Markus Pasanen

SUUNNITTELUMALLIT VERKKO-SOVELLUKSESSA

Opinnäytetyö

Liiketalouden ammattikorkeakoulututkinto

Tietojenkäsittely

2022

Tutkintonimike	Tradenomi (AMK)
Tekijä/Tekijät	Markus Pasanen
Työn nimi	Suunnittelumallit verkkosovelluksessa
Toimeksiantaja	Oy Gambit Labs Ab
Vuosi	2022
Sivut	34 sivua
Työn ohjaaja	Janne Turunen

TIIVISTELMÄ

Opinnäytetyön tavoitteena oli kehittää olemassa olevaan verkkosovellukseen yksinkertainen käyttäjähallinta. Toimeksiantajana työlle oli IT-alan yritys Oy Gambit Labs Ab. Raportti kuvailee yleisellä tasolla Microsoftin .NET-teknologioita verkkosovellusten kehityksessä sekä miltä erilaiset suunnittelumallit näyttävät käytännön projektissa. Varsinainen toteutus oli uuden ominaisuuden toteuttaminen yrityksen sisäiseen projektiin, jonka tarkoitus oli toimia eri tiimien yhteishengen rakentajana projektipäällikön apuna.

Suunnittelumallit käytiin läpi ensin komponenttitasolla ja sen jälkeen arkkitehtuurin tasolla, jolloin saatiin kokonaiskuva siitä, miten eri palaset ovat vuorovaikutuksissa toisiinsa kokonaisen verkkosovelluksen sisällä. Näistä selvisi myös tärkeänä osana, etteivät suunnittelumallit ole konkreettinen itseisarvo hyvän ohjelmistosuunnittelun kannalta vaan ennemminkin mallipohja, joita yhdistelemällä saadaan aikaiseksi hyvä lopputulos.

Koodaustyö toteutettiin Jetbrainsin työkaluja monipuolisesti avuksi käyttäen, ja kehitysympäristössä avuksi otettiin myös Docker-konttiin pystytettävä testitietokanta, jolloin sovellusta pystyttiin testaamaan mahdollisimman todenmukaisesti, mutta silti helposti oikeaa tietoa vasten. Versionhallinnan käyttäminen teki kehitysympäristön pystyttämisestä jouheaa ja mahdollisti muutosten vertaamisen alkuperäiseen ratkaisuun helposti, jolloin koodikatselmointi voitiin suorittaa jouhevasti.

Projektilla ei ollut varsinaista aikataulua, mutta käyttäjäpalaute oli positiivista ja toteutus tarjoaa hyvän pohjan jatkokehitykselle tarvittaessa. Myös varsinainen kehitysongelma ratkaistiin riittävällä tavalla ja lopputulos saatiin toimintakelpoisena suoraan käyttäjien käytettäväksi.

Asiasanat: ohjelmistoarkkitehtuuri, ohjelmistosuunnittelu, suunnittelumallit, entity framework, MVC

Degree title	Bachelor of Business Administration
Author	Markus Pasanen
Thesis title	Design patterns in a web application
Commissioned by	Oy Gambit Labs Ab
Time	2022
Pages	34 pages
Supervisor	Janne Turunen

ABSTRACT

The thesis aimed to develop a simple user management system for an existing web application. The project was commissioned by Oy Gambit Labs Ab. The report described Microsoft's .NET technologies in the development of web applications at a general level, as well as how different design patterns look and feel in a practical project. The actual implementation was the creation of a new feature for an internal company project, the purpose of which was to act as a builder of the common team spirit of different teams helping the project manager.

The design patterns were reviewed first at the component level and then at the architecture level, giving an overall picture of how the different pieces interact within a complete web application. This showed as an important part that the design patterns themselves were not a concrete truth in terms of good software design, but rather an abstraction or a template, and a good result can be achieved by combining them.

The coding work was carried out using JetBrains's tools in a versatile manner. The test database was set up inside a Docker container and was also used to ease the development, so that the application could be tested as realistically as possible, but still easily against the right information. Using version control made setting up the development environment easy and made it possible to easily compare changes to the original solution, so that code review could be done easily.

The project did not have an actual schedule, but user feedback was positive and the implementation provides a good basis for further development. Also, the actual development problem was dealt with sufficiently and the result is ready to use for the end user.

Keywords: software architecture, software design, design patterns, entity framework, MVC

SISÄLLYS

1 JOHDANTO 5
2 TYÖKALUT JA TEKNOLOGIAT 6
2.1 .NET Core 6
2.2 Razor Pages 7
2.3 Entity Framework Core 7
2.4 MySQL 9
2.5 Docker 10
2.6 Rider 11
3 SUUNNITTELUMALLI 12
3.1 Mikä on suunnittelumalli? 12
3.2 Miten suunnittelumalli valitaan? 14
3.3 SOLID-periaatteet 15
3.4 Antisuunnittelumalli 18
3.5 MVC-Malli 20
3.6 ORM 21
3.7 Sipulimalli 22
4 TOTEUTUS 24
4.1 Projektinhallinta 24
4.2 Työkalujen asennus ja työtilan pystytys 25
4.3 Ohjelmointi 27
4.4 Toiminnallisuuden testaaminen ja julkaisu 29
5 PÄÄTÄNTÖ 31
LÄHTEET 32

1 JOHDANTO

Tämän opinnäytetyön tavoite on esitellä .NET-ympäristöön kuuluvien työkalujen käyttöä oikean maailman projektissa sekä perustellusti kuvailla suunnittelumallin implementointi. Opinnäytteen sisältöön kuuluu varsinaisten työkalujen sekä -ympäristön esittely, yleinen kuvaus projektissa käytetystä suunnittelumallista sekä lopuksi raportti varsinaisesta implementaatiotyöstä.

Toimeksiantajana työlle toimii Gambit, joka on nykyään Atea Oy:n omistuksessa. Opinnäytetyössä esitetty kokonaisuus on osa sisäistä projektia, jonka tarkoitus on kartoittaa leikkimielisesti viikoittain opittuja asioita tiimien kesken ja tarjota näin projektipäälliköille mahdollisuuden selvittää eri tiimien osaamistasoa teknologioissa sekä yleistä jaksamista.

Työn tarkoitus on keskittyä riittävien määreiden keräämiseen käyttäjähallinnan toteutusta varten sekä varsinaisen implementaatiotyön suorittaminen. Opinnäytetyössä myös käydään yleisellä tasolla läpi käytettyjä työkaluja, -ympäristöä sekä suunnittelumallia. Mallin valintaa sekä varsinaista implementaatiota tutkitaan käytännön tasolla ja annetaan esimerkki, miten eri abstraktioita voidaan pitää hyvien ohjelmointitapojen mukaisesti erillään varsinaisesta datasta.

Luvussa 2 on esitelty yleiset työkalut ja teknologiat toteutukseen. Luvussa 3 esitellään suunnittelumallien teoriaa. Seuraavaksi raportissa esitellään käytännön toteutus ja lopuksi tiivistetään asiat päätäntöluvussa. Työstä on rajattu pois tekoälymallien sekä -teknologioiden kuvaus, koska ne eivät varsinaisen käyttäjähallinnan lisäämiseen tai kuvailuun tuo lisäarvoa.

2 TYÖKALUT JA TEKNOLOGIAT

2.1 .NET Core

.NET Core on Microsoftin ja .NET-yhteisön yhdessä kehittämä avoimeen lähdekoodiin perustuva kehitysympäristö, joka on modulaarisempi ja nykyaikaisempi vaihtoehto vanhalle .NET-kehitysympäristölle. Kehystä voidaan käyttää joustavasti rakentamaan erilaisia sovelluksia verkkosovelluksista aina IoT:n (internet of things) saakka. (.NET Core Overview 2022.)

.NET Core:n tarkoitus on olla huomattavasti kevyempi ja modulaarisempi ratkaisu, joka sopii mahdollisimman moneen eri käyttötarkoitukseen monella eri alustalla. Muina ominaisuuksina kehys sisältää NuGet-pakettienhallinnan, jolla uusia moduuleja voidaan lisätä sovellukseen mahdollisimman optimaalisesti. (.NET Core Overview 2022.)

Avoin lähdekoodi on Microsoftin itsessään ylläpitämä, jolloin tuki ja koodipohjan kehitys pysyvät hyvin hallinnassa. Kehys on myös täysin järjestelmäriippumaton, jolloin se sopii huomattavasti vanhaa .NET-ohjelmistokehystä paremmin monialustajulkaisuja varten. Sama koodipohja voidaan rakentaa yhtä aikaa, vaikka kolmelle käyttöjärjestelmälle ja lopputulos pysyy yhdenmukaisena. .NET Core tukee kolmea eri ohjelmointikieltä: C#, F# ja Visual Basic. (.NET Core Overview 2022.)

