TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

LAURI KOSKELA

AUTOMAATTINEN OHJELMISTOTESTAUS MIKROPALVELUARKKITEHTUURISSA

Kandidaatintyö

Tarkastaja: Tiina Schafeitel-Tähtinen

Jätetty tarkastettavaksi 10. heinäkuuta 2018

TIIVISTELMÄ

LAURI KOSKELA: Automaattinen ohjelmistotestaus mikropalveluarkkitehtuurissa
Tampereen teknillinen yliopisto
Kandidaatintyö, 29 sivua, 1 liitesivu
Heinäkuu 2018
Tietotekniikan tutkinto-ohjelma
Pääaine: Ohjelmistotekniikka
Tarkastaja: Tiina Schafeitel-Tähtinen

Avainsanat: mikropalvelut, automaattitestaus, Docker

Mikropalveluarkkitehtuuri on yleisesti pilviympäristöissä käytetty ohjelmistoarkkitehtuuri, jossa ohjelmisto koostuu useasta pienestä palveluohjelmasta eli mikropalvelusta. Kunkin palvelun vastuualue on kapea, ja palvelut hyödyntävät toisiaan koko ohjelman toiminnallisuuden saavuttamiseksi. Monista erillisistä osista koostuva arkkitehtuuri tuo haasteita myös ohjelmiston testaukseen. Tämän kandidaatintyön tarkoituksena on tutkia mikropalveluohjelmistojen testausmenetelmiä. Työssä keskitytään automaattitesteihin ja esitellään eri tekniikoita, joiden avulla mikropalveluohjelmiston testaus onnistuu tehokkaasti.

Hyvä tapa ohjata automaattitestien suunnittelua on testipyramidimalli, joka kuvaa testien jakamista usealle eri tasoille. Yksikkötestit ja erilaiset integraatiotestit ovat yleisesti käytössä kaikessa ohjelmistokehityksessä ja myös mikropalveluarkkitehtuurissa. Mikropalveluja käytettäessä palvelujen välisten rajapintojen testaaminen on tärkeää. Tämä tehdään usein käyttäen kuluttajavetoisia sopimustestejä. Testeissä pääpaino tulisi olla pienillä ja nopeilla yksikkö- ja integraatiotesteillä. Työssä havaittiin, että haasteet yksinkertaisten testitapauksien toteutuksessa liittyvät ensisijaisesti testiympäristön valmisteluun.

SISÄLLYSLUETTELO

LYHENTEET JA MERKINNÄT

CD	engl. *Continuous Delivery*, jatkuva toimitus
CI	engl. *Continuous Integration*, jatkuva integraatio
DAO	engl. *Data Access Object*, tietokantaa käsittelevä olio
E2E testing	engl. *End-to-end testing*, järjestelmätestaus
gRPC	Googlen kehittämä etäproseduurikutsuprotokolla
HTTP	engl. *Hypertext Transfer Protocol*, yleisesti Internetissä käytetty tiedonsiirtoprotokolla
TDD	engl. *Test Driven Development*, testivetoinen kehitys
UI	engl. *User Interface*, käyttöliittymä

1. JOHDANTO

Mikropalveluarkkitehtuuri on yleistynyt viime vuosina ohjelmistojen toteutusmallina. Sillä tarkoitetaan ohjelmiston rakentamista pienistä itsenäisistä palvelusovelluksista, joista kullakin on oma kapea vastuualueensa. Mikropalvelujen toteutuksessa hyödynnetään usein konttiteknologiaa. Kontit (container) ovat saman käyttöjärjestelmän alaisuudessa ajettavia kevyitä virtuaalikoneita. Niiden avulla mikropalveluita voidaan ajaa toisistaan eristettynä samalla isäntäkoneella. [13][18, s. 125]

Mikropalveluohjelmistojen kehittämiseen liittyy uusia haasteita perinteisten monoliittisten ohjelmistojen kehittämiseen verrattuna. Mikropalvelut muodostavat luonnostaan monimutkaisia kokonaisuuksia, mikä pitää huomioida myös niiden testauksessa. [13] Tämän kandidaatintyön tarkoituksena on tutkia mikropalveluohjelmiston automaattitestausta. Työssä perehdytään eri testaustasoihin ja testityyppeihin sekä niiden merkitykseen mikropalveluiden kannalta. Lisäksi tutkitaan automaattitestien toteutusta konttiteknologiaa hyödyntäen esimerkkitoteutusten kautta.

Luvussa 2 esitellään mikropalveluarkkitehtuuri ja sen mukanaan tuomia haasteita. Luku 3 käsittelee konttiteknologiaa ja Dockeria esimerkkialustana. Luvussa 4 syvennytään testaukseen ja käsitellään eri tapoja mikropalveluiden testaukseen. Testien toteutusta tutkitaan luvussa 5 esimerkkiprojektia hyödyntäen ja soveltaen aikaisempien lukujen käsittelemää teoriaa. Lopuksi yhteenvedossa käsitellään työssä tehdyt johtopäätökset.

2. MIKROPALVELUARKKITEHTUURI

Ohjelmistojen kehityksessä ja käyttöönotossa on perinteisesti noudatettu monoliittista arkkitehtuuria. Se tarkoittaa ohjelmiston rakentamista siten, että lopputuloksena muodostuu yksi käyttöönotettava ohjelmistopaketti tai -kokonaisuus. Tällainen ratkaisu soveltuu hyvin pieniin ohjelmistoihin. Monimutkaisempien projektien kanssa monoliittinen arkkitehtuuri aiheuttaa kuitenkin ongelmia esimerkiksi ohjelmiston skaalautuvuudessa sekä nopeassa ohjelmistokehityksessä. [13][26]

Monoliittisen arkkitehtuurin vaihtoehtona on viime vuosina yleistynyt mikropalveluarkkitehtuuri. Siinä ohjelmisto muodostetaan pienistä itsenäisistä palveluohjelmista, jotka voidaan ottaa käyttöön toisistaan riippumatta. Mikropalveluarkkitehtuuri ratkaisee joitakin suurissa monoliittisissa ohjelmistoissa esiintyviä ongelmia. Sen monimutkaisempi rakenne aiheuttaa kuitenkin haasteita kehitysprosessiin ja muun muassa testaukseen. [18, s. 11]

2.1 Mikropalveluarkkitehtuurin pääpiirteet

Mikropalvelut ovat itsenäisiä palveluohjelmistoja, joilla on kapea vastuualue. Vastuualueet pitäisi jakaa yleensä liiketoiminnan jaon mukaisesti eikä esimerkiksi teknisten ratkaisujen mukaan. Palveluista tulee luonnostaan itsenäisiä ja toisistaan riippumattomia niiden vastuujaon ollessa kehitysorganisaation vastuujaon mukainen [35, s. 41]. Palvelut kommunikoivat keskenään rajapintojen yli tyypillisesti verkon välityksellä. Ne ovat siis löyhästi sidottuja (loosely coupled), ja yhden palvelun muuttamisen ei pitäisi vaatia muutoksia toisiin palveluihin. [13][18, s. 2]

Mikropalvelujen itsenäisyyden ja pienen koon myötä niiden käytöstä saadaan monia hyötyjä. Palveluiden löyhän sidonnan ansiosta niitä voidaan ottaa käyttöön ja kehittää toisistaan riippumatta. Tämä mahdollistaa monoliittiseen arkkitehtuuriin verrattuna ketterämmät ohjelmiston käyttöönotot ja päivitykset. Mikropalvelujen kapeat vastuualueet taas tekevät koodista ymmärrettävämpää ja näin kehityksestä helpompaa. [13] Erään tutkimuksen mukaan juuri koodin ylläpidettävyys ja kehityksen helppous on yksi tärkeimmistä syistä mikropalveluarkkitehtuurin käyttöönottoon [31].

Pienistä osista koostuva arkkitehtuuri auttaa myös vastuunjaossa kehitystiimin sisällä. Kukin kehittäjä tai pieni tiimi voi keskittyä vain omiin palveluihinsa, eikä heidän välttämättä tarvitse koskaan katsoa muiden palveluiden koodia. Mikropalvelujen toteutuskielet ja -menetelmät voidaan valita vapaasti palvelujen tarpeiden tai kehitystiimin osaamisen mukaan, mikä ei ole mahdollista monoliittisessa arkkitehtuurissa. Koska mikropalvelut

kommunikoivat toistensa kanssa verkon yli, ei niiden tarvitse olla toteutettu samalla ohjelmointikielellä. [13][18, s. 4]

Mikropalvelujen käytöstä on hyötyjä myös kehityksen ja käyttöönoton jälkeen. Niitä voidaan skaalata järjestelmän tarpeiden mukaan tarkemmin kuin monoliittista ohjelmistoa. Usein käytetylle mikropalvelulle voidaan antaa enemmän resursseja kuin harvemmin käytetyille palveluille. [35, s. 7] Mikropalvelujärjestelmä voi olla myös vakaampi kuin vastaava monoliittinen järjestelmä: yhden mikropalvelun kaatuminen ei välttämättä vaikuta lainkaan muuhun järjestelmään. Monoliittisessa ohjelmistossa virheillä on helposti suuremmat vaikutukset koko järjestelmän toimintaan. [18, s. 5]

2.2 Mikropalveluarkkitehtuurin haasteet

Hajautetun ohjelmiston kehityksessä korostuvat monet haasteet monoliittiseen ohjelmistoon verrattuna. Mikropalveluja voi olla ohjelmistossa kymmeniä, ja jotkin ongelmat voivat toistua kussakin palvelussa erikseen. Erilaisiin virhetilanteisiin tulee varautua siten, että yhden palvelun vika ei lamauta koko järjestelmää. Samoin toimivan monitorointiratkaisun merkitys on suuri, jotta virheet löydetään ja voidaan korjata tehokkaasti. [9][13][35, s. 77]

Edellä mainittujen hyötyjen saavuttaminen vaatii myös ohjelmistokehitysprosessin mukautumista mikropalveluarkkitehtuuriin sopivaksi. Monien mahdollisesti paljonkin toisistaan poikkeavien palvelujen kehitys, käyttöönotto ja hallinta manuaalisesti on monimutkaista ja aikaa vievää. Automatisoinnin tehokas käyttö onkin olennainen osa mikropalveluarkkitehtuuria noudattavan ohjelmiston kehitystä. [18, s. 246][19]

Mikropalveluarkkitehtuurin monimutkaisuuden myötä myös ohjelmiston testaus on monimutkaisempaa. Testauksella voidaan varmistaa, että palvelut kommunikoivat keskenään oikein ja että niihin tehdyt muutokset eivät riko ohjelmiston toimintaa kokonaisuutena. Myös testauksessa automatisoinnin tarve on suuri, sillä koko hajautetun järjestelmän testaus manuaalisesti kehitysprosessin osana voi olla erittäin hankalaa. Testiautomaation myötä kehittäjät voivat keskittyä yhden palvelun muutoksiin luottaen, että sen toiminta muun järjestelmän osana varmistetaan automaattitestien avulla. [16, s. 55][35, s. 213]

3. KONTTITEKNOLOGIA JA DOCKER

Kontit (container) ovat tapa virtualisoida ohjelmistoja yhden käyttöjärjestelmäytimen (kernel) alaisuudessa. Ne mahdollistavat ohjelmien ja niiden tarvitsemien riippuvuuksien paketoinnin yhteen ja näin ohjelmien ajamisen toisistaan riippumatta. [29][33, s. 4] Tässä luvussa esitellään Docker-konttialusta ja konttien käyttöä sen avulla. Lisäksi tutkitaan hyötyjä, joita Docker tuo mikropalveluohjelmiston kehitykseen ja ajamiseen.

