta ja muut esitarkistusvaiheet voidaan kohdistaa. Samasta rajapintaosoitteesta on tarkoituksena muodostaa myös ainakin yksi tapa rajapintaosoitteiden versiointiin. Rajapintaversiointi mahdollistaa erilaisia käyttöönotonstrategioita, kuten luvussa 2.4.3 ja 2.4.4 mainitut canary- ja dark launch -julkaisut.

Tämän esimerkkisovelluksen rajapintaosoite on ’GET /pokemon-of-the-day’. Se palauttaa vastauksena takaisin päivän pokemonin nimen sekä muita valikoituja tietoja. Sovelluksen toimintalogiikka pidetään muuten yksinkertaisena (kuva 11). Sovellus määrittää kyseiselle päivälle indeksiarvon, jolla se sitten kyselee ulkoisesta rajapintapalvelusta indeksoidun pokemonin tietoja. Valikoidut tiedot päivän pokemonista palautetaan takaisin kyselyn lähettäjälle. Tällä ulkoisen rajapintaosoitteen käytöllä voidaan myös havainnollistaa joitain automaattitestauksen toimintoja, kuten mock-vastauksien käsittelyä.



Kuva 11: Esimerkkisovelluksen toiminta. Käyttäjän pyynnöstä muodostetaan indeksi, jolla haetaan pokemonin tiedot julkisesta PokeAPI-rajapintapalvelusta. Käyttäjälle lähetetään takaisin vastaanotetut tiedot, jonka sisältö riippuu alkuperäisen HTTP-pyynnön Version-otsikon arvosta.

4.1.1 Npm-pakettien asennus

Node.js-alusta käyttää npm-pakettien hallintaohjelmaa. Npm asentaa paketit oletuksena projektin paikalliseen hakemistoon, jolloin niitä voidaan käyttää sovelluksessa. Paketteja voi myös asentaa globaalisti, milloin ne ovat saatavilla koko tietokoneen käyttöjärjestelmässä. Globaalisti asennettuja paketteja voidaan käyttää työhakemistosta riippumatta komentorivisovelluksissa ja -apuohjelmissa, milloin halutaan. (Buckler 2022.)

Kun Node.js -ohjelmisto on asennettuna, uuden Node.js-projektin voi luoda pikaisesti kyseiseen hakemistoon komennolla ’npm init --yes’. Projektiin asennetaan myös npm-paketit Axios ja Express komennolla ’npm install express axios’. Expressiä käytetään rajapintasovelluksen viitekehyksenä ja Axios-paketilla pystytään luomaan omia rajapintakutsuja.

Projekteista löytyy usein kehitysaikaisia paketteja ja työkaluja, kuten lintteri tai testausohjelmat. Kehitysaikaisia paketteja ei käytetä silloin, kun sovellus on julkaistuna ja käytössä. Tämä tarkoittaa sitä, että näitä paketteja ei tarvita tuotantopalvelimella ollenkaan. Paketit, jotka ovat kehitysriippuvuuksia, on lueteltu erikseen package.json-tiedoston devDependencies-osiossa. Jos NODE_ENV-ympäristömuuttuja on asetettu tuotantotilaan, niitä paketteja ei asenneta silloin, kun pakettien asennuskomento ‘npm install’ ajetaan. (Buckler 2022.)

Projektin kehitysriippuvaisuuksiksi asennetaan JS-koodin lintteri ESLint, JS-testausviitekehys Jest, ja Express-sovelluksen testausta helpottavat ’superagent’ ja SuperTest komennolla ’npm install --save-dev eslint jest superagent supertest’. Asennetut paketit näkyvät listattuna package.json-tiedostossa ja node_modules-kansiosta löytyvät kaikki tarvittavat npm-paketit. Joitakin tyypillisiä paketteja, kuten ESLint tai Jest, asennetaan usein globaaliksi paketiksi. Näitä paketteja voi asentaa myös projektihakemistoon, jos halutaan varmistaa, että kaikilta projektin kehittäjiltä löytyy kyseinen moduuli (Buckler 2022). ESLint ja Jest voidaan asentaa globaalisti komennolla ’npm install jest eslint -g’.

4.1.2 Projektin sisältö

Sovelluksen JS-lähdekoodi sijoitetaan /src-hakemiston alle. Projektiin lisätään index.js-tiedosto (koodilohko 2), joka toimii rajapintasovelluksen lähtökohtana, ja app.js (koodilohko 3), jossa määritellään eri rajapintojen toiminta. Index.js importoi app.js-tiedostossa määritellyn Express-rajapintasovelluksen ja määrittää sen käynnistyksen listen()-funktion avulla. Kun palvelin käynnistyy, siitä kirjataan viesti konsoliin.

```
const app = require("./app");
const port = process.env.PORT || 3000;

// Run server
```

```
app.listen(port, () => {
  console.log(`Express app listening on port ${port}`);
});
```

Koodilohko 2. src/index.js-tiedoston sisältö.

```
const express = require("express");
const { GetDailyPokemonV1, GetDailyPokemonV2 } = require("./pokemon");

const app = express();

app.use(express.json({ limit: "1mb" }));

// Get pokemon of the day, API versioning in header
app.get("/pokemon-of-the-day", async function (req, res, next) {
  const version = req.get("Version");
  if (version === "1") {
    const result = await GetDailyPokemonV1();
    res.status(200).send(result);
    return;
  }
  if (version === "2") {
    const result = await GetDailyPokemonV2();
    res.status(200).send(result);
    return;
  }
  res.status(500);
  res.send("Invalid version header.");
  return;
});

module.exports = app;
```

Koodilohko 3. src/app.js-tiedoston sisältö.

App.js-tiedostossa määritellään Express-rajapintasovellukselle HTTP GET -tyyppinen rajapintaosoite app.get()-funktiolla. Rajapintaosoite on ”/pokemon-of-the-day”, ja se palauttaa takaisin kerran päivässä vaihtuvan pokemonin tiedot. Rajapinnasta löytyy kaksi versiota. Vastaukseen sisällytetään eri määrä dataa käytetystä versiosta riippuen. Versionumero selvitetään käyttäjän tekemän pyynnön otsikoista req.get()-funktiolla. Jos versiota ei ole määriteltynä, siitä palautetaan takaisin virheilmoitus koodilla 500. App.use(express.json({ limit: "1mb" }))-funktiolla määritellään, että rajapintasovellus pystyy käsittelemään json-tyyppisiä tiedostoja, rajoitettuna yhden megabitin kokoon.

