Rami Jarakivi

FARVIEW – JÄRJESTELMÄ AUTOMAATIOLOGIIKAN TIETOJEN TALLENNUKSEEN JA ETÄLUKUUN

Tietojenkäsittelyn koulutusohjelma

2017

samk
Satakunnan ammattikorkeakoulu
Satakunta University of Applied Sciences

FARVIEW - JÄRJESTELMÄ AUTOMAATIOLOGIIKAN TIETOJEN TALLENNUKSEEN JA ETÄLUKUUN

Jarakivi, Rami
Satakunnan ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma
Elokuu 2017
Ohjaaja: Grönholm, Jukka
Sivumäärä: 40
Liitteitä:

Asiasanat: automaatiologiikka, etävalvonta, .NET Core, parseri, testaus

---

Opinnäytetyöni tarkoitus oli luoda Raumaster Paper Oy:lle ohjelma joka auttaa heitä paperitehtaan toiminnan seuraamisessa ja mahdollisten virhetilanteiden diagnosoinnissa. Toteuttamani ohjelma mahdollistaa halutun datan lukemisen PLC-logiikkayksikön muistista haluttuna ajanhetkenä ja sen jälkeen luetun datan tallentamisen tietokantaan. Tallennetut muistiluvut voidaan myöhemmin lukea ohjelman UI-osan avulla. Esittelen tässä työssä käyttämäni teknologiat, ohjelman taustalla toimivan tietokantamallin sekä tietojen tallennukseen liittyvän parserin. Lisäksi kerron ohjelman testauksesta ja sen asennuksesta Raumaster Paperille ja yhteistyöstä heidän kanssaan projektin aikana.

FARVIEW – A SYSTEM FOR STORING AND REMOTE VIEWING DATA FROM A PROGRAMMABLE LOGIC CONTROLLER

Jarakivi, Rami
Satakunnan ammattikorkeakoulu, Satakunta University of Applied Sciences
Degree Programme in Business Information Systems
August 2017
Supervisor: Grönholm, Jukka
Number of pages: 40
Appendices:

Keywords: automation logic, remote monitoring, .NET Core, parser, testing

---

The purpose of this thesis was to create a program for Raumaster Paper Oy that will help them monitor the operations of a paper mill and to diagnose possible error states. The program I have written will allow them to read desired data from a PLC-logic unit at the desired time and then save that data to a database. The saved data can be later accessed through the UI component of the program. In this thesis, I present the technologies I used, the database model and the parser that is tied to saving the data. Furthermore, I report on the testing of the program, the process used to install it and the co-operation with Raumaster Paper.

SISÄLLYS

1 JOHDANTO

Opinnäytetyöni tarkoituksena on suunnitella ja toteuttaa järjestelmä paperitehtaan automaatiologiikkayksikön tietojen tallentamiseen ja esittämiseen. Järjestelmän on tarkoitus mahdollistaa tallennettavien tietojen määrittäminen dynaamisesti. Tietojen tallentamisen jälkeen järjestelmän tulee mahdollistaa tietojen tarkastelu etäyhteyttä käyttäen.

Käyn tässä työssä läpi ensin projektin lähtökohdat kuten projektin tilaaja, projektin vaatimusmäärittelyt ja projektissa käytettyjen automaatiologiikkayksikköjen perusteita. Seuraavaksi esittelen työssä käytettyjä termejä ja kerron, miten toteutettu ohjelma ratkaisee asiakkaan ongelman teoriassa ja käytännössä. Sitten esittelen ohjelman testauksen ja asennuksen asiakkaalle. Työn päättää muutama sana opinnäytetyön teon vaiheista.

2 OPINNÄYTETYÖN LÄHTÖKOHDAT

2.1 Raumaster Paper

Opinnäytetyön tilaaja Raumaster Paper on osa Raumaster Groupia. Raumaster Group on maailman johtava yritys materiaalikäsittelylaitteistoissa. He työllistävät noin 300 ammattilaista ja yrityksen liikevaihto on noin 100 miljoonaa euroa. Raumaster Paper on erikoistunut paperirullien käsittelyyn. He toimittavat rullankäsittelyratkaisuja ympäri maailmaa sekä uusiin tehtaisiin, että vanhojen tehtaiden modernisointiin.

2.2 Lähtökohtien esittely

Raumaster Paper on asettanut ratkaistavaksi ongelman, joka liittyy heidän keskeiseen osaamisalueeseensa, tehdasautomaatioon. He kytkevät ja ohjelmoivat tehtaan toimintaa ohjaavia logiikkayksiköitä asiakkaan laitteistoon. Joskus tehtaassa kuitenkin tulee

toimintahäiriöitä ja näitä tilanteita on vaikea diagnosoida ilman historiatietoja siitä mitä tehtaassa on ennen häiriötä tapahtunut.

Raumaster Paper on tilannut järjestelmän, joka tullaan sijoittamaan PC-tietokoneeseen, logiikkayksikön viereen, ja joka tallentaa heidän haluamiaan tietoja logiikkayksiköstä ja mahdollistaa niiden tarkastelun myöhemmin. Tämä mahdollistaa sekä häiriötilanteiden diagnosoinnin, että tehtaan yleisen toiminnan sujuvuuden tarkastelun ilman fyysistä paikallaoloa.

2.3 Aiemmat ongelmaa käsittelevät ratkaisut

Opinnäytetyö on asetettu tehtäväksi sen sijaan, että Raumaster Paper olisi ottanut käyttöön valmiin paketin joltain toimittajalta, koska he ovat havainneet, että valmiit ohjelmat tulevat hyvin isoina paketteina. Jotta pieni logiikkahistoriatallennus saataisiin käyttöön, pitäisi koko iso ohjelmistopaketti ottaa käyttöön. Samoin valmiit ohjelmat ovat raskaampia ja vähemmän joustavia kuin täsmäratkaisuna, alusta asti kehityksessä mukana ollen kehitetty ohjelma.

2.4 Automaatiologiikka

Kuva 1: Siemens SIMATIC S7-300 logiikkakontrolleri (Siemens 2017)

Raumaster Paper on tilannut järjestelmän, jonka on tarkoitus toimia Siemens Simatic S7 automaatiologiikkayksiköiden kanssa. Yllä olevassa kuvassa 1 näkyy kyseisiä logiikkayksikköjä.

S7-300 on modulaarinen PLC järjestelmä, joka on tarkoitettu pieneen ja keskisuureen tehontarpeeseen. Laaja moduulivalikoima tarjoaa optimaalisen soveltuvuuden automaatiotehtävään. Monipuolinen verkottuminen ja hajautettu rakenne mahdollistavat joustavan käytön. S7 yksiköitä käytetään mm. erityisteollisuudessa, tekstiiliteollisuudessa, pakkausteollisuudessa, yleiskoneiden valmistusteollisuudessa, ohjausrakennuksissa, koneistusteollisuudessa ja elektroniikkateollisuudessa. (Siemens AG 2017)

S7 järjestelmä on suunniteltu modulaariseksi. Järjestelmä sisältää seuraavat osat:

- Prosessori joka voidaan valita sopivaksi tehotarpeiden mukaisesti.
- Signaaliyksikkö joka hoitaa digitaalisen tai analogisen I/O:n.
- Kommunikaatioprosessori joka hoitaa väyläliitynnät ja point-to-point yhteydet.
- Funktiomoduuli joka hoitaa nopean laskennan, paikallistamisen ja PID ohjauksen.

(Siemens 2017)

S7 prosessoriyksikköjä ohjelmoidaan useimmiten STEP 7 ohjelmalla. Ne toimivat 24V tasavirralla, yleensä 0 - 60 Celsius - asteen alueella (osa jopa -25 °C), sisältävät monia tyypillisiä tietokoneisiin liittyviä asioita kuten laskureita, työmuistia, liityntöjä ja salausmahdollisuuksia. Fyysiseltä kooltaan prosessorimoduulit ovat luokkaa 40mm X 125mm X 130mm ja painoltaan noin 400 - 1200 grammaa. (Siemens 2017)

3 VAATIMUSMÄÄRITTELY

Tietokoneohjelmaa toteutettaessa on eräs ensimmäisistä ja tärkeimmistä vaiheista vaatimusmäärittely. Se on vaihe jossa ohjelman tekijä ja tilaaja istuvat alas ja keskustelevat siitä mitä ohjelman pitäisi tehdä. Vaatimusmäärittelyvaiheessa etsitään keskustelemalla ja jopa haastattelemalla asiakasorganisaation työntekijöitä niitä yleisiä käyttötapauksia joihin tulevan ohjelman pitäisi vastata. Näiden pohjalta voidaan, hieman käytetystä projektinhallintatavasta riippuen, kirjoittaa eri tasoisia vaatimusmäärittelydokumentteja joihin sisällytetään kuvauksia siitä, millainen tekeillä olevasta ohjelmasta tulee.

Tärkeimmät ja eniten järjestelmän kehitystä ohjaavat vaatimukset, jotka Raumaster Paper on asettanut, ovat:

- tallennettavien tietojen dynaaminen määrittely, mukaan lukien
    - tiedon tallennuksen aikaväli
    - tallennettavat muistipaikat
    - muistipaikat joita seurataan ja joiden ollessa tietyissä arvoissa halutut muistipaikat tallennetaan
- järjestelmän toiminta 24/7
- tallennettujen tietojen lukeminen VPN-yhteyttä käyttäen
- toive Microsoft lähtöisten tekniikoiden käyttämisestä

4 RATKAISUSSA KÄYTETYT TEKNOLOGIAT

Kehittäessäni FarViewtä olen käyttänyt mm. seuraavia teknologioita:

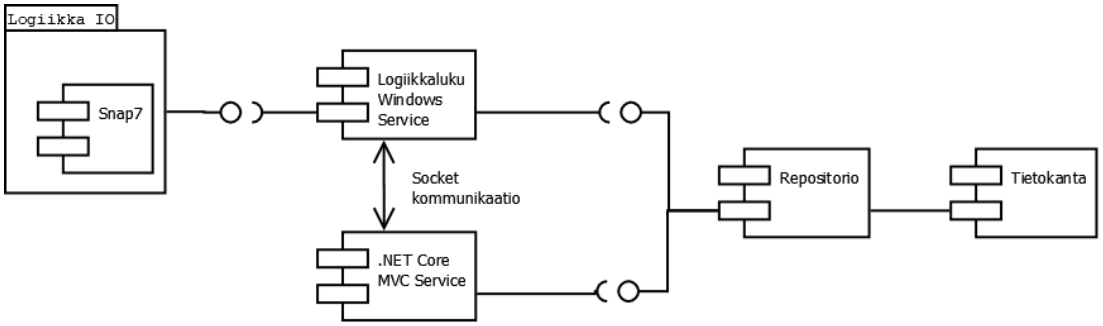
- Windows Service on erityinen ohjelmatyyppi, joka on suunniteltu pitkäkestoisiin operaatioihin ja joka ei näytä graafista käyttöliittymää. Windows Servicet voidaan myös automaattisesti käynnistää tietokoneen käynnistyessä ja ne voidaan pysäyttää ja uudellenkäynnistää. (Microsoft 2017). Näin ollen service sopii hyvin FarView-ohjelmaan ja ohjelma onkin jaettu kahteen erilliseen serviceen.
- Relaatiotietokanta on tietokanta, jonka rakenne perustuu E. F. Coddin 1970-luvulla julkaistuun tutkielmaan ”A Relational Model of Data for Large Shared Data Banks”. Siinä data on järjestetty riveittäin tauluihin niin, että taulun yksi rivi vastaa yhtä taulun tyyppistä tietuetta ja rivin sarakkeet kyseiseen tietueeseen kuuluvia arvoja. Jokaisen taulun jokainen rivi on myös erilainen kuin taulun kaikki muut rivit. Lisäksi eri taulujen rivejä voidaan viitata toisiin niin kutsutuilla avaimilla josta relaatiotietokanta-sanan ”relaatio” osuus tulee. (Powell 2017). FarViewssä relaatiotietokanta on kaiken toiminnan taustalla. Sinne tallennetaan niin ohjelman käyttäjien tunnukset ja oikeudet, muistipaikoista tallennetut tiedot, osa ohjelman käyttämistä konfiguraatiotiedoista kuin myös ohjeet siitä mitä muistipaikkoja tallennetaan ja milloin niitä tallennetaan.
- SQL Server on Microsoft Oy:n kehittämä relaatiotietokantaohjelma jota FarView käyttää tietokantansa tallennukseen.
- Entity Framework on objekti-relaatio-kuvaaja joka auttaa .NET kehittäjiä antamalla heidän käyttää relaatiopohjaista dataa objektien kautta. Se poistaa tarpeen kirjoittaa suurin osa datasaanti koodista joka normaalisti olisi pakko kirjoittaa. (Anderson;Dykstra ja Pasic 2010). Valitsin Entity Frameworkin FarViewn repositorioiden toteutusmalliksi juuri sen vuoksi, että tietokannan ollessa yksinkertainen, pääsin nopeammin kirjoittamaan varsinaista business-logiikkakoodia, kun varsinainen SQL oli jo valmiiksi debuggattu ja pakattu Entity Frameworkiksi. Samoin tietokanta itsessään oli helppo kehitysaikana tuottaa Entity Frameworkin migraatiotyökaluilla entity-luokista.
- .NET Framework on ohjelmistokehys, joka mm. tukee montaa ohjelmointikieltä, asynkronista ja samanaikaista ohjelmointia sekä natiivia yhteensopivuutta. (Carter, ym. 2016)