Konttiteknologia on kevyempi virtualisointiratkaisu kuin perinteinen *hypervisor*-pohjainen virtualisointi. Jälkimmäisessä isäntäkäyttöjärjestelmän (host operating system) alaisuudessa ajetaan kokonaista vieraskäyttöjärjestelmää (guest operating system). [32, s. 70] Kontteja sen sijaan ajetaan suoraan isäntäkäyttöjärjestelmällä, joten niiden avulla resursseja voidaan hyödyntää tehokkaammin. Samasta syystä ne ovat tavallisiin virtuaalikoneisiin verrattuna usein moninkertaisesti pienempiä ja nopeampia käynnistää. Kokonaisen käyttöjärjestelmän käynnistämisen sijaan konteille riittää usein vain yhden prosessin käynnistäminen. [13][33, s. 4][35, s. 262]

Linux-käyttöjärjestelmässä kontit on toteutettu käyttöjärjestelmäytimen tarjoamien nimiavaruuksien avulla. Nimiavaruudet tarjoavat tavan eristää prosesseja muusta järjestelmästä. [5] Näin kukin kontti näkyy siinä ajettavalle prosessille omana käyttöjärjestelmänään, jota voi muokata vapaasti ilman vaikutuksia muihin kontteihin tai koko järjestelmään [18, s. 125].

3.1 Docker

Docker on yleisesti käytetty Linux-pohjainen konttialusta. Se tarjoaa ratkaisun konttien luontiin, ajamiseen, jakeluun ja hallintaan.

Kontit ovat Dockerissa instansseja *imageista*. Docker-image sisältää kaikki sovelluksen ajamiseen tarvittavat kirjastot sekä sovelluksen itse, ja se on muuttumaton luomisensa jälkeen. Imaget rakennetaan *Dockerfile*-tiedoston perusteella. Se sisältää joukon käskyjä, joiden perusteella imageen esimerkiksi asennetaan ohjelmistoja tai määritellään sen konfiguraatio. Dockeria käytettäessä luodaan tyypillisesti jokaista ajettavaa sovellusta tai palvelua varten oma imagensa. [13][33, s. 7]

Sovellukset tai palvelut ajetaan luomalla imagen pohjalta kontti. Yksi kontti vastaa yhtä suorituksessa olevaa sovelluksen instanssia. Koska imaget ovat muuttumattomia, ja konttien sisäiset muutokset tiedostojärjestelmään eivät vaikuta niihin, voidaan saman imagen pohjalta luoda useita kontteja ajoon samaan aikaan. [33, s. 9]

Docker sisältää joitakin työkaluja konttiorkestrointiin (container orchestration) eli useiden konttien ajamisen hallintaan. Tätä varten on myös useita kolmannen osapuolen projekteja, kuten alun perin Googlen kehittämä Kubernetes [15] ja Mesosphere DC/OS [4].

Yksinkertaisiin monen kontin sovelluksiin, kuten paikallisen kehitys- ja testausympäristön rakentamiseen soveltuu Docker Compose. Se on työkalu, jolla voi hallita useiden konttien ajamista määrittelemällä ne yhteen tiedostoon. Näin esimerkiksi paikallisessa kehityksessä yhdellä komennolla voi ajaa sekä kehitettävää ohjelmistoa että sen riippuvuuksia, kuten tietokantaa tai muita palveluita. [20]

3.2 Konttiteknologian hyödyt

Koska kontit sisältävät koko tarvitsemansa ympäristön, ne ovat riippumattomia alustasta, jolla niitä ajetaan. Tämän vuoksi ne ovat helposti siirrettävissä ympäristöstä toiseen. Konttiteknologian käyttö vähentää myös kehitys- ja tuotantoympäristön eroista johtuvia ongelmia. Periaatteessa kontit toimivat identtisesti millä tahansa palvelimella tai tietokoneella, jossa konttialusta kuten Docker on saatavilla. [3]

Kontit ja Docker soveltuvat erittäin hyvin mikropalvelujen ajamiseen. Keveytensä ansiosta niitä voidaan ajaa tehokkaasti rinnakkain samoilla palvelimilla. Samoin mikropalvelujen skaalaus on helppoa ja nopeaa kontteja lisäämällä. Toisaalta Docker mahdollistaa palvelujen erilaiset toteutustekniikat, koska palvelun kaikki riippuvuudet sisältyvät Docker-imageen. [35, s. 264]

4. MIKROPALVELUJEN TESTAUS

Ohjelmistotestauksen tarkoituksena on paljastaa virheitä ohjelmistojen toiminnassa ja varmistaa ohjelmiston laatu ja vaatimusten mukainen toiminta. [25, s. 466] Testaus on oleellinen osa ohjelmistotuotantoprosesseja, ja testaustavat kehittyvät kehitysprosessien ja -teknologioiden kehittyessä. [34]

Mikropalveluohjelmistojen testaus on tärkeää aivan kuten muunkin ohjelmiston testaus. Mikropalvelujen pienistä osista koostuvan rakenteen vuoksi testaukseen tarvitaan uusia ratkaisuja, jotta testeillä voidaan löytää virheet sekä yksittäisistä palveluista että niiden toiminnasta kokonaisuutena. [16, s. 54]

Tässä luvussa esitellään automaattitestauksen menetelmiä ja tutkitaan mikropalvelujen testaamiseen soveltuvia testityyppejä. Lisäksi esitellään testien suunnittelussa ja mitoituksessa hyödyllinen testipyramidimalli.

4.1 Automaattitestaus

Ohjelmistotestaus voidaan tehdä manuaalisesti siten, että testaaja tai kehittäjä käy läpi määritellyt testitapaukset ja näin varmentaa ohjelmiston toimivuuden. Tällainen testaus on kuitenkin kallista ja hidasta. Tämän vuoksi testauksen automatisointi on keskeistä nykyaikaisessa ohjelmistokehityksessä. [10][24][34]

Automaattitestaus tarkoittaa testitapauksien ja testien tulosten tarkistuksen määrittelyä ohjelmaksi siten, että ne voidaan ajaa automaattisesti ilman ihmisten valvontaa [10]. Automatisointi mahdollistaa testauksen ottamisen olennaiseksi osaksi ohjelmistokehitysprosessia, ja se onkin yleistynyt erityisesti ketterien ohjelmistokehitysmenetelmien mukana. Testaus ei usein ole erillinen vaihe ketteriä menetelmiä käytettäessä, vaan sitä tehdään osana itse kehitystä. Testauksen automatisointi mahdollistaa nopeamman ohjelmistokehityksen, kun kehittäjien ei tarvitse käyttää aikaa ohjelmiston testaamiseen manuaalisesti. [2, s. 308]

Eräänlainen ääritapaus testiautomaatiosta on testivetoinen kehitys (test-driven development, TDD). Se tarkoittaa ohjelmistokehitysmenetelmää, jossa automaattitestitapaukset luodaan ennen varsinaista toteutusta. Kehittäjä ajaa testejä toteutuksen yhteydessä, ja näin saa jatkuvaa palautetta toteutuksen oikeellisuudesta. [30] Testivetoisen kehityksen pääperiaatteita on se, että muutoksia toteutukseen tehdään vain, jos jonkin testitapauksen läpäisy niitä vaatii. Käytännössä tämä tarkoittaa sitä, että ennen virheiden korjausta tai uusien ominaisuuksien toteutusta kirjoitetaan testitapaukset, jotka kyseistä asiaa testaavat. Näin varmistetaan myös se, että testitapauksia ei jää tekemättä millekään osalle ohjelmistoa. [35, s. 215]

Jatkuva integraatio (continuous integration, CI) on menetelmä, jossa ohjelmistoa koostetaan jatkuvasti jo kehityksen aikana. Käytännössä tämä tarkoittaa esimerkiksi sitä, että jokaisen versionhallintajärjestelmään siirretyn koodimuutoksen jälkeen koko ohjelmisto koostetaan. [27] Samoin usein jokaisen koodimuutoksen seurauksena ajetaan ohjelmiston automaattitestit. Näin kehittäjät saavat jatkuvasti palautetta ohjelmiston toimivuudesta ja muutosten vaikutuksista. [30]

Jatkuvasta integraatiosta jalostettuja menetelmiä ovat jatkuva toimitus (continuous delivery, CD) sekä jatkuva käyttöönotto (continuous deployment). Jatkuvalla toimituksella tarkoitetaan sitä, että jatkuvan integroinnin lisäksi ohjelmisto valmistellaan jatkuvasti ja automaattisesti käyttöönotettavaksi tuotantoympäristössä. Tämä voi tarkoittaa esimerkiksi monimutkaisempien automaattitestien ajamista sekä asennuspakettien luomista ohjelmistosta. Jatkuvassa käyttöönotossa ohjelmisto otetaan automaattisesti käyttöön tuotantoympäristössä, mikäli se läpäisee automaattitestit. Testien merkitys on siis näissä menetelmissä suuri, sillä ne voivat olla ainoa ohjelmiston toiminnan varmistus ennen julkaisua. [27]