App.js-tiedosto importoi GetDailyPokemon-funktiot pokemon.js-tiedostosta (koodilohko 4). Pokemon.js sisältää molemmille rajapintaversioille omat funktionsa. Molemmissa funktioissa määritellään kyseisen päivän indeksi ja sillä indeksillä haetaan ulkoisesta rajapintapalvelusta pokemon. Funktion toinen versio palauttaa pokemonista enemmän tietoja ensimmäiseen versioon verrattuna.

```
const { FetchPokemonByIndex } = require("./fetch");
const { GetRandomNumberOfTheDay } = require("./functions");

const maxPokeIndex = 1000;

async function GetDailyPokemonV1() {
  const seedNum = GetRandomNumberOfTheDay(maxPokeIndex);
```

```
  const pokemon = await FetchPokemonByIndex(seedNum);
  const result = { name: pokemon.name, types: pokemon.types };
  return result;
}

async function GetDailyPokemonV2() {
  const seedNum = GetRandomNumberOfTheDay(maxPokeIndex);
  const pokemon = await FetchPokemonByIndex(seedNum);
  const result = {
    name: pokemon.name,
    types: pokemon.types,
    abilities: pokemon.abilities,
    weight: pokemon.weight,
  };
  return result;
}

module.exports = { GetDailyPokemonV1, GetDailyPokemonV2 };
```

Koodilohko 4. src/pokemon.js-tiedoston sisältö.

Päivän indeksin hakeminen ja ulkoiseen osoitteeseen tehty rajapintakutsu on vielä määriteltynä eri moduuleissa, functions.js (koodilohko 5) ja fetch.js (koodilohko 6). Fetch.js-tiedostossa tehdään kutsu Axios-kirjastoa hyödyntäen julkiseen PokeAPI-rajapintapalveluun. Palvelu palauttaa indeksin mukaan tietoja kyseisestä pokemonista json-muodossa. Vastauksen tiedot saadaan palautettua vastausobjektin data-attribuutista.

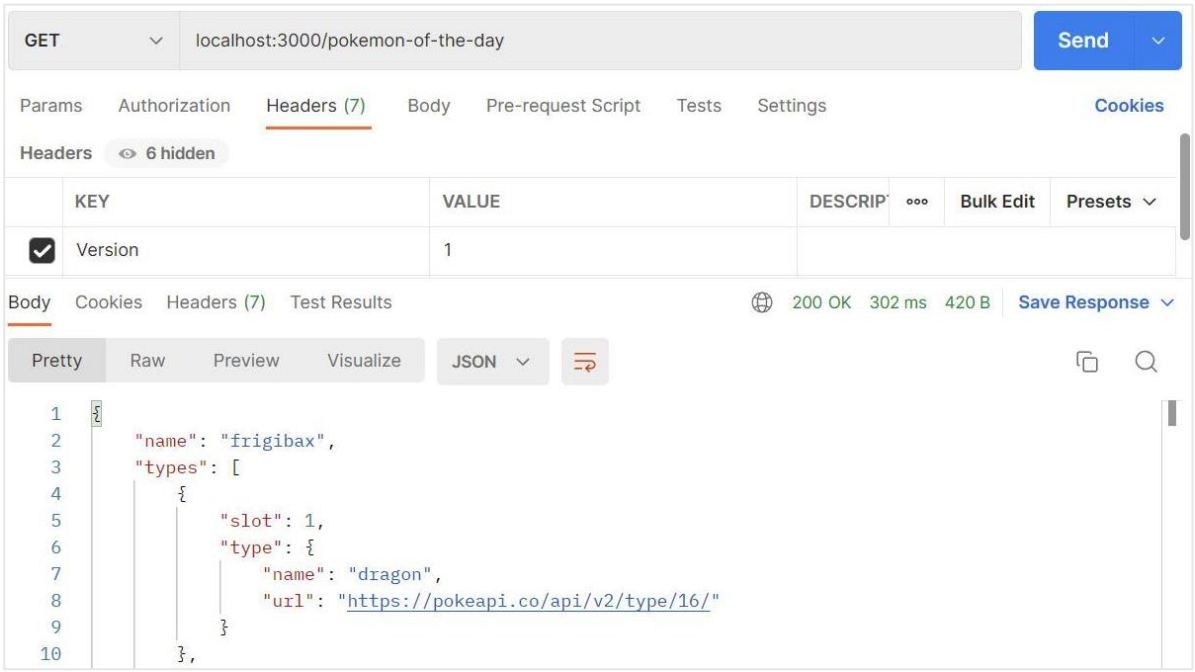
```
function GetCurrentDay() {
  const now = new Date();
  return now.getDate();
}

function GetRandomNumberOfTheDay(limit) {
  const useLimit = limit ?? 1000;
  const dateNum = GetCurrentDay();
  return Math.floor((dateNum * 666) % useLimit);
}

module.exports = { GetRandomNumberOfTheDay, GetCurrentDay };
```

Koodilohko 5. src/functions.js-tiedoston sisältö.

```
const axios = require("axios");

const pokeUrl = "https://pokeapi.co/api/v2/pokemon/";

async function FetchPokemonByIndex(index) {
  let response;
  try {
    response = await axios.get(pokeUrl + index);
  } catch (error) {
    console.log("error", error);
  }
  return response.data;
}

module.exports = { FetchPokemonByIndex };
```

Koodilohko 6. src/fetch.js-tiedoston sisältö.

Jos index.js ja muut JavaScript-tiedostot löytyvät /src-hakemistosta, rajapintasovelluksen saa käyntiin juurihakemistosta ajamalla komennon ‘node ./src/index.js’. Node.js-rajapintasovelluksen pitäisi olla nyt käynnissä ja vastaanottaa HTTP-pyyntöjä localhost:3000-osoitteeseesta. Esimerkiksi Postman-palvelua käyttäen sovelluksesta tulisi nyt saada toimiva vastaus, jos GET-pyyntö laaditaan osoitteeseen ’localhost:3000/pokemon-of-the-day’ (kuva 12). HTTP-pyynnön otsikkoon pitää muistaa myös lisätä ’Version’-arvo, sillä sovellus vaatii myös sen.



Kuva 12. Postman-sovelluksesta lähetetty HTTP GET -pyyntö rajapintasovellukseen otsikolla ’Version: 1’. Päivän pokemon on Frigibax!