- C# on yksinkertainen, moderni, oliopohjainen ja vahvasti tyypitetty ohjelmointikieli jonka juuret ovat C ohjelmointikieliperheessä (Wagner ja Wenzel 2016). FarViewn backend on ohjelmoitu C#:lla joka on yksi .NET kielistä ja näin ollen käyttää .NET frameworkia.
- Snap7 on avoimen lähdekoodin, 32/64 bittinen, monialustainen Ethernet kommunikaatiokirjasto jolla voi lukea ja kirjoittaa dataa Siemensin S7 logiikkayksiköiltä (Nardella ei pvm). FarView käyttää Snap7 kirjaston Sharp7 nimistä C#-porttausta logiikkayksikön kanssa keskusteluun.
- Bootstrap on suureen suosioon noussut ohjelmistokehys joka CSS:ää ja Javascriptiä käyttäen auttaa kehittämään responsiivisen mallin mukaisia html-sivuja. Se auttaa ohjelmistokehittäjiä vähentämään koodin kirjoitusta, saavuttamaan yhtenäisen ulkoasun ja varmistaa sivujen toiminnan kaikilla selaimilla. (Rascia 2015). FarViewssä käytän Bootstrappia käyttöliittymän CSS-tyylittelyyn.
- jQuery on nopea, pieni ja ominaisuusrikas JavaScript kirjasto (jQuery Foundation 2017). FarViewssä käytän jQueryä datan hakuun Ajax-kutsuina.
- Knockout.js on Javascript kirjasto joka auttaa kehittämään rikkaita ja responsiivisia käyttöliittymiä. Knockout auttaa aina kun on tarve päivittää käyttöliittymän osia dynaamisesti, esim. ulkoisen datalähteen muuttuessa. (Knockoutjs.com ei pvm). Olen käyttänyt Knockouttia näkymissä sitomaan käyttöliittymän elementtejä alla olevaan dataan.
- vis.js on dynaaminen, selainpohjainen visualisaatiokirjasto. Se on suunniteltu helppokäyttöiseksi, toimimaan suurien, dynaamisten datamäärien kanssa ja mahdollistamaan datan manipulointi. (Almende B.V. 2017). Olen käyttänyt vis.js:stä Timeline osaa joka mahdollistaa datapisteiden piirron dynaamisesti liikuteltavalle aikajanalle.
- Ajax on tekniikka jolla html-sivu voi asynkronisesti ladata Javascriptin avulla palvelimelta uutta sisältöä ja esittää sen sivulla päivittämättä koko sivua uudelleen (Segue Technologies 2013). Käytän Ajaxia FarViewssä mm. hakemaan rullaviestejä tietokannasta viestit näyttävässä näytössä. Ajaxin käyttö mahdollistaa sen, että selainikkuna ei ”jäädy” tiedonhaun ajaksi, joka saattaa isoilla viestimäärillä kestää muutaman sekunnin.

5 ONGELMAN RATKAISUN TEORIA

5.1 Yleistä

FarView on toteutettu kahtena osana, jotta molempia osia voidaan kehittää erillään toisistaan, ja jotta osien koko ja toimintalogiikka pysyvät paremmin hallittavana. Osajako näkyy alla olevassa kuvassa 2.



Kuva 2: Ohjelman jako komponentteihin

Ehkä tärkein osa käsillä olleen ongelman ratkaisua on tapa, jolla logiikan luvun dynaamisuus toteutetaan. Olen tähän tarkoitukseen ottanut käyttöön käsitteen trigger, joka tarkoittaa kokoelmaa niitä tietoja jotka määrittävät kuinka usein logiikasta yritetään lukea, minkä ehtojen pitää täyttyä, jotta luku suoritetaan ja ehtojen täyttyessä, mitä muistipaikkoja logiikasta luetaan ja tallennetaan tietokantaan. Koska kaikki edellä mainitut ominaisuudet voidaan määritellä dynaamisesti, täyttyvät asiakkaan vaatimukset logiikan luvun määritysten osalta.

Esiteltyäni projektin edistymistä Raumaster Paperille, tuli heiltä vielä toive saada myös kirjoitettua logiikalle tietoja. Tarve olisi toiminnallisuudelle, jossa esim. trigger lukee muistia jonkin bitin ollessa päällä ja luennan jälkeen kyseinen bitti nollataan. Toive oli onneksi helppo toteuttaa, sillä Snap7 kirjasto tuki logiikalle kirjoittamista ja pystyin uudelleenkäyttämään sekä osia käyttöliittymästä, että tietokannan toiminnallisuutta. Nämä logiikan kirjoitustiedot on myös tallennettu osaksi triggeriä luvussa 6.6 kerrotulla tavalla.

5.2 Dependency Injection

Eräs .NET Coren ominaisuuksista, jotka helpottavat ohjelmien luontia ja ohjelman eri osien rajaamista omiin laatikkoihinsa, on Dependency Injection. DI auttaa pitämään ohjelman luokat nk. loosely coupled, eli luokat eivät suoraan luo tai käytä toisia luokkia, vaan asiat sidotaan toisiinsa rajapintojen kautta. Tällöin voi rajapinnan toteuttavaa konkreettista implementaatiota muuttaa tai vaihtaa kokonaan toiseen ilman, että rajapintaa käyttävään luokkaan tarvitsee tehdä muutoksia.

Ratkaistakseen ongelman, jossa viite palvelun toteutukseen kovakoodataan ohjelmaan, DI tarjoaa välillisyyden tason siten, että sen sijaan, että ohjelma instansoi palvelun suoraan new-operaattorilla, ohjelma sen sijaan pyytää palvelua palvelukokoelmalta tai ”tehdas” - luokalta. Ja vieläpä, sen sijaan että ohjelma pyytää palvelukokoelmalta tiettyä, spesifistä tyyppiä (näin ollen luoden tiukan sidoksen), ohjelma pyytää rajapintaa (kuten vaikkapa ILoggerFactory) sillä odotuksella, että palvelun tarjoaja (tässä tapauksessa NLog, Log4Net tai Serilog) toteuttaa halutun rajapinnan. (Michaelis 2016)

Dependency Injectioniin liittyy oleisesti myös palveluiden elinikä. .NET Coressa elinikävaihtoehtoja on neljä: Instance, Singleton, Transient ja Scoped. AddInstance<TService> (TService implementationInstance) kutsulla lisätty palvelu varmistaa, että palvelua pyydettäessä ei ainoastaan palauteta TService-tyypin luokka, vaan, että palautettava palveluluokka on aina AddInstance kutsulla rekisteröity implementationInstance ilmentymä. (Michaelis 2016)

Vastapainona AddInstance kutsulle on käytössä AddSingleton<TService>() metodi jolle ei voi antaa instanssia parametrina vaan sen sijaan TService tyypistä pitää löytyä vakiokonstruktori. AddSingletonilla lisätystä palvelusta luodaan ensimmäisellä pyynnöllä instanssi, joka sitten aina palautuu seuraavilla pyyntikerroilla. Palvelukokoelmasta löytyy myös AddTransient() metodi, jolla lisätystä palvelusta palautetaan joka pyyntikerralla uusi instanssi. Viimeisenä on joukko AddScoped() metodeja. Näillä lisätyistä palveluista palautetaan sama instanssi koko kontekstin, eli scopen ajan. .NET Coressa konteksti tarkoittaa käytännössä HttpContext-oliota. (Michaelis 2016)

Tarpeesta riippuen voidaan siis palvelusta luoda aina uusi instanssi, luoda palvelusta jokaista kontekstia kohden uusi instanssi tai palauttaa aina sama instanssi joka luodaan joko ensimmäisellä kutsukerralla automaattisesti (singleton) tai jonka ohjelmoija itse luo eksplisiittisesti new-kutsulla (instance).

5.3 Käyttöliittymä

Ensimmäinen osa ohjelmasta on Windows Servicenä suoritettava UI, joka on toteutettu .NET Core MVC pohjaisesti. Olen valinnut Coren ratkaisuksi monestakin syystä. Suurin syy on, että se on minulle teknologiana kohtuullisen tuttu ja se toteuttaa asiakkaan vaatimuksen toimia selainpohjaisesti VPN yhteydellä. Toisekseen se on sisäisesti toteutettu eräänlaisella opt-in ajatusmallilla, eli ohjelmaan otetaan käyttöön erilaisia moduuleja sen mukaan mitä ohjelma tarvitsee. Tämä johtaa siihen, että valmiista ohjelmasta tulee kevyempi ja mahdollisesti nopeammin suorittuva koska siinä on vain ohjelman tarvitsemat asiat, ei kaikkea mitä jokin ohjelmistokehys sisältää. Kolmas syy on DI:n palvelukokoelma, joka helpottaa eri palveluiden käyttöä ja mm. ohjelman testaamista.

5.4 Logiikan lukemisen service

Toinen ohjelman osa on Windows Service joka hoitaa logiikan luvun käyttämällä Snap7 nimistä kirjastoa. Tämä kirjasto, ja sen C# porttaus jota käytän, Sharp7, mahdollistaa Ethernet kommunikaation Siemensin S7 logiikkamoduulin kanssa. Olen myös eriyttänyt logiikkaluvun rajapinnan taakse, jotta siitä saadaan riippumaton osa ja tulevaisuudessa se voidaan, vaikka vaihtaa kokonaan tai laajentaa käsittelemään myös muita logiikkakirjastoja.

Osat keskustelevat myös keskenään. Tämä on hoidettu käyttämällä socketteja, joiden läpi lähetetään TCP-paketteja sovitun protokollan mukaisesti. Se, että osat pystyvät keskustelemaan toistensa kanssa, mahdollistaa mm. sen, että UI pystyy seuraamaan reaaliaikaisesti logiikkalukuosan toimintaa ja tilaa. Samoin kun käyttäjä tekee käyttöliittymässä muutoksia logiikkaluvun triggereihin, voi käyttöliittymä heti ilmoittaa logiikkaservicelle, että muutoksia on tapahtunut, jolloin logiikkaservice voi käydä ne

tietokannasta lukemassa ja aloittaa uusien tai päivitettyjen triggereiden seurannan. Tämä poistaa mm. sen tarpeen, että logiikkaservice joutuisi koko ajan käymään tietokannasta katsomassa onko sinne tullut kokonaan uusia triggereitä tai päivityksiä vanhoihin. Socket-kommunikaatio mahdollistaa myös sen, että Raumasterin mahdollisesti joskus laajentaessa järjestelmää, voidaan esim. logiikan luku sijoittaa, vaikka vallan eri tietokoneeseen ja kommunikaatio osien välillä saadaan silti helposti toimimaan.

