Opinnäytetyö (AMK)

Tieto- ja viestintätekniikka

2021

Aleksanteri Jaakola

OHJELMISTOTESTAAMINEN JA TESTIAUTOMAATIO OSANA OHJELMISTOKEHITYSTÄ

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tieto- ja viestintätekniikka

2021 | 29 sivua

Aleksanteri Jaakola

OHJELMISTOTESTAAMINEN JA TESTIAUTOMAATIO OSANA OHJELMISTOKEHITYSTÄ

Ohjelmistotestauksen ja testiautomaation merkitys on suuri ohjelmistokehityksessä. Lisäämällä ohjelmistotestaamisen osaksi sovelluskehitystä saavutetaan monia hyötyjä, joista yksi tärkeimmistä on kustannustehokkuus. Tällä tarkoitetaan sitä, kun sovelluksen tai ohjelmiston viat havaitaan aikaisin, säästetään rahaa pitkällä aikavälillä. Tässä opinnäytetyössä tarkastellaan ohjelmistotestausprosessia ohjelmistokehityksessä ja ohjelmistotestaamisen automatisointia osana ohjelmistokehitystä.

Testiautomaation merkitys korostuu ohjelmistokehityksessä, kun ohjelmistotestauksessa on automatisoitavia työtehtäviä ja ohjelmistotestauksen testitapaukset toistettavia. Testiautomaatio parantaa ohjelmistotestaamisen tarkkuutta ja ylläpitää kilpailukykyä, kun manuaaliset toimenpiteet automatisoidaan. Yrityksen kilpailukykyä ylläpidetään, kun voidaan tuoda markkinoille uusia testattuja tuotteita nopeasti.

Opinnäytetyö on toteutettu kolmessa vaiheessa. Ensimmäisessä vaiheessa on tutustuttu ohjelmistotestaamisen teoriaosuuteen ja sen vallitseviin peruskäytäntöihin. Toisessa vaiheessa on tarkasteltu ohjelmistotestaamisen automatisoinnin teoriaosuutta ja sen vallitsevia käytäntöjä. Viimeisessä osuudessa nämä molemmat teoriaosuudet on yhdistetty ja toteutettu www-sovellusprojektiin käytännössä.

Tämän opinnäytetyön johtopäätöksenä voidaan todeta, että ohjelmistotestaaminen on suuressa roolissa ohjelmistotestaamisessa ja tuotteen laadun varmistamisessa. Ohjelmistotestaaminen auttaa ohjelmistoyrityksiä suorittamaan kattavan arvioinnin ohjelmistosta tai sovelluksesta ja varmistamaan, että tuote täyttää asiakkaiden tarpeet.

ASIASANAT:

Ohjelmistotestaaminen, ohjelmistotestaamisen tasot, ohjelmistotestauksen automatisointi, gitlab, jatkuva integraatio, jatkuva julkaisu

TURKU UNIVERSITY OF APPLIED SCIENCES

Information and Communication Technologies

2021 | 29 pages

Aleksanteri Jaakola

SOFTWARE TESTING AND TEST AUTOMATION AS PART OF SOFTWARE DEVELOPMENT

The importance of software testing and test automation is great in software development. The importance of software testing is highlighted by the reduction in maintenance costs and the improvement in the availability and functionality of the product.

Thesis examines the software testing process in software process in software development and the automation of software testing as part of software development.

The importance of test automation is emphasized in software development when software testing involves automated work tasks and software testing test cases to replicate. In this case, test automation increases the overall efficiency of the application and ensures strong quality. The main advantages of test automation are repeatability and high accuracy of test cases.

The thesis has been implemented in three phases. The first phase has explored the theory portion of software testing and it its prevailing basic practices. The second phases have examined the theory share of software testing automation and its prevailing practices. In the last section, both theory contributions have been combined and implemented into progressive web application project.

KEYWORDS:

Software testing, software testing levels, software testing automation, gitlab, continuous integration, continuous deployment

SISÄLTÖ

.

KUVAT

KÄYTETYT LYHENTEET

React Native	Mobiilikehityksessä käytetty framework
Storybook	Testaus työkalu komponenteille
Jest	JavaScript yksikkötestaus kehysympäristö
Framework	Kehysympäristö
UI	Käyttöliittymä
Komponentti	Luokka tai funktio
Funktio	Aliohjelma, tietyn toiminnon suorittamiseen tarkoitettu käsky
Pipeline	Deployable unit path is called pipeline
Script	Ohjelmakoodi
Container	Kontti
Npm	Tietokoneohjelmisto kirjasto
CRA	Create-React-App komento, jolla luodaan uusi react projekti
CLI	Komentorivi

1 JOHDANTO

Ohjelmistotestauksesta on monia hyötyjä ja yksi tärkeimmistä on kustannustehokkuus. Lisäämällä ohjelmistotestauksen projektiin voidaan sillä säästää rahaa pitkällä aikavälillä. Ohjelmistokehittäminen koostuu monista eri vaiheista ja jos viat löydetään aikaisin, niiden korjaaminen maksaa vähemmän. Siksi on tärkeää, että ohjelmistotestaaminen tehdään mahdollisimman pian (Ilze, 2018).

Ohjelmistotestaamisen tärkeys ei johdu pelkästään kustannustehokkuudesta, vaan myös siitä, että jotkut viat voivat olla hengenvaarallisia. Tästä hengenvaarallisuudesta on paljon erilaisia esimerkkejä.

- Vuonna 2014 Nissan kutsui korjattavaksi miljoona ajoneuvoa, koska turvatyyny-järjestelmä ei toiminut oikein. Tästä syystä kolarin sattuessa, turvatyyny ei välttämättä täyttynyt ilmalla (Healey, 2914).
- Ohjelmistohäiriö aiheutti F-35 havaitsemaan muodostuksessa olevat kohteet virheellisesti (Military.com, 2016).

Kustannustehokkuuden lisäksi ohjelmistotestauksessa pyritään löytämään virheitä kehitetystä sovelluksesta tai ohjelmistosta. Ohjelmistotestaus on prosessi, jossa jatkuvasti arvioidaan ominaisuuksien toimivuutta sekä virheettömyyttä (Thelin, 2020).

Ohjelmistotestaamisen tavoitteena on ymmärtää eri sovellusten tai ohjelmistoversioihin liittyvät riskit. Ohjelmistosovellusten jakautuessa yhä monimutkaisimmiksi kasvaa testitapausten suorittamisen määrä ja toistettavuus. Testitapausten toistettavuuden ja määrän kasvaessa on alettu puhumaan testiautomaatiosta. Testiautomaatio on käytäntö, jossa testausprosessi automatisoidaan siten, että ohjelmistosovelluksen todellista suorituskykyä verrataan odotettuihin tuloksiin (Raza, 2020).

Tässä opinnäytetyössä tarkastellaan ohjelmistotestaamista ja testiautomaatiota. Ohjelmistotestaaminen ja testiautomaatio toteutetaan olemassa olevaan www-sovellusprojektiin. Opinnäytetyön tarkoituksena on selvittää mitä on ohjelmistotestaaminen ja testiautomaatio sekä miten ohjelmistotestaaminen ja testiautomaatio on oleellinen osa ohjelmistokehitystä.

Ohjelmistotestaamisesta ja testiautomaatiosta on tehty paljon opinnäytetöitä ja tutkimuksia jo aiemmin, esimerkiksi Laurean ammattikorkeakoulussa (Närhi, 2019) ja Centrian

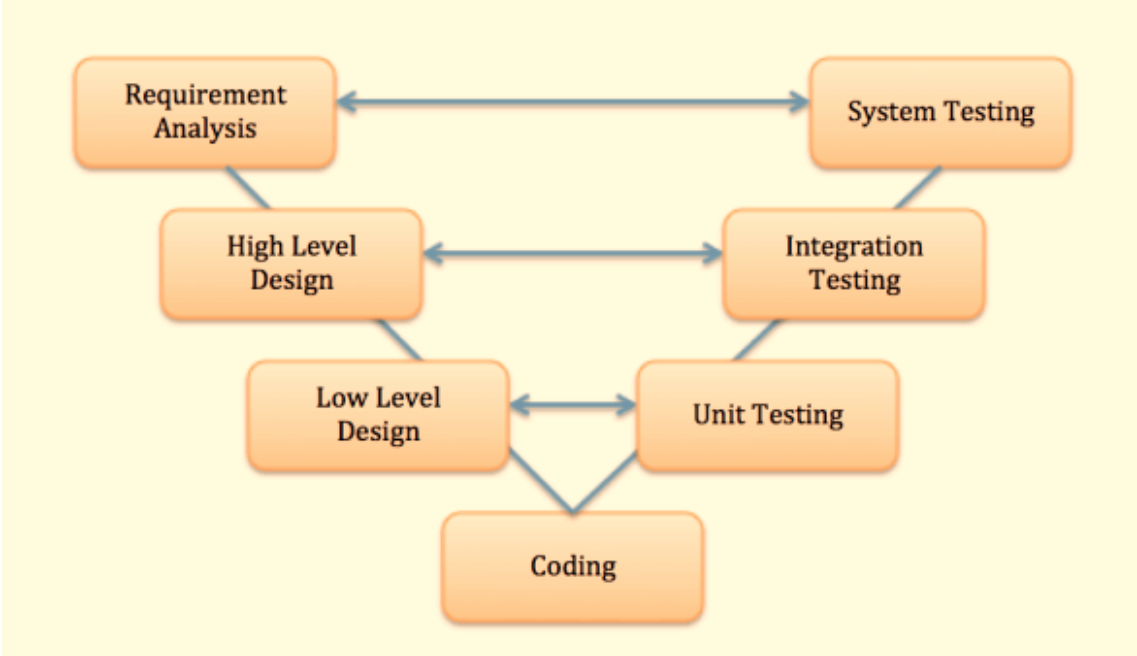
ammattikorkeakoulussa (Pollari, 2014). Närhen opinnäytetyössä käsitellään pääasiassa testiautomaation nykytilaa ja siinä tulleita haasteita sekä testiautomaation parantamista. Pollarin opinnäytetyössä käsitellään enemmänkin suurten ohjelmistojen testauksen perusteita ja GSM-verkon elementtien testausta. Tässä opinnäytetyössä ei keskitytä pelkästään testiautomaation parantamiseen tai ohjelmistojen testauksen perusteisiin, vaan keskitytään ohjelmistotestaamisen ja testiautomaation vallitseviin käytäntöihin sekä näiden molempien toteuttamiseen konkreettisesti verkkosovellukseen.

