React.js websovelluksen testaus

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Ammattikorkeakoulututkinnon opinnäytetyö

Tietojenkäsittelyn koulutus, Hämeenlinnan korkeakoulukeskus
kevät, 2021

Petri Lindholm

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tietojenkäsittelyn koulutus — Tiivistelmä

Hämeenlinnan korkeakoulukeskus

Tekijä	Petri Lindholm	Vuosi 2021
Työn nimi	React.js websovelluksen testaus	
Ohjaajat	Mirlinda Kosova-Alija	

TIIVISTELMÄ

Opinnäytetyön tarkoituksena oli tehdä selvitystä ohjelmistotestauskäytännöistä sekä työkaluista React.js -kirjastolla kehitetylle websovellukselle. Opinnäytetyön toimeksiantajana toimi Hämeen Ammattikorkeakoulun Smart-tutkimusyksikkö. Tavoitteena oli luoda luotettavia testitapauksia toimeksiantajan kehittämälle Kaupunki taskussa-websovellukselle.

Opinnäytetyön teoreettisessa osuudessa määritellään työn kannalta keskeiset käsitteet sovellustestauksesta, JavaScript-ohjelmointikielestä, React.js -kirjastosta, sekä testaustyön suorittamiseen käytetyistä työkaluista. Opinnäytetyön toiminnallisessa osuudessa testitapauksia kehitetään käyttäen Jest-, React Testing Library-, ja Cypress-testaustyökaluja.

Opinnäytetyössä havaittiin Jestin olevan keskeisessä roolissa yksikkö- ja integraatiotestejä kehittäessä sen sisältämien kattavien ominaisuuksien sekä testiajurin vuoksi. React Testing Library, jota käytetään Jestin yhteydessä React.js -sovelluksia testattaessa, mahdollistaa luotettavien, yksinkertaisten ja helposti luettavien testitapausten kehittämisen. Cypress osoittautui mainioksi valinnaksi käyttöliittymätestien suorittamiseen, sillä se on helppo asentaa, käyttää, ja sen virallinen dokumentaatio on laadultaan erinomainen.

Avainsanat JavaScript, React, ohjelmistotestaus

Sivut 58 sivua ja liitteitä 1 sivu

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Degree Programme in Business Information Technology
Hämeenlinna University Centre

Abstract

Author Petri Lindholm
Year 2021
Subject React.js web application testing
Supervisors Mirlinda Kosova-Alija

ABSTRACT

The purpose of this thesis was to examine suitable tools and methods for testing a web application developed with the React.js library. This thesis was commissioned by Häme University of Applied Sciences' Smart Research Unit. The goal was to write reliable test cases for the commissioner's Kaupunki taskussa web application.

The theoretical part of the thesis defines the key aspects regarding software testing, the challenges JavaScript as a language brings to the testing process, the core concepts of the React.js library and the tools used for testing web applications. The thesis proceeds to describe how unit, integration and end to end testing is done in practice for a web application using Jest, React Testing Library and Cypress as testing tools.

The research demonstrates the central role Jest plays in unit and integration testing with its fantastic API, assertion library and test runner. React Testing Library, used alongside Jest provides a reliable, logical way of writing test cases for React applications. Cypress showed its strength as a user interface end to end testing tool due to its ease of use, simple installation process and outstanding official documentation.

Keywords JavaScript, React, software testing
Pages 58 pages and appendices 1 page

Sanasto

HTML	HyperText Markup Language, verkkosivujen määrittelykieli
JavaScript	Ohjelmointikieli
React / React.js	JavaScript-kirjasto
JSX	JavaScript XML, HTML-tyylistä syntaksia JavaScript-tiedostoissa
Babel	Kääntäjä, transpiloi uutta JavaScriptia kaikille selaimille sopivaksi
Node.js	JavaScript-ajoympäristö, joka mahdollistaa JavaScriptin suorittamisen selaimen ulkopuolella
PWA	Progressive Web App, websovellus joka sisältää natiivien mobiilisovellusten tavoin esim. offline-tilan käytön
Unit test	Yksikkötesti, pieni yksittäisen funktion toimintaa arvioiva testi
Integration test	Integraatiotesti, useamman sovelluksen osan yhtenäistä toimintaa arvioiva testi
End-to-end test	Sovelluksen toiminnallisuutta ulkopuolisena käyttöliittymän kautta kokonaisuutena arvioiva testi
Rajapinta / API	Ohjelmointirajapinta, engl. Application Programming Interface, mahdollistaa tiedon siirron ohjelmistojen välillä
IDE	Integrated Development Environment, kehitysympäristö
VS Code	Microsoftin kehittämä kehitysympäristö

Sisällys

Kuvat, ohjelmakoodit ja taulukot

Liitteet

1 Johdanto

Sovellustestauksella tarkoitetaan ohjelmistotuotteen toiminnallisuuden varmentavaa työtä, jonka voi automatisoida koodiksi käyttäen erilaisia testaustyökaluja. Jokaisen sovellusta kehittävän yrityksen kuuluisi käyttää huomattavasti aikaa luotettavien testitapausten kirjoittamiseen, sillä testauksesta saatava hyöty voidaan mitata muun muassa asiakastyytyväisyydessä, kehitykseen kuluvassa ajassa, sekä tuotteen ylläpidon kustannuksissa. Ohjelmistotuotteessa esiintyvän virheen korjaaminen kehitysvaiheessa maksaa yritykselle vain murto-osan niistä kuluista ja mahdollisista menetetyistä tuloista joita asiakkaan käsiin toimitetusta virheellisestä sovelluksesta koituu.

Tämä opinnäytetyön toimeksiantajana toimii Hämeen Ammattikorkeakoulun Smart-tutkimusyksikkö. Toimeksiantaja tuottaa React.js -kirjastolla ohjelmoitua Kaupunki taskussa -websovellusta Hämeenlinnan kaupungille ja kaipaa selvitystä React.js -kirjastolle räätälöidyistä ohjelmistotestauksen työkaluista sekä hyvistä testauskäytännöistä.

Teoriaosuudessa lukija perehdytetään aluksi ohjelmistotestauksen peruskäsitteisiin ja syihin miksi sovelluksia testataan, sekä hyötyihin joita testauksesta saadaan. Teoriaosuuden toisella puoliskolla keskitytään JavaScript-ohjelmointikieleen, React.js-kirjastoon, sekä testaus- ja kehitystyössä tarvittaviin työkaluihin kuten Babel-kääntäjään ja ajoympäristöihin.

Käytännön osuuden ensimmäisellä puoliskolla lukijalle havainnollistetaan JavaScript- ja React-testausta lyhyillä käytännön esimerkeillä testaustyökalujen asennuksesta ja konfiguroinnista aina varsinaisen testitapauksen kirjoittamiseen, sekä ajamiseen. Testien toimivuus varmennetaan rikkomalla testit tarkoituksella, sillä testi, joka ei ilmoita virheestä sen sattuessa on hyödytön. Käytännön osuuden toisella puoliskolla toimeksiantajan Kaupunki taskussa-sovellukselle erittelemiä testejä tutkitaan tapauskohtaisesti. Opinnäytetyö pyrkii vastaamaan seuraaviin kysymyksiin:

- Mitä sovellustestauksella tarkoitetaan?
- Millä työkaluilla websovelluksia kannattaa testata?
- Miten React-komponentteja kannattaa testata?

2 Testaustyön tietoperusta

Tässä luvussa kerrotaan mitä sovellustestaus on, mitä hyötyä testaamisesta saadaan, mitä erilaiset testityypit kattavat ja miten hyvien testikäytäntöjen noudattaminen nopeuttaa sovelluskehityksen sylkiä tuottaa luotettavaa, hyvin dokumentoitua ohjelmistotuotetta.

2.1 Miksi sovelluksia testataan

Sovellustestauksella tarkoitetaan ohjelmistotuotteelle määritettyjen vaatimusten täyttymistä, sekä toiminnallisuuksia vertailevaa työtä, jolla varmistetaan, että tuote käyttäytyy kuten oli tarkoitus. Ohjelmistoprojektien sisältäessä suuria määriä toimintoja tarkastettavien asioiden ja mahdollisten vikatilojen määrä kasvaa valtavaksi. Sovellustestaus juontaa juurensa 1960-luvulle, jolloin tietokoneiden suoritustehon kasvaessa ohjelmistojen sisältämien ominaisuuksien määrä kasvoi liian suureksi, jotta yksittäinen ohjelmoija pystyisi muistamaan kaikkia vialliseen toimintaan johtavia yksityiskohtia. (Kasurinen, 2013)

Yrityksen toimialasta ja yksittäisen sovelluksen käyttökohteesta riippuen sovellustestauksella saatetaan hakea erilaisia tuloksia. Valtion virallisten laitosten tai pankkialan toimijoiden keskeisimpiä kriteerejä saattavat olla saavutettavuusdirektiivin noudattaminen, palvelun helppokäyttöisyys, sekä maksu- tai tunnistautumistapahtumiin liittyvän turvallisuuden takaaminen. Peliyritystä taas saattaa kiinnostaa enemmän graafisten elementtien toimivuus, sekä käyttäjäkokemus, jotta tuotteen parissa on hauskaa viettää aikaa. Autojen ajotietokoneita rakentavaa yritystä tuskin saavutettavuus tai hauskuus kiinnostaa, sillä heidän prioriteettinsa on tuottaa luotettavia hallintalaitteita. (Kasurinen, 2013)

Kun valtaosa maailmasta on riippuvainen ohjelmistoista uusien toiminnallisuuksien tarve kasvaa eksponentiaalisesti. Siksi on kriittistä tuottaa ja toimittaa luotettavasti toimivaa koodia mahdollisimman nopeasti. Koodin luotettavuuden varmistamiseen tarvitaan testejä, varsinkin automatisoituja testejä, joiden ajamiseen kuluva aika voidaan minimoida. Tässä kohtaa testien kirjoittaminen ei ole pelkästään hyvä käytäntö, vaan alan standardi. Automatisoiduilla testeillä ei voida kuitenkaan varmistaa ohjelmistotuotteen toimivuutta. Testit pystyvät ainoastaan todistamaan sen, että tuote ei toimi. (da Costa, 2021, An introduction to automated tests-luku)

2.2 Automatisoitu testaus

Kuvitellaan tilanne, jossa kehittäjä rakentaa paikalliselle pienyrittäjälle websovellusta. Vaatimuksena websovellukselle on, että asiakas voi tilata tuotteita websovelluksen kautta, syöttää toimitusosoitteen, ja vahvistaa tilauksen. Kehittäjän luotua nämä toiminnallisuudet, hän haluaa varmistaa, että tuote toimii kuten pitää. Kehittäjä luo tietokannan yrittäjän myymistä tuotteista, käynnistää websovelluksen palvelimen, ja avaa websovelluksen selaimessa tarkoituksena tilata muutama tuote. Tuotteita lisätessä ostoskoriin kehittäjä huomaa, että ostoskoriin voidaan lisätä vain yksi tuote kerrallaan. Mitä jos asiakkaat haluavat tilata useamman tuotteen kerralla? Kehittäjä lisää testitapauksen, jossa useiden tuotteiden lisäys pitää testata aina kun websovellusta päivitetään. Tätä kutsutaan manuaaliseksi testaamiseksi. Websovelluksen kasvaessa testattavien toimintojen määrästä voidaan pian todeta, ettei jokaisen testitapauksen käsittely manuaalisesti skaalaudu hyvin. Manuaalinen testaus vie kehittäjältä runsaasti aikaa, ja kuten kaikissa manuaalisissa tehtävissä, testaustyössä voidaan sortua inhimillisiin virheisiin. (da Costa, 2021, An introduction to automated testing-luku, What is an automated test-alaluku)

Manuaalisten testien tuottama ongelma voidaan ratkaista korvaamalla asiakas, tässä tapauksessa testiä suorittava kehittäjä, koodilla. Automatisoidut testit ovat sovelluksen osia, joilla sovellustuotteen testaustyön taakka voidaan automatisoida. Automatisoidut testit suorittavat sovelluksen toimintoja ja vertaavat niiden tuloksia odotettuun tapahtumaan. Automatisoidulla testillä voidaan nyt varmistaa usean tuotteen lisääminen ostoskoriin. Mitä jos joku sattuu tilaamaan 10 000 tuotetta, mutta yrityksellä ei ole tarpeeksi tuotteita varastossa? Tätä varten kehittäjä luo toiminnallisuuden, jossa tuote voidaan lisätä ostoskoriin vain jos sitä on yrityksen varastossa saatavilla. Ostoskoriin lisättäessä websovellus tarkistaa tietokannasta tuotteen saatavuuden, ja asiakkaan varmistaessa tilauksensa tuotteen saatavuus päivitetään tietokantaan. Näille toiminnallisuuksille pitää myös luoda omat automatisoidut testitapaukset, joilla varmistetaan niiden toimivuus. Jos asiakas pystyy edelleen tilaamaan enemmän tuotteita kuin mitä on saatavilla, testitapaukset ilmoittavat niistä virheviestillä. (da Costa, 2021, An introduction to automated testing-luku, What is an automated test-alaluku)

2.3 Testauksesta saatava hyöty

Yrityksen asiakkaiden määrästä tai käsiteltävän datan laajuudesta riippumatta hyvien testien kirjoittaminen on arvokas osa luotettavan ohjelmiston tuottamista. Oli projekti sitten suuri tai pieni, testien teko on kannattavaa, sillä se edesauttaa kehittäjien välistä kommunikointia ja vähentää sovelluksessa esiintyvien vikojen määrää. Kehitystiimin jäsenten määrän kasvaessa testien kirjoittamisen tärkeys kasvaa samassa suhteessa, sillä mitä enemmän rahaa ja aikaa projektin kehitykseen kuluu, sitä kalliimmaksi virheiden aiheuttamien vikojen korjaaminen käy. (da Costa, 2021, An introduction to automated tests-luku)

Kasurisen (2013, s. 13) mukaan testaustyö on tärkein työvaihe kannattavuuden kannalta. Yritykset, joissa ohjelmistotestaus suoritetaan huolellisesti, saavat paremman katteen tuotteilleen niihin yrityksiin verrattuna, jotka testaavat tuotteensa huolimattomasti.

Testaus on myös erinomainen tapa dokumentoida koodia. Toisin kuin esimerkiksi erilliseen tekniseen dokumentaatioon kirjoitettu kuvaus sovelluksen osan toiminnallisuudesta, tai lähdekoodiin kirjoitetut kommentit, automatisoidut testit ilmoittavat välittömästi, kun osan toiminnallisuus poikkeaa odotetusta. Tällöin testitapauksia joudutaan päivittämään, jolloin niiden kuvaus osan toiminnasta pysyy ajan tasalla. Testit takaavat myös sen, että muut kehitystiimin jäsenet pystyvät testejä lukemalla ymmärtämään miten ominaisuudet toimivat. (da Costa, 2021, An introduction to automate testing-luku, Collaboration-alaluku)

Testaukseen sijoitettavassa budjetissa kannattaa ottaa huomioon viallisesta tuotteesta koituvat tappiot. Huonot käyttäjäkokemukset usein johtavat negatiivisiin tuotearvosteluihin, jotka voivat karkottaa tulevia asiakkaita, sekä ajaa jo palvelusta maksaneet asiakkaat vaatimaan rahojansa takaisin. Asiakkaiden luottamuksen palauttaminen on hankalaa. (Miller, 2020)

Ohjelmistoissa esiintyvät virheet voivat myös johtaa ihmishenkiä vaativiin tapaturmiin. Helmikuun 25. päivä 1991, Persianlahden sodassa Yhdysvaltojen PATRIOT-ohjuksentorjuntajärjestelmä epäonnistui jäljittämään ja torjumaan vihollisen laukaisemaa Scud-ohjusta. Ohjus osui Yhdysvaltojen armeijan kasarmiin Dhahranissa, Saudi Arabiassa, tappaen 28 amerikkalaista. Vika johtui ohjelmistovirheestä, joka tuotti väärää jäljityslaskelmaa, ja sen virhemarginaali kasvoi mitä pidempään järjestelmä oli päällä.

Tapaturman sattuessa ohjuksentorjuntajärjestelmä oli ollut yli 100 tuntia jatkuvassa käytössä, jonka takia jäljityslaskenta heitti noin 687 metrillä. Virheen takia torjuntaohjusta ei laukaistu. Ohjelmiston epätarkkuudesta oli ilmoitettu jo kaksi viikkoa etukäteen, ja ohjelmistoon oli tehty muutoksia tarkkuuden parantamiseksi, mutta päivitetty ohjelmisto saapui Dhahraniin vasta helmikuun 26., yhden päivän myöhässä. (U.S. Government Accountability Office, 1992, s. 1, s. 15)

Yrityksen liiketoiminnan kannalta ohjelmistotuotteen kehityksen nopeus ja toimivuuden luotettavuus ovat tärkeitä, ei se miten paljon testejä kehittäjät ovat kirjoittaneet. Kun kehittäjä pystyy tuottamaan koodia nopeasti, todistaen ettei se sisällä tiettyjä, testeillä todennettuja bugeja, ja toiminnallisuus voidaan integroida muuhun järjestelmään turvallisesti, yritys menestyy. Automatisoitujen testien kirjoittamiseen kuluva aika tuottaa kustannuksia, mutta niistä saatava hyöty kasvaa mitä enemmän testejä ajetaan. Jos toiminnon testaaminen manuaalisesti veisi minuutin, ja automatisoidun testin kirjoittaminen veisi viisi minuuttia, automatisoitua testiä ajettaessa viidettä kertaa sen kirjoittamiseen kulunut aika on maksanut itsensä takaisin. Ohjelmistokehitysprojektin aikana automatisoituja testejä voidaan ajaa tuhansia kertoja. (da Costa, 2021, An introduction to automated testing-luku, Speed-alaluku)

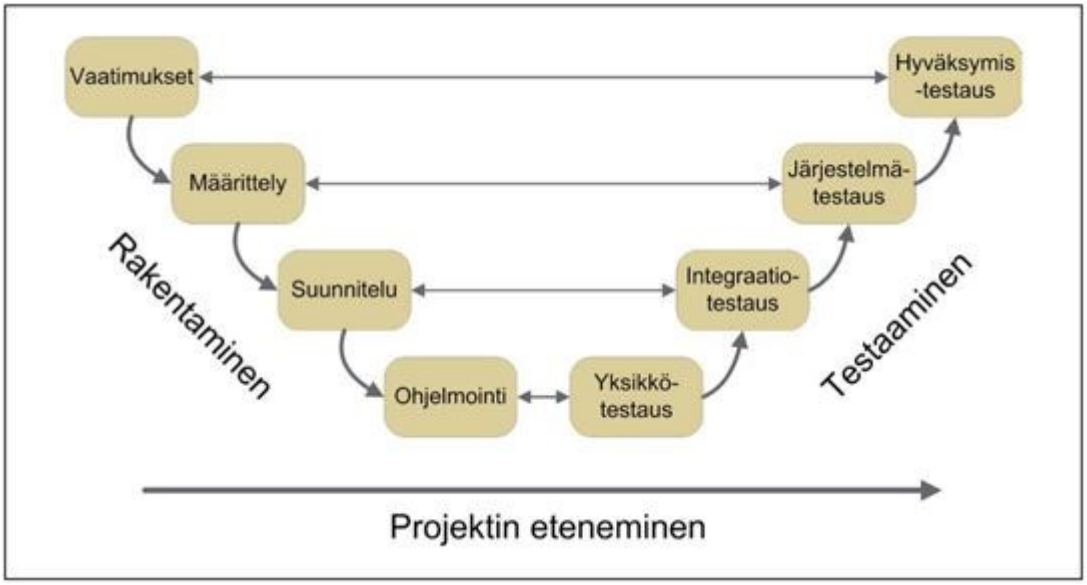
2.4 Testaustasot

Testejä voidaan jakaa useampaan eri kategoriaan, jotka palvelevat eri tarkoituksia. On tärkeää testata yksittäisiä osia erillään muusta ympäristöstä, kuten funktioita ja niistä palautuvia arvoja, varmistaakseen niille asetettujen kriteerien täyttävä toiminta. Vähintään yhtä tärkeää on myös varmistaa useiden yksittäisten osien toimivuus toistensa kanssa integroituina kokonaisuutena. Autossa toimiva moottori on turha, jos se ei onnistu siirtämään kehittämäänsä voimaa vaihteiston kautta renkaille. (da Costa, 2021, What to test and when-luku)

Kasurinen (2013, s. 41) määrittelee testaamisen V-mallissa (Kuva 1) kolme päävaihetta kehitysvaiheessa testaamiseen: yksikkötestauksen, integraatiotestauksen, ja järjestelmätestauksen. Näitä kutsutaan testaustasoiksi, koska niissä testausta tehdään eri tasoilla. Yksikkötestauksessa testitapaus rajoittuu yhteen komponenttiin, tai funktioon.

