Tampereen ammattikorkeakoulu

Ohjelmistotestaus Angular-ohjelmistokehyksellä

Juho Taakala

OPINNÄYTETYÖ
Tammikuu 2021

Tietojenkäsittely
Ohjelmistotuotanto

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittely
Ohjelmistotuotanto

TAAKALA JUHO:
Ohjelmistotestaus Angular-ohjelmistokehyksellä

Opinnäytetyö 49 sivua, joista liitteitä 3 sivua
Tammikuu 2021

Tämän opinnäytetyön tavoitteena oli tuottaa selvitys ohjelmistotestauksesta sekä esitellä yleisiä käytänteitä sen toteuttamiseen Angular-ohjelmistokehyksellä. Tarkoituksena oli tehdä kattava selvitys ohjelmistotestauksesta Angular-ohjelmistokehyksellä esitellen tätä työtä varten tehtyjen Angular-sovelluksen osien testiesimerkkejä sekä Angularin kanssa käytettäviä testaustyökaluja ja niiden toimintaa.

Työssä testien kirjoittamiseen käytettiin TypeScript-ohjelmointikieltä sekä Angular-projektin luomisen yhteydessä riippuvuutena tulevaa Jasmine-testauskehystä, Karma-testiajajaa sekä Protractor päästä päähän -testauskehystä, jolla voidaan suorittaa testit oikeassa selaimessa simuloiden käyttäjän toimintaa.

Opinnäytetyössä käydään läpi ohjelmistotestausta yleisesti, jonka jälkeen selvitetään tarvittavat tekniikat ohjelmistotestaukseen Angular-ohjelmistokehyksellä. Työssä käydään läpi Angular-ohjelmistokehystä ja selvitetään mistä osista Angular-sovellus koostuu, tutkitaan Angularin kanssa testaukseen käytettyjä testauskehyksiä ja työkaluja sekä niiden toimintaa sekä keskitytään eri Angular-sovelluksen osien testaukseen konkreettisten testiesimerkkien avulla.

Asiasanat: ohjelmistotestaus, angular-ohjelmistokehys, typescript, jasmine, karma, protractor

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Business Information Systems
Software Development

TAAKALA JUHO:
Software testing with Angular Framework

Bachelor's thesis 49 pages, appendices 3 pages
January 2021

The aim of this thesis was to produce a report of software testing, and to present general practices for its implementation using the Angular framework. The purpose was to conduct a comprehensive report of software testing with the Angular framework, present test examples of the parts of the Angular application created for this work, as well as introduce testing tools used with Angular and their operations.

In the work, the tests were written with TypeScript programming language. Jasmine test framework, Karma test runner, and Protractor end-to-end test framework, which can run the tests on a real browser simulating user's actions, were also used.

The thesis reviews software testing in general, after which the necessary techniques for software testing with the Angular framework are explained. The work reviews the Angular framework and explains the parts that make up an Angular application, examines the test frameworks and tools used for testing with Angular and their operation, and focuses on testing different parts of the Angular application using concrete test examples.

Key words: software testing, angular framework, typescript, jasmine, karma, protractor

SISÄLLYS

1 JOHDANTO ... 9
2 OHJELMISTOTESTAUS ... 10
2.1 Yleistä ... 10
2.2 Testausmenetelmiä ... 10
2.2.1 Yksikkötestaus ... 11
2.2.2 Savutestaus ... 11
2.2.3 Integraatiotestaus ... 12
2.2.4 Järjestelmätestaus ... 12
2.2.5 Hyväksyntätestaus ... 12
2.2.6 Päästä päähän -testaus ... 13
2.2.7 Regressiotestaus ... 13
2.2.8 Automaatiotestaus ... 14
3 JAVASCRIPT JA TYPESCRIPT ... 15
3.1 JavaScript ... 15
3.2 TypeScript ... 16
3.3 TypeScript ja Angular ... 17
4 ANGULAR ... 18
4.1 Yleistä ... 18
4.2 Komponentit ... 19
4.3 Palvelut ... 20
4.4 Moduulit ... 21
4.5 Direktiivit ... 22
4.6 Putket ... 23
5 OHJELMISTOTESTAUS ANGULARILLA ... 25
5.1 Yleistä ... 25
5.2 Testauskehykset ja testaustyökalut ... 25
5.2.1 Jasmine ... 26
5.2.2 Karma ... 28
5.2.3 Protractor ... 31
5.3 Komponenttien testaus ... 32
5.4 Palvelujen testaus ... 35
5.5 Direktiivien testaus ... 37
5.6 Putkien testaus ... 39
5.7 Päästä päähän -testaus ... 40
6 POHDINTA ... 44
LÄHTEET ... 45

LIITTEET 47
Liite 1. Testidata 1.. 47
Liite 2. Testidata 2.. 48

LYHENTEET JA TERMIT

Ohjelmistotestaus	Ohjelmistotestaus tarkoittaa prosessia, jossa varmistetaan, että kehitettävä ohjelmisto toimii niin kuin sen on suunniteltu toimivan. Ohjelmistotestauksen pääasiallinen tarkoitus on löytää ohjelmistovirheitä eli bugeja ja näin ollen varmistaa ohjelmiston oikea toiminnallisuus.
Yksikkötestaus	Yksikkötestaus on ohjelmistokehityksessä käytetty testaustapa, jossa kehitettävän ohjelman lähdekoodin yksittäisille osille kirjoitetaan testejä, kuten esimerkiksi yhdelle metodille erilliset testit, jotka vahvistavat metodin halutun toiminnallisuuden.
SPA	SPA tulee sanoista single page application ja se tarkoittaa sovellusta, joka sisältää nimensä mukaisesti vain yhden sivun. Sivun ulkoasua päivitetään dynaamisesti hakemalla käyttäjän toimien perusteella palvelimelta uutta käyttäjälle näytettävää dataa.
HTML	HTML tulee sanoista Hypertext Markup Language eli hypertekstin merkintäkieli. Se on standardisoitu merkintäkieli, jolla muodostetaan dokumentin sisällön näyttäminen verkkosivustolla.
CSS	CSS tulee sanoista Cascading Style Sheets eli porrastetut tyyliarkit. CSS:llä voidaan tyyliohjeiden avulla määritellä, miltä HTML-elementit näyttävät verkkosivustolla.
DOM	DOM tulee sanoista Document Object Model eli dokumenttioliomalli. Se on tapa esittää dokumentin, kuten esimerkiksi HTML:n rakenne puuna, jonka olioita voi hakea ja muokata esimerkiksi JavaScriptin avulla.

JavaScript	JavaScript on yksi maailman suosituimmista web-ohjelmointikielistä ja se on HTML:n ja CSS:n rinnalla webin yksi ydintekniikoista. JavaScript mahdollistaa interaktiiviset web-sivustot ja se on olennainen osa verkkosovelluksia.
ECMAScript	ECMAScript (ES) on yleiskäyttöinen ohjelmointikieli, jonka on standardisoinut European Computer Manufacturers Association (Ecma). Kielen standardisoinnin taustalla on ollut tarve standardisoida JavaScript-ohjelmointikieli.
TypeScript	TypeScript on Microsoftin luoma ja ylläpitämä ohjelmointikieli, joka on syntaktisesti sama ohjelmointikieli kuin suosittu varsinkin web-ohjelmoinnissa käytetty JavaScript-ohjelmointikieli. TypeScript lisää JavaScriptiin tyypityksen mahdollisuuden, mahdollistaen ohjelmien paremman ylläpidettävyyden ja testattavuuden.
Angular	Angular on TypeScript-ohjelmointikieleen perustuva avoimen lähdekoodin web-ohjelmistokehys, jonka on kehittänyt Google ja jonka kehitykseen osallistuu myös aktiivinen seuraajayhteisö. Angular-ohjelmistokehys soveltuu sekä web- että mobiilisovellusten kehittämiseen.
CLI	CLI tulee sanoista Command Line Interface eli komentorivikäyttöliittymä. Komentorivikäyttöliittymän avulla käyttäjä voi antaa komentoja tietokoneohjelmistolle tekstin muodossa.
Jasmine	Jasmine on avoimen lähdekoodin testauskehys JavaScriptille ja se on suunniteltu niin, että siinä on helposti luettava syntaksi.

Karma	Karma on testiajaja JavaScriptille. Karman päätavoite on luoda tuottava testausympäristö ohjelmistokehittäjille.
Protractor	Protractor on päästä päähän -testauskehys Angular-sovelluksille. Protractor ajaa ohjelmaan kirjoitetut testit oikeassa selaimessa simuloiden käyttäjän toimintaa.

1 JOHDANTO

Tänä päivänä erilaisten ohjelmistojen tarve ja määrä on valtava. Ohjelmistoja käytetään joka paikassa ja koko ajan, joten niiden oikean toiminnallisuuden varmistaminen on välttämätöntä.

Ohjelmistotestaus tarkoittaa prosessia, jossa varmistetaan, että kehitettävä ohjelmisto toimii niin kuin sen on suunniteltu toimivan. Ohjelmistotestauksen pääasiallinen tarkoitus on löytää ohjelmistovirheitä eli bugeja ja näin ollen varmistaa ohjelmiston oikea toiminnallisuus. Ohjelmistotestaus on erittäin tärkeä prosessi ja sen tarve kasvaa koko ajan, kun kehitetään uusia ohjelmistoratkaisuja.

Tämän opinnäytetyön tavoitteena oli tuottaa selvitys ohjelmistotestauksesta sekä esitellä parhaita käytänteitä sen toteuttamiseen Angular-ohjelmistokehyksellä. Työssä tehtiin kattava selvitys ohjelmistotestauksesta Angular-ohjelmistokehyksellä esitellen tätä työtä varten tehtyjen Angular-sovelluksen osien testiesimerkkejä sekä Angularin kanssa käytettäviä testaustyökaluja ja niiden toimintaa.

Testien kirjoittamiseen käytettiin TypeScript-ohjelmointikieltä sekä Angular-projektin luomisen yhteydessä riippuvuutena tulevaa Jasmine-testauskehystä, Karma-testiajajaa sekä Protractor päästä päähän -testauskehystä. Näiden toimintaa käydään työssä läpi yksityiskohtaisesti.

