ESISELVITYS ORGANISAATIOIDEN VÄLISEN MAKSU- JA KIRJAUSPROSESSIN AUTOMATISOINNISTA

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Ammattikorkeakoulututkinnon opinnäytetyö

Tietojenkäsittelyn koulutusohjelma

Hämeenlinna, syksy 2019

Ilkka Lemmetti

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tietojenkäsittelyn koulutusohjelma
Hämeenlinnan korkeakoulukeskus

Tekijä Ilkka Lemmetti **Vuosi** 2019

Työn nimi Esiselvitys organisaatioiden välisen maksu- ja kirjausprosessin automatisoinnista

Työn ohjaaja/t Tommi Lahti

TIIVISTELMÄ

Opinnäytetyön lähtökohtana oli manuaalisen työn vähentäminen tilaajayrityksen talousosastolla toimivan, päivittäisistä rahasiirroista vastaavan AP/AR-tiimin maksu- ja kirjausprosessissa, jossa tasataan tilit päivittäin eri organisaatioiden välillä. Työn tavoitteena oli optimaalisen toteutustavan löytäminen ja toteutuksen suunnittelu.

Teoriaosassa perehdyttiin eri tapoihin lisätä automaatioastetta prosesseissa sekä syvennyttiin tarkemmin sovelluskehyksiin ja -arkkitehtuurimalleihin. Lisäksi käsiteltiin rahaliikennettä ja kirjanpitoa prosessiin liittyviltä osin.

Käytännön osassa lähdettiin kuvaamaan nykyprosessia ja tavoiteprosessia vaatimusmäärittelydokumentin pohjalta. Tästä päädyttiin perusteltuun toteutustavan valintaan ja siitä edelleen ratkaisun suunnitteluun. Tuloksena saatiin tilaajan toivomaa lisätietoa toteutustavan valintaan vaikuttavista tekijöistä sekä alustava suunnitelma sovelluksen toteuttamiseksi.

Avainsanat Ohjelmistorobotiikka, Sovelluskehitys, ASP.NET Core, Entity Framework Core, Razor Pages.

Sivut 26 sivua

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Degree Programme in Business Information Technology
Hämeenlinna

Author Ilkka Lemmetti **Year** 2019

Subject A pre-study on the automation of cross-organizational payment and booking processes

Supervisors Tommi Lahti

ABSTRACT

The incentive for the work was to reduce manual work in the payments and bookings of the daily cross-organizational balancing activities carried out by the commissioner's Finance department's AP/AR team, who is responsible for the accounts payable and accounts receivable. The aim was to find an optimal solution and to plan its development.

Different ways to increase the automation level of the processes in general and a deeper dive into application frameworks and architecture models were described in the theory part, as well as the relevant parts of transactions and bookkeeping.

The practical part began with a description of the current and target process from the basis of the business requirement document. The insights were then used to decide on the method and to justify the selection. Ultimately, a development plan for the solution was described.

As a conclusion, the commissioner received the requested insight on the reasons behind selecting one method over another and a plan for the development of the solution.

Keywords Robotic process automation, application development, ASP.NET Core, Entity Framework Core, Razor Pages.

Pages 26 pages

SISÄLLYS

1 JOHDANTO

Työn tilaaja on Santander Consumer Finance Oy, jonka päivittäisistä rahasiirroista vastaavan AP/AR-tiimin maksu- ja kirjausprosessiin eri organisaatioiden välillä kaivataan automaation lisäystä. Yrityksellä on järjestelmässään usean eri organisaation sopimuksia ja maksut ohjautuvat sopimuksille, vaikka ne maksettaisiin väärän organisaation tilille. Tästä johtuen päivän päätteeksi on tasattava tilit ja kirjanpito organisaatioiden kesken, mikä on toistaiseksi ollut täysin manuaalinen prosessi.

Opinnäytetyön tavoitteena on löytää perusteltu toteutustapa sekä laatia alustava suunnitelma tietoteknisen toteutuksen pohjaksi. Selvitettävänä on kysymys siitä, mikä on oikea toteutustapa prosessin automatisoimiseksi ja millä perustein? Tässä raameina toimivat tilaajalta saatu vaatimusmäärittelydokumentti, tilaajan sisäiset käytänteet, sekä pankki- ja rahoitusalan lainsäädäntö ja sääntely.

Näkökulmana toteutustavan valinnassa toimii pohdinta ohjelmistorobotiikan sekä sovelluspohjaisen automaation hyödyistä ja soveltuvuudesta peilaten työn aiheena olevaan prosessiin. Lisäksi käsitellään niitä tekijöitä, joiden yleisesti ottaen voidaan katsoa vaikuttavan toteutustavan valintaan. Valinnan jälkeen ratkaisua suunnitellaan yksityiskohtaisemmin ja käydään läpi sovellettavia tekniikoita, jotta lopputuloksena tilaaja saisi alustavan suunnitelman teknistä toteutusta varten.

2 LIIKETOIMINTAPROSESSIEN AUTOMATISOINTI

Liiketoimintaprosessien automatisointi on osa liiketoiminnan yleisestä digitalisoitumista. Asiakkaille kohdennetut palvelut ovat muuttuneet ja edelleen muuttumassa sähköisiksi, tuotteita ostetaan enenevissä määrin verkkokaupoista ja kivijalkaliikkeet taistelevat olemassaolonsa puolesta. Kaiken tämän ulkoisen liiketoiminnan digitalisoitumisen ohella myös yritysten sisäiset liiketoimintaprosessit ovat muutoksen kourissa.

Liiketoimintaprosessien automatisoinnilla tarkoitetaan yleisesti ottaen sovelluspohjaista automaatiota, jolla manuaalisten työvaiheiden sijaan käyttäjä tekee vastaavat toiminnot tietokoneen avulla. Liiketoimintaprosessien automatisoinnin lopputulemana on perinteisesti pidetty toiminnanohjausjärjestelmän käyttöönottoa. (Techopedia Inc., n.d.)

Toiminnanohjausjärjestelmät eivät kuitenkaan vastaa kaikkiin yrityksen virtaviivaistus- ja automatisointitarpeisiin. Lisäksi niiden toiminnallisuuksien jatkokehittäminen on sekä kallista että hidasta. Muutostarve on kuitenkin monella toimialalla niin nopea, että tarvitaan kevyempiä, kohdennettuja ratkaisuja.

Automatisoinnilla tavoitellaan yleensä prosessin skaalautuvuuden parantamista, yleistä nopeuttamista sekä inhimillisten virheiden aiheuttamien riskien vähentämistä. Joskus automatisoinnilla haetaan työntekijän vapauttamista kokonaan toisiin tehtäviin, toisinaan taas työntekijän rooliksi jää toimia varmistajana ja prosessin kontrolloijana. Muun muassa näistä lähtökohdista voidaan valita prosessin automatisoinnille oikea toteutustapa ja pohtia, halutaanko kehittää työntekijän avuksi uusi sovellus vai ohjelmoida ohjelmistorobotiikan avulla botti suorittamaan työtehtävää työntekijän puolesta. (Barkham, Cannata, Chitre & Lowes, 2016, s. 6)

2.1 Ohjelmistorobotiikka

Prosessien viilaaminen ja automatisointi on viimeisen reilun kymmenen vuoden aikana noussut näkyväksi ilmiöksi ja siitä uutisoidaan mediassakin varsin tiheään. Samaan aikaan ohjelmistorobotiikka on kehittynyt suurin harppauksin ja saavuttanut tason, jolla prosessit voidaan parhaimmillaan automatisoida alusta loppuun. (Barkham ym., 2016, s. 5)

Automaatiotarpeille voi olla monia eri lähtökohtia. Esimerkiksi yrityksen kasvaessa syntyy usein ajan myötä ns. teknistä velkaa, kun kehitetään nopeita ratkaisuja apukeinoiksi yksittäisiin ongelmiin ottamatta kokonaisuutta huomioon. Kasvun myötä saatetaan jättää vähälle huomiolle olemassa olevien prosessien ja käytössä olevien teknologioiden skaalautuvuus. Perinteisesti tämä on johtanut uusien järjestelmien hankkimiseen, prosessioptimointiin ja ulkoistamiseen. Uudet järjestelmät ja niiden lan-

seeraus ovat kuitenkin kalliita ja niiden käyttöönotot ovat mittavia, pitkäkestoisia projekteja. Prosessioptimoinnilla hyötyjä on mahdollista saavuttaa vain tiettyyn pisteeseen asti, eikä optimointi itsessään yleensä johda prosessin skaalautuvuuden paranemiseen. Ulkoistamisella saadaan puolestaan yleensä kertahyöty siirrettäessä palvelu tai prosessi kolmannelle osapuolelle, mutta kustannusten ja hyötyjen suhde ei ajan myötä enää juurikaan parane. (Barkham ym., 2016, s. 4)