5.5 Tietokanta

Molemmat osat keskustelevat saman tietokannan kanssa. Tämä tapahtuu molemmissa tapauksissa repository arkkitehtuurimallia käyttämällä. Repository malli on otettu ohjelmaan käyttöön koska se mahdollistaa mm. ohjelmakoodin uudelleenkäytön, sillä molemmat osat käyttävät osittain samoja tauluja ja suorittavat samantyylisiä kyselyjä. Repository myös edistää kerrosarkkitehtuuria jossa sovelluksen sovelluslogiikka eristetään data access layerista. Tämä johtaa siihen, että molemmat osat voidaan ohjelmoida ja testata toisistaan erillään ja vaikkapa vaihtaa kokonaan sovelluksen siitä kärsimättä koska layerit keskustelevat toistensa kanssa rajapintojen välityksellä.

6 ONGELMAN RATKAISUN KÄYTÄNTÖ

6.1 Yleistä

Opinnäytetyössä tuotetun ohjelman molemmat osat on toteutettu Windows Serviceinä. Tämä sen takia, koska molempien osien tulee olla koko ajan ajossa ja saatavilla. Molemmat myös suorittavat pitkäaikaista toimintaa, jolloin paras ratkaisu on Windows Service. Kuten Msdn.com toteaa: ”Microsoft Windows services, formerly known as NT services, enable you to create long-running executable applications that run in their own Windows sessions. These services can be automatically started when the computer boots, can be paused and restarted, and do not show any user interface. These features make services ideal for use on a server or whenever you need long-running

functionality that does not interfere with other users who are working on the same computer" (Microsoft 2017).

Toisaalta se, että FarView ei avaa missään vaiheessa perinteistä Windows ohjelmaikkunaa aiheuttaa esim. ohjelman kohtaamien virhetilanteiden käsittelyyn pientä lisähaastetta. Sillä vaikka käyttöliittymään pääseekin verkkoselaimella käsiksi, voi myös itse käyttöliittymäosa kohdata virheitä, joiden vuoksi se ei välttämättä pääse koskaan edes html-sivuja renderöimään.

Tämän vuoksi käytän FarViewn kanssa Windowsin Event Log palveluja, jotka mahdollistavat lokitietojen kirjoituksen käyttöjärjestelmän sisäisellä mekanismilla lokiin johon pääsee käsiksi Windows Event Viewer ohjelmalla. Näin ollen sekä FarViewn käyttöliittymäosan, että logiikkalukuosan kohdatessa virheitä voivat ne kirjoittaa lokitiedostoa jota lukemalla voidaan ongelmat mahdollisesti korjata.

Koska ohjelmia suoritetaan erilaisissa ympäristöissä ja niiden halutaan tekevän hieman eri asioita, pitää niiden asetuksiin päästä vaikuttamaan. Muutetut asetukset sitten tallennetaan, ja seuraavalla kerralla luetaan, jonkinlaisesta konfiguraatiotiedostosta. FarView käyttää konfiguraatiotiedostoja, joissa mm. kerrotaan tietokantayhteys ja käyttöliittymäosalle se, mitä porttia http yhteyttä varten halutaan kuunnella. Näiden lisäksi on molemmille FarViewn osille yhteisiä konfiguraatiotietoja, jotka tallennetaan tietokantaan ja joita voi ohjelman ajon aikana muuttaa ja joiden muutokset vuorostaan vaikuttavat ohjelman toimintaan.

6.2 Dependency Injection

Käytännössä FarViewn tapauksessa, ja yleisesti .NET Coressa, DI näkyy siten, että ohjelmalla on Startup-luokka jonka ConfigureServices-metodin ajoympäristö suorittaa, antaen tälle IServiceCollection-rajapinnan toteuttavan luokan, kuten:

```
public void ConfigureServices(IServiceCollection services) {
```

Tähän luokkaan voi metodissa sitten lisätä palveluita, kuten vaikkapa:

```
services.AddScoped<ITriggerRepository, TriggerRepository>();
```

kutsulla joka lisää palvelukokoelmaan TriggerRepository-olion joka toteuttaa ja jonka voi palvelukokoelmalta myöhemmin pyytää, ITriggerRepository-rajapinnan toteuttavana.

FarViewn tapauksessa olen käyttänyt lähinnä scoped ja singleton eliniän palveluita. Scoped palveluina ovat sellaiset jotka käytetään yhtä http-kutsua varten ja sen lopuksi hylätään, kuten tietokantayhteys ja yhteys lukijan ja käyttöliittymän välillä. Tämä sen takia, että näin varmistutaan mm. siitä, että kyseisiä palveluja tuottavat oliot ja niiden resurssit vapautetaan ja poistetaan säännöllisesti, jolloin ne eivät jää suotta elämään.

Singleton palveluiksi olen lisännyt sellaisia, joita käytetään joko kerran, kuten luokka joka huolehtii, että tietokanta on luotu ja siellä on vähintään vakiokäyttäjä, tai lokikirjoituspalvelu joka voi hyvin olla sama koko ohjelman ajon ajan, sillä se ei tarvitse yhteyksiä eikä loki muutu ohjelman ajon aikana.

6.3 Käyttöliittymä

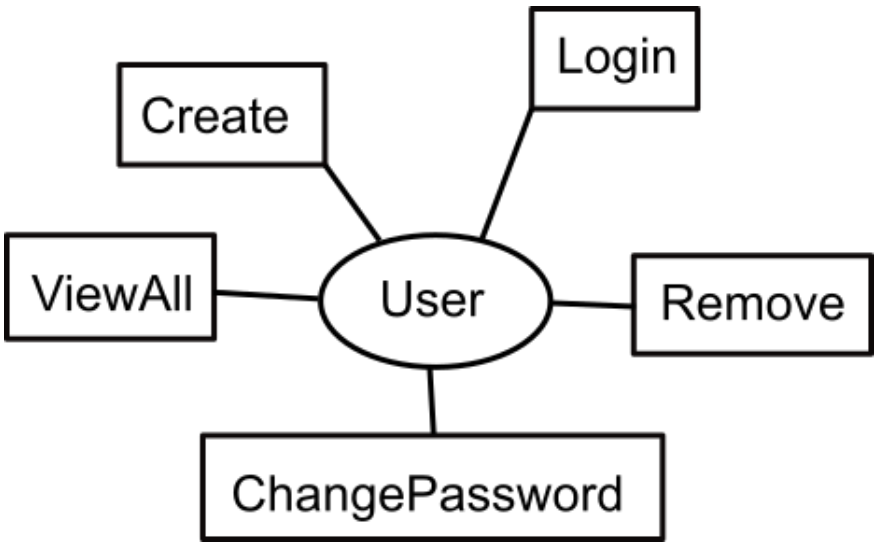
Tuotetun ohjelman UI on toteutettu .NET Core MVC tekniikalla. Lyhenne MVC tarkoittaa tässä kontekstissa sanoja Model, View ja Controller. MVC on arkkitehtuurimalli joka auttaa eriyttämään ohjelman osat toisistaan. Tämä johtaa siihen, että ohjelman eri osia voidaan mahdollisesti käyttää uudelleen toisessa projektissa. Samaten ne on helppo korvata tarpeen mukaan uusilla toteutuksilla.

MVC koostuu mallista (Model), näkymästä (View) ja kontrollerista (Controller). Malli tarkoittaa dataa, näkymä sen esittämistä näytöllä ja kontrolleri syötteen käsittelyä. FarViewn tapauksessa View koostuu html-sivuista ja niiden Javascript-skripteistä. Vieweihin kuuluvat myös .NET Coren Viewmodelit jotka ovat tavallaan kavennettuja Modeleita tapauksissa joissa Viewn ei haluta näkevän kaikkea dataa (esim. kaikkia tietokannan taulun sarakkeita). Model on FarViewssä ne luokat ja palvelut jotka hoitavat tiedon tallennusta ja hakua tietokannasta. Controllerit vuorostaan käsittelevät

käyttäjän syötteen (esim. pyyntö saada html-sivu) ja päättävät mikä View ja millä Modelilla lähetetään siihen vastaukseksi.

Toki, kuten aina tosielämässä, Modelin, Viewn ja Controllerin vastuualueet aavistuksen sekoittuvat FarViewn tapauksessa. Esimerkiksi täysin oikeaoppinen View on täysin ”tyhmä” eikä se sisällä kuin korkeintaan automaattisesti tapahtuvaa Form-datan validointia. FarViewn tapauksessa kuitenkin Viewt tekevät hieman enemmän ja niihin on sijoitettu aavistus bisneslogiikkaa (mm. muistiosoitteiden validointia joka on sidottu yhden valmistajan muistiosoitustyyliin ja rikkoo rajapinnoilla muuten toteutettua eriyttämispolitiikkaa).

FarView ohjelmana koostuu useasta kontrollerista jotka tuottavat käyttäjän ruudulle usean näkymän jotka esittävät usean mallin tietoja. Alla olevassa kuvassa 3 nähdään esimerkki FarViewn User-kontrollerin tuottamista näkymistä.



Kuva 3: User-kontrollerin näkymät

User-kontrolleri vastaa FarViewssä kaikista suoraan käyttäjään liittyvistä toimista. Eri toiminnot on eriytetty eri näkymiin, jotta niitä voidaan helposti muokata erillään toisistaan. Käyttäjää koskevia toimia, ja näin ollen nämä toimet mahdollistavia näkymiä, ovat mm. uuden käyttäjän luonti, sisäänkirjautuminen, käyttäjän poisto, salasanan vaihto ja kaikkien käyttäjien listaaminen.

Kontrollereissa on myös toteutettu pääsykontrolli ohjelman eri osiin. FarView käyttää kahta eri käyttäjäroolia, Admin ja User, joista Admin on rooli jossa oleva käyttäjä voi suorittaa ohjelman kaikkia toimintoja. User taasen on rajattu rooli jossa oleva käyttäjä ei voi esimerkiksi lisätä järjestelmään uusia käyttäjiä tai poistaa triggereitä, käyttäjiä tai konfiguraatioita.

6.3.1 Responsiivisuus

FarViewn käyttöliittymä on pyritty rakentamaan nykyaikaisia suunnitteluperiaatteita käyttäen. Tämä tarkoittaa html-käyttöliittymässä nk. responsiivista suunnittelua.

Responsiivinen suunnittelu tarkoittaa lyhyesti sitä, että riippuen käyttäjän käyttämän päätelaitteen näytön koosta, käyttöliittymää skaalataan sopimaan paremmin näytölle. Eri elementtejä, joiden katsotaan olevan ei-kriittisiä, voidaan myös piilottaa pienemmiltä näytöiltä. Jotkin sivut ovat myös kokeilleet mm. tekstin ja kuvien koon muuttamista päätelaitteen näytöstä riippuen. (Leiniö 2012)

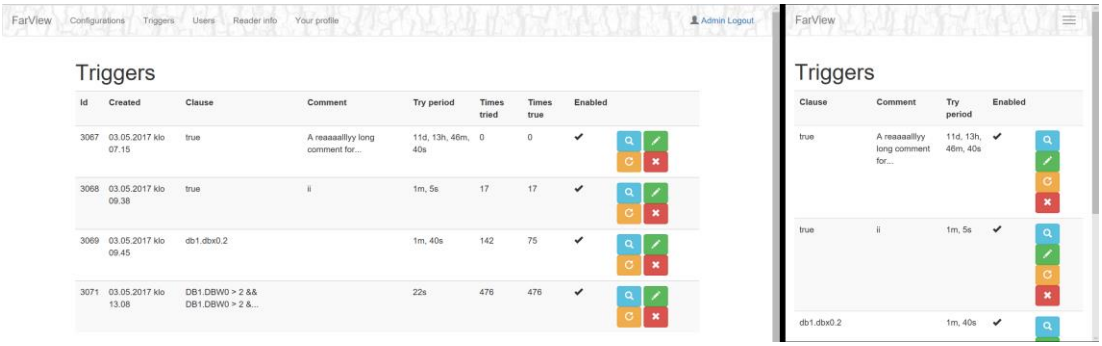
FarViewssä käyttöliittymän responsiivisuus, kuten sen tyylityskin, on toteutettu Bootstrap kirjastoa käyttäen. Tämä sen takia, että Bootstrap tarjoaa monia hyödyllisiä palveluja ja yhtenäisen, nykyaikaisen tyylin. Näin ollen pystyn ohjelman kehittäjänä keskittymään toiminnallisuuteen tyylimääritysten luonnin sijaan.