Modulaarinen rakenne yhdessä NuGet-paketinhallinnan kanssa mahdollistaa laajan skaalan aina prototyypeistä suuriin tuotantosovelluksiin saakka ja projektin skaalaaminen onnistuu helposti tämän edetessä. Paketinhallinta myös pitää parhaansa mukaan huolen suorituskyvystä, jolloin suorituskykyongelmien riski pienenee huomattavasti. (.NET Core Overview 2022.)

Ympäristö sisältää komentorivityökalut sisällään vakiona, joiden avulla esimerkiksi versionhallinnan ja julkaisujen tekeminen helpottuu, koska nämä voidaan tehdä yksinkertaisesti kehitystyökalujen sisällä. Julkaisuja helpottaa myös

laaja yhteensopivuus Docker-konttiympäristön kanssa. (.NET Core Overview 2022.)

2.2 Razor Pages

Razor Pages on Microsoftin kehittämä käyttöliittymäkehys .NET CORE ympäristöön, jonka tarkoitus on selkeyttää ja yksinkertaistaa käyttöliittymän implementointia .NET:n sisällä luotuihin sovelluksiin. Erona MVC-malliin on muun muassa kansiorakenne. Kyseisen käyttöliittymän etuja ovat muun muassa seuraavat:

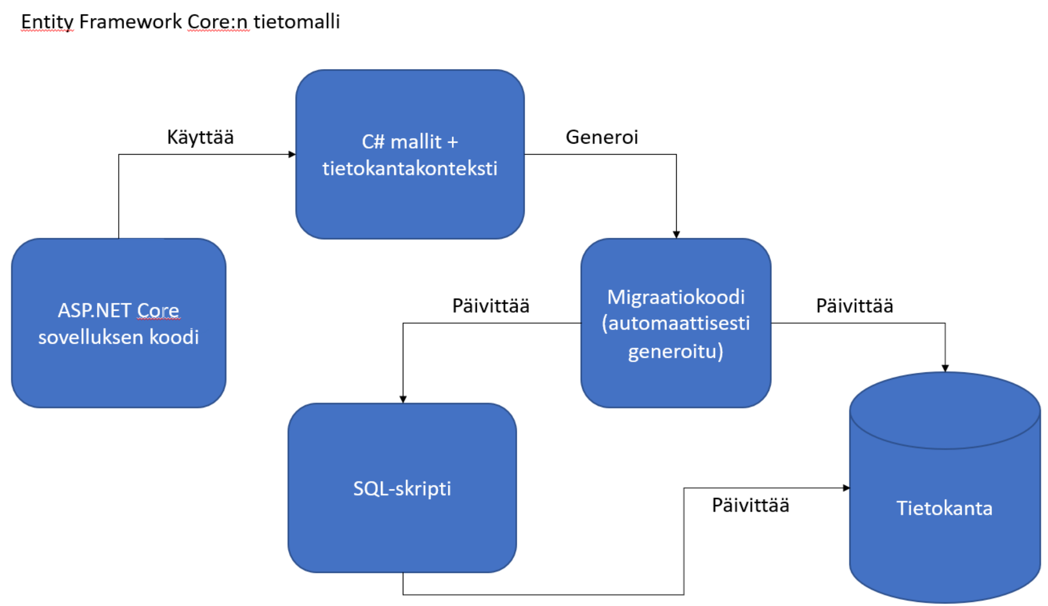
1. Kehys on järjestelmäriippumaton, joten kerran toteutettu käyttöliittymä voidaan kääntää sellaisenaan useille eri käyttöjärjestelmille kuten Windows, MacOS ja Unix.
2. Syntaksi on helppo omaksua varsinkin, jos käyttäjällä on ennestään kokemusta .NET:ssa käytettävistä kielistä.
3. Kevyt ja helposti laajennettava kehys, joka toimii sekä nopeisiin prototyyppeihin ja suurempiin toteutuksiin
4. Toimii suoraan C# ohjelmointikielen kanssa oman Razor merkintäkielen avulla.
5. Koodin organisointi on helpompaa kuin ennen. (Introduction To ASP.NET Core Razor Pages, 2021)

Razor Pages sopii kaiken tasoisille kehittäjille aina aloittelevista koodareista yritysmaailman konkareihin. Se perustuu sivukeskeiseen kehitysmalliin, joka tarjoaa tutun oloisen kehyksen vanhoille verkkokehittäjille, joilla on kokemusta muista sivukeskeisistä kehyksistä, kuten PHP, Classic ASP, Java Server Pages, ASP.NET Web Pages sekä ASP.NET Web Forms. Kehys on myös suhteellisen helppo oppia ja omaksua aloittelijan näkökulmasta, ja se sisältää lisäksi kaikki ASP.NET Coren edistyneet ominaisuudet (kuten riippuvuusinjektion), joten se sopii hyvin myös suuriin, skaalautuviin, tiimipohjaisiin projekteihin pienempien sovelluksien lisäksi. (Learn Razor Pages 2021.)

2.3 Entity Framework Core

Entity Framework Core on avoimen lähdekoodin ORM (Object–relational mapping) kehys .NET CORE-kehitysympäristöön. Tarkoituksena on luoda abstraktiota palvelinpään ja tietokannan välille, jolloin suoria tietokantakutsuja ei koskaan suoriteta palvelinpäässä tai varsinkaan käyttöliittymän puolella. (Entity Framework – Overview 2022.)

Tarkoitus on vähentää varsinaisen datan sekoittumisen koodin abstraktioihin eri ohjelmistokerrosten välillä ja siten hyvien ohjelmointitapojen mukaisesti pitää komponentit mahdollisimman riippumattomina toisistaan. Käytännössä tietokantataulut kartoitetaan olioihin relaatioiden perusteella ja kehys hoitaa varsinaiset tietokantaoperaatiot optimoidusti ja tietoturvallisesti, jolloin kehittäjän tehtäväksi jää ainoastaan olioiden käsittely. Tiedon kulkeminen on havainnollistettu kuvassa 1. (SOLID: The First 5 Principles of Object Oriented Design 2020.)

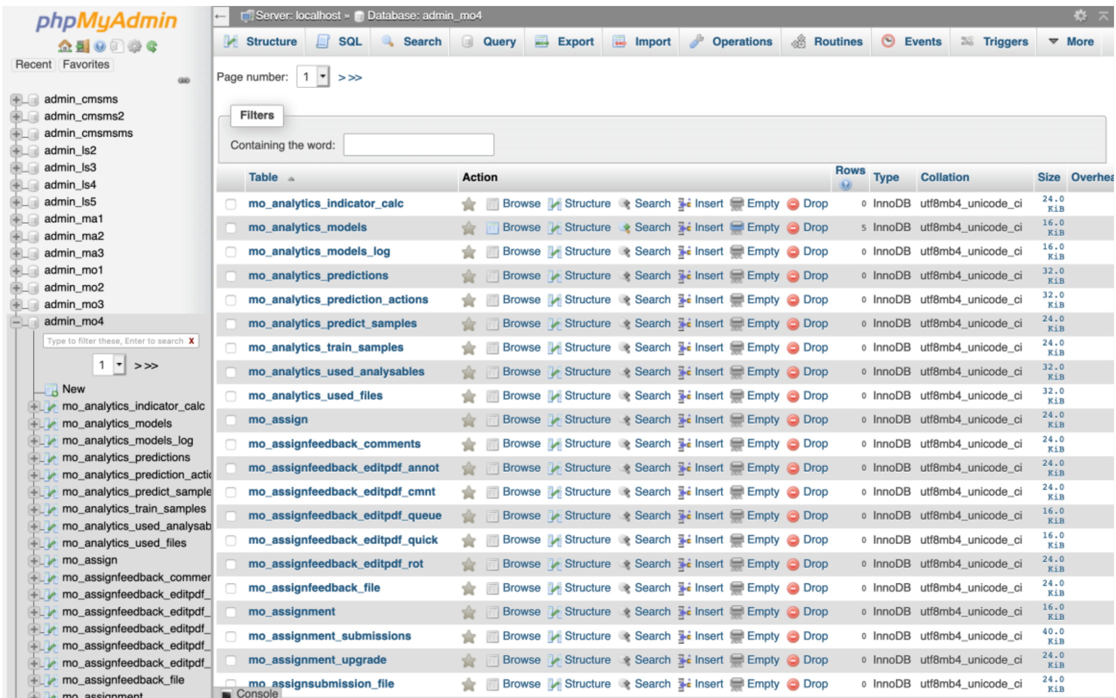


Kuva 1. Esimerkki tiedon kulkemisesta Entity Frameworkin sisällä

Entity framework tukee pääosin kolmea erilaista tietomallia, jotka ovat: mallin generointi olemassa olevasta tietokannasta, mallin käsin kirjoittaminen peilaamaan tietokantaa ja tietokannan generointi olemassa olevasta mallista. Tietojenkäsittely hoidetaan olioiden ja luokkien avulla, jotka muodostuvat entiteetti- sekä kontekstiluokista. Kontekstiluokasta luotu olioinstanssi käytännössä edustaa tietokantasessiota. Tämän avulla voidaan muokata ja lisätä tietoa tietokantaan ilman että ohjelmoijan tarvitsee kirjoittaa käsin tietokantaoperaatioita. Kehys tällöin itsessään pyrkii hoitamaan varsinaisen abstraktion tietokannan ja rajapinnan välillä parhaansa mukaan. (Entity Framework Core 2021.)

