Mika Lammi

Automaattinen E2E-testaus hajautetussa tapahtumaohjatussa FaaS-sovelluksessa

Tietotekniikan Pro Gradu -tutkielma

8. kesäkuuta 2024

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Mika Lammi

Yhteystiedot: mika.j.lammi@student.jyu.fi

Ohjaaja: Tommi Mikkonen

Työn nimi: Automaattinen E2E-testaus hajautetussa tapahtumaohjatussa FaaS-sovelluksessa

Title in English: Automatic E2E testing in distributed event-driven FaaS application

Työ: Pro Gradu -tutkielma

Opintosuunta: Ohjelmistotekniikka

Sivumäärä: 66+11

Tiivistelmä: Hajautettujen tapahtumaohjattujen FaaS-sovelluksien päästä päähän -testaus (E2E Testing) on haastavaa, koska pilviympäristöjä ei voida replikoida lokaalisti, ja siksi testit pitää suorittaa todellisessa pilviympäristössä. Tässä tutkielmassa tutkitaan, miten hajautetuille FaaS-sovelluksille voidaan toteuttaa päästä päähän -testejä toteuttamalla sidosryhmänä toimivan organisaation sovellukseen toimiva ohjelmistokehys päästä päähän -testausta varten. Toteutettua ohjelmistokehystä arvioidaan organisaation asettamien toiminnallisten ja laadullisten vaatimuksien pohjalta, ja sen toimivuutta käytännössä testataan toteuttamalla sen avulla sovelluksen käyttötapaukselle testiskripti. Tutkimusmenetelmänä käytetään suunnittelutieteellistä tutkimusmenetelmää, jossa luodun ohjelmistokehyksen pohjalta tehty dokumentaatio toteutuksen arkkitehtuurista ja siinä käytetyistä teknologioista toimii artefaktina. Toteutuksessa testit suoritetaan automaattisesti pitkäaikaisessa pilviympäristössä AWS CodeBuild-projektissa. Toteutus tukee rinnakkaisia testiajoja, ja siihen lisättiin tuki ulkoisten rajapintojen korvaamiseen testisijaisten avulla. Jatkokehityksenä sovellukselle voisi lisätä tuen lyhytaikaisten ympäristöjen käyttöönottamiseen, jolloin testejä voitaisiin ajaa automaattisesti suoraan CI/CD-prosessin osana.

Avainsanat: päästä päähän -testaus, E2E, mikropalvelu, pilvipalvelu, palveliton, FaaS, hajautettu järjestelmä, järjestelmätestaus, AWS

Abstract: End-to-end testing (E2E Testing) of distributed and event-driven FaaS-applications is challenging because real cloud environments cannot be replicated locally and thus the tests need to be executed in the cloud. This thesis aims to investigate how end-to-end tests can be implemented for a distributed FaaS-application by implementing a working end-to-end testing framework for stakeholder organisation's application. The implementation is evaluated by functional and non-functional requirements set by the organisation and its functionality in practice is tested by using the framework to implement a real use case of the application as a test script. This thesis uses design science as a research method and the design science artifact will be the documentation of technologies and architectural decisions used in the implementation. In the final implementation, end-to-end tests are executed in an AWS CodeBuild-project in a long-term cloud environment. The implementation also supports concurrent test runs and mocking of external APIs. As a further development, a support for deployments of short-term environments could be added to the application, so that the end-to-end tests could be executed directly as a part of CI/CD execution process.

Keywords: end-to-end testing, E2E, microservice, cloud service, serverless, FaaS, distributed system, system testing, AWS

Kuviot

Sisällys

1 Johdanto

Ohjelmistojen testaaminen niiden toimivuuden varmistamiseksi on tärkeää, sillä ohjelmistojen viat voivat aiheuttaa projektien viivästymistä, keskeytymistä, kustannuksia tai pahimmillaan jopa ihmishenkien menetystä. Esimerkiksi Nasan Mars Polar Lander epäonnistui laskeutumaan Marsin pinnalle vuonna 1999 yhdessä koodirivissä olleen virheen takia. Polar Landerin menetyksestä koitui Nasalle 165 miljoonan dollarin tappiot. (McQuaid 2012)

Usein ajatellaan, että ohjelmistotestaamisen tarkoituksena on todentaa ohjelmiston toimivuus tai vähentää ohjelmiston käyttöön liittyviä riskejä. Pohjimmiltaan ohjelmistotestaus on kuitenkin ajattelutapa, jota noudattamalla voidaan tuottaa laadukkaita ohjelmistoja. Testaus jaetaan usein verifiointiin ja validointiin. Verifioinnin tarkoituksena on varmistaa, että ohjelmisto vastaa sille annettuja vaatimuksia, kun taas validoinnin tarkoituksena on varmistaa, että ohjelmisto ja sille asetetut vaatimukset sopivat ohjelmiston käyttötarkoitukseen. (Ammann ja Offutt 2017)

Ohjelmistotestaus on hyvin kallista ja aikaa vievää työtä, joten mahdollisimman suuri osa testeistä pyritään automatisoimaan. Nimensä mukaisesti automaatiotestit ovat testejä, jotka ajetaan automaattisesti, esimerkiksi ennen käyttöönottoa automaatioputkessa (Ammann ja Offutt 2017).

(Cohn 2009) esitti ensimmäisenä kirjassaan testauspyramidin käsitteen, jossa automaatiotestit ollaan jaettu kolmeen eri kategoriaan: yksikkötesteihin, intgeraatiotesteihin sekä päästä päähän -testeihin. Testauspyramidin alimmalla tasolla ovat yksikkötestit, joilla testataan toteutuksen ”yksiköiden“ toimivuutta, jotka ovat yleensä yksittäisiä funktioita (Ammann ja Offutt 2017). Koska suurin osa automaatiotesteistä koostuu yksikkötesteistä, niitä on suhteellisen helppoa ja kustannustehokasta tehdä ja niiden avulla pystytään paikantamaan virheet ohjelmistokoodista varhaisessa vaiheessa (Cohn 2009). Testauspyramidin keskitasolla ovat integraatiotestit, joita (Cohn 2009) kutsuu palvelutason testeiksi (Service-level testing). Niiden avulla testataan palveluiden keskinäistä toiminnallisuutta järjestelmän sisällä ilman loppukäyttäjille tarkoitettuja rajapintoja. Testauspyramidin huipulla ovat päästä päähän -testit, joita (Cohn 2009) kutsuu UI-testeiksi (User Interface tests). Automaattisissa päästä päähän

-testeissä loppukäyttäjän tekemiä manuaalisia toimintoja jäljitellään automaattisten skriptien avulla, jolloin saadaan verifioitua sovelluksen toimivuus sen erilaisissa käyttötapauksissa (Ricca ja Stocco 2021). Nimensä mukaisesti päästä päähän -testeillä testataan koko järjestelmän toimivuutta alkaen loppukäyttäjän tekemistä operaatioista käyttöliittymällä tai rajapinnoista ja päättyen siihen että tarkistetaan vastaavatko järjestelmän palauttamat arvot odotettuja arvoja. Niiden avulla pystytään todentamaan järjestelmän liiketoimintalogiikan toimivuus. Päästä päähän -testit toimivat tyypillisesti regressiotesteinä, eli niiden avulla varmistetaan ettei sovellukseen lisätyt uudet toiminnallisuudet hajoita aiempia toiminnallisuuksia (Ricca ja Stocco 2021). (Cohn 2009) perustelee, että päästä päähän -testauksella tulee testata vain kriittisimmät toiminnallisuudet, sillä ne ovat hauraita rikkumaan pienistä käyttöliittymän muutoksista, niiden testiajossa menee kauan aikaa ja niitä on vaikea kirjoittaa.

Mikropalveluarkkitehtuuria hyödyntävien sovelluksien suosio on ollut tasaisessa nousussa ja yritykset kuten Amazon, Netflix ja Spotify hyödyntävät sitä omissa järjestelmissään (Bushong ym. 2021). Ensimmäinen ja eräs tunnetuimmista määritelmistä mikropalveluarkkitehtuurille on artikkelista (Fowler ja Lewis 2014) (Di Francesco, Lago ja Malavolta 2019; Bushong ym. 2021). Lyhyesti sanottuna mikropalveluarkkitehtuurissa sovellus luodaan useista pienistä mikropalveluista, jotka kommunikoivat toistensa kanssa, esimerkiksi HTTP-viestien avulla.

Eräs huolenaihe mikropalveluissa liittyy palvelimien hallinnoimiseen. DevOps automaatiotyökalujen (kuten Docker ja Auto Scaling) implementointi ja käyttöönotto voivat kuluttaa paljon organisaation aikaa ja resursseja. Tästä syystä pilvipalveluntarjoajat tarjoavat nykyään palvelittomia (Serverless) pilvilaskentaratkaisuja, kuten AWS Lambda, joiden avulla yksittäisiä mikropalveluita pystytään käyttöönottamaan ja skaalaamaan automaattisesti ilman, että organisaation tarvitsee hallinnoida palvelininfrastruktuuria. (Villamizar ym. 2017)

Palvelittomia funktioita voidaan kutsua joko asynkronisesti tai synkronisesti. Synkronisissa kutsuissa kutsuja odottaa, kunnes kutsuttu funktio on käsitellyt kutsun loppuun asti. Asynkroniset kutsut sen sijaan perustuvat tapahtumapohjaiseen menettelytapaan. (Winzinger ja Wirtz 2019). Tapahtumapohjaisen arkkitehtuurin keskeinen idea on, että jostakin merkittävän asiasta luodaan tapahtuma (Event), ja kyseisestä tapahtumasta kiinnostuneet tahot voivat vastaanottaa tapahtuman ja tarvittaessa reagoida siihen suorittamalla prosesseja tai ohjelma-

koodia. Oleennaista on, että tapahtuman luoja (Source) ei tiedä tapahtuman vastaanottavista tahoista mitään, tai vastaanotettiinko tapahtumaa ylipäätään. Tämä ominaisuus takaa löyhät kytkökset eli riippuvuussuhteet eri tahojen väleille. (Michelson 2006)

Toisaalta, mikropalveluarkkitehtuuriin liittyy myös haasteita. Integraatiotestaus ja päästä päähän -testaus hajautetussa pilvi-infrastruktuurissa on haastavampaa kuin tavallisessa monoliittisessa-sovelluksessa. Testien ajamiseen tarvittavien työkalujen lisäksi tulisi olla työkaluja ja infrastruktuuria lokitietojen ja metriikkatietojen keräämiseen ja välittämiseen mikropalveluiden välillä, jotta testeissä ilmenevät viat voitaisiin paikantaa. Testiinfrastruktuurin ja työkalujen toteuttamiseen saattaa joutua uhraamaan lukuisia tunteja työaikaa. (Gortázar ja Gallego 2018).

Hajautettujen FaaS-funktioita hyödyntävien sovelluksien päästä päähän testaamiseen liittyy useita haasteita. Pilviympäristöä ei pysty replikoimaan lokaalissa testausympäristössä, joten testit suositellaan suoritettavan pilviympäristössä (Leitner ym. 2018). Koska testausympäristön käyttöönotossa jokaisessa testiajossa erikseen voisi kulua pitkä aika, testit suoritetaan usein omassa pitkäaikaisessa testausympäristössään. Tällöin yhtenä ongelmana on yhtäaikaisten testitapausten suorituksien eristäminen toisistaan, sekä ulkoisten järjestelmien injektoiminen testisijaisilla. FaaS-funktioiden kylmäkäynnistys saattaa aiheuttaa HTTP-kutsujen aikakeskeytyksen, ja verkkoyhteyksien katkeamiset saattavat aiheuttaa testien epäonnistumisen (Rinta-Jaskari ym. 2022). Testien epäonnistuessa vikojen paikantaminen hajautetussa järjestelmässä on myös haastavaa (Gortázar ja Gallego 2018).

Tässä tutkielmassa selvitetään, miten automaattiset päästä päähän -testit voidaan toteuttaa hajautetussa pilvipohjaisessa järjestelmässä simuloiden mahdollisimman aitoja liiketoimintamalleja. Tutkielmassa toteutetaan suunnittelutieteen menetelmällä artefakti, jota voidaan soveltaa edellä mainitun kaltaisten päästä päähän -testien kehittämisprosessissa. Tavoitteena on ensisijaisesti tuottaa päästä päähän testejä organisaation sovellukselle, mutta samalla tuottaa arvoa tieteen näkökulmasta luomalla hyödyllinen artefakti, jota voidaan käyttää yleispätevästi päästä päähän -testien kehityksen apuna vastaavanlaisiin sovelluksiin.

Tutkielman sidosryhmänä toimii organisaatio Akamon Innovations Oy, jonka Dataplatform-sovellukselle luodaan päästä päähän -testaukseen ohjelmistokehys. Organisaation kohdeso-

vellus Dataplatform on pilvinatiivi palvelualusta, jota ulkopuoliset järjestelmät ja tahot voivat käyttää sekä SaaS- (Software-as-a-Service) että PaaS-palveluina (Platform-as-a-Serice). Sen tarkoituksena on kerätä ja varastoida dataa eri lähteistä ja jalostaa sitä tiedoksi, jota ulkopuoliset tahot voivat hyödyntää. Se on multi-tenant sovellus, eli alustaa hyödyntävät useat eri organisaatiot, joita yleensä kutsutaan tenanteiksi, mutta toisinaan organisaation alla voi olla useita tenantteja asiakasyhtiöinä. Tenanteilla ei ole pääsyä toistensa dataan sovelluksessa. Tenanteille kuuluu käyttäjiä, jotka kuuluvat organisaation 0-N asiakasyhtiön asiakkaiksi. Sovellus toimii PaaS-palveluna REST-rajapinnan kautta sekä SaaS-palveluna GraphQL-rajapinnan kautta. Sovelluksella on myös ETRM-rajapinta (Energy Trading and Risk Management), jota hyödynnetään automatisoimaan sähkön hintaa loppuasiakkaille sekä auttamaan yhtiöitä riskien hallinnassa. Sovelluksen arkkitehtuuri pohjautuu mikropalveluarkkitehtuuriin, joka on toteutettu pilvinatiivisti AWS Lambdan palvelittomia pilvilaskentapalveluja hyödyntämällä. Mikropalveluiden välinen viestintä toimii tapahtumapohjaisesti. Suurin osa sovelluksesta on toteutettu Typescript-ohjelmointikielellä Node.js-ajoympäristössä.

Luvussa 2 käydään läpi tutkielman kannalta oleellisia käsitteitä ja teknologioita, joita ovat mikropalveluarkkihtehtuuri, AWS Lambda ja Amazon EventBridge. Luvussa 3 käydään läpi palvelittomien pilvipalveluiden testaamiseen liittyvää aikaisempaa tutkimustietoa sekä teollisuuden parhaita käytänteitä. Luvussa 4 käydään läpi suunnittelutieteellinen tutkimusmenetelmä ja selitetään, miten sitä sovelletaan tässä tutkielmassa. Lisäksi määritellään vaatimukset tutkielman osana toteutettavalle päästä päähän -testauksen ohjelmistokehykselle, sekä esitellään käyttötapaus, jonka testiskriptin avulla vaatimuksien onnistumista arvioidaan käytännössä. Luvussa 5 käydään läpi toteutetun ohjelmistokehyksen arkkitehtuurisia ominaisuuksia ja siinä käytettyjä teknologioita. Luvussa 6 arvioidaan, kuinka hyvin toteutettu ohjelmistokehys toteutui sille asetettujen vaatimuksien pohjalta, sekä kuinka käyttötapauksen toteutus testiskriptinä onnistui. Luvussa 7 pohditaan mahdollisia jatkokehitystarpeita, mitä tutkielmasta opittiin ja miten kehitysprosessin kulku eteni. Viimeisessä luvussa 8 on vielä yhteenveto tutkielmasta.

2 Taustateknologiat

Tässä luvussa kuvataan tutkielman kannalta keskeiset teknologiat ja käsitteet. Näitä ovat mikropalveluarkkitehtuuri, AWS Lambda sekä Amazon EventBridge.

2.1 Mikropalveluarkkitehtuuri

Järjestelmän arkkitehtuuri on sen komponenttien kokoonpano, joka tuottaa sen järjestelmätasoisen käyttäytymisen. Arkkitehtuuri koostuu komponenteista, jotka ovat toistensa kanssa vuorovaikutuksessa. Komponentit määritellään niiden kykyjen ja rajoitteiden perusteella ja ne voivat vastata esimerkiksi tiettyyn stimuliin tai tilojen muutoksiin. Stimuli aiheuttaa järjestelmässä mekanismin eli komponenttien välisen vuorovaikuksien mallin, joka saa aikaan muutoksen järjestelmässä. (Wieringa 2014, 76)

Järjestelmän rakenteen abstraktoiminen arkkitehtuuriksi tuo monenlaisia hyötyjä. Kun järjestelmän arkkitehtuuri tiedetään, sen avulla voidaan jakaa järjestelmätason monimutkaiset ongelmat pienempiin osiin, jota ne ovat helpompia ratkaista. Arkkitehtuurikuvauksen avulla pystytään ymmärtämään ja suunnittelemaan järjestelmän luonnetta, löytämään siitä vikoja, organisoida sitä paremmaksi sekä tutkia sen yksittäisten komponenttien toimintaa. (Wieringa 2014, 78)

Mikropalveluarkkitehtuuri koostuu hajautetuista mikropalveluista, jotka ovat toistensa kanssa vuorovaikutuksessa. Mikropalveluarkkitehtuurissa on monenlaisia eroja perinteiseen arkkitehtuuriin nähden. Perinteisessä monoliittisessa kolmitaso-arkkitehtuurissa sovellus on jaettu kolmeen osaan: käyttöliittymäpuoleen, tietokantaan ja palvelinpuolen sovellukseen. Mikä tahansa päivitys monoliitin ohjelmakoodissa vaatii monoliitiin uudelleen rakentamisen ja käyttöönottamisen (Fowler ja Lewis 2014). Monoliitti-sovelluksien rakenteen monimutkaistuessa niiden modulaarisuus voi heikentyä, koska ajan saatossa entropia lisääntyy ja moduulien väleille voi kehittyä tiukkoja kytköksiä (Söylemez, Tekinerdogan ja Tarhan 2023). Monoliitin skaalautuminen tapahtuu aina replikoimalla koko sovellus uusille palvelimille. Mikropalveluarkkitehtuurissa jokainen palvelu on oma kokonaisuutensa, jota voidaan kehittää, testata, skaalata ja käyttöönottaa itsenäisesti (Söylemez, Tekinerdogan ja Tarhan 2023).

Jokaiselle palvelulle voidaan käyttää niiden tarkoitukseen parhaiten soveltuvia teknologioita (Söylemez, Tekinerdogan ja Tarhan 2023). Usein eri mikropalveluilla on omat kehitystiiminsä, minkä takia uusien työntekijöiden lisääminen projektiin on skaalautuvampaa (Villamizar ym. 2017). Yhden mikropalvelun muuttaminen, tai jopa sen kokonaan korvaaminen toisella, ei vaadi muiden palveluiden uudelleen rakentamista ja käyttöönottoa, kunhan rajapinnat pysyvät taaksepäin yhteensopivina. Sovellus pysyy modulaarisena kapseloinnin (Separation of concern principle) avulla, kunhan jokaiselle palvelut on rajattu selkeisiin liiketoimintalogiikan mukaisiin kokonaisuuksiin(Söylemez, Tekinerdogan ja Tarhan 2023).