Yllä mainittuihin ongelmiin yhtenä ratkaisuna on yleistynyt sovellusrobotiikka (RPA, Robotic Process Automation). Robotiikka soveltuu erityisesti pienen mittakaavan prosesseihin, jotka ovat rutiininomaisia ja sellaisenaan toistuvia. Robotiikka toimii pääosin käyttöliittymien kautta, toimien ikään kuin työntekijä, joka käyttää eri sovelluksia graafisten käyttöliittymien kautta ja siirtyy sujuvasti sovelluksesta toiseen. Tätä kutsutaan front-end-pohjaiseksi automaatioksi. (Barkham ym., 2016, s. 5)

Perinteinen sovellusteknologia vaatii kommunikointiin API- eli ohjelmistorajapintoja, joiden avulla eri järjestelmät voivat kommunikoida keskenään. Tällaiset rajapinnat kuitenkin puuttuvat monista vanhoista järjestelmistä ja tällöin automaatio on työlästä, kallista tai jopa mahdotonta toteuttaa. Sovellusrobotiikka on näissä tapauksissa helpommin, nopeammin ja taloudellisemmin toteutettava vaihtoehto, jolla voidaan ylittää eri järjestelmien rajat. (Vadivelrajan, 2017)

Robotiikka on kehittynyt sille tasolle, että se toimii luotettavasti toisteisten, rutiininomaisten ja strukturoitua dataa käsittelevien prosessien automatisoinnissa. Taloudellisten hyötyjen lisäksi hyvin toteutettu automaatio vähentää inhimillisten virheiden aiheuttamia operatiivisia riskejä. Robotiikan toimintaa voidaan myös monitoroida ja tallentaa, jolloin saadaan parhaimmillaan aivan uudenlaista tietoa prosessista ja sen toimivuudesta. Tällä voi olla prosessista riippuen suurikin merkitys esimerkiksi auditointia ajatellen. Ihmisen rooli robotiikassa liittyykin lähinnä robotin ylläpitoon, poikkeustilanteiden hallintaan sekä prosessin toimivuuden arviointiin ja kehittämiseen. (Barkham ym., 2016, s. 5)

2.2 **Sovelluspohjainen automaatio**

Perinteinen tapa automatisoida ja virtaviivaistaa prosesseja on kehittää ohjelmisto, jonka avulla manuaalisia työvaiheita vähennetään ja jolla ohjataan käyttäjää prosessin läpi. Robotiikasta poiketen ohjelmistoille olennaista on tarjota käyttäjälle käyttöliittymä, josta prosessi käynnistetään ja johon yleensä prosessin tulokset myös tuodaan käyttäjän nähtäville.

Siinä missä robotiikalla pyritään tavallisesti toimimaan kuin ihmiskäyttäjä, eli navigoidaan eri järjestelmien käyttöliittymissä ja tehdään toimintoja kuten ihminen niitä toteuttaisi, perustuu erillisen käyttöliittymän sisältävän

ohjelmiston toiminta yleensä API-rajapintoihin ja tietokantoihin. Tätä kutsutaan back-end-pohjaiseksi automaatioksi.

API-rajapintaa käyttävän sovelluksen ajatellaan yleisesti ottaen olevan luotettavampi tapa toteuttaa automaatio kuin robotiikka. Näin ollen tilanteessa, jossa ratkaisu voidaan toteuttaa tehokkaasti API-rajapintoja hyödyntävän sovelluksen avulla, niin tällöin se myös yleensä kannattaa toteuttaa niin. Jos kuitenkin prosessissa vaaditaan toimintoja, jossa API-rajapinnat eivät riitä, niin robotiikan käyttö voi osoittautua paremmaksi ratkaisuksi. (Camp, 2019)

Sovelluksen toteutustapaa mietittäessä valitaan tarpeisiin sopiva sovelluskehys, joka puolestaan rajaa muita valintoja. Moderni, alustariippumaton vapaan lähdekoodin vaihtoehto sovelluskehykseksi on ASP.NET Core, jossa puolestaan tietokantapohjaisen datan käsittelyn osalta toimiva ratkaisu on Entity Framework Core. Sovelluksen arkkitehtuurimalli on ASP.NET Coressa oletuksena MVC-malli. Yksinkertaisten, muutamaan sivuun pohjautuvien web-sovellusten kehittämiseen erityisen hyvin sopii MVC-malliin pohjautuva, mutta hieman muunneltu, Razor Pages-toteutusten suunnittelumalli. Tällainen kokonaisuus on lähtökohtana sovelluspohjaisen automaation käsittelemisessä tämän työn puitteissa.

2.2.1 ASP.NET Core

ASP.NET on Microsoftin kehittämällä .NET-alustalla tapahtuvaan web-kehitykseen tarkoitettu sovelluskehys. ASP.NET:n pohjalta kehitetty ja vuonna 2016 julkaistu ASP.NET Core on avoimen lähdekoodin versio, joka toimii niin macOS:lla, Linuxilla kuin Windowsillakin, toisin kuin aiemmat versiot. (Microsoft, 2019a)

ASP.NET Core sisältää suunnittelumallin, jossa on yhdistetty ominaisuuksia MVC:stä ja Web API:sta yhtenäiseksi sovelluskehykseksi. ASP.NET Coren hyviä puolia ovat modulaarisuus, suorituskyky, tietoturva ja alhaisemmat kustannukset. ASP.NET Coressa on myös sisäänrakennettu tuki riippuvuusinjektiolle. Kätevä ominaisuus Razoria käytettäessä ovat myös Tag Helperit, joiden avulla palvelinkoodia voidaan käyttää luomaan ja renderöimään HTML-elementtejä. (Tutorialspoint, n.d.)

2.2.2 Entity Framework Core

Entity Framework Core on Microsoftin kehittämä ORM, eli olio-relaatiomallintaja, joka mahdollistaa datan käsittelyn olio-pohjaisesti. Entity Framework Coren avulla voidaan suorittaa CRUD-operaatioita ilman SQL-kielisiä tietokantakyselyitä ja sen käyttö vähentää datan käsittelyyn vaadit-

tavan koodin määrää merkittävästi. CRUD-operaatioilla tarkoitetaan toimintoja, joilla tietoa luodaan (Create), luetaan (Read), päivitetään (Update ja poistetaan (Delete). (ZZZ Projects, n.d.)

Datan käsittelyyn Entity Framework Coren avulla käytetään mallia, joka muodostuu entiteettiluokista sekä tietokantasessiota vastaavasta kontekstioliosta (context object), joka mahdollistaa tietokantakyselyiden tekemisen ja datan tallentamisen tietokantaan. Malli voidaan luoda olemassa olevasta tietokannasta, koodata itse vastaamaan tietokantaa tai käyttää Migrations-ominaisuutta luomaan mallin pohjalta tietokanta, jota voidaan muokata mallin kehittyessä ja muuttuessa. Tietokantahakuihin käytetään Language Integrated Querya (LINQ). CRUD-operaatioita suoritetaan käyttämällä entiteettiluokkien instansseja. (Microsoft, 2016)

Entity Framework Coren mallissa entiteettiluokat ovat tavallisia, sovelluskehyksestä riippumattomia luokkia, eli POCO-luokkia, joilla on ominaisuuksia. Niiden avulla luodaan entiteettejä, jotka vastaavat tietokannassa yhtä riviä tietoa. Yhden entiteettiluokan entiteetit muodostavat tietokantataulua vastaavan kokoelman, DbSet:n, joista puolestaan relaatiotietoineen muodostuu tietokantaskeemaa vastaava konteksti, DbContext. (Timms, 2018)

Useimmat datan käsittelyyn tarkoitetut kehykset, kuten ADO.NET, sisältävät kirjastoja, joiden avulla datan käsittely tapahtuu tietuejoukkojen (recordset) kaltaisten datarakenteiden avulla. Tällainen lähestymistapa on kuitenkin virhealtis. Esimerkiksi sarakkeen nimen väärinkirjoittaminen tai sen muuttuminen tietokannassa ovat tavallisia virhelähteitä. Myös kenttien määrittelyjärjestyksen muuttuminen SQL-kyselyssä johtaa tällöin virheisiin, ellei vastaavia muutoksia tehdä myös sovelluksen koodiin. Lisäksi tietotyypin muutokset saattavat epäonnistua. Näissä tapauksissa koodin kääntäminen saattaa onnistua ilman virheitä, mutta ajossa (runtime) ilmeneekin virheitä. (Learn Entity Framework Core, n.d.)