2.4 MySQL

MySQL on suhteellisen uusi tulokas relaatiotietokannan hallintajärjestelmien (RDBM) joukkoon, jonka konseptin kehitti IBM:n tutkija Edgar Frank Codd vuonna 1970. Huolimatta uudempien tietokantateknologioiden saapumisesta viimeisten 35 vuoden aikana ovat relaatiotietokannat pysyneet silti tietotyön työjuhtina. Näiden avulla käyttäjät voivat muodostaa hyvinkin monimutkaisia suhteita ja kytköksiä tietueitten välillä ja laskea näiden väliset suhteet päätöksentekoon riittävällä nopeudella nykyaikaisissa organisaatioissa. Tämä mahdollistaa nopean siirtymisen suunnitteluvaiheesta toteutusvaiheeseen vain muutamassa tunnissa ja käyttäjä kykenee helposti kehittämään verkkosovelluksia, jotka pääsevät käsiksi teratavuihin dataa ja palvelevat tuhansia verkon käyttäjiä sekunneissa. (Tahaghoghi & Williams 2007, 3.)



Kuva 2. Esimerkki MySQL-tietokannan hallinnasta phpMyAdmin työkalulla

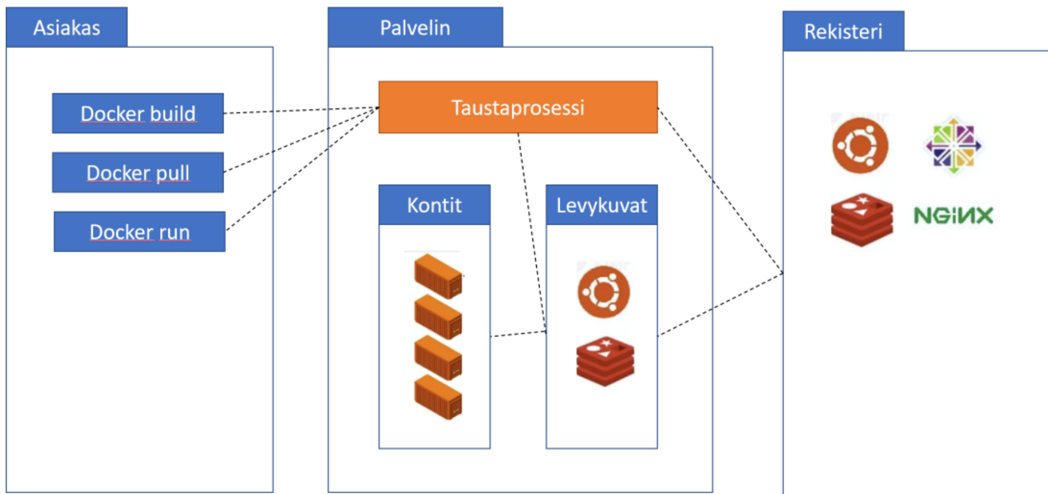
MySQL:n SQL-osa tulee sanoista "Structured Query Language", joka on yleisin standardisoitu kieli, jota käytetään tietokantojen käsittelyssä. Työympäristön mukaan voidaan SQL-lauseke kirjoittaa suoraan esimerkiksi raporttien luomiseksi, upottaa SQL-lauseita toisella ohjelmointikielellä kirjoitettuun sovellukseen tai käyttää kielikohtaista rajapintaa, joka piilottaa SQL-syntaksin käyttäjän nähtäviltä. Kuvassa 2 on havainnollistettu MySQL tietokannan hallintaa graafisella työkalulla. (What is MySQL? 2022.)

MariaDB

MariaDB on kehityshaara MySQL ja se toimii käytännössä suoraan SQL:n tilalla. Voidaan korvata tavallisen MySQL-palvelimen MariaDB-palvelimen analogisella versiolla ja hyödyntää täysimääräisesti tämän tuomia parannuksia ilman, että kehittäjän tarvitsee muokata sovelluskoodia. MariaDB on nopeampi, skaalautuvampi ja vakaampi kuin normaali MySQL ja siksi se onkin suosittu teollisuuden tarpeisiin. Se tukee myös enemmän tallennusmoottoreita kuin MySQL. Sun Microsystems osti MySQL:n vuonna 2008. Tämän jälkeen Oracle osti Sun Microsystemsin vuonna 2010 MySQL:n kanssa. Jostain syystä Michael Monty Widenius, joka oli alkuperäinen MYSQL:n perustajajäsen, loi uuden kehityshaaran tästä ja loi oman yrityksen nimeltä Monty Program AB. Hän nimesi MariaDB:n toisen tyttärensä Marian mukaan. MariaDB-säätiö perustettiin joulukuussa 2012, koska Monty ei halunnut, että MariaDB:tä voisi ostaa samalla tavalla kuin MySQL:n kanssa kävi. MariaDB-säätiön tarkoituksena on tukea MariaDB:n jatkuvuutta ja toimia globaalina avoimen kehityksen kontaktipisteenä. (What is MariaDB 2022.)

2.5 Docker

Docker esitteli vuonna 2013 konttiteknologian, joka tulisi myöhemmin olemaan alan standardi sovelluksien eristämiseen. Docker-kontit ovat nykyään standardoitu osa ohjelmistokehitystä, jonka avulla kehittäjät voivat eristää sovelluksensa ympäristöstään. Miljoonille kehittäjille Docker on yksi tärkeimmistä työkaluista heidän työkalupakissaan, jolla voidaan jakaa sovelluksia muille työpöydältä aina ohjelmistopilveen saakka. (Containers were just the Beginning 2022.)



Kuva 3. kuvailu Docker-konttien arkkitehtuurista

Kontti on tavallaan riisuttu virtuaalikone, johon kehittäjä voi asentaa haluamansa käyttöjärjestelmän riisutun version. Ideana on täyttää ainoastaan yksittäisen sovelluksen käyttöjärjestelmävaateet. Kehittäjä luo sovelluksestaan docker-levykuvan valitsemallaan käyttöjärjestelmäpohjalla (kuva 3). Kuva itsessään ei ole muuta kuin resepti, jolla voidaan pystyttää useita kopioita samasta sovelluksesta tämän vaatimilla ympäristömuuttujilla ja muilla vaateilla. Nyt kehittäjä, testaaja ja esimerkiksi vaikkapa järjestelmänvalvoja voivat käyttää luotua kuvaa ottaakseen käyttöön identtisen version sovelluksesta. (Chaturvedi 2021.)

2.6 Rider

Rider on IDE (Integrated Development Environment), joka tukee .NET-ohjelmistokehystä, uutta monialustaista .NET Corea ja Mono-pohjaisia projekteja. IDE:n avulla voit kehittää laajan skaalan erityyppisiä sovelluksia, kuten .NET-työpöytäsovelluksia, palveluita ja kirjastoja, Unity-videopelejä, Xamarin-sovelluksia, ASP.NET- ja ASP.NET Core -verkkosovelluksia. Lisäksi Riderissa on tehokkaita virheiden korjausta avustavia järjestelmiä, ja se toimii useilla alustoilla: Windows, macOS ja Linux. (What is Rider? 2022.)

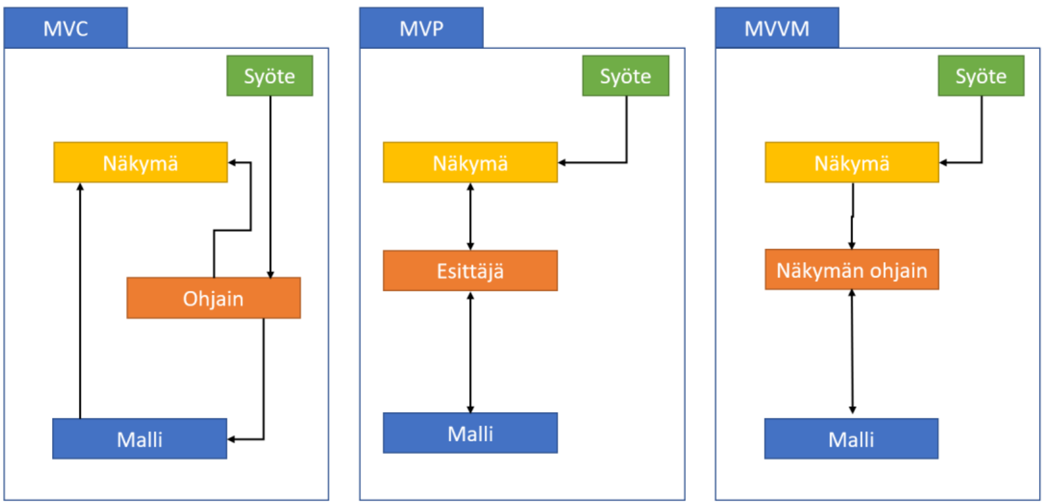
Kuva 4. Rider-työkalun käyttöliittymä

Rider käyttää käyttöliittymää + useita muita ominaisuuksia IntelliJ-alustasta, joka takaa toiminnallisuuden isolle osalle JetBrains:n kehitysalustoja ja työkaluja (kuva 4). Työkalu tarjoaa myös ydintoiminnallisuutenaan sisäänrakennetun tuen eri versionhallintatyökaluille sekä tuen eri tietokannoille. Lisäksi tuki ReSharper:in ominaisuuksiin kuten kehittynyt navigointi, haku, koodin uudelleenjärjestelyn ja muiden ominaisuuksien kanssa auttaa kehittäjiä toteuttamaan suuriakin .NET-projekteja yhdessä. Työkalu on myös huomattavan suorituskykyinen ja tämä lisää sen houkuttelevuutta monen kehittäjän silmissä. (ReSharper + IntelliJ platform 2022.)

