Tampereen ammattikorkeakoulu

Verkkosovelluksen päästä päähän -testaaminen

Panu Soisenniemi

OPINNÄYTETYÖ
Lokakuu 2021

Tieto- ja viestintätekniikka
Ohjelmistotekniikka

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tieto- ja viestintätekniikka
Ohjelmistotekniikka

SOISENNIEMI, PANU:
Verkkosovelluksen päästä päähän -testaaminen

Opinnäytetyö 38 sivua, joista liitteitä 3 sivua
Lokakuu 2021

Ohjelmistojen tarve ja määrä yhteiskunnassa kasvavat digitalisaation myötä jatkuvasti. Ohjelmistot ovat entistäkin suurempia ja monimutkaisempia, joten testaaminen on tärkeämmässä roolissa kuin koskaan ennen.

Työn tarkoituksena oli tehdä kattava selvitys testien toteuttamisesta ja liittämisestä osaksi jatkuvaa integraatiota sekä myös taustoittaa ohjelmistotestausta ja sen merkitystä osana ohjelmistokehityksen prosesseja.

Opinnäytetyössä toteutettiin työtä varten tehtyyn nykyaikaiseen verkkosovellukseen päästä päähän -testit sekä liitettiin ne osaksi jatkuvan integraation putkea. Lisäksi tehtiin selvitys ohjelmistotestauksesta sekä sen merkityksestä osana ohjelmistokehityksen prosessia. Päästä päähän -testaamisen työkaluksi valittiin Cypress ja jatkuva integraatio toteutettiin Gitlabin tarjoamaan ympäristöön.

Asiasanat: ohjelmistotestaus, päästä päähän -testaus, Cypress, jatkuva integraatio

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in ICT Engineering
Software Engineering

SOISENNIEMI, PANU:
Web Application End-to-End testing

Bachelor's thesis 38 pages, appendices 3 pages
October 2021

Digitization has increased the need for software everywhere. Software is now more extensive and complex than before, so the importance of testing has also increased.

In the thesis, end-to-end tests were created for a modern web application which was created for the thesis. Running of these tests was implemented to be part of a continuous integration pipeline. In addition, the thesis reviews software testing in general and its importance as part of the software development processes. Cypress was chosen as the tool for end-to-end testing and continuous integration environment was provided by Gitlab.

The purpose of the work was to make a comprehensive study of the process of implementing end-to-end tests in a modern web application and integrating tests to be part of continuous integration. An another objective was to review software testing in general and its importance as part of the software development processes.

Key words: software testing, end-to-end testing, Cypress, continuous integration

SISÄLLYS

LYHENTEET JA TERMIT

Annotaatio	Tapa määritellä metadataa Java-ohjelmointikielessä.
CI	Jatkuva integraatio (eng. Continuous Integration).
DOM	Dokumenttioliomalli (eng. Document Object Model) on tapa kuvata merkintäkielillä tehtyjen dokumenttien rakennetta puuna, jota pystytään manipuloimaan esimerkiksi JavaScriptillä.
Haara	Versionhallinnassa toisistaan erillään olevia kopioita, joihin voidaan tehdä muutoksia. Haaroja voidaan luoda olemassa olevista haaroista ja yhdistää takaisin joko alkuperäiseen tai johonkin toiseen haaraan.
HTML	Hypertext Markup Language. Verkkosivujen merkintäkieli.
HTTP	Hypertext Transfer Protocol. Protokolla, jota käytetään tiedostojen siirtoon selaimen ja palvelimen välillä.
JavaScript	Käytetyimpiä verkkosivustojen ohjelmointikieliä. Ajetaan suoraan selaimessa. Olennainen osa nykyaikaisia verkkosovelluksia.
Kirjasto	Kokoelma aliohjelmia, luokkia ja/tai ohjelmia, joita hyödynnetään ohjelmiston modulaarisessa kehittämisessä.
NodeJS	Avoimeen lähdekoodin perustuva alustariippumaton JavaScript-koodin ajoympäristö.
Ohjelmistokehys	Ohjelmistokehityksessä käytettävä apuväline, joka muodostaa rungon sen päälle rakennettavalle ohjelmistolle.
TypeScript	Microsoftin luoma ja ylläpitämä ohjelmointikieli, joka laajentaa JavaScriptiä mm. tyypityksillä. Käännetään ennen ajamista JavaScriptiksi.
YAML	YAML Ain’t Markup Language. Usein konfiguraatiotiedostoissa käytetty merkintäkieli.
Yhdistämispyyntö	Pyyntöä yhdistää versionhallinnan haara toiseen haaraan kutsutaan yhdistämispyynnöksi (eng. merge request).

1 JOHDANTO

Ohjelmistojen tarve ja määrä yhteiskunnassa kasvaa digitalisaation myötä jatkuvasti. Tänä päivänä ohjelmistot ovat entistäkin suurempia ja monimutkaisempia, joten niiden laadun ja luotettavuuden varmistaminen on suuremmassa osassa kuin koskaan ennen.

Ohjelmistontestaus on oleellinen osa ohjelmistokehitystä, jolla pyritään varmistamaan, että ohjelmisto toimii halutulla tavalla. Testauksen pääasiallinen tarkoitus on löytää ohjelmistovirheitä ja varmistaa ohjelmiston oikea toiminta. Ohjelmistotestauksen tarve kasvaa ohjelmistojen kasvaessa isommiksi kokonaisuuksiksi.

Työn tavoitteena on toteuttaa selvitys siitä, kuinka toteuttaa päästä päähän -testit sekä ajaa ne osana jatkuvan integraation putkea. Työssä tullaan taustoittamaan yleisesti ohjelmistotestausta sekä sen merkitystä osana ohjelmistokehitystä. Työssä tehdään kattava selvitys päästä päähän -testien toteuttamisesta Cypress-työkalulla sekä sen liittämisestä osaksi Gitlabin jatkuvan integraation putkea.

Luku 2 käsittelee ohjelmistotestausta. Luvussa 3 käydään läpi testattava kohdesovellus sekä testauksen suunnitelmaa. Luvut 4 ja 5 sisältävät työn käytännön osuuden. Lopuksi on vielä luvussa 6 koottu työn tuloksista yhteenveto ja pohdintaa.

2 OHJELMISTOTESTAUS

2.1 Testaaminen yleisesti

Ohjelmiston testaaminen on prosessi, jossa varmistetaan sekä vahvistetaan että ohjelmisto täyttää sille asetetut liiketoiminnalliset ja tekniset vaatimukset sekä toimii odotetulla tavalla. Testaamisella tunnistetaan ohjelmistokoodissa olevia vikoja, puutteita sekä virheitä. (Bentley J., Bank W. & NC C. n. d.)

Testaamisella halutaan tuoda ohjelmistoon lisäarvoa parantamalla sen laatua ja luotettavuutta. Luotettavuutta voidaan parantaa etsimällä ja korjaamalla ohjelmistovirheitä, mistä johtuen ohjelmiston testauksessa ei ole tärkeimpänä tavoitteena todeta ohjelman toimivan oikein, vaan yrittää löytää mahdollisia virheitä. (Myers, Sandler & Badgett 2012, 6.)

2.2 Testauksen osa-alueita

2.2.1 Toiminnallinen testaus

Toiminnallisessa testaamisessa testataan ohjelmiston toiminnallisia vaatimuksia. Testit suoritetaan yleensä manuaalisesti siten että testaaja suorittaa ennalta määriteltyjä testitapauksia ja vertaa lopputuloksia odotettuihin lopputuloksiin. Toiminnallisella testaamisella varmistetaan, että ohjelmisto noudattaa sille asetettuja teknisiä ja liiketoiminnallisia vaatimuksia. (Utor 2020.)

2.2.2 Ei-toiminnallinen testaus

Ei-toiminnallisessa testaamisessa testataan niitä asioita, joita toiminnallinen testaaminen ei kata. Näihin kuuluu mm. ohjelmiston suorituskyvyn, skaalautuvuuteen ja turvallisuuteen liittyvä testaaminen. Testaaminen tapahtuu yleensä automaattisesti hyödyntäen siihen tarkoitettuja työkaluja. (Utor 2020.)

2.2.3 Regressiotestaus

Regressiotestauksessa varmistetaan, että ohjelmistoon tehdyt muutokset eivät riko jo toimiviksi todennettuja ominaisuuksia eikä jo aikaisemmin löydettyjä ja korjattuja virheitä. (Nancy J. W. 1999.)

2.2.4 Käytettävyystestaus

Käytettävyystestauksessa tutkitaan kuinka loppukäyttäjät käyttävät ohjelmistoa, minkä tuloksien perusteella havaitaan mahdollisia ongelmia ohjelmistossa. Testausta tulisi tehdä jo ohjelmiston suunnittelu- ja kehitysvaiheessa, jotta oikeaa palautetta käyttäjiltä saadaan mahdollisimman ajoissa, jolloin korjausliikkeet ovat vielä helppo toteuttaa. (Agenda 2019.)

2.2.5 Tutkiva testaus

Tutkivassa testauksessa ei hyödynnetä ennalta määritettyjä testitapauksia, vaan testaaminen tapahtuu lennosta tehdyillä testitapauksien luomisella ja suorittamisella samalla jatkuvasti oppien uutta ohjelmistosta. Testaamiselle voidaan määritellä tietty ohjelmiston osa-alue, johon keskitytään. Tutkivan testauksen tehokkuus riippuu pitkälti testaajan kokemuksesta testattavan ohjelmiston parissa. (Itkonen J. & Rautiainen K. 2005.)

2.3 Manuaalinen testaaminen

Manuaalisessa testauksessa testaaja asettuu loppukäyttäjän rooliin ja testaa ohjelmistoa käyttöliittymästä käsin. Testaaja usein noudattaa testaussuunnitelmaa, jossa on määritetty testitapaukset, joiden avulla voidaan varmistaa, että ohjelmisto toimii oikein. Ohjelmiston toiminnallinen testaus sekä käytettävyystestaus suoritetaan usein manuaalisesti. Manuaalinen testaus vaatii paljon aikaa sekä resursseja mistä johtuen varsinkin teknisellä tasolla tapahtuvat testitapaukset pyritään automatisoimaan.

2.4 Automaattinen testaaminen

Automaattisessa testaamisessa hyödynnetään erilaisia testaustyökaluja, jotka suorittavat työvaiheita, joita muuten tehtäisiin manuaalisesti. Työkalut mahdollistavat mm. erilaisten testausympäristöjen pystyttämisen, testitapauksien suorittamisen, nopeamman ja laajemman testaamisen sekä erilaisten raporttien luomisen. Erilaisia työkaluja hyödyntämällä on mahdollista parantaa ohjelmisto ylläpidettävyyttä, laatua sekä luotettavuutta. Automaattista testausta hyödynnytetään varsinkin jatkuvan integraation putkessa. Automaattisella ja jatkuvalla testaamisella tehostetaan kehitys- ja julkaisuprosessia sekä vähennetään paikalliseen testaukseen kuluvaa aikaa.