4.1.3 Yksikkötestit

Sovelluksen yksikkötestit sijoitetaan /tests-unit’-hakemistoon. Jest tunnistaa testitiedoston ’.test.js’-päätteestä. App.js-moduulista laaditaan yksikkötestitiedosto ’app.test.js’ (koodilohko 7).

```
const request = require("supertest");
const app = require("../src/app");
const poke = require("../src/pokemon");

const mockPokemon = {
  name: "Pokemon 1",
  types: [{ slot: 1, type: { name: "Type name" } }],
};

jest.mock("../src/pokemon");
poke.GetDailyPokemonV1.mockReturnValue(mockPokemon);

describe("GET /pokemon-of-the-day", () => {
  test("Version header 1 returns name and type", () => {
    return request(app)
      .get("/pokemon-of-the-day")
```

```
      .set("Version", "1")
      .then((response) => {
        expect(response.statusCode).toBe(200);
        expect(response.body.name).toBe(mockPokemon1.name);
        expect(response.body.types).toStrictEqual(mockPokemon1.types);
      });
  });
});
```

Koodilohko 7. tests-unit/app.test.js-tiedoston sisältö.

Importoitu Express-app-objekti kääritään supertest-kirjaston request()-funktioon, ja sitä kautta se on helposti testattavissa. Koska app.js kutsuu toisen moduulin funktioita, näiden funktioiden paluuarvot olisi yksikkötestauksen käytäntöjen mukaan hyvä tekaista mock-vastauksella (Fernandes da Costa 2021).

Koko moduuli saadaan tekaistua käyttäen Jest-funktiota jest.mock(). Sitten moduulin yksittäisen funktion paluuarvon saa tekaistua funktiolla mockReturnValue(). Describe()-funktiolla otsikoidaan testattava rajapintaosoite, ja test()-funktion sisällä toteutetaan testitapaus. Testitapauksessa laaditaan GET-pyyntö version-otsikolla ’1’, ja then()-funktion sisällä käsitellään saatu vastaus käyttäen funktiorakennetta expect().toBe(). Koska pokemonin hakufunktio on tekaistu mock-vastauksella, testissä tarkistetaan, oliko palautettu objekti todella se sama mikä alun perin funktion paluuarvoon tekaistiin.

Projektin juurihakemistoon lisätään Jest-testausta varten jest.config.js-tiedosto, johon voidaan kirjoittaa Jest-testiajoon liittyviä asetuksia (koodilohko 8). Jest tunnistaa tämän tiedoston automaattisesti.

```
const config = {
  verbose: true,
  coverageThreshold: {
    global: {
      branches: 80,
      functions: 80,
      lines: 80
    },
  },
};
module.exports = config;
```

Koodilohko 8. test.config.js-tiedoston sisältö.

Verbose-asetuksella testiajon aikana tulostuu jokaista testiä kohden kirjaus konsoliin. CoverageThreshold-asetuksella on asetettu globaali koodikattavuuden raja. Jos koodikattavuuden raja ei ylity, prosessin ajo palauttaa virhekoodin ja testiajo julistetaan sitä kautta epäonnistuneeksi. Näillä asetuksilla testikoodikattavuuden pitää olla yli 80 %. Yksikkötestit ja koodikattavuus voidaan ajaa komennolla ’jest ./tests-unit/ --collect-coverage’.

4.1.4 Koodin lintteri

JavaScript-koodin linttausta varten luodaan .eslint.yml-tiedosto projektin juurihakemistoon. Tämän konfiguraatiotiedoston voi kirjoittaa monella eri kielellä. YAML-kieli valittiin sen syntaktisen selkeyden vuoksi. Tiedostoon kirjoitetaan muutama ESLint sääntö YAML-kielellä, kuten koodilohkosta 9 näkyy.

```
env:
  node: true
  jest: true
  es2022: true
rules:
  eqeqeq: error
  no-const-assign: error
```

Koodilohko 9. .eslint.yml-tiedoston sisältö.

Env-asetuksissa määritellään Node-, Jest- ja ECMAScript2022-ympäristöt, jotta ESLint jäsentäjä tunnistaa niiden koodisymbolit oikein. Rules-asetuksista löytyy kaksi error-tasoista ESLint-sääntöä, joista linttaus julistettaisiin epäonnistuneeksi. ’eqeqeq’ kieltää löysän vertailuoperaattorin käytön ja ’no-const-assign’ kieltää const-muuttujan uudelleenmäärittelyn. Kaikki projektin juurihakemiston alapuolelta löytyvät JavaScript-tiedostot voidaan lintata komennolla ’npx eslint ./**/*.js’.

4.2 Jatkuva integraatio ja vetopyynnöt

GitHubin käyttöön tarvitaan GitHub-käyttäjätunnus ja Git-ohjelma. Projektin juurihakemistoon laaditaan ensiksi ’.gitignore’-tiedosto, ja siihen määritellään, että node_modules/- ja coverage/ -hakemistoja ei työnnetä mukaan lähdehallintajärjestelmään (koodilohko 10).

```
/node_modules
/coverage
```

Koodilohko 10. .gitignore-tiedoston sisältö.

Uuteen tyhjään GitHub-tietovarastoon (repository) voidaan työntää projekti komennoilla ’git init && git add . && git commit -m "init" && git remote add origin <github-repository-url> && push -u origin master’. Muutokset on nyt työnnetty suoraan GitHub-tietovaraston master-haaraan. Jatkuvan integraation käytänteiden mukaisesti koodimuutoksia tulisi edeltää esitarkastusvaihe (Laster 2020). GitHub-projektiin määritellään ’Branch protection rules’-asetuksista ‘Require a pull request before merging’ voimaan master-haaralle. Nyt master-haaraan työntäminen on kiellettyä. Muutokset pitää tehdä ensiksi toiseen haaraan ja liittää master-haaraan vetopyynnön (pull request) avulla. GitHub-vetopyyntö toimii erinomaisena esitarkistusvaiheena.

