Opinnäytetyö (AMK)

Tieto- ja viestintätekniikka

2024

Ville Laitinen

Next.js-sovelluksen automaatiotestauksen suunnittelu ja implementointi

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

Opinnäytetyö (AMK) | Tiivistelmä

Turun ammattikorkeakoulu

Tieto- ja viestintätekniikka

2024 | 50 sivua

Ville Laitinen

Next.js-sovelluksen automaatiotestauksen suunnittelu ja implementointi

Automaattiset testit ovat ohjelmia, jotka automatisoivat sovelluksen testauksen. Ne suorittavat testattavan sovelluksen toimintoja ja vertaavat saatuja tuloksia odotettuihin, ennalta määriteltyihin tuloksiin.

Opinnäytetyön tavoitteena oli vertailla sopivia automaatiotestaustyökaluja toimeksiantajan kehitysvaiheessa olevaan Next.js-sovellukseen, suunnitella tarvittavat testit sovelluksen vaatimiin käyttötapauksiin, sekä implementoida suunnitellut testit sovellukseen.

Tuloksena luotiin sovellukselle testaussuunnitelma, valittiin sopivat automaatiotestaustyökalut sekä toteutettiin suunnitellut testit niiltä osin kuin sovelluksen toiminnot olivat valmiina testattavaksi.

Lisäksi tehtiin aloittelijaystävällinen dokumentaatio implementoidusta testausympäristöstä, ohjeet testien ajamiselle, sekä lista tarvittavista jatkotoimenpiteistä sovelluksen tuleville ylläpitäjille.

Asiasanat:

automaatiotestaus, JavaScript, TypeScript, React, Next.js.

Bachelor’s Thesis | Abstract

Turku University of Applied Sciences

Information and Communications Technology

2024 | 50 pages

Ville Laitinen

Design and Implementation of Automated Testing for a Next.js Application

Automated tests are programs that automate the testing of an application. They execute the functions of the application under test and compare the results with the expected, predetermined results.

The objective of the thesis was to compare suitable automation testing tools for the thesis client’s Next.js application under development, design the necessary tests for the use cases required by the application, and implement the designed tests in the application.

As a result, a test plan was created for the application, suitable automation testing tools were selected, and the planned tests were implemented as far as application functions were ready to be tested.

Additional outputs of this thesis include the compilation of a beginner-friendly documentation of the implemented testing environment, instructions for running the tests, and a list of necessary follow-up actions for future application maintainers.

Keywords:

automated testing, JavaScript, TypeScript, React, Next.js

Sisältö

Kuvat

Kuviot

Taulukot

Sanasto

API	Application Programming Interface eli ohjelmointirajapinta.
CSS	Cascading Style Sheets on tyylittelykieli, jolla määritellään verkkosivujen ulkoasu ja tyyli.
Figma	Käyttöliittymien suunnitteluun käytetty työkalu.
HTML	Hypertext Markup Language on kieli, jolla määritellään verkkosivujen rakenne ja sisältö.
HTTP	Hypertext Transfer Protocol on verkkoprotokolla, jota käytetään tietojen siirtämiseen internetissä.
Node.js	Ajoympäristö JavaScript-koodin suorittamiseen palvelinpuolella.
NPM	Node Package Manager on Node.js mukana tuleva, JavaScript-kirjastojen asentamiseen ja jakamiseen tarkoitettu pakettien hallintaohjelma.
RESTful API	Representational State Transfer Application Programming Interface on rajapintamalli tiedon siirtämiseen HTTP-protokollalla.

1 Johdanto

Opinnäytetyön tavoitteena on vertailla sopivia automaatiotestaustyökaluja toimeksiantajan kehitysvaiheessa olevaan Project Gate 2.0 nimiseen Next.js-sovellukseen, suunnitella tarvittavat testit sovelluksen vaatimiin käyttötapauksiin, sekä implementoida suunnitellut testit sovellukseen.

Opinnäytetyön toimeksiantaja on Turun ammattikorkeakoulussa toimiva opiskelijakeskeinen ICT-projektitoimisto theFIRMA. Project Gate 2.0-sovelluksen toiminnallisena tavoitteena on theFIRMAn sisäisten projektien ja niissä toimivien projektityöntekijöiden hallinnoinnin helpottaminen.

Aiheen valintaan vaikuttivat sovelluksen kehitystyön aikana havaittu automaatiotestien tarve yhdistettynä henkilökohtaiseen mielenkiintoon oppia aiheesta enemmän ja hyödyntää opittua omissa henkilökohtaisissa sovelluskehitysprojekteissa. Automatisoitu sovelluksen testaus on tärkeä osa modernia sovelluskehitystä ja parantaa sovelluksen laatua ja ylläpidettävyyttä.

Aihetta lähestytään tutustumalla käytetyimpiin alan testaustyökaluihin ja vertailemalla niiden sopivuutta opiskelijakeskeiseen kehitystyöhön sekä Project Gate 2.0-sovelluksen teknologioihin.

Oman kehittymisen tavoitteita opinnäytetyöprosessin aikana ovat tekniseen kirjoittamiseen liittyvien taitojen parantaminen sekä automaatiotestauksen teorian sekä työkalujen erojen syvempi ymmärtäminen.

2 Project Gate 2.0 -sovelluksen teknologiat

2.1 JavaScript-ohjelmointikieli

JavaScript on alun perin Netscapen kehittämä, nykyään Ecma Internationalin ylläpitämä, dynaamisesti tyypitetty ohjelmointikieli, jota käytetään laajasti web-sivustojen ja -sovellusten kehittämisessä. Se on yksi keskeisistä web-teknologioista yhdessä HTML:n ja CSS:n kanssa. JavaScript mahdollistaa vuorovaikutuksen käyttäjän ja verkkopalvelimen kanssa, animoidun grafiikan ja dynaamisesti päivittyvän sisällön verkkosivuilla. (Mozilla 2024.)

Vuoden 2023 Stack Overflow’n kehittäjäkyselyn mukaan JavaScript oli vuonna 2023 kaikista yleisimmin käytetty ohjelmointikieli ohjelmistokehittäjien keskuudessa jo yhdettätoista vuotta peräkkäin (Stack Overflow 2023).

Tämän lisäksi vuoden 2020 jälkeen yli 97 % verkkosivuista käytti JavaScriptiä asiakaspuolen ohjelmointikielenä (W3Techs 2024). JavaScriptin asema modernina web-teknologiana on siis hyvin keskeinen.

2.2 TypeScript-ohjelmointikieli

TypeScript on Microsoftin kehittämä ja ylläpitämä ohjelmointikieli, joka mahdollistaa staattisen tyypityksen käytön JavaScriptissä. TypeScript on JavaScriptin ylijoukko, minkä ansiosta se sisältää kaikki JavaScriptin ominaisuudet omien ominaisuuksiensa lisäksi (Microsoft 2024a).

Kaikesta TypeScriptillä kirjoitetusta koodista tulee kääntämisvaiheessa JavaScriptiä ja kaikki TypeScript-pohjainen ohjelmakoodi poistuu. Täten valmiin sovelluksen toiminnallisuuden kannalta käännetty lopputulos on sama (Torppa ym. 2023a).

TypeScriptin pääasialliset hyödyt esiintyvät sovelluksen kehitysvaiheessa. TypeScript mahdollistaa tyyppitarkistuksen ja staattisen koodianalyysin käytön ohjelmistokehityksessä. Tällöin voidaan etukäteen määritellä tyyppivaatimuksia

sovelluksen muuttujille ja funktioille. Koodin kääntäjä varoittaa mahdollisista tyyppivirheistä koodia kirjoitettaessa, mikä osaltaan vähentää koodin ajon aikana mahdollisia virhetilanteita. Vahvasti tyypitetty lähdekoodi on myös itseään dokumentoivaa. Tyyppihuomautukset (eng. type annonation) antavat yksiselitteisen vastauksen sille, minkä tyyppistä dataa minkäkin funktion odotetaan ottavan vastaan ja palauttavan. (Torppa ym. 2023b.)

Lähdekoodin kirjoittaminen TypeScriptillä JavaScriptin sijasta on yleensä suositeltavaa, jos sovelluksesta on tulossa laaja. Pääsääntöisesti mitä enemmän lähdekoodia ja kehittäjiä sovelluksella on, sitä enemmän TypeScriptin hyödyt tulevat esiin (Monesi 2024). Puhdas JavaScript taas voi sopia paremmin pieniin harrasteprojekteihin sekä itse ohjelmointikielen perusteiden oppimiseen ja opettamiseen pienemmän koodimäärän ja yksinkertaisemman rakenteen vuoksi.

2.3 React-kirjasto

React on Facebookin (nyk. Meta) kehittämä JavaScript-kirjasto käyttöliittymien rakentamiseen modulaarisen, komponenttipohjaisen arkkitehtuurin avulla. React-komponentit ovat JavaScript-funktioita, jotka mahdollistavat käyttöliittymän osien kapseloinnin itsenäisiksi ja uudelleenkäytettäviksi osiksi. (Meta 2022.)

Esimerkki uudelleenkäytettävästä React-komponentista voisi olla komponentti ”Nappi”, jolla olisi parametrit ”teksti” ja ”toiminto”. Napilla olisi ennalta määritelty CSS-tyyli, HTML-rakenne ja JavaScript logiikkapohja, joten sovelluskehittäjän täytyisi vain syöttää komponentille napissa lukeva teksti sekä funktio, joka suoritetaan nappia painettaessa.

2.4 Next.js-kehys

Next.js on Vercelin kehittämä ja ylläpitämä avoimen lähdekoodin React-kehys, joka abstrahoi ja automatisoi Reactin tarvitsemia työkaluja tavoitteena tehdä ohjelmistokehityksestä suoraviivaisempaa (Vercel 2024).

Next.js sisältää esimerkiksi oman reititysratkaisun sille, miten sivujen ja toimintojen välillä navigoidaan. Kehys mahdollistaa myös sivujen palvelinpuolen renderöinnin (eng. server-side rendering, SSR), mikä parantaa sovelluksen hakukoneoptimointia ja varmistaa, että sisällön rakenne on johdonmukainen mm. sosiaalisessa mediassa jaettaessa (Risad 2023).

Kehyksen suosio on ollut tasaisessa kasvussa vuodesta 2020 eteenpäin ja vuoden 2024 ensimmäisen puoliskon ajan saavuttanut yli 5 miljoonaa viikoittaista latausta (NPM trends 2024a; NPM 2024a).

