OPINNÄYTETYÖ - **AMMATTIKORKEAKOULUTUTKINTO**
TEKNIIKAN JA LIIKENTEEN ALA

AUDIT TRAIL -JÄRJESTELMÄN SUUNNITTELU JA KEHITYS ENTITY FRAMEWORKISSÄ

Tuukka Heiskanen

TEKIJÄ: Tuukka Heiskanen

SAVONIA-AMMATTIKORKEAKOULU | OPINNÄYTETYÖ
Tiivistelmä

Koulutusala Tekniikan ja liikenteen ala	
Koulutusohjelma/Tutkinto-ohjelma Tietotekniikan koulutusohjelma	
Työn tekijä Tuukka Heiskanen	
Työn nimi Audit Trail -järjestelmän suunnittelu ja kehitys Entity Frameworkissä	
Päiväys 1. Maaliskuuta 2018	Sivumäärä/Liitteet 48
Ohjaajat Kuosmanen Keijo, lehtori ja Koistinen Jussi, lehtori	
Toimeksiantaja/Yhteistyökumppanit Solteq Oyj, Utilities-liiketoimintayksikkö (InPulse Works Oy)	

Tiivistelmä

Tässä opinnäytetyössä suunniteltiin ja kehitettiin järjestelmää, joka tallentaa kaikki tietokantaan tehdyt muutokset. Toiselta nimeltään tätä voisi kutsua Audit Trail (AT) -järjestelmäksi. Asiakkaana toimi energiatoimialan sähköisiin järjestelmiin, tietoanalytiikkaan ja BI-asiantuntemukseen erikoistunut Solteqin Utilities-liiketoimintayksikkö.

Opinnäytetyö aloitettiin tutustumalla käsitteeseen Audit Trail ja kuinka se tultaisiin toteuttamaan yrityksen olemassa oleviin järjestelmiin. Työn lähtökohtana oli suunnitella ja toteuttaa AT -ratkaisu, joka seuraisi muutoksia tietokantaan ja mahdollistaisi myös AT -tiedon tallennuksen, hakemisen ja näyttämisen käyttöliittymän kautta.

Asiakkaan ehdotuksesta AT -järjestelmä kehitettiin Entity Frameworkin (EF:n) sisälle seuraamaan ja tallentamaan seurattavan tietokannan muutoksia erilliseen AT -tietokantaan. AT -tietokantataulut toteutettiin omassa skeemassaan, jolla vältettiin erillisten tietokantamigraatioiden tarve muissa tietokantakonteksteissa. Audit-lokituksen ohjelmallinen toiminnallisuus kehitettiin mahdollisimman geneeriseksi ja modulaariseksi. Apuna käytettiin tarjolla olevia luokkakirjastoja ja toteutusluokkien abstrahointia, sekä hyväksi todettuja ohjelmointiarkkitehtuureja. Toteutusta Vertailtiin myös kolmannen osapuolen EF:n sisällä toimivaan AT -luokkakirjastoon.

Työn tuloksena asiakas sai AT -järjestelmän, joka seuraa EF:n sisällä tapahtuvia muutoksia tallentaen muutoshistorian ja tietokantatapahtuman tilan erilliseen tietokantaskeemaan. AT tiedot voidaan hakea käyttäjäystävällisen selainpohjaisen käyttöliittymän kautta haku- ja asemointiparametrien määrittämällä tavalla, käyttäen apuna kehitettyä tiedonhakukerrosta. Tämän lisäksi asiakas sai toteutuksen, joka käyttää apunaan kolmannen osapuolen luokkakirjastoa toteuttaakseen vastaavanlaisen toiminnallisuuden.

Avainsanat

Audit Trail, Entity Framework, EF, DDD, Code First, POCO, POCO Proxy, Lokitus, .NET, .NET Framework, C#, MVC, ASP.NET MVC, GDPR, tietokanta, UI, luokkakirjasto, järjestelmä, muutokset, muutostenseuraus, tallennus, migraatiot, rajapinta, EDM, tietosuoja, EU

SAVONIA UNIVERSITY OF APPLIED SCIENCES

THESIS
Abstract

Field of Study Technology, Communication and Transport			
Degree Programme Degree Programme in Information Technology			
Author(s) Tuukka Heiskanen			
Title of Thesis Design and Development of Audit Trail System within Entity Framework.			
Date	1 March 2018	Pages/Appendices	48
Supervisor(s) Mr Keijo Kuosmanen, Senior Lecturer and Mr Jussi Koistinen, Senior Lecturer			
Client Organisation /Partners Solteq Corporation, Utilities business Unit (InPulse Works Ltd.)			

Abstract

The purpose of this thesis was to design and develop an audit trail (AT) system that would track and log all the changes occurred in the underlying database. Utilities business unit of Solteq Corporation, the client, implements specialized digital solutions and offers data analytical services with Business Intelligence (BI) expertise for clients that operate mostly in energy industry.

The thesis work began by getting familiar with concept of audit trail and how AT system would be implemented in the client's existing systems. The purpose was to design and implement an AT solution that could track and log the changes in the underlying database. The requirement was also to develop an easy way to retrieve tracked AT data from the browser based graphical user interface.

The client proposed that change tracking and logging of database changes would be developed within Entity Framework (EF). The database of the AT system was implemented in a separate schema in order to avoid distinct database migrations to the tracked databases. The logging of the AT system was developed as generic and modular as possible. Existing class libraries, abstraction and proven programming architecture patterns were used to achieve the demanded functionality. The solution was crosschecked against a third-party AT class library that also functions within EF.

As a result of this thesis, the client received an AT system that can automatically track changes within the premise of EF and store the changes and transaction states to a separate database schema. AT data can be retrieved from a user-friendly browser based UI with given search and positioning parameters using the developed services. The client also received a similar solution that implements a third-party AT class library in their system.

Keywords

Audit trail, Entity Framework, EF, DDD, Code First, POCO, POCO Proxy, Logging, .NET, .NET Framework, C#, MVC, ASP.NET MVC, GDPR, Database, UI, Class library, System, Changes, Change Tracking, Saving, Migrations, Interface, EDM, Data Protection, EU

SISÄLTÖ

1 LYHENTEET JA MÄÄRITELMÄT 5
2 JOHDANTO 8
3 TILAAJA 8
4 AUDIT TRAIL – AUKOTON KIRJAUSKETJU 9
4.1 Määritelmä 9
4.2 Tarpeet 9
5 JÄRJESTELMÄYMPÄRISTÖ 13
5.1 Entity Framework (EF) 14
5.1.1 Entity Data Model (EDM), Plain Old CLR Object (POCO) ja tietokantakyselyt 18
5.1.2 Muutostenseuraus, tiedontallennus ja DbContext 21
5.1.3 Code First ja migraatiot 22
6 TIEDONTALLENNUS 26
6.1 Entiteettien ja taulujen määritys 26
6.1.1 Konseptimallin luokat ja tietokannantaulut 27
6.2 Rajapintojen ja luokkien määritys 30
6.3 Geneerinen tiedontallennus 33
7 TIEDONHAKU 36
7.1 Rajapinnanmääritys 36
7.2 Geneerinen tiedonhaku 37
8 KÄYTTÖLIITTYMÄ 38
8.1 Tiedon näyttäminen käyttöliittymässä 39
9 TOTEUTUS KOLMANNEN OSAPUOLEN KIRJASTOON 42
10 KEHITYSIDEAT JA JATKOKEHITYSTARPEET 43
11 POHDINTA 44
12 LÄHDELUETTELO 45

1 LYHENTEET JA MÄÄRITELMÄT

ORM – Object Relational Mapper on suomennettuna objektirelaatiokartoittaja tai olio-relaatiokonvertteri.

.NET Framework - .NET:in kuuluva ohjelmistokehys, joka on ohjelmointialusta rakennettaessa sovelluksia. (Microsoft, Microsoft Docs, .NET Framework Guide, 2018)

EF – **Entity Framework** on Microsoft .NET alustalle Microsoftin suosittelema objektirelaatiokartoittaja, joka tarjoaa ohjelmistokehyksen objektiorientoituvasta toimialuemallista perinteiseen relaatiotietokantaan. EF:n käyttö vähentää ohjelmistokehittäjien tarvetta kirjoittaa tiedonhakukoodia. (Microsoft, MSDN, Introduction to Entity Framework, 2017)

Dapper ORM – Stack Overflow tiimin kehittämä Microsoftin .NET alustalle tarkoitettu objektirelaatiokartoittaja, joka tarjoaa ohjelmistokehyksen objektiorientoituvasta toimialuemallista perinteiseen relaatiotietokantaan. (GitHub, StackExchange/Dapper, 2018)

Instanssi – Objektiorientoituvissa kielissä instanssit ovat konkreettisia ilmentymiä objekteista, jotka yleisesti ovat olemassa sovelluksen ajon aikana.

CRM – Customer relationship management tarkoittaa strategiaa jolla hoidetaan yrityksen suhteita olemassa oleviin ja mahdollisiin asiakkaisiin. Yleisesti CRM:llä viitataan CRM -järjestelmiin, jotka helpottavat asiakaskontaktien hallintaa, myyntiä, prosessien työnkulkua ja tuottavuutta. (Salesforce - What is CRM?, 2018)

ERP – Enterprise resource planning eli toiminnanohjausjärjestelmä on yrityksen tietojärjestelmä, joka integroi yrityksen ydinprosessien hallintaa esimerkiksi tuotantoa, jakelua, varastonhallintaa, laskutusta ja kirjanpitoa.

CIS – Customer Information System eli asiakastietojärjestelmä on sähköinen järjestelmä, joka sisältää tietoa yrityksen asiakkaista.

AT – Audit Trail eli katkeamaton kirjausketju, joka työssä tarkoittaa tietomuutosten seurausta ja tallennusta.

Azure – Microsoftin julkinen pilvipalvelu, jota voidaan käyttää virtuaalipalvelinten alustana.

Microsoft SQL Server – Operationaalinen tietokantahallinnointijärjestelmä eli ODBMS (**Operational Database Management System**). (Microsoft, Microsoft Docs, SQL Server Documentation, 2018)

API – **Application Programming Interface** eli ohjelmistosovellusrajapinta, mikä on ohjelmoinnissa käytetty termi. API on funktioiden ja proseduurien joukko, mikä mahdollistaa sovellusten kehittämisen tavalla, että sovelluksella on pääsy käyttöjärjestelmän, sovelluksen tai palvelun ominaisuuksiin tai tietoon käyttäen API:n tarjoamaa rajapintaa. (TechTerms, API (Application Programming Interface) Definition, 2018)

ASP.NET MVC – Yksi ASP.NET -ohjelmistokehyksen tarjoamista ohjelmistokehyksistä. (Microsoft, Microsoft Docs, ASP.NET overview, 2018)

MVC – Model-View-Controller on ohjelmistoarkkitehtuuri ja tarkemmin suunnittelumalli, joka eriyttää sovelluksen kolmeen keskeiseen komponenttiin: malliin (Model), näkymään (View) ja kontrolleriin (Controller). (Microsoft, MSDN, ASP.NET MVC Overview, 2018)

MVC Controller - ASP.NET MVC -ohjelmistokehyksessä oleva objekti, joka käsittelee saapuvia kutsuja selaimelta, palauttaen mallien tietoja ja määritellen näkymien (View) rungot, mitkä palautetaan vastauksena selaimelle. (Microsoft, Microsoft Docs, Adding a Controller, 2018)

ASP.NET Web API – ASP.NET:n tarjoama ohjelmistokehys, joka mahdollistaa http -palveluiden (http services) rakentamisen. (Microsoft, Microsoft Docs, ASP.NET overview, 2018)

ApiController – ASP:NET Web API -ohjelmistokehyksessä oleva objekti, joka käsittelee http -kutsut. (Microsoft, Microsoft Docs, Get Started with ASP.NET Web API 2 (C#), 2018)

JS – JavaScript on kevyt, tulkattava ohjelmointikieli. Parhaiten tunnettu Web-sivuilla käytettynä skriptauskielenä. (Mozilla, MDN Web Docs, JavaScript, 2018)

GDPR - General Data Protection Regulation eli tässä yhteydessä EU:n yleinen tietosuoja-asetus, joka hyväksyttiin 14.4.2016 (Oikeusministeriö, 2017).

IDE – Integrated Development Environment eli integroitu ohjelmointikehitysympäristö, joka on sovellus, mikä tarjoaa kattavat mahdollisuudet ohjelmistokehittäjille ohjelmistokehitykseen.

VS – Visual Studio eli Microsoftin tarjoamasta IDE.

VS -Projekti – Lähdekooditiedostoista, ikoneista, kuvista ja muista tiedostoista koostuva kokonaisuus, jotka ohjelmointikielen kääntäjän tekemän kääntämisen jälkeen muodostavat ajettavan ohjelman tai verkkosivun (Microsoft, MSDN, Solutions and Projects in Visual Studio, 2018).

Solution – VS:n sisällä oleva ratkaisu, joka voi sisältää yhden tai useamman VS -projektin, sekä tarvittavat rakennusohjeet (build information) lähdekoodin muutoksesta ohjelmistokirjastoiksi (Microsoft, MSDN, Solutions and Projects in Visual Studio, 2018).

SQL - Structured Query Language eli IBM:n kehittämä standardoitu kyselykieli, jolla voidaan tehdä erilaisia muutoksia, hakuja ja lisäyksiä relaatiotietokantaan (IBM, IBM knowledge Center, Structured Query Language (SQL), 2018).

C# - Yksinkertainen, moderni, objektiorientoituva, tyypitetty ohjelmointikieli. (Microsoft, Microsoft Docs, Introduction, 2018)

CLR – Common Language Runtime eli .NET Framework:n virtuaalikone-komponentti, joka hoitaa .NET ohjelmien suorittamista.

SaveChanges() – DbContext-luokassa oleva metodi, jonka läpi synkroniset tiedontallennuskutsut menevät. Metodilla on myös asynkroninen vastine **SaveChangesAsync()** ja kolmannen osapuolen kirjastossa oli lisäksi massaoperaatioille oma metodinsa **BulkSaveChanges().**

DDD - Domain-Driven Design eli Toimialuepainotteinen suunnittelu on lähestymistapa, jossa ohjelmistokehityksen monimutkaisiin tarpeisiin vastataan yhdistämällä täytäntöönpano (implementation) kehittyviin malleihin (model). (Nilsson, 2006)

Tyypitys - Tyypitetyissä ohjelmointikielissä objekteille voi määrittää tietotyypin, jonka perusteella tiedetään objektin ominaisuudet. Tyypitys tarkoittaa, että objektille määritetään tietotyyppi.

Skeema – Tiedon organisointitapa tietokannan tiedon rakenteesta ja sen jakamisesta eri lohkoihin. Relaatiotietokannoissa tietokannan taulut voidaan jakaa eri skeemoihin. (Oracle, Oracle docs, Schema Objects, 2018)

2 JOHDANTO

Opinnäytetyön tilaajana ja toimeksiantajana toimi aluksi InPulse Works Oy. Kesällä 2017 Solteq Oyj kuitenkin osti Inpulse Works Oy:n. Yrityskaupan johdosta Inpulsesta tuli Solteqin Utilities-liiketoimintayksikkö. Opinnäytetyö oli projektiluonteinen tilaustyö, jossa kehitettiin Audit Trail -järjestelmää (AT -järjestelmää) yrityksen olemassa olevaan tuoteperheeseen.

Työssä kehitettiin yrityksen järjestelmissä tapahtuvien tietokannan tietuemuutosten seurausta ja tallennusta. Järjestelmän avulla pystytään tarkastelemaan tietuemuutosten historiaa niiden luomisesta tuhoamiseen asti. Tarpeet järjestelmän kehitykseen tulivat muuttuvista EU:n tietoturva-asetuksista ja asiakkaan toiveista. Järjestelmää voidaan kutsua AT -järjestelmäksi.

Tietuemuutosten seuranta ja tallennus kehitettiin EF:n sisään. Tietuemuutokset tallennettiin erilliseen tietokannan skeemaan, jotta AT -skeema ei olisi riippuvainen seurattavasta skeemasta. Tällä erittelyllä vältettiin ylimääräisten tietokantamigraatioiden suorittaminen yrityksen olemassa olevissa tuotanto- ja kehitysympäristöissä.

Kehitettävä järjestelmä mahdollisti EF:n sisällä tapahtuvien tietuemuutosten tallennuksen erilliseen tietokannan skeemaan. AT –tietojen hakemiseen kehitetiin lisäksi selainpohjainen UI. UI:lta pystyttiin kyselemään AT -tietoja. Tiedonhaussa pystyttiin käyttämään eri haku- ja asemointiparametrejä. UI:lla tiedot pystytiin näyttämään taulukkomuotoisessa esitysmuodossa.

Työ sisälsi suunnittelua, määrittelyä ja ohjelmistokehitystä, jonka seurauksena saatiin toteutettua AT -järjestelmä, joka kirjaa suurimman osan seuratun tietokannan muutoksista erilliseen AT -tietokantaan. AT -tietokannan tietuemuutoshistoriaa voitiin hakea tarkasteltavaksi selainpohjaisen käyttöliittymän kautta.

3 TILAAJA

Opinnäytetyön tilaajana ja asiakkaana toimi aluksi InPulse Works Oy. Kesällä 2017 Solteq Oyj kuitenkin osti Inpulsen. Yrityskaupan johdosta Inpulsesta tuli Solteqin Utilities-liiketoimintayksikkö. Utilities-liiketoimintayksikkö tarjoaa digitaalisia palveluita ja tuottaa muun muassa energia-alan toimijoille sähköisiä CRM-, ERP- ja laskutusjärjestelmiä. Tämän lisäksi liiketoimintayksikkö tarjoaa BI- ja analytiikkapalveluita. Solteq on toimialariippumaton pohjoismainen digitaalisiin liiketoimintaratkaisuihin erikoistunut ohjelmistotalo.

Utilities-liiketoimintayksikön lähtökohtana on toimia asiakasrajapinnassa ja parantaa asiakasprosessien sujuvuutta sähköisillä järjestelmillä, tarjoten laadukkaita järjestelmiä asiakkaiden ydinprosessien hoidossa. Liiketoimintayksikkö on painottunut Microsoft-osaamiseen ja onkin Microsoft Gold Partner data analytiikassa, sovellusintegraatioissa, älykkäissä järjestelmissä, pilvialustoissa, sovelluskehityksessä ja tietojärjestelmissä.