Projekti sisältää jo koodin linttaus-, testaus- ja testikattavuustarkistukset. Nämä tarkistukset saadaan asetettua pakollisiksi vetopyyntökriteereiksi luomalla näistä prosesseista GitHub Action. Pull-

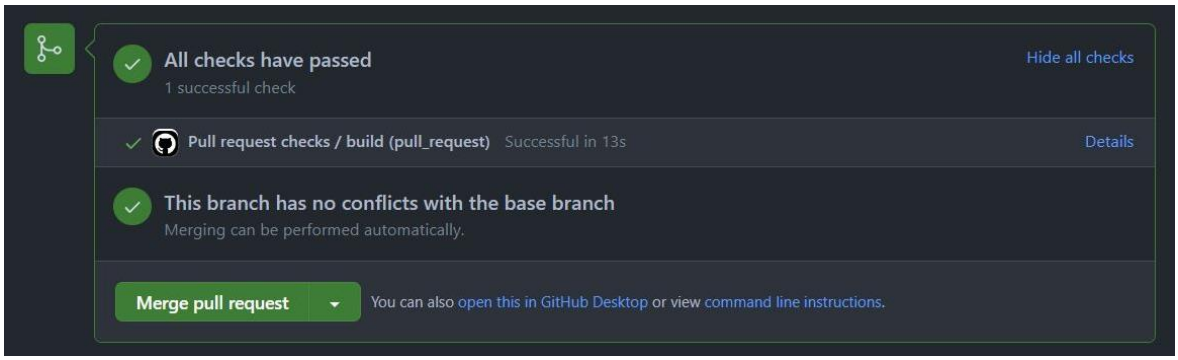
request-checks.yml-tiedosto luodaan .github/workflows-hakemiston alle (koodilohko 11). GitHub tunnistaa sieltä löytyvät .yml-tiedostot automaattisesti GitHub Actioneiksi. Asetetaan yml-tiedostoon linttaus- ja testikattavuus-komennot.

```
name: Pull request checks
on:
  pull_request:
    branches: [master]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Run npm
        uses: actions/setup-node@v3
        with:
          node-version: "16.x"
          cache: "npm"
      - name: Install
        run: npm ci
      - name: Lint
        run: npm run lint
      - name: Unit tests & coverage
        run: npm run tests:unit:coverage
```

Koodilohko 11. pull-request-checks.yml-tiedoston sisältö.

Tämä GitHub Action on määritelty käynnistymään aina, kun master-haaraan tehdään vetopyyntö komennolla ’on: pull_request: branches [master]’. Tämä prosessi on määritelty käyttämään node-versiota 16. Npm asennukset sekä linttaus- ja testikomennot suoritetaan peräkkäisessä järjestyksessä komennolla ’run’. Alkuperäiset komennot ovat aliasoitu npm-skripteillä, kuten ’npm run lint’. Npm-skriptejä voidaan määritellä package.json-tiedostossa.

Projektin master-haarasta voidaan muodostaa uusi kehityshaara ’develop1’ komennolla ’git checkout -b develop1’. Projektiin tehdyt muutokset työnnetään komennolla ’git add . && git commit -m "muutokset" && git push origin develop1’. Nyt GitHub-sivustolta voidaan luoda uusi vetopyyntö ’Pull requests’-osiosta, missä develop1-haaran muutokset liitetään master-haaraan. Vetopyynnön hyväksyntäkriteereissä näkyy nyt pull-request-checks.yml-tiedoston GitHub Action, joka lähtee automaattisesti käyntiin jokaisesta uudesta Git-työnnöstä (kuva 13).



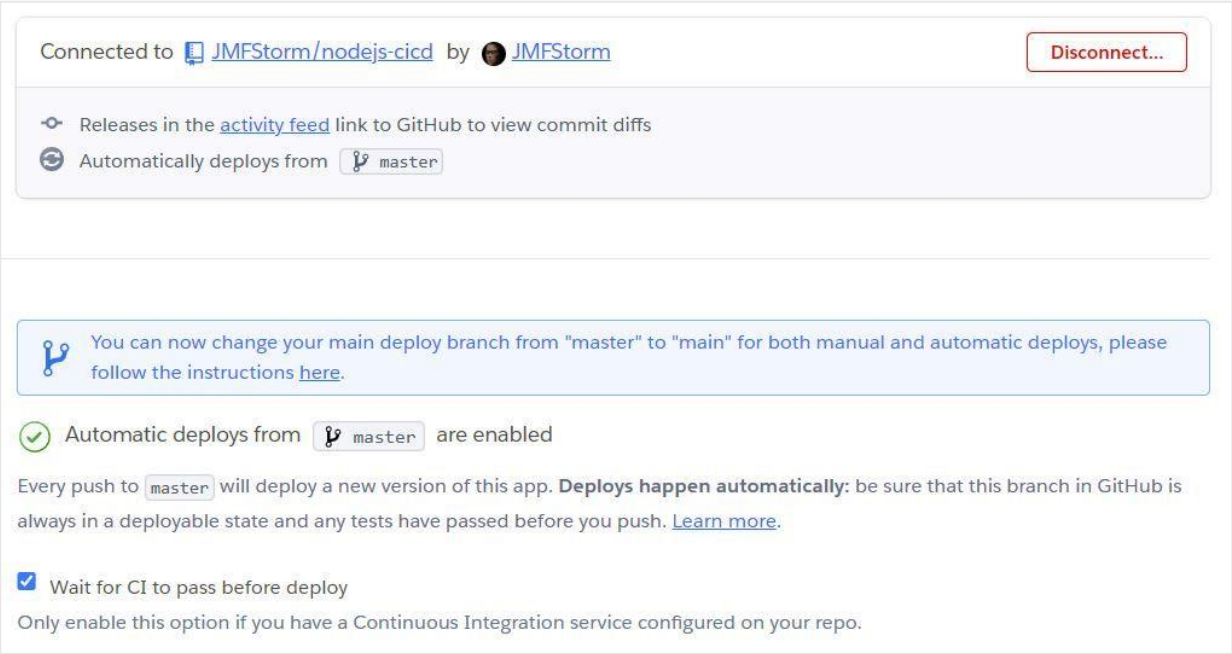
Kuva 13. pull-request-checks.yml-tiedostossa määritelty esitarkistusvaihe on suoritettu vetopyynnössä onnistuneesti ja vetopyyntö on valmis liitettäväksi master-haaraan.