2.5 MariaDB

MariaDB on avoimen lähdekoodin SQL-relaatiotietokannan hallintajärjestelmä, joka on suunniteltu tehokkaaseen tietojen tallennukseen, hallintaan ja hakemiseen. (MariaDB Foundation 2024).

3 Automaatiotestaus

3.1 Määritelmä

Automaattiset testit ovat ohjelmia, jotka automatisoivat sovelluksen testauksen. Ne suorittavat testattavan sovelluksen toimintoja ja vertaavat saatuja tuloksia odotettuihin, ennalta määriteltyihin tuloksiin. Monimutkaisissa sovelluksissa testit eivät itsessään voi osoittaa, että sovellus toimisi täysin odotetusti. Sen sijaan testit ovat hyödyllisiä ilmoittamaan, kun sovellus ei toimi. (Da Costa 2021, 24–26.)

Automaatiotestaus ei kuitenkaan voi täysin korvata manuaalista testausta, esimerkiksi kun halutaan selvittää sovelluksen käyttökokemusta. Monet asiat, jotka ovat sovelluskehittäjille ja käyttöliittymien suunnittelijoille helposti ymmärrettäviä, saattavat olla vaikeasti hahmoteltavia sovelluksen kohdekäyttäjille, eivätkä automaattiset testit voi ottaa inhimillisiä näkökulmia huomioon. (Son 2023.)

3.2 Automaatiotestauksen hyötyjä sovelluskehityksessä

Automaatiotestaus mahdollistaa useiden sovelluksen toimintojen testaamisen ilman, että sovelluksen käyttöliittymää on vielä toteutettu. Tällöin toimivuus voidaan varmistaa ja tarvittaessa korjata paljon aikaisemmassa vaiheessa sovelluksen kehityskaarta. Kehityskaaren alkuvaiheissa lähdekoodin määrä sovelluksessa on pienempi, jolloin virheellisen koodin löytäminen on helpompaa. (Da Costa 2021, 27.)

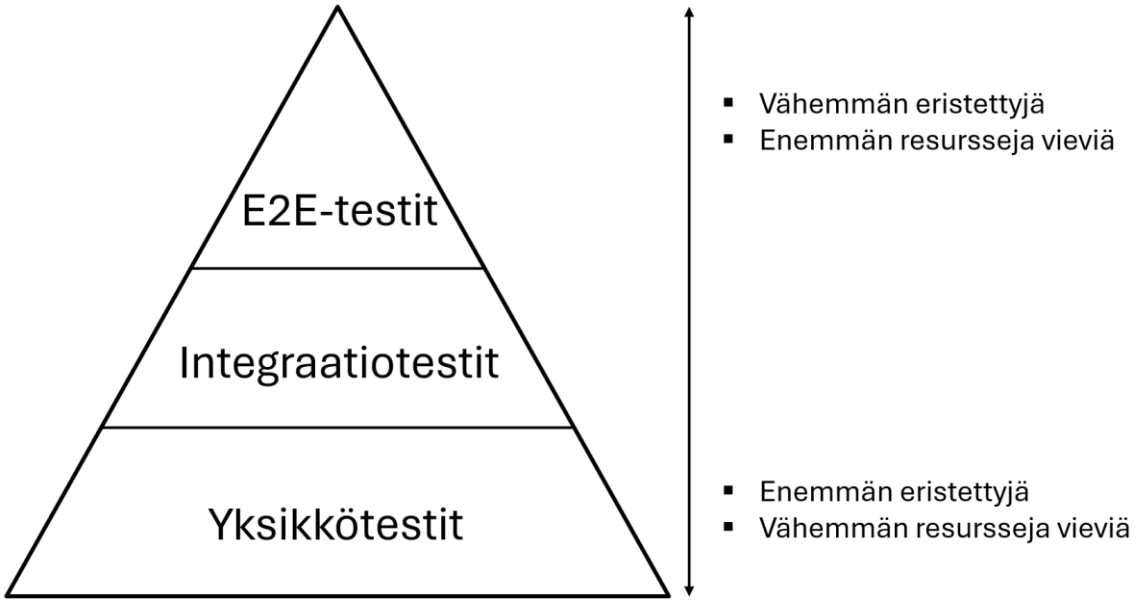
Automaatiotestit eivät ainoastaan varmista sovelluksen toimivuutta testien luontihetkellä, vaan auttavat tekemään sovelluksesta kestävämmän mahdollisille koodipohjan päivityksille. Monesti täysin toimivaan sovellukseen ilmestyy uusia virheitä, kun sovellukseen lisätään uusia ominaisuuksia.

Testit voivat myös tehdä lähdekoodista itseään dokumentoivaa. Ohjelmistoprojektiin saapuvien uusien työntekijöiden voi olla helpompi ymmärtää sovelluksen koodin toiminnallisuuta lukemalla sille luotuja testejä. (Meta 2024a.)

3.3 Automaatiotestaustyypit

Automaatiotestit on usein jaettu kolmeen pääkategoriaan: yksikkötesteihin (unit tests), integraatiotesteihin (integration tests) ja E2E (end-to-end) testeihin (Bose 2022).

Kuvassa 1. olevassa, alun perin Mike Cohnin luomassa testauspyramidikonseptissa havainnollistetaan, että yksikkötestit testaavat toisistaan eristettyjä toimintoja mahdollisimman pieninä yksikköinä. Yksikkötestien ajaminen on vähemmän laskentatehoa ja aikaa vievää mikä mahdollistaa testien ajamisen tiheämmin. Pyramidin toisessa päässä olevat E2E-testit taas voivat olla hyvinkin paljon laskentatehoa ja aikaa vieviä, jolloin niiden ajaminen tapahtuu testityypeistä harvimmin. E2E-testit toisaalta voivat olla hyvinkin kattavia kokonaisuuksia. (Vocke 2018.)



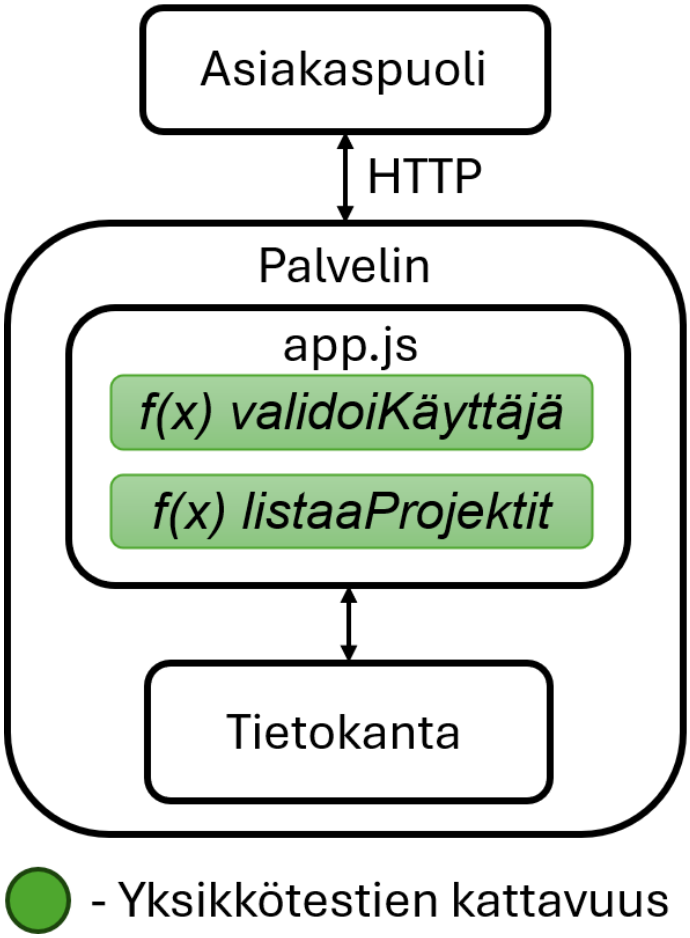
Kuva 1. Testauspyramidi.

Kyseisiin kategorioihin liittyvästi on hyvä huomioida, ettei niiden tulkinta ole aina johdonmukaista ja raja testaustyyppien välillä voi olla häilyvä (Da Costa 2021, 35). React-kirjaston omassa dokumentaatiossa pohditaan tapausta, jossa lomakekomponentin testaamisen sijoittuminen yksikkötestiksi tai integraatiotestiksi on häilyvää riippuen siitä, miten lomakefunktion testi reagoi lomakkeen sisältämän napin refaktorointiin (Meta 2024b).

3.3.1 Yksikkötestit

Yksikkötestin tehtävänä on testata pienintä mahdollista osaa sovelluksen lähdekoodissa, käytännössä yksittäistä funktiota, erillään muista sovelluksen osista. Yhtä funktiota kohden luodaan tavallisesti useampi yksikkötesti, jos funktio ottaa sisäänsä muuttujia, jotka vaikuttavat funktion toimintaan. Tällöin testin kohdatessa virhetilanteen on nopeampaa yksilöidä virheen syy verrattuna laajoihin, useaa väitettä (assertion) sisältäviin testeihin. (Bakharev 2023.)

Kuva 2. havainnollistaa sovellusta, joka sisältää tietokannan, palvelimen, käyttäjälle näkyvän asiakaspuolen käyttöliittymän sekä näiden välisen kommunikaation. Yksikkötestien kattavuutta sovelluksessa havainnollistetaan vihreäksi väritetyillä yksittäisillä funktioilla app.js-lähdekooditiedoston sisällä. Tässä tapauksessa yksikkötestien tarkoituksena on ainoastaan selvittää funktioiden `validoiKäyttäjä` sekä `listaaProjektit` toimivuus muusta sovelluksesta erillään.



Kuva 2. Esimerkki yksikkötestien kattavuudesta (Da Costa 2021, 38. Muokattu alkuperäisestä).

Yksikkötestien ajaminen on hyvin nopeaa, tyypillisesti muutamia millisekunteja. Tämä mahdollistaa yksikkötestien ajamisen tarvittaessa aina, kun lähdekoodiin tehdään muutoksia. Täten virhetilanteet yksittäisissä funktioissa voidaan havaita hyvin nopeasti, jos yksikkötestit ovat kattavia. Yksikkötestit voidaankin ottaa sovelluskehityksessä käyttöön hyvin aikaisessa vaiheessa sovelluksen kehityskaarta, sillä ne eivät ole riippuvaisia sovelluksen toimivuudesta kokonaisuutena toisin kuin E2E-testit. (Da Costa 2021, 37.)

