Yksikkö- ja integraatiotestaus .NET ympäristössä

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Ammattikorkeakoulun opinnäytetyö

Tieto- ja viestintätekniikka, insinööri (AMK)

kevät 2023

Joona Kleemola

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tieto- ja viestintätekniikan koulutusohjelma
Tekijä Joona Kleemola
Työn nimi Yksikkö- ja integraatiotestaus .NET ympäristössä
Ohjaaja Joni Järvenpää

Tiivistelmä
Vuosi 2023

Opinnäytetyön tavoitteena oli luoda sovelluksen .NET-rajapintasovellukseen automaatiotestausta yksikkö- ja integraatiotestien muodossa. Kyseisessä sovelluksessa ei automaatiotestausta ole aikaisemmin ollut käytössä, joten testauksen luontiin lähdetään tyhjältä pöydältä, ja tällöin testien pohjatyön luominen on tärkeässä asemassa. Kun testien pohjat ja mallit on tehty hyvin, niin muiden kehittäjien on helppo lähteä kasvattamaan testikattavuutta.

Toimeksiantajana työssä toimi IT-alan yritys, Riihicloud Oy. Riihicloudin ohjelmiston luotettavuutta ja vakautta kehitetään jatkuvasti, ja tätä tukemaan haluttiin ottaa käyttöön automatisoitua testausta, jolloin manuaalisen testauksen tueksi saadaan nopeasti ajettavia yksikkötestejä ja laajempaa kokonaisuutta testaavia integraatiotestejä. Tällöin myös kehitysvaiheessa tulleita mahdollisia virheitä ja bugeja voidaan saada kiinni ennen manuaalisen testauksen aloitusta.

Toteutuksessa haettiin testaukseen käytettävien pohjaluokkien luomista ja testimallien tekoa, eli laajempi testikattavuus jätetään sovelluksen kehitystiimin tehtäväksi. Työn valmistuttua testikattavuuden laajentaminen on kevyempi tehdä, kun testeistä löytyy malleja ja esimerkkejä, sekä niiden kehittämisen tukemiseksi on lisätty apuluokkia. Tällöin myös vähemmän testejä tehnyt kehittäjä pääsee tehokkaammin alkuun testien luomisessa.

Testit toteutettiin käyttämällä XUnit .NET-kirjastolla. Testien pohjien ja apuluokkien luomisen jälkeen sovellukselle luotiin automatisoituja testejä, joita kehittäjä voi ajaa esimerkiksi Visual Studio työkalulla, ja tämän lisäksi ne lisättiin ajettavaksi sovelluksen Azure Pipeline julkaisuputkeen, jolloin testit ajetaan ennen, kun sovelluksesta tehdään uutta julkaisuversiota.

Avainsanat .NET, yksikkötestaus, integraatiotestaus, XUnit
Sivut 28 sivua

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Information and communication technology, Bachelor of Engineering
Author Joona Kleemola
Subject Unit- and integration tests in .NET framework
Supervisor Joni Järvenpää

Abstract
Year 2023

The aim of the thesis was to create an application for .NET interface in the form of automated testing, including unit and integration tests. The application in question had no previous automated testing, so the creation of the testing was started from scratch, and thus, the creation of base classes for testing was crucial. Once the test groundwork and models are well established, it will be easier for other developers to increase test coverage.

The client for this project was an IT company called Riihicloud Oy. Riihicloud's software reliability and stability are continuously being improved, and automated testing was introduced to support this. This enables quickly executable unit and integration tests, that test a larger context, in addition to manual testing. In this way, potential errors and bugs that may occur during the development phase can be caught before manual testing begins.

The implementation of the thesis involved the creation of base classes and test models for testing, with the responsibility of developing a broader test coverage left to the application development team. After the completion of the thesis, expanding test coverage will be easier to do since there are models and examples of tests and supporting helper classes that have been added. This way, even a developer with less experience in testing can begin creating tests more efficiently.

The tests were implemented using the XUnit .NET library. After creating the test base and helper classes, automated tests were created for the application. Developers can run these tests, for example, using the Visual Studio tool, and the tests were also added to be run in the application's Azure Pipelines. This way, the tests are run before a new release version of the application is made.

Keywords .NET, unit testing, integration testing, XUnit
Pages 28 pages

Sisällys

Kuvat, taulukot ja kaavat

1 Johdanto

Tämän opinnäytetyön aiheena on yksikkö- sekä integraatiotestaamisen suunnittelu ja implementointi .NET pohjaiseen REST-rajapintaan, sekä testien integrointi sovelluksen julkaisuputkeen. Työn tavoitteena on luoda pohjaa laajalle testikattavuudelle luomalla yksikkö- sekä integraatiotestiprojektit, ja näiden alle muutamia mahdollisimman monipuolisia testejä. Näitä testejä voidaan jatkossa käyttää esimerkkeinä sovelluksen kehittäjien toimesta, jolloin myös vähemmän ohjelmistotestejä tehnyt kehittäjä pääsee nopeasti ja tehokkaasti alkuun testien teon kanssa, ja täten testauskattavuutta pystytään jatkossa kasvattamaan muun kehityksen ohessa.

1.1 Toimeksiantaja

Opinnäytetyön toimeksiantajana ja tilaajana toimi Riihicloud Oy, jonka ohjelmistoon oli tavoitteena lisätä automatisoituja yksikkö- ja integraatiotestejä. Riihicloud Oy on perustettu vuonna 2021, kun Riihicloud eriytettiin ohjelmistotalo Riihisoft Oy:stä. Riihicloudin tarkoituksena on keskittyä ICT-palveluiden tuottajien palveluliiketoiminnan tehokkuuden ja tuottavuuden parantamiseen. (Riihicloud, n.d.) Opinnäytetyössä luotavat testit tulevat koskemaan vain Riihicloudin kehittämän ohjelmiston .NET-pohjaista REST-rajapintaa.

1.2 Lähtötilanne ja tavoitteet

Projektin työlistalla on jo pitkään ollut automatisoidun testaamisen lisääminen, jotta saataisiin helpotettua jokaisen julkaisun yhteydessä tehtävää manuaalista testaamista. Tätä ei kuitenkaan ole resurssien puutteen takia pystytty priorisoimaan korkeammalle. Opinnäytetyön tavoitteena ja tarkoituksena oli lähteä luomaan pohjaa automaatiotestaukselle, sekä luoda hyviä ohjenuoria ja esimerkkejä tulevaa testikattavuuden lisäämistä varten. Pohjaksi testikattavuuden lisäykselle haluttiin lähteä luomaan apu- ja pohjaluokkia, joiden päälle olisi helppo tehdä testejä myös semmoisten kehittäjien toimesta, ketkä eivät ole aikaisemmin testejä kirjoittaneet. Tätä tukemaan tavoitteeksi otettiin myös mahdollisimman kattavan dokumentoinnin teko pohjaluokkien puolelle, sekä esimerkkitestien luominen. Esimerkkitestien luomisella haluttiin antaa esimerkkejä ja perusideaa siitä, mitä yksikkö- ja integraatiotesteillä halutaan testata.

2 Teknologiat ja työkalut

Käytettävien teknologien osalta ei tarvinnut tehdä laajaa kartoittamista, tai arkkitehtuurin miettimistä, sillä itse sovellus on jo pitkällä kehityksessä, eli suuremmat arkkitehtuuriset valinnat on tehty ennen tämän projektin aloitusta. Suunnittelu ja käytettävien työkalujen miettiminen rajautui hyvin pitkälti käytettävän testauskehyksen valitsemiseen ja käytettävissä olevien kehyksien vertailuun.

2.1 .NET ohjelmistokehys

Microsoftin kehittämä .NET-ohjelmistokehys on ilmainen järjestelmäriippumaton avoimen lähdekoodin ohjelmistokehys, joka mahdollistaa ohjelmien kehittämisen monipuolisesti eri alustoille. .NET-sovelluksia voi esimerkiksi kehittää Windowsin, Linuxin, macOS:n tai Dockerin päälle. (Microsoft, n.d.-b) Ohjelmistokehys tukee useita eri kieliä, näistä eniten esillä ovat Visual Basic, C#, sekä F# (Microsoft, n.d.-c). .NET-kehyksen ominaisuuksiin kuuluvat esimerkiksi asynkroninen koodin ajo, attribuuttien käyttö, sekä tyypitys (Microsoft, 2023-d). Se on myös suunniteltu toimimaan saumattomasti Microsoftin kehittämien työkalujen, kuten Visual Studion ja Azure-pilviympäristön kanssa.

.NET-ohjelmistokehys käyttää Common Language Runtime (CLR) -komponenttia, joka vastaa .NET-ohjelmien suorittamisesta. CLR käyttää just-in-time (JIT) -kääntämistä, eli se kääntää ohjelman lähdekoodin konekielelle vasta suoritusaikana, kun ohjelmaa käytetään ensimmäistä kertaa. Tämä auttaa vähentämään käynnistysaikaa ja vähentää myös muistin käyttöä. CLR huolehtii myös muistin hallinnasta .NET-ohjelmassa. CLR:n ominaisuuksiin kuuluvat esimerkiksi tyypityksen varmistus, usean ohjelmointikielen tuki, sekä automaattinen roskankeruun (eng. garbage collection), joka vapauttaa muistin, jota ei enää käytetä. CLR myös valvoo sovelluksen suoritusta ja havaitsee virheet, jotka voivat aiheuttaa sovelluksen kaatumisen. (Microsoft, 2023-a)

2.2 XUnit testauskirjasto

XUnit on ilmainen, avoimen lähdekoodin yksikkötestauskirjasto, joka on rakennettu .NET-ohjelmistokehykselle, jonka on kehittänyt toisen .NET-testauskirjaston NUnit alkuperäinen

luoja. XUnit tuo viimeisintä teknologiaa .NET-kielien kuten C# ja F# yksikkötestaamiseen. (.NET Foundation, n.d.)

XUnit tukee useita erilaisia testin suoritusmetodeja, kuten faktat (eng. facts) ja teoriat (eng. theories). Faktat ovat yksinkertaisia testejä, jotka ajetaan aina samalla kaavalla. Teoriat puolestaan ovat testejä, jotka hyväksyvät parametreja. Teoriat mahdollistavat parametrisoidut testit, joissa sama testifunktio suoritetaan eri syötteillä. XUnit tukee myös testin suoritusjärjestyksen määrittämistä, testien priorisointia, ja testitulosten luokittelua luokkiin. Tämä auttaa kehittäjiä pitämään testit järjestyksessä ja antaa heille mahdollisuuden priorisoida tärkeät testit. (Vinugayathri, n.d.)