3 SUUNNITTELUMALLI

3.1 Mikä on suunnittelumalli?

Suunnittelumallit ovat tyypillisesti malleja ratkaisemaan usein esiintyviä ja toistuvia ongelmia ohjelmistokehityksessä. Niiden voidaan olevan käytännössä reseptejä, miten tietynlainen ongelma ratkaistaan koodin tai projektin sisällä. Esimerkiksi abstraktion ja datan erottelu toisistaan helpottuu huomattavasti oikeanlaisen suunnittelumallin avulla ja hyvien ohjelmointitapojen noudattaminen helpottuu. (What's a design pattern? 2022.)



Kuva 5. Esimerkki yleisistä arkkitehtuurimalleista

Oikeanlaisen mallin valitseminen yleisesti ottaen vaatii kokemusta, mutta projektin ongelmien tunnistaminen pääpiirteittäin auttaa arkkitehtiä tai ohjelmoijaa valitsemaan oikeanlaisen ratkaisun kyseiseen ongelmaan. Mallia valitessa täytyy muistaa, ettei niinkään käytetyt teknologiat tai ohjelmointikieli ole päättävässä asemassa mallia valitessa, vaan yleensä projektin luonne on suurin määrittävä tekijä ratkaisua pohtiessa. Kuvassa 5 on esitelty yleisimmät verkkosovelluksissa käytetyt arkkitehtuurimallit. (What's a design pattern? 2022.)

Suunnittelumalli on helppo sekoittaa algoritmin kanssa, koska molempien konseptien tarkoituksena on ratkaista tietyntyyppinen ongelma. Tämä toisaalta ei ole validi ajattelutapa, koska suunnittelumalli on yleensä abstraktimpi ja enemmän korkeamman tason ajattelumalli tietyntyyppisen ongelman ratkaisuun, kun taas algoritmi on enemmän konkreettinen ratkaisu matalamman tason ongelmaan, kuten vaikkapa esimerkkinä erilaiset lajittelualgoritmit. Voidaan siis ajatella, että algoritmi on enemmän kuin ruokaresepti, joka kertoo tarvittavat raaka-aineet konkreettisesti ja niiden oikeat määrät. Suunnittelumalli taas on enemmän suunnitelma lopputulemasta ja mitä sen tulisi sisältää, mutta toteutus on taas kehittäjän tai arkkitehdin käsissä (What's a design pattern? 2022.)

3.2 Miten suunnittelumalli valitaan?

Vaikka suunnittelumallit ovat tärkeä osa ammattimaista ohjelmistokehitystä, eivät ne silti ole hopealuoti kaikkeen. Niitä tulisi lähinnä käsitellä ohjeistuksena, miten eri komponentit liitetään toisiinsa sovelluksen sisällä, eikä niiden tulisi ajaa sovelluksen varsinaisen käyttötarkoituksen yli tai lisätä ylimääräistä monimutkaisuutta. Suunnittelumallit auttavat kehittäjää ymmärtämään ja hahmottamaan sovelluskehityksen prosessin aikana nousevia haasteita tiedonkulun ja komponenttien asettelun kanssa. Perusideana on, että geneerinen ja useamman henkilön tai tiimin yleisesti käyttämä malli on monella tapaa kestävämpi ratkaisu, kuin yksittäisen kehittäjän itsensä kehittämä arkkitehtuuri. (GolfPatterns 2022.)

Luontimallit	Rakennemallit	Käyttäytymismallit
Epäkonkreettinen tehdas	Sovitin	Yleisfunktio
Rakentajarajapinta	Silta	Tulkki
Tehdasfunktio	Koosteensa sisältävä kooste	Tarkkailija
Prototyyppirajapinta	Kuorruttaja	Vastuuketju
Ainokainen	Julkisivu	Välittäjä
	Hiutale	Iteraatio
	Edustaja	Komento
		Muisto
		Strategia
		Tila
		Vierailija

Kuva 6. Erilaiset suunnittelumallit jaoteltuna kolmeen ryhmään

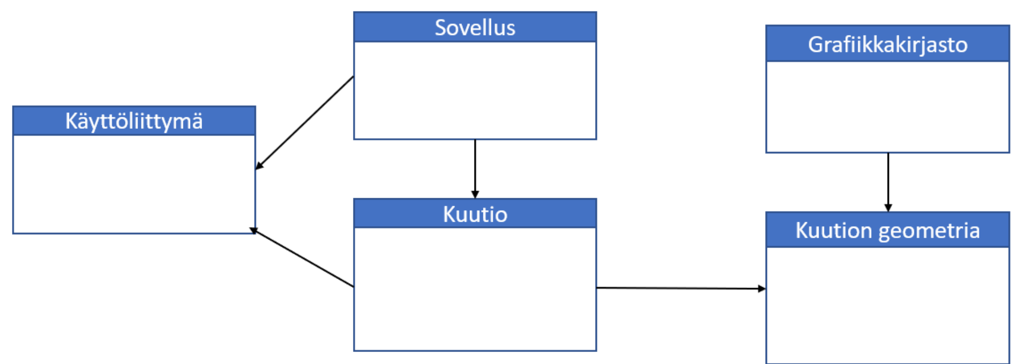
Kun tarkastellaan erilaisten mallien suurta määrää, kehittäjän voi olla vaikea valita paras malli ratkaisemaan juuri hänen kehitysongelmansa. Tällöin kannattaa analysoida, onko ongelman laatu arkkitehtoninen vai ennemminkin toiminnallinen. Toimiva ratkaisu saattaa vaatia kehittäjää sekoittamaan ja soveltamaan useita erityyppisiä suunnittelumalleja keskenään (kuva 6). (GolfPatterns 2022.)

Kaikkien mallien yhtäaikainen käyttö on myös negatiivinen asia, koska silloin sovelluksen kompleksiivisuus kasvaa ylenmääräisesti. Tällöin jokainen ratkaisu on suunniteltu antamaan sopeutuvuutta, jota ei oikeasti tulla koskaan käyttämään. Ohjelmistokehityksen pääasiallinen tarkoitus on kumminkin keskittyä mahdollisimman paljon tietyn ongelman olennaiseen ratkaisemiseen

mahdollisimman yksinkertaisella tavalla, kuitenkin laajentumisen estämättä. (Gamma 2005.)

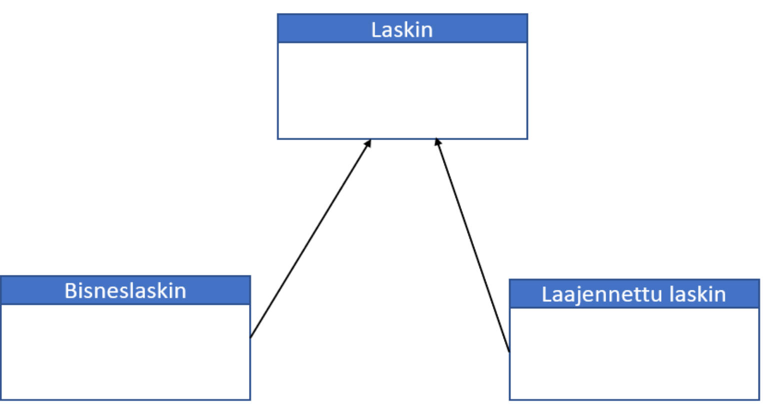
3.3 SOLID-periaatteet

SOLID-periaatteet koostuvat seuraavista: yhden vastuun periaate, avoimen / suljetun luokan periaate, liskovin korvattavuuden periaate, rajapintojen hajoittamisperiaate, riippuvuuden kääntämisen periaate. Nämä periaatteet määrittävät käytännöt, jotka soveltuvat ohjelmistojen kehittämiseen ottaen huomioon ylläpitotarpeet sekä laajenemisen projektin kasvaessa. Näiden käytäntöjen käyttöönotto voi myös auttaa välttämään koodin hajuja, koodin uudelleenmuodostusta. (SOLID: The First 5 Principles of Object Oriented Design 2020.)