2 OHJELMISTOTESTAUKSEN TASOT

Ohjelmistotestaus on prosessi, jossa arvioidaan ohjelmistosovelluksen toimivuutta tarkoituksena selvittää, täyttääkö sovellus sille asetetut vaatimukset ja tunnistaa viat sovelluksessa tuottaakseen laadukkaan sovelluksen (Rajkumar, 2021).

Yleisesti ohjelmistotestaus on jaettu neljään eri tasoon: yksikkötestaukseen-, järjestelmätestaukseen-, integraatiotestaukseen- ja hyväksyntätestaukseen. Tasojen tarkoitus on tehdä ohjelmistotestauksesta systemaattista ja helposti tunnistaa kaikki mahdolliset testitapaukset. (Guru99)

Ohjelmistotestauksessa on olemassa muitakin tasoja, mutta nykyään käytettävässä V-mallissa tasot on jaettu neljään pääkategoriaan. Kuva 1 tuo esille selkeästi sen, mistä V-mallissa on kyse: jokaiselle ohjelmistokehityksen elämänkaaren vaiheelle on oma ohjelmistotestauksen vaiheensa. (Guru99)



Kuva 1. Ohjelmistokehityksen- ja ohjelmistotestauksen vaiheet (Guru99)

2.1 Yksikkötestaus

Yksikkötestauksella tarkoitetaan ohjelmistokoodiin kuuluvien yksittäisten osien kuten luokkien ja funktioiden testaamista. Yksikkötestauksen testien avulla tavoitteena on selvittää, että yksittäiset osat toimivat halutulla tavalla. (htt21)

2.2 Järjestelmätestaus

Järjestelmätestauksessa testataan koko järjestelmän toimintaa yksittäisten osien sijaan. Testaus suoritetaan käyttäjän näkökulmasta ilman automatisointia, joka vie tämän takia yleensä paljon aikaa. Järjestelmätestausvaiheessa löytyneet viat yleensä viittaavat siihen, että yksikkö- tai integraatiotestauksessa on ollut puutteita. (Vertics Oy)

2.3 Integraatiotestaus

Integraatiotestauksessa testataan integrointia tai käyttöliittymiä komponenttien välillä, järjestelmän vuorovaikutusta testattavan sovelluksen tai järjestelmän muihin osiin, kuten käyttöjärjestelmää, tiedostojärjestelmää tai järjestelmien välisiin rajapintoihin. Integrointitestaus on ohjelmistotestauksen keskeinen osa (Tryna.com).

2.4 Hyväksyntätestaus

Hyväksyntätestauksessa testataan kokonaista tuotetta. Tämän testausvaiheen tarkoituksena on selvittää, täyttääkö tuote sille ennalta määrätyt vaatimukset. Yleensä testaajina ovat loppukäyttäjät itse tai heidän edustajansa. (Katara, 2011)

3 OHJELMISTOTESTAUKSEN TEKNIIKAT

3.1 Mustalaatikkotestaus

Mustalaatikkotestaus on testausmenetelmä, jota käytetään ohjelmiston tai sovelluksen testaamiseen tuntematta ohjelmistokoodin tai ohjelman sisäistä rakennetta. Mustalaatikkotestauksessa käsitellään ohjelmistoa mustana laatikkona, josta ei tiedetä mitään. Ohjelmisto voi olla mikä tahansa järjestelmä esimerkiksi: käyttöjärjestelmä, tietokanta, verkkosivusto tai yrityssovellus (Hoogenraad, 2020).

Mustalaatikkotestauksen tarkoituksena on selvittää täyttääkö ohjelmisto käyttäjän odotukset. Testauksen avulla pyritään löytämään seuraavia virheitä; alustus- ja lopetusvirheet, virheelliset- ja puuttuvat toiminnot, virheet tietorakenteissa tai ulkoisessa tietokannassa ja käyttäytymis- ja toteutusvirheet. Mustalaatikkotestimenetelmää sovelletaan integraatiotestauksessa-, järjestelmätestauksessa- ja hyväksyntätestauksessa (Hoogenraad, 2020).

3.2 Valkolaatikkotestaus

Valkolaatikkotestauksessa arvioidaan ohjelmiston tai sovelluksen sisäistä rakennetta ja tunnistetaan mahdolliset suunnitteluvirheet. Valkolaatikko viittaa termiin nähdä ohjelmiston uloimmalle ’boksille’ nähdäkseen ohjelmiston sisäisen rakenteen. Valkolaatikkotestausta kutsutaan myös termeillä lasilaatikkotestaus-, koodipohjainentestaus sekä läpinäkyvä laatikkotestaus (Johnson, 2020).

Valkolaatikkotestauksessa tiedetään ohjelmiston sisäinen rakenne, mukaan luettuna lähdekoodi, IP-osoitteet ja tietoverkko protokollat. Valkolaatikkotestauksen tarkoituksena on arvioida ohjelmiston sisäinen rakenne ohjelmistokehittäjän näkökulmasta. (Johnson, 2020).

3.3 Harmaalaatikkotestaus

Harmaalaatikkotestauksessa yhdistetään mustalaatikko- ja valkolaatikkotestaus. Testaajalla on osittainen ymmärrys ohjelmiston tai sovelluksen sisäisestä rakenteesta.

Ohjelmiston käyttötarkoitus ja sen tärkeimmät toiminallisuudet tiedetään. Mustalaatikkotestaus on prosessi, jossa etsitään ohjelmistosta virheitä antamalla syöttöarvo käyttöliittymästä ja todentamalla tämä arvo palvelinpuolella (Bartlett, 2018).

Harmaalaatikkotestauksen tarkoituksena on löytää virheet järjestelmän kaatumisesta, kun järjestelmää testaamiseen ulkoisia tekijöitä vastaan. Kyseinen testausmenetelmä on tehokas, kun testataan ohjelmistoa loppukäyttäjän näkökulmasta (Bartlett, 2018).

4 OHJELMISTOTESTAAMISEN TAVOITTEET

Ohjelmistotestaaminen on yksi kriittisimmistä prosesseista ohjelmistokehityksen elinkaaressa. Se auttaa yrityksiä suorittamaan kattavan arvioinnin ohjelmistosta tai sovelluksesta ja varmistamaan, että tämä täyttää asiakkaan tarpeet. Ohjelmistokehityksen elinkaaren testausvaiheet auttavat yrityksiä tunnistamaan ohjelmiston virheet ennen toteutusvaiheen alkamista. Mitä enemmän viivytetään ongelmien havaitsemista, sitä suuremmat kustannukset yritykset todennäköisesti kohtaavat (Performance Lab, 2008-2021).

Ohjelmistotestaamisen tavoitteet riippuvat paljon yrityksestä ja projektista. Tässä opinnäytetyössä ohjelmistotestaamisen tavoitteet jaetaan karkeasti kolmeen kategoriaan havainnollistaakseen testaamisen tavoitteita.

4.1 Lyhyen aikavälin tavoitteet

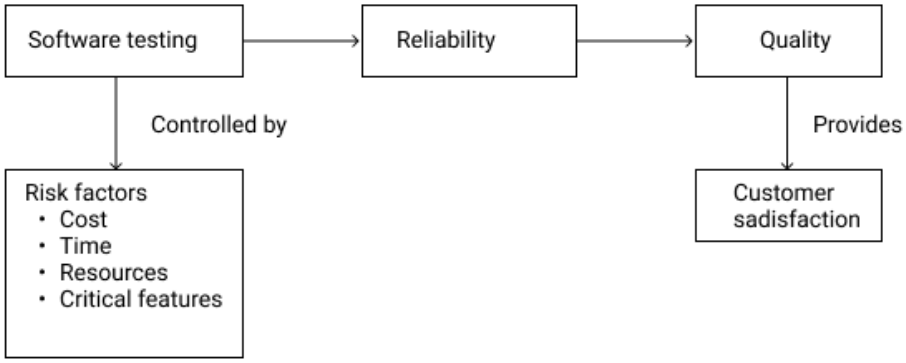
Ohjelmistotestaamisessa lyhyen aikavälin tavoitteena on virheiden löytäminen. Mitä enemmän virheitä löydetään ohjelmistokehityksen elinkaaren varhaisessa vaiheessa, sitä parempi on ohjelmistotestaamisen onnistumisprosentti (Nijhawan, 2012).