2 OHJELMISTOTESTAUS

2.1 Yleistä

Yksinkertaisuudessaan ohjelmistotestaus tarkoittaa prosessia, jossa varmistetaan, että kehitettävä ohjelmisto toimii niin kuin sen on suunniteltu toimivan. Ohjelmistotestauksen pääasiallinen tarkoitus on löytää ohjelmistovirheitä eli bugeja. Ohjelmistotestauksesta saatavia hyötyjä ovat esimerkiksi ohjelmistovirheiden estäminen, vähentyneet ohjelmiston kehityskustannukset sekä ohjelmiston suorituskyvyn parantuminen. (IBM)

Monet asiat määrittelevät ohjelmistotestausta ja sen suunnittelua. Tärkein niistä lienee se, että kaikkien testien tulisi olla jäljitettävissä asiakkaan asettamiin vaatimuksiin ohjelmistolle. Testien tulisi olla suunniteltu hyvissä ajoin ennen testien teon aloittamista. Testauksen olisi hyvä alkaa pienistä osista ja edetä systemaattisesti kohti suurempia kokonaisuuksia, eli ensiksi testataan esimerkiksi yksittäisiä komponentteja ja edetään kohti koko järjestelmän kattavia testejä. (Tayal, Gupta ja Agarwal 2009)

Ohjelmistotestausta suunniteltaessa ja suoritettaessa on myös hyvä ymmärtää, että koko ohjelmiston kattavaa testausta on lähes mahdoton tehdä. Jopa kohtalaisen kokoisilla ohjelmistoilla ohjelmiston sisältämien mahdollisten polkujen permutaatioiden määrä on erittäin suuri. Silti testauksella on mahdollista kattaa riittävä määrä ohjelmiston logiikkaa ja varmistaa, että suurin osa vaatimusten mukaisista komponenttitason testeistä on hyväksytysti tehty. (Tayal, Gupta ja Agarwal 2009)

2.2 Testausmenetelmiä

Ohjelmistotestaukseen liittyy useita eri testausmenetelmiä, joita käytetään tarpeen mukaan eri vaiheissa ohjelmiston kehitystä. Funktionaalisia testausmenetelmiä ovat esimerkiksi yksikkötestaus, savutestaus, integraatiotestaus, järjestelmätestaus, hyväksyntätestaus, päästä päähän -testaus ja regressiotestaus

sekä automaatiotestaus. (IBM) Tässä työssä keskitytään lähinnä yksikkötestauksen menetelmään sekä päästä päähän -testausmenetelmään Angular-ohjelmistokehyksellä.

2.2.1 Yksikkötestaus

Yksikkötestauksessa testataan yksittäisiä komponentteja ja esimerkiksi niiden yksittäisiä funktioita, ja todetaan niiden haluttu oikea toiminnallisuus. Yksikkötestejä tekemällä voidaan saavuttaa useita erilaisia hyötyjä. Ohjelmiston yksittäisten moduulien koko on tarpeeksi pieni, joten virheiden paikantaminen on helppoa. Yksikkötestien avulla voidaan löytää virheitä ohjelmistosta jo aikaisessa vaiheessa ohjelmistokehitystä. Tekemällä kattavasti yksikkötestejä yksittäiselle komponentille, voidaan huomata, jos jokin vaadittu ominaisuus puuttuu yksiköltä tai jos jokin vaatimus on puutteellinen. (Tayal, Gupta ja Agarwal 2009)

Testivetoisessa ohjelmistokehityksessä hyödynnetään yksikkötestejä tekemällä ne etukäteen kehitettävälle komponentille. Kun etukäteen kirjoitetut yksikkötestit läpäisevät, voidaan komponentin todeta toimivan halutulla tavalla. Kun komponenteille on saatu tehtyä kattava määrä yksikkötestejä, on refaktorointi helpompaa. Tämä siksi, että yksikkötestejä uudelleen suorittamalla voidaan huomata, jos jokin vanha yksikkötesti menee rikki refaktoroinnin seurauksena. (Johansen 2011)

2.2.2 Savutestaus

Savutestaus on tyypiltään eräänlaista regressiotestausta, jossa tarkoitus on selvittää, kannattaako ohjelmiston käännösversiota lähteä ylipäätään testaamaan sen enempää. Savutestit ovat yleensä automatisoituja testejä, jotka ajetaan aina uudelleen uudessa käännösversiossa. Näiden testien oletetaan siis toimivan ja jos testit eivät mene läpi, voidaan olettaa, että jotakin perustavanlaatuista on ohjelmistossa rikki. (Kaner, Bach ja Pettichord 2001)

2.2.3 Integraatiotestaus

Integraatiotestaus on systemaattinen testaustekniikka, jolla yritetään paljastaa rajapintoihin liittyvät virheet. Integraatiotesteillä yritetään selvittää sitä, että kehitettävän ohjelmiston moduulit ja palvelut toimivat hyvin keskenään.

Integraatiotestauksen ensisijaisena tavoitteena on testata moduulirajapintoja sen varmistamiseksi, ettei parametrien välityksessä ole virheitä, kun yksi moduuli kutsuu toista moduulia. Integraatiotestauksen aikana järjestelmän eri moduulit integroidaan suunnitellusti integrointisuunnitelman avulla. Integrointisuunnitelma määrittää vaiheet ja järjestyksen, jossa moduulit yhdistetään koko järjestelmän toteuttamiseksi. Jokaisen integraatiovaiheen jälkeen osittain integroitu järjestelmä testataan. (Tayal, Gupta ja Agarwal 2009)

2.2.4 Järjestelmätestaus

Järjestelmätestaus on prosessi, jossa testataan nimensä mukaisesti koko järjestelmän toimintaa. Tämän prosessin tarkoitus on se, että varmistetaan järjestelmän toiminta asiakkaan vaatimusten mukaiseksi. Usein järjestelmätestausta harjoitetaan integraatiotestauksen rinnalla. (Educba)

Järjestelmätestaus vaatii erillisen testaussuunnitelman sekä erillisen testausdokumentaation, joka on johdettu sekä ohjelmisto- että laitteistovaatimuksista. Järjestelmätestauksella paljastetaan siis järjestelmätason virheet ja varmistetaan, että koko järjestelmä toimii oletetusti sekä suorituskyvyn että toiminnallisuuden näkökulmasta. Järjestelmätestaus kattaa näin ollen kokonaisen päästä päähän -skenaarion asiakkaan näkökulmasta. (Educba)

2.2.5 Hyväksyntätestaus

Hyväksyntätestaus on testauksen menetelmä, jossa varmistetaan ohjelmiston todellinen toiminta, ja että ohjelmisto toimittaa loppukäyttäjille juuri sen, mitä he

odottavat. Hyväksyntätestaus on siitä hyvä testauksen menetelmä, että sitä voivat harjoittaa myös vähemmän tekniset henkilöt, sillä usein tämän tyyppiset testit on kirjoitettu selvällä kielellä, joten kuka vaan voi ymmärtää mitä ja miten testataan. (Sale 2014)

2.2.6 Päästä päähän -testaus

Päästä päähän (end-to-end, E2E)-testaus on testausmenetelmä, johon sisältyy sovelluksen työnkulun testaus alusta loppuun. Tällä testausmenetelmällä pyritään toistamaan todelliset käyttäjäskenaariot, jotta voidaan varmistua siitä, että järjestelmä toimii oikein integraatioiden ja datan eheyden näkökulmasta. (BrowserStack 2020)

Periaatteessa päästä päähän -testauksessa käydään läpi kaikki toiminnot, joita sovelluksessa voi suorittaa, ja testataan, miten sovellus kommunikoi laitteiston, verkkoyhteyksien, ulkoisten riippuvuuksien, tietokantojen ja muiden sovellusten kanssa. Yleensä päästä päähän -testaus suoritetaan sen jälkeen, kun toiminnallinen testaus ja järjestelmätestaus on saatu valmiiksi. (BrowserStack 2020)

2.2.7 Regressiotestaus

Regressiotestaus on testauksen muoto, jonka tavoitteena on testata se, että ohjelmistoon tehdyt uudet muutokset eivät tuo mukanaan uusia virheitä tai tahatonta käyttäytymistä. Regressiotestauksessa siis suoritetaan jo ennalta suoritettuja testejä uudelleen, jotta voidaan varmistua siitä, että kaikki toimii samalla tavalla kuin ennen uutta muutosta ohjelmistoon. Tarve regressiotestaukseen syntyy siis aina, kun on vaatimus muuttaa ohjelmistokoodia, ja on tarkistettava vaikuttaako uusi koodi ohjelmiston johonkin toiseen osaan. Lisäksi regressiotestausta tarvitaan aina, kun ohjelmistoon lisätään jokin uusi ominaisuus. (Tayal, Gupta ja Agarwal 2009)

2.2.8 Automaatiotestaus

Automaatiotestaus tarkoittaa erillisen ohjelmiston käyttämistä ohjelmiston testaamiseen. Erillisellä ohjelmistolla kontrolloidaan testien ajamista testattavassa ohjelmistossa ja tavoitteena on vertailla testistä saatuja tuloksia oletettuihin lopputuloksiin. Testiautomaatiolla voidaan automatisoida toistuvia tehtäviä, joita yleensä tehtäisiin manuaalisesti.

Automaatiotestausta voidaan myös hyödyntää automatisoidessa vaikeampia tehtäviä, jotka olisivat työläitä tehdä manuaalisesti. Automaatiotestaus on erittäin hyödyllinen apuväline esimerkiksi matalan tason käyttöliittymä-regressiotestaukseen, joka olisi muutoin hyvin työlästä ja aikaa vievää tehdä manuaalisesti. Automaatiotestaus luokin oivan mahdollisuuden tehdä tämän tyyppiset aikaa vievät testaukset automaattisesti. Kun automaattiset testaukset on kerran ohjelmoitu, on ne helppo ajaa nopeasti ja useaan otteeseen testattavalle ohjelmistolle. Näin säästyy kuluja ja testaajien aikaa muuhun manuaaliseen ohjelmistotestaukseen. (Gregory ja Crispin 2014)

3 JAVASCRIPT JA TYPESCRIPT

3.1 JavaScript

JavaScript (JS) on yksi maailman suosituimmista web-ohjelmointikielistä ja se on HTML:n ja CSS:n rinnalla webin yksi ydintekniikoista, kun luodaan interaktiivisia verkkosivustoja. JavaScript on myös kaikkein käytetyin ohjelmointikieli ohjelmistokehittäjien keskuudessa. JavaScript on tehokas yleistarkoituksellinen ohjelmointikieli, joka soveltuu monien ohjelmointitapojen käyttöön, kuten esimerkiksi funktionaalisen- ja imperatiivisen ohjelmointitavan käyttöön sekä olio-ohjelmoinnin käyttöön. (Flanagan 2020; Mozilla 2020)

ECMAScript (ES) on yleiskäyttöinen ohjelmointikieli, jonka on standardisoinut European Computer Manufacturers Association (Ecma). Tämän työn kirjoitushetkellä nykyinen standardi on ECMA-262 11th edition. (Ecma International 2020) Kielen standardisoinnin taustalla on ollut tarve standardisoida JavaScript-ohjelmointikieli. Vaikka ECMAScript ja JavaScript luetaan yleensä synonyymeiksi toisilleen, on JavaScript paljon muuta kuin pelkästään ECMA-262:ssa määritelty standardi. JavaScript sisältää ECMAScriptin lisäksi dokumenttiolio-mallin (Document Object Model, DOM), jonka avulla voidaan kommunikoida HTML/XML-dokumenttien kanssa. JavaScript sisältää myös selainoliomallin (Browser Object Model, BOM), jonka avulla voidaan kommunikoida selaimen komponenttien kanssa. (Frisbie 2019; Mozilla 2020)