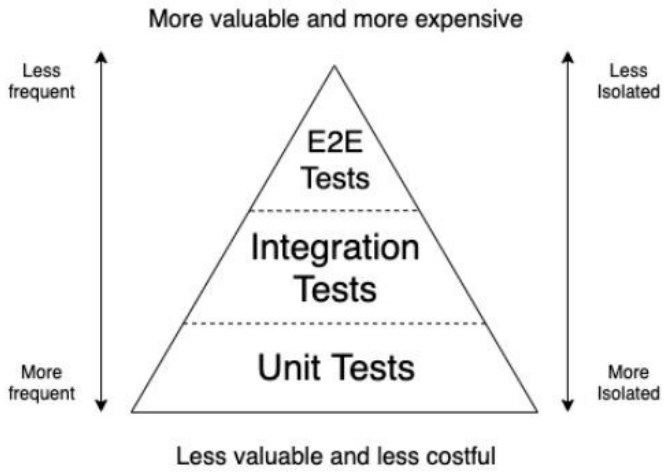
Integraatiotestauksessa testataan muutaman komponentin välistä toimintaa. Järjestelmätestauksessa testaus kattaa koko järjestelmän, mutta vielä toistaiseksi testiympäristössä. Hyväksymistestauksessa testausta tehdään jo kohdeympäristössä tai sitä tarkkaan simuloiden.

Kuva 1. Testauksen V-malli (Kasurinen, 2013)



Da Costa (2021, What to test and when-luku, The testing pyramid-alaluku) jakaa testauspyramidissaan (Kuva 2) testit kolmeen eri tasoon niistä saatavan hyödyn, testien kirjoittamiseen ja ajamiseen kuluvan ajan, sekä testien määrän mukaan. Pyramidin pohjalla olevat yksikkötestit (unit tests) vievät vähiten aikaa niitä ajettaessa, ovat helppoja kirjoittaa, päivittää ja ymmärtää, joten niitä tulisi tehdä eniten. Integraatiotestit (integration tests) kattavat yksikkötestejä laajemman osan ohjelmistosta, joten niistä saadaan enemmän hyötyä. Integraatiotestien kirjoittamiseen ja ajamiseen kuluu enemmän aikaa, jonka takia niitä tulisi tehdä vähemmän, joten ne sijoittuvat korkeammalle pyramidissa. Järjestelmätestit (E2E tests, end-to-end tests) kattavat laajuudellaan koko ohjelmiston, jolloin niistä saatava hyöty on kaikkein arvokkainta. Järjestelmätestien kirjoittamiseen ja ajamiseen kuluu kuitenkin eniten aikaa, joten niitä tulisi tehdä määrällisesti vähiten.

Kuva 2. Luca da Costan testauspyramidi. (da Costa, 2021, What to test and when-luku, The testing pyramid-alaluku)

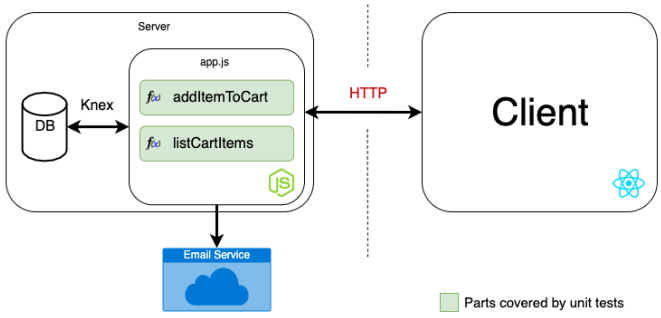


Kent C. Dodds (Dodds, 2019) muokkaa testauspyramidin käsitettä lisäämällä siihen staattisen testauksen. Staattisella testauksella tarkoitetaan koodin tarkistusta kirjoitusvaiheessa. Lintterit, kuten ESLint, ovat erinomaisia työkaluja, sillä ne tarkistavat koodia editorissa kehittäjän sitä kirjoittaessa, ja ilmoittavat kirjoitusvirheistä ja hyviin ohjelmointikäytäntöihin liittyvistä säännöistä välittömästi. Doddsin mielestä integraatiotesteihin pitäisi panostaa eniten, sillä niistä saatavat hyödyt ja luotettavuus ovat testien kirjoittamiseen kuluvan ajan ja sitä myötä käytettävän rahan kanssa parhaiten tasapainossa. Dodds korvaa perinteisen testauspyramidin testauspokaalilla (Kuva 3), jossa staattiset testit luovat kehitystyölle vakaan jalustan, ja integraatiotesteihin käytettäisiin eniten resursseja.

Kuva 3. Kent C. Doddsin testauspokaali (Dodds, n.d.)

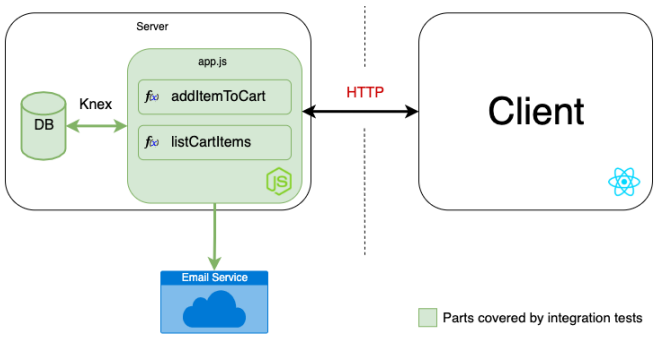
Yksikkötestaus on yleisin kaikissa ohjelmisto-organisaatioissa käytetty testaustoimenpide. Yksikkötestauksella (Kuva 4) tarkoitetaan testiä, jossa yhden yksittäisen funktion toimintaa tarkastellaan välittömästi toteutuksen yhteydessä, useimmiten itse kehittäjän toimesta. (Kasurinen, 2013) Testillä määritetään skenaario, jossa testattavan funktion koodi suoritetaan, ja funktiosta palautuvaa arvoa verrataan oletettuun arvoon. (da Costa, 2021, What to test and when-luku, Unit tests-alaluku) Testien epäonnistuessa kehittäjä tietää testattavan komponentin toiminnallisuuden olevan pielessä, jolloin vika voidaan korjata jo ennen sen yhdistämistä muihin ohjelmiston osiin. (Kasurinen, 2013)

Kuva 4. Yksikkötesti kattaa yksittäisiä osia sovelluksen toiminnallisuudesta. (da Costa, 2021, What to test and When-luku, Unit tests-alaluku)



Integraatiotestauksessa (Kuva 5) ohjelmiston eri osia aletaan sovittaa yhteen, varmistaen osien halutunlainen toiminta kokonaisuutena. Integraatiotestauksessa uusi yksikkötestit läpäissyt komponentti kytketään osaksi aiemmin testattua kokonaisuutta. Jos uutena osana lisättävä komponentti sisältää kytkentöjä, joita ei ole vielä olemassa, joudutaan testausta varten rakentamaan tynkiä (stub, mock), sijaiskomponentteja, joilla järjestelmä saadaan käynnistymään testausta varten. (Kasurinen, 2013)

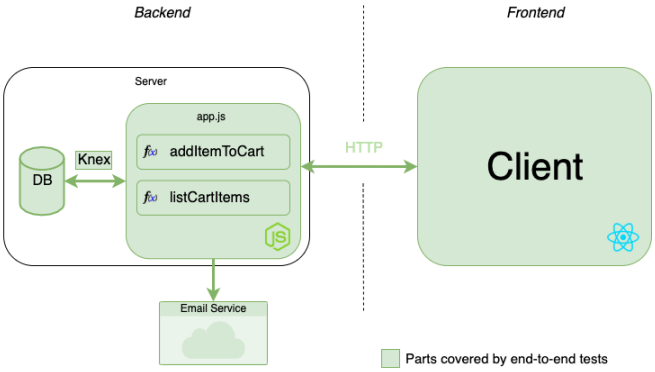
Kuva 5. Integraatiotesti kattaa useiden osien toimintaa kokonaisuutena. (da Costa, 2021, What to test and When-luku, Integration tests-alaluku)



End-to-end testauksessa (Kuva 6) järjestelmää testataan kokonaisuutena. Järjestelmän komponentit ovat läpäisseet yksikkötestauksen, jonka jälkeen niistä on rakennettu kokonaisuuksia, joiden toiminta on varmistettu integraatiotestauksella. (Kasurinen, 2013) End-to-end testeissä järjestelmää testataan ulkopuolisesti käyttäjän näkökulmasta, esimerkiksi websovelluksen kohdalla käyttöliittymän kautta selaimessa. Kuvassa 6 tuotteen lisääminen ostoskoriin end-to-end testillä avaisi websovelluksen selaimessa, klikkaisi ”Lisää ostoskoriin”-painiketta, ja tarkastaisi ostoskorin sisällön sivulta, jossa se on nähtävissä. (da Costa, 2021, What to test and When-luku, End-to-end tests-alaluku)

End-to-end testillä voidaan myös tarkoittaa sovelluksen tietokannan kanssa keskustelevalle rajapinnalle suoritettavaa testiä, jossa rajapinnalle lähetetään HTTP-kutsu tuotteen lisäämisestä, sekä kutsu ostoskorin sisällöstä. Kun sovellustuotteen kanssa ollaan vuorovaikutuksessa käyttäjän näkökulmasta jossa lähdekoodiin ei olla suoraan yhteydessä, tässä tapauksessa rajapinnan kautta, voidaan sitä kutsua end-to-end -testiksi. (da Costa, 2021, What to test and When-luku, End-to-end tests-alaluku)

Kuva 6. End-to-end testi kattaa koko sovelluksen. (da Costa, 2021, What to test and When-luku, End-to-end tests-alaluku)



2.5 AAA-testauskuvio

Testauksessa käytetään yleisesti kolmen A:n (arrange, act, assert) kuviota sen yksinkertaisuuden vuoksi, ja testitapausten yhtenäisyyden ylläpitämiseksi. Käyttämällä yhtä tiettyä tapaa kirjoittaa testejä, niistä tehdään helposti luettavia ja ymmärrettäviä, joka vähentää testien päivittämiseen kuluvaa aikaa. (Khorikov, 2020)

Arrange-vaiheessa, suomeksi järjestä, testattavan komponentin tila valmistellaan halutun toiminnallisuuden varmistamiseksi valmiiseen tilaan. Testattavasta kohteesta ja testityypistä riippuen valmisteluvaiheessa voidaan esimerkiksi määrittää funktiolle syötettävät parametrit, muodostaa yhteys palvelimeen, tai avata websovellus selaimessa. (Khorikov, 2020)

Act-vaiheessa, suomeksi toimia, testattava ominaisuus suoritetaan. Funktiota kutsutaan, syöttäen sille valmistellut parametrit, ja sen palauttama arvo tallennetaan. Palvelimelle tai rajapinnalle lähetetään HTTP-kutsu, ja siltä saatu vastaus tallennetaan. Selaimessa sivulta etsitään painike ja klikataan sitä. (Khorikov, 2020)

Assert-vaiheessa, suomeksi varmenna, suoritettua toimintoa verrataan odotettuun tulokseen. Funktion tapauksessa palautusarvoa verrataan siihen, mitä funktion oletetaan palauttavan. Palvelin- tai rajapintakutsun vastausta verrataan odotettuun vastaukseen. Selaimessa varmennetaan sivulla tapahtunut muutos. (Khorikov, 2020)

Testejä kirjoittaessa kannattaa kiinnittää huomiota siihen, etteivät testit ole toisistaan riippuvaisia. Kaikkien testien tulisi suoriutua tehtävästään samalla tavalla yksinään, tai tuhannen muun testin rinnalla ajettuna. Kirjoittamalla toisistaan riippuvaisia testejä on hankalaa todentaa johtuuko testin epäonnistuminen virheestä lähdekoodissa vai virheestä testeissä. (da Costa, 2021, Testing techniques-luku, Atomicity-alaluku)

3 JavaScript-ohjelmointikieli

Tässä luvussa kerrotaan JavaScriptin 25-vuotisesta historiasta, kielen määritelmää hallitsevasta komiteasta, ajoympäristöistä joissa koodia suoritetaan, React-kirjastosta, sekä haasteista, joita kieli tuottaa testaustyötä suoritettaessa.

3.1 Historia

JavaScriptia alettiin kehittää vuonna 1995. Netscape palkkasi nuoren insinöörin Brendan Eichin kirjoittamaan uuden, selaimissa ajettavan skriptauskielen. Eichille annettiin kymmenen päivää aikaa suunnitella ja kirjoittaa kieli, jotta se ehtisi Netscape Navigator 2.0 Beta-selaimen julkaisuun. Eich kutsui kehityksen aikana kieltä nimellä Mocha. Netscape aikoi julkaista kielen käyttämällä nimeä LiveScript, kunnes markkinointikikkana kielen nimeksi päätettiin JavaScript, Sun Microsystemsin Java-ohjelmointikieltä mukaillen. Microsoft julkaisi vuotta myöhemmin takaisinmallinnetun (reverse engineered) version kielestä Internet Explorer-selaimelle nimellä JScript. (Kaushik, 2020)

Netscape lähestyi eurooppalaista tietokonevalmistajien yhdistystä (European Computer Manufacturers Association), tästä eteenpäin viitattuna lyhenteellä ECMA, pyytäen heitä kehittämään yhteistä standardia, jota Netscape ja Microsoft noudattaisivat. ECMA kokosi teknisen komitean nimeltä TC39 hallinnoimaan kielen määritelmää. ECMAScript ohjelmointikielen määritelmää alettiin kehittää marraskuussa 1996. Kielen määritelmän ensimmäinen versio (ECMAScript 1, ES1) julkaistiin kesäkuussa 1997 sarjanumerolla ECMA-262. (Kuva 7) (Kaushik, 2020)

Kuva 7. ECMAScript-kielen määritelmä, ECMAn arkistossa (ECMA International, n.d.)

ECMA-263	Private Integrated Services Network (PISN) - Specification, functional model and information flows - Call priority interruption and call priority interruption protection supplementary services (CPI(P)SD)	December 2001
ECMA-262	ECMAScript® 2020 language specification, 11th edition (June 2020)	June 2020
ECMA-261	Broadband Private Integrated Services Network (B-PISN) - Service description - Broadband connection oriented bearer services (B-BCSD)	June 1997

Standardilla tarkoitetaan dokumenttia, joka rajaa säännöt, ohjeistukset, tai ominaisuudet yleistä ja toistuvaa käyttöä varten. Standardeja luodaan kokoamalla yhteen tuotteesta, prosessista, palvelusta tai tietystä materiaalista kiinnostuneiden osapuolien edustajista koostuva ryhmä, joka yhdessä tekee päätökset standardin määritelmästä. (ECMA International, n.d.)

Kielen määritelmää päivitettiin vuosina 1998 (ECMAScript 2, ES2) ja 1999 (ECMAScript 3, ES3). Kolmannen version julkaisun jälkeen kielen suosio internetin yleistyessä lähti hurjaan nousuun. (ECMA International, 2021) Kielen neljättä versiota kehitettiin useita vuosia, mutta poliittisista eroavaisuuksista ja kielen monimutkaisuudesta johtuen ECMAScriptin neljättä versiota ei koskaan julkaistu. Monista sen osista kuitenkin kehitettiin perusta vuonna 2009 julkaistulle ECMAScript 5 (ES5), sekä 2015 julkaistulle ECMAScript 6 (ES6 / ES2015) versioille. (Mozilla, n.d.-a)

2015 julkaistun ECMAScript 6 version kehitys alkoi vuonna 2009, ja kehitykseen sisältyi merkittävä määrä kokeilua ja kielen parantamisen suunnittelua, joiden alkuvaiheita voidaan jäljittää taaksepäin jopa kolmannen version julkaisuun vuonna 1999. ECMAScript 6:n julkaisua voidaan kuvailla siis 15 vuoden työn tuloksena. Vuodesta 2015 lähtien ECMAScriptin määritelmää on päivitetty vuosittain, ja nimeämiskäytäntö vaihtui vuosiperäiseen formaattiin. (ECMA International, 2021)

TC39 komitea koostuu noin 50-100 ihmisen ryhmästä, jotka edustavat eri web-teknologioihin sijoittaneita yrityksiä kuten selainvalmistajat Mozilla, Google ja Apple, sekä laitevalmistajia kuten Samsung. TC39 komitea tapaa keskimäärin joka toinen kuukausi keskustelemaan tapaamisten välissä tehdystä työstä, sekä äänestämään ehdotuksista. (Simpson, 2020, What is JavaScript-luku, Language Specification-alaluku)

Yhtenä TC39 komitean alkuperäisten määritelmän kulmakivistä oli kielen taaksepäin yhteensopivuuden takaaminen. Toisin kuin esimerkiksi ohjelmointikieli Pythonin tapauksessa, JavaScriptista ei ole saatavilla useita eri versioita. Määritelmää on vain päivitetty ajan myötä. Taaksepäin yhteensopivuudella tarkoitetaan sitä, että kun jokin osa hyväksytään osaksi JavaScriptia, sitä ei tulla tulevaisuudessa muuttamaan siten, ettei se olisi enää validia JavaScriptia. Tämä tarkoittaa, että vuonna 1996 kirjoitetun koodin pitäisi

edelleen toimia nykyaikaisissa selaimissa ja ajoympäristöissä. Taaksepäin yhteensopivuudella varmennetaan ettei kieleen tehdyt muutokset riko verkkosivuja tai websovelluksia. Tämän vuoksi JavaScriptin valitseminen ohjelmistoprojektin kieleksi on erittäin turvallinen vaihtoehto. (Simpson, 2020, What is JavaScript-luku, Backwards and Forwards -alaluku)

3.2 JavaScript-moottorit

JavaScript-moottorilla tarkoitetaan ohjelmistoa, joka toimii tulkkina JavaScriptin ja tietokoneen suorittimen välillä. JavaScript-moottorin voi kirjoittaa usealla eri ohjelmointikielellä. Googlen kehittämä V8 -moottori, jota käytetään Chrome- ja Chromium-pohjaisissa selaimissa, sekä Node.js-ajoympäristössä, on kirjoitettu C++ -kielellä. Mozillan Firefox-selaimen SpiderMonkey-moottorissa ohjelmointikieliksi valittiin C ja C++. (Rezvi, 2020)

Uuden ECMAScript-määritelmän julkaiseminen ei tarkoita sitä, että kaikki JavaScript-moottorit olisivat yllättäen varustettuja suorittamaan uusimman määritelmän mukaista JavaScript-koodia. Selainten kehittäjät ovat vastuussa uusien ominaisuuksien lisäyksestä JavaScript-moottoreihinsa, ja päivitysten jakamisesta käyttäjille. Tämä johtaa eroavaisuuksiin eri selainten yhteensopivuuden kanssa uusinta ECMAScript-määritelmää käyttävää JavaScript-kieltä suoritettaessa. (Aranda, 2017)

Kuten edellä todettiin JavaScriptin olevan taaksepäin yhteensopiva, se ei kuitenkaan tarkoita, että JavaScript olisi eteenpäin yhteensopiva. Eteenpäin yhteensopivalla tarkoitetaan sitä, että kieleen lisätyllä uudella toiminnolla ei olisi haitallisia vaikutuksia ohjelmaa ajettaessa vanhemmalla JavaScript-moottorilla. (Simpson, 2020, What is JavaScript-luku, Backwards and Forwards -alaluku) Käytännössä tämä tarkoittaa sitä, että esimerkiksi ECMAScript 2017-versioon lisätty tapa kirjoittaa asynkronisia funktioita käyttäen async/await-syntaksia ei toimisi vanhemmissa selainversioissa, joiden moottorille tätä ominaisuutta ei olla lisätty. Esimerkiksi Internet Explorer 11 on selain, joka ei tue async/await-toimintoja, joten kehittäjien on otettava vanhempien selainversioiden tuki huomioon kirjoittaessaan JavaScript-koodia. (McBride, 2018)

3.3 Babel-kääntäjä

JavaScriptin uusimpien ominaisuuksien käyttö siis tarkoittaa, että kehittäjän tulee kirjoittaa vanhemmalla standardilla määriteltyä koodia uuden rinnalle tukeakseen niitä selain- tai ajoympäristö-versioita, joihin päivityksiä ei olla lisätty. Tämän ongelman ratkaisemiseksi kehitettiin kääntäjiä (transpiler), kuten Babel. Kääntäjät voivat ottaa vastaan esimerkiksi ES6:n syntaksilla kirjoitettua JavaScriptia (Ohjelmakoodi 1), ja muuttaa sen kaikkien yleisimpien selainten ja ajoympäristöjen versioiden tukemaan ES5:n muotoon (Ohjelmakoodi 2). (Simpson, 2020, What is JavaScript-luku, Jumping the Gap-alaluku)