2.3 Azure DevOps CI/CD

Azure DevOps on Microsoftin kehittämä pilvipohjainen työkalu, joka tarjoaa kattavan työkalupaketin ohjelmistokehityksen- ja projektinhallintaan. Azure DevOpsin osio, jolla voidaan implementoida CI/CD julkaisuputki on nimeltään Azure Pipelines. (Compete 366, n.d.)

2.3.1 CI/CD

CI/CD lyhenne tulee englannin kielen sanoista Continuous Integration/Continuous Delivery or Deployment. Vapaa käännös suomeksi on jatkuva integrointi/jatkuva toimitus tai julkaisu. Tällä tarkoitetaan ohjelmistokehityksen käytäntöä, joka korostaa koodimuutosten säännöllisen usein automaattisesti toistuvaa testausta ja integroimista koodipohjaan, sekä ohjelmiston nopeaa toimittamista loppukäyttäjille. (Red Hat, 2022) CI/CD:n tavoitteena on mahdollistaa kehittäjille korkealaatuisen ohjelmiston tuottaminen tehokkaammin ja suuremmalla varmuudella.

Jatkuva integrointi on prosessi, jossa useiden kehittäjien koodimuutokset yhdistetään usein keskitettyyn koodipohjaan, jossa automatisoituja testejä ajetaan varhaisessa kehitysvaiheessa virheiden havaitsemiseksi. Tällä saadaan varmistettua, että koodimuutokset testataan ja integroidaan mahdollisimman pieninä palasina, jolloin vähennetään konfliktien ja muiden ongelmien riskiä.

Jatkuva toimitus on CI/CD putken toinen osa, jossa ohjelmiston julkaisu loppukäyttäjien käytössä oleviin tuotantoympäristöihin automatisoidaan, varmistaen julkaisujen olevan yhdenmukaisia, toistettavia ja luotettavia. Tällä eliminoidaan ongelmat, joita voi syntyä manuaalisten tuotantojulkaisuiden yhteydessä. Tämän lisäksi julkaisusyklit saadaan pidettyä nopeampina. (Red Hat, 2022)

2.3.2 Azure Pipelines

Azure Pipelines on Microsoftin pilvipohjainen CI/CD-palvelu, joka mahdollistaa sovelluksen automaattisen rakentamisen, testaamisen ja julkaisemisen. Se tukee kaikkia yleisiä ohjelmointikieliä ja projektityyppejä (Microsoft, 2023-c).

Palvelun avulla kehittäjät voivat luoda ja hallita julkaisuputkia erilaisten sovellustyyppien, kuten mobiili-, web- ja työpöytäsovellusten osalta. Putket voidaan räätälöidä sisältämään automatisoituja testejä ja koodilaadun tarkistuksia, mikä mahdollistaa kehittäjien virheiden havaitsemisen jo varhaisessa kehitysvaiheessa ja varmistaa, että koodi täyttää vaaditut standardit, ennen kuin sovellusta julkaistaan ajoympäristöön.

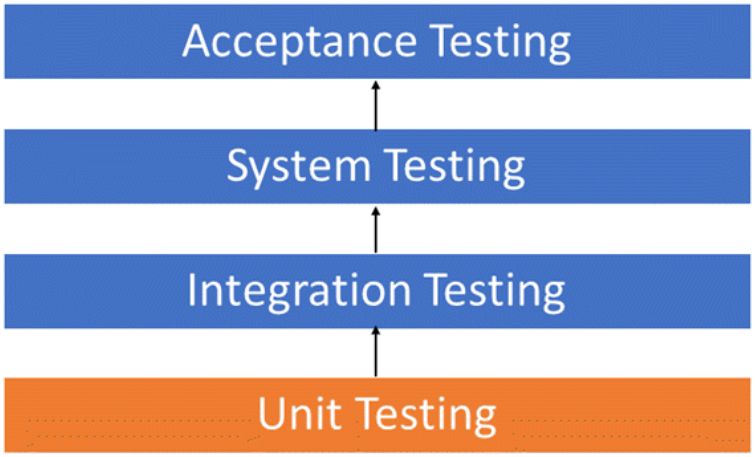
Azure Pipelines on saatavilla osana Azure DevOps -palvelua, joka tarjoaa kattavan joukon työkaluja ja palveluita ketterään ohjelmistokehitykseen, kuten lähdekoodinhallintaan, työn seurantaan ja projektinhallintaan. (Microsoft, n.d.-a)

2.4 Yksikkötestaus

Yksikkötestaus on ohjelmistokehityksen käytäntö, jossa testataan ohjelmiston yksittäisiä toiminnallisuuksia, kuten luokkia tai metodeja, erillisinä moduuleina. Tätä voidaan pitää ohjelmistotestauksen ”matalimman” tason testauksena. (Devolution, 2021) Ohjelmistotestauksen tasoja on kuvattu kuvassa 1. Yksikkötestauksen tarkoituksena on varmistaa, että ohjelmiston yksittäiset osiot toimivat ilman riippuvuuksia oikein ja täyttävät määrittelyvaatimukset ennen kuin ne yhdistetään kokonaisjärjestelmäksi. Yksikkötesteissä riippuvuuksia muihin luokkiin tai palveluihin voidaan katkaista käyttämällä mock-objekteja tai mockeja, joilla luodaan omanlainen simulaatio todellisen luokan tai palvelun toiminnasta. Testaaja voi määritellä mock-objektille, mitä sen pitäisi palauttaa tietyllä syötteellä tai millä

tavalla sen pitäisi käyttäytyä tietyissä tilanteissa, jolloin testi ei ole enää riippuvainen simuloidusta luokasta tai palvelusta.

Kuva 1. Ohjelmistotestauksen tasot (Guru99, n.d.).



Yksikkötestien kirjoittamiseen on useita eri tapoja. Testejä voidaan kirjoittaa ennen varsinaisen koodin kirjoittamista, jos seurataan TDD-ohjelmistokehitysprosessia (Test Driven Development), mutta ne voidaan kirjoittaa myös samanaikaisesti kehityksen ohessa, tai myös sen jälkeen. Yksikkötestien kirjoittamiseen on luotu useita kirjastoja, joilla testien generointia saadaan automatisoitua, mutta näiden käyttöä tulee harkita tarkkaan, sillä ne eivät jokaiseen projektiin sovi. (Software Testing Help, 2023) Yksikkötestejä voidaan kirjoittaa myös kehittäjän toimesta manuaalisesti, jolloin ne ovat parhaiten hallittavissa.

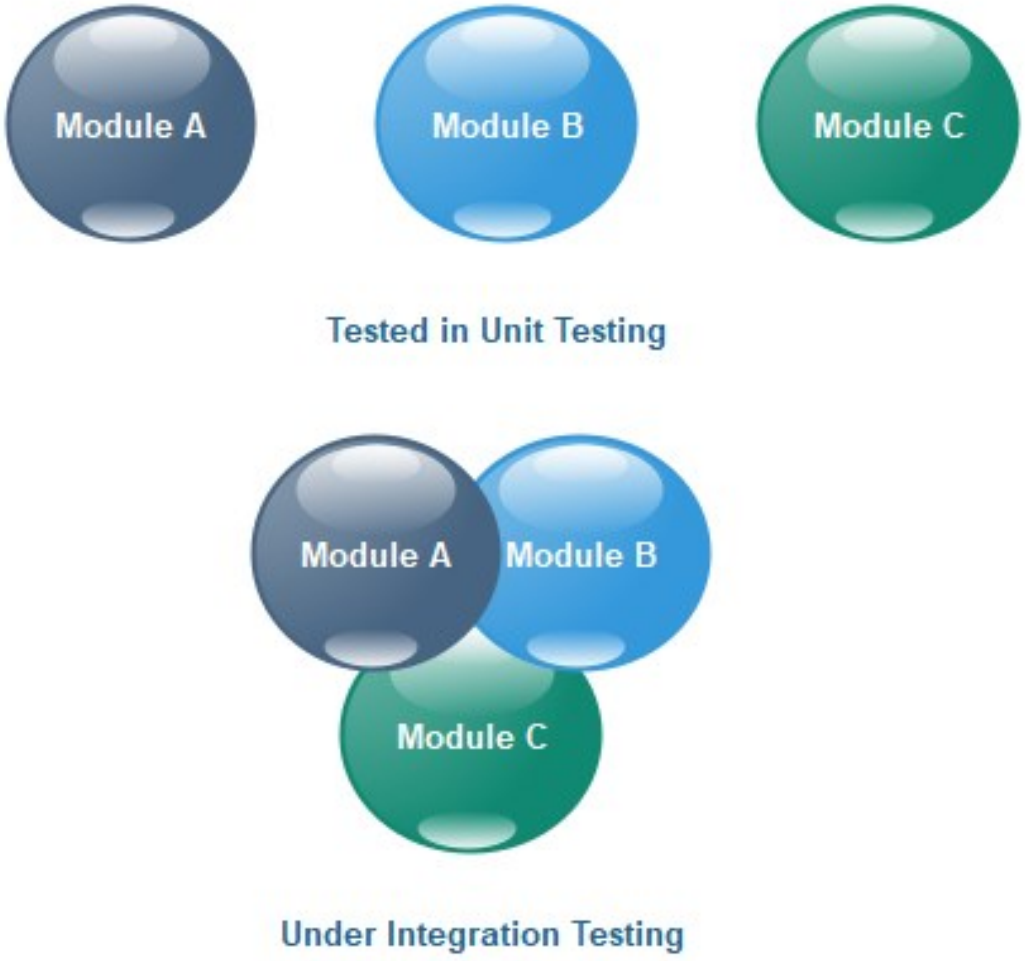
Yksikkötestit ovat tärkeä osa CI/CD-prosessia, sillä automatisoiduilla testeillä pystytään varmistumaan, että uudet koodimuutokset eivät riko vanhoja toiminnallisuuksia. Sen lisäksi yksikkötestauksen avulla voidaan myös parantaa ohjelmiston suunnittelua ja ylläpidettävyyttä, koska se vaatii kehittäjiä hajottamaan monimutkaiset toiminnot pienempiin, helpommin hallittaviin moduuleihin.