2.2 AWS Lambda

Nykyään pivlilaskentapalvelut, kuten Microsoft Azure, Amazon AWS ja Google Cloud Platform tarjoavat myös palvelittomia FaaS (Function-as-a-Service) ratkaisuja, jotka ovat PaaS-palveluiden (Platform-as-a-Service) alakategoria. Palvelittomat ratkaisut tarjoavat palvelun käyttöasteeseen perustuvan automaattisen skaalauksen sekä palvelumaksun (Pay-for-value), mikä tekee niistä joustavia ja kustannustehokkaita vaihtoehdon tavallisille palvelinratkaisuille (“AWS Serverless Computing”, n.d). Toisin sanoen organisaation ei tarvitse maksaa palvelimen joutukäynnistä, jos sovellus ei käytä koko laskentakapasiteettiaan (“AWS Serverless Computing”, n.d). Pilvipalveluntarjoaja huolehtii palvelimiin liittyviästä infrastruktuurista, jolloin ohjelmistokehittäjät voivat keskittää aikansa ohjelmakoodin kirjoittamiseen (“AWS Lambda The Ultimate Guide”, n.d). Tutkimuksen (Villamizar ym. 2017) mukaan palvelittomat ratkaisut voivat vähentää jopa 70 prosenttia infrastruktuuriin kuluvia kustannuksia. Sana “palvelittomuus“ tuottaa monelle hämmennystä; FaaS-palvelut toimivat palvelimilla, mutta kehittäjien ei tarvitse huolehtia palvelimien toimminnasta tai niihin liittyvästä infrastruktuurista (Leitner ym. 2018).

Sovellus käyttää Amazonin AWS Lambdaa palvelittomana ratkaisuna. AWS Lambdan toiminta perustuu tapahtumapohjaisiin Lambda funktioihin (“AWS Lambda”, n.d). Lambda funktion ohjelmakoodi suoritetaan omassa eristetyssä kontissaan, ja funktiolle määritetään tietyn verran työmuistia ja CPU-kapasiteettia. Lambda funktiota voidaan suorittaa useita samanaikaisesti (“AWS Lambda The Ultimate Guide”, n.d) ja niiden tilat eivät vaikuta toisiinsa (“AWS Lambda”, n.d). Lambda funktiot voidaan määrittää aktivoitumaan tiet-

tyjen tapahtumien yhteydessä, kuten palveluun tulevista HTTP-pyynnöistä tai tietokanta-operaatioista. Yhden palvelun ohjelmakoodissa voi olla useita toisistaan eristyksessä olevia Lambda-funktioita, jotka aktivoituvat erilaisista tapahtumista.

FaaS-palveluiden käytössä on toisaalta myös haittapuolia. Ne ovat pilvinatiiveja ratkaisuja, eli sovellusta ei voi käyttää pilvipalvelualustan ulkopuolella, jonka takia se on riippuvainen pilvipalveluntarjoan tarjoamasta pilvi-infrastruktuurista (Vendor lock-in) (Leitner ym. 2018). Suuri osa ohjelmakoodista tulee uudelleenkirjoittaa migratoidessa sovellus toiseen pilvipalvelualustaan. IaaS-palveluita (Infrastructure-as-a-Service) hyödyntämällä kokonainen sovellus pystyttäisiin migratoimaan muuttumattomana pilvialustalle lift-and-shift tyylisesti esimerkiksi Docker-konteissa (Leitner ym. 2018). Tällöin Fullstack-sovellusta pystyttäisiin ajamaan lookaalissa ympäristössä virtuaalisesti, mikä helpottaa huomattavasti Fullstack-kehitystä ja päästä päähän -testaamista. PaaS-palveluita hyödyntävä sovellus pitää testata todellisessa pilvi-infrastruktuurissa, mikä tekee esimerkiksi päästä päähän -testauksesta vaivalloisempaa. Ohjelmistokehitys FaaS-palveluissa vaatii erilaisen ajattelutavan verrattuna perinteisempään ohjelmistokehitykseen, joten ohjelmistokehittäjien perehdyttämisessä uuteen työskentelytapaan voi kulua aikaa (Leitner ym. 2018). Palvelittomat ratkaisut ovat myös verrattaen uusi teknologia (Leitner ym. 2018), joten parhaista käytänteistä, hyödyistä, haitoista ei ole vielä kertynyt paljoa tutkimustietoa. Koska PaaS-ratkaisuja hyödyntävät tahot eivät hallinnoi palvelimia, se heikentää sovelluksen rakenteen läpinäkyvyyttä.

2.3 Amazon EventBridge

Kohdesovelluksessa palveluiden tapahtumapohjainen kommunikointi tapahtuu Amazonin EventBridge-tapahtumaväylän (Event bus) avulla. Tapahtumat lähetetään asynkronisesti tapahtumaväylään, joka hoitaa tapahtumien suodattamisen ja välityksen oikealle tapahtuman “kuluttajalle“ (Consumer) eli vastaanottajalle. Suodattimien avulla palvelut voivat suodattaa pois tapahtumat, jotka eivät ole niiden näkökulmasta kiinnostavia ja joihin niiden ei tarvitse reagoida. Amazon EventBridge on palveliton ratkaisu, eli palvelun hinta määräytyy käytön mukaan (Pay-as-you-go) ja pilvipalvelu hoitaa palvelininfrastruktuurin ja sen skaalautumisen automaattisesti (“Amazon EventBridge Documentation”, n.d).

Koodikatkelma 2.1: Esimerkkisääntö tapahtumien vastaanottamisesta ja ohjaamisesta palvelun sisällä.

```
new evt.Rule(this, 'add-linked-user-event-rule', {
  ruleName: `dataplatform-${props.environment}-add-linked-user-event-
     rule`,
  description: 'Rule matching all graphql add-linked-user events',
  eventBus,
  eventPattern: {
    source: ['akamon.dataplatform.api.graphql'],
    detailType: [
      'AddLinkedUser',
    ],
  },
  targets: [
    new targets.SqsQueue(addLinkedUserEventQueue, {
      deadLetterQueue: dlQueue,
      retryAttempts: 10,
    }),
  ],
});
```

Kuvan 2.1 koodikatkelmassa luodaan sääntö tapahtumien vastaanottamiseen, joka määrittelee, mitä tapahtumia tapahtumaväylästä vastaanotetaan ja minne ne ohjataan palvelun sisällä. `eventPattern` määrittää tapahtumien suodatussäännöt, eli millaisia tapahtumia säännöllä halutaan vastaanottaa. `targets` määrittää, minne tapahtumat välitetään.

Tapahtumat voivat sisältää liiketoiminnan kannalta tärkeää informaatiota, joka vastaanottavan palvelun on ehdottomasti käsiteltävä. Tällöin on kriittistä varmistaa, että tapahtuman sisältämä data ei katoa virhetilanteiden sattuessa. Entä jos tapahtumaa käsiteltäessä palvelussa tapahtuu virhe- tai poikkeustilanne, jolloin liiketoiminnan kannalta oleennainen prosessi ei suoriudu loppuun asti? Edellä mainittu ongelma on ratkaistu sovelluksessa käyttämällä Storage-first periaatetta. (Daly, n.d) kuvailee sivustollaan erilaisia periaatteita, joita voidaan hyödyntää palvelittomassa pilvipalveluratkaisuissa. Storage-first periaatteen oleennainen ajatus on, että palveluun tuleva data varastoidaan ennen kuin sitä aletaan prosessoi-

maan ohjelmakoodissa. Jos prosessointi epäonnistuu, alkuperäisen datan kopio on edelleen tallessa varastossa.

Vastaanotetut tapahtumat tallennetaan Amazon Simple Queue Serviceen (SQS). SQS varastoi tapahtumat jonomuotoiseen varastorakenteeseen. SQS lähettää vastaanotetun tapahtuman siihen kiinnitetylle Lambda-funktiolle, joka aloittaa Lambda-funktion ohjelmakoodin suorittamisen, jossa tapahtuma voidaan käsitellä. Jos tapahtuman prosessointi epäonnistuu, tapahtumaväylä yrittää lähettää uudelleen tapahtuman `retryAttempts`:in monta kertaa. Tapahtuman vastaanotto säännöissä voidaan määrittää Dead Letter Queue (DLQ), johon tapahtumat tallentuvat, jos tapahtuman käsittely tai vastaanottaminen epäonnistuu kaikilla uudelleenyritys kerroilla. DLQ varastoi käsittelemättömät tapahtumat, josta ne voidaan myöhemmin lähettää uudelleen SQS:lle, kun palvelussa tapahtunut vika on saatu korjattua. Tämä takaa sen, että kriittinen informaatio ei katoa palvelusta.

3 Dataohjatun järjestemän testaus

Tässä luvussa käydään läpi palvelittomien pilvipalveluiden testaukseen liittyvää tutkimustietoa, sekä niiden testaukseen liittyviä teollisuuden parhaita käytänteitä.

3.1 Palvelittomat pilvipalvelut

Palvelittomat pilvipalvelut ovat verrattaen uusi teknologia, jonka suosion aloitti AWS Lambda vuonna 2014 (Baldini ym. 2017). Koska ne ovat verrattaen uusi trendi, niihin liittyvä tutkimus on edelleen varhaisessa vaiheessa. Mikropalveluarkkitehtuureihin liittyvä tutkimus on harvoin sovellettavissa palvelittomiin pilvipalvelusovelluksiin, koska niiden käyttöönotossa hyödynnetään perinteisempiä pilvipalvelumalleja.

(Baldini ym. 2017) käy läpi palvelittoman pilvilaskennan historiaa, käyttötarkoituksia ja sen käyttöön liittyviä haasteita. Koska FaaS-sovelluksissa lyhyitä aikoja suoritettavia funktioita suoritetaan rinnakkain, se vaikeuttaa vikojen ja pullonkaulojen paikantamista. Koska palvelimia ei ole, funktion suorituksen aikana on kerättävä kaikki vaadittava data talteen pilvialustalle, jotta vikoja pystyttäisiin selvittämään.

(Soldani, Tamburri ja Heuvel 2018) tutkivat mikropalveluarkkitehtuurin hyötyjä ja haittoja tekemällä systemaattisen kirjallisuuskatsauksen, jossa he analysoivat harmaan kirjallisuuden lähteitä. Erityisesti suorituskyvyn testaaminen (Performance testing) tuottaa hankaluuksia mikropalveluarkkitehtuurissa; Koska sovellus on hajautettu useisiin itsenäisiin osiin, yksittäisten palveluiden suorituskykyä ja sovelluksen suorituskykyä kokonaisuudessaan on hankala mitata. Lokitiedot ovat pirstaloituneet eri palveluihin, mikä tekee vikojen paikantamisesta haastavaa. Toisaalta mikropalvelut ovat luonnostaan hyvin vikoja sietävä, koska viat pystytään yleensä isoloimaan yhteen palveluun, jolloin ne eivät vaikuta muiden palveluiden toimintaan.

(Leitner ym. 2018) tutkivat monimenetelmällisen empiirisen tutkimuksen avulla FaaS kehityksen teollisuuden käytänteitä. Tutkimuksen tuloksista ilmenee, että haasteeksi kehityksessä osoittautui etenkin integraatiotestaus, koska pilviympäristöä ei pysty replikoimaan lo-

kaalissa ympäristössä. Eräänä ratkaisuna ehdotetaan testaamista joko suoraan tuotantoympäristössä tai erillisessä testausympäristössä. Testausympäristön haittapuolena on se, että sen käytöstä joutuu maksamaan samalla tavalla kuin tuotantoympäristön käytöstä. Tuotantoympäristössä testaamisesta voi taas olla haitallisia sivuvaikutuksia tuotantoympäristöön, joiden lieventämiseen voidaan käyttää Canary-testausta tai A/B testausta. Suurin osa kyselyyn vastanneista ilmoitti suorittavansa testit kehitysympäristössä tai erillisessä testausympäristössä. Tutkimuksen mukaan toinen selkeä haaste FaaS-kehityksessä on työkalujen ja dokumentaation puutteellisuus sekä tietämättömyys teollisuuden parhaista käytänteistä. Työkaluissa puutetta löytyi ainakin käyttöönottoon, monitorointiin ja lokitukseen liittyvissä työkaluissa. Kyselyyn vastanneista suurin osa hyödynsi Serverless Framework -ohjelmistokehystä työkaluna (79.7 %). Tutkimuksessa kävi myös ilmi, että eräs hyöty palvelittomissa ratkaisuissa on parantunut tietoturva, koska palvelimien päivittäminen ja ylläpito on pilvipalveluntarjoajan vastuulla. Pilvipalvelualustat tarjoavat työkaluja (SDK ja CLI) ja muita resursseja edistämään palveittomien ratkaisujen kehitystä, testausta ja käyttöönottoa. Jotkin alustat tarjoavat orkestrointi-työkaluja (esimerkiksi AWS Step Functions), joiden avulla pystytään muodostamaan suurempia sovelluksia funktioista tai suorittamaan monimutkaisia työnkulkuja. FaaS-sovelluksille tarjotaan myös välineitä mittaus-, jäljentämis- ja lokitietojen keräämiseen (esimerkiksi AWS X-Ray). Virallisten työkalujen lisäksi on olemassa myös useita kolmansien osapuolien työkaluja, joita voidaan hyödyntää sovelluksen kehityksessä, kuten Serverless Framework.

Tutkimuksen (Hassan, Barakat ja Sarhan 2021) tavoitteena oli vastata useisiin palvelittoman pilvilaskentaa koskeviin kysymyksiin kirjallisuukatsauksen muodossa, jotta kehittäjät pystyisivät paremmin ymmärtämään palvelitonta pilvilaskentaa ja edistää sen kehitystä. Koska mikropalvelusovellusten tavoin palvelittomat sovellukset koostuvat useista funktioista, integraatiotestaaminen eli funktioiden yhteistoiminnan testaus on tärkeää. Tutkimuksen tuloksessa selviää, että testaamiseen, debuggaukseen ja suorituskyvyn mittaukseen liittyvät työkalut ovat puutteellisia tai epäkypsiä, jonka takia kehittäjät turvautuvat usein testaamaan pilviympäristössä. Etenkin työkalujen puutteellisuus nostaa kehittäjillä kynnystä siirtyä hyödyntämään palvelittomia ratkaisuja.

(Lenarduzzi ym. 2021) etsivät mahdollisia keinoja vähentää teknisen velan kerääntymistä

FaaS-sovelluksissa. Tutkimus toteutettiin kyselytutkimuksena, jossa haastateltiin alan asiantuntijoita. Vastauksissa testaamisen puutteellisuutta perusteltiin sillä, etteivät automaattiset työkalut tue integraatiotestausta hyvin, minkä takia sovelluksen liiketoiminnan kannalta kriittiset osat jäävät usein testaamatta. Koska integraatiotestejä on vaikea toteuttaa, sovellusta testataan usein manuaalisesti ajamalla sitä testausympäristössä tietyillä syötteillä. Testaamatta jättäminen lisää sovellukseen teknistä velkaa, jota yritetään kompensoida suurella määrällä lokitusta. Asiantuntijoiden vastauksien mukaan yksikkötestaamisessa eristyksissä olevien atomisten liiketoimintalogiikan yksiköiden identifioiminen on vaikeaa. Monitorointi- ja debuggaustyökalut koetaan myös rajallisiksi. Teknisen velan vähentämisen keinoina mainitaan muun muuassa yksitarkoituksisten funktioiden käyttämistä (Single-purpose functions) sekä heksagonaalisen arkkitehtuurin (Hexagonal architecture) soveltamista. Yksitarkoituksisia funktioita pystytään orkestroimaan esimerkiksi AWS Step Functionsin avulla.

Tutkimuksessa (Lenarduzzi ja Panichella 2021) käydään läpi FaaS-sovelluksien testauksen ja debuggauksen hyviä ja huonoja käytänteitä. Tutkimus toteutettiin kyselytutkimuksena, jossa haastateltiin alan asiantuntijoita. Vastauksissa suositellaan kääntämään testauspyramidi ylösalaisin FaaS-sovelluksissa ja keskittymään etenkin ”integraatiopisteisiin“, eli komponenttien välisiin kytköksiin, joihin suurin osa vioista usein keskittyy. Päästä päähän -testit suositellaan ajettavan pilviympäristössä, ja niiden tulisi kattaa kokonaiset liiketoimintaprosessit alusta loppuun. Kyselyn vastauksissa ilmeni useita FaaS-sovelluksien testaukseen liittyviä haasteita: Lokaali testausympäristö ei yleensä vastaa oikeaa pilviympäristöä, joten järjestelmätestaus tulisi suorittaa pilviympäristössä. Tapahtumaohjatuissa sovelluksissa haastavaa on testata useiden palveluiden välisiä prosesseja. Testien kattavuuden mittaamisessa on vaikeaa varmistaa, että kaikki funktion kuuntelemat tapahtumat testataan. Asynkroninen tapahtumapohjainen viestinvälitys vaikeuttaa testien verifiointia, sillä jotta voidaan verifioida tiettyjen tapahtumien julkaisut tapahtumaväylään, ne pitää taltioida tapahtumahetkellä kuuntelijan avulla.

Tutkimuksessa (Rinta-Jaskari ym. 2022) identifioidaan testaustapoja ja työkaluja AWS-Lambdaa hyödyntäville FaaS-sovelluksille ja toteutetaan yksikkö-, integraatio- ja päästä päähän -testit yksinkertaiselle Fullstack-sovellukselle. Sovelluksen backend-puolella hyödynnettiin Serverless ja Express-ohjelmistokehyksiä, ja ohjelmakoodi kirjoitettin Typescrip-

tillä Node.js-ajoympäristössä. Pilvi-infrastruktuurin käyttöönottamiseen ja hallinnoimiseen käytettiin Pulumi-ohjelmistokehystä. Yksikkö- ja integraatiotestit kehitettiin hyödyntämällä Jest-testauskehystä; Muina Typescriptiä tukevina testauskehykseksinä mainittiin Mocha ja Jasmine. Yksikkötestit testattiin Path-testing tekniikalla, jonka tarkoituksena on testata kaikki ohjelmakoodin suorituspolut kattavasti. Testisijaisten injektoimiseen hyödynnettiin Jestin Spy-ominaisuutta. AWS:n NoSQL DynamoDB-tietokantaan luotiin testisijainen käyttämällä `aws-sdk-mock` -kirjastoa. FaaS-arkkitehtuuri ei juurikaan vaikeuttanut yksikkötestien toteutusta. Integraatiotestit jaettiin kolmeen eri kategoriaan: lokaalit integraatiotestit, hybridi-integraatiotestit sekä pilvi-integraatiotestit. Pilvi-integraatiotestestit vastaavat eniten todellistä pilviympäristöä, joten niiden tulokset ovat kaikkein luotettavimpia. Integraatiotesteissä hyödynnettiin Big-bang testaustekniikkaa, jossa kaikki komponentit integroidaan ja testataan yhtenä yksikkönä. Integraatiotestien sovelluskutsut implementoitiin kahdella eri tavalla: kutsumalla suoraan Lambda-funktion käsittelijää ja käyttämällä APIen testaukseen tarkoitettua `supertest` -kirjastoa. Lokaalien integraatiotestien testisijaisissa käytettiin samoja tekniikoita kuin yksikkötesteissä. Hybridi-integraatiotesteissä testauksen apuna käytettiin pilvessä olevaa DynamoDB-tietokantaa. Päästä päähän- ja järjestelmä-testauksessa hyödynnettiin Cypress-kehystä. Vaihtoehtoisen testikehyksinä ja kirjastoina päästä päähän -testeille mainittiin Selenium, Nightwatch.js ja WebdriverIO.