Ohjelmakoodi 1. ES6 standardin mukainen nuolifunktio (Babel, n.d.-b)

```
[1, 2, 3].map((n) => n + 1);
```

Ohjelmakoodi 2. Babelin ES5-standardin mukainen käännös nuolifunktiosta (Babel, n.d.-b)

```
[1, 2, 3].map(function (n) {
  return n + 1;
});
```

Ohjelmakoodi 1:ssä käytetty nuolifunktio on niin kutsuttua syntaktista sokeria, kehittäjien arkea helpottavaa tapaa kirjoittaa koodia lyhyemmässä, helpommin luettavassa muodossa. Ohjelmakoodi 3 sen sijaan käyttää ES2019 standardin mukaista promise-prototyypin finally-funktiota, toimintoa, joka vanhemmista JavaScript-moottoreista puuttuu. Ohjelmakoodia 3 ajettaessa vanhemmassa ajoympäristössä, finally-funktion kutsu aiheuttaisi virheen. (Simpson, 2020, What is JavaScript-luku, Filling the Gaps-alaluku)

Ohjelmakoodi 3. ES2019 standardin finally-funktion käyttöesimerkki (Simpson, 2020)

```
var pr = getSomeRecords(); // palauttaa promisen hakemastaan datasta
startSpinner();            // käynnistää latausanimaation
pr.then(renderRecords)     // näyttää haun tulokset jos haku onnistuu
  .catch(showError)        // näyttää virheviestin jos haku epäonnistuu
  .finally(hideSpinner)    // piilottaa latausanimaation joka tapauksessa
```

Tämän ongelman ratkaiseminen vanhemmissa selaimissa tai ajoympäristöissä vaatisi tarkistusta, jossa finally-funktion puuttuessa sen toiminnallisuus korvattaisiin vanhempaa standardia noudattavalla tavalla. Tätä käytäntöä kutsutaan nimellä polyfill. Ohjelmakoodissa 4 esitetään Simpsonia lainaten yksinkertaistettu kuvitteellinen tapa, jolla finally-funktion polyfill voitaisiin kirjoittaa. (Simpson, 2020, What is JavaScript-luku, Filling the Gaps-alaluku)

Ohjelmakoodi 4. Ajoympäristöstä puuttuvan finally-funktion polyfill (Simpson, 2020)

```
// Jos Promise-prototyypillä ei ole finally-nimistä arvoa
if (!Promise.prototype.finally) {
  // Lisää Promise-prototyypille korvaava funktio
  Promise.prototype.finally = function f(fn) {
    return this.then(
      function t(v) {
        return Promise.resolve(fn())
          .then(function t() {
          return v;
        });
      },
      function c(e) {
        return Promise.resolve(fn())
          .then(function t() {
          throw e;
        });
      }
    );
  };
}
```

Babelin tyyppiset kääntäjät helpottavat kehittäjän työtä havaitsemalla polyfillien tarpeen automaattisesti. Konfiguroimalla babelin oikein, tarvittavat polyfillit ladataan koodia käännettäessä automaattisesti. Babelin asetuksissa (Ohjelmakoodi 5) voidaan määritellä tarkalleen vanhimmat kohdennettavat selainversiot, jolloin Babel tekee kääntöprosessinsa aikana vain niihin versioihin asti tarvittavat muutokset, ja lisää tarpeelliset polyfillit. Ohjelmakoodi 6:ssa käytetty finally-funktiolle käännettäisiin Ohjelmakoodi 5:n asetusten mukaisesti hakemaan polyfill finally-funktiolle Babelin kirjastosta (Ohjelmakoodi 7), sillä Edge-selaimen versio 17 ei tue sitä. Babelin kaltaisia kääntäjiä hyödyntämällä JavaScript-koodia voidaan kirjoittaa uusimman ECMAScript-standardin mukaisesti, murehtimatta vanhempien selain- ja ajoympäristöjen tukemisesta. (Babel, n.d.-a)

Ohjelmakoodi 5. Babel-kääntäjän asetukset (Babel, n.d.-a)

```
{
  "presets": [
    [
      "@babel/env",
      {
        "targets": {
          "edge": "17",
          "firefox": "60",
          "chrome": "67",
          "safari": "11.1"
        },
        "useBuiltIns": "usage"
      }
    ]
  ]
}
```

Ohjelmakoodi 6. finally-funktion kutsu, joka vaatii kääntäjältä polyfillin (Babel, n.d.-a)

```
Promise.resolve().finally();
```

Ohjelmakoodi 7. Babelin tekemä käännös Ohjelmakoodi 6:sta (Babel, n.d.-a)

```
require('core-js/modules/es.promise.finally');
Promise.resolve().finally();
```

3.4 Node.js -ajoympäristö

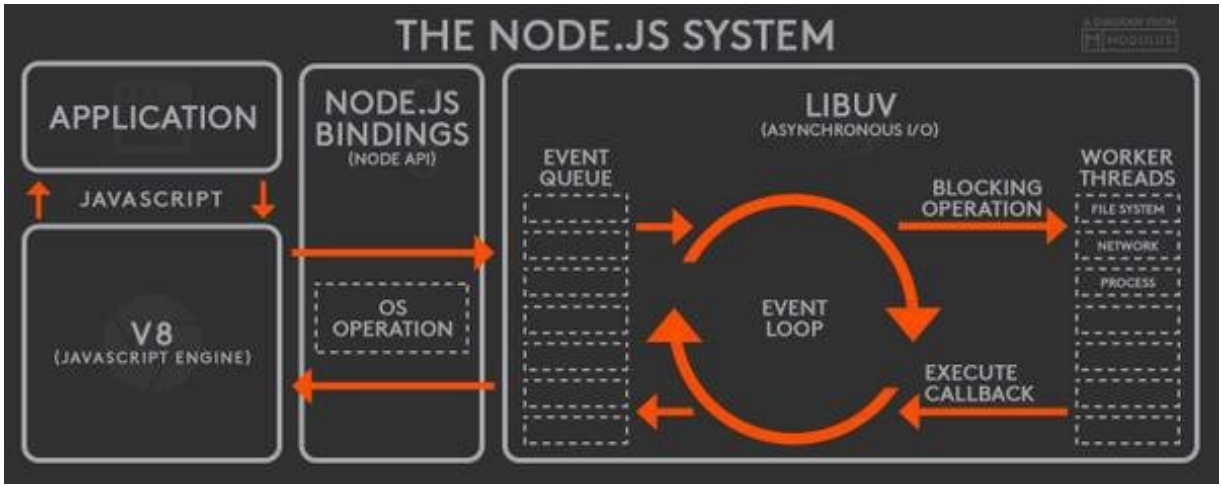
Node.js on avoimen lähdekoodin asynkroninen, tapahtumavetoinen (event-driven), Googlen V8-moottoria käyttävä JavaScript-ajoympäristö (runtime environment). Node mahdollistaa JavaScriptin ajamisen selaimen ulkopuolella. Ryan Dahl kehitti Noden vuonna 2009, tavoitteenaan luoda paremmin skaalautuva vaihtoehto Apachen HTTP Serverille. Noden omistus siirtyi Joyent-pilvipalveluyritykselle 2010. Vuonna 2015 projekti luovutettiin Node.js Foundationin käsiin. Vuonna 2019 Node.js Foundation ja JS Foundation yhdistyivät, muodostaen OpenJS Foundationin, joka tänä päivänä omistaa ja hallitsee Noden kehitystä. (Heller, 2020)

Synkronisella ohjelmoinnilla tarkoitetaan mallia, jossa komennot tapahtuvat yksi kerrallaan. Ohjelmakoodia suoritetaan rivi kerrallaan, kunnes sen toiminto on saatu päätökseen. Nopeus, jolla toimintoja suoritetaan riippuu täysin komentoja suorittavan koneen prosessorista. Monet toiminnot kuitenkin lähettävät HTTP-protokollan kautta kutsuja palvelimelle, tai noutavat dataa koneen kovalevyltä. Näissä tapauksissa synkronisessa ohjelmassa suoritus pysähtyy, kunnes palvelimelta tai kovalevyltä saadaan vastauksena jotain takaisin. Synkronisen ohjelmoinnin käyttö palvelinkutsuja suorittaessa ei siis ole hyvä käytäntö. (Haverbeke, 2018a)

Asynkronisella ohjelmoinnilla tarkoitetaan mallia, jossa montaa komentoa voidaan suorittaa yhtäaikaisesti. Pitkään kestävää toimintoa suoritetaan taustalla, samalla kun muut osat ohjelmistosta jatkavat toimintaansa. Synkronisella mallilla kahta peräkkäistä palvelinkutsua suoritettaessa ensimmäisen on päästävä päätökseen, kunnes toinen voi aloittaa toimintansa. Asynkronisesti suoritettuna sama toiminto voi lähettää molemmat palvelinkutsunsa ja vapauttaa ohjelman jatkamaan muuta toimintaansa kunnes palvelimelta lähetetyt viestit otetaan vastaan ja prosessoidaan sopivan tilaisuuden tullen. (Mozilla, n.d.-b)

JavaScript on yksisäikeinen ohjelmointikieli, mutta siitä huolimatta Node ja selaimet voivat suorittaa pitkään kestäviä toimintoja asynkronisesti käyttäen yksisäikeistä tapahtumaketjua (event loop) (Kuva 8), jossa useiden asynkronisten pyyntöjen suorittaminen siirretään monisäikeistä teknologiaa käyttävien ytimien (kernel) vastuulle. (Heller, 2020)

Kuva 8. Node.js:n tapahtumaketju havainnollistettuna (Witke, 2019)



Noden suosio serveriohjelmoinnissa johtuu sen erinomaisesta skaalautuvuudesta yksisäikeisyytensä takia, sekä siitä, että webkehittäjät voivat käyttää yhtä ja samaa ohjelmointikieltä tuotteen käyttöliittymän sekä palvelinpuolen ohjelmointiin. (Cucciniello, 2017) Node mahdollistaa myös npm:n (Node Package Manager) käytön komentorivin kautta suoraan kehitysympäristössä. Npm on maailman laajin ohjelmistorekisteri, jonka kautta kehittäjät voivat ladata ja jakaa ohjelmistokirjastoja. (npm, n.d.)

3.5 React.js -kirjasto

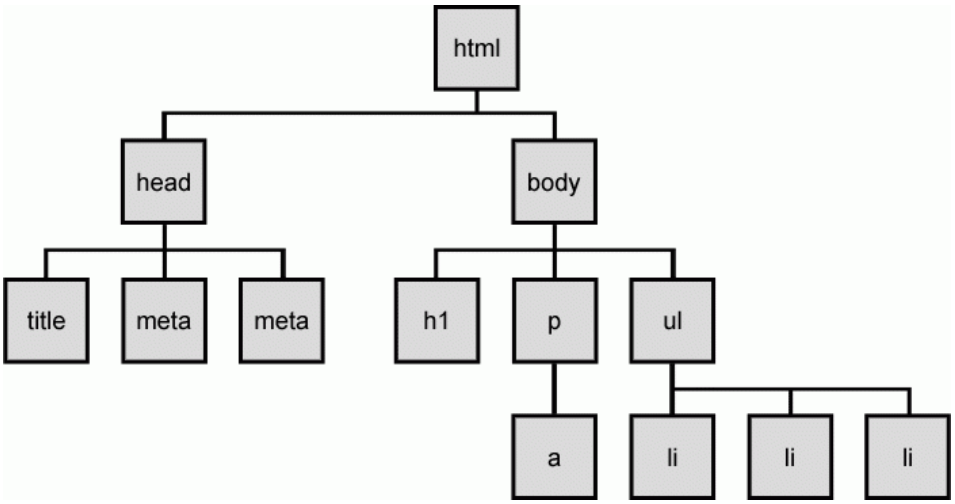
React on avoimen lähdekoodin komponentti-pohjainen käyttöliittymien rakentamiseen tarkoitettu JavaScript-kirjasto. (Facebook, n.d.-a) React syntyi 2010-luvun alussa Facebookin tarpeesta tuottaa helpommin ylläpidettävää koodia laajoissa koodikannoissa. Facebookin mainostajille tarjoaman websovelluksen monimutkaisuuden kasvaessa pienen muutoksen teko yhdessä komponentissa tuotti valtavan määrän työtä muiden toiminnallisuuksien päivittämiseen. Koodista tuli arvaamatonta, heikentäen tiimin luottamusta heidän tekemää työtänsä kohtaan. Tiimi alkoi rakentaa uutta JavaScript-kirjastoa Jordan Walken johdolla ratkaisemaan tämän ongelman. JSConf 2013 -tapahtumassa Tom Occhino ja Jordan Walke esittelivät React-kirjaston. Facebook julkaisi Reactin avoimen lähdekoodin kirjastona kaikkien käytettäväksi samana vuonna. (Occhino, 2015)

React mahdollistaa websovellusten rakentamisen uudelleenkäytettävillä komponenteilla, JavaScript-funktioilla tai luokilla, jotka palauttavat pienen palan HTML-tyylistä koodia. Komponentteja voidaan sisäistää toisiin komponentteihin, luoden monipuolisia, helposti ohjelmoitavia käyttöliittymän osia. Komponentit voivat ottaa vastaan syötteitä (props), joissa määritetään komponentin esittämää sisältöä. (Facebook, n.d.-b)

React-komponentteja kirjoittaessa suositellaan käyttämään JavaScriptia ja HTML-tageja yhdistävää JavaScript-syntaksin laajennusta nimeltä JSX (JavaScript XML). (Facebook, n.d.-b) JSX:n käyttö tekee React-koodista helpommin luettavaa ja kirjoitettavaa, mutta selaimet tai ajoympäristöt eivät sitä ymmärrä. Babel osaa kääntää JSX-syntaksia puhtaaseen JavaScript muotoon. (Babel, n.d.-b)

Verkkosivua avattaessa selain noutaa sivun HTML-koodin, luoden siitä puun kaltaisen haarautuvan tietorakenteen. Jokainen haara koostuu nodeista, objekteista, jotka sisältävät tietoa kyseisessä nodessa olevasta sisällöstä, sekä kaikista sen sisäkkäisistä nodeista. Tätä tietorakennetta kutsutaan nimellä **Document Object Model**, lyhennettynä **DOM** (Kuva 9). JavaScript mahdollistaa DOM-objektien manipuloinnin, jolloin sivulla esitettävää sisältöä voidaan muokata muutosten tapahtuessa. (Haverbeke, 2018b)

Kuva 9. Document Object Modelin rakenne (Kononenko, 2018)

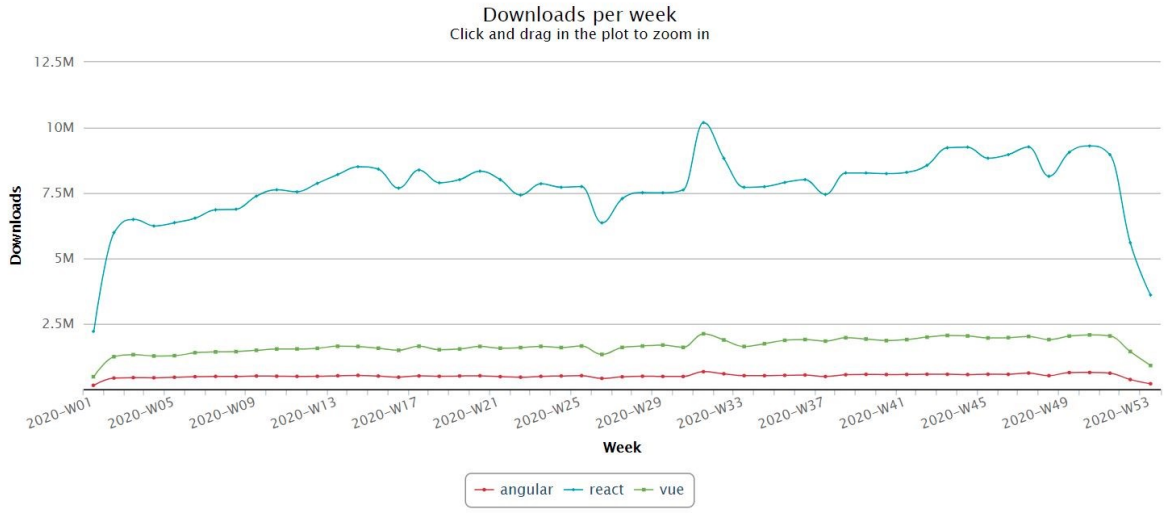


Useimmat JavaScript-ohjelmistokehykset päivittävät DOMia turhan paljon, luoden turhaa työtä, joka hidastaa sivun toimintaa. Tyypillisesti yhdessä elementissä tapahtuva muutos johtaa koko DOM-puun päivittämiseen, jolloin kaikki elementit joudutaan renderöimään uudestaan. React ratkaisee tämän ongelman käyttämällä virtuaalista DOMia, kevyttä kopiota jokaisesta DOM-objektista. React-komponentteja renderöitäessä selaimeen jokainen virtuaalinen DOM-objekti päivitetään. React tallentaa snapshotin virtuaalisen DOMin tilasta juuri ennen muutosta, vertaa päivitettyä tilaa vanhempaan versioon, ja päivittää pelkästään muuttuneet objektit varsinaiseen DOMiin. (Codecademy, n.d.) Käytännössä tämä tarkoittaa sitä, että esimerkiksi kymmenen tuotteen listassa yhden tuotteen tilan päivittyessä vain muuttunut elementti renderöidään uudestaan, sen sijaan että selain tekisi turhaa työtä uudelleen renderöidessään myös kaikki yhdeksän muuttumatonta elementtiä.

React on kolmesta suosituimmasta käyttöliittymien rakentamiseen tarkoitetuista JavaScript-ohjelmistokehyksestä ylivoimaisesti käytetyin. Npm-stat -sivuston mukaan (Kuva 10) vuonna 2020 React.js -kirjastoa ladattiin noin 408 miljoonaa kertaa, yli neljä ja puoli kertaa

enemmän kuin toiseksi suosituinta Vue.js -kirjastoa (noin 87 miljoonaa latausta), ja 15 kertaa enemmän kuin kolmannella sijalla olevaa Angular.js -kirjastoa (noin 26 miljoonaa latausta). (npm-stat, n.d.)

Kuva 10. React, Vue ja Angular -kirjastojen latauskerrat vuonna 2020 (npm-stat, n.d.)



3.6 Progressive Web App

Progressive Web App, lyhennettynä PWA, tarkoittaa websovellusta, jonka käyttökokemus on verrattavissa natiiviin mobiilisovellukseen. PWA:n kehitykseen käytetään samoja teknologioita kuin perinteisen websovelluksen kanssa, eli HTML määrittää sivun rakenteen, CSS tyylittelee HTML-elementit, ja JavaScript lisää websovellukseen toiminnallisuutta. PWA:n kehitykseen voidaan käyttää esimerkiksi React.js -kirjastoa, kuten perinteisissä websovelluksissa. (Magomadov, 2020)

PWA voidaan asentaa selaimesta käyttäjän mobiililaitteen aloitusruudulle. Palvelimelta ladattu käyttöliittymä tallennetaan mobiililaitteen välimuistiin (cache), jolloin sovelluksen sivujen väliset siirtymät voidaan noutaa välimuistista epävakaasta internet-yhteydestä huolimatta. PWA:n sisältöön päästään käsiksi URL-osoitteen kautta, jolloin sisältö on jaettavissa toisin kuin natiiveissa mobiilisovelluksissa. PWA eroaa perinteisestä websovelluksesta lisäämällä natiiveista mobiilisovelluksista tuttuja toiminnallisuuksia, kuten offline-tilan, push-notifikaatiot, sekä kameran tai mikrofonin käytön. PWA:t ovat selainpohjaisia websovelluksia, joka tekee niistä alustariippumattomia. (Magomadov, 2020)

PWA:ksi luokiteltavan websovelluksen täytyy sisältää vähintään seuraavat kolme asiaa:

Turvattu tiedonsiirto HTTPS-protokollan kautta. Turvatulla tiedonsiirrolla varmistetaan sivuston luotettavuus, jota tarvitaan useimpien natiivien mobiilisovellusten kaltaisten toiminnallisuuksien, kuten esimerkiksi GPS-paikannuksen, käyttämiseen. (Mozilla, n.d.-c)

Service worker. JavaScript-tiedosto, joka ajetaan irrallaan sivustosta. Service worker pystyy sieppaamaan ja hallitsemaan palvelinkutsujen lähettämistä, tallentamaan websovelluksen käyttöliittymässä esiintyviä elementtejä välimuistiin, sekä mahdollistamaan natiiveista mobiilisovelluksista tuttuja toimintoja, kuten push-notifikaatiot tai GPS-paikannuksen. (Mozilla, n.d.-c)

Manifest. JSON-tiedosto (JavaScript Object Notation), jolla hallitaan websovelluksen ulkonäköä mobiililaitteelle asennettuna, ja mahdollistetaan PWA:n asennus suoraan selaimesta. PWA-manifesti sisältää websovelluksen nimen, osoitteen, käytettävät ikonit, kuvauksen, sekä listan tarvittavista resursseista, joilla websovelluksesta käytöstä saadaan natiivin mobiilisovelluksen kaltainen. (Mozilla, n.d.-c)

4 Testaustyö käytännössä

Tässä luvussa kerrotaan käytännön osuuteen valituista testaustyökaluista, sekä havainnollistetaan testaustyötä käytännön esimerkeillä. Ensimmäisessä esimerkissä tehdään erittäin yksinkertainen JavaScipt-funktio, jonka toiminnallisuutta testataan Jest-testauskehyksellä. Toisessa esimerkissä testataan React-komponenttia Jestillä ja React Testing Library-testauskirjastolla.