Responsiivisuus on sisäänrakennettu jokaiseen FarViewn sivuun. Tämä on seuraus FarViewn käyttämästä NET Core MVC-tekniikasta, jossa päätelaitteelle lähetettävät sivut ”renderöidään” Razor nimisellä view enginellä. Tällöin sovellukseen voi määrittää jaettuja html-sivuja, jotka otetaan käyttöön jokaiseen View-sivuun, joka päätelaitteelle lähetetään. Eräs tällainen on nimeltään _Layout.cshtml. Se on sivu, joka toimii jokaisen View-sivun ”emona”.

_Layout sivu on normaali html-sivu, jossa on !DOCTYPE, head-, html- ja body-tagit. Olen siihen määritellyt joka sivulla näkyvän header-osion, jossa on navigaatiopainikkeita sovelluksen eri toimintoihin, sekä footer osion johon voin sijoittaa Ajax-kutsujen

vastauksia. Lisäksi eräässä kohtaa kutsutaan metodia RenderBody() jolle kohtaa Razor lisää Viewn jota päätelaitteelle ollaan lähettämässä.

Alla olevassa kuvassa 4 nähdään miten mm. navigaatiovalikko ja kaikki triggerit listaava näkymä reagoivat pienempään näyttöön. Isolla ruudulla näkymä on vasemman puolen näköinen ja kaikki sivun elementit ovat näkyvissä. Pienemmällä ruudulla navigaatioelementit sijoitetaan erillisen painikkeen taakse ja triggereitä esittävästä table-elementistä piilotetaan mm. Times tried ja Times true sarakkeet.



Kuva 4: FarView käyttöliittymä eri näyttökoilla

6.3.2 Ajax

Olen käyttänyt Ajaxia jQueryn avustuksella muutamissa näkymissä, kuten triggerin ja konfiguraation editointi ja tallennettujen tietojen esitys. Ajax mahdollistaa näissä näkymissä paremman käyttäjäkokemuksen tekemällä asioita taustalla samalla kun UI silti päivittyy koko ajan.

Esimerkiksi triggerin editointinäkymässä on ehtolauseen tarkistus toteutettu Ajax-kutsulla. Pelkkä malliin liitetty validointi ei olisi pystynyt varmistumaan ehtolauseen oikeellisuudesta, sillä ehto pitää parseroida jolloin siitä saadaan kaivettua esille toiminnot ja toimijat joille pitää suorittaa validointi vielä erikseen.

Ajaxilla ehdon tarkistus toimii niin, että trigger-kontrollerin erästä metodia kutsutaan ja annetaan sille argumentiksi ehtolause. Kontrolleri tekee lauseelle tarkistuksen ja tu-

loksen perusteella lähettää takaisin Json muotoisen vastauksen. Näkymä saa vastauksen ja päivittää sen perusteella näkymää josta käyttäjä sitten näkee ehtolauseen oikeellisuuden tai mahdolliset virheet.

6.3.3 Knockout.js ja vis.js

Olen käyttänyt monessa näkymässä knockout.js kirjastoa. Se noudattaa nk. MVVM-mallia joka tulee sanoista Model, View ja ViewModel. Se on hyvin läheistä sukua .NET Coren MVC:lle.

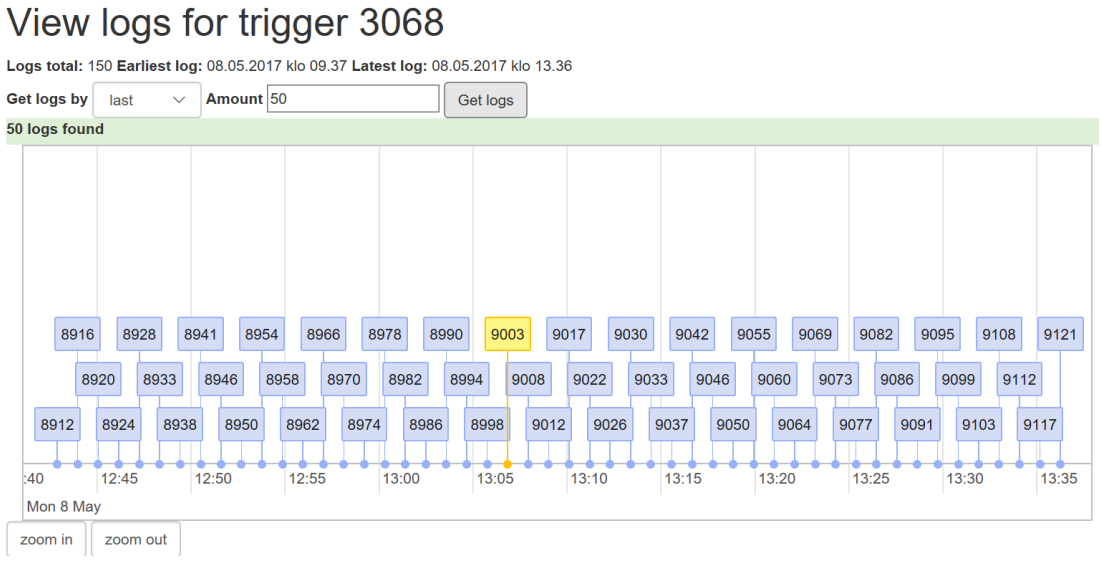
Knockoutissa esitettävä data sijoitetaan nk. view modeliin joka sitten ”sidotaan” käyttöliittymään, josta lähtien Knockout pitää huolta sekä siitä, että käyttöliittymäelementit näyttävät oikean sisällön, mutta myös siitä, että view model muuttuu, jos käyttöliittymäelementtiä, kuten vaikkapa tekstikenttä, muokataan.

Knockout view modelit kirjoitetaan Javascript funktioina, joihin sisällytetään propertyja. Nämä propertyt voivat olla ihan perustietotyyppejä kuten string tai int. Tällöin Knockout voi esittää kyseiset propertyt käyttöliittymän tekstilaatikoissa tai labeleissa. Knockoutin hienoin ominaisuus tulee esiin, kun propertyyn sijoitetaan Knockoutin oma ko.observable funktio, jolle annetaan parametriksi haluttu arvo. Tällöin sidonta toimii kahteen suuntaan ja esim. tekstilaatikossa muokattu teksti heijastuu takaisin view modeliin. Alla olevassa kuvassa 5 näkyy pätkä erästä FarViewssa käytettyä Knockout view modelia.

```
34  function messageViewModel() {
35      var self = this;
36
37      self.busyAjax = ko.observable(true);
38      self.messages = ko.observableArray([]);
39
40      self.filterDateStart = ko.observable();
41      self.filterDateEnd = ko.observable();
42      self.filterId = ko.observable();
43      self.filterLabel = ko.observable();
44      self.filterValue = ko.observable();
45
46      self.anyFilter = ko.computed(function () {
47          return (self.filterDateStart() || self.filterDateEnd()
48              || self.filterId() || self.filterLabel() || self.filterValue());
49      });
50
51      self.filteredMessages = ko.computed(function () {
52          if (!self.anyFilter()) {
53              return self.messages();
54          } else {
```

Kuva 5: Esimerkki Knockout view modelista

Vis.js kirjastosta olen käyttänyt osaa Timeline. Se mahdollistaa datapisteiden helpon esityksen aikajanalla jolla voi liikkua ja jota voi zoomata sisään ja ulos. FarViewssä Timelineä käytetään esittämään triggereiden tallentamia tietoja. Alla olevassa kuvassa 6 näkyy Timeline, johon tiedot on Ajax kutsulla ladattu. Yksi loki on valittu aktiiviseksi, jolloin sen tiedot näytetään Timelinen alla (ei näy kuvassa). Kuvan yläosassa näkyvät kontrollit, joilla voi valita miten kyseisen triggerin lokitiedostoja haetaan.



Kuva 6: Triggerin lokitietojen esitys

6.4 Logiikan lukemisen service

Logiikan luvun toteuttava service on itsessään jaettu kahteen pääkomponenttiin. Toinen osa lukee ja kirjoittaa tietokantaan sekä pitää kirjaa triggereistä. Toinen osa taas on ainoastaan vastuussa UI:n kanssa käytävästä socket-pohjaisesta kommunikaatiosta.

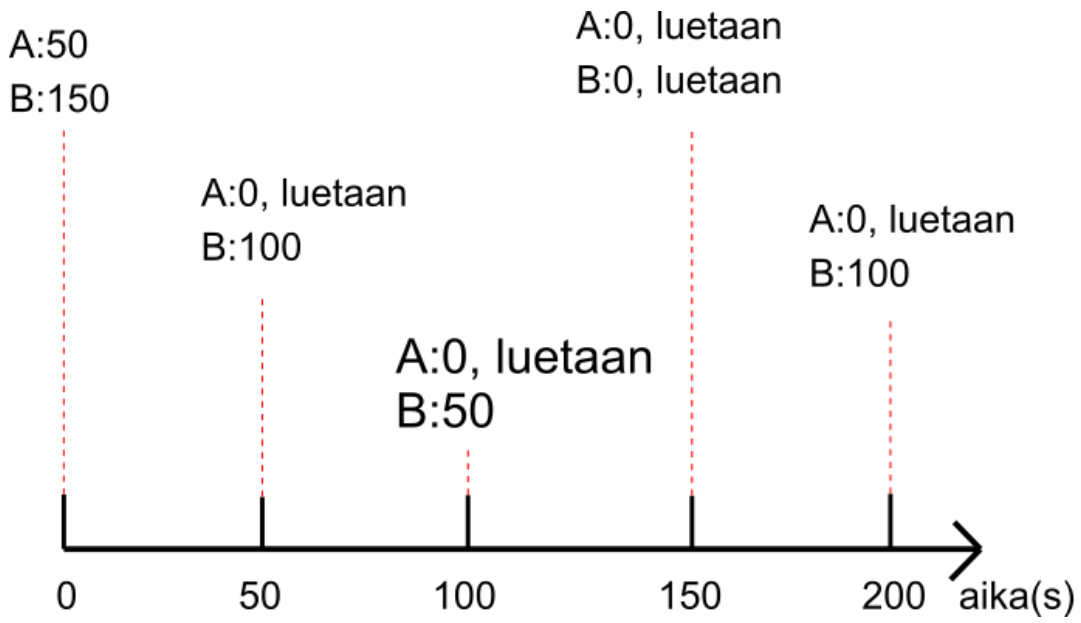
Kommunikaatio-osa on toteutettu omana SocketListener nimisenä C# luokkanaan ja se käynnistetään yksinkertaisesti luomalla luokasta uusi ilmentymä. Tällöin luokka luo uuden TcpListener tyyppisen olion ja kytkee sen kuuntelemaan paikallista IP-osoitetta 127.0.0.1 halutussa portissa. Sitten SocketListener käynnistää uuden säikeen, jolle annetaan työksi odottaa, kunnes joku ottaa yhteyttä äskeisessä vaiheessa avattuun porttiin. Kun näin tapahtuu, hoitaa SocketListener kommunikaation yhteyden ottajan kanssa ja lopulta sulkee yhteyttä varten avatun socketin. Näin tapahtuu, kunnes SocketListener lopetetaan ja se sulkee TcpListenerin.

Logiikka servicen varsinainen pääosa lukee ja kirjoittaa tietokantaan ja pitää huolta triggereistä. Triggereiden luvussa käytetään apuna C# luokkaa System.Threading.Timer. Sen avulla ohjelman säie saadaan halutuksi ajaksi ”nukkumaan” jolloin se ei turhaan käytä prosessoriaikaa pyöriessään tyhjässä silmukassa odottamassa hetkeä jolloin on taas aika lukea triggereitä.

Eri triggerit ajoittuvat mahdollisesti luettavaksi hyvinkin eri aikoina. On kuitenkin tärkeää, että kaikki triggerit tulevat luetuksi oikeina aikoina. Tämä mahdollistuu yksinkertaisella tekniikalla, jossa ohjelma nukutetaan aina ajanjaksoksi, joka on aika nykyhetkestä lähimpään seuraavaan hetkeen, jolloin jokin triggeristä tulee jälleen luettavaksi. Tätä varten täytyy seurata jokaisen triggerin kohdalla aikaa siihen, kunnes trigger pitää jälleen lukea.