Entity Framework Coren avulla voidaan siis automatisoida sovelluksen tietokantoihin liittyvät toiminnot, jolloin kehittäjän ei tarvitse keskittyä tietokantatauluihin ja -sarakkeisiin vaan datan käsittely tapahtuu olio-pohjaisesti. Datan abstraktimpi käsittely säästää koodia perinteisiin sovelluksiin verrattuna, erityisesti data-painotteisten sovellusten osalta. (EntityFrameworkTutorial.net, n.d.)

2.2.3 MVC ja MVVM -arkkitehtuurimallit

Web-sovellusten graafisten käyttöliittymien suunnittelussa ja ohjelmoinnissa käytetään varsin yleisesti MVC-ohjelmistoarkkitehtuuria. Kirjainlyhenne tulee sanoista Model, View ja Controller, eli suomennettuna malli, näkymä ja käsittelijä. Malli vastaa datasta ja sovelluksen logiikasta, eikä ole koskaan suoraan käyttäjän tavoitettavissa. Näkymä vastaa sovelluksen käyttöliittymää. Näkymän kautta käyttäjälle esitetään dataa ja tarjotaan

toiminnot, joilla käyttäjä voi sovellusta käyttää. Käsittelijä toimii välittäjänä näkymän ja mallin välissä. (Microsoft, 2019b)

Yksi MVC-arkkitehtuurimallin eduista on se, että jokaisella kolmella komponentilla on omat vastuualueensa, jolloin sovelluksen toimintalogiikka on aina erotettu käyttöliittymästä. Tällöin esimerkiksi käyttöliittymän muokkaaminen ei automaattisesti tarkoita kajoamista sovelluksen toimintalogiikkaan. (Code Maze, n.d.)

MVC-arkkitehtuurin yhtenä heikkoutena on käsittelijän merkittävä osuus. Kun käyttöliittymän vaatimukset kasvavat ja monipuolistuvat, niin vastaavasti käsittelijä paisuu jatkuvasti, jolloin sen luettavuus kärsii ja refaktorointi sekä testaaminen muuttuvat ajan myötä hankalaksi. (Vera, 2016)

Yksi keino välttää MVC-arkkitehtuurissa yleistä käsittelijän paisumista on MVVM-arkkitehtuurimalli. Lyhenne MVVM tulee sanoista Model, View, ja ViewModel. Se on varsin lähellä MVC-arkkitehtuuria, mutta erottavana tekijänä on ViewModel eli suoraan suomennettuna näkymämalli, jonka tehtävänä on muuntaa mallin informaatio näkymän ymmärtämiksi arvoiksi ja välittää dataa myös toiseen suuntaan ilman, että välissä olisi MVC-arkkitehtuurin mukaista käsittelijää. MVVM-suunnittelumallia voidaan myös yhdistellä muiden mallien kanssa, jolloin sovellus voi olla yhdistelmä esimerkiksi sekä MVVM- että MVC-suunnittelumalleja. (Luo, 2017)

2.2.4 Razor Pages

Razor Pages on osa .NET Core sovelluskehystä. Se sisältää kevyemmän version MVC-mallista. Itsessään Razor Page on samankaltainen kuin perinteisen MVC-mallin View-komponentti eli näkymä. Sillä on sama syntaksi ja samat toiminnallisuudet. Pääasiallisena erona on kuitenkin se, että Razor Pagesissa jokainen sivu muodostaa oman kokonaisuutensa, jossa näkymä ja koodi on järjestetty omaksi kokonaisuudekseen. Tämä noudattaa ohjelmoinnin niin sanottua Single Responsibility-periaatetta.

Razor Pages on lähempänä MVVM-mallia ja mahdollistaa kaksisuuntaisen tiedonsiirron (two-way data binding). Yksinkertaisemman rakenteensa ansiosta sitä pidetään myös kehittäjän kannalta selkeämpänä ratkaisuna, ja se soveltuukin erityisen hyvin sovelluksiin, joissa on melko yksinkertaisia dynaamisia sivuja, joilla tehdään lähinnä perustason datankäsittelyä. Razor Pages ei kuitenkaan rajoita perinteisten, käsittelijöihin perustuvien MVC-näkymien tai Web API:n käyttämistä samassa projektissa. (Watson, 2017)

MVC-mallin heikkoutena pidetty käsittelijän paisuminen ei Razor Pagesin mallissa ole ongelma. Sivupohjainen rakenne johtaa siihen, että sivun toiminnot on eritelty ja sijaitsevat loogisesti samassa yhteydessä itse sivun kanssa, muodostaen .cshtml ja .cshtml.cs -tiedostoparin, jossa ensin mainittu vastaa näkymää ja jälkimmäinen puolestaan MVVM-arkkitehtuurin näkymämallia. Razor Pagesissa näkymämalli on kuitenkin sivukohtainen ja

sitä kutsutaankin nimellä PageModel, eli sivumalli. Se sisältää sivun toiminnot ja välittää tietoa mallin ja näkymän välillä Näkymä ei siis tässäkään rakenteessa tiedä mitään itse mallista, eikä malli näkymästä. (Lock, 2019)

Oletuksena Razor Page sisältää yksittäiset OnGetAsync- ja OnPostAsync-metodit. Muiden toimintojen, kuten AJAX-kutsujen, useiden mahdollisten lomakkeiden lähetyksen yms. osalta on käytettävä niin sanottuja handlereita, eli käsittelijöitä. Kun sivu lähettää pyynnön, niin handlerin avulla valitaan pyynnön toteuttava metodi. (Lock, 2019)

3 RAHALIIKENNE JA KIRJANPITOVIENNIT

Yritysten rahansiirroissa jokaista tapahtumaa kohden tulee olla vastaava merkintä yrityksen kirjanpidossa. Kirjanpitovaatimukset ovat erityisen tiukat pankki- ja rahoitusalalla sekä eri markkinoilla toimivilla yrityksillä, koskien myös tytäryhtiöitä. Jos yritys esimerkiksi toimii Yhdysvaltain markkinoilla, niin kaikkia sen tytäryhtiöitäkin koskevat niin sanotut SOX-vaatimukset (Sarbanes-Oxley Act), riippumatta siitä, missä markkinoilla kyseinen tytäryhtiö toimii. Kaikkia Suomessa toimivia yrityksiä koskeva pienin yhteinen nimittäjä on kuitenkin kirjanpitovelvollisuus, joka koskee kaikkia liike- ja ammattitoiminnan harjoittajia. Liiketoiminnan harjoittajien osalta vaaditaan kahdenkertaista kirjanpitoa. (Kirjanpitolaki 1620/2015 § 2.)

3.1 Kahdenkertainen kirjanpito

Kahdenkertaisen kirjanpidon periaatteiden mukaan tapahtumat merkitään aina kahdelle tilille, jolloin jokaisesta tapahtumasta muodostuu selkeä polku, josta nähdään mistä raha on peräisin ja mihin se on käytetty. Näin muodostuvan kirjausketjun tulee olla katkeamaton.

Tämän työ puitteissa olennaista on, että tilaajan toiminnanohjausjärjestelmässä käsitellään usean eri yrityksen sopimuksia ja kirjanpitoa. Tapahtumien osalta tällöin kirjausketjun tulee olla katkeamaton yritystasolla. Tapahtumaa ei siis voida kirjata siten, että sen yksi puoli on ensimmäisen yrityksen kirjanpidossa ja toinen puoli toisen yrityksen kirjanpidossa, vaan ensimmäisen yrityksen osalta on kirjattava sekä rahan lähde että kohde, ja erikseen vastaavat kirjaukset toisen yrityksen kirjanpitoon.