4.1 Testauskirjastot

JavaScriptin testaukseen on saatavilla useita vaihtoehtoja työkaluja valittaessa. Tämän opinnäytetyön työkaluiksi valitut kolme testauskirjastoa ja kehystä ovat React-sovelluksille suositeltuja, hyvin dokumentoituja, monipuolisia ja helppokäyttöisiä. Työkalujen lähdekoodi on myös avoimesti kaikkien saatavilla.

Facebook suosittelee Reactin virallisessa dokumentaatiossa Jest ja React Testing Library -työkaluja React-sovellusten testaukseen. (Facebook, n.d.-c) State of JavaScript 2020 -kyselyyn vastasi 23 765 kehittäjää 137 eri maasta. Testauskehysten kategoriassa tyytyväisyyttä mitattaessa 97% vastaajista valitsi ”käyttäisin uudestaan”-vaihtoehdon Testing Libraryn kohdalla. Toiseksi kategoriassa sijoittui Jest 96% osuudella, ja kolmanneksi Cypress 94% osuudella (*State of JS 2020: Testing*, 2020)

Da Costa (The React testing ecosystem-luku, An overview of React testing libraries-alaluku) suosittelee React Testing Libraryn käyttöä React-projekteissa, sillä kuten nimestä voi arvata, se on räätälöity React-sovellusten testaukseen. Da Costa (2021, What to test and when-luku, End-to-end test-alaluku, Testing GUIs-alaluku) mainitsee myös, että Cypress, TestCafe ja Selenium ovat tällä hetkellä suosituimpia käyttöliittymän testaustyökaluja, joista Cypress valikoitui tässä työssä käytettäväksi.

Jest on Facebookin kehittämä JavaScript-testauskehys (testing framework). Jest sisältää loistavan rajapinnan testitapausten kirjoittamiseen, sekä testiajurin (test runner), jolla testit suoritetaan Node.js-prosessina. Jest tarjoaa testeissä suoritettavien varmentamisten (assert) toiminnot, testien kattavuusraportit (coverage), sekä mahdollisuuden mockata toimintoja,

rajapintoja, tai mitä tahansa muuta jota itse testitapauksessa halutaan rajata pois. Mockauksella voidaan esimerkiksi korvata palvelimelle lähetetyn kutsun vastaus kovakoodatulla tiedolla, jos testissä halutaan vain varmistaa kutsun lähettäminen ja palvelimelta saadun vastauksen oikeanlainen käsittely. Jest valikoitui tähän opinnäytetyöhön sen suuren suosion, virallisen React-dokumentaation suosituksen, sekä sen testiajurin ansiosta. Muut testikirjastot voivat käyttää Jestiä ajoalustana testejä suoritettaessa. (Jest, n.d.)

Testing Library on Kent C. Doddsin kehittämä JavaScript-testauskirjasto, joka sisältää suuren määrän toimintoja rajapintansa kautta. Toisin kuin Jest, Testing Library on vain JavaScript-kirjasto, jolla ei ole omaa testien ajamiseen vaadittavaa testiajuria. Testing Libraryn rajapintaa voidaan kuitenkin käyttää testien kirjoittamiseen, ja Jestiä tai muita testiajureita kuten Mocha voidaan käyttää testien ajamiseen. (Testing Library, n.d.)

Testing Libraryn perustana toimii *DOM Testing Library*, joka mahdollistaa websovelluksen testauksen JSDOMissa tai JestDOMissa simuloitujen DOM-nodejen kautta. Tällä lähestymistavalla vältytään testattavan komponentin implementoinnin testauksesta, joka tekee testeistä luotettavampia, ja mahdollistaa komponenttien refaktoroinnin rikkomatta testejä. Testing Library tarjoaa räätälöidyt testikirjastot esimerkiksi React.js-, Vue.js-, ja Angular.js-kirjastoille, sekä laajentaa Cypress- testauskehyksen toimintaa. (Testing Library, n.d.)

Cypress on end-to-end -testaukseen erikoistunut testauskehys, jolla testejä suoritetaan käyttäjän näkökulmasta selaimessa. Cypress-testit kirjoitetaan JavaScript-tiedostoihin käyttäen cy.*toiminto* -kutsuja. Cypress keskustelee testiä suoritettaessa suoraan selaimen rajapinnan kanssa. Cypress tallentaa testejä suoritettaessa jokaisesta vaiheesta tilannekuvan (snapshot), jolloin kehittäjä voi testin suorituksen jälkeen matkustaa testitapauksessa ajassa taaksepäin ja tarkastella sovelluksen tilaa koko testisuorituksen ajalta vaiheittain. Cypress asennetaan yhtenä pakettina, joten se on erittäin nopeasti käyttöön otettavissa. Cypress ei ole sidoksissa testattavan sovelluksen lähdekoodiin, joten se voidaan asentaa joko projektiin sisäisesti kehittäjäriippuvuutena tai sen ulkopuolelle omaan hakemistoonsa. Cypressiä käytetään sen oman graafisen käyttöliittymän kautta. (Cypress, n.d.-a)

4.2 JavaScript-testaus

Ennen ensimmäisen React-testin kirjoitusta on tärkeää ymmärtää miten testaus toimii JavaScript-kielellä. HAMK Smartin kehittäjätiimin ja opinnäytetyön tekijän valinta koodieditorista on Microsoftin kehittämä VS Code. Editorista voidaan integroidun terminaalin kautta asentaa kirjastoja ja ajaa testejä käyttämällä Node Package Managerin npm-komentoja. Komennot voidaan myös suorittaa cmd- tai powershell-komentorivillä.

JavaScriptin testauksen perusteiden esittelyä varten olen luonut uuden tyhjän projektikansion. Komennon 1 ensimmäisellä rivillä projektille alustetaan **package.json** -tiedosto, jotta npm-ohjelmistorekisteristä voidaan lisätä Jest projektin riippuvuuksiin. Komennon 1 toisella rivillä Jest asennetaan kehittäjä-riippuvuutena (dev dependency) lisäämällä komentoon `--save-dev`, sillä testaustyökaluja ei tarvita sovellusten julkaistaviin versioihin.

Komento 1 Package.json -tiedoston alustus, Jest-testauskehyksen asennus

```
npm init -y
npm install --save-dev jest
```

Ohjelmakoodi 8:sa package.json -tiedoston scripts-osioon lisätään komento *”test”: ”jest”*, jolla Jest suoritetaan. Jest osaa automaattisesti etsiä projektista *.test.js* -päätteiset tiedostot ja ajaa niihin kirjoitetut testit.

Ohjelmakoodi 8. Jest-komento npm-skriptinä

```
{
  "scripts": {
    "test": "jest"
  },
}
```

Yksinkertaista testiä varten luon projektiin **script.js** -tiedoston (Ohjelmakoodi 9), johon lisään testattavan funktion. Funktio ottaa vastaan kaksi parametria ja palauttaa niiden summan. Jotta funktiota voidaan käyttää testissä, se pitää viedä alkuperäisestä tiedostosta *module.exports*-komennolla.

Ohjelmakoodi 9. sum-funktio (Lahti, 2019)

```
const sum = (a, b) => {
  return a + b;
};

module.exports = sum;
```

Testiä varten luon **script.test.js**-tiedoston, käyttäen samaa nimeä kuin testattavassa tiedostossa, lisäten nimen päätteeseen sana test, jolloin Jest tunnistaa sen testitiedostoksi. Ohjelmakoodissa 10 tuomme sum-funktion testitiedoston käytettäväksi *require(path)*-komennolla.

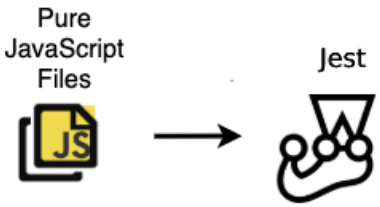
Testi alustetaan *test*- tai *it*-funktiolla, joka vastaanottaa ensimmäisenä argumenttina testin kuvauksen, ja toisena argumenttina funktion jossa testi suoritetaan. Testissä *sum*-funktio suoritetaan syöttäen sille numerot 1 ja 2, ja funktiosta palautuva arvo tallennetaan muuttujaan *result*. Tulosta verrataan, eli assertoidaan, *expect*-funktiolla syöttämällä sille argumenttina *sum*-funktion tulos.

Expect-funktio vaatii tulosta verratakseen *matcherin*, toisen funktion, jolle syötetään argumenttina oletettu arvo, tässä tapauksessa sum-funktiosta oletettu palautuva luku. Expect-funktion perään linkitetty *toEqual*-funktio suorittaa *deep equality*-vertauksen, eli tarkistaa ovatko primitiiviarvot samoja. Kuvassa 11 yksinkertainen polku, jossa ”puhdas” JavaScript, tarkoittaen ES5-standardin mukaista tai vanhempaa JavaScriptia, ajetaan ja vertaillaan käyttäen Jestin rajapinnan funktioita.

Ohjelmakoodi 10. Testitapaus: JavaScript, sum-funktio

```
const sum = require('./script');

test('sum returns correct result', () => {
  const result = sum(1, 2);
  expect(result).toEqual(3);
});
```

Kuva 11. Testin polku: JavaScript



Suoritan testin syöttämällä komennon 2 komentoriville. Kuvassa 12 esitetään testin tulos, *sum*-funktio toimii kuten oletettiin. Varmistan vielä, että testi ilmoittaa virheestä antamalla *toEqual*-funktiolle väärän oletetun arvon, ja ajan testin uudelleen. Kuvassa 13 Jest ilmoittaa, ettei funktiosta palautunut arvo vastaa oletettua. Jest avustaa kehittäjää tunnistamaan virheen käyttämällä vihreää ja punaista fonttia virheviestissä, jolloin oletettu arvo ja funktiosta saatu arvo tunnistetaan helposti. Jest printtaa myös komentoriville testin koodin, nuolilla osoittaen kohtaan jossa virhe ilmestyi.

Komento 2. Testien ajokomento

```
npm test
```

Kuva 12. Testitulos: sum-testi onnistuu

```
PASS ./script.test.js
  √ sum returns correct result (2 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.565 s
Ran all test suites.
```

Kuva 13. Testitulos: sum-testi rikotaan tahallaan, testi epäonnistuu

```
● sum returns correct result

  expect(received).toEqual(expected) // deep equality

  Expected: 4
  Received: 3

    3 | test('sum returns correct result', () => {
    4 |   const result = sum(1, 2);
  > 5 |   expect(result).toEqual(4);
      |                  ^
    6 | });
    7 |
```

React-kehittäjät käyttävät yleisesti ES6-standardin moduulien *import/export*-syntaksia tuodessaan komponentteja toisista tiedostoista tai kirjastoista. Ohjelmakoodissa 11 script.test.js-tiedoston *require(path)*-komento vaihdetaan uudempaan *import*-muotoon. Testi ajetaan uudelleen. Kuvassa 14 näkyy epäonnistuneen testin virheviesti ”*Jest encountered an unexpected token*”. Virheviestissä vihjataan koodin sisältävän syntaksia, jota Jest ei ymmärrä, sillä se ei ole niin sanotusti puhdasta JavaScriptia. Viestissä annetaan ohjeet, joita noudattamalla *import*-komento saadaan toimimaan Jestin kanssa. Viestissä mainitaan Jestin käyttävän Babelia automaattisesti tiedostojen kääntämiseen, jos se löytää projektin tiedostoista Babelin asetukset, eli *.babelrc*, *babel.config.js*, tai *babel.config.json* -tiedoston.

Ohjelmakoodi 11. Testitapaus: JavaScript, sum-funktio ES6 import-komennolla

```
import sum from './script';

test('sum returns correct result', () => {
  const result = sum(1, 2);
  expect(result).toEqual(3);
});
```

Kuva 14. Testitulos: sum-testi epäonnistuu, Jest ei osaa tulkata import-komentoa

```
 FAIL  ./script.test.js
  ● Test suite failed to run

    Jest encountered an unexpected token

    This usually means that you are trying to import a file which Jest cannot parse, e.g. it's not plain JavaScript.

    By default, if Jest sees a Babel config, it will use that to transform your files, ignoring "node_modules".

    Here's what you can do:
     • If you are trying to use ECMAScript Modules, see https://jestjs.io/docs/en/ecmascript-modules for how to enable it.
     • To have some of your "node_modules" files transformed, you can specify a custom "transformIgnorePatterns" in your config.
     • If you need a custom transformation specify a "transform" option in your config.
     • If you simply want to mock your non-JS modules (e.g. binary assets) you can stub them out with the "moduleNameMapper" confi
g option.

    You'll find more details and examples of these config options in the docs:
    https://jestjs.io/docs/en/configuration.html

    Details:

    D:\Programming\JS\opinnaytetyo\VanillaJS\script.test.js:1
    ({"Object.<anonymous>":function(module,exports,require,__dirname,__filename,global,jest){import sum from './script';
                                                                                             ^^^^^^

    SyntaxError: Cannot use import statement outside a module

      at Runtime.createScriptFromCode (node_modules/jest-runtime/build/index.js:1350:14)

Test Suites: 1 failed, 1 total
Tests:       0 total
Snapshots:   0 total
Time:        0.584 s, estimated 1 s
Ran all test suites.
```

Lisään projektiin Babel-kääntäjän kehittäjäriippuvuutena (Komento 3), jotta se voi kääntää *import/export*-komennot ES5-standardin muotoon jota Jest ymmärtää. Luon Babelille asetustiedoston nimellä **babel.config.js** (Ohjelmakoodi 13), johon lisään asetukset joilla saan Jestin toimimaan Babelin kautta käännetyn koodin kanssa. Preset-env mahdollistaa uusimman JavaScriptin käytön, tunnistaen sille *targets*-asetuksessa määriteltyjen selainten tai ajoympäristöjen versioiden tarvitsemat käännökset ja polyfillit. Kuvassa 15 testin polku, jossa ES6 moduuleja käyttävät tiedostot käännetään ennen Jestillä ajoa.

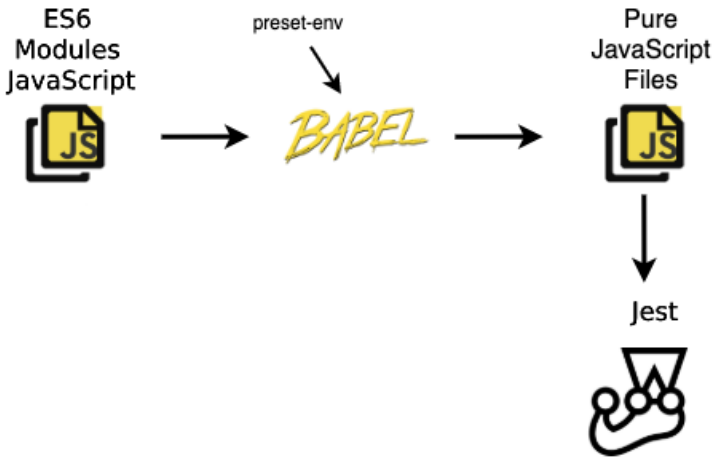
Komento 3. Babelin lisäys projektiin

```
npm install --save-dev babel-jest @babel/core @babel/preset-env
```

Ohjelmakoodi 12. Testitapaus: sum ES6 import, babel-kääntäjän asetukset

```
module.exports = {
  presets: [
    [
      '@babel/preset-env',
      {
        targets: { node: 'current' },
      },
    ],
  ],
};
```

Kuva 15. Testin polku: ES6 moduulit



4.3 React-testaus

Reactin testauksen perusteiden esittelyä varten luon toisen projektin. Tällä kertaa käytän *Create React App* -pohjaa, sillä React-projektin alustus ja asennukset eivät kuulu tämän opinnäytetyön aiheeseen. Syöttämällä komennon 4 komentoriville npm-ohjelmistorekisteristä haetaan ja asennetaan tarvittavat kirjastot, luodaan projektille tiedostot ja komennot joilla projekti saadaan nopeasti käyttöön.

Komento 4. React-projektin alustus Create React App-komennolla

```
npx create-react-app projektin-nimi
```

Create React App-pohjalla alustettu projekti sisältää valmiiksi konfiguroidun testausympäristön Jestiä ja React Testing Librarya käyttäen. Suurin osa CRA-pohjan mukana asennetuista toiminnallisuuksista ovat osa *react-scripts* -pakettia, jossa asennetaan muun muassa Babel-kääntäjä ja Webpack (bundler), joilla projektin saa heti ajettavaksi selaimeen ja testit pyörimään.

Tässä esimerkissä käytämme Jestiä, React Testing Librarya, sekä Jest-DOM -rajapintaa. Testityökalujen asennuksen voi tehdä manuaalisesti syöttämällä komennon 5 komentoriville. Ohjelmakoodissa 13 babel-asetuksiin on lisätty nyt myös @babel/preset-react.

Komento 5 React-projektin testauskirjaston asennukset

```
npm install --save-dev babel-jest @babel/core @babel/preset-env
@babel/preset-react @testing-library/jest-dom @testing-library/react
```

Ohjelmakoodi 13. Testitapaus: Counter, babel-kääntäjän asetukset

```
module.exports = {
  presets: [
    [
      '@babel/preset-env',
      {
        targets: { node: 'current' },
      },
    ],
    '@babel/preset-react',
  ],
};
```

Teen yksinkertaisen sovelluksen joka nostaa, laskee, tai resetoi selaimessa näytettävän numeron lukua painikkeita painamalla. Ohjelmakoodissa 14 Counter-komponentti on tavallinen JavaScript-funktio joka palauttaa JSX-syntaksia, eli esitettävän komponentin sisällön HTML-merkintäkieltä muistuttavassa muodossa.

Sovelluksessa näytettävän luvun tilan hallintaan käytetään Reactin useState-hookkia. Hookkia alustettaessa luvulle annetaan arvoksi luku nolla useState-funktion argumenttina. Tallennan hookista kaksi muuttujaa: count sisältää esitettävän numeron arvon, eli useState-funktiolle syötetyn luvun nolla, ja setCount-funktion jolla count-muuttujaa voidaan muuttaa. React päivittää selaimen sisällön aina hookkien arvojen muuttuessa, joten niihin pitäisi tallentaa ainoastaan arvoja, joita haluat selaimessa esittää. Painikkeille on luotu omat handler-funktiot joita ne kutsuvat painiketta klikatessa.

Ohjelmakoodi 14. React-komponentti: Counter

```
import React, { useState } from 'react';

const Counter = () => {
  const [count, setCount] = useState(0);

  const increment = () => setCount(count + 1);
  const decrement = () => setCount(count - 1);
  const reset = () => setCount(0);

  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={increment}> + </button>
      <button onClick={decrement}> - </button>
      <button onClick={reset}>Reset</button>
    </div>
  );
};

export default Counter;
```

Testaan painikkeiden toimintaa luomalla **Counter.test.jsx** -tiedoston. Ohjelmakoodin 14 testeissä käytetään React Testing Librarya ja Jestiä. React Testing Library tulostaa testattavan komponentin selainympäristöä simuloiden DOM-nodeina, niin sanotussa headless-tilassa eli Node.js-prosessina sen sijaan että se avaisi varsinaisen selaimen. Render-funktiolle

syötetään tulostettava komponentti argumenttina, ja se palauttaa objektin joka sisältää hakufunktiota, kuten *getByText*, joilla etsitään DOM-puusta elementtejä.