2.5 Integraatiotestaus

Integraatiotestaus on ohjelmistokehityksessä käytetty testauskäytäntö, jossa testataan usean komponentin tai moduulin toimintaa yhdessä. Tällöin saadaan testattua, että osat

toimivat odotetulla tavalla yhteen, kun osat on integroitu kokonaisuudeksi. Käytännössä kun yksikkötestissä on testattu kolmen luokan toimintaa erillisinä kokonaisuuksina, niin niiden toimivuutta yhtenä kokonaisuutena voidaan testata integraatiotestien kautta. Tätä rakennetta on havainnollistettu kuvassa 2.

Kuva 2. Integraatiotestaus havainnollistettuna (JavaTPoint, n.d.).



Yleensä integraatiotesteillä testataan myös laajemmin sovelluksen arkkitehtuuria, kuten tiedostojärjestelmän tai tietokantaan kohdistuvien transaktioiden toimintaa (Microsoft, 2023-b). Tämä tarkoittaa myös sitä, että toisin kuin yksikkötestien kohdalla, integraatiotestit käyttävät oikeita komponentteja testien ajamiseen, ja mock-objekteja pyritään käyttämään mahdollisimman vähän, jotta kaikki riippuvuudet saadaan testattua. Mock-objekteja pystytään kuitenkin käyttämään, jos joku sovelluksen komponenteista tai palveluista ei ole

saatavilla testiympäristössä. Tällöin voidaan käyttää mock service -ratkaisua, jossa simuloitu palvelu matkii todellisen palvelun käyttäytymistä ja vastauksia.

Vaikka sovelluksen eri osia olisikin testattu kattavasti yksikkötestien avulla, niin ongelmia ja virheitä voi kuitenkin syntyä eri komponenttien välillä useista syistä, kun ne yhdistetään kokonaisuudeksi. Esimerkiksi useamman moduulin kehityksessä on voinut olla mukana useampi sovelluskehittäjä, ja kehittäjien ymmärrys ja logiikka voivat poiketa kehittäjäkohtaisesti. Jatkokehitysvaiheessa on myös mahdollista, että tehdään tiettyihin komponentteihin uusia ominaisuuksia tai muutoksia, jotka ovat osaan kyseisiä komponenttia käyttävistä moduuleista rikkovia muutoksia. (Hamilton, 2023)

Integraatiotestien kirjoittaminen ja ajaminen on hitaampaa kuin yksikkötestien, joten on hyvä harkita tarkkaan, kuinka laajasti niitä alkaa kirjoittamaan. Sovelluksen yksittäisten osien logiikan ja rajatapauksien testaamiseen yksikkötestit ovat parempi ratkaisu. Integraatiotesteissä voidaan tällöin keskittyä tärkeiden koko sovelluksen infrastruktuuria koskevien tapausten käsittelyyn ja testaamiseen, sekä osioiden yhteen toimivuuden varmistamiseen.

3 Toteutuksen suunnittelu

Projektin toteuttaminen alkoi ohjelmistoon perehtymisen, sekä suunnitelman laatimisella. Suunnitelmaa varten aluksi oli hyvä kartoittaa mahdollisia testauskirjastoja, joita sovelluksen testaamiseen voitaisiin käyttää. Käytettävän testauskirjaston lisäksi suunnitteluvaiheessa mietinnässä oli se, että millä tapaa työstä olisi suurin hyöty jatkoa ajatellen. Testattava sovellus on hyvin laaja, joten ei ollut järkevää asettaa tavoitteeksi korkean testikattavuuden saamista, sillä se veisi hyvin paljon aikaa, eikä testien laatu tulisi olemaan sillä tasolla, kun se olisi sovelluksen parissa työskentelevän kehittäjän laatimana. Tämän takia tilaajan kanssa päädyttiin siihen lopputulokseen, että on parasta laatia pohjat ja raamit tulevalle testikattavuuden laajentamiselle kirjoittamalla testeille pohjaluokat, joiden avulla tulevien testien kirjoittaminen olisi mahdollisimman jouhevaa. Näiden pohjaluokkien lisäksi sovittiin, että pohjaluokkien luonnin ohessa tehtäisiin pieni määrä yksikkö- ja integraatiotestejä esimerkiksi, joista kehittäjät, jotka eivät ole testejä paljoa kirjoittaneet aikaisemmin, pystyvät ottamaan mallia, ja näin pääsevät paremmin alkuun testien kirjoittamisessa.

Esimerkkitestien tulisi näyttää mallia mahdollisimman eri tyylisten testien tekoon, ja eri testitapojen käyttöön.

3.1 Kirjastojen vertailu

Suunnittelu alkoi testauskirjastojen vertailulla. On olemassa monia mahdollisia kirjastoja, joilla kehittäjät voivat testata .NET-ohjelmistoja, mutta näistä kolme ovat ylitse muiden. Nämä kolme kirjastot ovat XUnit, NUnit sekä MSTest. Tämä kolmikko erottuu hyvin selvästi muista kirjastoista, joten muita vaihtoehtoja ei otettu mukaan vertailuun. Hyvin suuri painoarvo näiden kolmen valinnalle toi se, että Microsoft käyttää näitä kolmea myös omassa dokumentaatiossaan, sekä esimerkeissään.

3.1.1 MSTest

Kolme suosituinta .NET testikirjastoa ovat hyvin tasaväkisiä, ja kaikista löytyy omat hyvät, sekä huonot puolensa. MSTest on näistä kolmesta helpoin ottaa projektiin mukaan, sillä se ei vaadi erillisten pakettien lataamista, sillä MSTest tulee Visual Studio Code IDE:n mukana. MSTestillä voi suoraan kirjoittaa, ajaa, sekä tarkistella testien tuloksia. Tämän lisäksi sillä voidaan analysoida testien kattavuutta ilman, että ladataan erillisiä työkaluja. Huonona puolena MSTestin suorituskyky on hyvin huomattavasti heikompi kahteen isoimpaan kilpailijaansa verrattuna.

3.1.2 NUnit

NUnit-testauskirjasto on portattu Java ohjelmointikielelle luodusta testauskirjasto JUnit:sta. NUnit on paljon MSTestiä tehokkaampi testien ajossa, ja sen dokumentaatio on hyvin kattavaa. Huonoina puolina NUnit:lla ovat sen jokseenkin kompleksinen syntaksi, sekä tarve asentaa projektiin erillisiä paketteja ja työkaluja, jotta testausta pystytään tekemään.

3.1.3 XUnit .NET

XUnit on avoimen lähdekoodin testauskirjasto. Kirjasto on lähtöisin NUnit:n laajennoksesta, jonka on kirjoittanut yksi NUnit:n luojista. XUnit:n parhaita puolia ovat sen syntaksin

yksinkertaisuus ja helppo laajennettavuus. Sen takana on myös laaja yhteisö, jolloin sen kehitys pysyy hyvin .NET-versioiden mukana, ja sitä kehitetään nopealla syklillä. Huonona puolena muihin vaihtoehtoihin verrattuna XUnit:n käyttö voi kokemattomalle ohjelmoijalle olla aluksi hieman hankalaa muita kirjastoja heikomman dokumentaation takia.

3.1.4 Vertailun tulos

Kuvassa 3. on avattu testikirjastojen välillä olevia eroja karkeasti. Kuvassa oleva taulukko näyttää testien määrittelyyn käytettävien attribuuttien syntaksi erot. Testikirjaston valinta perustuu hyvin pitkälti kehittäjän tai kehitystiimin preferenssiin, mutta valinnassa on toki hyvä ottaa huomioon kirjaston tehokkuus, sekä dokumentoinnin ja tiedon saatavuus.

Näistä vaihtoehdoista XUnit valikoitui käytettäväksi kirjastoksi. Vaikka dokumentaatio ei ollut aivan muiden tasolla, niin sen syntaksi oli kehittäjän näkökulmasta helpointa kirjoittaa ja ymmärtää, sekä testien ajaminen on hyvin tehokasta. Microsoftin dokumentaatiosta löytyy myös paljon viittauksia XUnit-kirjastoon. Heikkoa dokumentaatiota korvaa laaja avoimen lähdekoodin yhteisö, josta saa apua tarvittaessa. Laajan yhteisön hyviin puoliin kuuluu, että suurella todennäköisyydellä eteen tuleviin ongelmiin löytyy myös valmiiksi vastaus internetin palstoilta.

Kuva 3. Testikirjastojen vertailu (BrowserStack, 2022).

Feature	MSTest	NUnit	xUnit
Test class	[TestClass]	[TestFixture]	NA
Test Method	[TestMethod]	[Test]	[Fact]
Initialization	[TestInitialize]	[Setup]	NA(constructor of the class is used for initialization)
Data driven test method	[DataTestMethod]	NA	[Theory]
Add parameters	[DataRow(_ , _)]	[TestCase(_ , _)]	[InlineData(_ , _)]
Documentation	Well Documented	Well Documented	Doesn't have good documentation
Tests Isolation	By default	Can be configured	By default

3.2 Pohjaluokat testaukselle

Toinen osa suunnittelua oli lähteä pohtimaan testeille pohjaa, jonka päälle pystyttäisiin rakentamaan erityyppisiä testejä. Yksikkö- ja integraatiotestit eroavat toisistaan, joten ne tulisivat molemmat tarvitsemaan omat pohja- ja apuluokkansa. Näiden pohja- ja apuluokkien määrittelyä varten tuli perehtyä ohjelmiston perustoimintaan, sekä mahdollisiin riippuvuuksiin, kuten tietokantaan ja sovelluksen käyttämiin Azure-resursseihin. Ohjelmiston kehitystiimin kanssa käytiin suunnitteluvaiheessa läpi kriittisiä luokkia ja toimintoja, joita vasten voitaisiin tehdä esimerkkitestit, jolloin ne saataisiin ainakin osin heti testikattavuuden piiriin.

Molemmat tulisivat tarvitsemaan omat tietokannat, joita vasten testejä voitaisiin ajaa, ja näihin olisi myös tarve ajaa alustavaa dataa, jotta testien ajo olisi ylipäänsä mahdollista. Paras tapa lähteä toteuttamaan tietokanta ratkaisua olisi se, että kanta pystytettäisiin aina

kun testiajo aloitetaan, jolloin testit eivät ole riippuvaisia kannasta löytyvästä datasta, vaan lähtötilanne olisi sama jokaisella ajokerralla.

Ohjelmiston oma tietokanta on pilvipalvelu Azuressa hostattu Azure SQL -tietokanta, joka vastaavat Microsoftin SQL server -tietokantaa. Tällöin luonnollinen valinta oli konfiguroida testit käyttämään Microsoftin LocalDB:tä, joka pystytettäisiin aina ennen testien ajoa, ja poistettaisiin ajon jälkeen. LocalDB on minimalistinen versio SQL serverin tietokanta moottorista, joka on kevyt ja kätevä käytettäväksi kehitys- ja testausvaiheissa. LocalDB-instanssin saisi myös pyörimään CI/CD putkessa, jolloin testien ajaminen on mahdollista myös CI/CD putken sisällä.