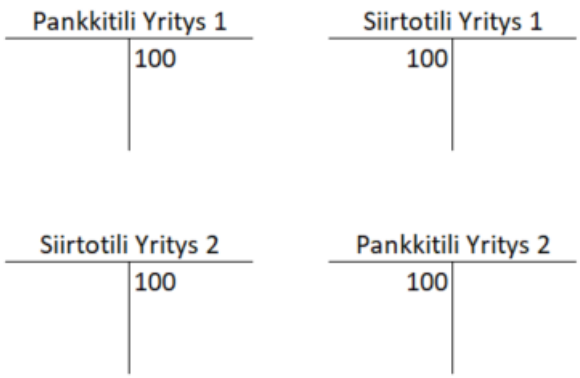
3.2 Rahaliikenne asiakkaalta yritykselle

Jokaisella yrityksellä on omat pankkitilinsä ja kirjanpitotilinsä. Toiminnanohjausjärjestelmässä sopimukselle kohdistetut suoritukset kuuluvat kyseisen sopimuksen omistavan yrityksen pankkitilille ja kirjanpitoon. Asiakkaat maksavat kuitenkin suorituksia eri yrityksen pankkitilille kuin heidän sopimuksensa edellyttäisi. Tämä johtaa siihen, että maksusuorituksia päätyy väärän yrityksen pankkitilille, mutta toiminnanohjausjärjestelmässä maksusuoritus kohdistetaan kuitenkin oikealle sopimukselle. Kun maksutapahtumat kirjataan sopimuksille, jotka kuuluvat eri organisaatiolle kuin pankkitili jolle suoritus on maksettu, niin yritysten välisiä kirjauksia varten luoduille kirjanpitotileille muodostuu tällöin saldoa.

3.3 Yritysten välinen rahaliikenne ja kirjanpito

Jotta kirjanpito ja pankkitilien saldot täsmäisivät päivätasolla yrityskohtaisesti, on yritysten pankkitilien välillä siirrettävä päivittäin rahaa, sekä tehtävä rahasiirtoja vastaavat kirjanpitoviennit. Näiden summat päätellään yritysten välisiin kirjauksiin tarkoitettujen kirjanpitotilien saldoista, joiden tulee täsmätä siten, että maksavan yrityksen kirjanpitotilin saldon tulee vastata vastaanottavan yrityksen kirjanpitotilin saldoa, mutta erimerkkisenä.

Organisaatioiden välisiä rahasiirtoja varten jokaisella yrityksellä on tilikartassa yksi kirjanpitotili per toinen yritys. Tilit on nimetty siten, että organisaatiot on eroteltu kirjanpitotilin numeron perässä olevalla organisaatiokohtaisella järjestysnumerolla, joka on erotettu alaviivalla kirjanpitotilinumerosta. Näin ollen esimerkiksi yritysten 1 ja 2 välillä tilit ovat muotoa 12345_1 ja 12345_2. Kyseisten organisaatioiden välillä on oltava kirjanpitotilit myös toiseen suuntaan (yritys 2 maksaa yritykselle 1), eli esimerkiksi kirjanpitotilit 14321_2 ja 14321_1. Kirjanpitotilien alkukirjaimesta voidaan päätellä niiden olevan tasetilejä, ja kirjanpitotilien nimet kuvaavat tilien tarkoitusta, esimerkiksi "Transfer from [Yritys 1] to [Yritys 2]". Kuvassa 1 on esimerkki kirjauksista, jotka tapahtuvat maksuprosessissa, jossa yritys 1 maksaa yritys 2:lle 100 euron summan. Kuvan ylemmän osan kirjaukset tulevat Yritys 1:n kirjanpitoon ja alemman osan kirjaukset Yritys 2:n kirjanpitoon.



Kuva 1. Tiliristikko kirjauksesta, jossa Yritys 1 maksaa 100 euroa yritykselle 2.

4 TILAAJAYRITYKSEN NYKYPROSESSI JA VAATIMUSMÄÄRITTELYN MUKAINEN TAVOITEPROSESSI

Nykyinen prosessi organisaatioiden välisten rahasiirtojen ja kirjanpitovientien toteuttamiseksi on varsin manuaalinen. AP/AR-tiimin prosessikehityksestä vastaava liiketoiminnan edustajan laatiman vaatimusmäärittelyn pohjalta voidaan kuvata tavoiteprosessi.

4.1 Nykyprosessin kuvaus

Tällä hetkellä sekä maksusuoritukset että kirjanpitoviennit tehdään manuaalisesti yksitellen toiminnanohjausjärjestelmän avulla. Tämä tehdään siten, että ensin tarkistetaan maksajan ja vastaanottajan kirjanpitotilien saldojen vastaavuus ja kun tämä on todettu, niin käydään ensin luomassa maksukirjaus ja sen jälkeen sitä vastaava kirjanpitovienti. Kumpikin työvaihe sisältää valikkojen kautta tapahtuvaa navigointia sekä manuaalista tiedonsyöttöä useaan eri kenttään. Sama työvaihe toistetaan useaan kertaan, jotta jokaisen organisaation rahasiirrot ja kirjanpitoviennit saadaan onnistuneesti toteutettua. Jokaisesta suorituksesta ja kirjanpitoviennistä otetaan lisäksi paperituloste tositteeksi maksujen tarkastajaa ja mahdollista auditointia varten.

Prosessin alkaessa tarkistetaan kirjanpitotilien saldot. Mikäli ne eivät täsmää, niin erot on selvitettävä ennen kuin prosessissa voidaan edetä. Tämän jälkeen valitaan maksavan yrityksen tili, valitaan toiminnanohjausjärjestelmän valikosta maksutoiminto, määritellään maksun saaja ja maksettava summa. Tämän jälkeen maksu voidaan toteuttaa nappia painamalla.

Maksutoiminnon jälkeen tehdään vastaanottavan yrityksen tilille kirjanpitovienti vastaamaan vastaanotettavaa summa. Tällöin valitaan vastaanottavan yrityksen tili, navigoidaan toiminnanohjausjärjestelmän valikosta hyvitystoimintoon, valitaan maksava taho ja vastaanotettava summa. Tämän jälkeen kirjanpitovienti voidaan toteuttaa nappia painamalla.

Maksuista tulostetaan tosite, joka lisätään maksuerittelyyn muiden maksujen joukkoon. Kun maksutiedosto luodaan, niin maksutosite kulkee muiden kyseisen tiedoston maksujen tositteiden kanssa tarkastajalle, joka hyväksyy maksut. Tällä hetkellä potentiaalisia maksutapahtumia on kaksikymmentä. Yllä kuvattu prosessi toistetaan siis jopa kaksikymmentä kertaa päivässä.

4.2 Tavoiteprosessi

Tavoitteena on automatisoida maksutapahtuma ja kirjanpitovienti, jotka nykyprosessissa ovat manuaalisia, toisteisia työvaiheita. Prosessi voidaan

aloittaa siinä vaiheessa, kun tietyt muut työtehtävät on tehty. Alkamisaikaa ei näin ollen voida määritellä vaan se riippuu muista tekijöistä. Tämän vuoksi prosessin käynnistys tulee tapahtua työntekijän toimesta. Työntekijän tulee myös varmistaa tilien saldot ja valita toteutettavat siirrot kirjanpitovienteineen.

Toteutettavat siirrot määritellään organisaatioiden tietojen perusteella. Tätä varten toteutuksen tulee sisältää käyttäjän muokattavissa oleva tietokantataulu, johon organisaatioiden perustiedot määritellään. Tietojen määritystä varten toteutuksessa tulee olla graafinen käyttöliittymä, jolla käyttäjät voivat muokata organisaatiotaulun tietoja CRUD-operaatioita käyttäen.

Varsinainen prosessi käynnistyy työntekijän hakiessa tilien saldot organisaatiotaulussa määritetyille tileille. Mikäli tilillä on saldoa, niin tuodaan tiliin liittyvä siirtotapahtuma näkyviin. Mikäli saldot vastaavat toisiaan, mutta erimerkkisinä, niin oletuksena merkitään tapahtuma toteutettavaksi, muussa tapauksessa näytetään ilmoitus käyttäjälle eikä merkitä tapahtumaa toteutettavaksi.

Rahasiirrot ja kirjanpitoviennit ovat eniten aikaa vieviä ja toisteisia työvaiheita, jonka vuoksi ne on oleellista automatisoida. Näin ollen, käyttäjän käynnistäessä siirtoprosessin, tulee toimenpiteiden toteutua automaattisesti siten, että lopputuloksena käyttäjälle muodostuvat tositteet tehdyistä rahasiirroista ja kirjauksista. Kaikkien valittujen tapahtumien tulee toteutua kerralla.

Yllä mainitun mukaisesti, jokaista tapahtumatyyppiä kohti muodostuu yrityksen toiminnanohjausjärjestelmässä rahasiirto ja kirjanpitovienti, sekä tosite maksun tarkastajaa varten. Tositteen tulee tallentua pdf-muodossa arkistoon, josta se on jälkikäteen haettavissa. Dokumentteja tulisi pystyä etsimään toteutusajan tai tiedostonimen perusteella.