Kuvassa 3 on esimerkkinä yksinkertainen JavaScript-funktio, joka palauttaa aina numeroarvon 2, sekä kaksi yksikkötestiä, joihin on määritelty kutsuttava funktio, funktion odotettu palautusarvo sekä testin epäonnistuessa tulostettu virheteksti. Testit ajettaessa konsoliin ilmestyy epäonnistuvaan testiin määritelty

virheviesti, sekä tietoa testin odotetun arvon ja funktion tuottaman arvon epäsuhdasta.

```
// Funktio, joka palauttaa arvon 2
function palauttaaKaksi() {
  return 2;
}

// Testitapaus, joka testaa, että funktio palauttaa arvon 2
assert.strictEqual(palauttaaKaksi(), 2, 'Funktio ei palauttanut lukua 2');
// Testitapaus, joka testaa, että funktio palauttaa arvon 3
assert.strictEqual(palauttaaKaksi(), 3, 'Funktio ei palauttanut lukua 3');
```

```
node:assert:125
  throw new AssertionError(obj);
  ^

AssertionError [ERR_ASSERTION]: Funktio ei palauttanut lukua 3
    at Object.<anonymous> (C:\Users\ville\Desktop\esimerkki\automaatiotesti.js:9:8)
    at Module._compile (node:internal/modules/cjs/loader:1369:14)
    at Module._extensions..js (node:internal/modules/cjs/loader:1427:10)
    at Module.load (node:internal/modules/cjs/loader:1201:32)
    at Module._load (node:internal/modules/cjs/loader:1017:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:122:12)
    at node:internal/main/run_main_module:28:49 {
  generatedMessage: false,
  code: 'ERR_ASSERTION',
  actual: 2,
  expected: 3,
  generatedMessage: false,
  code: 'ERR_ASSERTION',
  actual: 2,
  expected: 3,
```

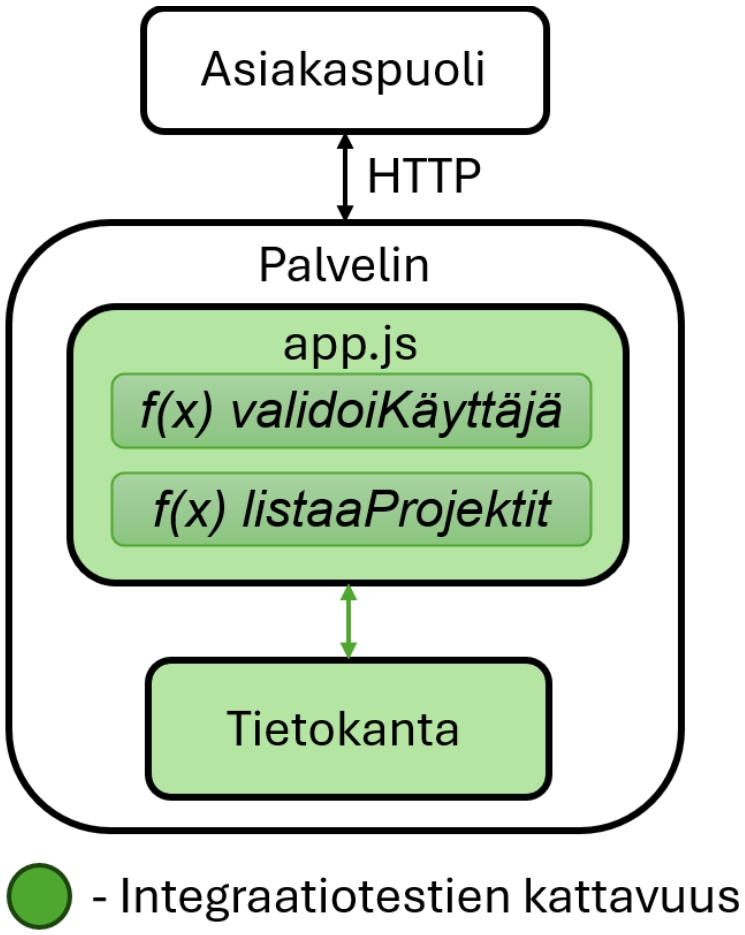
Kuva 3. Yksikkötestiesimerkki.

Testivetoisen kehityksen (test-driven development) ohjelmistokehitysmenetelmässä yksikkötestejä voidaan suunnitella ja toteuttaa jo ennen varsinaisten testattavien funktioiden ohjelmoimista. Tällöin määritellään testeillä ensin haluttu lopputulos funktiolle, jonka jälkeen funktiolle ohjelmoidaan toiminnallisuus, mikä johtaa haluttuun lopputulokseen. (Unadkat 2023.)

3.3.2 Integraatiotestit

Integraatiotestien tehtävänä on testata sovelluksen osia yhdessä kokonaisuuksina, ja varmistaa niiden toimivuus keskenään (Awati 2022).

Integraatiotestien mahdollista kattavuutta sovelluksessa havainnollistetaan kuvan 4. vihreäksi värjätyllä alueella. Kuvan sovelluksen integraatiotesti voisi esimerkiksi validoida tietokannassa oleva käyttäjä, ja tämän jälkeen listata tietokannassa olevat projektit, joiden työntekijä validoitu käyttäjä on. Esimerkkitesti ei kata asiakaspuolen käyttäjärajapintaa.



Kuva 4. Esimerkki integraatiotestien kattavuudesta (Da Costa 2021, 44. Muokattu alkuperäisestä).

Yksinkertaisimmillaan integraatiotesti voi kattaa kahden funktion yhteistoimintaa, kun taas monimutkaisimmillaan se voi olla hyvin laajojen palveluiden, ohjelmistokirjastojen ja sovellusten testaamista (Hu 2022).

Kuvan 5 esimerkissä on kaksi JavaScript-funktiota, joista ensimmäinen palauttaa luvun 2 ja toinen laskee kahden muuttujan summan. Kuvan yksinkertaisessa integraatiotestissä kutsumme funktion `yhteenlasku`

argumentteina funktiota `palauttaaKaksi` ja oletamme sen palauttavan lukuarvon 4. Ajettaessa testi odotetusti ei tuota virheilmoitusta.

```
// Funktio, joka palauttaa arvon 2
function palauttaaKaksi() {
  return 2;
}

// Funktio, joka laskee kahden luvun summan
function yhteenlasku(a, b) {
  return a + b;
}

// Testitapaus, joka testaa, että funktio palauttaa arvon 4
assert.strictEqual(
  yhteenlasku(palauttaaKaksi(), palauttaaKaksi()), 4,
  'Funktio ei palauttanut lukua 4'
);
```

Kuva 5. Integraatiotestiesimerkki.

Integraatiotestien ajaminen on usein enemmän laskentatehoa ja aikaa kuluttavaa kuin yksikkötestien. Tästä huolimatta kyse on keskimäärin edelleen vain muutamista sekunneista, joten yksikkötestien tavoin integraatiotestien ajaminen aina lähdekoodiin tehtyjen muutosten yhteydessä on usein käytännöllistä. (Da Costa 2021, 37.)

3.3.3 E2E-testit

E2E-testien tarkoituksena on testata sovelluksen kaikkia toiminnallisuuksia alusta loppuun. E2E-testit voidaan pääasiallisesti jakaa horisontaalisiin (eng. horizontal) ja vertikaalisiin (eng. vertical) testeihin. Toisin kuin yksikkötestien ja integraatiotestien tapauksessa, horisontaalisia E2E-testejä ei ajeta suoraan lähdekoodin funktioita käyttäen, vaan käyttäen samaa käyttöliittymää kuin sovelluksen kohderyhmänä olevat käyttäjät. (Gillis 2023.)

Siinä missä integraatiotesti saattaisi ajaa funktion, missä käyttäjän kirjautumistiedot sisältävä HTTP-pyyntö lähetetään palvelimelle ja tarkistetaan tästä seuraava vastaus, horisontaalisessa E2E-testissä avaisimme sovelluksen kirjautumisnäkymän verkkoselaimella, tarkistaisimme, että näkymä sisältää tekstisyöte-elementin, simuloisimme virtuaalisen näppäimistön kirjoittamaan tekstisyötteeseen kirjautumistiedot ja simuloisimme hiirikursorin klikkaamaan ”lähetä”-painiketta.

Vertikaalinen E2E-testi on käytännössä koko sovelluksen päästä päähän kattava integraatiotesti. Puhekielessä termillä ”E2E-testaus” viitataankin usein nimenomaan horisontaalisiin E2E-testeihin, ja tässä opinnäytetyössä keskitytään horisontaalisiin, verkkoselaimella näkyvän graafisen käyttöliittymän sisältäviin testeihin.

Kuvan 6 esimerkissä on Cypress-testikirjastolla luotuja E2E-testejä. Ohjelma tarkistaa testiselaimella käynnissä olevalta verkkosivulta, että käyttäjälle näkyy kirjautumislomake. Tämän jälkeen ohjelma syöttää lomakkeelle tietokannassa olevan käyttäjän tiedot oikein, klikkaa sisäänkirjautumispainiketta, ja olettaa saavansa oikean käyttäjälle näkyvän tervetuloviestin. Tämän jälkeen testi toistuu väärällä salasanalla, ja olettaa palvelimen vastauksen sisältävän virheilmoituksen. Viimeiseksi testi tarkistaa, ettei tervetuloviestiä näy virheellisen kirjautumisyrityksen yhteydessä.