4.3 Jatkuva toimitus ja käyttöönotto

4.3.1 Automaattinen julkaisu

Rajapintasovellukseen muodostetaan jatkuva käyttöönottovaihe, jossa sovellus julkaistaan palvelimelle. Tässä työssä rajapintasovellus julkaistaan Heroku-palveluun. Heroku tarjoaa palvelinympäristön monille eri ajoympäristöille, ja Node.js-palvelimen suhteen GitHubin kautta käyttöönotto on erittäin helppo prosessi. Kun Heroku-käyttäjätili ja uusi projekti on luotu, Heroku pystyy automaattisesti julkaisemaan sovelluksesta uuden version etukäteen määritellystä GitHub-projektin haarasta. Kun GitHub-käyttäjätili on liitetty Heroku-tilille ja projektin master-branch on määriteltynä alustaksi käyttöönottoon (kuva 14), Heroku julkaisee automaattisesti master-haaran sisällön palvelimelle joka kerta, kun siihen työnnetään muutoksia. Heroku tunnistaa Node.js-sovelluksen automaattisesti, ja koska palvelimen käynnistysskripti on ’npm run start’, Heroku osaa käynnistää rajapintasovelluksen automaattisesti.

Sovellus on tässä kohtaa jatkuvaa toimitusputkea yksikkötestattuna ainakin 80 prosenttisesti. Tosin mikään ei vielä takaa, että sovellus toimii kokonaisuutena oikein, kun se toimitetaan palvelimelle käyttöönotettavaksi. Sitä varten olisi syytä luoda integraatiotestejä, jotka varmistavat sovelluksen toiminnan kokonaisvaltaisemmin ennen käyttöönottoa. Heroku-palvelusta asetetaan ’Wait for CI to pass before deploy’ päälle (kuva 14), jolloin uutta versiota ei toimiteta käyttöönotettavaksi, ellei jatkuvan toimituksen esitarkistusvaihetta läpäistä onnistuneesti.



Kuva 14. Heroku deployment -asetukset.

4.3.2 Integraatiotestit

Rajapintasovellukseen muodostetaan integraatiotestejä hakemistoon /tests-integration. Ne muodostavat rajapintakyselyitä sovellukseen ja sitä kautta testaavat sekä sisäisten funktioiden että ulkoisen rajapintapalvelun yhteistoimintaa. Integraatiotestit voidaan ajaa komennolla ’jest ./tests-integration/’.

```
const axios = require("axios");

const baseUrl = "http://localhost:3000/";

describe("GET /pokemon-of-the-day", () => {
  test("Version header 1 returns name and types", async () => {
    const url = baseUrl + "pokemon-of-the-day";
    let res;
    try {
      res = await axios.get(url, {
        headers: {
          Version: "1",
        },
      });
    } catch (error) {
      console.log("error", error);
    }
    const result = res.data;
    expect(result).toBeDefined();
    expect(result.name).toBeDefined();
    expect(typeof result.name).toBe("string");
    expect(result.types).toBeDefined();
    expect(typeof result.types).toBe("object");
  });
});
```

Koodilohko 12. tests-integration/integrations.test.js-tiedoston sisältö.

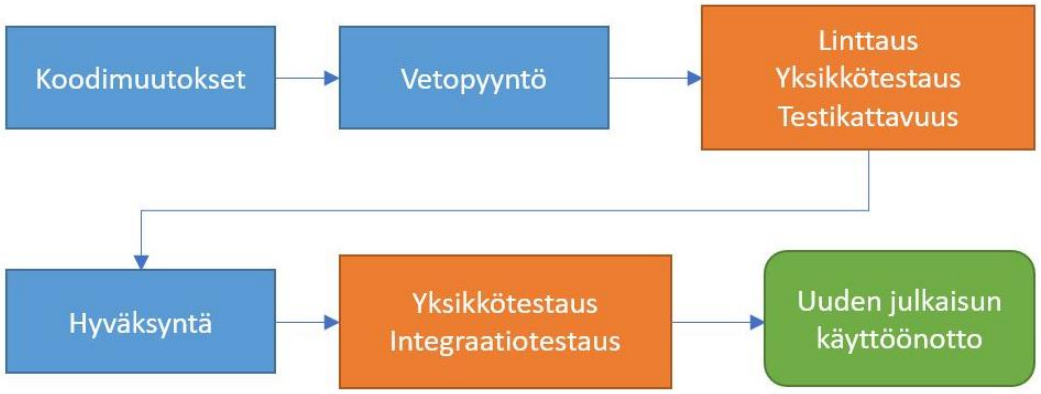
Integrations.test.js-tiedoston (koodilohko 12) testitapaus lähettää Axios-kirjastolla GET-pyynnön toimivalla ’version’-otsikolla sovelluksen rajapintaosoitteeseen ja olettaa saavansa validin vastauksen takaisin. Jotta tämän testitapauksen voi ajaa GitHub Actions -prosessissa, rajapintapalvelu pitäisi saada ensiksi käyntiin, ja vasta sitten tämä integraatiotesti voidaan suorittaa. Npm-paketti pm2 mahdollistaa useamman Node-prosessin suorittamisen samasta etäpäätteestä. Komennolla ’pm2 start ./src/index.js --name ci-test’ voidaan ensiksi käynnistää sovellus antaen sille prosessinimi ’ci-test’. Sen jälkeen integraatiotestit voidaan suorittaa, ja lopuksi sovelluksen käynnistänyt prosessi voidaan sulkea komennolla ’pm2 kill’. Tätä käyttöä varten pm2-paketti täytyy asentaa globaalisti.

Seuraavaksi muodostetaan uusi GitHub Action -tiedosto master-cicd.yml, joka suorittaa yksikkö- ja integraatiotestit joka kerta, kun projektin master-haaraan työnnetään uusi muutos (koodilohko 13).

```
name: Master CI/CD
on:
  push:
    branches: [master]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Run npm
        uses: actions/setup-node@v3
        with:
          node-version: "16.x"
          cache: "npm"
      - name: Install
        run: npm ci && npm install -g pm2
      - name: Unit tests
        run: npm run tests:unit
      - name: Integration tests
        run: npm run tests:integration:process
```

Koodilohko 13. tests-integration/master-cicd.yml -tiedoston sisältö.

Tämän jatkuvan testauksen automaatioprosessin npm-asennusvaiheeseen on lisätty pm2-paketin globaali asennus. Asennusten jälkeen ajetaan yksikkötestit ja sen jälkeen integraatiotestit. Pm2-prosessin käynnistys- ja sulkuvaihe, sekä integraatiotestaus on aliasoitu ’npm run tests:integration:process’-skriptillä. Nyt kun master-haaraan työnnetään muutoksia, sovellusta ei toimiteta Heroku-palvelimelle, ennen kuin nämä vaiheet ajetaan onnistuneesti GitHub Actions -työnkulkuprosessissa.



Kuva 15. Esimerkkisovelluksen jatkuvan kehitysputken vaiheet.