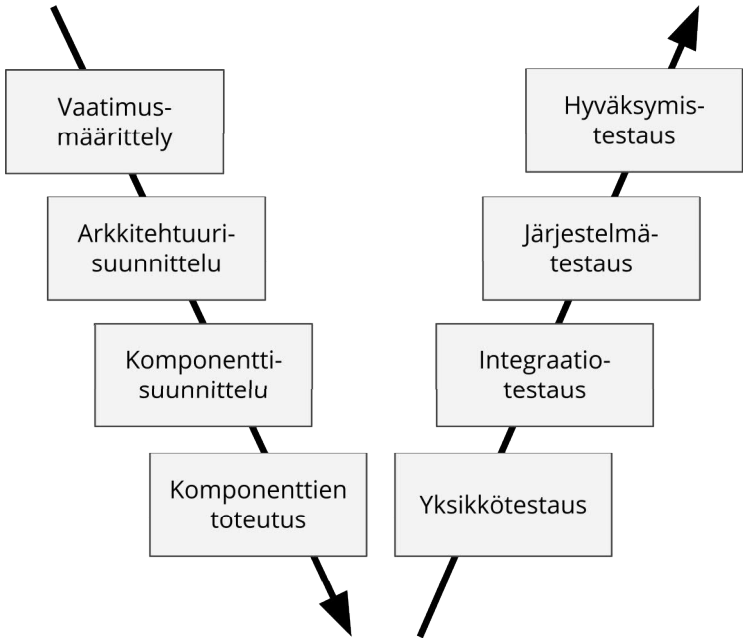
Kuten edellä on todettu, automaatio on tärkeää mikropalvelujen kehityksessä, ja toimiva automaatio vaatii kattavat automaattitestit. Jotta mikropalveluita voidaan kehittää ja ottaa käyttöön nopeasti, pitää automaattitestien olla riittävät varmentamaan ohjelmiston toiminnallisuus ennen käyttöönottoa. Tyypillisesti jatkuva integraatio on olennainen osa mikropalvelujen kehitysprosessia, sillä yksittäiset kehittäjät tai kehitystiimit saattavat hallita vain yhden palvelun muutoksia. Jatkuvan integraation avulla ajettavat automaattitestit antavat kaikille kehittäjille palautetta mahdollisista virheistä, joita yhteen palveluun tehdyt muutokset voivat aiheuttaa koko järjestelmässä. [9]

4.2 Testaustasot ja -tyypit

Ohjelmistokehityksessä aikaisemmin yleisen vesiputouskehitysmallin yhteydessä käytettiin testauksen V-mallia. Vesiputousmalli korostaa tarkkoja suunnitelmia ja kehitysvaiheita. V-mallissa suunnitelmia ja kehitysvaiheita vastaavat eri testauksen vaiheet. [25, s. 42] V-malli on esitetty kuvassa 1. Vaikka vesiputousmallin merkitys on vähentynyt ketterien ohjelmistokehitysmenetelmien suosion kasvaessa, ovat V-mallin mukaiset testausvaiheet edelleen yleisesti käytössä kaikessa ohjelmistokehityksessä [14].

V-mallin käsitteet, kuten yksikkö- ja integraatiotestaus, voivat testauksen vaiheiden lisäksi tarkoittaa myös testauksen tasoja tai tyyppejä. Ne tarkoittavat testien jakamista testattavan toiminnallisuuden perusteella. Usein tämä jako tehdään ohjelmiston komponenttien laajuuden mukaan. Testauksen tasojen käsite kuvaa tätä jakoa: matalan tason testeillä kuvataan ohjelmiston pienimpiä komponentteja varmentavia testejä, kun taas korkean tason testit testaavat komponenttien toimintaa yhdessä.

Testauksen tasoilla ja tyypeillä tarkoitetaan tässä työssä teknisiä ratkaisuja, joilla ohjelmiston testausta voidaan tehdä mahdollisimman nopeasti ja tehokkaasti. Testityyppejä tutkitaan erityisesti mikropalveluohjelmiston testauksen kannalta, mutta ne soveltuvat myös



Kuva 1. *Testauksen V-malli, perustuu lähteeseen [25, s. 43]*

muita arkkitehtuureita käyttävien ohjelmistojen testaukseen. Eri testityyppejä on esitelty alla.

4.2.1 Yksikkötestaus

Yksikkötestauksella tarkoitetaan ohjelmiston pienimpien komponenttien toiminnan testaamista. Nämä komponentit eli yksiköt voivat olla esimerkiksi funktioita tai luokkia riippuen ohjelmiston rakenteesta ja kehitystiimin käytännöistä. [1][34]

Koska yksikkötestien tarkoitus on testata vain pientä osaa ohjelmistosta, käytetään niiden kanssa usein testisijaisia, kuten tynkä- ja *mock*-olioita. Ne ovat tapa tarjota testattavalle yksikölle testikoodin hallitsema korvaava toteutus ulkoisista riippuvuuksista. Tynkäoliot (stub) ovat ulkoisten rajapintojen mahdollisimman yksinkertaisia toteutuksia. Ne palauttavat aina jonkin vakioidun vastauksen, eivätkä sisällä mitään oikeaa toiminnallisuutta. Mock-olioiden toiminta määritellään ennen testien ajamista. Niiden avulla voidaan myös varmistaa, että testattava koodi todella käyttää ulkoisia rajapintoja oikein. [6][34]

Yksikkötestit ovat suoritusnopeudeltaan nopeimpia ohjelmistotestejä. Edellä mainittujen testisijaisten ansiosta ne eivät tee hitaita kutsuja ulkoisiin palveluihin, ja niitä voidaan ajaa rinnakkain toisistaan riippumatta. Pienen vastuualueensa ansiosta ne myös antavat kehittäjille melko tarkkaa tietoa virheiden sijainnista koodissa. [2, s. 311][18, s. 134][34]

Yksikkötestit ovat yleinen työkalu testivetoista kehitystä käytettäessä [18, s. 134]. Ennen funktion tai luokan toteutusta sille kirjoitetaan yksikkötestit. Testit samalla määrittelevät, miten yksikön toteutuksen tulisi käyttäytyä. Vasta tämän jälkeen yksikölle tehdään testit läpäisevä toteutus. Koska yksikkötestit ovat nopeita, voivat kehittäjät ajaa niitä jatkuvasti toteutusta tehdessään, ja näin saada jatkuvaa palautetta toteutuksen oikeellisuudesta. [23]

Yksikkötestaus on tärkeää myös mikropalveluohjelmistoille. Yksittäiset mikropalvelut koostuvat pienistä yksiköistä, joita testataan aivan samoin kuin suuremmissa ohjelmistoissa. Pelkillä yksikkötesteillä ei kuitenkaan voida varmentaa mikropalvelujen toimintaa yhdessä. Tämän ongelman ratkaisuun tarvitaan korkeamman tason testausta, jota käsitellään seuraavaksi. [1]

4.2.2 Komponenttitestaus

Komponenttitestaus tarkoittaa ohjelmiston yksittäisen komponentin testaamista kokonaisuutena. Tässä komponentilla tarkoitetaan yleensä jotakin ohjelmistomoduulia, luokkien kokonaisuutta tai koko palveluohjelmaa, joka tarjoaa muulle ohjelmistolle palveluita. Komponentit muodostuvat edellä käsitellyistä pienistä yksiköistä. [2, s. 313][16, s. 52][18, s. 134] Komponenttitestausta kutsutaan myös palvelutestaukseksi [18, s. 134][34].

Komponenttitestaus on eräs integraatiotestauksen muoto. Integraatiotestauksella tarkoitetaan useiden ohjelmistomoduulien yhdessä toimimisen testausta. Vaikka moduulit toimisivat itsenäisesti virheittä, eivät ne välttämättä tuota yhdistettynä haluttua lopputulosta. Komponenttitestit varmentavat komponentin toiminnan kokonaisuutena, kun taas yksikkötestit testaavat komponentin osien toiminnan erikseen. [16, s. 52][35, s. 216] Testauksen V-mallin integraatiotestit viittaavat tyypillisesti komponenttitestien kaltaisiin testeihin, sillä niiden tarkoitus on testata juuri komponenttitason suunnittelua. [25, s. 475]

Mikropalvelujen tapauksessa komponenttitestaus kohdistuu yleensä yhteen mikropalveluun kokonaisuutena. Palvelua ajetaan erillisenä prosessina, ja sen tarvitsemat ulkoiset rajapinnat tarjotaan testisijaisten avulla. Testit kommunikoivat mikropalvelun kanssa sen julkisen rajapinnan kautta. Komponenttitestauksella varmennetaan, että yksikkötesteillä testattu toteutus muodostaa määrittelyn mukaisen ja toimivan mikropalvelun. [1][16, s. 52]

Mikropalvelujen komponenttitestejä automatisoitaessa on mahdollista hyödyntää konttiteknologiaa. Kun testaus suoritetaan kontissa ajettavaa palvelua vasten, on testattava palvelu samanlainen kuin tuotantoympäristössä, ja testien ajoympäristö ei vaikuta palvelun toimintaan. Testit voivat käyttää kontteja myös mikropalvelun tarvitsemien rajapintojen, kuten tietokantojen tarjoamiseen. Näin testiympäristön luonti voidaan tehdä osana testikoodia. [12]

4.2.3 Kapea integraatiotestaus

Integraatiotestauksella voidaan tarkoittaa myös ohjelmiston moduulien integraatiokohtien testaamista mahdollisimman kapealla alueella. Tällaisia testejä kutsutaan kapeiksi integraatiotesteiksi. Mikropalvelujen testauksessa ne voivat tarkoittaa esimerkiksi muiden mikropalvelujen tai tietokannan kanssa kommunikoivien luokkien testaamista. [1][8][16, s. 42][34]

Kapeat integraatiotestit muistuttavat yksikkötestejä. Ne testaavat ohjelmistojen pieniä osia kuten luokkia, jotka tässä tapauksessa tarvitsevat jotain ohjelman ulkoista riippuvuutta. Integraatiotestit tarjoavat nämä riippuvuudet hallitusti testattavalle yksikölle ja varmistavat, että yksikkö käsittelee niitä oikein. Ulkoisten riippuvuuksien takia integraatiotestit ovat kuitenkin usein paljon hitaampia kuin yksikkötestit. Ulkoisten palveluiden hallinta tekee niistä myös monimutkaisempia toteuttaa. [34]

Kuten yksikkötestit ja komponenttitestit, myös kapeat integraatiotestit hyödyntävät testisijaisia. Sijaiset korvaavat ulkoiset palvelut, joiden integraatiota testataan. Näin testit ovat riippumattomia ulkoisen palvelun toiminnasta ja varmentavat vain integraatiokohtien toiminnan. Kapeaa integraatiotestausta voi tehdä myös oikeaa ulkoista palvelua hyödyntäen, mikäli se on testien hallittavissa. Tämä voi olla hyödyllistä esimerkiksi tietokantaintegraatiota testattaessa. [1][8][34]

Myös kapeat integraatiotestit voivat hyödyntää konttiteknologiaa samoin kuin edellä kuvatut komponenttitestit. Testikoodi voi konttien avulla ajaa ulkoisia palveluita, joiden kanssa integraatiota testataan. [12] Mikropalvelujen tapauksessa testejä varten voidaan muodostaa kontti, jossa ajetaan tynkäversiota palvelusta. Muut palvelut voivat käyttää tätä tynkäversiota, kun ne ajavat omia integraatiotestejään.