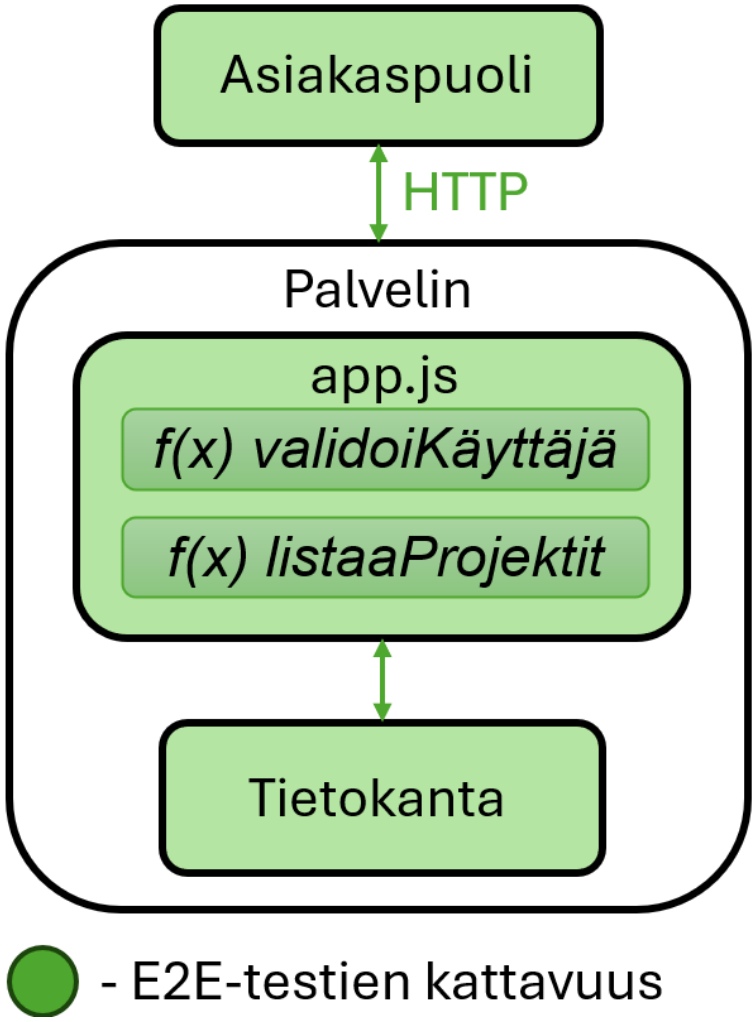
```
it('Kirjautumislomake on näkyvissä', function() {
  cy.contains('Kirjaudu sisään')
  cy.contains('käyttäjätunnus')
  cy.contains('salasana')
})

describe('Kirjautuminen sisään', function() {
  it('onnistuu oikeilla käyttäjätiedoilla', function() {
    cy.get('#username').type('Matti Meikäläinen')
    cy.get('#password').type('Salasana123')
    cy.get('#login-button').click()
    cy.get('.message')
      .should('contain', 'Tervetuloa Matti Meikäläinen')
      .and('have.css', 'color', 'rgb(0, 128, 0)')
  })

  it('epäonnistuu väärillä käyttäjätiedoilla', function() {
    cy.get('#username').type('Matti Meikäläinen')
    cy.get('#password').type('VääräSalasana321')
    cy.get('#login-button').click()
    cy.get('.error')
      .should('contain', 'Error: invalid username or password')
      .and('have.css', 'color', 'rgb(255, 0, 0)')

    cy.get('html').should('not.contain', 'Tervetuloa Matti Meikäläinen')
  })
})
```

Kuva 6. E2E-testiesimerkki.

E2E-testien ajaminen on yleensä todella hidasta verrattuna yksikkötesteihin ja yksinkertaisiin integraatiotesteihin. Tästä syystä niitä ajetaan yleensä harvoin, esimerkiksi vain, kun lähdekoodin muutoksia yhdistetään jaettuun tietovarastoon (repository) tai kun sovelluksen uusi versio on menossa tuotantoon. E2E-testien kirjoittaminen on yleensä myös automaatiotestausta rakentaessa viimeinen vaihe. (Da Costa 2021, 50.)

E2E-testien koko sovelluksen laajuista kattavuutta havainnollistetaan kuvassa 7. E2E-testi voisi esimerkiksi verkkoselaimella olevan graafisen käyttöliittymän kautta lähettää HTTP-pyynnön palvelimelle, joka validoisi käyttäjän, palauttaisi

listan tietokannassa olevista käyttäjän projekteista ja tarkistaisi, että projektit listataan käyttäjän verkkoselaimella olevaan näkymään.



Kuva 7. Esimerkki E2E-testien kattavuudesta (Da Costa 2021, 49. Muokattu alkuperäisestä).

Yksi heikkous, joka erityisesti käyttöliittymän kautta tapahtuvissa koko järjestelmän kattavissa testeissä voi esiintyä on, että testit saattavat tuottaa eri tuloksia toistuvasti ajettaessa, vaikka lähdekoodissa ei olisi tapahtunut muutoksia esimerkiksi asynkronisten toimintojen takia. Ilmiöön viitataan englannin kielen termillä ”flaky test”. Testatessa monimutkaisia kokonaisuuksia testiympäristön vakauden tärkeys korostuu ja haasteeksi nousee sovelluksen lähdekoodissa olevien vikojen lisäksi luotujen testien sisällä olevat viat. (Gitlab 2024.)

4 Testaustyökalut

Tässä luvussa olevaan tarkasteluun on valittu yleisimmin käytettyjä työkaluja JavaScript-pohjaisten sovellusten testaamiseen ohjelmistokehittäjäyhteisön antaman palautteen perusteella. State of JS 2022-kyselyn mukaan käytetyimmät testityökalut ovat Jest, Storybook, Mocha, Cypress, Testing Library, Puppeteer, Selenium, Jasmine, Playwright sekä Vitest (Greif ym. 2023). Edellä mainittujen erillisten testikirjastojen lisäksi Node.js versioista v.20 lähtien sisältää sisäänrakennetun testityökalun node:test (OpenJS Foundation 2024a).

Project Gate 2.0-sovellukseen implementoitavien testityökalujen olennaisimmat vaatimukset ovat helppokäyttöisyys sekä tuki käyttäjäyhteisön sekä dokumentaation muodossa. Tämä mahdollistaa sujuvan käyttöönoton opiskelijaympäristössä, missä projektin tulevat ylläpitäjät saattavat olla kokemattomia automaatiotestaamisen suhteen. Testityökalujen yhteisötuen laatua ja internetistä löytyvän tukimateriaalin saatavuutta kuhunkin mahdolliseen ongelmatilanteeseen on vaikeaa suoraan verrata, mutta korrelaationa voidaan pitää kunkin työkalun suosiota sekä maturiteettia. Mitä uudempi ja vähemmän käytetty työkalu on kyseessä, sitä todennäköisemmin ongelmatapauksien ratkaisuja on hankala löytää.

Teknisellä tasolla vaatimuksena on mahdollistaa kattavat yksikkö, sekä integraatiotestit palvelinpuolen toiminnoille Node.js ympäristössä, sekä React-komponenttitestit käyttöliittymäpuolella. Skaalautuvuus, suorituskyky ja mahdolliset muut ominaisuudet eivät ole prioriteetteina yhtä korkealla kuin helppokäyttöisyys ja tuen saatavuus, mutta otetaan työkalujen vertailussa huomioon.

4.1 Yksikkö- ja integraatiotestityökalut

Testaustyökaluja ei usein kategorisoida testaustyyppien mukaan, vaan niiden tarjoamien toiminnallisuuksien perusteella. Koska yksikkö- ja integraatiotestaaminen on funktioiden ja näiden yhteistoiminnan testaamista,

vaadittuja toiminnallisuuksia ovat väitekehys (assertion framework), joka määrittelee funktioiden odotetut tulokset ja vertaa niitä toteutuneisiin tuloksiin, sekä testiajurin (test runner), joka suorittaa testit ja raportoi väitekehykseen pohjaavat tulokset. (Zotti, 2015.)

4.1.1 Jest

Jest on Facebookin (nyk. Meta) kehittämä, erityisesti React-pohjaisten sovellusten testaamiseen luotu testaustyökalu. Jest on rakennettu Jasmine-kehyksen pohjalta (Sharanya, 2023a).

Jest on suunniteltu helppokäyttöiseksi eikä vaadi käyttäjältä paljon erillistä konfiguraatiota, mikä nopeuttaa testien toteuttamisen aloitusta. Jest pystyy suorittamaan testejä rinnakkaisesti, mikä mahdollistaa lyhyemmät testausajat. Testien rinnakkainen suoritus on kuitenkin ongelmallista, jos testit ovat yhteydessä tietokantaan, sillä tietokannan tila saattaa olla ennalta arvaamaton, jos Jest suorittaa testit ennalta määräämättömässä järjestyksessä kunkin testiajon aikana. (Luukkainen, 2023a.)

Jest on ollut käytetyin JavaScript-testikirjasto vuodesta 2019 lähtien (Greif ym. 2023). Täten yhteisöstä löytyvä tuki on laajaa ja ratkaisujen etsiminen ongelmatilanteisiin internetistä todennäköisempää kuin vähemmän suosituilla työkaluilla.

4.1.2 Mocha

Mocha on OpenJS Foundationin kehittämä ja ylläpitämä JavaScript testityökalu, joka on kehitetty erityisesti Node.js sovelluksille sekä selainpohjaisille sovelluksille (OpenJS Foundation 2024b).

Toisin kuin Jest, Mocha vaatii käyttäjältä enemmän konfigurointia. Se ei sisällä omaa väitekehystä tai mockaustyökaluja, vaan ne on erikseen määriteltävä ulkopuolisilla kirjastoilla ja lisäosilla. Mochan hyviin puoliin kuuluukin sen

joustavuus ja laajennettavuus, mutta se ei ole käytännöllinen ”out-of-the-box” työkaluna. (Sharanya, 2023b.)

4.1.3 Jasmine

Jasmine on avoimen lähdekoodin JavaScript testityökalu, jonka tavoitteena on tuottaa selkeää ja helppolukuista syntaksia testejä kirjoittaessa (Jasmine 2024).

Jasmine sisältyy Angular-kehyksellä luotuihin projekteihin riippuvuutena ja valmiiksi konfiguroituna oletusarvoisesti. Täten Angular-pohjaista sovellusta kehitettäessä Jasmine on mahdollisesti yksinkertaisin ja paras vaihtoehto jos testejä halutaan toteuttaa nopeasti. (Obregon, 2023.)

4.1.4 Node test runner

Node test runner on uudemmissa Node.js versioissa sisäänrakennettu testityökalu, mikä on mahdollistanut työkalun optimoinnin Node.js sovellusten testaamiseen kyseisessä ympäristössä. Koska Node test runner on natiivisti osa Node.js:ää, se mahdollistaa testien toteuttamisen ilman ulkopuolisia testityökaluja täten vähentäen riippuvuuksia. (Godwin 2023.)

Huomioitava haittapuoli on, että kirjaston vakaa versio on ollut saatavilla vasta vuoden 2023 huhtikuusta lähtien (Node 2023). Täten kirjastoa koskeva yhteisökeskustelu esimerkiksi virhetilanteiden osalta on melko vähäistä ja kokeneita käyttäjiä ei vielä juuri ollenkaan.

4.1.5 Vitest

Vitest on Vite-kehitystyökalun kehittäjien luoma testikirjasto (Vitest 2024).

Vitest käyttää samaa ohjelmointirajapintaa kuin Jest, joten testien siirtäminen näiden kahden työkalun välillä on suoraviivaista. Käytännössä näkyvin ero Jestin ja Vitestin välillä on konfiguraatioissa. Vitest on rakennettu erityisesti Vite-

kehitystyökalulla rakennettujen sovellusten testaamiseen, mutta sitä voidaan käyttää Jestin tapaan myös muissa JavaScript-pohjaisissa sovelluksissa. Vite-pohjaisten sovelluksien testaaminen Vitestillä vaatii kuitenkin vähemmän konfiguraatiota käyttäjältä ja työkalu on optimoitu kyseiseen ympäristöön. (Tozzi 2023.)