Tutkimuksessa (Ricca ja Stocco 2021) tehtiin kirjallisuuskartoitus päästä päähän -testauksen parhaista käytänteistä käyttämällä harmaan kirjallisuuden lähteitä. Yli 2400 lähteestä valittiin 142 dokumentin joukko kirjoituksia, joiden mukaan Ricca ja Stocco loivat listan päästä päähän -testauksen parhaista käytänteistä. Parhaita testauskäytänteinä lueteltiin muun muuassa seuraavat:

- Tee testauskoodista mahdollisimman uudelleenkäytettävää.
- Käytä hyviä nimeämis- ja koodaamiskäytänteitä.
- Pidä testit atomisina.
- Käytä tynkä tai mock-versioita ulkoisista järjestelmistä.
- Kirjoita sekä positiivisia että negatiivisia testejä.
- Tee testit riippumattomiksi toisistaan.
- Ryhmittele testit loogisiin ryhmiin.

- Älä korvaa manuaalista testausta automaatiotestauksella.
- Valitse sopivat teknologiat ja ohjelmistokehykset.
- Testaa varhain ja testaa usein.
- Testaa loppukäyttäjän näkökulmasta.
- Hyödynnä DDT:tä (Data-driven testing).
- Raportoi testien tulokset yksityiskohtaisesti.

Voidaan todeta, että hajautettujen FaaS-sovelluksien testaamiseen liittyy monenlaisia haasteita ja niiden testaamisen parhaista käytänteistä ei ole selvyyttä. Testaus-, debuggaus-, käyttöönotto- ja monitorointityökalujen puutteellisuuden koetaan hankaloittavan testausta. Pilviympäristöä ei voida replikoida lokaalisti, joten usein integraatio- ja päästä päähän -testit suoritetaan erillisessä testausmpäristössä, mikä lisää pilvipalvelun käyttökustannuksia. Usein integraatiotestit jätetään tekemättä, koska niiden toteuttaminen on hankalaa, mikä kerryttää sovelluksen teknistä velkaa. Mikropalveluarkkitehtuurissa suorituskyvyn testaaminen ja vikojen paikantaminen on vaikeaa. Tapahtumien testaamisessa ne tulee ensin vastaanottaa ja varastoida pilvessä, jotta niiden oikeellisuus voidaan verifioida. Testisijaisten käyttö ja injektoiminen todellisissa pilviympäristöissä on vaikeaa. Käytännön esimerkkejä tai ohjeita monimutkaisten, hajautettujen ja tapahtumaohjattujen FaaS-sovelluksien testaamisesta ei juurikaan löytynyt.

3.2 Teollisuuden parhaat käytänteet

(“Automated testing”, n.d) ohjeistaa FaaS-sovelluksien testaamisessa. He ehdottavat testivaljaiden (Test harness) hyödyntämistä tapahtumaohjattujen FaaS-sovelluksien integraatiotestaamisessa. Testivaljaat ovat todellisia pilviresursseja, jotka luodaan pilviympäristöön helpottamaan testaamista. Testivaljaat jaetaan usein tuottajiin (Producers), jotka lähettävät syötteet testauskohteelle, sekä kuluttajiin (Consumers), jotka vastaanottavat tapahtumia. Kuluttajan vastaanottamat tapahtumat lähetetään tuottajalle, joka arvioi vastaavatko ne odotettuja tapahtumia. Asynkronisen arkkitehtuurin testeille suositellaan aikarajan asettamista, jonka puitteissa testiajo tulee suorittaa tai ne epäonnistuvat. Tapahtumapohjaisissa arkkitehtuureissa tapahtumien vastaanottajat olettavat, että tapahtumilla on skeema. Tämän takia niissä voi olla hyödyllistä kehittää sopimustestejä (Contract tests), joilla validoidaan, ettei tapahtumien

skeemamuutokset riko palveluiden välistä kommunikointia.

(“What Is An Ephemeral Environment?”, n.d) kuvaa lyhytaikaisten ympäristöjen (Ephemeral Environment) hyödyllisyyttä sovelluskehitysessä ja testauksessa. Lyhytaikaiset ympäristöt ovat sovelluksen lyhytaikaisia ja isoloituja käyttöönottoja. Niiden hyödyntämisen avulla kehittäjät voivat ajaa päästä päähän -testejä automaattisesti jokaisen commitoidun koodin jälkeen repositoriossa automaatioputkessa. Tällöin suurempi osa sovelluksen vioista havaitaan varhaisessa vaiheessa, ennen kuin ne päätyvät tuotantoympäristöön. Lisäksi kehittäjät voivat varmistaa sovelluksen toimivuuden regressiotestien avulla ennen koodikatselmointia, mikä helpottaa koodikatselmoijien työtä. Koska ympäristöt ovat isolaatiossa, kehittäjät voivat testata toiminnallisuuksia samanaikaisesti häiritsemättä toistensa kehitysprosesseja.

4 Tutkimusmenetelmä

Tässä luvussa esitellään suunnittelutieteellinen tutkimusmenetelmä ja perustelut miksi kyseistä menetelmää päädyttiin käyttämään tutkielmassa. Lisäksi esitellään artefakti, joka menetelmällä on tarkoitus tuottaa.

4.1 Suunnittelutieteellinen tutkimus

Tutkielmassa hyödynnetään suunnittelutieteellistä tutkimusmenetelmää. Suunnittelutieteessä tuotetaan innovatiivinen artefakti, jonka on auttaa jonkin ratkaisemattoman ongelman ratkaisemisessa (Hevner ym. 2004). Artefaktia arvioidaan sen perusteella, kuinka hyödyllinen se oli kyseisen ongelman ratkaisemisessa (Hevner ym. 2004). Huomioitavaa on, että osana tutkimusta tulisi edistää myös itse kehitysprosessia, jolla artefaki luotiin (Hevner ym. 2004). Kyseinen menetelmä valittiin tutkielmaan siksi, koska tutkimuksen tuloksista olevan konkreettista hyötyä organisaatiolle, ja suunnittelutieteen päämääränä on tuottaa hyötyä (Utility) (Hevner ym. 2004). Konkreettinen hyöty saavutetaan ratkaisemalla organisaation liiketoiminnan kannalta kriittinen ongelma, ja ratkaisun pohjalta luodaan yleispätevä artefakti.

4.2 Sidosryhmät

Tämän tutkielman sidosryhminä ovat organisaatio, jolle sovellus luodaan, Jyväskylän yliopisto (Pro-gradun julkaisija ja hyväksyjä), organisaation kehittäjät (Toteuttavat päästä päähän -testejä) sekä organisaation asiakkaat ja loppukäyttäjät (Sovelluksen hyödyntäjät ja käyttäjät).

4.3 Kehitetyn artefaktin tavoitteet

Artefaktin tarkoituksena on vastata seuraavaan tutkimuskysymykseen:

Miten päästä päähän -testit voidaan toteuttaa automaatioputkessa hajautetussa pilvipohjaisessa järjestelmässä siten, että testiskriptien testausympäristö olisi mahdollisimman lähellä

tuotantoympäristöä ja että testeissä voidaan simuloida mahdollisimman aitoja liiketoimintamalleja?

Tutkielmassa toteutetaan päästä päähän -testausta varten toimiva ohjelmistokehys organisaation sovellukselle. Toteutuksen pohjalta luodaan kuvaus, jossa kuvaillaan toteutuksen luonnetta ja perustellaan siinä tehtyjä ratkaisuja ja käytettyjä teknologioita. Tämä kuvaus toimii tutkielman artefaktina. Artefaktista toivotaan olevan hyötyä niille kehittäjille, jotka pyrkivät toteuttamaan päästä päähän -testejä samankaltaisille sovelluksille.

Jotta artefaktin onnistumista tavoitteissaan pystytään arvioimaan, tulee olla arviointikriteereitä ja vaatimuksia joihin tuotettua artefaktia voidaan peilata. Suunnittelutieteessä artefaktia pyritään kuvaamaan ja arvioimaan “matemaattisen formaalisesti“ ja täsmällisesti (Hevner ym. 2004). Toisaalta, täsmällisistä tutkimuksista ei välttämättä seuraa käytännön hyötyä reaalimaailmassa (Lee 1999), ja liiallinen täsmällyys saattaa abstraktoida ongelmaa liikaa, jotta sen tuloksia voisi hyödyntää (Hevner ym. 2004). “Tutkimuksen täsmällisyys“ (Research rigor) ei ole tarkasti määritelty informaatiotieteen alalla, ja sen määritelmä vaihtelee tutkijasta toiseen (Soliman ja Siponen 2022). Artefaktin arvioinnissa tulee ottaa huomioon, että arviointiperusteet tulisi olla tarpeeksi formaaleja tutkimuksen näkökulmasta, mutta toisaalta niiden pitää myös perustua käytännön tarpeisiin.

Tässä tutkielmassa artefakti arvioidaan toimivan totetuksen pohjalta, jonka perusteella artefakti luotiin. Päästä päähän -testien ohjelmistokehyksen arviointikriteerit saatiin organisaatiolta, jolle testit toteutetaan. Toteutusta arvioidaan toteutukselle määriteltyjen toiminnallisten ja laadullisten vaatimuksien pohjalta. Lisäksi toteutukseen toteutetaan testiskripti yhdelle sovelluksen käyttötapaukselle, jonka toimivuus on organisaation liiketoiminnan kannalta tärkeää. Toteutetun käyttötapauksen tarkoituksena on todentaa vaatimuksien toteutuminen ja toteutuksen hyödyllisyys käytännössä.

4.3.1 Vaatimukset

Toteutukselle asetetaan vaatimuksia, jotka varmistavat, että sen ominaisuudet vastaavat organisaation asettamiin tarpeisiin. Vaatimukset voidaan jakaa toiminnallisiin ja laadullisiin vaatimuksiin (Wieringa 2014, 54). Toiminnalliset vaatimukset kuvaavat tuotteen toivottu-

ja toiminnallisuuksia, esimerkiksi “Testit voidaan ajaa automaattisesti“, kun taas laadulliset vaatimukset kuvaavat tuotteen laadullisia piirteitä, esimerkiksi “Testejä on käyttäjäystävällistä kirjoittaa“.

Toiminnalliset vaatimukset:

- Päästä päähän -testit ajetaan automaattisesti CI/CD-putkessa, mutta ne tulee voida ajaa myös manuaalisesti pilviympäristössä sekä lokaalissa ympäristössä.
- Tuki samanaikaisille testiajoille (joko jonotusjärjestelmällä tai tuki rinnakkaisille testiajoille).
- Testiskripteissä tulee voida kutsua sovelluksen rajapintojen asiakasohjelmia sekä rekisteröityä vastaanottamaan sovelluksen tapahtumia.
- Testien testidata ja lähtötila tulee olla alustettavissa.
- Testiajo tulee suorittaa eristetyssä testausympäristössä.
- Toteutuksessa pitää olla mahdollista tehdä sekä positiivisia- että negatiivisia testejä.
- Ulkoisten järjestelmien rajapintakutsut tulee olla korvattavissa ja testattavissa testisijaisilla.

Laadulliset vaatimukset:

- Toteusta voidaan käyttää tekemään testitapauksia useille sovelluksen käyttötapauksille.
- Testiskriptien kirjoittamisen tulee olla mahdollisimman helppoa ohjelmistokehittäjän näkökulmasta.
- Testausympäristön tulee olla mahdollisimman lähellä tuotantoympäristöä.

4.3.2 Käyttötapaus

Yllä mainittuja vaatimuksia testataan konkreettisesti toteuttamalla päästä päähän -ohjelmistokehyksellä testiskripti käyttötapauksen 1 pohjalta.

Käyttötapaus Asiakaskohtainen kiinteän hinnan tarjouksen laskenta ja sopimuksen hyväksyntä.

Toimija Asiakas.

Ennakkoehto Asiakkaalla on sähkösopimus jossakin yhtiössä.

Loppuehto Asiakkaalla on hyväksytyn tarjouksen mukainen uusi sopimustilaus.

Kuvaus Asiakkaan suorittamat toiminnot:

1. Asiakas käy antamassa oma.datahub.fi - sivustolla valtuutuksen omaan käyttöpaikkaansa sovellukselle, joka tukee tarjouslaskuria (kohdesovelluksen ulkopuolella).
2. Asiakas kirjautuu sisään tarjouslaskuriin.
3. Palvelu näyttää asiakkaan 1. kohdassa valtuuttamat käyttöpaikat listalla.
4. Asiakas valitsee halutun käyttöpaikan ja valitsee alkupäivän mikä tulee olemaan mahdollisen luotavan sopimuksen alkupäivä.
5. Tarjouslaskuri käynnistyy ja lataus-indikaattori näkyy näytöllä.
6. Asiakas näkee ETRM:n laskemat tarjousvaihtoehdot.
7. Asiakas valitsee tarjousvaihtoehdoista haluamansa ja siirrytään sopimustilauslomakkeelle.
8. Tilauslomakkeella asiakas syöttää omat tietonsa (Laskutusosoite, jne.). Kun tilauksen tiedot ovat kunnossa, asiakas klikkaa ”Tallenna“-nappia.
9. Sopimustilaus luodaan Dataplatformiin ja asiakkaalle viestitetään käyttöliittymällä, että tilaus on vastaanotettu.
10. Asiakas voi kirjautua ulos ja sulkea selaimen.
11. Asiakas saa hetken päästä sähköpostiin sopimustilauksen vahvistuksen.

Kuvio 1: Toteutettavan käyttötapauksen kuvaus.

5 Toteutus

Tässä luvussa kuvaillaan yksityiskohtaisesti toteutetun päästä päähän -testien ohjelmistokehyksen ominaisuuksia ja siinä käytettyjä teknologioita.

5.1 Yleiskuvaus

Kaikki ohjelmakoodi toteutettiin Typescript-ohjelmointikielellä Node.js-ajoympäristössä. Peruste Typescriptin valinnalle oli se, että sitä käytetään entuudestaan kohdesovelluksen palveluissa, joten se on yhteensopiva sovelluksen kirjastojen kanssa. Lisäksi AWS CDK ja AWS SDK tukevat Typescriptiä hyvin.

Jokainen testitapauksen suoritus käyttää samaa ”kovakoodattua“ tenanttia, koska sovellus ei tue tenanttien luontia automaattisesti. Sovelluksessa tenantit ovat asiakasorganisaatioita, joiden käyttäjä-pooliin voidaan lisätä Amazon Cognito-käyttäjiä. Niiden avulla organisaation asiakkaat voivat kirjautua asiakassovelluksiin ja hyödyntää sovelluksen tarjoamia palveluita.

Päästä päähän -testit toteutettiin omaan repositorioonsa `dataplatform-e2e-tests`, jonka kansiorakenne on seuraavanlainen:

```
/
├── infra
│   ├── bin
│   ├── lib
│   ├── scripts
│   └── src
└── dataplatform
    ├── bin
    ├── e2e-plugin
    ├── e2e-context
    └── e2e-tests
```

Repositoriossa on kaksi erillistä Node.js-projektia, `infra` ja `dataplatform`. `infra`-projektissa mallinnetaan testejä varten tarvittava pilvi-infrastruktuuri ohjelmakoodilla käyttäen AWS CDK:ta. `dataplatform`-projektissa ajetaan päästä päähän testit käyttäen Japa.dev-ohjelmistokehystä.

5.2 CDK-infrastruktuuri

`Infra`-projektissa mallinnetaan pilvi-infrastruktuuri ohjelmakoodilla käyttäen AWS Cloud Development Kittiä (AWS CDK). CDK on avoimen lähdekoodin ohjelmistokehys, joka mahdollistaa pilvi-infrastruktuurin määrittelyn koodissa (Infrastructure as code, IaC) ja sen provisionoinnin AWS CloudFormationin kautta ("What is the AWS CDK?", n.d). Kun infrastruktuuri määritellään ohjelmakoodissa, ohjelmistokehittäjät voivat soveltaa sen kehityksessä ohjelmistokehityksen yleisiä käytänteitä, kuten versionhallintaa. Tämä takaa sen, että syntynyt infrastruktuuri on aina toistettavissa, johdonmukainen ja luotettava ("Infrastructure as code", n.d). CDK tukee moina yleisesti käytettyjä ohjelmointikieliä, kuten Typescriptiä, Pythonia ja C#:ia.