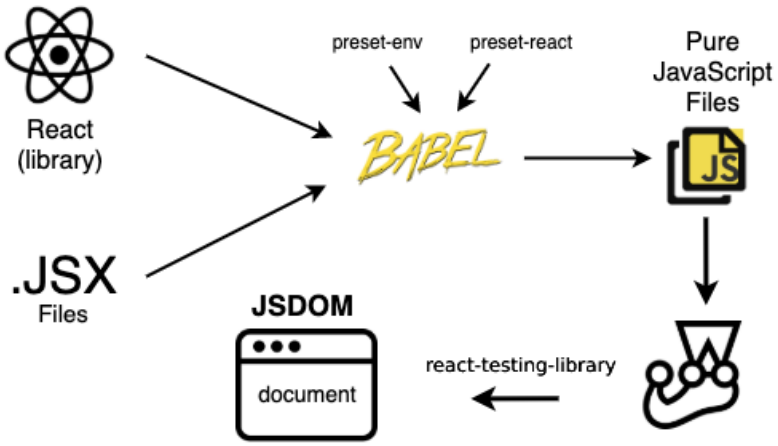
Painikkeiden klikkaukseen käytän React Testing Libraryn fireEvent.click-funktiota, jolle syötän klikkaukseni kohteeksi painikkeen tekstiä etsivän *getByText*-hakufunktion. Tulosta vertailtaessa käytän *@testing-library/jest-dom* -pakettiin kuuluvaa *toBeInTheDocument* matcher-funktiota, joka tarkistaa löytyykö elementti tulostetusta DOM-puusta. Jokaisessa testissä Counter-komponentti renderöidään alusta asti uudelleen, joten jokaisen testin alussa count-arvo nollataan.

Ohjelmakoodi 15. Testitapaus: React.js, Counter-komponentti

```
import React from 'react';
import { render, fireEvent } from '@testing-library/react';
import '@testing-library/jest-dom';
import Counter from './Counter';

test('Counter increments', () => {
  const { getByText } = render(<Counter />);
  fireEvent.click(getByText('+'));
  expect(getByText('Count: 1')).toBeInTheDocument();
});

test('Counter decrements', () => {
  const { getByText } = render(<Counter />);
  fireEvent.click(getByText('-'));
  expect(getByText('Count: -1')).toBeInTheDocument();
});

test('Counter resets', () => {
  const { getByText } = render(<Counter />);
  fireEvent.click(getByText('+'));
  fireEvent.click(getByText('Reset'));
  expect(getByText('Count: 0')).toBeInTheDocument();
});
```

Kuvassa 16 esitetään testin polku, jossa React-kirjastosta käytetty koodi sekä JSX-tiedostot käännetään Babelin kautta ”puhtaaksi” JavaScriptiksi ennen Jestillä ajoa. Jest käyttää React Testing Libraryn *render*-funktiota tulostaessaan komponentin JSDOMiin nodeina, *fireEvent*-funktiota painikkeita klikatessa, sekä *render*-funktiosta tallennettua *getByText*-hakufunktiota, jolla haetaan DOMissa esiintyviä elementtejä.

Kuva 16. Testin polku: React

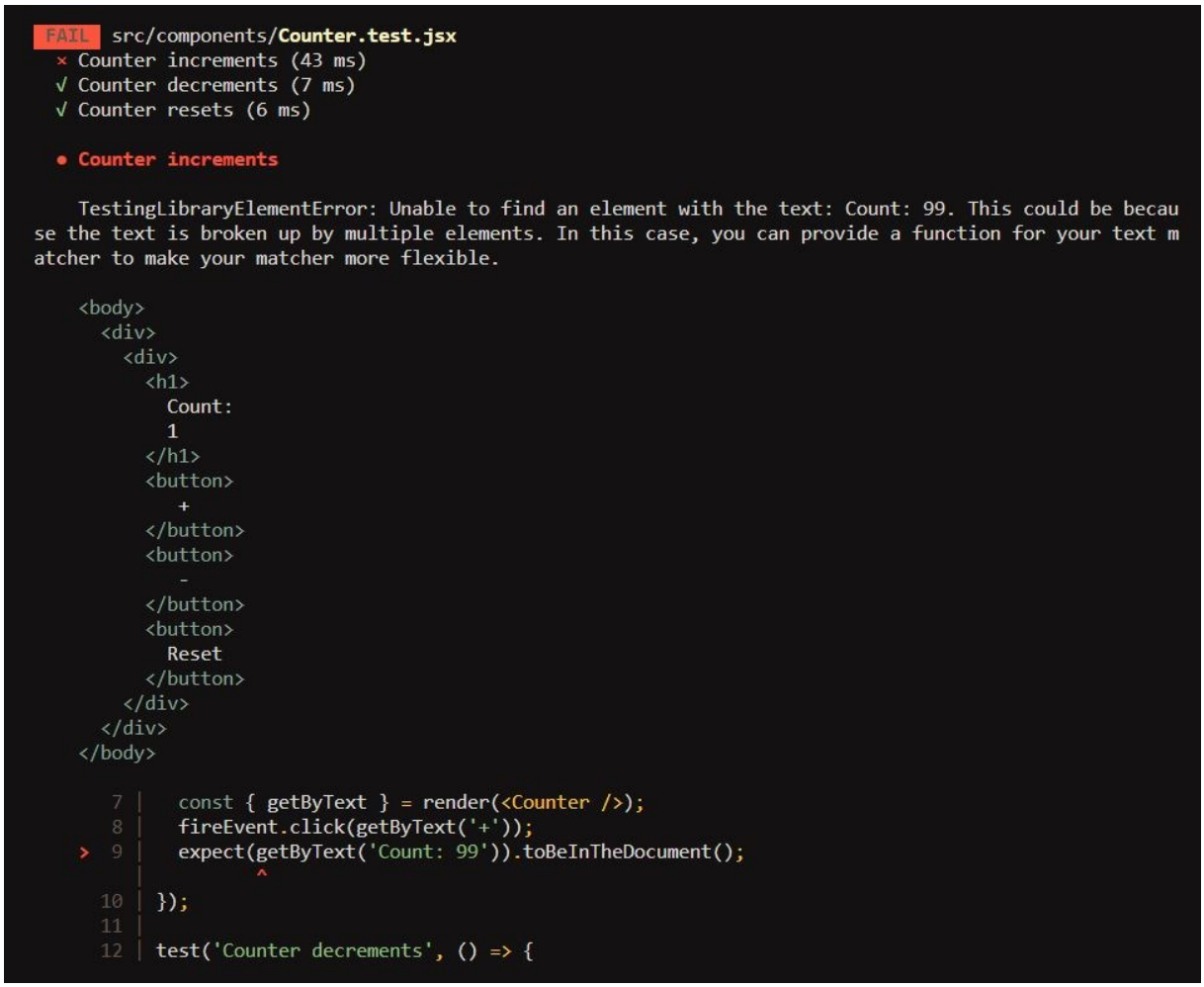


Suoritan testit npm test-komennolla. Kuvassa 17 Jest ilmoittaa onnistuneista testeistä. Varmistan testien toimivuuden rikkomalla ensimmäisen testin tahallaan syöttäen sille väärän oletetun arvon. Kuvassa 18 React Testing Library ilmoittaa virheestä tulostaen komentoriville DOM-rakenteen johon se vertausta teki.

Kuva 17. Testitulos: Counter-testi onnistuu

```
PASS src/components/Counter.test.jsx
  √ Counter increments (43 ms)
  √ Counter decrements (6 ms)
  √ Counter resets (6 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        2.975 s, estimated 4 s
Ran all test suites.
```

Kuva 18. Testitulos: Counter-testi rikotaan tahallaan, testi epäonnistuu



5 Testaustyön tulokset

Tässä luvussa kerrotaan Kaupunki taskussa -websovelluksesta ja sille opinnäytetyön aikana kirjoitetuista testitapauksista. Luvussa ei tarkastella toimeksiantajan lähdekoodia, vaan käydään läpi tapauskohtaisesti testattava toiminnallisuus perustellen sille valittu testaustapa. Testit selitetään, ajetaan, sekä rikotaan toimivuuden varmentamiseksi.

5.1 Kaupunki taskussa -websovelluksen esittely

Hämeenlinnan kaupunki osallistuu valtiovarainministeriön avustamaan Kaupunki taskussa -hankkeeseen, jonka yhteydessä tuotetaan kaikkien Suomen kuntien ja kaupunkien käyttöön tarjottava mobiilisovellus. Sovelluksen tuotannosta vastaa Hämeen Ammattikorkeakoulu. Hankkeen yhteydessä Hämeenlinna taskussa -sovellus (Kuva 19) päivitetään kevään 2021 aikana, ja sovellus julkaistaan uudistetulla teknologialla kehitettynä laite- tai pääteriippumattomana websovelluksena. (Hämeenlinnan kaupunki, 2021)

Kaupunki taskussa-websovellus tarjoaa tietoa kaupungin ajankohtaisista tapahtumista ja uutisista, sisältää sähköisen kirjastokortin, julkisen liikenteen bussiaikataulut ja paljon muuta. Tässä opinnäytetyössä käsiteltävä versio websovelluksesta sisältää rajallisen määrän toiminnallisuutta, sillä työn on tarkoitus avustaa Hämeen Ammattikorkeakoulun kehittäjätiimin testauskäytäntöjä. Sovellus on PWA, eli alustariippumaton mobiililaitteelle asennettavissa oleva websovellus, ja sen kehitys on toteutettu React.js -kirjastolla.

Kuva 19. Hämeenlinna taskussa -sovelluksen tuotekuva (Google Play Store, n.d.)

5.2 Yksikkötesti: Bussin mobiililippu-moduuli

Kaupunki taskussa-websovelluksen navigaatiopalkista Bussin mobiililippu-painiketta painamalla käyttäjä ohjataan laitteesta riippuen joko Waltti-matkakortin verkkosivulle, Apple Storeen tai Google Play Storeen Waltti mobiilisovelluksen asennussivulle. Testattava navigointi-komponentti käyttää laitteen tunnistukseen *react-device-detect* -kirjastoa. Navigointi-komponenttia tulostettaessa laitteen käyttöjärjestelmälle suoritetaan tarkistus käyttäen *react-device-detect* -kirjaston *isIOS*- ja *isAndroid*-funktioita. Funktiot palauttavat boolean-arvon true tai false. Testissä käytetään ainoastaan Navigointi-komponenttia, joka ei itsessään sisällä muita komponentteja. Täten testi voidaan luokitella yksikkötestiksi.

Jest mahdollistaa rajapintojen ja kirjastojen toiminnallisuuksien korvaamisen omalla mockatulla funktiolla. Sen sijaan, että linkin testiä varten yrittäisimme huijata *react-device-detect* -kirjastoa antamaan meille halutun tuloksen käyttöjärjestelmästä, voimme suoraan korvata *isIOS* ja *isAndroid*-funktioista palautuvat arvot itse valitsemillamme arvoilla käyttäen Jestin mock-toimintoa.

Teen projektin juureen __mocks__ -kansion, jonne luon **react-device-detect.js** -nimisen tiedoston. Ohjelmakoodissa 16 käytän Jestin *createMockFromModule*-funktiota, antaen sille argumenttina mockattavan kirjaston nimen, ja tallennan sen deviceDetect-muuttujaan. Muuttuja viedään tiedostosta *module.exports*-komennolla.

Ohjelmakoodi 16. Ulkoisen kirjaston korvaus Jestin mock-funktiolla

```
const deviceDetect = jest.createMockFromModule('react-device-detect');
module.exports = deviceDetect;
```

Ohjelmakoodissa 17 tuomme testitiedostoon *react-device-detect* -kirjaston käyttäen nimeä deviceDetect. Täten voimme hallita mockatuista kirjaston funktioista palautuvia arvoja itse testitiedostossa. Jest tarjoaa rajapintansa kautta pääsyn omiin hookkeihinsa. Jest-hookilla tarkoitetaan funktiokutsua, joka voidaan suorittaa esimerkiksi ennen kaikkien testien ajoa (beforeAll), kaikkien testien ajon jälkeen (afterAll), ennen jokaista yksittäistä testiä (beforeEach), tai jokaisen yksittäisen testin jälkeen (afterEach). Describe-funktiolla testejä

voidaan ryhmittää omiin pienempiin lohkoihinsa, ja niiden sisälle voidaan luoda vain kyseiseen lohkon testeihin vaikuttavat Jest-hookit.

Testatessa Bussin mobiililippu-painikkeen toimintaa kolmella eri testillä kannattaa mockatun deviceDetect-kirjaston *isIOS*- ja *isAndroid*-funktioista palautuvat arvot asettaa epätodeksi ennen jokaisen yksittäisen testin ajoa käyttäen beforeEach-hookkia. Tällöin testit voidaan ajaa missä järjestyksessä tahansa, toisistaan erillään tai yhdessä tuottaen saman tuloksen.

Ensimmäisessä testissä asetan mockatun deviceDetect-kirjaston *isIOS*-arvon todeksi. Käyttämällä React Testing Libraryn render-metodia Navbar-komponentti renderöidään Node.js-prosessina virtuaalisina DOM-nodeina JSDOMiin, jossa Navbarin käyttämä *react-device-detect* -kirjasto ladataan Jestin mockaamalla versiolla. Navigaatiolinkkejä valitessa *isIOS* palauttaa *true*-arvon, jolloin linkkiin valitaan Apple Storen osoite.

React Testing Libraryn render-funktio palauttaa objektin joka sisältää useita hakufunktioita, kuten *findByTitle*, *getByText* ja *queryByTestId*. Hakufunktioilla renderöidyistä DOM-nodeista voidaan etsiä elementtejä, kuten painikkeita käyttäen painikkeen tekstiä hakusanana. Hakufunktioita voidaan kutsua ohjelmakoodissa 17 ensimmäisen testin mukaan tallentamalla render-funktiosta palautuva objekti muuttujaan ja käyttämällä pistenotaatiota hakufunktioiden kutsumiseen. Toinen, minun mielestä parempi tapa hakufunktioiden kutsumiseen on käyttää ohjelmakoodin 17 toisen ja kolmannen testin mukaista objektin destrukturointia, jossa funktio tallennetaan omaan muuttujaan.

Komponentista haetaan elementtiä tekstillä ”Bussin mobiililippu” käyttäen *render*-metodista tallennettua *getByText* hakufunktiota. Komponentista palautuu objekti, joka tallennetaan link-muuttujaan. Link-objektin href-arvoa verrataan sovelluksen lähdekoodista haettuun Apple Storen osoitteeseen käyttäen Jestin *toBe* matcher-funktiota.

Toista testiä ennen beforeEach-hookissa *isIOS*- ja *isAndroid*-arvot muutetaan epätodeksi, ja testin ensimmäisellä rivillä *isAndroid*-arvo muutetaan todeksi. Navbar-komponentti renderöidään JSDOMiin ja siitä etsitään ”Bussin mobiililippu”-painike. Elementin href-arvoa verrataan Google Play Storen osoitteeseen. Kolmatta testiä ajettaessa *isIOS*- ja *isAndroid*-arvot ovat taas epätosia, joten niitä ei tarvitse testissä itsessään vaihtaa. Haussa käytän tällä kertaa tarkempaa *getByRole*-funktiota, joka ottaa vastaan elementin roolin sekä sitä

kuvaavan objektin. Elementin href-arvoa verrataan tällä kertaa Waltti-sivuston verkko-osoitteeseen.

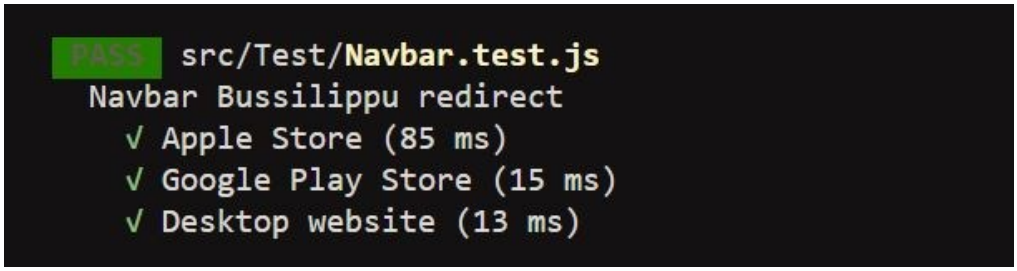
Ohjelmakoodi 17. Testitapaus: Bussin mobiililippu

```
import React from 'react';
import { render } from '@testing-library/react';
import Navbar from '../js/NavBar';
import * as deviceDetect from 'react-device-detect';
import navitems from '../../routekeys.json';

describe('Navbar Bussilippu redirect', () => {
  const ios_url = navitems.routekeys[2].market_links.ios_link;
  const android_url = navitems.routekeys[2].market_links.android_link;
  const pc_url = navitems.routekeys[2].url;

  beforeEach(() => {
    deviceDetect.isAndroid = false;
    deviceDetect.isIOS = false;
  });

  test('Apple Store', () => {
    deviceDetect.isIOS = true;
    const component = render(<Navbar />);
    const link = component.getByText('Bussin mobiililippu');
    expect(link.href).toBe(ios_url);
  });

  test('Google Play Store', () => {
    deviceDetect.isAndroid = true;
    const { getByText } = render(<Navbar />);
    const link = getByText(/bussin mobiililippu/i);
    expect(link.href).toBe(android_url);
  });

  test('Desktop website', () => {
    const { getByRole } = render(<Navbar />);
    const link = getByRole('link', { name: /bussin mobiililippu/i });
    expect(link.href).toBe(pc_url);
  });
});
```

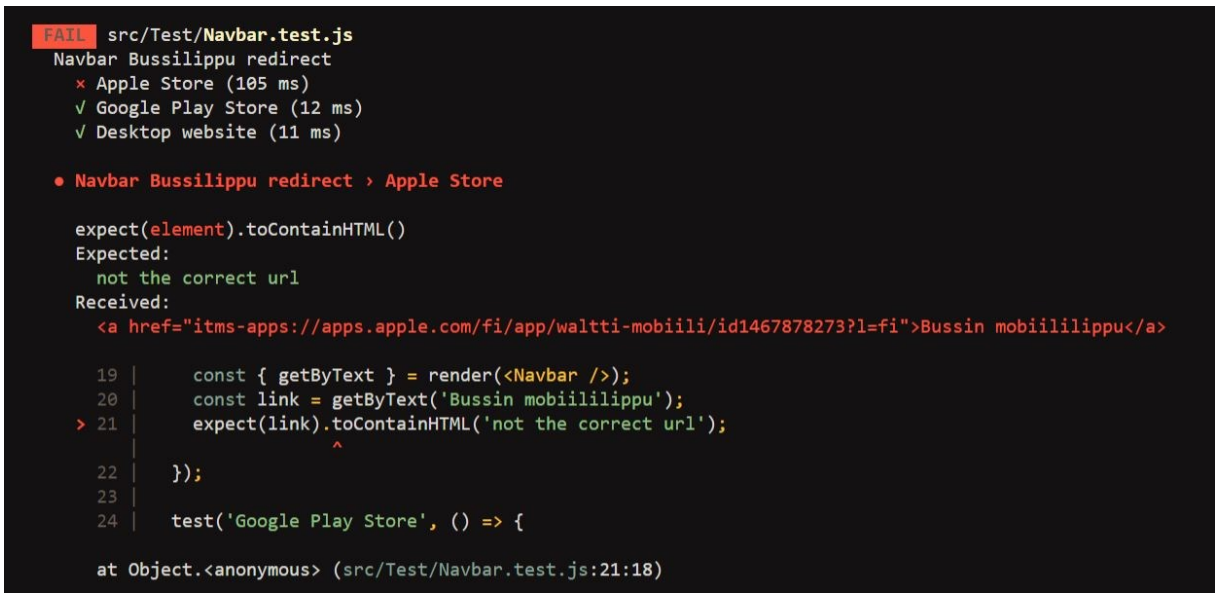
Kuvassa 20 esitetään onnistuneen testin tulos. Varmistaakseni testin toimivuuden vaihdan ensimmäisen testin matcher-funktion argumentiksi tekstin ”*not the correct url*” ja ajan testin

uudelleen. Kuvassa 21 Jest ilmoittaa virheviestillä testin epäonnistuneen, sekä tulostaa linkin varsinaisen sisällön komentoriville.

Kuva 20. Testitulos: Bussin mobiililippu-testi onnistuu



Kuva 21. Testitulos: Bussin mobiililippu-testi rikotaan tahallaan, testi epäonnistuu



5.3 Integraatiotesti: Tapahtumat-moduuli

Kaupunki taskussa -websovelluksen tapahtumat-sivulle listataan Hämeenlinnan alueella tulevia tapahtumia joiden tiedot noudetaan toimeksiantajan kehittämältä Häme Events-rajapinnalta. Tapahtumia voi suodattaa käyttäen hakusanoja, kategorioita, sijaintia, sekä päivämääriä. Hakutulokset jaetaan kahteen osioon: tänään tapahtuvat, sekä kaikki tapahtumat.