2.5 Päästä päähän -testaaminen

Päästä päähän -testeissä testataan ohjelmiston toiminnallisuuksia käyttöliittymästä käsin siten että koko ohjelmisto on käynnissä. Testaaminen on hitaampaa, koska ohjelmisto pitää kääntää lähdekoodista, pystyttää testiympäristö sekä käynnistää käyttöliittymä. Jos kyseessä on esimerkiksi verkkosovellus, käyttöliittymää käytetään verkkoselaimella, jota käytetään joko manuaalisesti tai sitten päästä päähän -testaamisen työkalu ohjaa sitä automaattisesti siten että se matkii ihminen toimintaa.

Päästä päähän -testit eivät korvaa muita testaamisen tapoja, mutta laajentaa niitä ja auttaa löytämään esimerkiksi käyttöjärjestelmästä riippuvaisia ongelmia. Testien kirjoittaminen on hankalaa ja työlästä ylläpitää sekä ongelmien paikantaminen voi olla hankalaa, koska testauksessa on mukana koko ohjelmisto.

Testeillä pyritään varmistamaan, että käyttöliittymä on käytettävissä mm. varmistamalla painikkeiden, lomakkeiden, linkkien ja työnkulkujen toimivuutta sekä varmistamalla, että regressiota ei pääse syntymään muutoksien yhteydessä. (Stan G. 2021.)

2.6 Päästä päähän -testaamisen työkalut

2.6.1 Selenium

Selenium on yksi suosituimmista päästä päähän -testauksen työkaluista. Seleniumia käytetään pääasiassa verkkosovellusten testaamiseen, mutta sitä voidaan käyttää myös automaatiotyökaluna. Selenium tukee useita eri selaimia niin sanottujen ajureiden avulla. (Selenium 2021.)

2.6.2 Cypress

Cypress on modernien verkkosovellusten testaamiseen kehitetty työkalu. Cypressin tavoitteena on helpottaa testausprosessia yksinkertaistamalla ympäristöjen pystystä, testien tekemistä sekä ajamista. Työkalu antaa uudenlaisia ominaisuuksia päästä päähän -testien toteuttamiseen verrattuna aikaisemmin esiteltyyn Seleniumiin. (Cypress 2021.)

2.6.3 Vertailua

Cypress ja Seleniumin välillä on merkittäviä arkkitehtuurillisia eroja. Selenium käyttää sovellusta selaimen ulkopuolelta, kun taas Cypress toimii testattavan sovelluksen sisällä. Edellä mainittu mahdollistaa Seleniumille laajan käytettävien ohjelmointikielien mahdollisuuden, kun taas Cypress tukee ainoastaan JavaScriptia. Cypressin kohderyhmänä ovat modernien JavaScript pohjaisten verkkosovellusten testaus, kun taas Seleniumia voidaan käyttää monipuolisemmin myös muissa sovelluksissa.

Koska Cypress suoritetaan samassa yhteydessä kuin testattava sovellus, tämä tarkoittaa sitä, että Cypressilla pääsee käsiksi sovelluksen sisäiseen tilaan. Sisäiseen tilaan kiinni pääseminen mahdollistaa sen, että Cypress voi tarkkailla ja manipuloida verkkoliikennettä, mikä mahdollistaa aikaisempaa monimutkaisempien testien toteuttamisen. (Cypress 2021.)

2.7 Jatkuvan integroinnin merkitys

Ohjelmistonkehityksessä on usein työskentelemässä useita kehittäjiä samanaikaisesti sovelluksen eri osien kanssa. Lähdekoodia haaroitetaan kehityksen aikana ja viedään yhteen päähaaraan. Manuaalisesti tämän tekeminen olisi työlästä sekä aikaa vievää.

Jatkuva integrointi tulee tässä avuksi. Koodimuutoksia yhdistäessä takaisin päähaaran, vahvistetaan muutokset jatkuvan integroinnin putkessa eli CI-putkessa. Putkessa voidaan suorittaa esimerkiksi sovelluksen automaattinen testaaminen eri tasoilla. Jos testauksessa esiintyy ongelmia, havaintaan virheet heti ja ne voidaan korjata nopeasti. (RedHat 2021.)

2.8 Yksisivuiset verkkosovellukset

Yksisivuiset verkkosovellukset ovat dynaamisia verkkosivuja, jotka rakentuvat yhden HTML-dokumentin ympärille. Sivun päivittäminen tapahtuu JavaScriptin rajapinnoilla manipuloimalla dokumentin sisältöä. Tällä tavalla parannetaan verkkosivun käyttökokemusta vähentämällä sivun latauksia ja luodaan käyttäjälle dynaamisempi käyttökokemus. (MDN Web Docs 2021.)

Samalla kuitenkin siirretään logiikkaa käyttäjän selaimelle, joten testaamisessa täytyy huomioida eri selaimet. Tämä hankaloittaa testaamista, koska sovelluksen toimivuus tarvitsee testata useilla eri selaimilla. Yksisivuiset verkkosovellukset usein lataavat uutta tietoa asynkronisesti ja manipuloivat HTML-dokumentin rakennetta käyttäjän toimien perusteella. Tämä vaikeuttaa testaamista, koska tarvittavien pyyntöjen tulee olla valmiita ennen kuin testit voidaan suorittaa sekä täytyy varmistua, että näkymä on valmis testattavaksi.

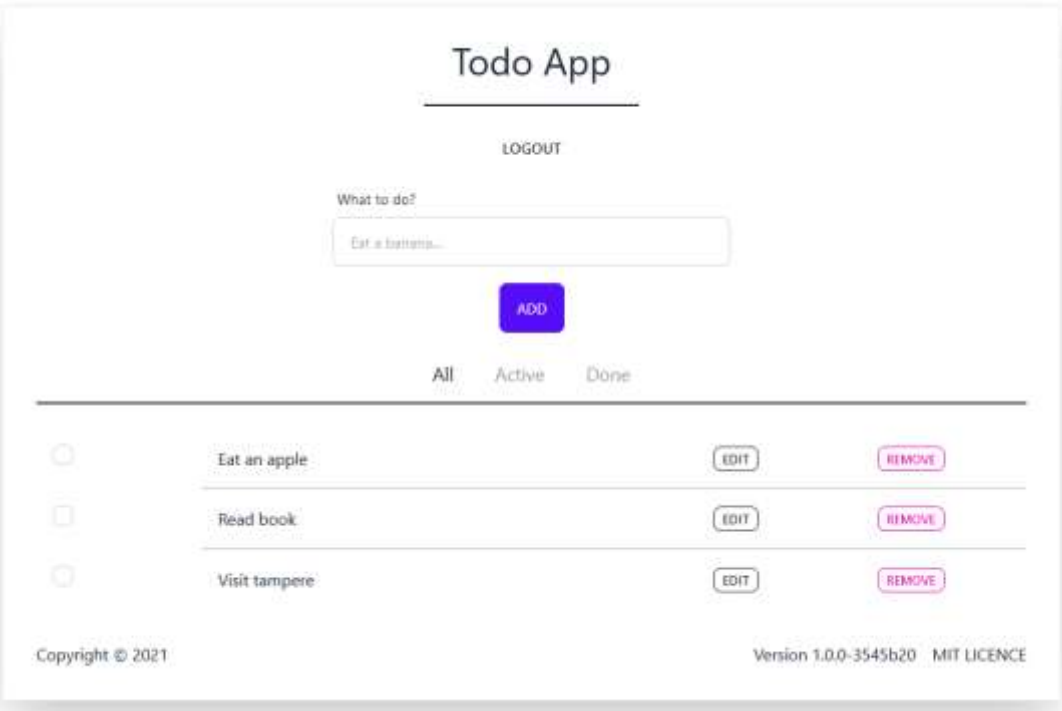
Yksisivuisten verkkosovellusten päästä päähän -testaamiseen on kehitetty työkaluja, kuten kappaleessa 2.6 esitellyt Selenium ja Cypress. Näissä työkaluissa on huomioitu edellä mainittuja ongelmia ja toteutettu ratkaisuja, joilla näitä voidaan ratkaista.

3 PÄÄSTÄ PÄÄHÄN -TESTIEN SUUNNITTELU

3.1 Kohdesovellus

Työtä varten toteutettiin yksisivuinen verkkosovellus. Käyttöliittymä toteutettiin TypeScriptillä käyttäen käyttöliittymäkirjastona Facebookin Reactia. Rajapinta toteutettiin Javalla Spring Boot -kehyksen päälle. Käyttöliittymä ja rajapinta paketoidaan yhdeksi .war-tiedostoksi, josta sovellusta voidaan ajaa. Tietokantana tuetaan PostgreSQL:n versiota 13. Sovelluksen arkkitehtuuria on kuvattu liitteessä 1.

Kohdesovellus on tarkoituksella toteutettu yksinkertaiseksi, koska työn tavoitteena on päästä päähän -testien suunnittelu ja toteuttaminen sekä niiden ajaminen CI-putkessa eli jatkuvan integraation putkessa. Kohdesovelluksessa on toimintoja kuten käyttäjän luominen, kirjautuminen ja käyttäjäkohtainen tehtävälista. Kuvassa 1 on esitetty kuvakaappaus kohdesovelluksen käyttöliittymästä.



KUVA 1. Kuvakaappaus käyttäjän tehtävälistasta kohdesovelluksessa

Sovelluksen käyttöliittymän ja rajapinnan lähdekoodit ovat samassa versionhallinnassa sekä sovellukselle on jo toteutettu CI-putki, jossa ajetaan sovelluksen yksikkö- ja integraatiotestit, paketoidaan .war-tiedosto sekä luodaan Docker-levykuva.

3.2 Testaussuunnitelma

Kohdesovellukseen on jo toteutettu yksikkö- ja integraatiotestejä, joita ajetaan osana CI-putkea. Tässä työssä toteutetaan kohdesovellukseen yksi päästä päähän -testi sekä lisätään päästä päähän -testien ajaminen osaksi CI-putkea. Testaussuunnitelma on esitetty liitteessä 2. Toteutettava päästä päähän -testi on uuden käyttäjän luomisen toiminto. Kyseinen toiminnon testaus toteutetaan siten että se täyttää testaussuunnitelmassa määritellyn toiminnan.

Päästä päähän -testien ajaminen osana CI-putkea saattaa kestää useita minuutteja, laajemmissa sovelluksissa jopa kymmeniä minuutteja. Toteutetaan sovellukseen mahdollisuus ajaa päästä päähän -testit lokaalisti aina tarvittaessa. CI-putkeen lisätessä asetetaan päästä päähän -testit ajettavaksi vain, jos haarasta on luotu yhdistämispyyntö versionhallinnan päähaaraan tai jos kyseessä ovat muutokset päähaaraan. Lisäksi toteutetaan päästä päähän -testeille ajastettu ajo sovelluksen päähaaraa vasten, jotta mahdolliset sovellukseen liittymättömät, esimerkiksi selainten virheet tulevat ilmi.