4.2.4 Sopimustestaus

Sopimustestaus on menetelmä, jossa ohjelmiston osien väliset rajapinnat määritellään sopimuksina. Sopimustestit varmentavat, että sekä rajapinnan tarjoajat että sen käyttäjät noudattavat sopimusta. Kuluttajavetoisessa sopimustestauksessa (consumer-driven contract testing) rajapinnan kuluttaja (consumer) eli rajapintaa käyttävä palvelu määrittelee testitapaukset sopimusta varten. Nämä testit käsittelevät rajapintaa samalla tavalla kuin kuluttaja itse tekee. Rajapinnan tarjoajaa (producer) kehitettäessä ajetaan kaikkien sen kuluttajien määrittelemät sopimustestit. Jos sopimustestit ovat kattavat ja vastaavat kuluttajien oikeaa toimintaa, testaavat ne kaikkia rajapinnan osia, joita jokin kuluttaja käyttää. [1][16, s. 53][18, s. 144][34][35, s. 231]

Mikropalvelujen välinen kommunikointi ja niiden välisten rajapintojen käyttö on koko ohjelmiston toimivuuden kannalta tärkeää. Tämän vuoksi sopimustestaus soveltuu erittäin hyvin mikropalveluohjelmistojen testaamiseen. Se esitelläänkin monissa mikropalvelujen testausta käsittelevissä lähteissä. [1][16, s. 53][18, s. 144][35, s. 231] Edellä kuvatut komponenttitestaus ja kapea integraatiotestaus varmentavat, että mikropalvelu käyttää ulkoisia palveluja oikein. Testaus tehdään usein kuitenkin testisijaisia vasten. Testikoodi luo sijaiset itse, ja ne eivät siis välttämättä täysin vastaa oikeita ulkoisia palveluja. Oikeiden rajapintojen testaus voidaan tehdä sopimustesteillä. [7]

Sopimustestauksella voidaan varmistaa, että rajapinnan yhteensopivuus säilyy, kun tarjoajaan tehdään muutoksia. Käytännössä testit varmistavat, että esimerkiksi rajapintakutsujen vastausviestissä on kaikki kuluttajan tarvitsemat kentät ja että tarvittavat operaatiot

ovat saatavilla kuluttajalle. [35, s. 230] Sopimustestit eivät testaa rajapinnan tarjoajien tai kuluttajien varsinaista toiminnallisuutta. Toiminnallisuus tulisi testata muilla testityypeillä kuten yksikkö- ja komponenttitesteillä. [1]

4.2.5 Järjestelmätestaus

Järjestelmätestit (system tests, end-to-end tests) testaavat kokonaisen järjestelmän toimintaa. Niissä testattava ohjelmisto ja kaikki sen riippuvuudet ovat kokonaan suorituksessa. Mikropalveluohjelmiston kohdalla tämä tarkoittaa sitä, että kaikkia mikropalveluja suoritetaan samaan aikaan. Järjestelmätestit käsittelevät ohjelmistoa sen julkisten rajapintojen tai käyttöliittymän välityksellä ja varmistavat, että mikropalvelut muodostavat kokonaisuutena toimivan ohjelmiston. [1][16, s. 53][18, s. 135]

Edellä kuvatut testityypit testaavat mikropalveluiden toimintaa kattavasti sekä erikseen että integroituna toistensa kanssa. Ne eivät kuitenkaan varmenna ohjelmiston toimintaa oikeassa ympäristössä. Järjestelmätesteillä voidaan varmistaa esimerkiksi mikropalvelujen verkkokonfiguraation oikeellisuus. [16, s. 53]

Jotkin lähteet käyttävät järjestelmätestien sijasta käyttöliittymätestien (UI tests) käsitettä. [2, s. 312] Nämä ovat jokseenkin rinnasteisia testityyppejä, sillä molempien tarkoituksena on testata järjestelmää mahdollisimman kattavasti kokonaisuutena. Käyttöliittymät ovat nykyään usein erillisiä muusta järjestelmästä, ja niitä voidaan testata erikseen yksikkö- ja integraatiotesteillä. [34] Tämän vuoksi järjestelmätestauksen käsite on suosittu uudemmissa lähteissä. Se kuvaa vain testien vastuualuetta eikä ota kantaa siihen, testataanko järjestelmää käyttöliittymän kautta. [1][34]

Järjestelmätesteihin liittyy monia ongelmia. Koska niitä varten tarvitaan kokonainen testiympäristö, niiden ajaminen on monimutkaisempaa kuin pienemmän laajuuden testien ajaminen. Monimutkaisen ympäristön myötä myös testitapauksiin liittymättömien virheiden mahdollisuus kasvaa. Vika testiympäristössä tai verkkoyhteyksissä voi aiheuttaa testien epäonnistumisen ohjelmiston toiminnasta riippumatta. Testiympäristön vaatimukset tekevät järjestelmätesteistä lisäksi kalliita ja vaikeita ylläpitää. [18, s. 140][34]

Järjestelmätestien suoritus on usein hidasta. Verkkokutsut, käyttöliittymän ohjelmallinen käyttö ja monimutkaiset valmistelut voivat saada yhden testitapauksen viemään useita sekunteja. Jos järjestelmätestejä on paljon, voi niiden suoritus olla niin hidasta, että niitä ei voida ajaa kehityksen ohella. [18, s. 141][34][35, s. 217] Hitaat testit hidastavat myös ohjelmiston kehitystä ja käyttöönottoa. Mikäli kaikki testit halutaan ajaa ennen ohjelmiston uuden version käyttöönottoa, voivat hitaat järjestelmätestit muodostua pullonkaulaksi ja estää nopeat ohjelmistopäivitykset. [18, s. 142][35, s. 224]

Mikropalveluarkkitehtuuria noudattavien ohjelmistoprojektien järjestelmätesteille tyypillistä on se, että niiden tekeminen ei ole varsinaisesti minkään kehitystiimin vastuulla.

Mikropalveluiden vastuujako on selvä kehittäjien ja kehitystiimien välillä. Koska järjestelmätestit testaavat koko järjestelmää ja näin kaikkia palveluita, ei kukaan välttämättä ole vastuussa niiden ylläpidosta. [18, s. 141][34]

Lukuisten ongelmien vuoksi järjestelmätestien rooli automaattitesteissä tulisi olla mahdollisimman pieni. Monet lähteet [1][2, s. 312][16, s. 54][18, s. 143][34][35, s. 224] suosittelevat käyttämään vain muutamia järjestelmätestitapauksia, jotka varmistavat tärkeimpien ohjelmiston ominaisuuksien toiminnan. Ohjelmiston toiminnallisuus tulisi ensisijaisesti varmistaa pienemmillä ja nopeammilla testeillä. [34]

4.2.6 Savutestaus

Savutestit (smoke tests) ovat nopeita ja yksinkertaisia testejä, jotka testaavat ohjelmiston perustoiminnallisuutta. Niiden tarkoituksena on paljastaa virheitä, jotka tekevät ohjelmistosta täysin käyttökelvottoman. [25, s. 479] Savutestien tarkoitus ei ole testata koko ohjelmiston toimivuutta, vaan pikemminkin tarkistaa, että ohjelmisto on edes jollain tasolla suorituksessa. Kun savutestit ovat varmentaneet tärkeimpien ominaisuuksien toiminnan, voi muun ohjelmiston toimintaa testata hitaammilla testeillä. [17]

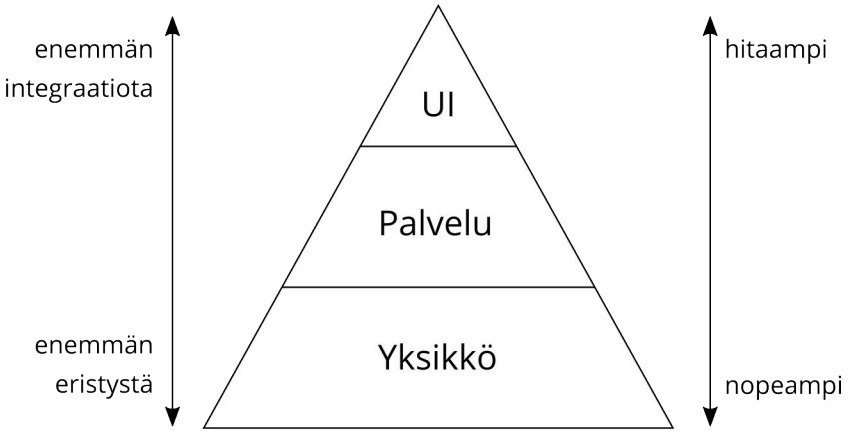
Savutestausta voi tehdä esimerkiksi välittömästi ohjelmiston käyttöönoton jälkeen. Savutestit voivat paljastaa virheitä ohjelmiston konfiguroinnissa tai ympäristössä. Käyttöönoton yhteydessä ajettaessa savutestit antavat heti varmuuden ohjelmiston osittaisesta toiminnasta. Toisaalta savutestien epäonnistuessa voidaan ohjelmistosta palauttaa heti käyttöön edellinen versio. [18, s. 148]

4.2.7 Ei-toiminnallinen testaus

Kaikki edellä kuvatut testityypit testaavat ohjelmiston toimintaa eri tasoilla. Automaattitestaus voi kuitenkin kohdistua myös ohjelmiston ei-toiminnallisiin vaatimuksiin, kuten suorituskykyyn ja kuormituksen kestävyyteen. Tällaista testausta kutsutaan ei-toiminnalliseksi testaukseksi (nonfunctional testing). Tehokkuuden testaamisen lisäksi ei-toiminnallinen testaus voi keskittyä esimerkiksi ohjelmiston käytettävyyteen tai tietoturvaan. [18, s. 151]

Mikropalvelujen tapauksessa erityisesti suorituskykytestaukseen tulee kiinnittää huomiota. Usean mikropalvelun yhteistoiminnasta muodostuva toiminnallisuus voi hidastua palvelujen välisten verkkokutsujen määrän kasvaessa. Automaattisilla suorituskykytesteillä voidaan havaita ohjelmiston muutoksista johtuva suorituskyvyn aleneminen. Suorituskykytestausta voidaan tehdä sekä koko järjestelmälle että yksittäisille palveluille. Näin voidaan esimerkiksi varmentaa tehokkuuskriittisen palvelun riittävä suorituskyky. [16, s. 54][18, s. 152]