4 AUDIT TRAIL – AUKOTON KIRJAUSKETJU

Audit Traillistä eli aukottomasta kirjausketjusta (Taloushallintoliitto, Audit trail aukoton kirjausketju, 2017) puhutaan yleensä kontekstissa, jossa puhutaan tuloslaskelman ja taseen tileistä. Tarpeet kuitenkin katkeamattomille kirjausketjuille muidenkin tietojen tallentamisesta kasvavat jatkuvan digitalisaation ja niiden mukana tulevien tietosuoja-asetusten myötä.

Näitä samoja periaatteita katkeamattomista kirjausketjuista voidaan soveltaa myös käyttötarkoituksiin, joissa seurataan tietoihin tapahtuvia muutoksia ja tallennetaan muutostiedot myöhempää tarkastelua varten. Erityisen tärkeäksi muutostenseuraus -ja muutostentallennusjärjestelmät nousevat, kun käsitellään väärinkäytöille alttiita sensitiivisiä tietoja esim. henkilötietoja (2, TaA 43 §, Valtiokonttori, 2017), (Oikeusministeriö, 2017).

4.1 Määritelmä

Audit Trail, eli katkeamaton kirjausketju (2, TaA 43 §, Valtiokonttori, 2017) tarkoittaa sitä, että suojatut, tietokonegeneroidut ja aikaleimatut "auditointi jäljet" pystyvät itsenäisesti kertomaan päivämäärän ja ajan, kun elektroniseen tietoon on tapahtunut muutoksia. Muutostapahtumia on luonti, muokkaus ja poisto. Tietomuutokset eivät saa syrjäyttää aikaisemmin tallennettua informaatiota tietomuutoksesta. (ECFP, GPO, Web archive, 2018)

Edellä mainittu määritelmä AT:stä ei kuitenkaan riitä kuvaamaan työssä käytettyä termiä Audit Trail, vaan se vaatii lisämäärityksiä. Kun tekstissä puhutaan AT -järjestelmästä, sillä tarkoitetaan tietomuutostenseuraus- ja tallennusjärjestelmää. Opinnäytetyössä järjestelmä tallentaa kaikki Entity Frameworkin sisällä tehdyt tietuemuutokset seurattavan tietokannan (voidaan kutsua myös Master-tietokannaksi) tauluista/entiteeteistä erilliseen relaatiotietokantaan, jota voidaan kutsua Audit Trail -tietokannaksi.

AT -järjestelmä pitää yllä tietomuutoshistoriaa kaikista seurattavista muutoksista. Tietojen luonnista, muutoksista ja poistamisesta jää jälki eli trail. Yksittäisen tiedon muutoksia pystytään seuraamaan sen luonnista tuhoamiseen asti. Tällä tavalla voidaan todentaa tietomuutostenhistoria. Audit Trailiä voitaisiin toiselta nimeltään kutsua siis jälkien auditoinniksi. Tallennettavaan AT -tietoon sisältyy aina aikaleima, muutoksen tekijä ja tapahtunut muutos. Nämä tiedot tallennetaan, jotta tiedon oikeellisuus pystytään todentamaan kullakin ajanhetkellä. Audit Trailiä pitää pystyä soveltamaan myös tilanteisiin, joissa halutaan seurata ja todentaa tiedon katselija. Tämä tulee tärkeäksi tilanteissa, joissa käsitellään väärinkäytölle altista sensitiivistä tietoa esim. henkilötietoja.

4.2 Tarpeet

Tarpeet AT -järjestelmään tulivat toimeksiantajan toimesta, sekä muuttuvista tietosuoja-asetuksista EU:ssa. Audit Trailiä tehdään, jotta elektronisen tiedon muutostapahtumat voidaan todentaa kullakin

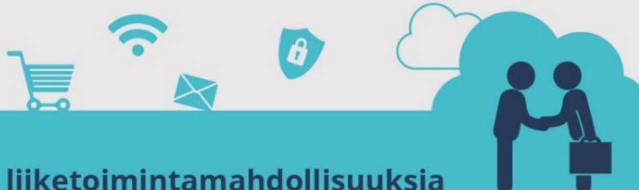
ajan hetkellä (ECFP, GPO, Web archive, 2018). AT -järjestelmässä on oleellista kirjata myös tietomuutoksen tekijä, jotta voidaan todentaa tiedonlähde, mikä parantaa tiedon uskottavuutta. AT -järjestelmän olisi hyvä voida todentaa myös tiedon katselija kullakin ajan hetkellä. Tiedon katselijan auditointi jäi kuitenkin opinnäytetyön ulkopuolelle.

Muutokset jotka liittyvät 14.4.2016 EU:n yleisen tietosuoja-asetuksen hyväksymiseen (Oikeusministeriö, 2017) GDPR eli General Data Protection Regulation, tulevat lisäämään rekisterin pitäjien ja sensitiivisten tietojenkäsittelijöiden velvoitteita henkilötietojen käsittelyn ja rekisteröinnin suhteen. Asetus pannaan täytäntöön 25.5.2018 (Tietosuojavaltuutettu, 24). Tietosuoja-asetuksen myötä tulevien vastuiden lisääntyessä tarpeet henkilötietojen, sekä muiden tietojen muutostenseuraus -ja muutostentallennusjärjestelmistä kasvavat, jonka seurauksena AT -järjestelmän kehitys nousi entistä tärkeämmäksi.

”EU:n tietosuoja-asetuksella pyritään takaamaan EU:n perusoikeuskirjan **8 artikla**: ihmisten oikeus henkilötietojen suojaan ja sen voimassaolo myös digitaaliaikana. Samalla uudistus hyödyttää digitaalisen talouden kehittämistä” (Oikeusministeriö, 2017). ”Muutokset päivittävät ja uudistavat periaatteita, joilla taataan oikeus henkilötietojen suojaan. Asetuksella vahvistetaan ihmisten oikeuksia ja EU:n sisämarkkinoita, sekä sujuvoitetaan henkilötietojen kansainvälistä siirtoa” (Oikeusministeriö, 2017). Uudistuksen myötä yksityishenkilöillä on enemmän valtaa kontrolloida henkilötietojaan ja rekisterin pitäjille tulee enemmän vastuuta suojella sitä (Oikeusministeriö, 2017). Alla olevassa infograafissa (KUVA 1.) on esitetty voimaan tulevan tietosuoja-asetuksen tärkeimmät kohdat.

Tietosuoja Euroopassa digitaaliajalla

Parempaa henkilötietojen suojaa



Enemmän liiketoimintamahdollisuuksia

Johdonmukaisempi soveltaminen ja tehokas täytäntöönpano

- Yksityishenkilöt ja yritykset voivat saada asiansa käsiteltyä heitä lähellä olevassa tietosuojaviranomaisessa ja tuomioistuimessa
- Keskitetty asiointipiste yksilöille ja yrityksille rajat ylittävissä tilanteissa kansallisten tietosuojaviranomaisten yhteistyön ansiosta

TAI

4% vuotuisesta kokonaisliikevaihdosta

Euroopan unionin neuvosto
Pääsihteeristö

© Euroopan unioni, 2015.
Tekstiä lainattaessa on mainittava lähde.

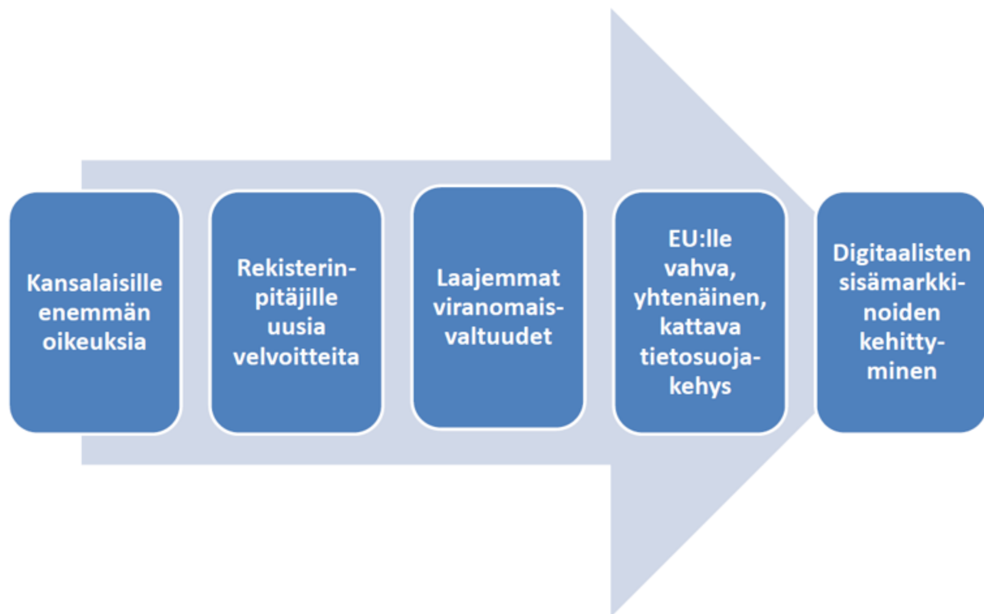
KUVA 1. EU:n neuvoston pääsihteeristön infograafissa 2018 voimaan tulevasta tietosuoja-asetuksesta. (Pääsihteeristö, EU, 2017)

Tietosuojauudistus tulee vaikuttamaan laajalta alueelta tietoturvaan liittyvissä kysymyksissä EU:n sisällä. Kuitenkin opinnäytetyön kannalta tärkeimmiksi uudistuksiksi nousi säännöt jotka vastaavat seuraaviin huolenaiheisiin: Oikeus tulla unohdetuksi, Tiedonsaanti helpottuu, Tiukemmat seuraukset rikkomuksista (Oikeusministeriö, 2017).

”**Oikeus tulla unohdetuksi:** Kun käyttäjä ei enää halua, että hänen tietojaan käsitellään, tiedot poistetaan, paitsi jos on olemassa jokin laillinen peruste säilyttää ne. Tarkoitus on taata kansalaisten yksityisyydensuoja, ei hävittää menneitä tapahtumia tai rajoittaa lehdistönvapautta” (Oikeusministeriö, 2017).

”**Tiedonsaanti helpottuu:** Ihmiset saavat tietoa siitä, miten heidän tietojaan käsitellään, ja tämä tieto on annettava heille selkeällä ja ymmärrettävällä tavalla**. Oikeus siirtää omat tietonsa tieto-järjestelmästä toiseen** helpottaa henkilötietojen siirtämistä palveluntarjoajalta toiselle” (Oikeusministeriö, 2017).

”Tiukemmat seuraukset rikkomuksista: tietosuojaviranomaiset voivat antaa EU-sääntöjä rikko-valle yritykselle sakon, joka on jopa neljä prosenttia yhtiön maailmanlaajuisesta liikevaihdosta” (Oikeusministeriö, 2017).



KUVA 2. EU:n tietoturva-asetuksen sisältöä ja tavoitteita. (Oikeusministeriö, 2017)

Samalla, kun yksityistenhenkilöiden oikeudet kasvavat. Rekisterinpitäjien ja henkilötietojen käsitteli-jöiden velvoitteet ja vastuut kasvavat. Asetuksen rikkomisesta voi seurata esimerkiksi sakkoja tai henkilötietojen käsittelykielto. (Tietosuojavaltuutettu, 2018)

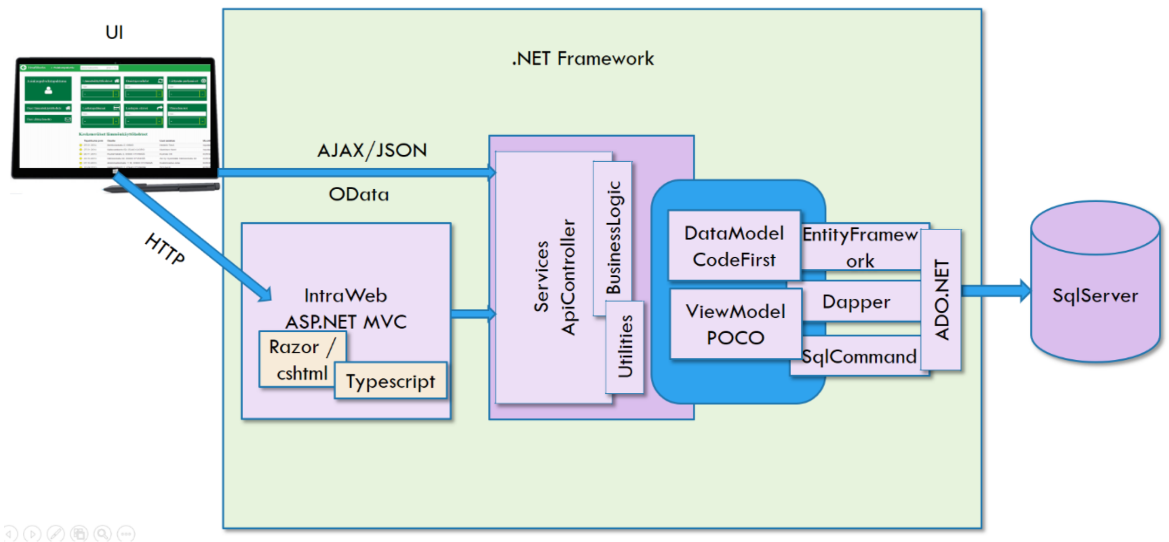
Uutta tietosuoja-asetuksessa on muun muassa riskiperusteinen lähestymistapa ja rekisterinpitäjän osoitusvelvollisuus. Mitä korkeampia riskejä henkilötietojen käsittelyyn liittyy, sitä suuremmat velvollisuudet rekisterinpitäjillä on. Rekisterinpitäjän on pystyttävä osoittamaan, että se noudattaa tietosuoja-asetusta. Tämän lisäksi rekisterinpitäjän on suunniteltava ja dokumentoitava henkilötietojen käsittely. (Tietosuojavaltuutettu, 2018)

Edellä mainitut kohdat EU:n tietoturva-asetuksesta johdetuista säännöistä ja kasvavista vastuista rekisterinpitäjillä johtaa siihen, että tarpeet henkilötietojen, sekä muiden tietojen muutostenseuraus -ja tallennuksenseurantajärjestelmistä kasvavat. Muuttuvat tietoturva-asetukset ja toimeksiantajan toiveet loivat tarpeen kehittää AT -järjestelmää, jotta yritys olisi myös lainopillisesti ajan tasalla.

5 JÄRJESTELMÄYMPÄRISTÖ

Tilaajan järjestelmät toimivat .NET ympäristössä ja Azure:ssa. Heidän tuoteperheensä on eriytetty eri ratkaisuihin (solution), jotka itsessään koostuvat useista erilaisista VS -projekteista, joissa on keskinäisiä riippuvuussuhteita eri VS projekteista syntyviin luokkakirjastoihin. Osa yrityksen VS -projekteista syntyvistä luokkakirjastoista on yleiskäyttöisiä ja kuuluvat siksi useaan eri ratkaisuun (solution). Tämä tarkoittaa, että eri ratkaisut (solution) jakavat samaa koodipohjaa eli lähdekoodia näiden VS -projektien osalta. Tästä seurasi, että osa kehityksestä tehtiin VS -projekteihin, jotka jakavat koodipohjaa eri ratkaisujen (solution) välillä. Tämän seurauksena AT -järjestelmän ydintoiminnallisuus pystytään ottamaan käyttöön yrityksen eri sovelluksissa. Edellä mainittujen asioiden pohjalta ohjelmistokehityksessä korostui koodin modulaarisuus, joten koodin abstrahointi oli ensisijaisen tärkeää.

Keskeisimmiksi kehitysprosessin kannalta AT -järjestelmää kehittäessä nousi muutama luokkakirjasto. Tietokerroksessä (Data Layer) oleva luokkakirjasto, joka hoitaa yhteyden .NET:n ja tietokannan välillä käyttäen EF:ä. Tämän luokkakirjaston tietokantakontekstia vasten testattiin AT -järjestelmän tietomuutostapahtumien seuranta ja tallennus erilliseen AT -tietokantaan. Tärkeä oli myös palvelukerroksessa (Service Layer) oleva luokkakirjasto, joka sisälsi business logiikkaa ja toimi samalla API -kerroksena. Tähän luokkakirjastoon kehitettiin AT -tietoja varten tiedonhakumoduuli. Näiden lisäksi oli ASP.NET MVC -luokkakirjasto, johon kehitettiin käyttöliittymä AT -tietojenhakua varten. Lopuksi oli myös luokkakirjasto, johon kehitettiin AT -tietokantakonteksti ja siihen kuuluvat entiteetit, sekä businesslogiikka. Alla olevassa kuvassa (KUVA 3.) esitellään asiakkaan ympäristöä.

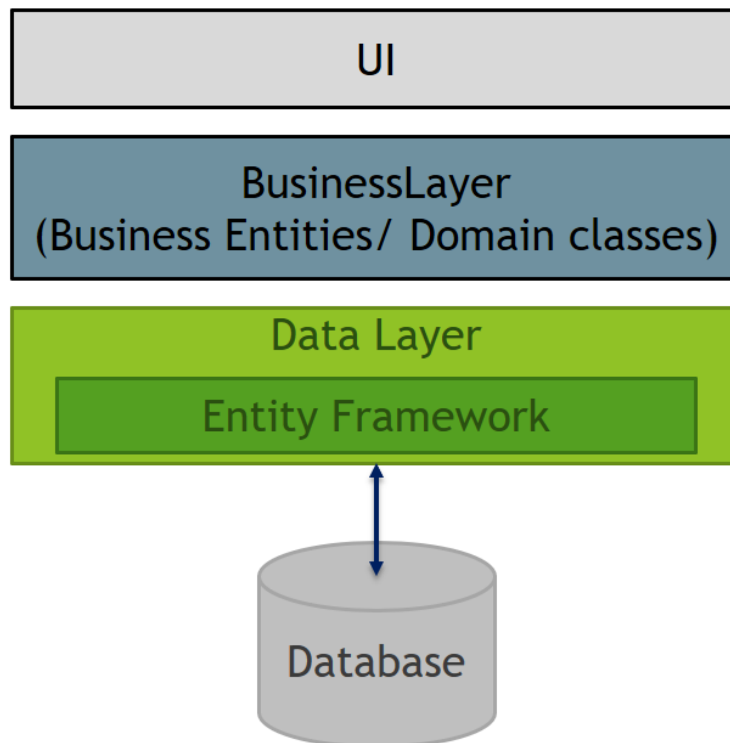


KUVA 3. Asiakkaan järjestelmäympäristö. (Solteq, inWorks-koulutusmateriaali, Tauno Hyvärinen, 2018)

Oleellista AT -järjestelmän kehityksessä on seurata tietokantaan lisättäviä, muutettavia ja poistettavia tietueita. Tässä voidaan onnistua, jos pääsemme seuraamaan tietokantaan tehtäviä tapahtumia, jotka muuttavat tietokannan taulujen sisältöä. Johtuen asiakkaan järjestelmästä, joka toimi .NET Framework 4.5 kehitysympäristössä, heille on ollut luontaista käyttää Entity Framework 6:a eli EF 6, jonka ohjelmistokehyksen avulla voimme seurata muutoksia .NET -ympäristön sisällä. Edellä mainittujen tietokannan tietueiden muutoksenseurannan lisäksi, AT -järjestelmän täytyisi pystyä näyttämään kuka, milloin ja mitä tietoja on päässyt näkemään. Tämä ominaisuus jäi kuitenkin opinnäytetyön ulkopuolelle, mutta siitä mainitaan osiossa **Kehitysideat ja Jatkokehitystarpeet**.