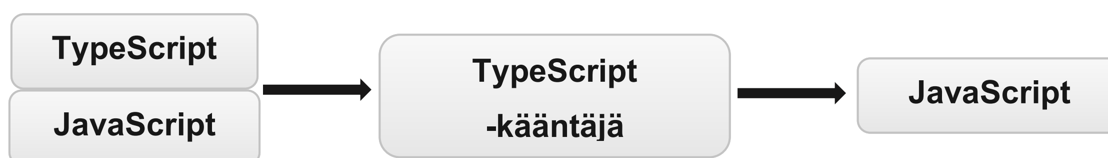
JavaScript-ohjelmointikieli on useimmiten käytössä selaimen puolelle ohjelmoitaessa ja sen avulla voidaan esimerkiksi kontrolloida sitä, miten web-sivusto käyttäytyy erilaisten käyttäjän laukaisemien tapahtumien seurauksena. JavaScriptillä voidaan esimerkiksi kontrolloida dynaamisen sisällön lisäämistä sivulle käyttäjän painaessa tiettyä nappia. Selaimen puolelle ohjelmoinnin lisäksi JavaScript-ohjelmointikieltä voidaan käyttää myös palvelinpuolen ohjelmointiin Node.js:llä, joka siis on ei-selainpohjainen ympäristö, jolloin esimerkiksi aiemmin mainittua selainoliomallia ei voida käyttää. (Mozilla 2020)

3.2 TypeScript

TypeScript on ohjelmointikieli, jonka on kehittänyt ja jota myös ylläpitää, Microsoft. TypeScriptin ensimmäinen versio julkaistiin vuonna 2012. TypeScript on superset JavaScript-ohjelmointikielestä, eli se sisältää kaikki samat ominaisuudet kuin JavaScript, mutta lisää siihen myös omia ominaisuuksiaan. TypeScriptin suurin ero ominaisuuksissa verrattuna JavaScriptiin on staattinen tyyppijärjestelmä. (Microsoft 2020a) Freemanin mukaan staattisen tyyppijärjestelmän ansiosta TypeScript-ohjelmointikielellä ohjelmointi on enemmän ennakoitavampaa verrattuna JavaScriptillä ohjelmointiin esimerkiksi ohjelmoijille, joilla on taustaa C# ja Java-ohjelmointikielten hallinnasta. (Freeman 2019)

TypeScriptin ominaisuuksien hyöty tulee parhaimmin ilmi silloin, kun kehitetään laajoja ohjelmistoja. Microsoftin mukaan tyypityksen ansiosta virheiden havaitseminen ja ennaltaehkäiseminen tuo huomattavat hyödyt ajansäästön muodossa. (Microsoft 2020a)

TypeScript-koodi käännetään JavaScript-koodiksi TypeScript-kääntäjän avulla. Kääntäjä tuottaa puhdasta JavaScript-koodia, jota voidaan suorittaa suoraan esimerkiksi selaimessa tai Node.js:ssä. (Microsoft 2020a) Kuvio 1 havainnollistaa tätä asiaa.



KUVIO 1. TypeScript-koodin muuntaminen JavaScript-koodiksi.

Kuvassa 1 havainnollistetaan JavaScriptin ja TypeScriptin ero tyypityksen osalta. Mikäli ”hello”-funktiolle yritettäisiin TypeScript-tiedostossa (kuvassa oikealla) antaa esimerkiksi ”number”-tyyppistä muuttujaa parametrina, ilmoittaisi TypeScriptin kääntäjä tästä virheilmoituksella, joka omalta osaltaan jo ehkäisisi virheen syntymistä. Mikäli ohjelmoija haluaisi käyttää dynaamista tyypitystä esimerkiksi siinä tilanteessa, että muuttujan tyyppiä ei tiedetä, voisi ”string”-tyypitykset korvata ”any”-tyypillä tai jättää tyypityksen kokonaan tekemättä.

```javascript
function hello(name) {
    return 'Hello, ' + name;
  }

const fullName = 'Brendan Eich';
console.log(hello(fullName));
```

```typescript
function hello(name: string): string {
  return 'Hello, ' + name;
}

const fullName: string = 'Anders Hejlsberg';
console.log(hello(fullName));
```

KUVA 1. Vasemmalla JavaScript-koodia ja oikealla TypeScript-koodia.

3.3 TypeScript ja Angular

Angular-ohjelmistokehys on rakennettu kokonaan TypeScript-ohjelmointikielellä. (Microsoft 2020b) Tämän vuoksi TypeScript toimiikin ensisijaisena ohjelmointikielenä kehitettäessä Angular-sovelluksia. (Angular 2020b)

Jotta Angular-sovellukseen kirjoitettu TypeScript-kieli voidaan kääntää ECMAScriptiksi, tarvitaan erillinen konfiguraatiotiedosto. Kuvassa 2 havainnollistetaan konfiguraatiotiedoston sisältö. Konfiguraatiotiedoston ”target”-kohdassa määritellään se, mille ECMAScriptin versiolle TypScript-koodit käännetään. Kaikki modernit selaimet tukevat vähintään ES2015 versiota. (Angular 2020d)

```json
{
  "compileOnSave": false,
  "compilerOptions": {
    "baseUrl": "./",
    "outDir": "./dist/out-tsc",
    "sourceMap": true,
    "declaration": false,
    "downlevelIteration": true,
    "experimentalDecorators": true,
    "module": "esnext",
    "moduleResolution": "node",
    "importHelpers": true,
    "target": "es2015",
    "typeRoots": [
      "node_modules/@types"
    ],
    "lib": [
      "es2018",
      "dom"
    ]
  },
  "angularCompilerOptions": {
    "fullTemplateTypeCheck": true,
    "strictInjectionParameters": true
  }
}
```

KUVA 2. TypeScript-konfiguraatiotiedosto.

4 ANGULAR

4.1 Yleistä

Angular on TypeScript-ohjelmointikieleen perustuva avoimen lähdekoodin web-ohjelmistokehys, jonka on kehittänyt Google ja jonka kehitykseen osallistuu myös aktiivinen seuraajayhteisö. Angular-ohjelmistokehys soveltuu sekä web- että mobiilisovellusten kehittämiseen. (Angular 2020a)

Angular-ohjelmistokehys on rakennettu Microsoftin kehittämän TypeScript-ohjelmointikielen pohjalta. TypeScript toimiikin pääasiallisena ohjelmointikielenä kehitettäessä Angular-verkkosovelluksia. Selaimet eivät voi suorittaa TypeScript-kieltä suoraan, joten TypeScriptin kääntäminen JavaScriptiksi pitää hoitaa Angular-verkkosovelluksissa erillisten konfiguraatiotiedostojen avulla (Angular 2020b)

Kuviossa 2 havainnollistetaan, mitä selaimia ja mitä selaimien versioita Angular 11 tukee tätä opinnäytetyötä kirjoittaessa tammikuussa 2021.

Selain	Tuetut versiot
Chrome	viimeisin
Firefox	viimeisin ja Extended Support Release (ESR)
Edge	2 viimeisintä versiota
IE	11
Safari	2 viimeisintä versiota
iOS	2 viimeisintä versiota
Android	Q (10.0), Pie (9.0), Oreo (8.0), Nougat (7.0)

KUVIO 2. Selaimet ja selaimien versiot, joita Angular 11 tukee.

4.2 Komponentit

Angular-ohjelmistossa varmasti tärkeimmässä osassa ovat komponentit. Angular-ohjelmisto koostuu pääosin erilaisista komponenteista, joilla kaikilla on oma käyttötarkoituksensa. Esimerkkejä komponenteista voisivat olla vaikkapa verkkosivujen ala- ja yläpalkit. Komponentit ovat erittäin hyödyllisiä, sillä ne mahdollistavat eri ohjelmiston osien uudelleenkäytön.

Kuvassa 3 havainnollistetaan Star Wars-hahmojen hakuun käytetyn Angular-komponentin rakenne. Angular-komponentti koostuu HTML-tiedostosta, tyylitiedostosta sekä TypeScript tiedostosta, joka sisältää komponentin logiikan. Angular CLI:n avulla luotu komponentti sisältää myös komponentin testaustiedoston, jonka nimi sisältää komponentin nimen sekä ”spec.ts”-loppuosan.

```
ANGULAR-TESTING
  src
    app
      mock
        mock-star-wars-character-response.ts
        mock-star-wars-characters-response.ts
      models
        star-wars-character.model.ts
        star-wars-characters.model.ts
      star-wars-characters
        star-wars-characters.component.html
        star-wars-characters.component.scss
        star-wars-characters.component.spec.ts
        star-wars-characters.component.ts
      app-routing.module.ts
      app.component.html
      app.component.sass
      app.component.ts
      app.module.ts
      fetch-star-wars-characters.service.spec.ts
      fetch-star-wars-characters.service.ts
      format-currency.pipe.spec.ts
      format-currency.pipe.ts
      highlight.directive.spec.ts
      highlight.directive.ts
    assets
    environments
    favicon.ico
    index.html
    main.ts
    polyfills.ts
    styles.sass
    test.ts
  .browserslistrc
  .editorconfig
  .gitignore
  angular.json
  debug.log
  karma.conf.js
  package-lock.json
  package.json
  README.md
```

```typescript
import { Component, OnInit } from '@angular/core';
import { FetchStarWarsCharactersService } from '../fetch-star-wars-characters.service';
import { StarWarsCharacters } from '../models/star-wars-characters.model';

@Component({
  selector: 'app-star-wars-characters',
  templateUrl: './star-wars-characters.component.html',
  styleUrls: ['./star-wars-characters.component.scss'],
})
export class StarWarsCharactersComponent implements OnInit {
  public starWarsCharacters: StarWarsCharacters;
  public idInput: string = '';
  public errorMsg: string;

  constructor(public fetchStarWarsCharactersService: FetchStarWarsCharactersService) {}

  ngOnInit() {}

  public fetchStarWarsCharacters() {
    this.starWarsCharacters = null;
    this.errorMsg = null;
    this.fetchStarWarsCharactersService.fetchCharacters().subscribe(
      res => {
        this.starWarsCharacters = res;
      },
      error => {
        this.handleError('Failed to fetch characters.');
      },
    );
  }

  public fetchCharacterById() {
    this.starWarsCharacters = new StarWarsCharacters();
    this.errorMsg = null;
    this.fetchStarWarsCharactersService.fetchCharacterById(this.idInput).subscribe(
      res => {
        this.starWarsCharacters.results.push(res);
      },
      error => {
        this.handleError('Failed to fetch a character with id ' + this.idInput);
      },
    );
  }

  public handleError(errorMsg: string) {
    this.errorMsg = errorMsg;
  }
}
```

KUVA 3. Esimerkki Angular-komponentista.