Kuormitustestauksella testataan ohjelmiston toimintaa suuren kuormituksen alla. Tämä on luonnostaan automatisoitava testaustyyppi, sillä usein suuri kuormitus on helpointa saavuttaa luomalla keinotekoisesti suuri määrä kutsuja palvelun rajapintoihin. [35, s. 217]



***Kuva 2**. Testipyramidi, perustuu lähteisiin [2, s. 312] ja [34]*

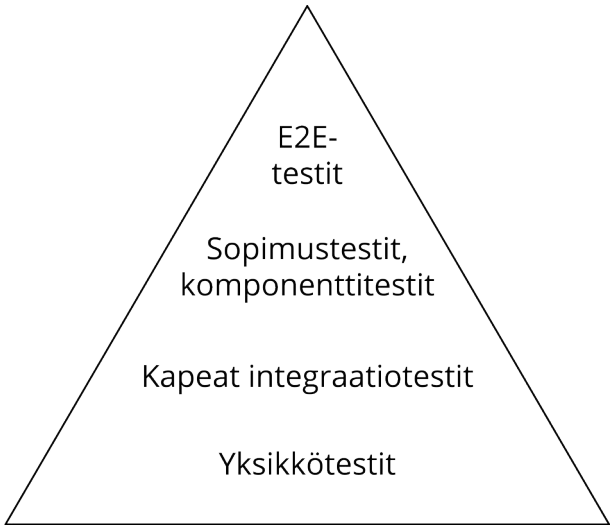
Yhteistä kaikille ei-toiminnallisille testityypeille on riippuvuus ympäristöstä. Testiympäristön tulisi vastata mahdollisimman tarkasti ohjelmiston tuotantoympäristöä, jotta testien tulokset vastaisivat tuloksia tuotantoympäristössä. Poikkeavat ympäristöt voivat johtaa suuriin eroihin esimerkiksi suorituskykytesteissä. [16, s. 54]

4.3 Testipyramidi

Testipyramidi on Mike Cohnin kirjassaan *Succeeding With Agile* [2] esittelemä malli automaattitestien suunnitteluun. Se kuvaa eri testauksen tasoja, jotka testien automatisoinnissa tulee huomioida, sekä niiden keskinäistä painotusta. Cohnin testipyramidi on esitetty kuvassa 2.

Automaattitestit on testipyramidissa jaettu kolmelle tasolle. Pyramidin perustana on Cohnin mukaan yksikkötestaus. Yksikkötestit ovat nopeita kirjoittaa ja antavat kehittäjille tarkkaa tietoa virheiden sijainnista. Tämän vuoksi niiden osuus testipyramidissa on suurin. Yksikkötestien yläpuolella on palvelutestaus, jolla Cohn tarkoittaa joko yksittäisten palveluohjelmien tai ohjelmiston sisäisten palvelujen testaamista kokonaisuutena. Tämä vastaa aikaisemmin käsiteltyä komponenttitestausta. Pyramidissa pienin osuus on käyttöliittymätesteillä (UI, user interface tests). Ne ovat Cohnin mukaan hitaita sekä luoda että ajaa, ja voivat hajota helposti pieniinkin muutoksiin käyttöliittymässä. Näiden ongelmien välttämiseksi ohjelmiston toiminnallisuuden testaus tulisi tehdä alemmilla tasoilla, ja käyttöliittymätesteillä tulisi varmentaa vain käyttöliittymän toimivuus. [2, s. 312]

Testipyramidin perusajatus ei välttämättä ole tarkka testauksen tasojen jako, vaan enemmänkin pienten ja nopeiden testien painotus korkeamman tason testeihin verrattuna [34]. Useat lähteet mainitsevat Cohnin testipyramidin olevan vanhentunut ja epäselvä tasojen nimeämisen ja niiden vastuualueiden osalta [18, s. 133][34], ja testipyramidista onkin tehty useita modernimpia versioita. Monissa lähteissä käyttöliittymätestaus on korvattu koko järjestelmän toimintaa testaavalla järjestelmätestauksella [1][18, s. 133][34]. Myös palvelutestaustaso on jaettu usein pienempiin osiin, kuten kapeisiin integraatiotesteihin [1][34],



Kuva 3. *Mikropalvelujen testaukseen mukautettu testipyramidi, perustuu lähteisiin [1] ja [18, s. 145]*

komponenttitesteihin [1] ja sopimustesteihin [1][18, s. 145][34].

Toby Clemsonin [1] ja Sam Newmanin [18, s. 145] esittelemissä mikropalveluille soveltuvissa testipyramidimalleissa edellä käsitellyt testityypit ovat laajemmin sisällytettynä. Clemsonin mallissa kapeat integraatiotestit ja komponenttitestit muodostavat omat tasonsa pyramidissa. Newmanin mallissa sopimustestit sisältyvät palvelutestauksen kanssa samalle tasolle. Molemmissa malleissa yksikkötestit ovat edelleen pyramidin pohjalla ja järjestelmätestit sen huipulla.

Mikropalvelujärjestelmiä varten mukautettu versio testipyramidimallista on esitetty kuvassa 3. Järjestelmätestit on kuvassa merkitty lyhenteellä E2E (end-to-end). Pyramidista selviää edellä käsiteltyjen testityyppien painotukset automaattitestejä suunniteltaessa. Matalan tason ja pienen vastuualueen testit ovat pyramidissa alempana, ja niiden määrä on siis suurempi. Vastaavasti korkean tason ja suurempien vastuualueiden testit ovat pyramidissa korkeammalla, ja niiden määrä on pienempi.

Ei-funktionaalista testausta ja savutestausta ei ole merkitty kuvan 3 pyramidiin. Suorituskykytestausta ja muuta ei-funktionaalista testausta voi tehdä kaikilla pyramidin tasoilla, joten sitä ei voi suoraan sijoittaa mihinkään kohtaan pyramidia. Savutestaus puolestaan ei varsinaisesti testaa ohjelmiston toiminnallisuutta, joten se on jätetty pois pyramidista.

Testipyramidimallin ylläpitämiseksi hyvä nyrkkisääntö on se, että kukin ohjelmiston toiminnallisuus testataan vain matalimmalla mahdollisella tasolla. Esimerkiksi yhden operaation eri virhetilanteita ei testata komponenttitesteillä vaan nopeilla yksikkötesteillä. Komponenttitestit voivat varmentaa esimerkiksi operaation yhteydessä tehtävän kommunikoinnin eri palvelujen välillä, jos sitä ei voi yksikkötesteillä testata. [34]

5. TESTIEN ESIMERKKITOTEUTUKSET

Tässä luvussa tehdään esimerkkitoteutukset luvussa 4 esitellyille mikropalvelujen testityypeille. Esimerkkitoteutusten tarkoituksena on toimia yksinkertaisina malleina, joita voi hyödyntää ja soveltaa testien toteuttamisessa laajemmin.

5.1 Kuvaus järjestelmästä

Testattavana järjestelmänä käytetään kuluttajille suunnatun mobiilisovelluksen taustapalvelua, joka on toteutettu mikropalveluarkkitehtuuria noudattaen. Esimerkkitestit tehdään yhdelle mikropalvelulle, joka vastaa mobiilisovellukseen lähetettävien markkinointiviestien hallinnasta. Palvelua nimitetään tässä työssä *AppMessagingServiceksi*.

Testattavalla mikropalvelulla on ulkoiset riippuvuudet PostgreSQL-tietokantaan sekä yhteen ulkopuoliseen mikropalveluun. Tietokantaan palvelu tallentaa markkinointiviestit, joita sen kautta on luotu. Ulkopuolisen *PushNotificationServicen* välityksellä AppMessagingService lähettää push-ilmoituksia käyttäjien mobiililaitteisiin. Palvelut kommunikoivat keskenään käyttäen gRPC-etäproseduurikutsuprotokollaa [11]. Käsiteltävä mikropalvelu on toteutettu käyttäen Microsoftin .NET Core -alustaa ja C#-ohjelmointikieltä. Sitä on mahdollista ajaa Dockerin avulla konttina.

Esimerkkien yksinkertaistamiseksi suurin osa niistä käsittelee AppMessagingServicen funktiota *GetAppMessages*, joka palauttaa listan palvelun hallinnoimista viesteistä. Palvelun julkinen rajapinta on gRPC-pohjainen. Koko järjestelmällä on kuitenkin julkinen HTTP-rajapinta, jonka kautta palvelua voidaan myös käyttää. Tätä hyödynnetään savutestien toteutuksessa.

5.2 Testeissä käytetyt teknologiat

Testit toteutetaan pääosin C#-ohjelmointikielellä. Testit kirjoitetaan hyödyntäen xUnit.net-työkalua [36], joka on yleisesti käytetty yksikkötestaustyökalu ja testiajuri .NET-alustoille. Se soveltuu yksikkötestien lisäksi myös korkeamman tason testitapausten määrittelyyn ja ajamiseen. Testisijaisten luontia varten käytetään Moq-kirjastoa.

Integraatio- ja komponenttitestien toteutuksessa käytetään Docker-konttialustaa. Sen avulla voidaan ajaa testattavaa mikropalvelua ja sen tarvitsemia riippuvuuksia, kuten tietokannanhallintajärjestelmää. Lisäksi sopimustestien esimerkkitoteutuksessa hyödynnetään Dockeria myös testien ajamiseen ja paketointiin.

5.3 Testien toteutukset

Esimerkkitoteutukset tehdään yksikkötesteistä, kapeista integraatiotesteistä, komponenttitesteistä, sopimustesteistä ja savutesteistä. Järjestelmätestit jätetään tässä käsittelemättä, sillä niissä itse testien toteutus muistuttaa pitkälti komponenttitestien toteutusta. Järjestelmätesteissä pääpaino on testiympäristön valmistelulla, joka on tässä työssä sivuutettu. Järjestelmätestiympäristön valmistelu voi vaatia monimutkaista ohjelmiston käyttöönoton automatisointia, joka ei varsinaisesti liity testaukseen.

Myös ei-funktionaaliset testit sivuutetaan esimerkkitesteissä. Järkevät suorituskyky- ja kuormitustestit vaativat myös tuotantoympäristöä vastaavan ympäristön, jonka valmistelu on monimutkaista. Nämä testit eivät myöskään vaadi mitään varsinaista toteutusta: tehokkuustestityökaluja on runsaasti, ja usein testejä voi ajaa yksinkertaisilla komentorivikomennoilla.

5.3.1 Yksikkötestien esimerkkitoteutus

Yksikkötestien toteutus riippuu paljon käytetystä ohjelmointikielestä ja sille saatavilla olevista testityökaluista. Olio-ohjelmointikielillä yksikkötesteihin käytetään usein xUnit-kirjastoperheen työkaluja, jotka tarjoavat suunnilleen samanlaisen toiminnallisuuden eri ohjelmointikielille. Tässä työssä käytetty xUnit.net on xUnit-perheen kirjasto C#-ohjelmointikielelle.