Testitapauksessa Häme Events-rajapinnalta saatu vastaus pitää mockata, eli korvata kovakoodatulla tiedolla, sillä testeissä varsinaisen rajapinnan käyttö tekee niistä hauraita. Rajapinta palauttaa päivittäin muuttuvaa tietoa, ja kutsussa saattaa kestää kauan tai se voi

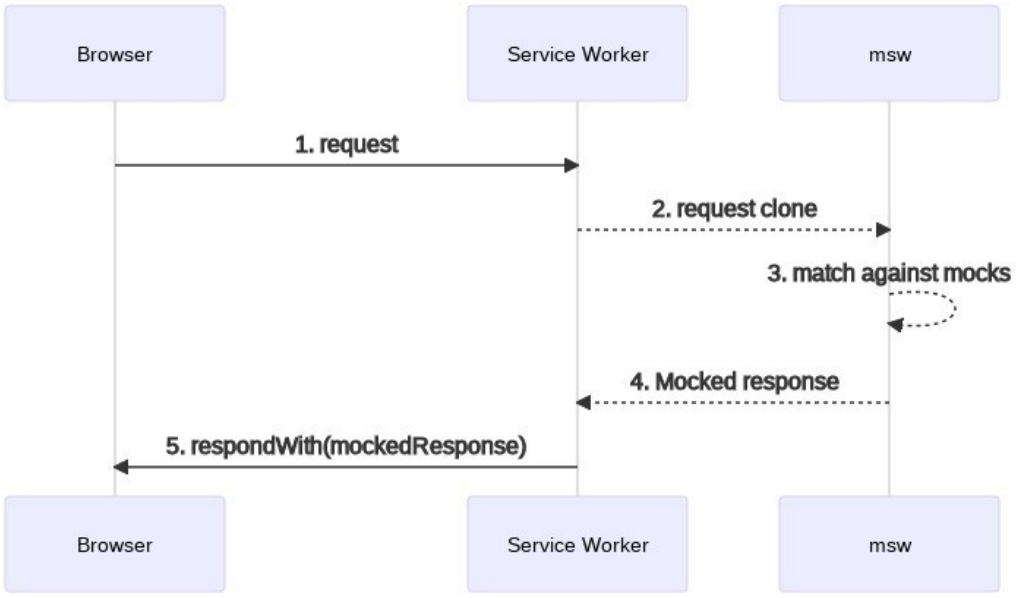
epäonnistua. Websovelluksen PWA-ominaisuuksiin kuuluu offline-toiminnallisuus, jossa Häme Events-rajapinnalta haettu data tallennetaan selaimen rajapinnan tietokantaan nimeltä indexedDB. Websovellus käyttää tietokannan kanssa Dexie-wrapperia, joka yksinkertaistaa tietokannalle tehtyjä kutsuja. Testitapauksessa pitäisi varmistaa tietokannan käytön toimivuus tapauksessa, jossa käyttäjän verkkoyhteys yllättäen katkeaa tai on erittäin heikko, jolloin oikealle palvelimelle tehty kutsu epäonnistuisi ja tietokantaan varmennuksena tallennettu data esitettäisiin käyttäjälle. Tietokannan toiminnallisuuden varmistamisen osalta testitapaus jäi kesken ajan puutteen vuoksi. Testitapauksessa kuitenkin käydään läpi vaiheet, joilla rajapinnalle lähetetyt kutsut saadaan kaapattua ja lähetettyä websovellukselle mockattu vastaus, eli kovakoodattua dataa, jota testeissä voidaan varmentaa.

Testeissä käytetään Jestiä ja React Testing Librarya testaustyökaluina. Rajapinnalle lähetettyjen palvelinkutsujen mockaukseen käytetään msw-kirjastoa (Mock Service Worker), joka elää websovelluksesta erillään omana prosessina ja kaappaa komponentissa tehdyt palvelinkutsut palauttaen kovakoodattua dataa. Kuvassa 22 havainnollistetaan msw:n toimintaperiaate, mutta tässä testitapauksessa selaimen (browser) sijasta testi ajetaan Node.js:ssä. Msw-kirjasto asennetaan projektin kehittäjäriippuvuuksiin komennolla 6.

Komento 6. msw-kirjaston asennuskomento

```
npm install --save-dev msw
```

Kuva 22. MSW kaappaa palvelinkutsun ja palauttaa kovakoodattua dataa (Mock Service Worker, n.d.)



Node.js-ajoympäristö ei itsessään sisällä palvelinkutsuihin käytettyä selaimissa natiivisti esiintyvää fetch-rajapintaa, joten sen polyfill täytyy lisätä projektin kehittäjäriippuvuuksiin komennolla 7. Fetchin polyfill lisätään Noden käytettäväksi konfiguroimalla se Jestin asetuksissa. Jestin asetuksia voi hallita useassa eri tiedostossa, tässä tapauksessa whatwg-fetch -kirjasto tuodaan *require*-komennolla projektin juureen luotuun **jest.setup.js** -tiedostoon (Ohjelmakoodi 18). Jestille täytyy myös kertoa mistä polusta kyseinen tiedosto löytyy, tämä tehdään projektin juuressa olevassa **jest.config.js** -tiedostossa (Ohjelmakoodi 19). *SetupFilesAfterEnv* sisältää listan poluista tiedostoihin, joita käytetään Jestin konfiguroimiseen.

Komento 7. Fetch API:n polyfill Node.js:lle

```
npm install --save-dev whatwg-fetch
```

Ohjelmakoodi 18. Jest.setup.js

```
require('whatwg-fetch');
```

Ohjelmakoodi 19. Jest.config.js

```
module.exports = {
  setupFilesAfterEnv: ['<rootDir>/jest.setup.js'],
};
```

Ohjelmakoodissa 20. testitiedostoon tuodaan msw-kirjastosta *rest*-rajapinnalle lähetettyä kutsua mockaava funktio, sekä msw/node-kirjastosta *setupServer*-funktio jossa rest-rajapinnalle tehdyt kutsut määritellään. Rest-rajapinnalle lähetettävä GET-kutsu määritellään rest.get -funktiolla, jolle syötetään rajapinnan URL-osoite ilman hakuparametreja, sekä funktio jossa kutsujen handlerit määritellään. Funktiolla on kolme parametria: req (request), res (response), sekä ctx (context). Funktio palauttaa vastauksen response-parametria käyttäen, jolle syötetään argumentteina *ctx.status(200)*, eli HTTP-tilakoodi 200 tarkoittaen että pyyntö on vastaanotettu, ymmärretty, ja käsitelty, sekä *ctx.json*-funktio jolle syötetään palautettava vastaus. Vastaus on tässä tapauksessa erillisestä tiedostosta tuotu objekti joka sisältää yhden tapahtuman tiedot.

Ohjelmakoodi 20. Testitapaus: Tapahtumat-moduuli, msw-serverin alustus

```
import { rest } from 'msw';
import { setupServer } from 'msw/node';
import { mockResponse } from '../../__mocks__/mockResponse';

const server = setupServer(
  rest.get('https://api.hameevents.fi/v1/event/', (req, res, ctx) => {
    return res(ctx.status(200), ctx.json(mockResponse));
  })
);

beforeAll(() => server.listen());
afterEach(() => server.resetHandlers());
afterAll(() => server.close());
```

Testejä ajettaessa msw-kirjaston luoma serveri pitää käynnistää käyttäen *listen*-komentoa. Testien jälkeen serveri pitää pysäyttää *close*-komennolla. Jotta testitapaukset eivät vaikuta toisiinsa kannattaa jokaisen testin jälkeen serverin handlereiden tila resetoida käyttäen *resetHandlers*-komentoa. Nämä komennot kannattaa siis suorittaa Jest-hookeissa, jotta niitä ei tarvitse kirjoittaa jokaiseen yksittäiseen testitapaukseen. Msw-kirjaston virallisessa dokumentaatiossa suositellaan ohjelmakoodissa 20 esitettyjen komentojen siirtämistä täysin testitiedostojen ulkopuolelle, jolloin kaikki testitiedostot voivat käyttää msw:n mockaamia palvelinkutsuja. Tehdään tämä muutos projektiin.

Luon __mocks__ -kansioon kaksi uutta tiedostoa: server.js (Ohjelmakoodi 21), ja handlers.js (Ohjelmakoodi 22). Handlers listaa kaikki mockatun palvelimen käyttämät handlerit, eli palvelinkutsut joita msw tarkkailee ja kaappaa. Tässä testitapauksessa mockaamme pelkästään GET-kutsun. Server tuo handlers-tiedoston sisällön spread-operaattoria (kolme pistettä) käyttäen setupServer-funktiolle argumentteina.

Ohjelmakoodi 21. server.js

```
import { setupServer } from 'msw/node';
import { handlers } from './handlers';

export const server = setupServer(...handlers);
```

Ohjelmakoodi 22. handlers.js

```
import { rest } from 'msw';
import { mockResponse } from './mockResponse';

export const handlers = [
  rest.get('https://api.hameevents.fi/v1/event/', (req, res, ctx) => {
    return res(ctx.status(200), ctx.json(mockResponse));
  }),
];
```

Ohjelmakoodissa 23 msw:n serveri tuodaan jest.setup.js -tiedostoon import-komennolla. Serverin käynnistys, handlereiden resetointi, ja sulkeminen voidaan hoitaa Jestin asetuksissa testitiedostoista erillään, joka tekee testeistä helpommin luettavia ja ymmärrettäviä. Asetuksiin on myös lisätty Jestin expect-funktion laajennus *@testing-library/jest-dom* -kirjastosta, joka sisältää esimerkiksi ohjelmakoodissa 24 käytetyn *toBeInTheDocument*-matcher-funktion. Tällöin jest-domia ei tarvitse tuoda yksittäisiin testitiedostoihin.

Ohjelmakoodi 23. jest.setup.js

```
// Fetch polyfill for Node.js
require('whatwg-fetch');

// JestDOM, extends Jest with custom matcher functions
const jestDom = require('@testing-library/jest-dom');
expect.extend(jestDom);

// Mock Service Worker
import { server } from './__mocks__/server.js';
beforeAll(() => server.listen());
afterEach(() => server.resetHandlers());
afterAll(() => server.close());
```

Ohjelmakoodin 24 ensimmäisessä testissä todennetaan mockatulta palvelimelta saatu vastaus varmistamalla lataus animaation poistuminen, sekä tapahtumien kategorisointiin renderöityjen välilehtien ilmestyminen DOM-puuhun. Testissä käytetään tällä kertaa asynkronista *findByText*-hakufunktiota elementtien hakuun jotka ilmestyvät vasta kun palvelimelta saatu vastaus on käsitelty. Toisena hakufunktiona käytetty *queryByTestId* palvelee kahta eri tarkoitusta. Latausanimaatio ei itsessään sisällä mitään tekstiä mihin

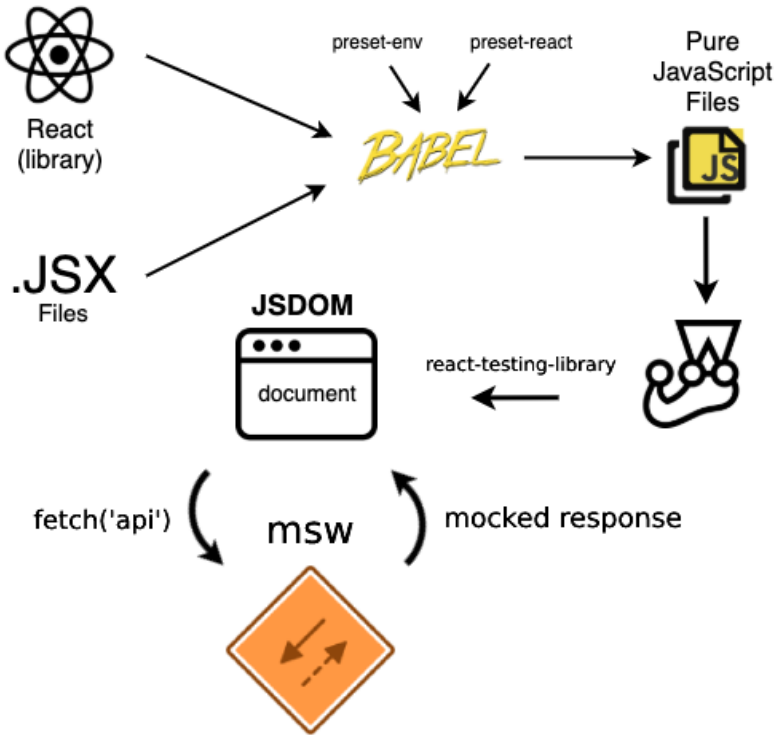
tarttua, joten lähdekoodiin pitää lisätä joku muu animaatiota suorittavaan komponenttiin viittaava parametri. Yksi tapa erottaa hankalasti valittava elementti muista on käyttää uniikkia testi-id:tä. Lisäsin websovelluksen lähdekoodiin animaatio-komponentin sisältävälle div-tagille parametrin *data-testid="loadingIcon"* jolla elementti saadaan valittua.

Toinen syy käyttää juuri *queryByTestId*-hakufunktiota on sen palauttama arvo. Query-alkuiset hakufunktiot palauttavat null-arvon jos kyseisillä hakuparametreilla ei löydy mitään, kun taas *findBy/getBy*-hakufunktiot antavat virheilmoituksen. Null kuuluu JavaScriptin niin sanottuihin falsy-arvoihin, joten animaation näkyvyyttä sovelluksen eri vaiheissa voidaan todentaa *toBeTruthy/toBeFalsy*-matchereilla. Ohjelmakoodin 24 toisessa testissä varmennetaan, että mockattu msw:n lähettämä tapahtuma ilmestyy DOM-puuhun. Kuvassa 23 havainnollistetaan msw:tä käyttävän testitapauksen polku, jossa sovellus lähettää fetchillä palvelinkutsun ja saa vastauksena mockattua dataa. Suoritan testit onnistuneesti (Kuva 24), ja kuten tähänkin asti on ollut tapana, rikon toisen testin varmistaakseni sen toimivuuden (Kuva 25).

Ohjelmakoodi 24. Testitapaus: Tapahtumat-moduuli

```
import React from 'react';
import { render } from '@testing-library/react';
import Tapahtumat_module from '../modules/Tapahtumat_module';

test('Waits for API response, stops loading animation', async () => {
  const { findByText, queryByTestId } = render(<Tapahtumat_module />);

  // Assert the app is loading at the start
  expect(queryByTestId('loadingIcon')).toBeTruthy();

  // Wait for elements to appear in the DOM, assert elements exist
  expect(await findByText(/tänään tapahtuu/i)).toBeInTheDocument();
  expect(await findByText(/kaikki tapahtumat/i)).toBeInTheDocument();

  // Assert loading has stopped
  expect(queryByTestId('loadingIcon')).toBeFalsy();
});

test('Renders mocked event', async () => {
  const { findByText } = render(<Tapahtumat_module />);
  expect(await findByText(/mockattu tapahtuma/i)).toBeInTheDocument();
});
```

Kuva 23. Testin polku: msw



Kuva 24. Testitulos: Tapahtumat-testi onnistuu

```
PASS src/Test/Tapahtumat_module.test.js
  √ Waits for API response, stops loading animation (275 ms)
  √ Renders mocked event (106 ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        4.567 s, estimated 5 s
Ran all test suites matching /Tapahtumat_module.test.js/i.
```

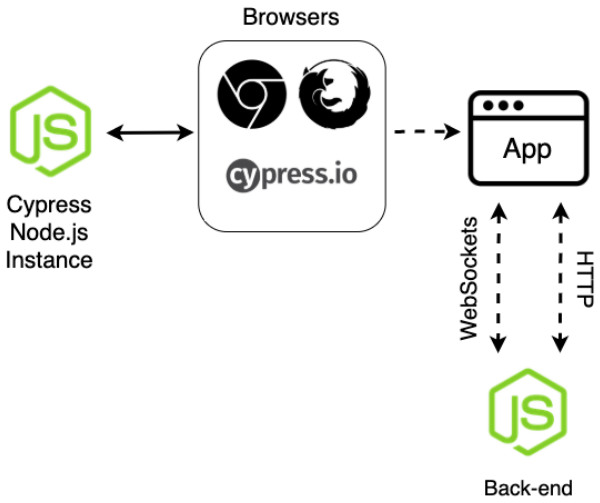
Kuva 25. Testitulos: Tapahtumat-testi rikotaan, testi epäonnistuu

```
FAIL src/Test/Tapahtumat_module.test.js
  √ Waits for API response, stops loading animation (265 ms)
  × Renders mocked event (110 ms)

  ● Renders mocked event

    expect(element).not.toBeInTheDocument()

    expected document not to contain element, found <span class="card-title">Mockattu tapahtuma</span> instead

      19 | test('Renders mocked event', async () => {
      20 |   const { findByText } = render(<Tapahtumat_module />);
    > 21 |   expect(await findByText(/mockattu tapahtuma/i)).not.toBeInTheDocument();
         |                                                    ^
      22 | });
      23 |

      at Object.<anonymous> (src/Test/Tapahtumat_module.test.js:21:55)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 passed, 2 total
Snapshots:   0 total
Time:        4.548 s
Ran all test suites matching /Tapahtumat_module.test.js/i.
```

5.4 End-to-end testi: Luontopolut-moduuli

Kaupunki taskussa -websovelluksen luontopolut-sivulle upotetaan Hämeenlinnan kaupungin verkkosivuilta Kanta-Hämeen kartta osoitteesta https://kartta.hameenlinna.fi/ims/. Kartta upotetaan sivulle iframe-elementtinä, jonka näkyvyyttä varmentavan automatisoidun testin toimeksiantaja on opinnäytetyöntekijää pyytänyt kirjoittamaan. Kartan näkyvyyttä voidaan testata käyttämällä Cypress-testauskehystä. Cypress suorittaa testin varsinaisessa selaimessa graafisen käyttöliittymän kautta (Kuva 26), lähettäen käskyjä selaimen rajapinnalle, joten se simuloi oikean käyttäjän kokemusta tarkemmin kuin aiemmat yksikkö- tai integraatiotestit. Cypressillä toteutetut testit voidaan luokitella käyttöliittymä-pohjaisiksi end-to-end testeiksi.

Kuva 26. Testin polku: Cypress (da Costa, 2021, UI-based end-to-end testing-luku, Cypress-alaluku)



Cypress asennetaan projektikohtaisesti kehittäjäriippuvuutena komennolla 8. Cypressin käyttöliittymä (Kuva 27) käynnistetään komennolla 9. Käyttöliittymää avatessa ensimmäistä kertaa projektin juureen luodaan cypress-kansio, jossa integration > examples-alikansiosta löytyy loistavia esimerkkejä erilaisista testitapauksista. Omat testitiedostot luodaan myös integration-kansioon jotta ne näkyvät käyttöliittymässä.