4.1.6 Testing Library ja React Testing Library

Testing Library on kokoelma testauskirjastoja käyttöliittymäkomponenttien testaamiseen. Testing Libraryn kirjastoja käytetään yleisesti ulkoisten testiajurien, kuten esimerkiksi tässä kappaleessa listattujen Jestin ja Mochan kanssa (Testing Library 2021).

Project Gate 2.0-sovelluksen ollessa Next.js pohjainen on Testing Library-kokoelmaan kuuluva React Testing Library huomionarvoinen. React Testing Library on erityisesti React-komponenttien renderöintiin testaamista varten luotu kirjasto (Testing Library 2024).

4.1.7 Yksikkö- ja integraatiotestityökalujen vertailu

Helppokäyttöisyyden arviointiin on otettu huomioon kunkin työkalun käyttövalmius mahdollisimman pienellä konfiguraatiolla ja ulkoisten riippuvuuksien määrä, mikä vaikuttaa käytettävän dokumentaation määrään ja hajanaisuuteen. Tältä osin Mocha edustaa yhtä ääripäätä, missä konfigurointi ja joustavuus tekevät siitä monimutkaisemman verrattuna Jasminen, Jestin tai Vitestin tapaisiin työkaluihin, missä kaikki tarvittava toiminnallisuus testien kirjoittamiseen on valmiiksi integroituna.

Yhteisön ja tuen tason arviointiin on käytetty State of JS-kehittäjäkyselyn tuloksia testityökalujen suosiosta vuosittain (Greif ym. 2023). Taulukko 1. havainnollistaa kuinka monta prosenttia vastaajista käytti kutakin arvioitua työkalua vuosittain. Jest on ollut selvästi suosituin vuodesta 2019 eteenpäin. Mocha ja Jasmine ovat olleet tasaisen suosittuja kirjastoja jo pitkään. Node test

runner ja Vitest ovat uusia tulokkaita, joten dataa ei Node test runnerin suhteen kyselyssä vielä ollut ja Vitest on ollut laajemmassa käytössä vasta vuodesta 2022.

Taulukko 1. Testityökalujen vuosittaiset käyttöprosentit State of JS -sovelluskehittäjäkyselyssä.

	2017	2018	2019	2020	2021	2022
Jest	26,4	41,1	63,6	68,3	72,7	68,3
Mocha	51,8	47,9	54,1	52,8	50,3	43,8
Jasmine	49,0	38,2	42,8	42,1	39,9	32,6
Vitest	0,0	0,0	0,0	0,0	2,7	14,1
Node test runner	0,0	0,0	0,0	0,0	0,0	0,0

Taulukkoon 2 on koottu tiivistetysti johtopäätökset kunkin vertailtavassa olevan työkalun helppokäyttöisyydestä, yhteisön koon ja tuen saavutettavuudesta sekä tärkeimmistä työkalukohtaisista ominaisuuksista.

Taulukko 2. Työkalujen vertailutaulukko.

	Helppokäyttöisyys	Yhteisö ja tuki	Ominaisuudet
Jest	Hyvä	Erinomainen	Testien siirto Vitestiin helppoa. Sopii hyvin React-pohjaisten sovellusten testaamiseen.
Mocha	Kohtalainen	Hyvä	Erittäin mukautettava ulkopuolisilla plugineilla ja kirjastoilla.
Jasmine	Hyvä	Hyvä	Sopii hyvin Angular-pohjaisten sovellusten testaamiseen.
Node test runner	Hyvä	Heikko	Sisäänrakannettu Nodeen, vähemmän ulkopuolisia riippuvuuksia.
Vitest	Hyvä	Heikko	Testien siirto Jestiin helppoa. Helppo integroida ja konfiguroida Vite-pohjaisiin sovelluksiin.

4.2 E2E-testityökalut

Käyttöliittymän kautta tapahtuvaan E2E-testaukseen tarvittavien työkalujen pääasiallinen lisävaatimus on verkkoselaimen automatisoidun käytön mahdollistaminen, jotta testitapauksia voidaan simuloida niin kuin kohdekäyttäjän oletetaan sovellusta käyttävän (Testim 2020).

4.2.1 Selenium

Selenium on Thoughtworksin kehittämä avoimen lähdekoodin verkkoselainten automaatiotyökalu (Selenium 2024a). Seleniumia käytetään yleisesti web-applikaatioiden testaamiseen verkkoselaimen kautta. Selenium voi emuloida selaimen käyttöliittymätoimintoja, kuten hiiren klikkauksia ja ruudulla näkyvän sisällön tunnistamista. Selenium julkaistiin alun perin vuonna 2004 ja se on modernien selainpohjaisten E2E-työkalujen edeltäjä. (Da Costa 2021, 303.)

Yleisten käyttäjätoimintojen lisäksi Selenium tarjoaa kattavasti muita selaimen kanssa vuorovaikuttavia toimintoja kuten videoiden tallentamista, kuvakaappausten ottamista sekä verkkoyhteyden nopeuden rajoittamista. Täten esimerkiksi testien luominen hitaita internetyhteyksiä silmällä pitäen on mahdollista. (Da Costa 2021, 304.)

Selenium ei itsessään ole testaustyökalu vaan automaatiotyökalu eikä se itsessään sisällä testiajureita tai väitekehystä, joten testien luomista varten Selenium tarvitsee ulkopuolisia työkaluja kuten esimerkiksi Jestiä tai Mochaa (Da Costa 2021, 305). Tämän takia Seleniumin käyttöönotto voi olla hieman monimutkaisempaa kuin vastaavien työkalujen, joissa on sisäänrakennetut testityökalut.

Selenium käyttää erillistä WebDriver-nimistä ohjelmointirajapintaa kommunikointiin eri verkkoselaimien ajureiden kanssa (Selenium 2024b). Kaikille suosituille selaimille on oma WebDriver rajapintansa. Seleniumin suuri etu onkin sen laaja tuki eri selaimille, joka voi olla tärkeää jos testattavan sovelluksen halutaan toimivan täysin odotetusti kaikilla määritetyillä selaimilla.

Selenium keskustelee selaimien kanssa HTTP-pyyntöjen muodossa, mikä tekee siitä suorituskyvyltään hitaamman kuin vaihtoehtoiset täysin selaimen sisäisesti ajettavat työkalut (Da Costa 2021, 306).

4.2.2 Puppeteer

Puppeteer on Googlen kehittämä ja ylläpitämä verkkoselainten automaatiotyökalu. Tosin kuin Selenium, Puppeteer tukee vakaasti ainoastaan Googlen Chrome ja Chromium selaimia. Puppeteer käyttää oletusarvoisesti ns. Headless-moodia, eli Chromea ilman graafista käyttöliittymää (Google 2024a).

Koska Puppeteer on rakennettu tukemaan natiivisti Googlen selaimia, se ei tarvitse WebDriverin tapaisia ulkopuolisia työkaluja voidakseen kommunikoida selaimen kanssa. Seleniumin tapaan Puppeteer ei kuitenkaan sisällä omia testaustyökaluja, vaan vaatii testien suorittamiseen ulkopuolisia testikirjastoja (Da Costa 2021, 306).

Koska Puppeteer käyttää suoraan Chromea, on virhetilanteiden debuggaaminen mahdollista käyttäen Chromen kehittäjätyökaluja. Tämä mahdollistaa selaimen toimintojen helpomman seuraamisen testejä ajettaessa. (Da Costa 2021, 307).

Versiosta 2.1.0 lähtien Puppeteer on tukenut kokeellisesti myös Firefox-selainta. Google aikoo laajentaa tuen myös muille selaimille kuten Safarille (Google 2024b). Tulevaisuudessa Puppeteer saattaa siis olla varteenotettava vaihtoehto Seleniumille useimpia selaimia testatessa.

4.2.3 Cypress

Cypress on Cypress.io:n kehittämä ja ylläpitämä testaustyökalu (Cypress.io 2024a). Cypress on suoraan yhteydessä selaimen etäohjaus API:in toimintojen suorittamiseksi. Tämä mahdollistaa nopeamman testien suorittamisen sekä yksinkertaisemman testiympäristön konfiguroinnin (Da Costa 2021, 307).

Seleniumista ja Puppeteerista poiketen Cypress on täysin automaatiotestaamiseen suunniteltu ja sisältää kattavasti työkaluja testien luomiseen ilman tarvetta ulkopuolisille kirjastoille. Koska Cypressin

ohjelmointirajapinta on täysin testaamista varten suunniteltu, Cypressillä toteutetut testit ovat usein tiiviitä ja helppolukuisia (Cypress.io 2024b).

4.2.4 Playwright

Playwright on Microsoftin kehittämä ja ylläpitämä, avoimen lähdekoodin automaatiokehys E2E-testaukseen, joka tukee kaikkia yleisimmin käytettyjä selaimia kuten Chromiumia, Firefoxia ja Safaria (Microsoft 2024b). Playwright tarjoaa Cypressin tapaan kattavasti testausominaisuuksia ilman tarvetta ulkoisille riippuvuuksille.

Playwright julkaistiin alun perin vuonna 2020 ja on täten listatuista E2E-työkaluista selvästi uusin. Tästä huolimatta Playwrightin suosion kasvu on ollut todella nopeaa ja Node.js pakettilatauksien statistiikkaa tarkasteltaessa sen suosio on saavuttanut vuoden 2024 aikana Puppeteerin ja Cypressin (NPM trends 2024b).