Alla oleva kuva 7 selventää asiaa. Siinä näemme tilanteen, jossa meillä on triggerit A ja B ja niiden jäljellä olevan ajan ennen kyseisen triggerin lukemista. A luetaan 50 sekunnin välein ja B 150 sekunnin välein. Hetkellä 0 olemme lukeneet tietokannasta molemmat triggerit ja etsimme lyhimmän ajanjakson, jonka välein triggereitä luetaan.

Tässä tapauksessa se on A:n 50 sekuntia. Ohjelma laitetaan siis nukkumaan 50 sekunniksi ja kun se herää, vähennämme molempien triggerien lukuodotusaikaa 50 sekuntia. Tällöin A:n odotusaika on 0 ja se siis luetaan, B:llä aika on 100 sekuntia.



Kuva 7: Esimerkki triggerien lukuajoituksista

Koska A:n odotusaika oli 0, eli se luettiin, asetamme uudeksi odotusajaksi A:n lukusyklin, eli tässä tapauksessa 50 sekuntia. Sitten etsimme jälleen lyhimmän sen hetkisen lukuodotusajan, joka on tässä kohtaa A:n 50 sekuntia. Näin ollen ohjelma laitetaan taas nukkumaan 50 sekunniksi. Triggerien lukemista jatketaan vastaavasti hamaan tulevaisuuteen. Ainoan poikkeuksen muodostavat hetket, jolloin UI:lta tulee socketin läpi käsky lukea triggerit uudelleen tietokannasta. Tällöin palaamme ajanhetkeen 0 ja jatkamme päivitetyillä triggereillä ja niiden lukuodotusajoilla.

6.5 Trigger parseri

Raumaster Paperin toivoma triggereiden dynaamisuus toteutetaan niin, että triggerit annetaan string muodossa, josta se käännetään tietokoneen ymmärtämään muotoon. Tällöin triggeri on mahdollisimman helppo ihmisen ymmärtää, mutta se on silti tarpeeksi ilmaisuvoimainen, jotta sillä voidaan toteuttaa myös komplekseja ehtolauseita. Tutkittuani ilmaiseksi saatavilla olevia kirjastoja, jotka mahdollistavat edellä kuvatun

asian, päädyin ohjelmoimaan asian itse. Saatavilla olevat kirjastot olivat mielestäni tähän tarkoitukseen liian isoja ja toimintarikkaita.

Esimerkkinä toimikoon trigger, joka on muotoa ”!DB0.DBX12.2 OR DB1.DW10 > 501”. Tämä tarkoittaa, että halutut muistipaikat tallennetaan tietokantaan, jos DB1.DW10 on enemmän kuin 501 tai jos DB0.DBX12.2 on false. Jotta tietokone ymmärtää halutun käyttäytymisen, on trigger käännettävä merkkijonosta tietokoneelle paremmin sopivaksi. Tätä prosessia kutsutaan parseroinniksi jonka dictionary.com määrittelee seuraavasti: ”Tietokoneissa. (merkkijonon) analysointi jonka seurauksena merkkiryhmiä saadaan assosioitua kontekstin mukaisen kieliopin syntaktisten yksiköiden kanssa” (Dictionary.com 2017).

Parserointi antaa tekemässäni toteutuksessa listan tokeneita, jotka on jaettu operandeihin ja operaattoreihin. Kuten nimetkin kertovat, operaattorit operoivat yhdellä tai useammalla operandilla. Esimerkkitapauksessa tokenit olisivat operaattori NOT, operandi DB0.DBX12.2, operaattori OR, operandi DB1.DW10, operaattori > ja operandi 501.

Tokenit ovat vielä tässä kohtaa nk. infix-muodossa. Siinä operaattorit ovat aina operandiensa välissä (Bam 2012). Jotta tietokoneen on helpompi operoida tokeneilla ja laskea lauseen lopputulos, käännetään tokenit nk. postfix tai reverse polish notation muotoon. Siinä operandit ovat pääasiassa vasemmalla ja operaattorit pääasiassa oikealla. Tähän käytän nk. Shunting Yard algoritmia (Tiliksew;Khim ja Silverman 2017). Esimerkkitapaus muuttuu muotoon DB0.DBX12.2, NOT, DB1.DW10, 501, >, OR.

Nyt tietokoneen on helppo pinoa käyttäen laskea lauseen lopputulos. Tokeneita aletaan tarkastella vasemmalta oikealle ja jos kyseessä on operandi, se painetaan pinoon. Jos kyseessä taas on operaattori, pinosta otetaan tarvittava määrä operandeja, joille suoritetaan kyseinen operaatio. Operaation tulos painetaan takaisin pinoon. Näin jatketaan, kunnes lopulta tokeneiden loppuessa pinossa tulisi olla ainoastaan yksi operandi jäljellä, joka on koko lauseen lopputulos. Algoritmi sisältää tietysti myös virhekäsittelyn tilanteiden varalle, joissa esim. pinossa ei ole tarpeeksi operandeja halutulle operaatiolle tai operandit ovat väärän tyyppisiä jne.

6.6 Tietokanta

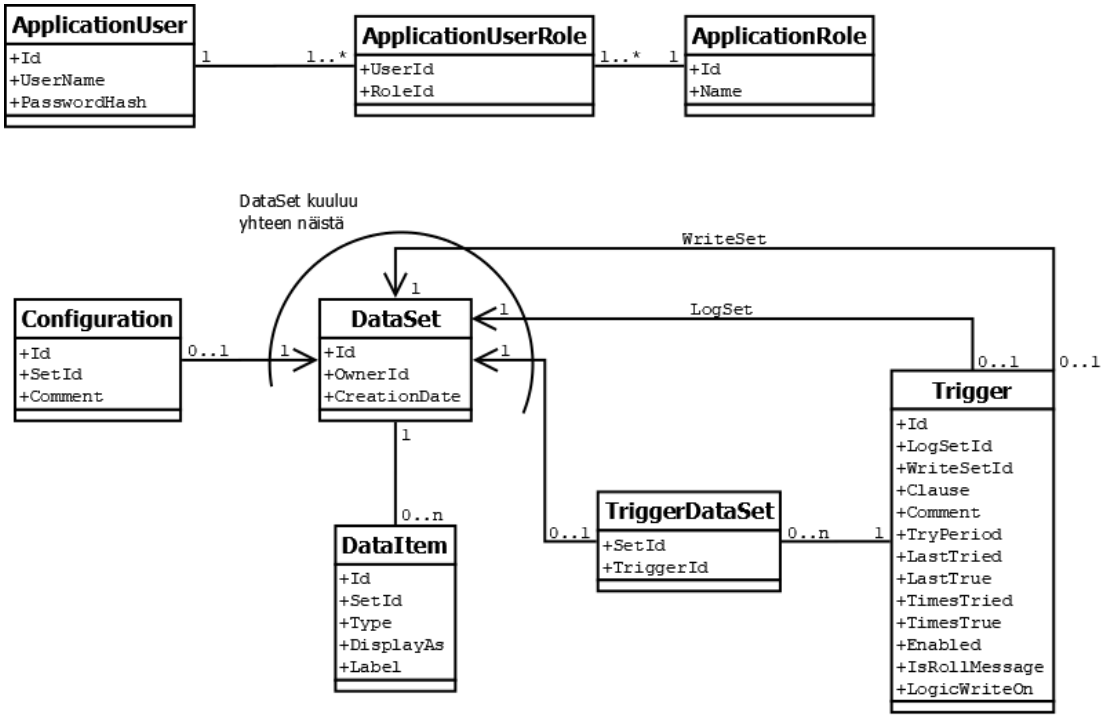
Vaikein asia Raumaster Paperin ongelman asettelussa tietokannan kannalta oli se, että he halusivat tietojen tallennuksesta dynaamista. Heidän käyttämänsä Siemensin logiikkayksiköt käyttävät useita eri datatyyppejä (bitti, sana, tuplasana jne.) joita kaikkia on tarpeen mukaan pystyttävä tallentamaan yhteen tallennukseen. Samoin erilaisia konfiguraatiotietoja ja rullaviestejä on tarpeen tallentaa niin, että tallennettavia tietokombinaatioita on teoriassa ääretön määrä. Tämä johti nopeasti niin kutsutun EAV-mallin jäljille.

Entity-Attribute-Value malleja on monia eri hienostuneisuustasoja. Yksinkertaisimmillaan ne ovat pelkkiä key-value-pareja, joihin on lisätty ylimääräinen kenttä, kertomaan minkä tyyppinen arvo on tallennettu. EAV - malli saattaa olla sopiva ratkaisu, kun on kyseessä esim. suuri joukko dataluokkia joilla on hyvin vähän yhteisiä attribuutteja. (Poole 2013)

Toteuttamassani tietokannassa EAV - malli näyttäytyy tavassa, jolla tietokantaan tallennetaan mm. konfiguraatiotiedot sekä ohjeet siitä mitä triggerit tallentavat tai kirjoittavat, että niiden varsinaiset tallennetut arvot. Kaikki yksittäiset tallennetut arvot tallennetaan DataItem-tauluun, jossa niille annetaan uniikki Id, SetId joka viittaa DataSet-tauluun, Type joka kertoo minä tietotyyppinä kyseistä tietoa käsitellään ja Label johon voi tallentaa jonkinlaisen selitteen tiedosta.

Jokainen DataSet-taulun rivi kuvaa kokoelmaa DataItem-arvoja. Koska DataItem viittaa aina DataSettiin johon se kuuluu, on mahdollista tallentaa yhteen DataSet-kokoelmaan niin monta DataItem-arvoa kuin halutaan. Jokaisesta DataSetistä tallennetaan myös uniikki Id, ja hetki jolloin kyseinen DataSet on luotu.

Tietokannan Trigger-taulu sisältää triggerit joita logiikan on tarkoitus tarkastella. Siinä voidaan yksittäiselle triggerille määrittää mm. suoritettava logiikkalause, vapaavalintainen kommentti ja aika jonka kuluttua trigger aina koitetaan. Tauluun tallennetaan myös tietoa siitä koska triggeriä on viimeksi koitettu ja koska se on viimeksi ollut tosi sekä kuinka monesti triggeriä on kokeiltu ja monestiko se on ollut tosi.



Kuva 8: Tietokannan rakenne

FarView ohjelmaa varten tietokantaan tallennetaan myös yksinkertainen käyttäjätunnistustieto. Ohjelmassa on etukäteen määriteltyjä ApplicationRoleja, joille on ohjelmassa annettu eri oikeuksia. Näitä rooleja annetaan ApplicationUsereille, joista tallennetaan mm. nimi ja salasanasta laskettu ja saltattu hasharvo. Koska roolit ja käyttäjät sidotaan toisiinsa kolmannen taulun, ApplicationUserRole avulla, on mahdollista, että yhteen rooliin voi kuulua useita käyttäjiä ja yhdellä käyttäjällä voi olla monta roolia.

Tietokannassa hyödynnetään muutamassa kohdassa viiteavainten ON DELETE CASCADE sääntöä. Sääntö tarkoittaa sitä, että poistettaessa entity B, johon entity A:lla on viiteavain, poistetaan myös entity A. Tämä mahdollistaa sen, että esim. Trigger poistettaessa poistetaan tietokannasta ensin kyseiseen triggeriin kuuluvat DataSetit. Koska jokaisella DataItemilla on viiteavain DataSettiin, johon se kuuluu, pois-

tuvat nämä DataItemit automaattisesti. Samoin automaattisesti poistuvat myös triggeriin kuuluvat TriggerDataSetit koska niissä on viiteavain DataSettiin. Kun vielä poistetaan DataSetit, jotka kertovat mitä trigger lukee logiikalta ehtolauseen ollessa tosi ja mitä logiikalle mahdollisesti kirjoitetaan, poistuvat automaattisesti niihin tallennetut DataItemit. Tämän jälkeen voidaan poistaa kyseinen rivi itse Trigger taulusta.