Rahasiirtoja toteutettaessa on olennaisen tärkeää, että vain määritellyillä henkilöillä on pääsy muokkaamaan organisaatio- ja maksutietoja sekä toteuttamaan maksuja. Tämän vuoksi toteutuksessa tulee huomioida myös käyttäjäoikeuksien hallinta.

5 PERUSTELUT TOTEUTUSTAVAN VALINNALLE

Määrittelydokumentin ja toimintaympäristön vaatimukset huomioiden päädyttiin siihen, ettei robotiikan hyödyntäminen ole oikea tapa toteuttaa tilaajan vaatimuksia. Sen sijaan ratkaisuksi valittiin selainpohjainen sovellus, joka on vuorovaikutuksessa yrityksen toiminnanohjausjärjestelmän kanssa hyödyntäen www-sovelluspalvelun kautta tehtäviä API-kutsuja.

5.1 Käyttäjän näkökulma

Ohjelmistorobotiikan mahdollisuutta tutkittiin vaatimusmäärittelyn mukaisesti, mutta toisaalta määrittelyssä on useita seikkoja, jotka eivät puolla ohjelmistorobotiikkaa. Ihmisen tulee aina käynnistää prosessi selainpohjaisesta näkymästä, tehdä valintoja prosessin etenemisen suhteen sekä päästä selaimen kautta käsiksi dokumenttiarkistoon, jonne tallennetaan prosessin lopputuloksena muodostuneet pdf-tiedostot. Arkiston tulee myös sisältää hakutoiminnallisuuksia. Lisäksi prosessia varten tulee laatia muokattava taulukko, jossa määritellään prosessissa käytettävät kirjanpitotilit, tilinumerot yms. Mikään näistä seikoista ei ole optimaalisesti robotiikalla ratkaistavissa olevia asioita.

5.2 Rajapinnat

Kohdeprosessissa käytetään yrityksen toiminnanohjausjärjestelmää, joka tarjoaa ohjelmointirajapinnan. Lisäksi yrityksellä on itse kehitetty www-sovelluspalvelu (web service), jonka metodien kautta ohjelmointirajapintaa voidaan kutsua. Tämä mahdollistaa toteutustavaksi myös back-end-pohjaisen ratkaisun. Robotiikka olisi erityisen kustannustehokas vaihtoehto, jos prosessissa käytettäisiin useita järjestelmiä, jotka eivät suoraan keskustele keskenään eivätkä tarjoa API-rajapintaa.

5.3 Auditointivaatimukset ja sääntely

Robotiikalla voidaan luoda lokeja ja tallentaa tapahtumat haluttuun tiedostomuotoon, mutta yksittäisten tapahtumien haku jälkikäteen ei ole erityisen kätevää puuttuvan käyttöjärjestelmän vuoksi. Näin ollen sisäisten ja ulkoisten auditointivaatimusten mukainen sekä vaatimusmäärittelyssäkin mainittu tositteiden haku ja läpikäynti edellyttävät käyttöliittymän ja hakutoiminnallisuuksien kehittämistä.

Myös riittävän luotettavan ja suojatun ympäristön luominen prosessille robotiikan avulla on haastavaa. Pankki- ja rahoitusala on tänä päivänä erittäin tiukasti säädelty. Yrityksen tulee pystyä täyttämään tiukat kontrollivaatimukset ja kyetä osoittamaan niiden olemassaolo sekä toimivuus. Erityisesti prosessit, joissa tehdään mittavia transaktioita tilien välillä, on tärkeää suojata väärinkäytöksiltä.

5.4 **Yrityksen käytänteet**

Myös yrityksen sisäiset käytänteet ohjaavat toteutustavan valintaa. Yksi olennainen seikka on kehitetyn työkalun/automaation ylläpito. Robotiikan osalta ylläpito luovutetaan aina sille taholle, jolle automatisoitu prosessi kuuluu. Tuotantoon viennin yhteydessä laaditaan sopimus vastuun siirrosta prosessin omistavalle taholle. Muissa toteutustavoissa ylläpitovastuu on aina IT-osastolla. Jos prosessin omistavalla taholla ei ole riittävää osaamista ja ymmärrystä robotin ylläpitoon, niin tällöin vastuu ylläpidosta on vain muodollisesti olemassa ja todellisuudessa ylläpitävää tahoa ei ole.

Olennaista robotiikan kannalta on myös se, missä ympäristössä robotti toimii. Tällä hetkellä yrityksellä ei vielä ole roboteille käytössä virtuaalikoneita, vaan ne joudutaan asentamaan tavallisille työasemille, jotka on osoitettu erityisesti robottien käyttöön. Tällöin työasemien rajallisuudesta johtuen robottien ajastus tulee koordinoida työasemakohtaisesti ja yhden automatisoidun prosessin kaatuminen saattaa johtaa seuraavienkin prosessin kaatumiseen, elleivät ylläpitäjät ole valppaana. Tämä lisää riskiä siitä, että vaadittavia rahasiirtoja ei tapahdu eikä tätä huomata riittävän ajoissa. Vaikka robotti voikin työskennellä vuorokauden ympäri, niin toistaiseksi pankkien välinen rahaliikenne katkeaa klo 16 ja tämän jälkeen tehdyt maksusuoritukset eivät päädy vastaanottajalle ennen seuraavaa pankkipäivää.

6 SUUNNITTELU JA RATKAISUKUVAUS

Määrittelydokumentin vaatimukset huomioiden sovellus sopii toteutettavaksi Razor Pagesin kevennetyllä MVC-mallilla. Malli mahdollistaa myös perinteisen MVC-mallin hyödyntämisen soveltuvilta osin, mutta on oletuksena lähempänä MVVM-mallia. Razor Pagesin sivupohjainen lähestymistapa vastaa hyvin määrittelydokumentin vaatimuksia.

6.1 Sovelluksen rakenne

Sovelluksen tulee sisältää vaatimusmäärittelyssä kuvatut näkymät ja sitä kautta niiden sivumallit. Taustalla logiikasta ja tietokantatiedon käsittelystä vastaavat mallit, joiden luonnissa käytetään Entity Framework Corea. Varsinaiset maksutoimenpiteet ja kirjanpitoviennit toteutetaan kutsumalla toiminnanohjausjärjestelmän API-rajapintaa www-sovelluspalvelun kautta, joten myös näiden yhteyksien luominen on olennainen osa sovelluksen taustarakennetta.

6.1.1 Näkymän ja sivumallin rakenne

Jokaisen cshtml-tiedoston alussa tulee olla @page-määrite, joka ohjaa toimintaa siten, ettei tapahtumia ohjata käsittelijälle (controller) vaan käytetään sivumallia. Haluttuun sivumalliin viitataan puolestaan @model-määritteellä. Nämä määritteet ovat Razor-syntaksin mukaisia. Kuvassa 2 on esimerkki näistä määritteistä.

```
1    @page
2    @model SPVTransfers.Pages.Organizations.IndexModel
```

Kuva 2. Näkymän aloitusmääritteet @page ja @model

Kun sivumalli on määritelty @model-määritteellä, niin siihen voidaan viitata esimerkiksi tuotaessa näkymään tietoa. Kuvan 3 esimerkissä näkyviin kenttiin haetaan tietoa mallista, josta se välitetään sivumallin kautta näkymälle. Mallit sijaitsevat oletusarvoisen rakenteen mukaisesti Models-kansiossa. Esimerkkitapauksessa on Entity Framework Coren avulla luotu oliopohjainen malli, jonka kautta tietokannasta välitetään tietoa sivumallin kautta näkymään, jossa foreach-lauseketta, HTML-avustimia sekä lambda-lauseketta hyödyntäen tuodaan tieto taulukkoon, joka näytetään käyttäjälle.

```
@foreach (var item in Model.Balance) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.GLAccount)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.BookValue)
        </td>
```

Kuva 3. HTML-avustimien käyttö näkymässä.