4.2.5 E2E-työkalujen vertailu

Samoin kuin yksikkö- ja integraatiotestaustyökalujen kanssa, helppokäyttöisyyden arviointiin on otettu huomioon kunkin työkalun käyttövalmius mahdollisimman pienellä konfiguraatiolla ja ulkoisten riippuvuuksien määrä, mikä vaikuttaa käytettävän dokumentaation määrään ja hajanaisuuteen

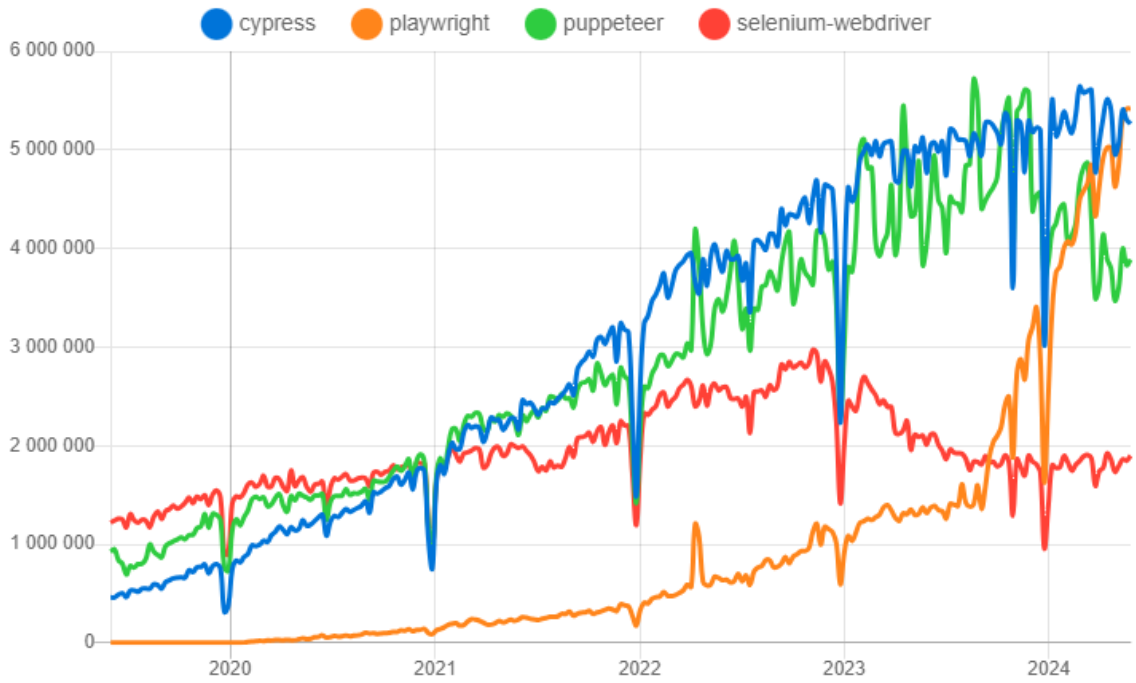
Yhteisön ja tuen tason arviointiin on käytetty State of JS-kehittäjäkyselyn tuloksia testityökalujen suosiosta vuosittain (Greif ym. 2023). Taulukko 3. havainnollistaa kuinka monta prosenttia vastaajista käytti kutakin arvioitua työkalua vuosittain.

Taulukko 3. E2E-testityökalujen vuosittaiset käyttöprosentit State of JS -sovelluskehittäjäkyselyssä.

	2019	2020	2021	2022
Cypress	25,6	35,1	42,8	42,3
Puppeteer	27,3	33,5	37,0	34,4
Playwright	0,0	3,4	9,5	15,8
Selenium	0,0	0,0	0,0	33,5

Data on kuitenkin rajallisempaa E2E-työkalujen osalta verrattuna yksikkö- ja integraatiotestityökaluihin, sillä ne ovat sisällytetty kyselyyn vasta vuodesta 2019, ja Selenium JavaScriptistä irrallisena työkaluna sisällytettiin vasta vuodesta 2022 eteenpäin.

Node.js pakettilatausstatistiikasta voidaan kuitenkin tarkastella Seleniumin ja muiden työkalujen suosiota paremmin pidemmällä aikavälillä JavaScript käyttöympäristössä. Kuvio 1. havainnollistaa Cypress, Playwright, Puppeteer sekä Selenium NPM-pakettien latausmääriä viimeisen viiden vuoden ajalta. Graafista nähdään Seleniumin olleen pitkään tasaisen suosittu, mutta vuodesta 2023 lähtien jääneen muista työkaluista jälkeen.



Kuvio 1. E2E-työkalujen latausstatistiikkaa (NPM trends 2024b).

Taulukkoon 4 on koottu tiivistetysti johtopäätökset vertailtavissa olevien E2E-työkalujen helppokäyttöisyydestä, yhteisön koon ja tuen saatavuudesta sekä työkalukohtaisista ominaisuuksista.

Taulukko 4. E2E-työkalujen vertailutaulukko.

	Helppokäyttöisyys	Yhteisö ja tuki	Ominaisuudet
Selenium	Kohtalainen	Hyvä	Tukee useaa selainta. Vaatii ulkopuolisia testityökaluja.
Puppeteer	Kohtalainen	Hyvä	Tukee vakaasti vain Chromea. Vaatii ulkopuolisia testityökaluja.
Cypress	Hyvä	Hyvä	Tukee useaa selainta. Ei vaadi ulkopuolisia testityökaluja.
Playwright	Hyvä	Kohtalainen	Tukee useaa selainta. Ei vaadi ulkopuolisia testityökaluja

5 Automaatiotestien suunnittelu sovellukseen

5.1 Project Gate 2.0-sovelluksen tavoite

Project Gate 2.0-sovelluksen käyttötarkoitus on mahdollistaa Turun ammattikorkeakoulussa toimivan opiskelijakeskeisen projektitoimisto theFIRMAn projektien ja niissä toimivien työntekijöiden hallinnointi. Sovelluksen kohderyhmät ovat Turun AMK:n opettajat sekä theFIRMAn vakituiset työntekijät, jotka toimisivat sovelluksessa ylläpitäjien oikeuksilla, sekä projekteissa työskentelevät opiskelijat, jotka toimisivat sovelluksessa normaalin käyttäjän oikeuksilla.

5.2 Sovelluksen käyttötapaukset

Sovelluksen automaatiotestien suunnittelua varten on kartoitettu sovelluksen tärkeimmät käyttötapaukset. Toiminnot, joita palvelinpuolen koodin täytyy voida suorittaa ilman graafisen käyttöliittymän käyttöä, on eritelty käyttäjäryhmien pääsyoikeuksien mukaan. Näille toiminnoille toteutetaan yksikkö ja integraatiotestit.

Uuden käyttäjän, jolla ei ole tunnuksia, tulisi voida:

- Luoda uusi käyttäjätili, jolla on opiskelijan pääsyoikeudet.

Käyttäjän, jolla on opiskelijan pääsyoikeudet, tulisi voida:

- Kirjautua sisään sovellukseen käyttäjätunnuksilla
- Hakea lista käyttäjistä, jotka kuuluvat henkilökuntaan tai ovat projektipäälliköitä
- Hakea lista kaikista projekteista
- Hakea lista projekteista, joissa on työntekijänä
- Hakea yksittäisen projektin tiedot
- Lähettää projektihakemus
- Hakea lista kaikista kursseista

- Hakea lista kaikista tapahtumista
- Hakea omat käyttäjätiedot
- Muokata omia käyttäjätietoja

Käyttäjän, jolla on ylläpitäjän pääsyoikeudet, tulisi pystyä kaikkien opiskelijan pääsyoikeuden sisältävien toimintojen lisäksi voida:

- Hakea lista kaikista käyttäjistä
- Luoda uusi projekti
- Muokata projektin tietoja
- Poistaa projekti
- Lisätä käyttäjä projektin työntekijäksi
- Poistaa käyttäjä projektista
- Hakea lista kaikista projektihakemuksista
- Hyväksyä projektihakemus
- Hylätä projektihakemus
- Lisätä uusi kurssi
- Muokata kurssin tietoja
- Poistaa kurssi
- Lisätä uusi tapahtuma
- Muokata tapahtuman tietoja
- Poistaa tapahtuma

5.3 Testitapaukset käyttötapauksista

Kustakin käyttötapauksesta on eritelty sarja testitapauksia, missä pyritään kattamaan keskeiset hyväksytyt tilanteet sekä virhetilanteet. Taulukko 5 sisältää otteen testitapauksia uuden käyttäjän rekisteröinnistä, esivaatimukset ennen kunkin testin suorittamista, sekä testin odotetun tuloksen. Testeihin, joiden odotettuna tuloksena on HTTP-vastaus, on pyritty selvittämään sopivin HTTP-tilakoodi.

Taulukko 5. Testitapauksia uuden käyttäjän rekisteröinnistä.

ID	Tapaus	Esivaatimukset	Odotettu tulos
1	Rekisteröidään onnistuneesti uusi käyttäjä	-	HTTP 201
2	Yritetään käyttäjän rekisteröintiä jo käytössä olevalla sähköpostilla	Tietokanta sisältää käyttäjän samalla sähköpostilla	HTTP 409
3	Yritetään käyttäjän rekisteröintiä ilman sähköpostia	-	HTTP 400
4	Yritetään käyttäjän rekisteröintiä ilman salasanaa	-	HTTP 400
5	Yritetään käyttäjän rekisteröintiä liian lyhyellä salasanalla	-	HTTP 400
6	Yritetään käyttäjän rekisteröintiä ilman erikoismerkkejä salasanassa	-	HTTP 400
7	Yritetään käyttäjän rekisteröintiä ilman isoa kirjainta salasanassa	-	HTTP 400

Virhetilanteen aiheuttavia testitapauksia laatiessa on pyritty tehokkuuteen ja pitämään testitapausten määrä kohtuullisena kuitenkin sisältäen yleisimmät sekä sovelluksen toiminnallisuutta haittaavat tapaukset. Testitapauksia laatiessa esimerkiksi kyseenalaistin tarvetta testata, että rekisteröitymisen

yhteydessä käyttäjän oikean nimen joukkoon ei vahingossa eksy erikoismerkkiä, jos käyttäjällä on mahdollisuus itse korjata virhe jälkikäteen käyttäjätietoja muokkaamalla, eikä se vaikuta sovelluksen muuhun toiminnallisuuteen millään lailla.

HTTP-tilakoodien määrittelemisen tarve odotettuihin tuloksiin tuli esiin testejä luodessa, sillä useat virhetilanteet aiheuttivat palvelimella yleisluonteisen, sisäistä palvelinvirhettä tarkoittavan `500-Internal Server Error`-vastauksen. Tämä on ongelmallista, sillä virhetilakoodien tulisi ilmoittaa mahdollisimman selkeästi virhetilanteen syy, jotta ongelman löytäminen ja korjaaminen olisi helpompaa (Moore 2023). Ilman tarkasti määriteltyjä tilakoodeja ja vastausviestejä on myös käytännössä mahdotonta olla varma, että esimerkiksi automaatiotestin odottama virhetulos on aiheutunut testattavan testitapauksen takia.

5.4 E2E-testitapaukset

E2E-testitapauksien suunnittelun ensimmäiset versiot tehtiin graafisesti Figma-työkalulla tehtyjen käyttöliittymäprototyyppien pohjalle. Tämä lähestymistapa osoittautui kuitenkin aikaa vieväksi ja sekavaksi, etenkin kun tiedossa oli käyttöliittymäprototyyppien vaihtuvuus sovelluskehityksen edetessä. Käyttöliittymäsuunnitelmien vaihtuvuus myös toi omat haasteensa testitapausten suunnitteluun.