4.2 Pitkän aikavälin tavoitteet

Pitkän aikavälin tavoitteet vaikuttavat ohjelmiston tai sovelluksen laatuun pitkällä tähtäimellä. Ohjelmisto on tuote, joten sen laatu on ensisijainen käyttäjien näkökulmasta. Perusteellinen ohjelmistotestaus takaa hyvän laadun, siksi testausprosessin ymmärtämisen ja suorittamisen ensimmäinen tavoite on parantaa ohjelmistotuotteen laatua. Vaikkakin laatu riippuu useista eri tekijöistä, kuten oikeellisuudesta, eheydestä, tehokkuudesta ja niin edelleen, on luotettavuus tärkein tekijä laadun saavuttamiseksi (Nijhawan, 2012)

Käyttäjien näkökulmasta testauksen tärkein huolenaihe on vain asiakastyytyväisyys. Jos halutaan asiakkaan olevan tyytyväinen ohjelmistotuotteeseen, testaus on oltava perusteellinen. Perusteellinen testausprosessi saavuttaa luotettavuuden, mikä parantaa laatua ja laatu puolestaan lisää asiakastyytyväisyyttä (Nijhawan, 2012).

Hallittu riskienhallinta on isossa roolissa pitkän aikavälin tavoitteissa. Jokaisesta ohjelmistokehityksen projektista löytyy riskejä, mutta riskien hallinta on suuressa roolissa siinä mihin lopputulokseen saavutaan. Riskejä on hallittava, jotta niitä voidaan kontrolloida helposti. Ohjelmistotestaus voi toimia kontrollina, mikä voi auttaa poistamaan tai minimoimaan riskejä. Organisaatiot ovat riippuvaisia ohjelmistotestauksesta, joka auttaa heitä hallitsemaan liiketoimintatavoitteitaan (Nijhawan, 2012). Kuvassa 2 on havainnollistettu ohjelmistotestauksen riskien hallinnoimista suhteessa ohjelmiston laatuun.



Kuva 2. Ohjelmistotestauksen riskitekijöiden suhde asiakastyytyväisyyteen. Muokattu lähteestä (Ques10, 2019).

4.3 Toteutuksen jälkeiset tavoitteet

Toteutuksen jälkeiset tavoitteet viittaavat siihen, kun ohjelmisto tai sovellus on julkaistu. Ohjelmistotuotteen ylläpitokustannukset eivät ole sen fyysiset kustannukset, koska ohjelmisto ei kulu loppuun. Ohjelmistokehityksessä ohjelmistotuotteen ainoa ylläpitokustannus on virheiden aiheuttama vika (Ques10, 2019).

Mitä pidemmällä ollaan ohjelmistokehityksen elinkaaressa, sitä kalliimmaksi tulee ohjelmistovirheiden korjaaminen. Ohjelmistovirheiden korjaaminen on taloudellisesti kannattavampaa korjata ajoissa kuin myöhemmin. Ohjelmiston kehityksessä on usein vähemmän dataa, käytetään yhtä selainta ja ohjelmistoa käytetään juuri niin kuin on tarkoitettu. Ohjelmiston tuotantoympäristö on paljon monimutkaisempi ympäristö. Kun ongelmia

löytyy käyttöliittymästä, ne voivat olla vielä monimutkaisempia diagnosoida ja ratkaista (Freyja, 2017).

5 JAVASCRIPT TESTAUS FRAMEWORKIT

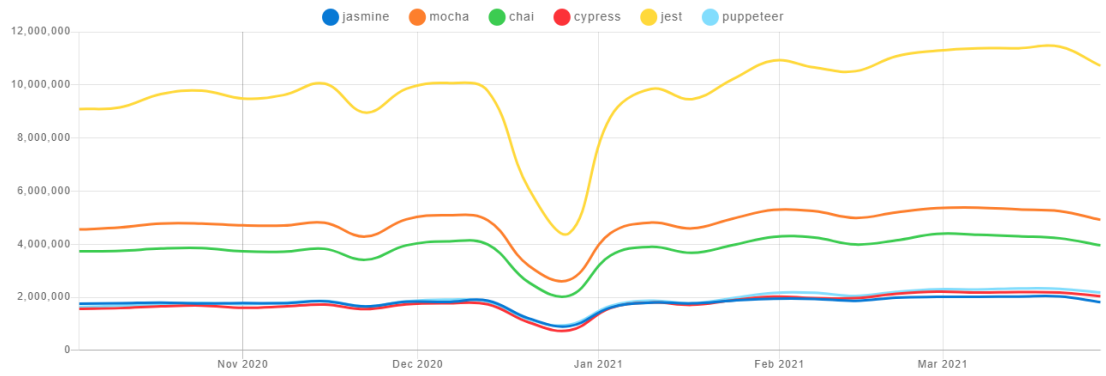
Ohjelmistotestauksessa käytettävät frameworkit eivät ole yksittäinen työkalu tai prosessi, vaan kokoelma työkaluja ja prosesseja, jotka toimivat yhdessä tukemaan minkä tahansa sovelluksen testausta. Ohjelmistotestauksen frameworkit yhdistävät joukon erilaisia toimintoja, kuten kirjastoja, testitietoja ja uudelleenkäytettäviä moduuleja (Clarion)

Näiden frameworkkien työkaluihin ja prosesseihin voivat kuulua koodausstandardit, testaustietojen käsittelymenetelmät, objektiarkistot tai testitulosten tallentamisprosessit. Objektiarkisto on kokoelma objekteja ja ominaisuuksia, joiden avulla objektiarkisto pystyy tunnistamaan objektit ja toimimaan sen kanssa. Esimerkiksi kun käyttäjä tallentaa testin, objektit ja sen ominaisuudet kaapataan oletusarvoisesti. Vaikka nämä eivät ole pakollisia ja testaajat voivat kommentoida tai tallentaa testejä noudattamatta niitä, frameworkin käyttäminen yleensä tarjoaa lisäetuja, jotka muutoin jäisivät pois (Aebersold).

5.1 Jest

Jest on Facebookin alun perin kehittämä JavaScript-testausframework, joka keskittyy verkkosovellusten yksinkertaisuuteen. Jest toimii muuan muassa Babel-, TypeScript-, Node.js-, ja React projektien kanssa (Wikipedia, 2021).

Jestin tärkeimmät edut ovat yhteensopivuus muiden kirjastojen kanssa, tavallinen JavaScriptiin perustuva syntaksi laajalla dokumentaation tuella, nopeus ja suorituskyky sekä testien hallinta suuremmilla objekteilla on mahdollista käyttäen Jestin ’Snapshot’-ominaisuutta (Unadkat, 2019). Kuvassa 3 havainnollistetaan erilaisten testaus frameworkkien suosiota.



Kuva 3. Erilaisten ohjelmistotestaus frameworkkien latausmäärät. (John Potter)

5.2 Mocha

Mocha on JavaScriptiin perustuva testaus- framework, jota käytetään laajalti asynkroniseen testaukseen. Asynkroninen kuvaa kahden tai useamman tapahtuman tai objektin välistä suhdetta, jotka toimivat vuorovaikutuksessa saman järjestelmän sisällä, mutta eivät esiinny ennalta määrätyin väliajoin. Ne eivät ole koordinoitu keskenään, jota tarkoittaa sitä, että ne voivat esiintyä samanaikaisesti. (Lutkevich, 2019)

Mochan suuria hyötyjä ovat toimivuus käyttöliittymässä ja palvelinpuolella, tarjoaa puhtaan pohjan kehittää testejä, tukee objektin pilkkaamista joustavien taustatestien suorittamiseksi sekä tukee mitä tahansa selainta. (Kothari, 2021).

6 STORYBOOK

Storybook on kehitystyökalu, jonka avulla voidaan rakentaa käyttöliittymäkomponentteja eristyksissä ja tallentaa komponenttien tilat tarinoina. Komponenttien rakentaminen eristyksissä tarkoittaa sitä, että ne ovat erillään siitä ohjelmistosovelluksesta, jossa niitä käytetään. Komponentin tilalla viitataan siihen, että komponentti voi pitää sisällään informaatiota, joka muuttuu ajan myötä. Tyypillisesti tilan muutos johtuu käyttäjän tapahtumista tai järjestelmätapahtumista. (Lindley)

Tarinoiden avulla on helppo tutkia komponenttia kaikissa sen eri muodoissa riippumatta siitä, kuinka monimutkainen komponentti on. Tarinat toimivat myös erinomaisena visuaalisena testitapauksena. Tarinat tallentavat tavan, jolla komponenttia voidaan käyttää. Tämä tarkoittaa sitä, että tarinoiden kokoelma on luettelo kaikista tärkeistä käyttötapauksista, joilla komponenttia voidaan testata. (Storybook)

Kuva 4. Storybookin suhde.

Tarinat ovat hyödyllisiä komponenttien tilojen tarkastamiseen, mutta joskus on testattava, miten komponentti muuttuu käyttäjän vuorovaikutuksessa. Tähän Storybook tarjoaa neljä menetelmää: yksikkötestit, regressiotestit, vuorovaikutustestit ja snapshot testit (Storybook).

Yksi merkittävistä eduista Storybook käyttämisessä on, että se pakottaa kirjoittamaan laadukkaampaa ohjelmistokoodia. Kun kirjoitetaan komponentteja eristyksissä, ei tarvitse huolehtia sovelluskohtaisista vaatimuksista, vaan keskitytään tekemään komponenteista uudelleenkäytettävä. Kaikkien reunatapausten seuraaminen helpottuu ja saadaan varmuus niiden riippumattomuudesta (Marek, 2020).