Kuvan 3 komponentin TypeScript-tiedostossa ”@Component metadata”-konfiguraation ”selector”-kohdassa määritellään komponentin nimi, jolla komponenttiin voidaan viitata jonkun toisen komponentin HTML-tiedostossa seuraavalla tavalla:

```
<app-star-wars-characters></app-star-wars-characters>
```

Komponentin ”templateUrl”-kohta määrittelee komponentin HTML-tiedoston nimen ja sijaintipolun. Komponentin ”styleUrls”-kohta määrittelee komponentin tyylitiedoston nimen ja sijaintipolun. Metadata-konfiguraatioon voidaan lisätä myös esimerkiksi ”providers”-taulukko, johon voidaan lisätä palveluja (services), joita komponentti tarvitsee sekä ”animations”-taulukko, johon voidaan lisätä animaatioita komponentin HTML-elementeille.

4.3 Palvelut

Palvelut ovat Angularissa tyypillisesti luokkia, joilla on jokin tietty, kapea ja hyvin määritelty tarkoitus. Palvelu voisi olla esimerkiksi luokka, jossa haetaan backendiltä jonkin verkkokaupan tuotteen tiedot identifikaattorin perusteella. Angular erottaa palvelut komponenteista, jotta modulaarisuus ja uudelleenkäytettävyys lisääntyisi. Myös komponentin luettavuus ja tehokkuus paranee, kun komponentin näkymään liittyvä funktionaalisuus on erillään muusta prosessoinnista, joka hoidetaan palveluissa.

Komponentti voi siis delegoida tietyt tehtävät palveluille, kuten esimerkiksi edellä mainittu verkkokaupan tuotteen datan haku palvelimelta. Kun palveluluokka on määritelty ”@Injectable()”-määreellä, voidaan palvelua ja sen tarjoamia funktioita käyttää muissa komponenteissa. Angular ei suoraan ”pakota” näihin toimintatapoihin, mutta auttaa noudattamaan näitä periaatteita helpottamalla sovelluslogiikan sisällyttämistä palveluihin ja saattamaan nämä palvelut komponenttien saataville helposti riippuvuusinjektiolla. (Angular 2020e)

Kuvassa 4 havainnollistetaan palvelu, jossa haetaan ohjelmointirajapinnalta Star Wars-hahmojen tietoja.

```typescript
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import { StarWarsCharacter } from './models/star-wars-character.model';
import { StarWarsCharacters } from './models/star-wars-characters.model';

@Injectable({
  providedIn: 'root',
})
export class FetchStarWarsCharactersService {
  public apiUrl: string = 'https://swapi.dev/api/people/';

  constructor(public http: HttpClient) {}

  public fetchCharacters(): Observable<StarWarsCharacters> {
    return this.http.get<StarWarsCharacters>(this.apiUrl);
  }

  public fetchCharacterById(id: string): Observable<StarWarsCharacter> {
    return this.http.get<StarWarsCharacter>(this.apiUrl + id);
  }
}
```

KUVA 4. Esimerkki palvelusta.

4.4 Moduulit

Angular sovellukset ovat hyvin modulaarisia ja Angularissa on oma moduulijärjestelmä nimeltään ”NgModules”. Moduulit ovat eräänlaisia säilöjä, joihin usein säilötään johonkin tiettyyn yhtenäiseen sovellusalueeseen liittyvät toiminnallisuudet. Moduulit voivat sisältää komponentteja, palveluntarjoajia (providers) tai muita kooditiedostoja. Moduulin sisällön määrittelee kyseiseen moduuliin liittyvän sovellusalueen laajuus. Moduulit voivat tuoda toiminnallisuuksia muista moduuleista ja vastavuoroisesti viedä valittuja toiminnallisuuksia muiden moduulien käyttöön. (Angular 2020f)

Jokaisessa Angular-sovelluksessa on vähintään yksi ”NgModule”-luokka, joka toimii juurimoduulina. Tämä juurimoduuli on perinteisesti nimeltään ”AppModule”, jonka tulee löytyä sovelluksesta käynnistyksen yhteydessä. Usein isommissa sovelluksissa on myös useita ominaisuusmoduuleita, joiden tarkoituksena on nimensä mukaan järjestää koodit yhtenäisiä ominaisuuksia sisältäviksi moduuleiksi. Tämän seurauksena juurimoduuli ei kasva liian suureksi ja myös kehittäjien yhteistyö helpottuu. (Angular 2020f)

4.5 Direktiivit

Angularissa direktiivit on tarkoitettu lisäämään toiminnallisuutta elementeille, poistamaan elementtejä tai lisäämään elementtejä dokumenttioliomallissa. Direktiivit ovat luokkia, jotka määritellään ”@Directive()”-määreellä. Komponentti on siis periaatteessa myös direktiivi, mutta sitä laajennetaan omalla komponentti-määreellä, joka tuo mukanaan lisäominaisuuksia. (Angular 2020i)

Angularissa direktiivejä on kahdenlaista tyyppiä: rakenteellisia direktiivejä ja attribuuttidirektiivejä. Attribuuttidirektiivien tarkoitus on muuttaa dokumenttioliomallin elementin tyyliä tai käyttäytymistä, kun taas rakenteellisen direktiivin tarkoitus on dokumenttioliomallin rakenteen muokkaaminen lisäämällä, poistamalla tai manipuloimalla elementtejä. Angularissa on valmiita rakenteellisia direktiivejä, joita ovat ”*ngIf”, ”*ngFor” ja ”*ngSwitch”. Angularissa on myös valmiita attribuuttidirektiivejä, joita ovat esimerkiksi ”ngClass”, ”ngStyle” ja ”ngModel”. (Angular 2020i)

Kuvassa 5 havainnollistetaan attribuuttidirektiiviä. Kyseisellä direktiivillä voidaan korostaa HTML-elementin tausta tietyllä värillä. Kyseistä direktiiviä voitaisiin käyttää HTML-tiedostossa esimerkiksi seuraavalla tavalla:

```
<h1 appHighlight="green">Green highlight</h1>
```

```
1  import { Directive, ElementRef, HostListener, Input } from '@angular/core';
2
3  @Directive({
4    selector: '[appHighlight]',
5  })
6  export class HighlightDirective {
7    public defaultColor = 'red';
8    @Input('appHighlight') highlightColor: string;
9
10   constructor(private elementRef: ElementRef) {}
11
12   @HostListener('mouseenter') onMouseEnter() {
13     this.highlight(this.highlightColor || this.defaultColor);
14   }
15
16   @HostListener('mouseleave') onMouseLeave() {
17     this.highlight(null);
18   }
19
20   private highlight(color: string) {
21     this.elementRef.nativeElement.style.backgroundColor = color;
22   }
23 }
```

KUVA 5. Esimerkki attribuuttidirektiivistä.

4.6 Putket

Angularissa putki (pipe) on tarkoitettu datan muokkaamiseen tiettyyn käyttäjälle esitettävään muotoon. Putket ovat hyödyllisiä, sillä niitä voidaan käyttää koko sovelluksen laajuudessa, kun putki on kerran määritelty. Esimerkiksi päivämäärä voidaan esittää tietyssä muodossa putken avulla. Angularissa on useita valmiiksi rakennettuja putkia, kuten esimerkiksi ”UpperCasePipe”, joka muuntaa tekstin isoiksi kirjaimiksi. Putkia voidaan myös itse rakentaa tarpeen mukaan. (Angular 2020g)

Kuvassa 6 havainnollistetaan rahayksikön formatointiin tehty putki. Kyseisellä putkella voidaan pyöristää sille annettu rahamäärä sekä muuntaa sille annettu rahayksikkö haluttuun, käyttäjälle näytettävään muotoon. Kyseistä putkea voitaisiin käyttää HTML-tiedostossa esimerkiksi seuraavalla tavalla:

```
<span>{{ cartItem.unitPrice | formatCurrency: cartItem.
currency }}</span>
```

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'formatCurrency',
})
export class FormatCurrencyPipe implements PipeTransform {
  transform(value: number, currency: string): string {
    const delimiter = ',';
    let strPriceValue: string;
    if (value === undefined || value === null || currency === undefined) {
      value = 0;
    }
    if (value === 0) {
      strPriceValue = '0' + delimiter + '00';
    }

    const valueIsNegative = value < 0;
    if (valueIsNegative) {
      value = value * -1;
    }

    const ints = Math.trunc(value);
    let cents = '' + Math.round(value * 100 - ints * 100);
    if (cents.length === 0) {
      cents = '00';
    } else if (cents.length < 2) {
      cents = '0' + cents;
    }
    strPriceValue = ints + delimiter + cents;

    if (valueIsNegative && currency === 'USD') {
      return '-$' + strPriceValue;
    } else if (valueIsNegative) {
      strPriceValue = '-' + strPriceValue;
    }

    if (currency === 'EUR') {
      return strPriceValue + ' €';
    } else if (currency === 'USD') {
      return '$' + strPriceValue;
    } else {
      throw new Error('no supported currency specified');
    }
  }
}
```

KUVA 6. Esimerkki putkesta.

5 OHJELMISTOTESTAUS ANGULARILLA

5.1 Yleistä

Angular CLI:lla luodun projektin yhteydessä tulee mukana kaikki, mitä tarvitaan Angular-sovellusten kokonaisvaltaiseen testaamiseen. Projektin luonnin yhteydessä tulee mukana Jasmine-testauskehys, Karma-testiajaja sekä Protractor-testauskehys. Näiden työkalujen avulla voidaan tehdä yksikkötestejä, päästä päähän -testejä sekä integraatiotestejä. On myös mahdollista vaihtaa edellä mainitut työkalut tarpeen mukaan joihinkin toisiin työkaluihin, jotka tarjoavat vastaavanlaisia ominaisuuksia. (Angular 2020c)

Seuraavien kappaleiden komponentin ja palvelun testiesimerkeissä käsitellään tätä työtä varten luodun Star Wars-hahmojen hakusovelluksen testausta. Testiesimerkeissä käytetään mock-dataa. (Liite 1 ja Liite 2)

5.2 Testauskehykset ja testaustyökalut

Testauskehykset ja testaustyökalut tuovat joukon ohjeita ja sääntöjä testitapausten luomiseen ja suunniteluun. Testauskehys koostuu yhdistelmästä käytäntöjä ja työkaluja, joiden avulla on tarkoitus helpottaa testaamista. Nämä ohjeet ja säännöt voivat sisältää esimerkiksi koodausstandardeja, menetelmiä testidatan käsittelyyn, objektivarastoja, erilaisia prosesseja testitulosten tallentamiseen sekä tietoja ulkoisten resurssien käytöstä.

Vaikka testauskehysten ja testaustyökalujen käyttö ei toki ole pakollista, on niistä saatava hyöty niin ajallisesti kuin testauksen helpottamisen näkökulmasta niin suuri, että niiden käyttämättä jättämisessä ei liene järkeä.

Kuten jo edellä on mainittu, Angular CLI:lla luodun projektin yhteydessä tulee mukana testausta helpottamaan Jasmine-testauskehys, Karma-testiajaja sekä Protractor-testauskehys. Näiden toimintaa käydään tarkemmin läpi seuraavissa kappaleissa.

5.2.1 Jasmine

Jasmine on käyttäytymisvetoinen testauskehys, jolla voidaan testata JavaScript koodia. Jasmine ei ole riippuvainen mistään muusta JavaScript-kehyksestä eikä se vaadi dokumenttioliomallia. Jasminessa on puhdas ja selkeä syntaksi, jotta testien kirjoittaminen olisi mahdollisimman sujuvaa. (Jasmine 2020a)

Jasmine-testauskehys tulee Angular CLI:lla luodun projektin yhteydessä oletuksena riippuvuutena. Jasmine tukee kaikkia ominaisuuksia, joita tarvitaan yksikkötestien kirjoittamiseen. Näitä ominaisuuksia ovat muun muassa assertiot, spy-funktiot, mock-metodit sekä tuki asynkronisten operaatioiden testaamiseen. Kuvassa 7 havainnollistetaan Jasminella testausta.