6.6.1 Entity Framework

Käytän FarViewssä tietokantaa Entity Framework Coren läpi. Entity Framework Core on viimeisin versio Microsoftin suosittelemasta datasaantiteknologiasta sovelluksille jotka käyttävät .NET Core sovelluskehystä. Se on suunniteltu kevyeksi, laajennettavaksi ja tukemaan monialustakehitystä. EF Core on nk. object-relational mapper. ORM on tekniikka joka mahdollistaa kehittäjien päästä käsiksi dataan oliomallin mukaisesti suorittamalla mappaus sovelluksen ohjelmointikielen ja relaatiomallin mukaisesti tallennetun datan välille. (Learn Entity Framework Core 2016)

Otin käyttöön Entity Frameworkin FarViewssä koska olen käyttänyt sitä ennenkin ja havainnut sen toimivaksi. EF nopeutti ja helpotti kehitystyötä mm. juuri oliopohjaisuudellaan. Sen sijaan, että olisin joutunut kirjoittamaan ja debuggaamaan kaikki dataa käsittelevät osat erillisinä SQL lauseina, käsittelinkin vain oliokokoelmia. Esimerkiksi kaikkien tietojen hakeminen tiettyä konfiguraatiota varten näyttäisi SQL:llä alla olevan kaltaiselta:

```
SELECT *
FROM Configuration AS C JOIN DataSet AS DS ON C.SetId = DS.Id JOIN DataItem AS
DI ON DI.SetId = DS.Id
WHERE C.Id = 3
```

Tämän jälkeen joutuisin vielä muokkaamaan saatuja rivejä, jotta saisin ne sijoitettua C#:n sisällä jonkinlaiseen olioon. Entity Framework sen sijaan antaa minulle saman datan suoraan oliona paljon helpommin:

```
context.Configurations.AsNoTracking().Include(c => c.DataSet).ThenInclude(s =>
s.Items).ToList();
```

Toinen suuri hyöty Entity Frameworkin käytöstä on vahva tyypitys joka mahdollistaa virheiden huomaamisen jo niiden kirjoitushetkellä eikä vasta SQL - lauseita suorittaessa. Visual Studio myös antaa täyden Intellisense tuen kirjoitettaessa EF datasaantikoodia, koska se on normaalia C# ohjelmointia.

6.6.2 Lokitietojen poisto

Eräs huomion arvoinen seikka tietojen tallennuksen yhteydessä on aina niiden tallennuskoon mahdollinen paisuminen. Tästä huolehtiminen tulee FarViewn tapauksessa kyseeseen ainoastaan triggereiden kanssa jotka, tarpeeksi tiheällä try periodilla varustettuna, saattavat jossain tilanteessa tallentaa isoja määriä lokeja.

Tämän estämiseksi olen ohjelmoinut tietokantaan nk. tietokantalaukaisimen, joka on tietokantaan tallennettava ohjelma, joka ajetaan aina halutussa tilanteessa. Tässä tapauksessa olen kytkenyt laukaisimen TriggerDataSet tauluun siten että, aina kun kyseiseen tauluun lisätään rivi, ajetaan ohjelmoimani laukaisin.

Kyseinen laukaisin hakee ensin konfiguraatiotiedoista mahdollisesti löytyvän määritteen yhden FarViewn triggerin lokitietojen maksimimäärälle. Jos sellaista ei löydy, käytetään ennalta asetettua arvoa. Tämän jälkeen laukaisin alkaa käydä läpi TriggerDataSet tauluun lisättyjä rivejä, siten että jokaista FarViewn triggeriä kohden algoritmi käydään läpi kerran.

Laukaisin laskee ensin, montako TriggerDataSettiä kyseisellä FarViewn triggerillä on, ja jos niitä on enemmän kuin sallittu määrä, poistetaan päivämääräjärjestyksessä vanhemmista alkaen DataSet taulun rivejä kyseiseltä triggeriltä niin monta, että päästään sallitun rajan alle. Koska TriggerDataSet taulussa on viiteavain joka osoittaa DataSet tauluun, ja sen ON DELETE säännöksi on asetettu CASCADE, johtaa tämä siihen, että DataSettejä poistettaessa poistuvat myös kyseiset TriggerDataSetit ja samalla perusteella DataSettiin kuuluvat DataItemit. Näin ollen FarView triggerkohtaiset lokitallennukset pysyvät aina alle maksimimäärän.

7 FARVIEWN TESTAUKSESTA

Tietokoneohjelman testaaminen on prosessi, jolla varmistetaan, että ohjelma toimii kuten odotettua. On ensiarvoisen tärkeää, että sen jälkeen, kun olemme selvittäneet mitä ohjelmaa olemme tekemässä, meillä on myös käsitys siitä, että olemme tekemässä sitä oikein. Yksinkertaistettuna ohjelmistotestaus vastaa kysymyksiin:

- Tekeekö ohjelmamme sitä mitä sen pitäisi tehdä?
- Toimiiko ohjelmamme siten kuin sen pitäisi toimia?

(BBC Academy 2017)

Ohjelmistotestauksessa on useita eri menetelmiä. Eräitä tällaisia ovat mm. yksikkötestaus ja integraatiotestaus. Yksikkötestauksessa testauksen kohteena ohjelman jokin pieni osa, esimerkiksi luokan metodi. Yksikkötestaus keskittyy vain siihen, että testauksen kohteena oleva yksikkö toimii oikein. (Agendaless Consulting 2017)

Näin ollen, jos testauksen kohteena oleva yksikkö käyttää jotain toista objektia, kuten vaikkapa tietokantayhteyttä, annetun tietokantayhteyden ei tarvitse olla, ja todennäköisemmin ei tulisi olla, oikea, toimiva yhteys. Sen sijaan annettu tietokantayhteys voi olla nk. mock, eli eräänlainen oikean ja toimivan objektin jäljitelmä. Mock-objekti kuitenkin riittää siihen, että yksikön toimivuus voidaan varmistaa esim. varmistamalla, että yksikkö kutsuu mockin odotettuja funktioita. (Agendaless Consulting 2017)

Yksikkötestauksen lisäksi on olemassa nk. integraatiotestaus. Siinä varmistutaan siitä, että yksikkötestatut yksiköt toimivat, kun niitä testataan yhdessä toistensa kanssa. (Agendaless Consulting 2017)

FarViewta testatessa olen käyttänyt sekä yksikkö- että integraatiotestausta. Näiden lisäksi olen myös ajanut ohjelmaa ja kokeillut sen eri toimintoja yrittäen löytää puutteita ja virheitä hieman kuin eksploratiivisessa testauksessa, mutta ilman varsinaista testiohjelmaa. Olen myös testannut ohjelman toimintaa fyysisten virhetilanteiden varalta mm. ottamalla verkkojohdon irti ja kytkemällä sen takaisin ohjelman ollessa ajossa.

Testaus on suoritettu James Newkirkin ja Brad Wilsonin ohjelmoimaa xunit kirjastoa ja sitä hyödyntävää lisäpakettia xunit.runner.visualstudio käyttäen. Apuna on ollut

myös Daniel Cazzulinon ohjelmoima Moq kirjasto, jota käyttäen olen pystynyt luomaan edellisessä luvussa kuvattuja Mock-objekteja. Testeissä olen mockannut mm. logiikkayksikön kanssa keskustelevan Sharp7-luokan ja repositorioita. Näin olen pystynyt tarkasti kontrolloimaan arvoja ja toimia joita mockatut luokat palauttavat ja tekevät. Testien ajo ja tulokset integroituvat Visual Studioon josta ne löytyvät Test Explorer ikkunasta.

7.1 Yksikkötestaus

Suurimmat yksiköt jotka yksikkötestasin ovat triggereiden parseroinnista huolehtiva ExpressionCore kirjasto ja viewt tuottavat controllerit. Näiden lisäksi yksikkötestauksessa olivat luokat jotka huolehtivat UI:n ja logiikkalukijan välisestä kommunikaatiosta. Teoriassa yksikkötestauksen alle menisivät myös mm. datan siirrosta huolehtivat Model-luokat, mutta koska niissä ei ole mitään metodeja, ja näin ollen mitään toiminnallisuutta, ei niitä ole tarpeen testata.

7.2 Integraatiotestaus

Vaikein osa FarViewn testauksessa olivat integraatiotestit. Sopivien testimetodien löytäminen vei aikansa ja senkin jälkeen edessä oli ongelmia. Mm. sen päättäminen, miten tietokantayhteyksiä käyttävät ohjelman osat testataan, ei ollut helppoa. Ensimmäinen vaihtoehto, jota yritin soveltaa, oli, että kaikki testit ajettaisiin oikeaa SQL Serveriä vasten. Näin jo testitilanteessa päästäisiin mahdollisimman lähelle ohjelman oikeaa ajoympäristöä.

Tässä lähestymistavassa oli kuitenkin muutamia ongelmia, jotka lähtivät siitä, että on toivottavaa, että aina testejä ajettaessa on tietokannan tilanne tunnettu. Näin ollen tulisi tietokanta esim. aina tyhjentää joka testimetodin ajoa ennen. Tämä on kuitenkin oikeassa tietokannassa aavistuksen muita vaihtoehtoja hitaampi operaatio sen vuoksi, että data kirjoitetaan aina levylle asti sekä datana, että lokimerkintöinä. Ja testaamisen luonteen vuoksi (toistuvaa, usein suoritettavaa) hylkäsin tämän vaihtoehdon.

Toisena vaihtoehtona esiin nousivat niin kutsutut in memory tietokannat jotka nimensä mukaisesti sijaitsevat ainoastaan keskusmuistissa eikä niihin sijoitettavaa dataa välillä kirjoiteta levylle asti. Tutkittuani in memory tietokantoja ja niiden soveltuvuutta tilanteeseen, päädyin lopulta käyttämään Sqlite nimistä tietokantaa. Kyseiseen ohjelmaan päädyin sen takia, että se tuki tarvitsemiani ominaisuuksia (mm. vierasavaimet) parhaiten ja oli silti ilmainen ja kevyt.

Kaiken kaikkiaan on integraatiotestauksessa ollut apuna myös FarViewn ohjelmistokehykseksi valitsemani .NET Core. Siihen sisäänrakennettu Dependency Injection helpottaa sekä ohjelman itsensä aikana tapahtuvaa palvelujen käyttöä, että testien aikana tapahtuvaa mockaamista. DI:n perusperiaate on, että ohjelman alussa luodaan eräänlainen palveluympäristö, josta sitten aina annetaan jotain luokkaa tarvitsevalle ohjelman osalle kyseisen luokan haluttu implementaatio. Ohjelmaa oikeasti ajettaessa annetaan esim. oikea tietokantayhteys, mutta testien aikana vaikkapa juuri in memory tietokantaa käyttävä mock-objekti.

7.3 Eksploratiivinen testaus

Eksploratiivisessa testauksessa testaajat voivat käyttää ohjelmaa, miten haluavat, ja käyttää informaatiota, jota ohjelma heille tarjoaa reagoidakseen, muuttaakseen testauksen suuntaa ja yleisesti kokeillakseen ohjelman toiminnallisuutta ilman rajoitteita. Eräs eksploratiivisen testauksen tavoite on päästä ymmärrykseen siitä, miten ohjelma toimii, miltä sen käyttöliittymä näyttää ja mitä toiminnallisuutta se sisältää. Eräs toinen tavoite on löytää virheitä, sillä ohjelman reuna-alueiden tutkiminen ja mätien kohtien löytäminen on eksploratiivisen testauksen yksi erikoisalue. (Whittaker 2012)

Olen FarViewta testatessani käyttänyt eksploratiivista testausta juuri ohjelman reuna-alueiden ja käyttöliittymän testaukseen. On ymmärrettävän vaikeaa testata käyttöliittymää ohjelmallisesti, sitä miltä käyttöliittymän tulisi näyttää ja mitä toiminnallisuutta sen tulisi sisältää. Tämä koska on vaikea sekä määritellä yksikäsitteisesti, että ohjelmallisesti testata, miltä esim. ohjelman jonkin näytön tulisi näyttää ja miten sen reagoida. Esimerkiksi näytön ulkoasun testaus vaatisi, että ohjelman renderöimää näyttöä

verrattaisiin vaikkapa aiemmin napattuun kuvankaappaukseen. Tällöin kuitenkin, tehdessä suora yksi-yhteen vertaus, pieninkin muutos renderöidyssä näytössä johtaisi testin epäonnistumiseen.