3.3 Käytettävä päästä päähän -testaamisen työkalu

Kappaleessa 2.6.3 käytiin lävitse Selenium- ja Cypress-testaustyökalujen eroja. Cypress on suunniteltu arkkitehtuurillisesti modernien yksisivuisten verkkosovellusten testaamiseen. Cypress on uusi ja erilainen tulokas päästä päähän -testaamisen saralla ja siksi se valittiin työn päästä päähän -testaamisen työkaluksi.

4 PÄÄSTÄ PÄÄHÄN -TESTIEN TOTEUTUS

4.1 Cypress

Cypress on JavaScript kirjasto ja on saatavilla NodeJS:n pakettienhallinnasta. Testit kirjoitetaan JavaScriptillä tai TypeScriptillä. Kohdesovellus on kirjoitettu TypeScriptillä, joten käytetään sitä myös testien kirjoittamisessa.

Cypress edellyttää, että testattavan sovelluksen DOM-puusta pystytään tunnistamaan eri elementtejä. Tällä varmistetaan, että Cypress on tekemässä toimintoja oikeille elementeille. Virallisen dokumentaation mukaan on suositeltavaa, että elementtien erottelu toteutetaan käyttämällä kustomoituja attribuutteja (Cypress 2021).

Yleisimmät käyttäjän toimenpiteet on mahdollista toteuttaa Cypressin omilla funktioilla. Lisäksi on mahdollista luoda uusia Cypress-funktioita tai ylikirjoittaa jo olemassa olevia funktioita. Taulukossa 1 on listattu yleisimmät Cypress-funktiot.

TAULUKKO 1. Yleisimmät Cypress-funktiot (Cypress 2021)

Funktio	**Kuvaus**
visit()	Avaa määritellyn URL-osoitteen
get()	Hakee määritellyn DOM-elementin
click()	Klikkaa DOM-elementtiä, ketjutetaan esim. get() kanssa
type()	Kirjoittaa DOM-elementtiin, ketjutetaan esim. get() kanssa
should()	Tarkistetaan, että DOM-elementti läpäisee ehdot, ketjutetaan esim. get() kanssa
request()	Suorittaa HTTP-pyynnön määrityksien mukaan

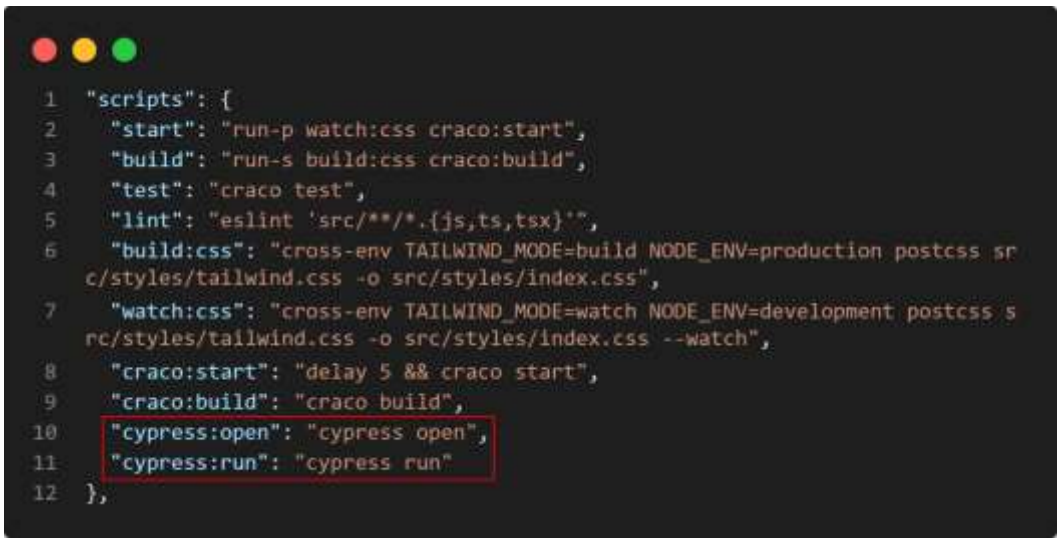
4.2 Cypressin käyttöönotto

Cypress-työkalun käyttäminen vaatii sen asentamista projektiin. NodeJS projekteissa helpointa on asentaa Cypress käyttämällä NodeJS:n pakettienhallintatyökalua kuten npm tai yarn. Kohdesovelluksessa pakettienhallintatyökaluna on käytetty Facebookin kehittämää yarn-työkalua. Cypress voidaan asentaa projektiin

komennolla ”yarn add cypress –dev”. Komento asentaa ja lisää Cypressin kehityksenaikaiseksi riippuvuudeksi projektiin. Kuvassa 2 on esitetty kohdesovelluksen kehityksenaikaiset riippuvuudet Cypressin lisäämisen jälkeen. Korostettuna on Cypressin kirjasto.

KUVA 2. Projektin kehitysajan riippuvuudet package.json tiedostossa

Cypressin asentamisen jälkeen lisätään projektin skripteihin Cypressin käynnistämiseen tarvittavat skriptit. Komennolla ”cypress open” voidaan avata Cypressin graafinen käyttöliittymä, kun taas komennolla ”cypress run” voidaan ajaa testit suoraan konsolissa. Cypress ei käynnistä testattavaa sovellusta, joten sovellus on käynnistettävä manuaalisesti. Kuvassa 3 on esitetty projektin skriptit. Korostettuna on Cypressin käytössä käytettävät skriptit.



```
"scripts": {
  "start": "run-p watch:css craco:start",
  "build": "run-s build:css craco:build",
  "test": "craco test",
  "lint": "eslint 'src/**/*.{js,ts,tsx}'",
  "build:css": "cross-env TAILWIND_MODE=build NODE_ENV=production postcss sr
c/styles/tailwind.css -o src/styles/index.css",
  "watch:css": "cross-env TAILWIND_MODE=watch NODE_ENV=development postcss s
rc/styles/tailwind.css -o src/styles/index.css --watch",
  "craco:start": "delay 5 && craco start",
  "craco:build": "craco build",
  "cypress:open": "cypress open",
  "cypress:run": "cypress run"
},
```

KUVA 3. Projektin skriptit package.json tiedostossa

Avataan Cypress komennolla ”yarn cypress:open” ja Cypress generoi kansion ”cypress” sekä useita alikansioita ja tiedostoja. Muutetaan kansioihin luodut .js-tiedostot .ts-tiedostoiksi ja lisätään ”cypress” kansion juureen TypeScriptin konfiguraatiotiedosto tsconfig.json, joka on esitetty kuvassa 4. Konfiguraatiossa laajennetaan kohdesovelluksen konfiguraatiota Cypressin tyypityksillä.

```
{
  "extends": "../tsconfig.json",
  "include": ["./*/*.ts"],
  "exclude": [],
  "compilerOptions": {
    "lib": ["es2015", "dom"],
    "types": ["cypress"],
    "isolatedModules": false,
    "allowJs": true,
    "noEmit": true
  }
}
```

KUVA 4. Cypressin tsconfig.json

Luodaan vielä Cypressin konfiguraatiotiedosto ”cypress.json” samalle tasolle kuin ”cypress” kansio. Konfiguraatiossa voidaan muokata Cypressin eri ominaisuuksia ja toimintoja. Otetaan käyttöön Cypressin mukana asennettu JUnit-raportointi. Konfiguraatio on näkyvissä kuvassa 5. Huomioitavaa on, että tiedostossa määritellyt konfiguraatiot voidaan ylikirjoittaa ympäristömuuttujilla. Kun edellä tehdyt toimenpiteet ovat tehty, voidaan alkaa kirjoittamaan päästä päähän -testejä.

KUVA 5. Cypressin cypress.json konfiguraatio

4.3 Testin toteuttaminen

Kuten aikaisemmin on jo mainittu, Cypress ei käynnistä sovellusta testejä ajaessa, joten sovelluksen on oltava käynnissä ennen kuin testejä ajetaan. Kohdesovelluksessa tämä on helpointa tehdä ajettava paketti Gradlella ja ajamalla sitä samalla kun suorittaa testejä.

Cypress etsii testejä oletuksena ”cypress” kansion sisällä olevasta ”integration” kansiosta. Luodaan toteutettava testi myöhemmin tähän kansioon, kuitenkin aluksi kansiosta voidaan poistaa kaikki esimerkkitestit.

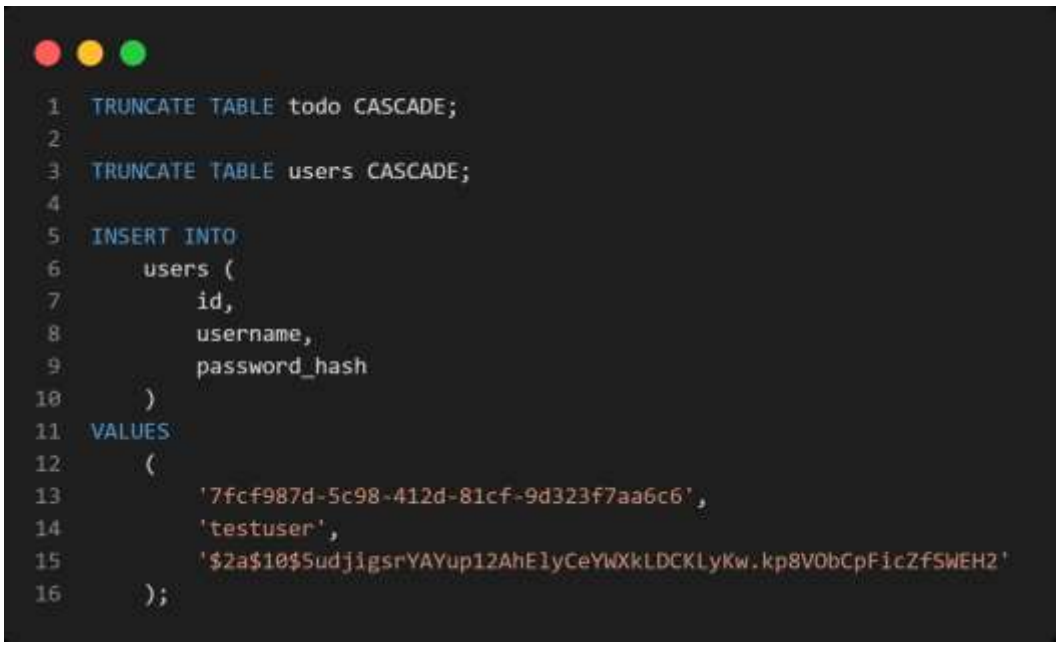
Kuten yleensä, testien tulisi olla luotettavia sekä toistettavassa olevia, mikä tarkoittaa sitä testeissä tulisi olla sama alkutilanne. Toteutetaan palvelinpuolelle rajapinta, jonka avulla voidaan tietokanta asettaa halutuksi päästä päähän -testien aikana. Rajapintaa tehtäessä on tärkeää rajoittaa sen toimintaa siten että se on

saatavilla vain päästä päähän -testien aikana. Käytössä oleva Spring Boot tarjoaa tähän sopivan annotaation ”Profile”, jonka avulla voidaan rajoittaa rajapinta olemaan saatavissa vain, jos tietty profiili on aktiivisena. Kuvassa 6 on esitetty rajapinta ”/e2e/reset-database”, jolla voidaan resetoida tietokanta sql-tiedoston avulla tiettyyn tilaan.

```java
@RestController
@Profile({ "e2e-local", "e2e-docker", "e2e-ci" })
public class E2eController {

    @Autowired
    NamedParameterJdbcTemplate jdbcTemplate;

    @Autowired
    private DataSource dataSource;

    @GetMapping("/api/e2e/reset-database")
    @ResponseStatus(HttpStatus.ACCEPTED)
    public void reset() {
        executeSqlScript("e2e-database.sql");
    }

    private void executeSqlScript(String file) {
        if (file.charAt(0) == '/') {
            file = new StringBuilder(file).deleteCharAt(0).toString();
        }

        try (Connection conn = dataSource.getConnection();) {
            ScriptUtils.executeSqlScript(conn, new ClassPathResource(String.format("/testdata/sql/%s", file)));
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

}
```