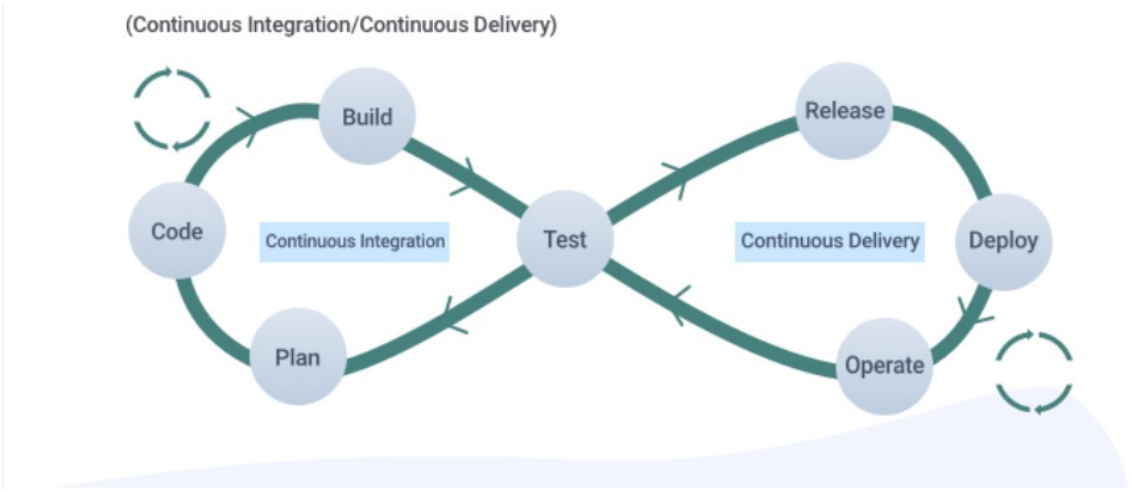
7 CI/CD METODOLOGIAT

CI/CD termi viittaa menetelmään toimittaa sovelluksia asiakkaille lisäämällä automaation osaksi ohjelmistokehityksen vaiheita. Pääkäsitteet CI/CD- menetelmässä ovat jatkuva integrointi, jatkuva toimitus ja jatkuva käyttöönotto (Brown, 2020).

CI/CD tuo käyttöön jatkuvan automaation ja seurannan sovellusten koko elinkaaren ajan integrointi- ja testausvaiheista aina toimitukseen ja käyttöönottoon. Yhdessä näitä yhdistettyjä käytäntöjä kutsutaan ’’CI/CD-pipelineksi” (Brown, 2020).

Jatkuva Integraatio (Continous Integration), johon usein viitataan sanalla ’CI’, on kokoelma käytäntöjä, jotka ajavat ohjelmistokehitystiimejä toteuttamaan pieniä muutoksia ja kirjaamaan muutokset versionhallinta-arkistoihin usein. Jatkuvan integraation tekninen tavoite on luoda johdonmukainen ja automatisoitu tapa rakentaa ja testata sovelluksia (Brown, 2020).

Jatkuva toimitus (Continous Deployment) jatkaa siitä, mihin jatkuva integraatio päättyy. Jatkuva toimitus automatisoi sovellusten toimittamisen erilaisille infrastruktuuriympäristöille. Tämä tarkoittaa sitä, että voidaan helposti esimerkiksi julkaista sovellus Apple Storessa (RedHat, 2021).



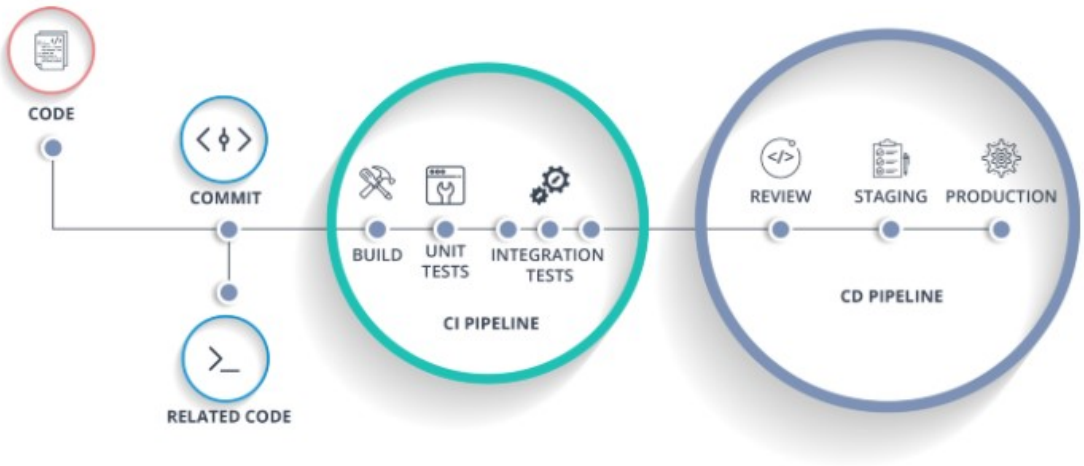
Kuva 5. Jatkuvan integraation ja jatkuvan toimituksen elämänkaari (Sing, 2020).

7.1 Gitlab CI/CD

Gitlabissa CI/CD konfiguroidaan tiedostosta nimeltään `.gitlab-ci.yml`, joka sijoitetaan ohjelmistoprojektin juureen. Tämä tiedosto luo CI/CD-pipelinen, joka on sarja vaiheita, jotka on suoritettava uuden ohjelmistoversion toimittamiseksi.

CI/CD-pipelinen muodostavat vaiheet ovat erillisten tehtävien osajoukkoja, jotka on ryhmitelty niin sanottuun pipeline vaiheeseen (RedHat, 2021). Tyypilliset pipelinen vaiheet ovat:

- Build- vaihe
- Test- vaihe
- Release- vaihe
- Deploy- vaihe



Kuva 6. Tyypillinen CI/CD-pipeline (Balajee, 2020).

Tyypillisesti Gitlabin CI/CD- järjestelmässä on kolme olennaista osaa: pipelinet, työt ja vaiheet. Työ on pienin yksikkö Gitlab CI/CD- järjestelmässä. Sitä kutsutaan usein nimellä ’build step’. Se voi olla yksikkötestien suorittamista, ohjelmistokoodin laadun tarkistamista tai ohjelmistokoodin kattavuuden tarkastamista. Kuvassa 8 on esimerkki työn suorittamisesta (Ryzycki, 2018).

```
unit tests:
  script:
    - npm install
    - npm run test
```

Kuva 7. Esimerkki työn kirjoittamisesta kooditasolla.

Jokainen työ kuuluu yhteen vaiheeseen. Vaihe voi sisältää yhden tai useamman työn, mutta voi olla myös, että vaiheeseen ei ole määritelty yhtäkään työtä. Kaikki työt samassa vaiheessa suoritetaan samanaikaisesti. Seuraavaan vaiheeseen päästään vain silloin, jos kaikki työt onnistuvat edellisestä vaiheesta (Ryzycki, 2018).

Pipeline termillä viitataan kokonaiseen järjestelmään, joka sisältää suoritettavia vaiheita. Pipeline pistää nämä vaiheet ja työt yhteen, ja aina kun työtä tai vaihetta suoritetaan tietävät ne oman järjestyksensä.

7.2 Gitlab runners

Gitlabissa käytettävä runner on eräänlainen työväline, joka suorittaa projektien CI töitä. Kun runner tulee saatavilla olevaksi se lähettää jatkuvasti pyyntöjä Gitlabin instanssiin, saadakseen toimeksiannon uudelle työlle. Erilaisilla runnereilla on erilaisia ominaisuuksia (esimerkiksi tietyllä runnerilla voi olla eri käyttöjärjestelmä ja toisella voi olla muistia enemmän). Runnerit sisällyttävät nämä ominaisuudet pyyntöihin tagien avulla. Tagien kautta saadaan tietyntyyppinen runner kiinnitettyä oikeaan työhön. Runnerit voidaan jakaa kolmeen eri kategoriaan spesific runners, group runners ja shared runners (Freeman, 2019).

7.3 Runnerin asentaminen

Käyttääksemme ’spesific runneria’ meidän on ensiksi asennettava se. Luodaan kansio osoitteeseen C:\Gitlab-runner. Ladataan binäärinen tiedosto Gitlabin sivuilta kyseiseen kansioon. Navigoidaan komentorivin avulla osoitteeseen C:\Gitlab-runner ja rekisteröidään runner. Kuvassa 10 on esimerkki komentoriviltä, kun spesific runner asennetaan.

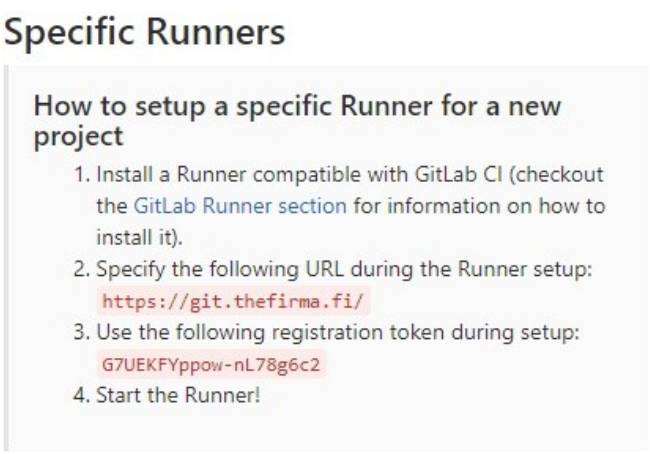
```
C:\Windows>cd..

C:\>cd gitlab-runner

C:\Gitlab-runner>gitlab-runner.exe register
Runtime platform                                    arch=386 os=windows pid=13960 revision=943fc252 version=13.7.0
Enter the GitLab instance URL (for example, https://gitlab.com/):
https://git.thefirma.fi/
Enter the registration token:
G7UEKFYppow-nL78g6c2
Enter a description for the runner:
[DESKTOP-PRH4IQ4]: my-runner
Enter tags for the runner (comma-separated):
test, build
Registering runner... succeeded                     runner=G7UEKFYp
Enter an executor: parallels, shell, ssh, docker+machine, kubernetes, custom, docker, docker-windows, docker-ssh, virtualbox, docker-ssh+machine:
shell
Runner registered successfully. Feel free to start it, but if it's running already the config should be automatically reloaded!

C:\Gitlab-runner>gitlab-runner.exe start
Runtime platform                                    arch=386 os=windows pid=12552 revision=943fc252 version=13.7.0

C:\Gitlab-runner>
```

Kuva 8. Spesific Runnerin asentaminen.