Testit olisi mahdollista määrittää käyttämään myös in-memory-tietokanta rakennetta, jolloin data tallennetaan palvelimen RAM-muistiin, jolloin se toimii välimuistin tapaan. Tämä ei kuitenkaan ole suositeltu keino, sillä in-memory-kanta ei vastaa relationaalista tietokantaa, joten testit eivät vastaa oikeaa tuotannossa olevaa sovellusta. Tämän lisäksi in-memory-kantaa vasten testaamista ei tueta testauskirjastojen toimesta, joten sen implementointi voisi myös tuottaa haasteita.

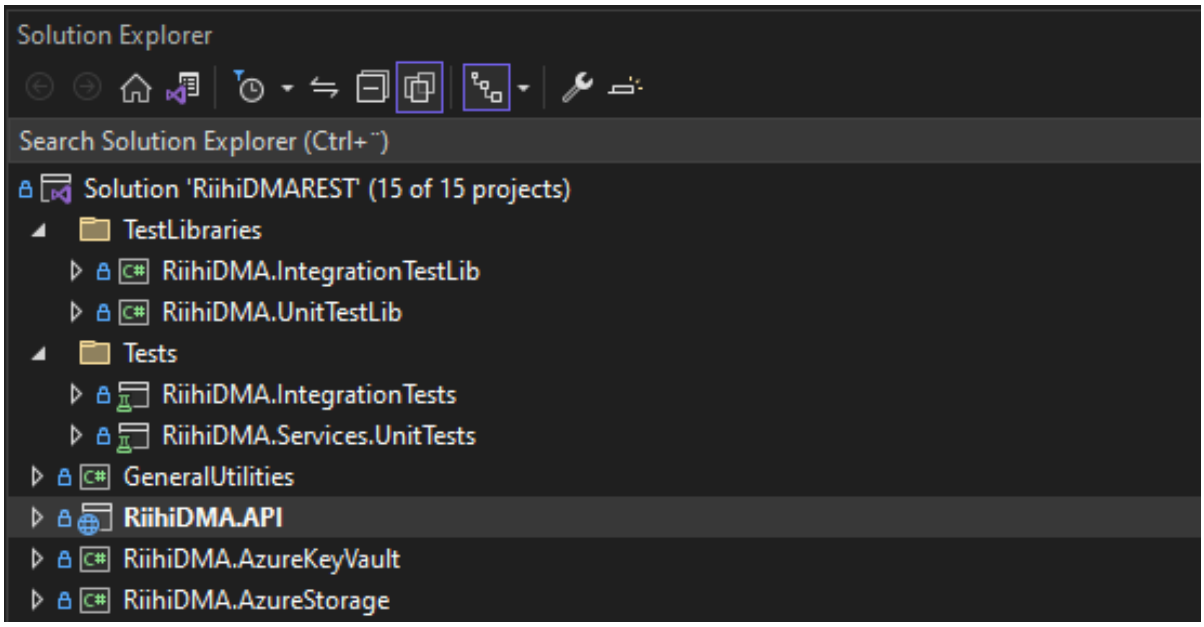
4 Testauksen toteuttaminen

Projektin toteutus aloitettiin kehittämällä yleiskäyttöisiä apuluokkia, joita pystyttäisiin käyttämään useissa eri projekteissa, jolloin testausprojektien luonti olisi jatkossa helpompaa myös muihin pienempiin ohjelmistoprojekteihin, jotka ovat pääprojektin alla. Kehityksessä oli käytössä Visual Studio Professional, joka on Microsoftin kehittämä IDE, eli integroitu kehitysympäristö. Visual Studio itsessään ei tue mitään kieltä, vaan asennuksen yhteydessä pystytään määrittämään liitännäisiä, jotka antavat tuen lähes millä vain kielellä kehittämiseen. Visual Studiosta löytyy hyvin laaja valikoiman .NET kehitystä tukevia liitännäisiä. Visual Studiossa kehitettäviä ohjelmisto- ja sovelluskehitysprojekteja kutsutaan nimellä solution. Solution voi sisältää yhden tai useamman erillisen projektin, jotka voivat olla esimerkiksi luokkakirjastoja, komponentteja, testikirjastoja tai konsolisovelluksia.

4.1 Yksikkötestit

Kun apuluokat oli saatu aluilleen, siirryttiin varsinaisten testiprojektien luomiseen. Projektisuunnitelman mukaan toteutusta lähdettiin tekemään yksikkötestien puolelta ensin, sillä ne ovat yksinkertaisempia rakenteelta, sekä niiden pohja on helpompi rakentaa. Tällöin projektin kanssa pääsee helpommin ja jouhevammin liikkeelle. Visual Studiossa luotiin omat kansiot testiprojekteille, jotta Visual Studio -projekti pysyisi mahdollisimman luettavana, ja tarvittavat projektit löytyisivät nopeasti. Projektin kansiorakennetta on kuvattu kuvassa 4.

Kuva 4. Sovelluksen projektirakenne Visual Studiossa.



Testauskansion alle lähdettiin luomaan ensimmäistä yksikkötestiprojektia. Yksikkötestejä varten on hyvä luoda oma testiprojekti vastaamaan solutionin alta löytyvää projektia. Ensimmäisenä työn alle päätyi testien luomisen sovelluksen service-tasolle, jossa tapahtuu suurin osa sovelluksen datan käsittelystä ja logiikasta. Testiprojektin luomiseen on hyvä käyttää yhtenäistä nimeämistapaa, jolloin testiprojektien rakentaminen ja ajo onnistuu CI/CD putkessa helposti.

4.1.1 Pohja- ja apuluokat

Pohja- ja apuluokilla saadaan luotua pohja ja tarvittavat rakenteet testien ajolle, kuten esimerkiksi testeihin käytettävän tietokannan määrittely sekä testidatan syöttö. Pohjaluokka toimii kaikkien testien perustana, ja se määrittelee testiajon alustuksen ja lopetuksen.

Apuluokat puolestaan tarjoavat hyödyllisiä apufunktioita, joiden avulla testien kirjoittaminen on helpompaa ja nopeampaa. Näiden suunnitteluun on hyvä käyttää aikaa, jotta niitä voidaan käyttää erilaisten testien yhteydessä. Näin ei tulisi tarvetta luoda erilaisia testipohjia eri yksikkötestiprojekteille.

Sovelluksen service-tason arkkitehtuuri on rakennettu siten, että tältä tasolta on yhteys sovelluksen tietokantaan, joten testien ajoa varten tarvittiin oma instanssi LocalDB, jossa on alustettuna testejä varten tarvittavaa dataa. Testikannan määrittelyyn on hyvä luoda oma DatabaseFixture-luokka, joka sisältää tietokannan luomisen ja testidatan syötön. Fikstuuri-luokkaa esitellään kuvassa 5.

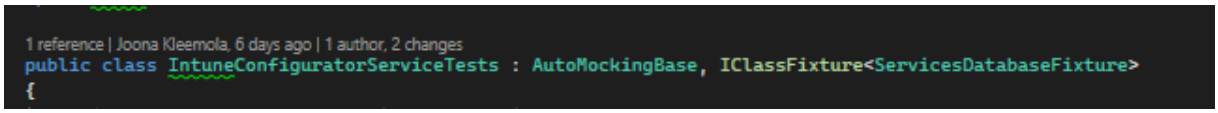
Kuva 5. Yksikkötestien fikstuuri-luokka.

```
4 references | Joona Kleemola, 6 days ago | 1 author, 2 changes
public class ServicesDatabaseFixture : AutoMockingBase
{
    private readonly UserResolverService _userResolverService;

    private const string ConnectionString = @"Server=(localdb)\mssqllocaldb;Database=RiihiDMAUnitTest;Trusted_Connection=True";
    private static readonly object _lock = new();
    private static bool _databaseInitialized;

    0 references | Joona Kleemola, 6 days ago | 1 author, 2 changes | 0 exceptions, - live
    public ServicesDatabaseFixture()
    {
        _userResolverService = new UserResolverService(GetMock<IHttpContextAccessor>().Object);

        lock (_lock)
        {
            if (_databaseInitialized) return;

            using var context = CreateContext();

            context.Database.EnsureDeleted();
            context.Database.EnsureCreated();

            ServicesUnitTestData.SeedData(context);

            context.SaveChanges();
            _databaseInitialized = true;
        }
    }

    2 references | Joona Kleemola, 45 days ago | 1 author, 1 change | 0 exceptions, - live
    public RiihiDMADbContext CreateContext()
        => new(
            new DbContextOptionsBuilder<RiihiDMADbContext>()
                .UseSqlServer(ConnectionString) // DbContextOptionsBuilder<RiihiDMADbContext>
                .Options, _userResolverService);
}
```

Testikanta määritellään käyttämään LocalDB-instanssia, jolloin testiympäristö saadaan nopeasti ajettua ylös ja alas, sekä testit saadaan ajettua aina samanlaista alkutilannetta vasten. Tietokanta puhdistetaan EnsureDeleted-metodin avulla, jolloin edellisen ajon mahdolliset muutokset kantaan saadaan nollattua. Tämän jälkeen kannan olemassaolo varmistetaan EnsureCreated-metodilla, jonka jälkeen kantaan syötetään haluttu lähtötilanne SeedData-metodilla. Ennen kun itse testiluokka saa kannan kontekstin käyttöön SeedData-metodin lisäämät datat tallennetaan kantaan SaveChanges-metodia kutsumalla. Testiluokkaan tämän fikstuurin saa implementoimalla XUnitin IClassFixture-rajapinnan, jota

käytetään testiluokkien aloitustilan luomiseen ja tilan purkamiseen liittyvää koodia. Rajapinnalle asetetaan parametrina aikaisemmin luotu ServiceDatabaseFixture kuvassa 6 esitetyllä tavalla. Tällöin haluttua fikstuuriluokkaa käytetään testiluokan luomiseen, sekä purkamiseen.