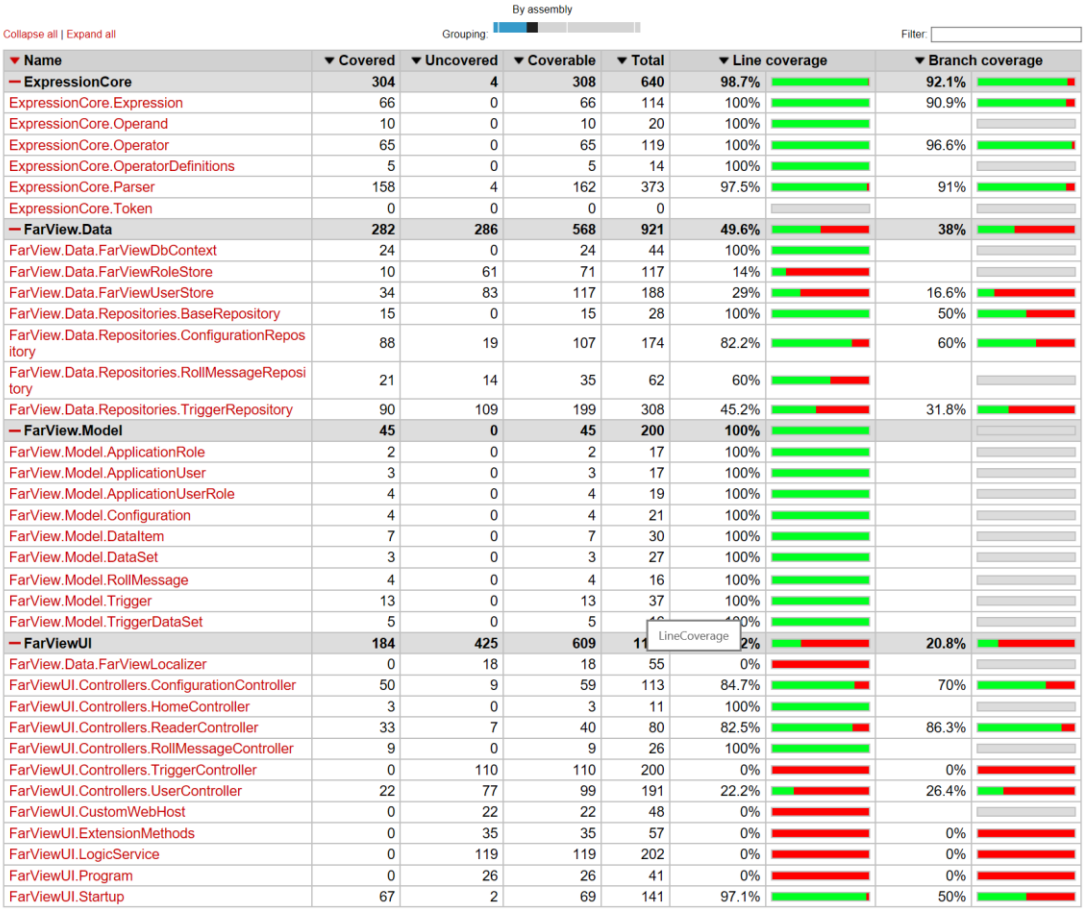
Ohjelman teon aikana olen toistuvasti simuloinut eri näytöissä erilaisia tilanteita vaihdellen mm. syötteen oikeellisuutta, syötteen määrää, suorittamiani toimenpiteitä, käyttämääni selainta ja niin edelleen. Näiden testien paljastamat virheet ovat omalta osaltaan ohjanneet kehittämään sekä FarViewn oikeakoodisuutta, että sen käyttöliittymän näyttöjen toimivuutta ja toiminnallisuutta.

7.4 Testikattavuus

Lyhyesti sanottuna testikattavuus on menetelmä, jolla voidaan varmistua siitä, että edellä kuvatut testit oikeasti testaavat kirjoitettua ohjelmakoodia. Vaikka kaikki testit menisivät läpi, ei se vielä riitä siihen, että voidaan olla varmoja siitä, että ohjelma on testattu hyvin. Testien läpäisyn lisäksi on tärkeää, että on testattu suuri osa koodista. Testikattavuuden mittaamiseen on myös olemassa monia eri kriteerejä kuten lausekattavuus, polkukattavuus, ehtokattavuus, haarakattavuus jne. (Johnson ei pvm)

FarViewta testatessani löysin ilmaisohjelman OpenCoverage, joka pystyi yhdessä ReportGeneratorin kanssa tuottamaan testikattavuustiedoista suoraan html-sivuja, joita oli helppo selata. Näiden avulla oli helppo huomata luokat ja metodit joita ei oltu vielä testattu ja näin ollen kirjoittaa niille testejä.

Alla olevassa kuvassa 9 on esitetty eräs tällainen testiraportti FarViewn testauksen ollessa vielä kesken. Siitä nähdään, että ExpressionCore nimiavaruudessa sijaitsevat, triggerien ehtolauseiden parserointiin liittyvät luokat, on testattu kohtuullisesti, sillä sekä lause-, että haarakattavuus on yli 90%. Toisaalta taas FarViewUI nimiavaruudessa on vielä paljon testaamatonta koodia.



Name	Covered	Uncovered	Coverable	Total	Line coverage	Branch coverage
ExpressionCore	**304**	**4**	**308**	**640**	**98.7%**	**92.1%**
ExpressionCore.Expression	66	0	66	114	100%	90.9%
ExpressionCore.Operand	10	0	10	20	100%	
ExpressionCore.Operator	65	0	65	119	100%	96.6%
ExpressionCore.OperatorDefinitions	5	0	5	14	100%	
ExpressionCore.Parser	158	4	162	373	97.5%	91%
ExpressionCore.Token	0	0	0	0		
FarView.Data	**282**	**286**	**568**	**921**	**49.6%**	**38%**
FarView.Data.FarViewDbContext	24	0	24	44	100%	
FarView.Data.FarViewRoleStore	10	61	71	117	14%	
FarView.Data.FarViewUserStore	34	83	117	188	29%	16.6%
FarView.Data.Repositories.BaseRepository	15	0	15	28	100%	50%
FarView.Data.Repositories.ConfigurationRepository	88	19	107	174	82.2%	60%
FarView.Data.Repositories.RollMessageRepository	21	14	35	62	60%	
FarView.Data.Repositories.TriggerRepository	90	109	199	308	45.2%	31.8%
FarView.Model	**45**	**0**	**45**	**200**	**100%**	
FarView.Model.ApplicationRole	2	0	2	17	100%	
FarView.Model.ApplicationUser	3	0	3	17	100%	
FarView.Model.ApplicationUserRole	4	0	4	19	100%	
FarView.Model.Configuration	4	0	4	21	100%	
FarView.Model.DataItem	7	0	7	30	100%	
FarView.Model.DataSet	3	0	3	27	100%	
FarView.Model.RollMessage	4	0	4	16	100%	
FarView.Model.Trigger	13	0	13	37	100%	
FarView.Model.TriggerDataSet	5	0	5			
FarViewUI	**184**	**425**	**609**	**11**	**2%**	**20.8%**
FarView.Data.FarViewLocalizer	0	18	18	55	0%	
FarViewUI.Controllers.ConfigurationController	50	9	59	113	84.7%	70%
FarViewUI.Controllers.HomeController	3	0	3	11	100%	
FarViewUI.Controllers.ReaderController	33	7	40	80	82.5%	86.3%
FarViewUI.Controllers.RollMessageController	9	0	9	26	100%	
FarViewUI.Controllers.TriggerController	0	110	110	200	0%	0%
FarViewUI.Controllers.UserController	22	77	99	191	22.2%	26.4%
FarViewUI.CustomWebHost	0	22	22	48	0%	
FarViewUI.ExtensionMethods	0	35	35	57	0%	0%
FarViewUI.LogicService	0	119	119	202	0%	0%
FarViewUI.Program	0	26	26	41	0%	0%
FarViewUI.Startup	67	2	69	141	97.1%	50%

Kuva 9: Esimerkki FarViewn testikattavuusraportista

7.5 Testaamisen oikeellisuudesta

Työn alla olevan ohjelmiston testaaminen eri tavoin ja suuri testikattavuusprosentti ovat molemmat tärkeitä asioita ja melkeinpä vaatimuksia hyvälle koodille. Ne eivät kuitenkaan ole riittäviä takeita yksinään varmistamaan, että ohjelmisto on laadukas.

Sataprosenttinen yksikkötestikattavuus ei tarkoita, että meillä on hyvät testit tai edes sitä, että testit ovat valmiit. Testeistä saattaa puuttua tärkeää dataa ja saatamme testata

vain datalla jolla testit onnistuvat, näin ollen emme testaa lainkaan datalla joka aiheuttaa epäonnistumisia. Sataprosenttinen testikattavuus ei kerro mitään puuttuvasta koodista, puuttuvasta virheenkäsittelystä tai puuttuvista vaatimuksista. (Ruberto 2017)

On helppo luoda esimerkki testeistä, joissa saavutetaan 100% testikattavuus, mutta jotka silti päästävät virheitä läpi. Oletetaan, että meillä on alla olevan kaltainen metodi, joka laskee, montako prosenttia luku a on b:stä ja palauttaa tuloksen doublena:

```
public double inPercent(int a, int b) {
   return ((a / b) * 100);
}
```

Oletetaan myös, että yritämme olla mahdollisimman fiksuja ja testaamme metodia arvoilla, joissa a on suurempi kuin b, b on suurempi kuin a ja jompikumpi tai molemmat ovat negatiivisia ja positiivisia vuorotellen. Näillä testeillä saavutamme 100% lause-, ja polkukattavuuden ja oletamme, että testit ja koodi ovat hyvin suunniteltuja ja laadukkaita. Tosimaailmassa kuitenkin joku kutsuu metodia siten, että b on 0 ja joko kaataa koko ohjelman tai vähintään ei ainakaan saa oikeaa tulosta.

Näinkin pienestä esimerkistä on helppo nähdä, että ohjelmistojen testaaminen on vaikeaa ja usein niiden 100% testaaminen on täysin mahdotonta johtuen mm. erilaisten syötteiden ja ohjelmiston suorituspolkujen määrästä. FarViewtä ohjelmoidessani ja testatessani olen pyrkinyt ottamaan tämän huomioon siten, että yksikkötestit olisivat oikeasti mahdollisimman 100% vedenpitäviä ja ottaisivat huomioon eri syöte ”luokat” (negatiiviset arvot, ääriarvot, 0, jne.). Siitä eteenpäin siirryttäessä olen yrittänyt integraatiotestauksella päästä siihen, että useimmissa tapauksissa eri osien välinen yhteistyö toimisi ilman ongelmia ja lopulta eksploratiivisella testauksella olen yrittänyt simuloida sitä, miten käyttäjä mahdollisesti tekee erikoisia valintoja.

Tästä kaikesta huolimatta olen varma, että FarView sisältää ohjelmointivirheitä ja tilanteita, joita en ole edes osannut ottanut huomioon.

8 FARVIEWN ASENNUKSESTA RAUMASTER PAPERILLE

Ohjelman kehityksen aikana olen asentanut ja poistanut sekä itse ohjelman, että sen käyttämän tietokannan, useita kertoja kokeillessani eri asioita. Nämä toimet ovat auttaneet sekä luomaan kuvauksen FarViewn asennukseen käytettävästä prosessista, että varmistumaan siitä, että luomani prosessi todennäköisesti toimii, kun ohjelmaa asennetaan oikeaan käyttökohteeseen.