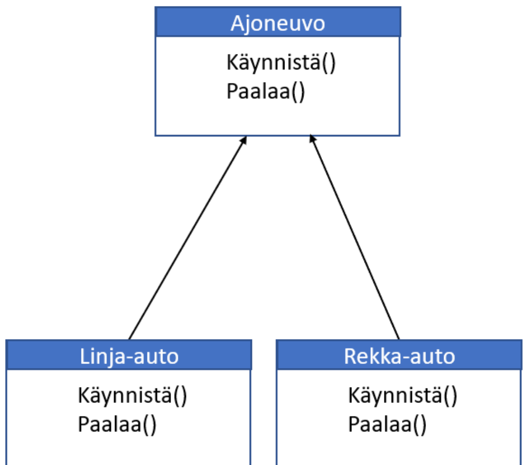
Kuva 7. Kuvailu yhden vastuun periaatteesta

Yhden vastuun periaate kertoo, että luokalla tulisi olla ainoastaan yksi tehtävä, eli ohjelmistomielessä voidaan ajatella luokan vastaavan ainoastaan yhdestä ohjelmiston sisällä toteutettavasta tehtävästä kuten vaikkapa tietokantaan kirjautuminen tai ohjelmiston toiminnan raportointi (kuva 7). Tämä tarkoittaa myös, että luokan toiminnallisuutta tarvitsee muuttaa ainoastaan varsinaisen tietomallin muuttuessa. Tämän ohjelmointiperiaatteen edut tulevat varsinkin esiin useamman kehittäjän työskennellessä samassa projektissa. Mikäli luokan vastuualue on rajattu yhteen asiaan, vähenee mahdollisuus yhteentörmäyksiin ja virheisiin huomattavasti. (The Single Responsibility Principle 2020.)



Kuva 8. Esimerkki periytymisestä avoimen / suljetun luokan periaatteessa

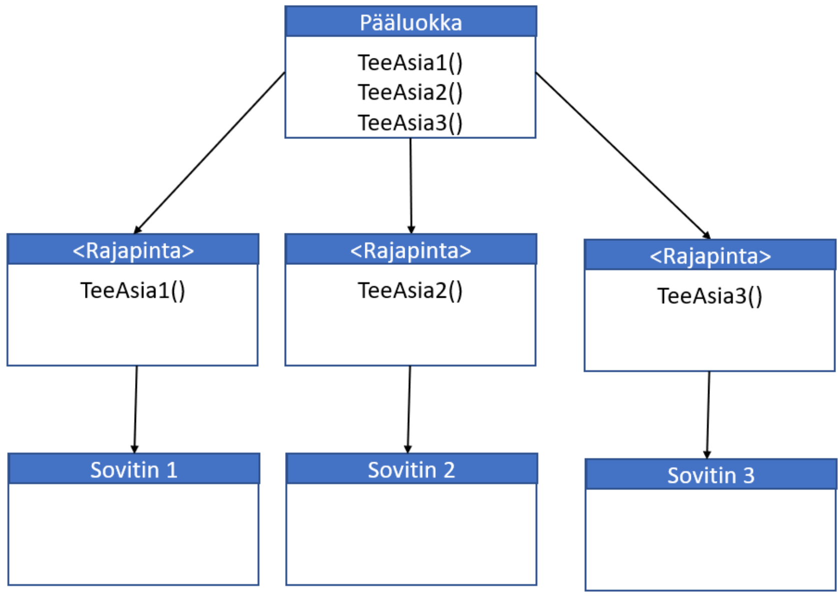
Avoimen / suljetun luokan periaatteen ideana on, että voidaan laajentaa olemassa olevaa toiminnallisuutta ilman muutoksien tekemistä alkuperäiseen luokkaan (kuva 8). Tämä auttaa välttämään tilanteita, joissa yksi luomistasi uusista luokista vaatisi myös riippuvuusluokkien muokkaamista, joka taas johtaa helposti ikäviin bugeihin. Alkuperäisen luokan tulisi aina olla suljettu mutaatioilta, mutta olla silti avoin laajentamista tai uusien toiminnallisuuksien lisäämistä varten. (Janssen 2018.)



Kuva 9. Esimerkki aliluokkien identtisistä funktiokutsuista

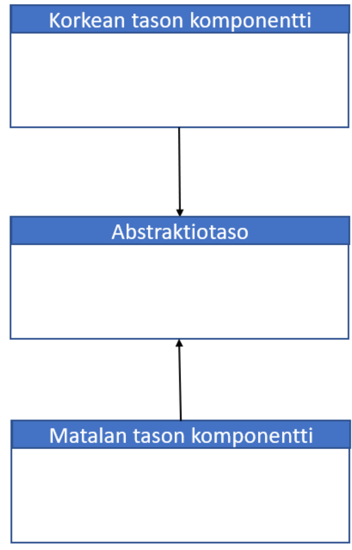
Liskovin korvattavuuden periaate kertoo, että kaikki samasta luokasta periytyvät oliot, jotka sijaitsevat samalla tasolla ovat korvattavissa toistensa kanssa

ilman sovelluksen rikkoontumista. Käytännössä tämä tarkoittaa, että olioinstanssit käyttäytyvät samalla tavalla (kuva 9), eikä vahingossa luotu väärä instanssi pysty rikkomaan sovellusta. (Saxena 2021.)



Kuva 10. Esimerkki hajotetuista rajapinnoista

Rajapintojen hajoittamisperiaatteen, joka tunnetaan myös nimellä ISP perusideana on määrittää, ettei sovellus sisällä massiivisia yleismaailmallisia rajapintoja, vaan nämä olisivat kevyitä yksittäisesti määriteltyjä (kuva 10). Tämän ansiosta luokka sisältää ainoastaan metodeja, joita se oikeasti käyttää eikä sisällä turhia metodikutsuja. (Software Craft 2020.)



Kuva 11. Esimerkki matalan ja korkean tason komponenttien riippuvuuksista

Riippuvuuden kääntämisen periaate (DIP) määrittää, että jokaisen korkeamman tason moduuli ei saa olla riippuvainen matalan tason moduuleista. Molempien tulisi olla riippuvainen abstraktioista. On myös tärkeää, että abstraktio ei koskaan ole riippuvainen yksityiskohdista vaan nimenomaan yksityiskohdat ovat aina riippuvaisia abstraktioista (kuva 11). Komponentit, jotka eivät noudata tätä ovat vaarassa rikkoa toisensa, mikäli muutoksia tehdään yksittäiseen komponenttiin. (Gudabayev 2021.)

3.4 Antisuunnittelumalli

Antisuunnittelumallit tarkoittavat ratkaisuja, jotka on toteutettu projektiin yleensä nopealla aikataululla välittämättä ympärillään olevista käytännöistä. Käytännössä tämä näkyy erikoisina purkkaratkaisuina, joilla ongelma on pyritty ratkaisemaan lyhyellä ja ongelmaan tarkennettuna ratkaisuna, vaikka ratkaisu toimisi sillä hetkellä, kun se on toteutettu. Vaarana on, että ohjelmiston jatkokehitys hidastuu huomattavasti näiden ratkaisujen aiheuttaessa erilaisia virheitä ohjelmiston toiminnassa. Tämä aiheuttaa pahimmillaan koko projektin myöhästymisen aikataulusta, koska jokainen lisätty ominaisuus vaatii olemassa olevan koodin uudelleenohjelmointia. (Lucidchart 2022.)

Spagettikoodi

Spagettikoodi on nimitys koodille, joka on lisätty projektiin yleensä ymmärtämättä vallitsevaa arkkitehtuuria ja suunnittelumalleja. Yleensä spagettikoodia muodostuu projekteissa, jossa useampi kehittäjä työskentelee saman koodipohjan kimpussa pitkän aikaa ja he muokkaavat olemassa olevia komponentteja ja toiminnallisuuksia yhtäaikaisesti yrittäen optimoida omia komponenttejaan. Mitä pidemmälle tämä antisuunnittelumalli pääsee, niin sitä hitaammaksi jatkokehitys muuttuu, koska jokaista uutta komponenttia kohti on vaarassa, että useampi olemassa oleva rikkoutuu tämän seurauksena. (Watts 2020.)