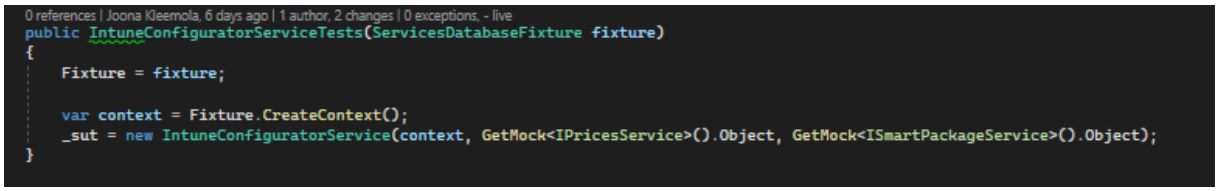
Kuva 6. Testiluokan määrittely.



Testiluokka perii myös AutoMockingBase-luokan, joka on aikaisemmassa projektin vaiheessa luotu apuluokka. Tämä luokka sisältää geneerisiä ja hyödyllisiä apufunktioita, joiden avulla esimerkiksi mock-objektien luonti on testiluokan sisällä helppoa. AutoMockingBase-luokkaan on implementoitu metodi GetMock, joka palauttaa mock-objektin halutusta luokasta. Käyttäen GetMock-metodia testattavan luokan instanssiointiin vaadittavat riippuvuudet saadaan mockattua, jolloin yksikkötestit saadaan tehtyä ilman riippuvuutta toiseen luokkaan.

Kuvassa 7. näkyvä IntuneConfiguratorService on testattava luokka, ja sen käyttämiä riippuvuuksia IPricesService- ja ISmartPackageService-rajapintoihin on korvattu mock-objekteilla. Ensimmäisenä parametrinä on IntuneConfiguratorService:n käyttämä tietokantakonteksti (DBContext), joka luodaan fikstuuri-luokan CreateContext-metodilla.

Kuva 7. Testiluokan konstruktori.



4.1.2 Testien kirjoittaminen

Pohja- ja apuluokkien jälkeen tulee itse testien kirjoittaminen. Yksikkötestit voivat testata hyvin yksinkertaisia asioita. Testeillä voidaan esimerkiksi varmistua, että tietyissä tilanteissa saadaan haluttu virheilmoitus, tai että tietty lähtödata palauttaa käsittelyn jälkeen oikean lopputuloksen. Tämän pohjalta lähdettiin luomaan malliksi muutamia testitapauksia sovelluksen service-tasoa vasten.

Kuvassa 8 esitellään esimerkkinä olevaa yksinkertaista testiä. Testimetodille annetaan XUnitin Fact-attribuutti, joka indikoi, että kyseessä on testi metodi. Fact-attribuutilla kerrotaan, että testimetodissa ei käytetä parametreja, vaan testiajo ajaa pelkän metodin ja käyttää sen sisällä määriteltyjä tietoja testin ajamiseen. Tämä mallitesti varmistaa, että sovellus antaa virheen, jos tilauksen käsittelijäksi liitettävää käyttäjää ei ole olemassa. Metodien nimeämisessä on hyvä käyttää yhtenäistä nimeämistapaa, jotta testin nimen lukemalla saa jo selville mitä tapahtumaa testataan. Jos testattava asia on monimutkainen, niin testin kuvausta voi myös avata esimerkiksi lisäämällä kommentteja tai käyttämällä metodin yhteenveto (eng. summary) tageja

Kuva 8. Yksinkertainen yksikkötesti.

```
[Fact]
0 references | Joona Kleemola, 6 days ago | 1 author, 2 changes | 0 exceptions, - live
public async Task AssignToUser_UserNotFound_ThrowsException()
{
    // Arrange
    var order = new OrderDao { OrderId = "FooFoo", PartnerId = "BarFoo" };
    var user = new DMAUserDao
    {
        Id = "DoesNotExist"
    };

    // Act and Assert
    await Assert.ThrowsAsync<IntuneConfiguratorServiceException>( testCode: () => _sut.AssignToUser(order, user.Id));
    Assert.Null(order.Worker);
    Assert.Equal( expected: OrderDao.OrderStatus.New,  actual: order.Status);
}
```

Testeillä voidaan myös varmistaa, että joku tietty logiikka toteutuu, kun metodia kutsutaan. Tilauksen luonnin yhteydessä on logiikka, että jos tilaukselle on määritelty IsTemplate ominaisuuden arvoksi tosi (eng. true), niin tällöin tilauksen statuksen arvoksi pitäisi tulla ”Complete”. Tätä yksinkertaista logiikkaa voitaisiin testata esimerkiksi kuvassa 9. olevalla yksikkötestillä. XUnitin Assert-metodilla voidaan tarkastaa, että saadun vasteen (eng. result) status on haluttu ”Complete”-status.

Kuva 9. Logiikan testausta yksikkötestillä.

```
[Fact]
0 references | Joona Kleemola, 6 days ago | 1 author, 1 change | 0 exceptions, - live
public async Task Create_IfOrderIsTemplate_StatusIsComplete()
{
    // Arrange
    var order = new OrderDao
    {
        OrderId = "ABC123",
        PartnerId = "BarFoo",
        IsTemplate = true,
        Configurations = new List<PackageConfigurationDao>()
    };

    // Act and Assert

    var result :OrderDao? = await _sut.Create(order);

    Assert.Equal( expected: OrderDao.OrderStatus.Complete,  actual: result.Status);
}
```

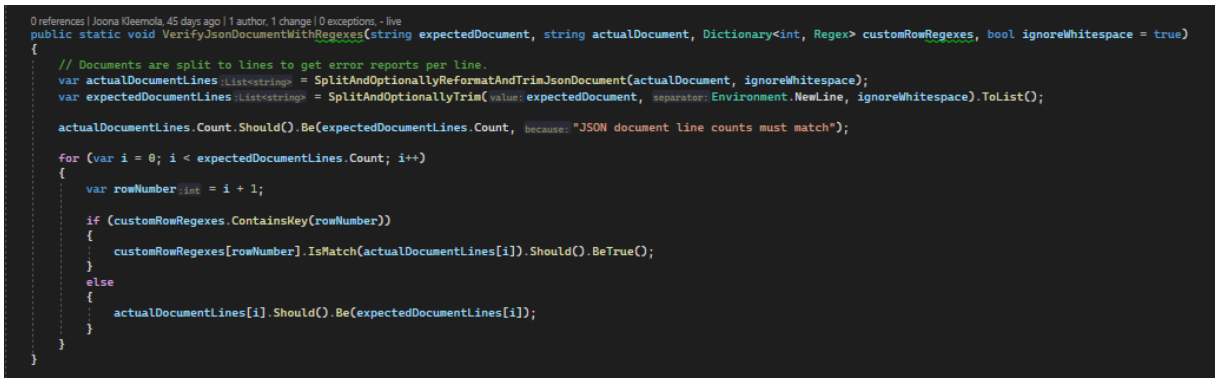
4.2 Integraatiotestit

Yksikkötestien luomisen jälkeen siirryttiin integraatiotestauksen puolelle. Integraatiotestien pohjan tekeminen on paljon työläämpää, kuin yksikkötestien, sillä integraatiotestejä varten tarvitsee konfiguroida yksikkötestien vaatiman tietokanta fikstuurin lisäksi myös muut sovelluksen osiot, kuten esimerkiksi auktorisointi, jotta testikutsut menevät läpi. Integraatiotesteillä testataan yleensä koko ketjua, joka tapahtuu ohjelmistoa kutsuttaessa, mutta tarvittaessa myös integraatiotestien kanssa voidaan käyttää mock-objekteja, jos eteen tulee riippuvuuksia, joita ei haluta testien piiriin ottaa. Mock-objekteilla voidaan esimerkiksi korvata riippuvuuksia, jotka ovat arvaamattomia, mahdollisesti välillä poissa käytöstä, tai riippuvuutena olevasta sovelluksesta ei ole testiversiota saatavilla, jota vasten testejä voitaisiin ajaa.

4.2.1 Pohja- ja apuluokat

Integraatiotestien apuluokkiin pyrin luomaan metodeja, joilla saisi vähennettyä testien puolelle kirjoitettavaa logiikkaa, jolloin itse testiluokat olisivat mahdollisimman siistejä, ja helposti luettavia. Tällöin esimerkiksi isommat JSON-vasteet, joita REST-rajapinta palauttaa, voidaan tarkistaa kutsumalla JSON tarkistuksiin luodulla apumetodilla VerifyJsonDocumentWithRegexes, ja samaa logiikkaa ei tarvitse kirjoittaa jokaiseen kohtaan, jossa halutaan verrata palautunutta JSON-vastetta, sekä odotettua JSON-vastetta. Regular expressioneilla, eli regexeillä saadaan helposti jätettyä vertailusta pois esimerkiksi päivämäärät, jotka ovat dynaamisesti muuttuvia.

Kuva 10. Apuluokan JSON-tarkistus metodi.