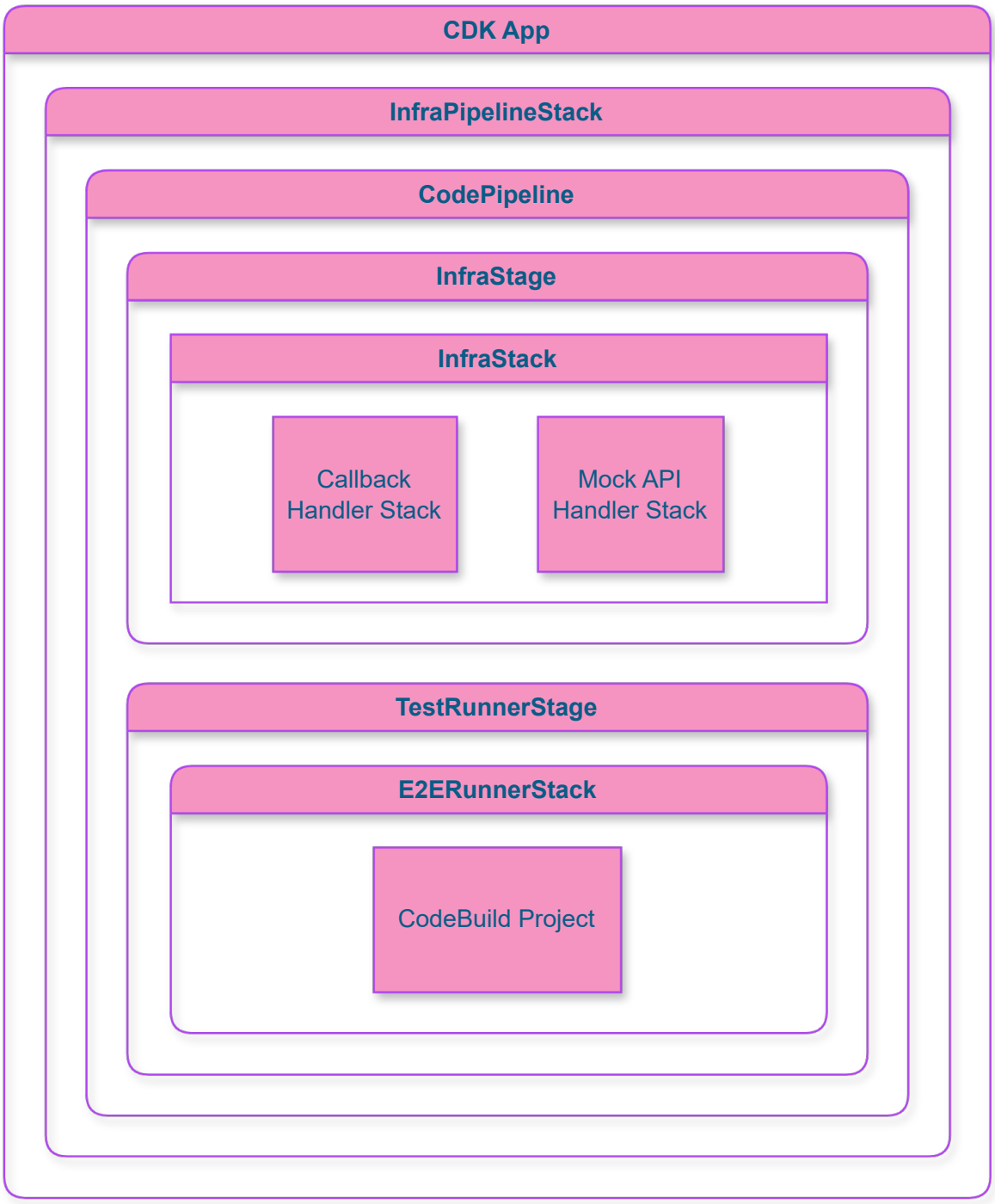
Pilvi-infrastruktuuri luodaan hierarkisesti AWS Construct -kirjaston Construct-luokan instanssien avulla. CDK-sovellus luodaan käyttämällä App constructia, joka toimii CDK sovelluksen pohjana. CDK-sovellus (CDK App) sisältää yhden tai useampia pinoja (Stack). Pinot taas sisältävät kokoelman rakenteita (Construct), jotka määrittelevät AWS pilviresursseja ja niiden ominaisuuksia. ("AWS CDK Concepts", n.d)

Kuviossa 2 esitetään infra-projektin CDK-sovelluksen hierarkinen rakenne. CDK sovellus koostuu yhdestä pinosta, `InfraPipelineStack`, jonka sisällä määritellään CI/CD-prosessissa käyttöönotettavat pilviresurssit. CI/CD-putkella on kaksi käyttöönoton vaihetta, `InfraStage` ja `TestRunnerStage`, jotka ajetaan järjestyksessä CI/CD-putkessa. CI/CD-putken ajon aikana `InfraStagessa` käyttöönotetaan pilviresurssit, joita tarvitaan päästä päähän-testien ajon aikana testivaljaina: `MockAPI` ja `CallbackHandlerAPI`. `TestRunnerStagessa` käyttöönotetaan AWS CodeBuild-projekti, jossa päästä päähän -testit, eli `dataplatform`-projektikansion ohjelmakoodit, suoritetaan pilviympäristössä. Testiskriptit ajetaan automaattisesti kerran päivässä ja aina silloin kun sovelluksen pinoja päivitetään (Koodikatkelma 5.1).

Koodikatkelma 5.1: Testiohjelman automaatiokäynnistyksen kriteerit.

```
new Rule(this, 'Schedule-Tests-Rule', {
  schedule: Schedule.cron({minute: '0', hour: '0'}),
  targets: [new CodeBuildProject(e2eTests)],
});

new Rule(this, 'Start-Tests-Rule', {
  eventPattern: {
        source: [
          'aws.cloudformation',
        ],
        detailType: [
          'CloudFormation Stack Status Change',
        ],
        detail: {
          'stack-id': [
                'dataplatform-app-etrm',
                'dataplatform-api-rest',
                'dataplatform-api-graphql',
                'dataplatform-service-contract',
                ...
                'dataplatform-e2e-tests',
          ],
          'status-details': {
                status: [
                  'CREATE_COMPLETE',
                  'UPDATE_COMPLETE',
                ],
          },
        },
  },
  targets: [new CodeBuildProject(e2eTests)],
});
```



Kuvio 2: CDK-sovelluksen hierarkinen rakenne.

5.3 Arkkitehtuurin kuvaus

Kuviossa 4 esitetään päästä päähän -testien ohjelmistokehyksen pilvi-infrastruktuurin komponentit eli pilviresurssit sekä niiden väliset kytkökset. Kuviossa nuolien suunta kuvastaa HTTP-kutsujen tai tapahtumien kulkusuuntaa komponenttien välillä. Jos nuoli osoittaa komponentista A komponenttiin B ($A \rightarrow B$), se tarkoittaa, että A on kutsun tai tapahtuman lähettäjä ja B on sen vastaanottaja. Kaksisuuntainen nuoli tarkoittaa, että molemmat komponentit sekä lähettävät että vastaanottavat kustuja tai tapahtumia toisiltaan.

Kuviossa 4 `CodeBuild Project` on ohjelma, jossa päästä päähän -testiskriptit suoritetaan. Testiskripteissä kutsutaan ja vastaanotetaan kutsuja sovelluksen rajapinnoista, eli testeissä käytetään samoja rajapintoja kuin sovelluksen loppukäyttäjät käyttävät. ETRM API ja REST API ovat molemmat REST-rajapintoja, ja GraphQL API hyödyntää nimensä mukaisesti GraphQL-rajapintaa. REST API on PaaS-rajapinta, jota asiakasorganisaatiot eli tenantit käyttävät palvelualustana, ja siinä autentikointi tapahtuu tenant-kohtaisten API-avaimien avulla. ETRM API ja GraphQL API ovat SaaS-rajapintoja, joissa autentikaatio tapahtuu token-pohjaisesti. Kun rajapinnat vastaanottavat kutsuja, ne lähetetään tapahtumaväylän välityksellä EventBirdgeen, jolloin muut palvelut voivat vastaanottaa ja käsitellä niiden tietosisällön asynkronisesti. Tapahtuman sisällön mukana lähetetään `TenantId`, jonka perusteella tapahtuman vastaanottamat palvelut tietävät, mille tenantille tapahtuman tietosisältö kuuluu, sekä `CustomerId`, jos kutsu tuli asiakkaalta.

Jotta päästä päähän -testeissä ennakoitua dataa voidaan verrata vasteisiin, on tärkeää että testiskripteissä pystytään rajapinta-kutsujen lisäksi myös kuuntelemaan kohdesovelluksessa tapahtuvia tapahtumia. Koska sovellus toimii asynkronisesti ja tapahtumapohjaisesti, tapahtumia vastaanottamalla ja vertaamalla niiden tietosisältöä ennakoituun dataan voidaan varmistaa, että odotetut toiminnot ovat tapahtuneet odotetusti sovelluksen palveluissa. Tapahtumia kuuntelemalla voidaan esimerkiksi varmistaa, aloitettiinko tarjouslaskenta onnistuneesti tai tallentuiko tietosisältö tietokantaan onnistuneesti. Tapahtumien kuuntelu helpottaa vian paikantamista pitkissä tapahtumaketjuissa, koska silloin voidaan paikantaa, mikä tapahtuma ketjussa ei toiminut odotetulla tavalla. Huomioitavaa on, että sovelluksen käyttäjät pystyvät kuuntelemaan vain omia tapahtumiaan, toisin sanoen niitä tapahtumia, jotka ovat lähtöisin heidän omista rajapinta-kutsuistaan. Kaikkiin rajapintakutsuihin liitetään mukaan

`correlationId`, joka on sama jokaisella rajapintakutsusta lähtöisin olevasta tapahtumalla, jolloin sen avulla voidaan etsiä koko asynkronisten tapahtumien ketju sovelluksesta.

ETRM- ja GraphQL-rajapinnoissa tapahtumien kuuntelu tapahtuu websokettien avulla. Rajapintojen kautta on mahdollista “tilata” (Subscribe) tapahtumia, ja rajapinta välittää viestinä tapahtuman asiakkaan websokettiin sen tapahtuessa. REST APIssa tapahtumien kuuntelemiseen ei käytetä websoketteja, vaan tilaamiskutsussa annetaan webhook-osoite, jonne rajapinta lähettää tapahtuman kutsuna sen tapahtuessa.

Koska REST APIssa tapahtumien tilaaminen tapahtuu websokettien sijaan webhook-osoitteella, kaavion 4 mukaisesti tarvitaan erillinen Callback API, joka vastaanottaa REST APIn kutsuja ja tallentaa niiden tietosisällön väliaikaisesti tietokantaan. Testiskripteissä Callback APIn kautta voidaan hakea ja verifioida vastaanotetut tapahtumat DynamoDB-tietokannasta.

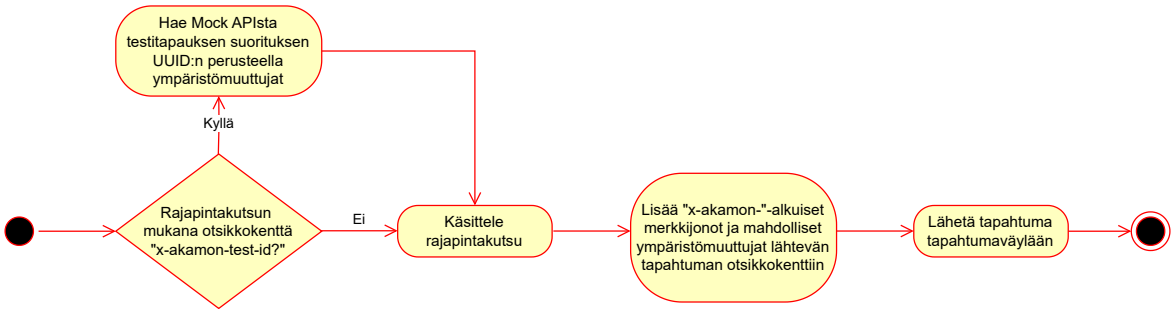
Kuten aiemmin todettiin, eräs haaste päästä päähän -testaamisessa todellisessa pilviympäristössä on ulkoisten palveluiden korvaus testisijaisilla (”mock“- tai tynkäversioilla). Toteutuksessa ulkoisten palveluiden korvaus testisijaisilla tapahtuu kaaviossa 4 näkyvän Mock APIn avulla. Testiskriptissä Mock APIin tehdään alustuskutsu, jolla alustetaan Mock API lähettämään halutunlainen testidata määritellylle rajapintakutsulle. Tämä testidata tallennetaan DynamoDB-tietokantaan odottamaan kutsun saapumista. Kun kohdesovelluksessa kutsutaan ulkoista palvelua, kutsu uudelleenohjataan Mock APIin, joka hakee DynamoDB-tietokantaan tallennetun testidatan ja lähettää sen vastauksena palvelulle. Kaaviossa 4 ”Palvelu“ on kohdesovelluksen mielivaltainen palvelu, jonka ulkoiseen palveluun tekemä kutsu ohjataan Mock APIin.

Koska päästä päähän -testaus tapahtuu oikeassa pilviympäristössä, jota päästä päähän -testiajojen lisäksi käyttävät oikeat asiakkaat ja organisaatiot, on toteutettava toimintalogiikka, joka erottaa testikutsut ja todellisten käyttäjien tekemät kutsut toisistaan siten, että vain testitapauksissa ulkoisten palveluiden kutsut ohjataan Mock APIin. Toisaalta, koska yhtenä vaatimuksena oli se, että testejä pitää pystyä ajamaan myös rinnakkain, pitää olla myös mahdollista erottaa yksittäisten testitapausten suorituksien rajapintakutsut ja niihin liittyvät tapahtumat toisistaan, vaikka kyseessä olisi organisaatiotason kutsut REST APIn kaut-

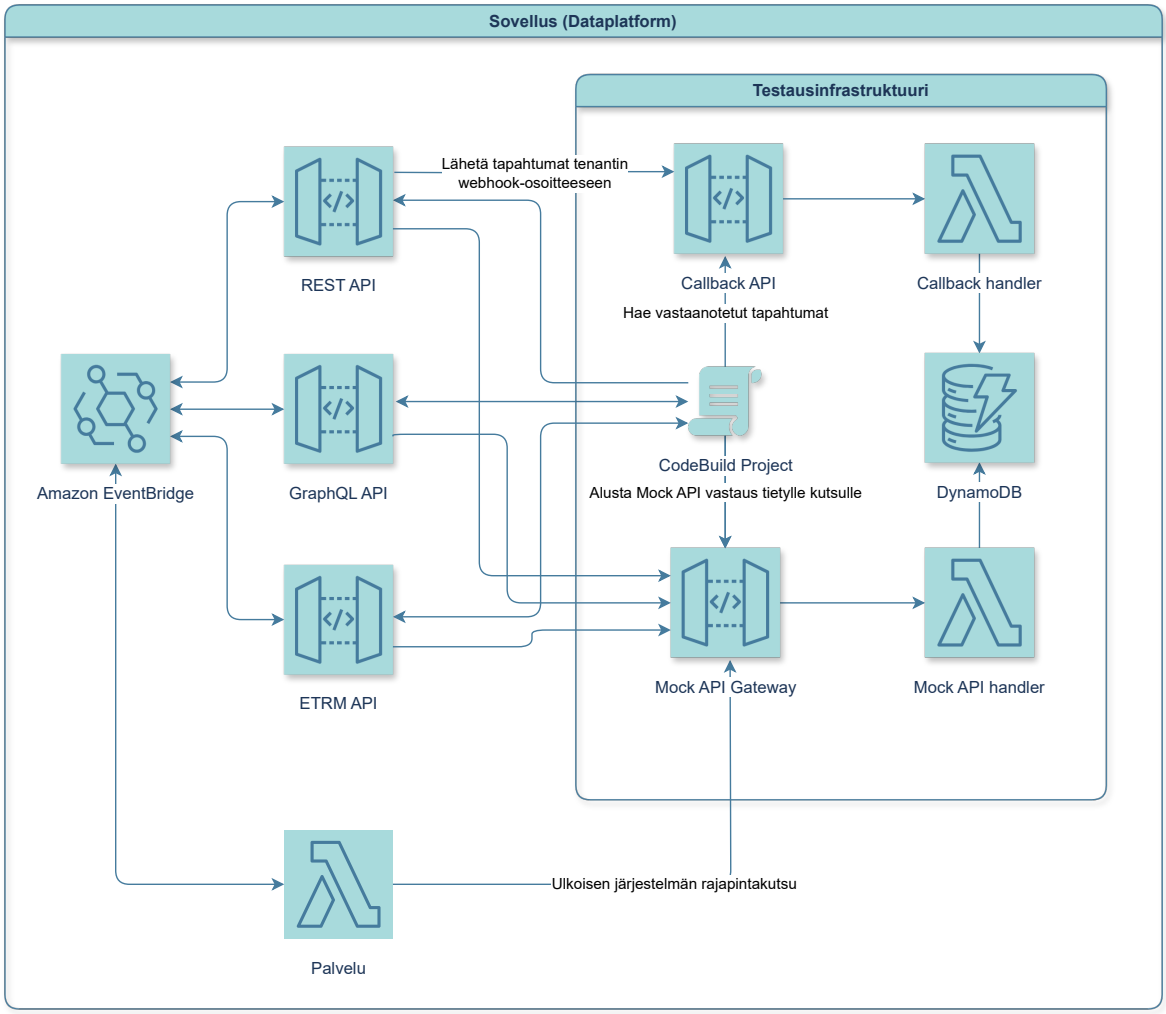
ta. Tämä toiminto toteutettiin siten, että jokaiseen testiskriptin suoritukseen liitetään uniikki UUID-tunniste (A Universally Unique Identifier), ja kyseinen tunniste lähetetään rajapintakutsun otsikkokentässä `x-akamon-test-id` sovellukseen. Tämä uniikki tunniste lähetetään `correlationId`:n tavoin tapahtumien mukana ”alavirtaan“ muille sovelluksen palveluille. Kun REST APIn lähettää kuunnellun tapahtuman kutsuna Callback APIlle, sen mukana lähetetään `x-akamon-test-id`, jonka avulla kuunneltavat tapahtumat voidaan erotella testitapausten suorituksien mukaan.

Toteutuksessa tapahtumatyyppiin `DataplatformEvent` lisättiin ominaisuus (Property) `headers`, joka on tyypiltään sanakirja, joka sisältää avain-arvo -pareja, jossa sekä avain että arvo ovat tyypiltään merkkijonoja. Rajapintakutsujen mukana tulevat otsikkokentät, jotka alkavat merkkijonolla `x-akamon-`, lisätään lähtevien kutsujen tai tapahtumien otsikkokenttiin. Muut otsikkokentät suodatetaan pois tietoturvasyistä, ettei tapahtumien mukana kulje turhaan http-kutsujen otsikkokenttien arkaluontoista tietoa, kuten autentikointitokeneita.

Ulkoisiin palveluihin tehtävien kutsujen uudelleenohjaus Mock APIin tapahtuu korvaamalla ulkoiseen palveluun osoittava ympäristömuuttuja Mock APIn osoitteella palvelussa, jossa kutsu tehdään. Korvaavat ympäristömuuttujat lähetetään tapahtumien tai rajapintakutsujen otsikkokentissä sovelluksessa sisäisesti palvelulta toiselle `x-akamon-env-` -alkuisissa otsikkokentissä. Palvelussa, jossa ulkoinen rajapintakutsu tehdään, ulkoisen palvelun osoite korvataan samannimisellä ympäristömuuttujalla, joka saadaan tapahtuman otsikkokenttien mukana. Tietoturvasyistä ympäristömuuttujia ei voida vastaanottaa julkisten rajapintojen otsikkokenttien mukana: pahantahtoinen tekijä voisi kokeilemalla keksiä ympäristömuuttujan ja uudelleenohjata kutsuja omaan palveluunsa, jolloin hän voisi saada arkaluontoista dataa, kuten API-avaimia, kutsujen mukana. Tämän takia ympäristömuuttujia ei oteta julkisten rajapintojen kautta, vaan ne haetaan sisäisesti GraphQL APIn, Rest APIn ja ETRM APIn käsittelijäfunktioissa Mock APIsta (kaavioiden 3 ja 4 osoittamalla tavalla). Lisäksi Mock API sijaitsee Amazonin Virtual Private Cloudissa, eli Mock APIia ei voida kutsua sovelluksen ulkopuolelta, joka takaa sen ettei ulkopuoliset tahot pysty väärinkäyttämään rajapintaa sovelluksen ulkopuolelta.



Kuvio 3: Ympäristömuuttujien haku rajapintojen käsittelijäfunktioissa.



Kuvio 4: Kuvaus pilviresurssien välisestä HTTP- ja tapahtumapohjaisesta viestinnästä.