Ratkaisuksi päädyttiin tekemään yksinkertaiset taulukoidut testitapaukset prototyyppien pohjalta, joissa määriteltiin automatisoidun selaimen kautta tapahtuvan testin eteneminen askeleittain. Täten testitapausten askelten muokkaaminen tarpeen mukaan on nopeaa, ja mahdollistaa graafisen esityksen tuomisen tarvittaessa myöhemmin käyttöliittymäsuunnittelun valmistuttua.

Taulukko 6. sisältää testitapauksen uuden käyttäjän rekisteröintiin, olemassa olevan käyttäjän sisäänkirjautumiseen ja näiden testitapausten verkkoselaimella tapahtuvat askeleet.

Taulukko 6. E2E-askeltaulukko

ID	Tapaus	Esivaatimukset	Askeleet
UI1	Uuden käyttäjän rekisteröinti	-	1. Klikkaa ’here’-linkkielementtiä 2. Täytä sähköpostikenttä 3. Täytä nimikenttä 4. Täytä salasanakenttä 5. Täytä ’toista salasana’-kenttä 6. Klikkaa ’Create Account’-nappia 7. Tarkista teksti ’Welcome to Project Gate!’
UI2	Kirjautuminen sisään olemassa olevalla käyttäjällä	Tietokanta sisältää käyttäjän	1. Täytä sähköpostikenttä 2. Täytä salasanakenttä 3. Klikkaa ’login’-nappia 4. Tarkista teksti ’Welcome to Project Gate!’

6 Automaatiotestauksen implementointi projektiin

6.1 Käytettävät testityökalut

Käytettäviksi testaustyökaluiksi valittiin Jest ja Cypress. Pääasiallisina valintakriteereinä toimivat laaja yhteisötuki, helppokäyttöisyys, hyvä dokumentaatio sekä Jestin tapauksessa projektissa toimineiden kokemus kirjaston käytöstä automaatiotestaamisessa aiemmissa projekteissa. Näiden lisäksi Jestin erityinen suhde React-kirjastoon molempien ollessa Metan ylläpitämiä tuotteita luo puitteet myös tulevalle yhteensopivuudelle.

6.2 Testiympäristö ja testitietokanta

Node.js-pohjaisille sovelluksille on tyypillistä asettaa ympäristömuuttuja `NODE_ENV`, joka määrittää missä moodissa sovellusta suoritetaan. Normaali käytäntö on määritellä erilliset moodit tuotantokäyttöön, kehitykseen sekä testaamiseen (Luukkainen, 2024).

Ympäristömuuttujan avulla sovellukseen voidaan luoda logiikkaa ja toimintoja, joita haluamme käyttää vain automaatiotestejä suorittaessa. Hyvä esimerkki tästä on erillisen testitietokannan käyttäminen, johon sovellus on yhteydessä vain, kun se suoritetaan testausmoodissa. Eristetyn testitietokannan käyttö mahdollistaa tietokannan alustamisen aina ennen testien ajamista, jolloin tietokannan tila on ennalta määritelty.

Kuva 8. näyttää sovellukselle määritetyt komentosarjat eri moodeissa käynnistämiseen. Komennolla `test:start` määritetään sovellus testimoodiin ja käynnistetään palvelin. Testimoodissa käynnistyvä sovellus asettaa ehtolausetta käyttäen `DATABASE_URL`-ympäristömuuttujan arvoksi testitietokannan URL-osoitteen ja API-palveluun rekisteröidään reitti testitietokannan alustamista varten. Cross-env kirjasto varmistaa, että komentosarja ja siihen liitetyt ympäristömuuttujat toimivat sekä Windows, Mac että Linux ympäristöissä (NPM 2024b).

```
"scripts": {
  "dev": "cross-env NODE_ENV=development nodemon src/index.ts",
  "start": "cross-env NODE_ENV=production node dist/index.js",
  "test:start": "cross-env NODE_ENV=test nodemon src/index.ts",
```

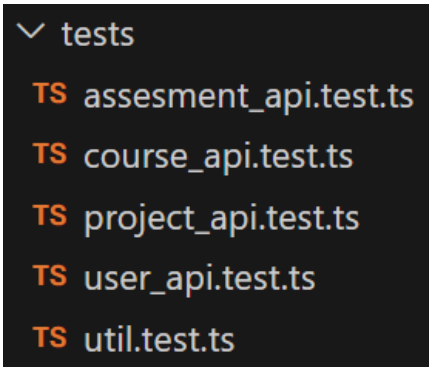
```
if (process.env.NODE_ENV === 'test') {
  console.log('NODE ENV IS TEST')
  process.env.DATABASE_URL = process.env.TEST_DATABASE_URL;
} else {
  console.log('NODE ENV IS DEV OR PRODUCTION')
  process.env.DATABASE_URL = process.env.DEV_DATABASE_URL;
}
```

```
if (process.env.NODE_ENV === "test") {
  app.register(resetRoutes.plugin, {
    prefix: resetRoutes.PREFIX_ROUTE,
  });
}
```

Kuva 8. Testiympäristön määrittäminen lähdekoodissa

6.3 Yksikkö- ja integraatiotestien implementointi

Jestillä toteutettavat testit jaettiin tietokannassa olevien mallien mukaan omiin testitiedostoihinsa kuvan 9. mukaisesti. Kuhunkin testitiedostoon sisällytettiin toteutetut mallikohtaisien apufunktioiden yksikkötestit ja mallien RESTful API-rajapinnan integraatiotestit. Näiden lisäksi luotiin erillinen `util.test.ts`-tiedosto, johon sisällytettiin malleihin liittymättömien tai näiden välillä jaettujen apufunktioiden, kuten esimerkiksi tietokantaan tallennettavien Snowflake-algoritmilla luotavien id-arvojen luomiseen käytettävät testit.



Kuva 9. Jest-testitiedostoja

Kuvassa 10. on Jestillä kirjoitettuja integraatiotestejä käyttäjien rekisteröimiseen ja sisäänkirjautumiseen. Ennen jokaista testiä tietokanta alustetaan `beforeEach`-lohkossa. Ensimmäinen it-lohko testaa onnistuneen rekisteröitymisen. Toinen it-lohko testaa epäonnistuneen rekisteröitymisen varatulla sähköpostilla. Kolmas it-lohko testaa epäonnistuneen rekisteröitymisen puuttuvalla sähköpostilla.

```
describe('User registration and login', () => {

  // Ennen jokaista testiä alustetaan tietokanta
  beforeEach(async () => {
    await supertest('http://localhost:3001').get('/api/v1/reset');
  })

  const testUser = {
    "firstName": "Matti",
    "lastName": "Meikäläinen",
    "email": "matti.meikalainen@turkuamk.fi",
    "password": "Password123",
    "enrollmentYear": 2023
  }

  // 1-Testataan uuden käyttäjän rekisteröinti
  it('should register a new user', async () => {
    const result = await supertest('http://localhost:3001')
      .post('/api/v1/user')
      .send(testUser)
      .expect(201)
  })

  // 2-Yritetään rekisteröidä käyttäjä, jonka email on jo käytössä
  it('should not register a user with an email that is already in use', async () => {
    await supertest('http://localhost:3001')
      .post('/api/v1/user')
      .send(testUser).expect(201)
    await supertest('http://localhost:3001')
      .post('/api/v1/user')
      .send(testUser).expect(409)
  })

  // 3-Yritetään rekisteröidä käyttäjä ilman emailia
  it('should not register a user without an email', async () => {
    const { email, ...testUserWithoutEmail } = testUser;

    await supertest('http://localhost:3001')
      .post('/api/v1/user')
      .send(testUserWithoutEmail).expect(400)
  })
```

Kuva 10. Jestin integraatiotestejä.

6.4 E2E-testien implementointi

Cypress-testejä luodessa esiintyi kaksi pääasiallista huomiota sovelluksen tulevaa ylläpitoa koskien. Useasta React-komponentista puuttui yksilöivä ID-arvo. Cypress kykenee tunnistamaan sivun elementtejä esimerkiksi napeissa tai

linkeissä olevan tekstin perusteella, mutta ilman ID-arvoja esimerkiksi rekisteröintilomakkeella oleville tekstisyötekentille tehtävät toiminnot olisi määriteltävä sen perusteella, missä järjestyksessä syötekentät sivulla esiintyvät, tai että kenttiä olisi vain yksi. Tämä on ongelmallista, koska testit menevät rikki, jos syötekenttien määrä tai järjestys sivulla muuttuu sovelluksen kehityksen aikana.

Toisena huomiona osa testeistä kohtasi odottamattoman virhetilanteen, kun testiympäristö käynnistettiin uudemmalla Node.js-versiolla kuin millä se oli alun perin kehitetty. Sovellusajoympäristön eroista johtuvia, odottamattomia virhetilanteita pystyy välttämään käyttämällä esimerkiksi kontteja (containers) määrittämään ympäristön, missä testit ajetaan (Docker 2024). Näin voidaan varmistaa, että testit toimivat odotetulla tavalla sekä sovelluksen kehitysvaiheessa, että tuotantoon viedessä.

Kuvassa 9. on osa luoduista E2E-testeistä. Alun `beforeEach`-lohko alustaa testitietokannan ja siirtyy sovelluksen kirjautumissivulle ennen jokaista testiä.

Ensimmäinen `it`-lohko testaa, että odotetut tekstielementit ovat näkyvillä kirjautumissivulla `contains`-funktiota käyttäen. Toinen `it`-lohko emuloi hiiren klikkaamista `click`-funktiolla linkkielementtiin, joka sisältää tekstin ’here’, ja tarkistaa että näkymä onnistuneesti siirtyy rekisteröintisivulle.