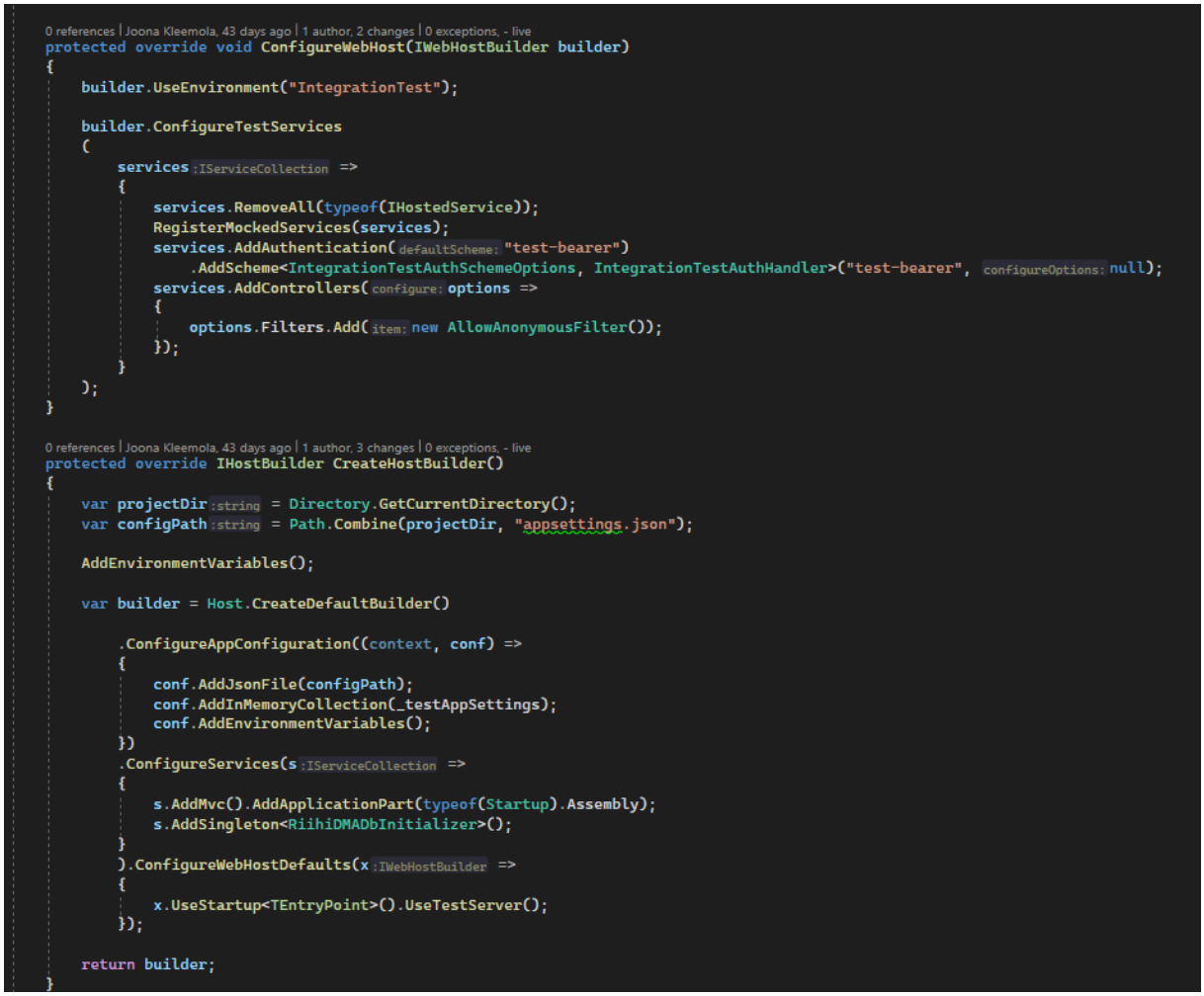
```
0 references | Joona Kleemola, 45 days ago | 1 author, 1 change | 0 exceptions. - live
public static void VerifyJsonDocumentWithRegexes(string expectedDocument, string actualDocument, Dictionary<int, Regex> customRowRegexes, bool ignoreWhitespace = true)
{
    // Documents are split to lines to get error reports per line.
    var actualDocumentLines :List<string> = SplitAndOptionallyReformatAndTrimJsonDocument(actualDocument, ignoreWhitespace);
    var expectedDocumentLines :List<string> = SplitAndOptionallyTrim( value: expectedDocument, separator: Environment.NewLine, ignoreWhitespace).ToList();

    actualDocumentLines.Count.Should().Be(expectedDocumentLines.Count, because: "JSON document line counts must match");

    for (var i = 0; i < expectedDocumentLines.Count; i++)
    {
        var rowNumber :int = i + 1;

        if (customRowRegexes.ContainsKey(rowNumber))
        {
            customRowRegexes[rowNumber].IsMatch(actualDocumentLines[i]).Should().BeTrue();
        }
        else
        {
            actualDocumentLines[i].Should().Be(expectedDocumentLines[i]);
        }
    }
}
```

Apuluokkien lisäksi myös integraatiotesteille luotiin oma pohjaluokka, jonka perusteella testiympäristö ja testisovellus ajetaan ylös. Pohjaluokista löytyy testisovelluksen tehdasmetodit, joilla sovellukselle saadaan asetettua haluttu konfiguraatio, kuten tarvittavat mock-objektit, sekä ympäristömuuttujat. Tehdasmetodien lisäksi loin testikäyttöön tarkoitetun HTTP-clientin konfiguroinnin, sekä kustomoidun auktorisoinnin käsittelyn, jotta integraatiotestien lähettämät HTTP-kyselyt menisivät läpi. Testeihin pystyy tällöin konfiguroimaan halutun käyttäjän, jonka tiedot asetetaan testikyselyn mukaan.

4.2.2 Testisovelluksen konfigurointi

Testien konfigurointiin käytetään kuvassa 11 näkyvää kustomoitua applikaatiotehdasta (eng. WebApplicationFactory), jolloin testeille saadaan luotua oma ajoympäristö. Applikaatiotehtaan tehtaan tehtävänä on .NET-ympäristössä hoitaa sovelluksen käynnistäminen testausta varten. Se luo web-palvelimen, joka käynnistää sovelluksen testien ajoa varten halutulla konfiguraatiolla, ja mahdollistaa HTTP-pyyntöjen lähettämisen sovellukselle ja vastausten vastaanottamisen. Testisovelluksen konfigurointi tapahtuu applikaatiotehtaan puolella yli kirjoittamalla .NET-sovelluksen käynnistysprosessissa kutsuttavia metodeja, kuten CreateHostBuilder- ja ConfigureWebHost-metodit.

Kuva 11. Integraatiotestin konfigurointi.



```
0 references | Joona Kleemola, 43 days ago | 1 author, 2 changes | 0 exceptions, - live
protected override void ConfigureWebHost(IWebHostBuilder builder)
{
    builder.UseEnvironment("IntegrationTest");

    builder.ConfigureTestServices
    (
        services :IServiceCollection =>
        {
            services.RemoveAll(typeof(IHostedService));
            RegisterMockedServices(services);
            services.AddAuthentication( defaultScheme: "test-bearer")
                .AddScheme<IntegrationTestAuthSchemeOptions, IntegrationTestAuthHandler>("test-bearer", configureOptions: null);
            services.AddControllers( configure: options =>
            {
                options.Filters.Add( item: new AllowAnonymousFilter());
            });
        }
    );
}

0 references | Joona Kleemola, 43 days ago | 1 author, 3 changes | 0 exceptions, - live
protected override IHostBuilder CreateHostBuilder()
{
    var projectDir :string = Directory.GetCurrentDirectory();
    var configPath :string = Path.Combine(projectDir, "appsettings.json");

    AddEnvironmentVariables();

    var builder = Host.CreateDefaultBuilder()

        .ConfigureAppConfiguration((context, conf) =>
        {
            conf.AddJsonFile(configPath);
            conf.AddInMemoryCollection(_testAppSettings);
            conf.AddEnvironmentVariables();
        })
        .ConfigureServices(s :IServiceCollection =>
        {
            s.AddMvc().AddApplicationPart(typeof(Startup).Assembly);
            s.AddSingleton<RiihiDMADbInitializer>();
        }
        ).ConfigureWebHostDefaults(x :IWebHostBuilder =>
        {
            x.UseStartup<TEntryPoint>().UseTestServer();
        });

    return builder;
}
```

Tällöin normaalin ympäristön käyttämiä konfigurointeja voidaan yliajaa testeissä käytettävillä asetuksilla. Kuvassa 11 yli kirjoitetaan ConfigureWebHost-metodi, ja korvataan oletustoteutus testispesifillä toteutuksella, joka sisältää mock-objektien asettamisen käyttöön, sekä testiautentikoinnin käyttöönoton. Tämän lisäksi ylikirjoitetaan CreateHostBuilder-metodi, jolla saadaan käyttöön testiin halutut applikaatioasetukset (eng. application settings), sekä ympäristömuuttujat (eng. environment variables).

Integraatiotesteissä testaaminen tapahtuu lähettämällä http-kutsuja rajapintaan, ja näiden lähettämiseen tarvitaan oma http-client. Http-client on .NET luokka, joka mahdollistaa http-pyyntöjen lähettämisen ja vastausten vastaanottamisen. Kuvassa 12 näkyy http-clientin konfigurointiin luotu metodi ConfigureHttpClient, jolla testien käyttämään http-clientiin pystyy testikohtaisesti määrittelemään haluttuja http-header tunnisteita, ja tätä kautta voidaan asettaa haluttu käyttäjä, jonka oikeudet voidaan konfiguroida kustomoidussa autentikointikäsittelijässä. Tällöin saadaan helposti hallintaan testeissä käytettävät käyttäjät

ja niiden hallinta. ConfigureHttpClient-metodi mahdollistaa myös käyttäjätietojen jättämisen tyhjäksi. Tällöin voidaan testata, että sovellus ei päästä kutsuja läpi, joista ei löydy autentikointitunnistetta.

Kuva 12. Http-clientin konfigurointi.

```
/// <summary>
/// Inheritor can override this method to configure HTTP client.
/// </summary>
/// <param name="client">HTTP client used in tests</param>
/// <param name="userName">Username for the test user. If left empty, no user will be added to authentication header</param>
/// <returns>Updated HTTP client</returns>
1 reference | Joona Kleemola, 25 days ago | 1 author, 2 changes | 0 exceptions, - live
protected virtual HttpClient ConfigureHttpClient(HttpClient client, string userName)
{
    // Additional configuration for HttpClient (e.g. default headers) can be done here.

    if (!string.IsNullOrEmpty(userName))
    {
        client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue(scheme: "test-bearer", parameter: userName);
        client.DefaultRequestHeaders.Add(name: "integration-test", value: "true");
    }

    return client;
}
```

Autentikoinnin käsittelijä lukee tunnisteesta käyttäjän nimen, ja asettaa sille konfiguroidut API-luvat (claims), joiden perusteella käyttäjällä on oikeus tiettyihin resursseihin rajapinnassa. Kuvassa 13 käyttäjälle ”TestAdmin” annetaan oikeus kaikkiin resursseihin. Tätä käyttäjää voi käyttää yleisenä testikäyttäjänä, jos ajettavan testin päällimmäisenä tarkoituksena on testata kyselyn sisällä tehtävää logiikkaa.