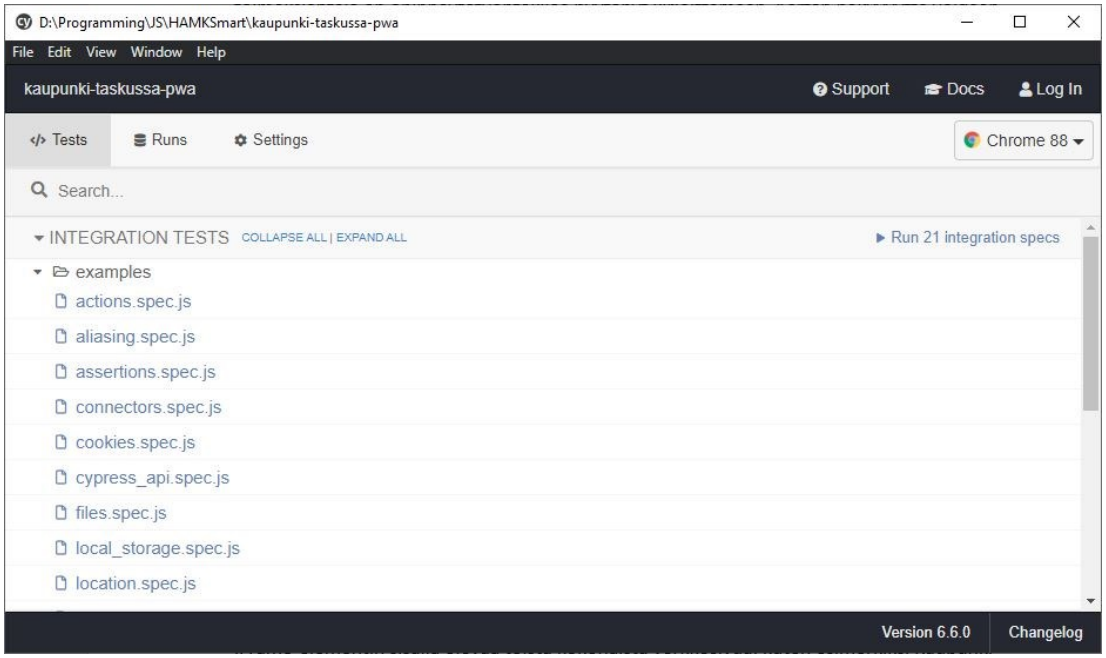
Komento 8. Cypress-testauskehyksen asennus

```
npm install --save-dev cypress
```

Komento 9. Cypress-testauskehyksen käynnistys

```
npx cypress open
```

Kuva 27. Cypress-testiajurin käyttöliittymä



Cypressin käyttöliittymässä voidaan oikeassa yläkulmassa valita selain jolla testit suoritetaan. Testejä voidaan suorittaa yksittäin klikkaamalla listasta tiedostonimiä, tai yhdessä painamalla ”Run X integration specs”. Cypress avaa selainikkunan ja suorittaa testit automaattisesti. Selainikkunan voi jättää auki odottamaan muutoksia, mutta muutos yhdessä tiedostossa aloittaa kaikki testit alusta. Tästä syystä suosittelen yksittäisten testitiedostojen ajoa kerrallaan, sillä toisin kuin Jest joka pystyy suorittamaan useita testejä rinnakkain vähentäen testeihin kuluvaa aikaa, Cypress suorittaa ne yksi kerrallaan johon saattaa kulua useita minuutteja. Cypressin testit voidaan ajaa rinnakkain esimerkiksi CI/CD-järjestelmässä käyttäen useita virtuaalikoneita. Lisätietoja aiheesta Cypressin sivuilla.

Inline Frame element, eli iframe tarkoittaa toisen verkkosivun sisällön esittämistä sisäkkäisesti alkuperäisellä sivulla. Käytännössä tämä tarkoittaa iframe-elementin sisällä olevaa toista kokonaista verkkosivua, kuten esimerkiksi Kaupunki taskussa -websovelluksen Luontopolut-sivulla käytettyä Kanta-Hämeen karttaa.

Selaimet noudattavat saman alkuperän käytäntöä (Mozilla, n.d.-d), turvallisuusmekanismia, joka rajoittaa eri lähteistä ladattujen kohteiden välistä kanssakäymistä. Testin kannalta tämä tarkoittaa että selaimet estävät pääsyn ulkoisesta lähteestä upotetun iframe-elementin sisäisiin DOM-nodeihin testiä ajettaessa. Suojauksen voi ohittaa chromium-pohjaisissa selaimissa lisäämällä projektin juuressa olevaan **cypress.json** -asetustiedostoon

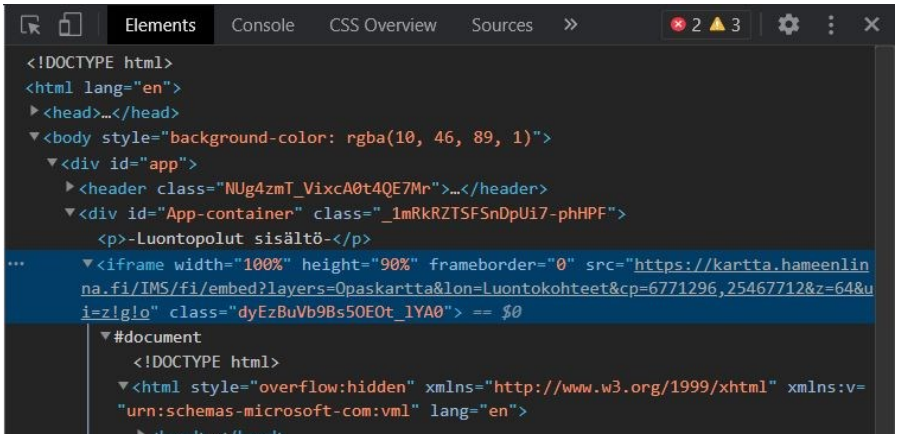
ohjelmakoodissa 25 esitetyn selaimen suojausasetuksen ohittamiskäskyn. Suojauksen ohittamalla päästään käsiksi ulkopuolisista lähteistä upotettuihin iframe-komponentteihin. (Cypress, n.d.-b) Tässä tapauksessa testi ajetaan kehittäjän paikallisessa verkossa, jolloin Hämeenlinnan kaupungin osoitteesta tuotu iframen käsittely estetään, ellei chrome-selaimen turvallisuutta olla ohitettu.

Ohjelmakoodi 25. Cypress asetukset, iFrame testi

```
{
  "chromeWebSecurity": false
}
```

Ohjelmakoodissa 26 testi alkaa *cy.visit*-komennolla, jolle annetaan argumenttina websovelluksen osoite. Tässä tapauksessa osoite viittaa kehittäjän paikallisessa verkossa käynnissä olevaan serverin osoiteeseen Luontopolut-sivulle. Kartan näkyvyys voidaan todentaa suorittamalla testi jossa iframe-elementin sisältä etsitään painike jota painetaan ja todennetaan sivulla tapahtunut muutos. Jotta elementin sisältä voidaan etsiä jotain, pitää iframe-elementtiin päästä käsiksi. Elementti voidaan valita käyttämällä id- tai class-arvoa, jotka saadaan selville joko lähdekoodista tai avaamalla selaimen kehittäjätyökalut F12-näppäimestä, aktivoimalla Inspect-osoitin kehittäjätyökalupaneelin vasemmasta ylälaidasta, ja klikkaamalla iframe-elementtiä. Tällöin Elements-välisivun sisältö kohdistuu iFrame-elementin sisään. Kuvassa 28 iframen class-arvolla saamme elementin valittua testissä. Toki helpompaa olisi jos elementille oltaisiin alun perin asetettu uniikki id-arvo jolla elementti voitaisiin valita.

Kuva 28. Chrome DevTools, iFrame-elementin luokka



Iframe-elementti valitaan käyttämällä cy.get-funktiota, syöttäen sille elementin class- tai id-arvo. Valittu kohde viittaa nyt itse iframe-elementtiin, mutta kuten kuvassa 28 näemme, elementti sisältää erillisen HTML-dokumentin johon testissä pitää päästä käsiksi. Kaikki Cypressin komennot ovat asynkronisia, tarkoittaen että kaikista komennoista palautuu promise, joka joko ratkeaa palautuvalla arvolla, tai hylätään asetuksissa määritetyn ajan kuluttua. Cypressin get-funktion palauttaa dokumentin, joten funktion perään voidaan linkittää *then*-funktiokutsu, jossa dokumenttia voidaan käsitellä. Dokumentin sisältöön päästään käsiksi käyttämällä *$document.contains().find('body')* -komentoa, joka palauttaa iframen sisäisen HTML-dokumentin bodyn. Sisältö tallennetaan muuttujaan contents.

Jotta body-elementin sisäisten painikkeita voidaan painaa, pitää käyttää Cypressin *wrap()*-komentoa, antaen sille argumenttina contents-muuttujan. Nyt dokumentin sisällöstä voidaan etsiä painikkeita ja klikata niitä käyttämällä *find*- ja *click*-komentoja. Ohjelmakoodissa 26 todennan kartan näkyvyyden etsimällä kartasta elementin id:llä ”scaletext”, joka kertoo kartan skaalan. Assertoin sen sisältöä *should*-funktiolla, joka ottaa vastaan kaksi argumenttia: chainer-käskyn, eli tavan jolla arvoa verrataan, sekä odotetun arvon. Tässä tapauksessa elementin pitäisi sisältää tekstiä jossa lukee ”5 km”.

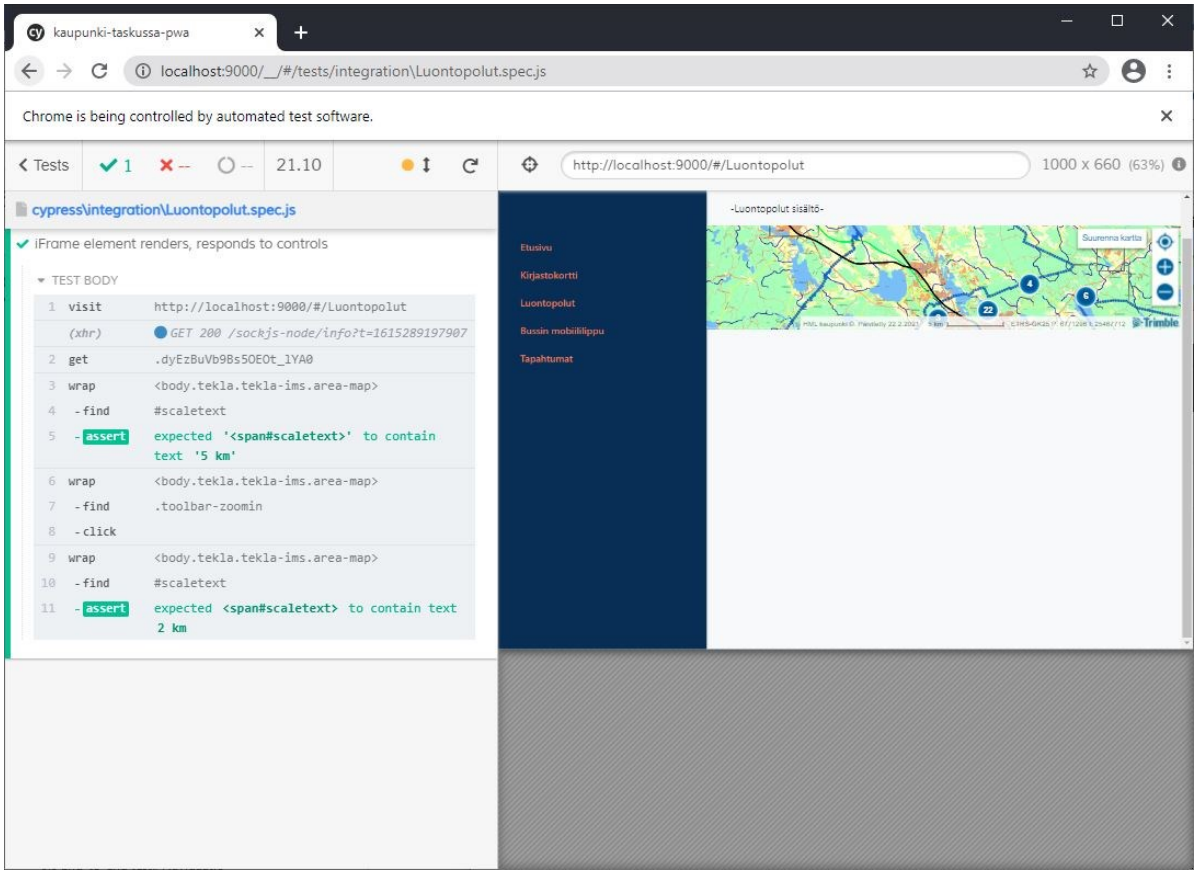
Varmennan kartan toimivuuden etsimällä toisessa *cy.wrap*-komennossa painikkeen, josta karttaa voi zoomata sisään, ja klikkaamalla sitä. Varmennan muutoksen tapahtuneen kolmannessa *cy.wrap*-komennossa tarkistamalla scaletext-elementin sisällön uudestaan. Tällä kertaa tekstin pitäisi olla ”2 km”. Jos testit onnistuvat, voidaan iframe-kartan todeta näkyvän ja toimivan kuten pitääkin. Toiminnallisuutta voidaan tässä tapauksessa testata vain chromium-pohjaisilla selaimilla, kuten Chrome ja Edge, sillä Firefoxin ja Safarin turvallisuutta ei voida muuttaa samalla tapaa ja iframen sisältöön pääsy blokataan.

Ajan testin avaamalla Cypressin käyttöliittymän ja klikkaamalla testin nimeä valikosta. Cypress avaa selaimen ja ajaa testin. Kuvassa 29 näkyy Cypressin hallitsema selainikkuna, joka on suorittanut testin onnistuneesti. Kehittäjä voi tarkastella sovelluksen tilaa testiä ajettaessa valitsemalla vasemmasta valikosta testin eri vaiheita. Varmentaakseni testin toimivuuden rikon sen tahallaan painamalla tällä kertaa zoomout-painiketta, jolloin selainikkunaan ilmestyy kuvassa 30 esitetty virheviesti.

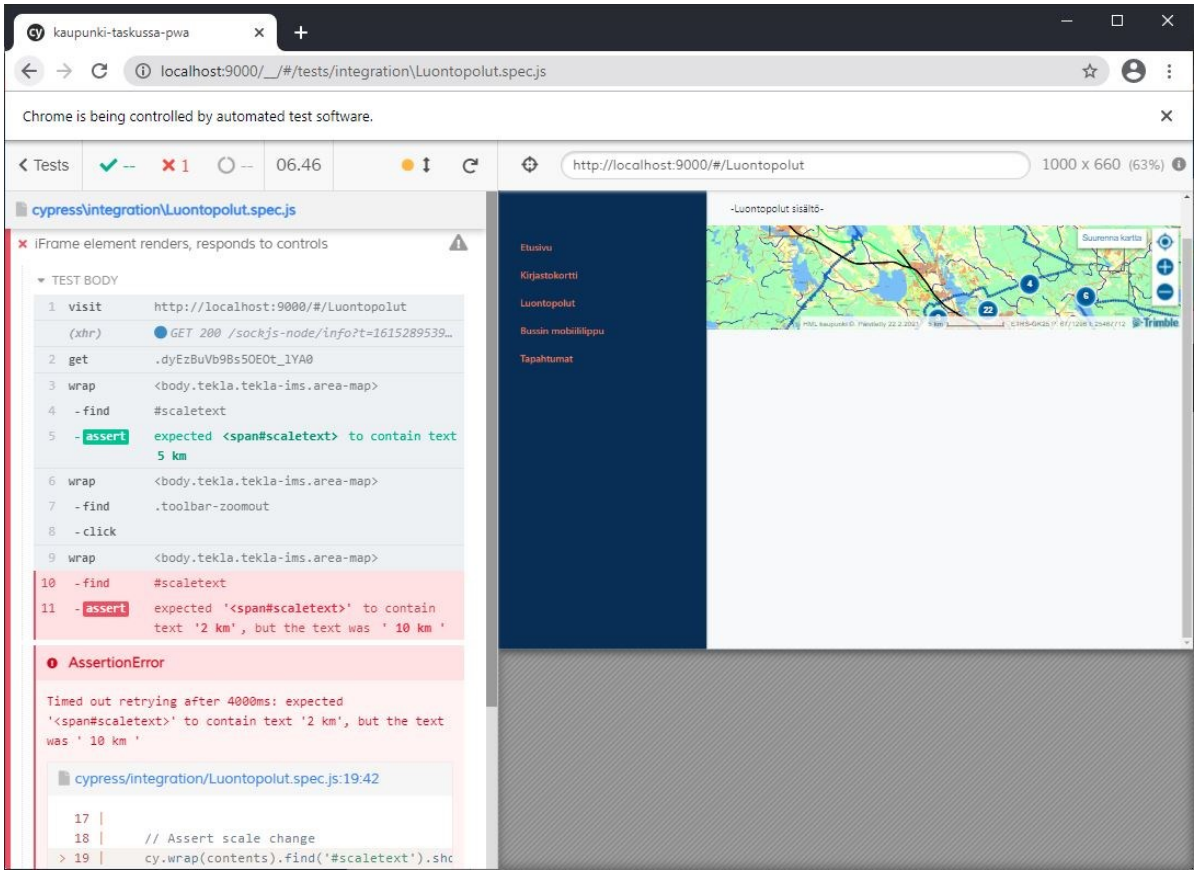
Ohjelmakoodi 26. Testitapaus: Luontopolku

```
// IntelliSense for cypress API
/// <reference types="Cypress" />

it('iFrame element renders, responds to controls', () => {
  // Arrange
  const iFrameClass = '.dyEzBuVb9Bs5OEOt_lYA0';
  cy.visit('http://localhost:9000/#/Luontopolut');

  cy.get(iFrameClass).then(($document) => {
    const contents = $document.contents().find('body');

    // Assert default scale at 5km
    cy.wrap(contents)
      .find('#scaletext')
      .should('contain.text', '5 km');

    // Act, find a button for zoom in and click it
    cy.wrap(contents)
      .find('.toolbar-zoomin')
      .click();

    // Assert scale change
    cy.wrap(contents)
      .find('#scaletext')
      .should('contain.text', '2 km');
  });
});
```

Kuva 29. Testitulos: Luontopolku-testi onnistuu



Kuva 30. Testitulos: Luontopolut-testi rikotaan tahallaan, testi epäonnistuu



5.5 End-to-end testi: Navigaatio

Neljäs ja viimeinen toimeksiantajan määrittelemä testitapaus varmentaa websovelluksen navigaation toimivuuden. Tämä testi on aiempiin verrattuna paljon yksinkertaisempi. Testissä käytetään Cypress-testauskehystä, sillä helpoin tapa varmentaa sivulla tapahtunut sisällön sekä URL-osoitteen muutos on suorittaa testi selaimessa.

Ohjelmakoodissa 27 beforeEach-hookin sisään kirjoitettu cy.visit -komento ohjaa selaimen kehittäjän paikallisen verkon portille 9000 jossa websovellus on käynnissä. Testitapaukset noudattavat kaavaa, jossa navigaatiopalkista etsitään painike kyseiselle sivulle, painiketta klikataan, ja sivulla tapahtunut muutos varmennetaan kahdessa osassa. Selaimen URL-osoitteen pitäisi sisältää kyseisen sivun nimi, johon käytetään Cypressin rajapinnan *url*- ja *should*-komentoja, Should-komennolle syötetään ensimmäisenä argumenttina string-muodossa vertaus *'include'*, ja toisena argumenttina polku, joka URL-osoitteessa pitäisi olla. Toisessa varmennuksessa sivulla tapahtunut muutos varmennetaan etsimällä sivulta ainoastaan sillä esiintyvää sisältöä, ja varmentamalla se jälleen should-komennolla antaen sille argumentiksi *'be.visible'*, eli elementin pitäisi olla nähtävissä. Cypressin komennot ovat luonnostaan asynkronisia, joten testitapauksissa ei ole tarvetta kirjoittaa muutoksia odottavia komentoja. Cypressin vakioasetus komentojen suorittamiselle on 4 sekuntia jonka jälkeen testitapaus heittää virheen jos komentoa ei onnistuneesti pystytty suorittamaan.

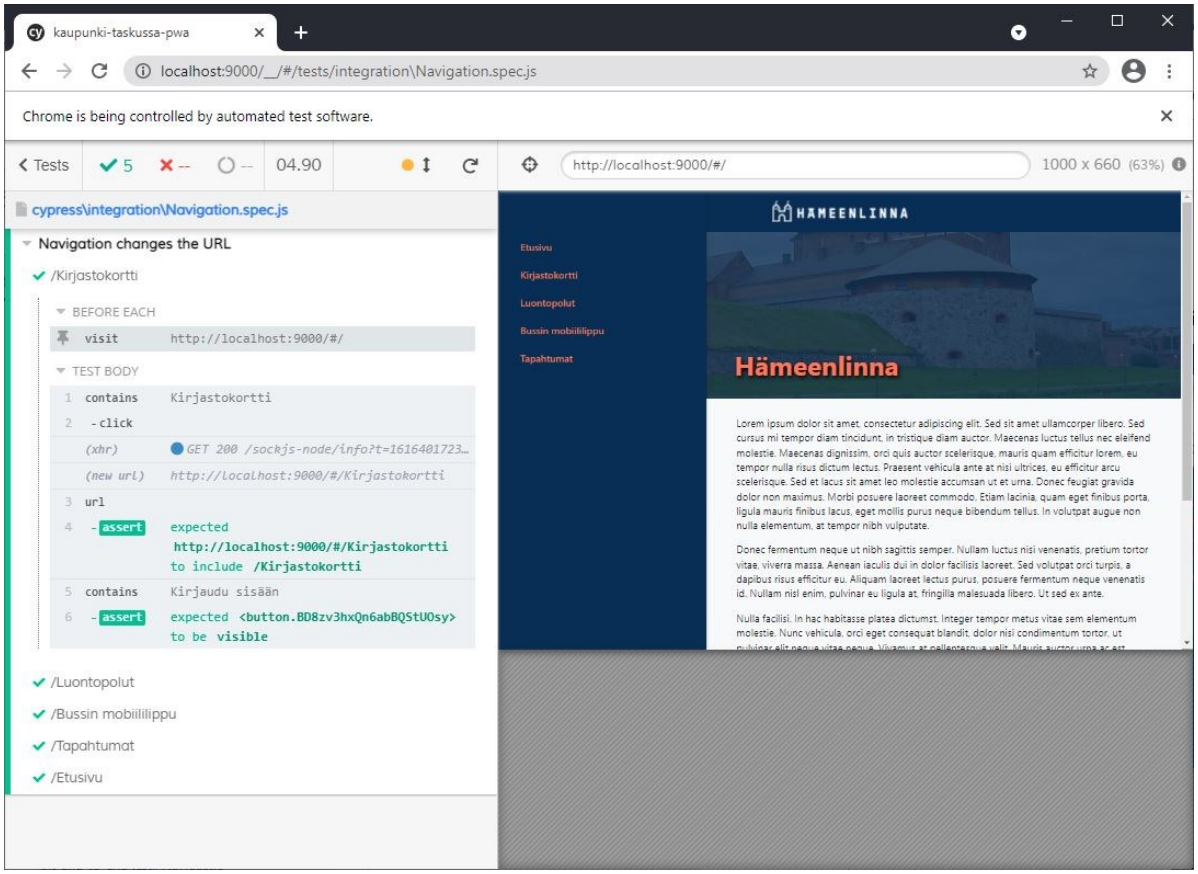
Kuvassa 31 testitapaus suoritetaan onnistuneesti. Cypress avaa chrome-selaimen, navigoi sivulle, ja suorittaa testit. Varmistaakseni testien toimivuuden rikon jälleen yhden testitapauksista antaen ensimmäiselle URL-osoitetta varmistavalle komennolle väärän syötteen. Kuvassa 32 Cypress ilmoittaa epäonnistuneesta testitapauksesta virheviestillä.