Komennolla **gitlab-runner.exe register** aloitetaan runnerin rekisteröiminen. Asennettaessa runneria on annettava tietyntyyppiset esitiedot runnerille: Gitlabin web-osoite, rekisteröinti token, runnerin kuvauksen, tagit ja toteuttajan.

CI työnkulku toimii siten, että esimerkiksi moduuleiden rakentamista ja niiden testaamista kutsutaan töiksi ja toteuttajat toteuttavat näitä töitä. Kun valitaan toteuttajaksi komentorivi, silloin Gitlabin runner integroituu komentorivin kanssa ja suorittaa CI töitä komentorivissä. (Singh, 2018)

GitLabin web-osoitteen ja rekisteröinti token saadaan, kirjautumalla sisään GitLabiin ja navigoimalla projektiin. Valitsemalla **Settings > CI / CD > Runners settings** ja painamalla näppäintä '**Expand**'. Kuvassa 11 on Gitlabin käyttöliittymästä löytyvä spesific runnerin asentamisen esitiedot.



Kuva 9. Spesific Runnerin asentaminen uudelle projektille.

8 WWW-SOVELLUSPROJEKTIN TEORIAOSUUS

Tässä projektissa toteutetaan ohjelmistotestaaminen ja testiautomaatio osaksi jo olemassa olevaa progressiivista verkkosovellusta. Verkkosovelluksen tavoitteena on mahdollistaa theFIRMAn luokkahuoneen tietokoneiden varaaminen verkkosovelluksen käyttöliittymästä.

Progressiivisella verkkosovelluksella tarkoitetaan verkkosovellusta, joka käyttää web-selaimen ohjelmointirajapintoja ja ominaisuuksia tuodakseen mobiilisovelluksen käyttäjäkokemuksen eri alustojen verkkosovelluksiin (Docs, 2021).

Sovellusprojektissa käsitellään käsitettä ohjelmistokoodin kattavuusraporttia, joka on mittari, jonka avulla tarkistetaan kuinka paljon lähdekoodista, on testattu. Kattavuusraportti on hyödyllinen, kun arvioidaan erilaisten testitapausten laatua.

Ohjelmistokoodin kattavuusraportissa käytetään yhtä tai useampaa kriteeriä määrittääkseen, miten ohjelmistokoodia käytetään testitapausten suorittamisen yhteydessä. Yleiset mittarit, jotka nähdään kattavuusraportissa ovat.

- Function coverage: kuinka monta määritellyistä funktioista on testattu
- Statement coverage: kuinka monta ohjelman lausekkeista on suoritettu
- Branches coverage: kuinka monta valvontarakenteiden haaraa on suoritettu
- Condition coverage: kuinka monta totuusarvomuuttujaa on testattu oikealla ja väärällä arvolla
- Line coverage: kuinka monta lähdekoodin riviä on testattu

9 WWW-SOVELLUSPROJEKTIN TOTEUTUS

Ennen itse ohjelmistotestausta on ladattava tarvittavat riippuvuudet testien kirjoittamiselle. Testien kirjoittamiseen tarvitaan tässä www-sovellusprojektissa Jest ja Enzyme. Enzyme on JavaScriptin ohjelmistotestauksen työkalu, jonka avulla on helpompi testata ja manipuloida komponenttien tulostusarvoja (Richardson). Koska Enzymen ohjelmointirajapinta pysyy muuttumattomana siitä huolimatta mitä React versiota käytetään, on ladattava adapteri tunnistamaan React versioiden muutokset. Kuvan 14 komennolla saadaan helposti ladattua molemmat.

```
npm i --save-dev enzyme enzyme-adapter-react-16
```

Kuva 10. Komento 2 riippuvuuden lataamiseksi.

Seuraavana toimenpiteenä on Enzymen adapterin konfiguraatio. Käytetään CRA mukana tulevaa setupTest.js tiedostoa siihen tarkoitukseen. Konfiguroidaan tiedosto kuvan 15 osoittamalla tavalla.

```
src > JS setupTests.js > ...
      You, seconds ago | 1 author (You)
1     // jest-dom adds custom jest matchers for asserting on DOM nodes.
2     // allows you to do things like:
3     // expect(element).toHaveTextContent(/react/i)
4     // learn more: https://github.com/testing-library/jest-dom
5     import '@testing-library/jest-dom';
6     import { configure } from "enzyme";
7     import Adapter from "enzyme-adapter-react-16";
8
9     configure({ adapter: new Adapter() });
```

Kuva 11. Ohjelmistokoodi tiedoston konfiguraatiosta.

9.1 Testitapausten kirjoittaminen www-sovellusprojektiin

Sen jälkeen, kun tarvittavat konfiguraatiot on tehty ja riippuvuudet ladattu kirjoitetaan testitapaukset. Kuvassa 16 on kirjoitettu kolme eri testitapausta www-sovellusprojektille. Ensimmäisessä testitapauksessa odotetaan, että Button komponentti on määritelty.

Toisessa testitapauksessa odotetaan, että komponentin snapshotit vastaavat toisiaan. Viimeisenä tarkistetaan, että www-sovellusprojekti renderöityy kaatumatta.

```
src > __tests__ > JS App.test.js > ...
      You, seconds ago | 1 author (You)
1     import React from "react";
2     import ReactDOM from "react-dom";
3     import { shallow } from "enzyme";
4     import Button from "../pages/Login";
5     import App from "../pages/App";
6
7     describe("Button", () => {
8       it("should be defined", () => {
9         expect(Button).toBeDefined();
10      });
11      it("should render correctly", () => {
12        const tree = shallow(<Button name="button test" />);
13        expect(tree).toMatchSnapshot();
14      });
15      it("renders without crashing", () => {
16        const div = document.createElement("div");
17        ReactDOM.render(<App />, div);
18      });
19    });
```

Kuva 12. Esimerkkinä ohjelmistokoodi testeistä.

Sitten kun testitapaukset ovat kirjoitettu on suoritettava testitapaukset. Kirjoittamalla komentoriville komento **npm run test,** suoritetaan kirjoitetut testitapaukset. Kuvassa 17 nähdään testitapausten suorittamisen tulokset.

```
Watch Usage
 › Press a to run all tests.
 › Press f to run only failed tests.
 › Press q to quit watch mode.
 PASS  src/__tests__/App.test.js
  Button
    √ should be defined (2 ms)
    √ should render correctly (7 ms)
    √ renders without crashing (18 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   1 passed, 1 total
Time:        5.157 s
Ran all test suites related to changed files.
```

Kuva 13. Ohjelmistotestien juoksemisen tulokset.

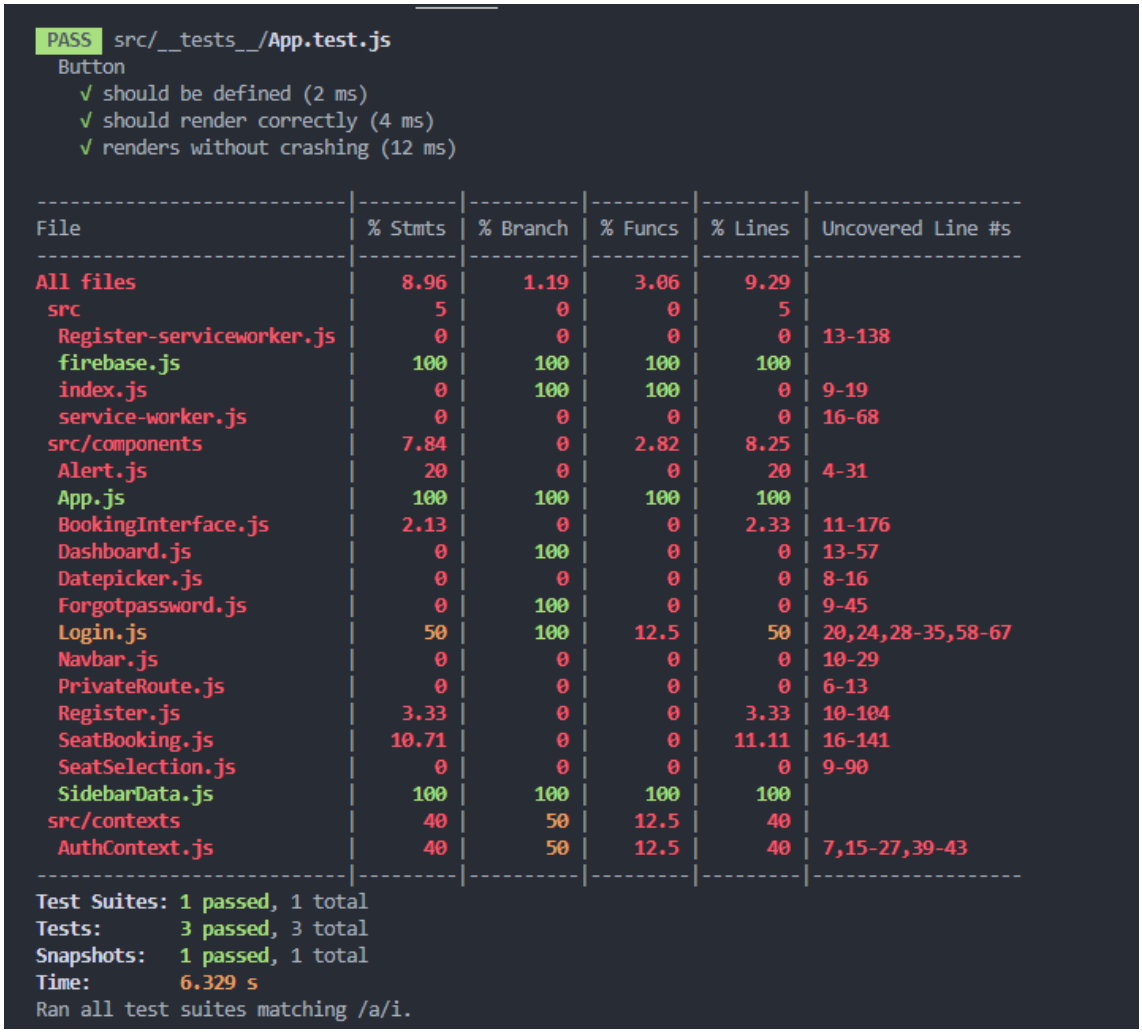
9.2 Ohjelmistokoodin kattavuus ja testauskattavuus