Kuva 13. Testikäyttäjän oikeuksien määrittely.

```
1 reference | Joona Kleemola, 25 days ago | 1 author, 3 changes | 0 exceptions, - live
private static IEnumerable<Claim> GetClaimsForUser(string? user)
{
    switch (user?.ToLowerInvariant())
    {
        // Add claims for the admin users
        case "testadmin":
            {
                yield return new Claim(type: ClaimTypes.Name, value: "TestAdmin", valueType: ClaimValueTypes.String);
                yield return new Claim(type: ClaimTypes.Role, value: "Access.All", valueType: ClaimValueTypes.String);
                yield return new Claim(type: "http://schemas.microsoft.com/identity/claims/objectidentifier", value: "AzureFooBar", valueType: ClaimValueTypes.String);
                break;
            }
```

Yksikkötestien tapaan myös integraatiotesteille luodaan oma tietokanta fikstuuri, jolloin testien ajon yhteydessä saadaan ajettua uusi instanssi tietokannasta ajon ajaksi pystyyn, ja ajon jälkeen se voidaan poistaa huoletta. Tällöin testit päästään ajamaan aina samanlaisesta alkuasetelmasta, eikä olla riippuvaisia jatkuvasti pystyssä olevasta tietokannasta. Testikanta ajetaan yksikkötestien tavoin Microsoftin LocalDB tietokantaan, jolloin testikanta vastaa mahdollisimman hyvin tuotantokäytössä olevaa tietokantaa. Testikantaan saadaan helposti ajettua sama tilanne luomalla Entity Frameworkin avulla tietokannan kontekstiin koodilla määriteltyä testidataa. Entity Framework on Microsoftin kehittämä työkalu, jolla

mahdollistetaan tietokannan parissa työskentely objektiorientoituneella tavalla suoraan sovelluksen lähdekoodin kautta ilman tarvetta esimerkiksi SQL-lauseiden luonnille. Testidataa pystyy tällöin myös helposti ylläpitämään kehittäjän toimesta, kun ei ole tarvetta ylläpitää erillisiä testausta varten luotuja SQL-tiedostoja, joiden perusteella kantaan ajettaisiin dataa. Kuvassa 14 on esitetty testidatan ajamista tietokannan kontekstiin käyttäen Entity Frameworkia.

Kuva 14. Testidatan määrittely käyttäen Entity Frameworkia.

```
    #region Packages

    public static readonly SmartPackageDao[] Packages =
    {
        new()
        {
            PackageId = "00000000-0000-0000-0000-000000000001",
            DisplayName = "SmartPackageOne",
            IsPublic = true,
            ShowNotifications = false
        }
    };

    #endregion

    #region Public methods

    /// <summary>
    /// Adds test data to database context.
    /// </summary>
    /// <remarks>
    /// Call to dbContext.SaveChanges is expected to be made by the caller.
    /// </remarks>
    /// <param name="dbContext">Target database context</param>
    1 reference | Joona Kleemola, 25 days ago | 1 author, 2 changes | 0 exceptions. - live
    public static void CreateData(RiihiDMADbContext dbContext)
    {
        dbContext.Users.AddRange(entities: Users);
        dbContext.Tenants.AddRange(entities: Tenants);
        dbContext.Partners.AddRange(entities: Partners);
        dbContext.Orders.AddRange(entities: Orders);
        dbContext.SmartPackages.AddRange(entities: Packages);
    }

    #endregion
}
```

4.2.3 Testien kirjoittaminen

Pohjatöiden jälkeen siirryttiin suunnitelman mukaisesti integraatiotestien toteutukseen. Käytännössä integraatiotestien kirjoittaminen ei eroa suuresti yksikkötestien kirjoittamisesta, sillä käytössä on sama testauskirjasto, eli XUnit. Tämän takia testien syntaksi ja käytettävissä olevat attribuutit pysyvät samoina, vain testattava asia ja laajuus muuttuvat. Integraatiotesteissä rajapintaan lähetetään HTTP-kutsu, joka simuloi tilannetta, että sovelluksen käyttöliittymästä tulisi HTTP-kutsu, joka käsiteltäisiin. Kuvassa 15 näkyvässä esimerkissä rajapinnan IntuneOrders-resurssiin lähetetään POST-kutsu PostAsync-metodilla, ja kutsun perusteella kantaan luodaan uusi tilaus. Kutsussa käytettävään HttpClientiin on

määritelty pohjaluokan CreateTestClient-metodin avulla käyttöön TestAdmin-käyttäjä. Kun kutsu on suoritettu, varmistetaan että palautuneen vasteen status on onnistunut. Tämän lisäksi integraatiotestien hengessä varmistetaan myös, että kutsuun liittyvät riippuvuudet ovat toimineet oikein. Kuvassa 15 olevassa integraatiotestissä varmistetaan lopuksi, että luotu tilaus löytyy myös tietokannasta. Tämä varmistetaan tekemällä SQL-kysely integraatiotestikantaan käyttämällä hyväksi Entity Frameworkia. Esimerkissä createdOrder-muuttuja pitää sisällään tietokannasta löytyneen tilauksen. Tietokannasta löytyneestä tilauksesta voidaan tarkistaa eri elementtejä riippuen siitä, kuinka tarkasti tiedot halutaan testata. Jos halutaan testata laajempaa vastetta, niin tällöin on hyvä ottaa käyttöön JSON-vertailu sen sijaan, että kirjoitettaisiin jokaisesta vasteessa tulevan olion ominaisuudesta (eng. property) oma rivinsä. JSON-vertailua voidaan toteuttaa luomalla JSON-tiedosto, johon lisätään odotettu vaste. Testissä oleva DeserializeJsonResultAsync-metodi jätettäisiin tällöin pois, ja response-muuttujaa verrattaisiin luotua JSON-tiedostoa vasten käyttämällä esimerkiksi aikaisemmin projektissa apuluokkaan tehtyä VerifyJsonDocumentWithRegexes-metodia.

Kuva 15. Uuden tilauksen luonnin testaus.

```
[Fact]
0 references | Joona Kleemola, 25 days ago | 1 author, 1 change | 0 exceptions, - live
public async Task Post_WhenSuccessful_ReturnsCreatedOrder()
{
    var configuration = new PackageConfigurationDto()
    {
        SmartPackageId = "00000000-0000-0000-0000-000000000001",
        NameOverride = "Override"
    };

    var order = new IntuneOrderDto
    {
        TenantName = "TenantFooBar",
        PartnerId = "Partner123",
        Configurations = new List<PackageConfigurationDto> { configuration }
    };

    var client = CreateTestClient(userName: "TestAdmin");

    var response = await client.PostAsync(requestUri: "odata/IntuneOrders", JsonContent(order));
    response.EnsureSuccessStatusCode();

    var actual = await DeserializeJsonResultAsync<IntuneOrderOData>(response);

    await using var dbConnection = new SqlConnection(TestFixture.TestDatabaseConnectionString);

    var createdOrder = TestFixture.RiihiDmaDbContext.Orders // DbSet<OrderDao>
        .Include(navigationPropertyPath: c :OrderDao => c.Configurations) // IIncludableQueryable<OrderDao,ICollection<…>>
        .FirstOrDefault(o :OrderDao => o.OrderId.Equals(actual.OrderId));

    createdOrder.Should().NotBeNull();
    createdOrder.PartnerId.Should().Be(order.PartnerId);
    createdOrder.TenantName.Should().Be(order.TenantName);
    createdOrder.Configurations.Count.Should().Be(1);
}
```

Testejä voidaan myös tehdä tietokantaan syötettyä testidataa vasten. Kuvassa 16 näkyvässä esimerkissä tehdään yksinkertainen testi, jolla varmistetaan, että testidatassa määritelty tilaus löytyy tietokannasta, ja kutsu palauttaa tämän tilauksen tiedot.

Kuva 16. Olemassa olevan tilauksen haun testaaminen.

```
[Fact]
0 references | Joona Kleemola, 25 days ago | 1 author, 1 change | 0 exceptions, - live
public async Task GetWithId_WhenSuccessful_ReturnsOrder()
{
    var orderId = "12345";

    var client = CreateTestClient(userName: "TestAdmin");

    var response = await client.GetAsync(requestUri: $"odata/IntuneOrders('{orderId}')?");

    VerifyStatusCodeIsOK(response);

    var actual = (await DeserializeJsonResultAsync<IntuneOrderOData>(response));

    actual.OrderId.Should().Be(orderId);
}
```

4.3 CI/CD integrointi

Kun testien pohjat, sekä itse yksikkö- ja integraatiotestejä oli valmiina, pystyttiin näiden ajo kytkemään kiinni sovelluksen Azure-julkaisuputkeen. Azure Build Pipelines -palvelun avulla voidaan lisätä erilaisia tehtäviä, joita Azure build agentit suorittavat. Build agent tarkoittaa ohjelmistoa, joka ajaa erilaisia ohjelmiston rakentamiseen ja kääntämiseen tähtääviä tehtäviä palvelimella (Sharp, n.d.). Tämän avulla voidaan asettaa testiprojektit rakennettavaksi (eng. build), ja tämän jälkeen rakennettujen testiprojektin alla olevat testimetodit voidaan ajaa esimerkiksi Azure Pipelines MSTest -tehtävällä, tai käyttämällä .NET Core ”test”-komentoa.

CI/CD integrointia varten DevOpsiin luotiin oma julkaisuputki, joka tekee vain sovelluksesta julkaistavan paketin, mutta ei lähde viemään pakettia ajoympäristöön. Tällöin CI/CD muutoksia pystyttiin testaamaan ilman, että siitä olisi häiriötä sovelluksen kehittäjille tai demoympäristölle. Julkaisuputken pohjaksi otettiin olemassa olevan demo-julkaisuputken tehtävät ja tähän lähdettiin liittämään luotuja testiprojekteja. Testit tarvitsivat julkaisuputkessa käyttöönsä LocalDB-instanssin. LocalDB:n saa otettua käyttöön lisäämällä putkeen PowerShell-skriptin, jossa ajetaan komento ”sqllocaldb start mssqllocaldb”. Tällä

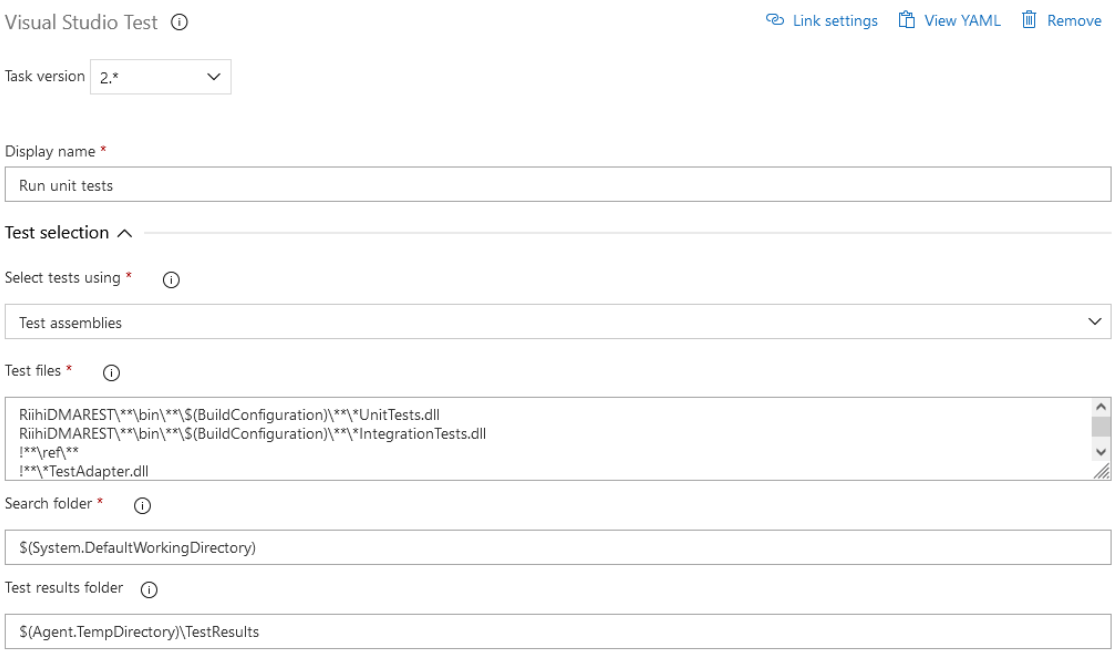
komennolla build-agentti käynnistää ajoympäristöönsä uuden LocalDB-instanssin, johon testiprojektit saavat luotua testeihin käytettävät tietokannat.