Asennusprosessi on sarja loogisesti eteneviä askeleita ja etenee tällä hetkellä seuraavasti:

- Asennetaan SQL Server tietokantapalvelu ja mahdollisesti SQL Server Management Studio ohjelma. Tietokantapalveluinstanssi nimetään tietyllä tavalla ja sen kirjautumisvaihtoehdot asetetaan nk. Mixed Mode Authentication tilaan, jolloin tietokantapalvelimeen pääsee kirjautumaan sekä Windows-tilillä, että SQL Serverin login-tunnuksilla.
- Luodaan varsinainen FarView-tietokanta valmiiksi tallennetuilla SQL skriptitiedostoilla, jotka ajetaan esim. juurikin SSMS ohjelmalla tietokantapalveluinstanssiin. Samalla tavalla luodaan tietokantalaukaisin, joka huolehtii ylimääräisten lokitiedostojen poistosta.
- Kopioidaan sekä käyttöliittymä, että logiikan luku ohjelmat omiin hakemistoihinsa ja varmistetaan, että appsettings konfiguraatiotiedoston määritykset vastaavat mm. tietokantayhdysmerkkijonon osalta edellä asennettuja.
- Tässä vaiheessa voidaan edellä suoritettujen toimenpiteiden toimivuus varmistaa ajamalla käyttöliittymä komentoriviltä ja sen jälkeen avaamalla internetselaimella osoite http://localhost josta käyttöliittymän pitäisi vastata ja siihen pitäisi päästä kirjautumaan sisään oletustunnuksilla.
- Asetetaan Windows Servicet toimintaan käyttämällä komentorivikehotteen sc komentoa, jolla voidaan rekisteröidä palveluja ja asettaa niille kuvaukset. Samalla komennolla voidaan palvelut myös käynnistää saman tien.
- Event Viewer ohjelmalla luetaan FarViewn tuottama lokitiedosto jossa pitäisi olla molemmilta serviceiltä varmistusviestit onnistuneesta käynnistymisestä.

Näiden askeleiden jälkeen FarViewlle on luotu tietokanta ja siihen sisäänkirjautumistunnus ja Windows Servicet on asetettu toimintaan. Tällöin järjestelmä on toiminnassa ja sen käyttäytymistä päästään jatkossa ohjaamaan käyttöliittymän kautta.

9 YHTEISTYÖ RAUMASTER PAPERIN KANSSA

Opinnätetyöprojektissani Raumaster Paperia edusti Automation Manager Joni Lahtonen. Yhteistyö hänen kanssaan alkoi alkupalaverissa 10.02.2017. Kävin tuolloin Raumalla tapaamisessa, jossa kävimme läpi toteutettavan ohjelman pääpiirteitä ja laitteistoa, jolla ohjelman tulee toimia. Lahtonen esitteli Raumaster Paperin toiminnan lisäksi tietysti myös Siemensin automaatiologiikkayksikön, ja tämä olikin ensimmäinen kerta, kun sellaisen näin. Keskustelimme myös luettavan datan rakenteesta ja tallentamisesta, sekä tavoista joilla kerättyä dataa tultaisiin tarkastelemaan.

Ensimmäisen tapaamisen jälkeen pääsin aloittamaan tosissani työn ohjelman parissa. Yhteistyö Raumaster Paperin kanssa toimi sähköpostitse Lahtosen kanssa ja kävimmekin muutaman keskustelun, jossa tarkensimme vielä teknisiä yksityiskohtia. Pystyin myös esittelemään ohjelmaa etänä Samkin virtuaaliympäristö Bokxin läpi. Sinne asentamallani koneella suorituksessa ollut aikainen versio FarViewstä mahdollisti sen, että Lahtonen pääsi käsiksi ohjelman UI-osaan ja pystyi antamaan siitä palautetta. Tämä palaute auttoi hiomaan ohjelman käyttöliittymää Raumasterin toivomaan suuntaan ja toi esiin asioita, joita en ollut itse edes huomannut ajatella.

Ensimmäisen kerran esittelin FarViewtä Raumaster Paperille ihan kädestä pitäen 06.07.2017, jolloin tapasimme jälleen Joni Lahtosen kanssa, tällä kertaa Raumaster Paperin Porin yksikössä. Pystyin tuolloin esittelemään pitkälle edennyttä FarViewtä, jossa oli olemassa suurin osa projektin alussa asetetuista toiminnallisuusvaatimuksista. Tosin tässä tapaamisessa esiin nousi vielä toive mahdollisuudesta saada FarViewn triggereille toiminnallisuus kirjoittaa logiikalle arvoja niiden lukemisen lisäksi. Hahmottelimme yhdessä Lahtosen kanssa nopeassa tahdissa kirjoitustarpeen yksityiskohdat ja projekti kääntyi loppusuoralle jäädessäni toteuttamaan logiikan kirjoitusta ja muutamaa muuta esiin tullutta muutostoivetta.

FarViewn osalta yhteistyö Raumaster Paperin kanssa kulminoitui 21.07 jolloin olin jälleen Raumalla esittelemässä FarViewtä. Ohjelma oli käytännöllisesti katsoen valmis ja saavuttanut projektin alussa, sekä sen aikana esitetyt vaatimukset. Tapaamiset, sähköpostikeskustelut ja ohjelman etäesittely ohjasivat mm. käyttöliittymässä käytettyjen termien valinnassa, Raumaster Paperin tarvitseman toiminallisuuden hiomisessa ja teknisten yksityiskohtien asettamisessa paikoilleen.

Yhteistyö Raumaster Paperin ja Joni Lahtosen kanssa sujui hyvin. Heiltä päin pystyi aina kysymään ja varmistamaan asioita, ja kaikesta saattoi aina keskustella ja esittää oman mielipiteensä. Vaikka ohjelma tehtiin tilaajalle, ja heidän vaatimustensa mukaisesti, oli yhteistyö kuitenkin koko ajan yksi projektin kantavista voimista.

10 LOPUKSI

Opinnäytetyön kirjoitus ja sen ohella FarViewn ohjelmointi oli mielenkiintoinen kokemus. Vaikka olen ohjelmoinut harrastuksena useita vuosia, on FarView selvästi ammattimaisin ja valmein ohjelma jonka olen koskaan tuottanut. Siinä yhdistyvät mm. etukäteissuunnittelu, kommenttien kirjoitus, toiminnan varmistus ja koodipohjan siisteydestä huolen pitäminen tarkkuudella johon omissa harrasteprojekteissa ei aina tule yllettyä.

Selvästi isoin motivaattori työn tuloksen takana oli ulkoinen tilaaja. Omia harrasteprojekteja tehdessä on helppo jättää kommentit kirjoittamatta koska eihän sitä lähdekoodia lue kukaan muu kuin minä. Samoin ohjelman toiminnallisuuden virheet on helppo sivuuttaa, koska eihän ohjelmaa käytä muut kuin minä. FarViewn kohdalla kaikesta oli kuitenkin pidettävä huolta koska Raumaster Paperin henkilökunta tulisi näkemään koodin ja käyttämään ohjelmaa. Tämä ajoi aina parantamaan ja hiomaan lopputulosta.

Samalla opinnäytetyön teko oli iso silmien avaaja isojen ohjelmaprojektien kompleksisuuteen. Vaikka sen työmäärän, joka ison ohjelman tekemiseen vaaditaan, on aina jollain tavalla hahmottanut, niin vasta nyt se konkretisoitui käsin kosketeltavaksi. Jos

näinkin pienessä ohjelmassa on tuhansia asioita jotka pitää muistaa tarkastaa ja ottaa huomioon, niin paljonko niitä on isommassa, mahdollisesti kymmenien ihmisten yhdessä tekemässä, ohjelmassa?

Päällimmäisenä huomiona kuitenkin käteen jää se, että kyllä ohjelmistojen kehitys on sitä mitä haluan isona tehdä. Ongelmien selvitys, tietokantojen suunnittelu, ominaisuuksien hiominen, ne ovat niitä minua kiinnostavia asioita.

LÄHTEET

Agendaless Consulting. *Unit, Integration, and Functional Testing.* 13. 5 2017. http://docs.pylonsproject.org/projects/pyramid/en/latest/narr/testing.html (haettu 15. 5 2017).

Almende B.V. *vis.js.* 2017. http://visjs.org/index.html (haettu 15. 5 2017).

Anderson, Rick, Tom Dykstra, ja Andy Pasic. *Entity Framework.* 12. 3 2010. https://docs.microsoft.com/en-us/aspnet/entity-framework (haettu 25. 4 2017).

Bam, Surya. *Data Structure and Algorithms.* 6 2012. https://itviewson.files.wordpress.com/2012/06/infixprefixpostfix.pdf (haettu 4. 8 2017).

BBC Academy. *Why do you need to test software?* 2017. http://www.bbc.co.uk/academy/technology/article/art20150223103348419 (haettu 15. 5 2017).

Carter, Phillip, Petr Onderka, Joe Sewell, ja Maira Wenzel. *Tour of .NET.* 9. 2 2016. https://docs.microsoft.com/en-us/dotnet/articles/standard/tour (haettu 25. 4 2017).

Dictionary.com. *Parse.* 2017. http://www.dictionary.com/browse/parsing (haettu 8. 4 2017).

Johnson, Paul. *Testing and Code Coverage.* ei pvm. https://pjcj.net/testing_and_code_coverage/paper.html (haettu 15. 5 2017).

jQuery Foundation. *jQuery.* 2017. http://jquery.com/ (haettu 9. 7 2017).

Knockoutjs.com. *Knockout : Introduction.* ei pvm. http://knockoutjs.com/documentation/introduction.html (haettu 15. 5 2017).

Learn Entity Framework Core. *An Introduction To Entity Framework Core* . 22. 11 2016. http://www.learnentityframeworkcore.com/ (haettu 16. 7 2017).

Leiniö, Timo. *Mitä on responsiivinen design?* 19. 7 2012. https://www.sofokus.com/blogi/mita-on-responsiivinen-design/ (haettu 4. 5 2017).

Michaelis, Mark. *Essential .NET - Dependency Injection with .NET Core.* 6 2016. https://msdn.microsoft.com/en-us/magazine/mt707534.aspx (haettu 15. 7 2017).

Microsoft. *Introduction to Windows Service Applications.* 2017. https://msdn.microsoft.com/en-us/library/d56de412(v=vs.110).aspx (haettu 11. 4 2017).

Nardella, Davide. *Snap7 Homepage.* ei pvm. http://snap7.sourceforge.net/ (haettu 4. 5 20107).

Poole, David. *Antipathy for Entity Attribute Value data models.* 21. 01 2013. http://www.sqlservercentral.com/articles/Data+Modeling/95918/ (haettu 16. 7 2017).

Powell, K. *What is a Relational Database?* 16. 4 2017. http://www.wisegeek.org/what-is-a-relational-database.htm (haettu 25. 4 2017).

Rascia, Tania. *What is Bootstrap and How Do I Use It?* 9. 11 2015. https://www.taniarascia.com/what-is-bootstrap-and-how-do-i-use-it/ (haettu 6. 5 2017).

Raumaster Paper Oy. *Company | Raumaster Paper Oy.* 2017. http://www.raumasterpaper.fi/en/company/ (haettu 13. 5 2017).

Ruberto, John. *100 Percent Unit Test Coverage Is Not Enough.* 10. 7 2017. https://www.stickyminds.com/article/100-percent-unit-test-coverage-not-enough (haettu 15. 7 2017).

Segue Technologies. *What is Ajax and Where is it Used in Technology?* 12. 3 2013. http://www.seguetech.com/ajax-technology/ (haettu 15. 5 2017).

Siemens AG. *S7-300 - Industry Mall - Siemens WW.* 12. 7 2017. https://mall.industry.siemens.com/mall/en/WW/Catalog/Products/5000013?tree=CatalogTree (haettu 15. 7 2017).

Siemens. *Universal Controller SIMATIC S7-300.* 2017. http://w3.siemens.com/mcms/programmable-logic-controller/en/advanced-controller/s7-300/Pages/Default.aspx (haettu 11. 4 2017).

Tiliksew, Beakal, Jimin Khim, ja Josh Silverman . *Shunting Yard Algorithm.* 2017. https://brilliant.org/wiki/shunting-yard-algorithm/ (haettu 8. 4 2017).

Wagner, Bill, ja Maira Wenzel. *A Tour of the C# Language.* 10. 8 2016. https://docs.microsoft.com/en-us/dotnet/articles/csharp/tour-of-csharp/ (haettu 25. 4 2017).

Whittaker, James. *Exploratory Software Testing.* 7 2012. https://msdn.microsoft.com/en-us/library/jj620911(v=vs.120).aspx (haettu 10. 7 2017).