```
function addNumbers(first: number, second: number): number {
  return first + second;
}

describe('Testing examples with Jasmine', () => {
  // Kutsutaan aina ennen testin ajamista
  beforeEach(() => {});

  // Kutsutaan aina testin ajamisen jälkeen
  afterEach(() => {});

  it('should add numbers', () => {
    expect(addNumbers(5, 6)).toEqual(11);
  });

  it('expectation examples', () => {
    expect(true).toBe(true);
    expect(false).not.toBe(true);
    expect('bar').toEqual('bar');
    expect('foo').toMatch('foo');
    expect(addNumbers).toBeDefined();
    expect(undefined).toBeUndefined();
    expect(null).toBeNull();
    expect(true).toBeTruthy();
    expect(false).toBeFalsy();
    expect('hello world').toContain('hello');
    expect(4).toBeGreaterThan(3);
    expect(3).toBeLessThan(4);
    expect(3.141592).toBeCloseTo(3.1415, 0.1);
    expect(() => {
      throw 'Error';
    }).toThrow('Error');
  });

  it('Jasmine spy example', () => {
    const spy = spyOn(Math, 'trunc');
    Math.trunc(3.1415);
    expect(spy).toHaveBeenCalledWith(3.1415);
  });

  // fit = vain tämä testi ajetaan
  fit('should run only this test', () => {});

  // xit = tätä testiä ei ajeta
  xit('should not run this test', () => {});
});
```

KUVA 7. Esimerkki testaamisesta Jasminella.

Jasmine-testit kirjoitetaan ”describe”-funktion sisään. Funktio saa kaksi parametria, joista ensimmäinen on string-tyyppinen, jolle annetaan arvoksi yleensä testattavan asian (esimerkiksi komponentti) nimi. Toinen parametri on funktio, joka on koodilohkolle, johon testit kirjoitetaan. Samalla periaatteella toimii myös ”describe”-funktion sisään kirjoitettavat ”it”-funktiot, joiden sisälle kirjoitetaan testit.

Jasminen ”expect”-funktiot on havainnollistettu kuvan 7 ”it”-funktiossa, johon on annettu ”expectation examples”-parametri. Jasminen ”expect”-funktiot ottavat parametrina annetun arvon ja funktio ketjutetaan vastaavuusfunktioon, joka ottaa odotusarvon vastaan. Vastaavuus-funktiossa tehdään vertaus, jonka tulos raportoidaan Jasminelle totena tai taruna. Tämän vertauksen seurauksena testi joko läpäisee tai ei. (Jasmine 2020b)

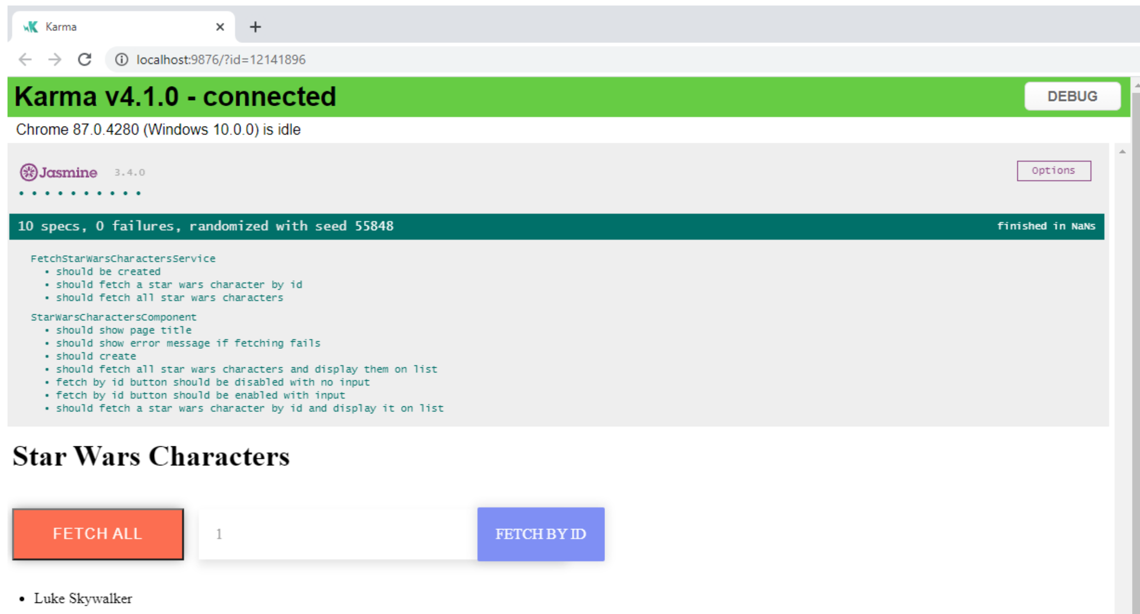
Jasmine tarjoaa globaaleja funktioita helpottamaan koodin toiston vähentämistä. Näitä funktioita ovat ”beforeEach”, ”afterEach”, ”beforeAll” ja ”afterAll”. Nämä funktiot ajetaan, kuten funktioiden nimistä voi päätellä, esimerkiksi kunkin testin jälkeen tai ennen. Näihin funktioihin voidaan siis lisätä testeissä tarvittavia toistuvia asioita.

Jasmine tarjoaa ”spy”-toimintoja, jotka on havainnollistettu kuvan 7 ”Jasmine spy example”-testissä. Jasminen ”spy”-toiminnoilla voidaan mockata haluttua funktiota ja seurata, onko funktiota kutsuttu tai onko sitä kutsuttu jollain tietyllä parametrilla. Jasminen ”spy”-funktioita ovat esimerkiksi ”toHaveBeenCalled”, ”toHaveBeenCalledTimes” ja ”toHaveBeenCalledWith”. Jasminen ”spy”-funktioita voidaan myös ketjuttaa esimerkiksi ”returnValue”-funktiolla, joka palauttaa sille annetun arvon. Hyödyllinen funktio on myös esimerkiksi ”callThrough”, jolla voidaan funktion mockaamisen sijaan ajaa itse funktion toiminnallisuus läpi.

5.2.2 Karma

Karma on testiajaja JavaScriptille ja se toimii työkaluna Jasmine-testien ajamiselle. Karman avulla voidaan yhdellä komennolla käynnistää selain, jossa kirjoi-

tetut testit ajetaan, ja nähdä samalla kaikki testien tulokset kerralla. Karma vähentää siis monia vaiheita, joita testaajan pitäisi ilman testiajajaa tehdä manuaalisesti. Karman avulla voidaan myös havaita testitiedostoihin kirjoitetut muutokset ja ajaa testit uudestaan automaattisesti. Karma tulee Angular CLI:lla luodun projektin yhteydessä automaattisesti riippuvuutena. (Angular 2020c) Karma ajaa testit CLI-komennolla ”ng test”, joka ajetaan projektin sisältämässä kansiossa. Kuvassa 8 havainnollistetaan Karman ajamat testit selaimessa.



KUVA 8. Karman ajamat testit selaimessa.

Jotta kuvan 8 testit on voitu ajaa, on pitänyt tehdä konfiguraatiotiedostoja. Karman täytyy siis tietää projektista, jonka testejä ajetaan, ja tämä tehdään konfiguraatiotiedostojen avulla. (Karma, 2020) Kuvassa 9 on Angular-CLI:n avulla luodun projektin yhteydessä luotu Karma-konfiguraatiotiedosto.

```
1   // Karma configuration file, see link for more information
2   // https://karma-runner.github.io/1.0/config/configuration-file.html
3
4   module.exports = function (config) {
5     config.set({
6       basePath: '',
7       frameworks: ['jasmine', '@angular-devkit/build-angular'],
8       plugins: [
9         require('karma-jasmine'),
10        require('karma-chrome-launcher'),
11        require('karma-jasmine-html-reporter'),
12        require('karma-coverage-istanbul-reporter'),
13        require('@angular-devkit/build-angular/plugins/karma')
14      ],
15      client: {
16        clearContext: false // leave Jasmine Spec Runner output visible in browser
17      },
18      coverageIstanbulReporter: {
19        dir: require('path').join(__dirname, './coverage/angular-testing'),
20        reports: ['html', 'lcovonly', 'text-summary'],
21        fixWebpackSourcePaths: true
22      },
23      reporters: ['progress', 'kjhtml'],
24      port: 9876,
25      colors: true,
26      logLevel: config.LOG_INFO,
27      autoWatch: true,
28      browsers: ['Chrome'],
29      singleRun: false,
30      restartOnFileChange: true
31    });
```

KUVA 9. Karma-konfiguraatiotiedosto.

Kuvan 9 konfiguraatiotiedoston ”frameworks”-kohdassa Jasmine-testauskehys asetetaan projektin testauskehykseksi. Tässä kohtaa voitaisiin myös käyttää jotain muuta testauskehystä, mutta suositeltavaa on käyttää Jasminea, sillä se on Angularin oletuksena tuleva testauskehys.

Konfiguraatiotiedoston ”browsers”-kohdassa määritellään, missä selaimessa testit ajetaan. Kuvan 9 konfiguraatiotiedostoon selaimeksi on määritelty Googlen Chrome-selain, joka mahdollistetaan ”plugins”-kohdan ”karma-chrome-launcher”-liitännäisellä. (Karma 2020)

Konfiguraatiotiedoston ”autoWatch”-kohdassa määritellään se, halutaanko seurata testitiedostoihin tehtyjä muutoksia ja suorittaa testit aina automaattisesti uudelleen, kun testitiedostoon tehdään muutoksia. (Karma 2020)

Kuvassa 10 on Karma-konfiguraatiotiedoston vaatima tiedosto, joka toimii sisääntulona kaikille projektin testitiedostoille (”spec.ts”-tiedostot).