Navigaation hyvistä käytännöistä mainittakoon websovellusten hyvin yleinen trendi. Sovelluksen tai yrityksen logon pitäisi aina olla nähtävissä sivun yläosassa, ja logon kuuluisi sisältää linkki websovelluksen pääsivulle. Kaupunki taskussa -websovelluksen tapauksessa tämä toiminnallisuus on joko tarkoituksella jätetty pois, tai epähuomiossa unohdettu. Sivun header-elementistä löytyy Hämeenlinnan kaupungin logo, mutta se ei toimi linkkinä websovelluksen pääsivulle. Tästä voisi kirjoittaa testitapauksen toiminnallisuuden varmentamiseksi, mutta jätän sen toimeksiantajan kehitystiimin päätettäväksi.

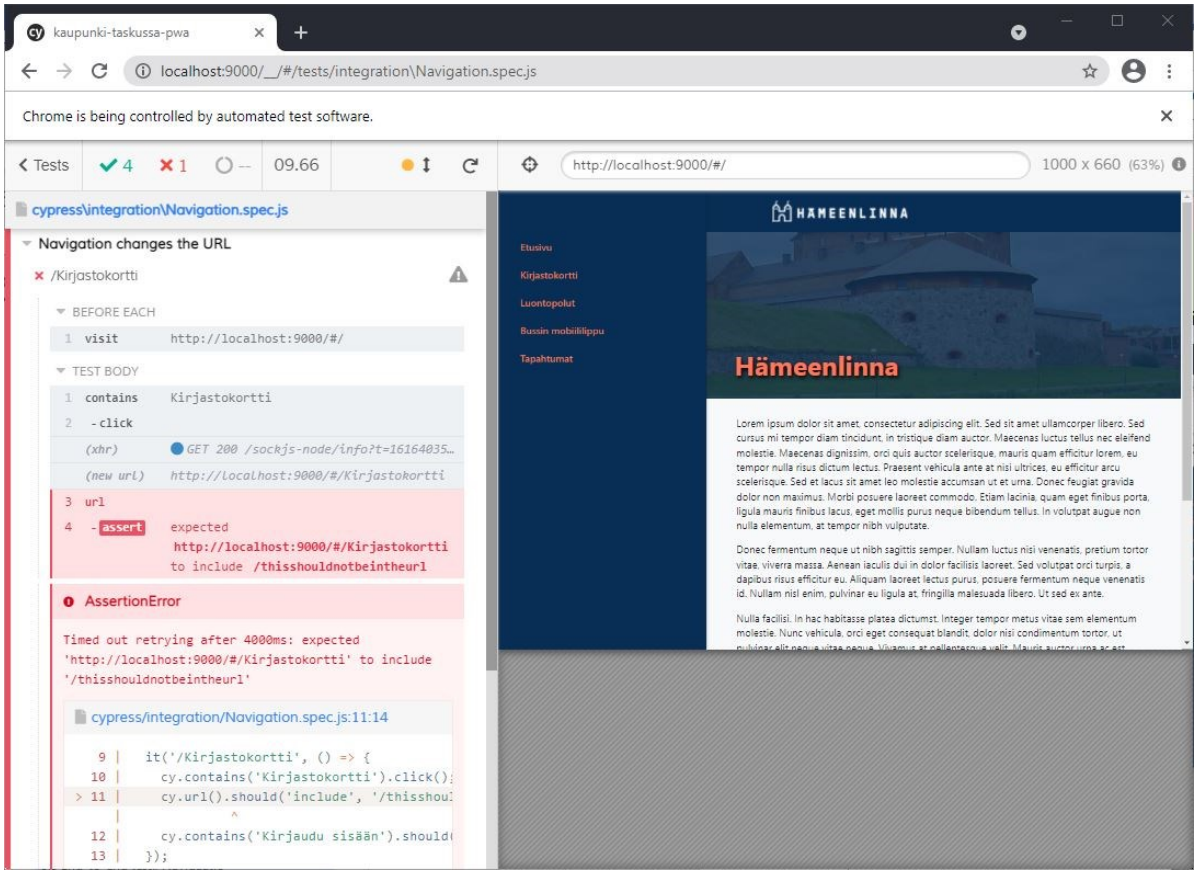
Ohjelmakoodi 27. Testitapaus: Navigaatio

```
// IntelliSense for cypress API
/// <reference types="Cypress" />

describe('Navigation changes the URL', () => {
  beforeEach(() => {
    cy.visit('http://localhost:9000/#/');
  });

  it('/Kirjastokortti', () => {
    cy.contains('Kirjastokortti').click();
    cy.url().should('include', '/Kirjastokortti');
    cy.contains('Kirjaudu sisään').should('be.visible');
  });

  it('/Luontopolut', () => {
    cy.contains('Luontopolut').click();
    cy.url().should('include', '/Luontopolut');
    cy.contains('Luontopolut sisältö').should('be.visible');
  });

  it('/Bussin mobiililippu', () => {
    cy.contains('Bussin mobiililippu').click();
    cy.url().should('include', 'https://waltti.fi/kaupungit/hameenlinna/');
    cy.contains('Lataa kortti').should('be.visible');
  });

  it('/Tapahtumat', () => {
    cy.contains('Tapahtumat').click();
    cy.url().should('include', '/Tapahtumat');
    cy.contains('Hae tapahtumia').should('be.visible');
  });

  it('/Etusivu', () => {
    cy.contains('Etusivu').click();
    cy.contains('Asenna Kaupunki Taskussa').should('be.visible');
  });
});
```

Kuva 31. Testitulos: Navigaatio-testi onnistuu



Kuva 32. Testitulos: Navigaatio-testi rikotaan tahallaan, testi epäonnistuu



6 Johtopäätökset ja pohdinta

Opinnäytetyön tutkimuskysymykset olivat mitä sovellustestauksella tarkoitetaan, millä työkaluilla websovelluksia kannattaa testata, sekä miten React-komponentteja kannattaa testata. Työn teoriaosuudessa selvitettiin sovellustestausta yleisellä tasolla, sekä mitä webtestauksessa käytetyt yksikkö, integraatio sekä end-to-end testit tarkoittavat, ja mitä sovelluksen osia kyseiset testit kattavat. Käytännön osuudessa suoritettiin testejä omille yksinkertaisille sovelluksille sekä toimeksiantajan kehittämälle websovellukselle. Työssä käytettiin React-sovelluksille suositeltuja Jest-, React Testing Library-, sekä Cypress-testaustyökaluja. Työn käytännön osuudessa selvitettiin miten React-komponentteja kannattaa testata käyttäjän näkökulmasta renderöimällä sovellus joko selaimissa Cypress-testeissä, tai virtuaalisina DOM-nodeina käyttäen Jestiä ja React Testing Librarya.

Toimeksiantajan määritellyissä testitapauksissa ratkaistiin työelämälähtöisesti varsinaisen websovelluksen testaukseen liittyviä ongelmia. Neljästä testitapauksesta kolme saatiin opinnäytetyön aikana suoritettua suunnitellusti. Ajan puutteen, sekä integraatiotestin monimutkaisuuden vuoksi yksi testitapaus Kaupunki taskussa-websovelluksen tapahtumat-moduulia koskevaan selaimen sisäisen indexedDB-tietokannan toimintaan jäi tekemättä. Testitapauksessa ulkoiselle rajapinnalle lähetetyt kutsut kaapattiin msw-kirjastoa käyttäen, ja sovellukselle palautettiin kovakoodattua tietoa. Tietokannan toimivuuden osalta testi jäi kuitenkin tekemättä. Toimeksiantajan kehittäjätiimi voi silti hyödyntää opinnäytetyössä aikaansaatuja tuloksia testitapauksia kirjoittaessaan.

Toimeksiantaja sai työssä tehdyllä selvityksellä ja testitapauksilla varmuutta tuotteensa toimivuudesta, sekä käytännön esimerkkejä siitä miten Reactilla kehitetylle tuotteelle luodaan testejä. Toimeksiantaja oli tyytyväinen opinnäytetyössä tuotettuihin tuloksiin. Opinnäytetyötä ja siitä seuranneesta kehitystyöstä sovelluksen parissa voidaan hyödyntää jatkossa myös muiden projektien parissa.

7 Yhteenveto

React-sovellusten testausta tulisi suorittaa ulkopuolisen näkökulmasta, välttäen komponenttien sisäiseen toimintaan liittyviä yksityiskohtia. Komponenttien tilan tai sisäisten funktioiden testaus johtaa hauraisiin testitapauksiin joita joudutaan päivittämään komponentteja refaktoroidessa, lisäten työtaakkaa. JSDOM mahdollistaa komponenttien yksittäisen tai yhtenäisen toiminnallisuuden varmentamisen tavalla, jossa ainoastaan tuloksella on merkitystä, ei sillä miten komponentti päätyi tuottamaan kyseisen tuloksen.

React Testing Library on erinomainen testaustyökalu React-komponenttien testaukseen, sillä se tarjoaa kattavat työkalut komponenttien renderöintiin, manipulointiin, sekä toiminnallisuuden varmentamiseen. Jest puolestaan täydentää React Testing Libraryn toiminnallisuutta tarjoamalla alustan jolla testejä suoritetaan, kattavan rajapinnan testien kirjoittamiseen, sekä ulkoisten osapuolten kuten tietokantojen tai rajapintojen mockaukseen tarvittavat työkalut. Cypress mahdollistaa sovelluksen testauksen kokonaisuudessaan käyttöliittymän kautta useilla eri selaimilla, joten sillä luodut testit ovat luotettavampia ja yllättävän helppoja kirjoittaa. Reactin suosio näyttää kasvavan vuosi vuodelta joten testattavien sovellusten määrä vain kasvaa. On hienoa nähdä että myös testaustyökalut ovat edistyneet ja niitä on räätälöity juurikin React-kirjaston tarpeille sopiviksi.

Opinnäytetyön aihe oli mielenkiintoinen sekä työelämälähtöinen, ja koen oppineeni runsaasti ohjelmistokehityksessä suoritettavasta testauksesta. Sovellustestaus on IT-alalla erittäin yleistä, joten koen opinnäytetyön olleen erinomainen mahdollisuus tutustua aiheeseen syvemmin. Työssä kerätyillä taidoilla on varmasti kysyntää, jonka vuoksi alun perin valitsinkin juuri tämän aiheen. Työtä aloittaessani minulla ei ollut käytännön kokemusta JavaScript- tai React.js -projektien testaustyöstä juuri lainkaan. Web-sovellusten ohjelmointi JavaScript-kielellä sekä React-kirjastolla olivat entuudestaan tuttuja sekä kouluprojektien että itsenäisen opiskelun kautta. Opinnäytetyö kehitti taitojani sovelluskehittäjänä sekä lisäsi ammatillista itsevarmuutta.

Lähteet

Aranda, M. (2017, October 28). *What's the difference between JavaScript and ECMAScript?* https://www.freecodecamp.org/news/whats-the-difference-between-javascript-and-ecmascript-cba48c73a2b5/

Babel. (n.d.-a). *Usage Guide · Babel*. Retrieved January 28, 2021, from https://babeljs.io/docs/en/usage

Babel. (n.d.-b). *What is Babel? · Babel*. Retrieved January 26, 2021, from https://babeljs.io/docs/en/index.html

Codecademy. (n.d.). *React: The Virtual DOM | Codecademy*. Retrieved February 8, 2021, from https://www.codecademy.com/articles/react-virtual-dom

Cucciniello, A. (2017). *How is Node.js Changing Web Development? | Packt Hub*. https://hub.packtpub.com/how-nodejs-changing-web-development/

Cypress. (n.d.-a). *JavaScript End to End Testing Framework | cypress.io*. Retrieved March 30, 2021, from https://www.cypress.io/

Cypress. (n.d.-b). *Web Security | Cypress Documentation*. Retrieved March 8, 2021, from https://docs.cypress.io/guides/guides/web-security.html#Limitations

da Costa, L. (2021). *Testing JavaScript Applications* (6th ed.). Manning. https://www.manning.com/books/testing-javascript-applications

Dodds, K. C. (n.d.). *Testing JavaScript with Kent C. Dodds*. Retrieved February 12, 2021, from https://testingjavascript.com/

Dodds, K. C. (2019). *Write tests. Not too many. Mostly integration.* https://kentcdodds.com/blog/write-tests

ECMA International. (n.d.). *Standards Archive - Ecma International*. Retrieved January 27, 2021, from https://www.ecma-international.org/publications-and-standards/standards/

ECMA International. (2021). *ECMAScript® 2021 Language Specification*. https://tc39.es/ecma262/

Facebook. (n.d.-a). *Getting Started – React*. Retrieved January 29, 2021, from https://reactjs.org/docs/getting-started.html

Facebook. (n.d.-b). *Introducing JSX – React*. Retrieved February 8, 2021, from https://reactjs.org/docs/introducing-jsx.html

Facebook. (n.d.-c). *Testing Overview – React*. Retrieved February 10, 2021, from https://reactjs.org/docs/testing.html

Google Play Store. (n.d.). *Hämeenlinna taskussa – Google Play -sovellukset*. Retrieved March 3, 2021, from https://play.google.com/store/apps/details?id=fi.hameenlinna.oca

Hämeenlinnan kaupunki. (2021). *Hämeenlinna taskussa -sovellus - Hämeenlinna*. https://www.hameenlinna.fi/hallinto-ja-talous/tietoa-hameenlinnasta/hameenlinna-taskussa-sovellus/

Haverbeke, M. (2018a). *Asynchronous Programming :: Eloquent JavaScript*. https://eloquentjavascript.net/11_async.html

Haverbeke, M. (2018b). *The Document Object Model :: Eloquent JavaScript*. https://eloquentjavascript.net/14_dom.html

Heller, M. (2020). *What is Node.js? The JavaScript runtime explained | InfoWorld*. https://www.infoworld.com/article/3210589/what-is-nodejs-javascript-runtime-explained.html

Lahti, H. (2019). *Henri Lahti JAVASCRIPT-TESTAUS JavaScript-ohjelmien testaus Opinnäytetyö CENTRIA-AMMATTIKORKEAKOULU Tieto-ja viestintätekniikan koulutusohjelma*.

Jest. (n.d.). *Jest · ? Delightful JavaScript Testing*. Retrieved March 30, 2021, from https://jestjs.io/

Kasurinen, J. P. (2013). *Ohjelmistotestauksen käsikirja*. Docendo.

Kaushik, V. (2020). *The history of JavaScript: A Journey from Netscape to Frameworks*. https://www.techaheadcorp.com/blog/history-of-javascript/

Khorikov, V. (2020). *Making Better Unit Tests: part 1, the AAA pattern - Manning*. Manning. https://freecontent.manning.com/making-better-unit-tests-part-1-the-aaa-pattern/

Kononenko, K. (2018). *Traversing the DOM Is Just Like Creating Your Personal Schedule (Visual Explanation) – CodeAnalogies Blog*. https://blog.codeanalogies.com/2018/01/06/traversing-the-dom-visual-explanation/

Magomadov, V. S. (2020). Exploring the role of progressive web applications in modern web development. *Journal of Physics: Conference Series*, *1679*, 22043. https://doi.org/10.1088/1742-6596/1679/2/022043

McBride, P. (2018, January 9). *An Introduction to Async / Await — JavaScript January*. https://www.javascriptjanuary.com/blog/an-introduction-to-async-await

Miller, W. (2020). *Why the Cost of Software Testing is Totally Worth It - Software Testing and Quality Assurance by iBeta*. https://www.ibeta.com/why-the-cost-of-software-testing-is-totally-worth-it/

Mock Service Worker. (n.d.). *Introduction - Mock Service Worker Docs*. Retrieved March 19, 2021, from https://mswjs.io/docs/

Mozilla. (n.d.-a). *A re-introduction to JavaScript (JS tutorial) - JavaScript | MDN*. Retrieved January 25, 2021, from https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript

Mozilla. (n.d.-b). *Asynchronous - MDN Web Docs Glossary: Definitions of Web-related terms | MDN*. Retrieved January 29, 2021, from https://developer.mozilla.org/en-US/docs/Glossary/asynchronous

Mozilla. (n.d.-c). *Progressive web apps (PWAs) | MDN*. Retrieved February 8, 2021, from https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps

Mozilla. (n.d.-d). *Same-origin policy - Web security | MDN*. Retrieved March 8, 2021, from https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

npm. (n.d.). *About npm | npm Docs*. Retrieved January 29, 2021, from https://docs.npmjs.com/about-npm

npm-stat. (n.d.). *npm-stat: react, angular, vue*. Retrieved February 8, 2021, from https://npm-stat.com/charts.html?package=react&package=angular&package=vue&from=2020-01-01&to=2021-01-01

Occhino, T. (2015). *React.js Conf 2015 Keynote - Introducing React Native - YouTube*. https://www.youtube.com/watch?v=KVZ-P-ZI6W4&t=0s&list=PLb0IAmt7-GS1cbw4qonlQztYV1TAW0sCr&index=1

Rezvi, M. (2020, August 27). *JavaScript Engines: An Overview*. https://blog.bitsrc.io/javascript-engines-an-overview-2162bffa1187

Simpson, K. (2020). *You Don't Know JS Yet: Get Started*. https://github.com/getify/You-Dont-Know-JS/blob/2nd-ed/get-started/README.md

State of JS 2020: Testing. (2020). https://2020.stateofjs.com/en-US/technologies/testing/

Testing Library. (n.d.). *Testing Library | Testing Library*. Retrieved March 30, 2021, from https://testing-library.com/

U.S. Government Accountability Office. (1992). *PATRIOT MISSILE DEFENSE Software Problem Led to System Failure at Dhahran, Saudi Arabia*. https://www.gao.gov/products/IMTEC-92-26

Witke, S. (2019). *Solving The Blocked Event Loop In Node.js. | by Samuel Witke | Medium*. https://medium.com/@witkesam/solving-the-blocked-event-loop-in-node-js-abb6cac281a7

Liite 1: Aineistonhallintasuunnitelma

Kehitysprojektin aikana pidetään päiväkirjaa excel-dokumentissa, johon kerätään teknistä tietoa projektista. Tämä tieto analysoidaan opinnäytetyötä varten. Päiväkirjaa säilytetään tekijän koulun antamalla OneDrive-pilvipalvelutilillä, ja siitä tehdään säännöllisesti varmuuskopioita tekijän oman tietokoneen D-asemalle. Päiväkirjaa säilytetään D-asemalla sekä OneDrive-pilvipalvelussa ainakin vuoden verran opinnäytetyön valmistumisesta.

Kehitysprojektin aikana pidetyistä kokouksissa kirjoitetaan muistiinpanoja, jotka säilytetään tekijän oman tietokoneen D-asemalla sekä OneDrive-pilvipalvelutilillä. Valmiin projektin onnistumisesta kerätään tietoa opinnäytetyön toimeksiantajan edustajalta, jota säilytetään tekijän oman tietokoneen D-asemalla sekä OneDrive-pilvipalvelutilillä.