HTML-avustimien lisäksi voidaan käyttää myös tag-avustimia (Tag Helpers), joskaan kaikille HTML-avustimille ei löydy vastaavaa tag-avustinta. Esimerkiksi kuvan 3 DisplayFor-HTML-avustimelle ei löydy vastinetta tag-avustimista. Kuvan 4 esimerkissä on form- eli lomake-elementin yhteydessä käytetty asp-for sekä asp-validation-for tag-avustimia. Labelin yhteydessä asp-for tuo näkyviin mallin mukaisen sarakeotsikon sille annetun arvon perusteella, kun taas asp-validation-for tarkistaa, että käyttäjän syöte on mallin vaatimassa muodossa ja antaa käyttäjälle virheilmoituksen, mikäli näin ei ole.

```
<form method="post">
    <div asp-validation-summary="ModelOnly" class="text-danger"></div>
    <div class="form-group">
        <label asp-for="Organization.Name" class="control-label"></label>
        <input asp-for="Organization.Name" class="form-control" />
        <span asp-validation-for="Organization.Name" class="text-danger"></span>
    </div>
```

Kuva 4. Tag-avustinten hyödyntäminen HTML-elementeissä.

6.1.2 Tietokannan käsittely

Tietokantaa voidaan hyödyntää ja käsitellä oliopohjaisesti Entity Framework Coren avulla. Tietokannan rivit vastaavat Entity Framework Coren entiteettejä, joiden määrittelemiseksi luodaan entiteettiluokka, joka puolestaan vastaa tietokantataulua. Entiteettiluokkien avulla tietokantarivejä käsitellään oliopohjaisesti. Käyttäjän toimenpiteiden seurauksena voidaan sivumallin kautta lähettää pyyntö mallille, joka toteuttaa pyynnön, esimerkiksi hakee tietokannasta tiedon, joka välitetään sivumallin kautta näkymään.

Kuvassa 5 on esimerkki yksinkertaisesta entiteettiluokasta. BookValue-tiedolle on kuvan esimerkissä määritelty myös tietotyypin validointia, joka on toteutettu hyödyntäen System.ComponentModel.DataAnnotations -nimiavaruuden attribuuttiluokkia. Validointiattribuuttien avulla voidaan määritellä tietotyypin ominaisuuksia ja näin esimerkiksi estää tiedon tal-

lentaminen tai syöttäminen virheellisessä muodossa. Lisäksi esimerkin entiteettiluokan viimeinen koodirivi sisältää navigointiominaisuuksia (navigation properties), joilla entiteettiluokka yhdistetään tietomallin muihin entiteettiluokkiin.

```
public class Balance
{
    11 references | 0 exceptions
    public int ID { get; set; }
    38 references | 0 exceptions
    public string GLAccount { get; set; }

    [DataType(DataType.Currency)]
    [Column(TypeName = "money")]
    36 references | 0 exceptions
    public decimal BookValue { get; set; }

    0 references | 0 exceptions
    public ICollection<Transfer> Transfers { get; set; }
}
```

Kuva 5. Balance-entiteettiluokka.

6.1.3 Www-sovelluspalveluyhteydet

Tilaajan toiminnanohjausjärjestelmän ohjelmointirajapintaa (API) kutsutaan WCF-pohjaisen www-sovelluspalvelun kautta. Yhteyden muodostamiseksi tulee ASP.NET Core-sovelluksissa käyttää Connected Services -lisäosaa, jonka kautta sovellukseen luodaan viittaukset kyseiseen www-sovelluspalveluun. Tämän jälkeen www-sovelluspalvelua kutsutaan niin sanotun clientin avulla. Kuvassa 6 on esimerkki clientin luomisesta. Esimerkissä on toteutettu niin sanottua factory- eli tehdasmallia, jossa hyödynnetään rajapintaluokkia (interface).

```
1 reference | 0 exceptions
public ICoreviewClient GetCoreClient()
{
    var client = new CoreViewService.ServiceClient();
    client.ClientCredentials.UserName.UserName = _conf.CVClientAppUser;
    client.ClientCredentials.UserName.Password = _conf.CVClientAppPassword;

    return new CoreviewClient(client, _conf);
}
```

Kuva 6. Www-sovelluspalvelu-clientin luominen.

Clientin avulla voidaan kutsua www-sovelluspalvelun metodeja. Kuvassa 7 on esimerkki kutsusta, jolla haetaan sopimustietoja www-sovelluspalvelun metodin avulla käyttäen parametrina tilin nimeä (accountName).

```
1 reference | 0 exceptions
public async Task<GetAccountInfoResponse> GetAccountInfoAsync(string accountName)
{
    var req = new GetAccountInfoRequest()
    {
        AccountName = accountName,
        PayPlan = true,
        UserName = _conf.CVClientAppUser,
        Password = _conf.CVClientAppPassword,
        Server = _conf.BLServer
    };
    return await _client.GetAccountInfoAsync(req);
}
```

Kuva 7. Metodi kutsun lähettämiseksi www-sovelluspalvelulle.

6.2 **Sovelluksen toiminnallisuudet**

Sovelluksessa tulee olla käyttäjälle kolme eri näkymää: pääsivu, dokumenttiarkisto sekä organisaatiotaulu. Sovelluksen pääsivuna toimii näkymä, josta haetaan tilien saldot, valitaan toteutettavat siirrot, käynnistetään toimenpiteet ja saadaan lopuksi tosite toteutuneista tapahtumista. Tositteet tallentuvat dokumenttiarkistoon, joka hakutoimintoineen muodostaa toisen näkymän. Pääsivun toimenpiteet puolestaan pohjautuvat organisaatiotauluun, jossa määritellään halutut tilit ja organisaatiot. Yhteistä kaikille näkymille on navigointipalkki, josta valitaan haluttu näkymä.

Myös käyttäjäoikeuksien määrittely on ensiarvoisen tärkeää sovelluksella tehtävien toimintojen luonteen kannalta. Hallinta tapahtuu kuitenkin sovelluksen ulkopuolella. Kun tehdään rajapintojen kautta toimenpiteitä toiminnanohjausjärjestelmään, niin kyseiseen järjestelmään asetetut käyttäjäprofiilit toimivat oikeuksien määrittelyn taustalla. Itse sovelluksen käyttäminen ja sovelluksen tietojen muokkaamisen mahdollistaminen tapahtuvat Windowsin Active Directoryyn asetettujen käyttäjäryhmien kautta.

6.2.1 Pääsivu

Pääsivun tulee sisältää hakunappi, jonka painallus vastaa http-metodia GET. Sen avulla haetaan organisaatiotaulun määritysten pohjalta ajantasaiset tilien saldot ja näytetään taulukkona kaikki ne siirtomahdollisuudet, joissa tileillä on saldoa. Taulukko sisältää sarakkeen, jossa viestitään käyttäjälle saldojen tilanteesta. Jos maksutilin ja kirjaustilin saldot rivitasolla vastaavat toisiaan, niin käyttäjälle viestitään, että siirto on toteutettavissa. Muussa tapauksessa varoitetaan eroavista saldoista. Viestikentän vieressä on valintalaatikko, joka oletuksena on valittuna niiden rivien osalta, joissa saldot täsmäävät. Valinnat ovat käyttäjän muokattavissa.

Kun saldot on haettu, aktivoituu pääsivulla nappi, jonka painallus toteuttaa POST-metodin. Sivumallin OnPostAsync-metodi toteuttaa maksu- ja kirjaustoimenpiteet kutsumalla www-sovelluspalvelua, joka puolestaan välittää pyynnön toiminnanohjausjärjestelmän API-rajapinnalle. Lopputulemana käyttäjälle muodostuu pdf-muotoinen tosite toteutetuista toimenpiteistä. Pdf-tiedostot myös arkistoidaan tietokantaan, josta ne ovat jälkikäteen käyttäjän haettavissa.

6.2.2 Dokumenttiarkisto

Dokumenttiarkistoon tallentuvat pdf-dokumentit, jotka jokaisen onnistuneen maksu- ja kirjaustapahtuman seurauksena muodostuvat. Näkymä sisältää listauksen dokumentteja. Listaus esitetään oletuksena uusimmasta alkaen, mutta lajittelujärjestystä voidaan muuttaa sarakeotsikkoa klikkaamalla. Lisäksi listausta voidaan suodattaa hakupalkista sekä tiedostonimen että päivämäärän perusteella. Alla on ensin lajittelun toteutuksen suunnittelua, jonka jälkeen käsitellään suodatuksen toteutusta.