Kun halutaan ohjelmistokoodin kattavuusraportti osaksi testien tuloksia, on muokattava package.json tiedostoa lisäämällä sinne uusi ohjelmistokoodi rivi kuvan 18 osoittamalla tavalla.

```
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "test:coverage": "react-scripts test -coverage",
  "eject": "react-scripts eject"
```

Kuva 14. Tiedostoon lisättävä rivi.

Kirjoittamalla komentoriville `npm run -- --coverage a` suorittaa ohjelmisto testitapaukset ja luo kattavuusraportin www-sovellusprojektista. Kuvasta 20 nähdään selkeästi, että testitapaukset on suoritettu onnistuneesti ja kuinka paljon ohjelmistokoodia eri tiedostoista on testattu.

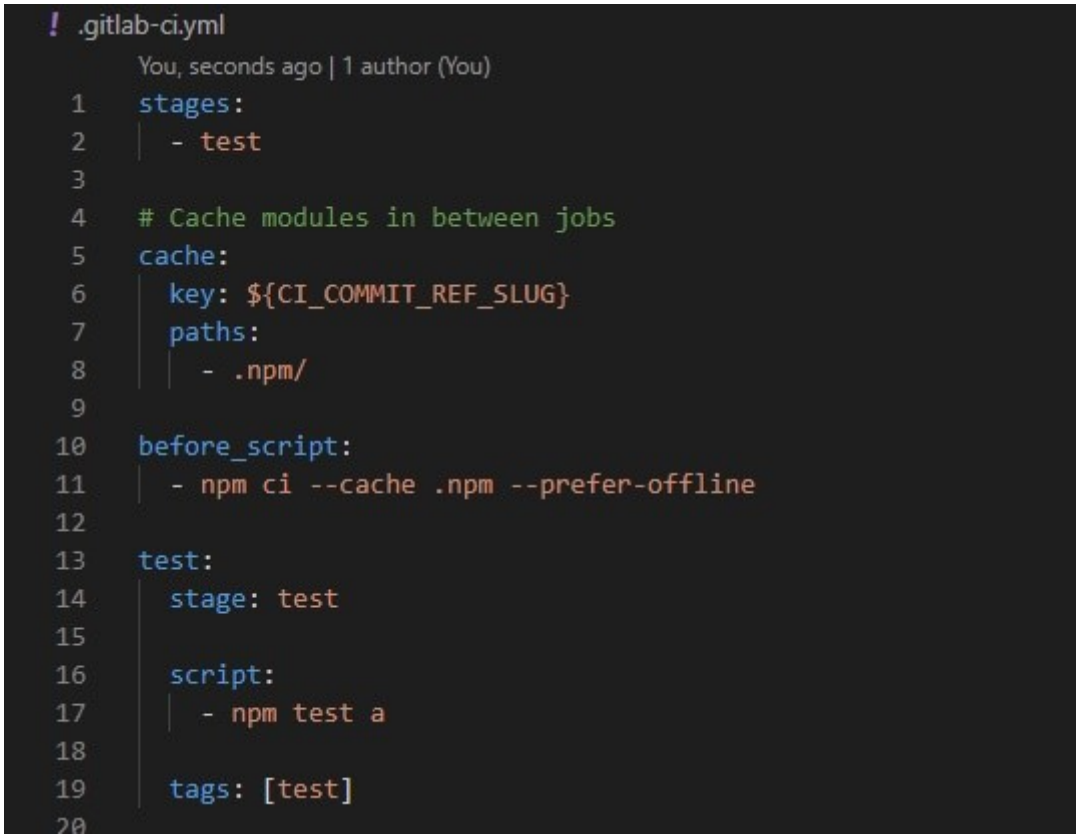


```
PASS src/__tests__/App.test.js
  Button
    √ should be defined (2 ms)
    √ should render correctly (4 ms)
    √ renders without crashing (12 ms)

----------------------------|---------|----------|---------|---------|-------------------
File                        | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------------------------|---------|----------|---------|---------|-------------------
All files                   |    8.96 |     1.19 |    3.06 |    9.29 |
 src                        |       5 |        0 |       0 |       5 |
  Register-serviceworker.js |       0 |        0 |       0 |       0 | 13-138
  firebase.js               |     100 |      100 |     100 |     100 |
  index.js                  |       0 |      100 |     100 |       0 | 9-19
  service-worker.js         |       0 |        0 |       0 |       0 | 16-68
 src/components             |    7.84 |        0 |    2.82 |    8.25 |
  Alert.js                  |      20 |        0 |       0 |      20 | 4-31
  App.js                    |     100 |      100 |     100 |     100 |
  BookingInterface.js       |    2.13 |        0 |       0 |    2.33 | 11-176
  Dashboard.js              |       0 |      100 |       0 |       0 | 13-57
  Datepicker.js             |       0 |        0 |       0 |       0 | 8-16
  Forgotpassword.js         |       0 |      100 |       0 |       0 | 9-45
  Login.js                  |      50 |      100 |    12.5 |      50 | 20,24,28-35,58-67
  Navbar.js                 |       0 |        0 |       0 |       0 | 10-29
  PrivateRoute.js           |       0 |        0 |       0 |       0 | 6-13
  Register.js               |    3.33 |        0 |       0 |    3.33 | 10-104
  SeatBooking.js            |   10.71 |        0 |       0 |   11.11 | 16-141
  SeatSelection.js          |       0 |        0 |       0 |       0 | 9-90
  SidebarData.js            |     100 |      100 |     100 |     100 |
 src/contexts               |      40 |       50 |    12.5 |      40 |
  AuthContext.js            |      40 |       50 |    12.5 |      40 | 7,15-27,39-43
----------------------------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   1 passed, 1 total
Time:        6.329 s
Ran all test suites matching /a/i.
```

Kuva 15. Www-sovellusprojektin testitapausten ja kattavuusraportin tulokset.

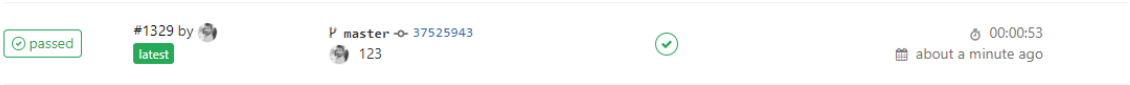
9.3 Pipelinen pystyttäminen projektiin

Seuraavana toimenpiteenä on testitapausten automatisointi www-sovellusprojektille, luomalla CI/CD- pipeline projektiin. Luodaan projektin juureen uusi tiedosto nimeltään gitlab-ci.yml. Tämä tiedosto pitää sisällään CI/CD konfiguraation kuvan 22 osoittamalla tavalla.