KUVA 6. Päästä päähän -testejä varten luodun rajapinnan logiikka

Tietokannan tyhjentäminen ja asettaminen hoidetaan sql-tiedoston avulla, joka ajetaan tietokantaan. Tiedoston sisältö on esitetty kuvassa 7. Suuremmissa ja monimutkaisissa järjestelmissä kannattaa hajauttaa sisältö useampaan tiedostoon ja muuttaa toteutusta siten että vain osa sisällöstä voidaan ajaa tietokantaan. Tässä työssä datan määrä on sen verran pieni, että se ei kasvata testien suoritusaikaa merkittävästi, joten kaikki kantaan tehtävät toimenpiteet ovat samassa tiedostossa.



```
TRUNCATE TABLE todo CASCADE;

TRUNCATE TABLE users CASCADE;

INSERT INTO
    users (
        id,
        username,
        password_hash
    )
VALUES
    (
        '7fcf987d-5c98-412d-81cf-9d323f7aa6c6',
        'testuser',
        '$2a$10$5udjigsrYAYup12AhElyCeYWXkLDCKLyKw.kp8VObCpFicZfSWEH2'
    );
```

KUVA 7. Tietokannan asettaminen tiettyyn tilaan E2E-testejä varten

Aloitetaan käyttäjän luomisen -testin tekeminen lisäämällä käyttöliittymän koodiin yksilöivät Cypressin data-cy HTML-attribuutit. Cypress voi hakea HTML-dokumentin DOM-puusta elementtejä usealla eri tavalla, mutta suositeltu tapa suurimmassa osassa tapauksia on luoda niille yksilöllinen data-cy attribuutti (Cypress 2021.). Kuvassa 8 on asetettu edellä mainittu yksilöllinen tunniste sovelluksen TextInput -komponentissa dynaamisesti syötekentälle sekä mahdolliselle virheelle. Attribuuttien lisääminen dynaamisesti vähentää kehittäjän manuaalista työtä.

```
<div className="form-control">
  <label className="label" id={ids.label}>
    <span className="label-text">{label}</span>
  </label>
  <input
    data-cy={`${props.name ?? ids.input}TextInput`}
    id={ids.input}
    aria-labelledby={ids.label}
    aria-describedby={error?.message ? ids.error : undefined}
    aria-invalid={error?.message ? true : false}
    className={classNames("input input-bordered", {
      "input-error": error?.message !== undefined,
    })}
    ref={ref}
    {...props}
  />
  {error?.message && (
    <div className="label" id={ids.error} data-cy={`${props.name ?? ids.input}TextInputError`}>
      <span className="label-text-alt">{error.message}</span>
    </div>
  )}
</div>
```

KUVA 8. TextInput -komponentin yksilöivän HTML-attribuutin asettaminen

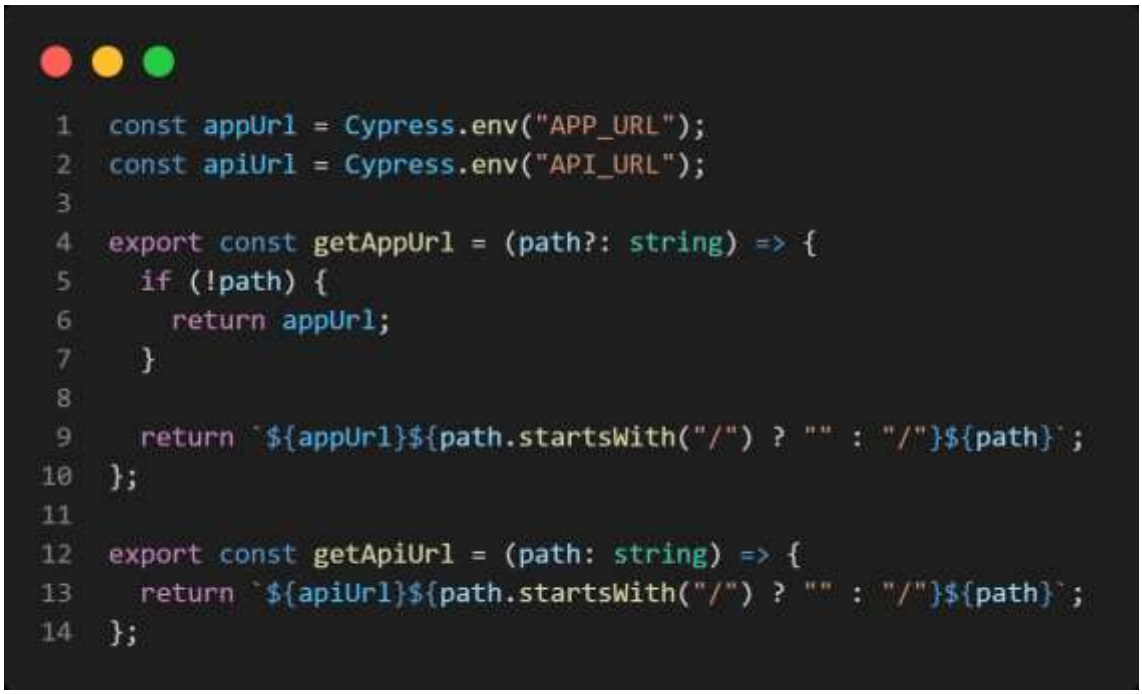
Määritellään muillekin tarpeellisille komponenteille sekä elementeille kuvan 8 mukaisesti data-cy attribuutit joko tekemällä dynaaminen logiikka tai lisäämällä ne kovakoodattuina manuaalisesti.

Käyttäjän luomisessa on tärkeää, että testi on toistettavissa, mikä vaatii edellä mainittua tietokannan resetointia. Testiä täytyy pystyä ajamaan useita kertoja peräkkäin onnistuneesti. Cypress mahdollistaa suoraan HTTP-pyyntöjen tekemisen, joten testissä voidaan kutsua rajapintaa, jolla tietokanta asetetaan ennen testin muun osan suorittamista. Huomioitavaa on kuitenkin, että kehitysympäristössä sovelluksen selain- ja rajapinta toimivat eri osoitteissa. Lisätään ”cypress.json” konfiguraatioon kuvassa 9 esitetyt oletusarvot ympäristömuuttujille APP_URL ja API_URL.

```
{
  "reporter": "junit",
  "reporterOptions": {
    "mochaFile": "cypress/results/junit-[hash].xml"
  },
  "env": {
    "APP_URL": "http://localhost:3000",
    "API_URL": "http://localhost:8080"
  }
}
```

KUVA 9. Lisätyt ympäristömuuttujat cypress.json tiedostossa

Ympäristömuuttujat ovat mahdollista ylikirjoittaa asettamalla ne konsolissa Cypressia ajettaessa, mukana täytyy olla etuliitteenä ”CYPRESS_”. Jotta Cypressin käyttö helpottuu, luodaan testejä varten apufunktiot getAppUrl() ja getApiUrl(). Funktiot ovat esitettynä kuvassa 10.



```
const appUrl = Cypress.env("APP_URL");
const apiUrl = Cypress.env("API_URL");

export const getAppUrl = (path?: string) => {
  if (!path) {
    return appUrl;
  }

  return `${appUrl}${path.startsWith("/") ? "" : "/"}${path}`;
};

export const getApiUrl = (path: string) => {
  return `${apiUrl}${path.startsWith("/") ? "" : "/"}${path}`;
};
```

KUVA 10. Apufunktiot, joilla käyttää ympäristömuuttujissa määritettyjä osoitteita helposti

Kun edellä mainitut apufunktiot ja DOM-elementtien yksilöinti on tehty, voidaan aloittaa testin kirjoittaminen. Aloitetaan se luomalla tiedosto ”register.spec.ts” kansioon ”integration”.

Kuvassa 11 on esitetty uuden käyttäjän luomisen -testi. Describe-funktiolla jaetaan testi loogisiin osiin sekä kuvataan testiä. It-funktion sisälle toteutetaan yksi testitapaus. Testitapauksien määrää ei ole rajoitettu. BeforeEach-funktion sisälle voidaan kirjoittaa ennen it-funktioita ajettava yhteinen osa. Testin jakaminen loogisiin osiin tekee testin lukemisesta helpompaa.

```
describe("Create a new user account", () => {
  const validUsername = "username1";
  const validPassword = "ValidP4ssw0rd!";

  const invalidUsername = "ab";
  const invalidPasswordLength = "test";
  const invalidPasswordRequirements = "Password123";

  const alreadyCreatedUsername = "testuser";

  beforeEach(() => {
    cy.request(getApiUrl("/api/e2e/reset-database"));
    cy.visit(getAppUrl());
    cy.get("[data-cy=loginView]").should("exist");
    cy.get("[data-cy=tabLoginButton]").should("have.class", "tab-active");
    cy.get("[data-cy=tabRegisterButton]").click().should("have.class", "tab-active");
    cy.get("[data-cy=tabLoginButton]").should("not.have.class", "tab-active");
    cy.get("[data-cy=registerForm]").should("exist");
  });

  it("User can create a new account", () => {
    cy.get("[data-cy=usernameTextInput]").type(validUsername);
    cy.get("[data-cy=passwordTextInput]").type(validPassword);
    cy.get("[data-cy=registerFormSubmit]").click();
    cy.get("[data-cy=successAlert]").should("exist");
    cy.get("[data-cy=loginForm]").should("exist");
  });

  it("User creation form validates inputs", () => {
    cy.get("[data-cy=usernameTextInput]").type(invalidUsername);
    cy.get("[data-cy=passwordTextInput]").type(invalidPasswordLength);
    cy.get("[data-cy=registerFormSubmit]").click();
    cy.get("[data-cy=usernameTextInputError]").containsTranslation("validation.username-length");
    cy.get("[data-cy=passwordTextInputError]").containsTranslation("validation.password-length");

    cy.get("[data-cy=usernameTextInput]").type(validUsername);
    cy.get("[data-cy=passwordTextInput]").type(invalidPasswordRequirements);
    cy.get("[data-cy=registerFormSubmit]").click();
    cy.get("[data-cy=usernameTextInputError]").should("not.exist");
    cy.get("[data-cy=passwordTextInputError]").containsTranslation("validation.password-characters");
  });

  it("User gets alert if username already taken", () => {
    cy.get("[data-cy=usernameTextInput]").type(alreadyCreatedUsername);
    cy.get("[data-cy=passwordTextInput]").type(validPassword);
    cy.get("[data-cy=registerFormSubmit]").click();
    cy.get("[data-cy=errorAlert]").containsTranslation("error.code-6"); // Check ApiErrorCode.java for error codes
  });
});
```

KUVA 11. Uuden käyttäjän luomisen testi

Testissä jokaisen testitapauksen alussa ajetaan beforeEach-funktiossa määritetty toiminta. Aluksi nollataan rajapinnan avulla tietokanta sekä siirrytään sovelluksen etusivulle. Koska kyseessä on yksisivuinen sovellus, sovelluksessa ei ole suoraa osoitetta rekisteröitymisen näkymään. Tehdään tarvittavat toimenpiteet, että päästään oikeaan näkymään.