5.4 Testiohjelma

5.4.1 Japa-ohjelmistokehys

Testien ohjelmistokehyksenä käytettiin toteutuksessa Japaa. Japa on kevyt ohjelmistokehys, joka on tarkoitettu ainoastaan backend-sovelluksien testaamiseen, mikä tekee siitä kevyemmän ja nopeamman kuin useat muut testausohjelmistokehykset, koska backend-koodia ei tarvitse transpiloida vanhemman Javascript-version koodiksi samalla tavoin kuin frontend-koodia (“Japa.dev documentation”, n.d). Japa tukee myös Typescript-ohjelmointikieltä, jota projekteissamme käytetään. Japa on suhteellisen uusi ohjelmistokehys, joka on julkaistu vuoden 2021 loppupuolella npm-paketinhallintatyökaluun. Viikottaisia latauksia kirjastolla on noin kuusituhatta (2024, viikko 15), mikä on suhteellisen vähän verrattuna esimerkiksi suositun Jest-ohjelmistokehyksen latausmääriin (noin 23 miljoonaa latausta samalla viikolla). Vaikka Japa ei ole suosituin ohjelmistokehys, se on osoittautunut hyväksi ja toimivaksi valinnaksi backend-sovelluksen päästä päähän -testaamiseen. Japa tukee kaikkia yleisiä testauskehyksien toiminnallisuuksia, kuten assertointia, asynkronisia testejä, kattavuusraportointia sekä testien ryhmittelyä ja suodatusta.

Japaan voi lisätä mukautettuja liitännäisiä (Plugins), jotka ovat Javascript-funktioita, jotka suoritetaan aina ensimmäisenä ennen testin suorittamista. Ne voivat muokata konfiguraatiota, muuttaa CLI-lippuja, rekisteröidä elinkaari-hookeja ja laajentaa luokkia. Toteutuksessa käytetään Japan virallisesti tuettua pluginia `@japa/assert` assertoimiseen testeissä. Lisäksi toteutuksessa käytetään mukautettua liitännäistä `endToEnd`, jonka tarkoituksena on toimia päästä päähän -testaamista helpottavana ohjelmistokehyksenä.

Koodikatkelma 5.2: Tiedostoa bin/test.ts käytetään aloitustiedostona Japa:n konfiguroimiseen ja testien suorittamiseen.

```
import {assert} from '@japa/assert';
import {expectTypeOf} from '@japa/expect-type';
import {configure, processCLIArgs, run} from '@japa/runner';
import {endToEnd} from '../e2e-plugin/e2e-plugin';
import {dataplatformContextBuilder} from '../e2e-context/dataplatform-
    context/dataplatform-context-builder';
```

```
processCLIArgs(process.argv.splice(2));

configure({
  suites: [
    {
      name: 'E2E',
      files: 'e2e-tests/**/*.spec.ts',
      timeout: 1000 * 60 * 60 * 6, // 6 hours timeout
    },
  ],
  plugins: [
    assert(),
    expectTypeOf(),
    endToEnd(dataplatformContextBuilder),
  ],
});

void run();
```

5.4.2 Mukautettu liitännäinen päästä päähän -testaukseen

Toteutuksessa päädyttiin tekemään mukautettu liitännäinen, joka toimii ohjelmistokehyksenä päästä päähän -testeille. Syy käyttää mukautettua liitännäistä oli se, että sen avulla pystytään tekemään testien kirjoittamisesta helpompaa; Liitännäisissä elinkaari-hookien avulla pystytään automaattisesti poistamaan päästä päähän -testin aikana luodun resurssit ja tilat. Esimerkiksi testiskripteissä luodut Cognito-käyttäjät voidaan automaattisesti poistaa testin suorituksen jälkeen ilman, että testiskriptin toteuttajan tarvitsee manuaalisesti kutsua erillisiä siivousfunktioita testiskriptissä. Tämä tekee testien kirjoittamisesta helpompaa ja käyttäjäystävällisempää.

Japassa jokaisen testin mukana parametrina jaetaan `TestContext`-luokan instanssi. Jokaiselle testille luodaan uusi `TestContext`-instanssi, mikä takaa sen, etteivät testien tilat ja ominaisuudet vaikuta toisiinsa (“Japa.dev documentation”, n.d). `TestContext`-luokan ominaisuuksien kautta voidaan käyttää liitännäisten avulla luotuja ominaisuuksia; Esimerkiksi `@japa`

`/assert`-liitännäinen lisää `TestContext`-luokkaan `assert`-ominaisuuden, jonka kautta assertointikirjaston ominaisuuksia pystytään testeissä käyttämään.

Toteutetussa liitännäisessä kaikki päästä päähän -testaukseen liittyvät ominaisuudet asetettiin `TestContext`-luokan `endToEnd`-ominaisuuden sisälle. Tämä takaa sen, etteivät päästä päähän -testeihin liittyvät ominaisuudet sekoitu muiden liitännäisten kanssa. Lisäksi se tekee testien kirjoittamisesta helpompaa, koska päästä päähän -tesiin liittyviä toiminnallisuuksia ei tarvitse tuoda erikseen omassa moduulissaan testeihin, vaan ne tulevat automaattisesti `TestContext`-instanssin mukana testiin.

Mukautetun liitännäisen toteuttamisen hyötynä on myös se, että sitä on mahdollista uudelleenkäyttää myös muiden sovelluksien päästä päähän -testauksessa. Lisäksi yleisesti käytettävän ja sovelluskohtaisen ohjelmakoodin erottaminen tekee koodin rakenteesta selkeämmän: Ennen yleisesti käytettävän ja kohdesovellukseen liittyvän ohjelmakoodin erottamista koodi oli merkittävästi vaikeampilukuisempaa, ja kun yleisesti käytettävä koodi oli siirretty omaan moduuliinsa liitännäisenä, ohjelmakoodin rakenteesta tuli helpommin ymmärrettävä.

Toteutettu liitännäinen hyödyntää rakentaja-suunnittelumallia (Builder pattern) päästä päähän -testien kontekstin luomiseen. Kontekstilla tarkoitetaan sen kohdesovelluksen rakennetta ja ominaisuuksia, jolle päästä päähän -testit toteutetaan. Toisin sanoen rakentaja-oliolla rakennetaan konteksti, jonka avulla pystytään käyttämään kohdesovelluksen rajapintoja testiskripteissä. Rakentaja-suunnittelumallin avulla pystytään ykinkertaistamaan monimutkaisten olioiden luomisen ohjelmakoodi pieniin askeliin ja se toimii ohjelmistokehyksenä liitännäisessä testikontekstin luomiselle, eli se tarjoaa helppokäyttöisen ohjelmistokehyksen mukautetun liitännäisen käyttämiseen.

Rakentaja-suunnittelumallin avulla pystytään myös luomaan automaattisesti Typescript-tyyppi päästä päähän -testien kontekstille, mikä tekee liitännäisen käyttämisestä helpompaa, koska liitännäisen käyttäjän ei itse tarvitse määritellä tyyppiä. Typescriptissä tyypitys on staattista, toisin sanoen tyypitykset määritellään käännösaikana (lukuunottamatta **`any`**-tyyppiä), eli muuttujien tyyppejä ei voida määrittää uudelleen ohjelmakoodin ajon aikana. Rakentaja-suunnittelumallia hyödyntämällä on mahdollista saada Typescript päättelemään rakennettavan kontekstin tyyppi käännösaikana. Rakentaja-mallissa rakentajalle annetaan as-

kelittain ohjeita, joiden perusteella se voi luoda olio-instansseja. Rakentajalle annettavia ohjeita voidaan ketjuttaa (Esimerkiksi `createBuilder().addA().addB().addC()`), jos metodit palauttavat paluuarvona viitteen rakentajaan. Typescript osaa päätellä ketjutetuista metodeista palautuvan rakentajan lopullisen tyypin, kunhan jokainen metodin paluuarvossa päivitetään rakentajan tilaa geneeristen tyyppien avulla. Typescriptin geneeriset tyypit ovat tyyppimuuttujia, jotka edustavat mitä tahansa tyyppivaihtoehtoa ja niiden avulla pystytään luomaan uudelleenkäytettäviä funktioita, jotka voivat toimia monien eri tyyppien kanssa. Rakennusmetodien paluuarvossa rakentajan palauttaman olion geneeristä tyyppiä voidaan päivittää lisäämällä siihen ominaisuuksia Typescriptin `&`-avainsanaa käyttämällä. Avainsanalla `&` (Ampersand) luodaan kahden eri tyypin yhdiste, joka vastaa joukko-opin yhdistettä $A \cap B$.

Tekijä aloitti rakentajan toteutuksen kehittämällä liitteen 5.3 mukaisen yksinkertaistetun prototyypin rakentajasta, jolla pystytään lisäämään ominaisuuksia tyhjälle Javascript-oliolle. Tekemällä ensin yksinkertaistetun prototyypin ideasta sen toimivuutta pystytään arvioimaan ennen kuin toteutukseen on käytetty liikaa aikaa. Toteutuksessa metodi `addProperty` lisää olioinstanssille `properties` uuden ominaisuuden sen tietylle avaimelle ja palauttaa viitteen rakentajaan, jonka tyypin tilaa päivitetään **`as`**-avainsanaa käyttämällä. Typescriptissä **`as`**-avainsanalla voidaan ilmoittaa sellaisista tyyppimuunnoksista Typescriptin-kääntäjälle, joita se ei osaa itse päätellä. `build`-metodi palauttaa `properties` muuttujan eli viitteen luotavaan olioon. `build`-metodi palauttaa `TResult` sijaan tyypin muodossa `{[K in keyof TResult]: TResult[K]}`, koska niin saa poistettua rakennusvaiheessa tyyppiin lisätyt tarpeettomat `&`-merkit. Toisin sanoen tyyppi `{ a: number; } & { b: string; }` saadaan helpompilukuisempaan muotoon `{ a: number; b: string; }`.

Koodikatkelma 5.3: Yksinkertainen prototyyppi rakentajasta.

```
export type Builder<TResult> = {
  addProperty: <K extends string, TProperty>(key: K, property:
     TProperty) => Builder<TResult & {[P in K]: TProperty}>;
  build: () => {[K in keyof TResult]: TResult[K]};
};

export const getBuilder = <TResult>(): Builder<TResult> => {
```

```
  const properties: Record<string, unknown> = {};

  const addProperty = <TKey extends string, TProperty>(key: TKey,
      property: TProperty): Builder<TResult & {[P in TKey]: TProperty
      }> => {
    properties[key] = property;
    return builder as Builder<TResult & {[P in TKey]: TProperty}>;
  };

  const build = (): TResult => properties as {[K in keyof TResult]:
      TResult[K]};

  const builder: Builder<TResult> = {
    addProperty,
    build,
  };

  return builder;
};
```

Kun prototyypin avulla konseptin toimivuus oli testattu, sen toteuttamista jatkettiin. Prototyypin toteutuksessa on kuitenkin keskeisiä puutteita: Rakentaja-suunnittelumallissa hyviin käytänteisiin kuuluu, että samalla rakentajalla voidaan rakentaa useita olioinstansseja, mutta prototyypissä `build`-metodi palauttaa aina viitteen samaan olioon. Lisäksi rakentajamallissa yleensä olion rakentaminen tapahtuu kokonaan `build`-metodin sisällä, toisin kuin toteutuksessani, jossa `addProperty`-metodin sisällä uusi ominaisuus lisätään heti olion ominaisuuksiin.

Yllä mainitut puutteet saatiin ratkaistua liitteen 5.4 mukaisesti hyödyntämällä tehdasfunktioita. Hyödyntämällä tehdasfunktioita olioinstanssin luominen tapahtuu kokonaan `build`-metodin sisällä; Metodi `addProperty` ottaa parametrina tehdasfunktion `propertyFactory`, joka on asynkroninen funktio, joka palauttaa geneerisen ominaisuuden `TProperty`. Tehdasfunktiot tallennetaan listassa rakentajan muuttujaan, ja `build`-metodissa luodaan uusi tyhjä olioinstanssi, jolle lisätään halutut ominaisuudet kutsumalla listan tehdasfunktioita järjestyksessä. Näin ollen olion rakentaminen tapahtuu vasta `build`-metodin sisällä, ja

jokainen metodikutsu palauttaa uuden erillisen instanssin.

Koodikatkelma 5.4: Paranneltu rakentaja.

```
type PropertyFactory<TProperty> = () => Promise<TProperty>;

type FactoryType = 'property' | ...;

type FactoryFunction = PropertyFactory<unknown> | ...;

type Factory = {
  type: FactoryType;
  key?: string;
  factory: FactoryFunction;
};

type Context = Record<string, unknown>;

export const getContextBuilder = <TResult>(): Builder<TResult> => {
  const factories: Factory[] = [];

  ...

  const addProperty = <K extends string, TProperty>(name: K,
     propertyFactory: PropertyFactory<TProperty>): Builder<TResult &
     {[P in K]: TProperty}> => {
    const factory: Factory = {
      type: 'property',
      key: name,
      factory: propertyFactory,
    };
    factories.push(factory);
    return builder as Builder<TResult & {[P in K]: TProperty}>;
  };

  const build = async (): Promise<TResult> => {
    const context: Context = {};
```

```
    const results = await Promise.allSettled(factories.map(async
        factory => {
      const {type, key, factory: factoryFunction} = factory;
      switch (type) {
        case 'property': {
          const property = await factoryFunction() as PromiseType<
              ReturnType<PropertyFactory<unknown>>>;
          context[key!] = property;
          break;
        }
        ...
        default: {
          throw new Error('Unknown factory type');
        }
      }
    }));

    const errors = results.filter(result => result.status === '
        rejected');
    if (errors.length > 0) {
      throw new Error(`${errors.length} factories failed to be created
          : ${JSON.stringify(errors, null, '\t')}`);
    }

    return context as {[K in keyof TResult]: TResult[K]};
  };

  ...

};
```

Seuraava ongelma oli, miten rakentaja-suunnittelumallin avulla voidaan luoda käyttäjiä testikontekstille. Kohdesovellukselle tarvitsee luoda AWS Cognito-käyttäjiä, joiden pitää kirjautua sisään, jotta saadaan tokenit joiden avulla pystytään käyttämään sovelluksen SaaS-rajapintoja. Toteutuksen tulisi olla yleiskäyttöinen niin, että sillä voisi tehdä käyttäjiä erilaisille sovelluksille, jotka käyttävät erilaisia menetelmiä ja rajapintoja käyttäjien luontiin.

Kontekstilla pitää siis olla funktio, jota kutsumalla pystytään luomaan uusia käyttäjiä. Sovelluksesta riippuen tälle funktiolle pitää pystyä antamaan mukautettuja parametreja, joiden avulla käyttäjien luontiprosessia pystytään kontrolloimaan.

Ongelma ratkaistiin tekemällä käyttäjien luomiseen oma rakentaja, joka konfiguroidaan testikontekstin rakentajalle. Vastaavasti kirjautuneelle käyttäjälle tehtiin oma rakentaja, joka konfiguroidaan käyttäjän rakentajalle. Liitteen 5.5 katkelmasta nähdään, että metodi `configureUserBuilder` ottaa parametrina tehdasfunktion, jossa käyttäjän rakentaja alustetaan. Tehdasfunktio palauttaa funktion geneerisillä parametreilla `TParameters`, jonka paluuarvona palautetaan käyttäjä. Kyseinen funktio asetetaan `build`-metodissa olion ominaisuuteen `createUser`. Näin ollen kutsumalla luodun kontekstin metodia `createUser` mukautetuilla parametreilla `TParameters` pystytään luomaan uusi testikäyttäjä testiskripteissä.

Koodikatkelma 5.5: Paranneltu rakentaja.

```
type BuilderContext = {
  clientFactories?: unknown[];
  userBuilder?: ConfigureUserBuilderFactory<any, any>;
};

type Context = {
  [key: string]: unknown;
  createUser?: <TParameters extends any[]>(args: TParameters) =>
     Promise<unknown>;
};

type ConfigureUserBuilderFactory<
  TParameters extends any[],
  TUserBuilderResult,
> = () => Promise<(
  ...args: TParameters
) => Promise<UserBuilder<TUserBuilderResult>>>;

...
```

```
export const getContextBuilder = <TResult>(): Builder<TResult> => {
  const builderContext: BuilderContext = {};
  // eslint-disable-next-line @typescript-eslint/array-type
  const factories: Array<Factory> = [];

  ...

  const configureUserBuilder = <TParameters extends any[],
     TUserBuilderResult>(userBuilderConfigurationFactory:
     ConfigureUserBuilderFactory<TParameters, TUserBuilderResult>):
     Builder<TResult & {[P in 'createUser']: (...args: TParameters)
     => Promise<{[K in keyof TUserBuilderResult]: TUserBuilderResult[
     K]}>}> => {
    builderContext.userBuilder = userBuilderConfigurationFactory;
    return builder as Builder<TResult & {[P in 'createUser']: (...args
       : TParameters) => Promise<{[K in keyof TUserBuilderResult]:
       TUserBuilderResult[K]}>}>;
  };

  const build = async (): Promise<TResult> => {
    const context: Context = {};

    ...

    const configureUserFactory = builderContext.userBuilder ? await
       builderContext.userBuilder() : undefined;

    if (configureUserFactory) {
      context.createUser = async (...args: any[]) => {
        const userBuilderInstance = await configureUserFactory(...args
           );
        const user = await userBuilderInstance.build();
        return user;
      };
    }
```

```
    return context as {[K in keyof TResult]: TResult[K]};
  };

  ...
};
```

Edellä mainittuun toteutukseen liittyy vielä oleennaisia puutteita. Eräs keskeinen motiivi käyttää mukautettua liitännäistä oli Japan elinkaari-hookien hyödyntäminen, joiden avulla pystyttäisiin purkamaan testien aikana luodut resurssit automaattisesti. Lisäksi rakentajilla luodut oliot pitää voida alustaa ja purkaa; Esimerkiksi kohdesovelluksen Cognito-asiakasohjelmaa pitää kutsua käyttäjän luonnin ja poistamisen yhteydessä. Olioilla pitää olla myös sisäinen tila, johon voidaan tallentaa tietoa ja joka voidaan välittää parametrina rakentajan metodien tehdasfunktioille. Esimerkiksi jotkin API-asiakasohjelmat tarvitsevat alustuksessaan käyttäjän kirjautumisen yhteydessä saatuja tokeneita, joten ne pitää tallentaa käyttäjäolion tietoihin.

Yllä mainitut ongelmat ollaan korjattu toteutuksessa 5.6 hyödyntämällä käärijätyyppiä `CleanableResource<T>`, joka toimii liitännäisessä resurssien alustamisen ja purkamisen rajapintana. Sille kuuluu kaksi ominaisuutta: `instance`, joka on viite tyyppiä `T` olevaan olioon, sekä `cleanup`-funktio, jota kutsumalla olioon liittyvät resurssit voidaan purkaa. Jokainen olio, jonka yhteydessä luodaan purettavia resursseja, kääritään tyyppiin `CleanableResource<T>`. Kaikki olion rakennusprosessin aikana luodut resurssit puretaan rekursiivisilla `cleanup`-funktioiden kutsuilla: Kontekstin `cleanup`-funktio, jota kutsutaan testiskriptin suoriutumisen lopuksi purku-hookissa, kutsuu kontekstille luotujen olioiden `cleanup`-funktioita, kun taas niiden `cleanup`-funktiot kutsuvat niille luotujen olioiden `cleanup`-funktioita (ja niin edelleen). Rakentajalle määritellään sen luoman olion sisäinen tila tai konfiguraatio `addConfiguration`-metodilla. Metodi ottaa parametrina tehdasfunktion, joka palauttaa konfiguraation käärittynä `CleanableResource<T>`-olioon. Tämä konfiguraatio välitetään parametrina jokaiselle tehdasfunktioille, jolloin tehdasfunktioissa pääsee käsiksi olion sisäiseen tilaan.