Kolmas `it`-lohko siirtyy rekisteröintisivulle, valitsee ja täyttää tekstisyötekenttiä ID-arvon perusteella ja klikkaa rekisteröitymispainiketta. Onnistunut rekisteröinti tarkistetaan tervetuloviestin perusteella, jonka jälkeen testataan sivun yläpalkin linkkien toimivuus. Viimeiseksi testi siirtyy kirjautuneen käyttäjän profiilisivulle dropdown-valikon kautta.

```
beforeEach(() => {
  cy.request('GET', 'http://localhost:3001/api/v1/reset')
  cy.visit('http://localhost:3000')
})

it('login page shows correct text', () => {
  cy.contains('Email address')
  cy.contains('Password')
  cy.contains('New to Project Gate? Register here')
})

it('register a new user page works', () => {
  cy.contains('here').click()
  cy.contains('Create Account')
})

it('registering a new user and navigating the main pages works', () => {
  cy.contains('here').click()
  cy.get('#firstname').type('Ville')
  cy.get('#lastname').type('Laitinen')
  cy.get('#email').type('ville-laitinen@turkuamk.fi')
  cy.get('#password').type('Salasana123')
  cy.get('#confirmpassword').type('Salasana123')
  cy.contains('Create Account').click()
  cy.contains('Welcome to Project Gate!')
  cy.contains('Projects').click()
  cy.contains('My projects')
  cy.contains('Staff').click()
  cy.contains('theFirma Teachers')
  cy.contains('Events & Social Media').click()
  cy.contains('Bread & Games')
  cy.contains('Ville').click()
  cy.contains('Your profile').click()
  cy.contains('Profile')
})
```

Kuva 11. Cypress-testejä Project Gate 2.0 sovellukselle

7 Loppupäätelmät

Opinnäytetyön tavoitteena oli vertailla sopivia automaatiotestaustyökaluja toimeksiantajan kehitysvaiheessa olevan sovelluksen tarpeisiin, suunnitella tarvittavat testit sovelluksen toiminnallisuuksiin, sekä implementoida suunnitellut testit sovellukseen. Lopputuloksena syntyi toimiva testiympäristö, testitapausten suunnittelu ja kirjaaminen valmiille toiminnallisuuksille, sekä näiden testitapausten onnistunut implementointi automaatiotesteiksi käyttäen Jestiä ja Cypressiä.

Oman kehittymisen tavoitteista saavutin automaatiotestaukseen liittyvän teorian syvemmän perehtymisen sekä työkalujen ja niiden erojen oppimisen. Teknisen kirjoittamisen paremmassa osaamisessa on vielä tarvetta tulevaisuudessa.

Opinnäyteprosessin loppupuolella Project Gate 2.0-sovellus päätettiin yhdistää muihin theFIRMAn sisäisiin hallinnointiohjelmiin. Tämä vaatii sovelluksen jatkokehittäjiltä mahdollisesti testityökalujen uudelleenarviointia uusia toiminnallisuuksia ja teknologioita silmällä pitäen.

Automaatiotestauksen muihin jatkokehittämistarpeisiin lukeutuu testiympäristön siirtäminen kontteihin, jotta testit toimivat odotetulla tavalla sekä sovelluksen kehitysvaiheessa että tuotantoon vietäessä. Myös automaatiotestien vieminen osaksi theFIRMAn GitLab-repositorion jatkuvan integroinnin ja jatkuvan toimituksen (CI/CD) prosessia auttaisi varmistamaan koodin luotettavuutta.

Lähteet

Awati, R. 2022. Integration testing or integration and testing (I&T). Viitattu 3.6.2024. https://www.techtarget.com/searchsoftwarequality/definition/integration-testing

Bakharev, N. 2023. Unit Testing: Definition, Examples, and Critical Best Practices. Viitattu 10.6.2024. https://brightsec.com/blog/unit-testing/

Bose, S. 2022. The different levels of a Test Automation Pyramid. Viitattu 23.5.2024. https://www.browserstack.com/guide/testing-pyramid-for-test-automation

Cypress.io 2024a. Testing is the Key to Continuous Innovation – The Story of Cypress.io. Viitattu 22.5.2024. https://www.cypress.io/about-us/our-story

Cypress.io 2024b. Why Cypress? Viitattu 10.6.2024. https://docs.cypress.io/guides/overview/why-cypress

Da Costa, L. 2021. Testing JavaScript Applications. Shelter Island: Manning

Docker 2024. Use containers to Build, Share and Run your applications. Viitattu 10.6.2024. https://www.docker.com/resources/what-container/

Gillis, A. 2023. What is end-to-end testing?. Viitattu 4.6.2024. https://www.techtarget.com/searchsoftwarequality/definition/End-to-end-testing

Gitlab 2024. Flaky tests. Viitattu 4.6.2024. https://docs.gitlab.com/ee/development/testing_guide/flaky_tests.html

Godwin, A. 2023. Exploring the Node.js native test runner. Viitattu 21.5.2024. https://blog.logrocket.com/exploring-node-js-native-test-runner/

Google 2024a. Puppeteer. Viitattu 22.5.2024. https://pptr.dev/

Google 2024b. What is the status of cross-browser support? Viitattu 22.5.2024. https://pptr.dev/faq

Greif, S. & Burel, E. 2023. Testing. Viitattu 7.5.2024. https://2022.stateofjs.com/en-US/libraries/testing/

Hu, T. 2022. Viitattu 12.6.2024. https://about.codecov.io/blog/types-of-testing-unit-vs-integration/

Jasmine 2024. Unit Testing in Angular with Jasmine and Karma. Viitattu 10.6.2024. https://jasmine.github.io/index.html

Luukkainen, M. 2024a. Legacy: testaaminen Jestiä käyttäen. Viitattu 4.5.2024. https://fullstackopen.com/osa4/legacy_testaaminen_jestia_kayttaen

Luukkainen, M. 2024b. test-ympäristö. Viitattu 23.5.2024. https://fullstackopen.com/osa4/backendin_testaaminen

MariaDB Foundation 2024. MariaDB Server: the innovative open source database. Viitattu 16.3.2024. https://mariadb.org/

Meta 2022. Your First Component. Viitattu 3.4.2024. https://18.react.dev/learn/your-first-component

Meta 2024a. Why Test. Viitattu 23.5.2024. https://reactnative.dev/docs/testing-overview

Meta 2024b. Testing Overview. Viitattu 24.5.2024. https://legacy.reactjs.org/docs/testing.html

Microsoft 2024a. What is TypeScript?. Viitattu 3.6.2024. https://www.typescriptlang.org/why-create-typescript/

Microsoft 2024b. Playwright. Viitattu 17.5.2024. https://playwright.dev/

Monesi, D. 2024. TypeScript vs. JavaScript From 30,000 Feet: Scalability Challenges. Viitattu 9.6.2024. https://www.toptal.com/typescript/typescript-vs-javascript-guide

Moore, E. 2023. HTTP status codes 101: A guide implementing status codes in REST APIs. Viitattu 7.6.2024. https://www.lonti.com/blog/http-status-codes-101-a-guide-implementing-status-codes-in-rest-apis

Mozilla 2024. What is JavaScript? Viitattu 24.3.2024. https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript

Node 2023. Node.js 20 is now available! Viitattu 10.6.2024. https://nodejs.org/en/blog/announcements/v20-release-announce

NPM 2024a. Next.js. Viitattu 9.6.2024. https://www.npmjs.com/package/next.

NPM 2024b. Cross-env. Viitattu 6.6.2024. https://www.npmjs.com/package/cross-env

NPM trends 2024a. Next. Viitattu 9.6.2024. https://npmtrends.com/next

NPM trends 2024b. Cypress vs playwright vs puppeteer vs selenium-webdriver. Viitattu 6.6.2024. https://npmtrends.com/cypress-vs-playwright-vs-puppeteer-vs-selenium-webdriver

Obregon, A. 2023. Unit Testing in Angular with Jasmine and Karma. Viitattu 10.6.2024. https://medium.com/@AlexanderObregon/unit-testing-in-angular-with-jasmine-and-karma-63b1da49320b

OpenJS Foundation 2024a. Test runner. Viitattu 18.5.2024. https://nodejs.org/docs/latest/api/test.html

OpenJS Foundation 2024b. Mocha. Viitattu 15.5.2024. https://mochajs.org/

Risad, H. 2023. There Are 8 Benefits of Server Side Rendering in Next.js. Viitattu 3.4.2024 https://www.linkedin.com/pulse/8-benefits-server-side-rendering-nextjs-hafez-risad-eqznc

Selenium 2024a. About Selenium. Viitattu 22.5.2024 https://www.selenium.dev/about/

Selenium 2024b. Getting Started. Viitattu 22.5.2024. https://www.selenium.dev/documentation/webdriver/getting_started/

Sharanya, R.C. 2023. Jest vs Mocha vs Jasmine: Which JavaScript framework to choose? Viitattu 6.4.2024. https://www.browserstack.com/guide/jest-vs-mocha-vs-jasmine

Son, H. 2023. Manual Testing vs Automated Testing: Key Differences. Viitattu 10.6.2024. https://www.testrail.com/blog/manual-vs-automated-testing/

Stack Overflow 2023. 2023 Developer Survey. Viitattu 24.3.2024. https://survey.stackoverflow.co/2023/#programming-scripting-and-markup-languages

Testim 2020. What is Browser Automation? Definition and Getting Started. Viitattu 10.6.2024. https://www.testim.io/blog/browser-test-automation/

Testing Library 2021. Introduction. Viitattu 6.6.2024. https://testing-library.com/docs/

Testing Library 2024. React Testing Library. Viitattu 6.6.2024. https://testing-library.com/docs/react-testing-library/intro/

Torppa, T.; Peuraniemi, T. & Rapo, J. 2023a. Background and introduction. Viitattu 8.6.2024. https://fullstackopen.com/en/part9/first_steps_with_type_script

Torppa, T.; Peuraniemi, T. & Rapo, J. 2023b. Background and introduction. Viitattu 27.3.2024. https://fullstackopen.com/en/part9/background_and_introduction

Tozzi, C. 2023. Vitest vs. Jest: Differences, Benefits, and Challenges. Viitattu 21.5.2024. https://saucelabs.com/resources/blog/vitest-vs-jest-comparison

Unadkat, J. 2023. What is Test Driven Development (TDD)? Viitattu 10.4.2024. https://www.browserstack.com/guide/what-is-test-driven-development

Vercel 2024. What is Next.js? Viitattu 3.4.2024 https://nextjs.org/docs#what-is-nextjs

Vitest 2024. Vitest Next Generation Testing Framework. Viitattu 21.5.2024. https://vitest.dev/

Vocke, H. 2018. Practical test pyramid. Viitattu 7.6.2024. https://martinfowler.com/articles/practical-test-pyramid.html

W3Techs 2024. Historical yearly trends in the usage statistics of client-side programming languages for websites. Viitattu 24.3.2024. https://w3techs.com/technologies/history_overview/client_side_language/all/y

Zotti, A. 2015. What is the difference between a test runner, testing framwork, assertion library, and a testing plugin? Viitattu 30.5.2024. https://amzotti.github.io/testing/2015/03/16/what-is-the-difference-between-a-test-runner-testing-framework-assertion-library-and-a-testing-plugin/