Esimerkkinä on tehty *AppMessagingServiceImpl*-luokan funktiolle *GetAppMessages* yksi yksikkötesti. Kyseinen luokka toteuttaa mikropalvelun julkisen rajapinnan, ja käsittelee tietokantaa erillisen luokan kautta (DAO, database access object). Kaikki luokan ulkoiset riippuvuudet on korvattu Moq-kirjastolla luoduilla mock-olioilla, joten yksikkötesti varmistaa vain yhden funktion toiminnan. Toteutettu testi on esitetty alla.

```
1  [Fact]
2  public async Task GetAppMessages_NoMessagesFound_ReturnsEmptyList()
3  {
4    // Arrange
5    _daoMock.Setup(d => d.GetAppMessages())
6      .ReturnsAsync(Faultable<IEnumerable<AppMessageFromDb>>.WithValue(
7        Enumerable.Empty<AppMessageFromDb>()
8      ));
9    var service = new AppMessagingServiceImpl(
10     _loggerFactoryMock.Object, _daoMock.Object,
11     _hangfireHelperMock.Object, _configMock.Object);
12
13   // Act
14   var request = new GetAppMessagesRequest();
15   var response = await service.GetAppMessages(request, null);
16
17   // Assert
18   _daoMock.Verify(d => d.GetAppMessages(), Times.Once);
```

```
19    Assert.Null(response.Error);
20    Assert.Equal(0, response.AppMessages.Count);
21  }
```

***Ohjelma 1**. Yksikkötestin esimerkkitoteutus*

Testitapauksen aluksi DAO-olio määritellään palauttamaan tyhjä lista, kun siltä pyydetään markkinointiviestejä. Tämän jälkeen luodaan testattavasta luokasta uusi olio, joka käyttää edellä valmisteltua DAO-oliota. Testikoodi kutsuu testattavaa operaatiota normaalisti, ja tarkistaa paluuarvon oikeellisuuden. Tarkistukset tehdään xUnit.netin Assert-luokan funktioiden avulla. Jos paluuarvot eivät läpäise tarkistuksia, merkitsee xUnit.net testitapauksen epäonnistuneeksi.

Testikoodista ilmenee yksikkötestien ominaispiirteitä: testisijaisia käytetään runsaasti ja testattava yksikkö on pieni. Tämän testitapauksen suoritus kestää noin 0,3 sekuntia kehitykseen käytetyllä vähävirtaisella kannettavalla tietokoneella. Testin suoritus on siis myös nopeaa. Vastaavia testitapauksia tulisi tehdä niin paljon, että kaikki funktion toiminnallisuus testataan.

5.3.2 Kapeiden integraatiotestien esimerkkitoteutus

Kapeat integraatiotestit voidaan toteuttaa yksikkötestejä muistuttaen. Testien tulee kuitenkin myös hallita ulkoisia riippuvuuksia, joiden kanssa integraatiota testataan. Tässä käytetään esimerkkinä edellä mainittua DAO-luokkaa, joka kommunikoi PostgreSQL-tietokannan kanssa.

Luvussa 4 mainittiin kapeiden integraatiotestien usein hyödyntävän testisijaisia ulkoisten palvelujen hallitsemiseen testeissä. Kokonaisen relaatiotietokannanhallintajärjestelmän korvaaminen testisijaisella on hankalaa, joten DAO-luokan testausta varten luodaan kokonaan uusi tietokantainstanssi. Tässä käytetään apuna Docker-kontteja: ennen testien suoritusta käynnistetään ja valmistellaan kokonaan uusi kontti valmiin PostgreSQL-imagen pohjalta.

Kapean integraatiotestin esimerkkitoteutus on esitetty alla ohjelmassa 2. Testitapauksen suorituksen alussa luodaan Dockerin avulla uusi tietokantainstanssi. Tämän tekee *_dockerPostgresProvider*-olio, joka käynnistää kontit Dockerin rajapintaa käyttäen. Kun tietokantainstanssi on luotu, luodaan *IDatabaseConnectionFactory*- rajapinnasta uusi toteutus, jonka avulla DAO-luokka ottaa yhteyden oikeaan tietokantaan.

Testejä varten tietokantaan luodaan tarvittavat taulut ja sijoitetaan testidataa. Tämän jälkeen suoritetaan testattava operaatio, joka tässä on DAO-luokan *GetAppMessages*-funktio. Lopuksi funktion paluuarvojen oikeellisuus tarkistetaan. Tässä testattu funktio ei tee muutoksia tietokannan tilaan. Muokkaavaa operaatiota testaavat testitapaukset voivat tarkistaa tietokantaan tehdyt muutokset erikseen paluuarvon lisäksi.

```
1  [Fact]
2  public async Task GetAppMessages_OneMsgInTimePeriod_ReturnsCorrectMsg()
3  {
4    // Get a database instance
5    using (var postgresInstance =
6      await _dockerPostgresProvider.GetPostgresInstanceAsync())
7    {
8      // Setup a mock connection factory which
9      // gives out connections to the new instance
10     var dbConnFactory = new Mock<IDatabaseConnectionFactory>();
11     dbConnFactory
12       .Setup(f => f.CreateDatabaseConnectionAsync())
13       .Returns(Task.Run<IDbConnection>(async () =>
14       {
15         return await postgresInstance
16           .GetDatabaseConnectionAsync();
17       }));
18
19     // Initialize & prepare the database
20     using (var dbConnection =
21       await postgresInstance.GetDatabaseConnectionAsync())
22     {
23       await ExecuteSqlFile(dbConnection, "Integration/schema.sql");
24       await ExecuteSql(dbConnection,
25         @"INSERT INTO app_message(created, sent, json_data)
26           VALUES ('now', '2018-02-11T11:11:11.000Z',
27                   '{""Title"": ""Lorem"", ""Body"": ""Ipsum""}');");
28     }
29
30     // Do the actual operation
31     var dao = new DAO(
32       dbConnFactory.Object, _mockLoggerFactory.Object);
33     var result = await dao.GetAppMessages(
34       new DateTimeOffset(2018, 02, 11, 0, 0, 0, TimeSpan.Zero));
35
36     // Assert that the results are correct
37     Assert.True(result.HasValue);
38     Assert.Null(result.Error);
39     Assert.Equal(1, result.Value.Count());
40     var msg = result.Value.First();
41     Assert.Equal("Lorem", msg.Title);
42     Assert.Equal("Ipsum", msg.Body);
43     Assert.Equal(
44       DateTimeOffset.Parse("2018-02-11T11:11:11.000Z"),
45       msg.Sent);
46   }
47 }
```

***Ohjelma 2**. Kapean integraatiotestin esimerkkitoteutus*

Tässä esimerkkitoteutuksessa noin puolet koodiriveistä ovat testien valmistelua varten.

Valmistelut, eli tietokannan luominen ja alustus, vievät myös suurimman osan testin suoritusajasta. Näitä operaatioita ei kannatakaan tehdä erikseen kaikissa testitapauksissa, vaan ennen testien suoritusta. Kaikki testitapaukset voivat käyttää samaa tietokantainstanssia ja suoritus on näin nopeampaa.

5.3.3 Komponenttitestien esimerkkitoteutus

Komponenttitestejä varten palvelua suoritetaan ja testataan kokonaisuutena. Tämä voidaan tehdä Dockerin avulla, sillä testattava palvelusta on saatavilla valmis Docker-image. Kuten luvussa 4 mainittiin, konttien käytöllä voidaan poistaa eroja testi- ja tuotantoympäristöjen välillä. Myös palvelun riippuvuudet, eli tietokanta ja PushNotificationService ovat saatavilla kontteina. Jälkimmäisestä käytetään tässä tynkäversiota, joka palauttaa vakioidun vastauksen kaikkiin kutsuihin eikä sisällä varsinaista toimintalogiikkaa.

Kontit saatetaan ajoon Docker Composen avulla. Sillä määritellään usean kontin kokonaisuus, jossa kontit voivat kommunikoida keskenään eristetysti. Composen määrittelytiedosto sisältää myös konttien konfiguraatiot, kuten tietokantayhteysasetukset. Komponenttitesteihin käytetty Compose-määrittely on esitetty liitteessä A.

Testitapauksen yksinkertaistamiseksi konttiympäristön valmistelua ei tässä ole tehty testikoodissa, vaan testit olettavat ympäristön olevan valmis ennen niiden suoritusta. Testit saavat testattavan palvelun verkko-osoitteen ympäristömuuttujan välityksellä. Ympäristö voitaisiin valmistella myös testikoodin aluksi, kuten kapeiden integraatiotestien tapauksessa edellä tehtiin. Tällöin testit voisivat myös hallita ulkoisia palveluja tarkemmin ja muuttaa esimerkiksi tynkäpalvelujen toimintaa.

Testikoodi tekee kutsun mikropalvelun julkiseen rajapintaan ja varmistaa vastauksen oikeellisuuden. Jos testattava operaatio käyttää jotain ulkoista palvelua, voidaan varmistaa myös, että sitä kutsuttiin oikein. Tässä tapauksessa operaatio tekee vain yhden tietokantakutsun, joka ei muuta tietokannan tilaa. Ulkoisten palveluiden käyttöä ei tässä siis tarvitse erikseen tarkistaa palvelukutsun oikeellisuuden lisäksi. Komponenttitestikoodi on esitetty alla ohjelmassa 3.

```
1  [Fact]
2  public async Task TestGetAppMessages()
3  {
4    // Get the target service's address and connect to it
5    var appMessagingServiceAddress =
6      Environment.GetEnvironmentVariable("APP_MESSAGING_TEST_HOST");
7    var grpcChannel = new Channel(
8      appMessagingServiceAddress, ChannelCredentials.Insecure);
9    try
10   {
11     await grpcChannel.ConnectAsync();
12     var client = new AppMessaging.AppMessagingClient(grpcChannel);
13
```

```
14      // Use the service's public api
15      var request = new GetAppMessagesRequest
16      {
17        OnlyMessagesSince =
18          DateTimeOffset.Parse("2018-01-01").ToTimestamp()
19      };
20      var response = await client.GetAppMessagesAsync(request);
21
22      // Validate the response
23      Assert.Equal(0, response.AppMessages.Count);
24      Assert.Null(response.Error);
25    }
26    finally
27    {
28      await grpcChannel.ShutdownAsync();
29    }
30  }
```

***Ohjelma 3**. Komponenttitestin esimerkkitoteutus*

Järjestelmätestitapaukset ovat hyvin samankaltaisia tässä esitellyn komponenttitestitapauksen kanssa. Kummassakin testityypissä tehdään kutsuja julkisiin rajapintoihin, ja varmistetaan vastauksien ja operaatioiden sivuvaikutusten oikeellisuus. Erot testityypeissä johtuvatkin testiympäristön valmistelun ja testitapausten monimutkaisuudesta järjestelmätesteillä.