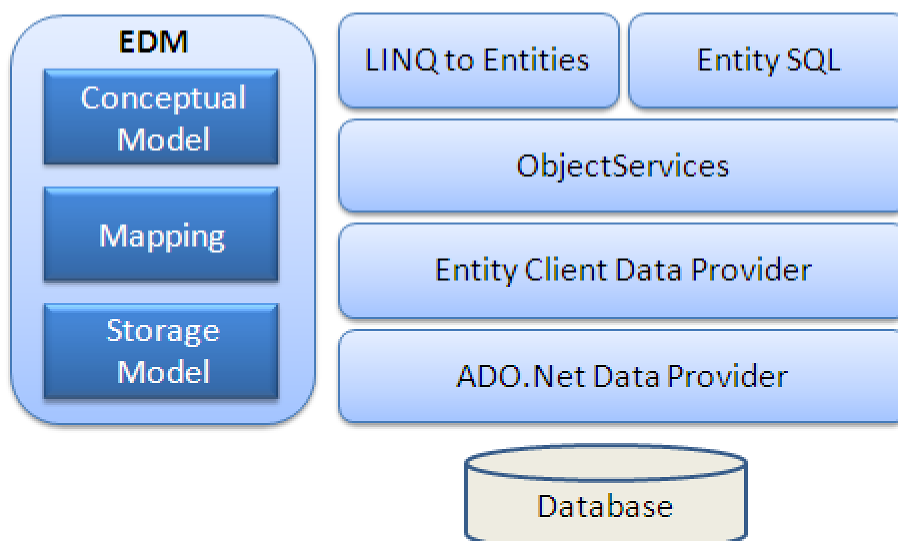
5.1 Entity Framework (EF)

EF on **ADO.NET**:ssä oleva joukko teknologioita, jotka tukevat tieto-orientoituvien ohjelmistosovellusten kehittämistä. EF vapauttaa kehittäjät tarpeesta huolehtia taustalla olevista tietokantojen tauluista ja sarakkeista, koska EF abstrahoi tietokantayhteyden ja tietokannan taulujen, sekä niiden sisältämien tietojen käsittelyn mahdollistamalla objektien ja propertyjen käytön, kuten *Customers* ja *CustomerAddresses*. EF:n avulla kehittäjät voivat työskennellä korkealla abstraktion tasolla käsitellessään tietoa ja voivat luoda ja säilyttää tieto-orientoituvia sovelluksia vähemmällä koodin määrällä verrattuna tavanomaisiin sovelluksiin. (Microsoft, Microsoft Docs, Entity Framework Overview, 2017)



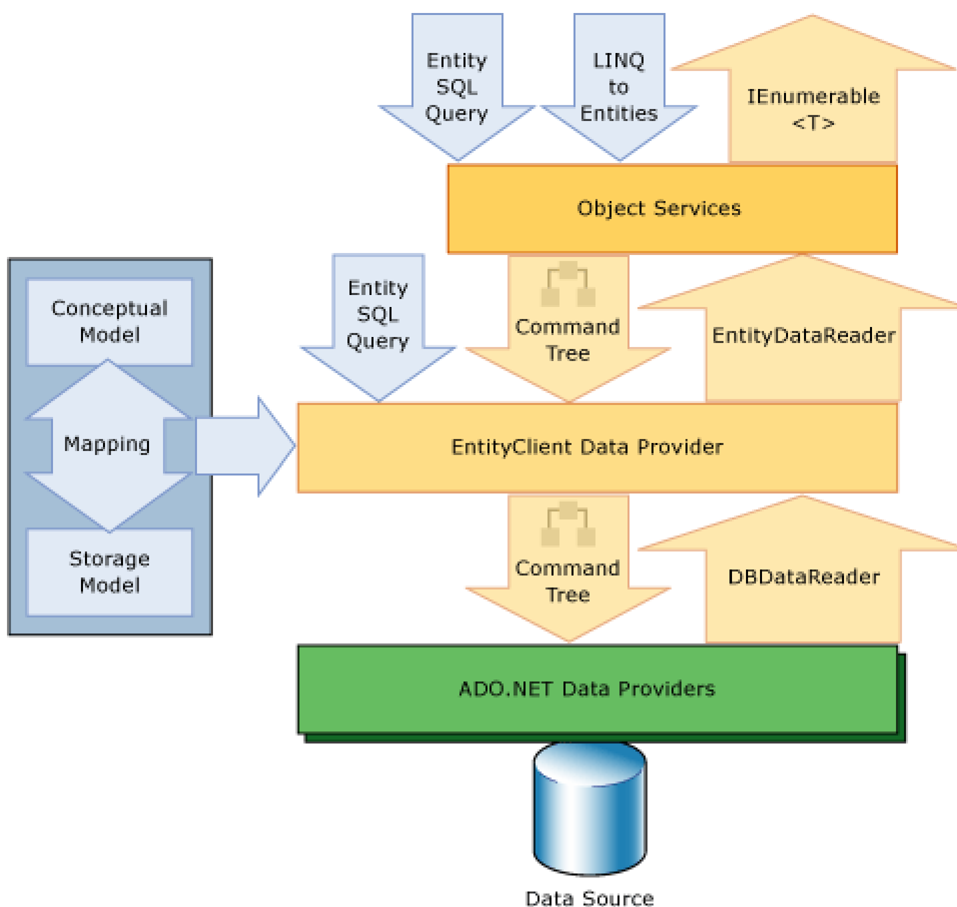
KUVA 4. EF kuuluu tietokerrokseen (Data Layer).

Edellä mainitun lisäksi Microsoft Developer Network eli MSDN antaa seuraavanlaisen selityksen EF:stä: EF on Microsoft .NET alustalle Microsoftin suosittelema objektirelaatiokartoittaja tai olio-relaatiokonvertteri, joka tarjoaa ohjelmistokehyksen objektiorientoituvasta toimialuemallista perinteiseen relaatiotietokantaan. EF:n käyttö vähentää ohjelmistokehittäjien tarvetta kirjoittaa tiedonhakukoodia eli tässä tapauksessa Structured Query Languagea eli SQL. (Microsoft, MSDN, Introduction to Entity Framework, 2017)



KUVA 5. EF:n rakenteellinen arkkitehtuuri. (Entity Framework Tutorial, Entity Framework Architecture, 2018)

Ohjelmistokehyksen avulla voidaan liittää .NET -kontekstin olioita kartoitettuun relaatiotietokantaan. EF mahdollistaa tietokantaan suoritettavien kyselyistä aiheutuvien tietuemuutosten seurannan, näyttäen muutokset olioissa .NET -kontekstin sisällä. Nämä oliot tai toiselta nimeltään entiteetit on liitetty tietokannan tauluihin ohjelmistokehyksen tarjoaman toiminnallisuuden toimesta. EF voidaan jakaa seitsemään kokonaisuuteen: **EDM**, **LINQ to Entities** (L2E), **Entity SQL**, **ObjectServices**, **Entity Client Data Provider** ja **ADO.NET Data Provider**.



KUVA 6. Diagrammi kuvastaa tiedonhakulogiikkaa EF:n sisällä (Microsoft, Microsoft Docs, Entity Framework Overview, 2017)

LINQ to Entities (L2E) tarjoaa integroidun kyselykielituen (Language-Integrated Queries (LINQ) support), joka tarjoaa kehittäjille mahdollisuuden kirjoittaa kyselyitä EF:n käsitemalleja (Conseptual Model) vasten, tässä tapauksessa käyttäen C#:a. Kyselyt EF:ä vasten on esitetty komentopuukyselyinä (Command Tree Queries), jotka suoritetaan objektikontekstia vasten. **L2E** kääntää LINQ-kyselyt komentopuukyselyiksi ja suorittaa kyselyt EF:ä vasten, palauttaen objekteja, jotka ovat sekä EF:n ja LINQ:n käytettävissä. (Microsoft, Microsoft Docs, LINQ to Entities, 2017)

Entity SQL on SQL:n tapainen kieli, joka mahdollistaa konseptimallien (Conseptual models) kyselemisen EF:ssä. Entity SQL on vaihtoehtoinen kyselykieli **L2E**:n lisäksi. Konseptimallit kuvaavat tietoa entiteetteinä ja relaatioina. EF toimii varastointispesifin tietopalvelutarjoajan (Data Provider) kanssa kääntäessään geneerisen Entity SQL:n varastointispesifiksi kyselyksi. EntityClient provider tarjoaa tavan suorittaa Entity SQL -komentoja entiteettimalleja vasten palauttaen rikkaita tietotyyppejä (rich types of data) sisältäen skalaariset tulokset, tulosjoukot ja objektigraafit. (Microsoft, Microsoft Docs, Entity SQL Overview, 2017)

Object Service on tärkein sisääntulopiste haettaessa tietoja tietokannasta ja palauttaessaan ne takaisin. **Object Service** on vastuussa materialisoinnista, mikä on prosessi, jossa **Entity Client Data Provider** -kerrokselta palautettu tieto muunnetaan entiteettiobjektistruktuureiksi (entity object structure). (Klein, 2010)

EF sisältää **Entity Client Data Providerin**. Tämä provider hoitaa yhteyksiä, muuntaa entiteettikyselyt tietolähdespesifeiksi kyselyiksi (data source-specific queries) ja palauttaa lukijan (reader), jota EF käyttää materialisoidessaan entiteettitietoa objekteiksi. Ellei objektien materialisointi ole vaatimuksena, **Entity Client Provideria** voidaan käyttää myös tavallisen ADO.NET tietopalvelutarjoajan (data provider) tavoin sallimalla sovelluksen Entity SQL-kyselyiden suorittaminen ja käyttämään palautunutta read-only-lukijaa (read-only data reader). (Microsoft, Microsoft Docs, Entity SQL Overview, 2017)

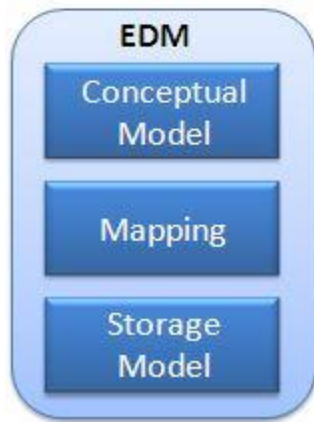
Tärkein tehtävä **Entity Client Provider** kerroksella on muuntaa L2E- tai Entity SQL-kyselyt tietokantaspesifiksikyselyksi esim. SQL. **Entity Client Provider** kommunikoi ADO.NET tietopalvelutarjoajan (data provider) kanssa, joka puolestaan lähettää tai palauttaa tietoa taustalla olevasta tietokannasta. (Klein, 2010)

ADO.NET tarjoaa johdonmukaisen tavan muodostaa yhteyden tietolähteisiin, kuten SQL Serveriin, XML-tietorakenteisiin, sekä muihin tietolähteisiin käyttäen OLE DB ja ODBC:tä. Tietoa jakavat kuluttajasovellukset voivat käyttää ADO.NET:ä yhdistäessään näihin tietolähteisiin. ADO.NET mahdollistaa tietolähteiden sisältämän tiedon palauttamisen, käsittelemisen ja päivittämisen. ADO.NET jakaa tietoyhteyden (data access) tietoa manipuloivasta tahosta erillisiin komponentteihinsa, jotta niitä voidaan käyttää erikseen tai yhdessä. ADO.NET sisältää .NET Frameworkin tietopalvelutarjoajat (data providers). (Microsoft, Microsoft Docs, ADO.NET Overview, 2017)

.NET Framework Data Provideria käytetään tietokantoihin yhdistämiseen, komentojen suorittamiseen ja tuloksien palauttamiseen. Tulokset prosessoidaan suoraan tai ne asetetaan DataSet:n, jotta ne olisivat näkyvillä käyttäjän tarvitsemassa muodossa. Tieto voi olla yhdistettynä useammasta lähteestä tai säädetty eri tasojen välille. (Microsoft, Microsoft Docs, .NET Framework Data Providers, 2017)

5.1.1 Entity Data Model (EDM), Plain Old CLR Object (POCO) ja tietokantakyselyt

EF luo Entity Data Model:ta (EDM) eli entiteettitietomalleja. **EDM** on koko metadatan muistikuvaus, joka koostuu **konseptimallista** (Conseptual Model), **varastointimallista** (Storage Model) ja niiden välisestä liitoksesta (Mapping). Alla olevassa kuvassa (KUVA 7.) on esitelty EDM:n tietorakenne.



KUVA 7. Esitellään EDM:n tietorakenne. (Entity Framework Tutorial, Architecture, 2018)

Konseptimalli rakentuu .NET -kontekstin luokista, konseptiluokista ja yleisistä luokkien ja konfiguraatioiden säännöistä. Konseptimalli on erillinen varastointimallista. **Varastointimalli** rakentuu taustalla olevan tietokannan skeemasta. Se on tietokantakuvausmalli, joka sisältää taulut, näkymät, proseduurit, relaatiot ja avaimet.

EF:ssä olevat **entiteetit** ovat sovellusympäristön luokkia, jotka liitetään alla olevaan tietokantakontekstiluokkaan (**DbContext**). EF:ssä on kahta erilaista entiteettityyppiä: POCO-entiteetit (POCO) ja Dynamic proxy entiteetit (POCO Proxy).

POCO-entiteetit ovat luokkia ilman riippuvuutta .NET -kontekstin kantaluokkiin. Ne ovat kuin muutkin **.NET CLR luokat**, jonka seurauksena niitä kutsutaan nimellä ”Plain Old CLR Objects”. Nämä entiteetit tukevat suurinta osaa samoista kyselyistä kuin entiteettityypit, jotka on generoitu käyttäen **EDM**:ää. (Klein, 2010) Alla olevassa kuvassa (KUVA 8.) esitellään **POCO** -entiteetti, joka vastaa kaikkia **POCO Proxylle** asetettuja vaatimuksia, toteuttaen Lazy Loading-ja muutostenseurausominaisuudet.

```
public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public DateTime? DateOfBirth { get; set; }
    public byte[]  Photo { get; set; }
    public decimal Height { get; set; }
    public float Weight { get; set; }

    public StudentAddress StudentAddress { get; set; }
    public Grade Grade { get; set; }
}
```

KUVA 8. Esitellään **POCO** -entiteetti.

Dynamic Proxy Entiteetit (**POCO Proxy**) on ajonaikainen proxy-luokka, joka kietoo POCO-entiteetin itseensä. EF luo **POCO**-entiteeteille Proxyjä, jos **POCO** vastaa Proxylle asetettuja vaatimuksia. **POCO Proxy**:llä voi olla Lazy Loading -ominaisuus, joka mahdollistaa entiteettien ja kokoelman entiteettejä latautuvan automaattisesti tietokannasta, kun ensimmäisen kerran käsitellään viitatun entiteetin propertyä (Microsoft, MSDN, Entity Framework Loading Related Entities, 2017). Tämän lisäksi **POCO**:lla voi olla muutostenseurausominaisuus (Change tracking).

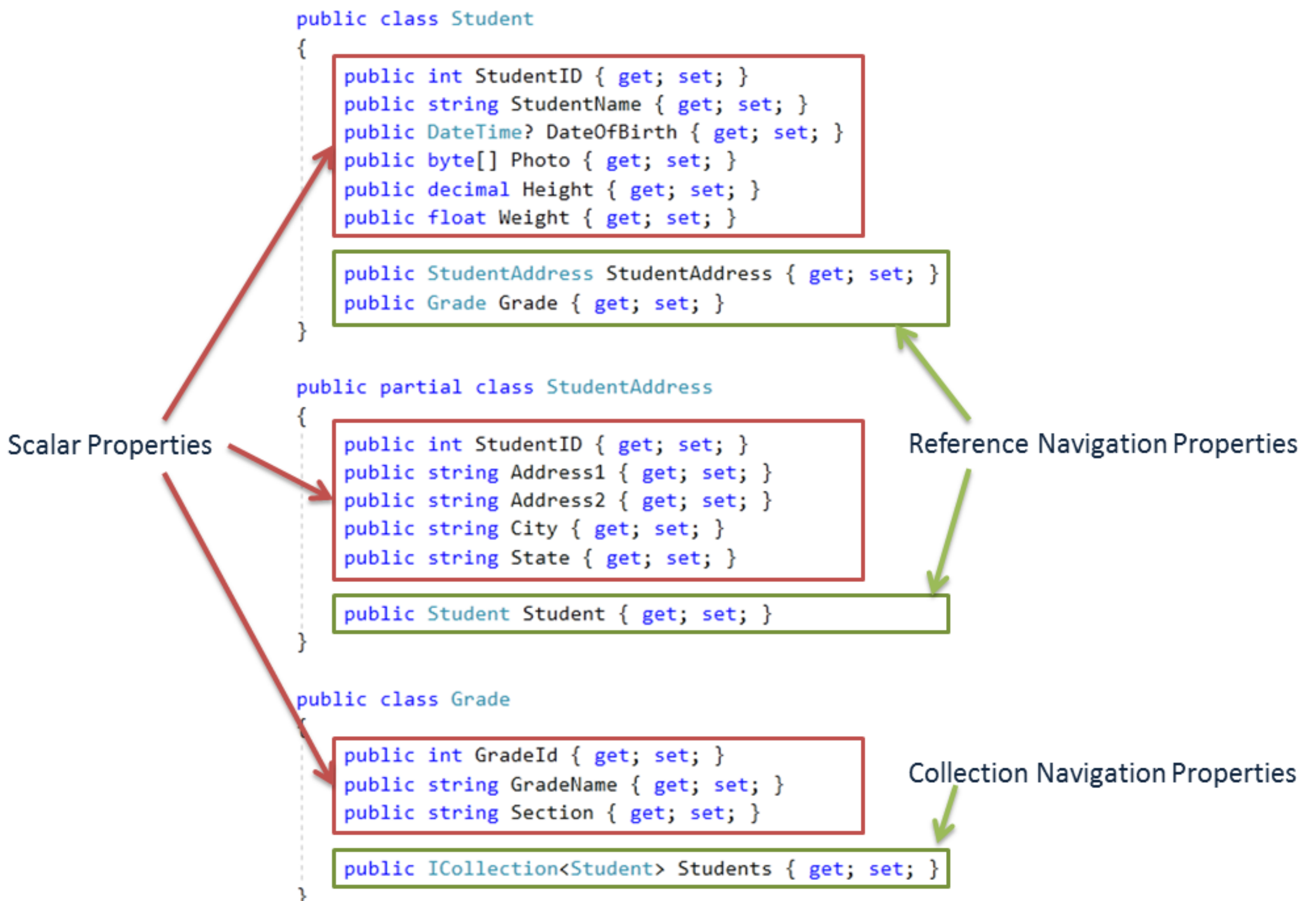
Jos **POCO** vastaa muutostenseurausominaisuudelle asetettuja vaatimuksia, sille luodaan Lazy Loading Proxy samalla kertaa. Lazy Loading -ominaisuus voi olla ilman, että se vastaa muutostenseurausominaisuudelle asetettuja vaatimuksia. Molemmissa tapauksissa **POCO**:n täytyy vastata seuraaviin vaatimuksiin, jotta se voi saada Proxyn. Luokalle täytyy määritellä **Public**-pääsyoikeus, luokka ei saa olla **Sealed** tai **abstract**, luokalla pitää olla **Public** tai **Protected** konstruktori, luokka ei voi toteuttaa (implement) IEntityWithChangeTracker- tai IEntityWithRelationships-rajapintoja, koska **POCO Proxy** toteuttaa (implement) kyseiset rajapinnat. Tämän lisäksi ProxyCreationEnabled asetuksen täytyy olla arvossa **true**. (Microsoft, MSDN, Requirement for Creating POCO Proxies, 2018)