Nyt rajapintasovelluksesta on luotu jatkuvan kehityksen kokonaisuus (kuva 15). Uudet muutokset tarkistetaan pakollisessa vetopyyntövaiheessa jatkuvan integraation prosessien mukaisesti. Sitten vielä ennen toimitusvaihetta suoritetaan kattavampi testausprosessi, jonka jälkeen sovellus julkaistaan automaattisesti palvelimelle.

5 Tulokset

5.1 Lopullinen tuotos

Opinnäytetyön tuotoksena kehitettiin Node.js-rajapintasovellusesimerkki, jonka ympärille laadittiin automatisoitu jatkuva kehitysputki. Sovelluksen jatkuvan kehitysputki kattaa jatkuvan integraation ja käyttöönoton vaiheet käyttäen GitHub Actions -työnkulkuprosesseja. Sovellus tarjoaa esimerkillisen lähtötilanteen uuden Node.js-rajapintasovelluksen kehitykseen ja jatkuvan kehitysputken määrittelyyn. Esimerkkisovelluksen lähdekoodi on julkisesti saatavilla GitHub-sivustolta (liite 1) MIT-lisenssin alaisena.

Tämän työn esimerkkisovellus sisältää yhden toimivan rajapintaosoitteen, jossa esitellään yksi lähestymistapa rajapintojen versiointiin. Sovelluksen projektin ympäriltä löytyy valmiit skriptit ajamaan yksikkö- ja integraatiotestejä sekä koodin linttaus- ja testikattavuusanalyyseja, käyttäen Jest ja ESLint JavaScript-kirjastoja. Esimerkkisovellus hakee tietoa ulkoisesta rajapintaosoitteesta, jonka avulla projektissa pystytään havainnollistamaan yksikkötestien ja integraatiotestien eroavaisuuksia.

Lähdekoodi säilytetään GitHub-sivustolla, jota käytetään myös osana jatkuvaa kehitysputkea. Tämän GitHub -projektin Git master -haara on suojattuna suoralta työnnöltä, joten uusia muutoksia saadaan ainoastaan vetopyynnön kautta master-haaraan. Kun vetopyyntöön vaaditut kriteerit, kuten yksikkötestit ja koodin analyysit ovat suoritettu hyväksytysti, uudet muutokset siirtyvät master-haaraan ja sovelluksen uusi versio julkaistaan automaattisesti Heroku-palvelimelle.

5.2 Jatkokehitykset

JavaScript-kieli tekee jatkuvan toimituksen prosessista erittäin yksinkertaisen. JavaScript-lähdekoodi ei edes vaadi erillistä koontivaihetta toimiakseen, vaan tiedostot voidaan lähettää suoraan Node.js-palvelimelle ajettavaksi. Kuitenkin tiedostojen versiointi auttaa selkeyttämään eri julkaisujen eroavaisuuksia (Laster 2020). Eli myös JavaScript-lähdekoodi olisi hyvä versioida jollain automaattisella järjestelmällä ennen kuin se lähetetään palvelimelle.

Tämän työn jatkuva kehitysputki julkaisee esimerkkirajapintasovelluksen Heroku-palvelimelle. Käyttöönottostrategia on helposti laajennettavissa tai vaihdettavissa eri palvelinympäristöön. Käyttöönoton vaihetta voidaan laajentaa esimerkiksi hyväksyntätestausvaiheella, joka voitaisiin suorittaa palvelinympäristöstä. Jatkuvan käyttöönoton prosessin laajentaminen on tästä kohdasta täysin vapaasti valittavissa. Luvussa 6.3. pohditaan vielä tarkemmin eri julkaisutapojen toteuttamista.

6 Pohdinta

6.1 Johtopäätökset

Jatkuvan kehityksen toimintamalli on ajankohtainen taito ohjelmistokehityksessä. Node.js-kehitysalusta on edelleen laajasti käytössä monissa erikokoisissa projekteissa. (Herron 2020; Laster 2020.)

Työn valmistuessa tuli selväksi, että automatisoidun kehitysputken määrittely projektiin ei ole vaikeaa, kunhan se määritellään selkeästi projektin alkuvaiheissa ja sitä päivitetään uusien projektimuutosten tullessa. Jälkeenpäin tehtynä jatkuvan kehitysputken kehittely voi tulla projektissa haastavaksi arkkitehtuurillisista syistä.

Teoriaosuuteen kerätty kirjallisuus auttoi selkeyttämään jatkuvan kehitysputken tarkoitusta ja miten kehitysputkea voidaan soveltaa käytännössä. Teorialuvussa 2.1. mainitaan, että jatkuvan integraation vaihe lähtee tyypillisesti käyntiin uuden Git-haaran muodostamisesta ja päättyy siihen, kun Git-haarasta tehty vetopyyntö on suoritettu valmiiksi (Rossel 2017; Laster 2020; Dingare 2022). Tässä työssä jatkuvan integraation prosessiin sisältyi GitHub-ympäristössä tehty vetopyyntö, sekä siihen liitetyt GitHub Actions -toiminnot, jossa suoritetaan koodin linttaus, yksikkötestit ja koodin testikattavuusanalyysi.

Tätä työtä tehdessä tuli myös ilmi, että jatkuvan toimituksen tai käyttöönoton vaihe ei ole aina yhtä selkeästi määriteltävissä. Laster (2020) on myös viitannut siihen, että toimituksen ja käyttöönoton nimikkeitä käytetään usein synonyymisesti, vaikka ne tarkoittavat vähän eri asiaa. Teorialuvussa 2. Rossel (2017) mukaan jatkuva käyttöönotto on osa jatkuvan toimituksen vaihetta (kuva 1), ja tämä kuulostaa järkeenkäyvältä tavalta käsitellä termien eroavaisuuksia.

6.2 Testityyppien määrittely

Kuten myös Fernandes da Costa (2021) asiaan viittasi, eri testityypit eivät ole aina selkeästi määriteltyjä. Esimerkkisovellusta kehitettäessä yksikkötestien ja integraatiotestien välille ei syntynyt aluksi selkeää rajaa. Sovelluksessa yksikkötestin laajuus määrittyi olemaan pienin yksittäisen JavaScript-moduulin osa, jonka sisällä kutsut toisiin moduuleihin tekaistaan mock-vastauksilla. Tosin yksikköteiksi voitaisiin myös määrittää sellaiset testitapaukset, joissa funktion sisäisiä kutsuja ei tekaista, mutta kokonaisuudessaan funktion toiminta ei ylety ulkoiseen palveluun asti. Esimerkkisovelluksen GetRandomNumberOfTheDay()-funktio (koodilohko 7) on tästä yksi esimerkki.