Tämän jälkeen suoritetaan kolme eri testitapausta. Ensimmäinen testitapaus suorittaa toimenpiteet:

1. Täytä lomakkeen tiedot
2. Lähetä lomake
3. Tarkista, että ilmoitus onnistuneesta toimenpiteestä on näkyvissä sekä että kirjautumisnäkymä on näkyvissä

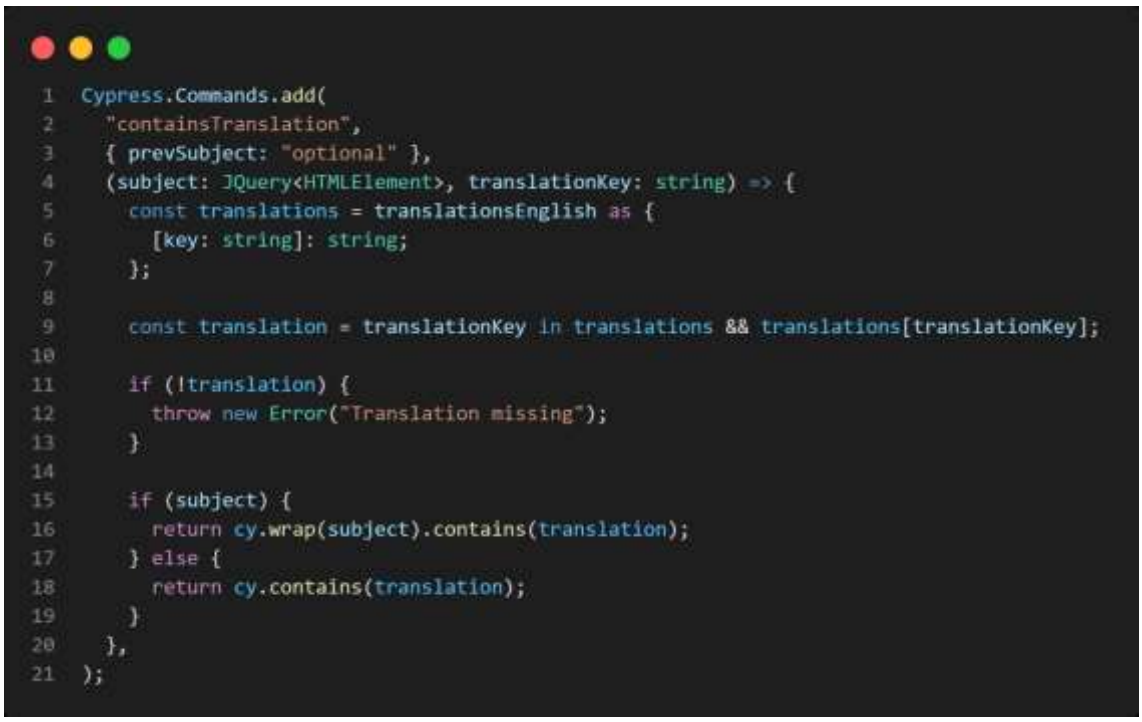
Toisessa testitapauksessa suoritetaan toimenpiteet:

1. Täytä virheelliset lomakkeen tiedot
2. Lähetä lomake
3. Varmista, että syötekenttien virheet ovat näkyvissä ja sisältävät oikean tekstin

Kolmannessa testissä suoritetaan toimenpiteet:

1. Täytä lomakkeen tiedot sellaisella käyttäjätunnuksella, joka on jo käytössä
2. Lähetä lomake
3. Varmista, että näkyvissä on virheilmoitus ja se sisältää oikean tekstin

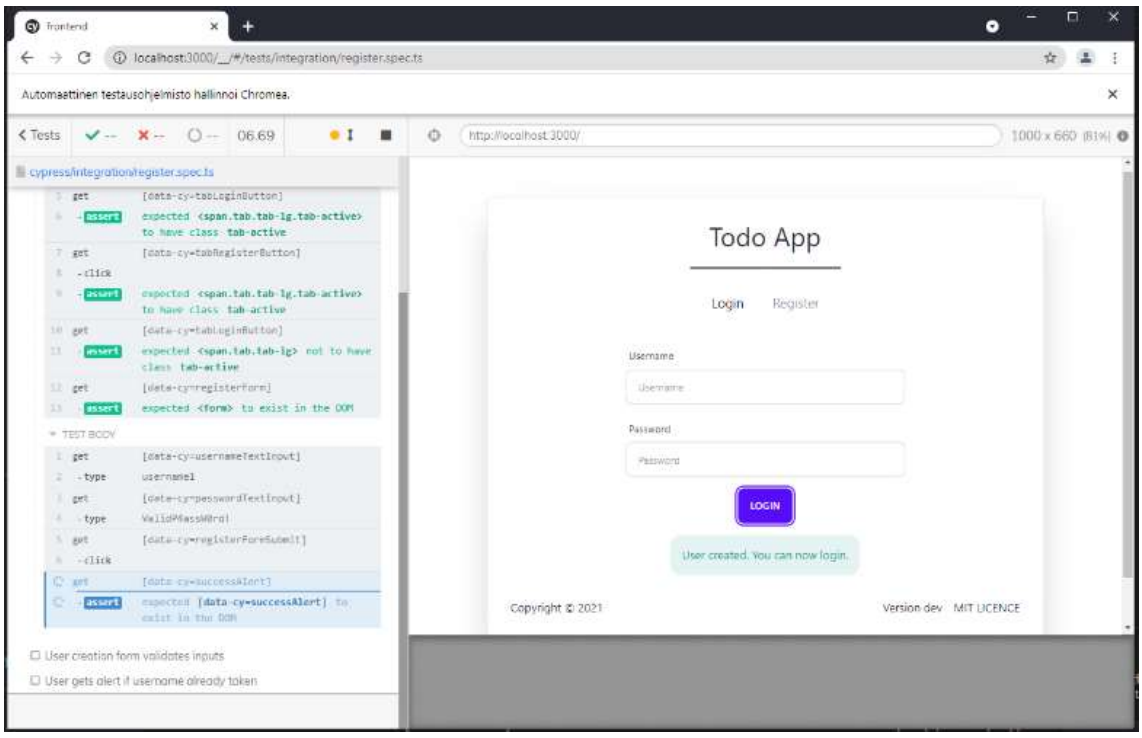
Ensimmäisessä testitapauksessa on käytetty taulukossa 1 esitettyjä Cypressin funktioita. Toisessa ja kolmannessa testitapauksessa on näiden lisäksi käytetty omaa kustomoitua funktiota. Koska kohdesovellus on lokalisoitu, tosin vain yhdelle kielelle, laajennettiin Cypressin komentoja omalla containsTranslation-funktiolla, joka hakee sovelluksen lähdekoodista halutun käännöksen ja välittää sen contains-funktiolle. Kustomoitu komento on esitetty kuvassa 12. Tämä mahdollistaa sen, että käännöksiä voidaan muokata rikkomatta testejä sekä helpottaa tulevaisuudessa useamman kielen implementointia, koska testejä ei tarvitse täysin uudelleen kirjoittaa vaan riittää pelkästään pieni refaktorointi.



```
Cypress.Commands.add(
  "containsTranslation",
  { prevSubject: "optional" },
  (subject: JQuery<HTMLElement>, translationKey: string) => {
    const translations = translationsEnglish as {
      [key: string]: string;
    };

    const translation = translationKey in translations && translations[translationKey];

    if (!translation) {
      throw new Error("Translation missing");
    }

    if (subject) {
      return cy.wrap(subject).contains(translation);
    } else {
      return cy.contains(translation);
    }
  },
);
```

KUVA 12. Kustomoidun Cypress-funktion lisääminen

Jos Cypressia ajetaan käyttöliittymällä, suorittaa Cypress testit kuvan 13 näköisessä ikkunassa. Ikkunasta kehittäjä voi seurata testien sujumista sekä se tarjoaa myös kehittäjälle erilaisia toimintoja testin analytiikan seuraamiseen sekä sovelluksen käyttäytymiseen testin eri vaiheissa. Cypress pitää tarkkaa lokia suoritetuista komennoista. Loki on näkyvissä kuvan 13 vasemmassa reunassa. Ikkunassa myös ilmoitetaan onnistuneet sekä epäonnistuneet testit sekä myös testien kesto. Edellä mainitut ilmoitukset ovat näkyvissä kuvassa 13 vasemmassa yläkulmassa. Jos Cypressiä ajetaan konsolista cypress run komennolla tai CI-putkessa, ottaa Cypress virhetilanteissa kuvakaappauksen sekä myös videon testin suorituksesta (Cypress 2021). Kuvia sekä videoita voidaan sitten hyödyntää ongelmien selvittelyssä.



KUVA 13. Kuvakaappaus Cypressin käyttöliittymästä testien ajon aikana

5 TESTIN AJAMINEN CI-PUTKESSA

5.1 Gitlab CI-putken luominen

Sovelluksen lähdekoodia säilytetään TAMKin tarjoamassa Gitlab-versionhallinnassa, minkä takia testien ajaminen automaattisesti toteutetaan samassa ympäristössä käyttäen Gitlabin tarjoamaa CI/CD-ratkaisua. Gitlabin CI/CD-ratkaisun pääelementit ovat putket (eng. pipelines), vaiheet (eng. stages) ja työt (eng. jobs) (Gitlab 2021).

Työ on pienin ajettava yksikkö, joka koostuu vähintään ajettavasta skriptistä. Työlle voidaan määritellä mm. käytettävä Docker-levykuva, palveluita ja ehtoja. Jokainen työ kuuluu johonkin vaiheeseen. Samassa vaiheessa olevat työt ajetaan rinnakkain. Kun vaiheen kaikki työt ovat ajettu, siirrytään ajamaan seuraavan vaiheen työt. Putki kokoaa vaiheet ja työt yhteen. Edellä mainittujen konfiguraatiot toteutetaan YAML-merkintäkielellä. (Gitlab 2021.)