Jotta Lazy loading saadaan käyttöön, jokaisen navigaatio propertyn (Navigation Property) GET-aksessori täytyy olla **Public**, **Virtual**, eikä **Sealed**. Muutostenseurausominaisuus vaatii lisäksi, että jokainen entiteetin kartoitettu (mapped) property entiteettimallissa täytyy olla julkinen eli **Public** tai **Virtual** GET-aksessori. Jokainen navigaatio property, joka kuvaa useaa relaatiota (esim. listat) täytyy palauttaa tyyppi, joka toteuttaa (implement) ICollection-rajapinnan. Tämän kokoelman geneerinentyyppi T on tyypitetty. Tämän tyypitetyn ICollection-rajapinnan tyyppi kuvastaa relaation toisessa päässä olevan objektin tyyppiä. (Microsoft, MSDN, Requirement for Creating POCO Proxies, 2018) Alla olevassa esimerkissä navigaatio property toteuttaa (implement) ICollention-rajapinnan ja sen geneerinen tyyppi T on tyypitetty Student-entiteetiksi.

```
Public ICollection<Student> Students { get; set; }
```

EF API liittää jokaisen entiteetin tauluun ja luo jokaiselle propertylle vastaavan sarakkeen kyseiseen tauluun. Entiteeteillä voi olla skalaarisia propertyjä (Scalar Property), jotka ovat primitiivityyppisiä ja varastoivat todellista tietoa. Tämän lisäksi entiteeteillä voi olla navigaatio propertyjä (Navigation Property).

Navigaatio propertyt kuvaavat relaatioita toisiin entiteetteihin ja eivät itsessään varastoi mitään arvoa. Navigaatio propertyjä on kahdenlaisia: referenssi navigaatio propertyt (Reference Navigation Property), jotka viittaavat yhteen eli yhden suhde yhteen ja kokoelma navigaatio propertyt (Collection Navigation Property), jotka voivat viitata useampaan eli yhden suhde moneen.



KUVA 9. Kuvassa esitellään entiteetin eri propertyjä. (Entity Framework Tutorial, 2018)

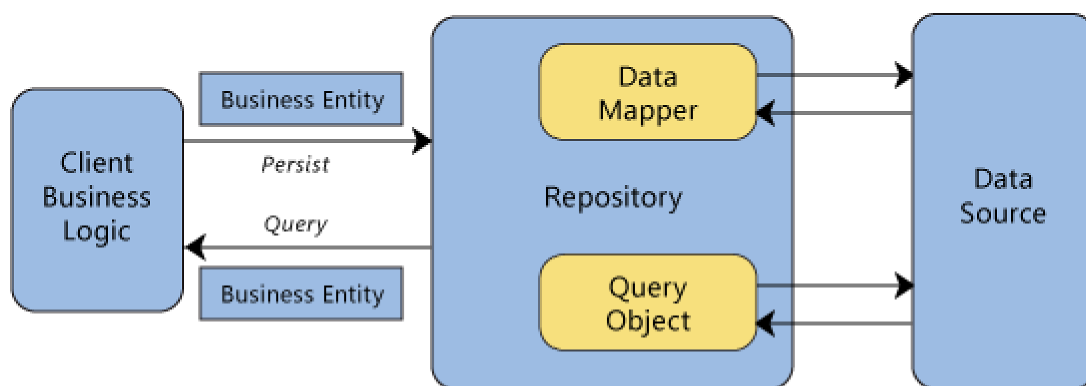
EF mahdollistaa LINQ -kyselyiden käytön, jotka mahdollistavat tiedon hakemisen taustalla olevasta tietokannasta. Tietokanta toimittaja (provider) muuntaa LINQ -kyselyt tietokanta spesifiksi kyselykieleksi esimerkiksi Sequence Query Language eli SQL. EF mahdollistaa näin karkeiden SQL-kyselyiden ajamisen suoraan tietokantaan. (Lerman, 2010)

EF suorittaa myös automaattisen tietokantapahtumakäsittelyn (Automatic Transaction Management), tallennettaessa tai kysellessään tietoja. Käsittelyä on mahdollista tarvittaessa muokata. EF sisältää myös ensimmäisen asteen kätkön (cache), jossa peräkkäiset kyselyt palauttavat tietoa kätköstä (cache) tietokannan sijaan. (Klein, 2010)

5.1.2 Muutostenseuraus, tiedontallennus ja DbContext

Tärkeänä ominaisuutena tiedonmuutosten seuraamisessa nousi EF:n sisällä oleva tietomuutostenseurausominaisuus (Change tracking). Tämä ominaisuus (feature) seuraa arvojen (value) muutoksia entiteetti-ilmentymien propertyissä. EF pitää kirjaa muutoksista, jotka pitää tehdä taustalla olevaan tietokantaan, varmistaen näin tietojen oikeellisuuden. (Klein, 2010)

EF suorittaa INSERT, UPDATE ja DELETE -komentoja tietokantaan, perustuen muutostenseuraajan (ChangeTracker) kirjaamiin tietomuutoksiin entiteeteissä. Nämä tietomuutokset tallentuvat kutsuttaessa **SaveChanges()** -metodia. (Klein, 2010) **SaveChanges() -**metodi kuuluu **DbContext**-luokkaan ja se täytyy olla kaikissa luokissa, jotka perivät **DbContext**-luokan. Alla olevassa kuvassa (KUVA 10.) esitellään miten säilytyspaikkasuunnitteluperiaate (Repository Pattern) eriyttää businesslogiikkakerroksen tietolähteestä (Data Source). Periaate liittää tietolähdekerroksen (data source layer) eli tässä tapauksessa tietokannassa olevan tiedon entiteetteihin, joita voidaan käyttää businesslogiikkakerroksessa. Periaate parantaa koodin ylläpidettävyyttä ja lukua. Periaate myös lisää koodin määrää, mitä on mahdollista automaatio- ja yksikkötestata. (Microsoft, MSDN, The Repository Pattern, 2018)



KUVA 10. Säilytyspaikkasuunnitteluperiaate (Repository Pattern) eriyttää businesslogiikkakerroksen tietolähdekerroksesta. (Microsoft, MSDN, The Repository Pattern, 2018)

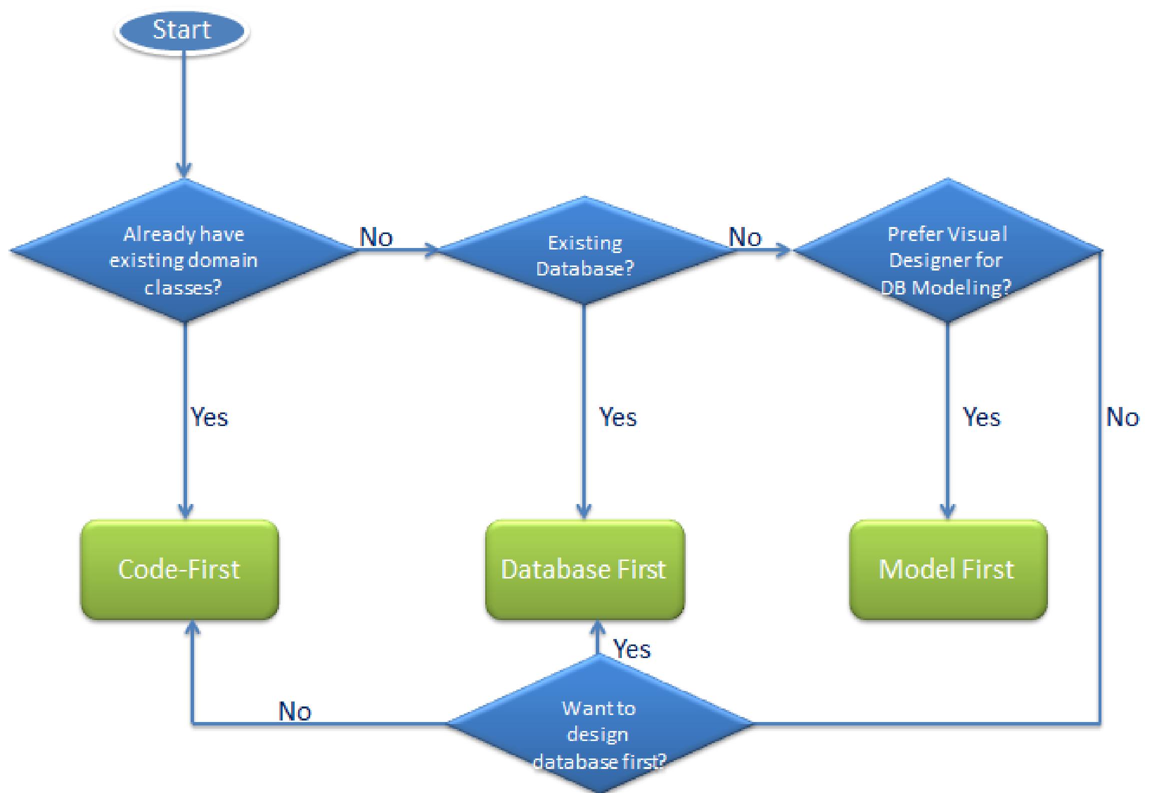
DbContext-ilmentymä edustaa kombinaatiota Työn yksikkö- ja säilytyspaikkasuunnitteluperiaatteista (Unit of Work and Repository Pattern). Työn yksikkösuunnitteluperiaate säilyttää muutostiedot muistissa, sekä lopuksi lähettää muistissa olevat muutostiedot tietokantaan yhdellä tietokantatapahtumalla (transaction). DbContext-ilmentymää voidaan käyttää tietokantakyselyissä. Koska ilmentymä toteuttaa Työn yksikkösuunnitteluperiaatteen, mahdollistaa se kirjattujen muutosten kokoamisen

yhteen yksikköön (one unit), jonka jälkeen kaikki tehdyt muutokset entiteetteihin voidaan tallentaa yhden tietokantatapahtuman (transaction) aikana. (Microsoft, MSDN, DbContext Class (System.Data.Entity), 2018) Luokka pitää sisällään myös yhteysmerkkijonon (connectionString), joka sisältää tarvittavan tiedon tietokantayhteyden muodostamiseksi.

SaveChanges() -metodi on **DbContext**-luokassa. **DbContextin** sisällä on mahdollista tarkastella koottuja tietomuutoksia entiteetti-ilmentymistä. Kaikki tietokantamuutokset, jotka tehdään käyttäen EF:ä kulkevat edellä mainitun metodin läpi (pois lukien asynkroninen vastine). Ajoympäristön kyseisessä kontekstissa on mahdollista käsitellä tietoja **EDM** -tietomalleina. Tästä johtuen tietomuutostenseuraus oli helppoa tehdä tämän metodin sisään. **SaveChanges() -**metodista on olemassa myös asynkroninen vastine **SaveChangesAsync()**.

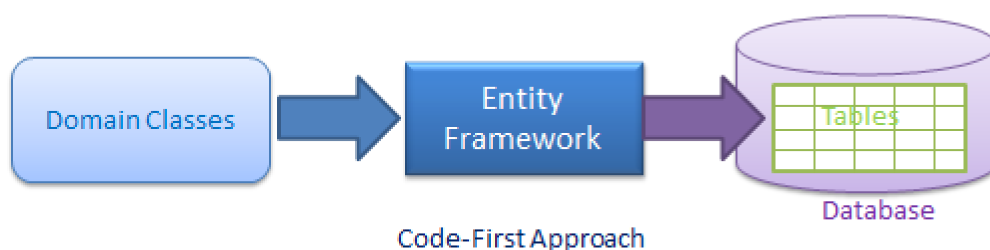
5.1.3 Code First ja migraatiot

EF 6.0:sa on mahdollista valita kolmesta kehitystavasta, joilla **objektirelaatiokartoittaja** luodaan. **Tietokanta ensin lähestymistapa** (Database-First Approach), jossa ensin luodaan konteksti ja entiteetit olemassa olevalle tietokannalle käyttäen EDM Wizard:ia. Wizard on integroitu Visual Studioon (VS). **Malli ensin lähestymistavassa** (Model-First Approach) luodaan ensin entiteetit, relaatiot ja periytyvyydet käyttäen VS:n integroitua visuaalista piirtäjää, mikä lopulta generoi entiteetit, konseptiluokat ja tietokantaskriptit visuaalisen mallin pohjalta. (Entity Framework Tutorial, 2018) Kehitystyön kannalta oleellinen lähestymistapa oli kuitenkin **koodi ensin lähestymistapa** (Code-First Approach), koska sitä käytetään yrityksen järjestelmissä.



KUVA 11. Vuokaavio kolmesta EF:n kehityksessä käytettävistä lähestymistavoista. (Entity Framework Tutorial, 2018)

Koodi ensin -lähestymistavassa ensin luodaan EF -kontekstin entiteetit ja kontekstiluokka. Näiden luokkien pohjalta luodaan tietokanta käyttäen migraatiokomentoja. (Lerman, 2010) Migraatio sisältää tiedon tietokannan rakenteellisesta muutoksesta. Koodi ensin -lähestymistavassa migraatiot luovat tietokannan rakenteellisen muutoshistorian sen luonnista nykyhetkeen. Yksi migraatio voisi sisältää tiedon uuden taulun lisäämisestä tietokantaan tai sarakemuutoksesta tietokannan taulussa. Koodi ensin on hyvä lähestymistapa, jos järjestelmässä on olemassa olevia luokkia, koska muutokset saadaan tietokannan tasolle helposti migraatiokomentojen avulla. Kehitystyössä seurattiin **ympäristö painotteisia suunnitteluperiaatteita** (Domain-Driven Design (**DDD**) principles), joita koodi ensin lähestymistapa tukee. (Nilsson, 2006)



KUVA 12. Kuvataan koodi ensin lähestymistavan kehitysjärjestystä. (Entity Framework Tutorial, 2018)

Koodi ensin -lähestymistavassa kehitystyö aloitetaan yleensä kirjoittamalla .NET ohjelmistokehyksen luokat, jotka määrittävät **EDM:n konseptimallin**. **Konseptimallin POCO**-luokat täytyy

määrittää **DbContext**-luokassa, jotta tiedetään tyypit, jotka liitetään EDM-malliin. Tämän saavuttamiseksi täytyy määritelle kontekstiluokka, joka periytyy **DbContext**:sta ja paljastaa **DbSet-**propertyt tyypeille, jotka halutaan sisällyttää konseptimalliin. **Koodi ensin** sisällyttää nämä tyypit ja kaikki referenssityypit, vaikka referenssityypit olisi määritetty eri ohjelmistokirjastossa. (Microsoft, MSDN, Entity Framework Code First Conventions, 2016) Alla olevassa kuvassa (KUVA 13.) esitellään **AuditTrailDbContext** ilmentymä, jossa on DbSet-propertyt tyypitettynä **AuditTrailTransaction**, **AuditTrailBody** ja **AuditTrailContent**.

```
public class AuditTrailDbContext : DbContext, IAuditTrailDataContext
{
    private const string DefaultNameOrConnectionString = "AuditTrailDataBase";
    private static readonly ILog Logger = LogManager.GetLogger<AuditTrailDbContext>();

    public AuditTrailDbContext()
        : base(GetConnectionStringName())
    {
    }
    private static string GetConnectionStringName()
    {
         var connectionStringName = ConfigurationManager.AppSettings["AuditTrailConnectionStringName"];
        if (string.IsNullOrWhiteSpace(connectionStringName))
            connectionStringName = DefaultNameOrConnectionString;
        return connectionStringName;
    }

    //
    // Summary:
    //     Constructs a new context instance using the given string as the name or connection
    //     string for the database to which a connection will be made. See the BASEclass remarks
    //     for how this is used to create a connection.
    //
    // Parameters:
    //   nameOrConnectionString:
    //     Either the database name or a connection string.
    public AuditTrailDbContext(string nameOrConnectionString)
      : base(nameOrConnectionString)
    { }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.HasDefaultSchema("AuditTrail");

    }

    public static AuditTrailDbContext Create()
    {
        return new AuditTrailDbContext();
    }
    public DbSet<AuditTrailTransaction> AuditTrailTransactions { get; set; }
    public DbSet<AuditTrailBody> AuditTrailBodys { get; set; }
    public DbSet<AuditTrailContent> AuditTrailContents { get; set; }
}
```

KUVA 13. Esitellään **AuditTrailDbContext** ilmentymä.