Kultainen vasara

Kultainen vasara tarkoittaa yksiulotteista ajattelua, joissa kaikissa ratkaisuissa yritetään käyttää samoja ohjelmistotekniikoita. Esimerkiksi tietokantoihin erikoistunut kehittäjä saattaa saada tehtäväkseen kehittää yksinkertainen verkkosivu, jonka toiminnallisuus on näyttää pelkästään staattista sisältöä. Ole-

massa olevaan kokemukseensa pohjaten hän suunnittelee aivan ensimmäisenä tietokantaskeeman vaikkei projekti staattiselta luonteeltaan vaatisin ollenkaan edes tietokantaa pohjalle. (Jones 2015.)

Ankkuroituminen

Ankkuroituminen tarkoittaa osaa ohjelmistossa, joka ei tarjoa minkäänlaista lisäarvoa tai toiminnallisuutta projektille. Yleensä nämä ankkurit ovat vielä kalliita lisäpalikoita, jotka ovat suunnitteluhetkellä tuntuneet tärkeiltä osilta projektiin lisäyshetkellä. Nämä lisäykset on yleensä kumminkin tehty vastoin kunnollista teknistä evaluointia. (Source making 2022.)

Kuollut koodi

Kuollut koodi on yleensä jäänne esimerkiksi kehittäjältä, joka ei ole työskennellyt projektin parissa vuosikausiin tai pahimmillaan ei ole edes koko yrityksessä töissä. Käytännössä kuollut koodi on esimerkiksi metodeja, jotka eivät näytä tekevän enää mitään sovelluksen sisällä. Myös kommentit yleensä päätyvät kuolleeksi koodiksi ohjelmistoon, koska ne ovat loppujen lopuksi samanlailla ylläpidettävää koodia, kuin varsinaiset kuvailtavat metoditkin. Kukaan ei oikein uskalla poistaakaan tätä kuollutta koodia, koska kenelläkään ei ole harmainta hajua mikä sen toiminnallisuus on projektissa ja missä kontekstissa. (Parr 2020.)

Jumalolio

Jumalolio on lyhyesti tietorakenne, joka yksinkertaisesti tekee ja tietää liikaa. Kun yhdelle luokalle annetaan useita toisistaan riippumattomia vastuita ja tehtäviä kasvaa tämän koko luvattoman isoksi. Tämän takia toiminnallisuuksien muokkaaminen ja siivoaminen vaikeutuu huomattavasti, koska yksi komponentti saattaa rikkoa todella monia toiminnallisuuksia yhtä aikaa ohjelmiston sisällä. (Mamani 2020.)

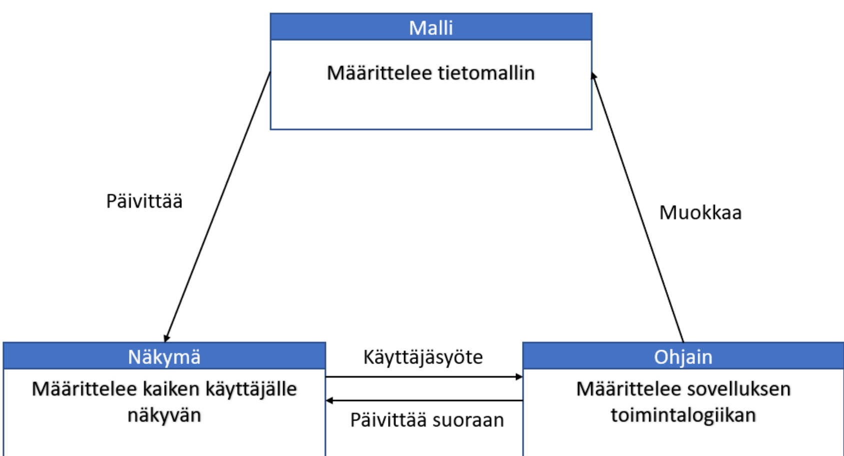
Tämä antisuunnittelumalli pystytään välttämään tehokkaasti käyttämällä aiemmin tekstissä mainittua yhden vastuun periaatetta. Tällöin suurien yleisluokkien muodostuminen on käytännössä mahdotonta ja sovellus pysyy helposti muokattavissa ja laajennettavissa. (sivu 14.)

Kopioi ja liitä -ohjelmointi

Kopioi ja liitä -ohjelmointi on nähtävissä yleensä todella paljon itseään toistavissa metodeissa ja saman rivin toistoista. Yleensä tämä johtuu yksinkertaisesti siitä, ettei kehittäjällä ole riittävää kompetenssia suorittaa ohjelmointitehtävää ja ratkaisu on tehty helpoimman tien kautta. Monesti tämä aiheuttaa bugeja myös sen takia, koska kehittäjä ei ole välttämättä edes täysin varma mitä kyseinen metodi tai komponentti tekee käytännössä ja se edes ole hänen itsensä kirjoittama alun perinkään. Tässä on myös isona riskinä samojen ongelmien toistuminen koodissa aina uudestaan ja uudestaan. (Broadhead 2019.)

3.5 MVC-Malli

MVC (Model-View-Controller) on ohjelmointimalli, jota käytetään yleisesti käyttöliittymien, tietojen ja ohjauslogiikan yhdistämiseen ja toteuttamiseen. Malli korostaa sovelluksen toimintalogiikan ja käyttöliittymän erottamista toisistaan. Tämä helpottaa sovelluksen toiminnallisuuksien hahmottamisen ja ylläpitämisen erottelemalla sovelluksen toimintaa eri paikkoihin. Myös MVC-malliin (kuva 12) perustuvia muita suunnittelumalleja kuten MVVM (Model-View-Viewmodel), MVP (Model-View-Presenter) ja MVW (Model-View-Whatever) käytetään yleisesti. (MVC 2022.)



Kuva 12. Esimerkki MVC-mallin tietomallista

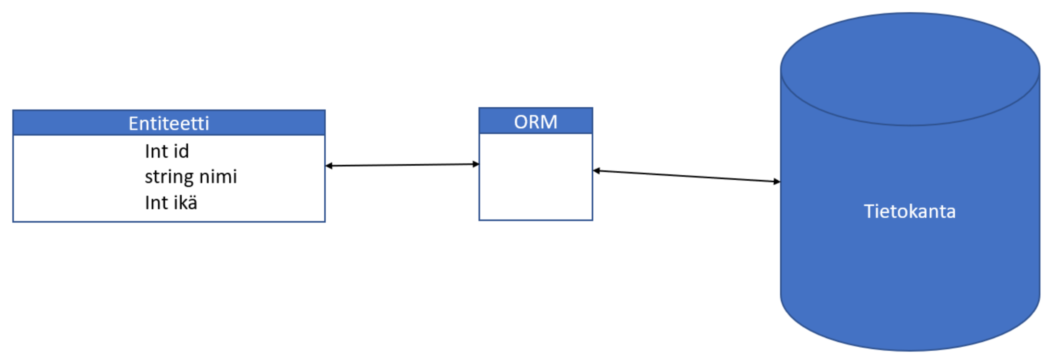
Malli (model) normaalisti kuvastaa oikeassa maailmassa olevia entiteettejä. Tämä koodi voi sisältää raakaa tietoa tai se voi määrittää oleellisia komponentteja sovelluksen sisällä. Esimerkkinä to-do sovelluksen sisällä malli:n sisällä oleva koodi voi määrittää, että mikä ”tehtävä” on. (Wrapping up 2022.)

Näkymä (view) pitää sisällään käytännössä kaiken käyttäjälle näkyvän ja hallitsee myös käyttöliittymän toiminnallisuuksia. (Wrapping up 2022.)

Ohjain (controller) Ohjainkoodi toimii käytännössä yhteyshenkilönä mallin ja näkymän välillä, vastaanottaen käyttäjän syötteitä päättäen, mitä sille tehdään. Tämä taso on sovelluksen aivot ja yhdistää mallin ja näkymän koodin keskenään. (Wrapping up 2022.)

3.6 ORM

Oliorelaatioiden kartoittaja tarjoaa oliopohjaisen kerroksen relaatiotietokantojen ja olio-ohjelmointikielellä kirjoitetun koodin välille ilman, että kehittäjän tarvitsee kirjoittaa suoria SQL-kyselyitä. Standardisoidut liitännät vähentävät boilerplate-koodin määrää ja nopeuttaa huomattavasti ohjelmointiin käytettyä aikaa. Olio-ohjelmointi sisältää monia erilaisia tila, tilakoneita ja muunlaisia toiminnallisuuksia, jotka voivat olla ajoittain vaikeita tulkita ja ymmärtää kehittäjän näkökulmasta. ORM ratkaisee tämän kytkemällä nämä tiedot jäsennellyn kartan avulla suoraan tietokantaan (kuva 13), jolloin kehittäjä voi keskittyä ominaisuuksien luontiin ja antaa kartoittajan hoitaa varsinaiset relaatiot tietokantaan. (Liang 2021.)