Käyttöönotto tapahtuu niin yksinkertaisesti, että luodaan ”.gitlab-ci.yml” tiedosto-projektin juureen. Tiedosto voi sisältää kaikki CI/CD-putken konfiguraatiot, mutta selkeyden vuoksi on hyvä eriyttää eri vaiheet omiksi tiedostoiksi, jotka sitten sisällytetään konfiguraatioon.

5.2 Päästä päähän -testien toteuttaminen CI-putkeen

Kohdesovellukseen on jo toteutettu CI-putkeen vaiheet ja omat työt yksikkötesteille, sovelluksen paketoinnille ja Docker-levykuvan luonnille. Vaiheiden konfiguraatiot on eriytetty omiksi konfiguraatiotiedostoiksi.

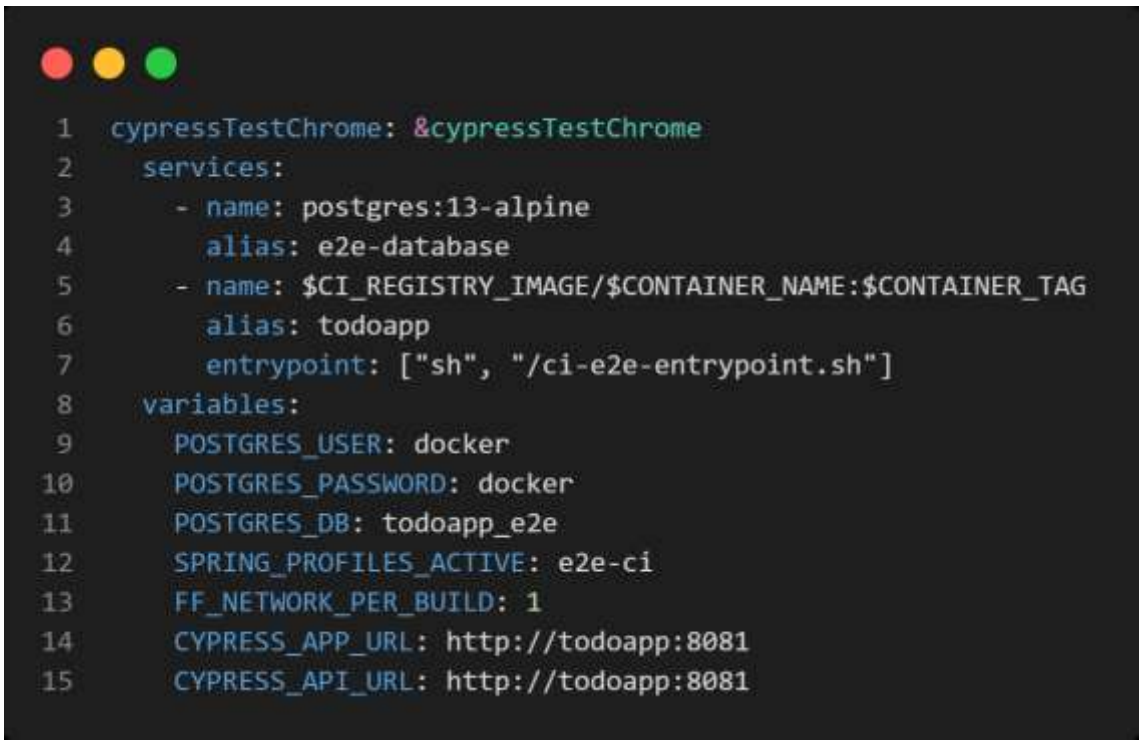
Toteutetaan päästä päähän -testeille oma vaihe ja työ osaksi jo olemassa olevaa putkea. Kuten kappaleessa 3.2 on mainittu, ajetaan päästä päähän -testit osana putkea vain silloin kun kyseessä on yhdistämispyyntö, päähaaran tulleet muutokset tai ajastettu ajo. Cypress tukee useita eri selaimia, joilla testit voidaan ajaa. Toteutetaan putkeen päästä päähän -testaus siten että ajetaan rinnakkain testit

Chrome- ja Firefox-selaimilla. Tämä tarkoittaa sitä, että toteutetaan kaksi melkein lähes identtistä työtä osaksi putkea. Päästä päähän -testien ajaminen voi kestää monimutkaisemmassa sovelluksessa useita kymmeniä minuutteja, joten suorituksen kestoa voidaan minimoida rinnakkaisajolla.

Aloitetaan uuden CI-putken työn tekeminen luomalla uusi konfiguraatiotiedosto nimeltä ”e2e.gitlab-ci.yml”. Toteutetaan ensiksi päästä päähän -testeille työ, jossa ajetaan testit Chromella. Tämän jälkeen hyödyntäen YML-ankkuria periytetään edellä mainitusta työstä uusi työ, jossa kaikki vaiheet suoritetaan vastaavasti, mutta testit ajetaan Firefoxilla. Töillä erona on siis vain skripti, joka ajetaan työn yhteydessä. Tällä tavalla säästytään turhalta toistuvalta konfiguraatiolta sekä pidetään työt yhtenäisinä.

Aloitetaan siis konfiguroimaan päästä päähän -testin Gitlab CI-putken työtä, jossa testit ajetaan Chromella. Annetaan työlle nimi ”cypressTestChrome” ja lisätään jo valmiiksi perään YML-ankkuri. Asetetaan työssä käynnistymään omina palveluinaan tietokanta sekä testattava sovellus. Käytetään tietokannan levykuvana julkisesta levykuvarekisteristä löytyvää PostgreSQL:n levykuvaa. Sovelluksen levykuvana käytetään projektin omaan levykuvarekisteriin luotua levykuvaa. Sovelluksen levykuvana käytettävä versio määräytyy muuttujien $CONTAINER_NAME ja $CONTAINER_TAG perusteella. Jokaisella suoritettavalla putkella on oma levykuvansa, jotta on mahdollista ajaa useita suorituksia putkesta samaan aikaan sekoittamatta eri haaroja, pois lukien versionhallinnan päähaara, jonka tagina toimii aina ”latest”.

Sovelluksen käynnistämisessä tietokannan tulee olla päällä. Konfiguraatio ei kuitenkaan mahdollista palveluiden välistä odottamista kirjoitushetkellä, joten toteutetaan kustomoitu skripti, joka odottaa noin 10 sekuntia ennen kuin käynnistää sovelluksen. Lisätään skripti kopioitumaan mukaan Docker-levykuvaan, jotta se on käytettävissä levykuvasta käsin. Kuvassa 14 on määritettynä työn palvelut sekä ympäristömuuttujia, joita palvelut sekä Cypress käyttävät.

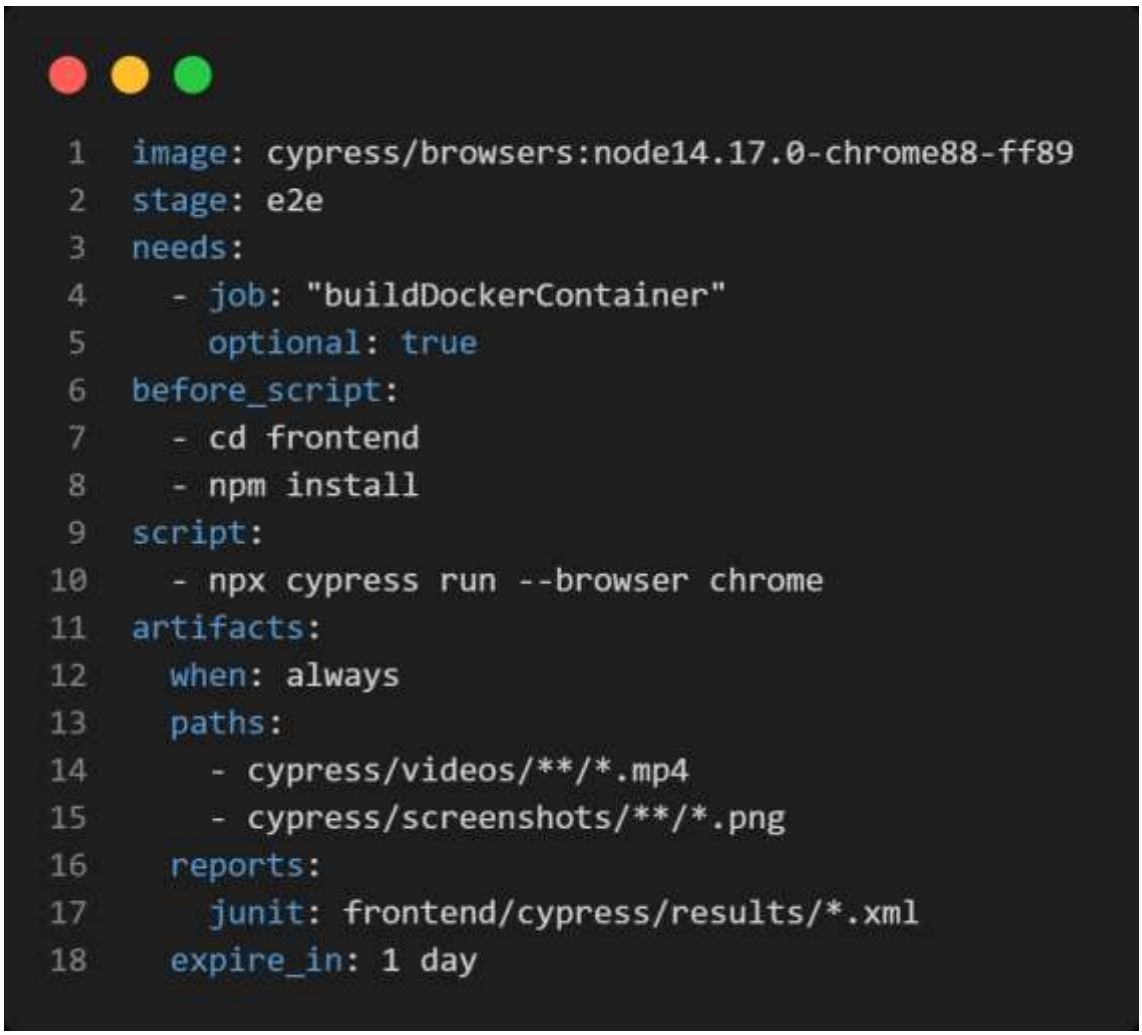


```
1  cypressTestChrome: &cypressTestChrome
2    services:
3      - name: postgres:13-alpine
4        alias: e2e-database
5      - name: $CI_REGISTRY_IMAGE/$CONTAINER_NAME:$CONTAINER_TAG
6        alias: todoapp
7        entrypoint: ["sh", "/ci-e2e-entrypoint.sh"]
8    variables:
9      POSTGRES_USER: docker
10     POSTGRES_PASSWORD: docker
11     POSTGRES_DB: todoapp_e2e
12     SPRING_PROFILES_ACTIVE: e2e-ci
13     FF_NETWORK_PER_BUILD: 1
14     CYPRESS_APP_URL: http://todoapp:8081
15     CYPRESS_API_URL: http://todoapp:8081
```

KUVA 14. Tiedostossa e2e.gitlab-ci.yml määritetyt palvelut ja ympäristömuuttujat

Tietokannan ympäristömuuttujat ovat samat kuin sovelluksen e2e-ci profiilissa määritetyt tietokannan yhdistämiseen vaaditut tiedot. Ympäristömuuttujalla FF_NETWORK_PER_BUILD asetetaan kaikki työn Docker-levykuvat ajettavaksi samassa verkossa. Tämä vaaditaan, jotta palvelut voivat keskustella keskenään. Lisätään myös kappaleessa 4.3 määritetyt Cypressin ympäristömuuttujat vastaamaan sovelluksen osoitetta.