```
1  // This file is required by karma.conf.js and loads recursively all the .spec and framework files
2
3  import 'zone.js/dist/zone-testing';
4  import { getTestBed } from '@angular/core/testing';
5  import {
6    BrowserDynamicTestingModule,
7    platformBrowserDynamicTesting
8  } from '@angular/platform-browser-dynamic/testing';
9
10 declare const require: any;
11
12 // First, initialize the Angular testing environment.
13 getTestBed().initTestEnvironment(
14   BrowserDynamicTestingModule,
15   platformBrowserDynamicTesting()
16 );
17 // Then we find all the tests.
18 const context = require.context('./', true, /\.spec\.ts$/);
19 // And load the modules.
20 context.keys().map(context);
```

KUVA 10. Karma-konfiguraatiotiedoston vaatima tiedosto.

5.2.3 Protractor

Protractor on tarkoitettu Angular-sovellusten päästä päähän -testaukseen ja integraatiotestaukseen. Protractor ajaa sovellukseen kirjoitetut testit oikeassa selaimessa simuloiden käyttäjän toimintaa. Toisin kuin Karma, joka on tarkoitettu lähinnä yksikkötesteille, Protractorilla on tarkoitus tehdä laajemmat testit, joissa on käytössä koko sovelluksen instanssi. Protractor käyttää oletusarvoisesti Jasmine-testauskehystä testausrajapintanaan. (Protractor 2020a.)

Protractorin pitää tietää, miten se yhdistää selaimen ajureihin. Yhdistäminen tehdään yleensä Selenium-palvelimen avulla, mutta yhdistäminen onnistuu myös suoraan Firefox ja Chrome-selaimiin. (Protractor 2020b.) Kuvan 11 konfiguraatiotiedostossa yhdistäminen tapahtuu suoraan Chrome-selaimeen ”directConnect”-kohdan ”true”-arvon takia. Protractorilla testaamista käydään läpi kappaleessa 5.7.

```
// @ts-check
// Protractor configuration file, see link for more information
// https://github.com/angular/protractor/blob/master/lib/config.ts

const { SpecReporter } = require('jasmine-spec-reporter');

/**
 * @type { import("protractor").Config }
 */
exports.config = {
  allScriptsTimeout: 11000,
  specs: [
    './src/**/*.e2e-spec.ts'
  ],
  capabilities: {
    browserName: 'chrome'
  },
  directConnect: true,
  baseUrl: 'http://localhost:4200/',
  framework: 'jasmine',
  jasmineNodeOpts: {
    showColors: true,
    defaultTimeoutInterval: 30000,
    print: function() {}
  },
  onPrepare() {
    require('ts-node').register({
      project: require('path').join(__dirname, './tsconfig.json')
    });
    jasmine.getEnv().addReporter(new SpecReporter({ spec: { displayStacktrace: true } }));
  }
};
```

KUVA 11. Protractor-konfiguraatiotiedosto.

5.3 Komponenttien testaus

Angular-komponentit sisältävät HTML-mallin sekä TypeScript-luokan. Komponenttien testaamisen tarkoituksena on selvittää, että edellä mainitut toimivat keskenään juuri halutulla tavalla.

Komponentin testauksessa vaaditaan, että komponentin isäntäelementti luodaan selaimen dokumenttioliomallissa. Näin voidaan tutkia komponenttiluokan vuorovaikutusta dokumenttioliomallin kanssa komponentin mallin kuvaamalla tavalla. Angularin ”TestBed” luo perustan tämänkaltaiselle testaustavalle. Sen avulla konfiguroidaan ja alustetaan ympäristö yksikkötestejä varten ja se tarjoaa myös funktioita komponenttien ja palvelujen luontiin yksikkötestejä varten. (Angular 2020h)

Kuvassa 12 ja kuvassa 13 havainnollistetaan aiemman kappaleen komponenttiesimerkin (kuva 3) testausta.

```
import { HttpClientTestingModule } from '@angular/common/http/testing';
import { ComponentFixture, TestBed, waitForAsync } from '@angular/core/testing';
import { FormsModule } from '@angular/forms';
import { By } from '@angular/platform-browser';
import { throwError } from 'rxjs';
import { of } from 'rxjs/internal/observable/of';
import { FetchStarWarsCharactersService } from '../fetch-star-wars-characters.service';
import { TEST_CHARACTER } from '../mock/mock-star-wars-character-response';
import { TEST_ALL_CHARACTERS } from '../mock/mock-star-wars-characters-response';
import { StarWarsCharactersComponent } from './star-wars-characters.component';

describe('StarWarsCharactersComponent', () => {
  let component: StarWarsCharactersComponent;
  let fixture: ComponentFixture<StarWarsCharactersComponent>;
  let fetchStarWarsCharactersService: FetchStarWarsCharactersService;

  beforeEach(
    waitForAsync(() => {
      TestBed.configureTestingModule({
        imports: [HttpClientTestingModule, FormsModule],
        declarations: [StarWarsCharactersComponent],
      }).compileComponents();

      fetchStarWarsCharactersService = TestBed.inject(FetchStarWarsCharactersService);
    }),
  );

  beforeEach(() => {
    fixture = TestBed.createComponent(StarWarsCharactersComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('should show page title', () => {
    const titleElement = fixture.debugElement.query(By.css('.page-title'));

    expect(titleElement).toBeDefined();
    expect(titleElement.nativeElement.textContent).toEqual('Star Wars Characters');
  });

  it('should fetch all star wars characters and display them on list', () => {
    const fetchAllButton = fixture.debugElement.query(By.css('.fetch-all-button'));

    const fetchStarWarsCharactersServiceSpy = spyOn(fetchStarWarsCharactersService, 'fetchCharacters').and.returnValue(
      of(TEST_ALL_CHARACTERS),
    );

    fetchAllButton.nativeElement.click();
    fixture.detectChanges();

    expect(fetchStarWarsCharactersServiceSpy).toHaveBeenCalled();

    const firstListItem = fixture.debugElement.queryAll(By.css('li'))[0];
    const secondListItem = fixture.debugElement.queryAll(By.css('li'))[1];

    expect(firstListItem.nativeElement.textContent).toEqual('Luke Skywalker');
    expect(secondListItem.nativeElement.textContent).toEqual('C-3PO');
  });

```

KUVA 12. Esimerkki kuvan 3 komponentin testaamisesta.

```
    it('fetch by id button should be disabled with no input', () => {
      const fetchByIdButton = fixture.debugElement.query(By.css('.fetch-by-id-button'));

      expect(component.idInput.length).toBe(0);
      expect(fetchByIdButton.nativeElement.disabled).toBe(true);
    });

    it('fetch by id button should be enabled with input', () => {
      const fetchByIdButton = fixture.debugElement.query(By.css('.fetch-by-id-button'));

      expect(component.idInput.length).toBe(0);
      expect(fetchByIdButton.nativeElement.disabled).toBe(true);

      component.idInput = '1';
      fixture.detectChanges();

      expect(component.idInput.length).toBe(1);
      expect(fetchByIdButton.nativeElement.disabled).toBe(false);
    });

    it('should fetch a star wars character by id and display it on list', () => {
      const fetchByIdButton = fixture.debugElement.query(By.css('.fetch-by-id-button'));

      const fetchStarWarsCharactersServiceSpy = spyOn(
        fetchStarWarsCharactersService,
        'fetchCharacterById',
      ).and.returnValue(of(TEST_CHARACTER));

      component.idInput = '1';
      fixture.detectChanges();

      fetchByIdButton.nativeElement.click();
      fixture.detectChanges();

      expect(fetchStarWarsCharactersServiceSpy).toHaveBeenCalledWith('1');

      const listItemElement = fixture.debugElement.query(By.css('li'));
      expect(listItemElement.nativeElement.textContent).toEqual('Luke Skywalker');
    });

    it('should show error message if fetching fails', () => {
      const fetchByIdButton = fixture.debugElement.query(By.css('.fetch-by-id-button'));

      const fetchStarWarsCharactersServiceSpy = spyOn(
        fetchStarWarsCharactersService,
        'fetchCharacterById',
      ).and.returnValue(throwError({ status: 404 }));

      component.idInput = '254';
      fixture.detectChanges();

      fetchByIdButton.nativeElement.click();
      fixture.detectChanges();

      expect(fetchStarWarsCharactersServiceSpy).toHaveBeenCalledWith('254');

      const errorMsgElement = fixture.debugElement.query(By.css('.error-msg'));
      expect(errorMsgElement.nativeElement.textContent).toEqual('Failed to fetch a character with id 254');
    });
  });
```

KUVA 13. Jatkoa kuvan 12 komponenttitesteille.

Ensimmäisessä ”beforeEach”-funktiossa konfiguroidaan komponentti yksikkötestejä varten asettamalla yksikkötestien vaatimat komponentit, palvelut ja moduulit. Seuraavassa ”beforeEach”-funktiossa luodaan itse komponentti sekä fikstuuri, joita käytetään testien kirjoittamisessa. Testeissä esiintyvillä ”detectChanges”-funktioilla käynnistetään muutosten havaitseminen ja päivitetään näkymä. Muita

tärkeitä komponenttien testaukseen liittyviä asioita ovat muun maussa ”debugElement”, jonka avulla voidaan hakea ”By.css”-metodilla HTML-elementtejä sekä ”nativeElement”, jolla päästään käsiksi kyseiseen elementtiin ja voidaan testata sen toiminnallisuuta. (Angular 2020h)

5.4 Palvelujen testaus

Angularin palvelut sisältävät yleensä uudelleenkäytettävää logiikkaa ja usein palveluissa tehdään kutsuja backendiin, joten testaaminen on hyvin suoraviivaista. Testataan esimerkiksi sitä, että kutsu lähtee oikeaan osoitteeseen oikeilla parametreilla ja vastaus palautuu halutunlaisena. Angular sisältää myös valmiita kirjastoja HTTP-kutsujen ja vastauksien mockaamiseen, joten palvelujen, joissa tehdään kutsuja, yksikkötestaus on helpohkoa.

Kuvassa 14 havainnollistetaan aiemmassa kappaleessa esitellyn palvelun (kuva 4) testaamista.