Koodikatkelma 5.6: Paranneltu rakentaja.

```
export type CleanableResource<T> = {
  instance: T;
  cleanup: () => Promise<void>;
};

type ConfigureUserBuilderFactory<
  TConfig,
  TUserConfig,
  TParameters extends any[],
  TUserBuilderResult,
> = (config: TConfig) => Promise<(
  ...args: TParameters
) => Promise<UserBuilder<TUserBuilderResult, TUserConfig>>>;

type PropertyFactory<
  TConfig,
  TProperty,
> = (config: TConfig) => Promise<TProperty>;

type ConfigurationFactory<TConfig> = () => Promise<CleanableResource<
   TConfig>>;

type Context = {
  [key: string]: unknown;
  createUser?: <TParameters extends any[]>(args: TParameters) =>
     Promise<unknown>;
};

type BuilderContext = {
  clientFactories?: unknown[];
  userBuilder?: ConfigureUserBuilderFactory<any, any, any, any>;
};

type FactoryType = 'property' | ... ;
```

```
type FactoryFunction<TConfig> = PropertyFactory<TConfig, unknown> |
    ...;

type Factory<TConfig> = {
  type: FactoryType;
  key?: string;
  factory: FactoryFunction<TConfig>;
};

export const getContextBuilder = <TResult, TConfig = Record<string,
    unknown>>(): Builder<TResult, TConfig> => {
  const builderContext: BuilderContext = {};
  let configurationFactory: undefined | (ConfigurationFactory<any>);
  const factories: Array<Factory<TConfig>> = [];

  const addConfiguration = <TConfigNew>(configFactory:
      ConfigurationFactory<TConfigNew>): Builder<TResult, TConfigNew>
      => {
    configurationFactory = configFactory;
    return builder as unknown as Builder<TResult, TConfigNew>;
  };

  const addProperty = <K extends string, V>(name: K, propertyFactory:
      PropertyFactory<TConfig, V>): Builder<TResult & {[P in K]: V},
      TConfig> => {
    const factory: Factory<TConfig> = {
      type: 'property',
      key: name,
      factory: propertyFactory,
    };
    factories.push(factory);
    return builder as Builder<TResult & {[P in K]: V}, TConfig>;
  };

  const configureUserBuilder = <TParameters extends any[],
      TUserBuilderResult, TUserConfig>(userBuilderConfigurationFactory
      : ConfigureUserBuilderFactory<TConfig, TUserConfig, TParameters,
```

```
    TUserBuilderResult>): Builder<TResult & {[P in 'createUser']:
    (...args: TParameters) => Promise<{[K in keyof
    TUserBuilderResult]: TUserBuilderResult[K]}>}, TConfig> => {
  builderContext.userBuilder = userBuilderConfigurationFactory;
  return builder as Builder<TResult & {[P in 'createUser']: (...args
      : TParameters) => Promise<{[K in keyof TUserBuilderResult]:
      TUserBuilderResult[K]}>}, TConfig>;
};

const cleanup = async (cleanups: Array<() => Promise<void>>) => {
  const results = await Promise.allSettled(cleanups.reverse().map(
      async func => {
    await func();
  }));
  const errors = results.filter(result => result.status === '
      rejected');
  if (errors.length > 0) {
    throw new Error(`${errors.length} cleanup functions failed to be
        teared down: ${JSON.stringify(errors, null, '\t')}`);
  }
};

const resolveConfiguration = async <TConfigNew>(configFactory:
    ConfigurationFactory<TConfigNew>, cleanups: Array<() => Promise<
    void>>): Promise<TConfigNew> => {
  const config = await configFactory();
  cleanups.push(config.cleanup);
  return config.instance;
};

const build = async (): Promise<CleanableResource<TResult>> => {
  const cleanupFunctions: Array<() => Promise<void>> = [];
  const context: Context = {};
  const config = (configurationFactory ? await resolveConfiguration(
      configurationFactory, cleanupFunctions) : undefined) as
      TConfig;
```

```
const results = await Promise.allSettled(factories.map(async
    factory => {
  const {type, key, factory: factoryFunction} = factory;
  switch (type) {
    case 'property': {
      const property = await factoryFunction(config) as
          PromiseType<ReturnType<PropertyFactory<TConfig, unknown
          >>>;
      context[key!] = property;
      break;
    }

    ...

    default: {
      throw new Error('Unknown factory type');
    }
  }
}));

const configureUserFactory = builderContext.userBuilder ? await
    builderContext.userBuilder(config) : undefined;

const errors = results.filter(result => result.status === '
    rejected');
if (errors.length > 0) {
  throw new Error(`${errors.length} factories failed to be created
      : ${JSON.stringify(errors, null, '\t')}`);
}

if (configureUserFactory) {
  context.createUser = async (...args: any[]) => {
    const userBuilderInstance = await configureUserFactory(...args
        );
    const user = await userBuilderInstance.build();
    cleanupFunctions.push(user.cleanup);
    return user.instance;
```

```
      };
    }

    return {instance: context as {[K in keyof TResult]: TResult[K]},
        cleanup: async () => cleanup(cleanupFunctions)};
  };

  const builder: Builder<TResult, TConfig> = {
    addProperty,
    addConfiguration,
    configureUserBuilder,
    build,
  };

  return builder;
};
```

Esitetyn toteutuksen perustoiminnallisuudet ovat nyt kunnossa, mutta rakentajille tulee vielä lisätä apumetodeja, jotka tekevät ohjelmistokehyksen käyttämisestä helpompaa päästä pääḧn -testauksessa. Kuvioissa 6, 7 ja 8 kuvaillaan, mitä rakentajien ero metodit tekevät sekä niiden ottamat parametrit.

5.4.3 Mukautetun liitännäisen käyttö

Koodikatkelmassa 5.7 näytetään käytännön esimerkki rakentajien käytöstä. Esimerkissä `addConfiguration`-metodissa alustetaan `CognitoIdentityProviderClient`-asiakasohjelma, jota käytetään metodissa `configureUserBuilder` käyttäjien luomiseen ja poistamiseen. Konfiguraatiossa haetaan myös vaadittavia ympäristömuuttujia `getEnvs` -metodilla sekä asetetaan kontekstille UUID-tunniste. Rakentajassa asetetaan kontekstille tenant-kohtainen REST-asiakasohjelma metodin `addClientFactory` avulla. Käyttäjien luominen alustetaan metodilla `configureUserBuilder`, jossa `getUserBuilder()`-metodilla luodaan rakentaja käyttäjän luomiseen, jonka avulla määritellään käyttäjän ominaisuudet. Käyttäjän rakentaja palautetaan `configureUserBuilder`-metodin paluuarvossa, jota konteksti käyttää käyttäjien instanssien luomiseen testiskripteissä. Käyttä-

jäjän rakentajan `addConfiguration`-metodissa kutsutaan metodia `createCognitoUser` uuden Cognito-käyttäjän luomiseen. Palautettava konfiguraatio palautetaan käärittynä `CleanableResource`-tyyppiin kutsumalla `createCleanableResource`-metodia, joka ottaa parametrina olion sekä purkufunktion. Purkufunktiossa poistetaan luotu Cognito-käyttäjä kutsumalla metodia `deleteCognitoUser`. Käyttäjälle lisätään rakentajalla ominaisuudet `identification` ja `email`, jotta niitä voidaan lukea testeissä. Käyttäjän rakentajan `configureAuthenticatedUserBuilder`-metodi määrittelee rakentajan, jota käytetään kirjautuneen käyttäjän luomiseen, kun käyttäjällä kirjaudutaan sisään `authenticateUser`-metodilla. Kirjautuneen käyttäjän `addConfiguration`-metodissa käyttäjälla kirjaudutaan sisään `Cognito`-asiakasohjelman avulla, ja kirjautumisen yhteydessä saadut tokenit varastoidaan kirjautuneen käyttäjän tilaan. Kirjautuneelle käyttäjälle lisätään `.addClientCreatorFactory`-metodilla graphql-asiakasohjelma, joka voidaan alustaa testiskripteissä kutsumalla metodia `clients.graphql.init()`.

REST-rajapintojen asiakasohjelmat toteutettiin hyödyntämällä `openapi-fetch` ja `openapi-typescript` -kirjastoja. `opentapi-typescript` -kirjaston avulla pystytään automaattisesti generoimaan REST-rajapinnalle Typescript-ohjelmakoodi sen JSON-skeeman avulla. Kohdesovelluksen REST-rajapinnoille on olemassa Swaggerin avulla luotu rajapintadokumentaatio, jonka kirjasto kääntää Typescript-kielelle. `openapi-fetch` -kirjaston avulla pystytään generoimaan Typescriptille käännetty ja tyypitetty asiakasohjelma REST-rajapinnalle, jonka avulla pystytään tekemään rajapintakutsuja testeissä.

Kohdesovelluksen ETRM-palvelun tapahtumia kuunntellaan luomalla websoketti-yhteys palvelun websoketti-palvelimelle. websoketti-yhteyden luomiseen käytetään `ws`-pakettia. Asiakasohjelman alustamisen yhteydessä avataan websoketti-yhteys palvelimelle, ja asiakasohjelma lähettää sille tilauspyynnön tapahtumien kuuntelemiseen. Kun palvelin on käsitellyt ja hyväksynyt tilauspyynnön, palvelin lähettää yhteyden kautta tiedon palvelussa tapahtuneista tapahtumista asiakasohjelmalle.

Kohdesovelluksen GraphQL-rajapinnan asiakasohjelma toteutettiin `urql`-kirjaston avulla. GraphQL-rajapinnan tapahtumien kuuntelemiseen hyödynnetään `graphql-ws` sekä `ws` -kirjastoja. Arkkitehtuurikuvauksessa 4 kuvattujen `Mock APIn` sekä `Callback APIn` -rajapintojen asiakasohjelmat toteutettiin käyttämällä `axios`-kirjastoa, koska niille ei olla

toteutettu JSON-skeemaa, jonka avulla voitaisiin luoda automaattisesti asiakasohjelmat `openapi-typescript` ja `openapi-fetch` -kirjastojen avulla.

Koodikatkelma 5.7: Esimerkki rakentajan käytöstä.

```
export const dataplatformContextBuilder = getContextBuilder()
  .addConfiguration(async (): Promise<CleanableResource<Config>> => {

    const cognitoClient = new CognitoIdentityProviderClient({
      credentials: CREDENTIALS,
      region: process.env.AWS_REGION,
    });

    const envs: EnvironmentVariables = envCahche ?? await getEnvs();
    if (!envCahche) {
      envCahche = envs;
    }

    const testContextId = randomUUID();

    const instance = {
      cognitoClient,
      testContextId,
      envs,
    };

    return createCleanableResource(instance, async () => {});
  })
  .addProperty('testContextId', async (config: Config) => config.
     testContextId)
  .acceptHooks()
  .addClientFactory('rest', async (config: Config) => {
    const restClient = await createRestApiClient(
      config.testContextId,
      config.envs.callbacksTableName,
      config.envs.restApiKey,
      config.envs.callbacksTableUrl);
```

```
  return restClient;
})
.configureUserBuilder(async (config: Config) => async () =>
   getUserBuilder()
  .addConfiguration(async (): Promise<CleanableResource<UserConfig>>
      => {
    const testContextId = config.testContextId;

    const output = await createCognitoUser(testContextId, config.
       cognitoClient, config.envs.userPoolId);

    const instance = {
      testContextId,
      cognitoClient: config.cognitoClient,
      username: output.username,
      password: output.password,
      groups: output.groups,
      attributes: output.attributes,
      envs: config.envs,
    };

    return createCleanableResource(instance, async () => {
      await deleteCognitoUser(output, config.cognitoClient, config.
         envs.userPoolId);
    });
  })
  .addProperty('email', async (config: UserConfig) => config.
     attributes?.find(attribute => attribute.Name === 'email')?.
     Value)
  .addProperty('identification', async (config: UserConfig) =>
     config.attributes?.find(attribute => attribute.Name === '
     custom:Identification')?.Value)
  .configureAuthenticatedUserBuilder(async (config: UserConfig) =>
     async (appClientType: AppClientType) =>
     getAuthenticatedUserBuilder()
    .addConfiguration(async () => {
      const appClientId = await getAppClientId(
```

```
      config.cognitoClient,
      appClientType,
      config.envs.userPoolId,
    );

    const authUser = await initiateSignInCognitoUser({
      username: config.username,
      password: config.password,
      attributes: config.attributes,
      groups: config.groups,
    },
    config.cognitoClient,
    appClientId,
    config.envs.userPoolId,
    );

    // Update the temporary password with the new password
    config.password = authUser.password;

    const instance = {
      testContextId: config.testContextId,
      appClientType,
      appClientId,
      token: {
        accessToken: authUser.authenticationResult?.AccessToken,
        idToken: authUser.authenticationResult?.IdToken,
        refreshToken: authUser.authenticationResult?.RefreshToken,
      },
      envs: config.envs,
    };

    return createCleanableResource(instance, async () => {});
  })
  .addClientCreatorFactory('graphql', async config => async () =>
     createGraphQLApiClient(
    config.testContextId,
    config.token.idToken ?? '',
```

```
        config.appClientType,
        config.envs.userPoolId))));
```

6 Arviointi

Tässä luvussa käydään läpi, täyttikö luotu päästä päähän -testien ohjelmistokehys sille asetetut vaatimukset ja onnistuiko käyttötapauksen toteuttaminen testiskriptinä.

6.1 Toiminnalliset vaatimukset

Päästä päähän -testit ajetaan automaattisesti CI/CD-putkessa, mutta ne tulee voida ajaa myös manuaalisesti pilviympäristössä sekä lokaalissa ympäristössä

Vaatimus täyttyi osittain. Testiskriptit voidaan ajaa lokaalisti asettamalla vaadittavat ympäristömuuttujat `.env`-tiedostoon, joiden avulla rajapintojen asiakasohjelmat voidaan yhdistää pitkäaikaiseen hiekkalaatikko- tai kehitysympäristöön. Testit suoritetaan pilviympäristössä automaattisesti aina, kun sovellusta päivitetään koodikatkelman 5.1 mukaisesti. Testiohjelman AWS CodeBuild-projektin voi myös käynnistää manuaalisesti AWS:n konsolista. Testejä ei kuitenkaan voida vielä suorittaa osana CI/CD-prosessia, koska sitä varten sovelluksessa tulisi olla tuki lyhytaikaisten ympäristöjen käyttöönottamiseen.

Tuki samanaikaisille testiajoille (joko jonotusjärjestelmällä tai tuki rinnakkaisille testiajoille)

Vaatimus täyttyi. Testit voidaan ajaa rinnakkain, koska jokaisessa testiskriptissä luodaan omat uniikit käyttäjät ja testidata, ja testeissä lähetetyt tapahtumat erotetaan toisistaan niiden otsikkokentässä olevan testitapauksen suorituskerran UUID-tunnisteen avulla.

Testiskriptien tulee voida kutsua sovelluksen rajapintojen asiakasohjelmia sekä rekisteröityä vastaanottamaan sovelluksen tapahtumia

Vaatimus täyttyi. Kaikkia sovelluksen rajapintoja voidaan kutsua testiskripteissä ja jokaisen rajapinnan asiakasohjelman kautta on mahdollista rekisteröityä kuuntelemaan sovelluksen tapahtumia.

Testien testidata ja lähtötila tulee olla alustettavissa

Vaatimus täyttyi. Testiskripteissä pystytään luomaan testidataa ja käyttäjiä, autentikoimaan käyttäjät vaadittaviin asiakasohjelmiin, rekisteröitymään sovelluksen tapahtumien kuuntelijoiksi sekä korvaamaan ulkoisten järjestelmien rajapinnat testisijaisilla.

Testiajo tulee suorittaa eristetyssä testausympäristössä

Vaatimus täyttyi osittain. Testitapauksien suoritukset ovat riittävässä määrin eristyksissä toisistaan siten, ettei testien rinnakkaisajosta koidu käytännön ongelmia. Teoriassa on mahdollista hakea ja manipuloida PaaS-rajapinnan kautta testin ulkopuolista dataa, mutta käytännössä jokaiselle testille luodaan oma uniikki testidata.

Toteutuksessa pitää olla mahdollista tehdä sekä positiivisia- että negatiivisia testejä

Vaatimus täyttyi. Japan `assert`-liitännäisen avulla on mahdollista testata myös virhetilanteita. On esimerkiksi mahdollista suorittaa virheellisiä rajapintakutsuja ja verifioida, että niiden vastauksessa on odotettu virheen statuskoodi ja viesti.

Ulkoisten järjestelmien rajapintakutsut tulee olla korvattavissa ja testattavissa testisijaisilla

Vaatimus täyttyi. Luvussa 5.3 kuvailtiin, kuinka ulkoiset rajapintakutsut voidaan korvata testisijaisilla Mock APIn avulla.

6.2 Laadulliset vaatimukset

Testausympäristön tulee olla mahdollisimman lähellä tuotantoympäristöä

Vaatimuksen voidaan ajatella täyttyneen. Testit voidaan suorittaa sovelluksen kehitysympäristössä, jossa uusien toiminnallisuuksien toimivuus verifioidaan ennen niiden vientiä tuotantoympäristöön. Testiskripteissä käytetään samoja sovelluksen rajapintoja kuin asiakkaat ja tenantit käyttävät, ja ainoastaan ulkopuolisten järjestelmien rajapinnat korvataan testisijaisilla.

Toteusta voidaan käyttää tekemään testitapauksia useille sovelluksen käyttötapauksille

Vaatimuksen voidaan olettaa täyttyneen. Koska testiskripteissä voidaan hyödyntää kaikkia

sovelluksen rajapintoja sekä korvata ulkoisten rajapinnat testisijaisilla, voidaan olettaa että testeissä voidaan testata lähes mitä tahansa sovelluksen käyttötapauksia.

Testiskriptien kirjoittamisen tulee olla mahdollisimman helppoa ohjelmistokehittäjän näkökulmasta

Vaatimuksen voidaan olettaa täyttyneen. Asiantuntijoilta saatu palaute on ollut positiivista testien kirjoittamisen helppouteen liittyen. Toteutuksessa on monia käytettävyyttä helpottavia piirteitä: mukautettu `endToEnd`-liitännäinen abstraktoi ja automatisoi toistuvia toiminnallisuuksia siten, ettei testiskriptien kehittäjien tarvitse kirjoittaa niitä manuaalisesti. REST-rajapinnat ovat vahvasti tyypitettyjä ja antavat virheilmoituksia, jos testiskriptien rajapintakutsujen parametrit ovat väärin. Useat operaatiot, kuten käyttäjien luominen ja autentikointi, tapahtuu yhdellä koodirivillä.