5.3.4 Sopimustestien esimerkkitoteutus

Kuluttajavetoisten sopimustestien toteutukseen on joitakin työkaluja. Tällainen työkalu on esimerkiksi *Pact* [21], jolla voi toteuttaa sopimustestejä JSON-rajapinnoille. Pact tallentaa sopimukset JSON-tiedostoihin, ja rajapinnan tarjoaja voi niiden perusteella ajaa sopimustestit [22].

Tässä työssä esimerkkinä käytetyt palvelut kommunikoivat käyttäen gRPC-protokollaa, jolle ei ole valmiita sopimustestaustyökaluja. Protokollan käyttöä varten tulee määritellä vahvasti tyypitetyt rajapinnat, jotka voidaan tulkita eräänlaisiksi sopimuksiksi. Sopimustestit voivat kuitenkin olla edelleen tarpeellisia, sillä ne havaitsevat muutokset rajapinnoissa.

Myös sopimustesteissä on hyödynnetty Dockeria, jolla kuluttajatestitapaukset voidaan määritellä yhtenäisesti rajapinnan tarjoajaa varten. Kukin rajapinnan kuluttaja voi luoda omat sopimustestitapauksensa ja tehdä niistä Docker-imagen. Rajapinnan tarjoaja ajaa kaikki itselleen määritellyt testit näistä imageista luotuina Docker-kontteina. Testien toteutusteknologioilla ei näin ole tarjoajan kannalta merkitystä.

Esimerkkitestejä varten määriteltiin ensin kuluttajan testitapaus, jonka toteutus on esitetty alla ohjelmassa 4. Testitapaus on määritelty kuluttajalle, joka käyttää ainoastaan App-

MessagingServicen GetAppMessages-funktiota, ja tarvitsee kustakin palautetusta viestistä otsikon ja sisällön. Testikontissa testit suoritetaan xUnit.netin avulla. Mikäli rajapinnan tarjoaja ei läpäise sopimustestejä, tai testien suoritus epäonnistuu, testiprosessi palauttaa virhekoodin. Tarjoaja voi siitä testejä ajaessaan päätellä, läpäistiinkö testit vai ei.

```
1  [Fact]
2  public async Task TestGetAppMessages()
3  {
4    var appMessagingServerAddress =
5      Environment.GetEnvironmentVariable("APPMESSAGING_ADDRESS");
6
7    var channel = new Channel(
8      appMessagingServerAddress, ChannelCredentials.Insecure);
9    var client = new AppMessaging.AppMessagingClient(channel);
10
11   // Assume this returns 2 messages
12   var resp = await client.GetAppMessagesAsync(
13     new GetAppMessagesRequest());
14   Assert.Equal(2, resp.AppMessages.Count);
15   // Check that we got the title and the body
16   Assert.Equal("Message 1 Title", resp.AppMessages[0].Title);
17   Assert.Equal("Message 1 Contents", resp.AppMessages[0].Body);
18   Assert.Null(resp.Error);
19 }
```

***Ohjelma 4**. Kuluttajan sopimustestitoteutus*

Rajapinnan tarjoaja voi ajaa myös kuluttajien määrittelemät sopimustestit testityökalun, kuten xUnit.netin avulla. Näin myös sopimustestit ovat yksinkertaisia ajaa muiden testien ohessa. Esimerkkitoteutus sopimustestien ajamisesta on esitetty alla ohjelmassa 5. Tässä hyödynnetään xUnit.netin *Theory*-ominaisuutta, jolla voidaan ajaa sama testitapaus useilla eri parametreilla. Tässä on parametrina annettu kuluttajan määrittelemä Docker-image.

Ennen kuluttajan määrittelemien testien ajamista testikoodi valmistelee oman tilansa. Koska sopimustestit testaavat vain palvelun julkista rajapintaa eikä palvelun varsinaista toimintalogiikkaa, suurin osa muusta palvelusta korvataan testisijaisilla. Tässä DAO-olio korvataan mock-oliolla, joka määritellään palauttamaan vakioitu vastaus GetAppMessages-funktiosta. Palvelu käynnistetään siten, että kuluttajan määrittelemä testikoodi voi ottaa siihen yhteyden. Lopuksi ajetaan kuluttajan testikontti. Mikäli se palauttaa virhekoodin, tulostetaan kontin lokiviestit ja merkitään testitapaus epäonnistuneeksi.

```
1  [Theory]
2  [InlineData("appmessaging-consumer:1-latest")]
3  public async Task RunContractTests(string image)
4  {
5    // Setup state to have two messages
6    _daoMock.Setup(d => d.GetAppMessages(It.IsAny<DateTimeOffset?>()))
7      .ReturnsAsync(
```

```
8         Faultable<IEnumerable<AppMessageFromDb>>.WithValue(
9           new List<AppMessageFromDb>
10          {
11            new AppMessageFromDb(
12              1, DateTimeOffset.Parse("2018-01-01T00:00:00Z"),
13              true, DateTimeOffset.Parse("2018-01-01T00:00:00Z"),
14              "Message 1 Title", "Message 1 Contents"),
15            new AppMessageFromDb(
16              2, DateTimeOffset.Parse("2018-01-02T00:00:00Z"),
17              true, DateTimeOffset.Parse("2018-01-02T00:00:00Z"),
18              "Message 2 Title", "Message 2 Contents")
19          }));
20
21    var service = new AppMessagingServiceImpl(
22      _loggerFactoryMock.Object,
23      _daoMock.Object,
24      _hangfireHelperMock.Object,
25      _configMock.Object);
26
27    // Start the server the consumer tests will connect to
28    var server = new Server
29    {
30      Ports =
31      {
32          new ServerPort("0.0.0.0", 0, ServerCredentials.Insecure)
33      },
34      Services = { AppMessaging.BindService(service) }
35    };
36    server.Start();
37
38    try
39    {
40      var port = server.Ports.First().BoundPort;
41      var envVars = new List<string>
42      {
43        $"APPMESSAGING_ADDRESS=localhost:{port}"
44      };
45
46      using (var container = await RunContainerAsync(image, envVars))
47      {
48        if (container.StatusCode != 0) {
49          var logs = await container.GetLogsAsync();
50          _output.WriteLine(
51              $"Logs from container {image}:\n{logs}");
52          // fail the test
53          Assert.True(
54            false, $"Running consumer tests from {image} failed");
55        }
56      }
57    }
58    finally
```

```
59    {
60      await server.ShutdownAsync();
61    }
62  }
```

***Ohjelma 5**. Rajapinnan tarjoajan sopimustestiajuri*

Tässä sopimustestien esimerkkitoteutuksessa on monia ongelmia. Kuluttajatestit eivät voi määrittää tarjoajan sisäistä tilaa, joten esimerkiksi virhetilanteiden testaus on mahdotonta. Kuluttajatesteille ei myöskään ole tarkkaa rajapintaa, joten niiden kirjoittaminen vaatii tuntemusta tarjoajan testiajurista. Lisäksi uusien kuluttajatestien lisääminen vaatii muutoksia myös tarjoajan testikoodiin.

Sopimustestaustyökalut, kuten Pact sisältävät ratkaisut näihin ongelmiin. Pactin sopimustiedostot ovat vakiomuotoisia, joten kuluttajien ja tarjoajien ei tarvitse välittää toistensa testien suorituksesta [22]. Sopimustiedostojen jakelua varten Pactissa on erillinen palvelin, jonka välityksellä tarjoajat voivat hakea itseään koskevat kuluttajatestit [28]. Lisäksi kukin kuluttajatestitapaus määrittelee tarjoajan tilan ennen testien suoritusta [22]. Vastaavat ominaisuudet olisi mahdollista toteuttaa myös tämän työn sopimustesteihin, mutta ne on sivuutettu esimerkkien yksinkertaistamiseksi.

5.3.5 Savutestien esimerkkitoteutus

Savutestien on tarkoitus testata nopeasti, että ohjelmisto ei ole täysin toimintakelvoton. Esimerkkisavutestit on tässä toteutettu yksinkertaisina HTTP-kutsuina. Niiden vastauksista varmistetaan ainoastaan, että tilakoodit eivät ilmaise virheitä. Tällaiset testit sopisivat esimerkiksi ohjelmiston käyttöönoton onnistumisen varmistukseen.

Testit on toteutettu shell-skriptinä, jossa HTTP-kutsut tehdään curl-työkalua käyttäen. Testikoodi on esitetty ohjelmassa 6.

```
1  #!/bin/sh
2
3  # Exit immediately after any command returns an error
4  set -e
5
6  # This script takes the base URL
7  # (e.g. http://example.com/api/) as a parameter
8  BASEURL="$1"
9
10 # Test app messaging
11 echo "Testing $BASEURL/messages..."
12 curl \
13     --fail \          # Fail if the response status indicates an error
14     --max-time 20 \   # Fail if the request takes more than 20 seconds
15     "$BASEURL/messages"
16
```

```
17 # ...
18 # More test cases here
19 # ...
20
21 echo "All smoke tests completed succesfully."
```

***Ohjelma 6**. Savutestien esimerkkitoteutus*

Savutestit ovat hyvin yksinkertaisia. Yllä yhtä testitapausta vastaa käytännössä vain yksi komento. Muihin testityyppeihin verrattuna savutesteistä puuttuu testien valmistelu käytännössä kokonaan. Savutestien tarkoitus on testata, että järjestelmä on suorituksessa oikein, joten niiden ei kuulu valmistella testiympäristöä.

6. YHTEENVETO

Tässä työssä tutkittiin tapoja mikropalveluohjelmistojen tehokkaaseen automaattitestaamiseen. Työssä esiteltiin mikropalveluarkkitehtuuri sekä sen kanssa yleisesti käytetty konttiteknologia ja Docker-konttialusta. Tämän jälkeen työssä esiteltiin eri tekniikoita mikropalveluohjelmistojen testaamiseen sekä testipyramidimalli, joka ohjaa testien suunnitteluprosessia. Lopuksi tehtiin esimerkkitoteutukset eri testityypeille.