Koodi ensin lähestymistapaan olennaisena osana kuuluvat tietokantamigraatiot. Migraatiot tarjoavat tavan tehdä tietokantamuutoksia vähitellen, pitäen EF:n entiteetti mallit synkronoituina säilyttäen olemassa olevan tiedon tietokannassa (Microsoft, Microsoft Docs, Migrations - EF Core, 2017). **Migraatiot** generoidaan **DbContext** ilmentymään liitettyjen konseptimallien pohjalta (**DbSet**-propertyt, jotka on tyypitetty luokkatyyppisiksi olioiksi). Migraatioiden avulla ohjelmistokehittäjä voi tehdä muutoksia konseptimalleihin ja päivittää **DbContextiin** liitetyn tietokannan niiden mukaiseksi, käyttäen migraatiokomentoja. Kehitettävän järjestelmän jatkuvan kehittymisen vuoksi, migraatioita

on tullut ja tulee jatkuvasti lisää. Uusien migraatioiden johdosto, kaikkien järjestelmää käyttävien tahojen täytyy päivittää tietokannat ajan tasalle versiopäivitysten yhteydessä.

AT -tietokannan lyhyen kehittämisen ja testauksen johdosta, voi olla, että siihenkin tulee muutoksia konseptimalliin. Tästä seuraa uusia migraatioita. Välttääkseen migraatiomuutosten tarpeet seurattavassa tietokannassa, AT -järjestelmä haluttiin toteuttaa omaan skeemaansa. Tämän johdosta muutokset, jotka vaikuttavat vain AT -tietokantaan, voidaan päivittää migraatioilla, jotka eivät vaikuta millään tavalla seurattavaan tietokantaan. Eli tällä tavalla vältetään ylimääräiset migraatiot seurattavassa tietokannassa.

6 TIEDONTALLENNUS

Tiedontallennusominaisuus liitettiin EF:n sisään. Entiteettitiedonmuutoksia seurattiin ChangeTrackerin avulla. ChangeTracker pystyi seuraamaan POCO Proxy entiteettien tietojen muutoksia DbContextin sisällä. DbContextin sisällä oleva ChangeTracker piti sisällään kootut tiedot muutoksista. Nämä muutokset muunnettiin ja tallennettiin vaiheistetusti AT -tietokantaan. AT -taulut kehitettiin omaan skeemaansa eli niillä oli oma DbContext luokka, jonka SaveChanges() -metodia kutsuttiin seurattavan kontekstin (DbContext) aloittamien kutsujen pohjalta. AT -kontekstille välitettiin tieto, joka prosessoitiin matkalla muotoon, jota kontekstin entiteetit tukivat. AT -konteksti tallensi muutokset taustalla olevaan AT -tietokantaan.

6.1 Entiteettien ja taulujen määritys

Muutostietojen tallennuksen kehityksessä yksi tärkeimmistä tehtävistä oli suunnitella Audit Trail entiteetit ja niitä vastaavat taulut. Tauluihin pitäisi pystyä tallentamaan kaikenlaista tietoa, sekä tiedot pitäisi olla sellaisessa muodossa, että ne olisi helppo hakea tarvittaessa. Tietojen tallennuksessa ei siis saisi olla rajoitteita mihinkään tiettyyn tietotyyppiin ja tietojen tarkastelu täytyisi olla mahdollista, vaikka tiedot eivät olisi alkuperäisessä tietotyypissä. Lopulta päädyttiin kolmen taulun ratkaisuun.

Taulujen suunnittelussa mietittiin erilaisia vaihtoehtoja taulujen rakenteeksi. Yksi mahdollinen oli, että kaikista seurattavan tietokannan tauluista luotaisiin erilliset AT -taulut, mutta tämän ratkaisun ongelmaksi nousisi jatkuvat ylläpidettävyys tarpeet. Migraation seurauksena tehdyt muutokset seurattavaan tietokantaan, täytyisi tehdä myös vastaavaan AT -tauluun. Tämä loisi turhaa lisätyötä ja AT -tietokanta olisi aina erilainen riippuen seurattavasta tietokannasta.

Koska ei ollut hyvä vaihtoehto luoda erillisiä AT -tauluja jokaisesta seurattavan tietokannan taulusta, ainoaksi vaihtoehdoksi jäi, että AT -tauluja olisi vakio määrä. Tässä ongelmaksi nousi vain se, että minkälaiset tietorakenteet tauluilla tulisi olemaan, jotta niihin voitaisiin tallentaa kaikenlaista tietoa geneeriseen muotoon. Tiedon tulisi olla myös helposti tarkasteltavissa ja haettavissa. Tallennukseen pitäisi sisällyttää ainakin seuraavat tiedot: **muutoksentekijä**, **muutosaika**, **muutostyyppi** eli

mikä operaatio tietoon tehtiin, miten tieto muuttui eli sarake **vanhalle** ja **uudelle arvolle** ja tiedon **yksilöivä tunniste**.

Saraketasonmuutos tieto täytyisi tallentaa muodossa, joka olisi helposti luettavissa, tästä johtuen arvojen muutoksia ei voinut tallentaa binäärityyppisenä. Tässä ei ollut myöskään järkevää käyttää useampaa saraketta kaikille mahdollisille tietotyypeille, koska se olisi turhaan kasvattanut taulujen kokoa. Riveille jäisi turhaan tyhjiä sarakkeita ja uusien tietotyyppien ilmestyessä tauluihin täytyisi tehdä migraatiomuutoksia. Tästä johtuen arvojenmuutossarakkeet päädyttiin luomaan primitiivisessä tietotyypissä string, koska sen luettavuus ja tietotyyppimuutos alkuperäisestä stringiksi on yleisesti tuettu kaikissa muissa primitiivitason tietotyypeistä.

Koska taulujen luonnissa käytettiin EF:n koodi ensin metodiikkaa, taulut luotiin kirjoittamalla ensin toimialuekohtaiset entiteettiluokat ja DbContext-luokka. Entiteettiluokat eli konseptimallin toimialuekohtaisialuokkia tehtiin neljä **AuditTailTransaction**, **AuditTrailBody** ja **AuditTrailContent,** sekä **AuditTrailDbContext**, joka peri DbContext luokan. Näille konseptimallin luokille oli vastaavat varastointimallit, jotka liittivät toimialuekohtaiset luokat tietokannan tauluihin.

6.1.1 Konseptimallin luokat ja tietokannantaulut

Transaction-luokan (entiteetti) tarkoitus oli pitää sisällään tieto yhdestä seurattavan DbContext:n tekemästä tietokantatapahtumasta, sen tilasta ja toimia AT -taulujen perustana. Tässä taulussa oli vain juokseva id-sarake, tilatieto ja kokoelma navigaatio referenssi AuditTrailBody-tauluun. **Body-luokka** (entiteetti) piti sisällään tiedon, mihin tauluun muutos oli tehty. Luokassa myös kerrottiin muutoksen kellonaika, muutostyyppi, muutostaulussa olevan uniikin id:n tieto, muuttajantieto, juoksevan id-sarakkeen ja kokoelma navigaatio referenssin AuditTrailContent-luokkaan, sekä navigaatio referenssi AuditTrailTransaction-luokkaan. **Content-luokassa** (entiteetti) oli sarakekohtainen muutostieto. Luokka piti sisällään sarakkeen nimen, vanhan arvon ja uuden arvon, sekä juoksevan id-sarakkeen ja navigaatio referenssin AuditTrailBody-luokkaan. Kuvissa (KUVA 14, 15, 16) esitellään toteutettuja entiteettejä.

```
public class AuditTrailTransaction
{
    public AuditTrailTransaction()
    {
        Bodies = new List<AuditTrailBody>();
    }
    [Key]
    public long AuditTrailTransactionId { get; set; }
    public ApplicationContextError State { get; set; }
    [NotMapped]
    public string StateText { get { return State.GetDescription(); } }

    public virtual ICollection<AuditTrailBody> Bodies { get; set; }

}
```

KUVA 14. Esitellään **AuditTrailTransaction** konseptimallin luokkatoteutus.

```csharp
public class AuditTrailBody
{
    [Key]
    public long AuditTrailBodyId { get; set; }
    public DateTime Modified { get; set; }

    [MaxLength(200)]
    public string ModifiedTable { get; set; }

    public DbOperations OperationType { get; set; }
    [NotMapped]
    public string OperationTypeText { get { return OperationType.GetDescription(); } }

    [MaxLength(200)]
    public string Modifier { get; set; }

    public int TablesIdentifier { get; set; }

    public long AuditTrailTransactionId { get; set; }
    [ForeignKey("AuditTrailTransactionId")]
    public virtual AuditTrailTransaction AuditTrailTransaction { get; set; }
    public virtual ICollection<AuditTrailContent> Contents { get; set; }
}
```

KUVA 15. Esitellään **AuditTrailBody** konseptimallin luokkatoteutus.

```csharp
public class AuditTrailContent
{
    [Key]
    public long Id { get; set; }
    [MaxLength(500)]
    public string OldValue { get; set; }
    [MaxLength(500)]
    public string NewValue { get; set; }
    [MaxLength(200)]
    public string ColumnName { get; set; }
    public long AuditTrailBodyId { get; set; }
    [ForeignKey("AuditTrailBodyId")]
    public virtual AuditTrailBody AuditTrailBody { get; set; }


}
```

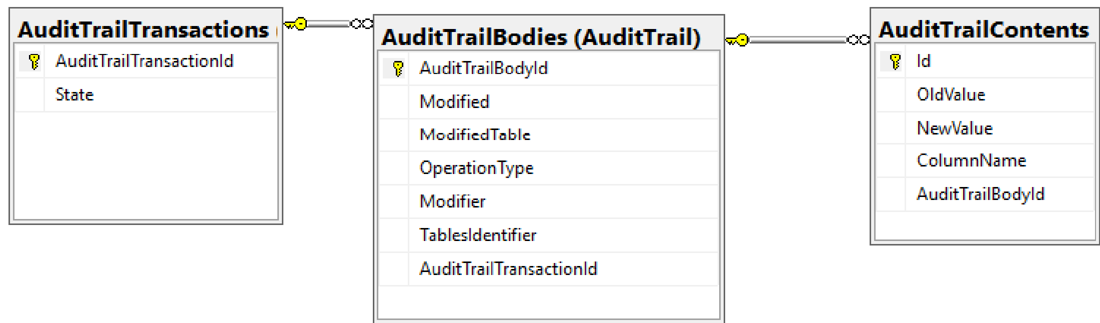
KUVA 16. Esitellään **AuditTrailContent** konseptimallin luokkatoteutus, joka EF:n DbContext:n yhteydessä on entiteetti.

Context-luokka tehtiin, jotta AT -tiedot saataisiin eriytettyä seurattavan tietokannan skeemasta ja vältettäisiin ylimääräiset tietokantamigraatiot seurattavassa tietokannassa. Oma skeema AT -tiedoille on lisäksi modulaarisempi ja kestävämpi ratkaisu, koska se mahdollistaa erilaisten tietokantojen muutostenseurauksen välttäen riippuvuudet yhteen järjestelmään (tai tietokantaan). Oma skeema mahdollistaa myös AT -skeeman sijaitsemisen eri tietokannassa kuin seurattavatietokanta.

```
public class AuditTrailDbContext : DbContext, IAuditTrailDataContext
{
    private const string DefaultNameOrConnectionString = "AuditTrailDataBase";
    private static readonly ILog Logger = LogManager.GetLogger<AuditTrailDbContext>();

    public AuditTrailDbContext()
        : base(GetConnectionStringName())
    {
    }
    private static string GetConnectionStringName()
    {
         var connectionStringName = ConfigurationManager.AppSettings["AuditTrailConnectionStringName"];
        if (string.IsNullOrWhiteSpace(connectionStringName))
            connectionStringName = DefaultNameOrConnectionString;
        return connectionStringName;
    }

    //
    // Summary:
    //     Constructs a new context instance using the given string as the name or connection
    //     string for the database to which a connection will be made. See the BASEclass remarks
    //     for how this is used to create a connection.
    //
    // Parameters:
    //   nameOrConnectionString:
    //     Either the database name or a connection string.
    public AuditTrailDbContext(string nameOrConnectionString)
      : base(nameOrConnectionString)
    { }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.HasDefaultSchema("AuditTrail");

    }

    public static AuditTrailDbContext Create()
    {
        return new AuditTrailDbContext();
    }
    public DbSet<AuditTrailTransaction> AuditTrailTransactions { get; set; }
    public DbSet<AuditTrailBody> AuditTrailBodys { get; set; }
    public DbSet<AuditTrailContent> AuditTrailContents { get; set; }
}
```

KUVA 17. Kuvassa esitellään AuditTrailDbContext ilmentymä

Konseptimallinluokkien pohjalta EF API generoi migraatioiden avulla vastaavanlaisen tietokannan skeeman. Skeema sisältää taulut, jotka vastaavat edellä mainittuja toimialuekohtaisia konseptimallinluokkia. Alla olevassa kuvassa (KUVA 18.) on esitetty toteutettua AT – tietokantaa, joka pohjautuu AT -entiteetteihin, koska kehityksessä käytettiin Koodi ensin lähestymistapaa.



KUVA 18. Audit Trail -skeeman Audit Trail -taulut esiteltynä.

6.2 Rajapintojen ja luokkien määritys

AT -järjestelmästä haluttiin tehdä mahdollisimman irrallinen kokonaisuus, jotta se olisi mahdollista ottaa käyttöön erilaisille tietokantakonteksteille. Tämän ja hyvien ohjelmointikäytänteiden mukaisesti toteutusluokista olisi hyvä olla aina rajapintamääritykset. Abstrahoinnin avulla vältetään riippuvuudet, jotka saattaisivat vaikeuttaa jatkokehitystä. Seurauksena tästä kehitetyt toteutusluokat pohjautuivat abstrakteihin määrittelyluokkiin.

Luotujen abstraktien määrittelyjen avulla voidaan käsitellä erilaisia abstraktien määrittelyjen toteuttavia toteutusluokkia. Abstraktin määrittelyn avulla tunnemme sen yleisen rajapinnan, jolla toteutusluokkaan voidaan olla yhteydessä. Abstrahoinnin avulla koodista saadaan modulaarisempaa, koska määrittelyn avulla voimme luoda erilaisia luokkia, jotka toteuttavat abstraktin määrittelyn. Tarvittaessa toteutusluokka voidaan vaihtaa, koska siihen ollaan yhteydessä abstraktin määrittelyn tarjoaman rajapinnan avulla.

AT -tietokantakontekstille luotiin rajapinta, joka määrittää, että sen täytyy sisältää DbSet<TEntity> -tyyppiset propertyt, jossa määritellään **AuditTrailTransaction**, **AuditTrailBodies** ja **AuditTrail-Contents**. Tämä rajapinta olisi tarkoitus liittää DbContext-luokan periviin luokkiin, jotka haluavat toteuttaa rajapinnan määrityksissä olevat entiteetit (ja taulut). Alla olevassa kuvassa (KUVA 19.) esitellään **IAuditTrailDataContext**, joka määrittelee vähimmäis DbSet-ominaisuudet, jotka AT -DbContextin pitää määritellä.

```
/// <summary>
/// This interface should be added to DataContext-classes that use System.Data.Entity.DbContext
/// </summary>
public interface IAuditTrailDataContext
{
    DbSet<AuditTrailBody> AuditTrailBodys { get; set; }
    DbSet<AuditTrailContent> AuditTrailContents { get; set; }

    DbSet<AuditTrailTransaction> AuditTrailTransactions { get; set; }
}
```

KUVA 19. Kuvassa esitellään **IAuditTrailDataContext**.

AT -kontekstille tehtiin myös sitä vastaava Repository **IAuditTrailRepository**. Tämän rajapinnan täytäntöönpano (implementation) luokkaan toteutettiin logiikka, joka muuntaa tietokantakontekstin keräämät muutostiedot AuditTrailTransaction-tyyppisiksi entiteeteiksi ja tallentaa tiedot, alla olevaan tietokantaan. Tähän luokkaan myös toteutettiin logiikka, joka käy päivittämässä **AuditTrailTransactionin** tilan (State), jos tallennuksen yhteydessä tapahtui virheitä. Tallentaessa tietoja seurattavaan tietokantaan, voisi tapahtua virhe jonka seurauksena tiedot eivät oikeasti tallentuneet, mutta vastaavat AT -tiedot ovat kuitenkin tallentuneet. Tällaisissa tapauksissa Transactionin tila vaihdetaan automaattisesti ja sille asetetaan virhetyypin mukainen tilatieto. Alla olevassa kuvassa (KUVA 20.) esitellään rajapinta **IAuditTrailRepository**, jossa määriteltiin vähimmäisvaatimukset säilytyspaikalle (Repository), jota käytetään **IAuditTrailDataContextin** kanssa.

```
/// <summary>
/// Audit-log repository
/// </summary>
public interface IAuditTrailRepository : IRepository
{
    long SaveAuditTrail(AuditTrailTransaction transaction);
    IQueryable<AuditTrailBody> GetAuditTrailBodies();
    IQueryable<AuditTrailTransaction> GetAuditTrailTransactions();
    IQueryable<AuditTrailContent> GetAuditTrailContents();

    AuditTrailDbContext GetContext();
}
```

KUVA 20. Kuvassa esitellään rajapinta **IAuditTrailRepository**.

AuditTrailTransactionien tilatiedoille (nimetty huonosti **ApplicationContextError**) luotiin oma enum-toteutus. Tätä käytetään määrittäessä Transactionin tila, joka voi päivittyä, jos tiedontallennuksen yhteydessä on tapahtunut virheitä. Tilat ovat riippuvaisia tiedontallennustapahtuman onnis-

tumisasteesta ja sen aikana mahdollisesti tapahtuneista virheistä, jossa yleisimmille virheille on määritetty oma tilansa. Virheet liitettiin Transactionin tilatietoon käyttäen apuna **AuditTrailErrorAttribute**:a, joka periytyy System.**Attribute** luokasta.

```
public enum ApplicationContextError
{
    [Description("Not Defined")]
    NotDefined = 0,
    [Description("Ok")]
    OK = 1,
    [AuditTrailError(typeof(System.Exception))]
    [Description("Exception")]
    Exception = 2,
    [AuditTrailError(typeof(System.Data.SqlClient.SqlException))]
    [Description("SqlException")]
    SqlException = 3,
    [AuditTrailError(typeof(System.Data.Entity.Infrastructure.DbUpdateConcurrencyException))]
    [Description("DbUpdateConcurrencyException")]
    DbUpdateConcurrencyException = 4,
    [AuditTrailError(typeof(System.Data.Entity.Validation.DbEntityValidationException))]
    [Description("DbEntityValidationException")]
    DbEntityValidationException = 5,
    [AuditTrailError(typeof(System.Data.Entity.Infrastructure.DbUpdateException))]
    [Description("DbUpdateException")]
    DbUpdateException=6

}
```

KUVA 21. Kuvassa esitellään **ApplicationContextError** enum, joka määrittelee **AuditTrailTransactionin** tilatiedon siitä, miten tietokantapahtuma meni.