Lajittelun toteuttamisessa voidaan hyödyntää tag-avustimia, joiden perusteella määritellään haluttu lajittelujärjestys. Lajitteluominaisuudet koodataan sivumalliin, johon aluksi määritellään muuttujat lajitteluparametreja varten. Dokumenttiarkistossa lajitellaan joko tiedostonimen tai päivämäärän perusteella, joten kumpaakin varten tarvitaan oma muuttujansa, jonka lisäksi tämän hetkinen lajittelujärjestys tallennetaan omaan muuttujaansa (Kuva 8).

```
2 references | 0 exceptions
public string NameSort { get; set; }
2 references | 0 exceptions
public string DateSort { get; set; }
3 references | 0 exceptions
public string CurrentSort { get; set; }
```

Kuva 8. Lajittelun toteuttamiseen tarvittavat muuttujat.

Muuttujia hyödynnetään sivumallin OnGetAsync-metodissa, jossa niille aluksi annetaan arvot sen mukaan, mitä näkymästä tag-avustimien kautta vastaanotetaan. Kuvan 9 esimerkissä CurrentSort-muuttujan arvoksi annetaan sortOrder-muuttujan arvo. Näkymässä käytetään tag-avustinta asp-route-sortOrder, josta muuttujan arvo kuljetetaan URL:n mukana. CurrentSort-muuttujan lisäksi sitä hyödynnetään annettaessa DateSort ja NameSort-muuttujille arvot. Näkymän lajittelun toteuttamiseksi luodaan myös viittaus tietokannan sisältämiin dokumentteihin IQueryable-tyyppisen muuttujan documentIQ avulla.

```
CurrentSort = sortOrder;
DateSort = String.IsNullOrEmpty(sortOrder) ? "date" : "";
NameSort = sortOrder == "Name" ? "name_desc" : "Name";
IQueryable<Document> documentIQ = from d in _context.Document
                                  select d;
```

Kuva 9. Esimerkki muuttujien arvojen määrittämisestä, osa metodia OnGetAsync.

Tämän jälkeen lajittelu tehdään hyödyntämällä sortOrder-muuttujan arvoa switch-lausekkeessa, ja lopputuloksena palautetaan määritysten mukaisesti lajiteltu lista näkymään (Kuva 10). Vaikka documentIQ-muuttujan sisältö muodostuu tietokantakyselystä, niin vasta ToListAsync-metodin suorittamisen yhteydessä kysely toteutetaan, joten ylimääräisiä tietokantakyselyitä ei tapahdu. Näin toteutettuna OnGetAsync-metodin toteutuksen aikana tehdään siis ainoastaan yksi kysely tietokantaan.

```
switch (sortOrder)
{
    case "date":
        documentIQ = documentIQ.OrderBy(d => d.DocDate);
        break;
    case "Name":
        documentIQ = documentIQ.OrderBy(d => d.DocName);
        break;
    case "name_desc":
        documentIQ = documentIQ.OrderByDescending(d => d.DocName);
        break;
    default:
        documentIQ = documentIQ.OrderByDescending(d => d.DocDate);
        break;
}
Documents = await documentIQ.AsNoTracking().ToListAsync();
```

Kuva 10. Switch-lauseke, jolla määritetään listan lajittelujärjestys.

Kun sivu ladataan ensimmäisen kerran, niin URL ei sisällä sortOrder-muuttujaa. Tällöin kuvan 10 switch-lausekkeen default-määrityksen mukaisesti dokumentit ovat päivämäärän mukaisesti lajiteltu uusimmasta vanhimpaan. Sivulatauksen yhteydessä muuttujille annetut arvot puolestaan määrittävät mitä tapahtuu, kun sarakeotsikoita klikataan, sillä tag-avustimissa on viittaus näihin muuttujiin (Kuva 11).

```
<th>
    <a asp-page="./Index" asp-route-sortOrder="@Model.NameSort"
```

Kuva 11. Tag-avustin asp-route-sortOrder.

Listauksen suodattamiseksi tulee sivulla olla tekstilaatikko käyttäjän syötettä varten. Toteutus on varsin samankaltainen kuin lajittelunkin kohdalla. Suodatuksen toteuttamiseksi tarvitaan yksi muuttuja, jonka arvoksi määritetään käyttäjän syöte (Kuva 12), sekä lisäyksiä OnGetAsync-metodiin.

```
6 references | 0 exceptions
public string CurrentFilter { get; set; }
```

Kuva 12. Muuttuja käyttäjän syötteen tallentamiseksi.

Muuttujan arvon vastaanottamiseksi tarvittavaa tekstikenttää varten näkymään laaditaan form-elementti (Kuva 13). Oletustoiminto form-elementeille on POST, mutta tässä tapauksessa toteutusmetodiksi määritellään GET, jolloin syöte saadaan kulkemaan URL:n mukana. Syöte lähetetään submit-painikkeella, jonka painallus määritellyn metodin mukaisesti kutsuu sivumallin OnGetAsync-metodia. Näkymässä painikkeen vieressä, koodissa alapuolella, on suodatuksen poistamiseksi linkki, joka tekee ohjauksen sivulle lähettämättä parametreja.

```
<form asp-page="./Index" method="get">
<div class="form-actions no-color">
    <p>
        Search documents:
        <input type="text" name="SearchString" value="@Model.CurrentFilter" />
        <input type="submit" value="Search" class="btn btn-primary" />
        <a asp-page="./Index">Reset filter</a>
    </p>
</div>
</form>
```

Kuva 13. Form-elementti GET-metodilla, sisältäen submit-napin ja linkin suodatuksen poistamiseksi.

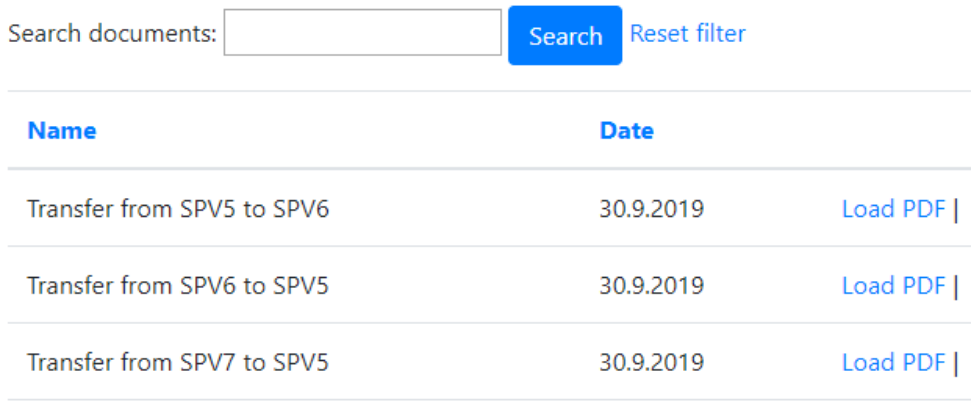
Sivumallin OnGetAsync-metodia muokataan vastaanottamaan lajittelussa käytetyn sortOrder-muuttujan lisäksi myös käyttäjän tekstisyöte, joka määritellään CurrentFilter-muuttujan arvoksi. Tämä auttaa säilyttämään suodatuksen silloinkin, kun käyttäjä lajittelee suodatuksen tuloksia, sillä tekstikentän arvoksi on määritelty näkymässä CurrentFilter-muuttujan arvo.

Suodatusta varten sivumallin OnGetAsync-metodiin lisätään if-lause (Kuva 14). If-lause tulee sijoittaa ennen aiemmin määriteltyä switch-lauseketta, mutta documentIQ-muuttujan määrittelyn jälkeen, jolloin lajittelu kohdistuu suodatettuihin tuloksiin.

```
if (!String.IsNullOrEmpty(searchString))
{
    if (!searchString.Any(x => char.IsLetter(x)))
    {
        documentIQ = documentIQ.Where(d => d.DocDate.ToString("d.M.yyyy").Contains(searchString));
    }
    else
    {
        documentIQ = documentIQ.Where(d => d.DocName.Contains(searchString));
    }
}
```

Kuva 14. If-lauseke listauksen suodattamiseksi.

Kuvassa 15 on esimerkki näkymästä, joka sisältää yllä kuvatun toteutuksen mukaiset suodatus- ja lajittelutoiminnallisuudet.



Kuva 15. Dokumenttiarkiston käyttöliittymä.

6.2.3 Organisaatiot

Organisaatiot muodostavat pohjan sovelluksen toiminnoille. Maksu- ja kirjausprosessissa on aina mukana maksun suorittava organisaatio sekä maksun vastaanottava organisaatio. Maksajalle tehdään maksutoimenpide ja vastaanottajalle kirjanpitovienti. Vaatimusmäärittelyn mukaisesti pääkäyttäjän tulee lisätä, muokata ja tarvittaessa poistaa organisaatioita sovelluksesta.