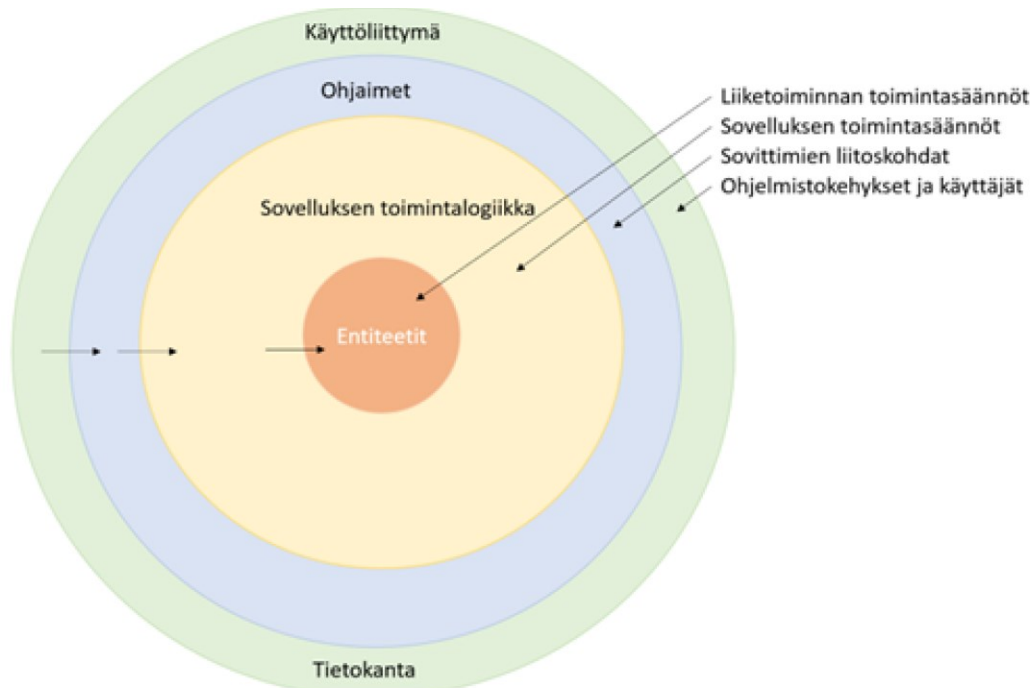


Kuva 13. Kuvassa näkyy ORM toiminnallisuus tietokannan ja sovelluksen välillä

ORM tarjoaa yksinkertaisen rajapinnan olioiden tallentamiseen ja hakemiseen suoraan relaatiotietokannasta ja sieltä takaisin. Työkalun avulla sovellus käsittelee tietoa olioina sen sijaan, että se käyttäisi tietokantakohtaisia käsitteitä, kuten rivit, sarakkeet ja taulukot. (Varma 2022.)

3.7 Sipulimalli

Suurin osa perinteisistä arkkitehtuureista herättää perustavanlaatuisia kysymyksiä liian tiiviistä kytkennöistä ja niistä muodostuvista riippuvuussilmukoista. Sipulimallin esitteli Jeffrey Palermo tarjotakseen paremman tavan rakentaa sovelluksia testattavuuden, ylläpidettävyyden ja luotettavuuden näkökulmasta. (Understanding Onion Architecture 2018.)



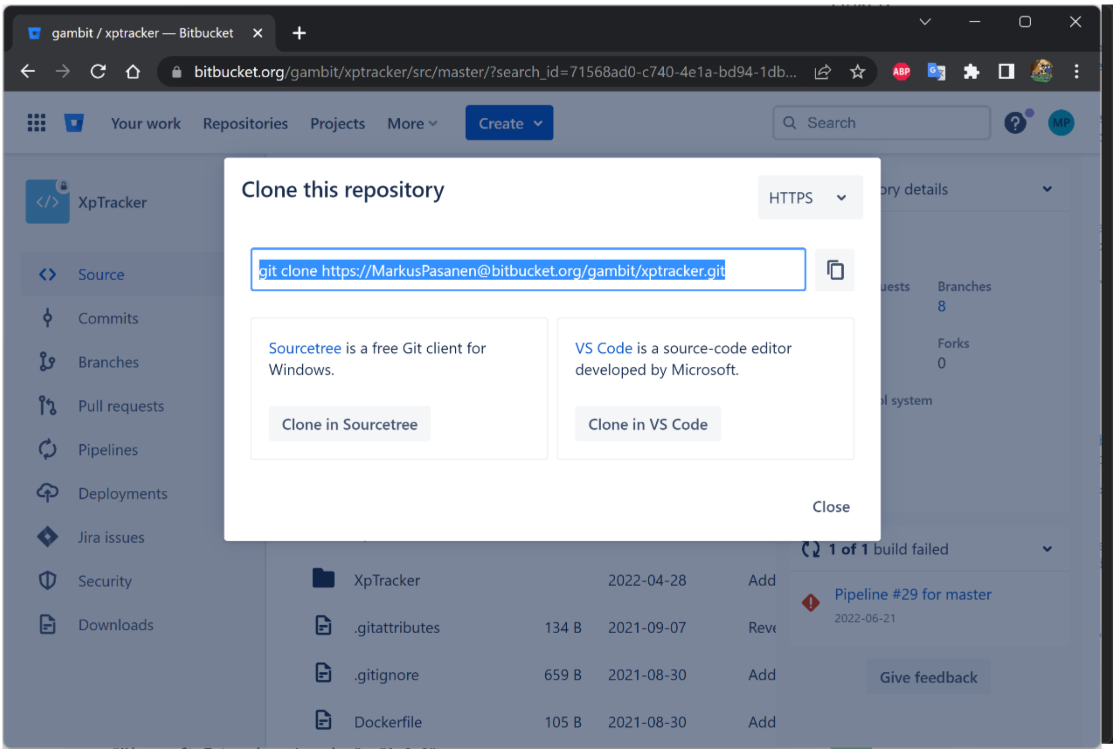
Kuva 14. Havainnekuva sipumallista

Sipuliarkkitehtuurin eri kerrokset ovat vuorovaikutuksessa toistensa kanssa porttien ja sovittimien avulla, kuten kuvassa 14 esitetään. Entiteettitaso on mallin sisäisin ja keskeisin taso, jolla voi olla riippuvuuksia ainoastaan itsensä kanssa. Tämä edustaa liiketoiminnan entiteettejä ja niiden käyttäytymistä ohjelmistossa. Toimintalogiikan taso sisältää ohjainkerroksessa määriteltyjen toimintamallien toiminnallisuuden varsinaisen implementaation. Tällöin saadaan eroteltua varsinainen toiminnallisuus muista riippuvuuksista, joka mahdollistaa aidosti irrallisten toiminnallisuuksien kasaamisen. Ohjaintaso on silta ulkoisen infrastruktuurin ja toimintalogiikan välillä. Ohjelmiston toimintalogiikkakerros tarvitsee usein tietoja tai toimintoja toteuttaakseen oman toiminnallisuutensa, mutta ei saisi olla riippuvainen näistä. Tämän takia ohjainkerros välittää tarvittavat tiedot ja siivoaa ne tarvittaessa. Käyttöliittymätaso on sovelluksen ulkoisin taso, joka voi käytännössä sisältää mitä vain. (Understanding Onion Architecture 2018.)

4 TOTEUTUS

4.1 Projektinhallinta

Toteutus tehtiin toimeksiantona Oy Gambit Labs Ab -nimiselle ohjelmistoalan yritykselle. Yritys on perustettu vuonna helmikuussa 2011 Vaasassa ja työllistää nykyään 50 ihmistä. Pääasiallinen toimiala on yleisen järjestelmäkehityksen, integraatioiden sekä data-analytiikan puolelta, joita toteutetaan pääosin konsultointina avainasiakkaille. Tarkoituksena oli toteuttaa yrityksen sisäisessä käytössä olevaan sovellukseen uutena ominaisuutena vanhojen käyttäjien poistaminen ja uusien lisääminen.