Erilaisille tietokantapahtumille luotiin oma enum toteutuksensa **DbOperations**, joka määritteli tietokantapahtuman tyypin. Enum toteutuksen pohjana käytettiin System.Data.Entity.**EntityState** enumia, joka määrittelee entiteetin tilan. **EntityState** liitettiin vastaavaan **DbOperations** tilaan käyttäen apuna kehitettyä System.**Attribute** luokkaa **AuditTrailDbOperationAttribute**. Alla olevassa kuvassa (KUVA 22.) **DbOperations-**enumilla on arvot 0 = tilaa ei määritetty, 1 = tieto päivitetty, 2 = uusi tieto lisätty, 3 = tieto poistettu, sekä 4 = tieto haettu, mitä ei kuitenkaan käytetty toteutuksessa. Tila neljä lisättiin, jotta kehittäjät huomaisivat, että historiatietojen tallennusta voisi laajentaa seuraamaan myös tiedonhakua.

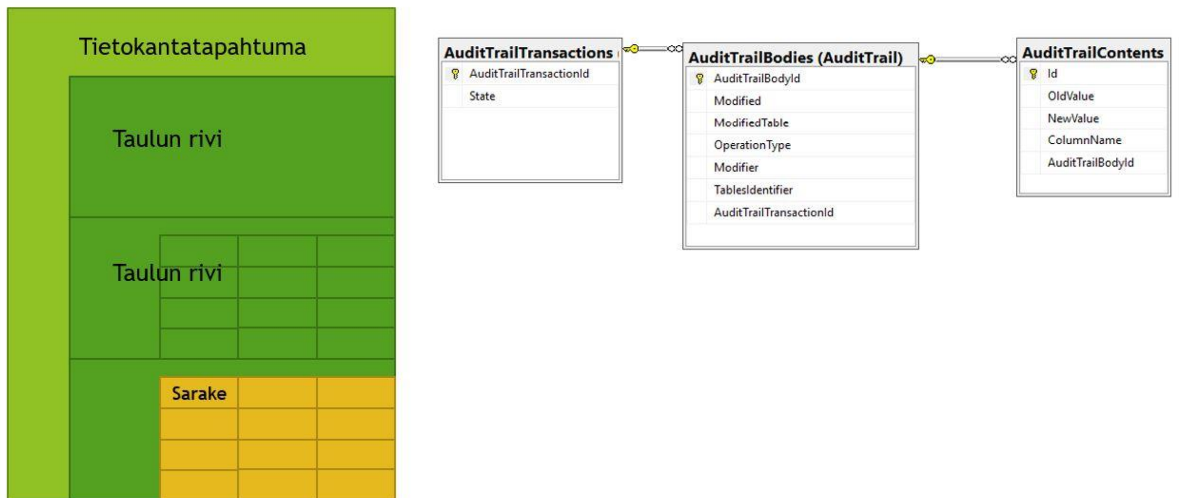
```
public enum DbOperations
{
    [Description("Not Defined")]
    NotDefined = 0,
    [AuditTrailDbOperation(EntityState.Modified)]
    [Description("Update")]
    Update =1,
    [AuditTrailDbOperation(EntityState.Detached)]
    [AuditTrailDbOperation(EntityState.Added)]
    [Description("Insert")]
    Insert = 2,
    [AuditTrailDbOperation(EntityState.Deleted)]
    [Description("Delete")]
    Delete =3,
    [Description("Select")]
    Select =4
}
```

KUVA 22. Kuvassa esitellään **DbOperations** enum, joka määrittelee tietokantapahtuman tyypin.

6.3 Geneerinen tiedontallennus

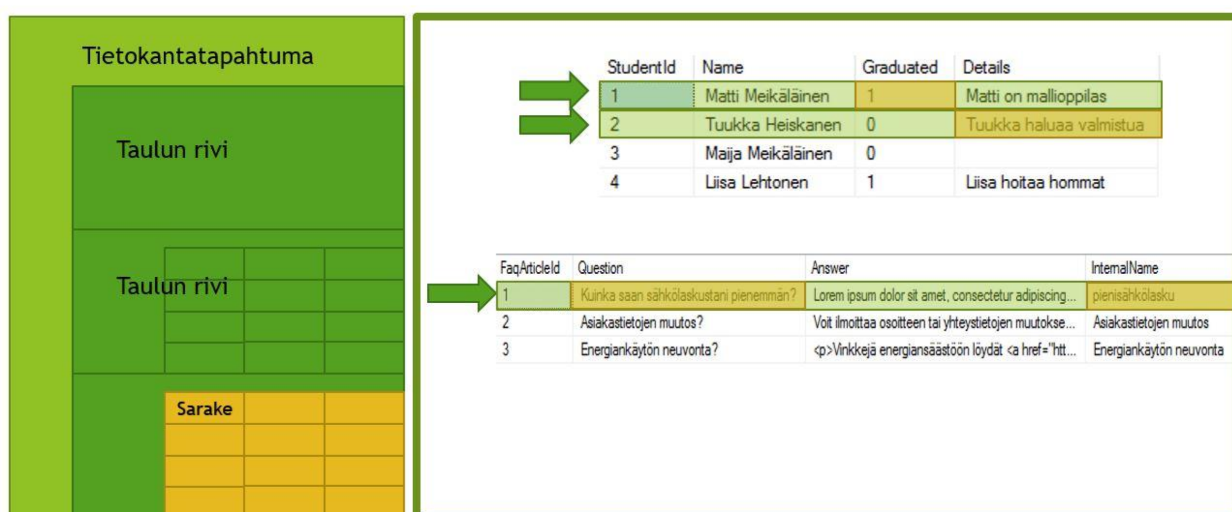
Tiedontallennusominaisuus liitettiin EF:n sisään. Entiteettitiedonmuutoksia seurattiin ChangeTrackerin avulla. ChangeTracker pystyi seuraamaan POCO Proxy entiteetti -tietojen muutoksia DbContext:n sisällä. MasterDbContextin sisällä oleva ChangeTracker piti sisällään kootut tiedot seurattavan tietokannan (MasterDb) muutoksista. Nämä muutokset muunnettiin ja lopulta tallennettiin AT -tietokantaan. AT -taulut kehitettiin omaan skeemaansa. Tauluilla oli oma DbContext luokka ja SaveChanges() -metodi. Seurattavakonteksti (MasterDbContext) välitti muutostiedot prosessoitavaksi ja lopulta ne päätyivät AT -kontekstille. AT -konteksti sai muuntostiedot, jota sen entiteetit tukivat ja tallensi muutokset taustalla olevaan AT -tietokantaan.

Entiteeteillä on neljä mahdollista **EntityState** tilaa: Modified, Detached, Added ja Deleted. Modified ja Deleted on tiloja entiteeteille, jotka tietokantakonteksti on jo tallentanut aikaisemmin tietokantaa, joten näiden entiteettien sijainti tunnetaan taustalla olevassa tietokannan taulussa. Näille riveille on jo muodostunut uniikit yksilöivät tunnisteet (yleensä juokseva avain). Detached ja Added on tiloja, jotka ovat tarkoitettu uusille entiteeteille, eli niitä ei vielä ole tallennettu tietokantaan. Tämän seurauksena niillä ei vielä ole yksilöivää tunnistetta, joka pitäisi kuitenkin liittää AT -tietoihin, jotta muutostiedot voidaan yksilöidä rivikohtaisesti. Alla olevassa kuvassa (KUVA 23.) havainnollistetaan AT -taulujen relaatioita toisiinsa, sekä niiden sisältämän tietosisällön näyttämistä. **AuditTrailTransaction** (tietokantatapahtuma) voi sisältää useamman **AuditTrailBodyn** (taulun rivi), joka taas voi sisältää useamman **AuditTrailContentin** (sarake).



KUVA 23. Kuvassa havainnollistetaan AT -taulujen relaatioita toisiinsa.

Yhdellä tietokantapahtumalle luodussa **AuditTrailTransaction:ssa** pystyy olemaan useita muutoksia eri entiteetteihin/tauluihin. Tiedot tauluista ja muutosriveistä on **AuditTrailBody:ssä**. Taulun rivillä pystyy olemaan useita muutoksia sen eri sarakkeisiin. **AuditTrailBody:ssä** pystyy olemaan useampia **AuditTrailContent:ja**, jotka sisältävät sarakekohtaisen muutostiedon. Alla olevassa kuvassa (KUVA 24.) havainnollistetaan **AuditTrailTransaction**, **AuditTrailBody:n** ja **AuditTrail-Content** entiteettien käyttäytymistä. Oikeanpuoleinen vihreä laatikko kuvastaa yhtä DbContext-luokan sisällä olevaa muutostietojenryhmittymää, jonka muuttuneet tiedot on sijoitettu **AuditTrailTransaction:in**. Tämä tietokantatapahtuma (Transaction) sisältää samat muutokset kuin mainittu ryhmittymä. Tässä tapauksessa muutostietoja on kolmella eri rivillä (Body), jossa kahdella ylimmällä rivillä Students-taulussa on yhdessä sarakkeessa muuttunut tieto. Tämän lisäksi FaqArticles-taulussa on yhdellä rivillä muuttuneita tietoja kahdessa eri sarakkeessa.



KUVA 24. Kuvassa havainnollistetaan **AuditTrailTransaction**, **AuditTrailBody** ja **AuditTrailContent** entiteettien käyttäytymistä.

Jotta seurattavan tietokantakontekstin muutostiedot saatiin tallennettua AT -tietokantaan, täytyi seurattavan tietokantakontekstin sisällä olevaan SaveChanges() -metodiin lisätä kaksi kutsua. Nämä

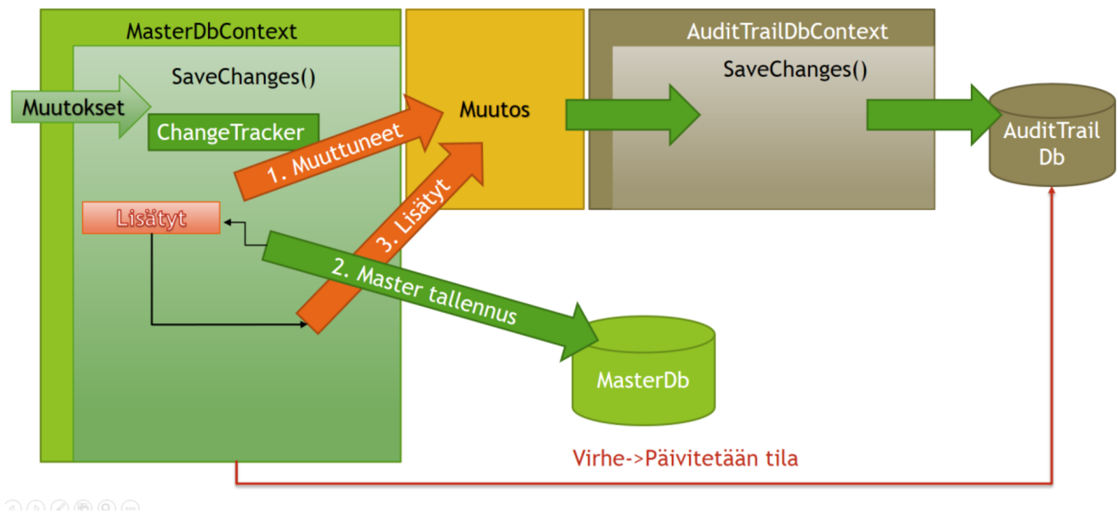
kutsut hoitivat muutostietojen tallennuksen taustalla olevaan AT -tietokantaan. Ensimmäinen kutsu tallensi muutostiedot olemassa olevista entiteeteistä, joiden yksilöivät tunnisteet tiedettiin eli tiedot, joiden tilat olivat Modified tai Deleted. Tämän jälkeen otettiin lisätyt tiedot talteen muuttujaan eli tiedot, jotka olivat tilassa Added ja Detached. Tämä tehtiin, jotta entiteettien tilatieto ei muuttuisi, kun ne tallennetaan seurattavaan tietokantaan. Lisättyjä entiteettejä ei voinut myöskään tallentaa ensimmäisessä vaiheessa, koska niille ei ollut vielä muodostunut tarvittavia yksilöiviä tunnisteita. Kun seurattavan tietokannan tiedot oli tallennettu, voitiin muuttujaan lisätyt entiteetit välittää metodille, joka hoiti lisättyjen entiteettien tallennuksen AT -tietokantaan. Tämä voitiin tehdä, koska entiteetit olivat saaneet yksilöivän tunnisteensa tallentuessaan seurattavaan tietokantaan. Alla olevassa kuvassa (KUVA 25.) on seurattavan tietokantakontekstin (Master DbContext) SaveChanges() -metodin sisällä olevat kaksi kutsua, jotka hoitavat AT -tietojen tallennuksen vaiheistetusti **base.SaveChanges()** -metodin ympärillä. **Base.SaveChanges()** -metodi tallentaa tiedot Master tietokantaan.

```
SaveUpdatedAuditTrails(auditUser, auditDate);
var addedEntities = ChangeTracker.Entries<IAuditTrailEntity>().GetAddedEntities().ToList();
var rowAffecteds = base.SaveChanges();
SaveAddedAuditTrails(auditUser, auditDate,addedEntities);
return rowAffecteds;
```

KUVA 25. AT -tietojen vaiheistettu tallennus.

Kolmivaiheisen tietojentallennuksen lisäksi tietomuutostentallennukseen kehitettiin ominaisuus, joka päivittää tapahtuneen tietokantatapahtuman (Transaction) tilan. Jos tallennuksen yhteydessä tapahtuu virheitä, tämä ominaisuus päivitti **AuditTrailTransaction:in** tilan. Tämän perusteella saatiin tieto, että tallennus ei ole välttämättä onnistunut seurattavaan tietokantaan.

Alla olevassa kuvassa (KUVA 26.) esitellään kolmivaiheinen tietojentallennus AT -tietokantaan ja seurattavaan tietokantaan eli Master-tietokantaan. Muutostiedot tulevat MasterDbContext:n SaveChanges-metodiin, jossa muutostiedot saadaan ChangeTracker-ilmentymältä. (Vaihe 1.) Ensiksi tallennetaan muuttuneet tiedot AT -tietokantaan eli tiedot, jotka on jo tallennettu Master-tietokantaan eli nämä tiedot ovat joko EntityState.Modified tai Deleted -tilassa (näillä tiedoilla on jo yksilöivätunnistetieto Master-tietokannassa). Nämä tiedot joudutaan muuttamaan AT -entiteettien tukemaan muotoon, jotta ne pystytään tallentamaan AT -tietokantaan. Tämän jälkeen uudet tiedot tallennetaan muuttujaan (lisätyt) eli tiedot, jotka ovat tilassa EntityState.Added. (Vaihe 2.) Tämän jälkeen kaikki muutostiedot tallennetaan Master-tietokantaan. Kun tiedot ovat tallentuneet Master-tietokantaan, saadaan lisätyille tiedoille yksilöivät tunnisteet, koska ne ovat syntyneet Master-tietojen tallennuksen yhteydessä. (Vaihe 3.) Lopuksi lisätyt tiedot tallennetaan AT -tietokantaan saman **AuditTrailTransaction:in** alle, kuin muuttuneet tiedot. Tietojenmuutoksessa käytetään samankaltaista logiikkaa kuin muuttuneille tiedoille. (Mahdollinen Vaihe 4.) Jos tässä prosessissa tapahtuu virheitä, virheenmukainen tilatieto päivitetään kyseiseen **AuditTrailTransaction:in**.



KUVA 26. Kuvassa esitellään kolmivaiheinen tietojentallennus.

7 TIEDONHAKU

AT -järjestelmän tiedonhakutoiminnallisuus toteutettiin omassa moduulissaan riippumattomana tiedon näyttävästä moduulista. Hakutoiminnallisuudelle kehitettiin omat rajapintansa ja luokkatoteutuksensa. Tämän lisäksi tiedonhaku eriytettiin tiedontallennusprosessista, koska tiedon tallennus AT -järjestelmässä tapahtuu automaattisesti muiden tietojenmuutoksen rinnalla, mutta tiedonhaku on taas prosessi, joka tehdään tilanteesta riippuen. Tällä jaolla haluttiin selkeyttää prosessien luonnetta, joka tallennuksella ja haulla on, toistensa vastakohdat automaattinen verrattuna manuaaliseen. Toinen syy tiedontallennuksen ja -haun eriyttämiseen oli, että tiedontallennukseen sisältyi paljon logiikkaa, joka muuntaa seurattavat entiteetit AT -entiteeteiksi. Tiedonhaun ydin toiminnallisuus taas ei sisällä paljoakaan logiikkaa, mutta mahdollistaa tarvittavat raamit tiedonhakuun.