Tämän jälkeen testiprojektit piti vielä rakentaa, ja ajaa. Testiprojektit saadaan kätevästi valittua rakennettavaksi ja ajettavaksi, kun projektit on nimetty yhtenäisesti ja asetettu omaan testikansioonsa. Valintaan voidaan käyttää wildcard-syntaksia, jolloin testiprojektit voidaan asettaa rakennettavaksi kuvassa 17. näkyvällä haulla. ”Project”-kenttä tukee wildcard-syntaksia. Asteriskilla (*) ilmaistaan mitä tahansa merkkijonoa, joka voi olla minkä tahansa pituinen.

Kuva 17. Rakennettavien projektien haku.

Testien rakentamisen jälkeen testit vielä ajetaan käyttämällä Kuvassa 18. näkyvää ”Visual Studio Test”-tehtävää. Ajoon valittavat testiprojektit valitaan niin ikään käyttämällä wildcard-syntaksia. Tällöin projektiin luotavat uudet testiprojektit tulevat automaattisesti ajoon ilman muutoksia, jos ne nimetään säännönmukaisesti.

Kuva 18. Ajettavien testien määritys julkaisuputkessa.



5 Projektin tulokset

Projektin tuloksena syntyi yksikkö- ja integraatiotesteille tarkoitetut apu- sekä pohjaluokat, ja näitä käyttämällä luotuja automaatiotestejä. Testit kiinnitettiin sovelluksen Azure-julkaisuputkeen, jolloin testit ajetaan aina ennen, kun sovelluksesta lähdetään julkaisemaan uutta versiota. Testiprojektien rakennetta, sekä testien luontia käytiin myös läpi sovelluksen johtavan kehittäjän kanssa. Jatkossa sovelluksen kehitykseen otetaan mukaan testien luonti, jolloin sovelluksen testikattavuutta saadaan kasvatettua. Lopputuloksesta voidaan katsoa, että tavoitteena ollut automaatiotestauksen raamien ja esimerkkien luonti oli saavutettu.

5.1 Haasteet

Suurin haaste projektin kanssa oli uuden ja suhteellisen laajan kokonaisuuden hahmottaminen, jotta apu- ja pohjaluokat palvelisivat mahdollisimman hyvin jatkossa tehtävää testikattavuuden lisäämistä. Tätä saatiin hyvin ratkottua, kun sovelluksen kehittäjien kanssa katsottiin tietty osio, jota vasten ensimmäisiä testejä on hyvä luoda. On myös hyvä tiedostaa, että projektin aikana luotuja pohjia voidaan laajentaa ja jatkokehittää tarpeiden mukaan. Integraatiotestien pohjaluokkien luonti aiheutti ajoittain haasteita, kun sovellus itsessään ei ollut täysin tuttu, ja integraatiotestejä varten tarvitsee tarkkaan tietää

miten esimerkiksi sovelluksen autentikointi ja auktorisointi saadaan toimivaksi integraatiotestiympäristössä. Tässä oma useamman vuoden sovelluskehityskokemus auttoi kohtuullisen paljon, jolloin työmäärä ei paisunut älyttömän suureksi.

5.2 Jatkokehitys

Projektin jälkeen voidaan lähteä laajentamaan sovelluksen testikattavuutta, kun pohjat on luotu valmiiksi. Testikattavuuden nostattamista on hyvä tehdä jokaisen kehitys tiketin yhteydessä, jolloin ajan kanssa testit alkavat kattamaan hyvän osan sovelluksen logiikasta. En itse lähtenyt suurta testikattavuutta tavoittelemaan projektin aikana, sillä sovellusta aktiivisesti kehittävillä on parempi tieto, miten saadaan simuloitua todelliset testitapaukset, ja mitä osia sovelluksesta on tärkeä asettaa testikattavuuden piiriin. Jatkokehityksen yhteydessä voidaan myös ottaa käyttöön kattavuuden seurantaan tarkoitettuja työkaluja, jolloin saadaan helposti havaittua osioita, joita ei vielä testata. Testikattavuuden ja testien määrän kasvaessa testien ajonopeuden optimointi on kanssa yksi mahdollinen jatkokehityksen kohde, jos koetaan että testien ajo alkaa viemään liikaa aikaa.

5.3 Henkilökohtaiset tavoitteet ja loppupohdintaa

Omana tavoitteenani oli kasvattaa .NET-automaatiotestauksen teoriatietämystä, ja päästä luomaan testien pohjia. Itsellä on jo useampi vuosi työelämänkokemusta, ja osassa projekteista testaus on ollut osa sovelluskehitys-sykliä, mutta itse sovellusten testauspohjien luonti oli itselle uutta, sillä nämä ovat olleet projekteissa tehtynä ennen, kun liityin mukaan.

Suurin osa kehittämistäni sovelluksista on myös kehitetty vanhempien .NET-versioiden päälle, ja niissä ei esimerkiksi ole ollut käytössä Entity Framework -tietokantakehystä, joten tämä projekti tarjosi myös paljon uutta, vaikka kokemusta aiheesta olikin jo kohtuullisesti. Koin myös mielenkiintoiseksi uuden kokonaisuuden haltuunoton, ja sen että pääsi luomaan suuntaviivoja sille, miten jatkossa testejä ja testikattavuutta tullaan tekemään. Tulen myös mielelläni tulevaisuudessa auttamaan ja konsultoimaan tiimiä, jos testaukseen liittyen tulee ongelmia tai kysymyksiä.

Omasta mielestäni suoriuduin työstä hyvin. Suunnittelin työlle työmääräarvion ennen työn toteutuksen aloittamista, ja pystyin sen alittamaan. Tavoitteeksi asetetut apu- ja pohjaluokat sain luotua, ja näiden lisäksi testien integrointi CI/CD-julkaisuputkeen ja esimerkkitestit tuli tehtyä. Työ tuli tehtyä varsinaisen päivätyön lisänä, joten ennen opinnäytetyön aloitusta itseäni mietitytti paljon oma jaksaminen ja aikatauluhaasteet, mutta myös näistä suoriuduin kohtuullisen hyvin. Tässä auttoi suuresti se, että projektille ei ollut tilaajan puolesta tiukkaa aikarajaa, tai ei tullut suurta painetta saada jotain tiettyä ominaisuutta tuotantoon, vaan pystyin etenemään sitä mukaa, kun oma aikataulu antoi periksi. Tämä vähensi huomattavasti työstä tulevaa stressiä, ja helpotti projektin edistämistä.

Lähteet

BrowserStack. (2022). *Testikirjastojen vertailu* [kuva]. https://www.browserstack.com/guide/c-sharp-testing-frameworks

Compete 366. (n.d.). *Azure DevOps – an overview*. https://www.compete366.com/blog-posts/azure-devops-an-overview/

Devolution. (25.01.2021). *Yksikkötestaus ja sen tarkoitus*. https://devolution.fi/yksikkotestauksen-tarkoitus/

Guru99. (2023). *Ohjelmistotestauksen tasot* [kuva] https://www.guru99.com/unit-testing-guide.html

Hamilton, T. (4.3.2023). *Integration Testing: What is, Types with example.* Guru99. https://www.guru99.com/integration-testing.html

JavaTPoint. (n.d.) *Integraatiotestaus havainnollistettuna* [kuva]. https://www.javatpoint.com/integration-testing

Microsoft. (n.d. -a). *Azure DevOps.* https://azure.microsoft.com/en-us/products/devops

Microsoft. (25.4.2023-a). *Common Language Runtime (CLR) overview*. Haettu 19.3.2023 osoitteesta https://learn.microsoft.com/en-us/dotnet/standard/clr

Microsoft. (17.3.2023-b). *Integration tests in ASP.NET Core*. Haettu 21.3.2023 osoitteesta https://learn.microsoft.com/en-us/aspnet/core/test/integration-tests?view=aspnetcore-7.0

Microsoft. (n.d -b). *.NET | Build. Test. Deploy*. https://dotnet.microsoft.com/en-us/

Microsoft. (n.d -c). *.NET Programming Languages*. https://dotnet.microsoft.com/en-us/languages

Microsoft. (12.4.2023-c). *What is Azure Pipelines?.* Haettu 19.3.2023 osoitteesta https://learn.microsoft.com/en-us/azure/devops/pipelines/get-started/what-is-azure-pipelines?view=azure-devops

Microsoft. (15.3.2023-d). *What is .NET? Introduction and overview.* Haettu 19.3.2023 osoitteesta https://learn.microsoft.com/en-us/dotnet/core/introduction

.NET Foundation. (n.d.). *About xUnit.net.* Haettu 19.3.2023 osoitteesta https://xunit.net/

Red Hat. (11.5.2022). *What is CI/CD?.* https://www.redhat.com/en/topics/devops/what-is-ci-cd

Sharp, K. (n.d.). *Azure Build Agents.* Cloud Academy. https://cloudacademy.com/course/implementing-and-managing-azure-build-infrastructure/azure-build-agents/

Software Testing Help. (28.4.2023). *12 Best Automated Unit Testing Tools.* Haettu 21.3.2023 osoitteesta https://www.softwaretestinghelp.com/best-automated-unit-testing-tools/

Vinugayathri. (n.d.) *Why Should You Use xUnit? A Unit Testing Framework For .Net.* Clarion technologies. https://www.clariontech.com/blog/why-should-you-use-xunit-a-unit-testing-framework-for-.net