```
import { HttpClientTestingModule, HttpTestingController } from '@angular/common/http/testing';
import { TestBed } from '@angular/core/testing';
import { FetchStarWarsCharactersService } from './fetch-star-wars-characters.service';
import { TEST_CHARACTER } from './mock/mock-star-wars-character-response';
import { TEST_ALL_CHARACTERS } from './mock/mock-star-wars-characters-response';
import { StarWarsCharacter } from './models/star-wars-character.model';

describe('FetchStarWarsCharactersService', () => {
  let httpTestingController: HttpTestingController;
  let fetchStarWarsCharactersService: FetchStarWarsCharactersService;
  const apiUrl = 'https://swapi.dev/api/people/';

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule],
      providers: [FetchStarWarsCharactersService],
    });

    httpTestingController = TestBed.inject(HttpTestingController);
    fetchStarWarsCharactersService = TestBed.inject(FetchStarWarsCharactersService);
  });

  afterEach(() => {
    httpTestingController.verify();
  });

  it('should be created', () => {
    expect(fetchStarWarsCharactersService).toBeTruthy();
  });

  it('should fetch a star wars character by id', () => {
    fetchStarWarsCharactersService.fetchCharacterById('1').subscribe(res => {
      expect(res.name).toEqual('Luke Skywalker');
    });

    const req = httpTestingController.expectOne(apiUrl + '1');

    expect(req.request.method).toEqual('GET');

    req.flush(TEST_CHARACTER);
  });

  it('should fetch all star wars characters', () => {
    fetchStarWarsCharactersService.fetchCharacters().subscribe(res => {
      const results = res.results as StarWarsCharacter[];
      expect(results[0].name).toEqual('Luke Skywalker');
      expect(results[1].name).toEqual('C-3PO');
    });

    const req = httpTestingController.expectOne(apiUrl);

    expect(req.request.method).toEqual('GET');

    req.flush(TEST_ALL_CHARACTERS);
  });
});
```

KUVA 14. Esimerkki kuvan 4 palvelun testauksesta.

Kuvan 14 esimerkissä testitiedostoon injektoidaan ”HttpTestingController”, jolla voidaan mockata HTTP-kutsuja. Testien ”expectOne”-funktiolla testataan, että haluttua päätepistettä kutsutaan vain kerran oikealla osoitteella. Tämän jälkeen testataan, että kutsun tyyppi on oikea, eli ”GET”. Sen jälkeen ”flush”-funktiolla

ratkaistaan pyyntö antamalla parametrina kutsun runkona (body) testidata. (Liite 1 ja Liite 2) Runko muunnetaan oletusarvoisesti JSON-muotoiseksi, jos kutsussa ei muuta tyyppiä määritellä. Kun ”flush”-funktio on suoritettu, palautuu ”subscribe”-funktioon testidata, joka todetaan oikeaksi. Testitiedoston ”afterEach”-funktiossa vahvistetaan, että odottavia HTTP-pyyntöjä ei ole enää jäljellä.

5.5 Direktiivien testaus

Direktiivien huolellinen testaus on erittäin tärkeää, sillä direktiivejä voidaan käyttää useissa eri sovelluksen osissa. Sen sijaan, että direktiivejä testattaisiin jossain tietyssä komponentissa, jossa direktiiviä käytetään, on hyvä luoda erillinen testikomponentti direktiiviä varten. Näin saadaan helposti ja selkeästi testattua kaikki direktiivin käyttötapaukset.

Kuvassa 15 havainnollistetaan aiemmassa kappaleessa esitellyn direktiivin (Kuva 5) testaamista.

```
import { Component, DebugElement } from '@angular/core';
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { By } from '@angular/platform-browser';
import { HighlightDirective } from './highlight.directive';

@Component({
  template: `
    <h1 appHighlight="green">This will be green</h1>
    <h1 appHighlight>This will be defaultColor (red)</h1>
    <h1>This will have no highlight</h1>
  `,
})
class TestHighlightDirectiveComponent {}

describe('HighlightDirective', () => {
  let fixture: ComponentFixture<TestHighlightDirectiveComponent>;
  let elementsWithHighlighDirective: DebugElement[];
  let elementWithoutHighlightDirective: DebugElement;

  beforeEach(() => {
    fixture = TestBed.configureTestingModule({
      declarations: [HighlightDirective, TestHighlightDirectiveComponent],
    }).createComponent(TestHighlightDirectiveComponent);

    fixture.detectChanges();

    elementsWithHighlighDirective = fixture.debugElement.queryAll(By.directive(HighlightDirective));
    elementWithoutHighlightDirective = fixture.debugElement.query(By.css('h1:not([appHighlight])'));
  });

  it('should have 2 elements with highlight directive', () => {
    expect(elementsWithHighlighDirective.length).toBe(2);
  });

  it('should color 1st <h1> background with green on mouse enter', () => {
    const event = new Event('mouseenter');
    elementsWithHighlighDirective[0].nativeElement.dispatchEvent(event);

    const backgroundColor = elementsWithHighlighDirective[0].nativeElement.style.backgroundColor;
    expect(backgroundColor).toBe('green');
  });

  it('should color 2nd <h1> background with default color (red) on mouse enter', () => {
    const event = new Event('mouseenter');
    elementsWithHighlighDirective[1].nativeElement.dispatchEvent(event);

    const dir = elementsWithHighlighDirective[1].injector.get(HighlightDirective) as HighlightDirective;
    const backgroundColor = elementsWithHighlighDirective[1].nativeElement.style.backgroundColor;
    expect(backgroundColor).toBe(dir.defaultColor);
  });

  it('should remove highlight color on mouse leave', () => {
    const mouseEnter = new Event('mouseenter');
    elementsWithHighlighDirective[0].nativeElement.dispatchEvent(mouseEnter);

    const backgroundColorEnter = elementsWithHighlighDirective[0].nativeElement.style.backgroundColor;
    expect(backgroundColorEnter).toBe('green');

    const mouseLeave = new Event('mouseleave');
    elementsWithHighlighDirective[0].nativeElement.dispatchEvent(mouseLeave);

    const backgroundColorLeave = elementsWithHighlighDirective[0].nativeElement.style.backgroundColor;
    expect(backgroundColorLeave).toBe('');
  });

  it('<h1> without highlight directive should not have a customProperty', () => {
    expect(elementWithoutHighlightDirective.properties.customProperty).toBeUndefined();
  });
});
```

Kuva 15. Esimerkki kuvan 5 direktiivin testauksesta.

Ennen testejä luodaan testikomponentti, joka sisältää kaikki direktiivin käyttötapaukset. Testeissä direktiivin toimintaa voidaan testata ”dispatchEvent”-funktion avulla, jossa lähetetään direktiiville tapahtumia (event), joita tässä tapauksessa ovat direktiivissä vastaanotetut ”mouseenter” ja ”mouseleave”. Näiden tapahtumien avulla voidaan testata, että direktiivi toimii halutulla tavalla, eli se värjää elementin taustan direktiiville annetun värin mukaan silloin, kun kursori viedään elementin päälle ja poistaa värin, kun kursori liikutetaan elementin päältä pois.

5.6 Putkien testaus

Angular-putkien testaus on melko helppoa, sillä siihen ei välttämättä tarvita mitään Angularin tarjoamia työkaluja, vaan testit voidaan tehdä käyttämällä pelkästään Jasminea. Putkelle voitaisiin toki tehdä myös DOM-testejä komponentin testitiedostossa, mikäli putkea käytetään jossakin komponentin HTML-elementissä. Putken toiminnallisuus voitaisiin todeta testaamalla, että elementti, jolle putki on annettu, muuttaa elementin sisältöä putkessa määritellyllä tavalla.

Kuvassa 16 havainnollistetaan aiemmassa kappaleessa esitellyn rahayksikön formatointiin käytetyn putken (Kuva 6) testaamista.

```
import { FormatCurrencyPipe } from './format-currency.pipe';

describe('FormatCurrencyPipe', () => {
  const pipe = new FormatCurrencyPipe();

  it('display EUR currency int values correctly', () => {
    expect(pipe.transform(123, 'EUR')).toBe('123,00 €');
    expect(pipe.transform(-123, 'EUR')).toBe('-123,00 €');
  });

  it('display EUR currency cents correctly', () => {
    expect(pipe.transform(123.32, 'EUR')).toBe('123,32 €');
    expect(pipe.transform(-123.32, 'EUR')).toBe('-123,32 €');
    expect(pipe.transform(0.32, 'EUR')).toBe('0,32 €');
    expect(pipe.transform(-0.32, 'EUR')).toBe('-0,32 €');
    expect(pipe.transform(0, 'EUR')).toBe('0,00 €');
  });

  it('display USD currency int values correctly', () => {
    expect(pipe.transform(123, 'USD')).toBe('$123,00');
    expect(pipe.transform(-123, 'USD')).toBe('-$123,00');
  });

  it('display USD currency cents correctly', () => {
    expect(pipe.transform(123.32, 'USD')).toBe('$123,32');
    expect(pipe.transform(-123.32, 'USD')).toBe('-$123,32');
    expect(pipe.transform(0.32, 'USD')).toBe('$0,32');
    expect(pipe.transform(-0.32, 'USD')).toBe('-$0,32');
    expect(pipe.transform(0, 'USD')).toBe('$0,00');
  });

  it('should throw error with currency that is not supported', () => {
    expect(() => pipe.transform(123.32, 'SEK')).toThrowError('no supported currency specified');
  });
});
```

KUVA 16. Esimerkki kuvan 6 putken testaamisesta.

Kuvan 16 testitiedostossa ”describe”-funktion sisällä ennen ”it”-testifunktioita luodaan muuttuja, johon luodaan uusi instanssi putkesta, jota ollaan testaamassa. Koska putki on puhdas tilaton funktio, ei ennen testejä tarvita ”beforeEach”-funktiota, jossa putki luotaisiin aina ennen kunkin testin ajamista. (Angular 2020j)

5.7 Päästä päähän -testaus

Aiemmin mainittu Protractor on tarkoitettu Angular-sovellusten päästä päähän -testaukseen. Päästä päähän -testeissä testataan sovelluksen toimintaa käyttöliittymän kautta.

Kuvassa 17 esitellään Protractor-funktioita, joiden tarkoituksena on olla vuorovaikutuksessa testattavien HTML-elementtien kanssa.

```
import { browser, by, element } from 'protractor';

export class AppPage {
  public navigateTo() {
    return browser.get(browser.baseUrl) as Promise<any>;
  }

  public setIdInput(id: number) {
    element(by.id('fetch-by-id-input')).sendKeys(id);
  }

  public clickFetchByIdButton() {
    element(by.id('fetch-by-id-button')).click();
  }

  public clickFetchAllButton() {
    element(by.id('fetch-all-button')).click();
  }

  public getAllDisplayedCharacters() {
    return element.all(by.css('.characters li'));
  }

  public getDisplayedCharacterName() {
    return element(by.id('character')).getText() as Promise<string>;
  }

  public getDisplayedErrorMsg() {
    return element(by.id('error-msg')).getText() as Promise<string>;
  }
}
```

KUVA 17. Protractor-funktioita päästä päähän -testaukseen.

Kuvan 17 ”AppPage”-luokka sisältää funktioita, joissa annetaan arvoja HTML input-elementille, palautetaan HTML-elementtejä ja niiden sisältämiä tekstejä sekä simuloidaan nappuloiden painamista.

Kuvassa 18 havainnollistetaan kuvan 3 komponentin päästä päähän -testausta.