Työssä muodostui selkeä kuva mikropalvelujen automaattitestauksesta. Mikropalvelujen tehokas testaaminen vaatii monien testityyppien käyttämistä. Nopeiden yksikkötestien rooli on mikropalveluohjelmistoa testattaessa suuri, mutta ne eivät yksin kykene varmistamaan ohjelmiston toimivuutta. Aihetta käsittelevässä kirjallisuudessa on hyvin yhteneväinen käsitys hyvästä mikropalvelujen testaustavasta: testausta tulisi tehdä monella eri tasolla testipyramidimallia noudattaen. Yksikkötestien lisäksi erilaiset integraatiotestit ovat tärkeässä osassa. Sen sijaan järjestelmätestien määrä tulisi monien lähteiden mukaan pitää mahdollisimman pienenä.

Mikropalvelujen testausta käsittelevissä lähteissä esiintyy hyvin usein sopimustestauksen ja erityisesti kuluttajavetoisten sopimustestien käsite. Sopimustestejä suositellaan yleisesti varmistamaan mikropalveluiden välisten rajapintojen käyttö. Mikropalveluarkkitehtuurille on tyypillistä, että kullakin palvelulla on eri kehittäjät tai kehitystiimit. Rajapintojen testaus on tärkeää, koska niiden eri osapuolet kehittyvät itsenäisesti, ja rajapintojen muutoksista johtuvat virheet voivat olla yleisiä. Sopimustestaus ei kuitenkaan vielä ole kovin laajalti käytössä, minkä vuoksi työkaluja siihen ei juurikaan ole. Tässä työssä tehtiin oma Dockerin käyttöön perustuva esimerkkiratkaisu sopimustestaukseen.

Testien esimerkkitoteutuksia tehdessä ilmeni, että suurin osa eri testityyppien toteutusten eroista tulee testien valmistelusta. Testien tulee tarjota testattavalle ohjelmistolle tai sen osalle ympäristö siten, että sen tilaa voidaan hallita. Mikropalvelujärjestelmässä ympäristön hallittavia osia voi olla paljon enemmän kuin monoliittisessa järjestelmässä. Jo tämän työn todella yksinkertaisissa testitapauksissa testien valmistelu ja ympäristön hallinta oli selvästi hankalin ja aikaa vievin toteutuksen osuus.

Toteutetut esimerkkitestit ovat melko yksinkertaisia, jotta ne pysyisivät riittävän pieninä tämän työn tarkoituksiin. Testitapausten suunnittelu jätettiin työn ulkopuolelle, sillä se on jo itsessään hyvin laaja aihealue. Esimerkkitestien tekniset ratkaisut voisivat kuitenkin toimia perustana testien toteutukselle oikeissa ohjelmistoprojekteissa. Dockerin käyttö testiympäristön valmisteluun vaikutti toimivalta ratkaisulta. Testitapauksia tulisi kuitenkin yleistää ja jatkokehittää siten, että koodin kopiointi vähenisi, resurssien uudelleenkäyttö testiympäristössä olisi mahdollista ja testien kirjoitus olisi yksinkertaisempaa. Muutoksia

testeihin tulisi tietenkin myös testien sovittamisesta muun ohjelmiston toteutustekniikoihin ja arkkitehtuuriin. Eri testityyppien perusajatukset ja tyypilliset vaiheet ovat kuitenkin esimerkkitesteissä hyvin esillä.

LÄHTEET

[1] T. Clemson, Testing Strategies in a Microservice Architecture, verkkosivu, 18 Nov, 2014. Saatavissa (viitattu 22.4.2018): https://martinfowler.com/articles/microservice-testing/

[2] M. Cohn, Succeeding with Agile: Software Development Using Scrum, Addison-Wesley, Upper Saddle River, NJ, 2010.

[3] Containers at Google, Google Cloud, verkkosivu, 2018. Saatavissa (viitattu 13.5.2018): https://cloud.google.com/containers/

[4] DC/OS, Mesosphere, Inc., verkkosivu, 2018. Saatavissa (viitattu 12.5.2018): https://dcos.io

[5] Docker Security, Docker Inc., verkkosivu, 2018. Saatavissa (viitattu 12.5.2018): https://docs.docker.com/engine/security/security/

[6] M. Fowler, TestDouble, Jan 17, 2006. Saatavissa (viitattu 15.5.2018): https://martinfowler.com/bliki/TestDouble.html

[7] M. Fowler, ContractTest, Jan 12, 2011. Saatavissa (viitattu 1.6.2018): https://martinfowler.com/bliki/ContractTest.html

[8] M. Fowler, IntegrationTest, Jan 16, 2018. Saatavissa (viitattu 1.6.2018): https://martinfowler.com/bliki/IntegrationTest.html

[9] M. Fowler, J. Lewis, Microservices, Mar 25, 2014. Saatavissa (viitattu 17.4.2018): https://martinfowler.com/articles/microservices.html

[10] V. Garousi, F. Elberzhager, Test Automation: Not Just for Test Execution, IEEE Software, Vol. 34, Iss. 2, March, 2017, pp. 90–96.

[11] gRPC, The gRPC Authors, verkkosivu, 2018. Saatavissa (viitattu 9.6.2018): https://grpc.io

[12] H. Harnisch, Integration Testing With Docker Compose, Jun 19, 2016. Saatavissa (viitattu 1.6.2018): https://hharnisc.github.io/2016/06/19/integration-testing-with-docker-compose.html

[13] D. Jaramillo, D.V. Nguyen, R. Smart, Leveraging microservices architecture by using Docker technology, in: SoutheastCon 2016, March, 2016, pp. 1–5.

[14] M. Kassab, J.F. DeFranco, P.A. Laplante, Software Testing: The State of the Practice, IEEE Software, Vol. 34, Iss. 5, 2017, pp. 46–52.

[15] Kubernetes, The Kubernetes Authors, verkkosivu, 2018. Saatavissa (viitattu 12.5.2018): https://kubernetes.io

[16] M. Martins, S. Joshi, R. Vennam, K. Eati, D. Glozic, S. Daya, M. Gupta, V. Lampkin, C.M. Ferreira, N.V. Duy, S. Narain, V. Gucer, Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach, 1st ed., IBM Redbooks, United States, Aug 26, 2015. Saatavissa (viitattu 22.4.2018): http://www.redbooks.ibm.com/redbooks/pdfs/sg248275.pdf

[17] S. McConnell, Daily build and smoke test, IEEE Software, Vol. 13, Iss. 4, July 1996, pp. 143–144.

[18] S. Newman, Building Microservices, 1st ed., O'Reilly, Sebastopol, CA, 2015.

[19] R.V. O'Connor, P. Elger, P.M. Clarke, Continuous software engineering—A microservices architecture perspective, Journal of Software: Evolution and Process, Vol. 29, Iss. 11, Nov, 2017.

[20] Overview of Docker Compose, Docker, Inc., verkkosivu, 2018. Saatavissa (viitattu 12.5.2018): https://docs.docker.com/compose/overview/

[21] Pact: Introduction, Pact Foundation, verkkosivu, 2018. Saatavissa (viitattu 9.6.2018): https://docs.pact.io

[22] Pact: Terminology, Pact Foundation, verkkosivu, 2018. Saatavissa (viitattu 25.6.2018): https://docs.pact.io/terminology

[23] J. Palermo, Guidelines for Test-Driven Development, verkkosivu, May, 2006. Saatavissa (viitattu 2.6.2018): https://msdn.microsoft.com/en-us/library/aa730844(v=vs.80).aspx

[24] M. Polo, P. Reales, M. Piattini, C. Ebert, Test Automation, IEEE Software, Vol. 30, Iss. 1, 2013, pp. 84–89.

[25] R.S. Pressman, B.R. Maxim, Software engineering : a practitioner's approach, 8th ed., McGraw-Hill Education, New York, 2014.

[26] C. Richardson, Pattern: Monolithic Architecture, verkkosivu, 2017. Saatavissa (viitattu 3.4.2018): http://microservices.io/patterns/monolithic.html

[27] M. Shahin, M.A. Babar, L. Zhu, Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices, IEEE Access, Vol. 5, 2017, pp. 3909–3943.

[28] Sharing Pacts with the Pact Broker, Pact Foundation, verkkosivu, 2018. Saatavissa (viitattu 25.6.2018): https://docs.pact.io/getting-started/sharing-pacts-with-the-pact-broker

[29] V. Singh, S.K. Peddoju, Container-based microservice architecture for cloud applications, in: 2017 International Conference on Computing, Communication and Automation (ICCCA), 2017, pp. 847–852.

[30] D. Spinellis, State-of-the-Art Software Testing, IEEE Software, Vol. 34, Iss. 5, 2017, p. 4.

[31] D. Taibi, V. Lenarduzzi, C. Pahl, Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation, IEEE Cloud Computing, Vol. 4, Iss. 5, 2017, pp. 22–32.

[32] A.S. Tanenbaum, H. Bos, Modern operating systems, 4. ed., Pearson Prentice-Hall, Upper Saddle River, NJ, 2014.

[33] C. de la Torre, B. Wagner, M. Rousos, .NET Microservices: Architecture for Containerized .NET Applications, 2nd ed., Microsoft Developer Division, Redmond, Washington, 2018. Saatavissa (viitattu 12.5.2018): https://aka.ms/microservicesebook

[34] H. Vocke, The Practical Test Pyramid, verkkosivu, 26 Feb, 2018. Saatavissa (viitattu 26.3.2018): https://martinfowler.com/articles/practical-test-pyramid.html

[35] E. Wolff, Microservices: Flexible Software Architecture, Addison-Wesley Professional, Oct 21, 2016.

[36] xUnit.net, .NET Foundation, verkkosivu, 2018. Saatavissa (viitattu 9.6.2018): https://xunit.github.io

LIITE A: DOCKER-COMPOSE.YML-TIEDOSTO KOMPONENTTITESTEILLE

```
1  version: "3"
2  services:
3   db:
4     image: postgres:9.6
5     ports:
6       - "0:5432"
7     labels:
8       origin: "appmessagingservice-componenttest"
9     volumes:
10      - "./db_init_sql:/docker-entrypoint-initdb.d"
11   appmessagingservice:
12    image: tests/appmessagingservice:latest
13    ports:
14      - "0:8080"
15    labels:
16      origin: "appmessagingservice-componenttest"
17    environment:
18      DATABASE: "Host=db;Port=5432;Database=postgres;\
19         Username=postgres;Password=postgres"
20      APP_MESSAGING_GRPC_HOST__HOST: "0.0.0.0"
21      APP_MESSAGING_GRPC_HOST__PORT: "8080"
22      NOTIFICATION_GRPC__HOST: "pushnotificationservice"
23      NOTIFICATION_GRPC__PORT: "5000"
24    restart: always
25    depends_on:
26      - db
27      - pushnotificationservice
28   pushnotificationservice:
29    image: tests/pushnotifservice-stub:latest
30    labels:
31      origin: "appmessagingservice-componenttest"
32    expose:
33      - "5000"
```