Organisaatiotietoja tallennettaessa voidaan hyödyntää handlereita. Oletusarvoisesti yhdellä Razor Pages-sivulla on yksi OnGetAsync() ja yksi OnPostAsync() -metodi, mutta handlereiden avulla voidaan luoda useampiakin toiminnallisuuksia sivulle.

Kuvan 16 esimerkissä sivulla on kaksi painiketta, joilla käyttäjä voi organisaatiotiedot syötettyään valita tallennetaanko kyseinen entiteetti maksajaksi (Payer) vai vastaanottajaksi (Receiver). Painikkeissa on käytetty han-

dler-metodia asp-page-handler ja annettu niille eri arvot. Nämä arvot vastaavat sivumallin vastaavia metodeja (Kuva 17). Handlerin arvo AddPayer vastaa sivumallin metodia OnPostAddPayerAsync, kun taas AddReceiver vastaa metodia OnPostAddReceiverAsync. Metodit ovat muuten identtiset, mutta ne tallentavat tietoa eri tauluun tietokannassa.

```
<div class="form-group">
    <input type="submit" asp-page-handler="AddPayer" value="Add New Payer" class="btn btn-primary" />
    <input type="submit" asp-page-handler="AddReceiver" value="Add New Receiver" class="btn btn-primary"
</div>
```

Kuva 16. Esimerkki asp-page-handlerin hyödyntämisestä näkymässä.

```
public async Task<IActionResult> OnPostAddPayerAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Organization.Add(Payer);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
public async Task<IActionResult> OnPostAddReceiverAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Balance.Add(Receiver);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

Kuva 17. Handlereiden arvoja vastaavat metodit sivumallissa.

Tietokanta organisaatioita varten voidaan suunnitella Entity Frameworkia hyödyntäen. Tällöin suunnitellaan ensin tietomalli ja sitä vastaavat entiteettiluokat, joiden pohjalta voidaan Entity Framework Coren scaffold-toimintoa hyödyntäen luoda tietokantakonteksti, tietokantayhteysmääreet (connection string) sekä CRUD-toimintoja hyödyntävät sivupohjat.

7 TULOKSET

Työn tilaajalla on ollut kiinnostusta robotiikan soveltamismahdollisuuksiin taloushallinnon prosesseissa. Sen vuoksi myös työn aiheena olevan prosessin osalta haluttiin saada näkemystä soveltuvasta toteutustavasta. Opinnäytetyön tuloksena syntyi perusteltu suunnitelma ratkaisuehdotuksineen. Lisäksi yritys sai työstä näkökulmia, joilla arvioida eri toteutustapojen soveltuvuutta muiden prosessien osalta.

Työn tuloksena on syntynyt myös koodikanta lopullisessa toteutuksessa hyödynnettäväksi, sisältäen tietomallin, tunnistetut www-sovelluspalvelun kutsumetodit toiminnallisuuksien loppuunsaattamiseksi sekä dokumenttiarkiston toteutusmallin.

Toteutettavan prosessin osalta todettiin, ettei robotiikka ole luontevin vaihtoehto. Tilaajayrityksen taloushallinnon osalta tulisi robotiikan suhteen miettiä, kuinka valmistautua robotiikan vaatimaan ylläpitovastuun ottamiseen, ennen kuin robotiikkaa voidaan tiimissä hyödyntää. Samoin auditointi- ja kontrollivaatimuksiin vastaamisen keinot robotiikan osalta tulisi selventää. Toisaalta myös vaatimusmäärittelyn laadinnassa tulisi jatkossa huomioida, että robotiikan hyödyntäminen ja prosessia varten laadittavien erillisten käyttöliittymien luonti eivät kulje kovin hyvin käsi kädessä, vaan robotiikkaa hyödyntävässä prosessissa tulisi pyrkiä mahdollisimman autonomiseen toteutukseen.

8 YHTEENVETO

Opinnäytetyön tavoitteena oli löytää sopiva toteutustapa prosessille sekä toteutuksen suunnittelu. Nämä tavoitteet myös saavutettiin onnistuneesti. Syventymällä toteutustavan valintaan vaikuttaviin syihin saatiin arvokasta tietoa myös kynnyskysymyksistä, joita liittyy erityisesti yrityksen taloushallinnon prosessien automatisointiin.

Henkilökohtaisesti opinnäytetyöprosessi oli opettavainen kokemus. Oli mielenkiintoista oppia ottamaan huomioon myös automatisoitavan prosessin ulkopuoliset, mutta toteutustapaan ja itse toteutukseenkin vaikuttavat seikat. Säntäämällä suoraan itse toteutuksen pariin olisi ollut mahdollista saada valmista merkittävästi nopeammassa ajassa, mutta lopputuloksena olisi todennäköisesti ollut ratkaisu, joka ei täyttäisi yrityksen sisäisiä ja lainsäädännöllisiä auditointi- ja kontrollivaatimuksia.

LÄHTEET

Barkham, J., Cannata, F., Chitre, S. & Lowes, P. (2016). Automate this - The business leader's guide to robotic and intelligent automation. Lontoo: Deloitte LLP. Haettu 11.9.2019 osoitteesta https://www2.deloitte.com/content/dam/Deloitte/uk/Documents/Innovation/deloitte-uk-leader-guide-to-robotics-automation.pdf

Camp, P. (2019). RPA vs API: Which is better? Blogijulkaisu 7.3.2019. Haettu 11.9.2019 osoitteesta https://www.campteksoftware.com/2019/03/07/rpa-vs-api-which-is-better/

Code Maze. (n.d.). ASP.NET Core MVC Series. Haettu 17.9.2019 osoitteesta https://code-maze.com/asp-net-core-mvc-series/

EntityFrameworkTutorial.net. (n.d.). What is Entity Framework? Haettu 17.9.2019 osoitteesta https://www.entityframeworktutorial.net/what-is-entityframework.aspx

Kirjanpitolaki 1620/2015. Haettu 20.11.2019 osoitteesta https://www.finlex.fi/fi/laki/ajantasa/1997/19971336#L1P2

Lock, A. (2019). Getting Started with ASP.NET Core Razor Pages. Blogijulkaisu 9.1.2019. Haettu 2.10.2019 osoitteesta https://www.twilio.com/blog/introduction-asp-net-core-razor-pages

Luo, Z. (2017). MVC vs. MVVM: How a Website Communicates With Its Data Models. Blogijulkaisu 16.10.2017. Haettu 17.9.2019 osoitteesta https://hackernoon.com/mvc-vs-mvvm-how-a-website-communicates-with-its-data-models-18553877bf7d

Microsoft. (2016). Entity Framework Core. Haettu 17.9.2019 osoitteesta https://docs.microsoft.com/en-us/ef/core/

Microsoft. (2019a). What is ASP.NET Core? Haettu 17.9.2019 osoitteesta https://dotnet.microsoft.com/learn/aspnet/what-is-aspnet-core

Microsoft. (2019b). What is the MVC pattern? Haettu 17.9.2019 osoitteesta https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-2.2

Techopedia Inc. (n.d.). Definition - What does Business Process Automation (BPA) mean? Haettu 18.9.2019 osoitteesta https://www.techopedia.com/definition/671/business-process-automation-bpa

Timms, S. (2018). Entity Framework Core Tutorial. Haettu 23.9.2019 osoitteesta https://stackify.com/entity-framework-core-tutorial/

Tutorialspoint. (n.d.). ASP.NET Core - Overview. Haettu 17.9.2019 osoitteesta https://www.tutorialspoint.com/asp.net_core/asp.net_core_overview.htm

Vadivelrajan, G. (2017). Robotic Process Automation(RPA) vs Traditional Automation. Blogijulkaisu 1.10.2017. Haettu 11.9.2019 osoitteesta https://medium.com/@gobiraj/robotic-process-automation-rpa-vs-traditional-automation-6f58c99f8e8e

Vera, D. (2016). Software Architecture: MVC Design Pattern. Blogijulkaisu 21.12.2016. Haettu 17.9.2019 osoitteesta https://medium.com/@dennis-vera.z/software-architecture-mvc-design-pattern-ceae5d5083d7

Watson, M. (2017). ASP.NET Razor Pages vs MVC: How Do Razor Pages Fit in Your Toolbox? Haettu 17.9.2019 osoitteesta https://stackify.com/asp-net-razor-pages-vs-mvc/

ZZZ Projects. (n.d.). Entity Framework Core Tutorial Overview. Haettu 17.9.2019 osoitteesta https://entityframeworkcore.com/overview