7.1 Rajapinnanmääritys

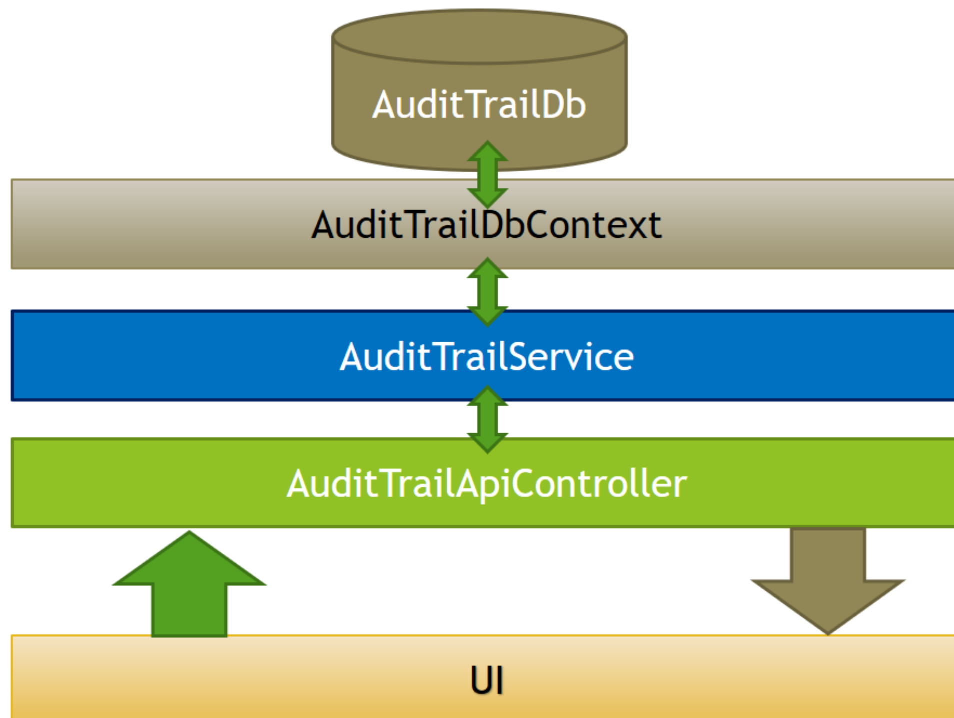
Tiedonhaku haluttiin toteuttaa tavalla, että AT -tietoja voidaan hakea geneerisesti eli tiedonhakemiselle ei asetettu tarkkoja rajoja, vaan tarjottiin raamit, jotka ohjaavat tiedon hakemiseen. Luotiin rajapinta, joka tulisi käyttämään **DbContext**-luokan ilmentymää ja pystyisi palauttamaan System.Linq.**IQueryable** -tyyppisiä AT -entiteettejä. Metodit ottivat vastaan lambda expressioneja, jotka oli tyypitetty AT -entiteeteiksi. Tämän rajapinnan toteutusluokka osasi siis hakea AT -tietoja tietokannasta. Metodien parametrit (Expression<Func<T,bool>>) olivat sen tyyppisiä, että hakuehdot hakemiseen voi määrittää tapauskohtaisesti. Palautetut tietotyypit (tyypitetyt IQueryable<T>) on kehitetty tilanteisiin, jossa haetaan tietoa erilaisista tietolähteistä. Palautettu tietotyyppi mahdollistaa erilaisten hakuehtojen määrittämisen kutsuvassa päässä ja on sen takia mainio kyseiseen tilanteeseen.

```
interface IAuditTrailService<T> where T : DbContext
{
    IQueryable<AuditTrailBody> GetAuditTrailBodies(Expression<Func<AuditTrailBody, bool>> predicate);
    IQueryable<AuditTrailTransaction> GetAuditTrailTransactions(Expression<Func<AuditTrailTransaction, bool>> predicate);
    IQueryable<AuditTrailContent> GetAuditTrailContents(Expression<Func<AuditTrailContent, bool>> predicate);
}
```

KUVA 27. IAuditTrailService-rajapinnan määritykset.

7.2 Geneerinen tiedonhaku

AT -tiedoille luotiin oma **ApiController** nimeltään **AuditTrailManageController**. Tämä luokka toimi palvelukerroksessa (Service Layer). Tämän kontrollerin tehtävä oli ottaa vastaan API -kutsuja ja hakea AT -tietoja käyttäen apuna **IAuditTrailService:n** luokkatoteutusta. Vastaavanlainen toteutus tehtiin myös kolmannen osapuolen toteutukseen.



KUVA 28. Kuvassa esitellään käyttöliittymältä tapahtuvan Audit Trail -tiedonhaun tekemää reittiä.

Kontrollerin metodit ottivat vastaan **AuditTrailSearchModel-**luokkia. Tämä luokka määritti palvelulta (Service) haettavat tiedot. Lisäksi luokka määritti millä tavalla tieto näytettiin käyttöliittymätasolla. Hakuluokalle pystyi määrittämään haettavan rivin, haettavan taulun nimen, tiedon siitä näytetäänkö vain onnistuneet muutostiedot vai myös virheelliset, näytettävien rivien lukumäärän, ensimmäisen näytettävän rivin indeksin, järjestys suunnan, järjestys sarakkeen ja filtterin. Näiden tietojen perusteella kontrolleri osasi hakea ja palauttaa oikeat tiedot, oikealla tavalla, loppukäyttäjän käyttöliittymälle.

```
[NotMapped]
public class AuditTrailSearchModel
{
    public int RowId {get;set;}

    public string TableName { get; set; }
    public bool ShowAll { get; set; }

    public int iDisplayLength { get; set; }
    public int  IDisplayStart { get; set; }
    public string sSortDir { get; set; }

    public AuditTrailSortColumnEnum SortColumn { get; set; }
    public string Filter { get; set; }
}
```

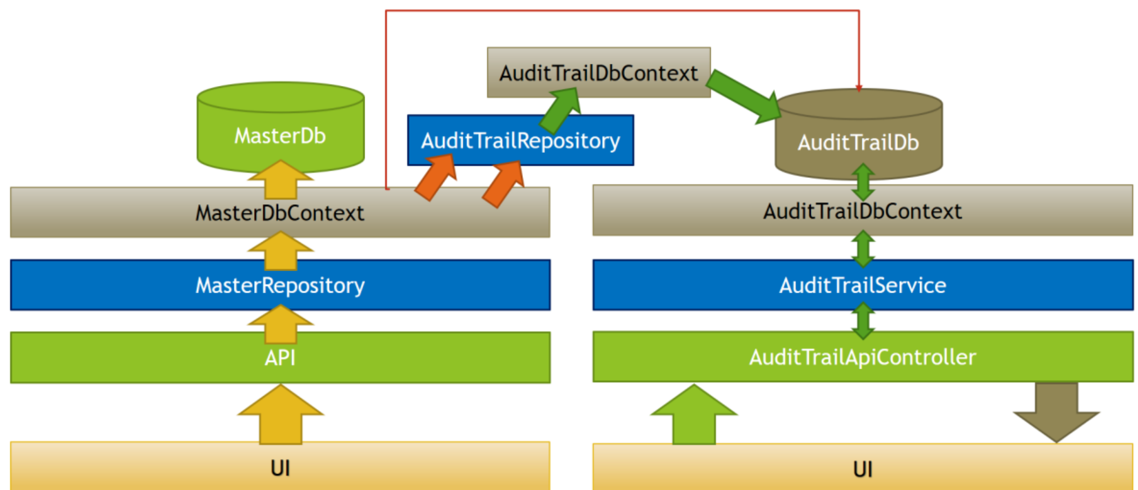
KUVA 29. Kuvassa esitellään **AuditTrailSearchModel**-luokka, joka toimii tiedonhaussa ja tiedonasemoinnissa käyttöliittymällä.

```
public enum AuditTrailSortColumnEnum
{
    AuditTrailTransactionId = 0,
    State = 1,
    StateText = 2,
    AuditTrailBodyId = 3,
    Time = 4,
    TableName = 5,
    OperationType = 6,
    OperationTypeText = 7,
    Modifier = 8,
    RowId = 9,
}
```

KUVA 30. Kuvassa esitellään AuditTrailSortColumnEnum-enum, joka määrittelee asemointisarakkeita.

8 KÄYTTÖLIITTYMÄ

Vaatimukset käyttöliittymästä olivat, että yhden entiteetin AT -tiedot voitaisiin näyttää käyttöliittymän kautta ainakin kronologisessa ja käänteisessä järjestyksessä erinäköisissä näkymissä. Projektissa kehitettiin yksi näkymä, joka näyttää yhden entiteetin tiedot riippumatta sen kanta entiteetti luokasta. Kanta entiteetti luokalla tarkoitetaan luokkaa, joka määrittää AT -tiedon alkuperäisen entiteettimuodon, josta se oli muunnettu AT -entiteeteiksi. Alla olevassa kuvassa (KUVA 31.) esitellään toteutettu kokonaisuus. Käyttöliittymällä tapahtuvista tietomuutoksista kolmivaiheiseen tietojentallennukseen ja lopulta AT -tietojen hakemiseen käyttöliittymältä.



KUVA 31. Kuvassa esitellään toteutettu kokonaisuus.

8.1 Tiedon näyttäminen käyttöliittymässä

Tiedonnäyttämiseen kehitettiin yksi näkymä, johon tieto voitiin hakea API-kontrollerilta ja näyttää taulukkomuotoisessa esitysmuodossa. Tiedon haussa ja sen asemoinnissa käytettiin kehitettyä **AuditTrailSearchModel**:a. Käyttöliittymältä tietoja pystyi hakemaan taulu ja rivikohtaisesti. Hakuun pystyttiin ottamaan mukaan myös mahdollisesti epäonnistuneet tietomuutokset. Tietoja pystyi asemoimaan rivikohtaisesti laskevaan tai nousevaan järjestykseen. Vastaavanlainen toteutus kehitettiin myös ratkaisussa, joka oli toteutettu kolmannen osapuolenkirjastoa käyttäen. Alla olevissa kuvissa (KUVA 32, 33, 34) esitellään toteutettua selainpohjaista käyttöliittymää. Kuvassa (KUVA 32.) Audit Trail -kannasta on haettu Customers -taulusta riville 2917 onnistuneet muutostapahtumat. Tiedot näytetään taulukossa. Kuvassa (KUVA 33.) Audit Trail -kannasta on haettu Customers-taulusta riville 2917 tapahtuneet onnistuneet ja epäonnistuneet muutokset. Epäonnistuneet tietomuutokset näytetään punaisilla palloilla. Jos hiiren vie pallon päälle, näyttää se virhetyypin. Kuvassa (KUVA 34.) Audit Trail -kannasta on haettu CustomerAddresses-taulusta riville 5836 tapahtuneet muutokset. Tietoja on asemoitu muokkaustyypin mukaan. Kuvassa (KUVA 35.) AT -kannasta on haettu CustomerAddresses-taulusta riville 5836 tapahtuneet muutokset ja yksi muutosrivi on avattu sarakekohtaista tarkastelua varten.

Audit trail

Etsi Audit lokitietoja

Taulun nimi: Customers

Rivinumero: 2917

☐ Näytä myös epäonnistuneet

Etsi

Onnistunut	Muutos Aika	▲ Muokkaus tyyppi	Muokkaaja	Taulu	Rivi
●	12.03.2018 10:33	Update	SQLAN\tuukka.heiskanen	Customers	2917
●	12.03.2018 10:46	Update	SQLAN\tuukka.heiskanen	Customers	2917
●	12.03.2018 14:57	Update	SQLAN\tuukka.heiskanen	Customers	2917
●	12.03.2018 15:00	Update	SQLAN\tuukka.heiskanen	Customers	2917
●	12.03.2018 15:01	Update	SQLAN\tuukka.heiskanen	Customers	2917
●	12.03.2018 15:07	Update	SQLAN\tuukka.heiskanen	Customers	2917
●	12.03.2018 15:09	Update	SQLAN\tuukka.heiskanen	Customers	2917

Ensimmäinen Edellinen **1** Seuraava Viimeinen

KUVA 32. Kuvassa esitellään toteutettua selainpohjaista käyttöliittymää.

Audit trail

Etsi Audit lokitietoja

Taulun nimi: Customers

Rivinumero: 2917

☑ Näytä myös epäonnistuneet

Etsi

Onnistunut	Muutos Aika	▲ Muokkaus tyyppi	Muokkaaja	Taulu	Rivi
●	12.03.2018 10:33	Update	SQLAN\tuukka.heiskanen	Customers	2917
●	12.03.2018 10:46	Update	SQLAN\tuukka.heiskanen	Customers	2917
●	12.03.2018 14:57	Update	SQLAN\tuukka.heiskanen	Customers	2917
●	12.03.2018 15:00	Update	SQLAN\tuukka.heiskanen	Customers	2917
●	12.03.2018 15:01	Update	SQLAN\tuukka.heiskanen	Customers	2917
●	12.03.2018 15:03	Update	SQLAN\tuukka.heiskanen	Customers	2917
●	12.03.2018 15:05	Update	SQLAN\tuukka.heiskanen	Customers	2917
●	12.03.2018 15:07	Update	SQLAN\tuukka.heiskanen	Customers	2917
●	12.03.2018 15:09	Update	SQLAN\tuukka.heiskanen	Customers	2917

Ensimmäinen Edellinen **1** Seuraava Viimeinen

KUVA 33. Esitellään toteutettua selainpohjaista käyttöliittymää.

Audit trail

Etsi Audit lokitietoja

Taulun nimi: Customeraddresses

Rivinumero: 5836

☐ Näytä myös epäonnistuneet

Etsi

Onnistunut	Muutos Aika	Muokkaus tyyppi	Muokkaaja	Taulu	Rivi
●	12.03.2018 10:21	Insert	SQLAN\tuukka.heiskanen	CustomerAddresses	5836
●	12.03.2018 10:33	Update	SQLAN\tuukka.heiskanen	CustomerAddresses	5836
●	12.03.2018 10:46	Update	SQLAN\tuukka.heiskanen	CustomerAddresses	5836
●	12.03.2018 14:57	Update	SQLAN\tuukka.heiskanen	CustomerAddresses	5836
●	12.03.2018 15:00	Update	SQLAN\tuukka.heiskanen	CustomerAddresses	5836
●	12.03.2018 15:01	Update	SQLAN\tuukka.heiskanen	CustomerAddresses	5836
●	12.03.2018 15:07	Update	SQLAN\tuukka.heiskanen	CustomerAddresses	5836
●	12.03.2018 15:09	Update	SQLAN\tuukka.heiskanen	CustomerAddresses	5836

Ensimmäinen Edellinen 1 Seuraava Viimeinen

KUVA 34. Esitellään toteutettua selainpohjaista käyttöliittymää.

Etsi Audit lokitietoja

Taulun nimi: Customeraddresses

Rivinumero: 5836

☐ Näytä myös epäonnistuneet

Etsi

Onnistunut	Muutos Aika	Muokkaus tyyppi	Muokkaaja	Taulu	Rivi
●	12.03.2018 10:21	Insert	SQLAN\tuukka.heiskanen	CustomerAddresses	5836

Sarake nimi	Vanha arvo	Uusi arvo
CustomerAddressId		5836
CustomerLine		Tuukan Yritys oy
Street		Testikatu 2 A 5
StreetName		Testikatu
StreetQualifier		2
Staircase		A
Apartment		5
PostalCode		70150
PostalPlace		KUOPIO
CountryCode		FI
AddressType		Home
Start		11/03/2018 22:00:00
Stop		31/12/2099 22:00:00
Created		12/03/2018 08:21:47
Creator		SQLAN\tuukka.heiskanen
Modified		12/03/2018 08:21:47
Modifier		SQLAN\tuukka.heiskanen
Deleted		False

Onnistunut	Muutos Aika	Muokkaus tyyppi	Muokkaaja	Taulu	Rivi
●	12.03.2018 10:33	Update	SQLAN\tuukka.heiskanen	CustomerAddresses	5836
●	12.03.2018 10:46	Update	SQLAN\tuukka.heiskanen	CustomerAddresses	5836

KUVA 35. Kuvassa esitellään toteutettua selainpohjaista käyttöliittymää.

9 TOTEUTUS KOLMANNEN OSAPUOLEN KIRJASTOON

Opinnäytetyössä kehitettiin myös vastaavanlainen toteutus, jossa käytettiin apuna kolmannen osapuolen valmista kirjastoa. Kirjasto tarjosi AT -tietojentallennuksen EF:n sisällä, sekä AT -tietojen hakemiseen kehitettyjä metodeja. Kirjaston ja muutaman koodirivin avulla saatin toteutettua samat toiminnallisuudet kuin kehitetyssä ratkaisussa.

Kirjaston avulla tiedot saatiin vähän vastaavalla tavalla tallennettua erilliseen skeemaan kuin omassa toteutuksessa. Eriyttäminen ei kuitenkaan onnistunut suoraan kirjaston tarjoaman rajapinnan avulla, vaan jouduttiin kirjoittamaan koodia. Tämän lisäksi tietojen hakuun kehitettiin vastaavanlainen Service, joka osasi hakea AT -tietoja. Tietojen näyttämiseen kehitettiin myös samanlainen käyttöliittymä, joka haki AT -tietoja API-kontrollerilta, joka taas käytti kolmannen osapuolen kirjastolle kehitettyä Serviceä. Kirjaston ja muutaman koodirivin avulla tietomuutokset saatiin tallennettua omaan skeemaansa, haettua käyttöliittymältä ja näytettyä taulukkomuotoisessa esitysmuodossa.

Kolmannen osapuolen järjestelmää on testattu ja kehitetty kauemmin. Tämän seurauksena voidaan olettaa, että se on varmempi tuotantokäytössä ja todennäköisesti nopeampi. Kolmannen osapuolen ratkaisussa on kuitenkin huono puolensa siinä, että koodipohja ei ole hallinnoitavissa, jonka seurauksena ollaan rajoittuneita kirjaston tarjoamiin raameihin.

Koska tilaaja käyttää EF:n koodi ensin metodiikkaa ja AT -taulut haluttiin eriyttää omaan skeemaansa, kolmannen osapuolen kirjaston tarjoamille entiteeteille jouduttiin luomaan oma tietokantakonteksti (DbContext:n perivä luokka). Tämän lisäksi kirjaston oletus entiteeteiltä puuttui muutostietojen pääavain, jonka seurauksena kehitetty geneerinen hakutoiminto ei toiminut suoraan, koska hakutoiminnallisuus pohjautui suurelta osin pääavain kenttään. Entiteettien rakenteeseen täytyi tämän seurauksena lisätä pääavain, jotta asiakkaan vaatimukset täyttyisivät kyseisen kirjaston avulla.

Mahdollisia ongelmia saattaa ilmetä muutoksien seurauksena, jos kirjastoa halutaan päivittää uudempaan versioon. Riippuen versiomuutoksien laajuudesta ja kohteesta, voidaan päätyä tilanteeseen, että versiopäivitystä ei voida suorittaa helposti. Mahdollinen ongelma saattaisi syntyä, jos kirjaston uudemmassa versiossa entiteetteihin tulee muutoksia kenttiin, jotka kartoittuvat tietokantaan. Entiteettien rakenteellinen muutos aiheuttaisi tilanteen, jonka seurauksena jouduttaisiin väistämättä tekemään uusi migraatio. Tämä heijastuisi kaikkiin seuraaviin ympäristöpäivityksiin, joissa päivitetty versio kirjastosta on otettu käyttöön. Pahimmassa tapauksessa entiteettien rakenteellinen muutos olisi niin laaja, että keskeisimmät kentätkin muuttuvat tai ne otetaan pois käytöstä. Tämän seurauksena voisi joutua tekemään tietokonversioita vanhasta uuteen tietomalliin. Tämä on kuitenkin epätodennäköistä.