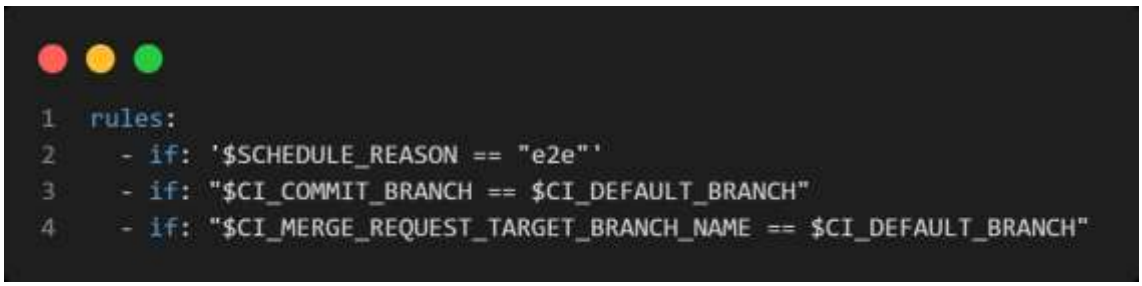
Cypress tarjoaa Docker-levykuvan CI-putkessa testaamista varten, joten hyödynnetään sitä työssä. Määritellään työlle levykuva, vaihe, riippuvuudet muihin töihin, ajettava skripti sekä työstä talteen otettavat artefaktit eli testien tulosten raportit sekä mahdolliset virheistä talteen otetut videot ja kuvakaappaukset. Määritetty konfiguraatio on esitetty kuvassa 15. Asetetaan riippuvuudeksi sovelluksen Docker-levykuvan työ, mutta siten että se on valinnainen. Tehdään näin siksi että työtä tullaan myös ajamaan ajastetusti versionhallinnan päähaaralle, jolloin Docker-levykuvaa ei luoda. Riippuvuuden ollessa valinnainen, sen valmistumista odotetaan, jos se on osana putkea.



```yaml
image: cypress/browsers:node14.17.0-chrome88-ff89
stage: e2e
needs:
  - job: "buildDockerContainer"
    optional: true
before_script:
  - cd frontend
  - npm install
script:
  - npx cypress run --browser chrome
artifacts:
  when: always
  paths:
    - cypress/videos/**/*.mp4
    - cypress/screenshots/**/*.png
  reports:
    junit: frontend/cypress/results/*.xml
  expire_in: 1 day
```

KUVA 15. Tiedostossa e2e.gitlab-ci.yml määritetyt työn kontin konfiguraatiot

Asetetaan vielä työlle ehdot, jolloin se ajetaan. Työ ajetaan aina, jos kyseessä on ajastettu ajo, jonka syynä on ”e2e” eli päästä päähän -testit. Lisäksi työ ajetaan aina kun kyseessä on muutokset versionhallinnan päähaaraan tai kyseessä on yhdistämispyyntö päähaaraan. Ehdot ovat näkyvillä kuvassa 16.



```yaml
rules:
  - if: '$SCHEDULE_REASON == "e2e"'
  - if: "$CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH"
  - if: "$CI_MERGE_REQUEST_TARGET_BRANCH_NAME == $CI_DEFAULT_BRANCH"
```

KUVA 16. Tiedostossa e2e.gitlab-ci.yml määritetyt työn ehdot

Luodaan vielä äsken luodusta työstä sellainen versio, jossa testit ajetaan Firefoxilla. Tämä tapahtuu hyödyntämällä jo aikaisemmin mainittua YML-ankkuria.

Periytetään uusi työ nimeltä ”cypressTestFirefox” juuri luodusta työstä, mutta muutetaan skripti osuus siten että ajetaan testit Firefoxilla. Kuvassa 17 on esitettynä koko e2e.gitlab-ci.yml tiedosto.

```yaml
cypressTestChrome: &cypressTestChrome
  services:
    - name: postgres:13-alpine
      alias: e2e-database
    - name: $CI_REGISTRY_IMAGE/$CONTAINER_NAME:$CONTAINER_TAG
      alias: todoapp
      entrypoint: ["sh", "/ci-e2e-entrypoint.sh"]
  variables:
    POSTGRES_USER: docker
    POSTGRES_PASSWORD: docker
    POSTGRES_DB: todoapp_e2e
    SPRING_PROFILES_ACTIVE: e2e-ci
    FF_NETWORK_PER_BUILD: 1
    CYPRESS_APP_URL: http://todoapp:8081
    CYPRESS_API_URL: http://todoapp:8081
  image: cypress/browsers:node14.17.0-chrome88-ff89
  stage: e2e
  needs:
    - job: "buildDockerContainer"
      optional: true
  before_script:
    - cd frontend
    - npm install
  script:
    - npx cypress run --browser chrome
  artifacts:
    when: always
    paths:
      - cypress/videos/**/*.mp4
      - cypress/screenshots/**/*.png
    reports:
      junit: frontend/cypress/results/*.xml
    expire_in: 1 day
  rules:
    - if: '$SCHEDULE_REASON == "e2e"'
    - if: "$CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH"
    - if: "$CI_MERGE_REQUEST_TARGET_BRANCH_NAME == $CI_DEFAULT_BRANCH"

cypressTestFirefox:
  <<: *cypressTestChrome
  script:
    - npx cypress run --browser firefox
```

KUVA 17. Lopullinen e2e.gitlab-ci.yml tiedosto

Aivan lopuksi lisätään vielä putken vaiheisiin e2e-vaihe sekä sisällytetään juuri luotu ”e2e.gitlab-ci.yml” tiedosto mukaan putken konfiguraatioon kuvassa 18 esitetyllä tavalla.

```yaml
stages:
  - test
  - build
  - package
  - e2e

include:
  - local: "gitlab-ci/test.gitlab-ci.yml"
  - local: "gitlab-ci/build.gitlab-ci.yml"
  - local: "gitlab-ci/package.gitlab-ci.yml"
  - local: "gitlab-ci/e2e.gitlab-ci.yml"
```

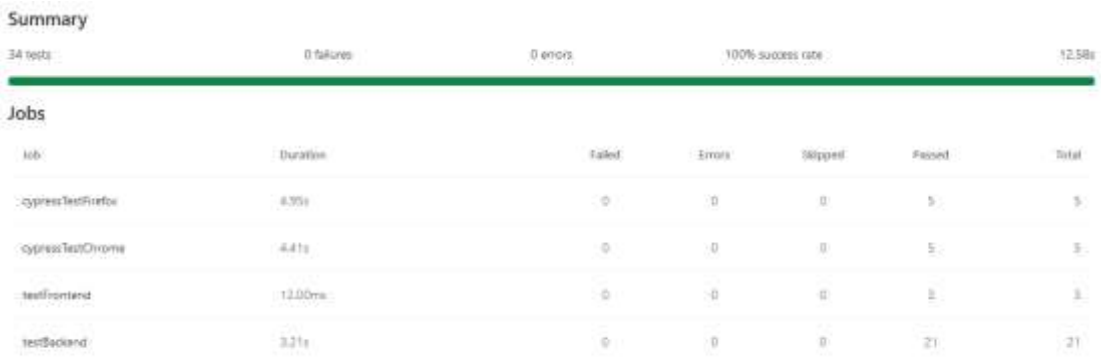
KUVA 18. Omissa konfiguraatiotiedostoissa olevien töiden sisällyttäminen osaksi putkea sekä vaiheiden määritys .gitlab-ci.yml -tiedostossa

5.3 Testien tulokset ja valvonta Gitlab CI-putkessa

Putkessa suorituksen tuloksia voidaan seurata projektin Gitlab projektin ”Pipeline” osiosta. Kuvassa 19 on esillä yhden yhdistämispyynnön yhteydessä ajetut työt sekä viivoilla esitetty niiden riippuvuudet toisiinsa.

KUVA 19. Yhdistämispyynnön CI-putken työt

Kuten huomataan, ensiksi ajetaan kohdesovelluksen yksikkötestejä ja tämän jälkeen paketoidaan sovellus. Paketoidusta sovelluksesta luodaan vielä Docker-levykuva. Koska kyseessä on yhdistämispyyntö, ajetaan vielä päästä päähän -testit, jotka käyttävät edellä luotua Docker-levykuvaa. Samalta sivulta on myös mahdollista tarkastella testien tuloksien yhteenveto. Tuloksien yhteenveto on esitetty kuvassa 20.



KUVA 20. Gitlab CI-putkessa ajettujen testien tulosten yhteenveto

Ajastetussa ajossa putkessa ei ole kuin päästä päähän -testit. Ajastetun ajon aikataulu määritellään Gitlab projektin ”Schedules” osiosta. Määritettiin projektille CI-putken ajaminen päivittäin päähaaraa vasten syyllä ”e2e”, jolloin kuvassa 16 määritellyt ehdot täyttyvät. Kuvassa 21 on esitetty kuvakaappaus Gitlabin CI-putkesta, jossa nähdään, että ajastetut testit ovat ajettu onnistuneesti useana eri päivänä.

KUVA 21. Kuvakaappaus projektin ajetuista ajastetuista putkista

6 POHDINTA JA TULOKSET

Ohjelmistotestauksen merkitystä osana ohjelmistokehityksen prosessia ei voida korostaa liikaa. Ohjelmistotestauksen tavoitteena on ohjelmistovirheiden löytäminen ja varmistaa, että ohjelmisto täyttää sille asetetut liiketoiminnalliset ja tekniset vaatimukset. Tavoitteen saavuttamiseksi on tärkeää suorittaa ohjelmistotestausta kehityksen jokaisessa kehitysvaiheessa kattavasti.

Tässä työssä taustoitettiin ohjelmistotestausta yleisesti sekä toteutettiin päästä päähän -testit nykyaikaiselle modernille verkkosovellukselle käyttämällä Cypress-työkalua sekä lisättiin päästä päähän -testaaminen osaksi jatkuvaa integraatiota. Opinnäytetyön tavoitteena oli tuottaa toimiva ja kattava katsaus edellä mainittuihin asioihin.