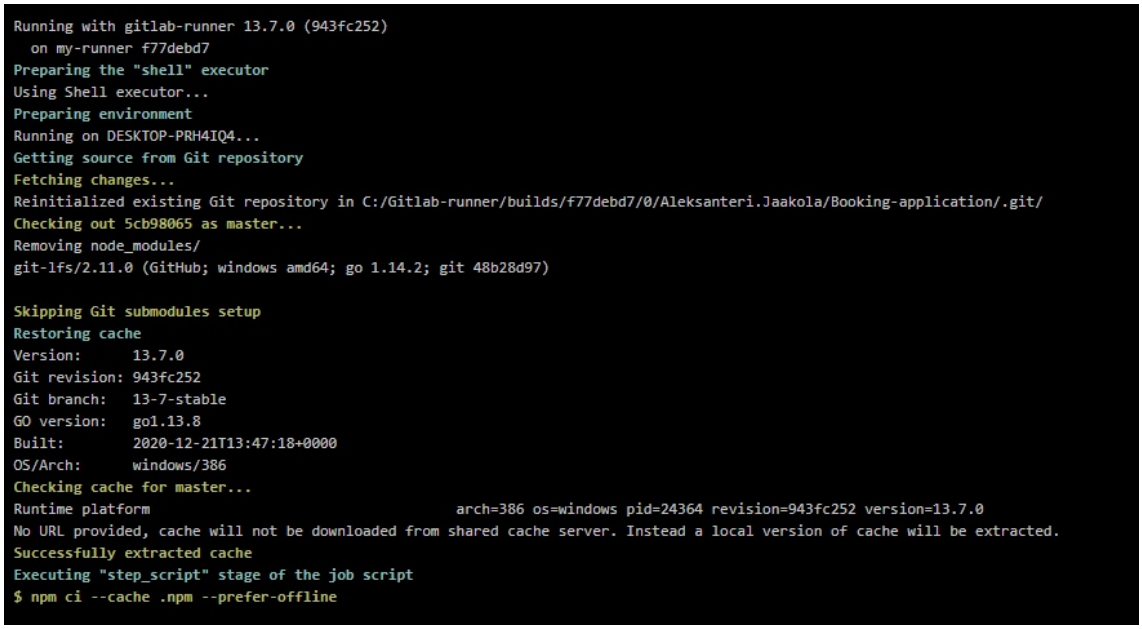
Kuva 16. Tiedoston gitlab-ci.yml konfiguraatio.

Navigoimalla Gitlabiin selaimesta ja sieltä navigoimalla kyseiseen projektiin sekä valitsemalla **CI/CD > Pipelines**. Kuvasta 23 nähdään, että työ on onnistuneesti suoritettu ja testitapausten suorittamisen yhteydessä ei virheitä ilmennyt.

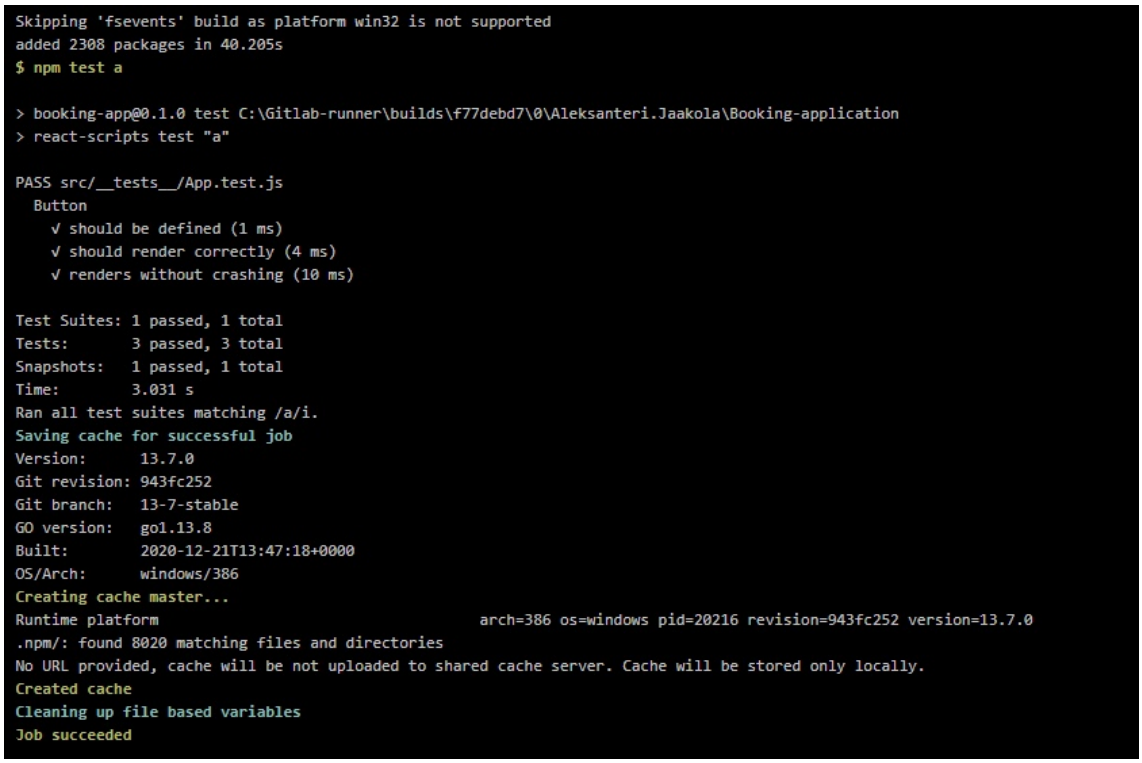


Kuva 17. Pipelinen tilanne katsottuna GitLabin käyttöliittymästä.

Jos halutaan lisää tietoa kyseisestä työstä, voidaan klikata kohtaa **passed > test** ja avautuu sieltä laajempi näkymä mitä oikeastaan tapahtui kyseisen pipelinen aikana. Kuvassa 24 haetaan uusimmat muutokset www-sovellusprojektista ja poistetaan node_modules kansio, jotta asennus voidaan aloittaa puhtaasti. Tämän jälkeen ohitetaan alimoduuleiden asentaminen, koska sitä ei konfiguraatiossa ole määritelty. Sen jälkeen palautetaan välimuisti, joka nopeuttaa pipelinen suorittamista.



Kuva 18. Mitä spesific runner tekee työn aikana.



Kuva 19. Testitapausten suorittaminen ja uuden välimuistin luominen

10 LOPPUPÄÄTELMÄT

Opinnäytetyön tarkoituksena oli selvittää mitä ohjelmistotestaaminen ja testiautomaatio on sekä miten nämä liittyvät olennaisesti ohjelmistokehitykseen. Www-sovellusprojektin myötä on pääteltävissä, että ohjelmistotestaaminen on tärkeää ohjelmistokehityksessä, kun havaitaan virheet ajoissa ja korjataan ne. Näin ollen ohjelmistokehityksessä pystytään käyttämään aikaa monimutkaisempien ongelmien ratkaisemiseen ja varmistamaan sen, että täytetään asiakkaan vaatimukset sovelluksesta tai ohjelmistosta.

Oikein ajoitettuna ja kohdistettuna testiautomaatio parantaa ohjelmistotestaamista. Vaikkakin testiautomaatio ei täysin korvaa esimerkiksi manuaalista testausta, vähentää se siinä tapahtuvia inhimillisiä virheitä. Automatisoimalla ohjelmistotestaaminen säästetään aikaan, jolloin siellä menetetty aika voidaan käyttää paremmin monimutkaisempien ongelmien ratkaisemiseen.

Testiautomaation tekeminen on työlästä ja vaatii investointeja, mutta kun se on tehty oikein maksaa testiautomaatio sen takaisin pitkällä aikavälillä. Tällä tavoin nopeutetaan ohjelmistokehityksen vaiheita ja saadaan palautetta toiminnallisuuksista nopeammin. Tämän lisäksi ohjelmistokehittäjät voivat käyttää aikansa uusien ominaisuuksien

Ohjelmistotestaaminen ei näy pelkästään siinä, että sovellus tai ohjelmisto täyttää asiakkaan vaatimukset tai vähentää siinä ilmeneviä virheitä, vaan tällä tavoin saadaan myös varmistus siitä, että koodikantaan ei pääse testaamatonta ohjelmistokoodia, joka voi haitallisesti vaikuttaa joihinkin muihin toiminallisuuksiin.

Ohjelmistotestaamisen ja testiautomaation tekeminen heti oikein projektin alusta saakka vähentää projektin teknistä velkaa, kun mietitään oikeita ratkaisuja heti alusta saakka. On paljon helpompaa heti projektin alusta unohtaa ohjelmistotestaaminen ja tehdä nopeita ratkaisuja, kun aikataulu on kireä ja sovellus tai ohjelmisto on saatava tuotantoon nopeasti, mutta tekemällä näin kasvatetaan vain teknistä velkaa, kun ei mietitä pitkän aikavälin hyötyjä.

Minulla oli tietyt ennakko-odotukset tuloksien suhteen. Tiesin etukäteen ohjelmistotestaamisen tärkeyden ohjelmistokehityksessä, mutta testitapausten kirjoittamisen hitaus ja työläisyys yllätti minut täysin. Ohjelmistotestauksen suunnittelu ja testitapausten rajaus sekä niiden toteuttaminen on työlästä ja aikaa vievää. Tämä on hyvä asia, koska tällä tavoin saadaan varmuus, että sovellus täyttää täsmälleen sille asetetut vaatimukset.

11 LÄHTEET

Guru99. [Online] [Viitattu: 5. 4 2021.] https://www.guru99.com/levels-of-testing.html.

Guru99 Tech Pvt Ltd. Guru99. [Online] [Viitattu: 6. 4 2021.] https://www.guru99.com/black-box-testing.html.

Aebersold, Kirsten. SmartBear Software. [Online] [Viitattu: 9. 4 2021.] https://smartbear.com/learn/automated-testing/test-automation-frameworks/.

Balajee, Nanduri. 2020. Medium.com. [Online] 7. Tammikuu 2020. [Viitattu: 22. 4 2021.] https://medium.com/@nanduribalajee/what-is-ci-cd-pipeline-e2f25db99bbe.

Bartlett, Jake. 2018. TestLodge. [Online] 10. 10 2018. [Viitattu: 6. 4 2021.] https://blog.testlodge.com/what-is-grey-box-testing/.

Brown, Taz. 2020. [Online] 9. Kesäkuu 2020. [Viitattu: 10. 4 2021.] https://opensource.com/article/20/7/automation-testing-cicd.