```
import { AppPage } from './app.po';

describe('Jasmine and Protractor E2E tests', () => {
  let page: AppPage;

  beforeEach(() => {
    page = new AppPage();
    page.navigateTo();
  });

  it('should fetch a character by id', () => {
    page.setIdInput(1);
    page.clickFetchByIdButton();

    const fetchedCharacter = page.getDisplayedCharacterName();

    expect(fetchedCharacter).toEqual('Luke Skywalker');
  });

  it('should fetch all characters', () => {
    page.clickFetchAllButton();

    page.getAllDisplayedCharacters().then(fetchedCharacters => {
      expect(fetchedCharacters.length).toBe(10);
      expect(fetchedCharacters[0].getText()).toEqual('Luke Skywalker');
      expect(fetchedCharacters[1].getText()).toEqual('C-3PO');
    });
  });

  it('should display error message if fetching fails', () => {
    page.setIdInput(200);
    page.clickFetchByIdButton();

    const errorMsg = page.getDisplayedErrorMsg();

    expect(errorMsg).toEqual('Failed to fetch a character with id ' + 200);
  });
});
```

KUVA 18. Kuvan 3 komponentin päästä päähän -testausta.

Kuvan 18 testeissä käytetään hyväksi kuvan 17 luokassa luotuja funktioita. Tiedoston ”beforeEach”-funktiossa luodaan uusi olio ”AppPage”-luokasta ja käytetään sen ”navigateTo”-funktiota, jonka avulla voidaan navigoida halutulle sivulle. Tämän jälkeen testeissä voidaan käyttää ”page”-muuttujaa apuna testattaessa sivun HTML-elementtien toimintaa.

Protractor suorittaa päästä päähän -testit CLI-komennolla ”ng e2e”, joka ajetaan projektin sisältämässä kansiossa. Kuvassa 19 havainnollistetaan Protractor päästä päähän -testien suorituksen jälkeinen palaute testien onnistumisesta.

```
** Angular Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200/ **
: Compiled successfully.
[11:16:48] I/launcher - Running 1 instances of WebDriver
[11:16:48] I/direct - Using ChromeDriver directly...

DevTools listening on ws://127.0.0.1:55982/devtools/browser/d12822b2-4945-4306-86f0-5e1a3dbcb9b2
Jasmine started

  Jasmine and Protractor E2E tests
    √ should fetch a character by id
    √ should fetch all characters
    √ should display error message if fetching fails

Executed 3 of 3 specs SUCCESS in 3 secs.
[11:16:54] I/launcher - 0 instance(s) of WebDriver still running
[11:16:54] I/launcher - chrome #01 passed
```

KUVA 19. Protractor-testitulokset.

6 POHDINTA

Ohjelmistotestauksen tärkeyttä ei voi tarpeeksi korostaa, kun rakennetaan uusia ohjelmistoja. Ohjelmistotestauksen lopullisena tavoitteena on ohjelmistovirheiden löytäminen ja lopulta täysin toimiva ohjelmisto. Tämän tavoitteen saavuttamiseksi on välttämätöntä suorittaa huolellinen ja kaiken kattava testaus, jota suoritetaan jatkuvasti ohjelmiston eri kehitysvaiheissa.

Tässä työssä selvitettiin ohjelmistotestausta yleisesti, yleisimpiä testausmenetelmiä sekä selvitettiin ohjelmistotestausta kehitettäessä uusia ohjelmistoja Angular-ohjelmistokehyksellä. Työssä keskityttiin siihen, mistä eri osista Angular-sovellukset koostuvat sekä miten ja millä testauskehyksillä ja testaustyökaluilla näitä osia kannattaisi testata. Opinnäytetyön tavoitteena oli tuottaa selvitys ohjelmistotestauksesta sekä esitellä yleisiä käytänteitä sen toteuttamiseen Angular-ohjelmistokehyksellä. Erityisesti työssä keskityttiin yksikkötestauksen menetelmään.

Angular on hyvin suosittu ohjelmistokehys kehitettäessä nykyaikaisia yksisivuisia web-sovelluksia. Angular tukee suosituimpia testauskehyksiä ja testaustyökaluja, jotka helpottavat testaajan työtä. Tässä työssä tutkittiin kattavasti erilaisia testauskehyksiä ja testaustyökaluja, joita on suositeltavaa käyttää Angularin kanssa, sekä tuotettiin kattava selvitys niiden käytöstä Angularin kanssa.

Tässä työssä esiteltiin tätä työtä varten tehtyjen Angular-sovelluksen osien testiesimerkkejä ja annettiin näin ollen kattava katsaus siihen, miten erilaisia Angular-sovelluksen osia kannattaisi testata. Lopputuloksena syntyi selvitys, jota voi käyttää apuna, kun harjoitetaan ohjelmistotestausta Angular-ohjelmistokehyksellä.

LÄHTEET

Angular. 2020a. One framework. Mobile & desktop. Luettu 19.12.2020. https://angular.io/

Angular. 2020b. TypeScript configuration. Luettu 20.12.2020. https://angular.io/guide/typescript-configuration

Angular. 2020c. Testing. Luettu 21.12.2020. https://angular.io/guide/testing

Angular. 2020d. Deployment. Luettu 22.12.2020. https://angular.io/guide/deployment

Angular. 2020e. Introduction to services and dependency injection. Luettu 23.12.2020. https://angular.io/guide/architecture-services

Angular. 2020f. Introduction to modules. Luettu 24.12.2020. https://angular.io/guide/architecture-modules

Angular. 2020g. Transforming Data Using Pipes. Luettu 29.12.2020. https://angular.io/guide/pipes

Angular. 2020h. Basics of testing components. Luettu 30.12.2020. https://angular.io/guide/testing-components-basics

Angular. 2020i. Directive. Luettu 30.12.2020 https://angular.io/api/core/Directive

Angular. 2020j. Testing Pipes. Luettu 2.1.2021. https://angular.io/guide/testing-pipes

BrowserStack. 2020. Luettu 20.12.2020. https://www.browserstack.com/guide/end-to-end-testing

Dooley, J. 2017. Software Development, Design and Coding: With Patterns, Debugging, Unit Testing, and Refactoring. California: Apress.

Educba. Luettu 25.12.2020. https://www.educba.com/system-testing/

Ecma International. 2020. Luettu 24.12.2020. https://www.ecma-international.org/publications/standards/Ecma-262.htm

Flanagan, D. 2020. JavaScript: The Definitive Guide, 7th Edition. California: O'Reilly Media, Inc.

Frisbie, M. 2019. Professional JavaScript for Web Developers, 4th Edition. Hoboken: John Wiley and Sons, Inc.

Freeman, A. 2019. Essential TypeScript: From Beginner to Pro. California: Apress.

Gregory, J. ja Crispin, L. 2014. More Agile Testing: Learning Journeys for the Whole Team. Boston: Addison-Wesley Professional.

IBM. Software testing. Luettu 20.12.2020. https://www.ibm.com/topics/software-testing

Jasmine. 2020a. Behavior-Driven JavaScript. Luettu 29.12.2020 https://jasmine.github.io/index.html

Jasmine. 2020b. Luettu 1.1.2021. https://jasmine.github.io/2.0/introduction

Johansen, C. 2011. Test-driven JavaScript development. Place of publication not identified Addison Wesley.

Kaner, C., Bach, J. ja Pettichord B. 2001. Lessons Learned in Software Testing: A Context-Driven Approach. Hoboken: John Wiley & Sons, Inc.

Karma. 2020. Configuration File. Luettu 21.12.2020. https://karma-runner.github.io/1.0/config/configuration-file.html

Mozilla. 2020. Luettu 21.12.2020. https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript

Microsoft. 2020a. Typed JavaScript at Any Scale. Luettu 22.12.2020. https://www.typescriptlang.org/

Microsoft. 2020b. Angular. Luettu 22.12.2020. https://www.typescriptlang.org/docs/handbook/angular.html

Protractor. 2020a. Luettu 23.12.2020. https://github.com/angular/protractor/blob/master/lib/config.ts

Protractor. 2020b. End to end testing for Angular. Luettu 23.12.2020. https://www.protractortest.org/#/

Sale, D. 2014. Testing Python: Applying Unit Testing, TDD, BDD and Acceptance Testing. Hoboken: John Wiley & Sons, Inc.

Seshadri, S. 2018. Angular: Up and Running. California: O'Reilly Media, Inc.

Tayal, S.P, Gupta, M. ja Agarwal B. 2009. Software Engineering and Testing. Burlington: Jones & Bartlett Learning.

LIITTEET

Liite 1. Testidata 1

```
export const TEST_CHARACTER = {
  name: 'Luke Skywalker',
  height: '172',
  mass: '77',
  hair_color: 'blond',
  skin_color: 'fair',
  eye_color: 'blue',
  birth_year: '19BBY',
  gender: 'male',
  homeworld: 'http://swapi.dev/api/planets/1/',
  films: [
    'http://swapi.dev/api/films/1/',
    'http://swapi.dev/api/films/2/',
    'http://swapi.dev/api/films/3/',
    'http://swapi.dev/api/films/6/',
  ],
  species: [],
  vehicles: ['http://swapi.dev/api/vehicles/14/', 'http://swapi.dev/api/vehicles/30/'],
  starships: ['http://swapi.dev/api/starships/12/', 'http://swapi.dev/api/star-
ships/22/'],
  created: '2014-12-09T13:50:51.644000Z',
  edited: '2014-12-20T21:17:56.891000Z',
  url: 'http://swapi.dev/api/people/1/',
};
```

Liite 2. Testidata 2


```
export const TEST_ALL_CHARACTERS = {
  count: 82,
  next: 'http://swapi.dev/api/people/?page=2',
  previous: null,
  results: [
    {
      name: 'Luke Skywalker',
      height: '172',
      mass: '77',
      hair_color: 'blond',
      skin_color: 'fair',
      eye_color: 'blue',
      birth_year: '19BBY',
      gender: 'male',
      homeworld: 'http://swapi.dev/api/planets/1/',
      films: [
        'http://swapi.dev/api/films/1/',
        'http://swapi.dev/api/films/2/',
        'http://swapi.dev/api/films/3/',
        'http://swapi.dev/api/films/6/',
      ],
      species: [],
      vehicles: ['http://swapi.dev/api/vehicles/14/', 'http://swapi.dev/api/vehi-
cles/30/'],
      starships: ['http://swapi.dev/api/starships/12/', 'http://swapi.dev/api/star-
ships/22/'],
      created: '2014-12-09T13:50:51.644000Z',
      edited: '2014-12-20T21:17:56.891000Z',
      url: 'http://swapi.dev/api/people/1/',
    },
    {
      name: 'C-3PO',
      height: '167',
      mass: '75',
      hair_color: 'n/a',
      skin_color: 'gold',
      eye_color: 'yellow',
      birth_year: '112BBY',
      gender: 'n/a',
      homeworld: 'http://swapi.dev/api/planets/1/',
      films: [
        'http://swapi.dev/api/films/1/',
        'http://swapi.dev/api/films/2/',
        'http://swapi.dev/api/films/3/',
        'http://swapi.dev/api/films/4/',
        'http://swapi.dev/api/films/5/',
        'http://swapi.dev/api/films/6/',
      ],
```

```
      species: ['http://swapi.dev/api/species/2/'],
      vehicles: [],
      starships: [],
      created: '2014-12-10T15:10:51.357000Z',
      edited: '2014-12-20T21:17:50.309000Z',
      url: 'http://swapi.dev/api/people/2/',
    },
  ],
};
```