Muutamia poikkeuksia lukuun ottamatta, kolmannen osapuolen kirjasto tarjosi hyvän integraatiotuen EF:n sisään. Kirjasto myös tarjosi ominaisuuksia, jotka kehitetystä toteutuksesta jäi uupumaan esimerkiksi asynkronisten- ja massatallennettavien tietojen auditointi. Näihin liittyi DbContextin metodit

SaveChangesAsync() ja BulkSaveChanges(). BulkSaveChanges() nopeuttaa tiedontallennus tietokantaan, koska se vähentää tietokantaan tehtäviä kierroksia. SaveChanges() -metodi tekee kierroksen tietokantaan ja takaisin jokaisesta muutoksesta, kun taas BulkSaveChanges() tekee niitä huomattavasti vähemmän. Kolmannen osapuolen kirjaston avulla tietomuutostenseurauksen ja tallennuksen saa kätevästi toteutettua EF:n sisälle.

10 KEHITYSIDEAT JA JATKOKEHITYSTARPEET

AT -järjestelmän kehityksen yhteydessä ilmeni muutamia kehitysideoita. Tiedon tallennukseen olisi hyvä saada konfigurointiluokka, jonka avulla pystyttäisiin määrittämään seurattavat entiteetit. Seurattavat entiteetit olisi hyvä pystyä asettamaan ajonaikana ja konfigurointiin täytyisi kehittää helppokäyttöinen käyttöliittymä, joka mahdollistaa edellä mainitun toiminnallisuuden. Tätä voisi kutsua, vaikka entiteettitietomuutosseurauksen -ja tallennuksen konfigurointi moduuliksi eli Configuration Module of Entity Data Change Tracking and Saving: **CMEDCTS** (Ehkä tuota nimeä voisi vielä hioa).

Toteutus seurasi vain tietomuutoksia, joten olisi hyvä, että käyttäjälle tekemät haut ja varsinkin käyttäjälle näytettävät tiedot tallennettaisiin vastaavanlaiseen tietokantaan, jotta niitä olisi helppo tarkastella. Tällainen toteutus tultaisiin todennäköisesti tekemään Businesslogiikkakerrokseen (Business Layer), koska kyseisen järjestelmän tapauksessa eri kontrollerit ovat yleisin paikka, mistä tietoja palautetaan loppukäyttäjälle.

Tietojentallennuksen yhteydessä tallennettiin tietokantatapahtuman tila, joka kertoi tapahtuman onnistumisen eli tiedon mahdollisista tallennuksen aikana tapahtuneista virheistä (tai onnistumisesta). Tällä hetkellä arvo ei kerro muuta, kuin virhetyypin. Tätä voitaisiin laajentaa siten, että virheille tehtäisiin oma taulu, josta olisi viittaus **AuditTrailTransactions**-tauluun. **AuditTrailTransaction:lla** voisi olla kokoelmanavigaatioreferenssi, josta virhepinojälki (Error Stack Trace) voitaisiin todentaa.

Tiedontallennus pitäisi laajentaa tukemaan massaoperaatioita, Dapper-kutsuja, sekä muita operaatioita, jotka jäävät EF:n ohjelmistorajapinnan ulkopuolelle tehdessään muutoksia tietokantaan. Tiedontallennusta voisi myös optimoida, jotta se toimisi nopeammin. **AuditTrailBody**-entiteetin TablesIdentifier täytyisi muuttaa erilaiseksi, jotta taulut ilman kokonaisluku (integer) -tyyppistä avainta pystyttäisiin identifioimaan. Tällä hetkellä toteutus tukee vain kokonaisluku (integer) -tyyppisten pääavainten tallentamista.

KUVA 36. Esitellään kehitysideoita toteutettuun käyttöliittymään.

Käyttöliittymää voitaisiin kehittää loputtomasti, koska loppujen lopuksi se on olemassa olevan tiedon näyttämistä erilaisissa esitysmuodoissa. Esimerkkinä kuitenkin voisi olla näkymät: AT -tietomuutoshistorian vertailu olemassa oleviin seurattavan tietokannan tietoihin, käyttäjäkohtaisten tietojen näyttäminen, käyttäjäkohtaistentietojen muutoshistoria ja muutosfrekvenssi, sekä vertailu muiden käyttäjien tekemiin muutoksiin ja tietomuutosten esiintyvyys eri ajanjaksoina. Kaikista tiedoista voisi myös kehitellä erilaisia diagrammeja ja kuvaajia, jotka selkeyttäisivät suurien tietomassojen tarkastelua. Nämä ja loputon määrä erilaisia näkymiä.

Tiedonhakuun täytyisi tehdä erilaiset roolitukset ja käyttäjäoikeustasot, jotta AT -tietojen pääsy voitaisiin rajata näkyväksi vain tietyille tahoille. Tätä ei välttämättä kannattaisi tehdä tiedontallennukseen, vaan erillisenä siitä, koska muuten oikeuksien muuttuessa täytyisi AT -tietoja mennä muuttamaan. Toteutus olisi järkevintä tehdä riippumattomana, erillisenä moduulina, joka voitaisiin liittää AT -tietojentallennukseen. Näiden kahden väliin voisi tehdä liitoksen, joka liittäisi kantatietoihin liitetyt käyttäjäoikeusvaatimukset AT -tietokannassa oleviin tietoihin. Tämä ominaisuus voisi toimia edellä mainitun entiteetti tietomuutosseurauksen -ja tallennuksen konfigurointi moduulin rinnalla.

11 POHDINTA

Tarpeet tietomuutostenseurausjärjestelmistä kasvavat digitaalisen kehityksen ja tietosuoja-asetusten myötä. Kasvavat tietomassat lisäävät tarpeita tietojen oikeellisuuden todentamisessa, jossa tietomuutostenseurantajärjestelmät tulevat avuksi.

EU:n tavoite yhtenäisestä tietoturvapolitiikasta ja sen tuomat tietosuoja-asetukset lisäävät yksityishenkilöiden oikeuksia ja samalla lisäävät yritysten vastuita. Kuinka yritykset vastaavat vaatimuksiin? Kuinka asetusta noudatetaan ja sovelletaan? Miten muualla maailmassa reagoidaan EU:n asettamaan linjaukseen? Tuleeko tietomuutostenseurantajärjestelmistä normi ohjelmistotuotannossa? Vain aika näyttää.

12 LÄHDELUETTELO

2, TaA 43 §, Valtiokonttori. (16. 3. 2017). *2, TaA 43 § | Valtiokonttori*. Noudettu osoitteesta Valtiokonttori: http://www.valtiokonttori.fi/kasikirja/public/default.aspx?nodeid=23977

ECFP, GPO, Web archive. (7. 5. 2018). *Web archive - Electronic Source of Federal Regulations*. Noudettu osoitteesta Web archive: https://web.archive.org/web/20100608090316/http://ecfr.gpoaccess.gov/cgi/t/text/text-idx?c=ecfr&sid=3094bb76639c37239557767756fd7878&rgn=div5&view=text&node=21%3A1.0.1.1.7&idno=21

Entity Framework Tutorial. (3. 3. 2018). *Database First development with Entity Framework*. Noudettu osoitteesta Entity Framework Tutorial: http://www.entityframeworktutorial.net/choosing-development-approach-with-entity-framework.aspx

Entity Framework Tutorial. (22. 2. 2018). *Types of Entities in Entity Framework*. Noudettu osoitteesta Entity Framework Tutorial: http://www.entityframeworktutorial.net/Types-of-Entities.aspx

Entity Framework Tutorial. (22. 2. 2018). *What is Entity Framework*. Noudettu osoitteesta Entity Framework Tutorial: http://www.entityframeworktutorial.net/basics/what-is-entity-in-entityframework.aspx

Entity Framework Tutorial, Architecture. (22. 4. 2018). *Entity Framework Tutorial - Architecture*. Noudettu osoitteesta Entity Framework Tutorial: http://www.entityframeworktutorial.net/EntityFramework-Architecture.aspx

Entity Framework Tutorial, Entity Framework Architecture. (22. 4. 2018). *Entity Framework Architecture - EntityFrameworkTutorial.net*. Noudettu osoitteesta EntityFrameworkTutorial.net: http://www.entityframeworktutorial.net/EntityFramework-Architecture.aspx

GitHub, Microsoft/TypeScript. (21. 4. 2018). *Microsoft/TypeScript | GitHub*. Noudettu osoitteesta GitHub: https://github.com/Microsoft/TypeScript

GitHub, StackExchange/Dapper. (22. 4. 2018). *StackExchange/Dapper | GitHub*. Noudettu osoitteesta GitHub: https://github.com/StackExchange/Dapper

IBM, IBM knowledge Center, Structured Query Language (SQL). (14. 4. 2018). *IBM knowledge Center - Structured Query Language (SQL)*. Noudettu osoitteesta IBM knowledge Center: https://www.ibm.com/support/knowledgecenter/en/SSEPGG_9.5.0/com.ibm.db2.luw.sql.ref.doc/doc/c0004100.html

Klein, S. (2010). *Pro Entity Framework 4.0, The future of data access in .NET programming.* -: Apress.

Lerman, J. (2010). *Programming Entity Framework, Second Edition.* 1005 Gravenstein Highway North, Sebastopol, CA 9547: O'Reilly Media, Inc.

Microsoft, Microsoft Docs, .NET Framework Data Providers. (30. 3. 2017). *.NET Framework Data Providers | Microsoft Docs*. Noudettu osoitteesta Microsoft Docs: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/data-providers

Microsoft, Microsoft Docs, .NET Framework Guide. (21. 4. 2018). *.NET Framework 4.7, 4.6 and 4.5 | Microsoft Docs*. Noudettu osoitteesta Microsoft Docs: https://docs.microsoft.com/en-us/dotnet/framework/

Microsoft, Microsoft Docs, Adding a Controller. (21. 4. 2018). *Adding a Controller | Microsoft Docs*. Noudettu osoitteesta Microsoft Docs: https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/introduction/adding-a-controller

Microsoft, Microsoft Docs, ADO.NET Overview. (30. 3. 2017). *ADO.NET Overview | Microsoft Docs*. Noudettu osoitteesta Microsoft Docs: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ado-net-overview

Microsoft, Microsoft Docs, ASP.NET overview. (21. 4. 2018). *ASP.NET overview | Microsoft Docs*. Noudettu osoitteesta Microsoft Docs: https://docs.microsoft.com/en-us/aspnet/overview

Microsoft, Microsoft Docs, Compiling and building in Visual Studio. (14. 7. 2017). *Compiling and building in Visual Studio | Microsoft Docs*. Noudettu osoitteesta Microsoft Docs: https://docs.microsoft.com/en-us/visualstudio/ide/compiling-and-building-in-visual-studio

Microsoft, Microsoft Docs, Entity Framework Overview. (30. 3. 2017). *Entity Framework Overview | Microsoft Docs*. Noudettu osoitteesta Microsoft Docs: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ef/overview

Microsoft, Microsoft Docs, Entity SQL Overview. (30. 3. 2017). *Entity SQL Overview | Microsoft Docs*. Noudettu osoitteesta Microsoft Docs: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ef/language-reference/entity-sql-overview

Microsoft, Microsoft Docs, Expression Trees (C#). (22. 4. 2018). *Expression Trees (C#) | Microsoft Docs*. Noudettu osoitteesta Microsoft Docs: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/expression-trees/index

Microsoft, Microsoft Docs, Get Started with ASP.NET Web API 2 (C#). (21. 4. 2018). *Get Started with ASP.NET Web API 2 (C#) | Microsoft Docs*. Noudettu osoitteesta Microsoft Docs: https://docs.microsoft.com/en-us/aspnet/web-api/overview/getting-started-with-aspnet-web-api/tutorial-your-first-web-api

Microsoft, Microsoft Docs, Introduction. (21. 4. 2018). *Introduction | Microsoft Docs*. Noudettu osoitteesta Microsoft Docs: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/introduction

Microsoft, Microsoft Docs, Language Integrated Query (LINQ). (21. 4. 2018). *Language Integrated Query (LINQ) | Microsoft Docs*. Noudettu osoitteesta Microsoft Docs: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/

Microsoft, Microsoft Docs, LINQ to Entities. (30. 3. 2017). *LINQ to Entities - Microsoft Docs*. Noudettu osoitteesta Microsoft Docs: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ef/language-reference/linq-to-entities

Microsoft, Microsoft Docs, Migrations - EF Core. (30. 10. 2017). *Migrations - EF Core | Microsoft Docs*. Noudettu osoitteesta Microsoft Docs: https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/

Microsoft, Microsoft Docs, Razor syntax reference for ASP.NET Core. (22. 4. 2018). *Razor syntax reference for ASP.NET Core | Microsoft Docs*. Noudettu osoitteesta Microsoft Docs: https://docs.microsoft.com/en-us/aspnet/core/mvc/views/razor?view=aspnetcore-2.1

Microsoft, Microsoft Docs, SQL Server Documentation. (21. 4. 2018). *SQL Server Documentation | Microsoft Docs*. Noudettu osoitteesta Microsoft Docs: https://docs.microsoft.com/en-us/sql/sql-server/sql-server-technical-documentation?view=sql-server-2017

Microsoft, MSDN, ASP.NET MVC Overview. (21. 4. 2018). *ASP.NET MVC Overview | MSDN*. Noudettu osoitteesta MSDN: https://msdn.microsoft.com/en-us/library/dd381412(v=vs.108).aspx

Microsoft, MSDN, DbContext Class (System.Data.Entity). (22. 4. 2018). *DbContext Class (System.Data.Entity) | MSDN*. Noudettu osoitteesta MSDN: https://msdn.microsoft.com/en-us/library/system.data.entity.dbcontext(v=vs.113).aspx

Microsoft, MSDN, Entity Framework Code First Conventions. (23. 10. 2016). *Microsoft Developer Network (MSDN) - Entity Framework Code First Conventions*. Noudettu osoitteesta Microsoft Developer Network (MSDN): https://msdn.microsoft.com/en-us/library/jj679962(v=vs.113).aspx

Microsoft, MSDN, Entity Framework Loading Related Entities. (23. 10. 2017). *MSDN - Entity Framework Loading Related Entities*. Noudettu osoitteesta Microsoft Developer Network (MSDN): https://msdn.microsoft.com/en-us/library/jj574232(v=vs.113).aspx

Microsoft, MSDN, Introduction to Entity Framework. (23. 9. 2017). *Introduction to Entity Framework | MDSN*. Noudettu osoitteesta MSDN: https://msdn.microsoft.com/en-us/library/aa937723(v=vs.113).aspx

Microsoft, MSDN, Requirement for Creating POCO Proxies. (14. 4. 2018). *MSDN - Requirement for Creating POCO Proxies*. Noudettu osoitteesta Microsoft Developer Network (MSDN): https://msdn.microsoft.com/en-us/library/dd468057(v=vs.110).aspx

Microsoft, MSDN, Solutions and Projects in Visual Studio. (14. 4. 2018). *Solutions and Projects in Visual Studio | MSDN*. Noudettu osoitteesta Microsoft Developer Network (MSDN): https://msdn.microsoft.com/fi-fi/library/b142f8e7.aspx

Microsoft, MSDN, The Repository Pattern. (22. 4. 2018). *The Repository Pattern | MSDN*. Noudettu osoitteesta MSDN: https://msdn.microsoft.com/en-us/library/ff649690.aspx

Microsoft, What is .NET. (21. 4. 2018). *What is .NET | Microsoft*. Noudettu osoitteesta Microsoft: https://www.microsoft.com/net/learn/what-is-dotnet

Mozilla, MDN Web Docs, JavaScript. (21. 4. 2018). *JavaScript | MDN*. Noudettu osoitteesta MDN (Mozilla Developer Network) Web Docs: https://developer.mozilla.org/bm/docs/Web/JavaScript

Nilsson, J. (2006). *Applying Domain-Driven Design and Patterns, With Examples in C# and .NET.* -: Addison Wesley Professional.

Oikeusministeriö. (14. 4. 2017). *Eu:n tietosuja asetukselle lopullinen hyväksyntä - Oikeusministeriö*. Noudettu osoitteesta Oikeusministeriö: http://www.oikeusministerio.fi/fi/index/ajankohtaista/tiedotteet/2016/04/euntietosuoja-asetuksellelopullinenhyvaksynta.html

Oikeusministeriö. (22. 5. 2017). *Oikeusministeriö.* Noudettu osoitteesta Kysymyksia_ja_vastauksia_tietosuojasta.pdf - Oikeusministeriö: http://www.oikeusministerio.fi/material/attachments/om/valmisteilla/lakihankkeet/informaatio-oikeus/xB1Vydn8T/Kysymyksia_ja_vastauksia_tietosuojasta.pdf

Oracle, Oracle docs, Schema Objects. (22. 4. 2018). *Schema Objects | Oracle docs*. Noudettu osoitteesta Oracle docs: https://docs.oracle.com/cd/B19306_01/server.102/b14220/schema.htm#i22627

Pääsihteeristö, EU. (22. 5. 2017). *Tietosuoja-asetus - Consilium*. Noudettu osoitteesta Euroopan unionin neuvosto: http://www.consilium.europa.eu/fi/policies/data-protection-reform/data-protection-regulation-infographics/

Salesforce - What is CRM? (10. 5. 2018). *What is CRM? — Customer Relationship Management - Salesforce EMEA*. Noudettu osoitteesta Salesforce: https://www.salesforce.com/eu/learning-centre/crm/what-is-crm/

Solteq, inWorks-koulutusmateriaali, Tauno Hyvärinen. (22. 4. 2018). inWorks akatemia, inWorks-arkkitehtuuri.

Taloushallintoliitto, Audit trail aukoton kirjausketju. (30. 4. 2017). *Taloushallintoliitto*. Noudettu osoitteesta Audit trail aukoton kirjausketju | Taloushallintoliitto: https://taloushallintoliitto.fi/kirjanpidon-abc-mita-jokaisen-tulisi-tietaa-kirjanpidosta/paakirjanpito-ja-osakirjanpidot/audit

TechTerms, API (Application Programming Interface) Definition. (21. 4. 2018). *API (Application Programming Interface) Definition | TechTerms*. Noudettu osoitteesta TechTerms: https://techterms.com/definition/api

Tietosuojavaltuutettu. (21. 2. 2018). *Tietosuojavaltuutettu*. Noudettu osoitteesta EU:n tietosuojauudistus - Tietosuojavaltuutettu: http://www.tietosuoja.fi/fi/index/euntietosuojauudistus.html

Tietosuojavaltuutettu. (2017. 1. 24). *Uusi opas auttaa rekisterinpitäjiä EU:n tietosuoja-asetukseen valmistautumisessa - Tietosuojavaltuutettu*. Noudettu osoitteesta Tietosuojavaltuutetun toimisto web site: http://www.tietosuoja.fi/fi/index/ajankohtaista/tiedotteet/2017/01/uusiopasauttaarekisterinpitajiaeuntietosuoja-asetukseenvalmistautumisessa.html