Clarion. Clariontech. [Online] [Viitattu: 9. 4 2021.] https://www.clariontech.com/blog/what-are-test-automation-frameworks-and-types.

Docs, MDN Web. 2021. developer.mozilla. [Online] 20. huhtikuu 2021. [Viitattu: 29. 4 2021.] https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps.

Freeman, John. 2019. Blogi. [Online] 22. Maaliskuu 2019. [Viitattu: 13. 4 2021.] https://jfreeman.dev/blog/2019/03/22/understanding-gitlab-runner/.

Freyja. 2017. *Raygun.com.* [Online] Raygun, 22. Maaliskuu 2017. [Viitattu: 7. 4 2021.] https://raygun.com/blog/cost-of-software-errors/.

Gitlab. *Gitlab sivusto.* [Online] [Viitattu: 12. 4 2021.] https://docs.gitlab.com/ee/ci/jobs/.

—. Gitlab. [Online] Gitlab.[Viitattu: 10. 4 2021.] https://docs.gitlab.com/ee/ci/pipelines/.

Guru99. *Guru99 sivusto.* [Online] [Viitattu: 5. 4 2021.] https://www.guru99.com/levels-of-testing.html.

—. Guru99 sivusto. [Online] [Viitattu: 5. 4 2021.] https://www.guru99.com/v-model-software-testing.html.

Healey, Fred Meier & James R. 2914. usatoday.com. [Online] 26. 3 2914. [Viitattu: 18. 5 2021.] https://eu.usatoday.com/story/money/cars/2014/03/26/nissan-recall-air-bags-altima-sentra-leaf/6902937/.

Hoogenraad, Wim. 2020. ITpedia. [Online] 29. 8 2020. [Viitattu: 6. 4 2021.] https://fi.itpedia.nl/2019/01/23/black-box-testing-software-op-de-pijnbank/.

https://ohjelmointi-19.mooc.fi/. [Online] Helsingin yliopisto.[Viitattu: 4. 5 2021.] https://ohjelmointi-19.mooc.fi/osa-5/6-testauksen-alkeet-jatkuvat.

Ilze. 2018. testdevlab.com. [Online] 3. 7 2018. [Viitattu: 18. 5 2021.] https://www.testdevlab.com/blog/2018/07/importance-of-software-testing/.

John Potter. npm trends. [Online] [Viitattu: 9. 4 2021.] https://www.npmtrends.com/.

Johnson, Patricia. 2020. WhiteSource Software. [Online] 12. 11 2020. [Viitattu: 6. 4 2021.] https://resources.whitesourcesoftware.com/blog-whitesource/white-box-testing.

Katara, Mika. 2011. Tampereen teknillinen yliopisto. [Online] 2011. [Viitattu: 5. 4 2021.] http://www.cs.tut.fi/~tie21201/s2011/luennot/OHJ-3060_2011_110-170.pdf.

Kinnunen, Hanna-Mari. 2018. *Arter.fi.* [Online] Arter Oy, 4. 9 2018. [Viitattu: 5. 4 2021.] https://www.arter.fi/ohjelmistojen-laadunvarmistuksen-abc/.

Kothari, Abhishek. 2021. Geekflare. [Online] 26. maaliskuu 2021. [Viitattu: 9. 4 2021.] https://geekflare.com/javascript-unit-testing/.

Lindley, Cody. React Enlightenment. [Online] [Viitattu: 9. 4 2021.] https://www.reactenlightenment.com/react-state/8.1.html.

Lutkevich, Ben. 2019. TechTarget. [Online] Joulukuu 2019. [Viitattu: 9. 4 2021.] https://searchnetworking.techtarget.com/definition/asynchronous.

Marek, Barbara &. 2020. merixstudio. [Online] 18. 5 2020. [Viitattu: 29. 4 2021.] https://www.merixstudio.com/blog/introduction-storybook-react/.

Military.com. 2016. foxnews.com. [Online] 3. 3 2016. [Viitattu: 18. 5 2021.] https://www.foxnews.com/tech/software-glitch-causes-f-35-to-incorrectly-detect-targets-in-formation.

Närhi, Eemi. 2019. theseus. [Online] 2019. [Viitattu: 19. Toukokuu 2021.] https://www.theseus.fi/bitstream/handle/10024/262618/Narhi_Eemi.pdf?sequence=2&isAllowed=y.

Nijhawan, Rakesh. 2012. testingbasicinterviewquestions.blogspot.com. [Online] maaliskuu 2012. [Viitattu: 7. 4 2021.] https://testingbasicinterviewquestions.blogspot.com/2012/03/what-are-different-goals-of-software.html.

Performance Lab. 2008-2021. [Online] Performance Lab, 2008-2021. [Viitattu: 6. 4 2021.] https://performancelabus.com/software-testing-importance-sdlc/.

Pittet, Sten. Atlassian.com. [Online] [Viitattu: 25. 4 2021.] https://www.atlassian.com/continuous-delivery/software-testing/code-coverage.

Pollari, Jukka. 2014. theseus. [Online] Joulukuu 2014. [Viitattu: 19. toukokuu 2021.] https://www.theseus.fi/bitstream/handle/10024/85400/Pollari_Jukka.pdf?sequence=1&isAllowed=y.

Ques10. 2019. Ques10. [Online] Ques10, Syyskuu 2019. [Viitattu: 7. 4 2021.] https://www.ques10.com/p/48598/goals-of-software-testing-1/.

Rajkumar. 2021. softwaretestingmaterial. [Online] 9. 1 2021. [Viitattu: 28. 4 2021.] https://www.softwaretestingmaterial.com/software-testing/.

Ravishankar, Avinash. 2020. Gitconnected sivusto. [Online] 29. Kesäkuu 2020. [Viitattu: 12. 4 2021.] https://levelup.gitconnected.com/a-gentle-introduction-to-gitlab-ci-cd-4be1d3ea7f19.

Raza, Muhammad. 2020. bmc.com. [Online] 2. Joulukuu 2020. [Viitattu: 19. Toukokuu 2021.] https://www.bmc.com/blogs/testing-automation/.

RedHat. 2021. [Online] Red Hat, Inc, 2021. [Viitattu: 10. 4 2021.] https://www.redhat.com/en/topics/devops/what-is-ci-cd.

Richardson, Leland. [Online] Airbnb, Inc.[Viitattu: 22. 4 2021.] https://airbnb.io/projects/enzyme/.

Ryzycki, Marcin. 2018. Medium. [Online] 26. Kesäkuu 2018. [Viitattu: 12. 4 2021.] https://medium.com/@ryzmen/gitlab-fast-pipelines-stages-jobs-c51c829b9aa1.

Sameer, Shaik. 2020. [Online] 2020. [Viitattu: 27. 4 2021.] https://www.westagilelabs.com/blog/why-is-software-testing-and-qa-important-for-any-business/.

Simmons, TJ. 2020. Plutora. [Online] 12. Marraskuu 2020. [Viitattu: 10. 4 2021.] https://www.plutora.com/blog/understanding-ci-cd-pipeline.

Sing, Gursimran. 2020. [Online] XenonStack, 27. Tammikuu 2020. [Viitattu: 10. 4 2021.] https://www.xenonstack.com/blog/continuous-integration-and-continuous-delivery/.

Singh, Ranvir. 2018. Linuxhint. [Online] 2018. [Viitattu: 21. 4 2021.] https://linuxhint.com/gitlab_runner_gitlab_ci/.

Storybook. Storybook. [Online] [Viitattu: 9. 4 2021.] https://storybook.js.org/docs/react/workflows/testing-with-storybook.

Stringfellow, Angela. 2017. Stackify. [Online] 25. syyskuu 2017. [Viitattu: 9. 4 2021.] https://stackify.com/when-to-use-asynchronous-programming/.

Techopedia. *Techopedia sivusto.* [Online] [Viitattu: 10. 4 2021.] https://www.techopedia.com/definition/540/compile.

Testim. 2021. Tesim. [Online] 17. Helmikuu 2021. [Viitattu: 9. 4 2021.] https://www.testim.io/blog/best-unit-testing-framework-for-javascript/.

Thelin, Ryan. 2020. educative.io. [Online] 20. lokakuu 2020. [Viitattu: 18. toukokuu 2021.] https://www.educative.io/blog/software-testing-types-101?aid=5082902844932096&utm_source=google&utm_medium=cpc&utm_campaign=blog-dynamic&gclid=CjwKCAjwy42FBhB2EiwAJY0yQh0ZPCXXkqBLlZBTS5b40xCxfoC7guO9hvM3fx3t3k_vivpfIa7gTxoC4qkQAvD_BwE.

Tryna.com. tryqa.com. [Online] [Viitattu: 28. 4 2021.] http://tryqa.com/what-is-integration-testing/.

Unadkat, Jash. 2019. BrowserStack. [Online] 23. Lokakuu 2019. [Viitattu: 9. 4 2021.] https://www.browserstack.com/guide/top-javascript-testing-frameworks.

Vala Group. Itewiki. [Online] [Viitattu: 5. 4 2021.] https://www.itewiki.fi/opas/laadunvarmistus-ja-ohjelmistotestaus/.

Vertics Oy. Vertics.co. [Online] Vertics Oy.[Viitattu: 5. 4 2021.] https://vertics.co/ohjelmistotestauksen-perusteet/.

Wikipedia. 2021. Wikipedia. [Online] 22. 1 2021. [Viitattu: 29. 4 2021.] https://en.wikipedia.org/wiki/Jest_(JavaScript_framework).