6.3 Käyttötapauksen toteutus ohjelmistokehyksellä

Kuvion 1 käyttötapauksen pohjalta luotiin testitapaus 5 kuvaamaan testiskriptin toiminnallisuutta. Käyttötapauksen viimeinen vaihe, eli vahvistussähköpostin vastaanottaminen jäi testitapauksesta pois, koska testiskripteissä luoduilla käyttäjillä ei ole oikeita sähköpostiosoitteita, joihin viestit voitaisiin vastaanottaa. Käyttäjien sähköposti päättyy merkkijonoon `@akamon.example`, ja `.example`-päätteiset sähköpostiosoitteet eivät koskaan ohjaudu oikeisiin sähköpostiosoitteisiin (“A list of fictitious numbers, domains, and more”, n.d).

Toteutettu testiskripti on esitettynä liitteessä 8.1. Testiskripti saatiin lähes kokonaan toimintakuntoiseksi. `salesOfferCreated`-tapahtumaa ei voitu vastaanottaa GraphQL-websokettiin, koska sovelluksen `ContractService`-palvelu ei lähettänyt tapahtumaa tapahtumaväylään, joten käyttötapaus ei siltä osin toiminut.

Testitapauksen ja testiskriptin toteuttamista vaikeutti se, ettei tekijällä ollut juuri lainkaan tietoa tai ymmärrystä käyttötapauksen kuvaamasta liiketoimintaprosessista. Siksi toteutettu testiskripti jäi osittain karkeaksi ja sitä voisi laajentaa testaamaan eri parametreja ja vaiheita yksityiskohtaisemmin.

Toteutetun käyttötapauksen tarkoitus oli validoida, toimiko toteutettu päästä päähän -

ohjelmistokehys käytännössä ja testata sen vaatimuksien toteutumista. Käyttötapaus onnistui tarkoituksessaan hyvin, koska testiskriptissä oli hyödynnettävä lähes jokaista ohjelmistokehyksen toimintoa, lukuunottamatta REST-rajapinnan tapahtumien kuuntelemista Callback APIn avulla. REST-rajapinnan tapahtumien kuuntelun ja vastaanottamisen toimivuus testattiin kuitenkin erillisessä testissään.

Testitapaus Asiakaskohtainen kiinteän hinnan tarjouksen laskenta ja sopimuksen hyväksyntä.

Ennakkoehto Luodaan uudet Cognito-käyttäjät sekä asiakkaalle että ETRM-asiakaspalvelijalle. Asiakkaan Cognito-käyttäjälle luodaan Dataplatform-asiakas ja Dataplatform-käyttöpaikka.

Kuvaus Testitapauksen vaiheet:

1. Kirjaudutaan asiakkaan käyttäjällä sisään asiakassovellukseen.
2. Kirjaudutaan hallinnoija-käyttäjällä sisään ETRM-palveluun.
3. Luodaan Datahub-kulutusdataa käyttöpaikalle testidataksi viimeisen kahden vuoden ajalle.
4. Alustetaan Mock API korvaamaan ulkoinen Datahub-rajapinta ja lähettämään luotua testidataa asiakkaan käyttöpaikalle.
5. Rekisteröidytään asiakas kuuntelemaan `salesOfferCreated` ja `salesOfferContractOrderCreated` -tapahtumia.
6. Asiakas käynnistää GraphQL-mutaation `calculateEtrmSalesOffer` avulla tarjouslaskennan.
7. Odotetaan, että asiakkaan GraphQL-websoketti vastaanottaa tapahtuman `salesOfferCreated`.
8. Odotetaan, että ETRM-hallinnoijan websoketti vastaanottaa tapahtuman `sales Offer Created`.
9. Varmistetaan ETRM-rajapinnan kautta, että `OfferCalculation` ja `SalesOffer` tietueet ovat tallentuneet tietokantaan.
10. Asiakas hyväksyy saadun tarjouksen ja aloittaa sopimustilauksen GraphQL-mutaation `createProductContractOrder` avulla.
11. Odotetaan, että asiakas vastaanottaa tapahtuman `salesOfferContractOrderCreated`.
12. Varmistetaan REST-rajapinnan kautta, että asiakkaan `ContractOrder` on tallentunut tietokantaan.

Kuvio 5: Käyttötapauksen 1 pohjalta luotu testitapaus.

7 Pohdinta

Tässä luvussa kuvaillaan kehitysprosessin kulkua, mitä tutkielmasta opittiin ja mitä jäi saavuttamatta.

7.1 Kehitysprosessin kulku

Kehitysprosessin alussa tekjälle ei ollut entuudestaan tuttua, miten päästä päähän -testejä voidaan toteuttaa pilvipohjaiselle hajautetulle FaaS-sovellukselle. Suurin osa aikaisemmasta tutkimustiedosta vaikutti keskittyvän päästä päähän -testaamisen haasteisiin ja ominaisuuksiin, ei siihen miten niitä voidaan käytännössä toteuttaa. Niissä tutkimuksissa, jossa toteutettiin testejä käytännössä, käytettiin taas liian yksinkertaisia monoliitti-sovelluksia esimerkkeinä, joiden koko ohjelmakoodi oli yhdessä repositoriossa. Harmaan kirjallisuuden lähteistäkään (dokumentaatiot, online-tutoriaalit, yms.) ei löytynyt monia käytännön esimerkkejä päästä päähän -testien toteuttamiseen.

Kehitysprosessin aikana tekijän tietämys mikropalveluista ja palvelittomista pilvipalveluista kasvoi merkittävästi, koska aikaisemmin hänellä ei ole ollut tarvetta perehtyä niihin syvällisemmin. Tästä syvällisestä perehtymisestä on ollut hyötyä myös muissa projekteissa, joten tekijä voi suositella kehittäjiä perehtymään teknologioihin, joiden parissa he työskentelevät.

Kehitysprosessin onnistumista edesauttoi prosessin saatu tuki organisaation kokeneilta asiantuntijoilta. He auttoivat muun muuassa ohjelmistokehyksien vaatimuksien määrittelemisessä ja CDK-infrastruktuurikoodin toteuttamisessa, kirjoittivat tutkielmassa käytetyn käyttötapauksen sekä antoivat tekijälle tarvittaessa ohjeita, ideoita ja mielipiteitä. Toisaalta, tekijä sai kehitysprosessin aikana myös paljon omaa vastuuta, joka edesauttoi hänen oppimisprosessiaan.

Ohjelmistokehyksen teknisessä toteutuksessa lähdettiin toteuttamaan siihen liittyviä toiminnallisuuksia yksi kerrallaan. Kun toiminnallisuudet oli todettu toimiviksi, alettiin keskittymään enemmän laadullisiin vaatimuksiin refaktoroimalla ohjelmakoodia ja toteuttamalla mukautettu liitännäinen. Tekijän mielestä kyseinen kehitysprosessimalli toimi hyvin hänelle

itselleen.

Päästä päähän -testit päätettiin suorittaa olemassaolevissa kehitys- ja hiekkalaatikkoympäristöissä, koska sovellus ei vielä tukenut lyhytaikaisten ympäristöjen käyttöönottoa. Jos ympäristö voitaisiin käyttöönottaa ohjelmallisesti, päästä päähän -testit voitaisiin helposti suorittaa osana CI/CD-prosessia. Lisäksi ulkoisten rajapintojen korvaus testisijaisilla olisi vaivatonta, jos yksittäisten palveluiden ympäristömuuttujat voitaisiin määritellä ennen käyttöönottoa osoittamaan haluttuun mock-osoitteeseen.

Tekijä oppi, että käyttötapauksen kirjoittamisesta on paljon hyötyä testiskriptien toteuttamiseen. Päästä päähän -testeissä testataan sovelluksen toimivuutta loppukäyttäjän näkökulmasta, ja loppukäyttäjän tekemät toiminnot käyttöliittymällä kuvataan selkeästi käyttötapauksessa. Käyttötapauksen kirjoittamatta jättäminen voi johtaa siihen, ettei kaikkia loppukäyttäjän tekemiä rajapintakutsuja testata testiskriptissä todenmukaisella tavalla.

Testien kirjoittaminen kohdesovellukselle voi olla haastavaa, koska palveluiden välillä tapahtuvat prosessit voivat olla hyvin monimutkaisia. Palveluiden välisten löyhien kytkösten takia voi olla vaikea selvittää, mitkä palvelut vastaanottavat mitäkin tapahtumia. Suositeltavaa olisi, että käyttötapauksen testit toteuttaisi sellainen henkilö, jolla on läheinen ymmärrys käyttötapauksen liiketoimintaprosessista sekä sen teknisestä toteutuksesta sovelluksessa. Muuten testien toteuttamiseen voi kulua paljon ylimääräistä aikaa.

7.2 Tiedostetut puutteet ja jatkokehitys

Japan mukautetulle `endToEnd`-liitännäiselle olisi tulevaisuudessa suositeltavaa toteuttaa yksikkötestejä, joilla varmistetaan että liitännäisen rakentajien ohjelmakoodi toimii moitteettomasti. Rakentajien sisäisessä toteutuksessa ollaan hyödynnetty Typescriptin **`as`**-avainsanaa, joka ohittaa Typescripin tyyppivirheiden tarkistamisen käännösaikana, ja pakottaa Typescripin hyväksymään käyttäjän määrittelemän tyypin muuttujalle. Tästä voi seurata bugeja, jos manuaalisesti määritelty tyyppi ei vastaakaan todellista tyyppiä.

GraphQL-rajapinnan asiakasohjelmaan voisi lisätä tyypitystä tukevan kirjaston. Kirjasto voisi esimerkiksi tehdä tyypitetyt Javascript-ominaisuudet GraphQL-skeeman perusteella, joita

voisi käyttää GraphQL-kyselyiden muodostamisessa, jolloin monimutkaisten kyselyiden toteuttaminen voisi olla vaivattomampaa.

Nykyisessä toteutuksessa päästä päähän -testejä ei voida suodattaa, vaan jokainen testitapaus suoritetaan sovelluksen ohjelmakoodin päivittyessä. Jos testejä on paljon ja niiden suorittamiseen kuluu paljon aikaa, niin voisi olla halvempaa ja vähemmän aikaa vievää, että vain tehtyjä muutoksia koskevat testit ajettaisiin.

Toteutukseen olisi suositeltavaa lisätä monitorointiominaisuuksia. Testien epäonnistuessa olisi hyvä lähettää esimerkiksi sähköpostiviesti kehittäjälle, jonka ohjelmakoodin muutokset aiheuttivat testien epäonnistumisen. Hyvät monitorointiominaisuudet helpottaisivat sovelluksen ylläpitotyötä ja virheiden löytämistä sovelluksesta.

Sovellukseen olisi suositeltavaa lisätä tuki lyhytaikaisten ympäristöjen käyttöönotolle, mikä voisi helpottaa päästä päähän -testausta. Käyttöönotettavat lyhytaikaiset ympäristöt olisivat isolaatiossa muista ympäristöistä, joka takaisi sen, että testausympäristö pysyy identtisenä testiajosta toiseen, eikä erinäisiä konflikteja tapahdu useiden kehitystiimien kehittäessä eri ominaisuuksia samanaikaisesti hiekkalaatikkoympäristössä. CI/CD-prosessissa voitaisiin kättöönottaa lyhytaikainen ympäristö, jota vasten päästä päähän -testit ajetaan, jolloin testien läpäisy voidaan asettaa vaatimukseksi tuotantoon menevälle ohjelmakoodille.

Nykyisessä toteutuksessa päästä päähän -testeillä testataan vain rajapintoja, ei mahdollisten asiakassovelluksien käyttöliittymiä. Tämä oli tietoinen valinta, mutta olisi syytä pohtia, voidaanko toteutusta laajentaa tulevaisuudessa siten, että sitä voitaisiin hyödyntää myös käyttöliittymäpuolella, esimerkiksi lyhytaikaisten ympäristöjen avulla.

8 Yhteenveto

Tutkielmassa toteutettiin artefakti, jota voidaan hyödyntää päästä päähän testien kehityksen apuna hajautettuihissa tapahtumaohjatuissa FaaS-sovelluksissa. Tutkielmaa varten luotiin päästä päähän -testeille ohjelmistokehys sidosryhmänä toimivan organisaation pilvinatiiviin multi-tenantti -sovellukseen, joka hyödyntää AWS Lambdan palvelittomia pilvilaskentapalveluita ja jossa on sekä asiakasorganisaatioille PaaS-rajapintoja sekä asiakassovellusksien käyttämiä SaaS-rajapintoja.

Tutkielman tarkoituksena oli tuottaa hyödyllinen artefakti ja vastata etenkin kysymykseen: miten päästä päähän -testejä voidaan luoda hajautettuihin tapahtumapohjaisiin FaaS-sovelluksiin? Kyseisten sovelluksien päästä päähän -testaus on vaikeaa, koska niissä päästä päähän -testit tulee suorittaa pilviympäristössä, pilvipalveluiden tarjoamat työkalut ovat testaamisen osalta puutteellisia ja teollisuuden parhaat käytänteet ovat testaamisen osalta tuntemattomia monille kehittäjille.

Tutkielmassa kuvataan kehitysprosessin kulkua ja sen aikana ilmenneitä ongelmia ja haasteita, sekä perustellaan kehitysprosessin aikana tehtyjä valintoja. Toteutettua päästä päähän -ohjelmistokehystä arvioitiin sille asetettujen toiminnallisten ja laadullisten vaatimusten pohjalta. Lisäksi toteutuksen toimivuutta käytännössä testattiin toteuttamalla laaditun käyttötapauksen pohjalta päästä päähän -testiskripti yhdelle sovelluksen käyttötapaukselle, jonka toimivuus on sovelluksen liiketoiminnan kannalta tärkeää.

Kaikki ohjelmakoodi kirjoitettiin Typescript-ohjelmointikielellä Node.js-ajoympäristössä. Toteutuksessa ohjelmoitiin CDK-sovellus, jonka avulla CI/CD-prosessissa käyttöönotetaan päästä päähän -testaamiseen vaadittava pilvi-infrastruktuuri. Testit ajetaan pitkäaikaisessa pilviympäristössä AWS CodeBuild-projektissa automaattisesti sovelluksen päivittyessä. Testit voidaan ajaa myös lokaalisti asettamalla vaadittavat ympäristömuuttujat `.env`-tiedostoon. Testiskripteissä hyödynnettiin backend-testausta varten kehitettyä Japa-kirjastoa päästä päähän -testiprojektin ohjelmistokehyksenä. Päästä päähän -testausta varten Japaan luotiin mukautettu liitännäinen, joka helpottaa testien kehitysprosessia abstraktoimalla monimutkaisia ja toistuvia toiminnallisuuksia toimimaan automaattisesti. Toteutukseen luotiin myös me-

netelmä, jossa ulkoisten järjestelmien rajapintoja voi korvata testisijaisten avulla pilviympäristössä. Toteutus tukee myös rinnakkaisia testiajoja, ja kaikki testitapaukset suoritetaan isolaatiossa toisistaan.

Toteutuksen avulla saatiin toteutettua päästä päähän -testejä onnistuneesti sovellukselle, ja lisäksi suurin osa päästä päähän -ohjelmistokehyksen toiminnallisista ja laadullisista vaatimuksista täyttyivät odotetulla tavalla. Käyttötapauksen toteuttaminen testiskriptinä onnisti lähes kokonaan, mutta kaikkia käyttötapauksen vaiheita ei pystynyt verifioimaan, koska kyseiset toiminnallisuudet olivat vielä toteuttamatta sovelluksessa.

Toteutukseen liittyi vielä puutteita, kuten monitoroinnin puutteellisuus eli esimerkiksi kehittäjien informoiminen testiajojen epäonnistumisista. Testejä ei myöskään voida nykyisen toteutuksen avulla suorittaa osana CI/CD-prosessia, vaan ne suoritetaan pitkäaikaisessa kehitys- tai hiekkalaatikkoympäristössä AWS CodeBuild-projektissa. Jatkokehityksenä sovellukselle voisi lisätä tuen testien ajamiseen CI/CD-prosessissa lyhytaikaisten ympäristöjen avulla, joilla ympäristöjä voitaisiin käyttöönottaa automaattisesti osana CI/CD-prosessia.

Lähteet

"A list of fictitious numbers, domains, and more". n.d. Viitattu 6. kesäkuuta 2024. https://ddbeck.com/fictitious-numbers/.

"Amazon EventBridge Documentation". n.d. Viitattu 17. syyskuuta 2023. https://aws.amazon.com/eventbridge/.

Ammann, Paul ja Jeff Offutt. 2017. *Introduction to Software Testing*. 2. painos. Cambridge University Press.

"Automated testing". n.d. Viitattu 26. syyskuuta 2023. https://serverlessland.com/event-driven-architecture/testing-introduction.

"AWS CDK Concepts". n.d. Viitattu 11. huhtikuuta 2024. https://docs.aws.amazon.com/cdk/v2/guide/core_concepts.html.

"AWS Lambda". n.d. Viitattu 16. syyskuuta 2023. https://aws.amazon.com/lambda/.

"AWS Lambda The Ultimate Guide". n.d. Viitattu 16. syyskuuta 2023. https://www.serverless.com/aws-lambda.

"AWS Serverless Computing". n.d. Viitattu 16. syyskuuta 2023. https://aws.amazon.com/serverless/.

Baldini, Ioana, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell ym. 2017. *Serverless Computing: Current Trends and Open Problems.* arXiv: 1706.03178 [cs.DC].

Bushong, Vincent, Amr S. Abdelfattah, Abdullah A. Maruf, Dipta Das, Austin Lehman, Eric Jaroszewski, Michael Coffey ym. 2021. "On Microservice Analysis and Architecture Evolution: A Systematic Mapping Study". *Applied Sciences* 11 (17). ISSN: 2076-3417. https://doi.org/10.3390/app11177856. https://www.mdpi.com/2076-3417/11/17/7856.

Cohn, Mike. 2009. *Succeeding with Agile: Software Development Using Scrum.* 1st. Addison-Wesley Professional. ISBN: 0321579364.

Daly, Jeremy. n.d. “Serverless Reference Architectures”. Viitattu 19. syyskuuta 2023. https://www.jeremydaly.com/serverless-reference-architectures/.

Di Francesco, Paolo, Patricia Lago ja Ivano Malavolta. 2019. “Architecting with microservices: A systematic mapping study”. *Journal of Systems and Software* 150:77–97. ISSN: 0164-1212. https://doi.org/https://doi.org/10.1016/j.jss.2019.01.001. https://www.sciencedirect.com/science/article/pii/S0164121219300019.

Fowler, Martin ja James Lewis. 2014. “Microservices a definition of this new architectural term”. Viitattu 16. syyskuuta 2023. https://martinfowler.com/articles/microservices.html.

Gortázar, Francisco ja Micael Gallego. 2018. “A Simple Path Towards Testing Cloud Applications”. Teoksessa *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion),* 28–29. https://doi.org/10.1109/UCC-Companion.2018.00029.

Hassan, Hassan B, Saman A Barakat ja Qusay I Sarhan. 2021. “Survey on serverless computing”. *Journal of Cloud Computing,* https://doi.org/https://doi.org/10.1186/s13677-021-00253-7.