Integraatiotestin määrittely oli esimerkkisovellusta tehtäessä vaikeampaa yksikkötestiin verrattuna. Sovelluksen yksinkertaisuuden vuoksi integraatiotestiksi määriteltiin kokonainen rajapintakutsu ja

sen paluuarvojen käsittely. Tällä tavoin rajapinnan täydellinen käyttötarkoitus tulee testattua kokonaan, ulkoinen rajapintapalvelu mukaan lukien. Toinen mahdollisuus integraatiotestin määrittelyyn olisi ollut muuten samanlainen tapaus, paitsi että kutsu ulkoiseen rajapintasovellukseen edelleen tekaistaisiin mock-vastauksella.

Jos integraatiotestaus on liitetty kehitysputken käyttöönoton esitarkistusvaiheeseen, ulkoisen rajapintavastauksen tekaisu voisi olla tyypillisesti parempi ratkaisu. Kuten teorialuvussa 2.3.2. mainittiin, integraatiotestin on tarkoitus testata sovelluksen toiminta ulkoista rajapintavastausta käsitellessä, eikä testata itse ulkoisen rajapintasovelluksen toimintaa (Fernandes da Costa 2021). Tämän esimerkkisovelluksen ulkoista rajapintapyyntöä ei tekaistu integraatiotesteissä, sillä esimerkkisovellus ei kuitenkaan toimisi, jos ulkoinen rajapintasovellus ei olisi myöskään toiminnassa. Testitapaukset ovat lopuksi aina kehittäjän päätettävissä.

6.3 Eri julkaisutapojen toteutus

Tämän työn esimerkkisovellus havainnollistaa yksinkertaisen käyttöönoton mallin, jossa automatisoitujen tarkastusten onnistuttua sovellus julkaistaan automaattisesti palvelimelle. Esimerkkisovelluksessa on jo esitelty yksi metodi eri rajapintaversioiden käyttöön, ja projektin mukauttaminen eri käyttöönoton malleihin ei olisi vaikeaa.

Jos käyttöönoton strategiana käytettäisiin blue-green-julkaisun strategiaa, julkaisu ei todennäköisesti saisi olla täysin automaattinen. Sen sijaan kehittäjällä tulisi olla mahdollisuus päättää, milloin uusi julkaisu toteutetaan, ja kumpaan ympäristöön julkaisu toimitetaan. Tästä toiminnasta tulisi muodostaa uusi GitHub Action, joka voidaan käynnistää ainoastaan käsin. Käsin käynnistettävään GitHub Action -prosessiin voidaan käyttää ’workflow_dispatch’-tapahtumaa (GitHub 2023).

Jos käyttöönoton strategiana käytettäisiin canary-julkaisua, automatisoitua kehitysputkea ei todennäköisesti tarvitsisi muuttaa ollenkaan. Canary-julkaisussa osajoukko käyttäjistä ohjataan käyttämään uudempaa palvelun versiota. Silloin Node.js-sovellukseen tulisi vain lisätä jokin logiikka, joka määrittäisi milloin kukakin käyttäjä ohjautuu käyttämään rajapintasovelluksen eri versiota.

Dark-launch-julkaisun käyttöönotossa käytännöllinen toteutus voi vaikuttaa blue-green- ja canary-metodien sekoitukselta. Rajapintasovelluksesta saatetaan edelleen ylläpitää toista kokonaista tai osittaista, uudempaa versiota, ja sovelluksen rajapintaliikenne peilataan uusiin ulkoisiin osoitteisiin. Esimerkkisovelluksessa on jo havainnollistettu, miten uusia rajapintakutsuja muodostetaan Axios-kirjaston avulla.

Ominaisuusliputuksen lisääminen sovellukseen on yksinkertainen prosessi. Kehitysympäristössä liputetut ominaisuudet listataan tyypillisesti ulkoiseen tekstitiedostoon, jota ei seurata Git-

versiohallinnassa. Sovelluksen käynnistyttyä arvot luetaan ja liitetään Node.js process.env -objektiin ympäristömuuttujiksi (Flanagan 2020). Tähän käyttötarkoitukseen löytyy npm-paketti ’dotenv’. Jatkuvassa kehitysputkessa process.env-objektin arvot ovat helposti määriteltävissä Jest-testien sisällä. Heroku mahdollistaa Node.js ympäristömuuttujien määrittelyn palvelinympäristössä ’Config Vars’-asetuksissa.

6.4 TypeScript-kielen lisääminen

Vahva koodin tyyppitarkistus on todistettavasti hyödyllinen ominaisuus ohjelmistokehityksessä. JavaScript-kieltä käytetään yhä enemmän monimutkaisissa sovelluksissa, joten koodin kääntäjästä olisi yhä enemmän hyötyä ohjelmointivirheiden havaitsemisessa. TypeScript-kieli on JavaScript-ympäristöstä laadittu merkittävä jatkotuote. Se tuo vahvan tyypityksen JavaScript-kieleen. (Herron 2020.)

Koodin vahva tyypitys on merkittävä etu ohjelmoinnissa, koska kehittäjä ei voi muistaa ulkoa kaikkia projektin funktioiden ja objektirakenteiden tyyppejä. Vahva tyypitys tuo välittömästi enemmän varmuutta kirjoitettuun koodiin, ja kääntäjä auttaa löytämään koodista mahdollisia virheitä. Sen sijaan pelkällä JavaScript-kielellä ongelmat saatetaan havaita vasta kun ohjelma on käynnissä ja kohdataan ajon aikainen suoritusvirhe.

TypeScript-kielen käytön hyöty kasvaa sitä mukaan, mitä isompi projekti on kyseessä ja kuinka monta kehittäjää on projektissa mukana. Kun käytetään vahvasti tyypitettyä koodikieltä, kehittäjät ymmärtävät paremmin sekä oman että myös toisten tekemän koodin toimintaa ja tarkoitusta.

Tässä esimerkkisovelluksessa ei ole käytetty TypeScript-kieltä, vaikka sen käyttö on tänä päivänä yleistä, ja on joillekin kehittäjille täysin välttämätön asia. Tämä työ on keskittynyt jatkuvien kehitysmenetelmien toteuttamiseen, ja TypeScript ei olisi itsessään tuonut siihen lisäarvoa. Jos esimerkkisovellusta käytetään uuden Node.js-projektin pohjana ja projekti halutaan kirjoittaa TyeScript-kielellä, TypeScript-asennukset tulisi tehdä siihen heti ensimmäiseksi.