Cypress on testaustyökalujen uusinta uutta ja soveltuu erinomaisesti nykyaikaisten verkkosovellusten testaamiseen. Gitlabin tarjoama jatkuvan integraation ratkaisu soveltuu myös lähes ongelmitta päästä päähän -testaamiseen.

Työssä annettiin kattava katsaus siihen, miten päästä päähän -testaaminen toteutetaan Cypress-työkalulla sekä kuinka se saadaan liitettyä osaksi Gitlabin jatkuvaa integraatiota. Lopputuloksena syntyi selvitys, jota voidaan hyödyntää apuna, kun toteutetaan päästä päähän -testausta nykyaikaiselle verkkosovellukselle.

LÄHTEET

Agenda. 2019. Digitaalisen palvelun käytettävyystestaus. Luettu 8.10.2021. https://agendahelsinki.fi/2019/08/08/kaytettavyystestaus-mita-miksi-miten/

Bentley J., Bank W. & NC C. n.d. Software Testing Fundamentals. Luettu 8.10.2021. https://support.sas.com/resources/papers/proceedings/proceedings/sugi30/141-30.pdf

Cypress. 2021. Best Practices. Luettu 10.10.2021. https://docs.cypress.io/guides/references/best-practices

Cypress. 2021. Introduction to Cypress. Luettu 16.10.2021. https://docs.cypress.io/guides/core-concepts/introduction-to-cypress

Cypress. 2021. Key differences. Luettu 9.10.2021. https://docs.cypress.io/guides/overview/key-differences

Cypress. 2021. Screenshots and Videos. Luettu 10.10.2021. https://docs.cypress.io/guides/guides/screenshots-and-videos

Cypress. 2021. Why Cypress. Luettu 9.10.2021. https://docs.cypress.io/guides/overview/why-cypress

Gitlab. 2021. Gitlab CI/CD. Luettu 10.10.2021. https://docs.gitlab.com/ee/ci/

Itkonen J. & Rautiainen K. 2005. Exploratory testing: a multiple case study. Luettu 8.10.2021. Vaatii käyttöoikeuden. https://ieeexplore.ieee.org/document/1541817

MDN Web Docs. 2021. SPA (Single-page Application). Luettu 16.10.2021. https://developer.mozilla.org/en-US/docs/Glossary/SPA

Myers, G. J., Sandler C. & Badgett T. 2012. The Art of Software Testing. 3. painos. John Wiley & Sons.

Nancy J. W. 1999. An overview of regression testing. Luettu 8.10.2021. Vaatii käyttöoikeuden. https://dl.acm.org/doi/abs/10.1145/308769.308790

RedHat. 2021. What is CI/CD. Luettu 16.10.2021. https://www.redhat.com/en/topics/devops/what-is-ci-cd

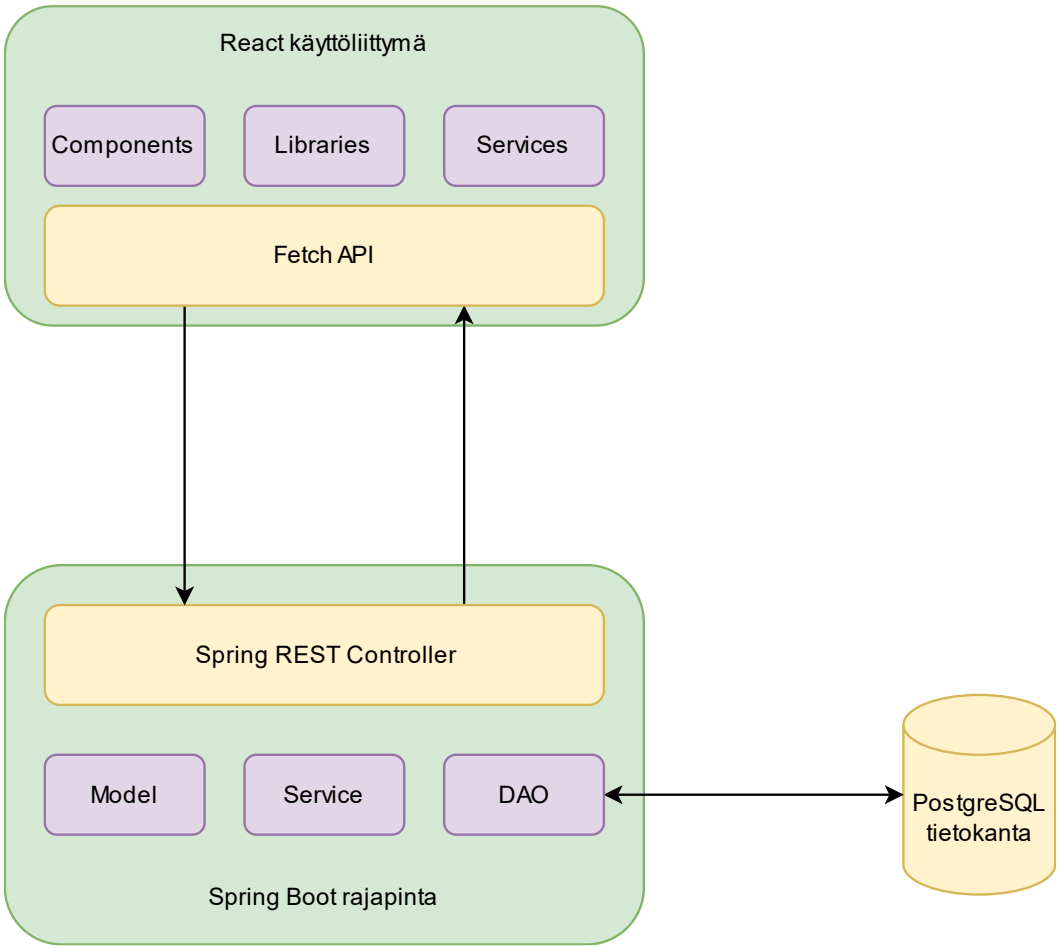
Selenium. 2021. Selenium Browser Automation Project. Luettu 9.10.2021. https://www.selenium.dev/documentation/

Stan G. 2021. What is End-to-End Testing and When Should You Use It. Luettu 8.10.2021. https://www.freecodecamp.org/news/end-to-end-testing-tutorial/

Utor. 2020. Functional vs Non-Functinal Testing. Luettu 8.10.2021.
https://u-tor.com/topic/functional-vs-non-functional

LIITTEET

Liite 1. Kohdesovelluksen arkkitehtuuri



Liite 2. Kohdesovelluksen testaussuunnitelma

TodoApp -sovelluksen testaussuunnitelma

Versio	Päivämäärä	
1.0	1.6.2021	Ensimmäinen versio
1.1	15.6.2021	Lisätty toiminallisuuksia taulukkoon 2

Tässä dokumentissa on määritelty opinnäytetyötä varten kehitetyn TodoApp -sovelluksen testaussuunnitelmaa. Testaussuunnitelma on toteutettu sellaisella olettamalla, että sovellusta kehitettäisiin aktiivisesti eteenpäin. Testien tarkoituksena on parantaa sovelluksen laatua sekä luotettavuutta.

Tässä dokumentissa määritellään testausstrategiaa, testattavia toiminallisuuksia sekä muita testauksessa huomioon otettavia asioita.

Testausstrategia

Tavoitteena on varmistaa, että sovellus toimii halutulla tavalla. Käyttöliittymän monimutkaisiin algoritmeihin ja util -luokkiin pyritään kirjoittamaan Jest -testit. Rajapintaa testataan JUnit -testeillä käyttäen hyödyksi MockMVC -kirjastoa, jolla testataan rajapintaa suoraan REST-rajapinnan kautta. Käyttöliittymässä ei siis lähtökohtaisesti toteuteta testejä jokaiselle komponentille eikä rajapinnan business-logiikkaa testata lähtökohtaisesti muuten kuin rajapinnan kautta. Lisäksi suoritetaan päästä päähän testaamista suosituimmilla selaimilla. Tavoitteena on, että suurin osa testeistä voidaan ajaa myös automaattisesti osana CI-putkea.

Testaukseen liittyviä oletuksia ja periaatteita

- Testauksessa käytettään tuotantoa vastaavaa dataa
- Testausympäristö vastaa mahdollisimman tarkasti tuotantoympäristöä
- Testauksessa ei toisteta samojen toiminallisuuksien testausta
- Testien tavoite ja lopputulokset ovat selkeitä

Testauksen suorittaminen

Testit tulee pystyä suorittamaan ohjelmistokehittäjän toimesta manuaalisesti tai automaattisesti lokaalissa ympäristössä. Testejä tulisi ajaa mahdollisimman usein, mutta vähintään enenen työn lähettämistä versionhallintaan.

CI-putkessa tulisi ajaa käyttöliittymän Jest -testit sekä rajapinnan JUnit -testit jokaisen versionhallintaan lähetetyn muutoksen yhteydessä haarasta riippumatta. Päästä päähän -testit voidaan ajaa ainoastaan, kun muutoksia ollaan viemässä päähaaraan.

Testattavat toiminallisuudet

Toiminallisuutta testatessa voidaan löytää ohjelmistovirheitä, joiden vakavuus määritellään taulukon 1 perusteella. Taulukossa 2 on esitetty testattavia toiminallisuuksia.

Luokka	**Kuvaus**
Blokkaava (Blocker)	Estää sovelluksen käytön. Ei voida viedä tuotantoon.
Kriittinen (Critical)	Aiheuttaa epävakautta tai tietojen menetystä sovellusta käytettäessä. Tuotantoon vieminen mahdollista harkinnan mukaan.
Merkittävä (Major)	Heikentää sovelluksen laatua ja vakautta, mutta ei aiheuta sellaista haittaa, joka estäisi sovelluksen käyttöä. Ei estä sovelluksen tuotantoon vientiä.
Mitätön (Trivial)	Ei vaikuta juurikaan sovelluksen käyttöön. Ei estä sovelluksen tuotantoon vientiä.

Taulukko 1. Testauksessa esiintyvien bugien luokittelu

Toiminto	**Kuvaus**
Uuden käyttäjän luominen	Sovellukseen voidaan luoda uusi käyttäjä
Sisäänkirjautuminen	Käyttäjä pystyy kirjautumaan sisään sovellukseen
Käyttäjäsession säilyminen	Käyttäjän sessio säilyy, vaikka sivu päivitetään
Uloskirjautuminen	Käyttäjä pystyy kirjautumaan ulos sovelluksesta
Uuden tehtävän luominen listalle	Käyttäjä pystyy luomaan uuden tehtävän listalle
Tehtävälista säilyy sessioiden välillä	Käyttäjän tehtävälista säilyy eri sessioiden välillä
Tehtävän merkkaaminen tehdyksi	Käyttäjä pystyy merkkaamaan tehtävälistalta tehtävän tehdyksi
Tehtävän poistaminen	Käyttäjä pystyy poistamaan tehtävälistalta tehtävän
Tehtävän muokkaaminen	Käyttäjä pystyy muokkaamaan tehtävälistalla tehtävän tietoja

Taulukko 2. Testattavat toiminallisuudet