Hevner, Alan, Alan R, Salvatore March, Salvatore T, Park, Jinsoo Park, Ram ja Sudha. 2004. “Design Science in Information Systems Research”. *Management Information Systems Quarterly* 28 (maaliskuu): 75–.

“Infrastructure as code”. n.d. Viitattu 11. huhtikuuta 2024. https://docs.aws.amazon.com/whitepapers/latest/introduction-devops-aws/infrastructure-as-code.html.

“Japa.dev documentation”. n.d. Viitattu 16. huhtikuuta 2024. https://japa.dev/docs/introduction.

Lee, Allen S. 1999. “Rigor and relevance in MIS research: beyond the approach of positivism alone”. *Management Information Systems Quarterly* 23:29–33. https://api.semanticscholar.org/CorpusID:85555901.

Leitner, Philipp, Erik Wittern, Josef Spillner ja Waldemar Hummer. 2018. “A mixed-method empirical study of Function-as-a-Service software development in industrial practice” (kesäkuu). https://doi.org/10.7287/peerj.preprints.27005.

Lenarduzzi, Valentina, Jeremy Daly, Antonio Martini, Sebastiano Panichella ja Damian Andrew Tamburri. 2021. “Toward a Technical Debt Conceptualization for Serverless Computing”. *IEEE Software* 38 (1): 40–47. https://doi.org/10.1109/MS.2020.3030786.

Lenarduzzi, Valentina ja Annibale Panichella. 2021. “Serverless Testing: Tool Vendors’ and Experts’ Points of View”. *IEEE Software* 38 (1): 54–60. https://doi.org/10.1109/MS.2020.3030803.

McQuaid, Patricia A. 2012. “Software disasters—understanding the past, to improve the future”. *Journal of Software: Evolution and Process* 24 (5): 459–470. https://doi.org/https://doi.org/10.1002/smr.500. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.500. https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.500.

Michelson, B.M. 2006. “Event-Driven Architecture Overview”. *Patricia Seybold Group, Feb.*

Ricca, Filippo ja Andrea Stocco. 2021. “Web Test Automation: Insights from the Grey Literature”. Teoksessa *SOFSEM 2021: Theory and Practice of Computer Science,* toimittanut Tomáš Bureš, Riccardo Dondi, Johann Gamper, Giovanna Guerrini, Tomasz Jurdziński, Claus Pahl, Florian Sikora ja Prudence W.H. Wong, 472–485. Cham: Springer International Publishing. ISBN: 978-3-030-67731-2.

Rinta-Jaskari, Eetu, Christopher Allen, Tamara Meghla ja Davide Taibi. 2022. “Testing Approaches And Tools For AWS Lambda Serverless-Based Applications”. Teoksessa *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops),* 686–692. https://doi.org/10.1109/PerComWorkshops53856.2022.9767473.

Soldani, Jacopo, Damian Tamburri ja Willem-Jan Heuvel. 2018. “The Pains and Gains of Microservices: A Systematic Grey Literature Review”. *Journal of Systems and Software* 146 (syyskuu). https://doi.org/10.1016/j.jss.2018.09.082.

Soliman, Wael ja Mikko Siponen. 2022. “What Do We Really Mean by Rigor in Information Systems Research?” *Proceedings of the 55th Hawaii International Conference on System Sciences (HICSS 2022),* 6583–6592. https://doi.org/10.24251/HICSS.2022.797.

Söylemez, Mehmet, Bedir Tekinerdogan ja Ayça Kolukısa Tarhan. 2023. “Microservice reference architecture design: A multi-case study”. *Software: Practice and Experience* n/a (n/a). https://doi.org/https://doi.org/10.1002/spe.3241. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.3241. https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.3241.

“What Is An Ephemeral Environment?” n.d. Viitattu 3. kesäkuuta 2024. https://ephemeralenvironments.io/.

“What is the AWS CDK?” n.d. Viitattu 11. huhtikuuta 2024. https://docs.aws.amazon.com/cdk/v2/guide/home.html.

Wieringa, Roelf J. 2014. *Design science methodology for information systems and software engineering* [kielellä Undefined]. 10.1007/978-3-662-43839-8. Springer. ISBN: 978-3-662-43838-1. https://doi.org/10.1007/978-3-662-43839-8.

Villamizar, Mario, Oscar Garcés, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano Merino, Rubby Casallas ym. 2017. “Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures”. *Service Oriented Computing and Applications* 11 (kesäkuu). https://doi.org/10.1007/s11761-017-0208-y.

Winzinger, Stefan ja Guido Wirtz. 2019. “Model-Based Analysis of Serverless Applications”. Teoksessa *Proceedings of the 11th International Workshop on Modelling in Software Engineerings,* 82–88. MiSE ’19. Montreal, Quebec, Canada: IEEE Press. https://doi.org/10.1109/MiSE.2019.00020. https://doi-org.ezproxy.jyu.fi/10.1109/MiSE.2019.00020.

Liitteet

Koodikatkelma 8.1: Käyttötapauksen testiskriptin toteutus.

```
test('Etrm test refresh consumption estimation', async (context:
    TestContext) => {
  const user = await context.endToEnd.createUser();
  const adminUser = await context.endToEnd.createUser();

  const customer: Customer = {
    identifier: randomUUID(),
    email: user.email,
    identification: user.identification,
    givenName: user.givenName,
    familyName: user.familyName,
    identificationType: 'PersonalId',
    customerType: 'Consumer',
    address: {
      streetName: 'TestStreet',
    },
  };

  const meteringPoint: MeteringPoint = {
    identifier: randomUUID(),
    gsrnIdentifier: randomUUID(),
  };

  await context.endToEnd.addHook(async () => {
    const createCustomerResponse = await context.endToEnd.clients.rest
        .POST('/api/customer', {body: customer});
    context.assert.equal(createCustomerResponse.response.status, 202);
    console.log('Added customer to dynamodb');

    const createMeteringPointResponse = await context.endToEnd.clients
        .rest.POST('/api/meteringpoint', {body: meteringPoint});
    context.assert.equal(createMeteringPointResponse.response.status,
        202);
```

```
    console.log('Added metering point to dynamodb');

    return async () => {
      await context.endToEnd.clients.rest.DELETE('/api/customer/{
          identifier}', {params: {path: {identifier: customer.
          identifier}}});
      console.log('Succesfully deleted customer from dynamodb');

      await context.endToEnd.clients.rest.DELETE('/api/meteringpoint/{
          identifier}', {params: {path: {identifier: meteringPoint.
          identifier}}});
      console.log('Succesfully deleted metering point from dynamodb');
    };
  });

  const userContext = await user.authenticateUser('SIERRA');
  const graphQlClient = await userContext.clients.graphql.init();

  const adminUserContext = await adminUser.authenticateUser('ETRM');
  const etrmClient = await adminUserContext.clients.etrm.init();

  const offerId = randomUUID(); // Sales offer id
  const orderId = randomUUID(); // Sales offer contract order id
  const gsrn = meteringPoint.gsrnIdentifier;
  const customerIdentification = customer.identification;

  const startDate = DateTime.now().plus({months: 2}).startOf('month').
      toUTC();
  const endDate = DateTime.now().plus({months: 8}).startOf('month').
      toUTC();

  const calculateSalesOfferProps = {
    tenantId: process.env.ORGANISATION ?? '',
    offer: {
      offerId,
      offerReason: 'ChangeOfSeller',
      gsrn,
```

```
      startDate: startDate.toISO(),
      customerType: 'Consumer',
      customerIdentification,
      customerCompanyName: 'String',
      customerEmail: 'String',
      customerPhone: 'String',
      consumptionEstimateSource: 'Datahub', // Or TypeCurve
      consumptionEstimate: 0,
      // TypeCurveId: 1,
      meteringPointAddress: {
        streetName: 'String',
        buildingNumber: 'String',
        stairwell: 'String',
        apartmentNumber: 'String',
        postalCode: 'String',
        postOffice: 'String',
        countryCode: 'String',
        poBox: 'String',
      },
    },
  };

  // Setup Mock API to mock Datahub service
  const tsStartDate = startDate.minus({years: 2});
  const timeseries = generateRandomHourlyObservations(tsStartDate,
     endDate);

  // Setup env variables for mock api
  const setVar = await context.endToEnd.clients.mockApi.
     setMockEnvironmentalVariables(
    {
      datahub_service_baseurl: context.endToEnd.clients.mockApi.
         baseUrl + 'datahub/',
    });
  context.assert.equal(setVar.status, 200);

  // Adds mock response to mock api
```

```
const mockResponse = await context.endToEnd.clients.mockApi.
    addMockResponse(
  {
    reqPath: 'ANY',
    reqMethod: 'GET',
    resBody: JSON.stringify(timeseries),
    resHeaders: {
      'Content-Type': 'application/json',
    },
    reqDomain: 'datahub',
    resStatusCode: 200,
    reqBody: '',
  });
context.assert.equal(mockResponse.status, 200);

const salesOfferCreatedSubscriptionProps = {
  tenantId: process.env.ORGANISATION ?? '',
  identifier: offerId,
};

const contractOrderSubscriptionProps = {
  tenantId: process.env.ORGANISATION ?? '',
  identifier: orderId,
};

// User interface is subscribed to listen for salesOfferCreated
    event
graphQlClient.subscription(
  `subscription salesOfferCreated($tenantId: ID!, $offerId: ID!) {
    salesOfferCreated(tenantId: $tenantId, offerId: $offerId) {
      offerId
    }
  }`, salesOfferCreatedSubscriptionProps).subscribe(result => {
  if (result.error) {
    context.assert.fail('Error subscribing to salesOfferCreated');
  } else {
    console.log('Received salesOfferCreated event');
```

```
    context.assert.equal(result.data?.salesOfferCreated.offerId,
       offerId);
    context.endToEnd.subscriptionHelper.increment('salesOfferCreated
       ');
  }
});

// User interface is subscribed to listen for
   salesOfferContractOrderCreated event
graphQlClient.subscription(
  `subscription salesOfferContractOrderCreated($tenantId: ID!,
     $identifier: ID!) {
       salesOfferContractOrderCreated(tenantId: $tenantId,
          identifier: $identifier) {
         identifier
       }
     }`, contractOrderSubscriptionProps).subscribe(result => {
  if (result.error) {
    context.assert.fail('Error subscribing to
       salesOfferContractOrderCreated');
  } else {
    console.log('Received salesOfferContractOrderCreated event');
    context.assert.equal(result.data?.salesOfferContractOrderCreated
       .identifier, orderId);
    context.endToEnd.subscriptionHelper.increment('
       salesOfferContractOrderCreated');
  }
});

console.log('Customer clicks on "Calculate offer" button...');
const calculateSalesOfferResponse = await graphQlClient?.mutation(
  `mutation calculateEtrmSalesOffer($tenantId: ID!, $offer:
     CalculateEtrmSalesOfferInput!) {
    calculateEtrmSalesOffer(tenantId: $tenantId, offer: $offer)
  }`, calculateSalesOfferProps);

if (calculateSalesOfferResponse?.error) {
```

```
  context.assert.fail('received error', JSON.stringify(
     calculateSalesOfferResponse.error, null, '\t'));
}

context.assert.notExists(calculateSalesOfferResponse.error);
context.assert.exists(calculateSalesOfferResponse.data);

console.log('Customer interface is waiting for sales offer created
   event...');
await context.endToEnd.subscriptionHelper.waitForAllEvents(['
   salesOfferCreated'], 60);
console.log('Customer interface received \'salesOfferCreated\' event
   ');

// Check that 'Sales offer created' event is also received in ETRM
   App
const message = await etrmClient.waitForMessage('Sales offer created
   ', (payload: any) => payload.offerId === offerId, 60);
context.assert.exists(message, 'Sales offer created received');

// Check that offer calculation exists in the database
const getOfferCalculationResponse = await etrmClient.GET('/
   offercalculation/{offerId}', {params: {path: {offerId}}});
context.assert.equal(getOfferCalculationResponse.response.status,
   200);
context.assert.exists(getOfferCalculationResponse.data);

// Check that sales offer exists in the database
const getSalesOfferResponse = await etrmClient.GET('/sales-offer/{
   offerId}', {params: {path: {offerId}}});
context.assert.equal(getSalesOfferResponse.response.status, 200);
context.assert.exists(getSalesOfferResponse.data);

const createSalesOfferProps = {
  tenantId: process.env.ORGANISATION ?? '',
  contractOrder: {
    billingInformation: {
```

```
      billingInterval: '1kk',
      eInvoice: false,
      eInvoiceBank: '',
    },
    billingAddress: undefined,
    contractInformation: {
      contractConfirmation: 'Email',
      additionalInformation: '',
    },
    contractEndDate: endDate.toISO() ?? undefined,
    offerId,
    customerType: 'Consumer',
    gsrn: meteringPoint.gsrnIdentifier,
    identifier: orderId,
    offerLengthCode: 6,
    optionalMargins: [],
  },
};

console.log('Customer accepts sales offer and submits application
   for contract order...');
const createContractOrderResponse = await graphQlClient?.mutation(
  `mutation createSalesOfferContractOrder($tenantId: ID!,
     $contractOrder: SalesOfferContractOrderInput!) {
    createProductContractOrder(tenantId: $tenantId, contractOrder:
       $contractOrder)
  }`, createSalesOfferProps);

if (createContractOrderResponse?.error) {
  context.assert.fail('received error', JSON.stringify(
     createContractOrderResponse.error, null, '\t'));
}

console.log('Customer interface is waiting for sales offer contract
   order created event...');
await context.endToEnd.subscriptionHelper.waitForAllEvents(['
   salesOfferContractOrderCreated'], 60);
```

```
  console.log('Customer interface received \'
     salesOfferContractOrderCreated\' event');

  // Check that contract order exists in the database
  const contractOrder = await context.endToEnd.clients.rest.GET('/api/
     contractorder/{orderId}', {params: {path: {orderId}}});
  context.assert.equal(contractOrder.response.status, 200);
  context.assert.exists(contractOrder.data);
  context.assert.equal(message?.payload.offerId, offerId);

  console.log('Contract order for order was created successfully!');
}).tags(['@etrm']);
```

addProperty Metodi lisää ominaisuuden rakennettavalle kontekstille.

Parametrit:

name **String** ominaisuuden nimi

propertyFactory `(config: TConfig)=> Promise<TProperty>`

addClientFactory Metodi lisää tenant-kohtaisen asiakasohjelman rakennettavalle kontekstille. Lisätyt asiakasohjelmat löytyvät kontekstin ominaisuuden `clients` sisältä.

Parametrit:

name **String** asiakasohjelman nimi

clientFactory `(config: TConfig)=> Promise<CleanableResource< TClientInstance>>`

acceptHooks Lisää kontekstille `addHook`-metodin, jonka avulla voidaan lisätä hookeja testiajon aikana. Metodi `addHook` ottaa yhden parametrin `hook`, joka on asynkroninen funktio jonka sisällä voidaan alustaa resursseja. Hookin paluuarvona palautetaan `cleanup`-funktio, jonka sisällä alustetut resurssit voidaan purkaa, ja sitä kutsutaan automaattisesti testiajon päätyttyä purku-hookissa.

addConfiguration Metodi alustaa ja lisää sisäisen tilan rakennettavalle kontekstille. Tila välitetään tehdasfunktioille parametrina rakennusprosessissa.

Parametrit:

configFactory `() => Promise<CleanableResource<TConfig>>`

configureUserBuilder Metodilla konfiguroidaan kontekstin metodi `createUser` käyttäjien luomiseen. Funktio ottaa parametrin `userBuilderConfigurationFactory`, joka palauttaa paluuarvona funktion, jolle voi antaa mukautetut parametrit `...args`, jotka kontekstin `createUser`-metodi ottaa. Funktio palauttaa paluuarvona rakentajan `UserBuilder`, jonka avulla kontekstille luodaan käyttäjien instansseja.

Parametrit:

userBuilderConfigurationFactory `(config: TConfig)=> Promise<(...args : TParameters)=> Promise<UserBuilder<TUserBuilderResult, TUserConfig>>>`

build Metodi luo ja palauttaa uuden instanssin kontekstista rakentajalle annettujen ohjeiden perusteella.

Kuvio 6: Konteksin rakentajan `Builder`-metodit.

`addProperty` Metodi lisää ominaisuuden rakennettavalle käyttäjälle.

Parametrit:

name **`String`** ominaisuuden nimi

propertyFactory `(config: TConfig)=> Promise<TProperty>`

`addConfiguration` Metodi alustaa ja lisää sisäisen tilan rakennettavalle käyttäjälle. Tila välitetään tehdasfunktioille parametrina rakennusprosessissa.

Parametrit:

configFactory `()=> Promise<CleanableResource<TConfig>>`

`configureAuthenticatedUserBuilder` Metodilla konfiguroidaan käyttäjän metodi `authenticateUser` käyttäjän kirjautumiseen. Funktio ottaa parametrin `authenticatedUserBuilderConfigurationFactory`, joka palauttaa paluuarvona funktion, jolle voi antaa mukautetut parametrit `...args`, jotka käyttäjän `authenticateUser`-metodi ottaa. Funktio palauttaa paluuarvona rakentajan `AuthenticatedUserBuilder`, jonka avulla käyttäjälle luodaan kirjautuneen käyttäjän instansseja.

Parametrit:

userBuilderConfigurationFactory `(config: TConfig)=> Promise<(...args: TParameters)=> Promise<AuthenticatedUserBuilder<TAuthenticatedUserBuilderResult, TUserConfig>>>`

`build` Metodi luo ja palauttaa uuden instanssin kontekstista rakentajalle annettujen ohjeiden perusteella.

Kuvio 7: Käyttäjän rakentajan `UserBuilder`-metodit.

`addProperty` Metodi lisää ominaisuuden rakennettavalle kirjautuneelle käyttäjälle.

Parametrit:

name **`String`** ominaisuuden nimi

propertyFactory `(config: TConfig)=> Promise<TProperty>`

`addConfiguration` Metodi alustaa ja lisää sisäisen tilan rakennettavalle kirjautuneelle käyttäjälle. Tila välitetään tehdasfunktioille parametrina rakennusprosessissa.

Parametrit:

configFactory `()=> Promise<CleanableResource<TConfig>>`

`addClientCreatorFactory` Metodilla lisätään käyttäjäkohtainen asiakasohjelma kirjautuneelle käyttäjälle. Funktio ottaa parametrin `clientCreatorFactory`, joka palauttaa paluuarvona funktion, jolle voi antaa mukautetut parametrit `...args`, jotka asiakasohjelman alustusmetodi `init`-metodi ottaa. Funktio palauttaa paluuarvona alustetun asiakasohjelman.

Parametrit:

name **`String`** asiakasohjelman nimi

clientCreatorFactory `(config: TConfig)=> Promise<(...args: TParameters)=> Promise<CleanableResource<TClientInstance>>>`

`build` Metodi luo ja palauttaa uuden instanssin kontekstista rakentajalle annettujen ohjeiden perusteella.

Kuvio 8: Kirjautuneen käyttäjän rakentajan `AuthenticatedUserBuilder`-metodit.