TypeScript-kielen käyttö Node.js-rajapintasovelluksessa tuo paljon etuja kehitykseen. Vahva tyypitys on etenkin tietokantamallien ja rajapintavastausten määrittelyssä hyödyllinen ominaisuus. Tyypitys vahvistaisi myös testausta, sillä tyypitysten avulla testitapauksiin pystytään määrittelemään tarkemmin ja helpommin testifunktioiden oikeat paluuarvot ja tietotyypit.

TypeScript-kielen käyttö edellyttää jonkin verran enemmän työtä projektin ohjelmistojen asennuksessa. Tänä päivänä kaikista suosituista npm-paketeista löytyy erillinen paketti TypeScript-tyypitykselle. Tämä tarkoittaa sitä, että projektiin pitää asentaa enemmän npm-paketteja, jos haluaa saada

kaiken hyödyn TypeScript-kielen käytöstä. Jotkut paketit edellyttävät vielä erillistä konfiguraatiota TypeScript-ympäristöä varten.

TypeScript-kielen käyttö monimutkaistaisi kehitysputken toteutusta jonkin verran. Node.js-palvelinalusta lukee ainoastaan JavaScript-kieltä. TypeScript-kääntäjän tehtävänä on lukea TypeScript-tiedostot ja kääntää ne JavaScript-tiedostoiksi. Jatkuvaan kehitysputkeen tulisi sisällyttää vaihe, missä TypeScript-kääntäjä muuntaa TS-tiedostot JS-tiedostoiksi. Sen jälkeen automatisoitu prosessi voi poimia JS-tiedostot ja lähettää ne palvelimelle. Tästä vaiheesta muodostuisi selkeämpi koodin rakennus- ja artefaktin luontivaihe, mikä on yksi osa jatkuvaa toimitusta (Laster 2020; Dingare 2022).

Lähteet

Buckler, C. 2022. Node.js: Novice to Ninja. SitePoint. Luettavissa: https://learning.oreilly.com/library/view/node-js-novice-to/9781098141004/. Luettu: 4.2.2023.

Dingare, P.P. 2022. CI/CD Pipeline Using Jenkins Unleashed: Solutions While Setting Up CI/CD Processes. Apress. Luettavissa: https://learning.oreilly.com/library/view/ci-cd-pipeline-using/9781484275085/. Luettu: 12.2.2023.

ESLint 2023. Core Concepts. Luettavissa: https://eslint.org/docs/latest/use/core-concepts. Luettu: 14.2.2023.

Fernandes da Costa, L. 2021. Testing JavaScript Applications. Manning Publications. Luettavissa: https://learning.oreilly.com/library/view/testing-javascript-applications/9781617297915/. Luettu: 4.2.2023.

Flanagan, D. 2020. JavaScript: The Definitive Guide, 7th Edition. O'Reilly Media, Inc. Luettavissa: https://learning.oreilly.com/library/view/javascript-the-definitive/9781491952016/. Luettu: 4.2.2023.

GitHub 2023. Manually running a workflow. Luettavissa: https://docs.github.com/en/actions/managing-workflow-runs/manually-running-a-workflow. Luettu: 19.2.2023.

Gough, J., Auburn, M. & Bryant, D. 2022. Mastering API Architecture. O'Reilly Media, Inc. Luettavissa: https://learning.oreilly.com/library/view/mastering-api-architecture/9781492090625/. Luettu: 5.2.2023.

Halmagiu, M., Hiltunen, T., Osipova, A. & Rautio, A. 2023. Getting started with GitHub Actions. Luettavissa: https://fullstackopen.com/en/part11/getting_started_with_git_hub_actions. Luettu: 18.2.2023.

Herron, D. 2020. Node.js Web Development. Packt Publishing. Luettavissa: https://learning.oreilly.com/library/view/navigating-the-continuous/9781492031949/. Luettu: 5.2.2023.

Kruchten, P., Nord, R. & Ozkaya, I. 2019. Managing Technical Debt: Reducing Friction in Software Development luettavissa: https://learning.oreilly.com/library/view/managing-technical-debt/9780135646052/. Luettu: 17.3.2023.

Laster, B. 2019. Navigating the Continuous Delivery Pipeline. O'Reilly Media, Inc. Luettavissa: https://learning.oreilly.com/library/view/continuous-integration-vs/9781492088943/. Luettu: 25.1.2023.

Laster, B. 2020. Continuous Integration vs. Continuous Delivery vs. Continuous Deployment, 2nd Edition. O'Reilly Media, Inc. Luettavissa: https://learning.oreilly.com/library/view/continuous-integration-vs/9781492088943/. Luettu: 25.1.2023.

MDN Web Docs 2023a. Express/Node introduction. Luettavissa: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction. Luettu: 12.2.2023.

MDN Web Docs 2023b. HTTP Messages. Luettavissa: https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages. Luettu: 12.2.2023.

MDN Web Docs 2023c. TCP. Luettavissa: https://developer.mozilla.org/en-US/docs/Glossary/TCP. Luettu: 6.3.2023.

Node.js 2023. Overview of Blocking vs Non-Blocking. Luettavissa: https://nodejs.org/en/docs/guides/blocking-vs-non-blocking/. Luettu: 12.2.2023.

Rossel, L. 2017. Continuous Integration, Delivery, and Deployment. Packt Publishing. Luettavissa: https://learning.oreilly.com/library/view/continuous-integration-delivery/9781787286610/. Luettu: 2.2.2023.

Scott, A.D., MacDonald, M. & Powers, S. 2021. JavaScript Cookbook, 3rd Edition. O'Reilly Media, Inc. Luettavissa: https://learning.oreilly.com/library/view/javascript-cookbook-3rd/9781492055747/. Luettu: 4.2.2023.

Van Winkle, L. 2019. Hands-On Network Programming with C. Packt Publishing. Luettavissa: https://learning.oreilly.com/library/view/hands-on-network-programming/9781789349863/. Luettu: 3.2.2023.

Liitteet

Liite 1. GitHub-linkki

GitHub-linkki opinnäytetyössä toteutettuun Node.js-rajapintasovellukseen.

https://github.com/JMFStorm/nodejs-cicd