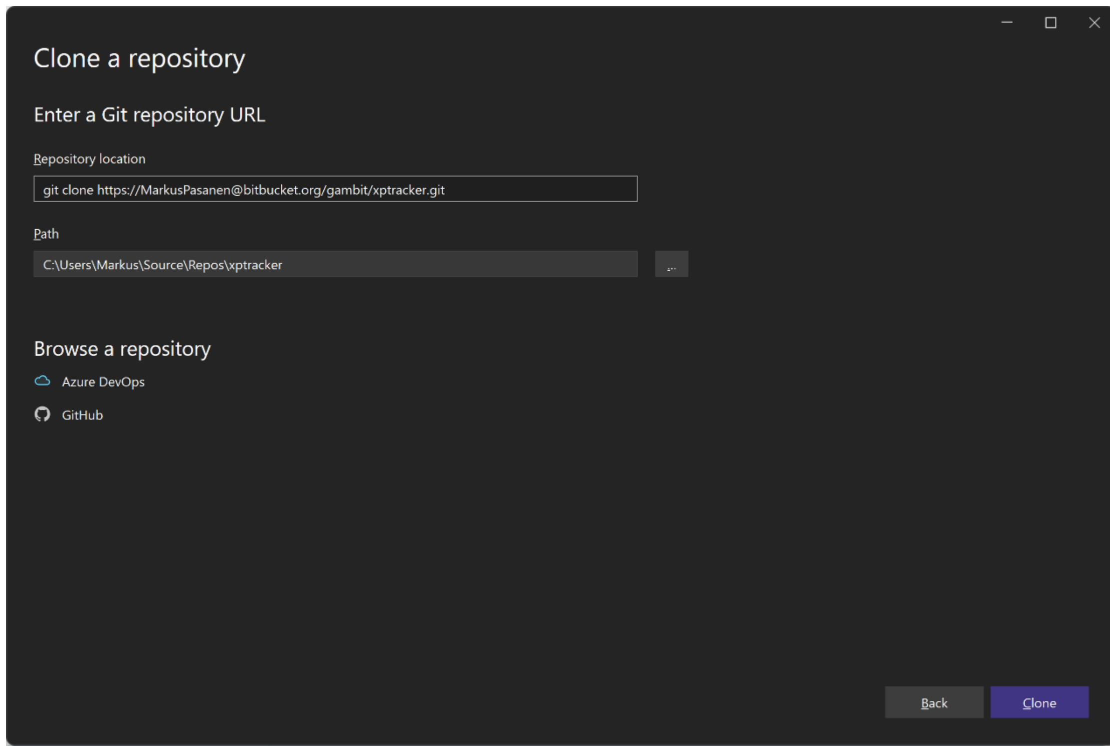


Kuva 15 Koodivaraston näkymä

Varsinainen työ aloitettiin JIRA-projektinhallintajärjestelmän kautta saadun taskin eli työtehtävän vastaanottamisella. Nopea läpikäymisen jälkeen kopioin koodivarastosta linkin lähdekoodiin (kuva 15) ja siirryin luomaan työympäristöä projektille versionhallinnan alle. Varsinaista julkaisu- tai versionhallintastrategiaa ei projektissa käytetä, joten uusi oksa työtehtävästä luotiin suoraan master-oksan alle.

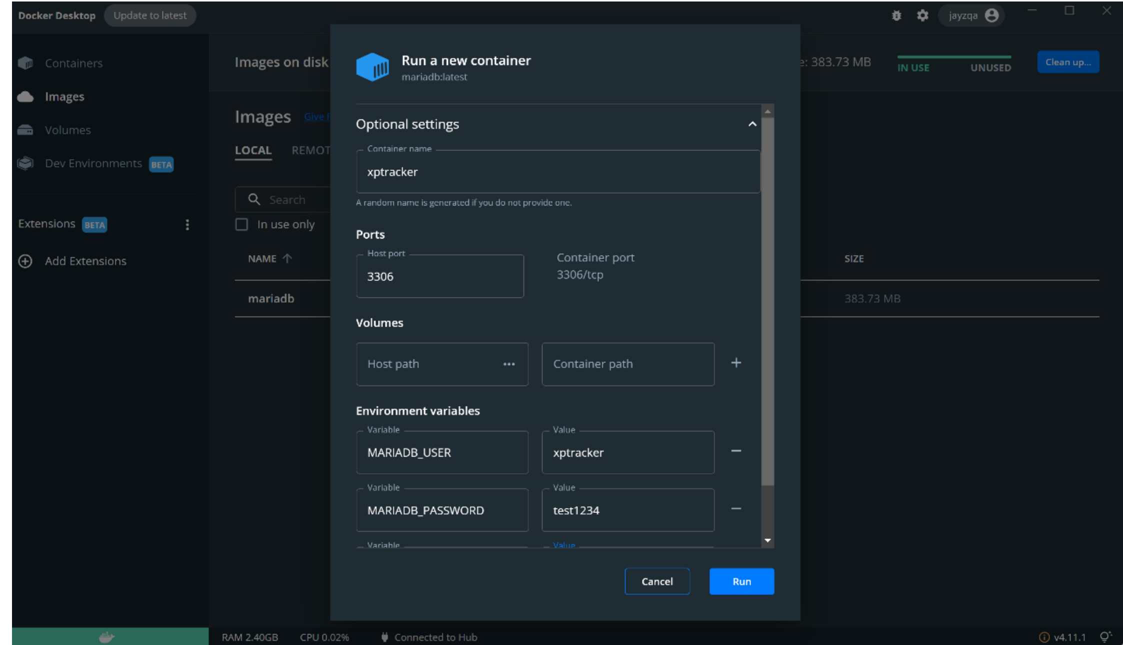
4.2 Työkalujen asennus ja työtilan pystytys

Työympäristön pystytys alkoi projektikopion luomisella bitbucket-koodivarastosta Rider-työkalun omien versionhallintatyökalujen avulla kuvan 16 mukaisella tavalla. Pienen silmäilyn jälkeen havaitsin, ettei projektissa ole käytetty varsinaisia yksikkötestejä missään, joten näin yksinkertaisimmaksi ratkaisuksi luoda työkoneelle paikallisen tietokannan yhteyksien testaamista varten.



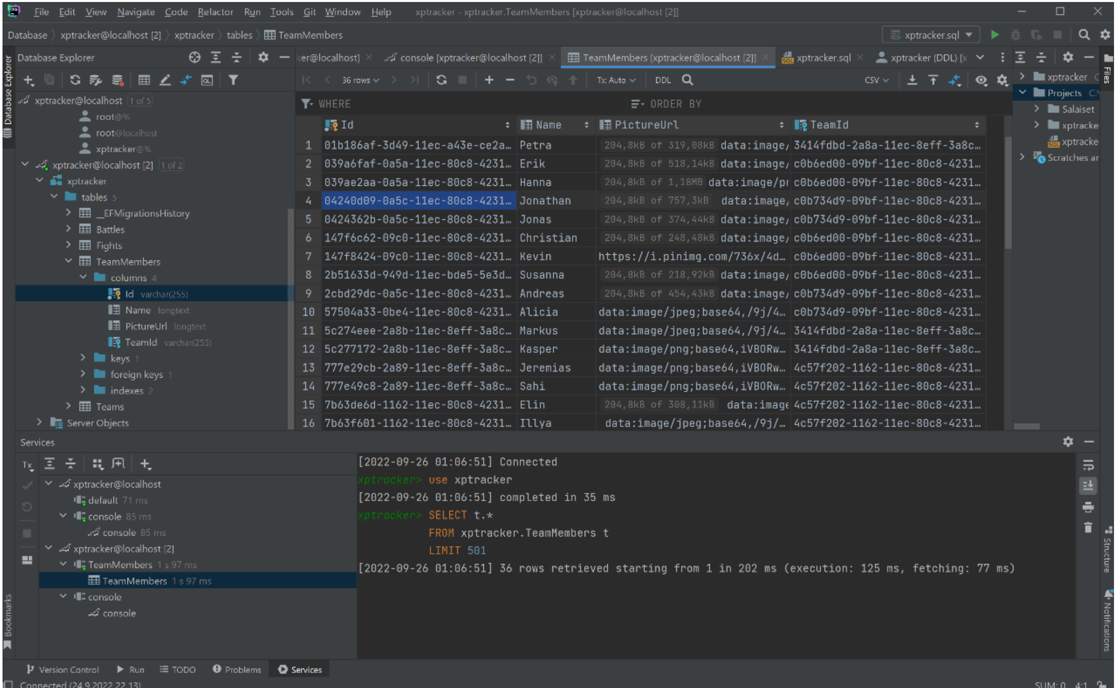
Kuva 16. Projektiympäristön luominen Rider työkalun toiminnoilla

Olin saanut vaateidenkeräysvaiheessa kopion tietokannasta sql-tiedostomuodossa, joten käytän tätä myös testidatana. Projektin luonne ei myöskään vaadi tietokantayhteydeltä salausta, joten täysin perusmuotoinen ratkaisu riittää tässä tapauksessa. SQL-tiedoston avaamalla selviää projektissa käytetyn tietokantahallintajärjestelmän olevan MariaDB, joten pystytän kuvan mukaisella tavalla tyhjän kopion tästä Docker-konttiympäristöön valmiista levykuvasta Docker hubista (kuva 17).



Kuva 17. Uuden docker-kontin luominen Docker hub-käyttöliittymän kautta

Tämän jälkeen testasin tietokannan toimivuuden JetBrainsin Datagrip tietokannanhallintasovelluksella. Työkalu valikoitui käyttöön yksinkertaiseksi siksi, että se kuuluu samaan ohjelmistopakettiin kehityksessä käytettävän Rider-työkalun kanssa. Ajan SQL-tiedoston DataGripin käyttöliittymän kautta (kuva 18). Normaalisti ulkopuolinen sovellus ei ole oikeutettu ajamaan pääkäyttäjänä tietokantaoperaatioita, joten teen uuden käyttäjän sovellusta varten. Tälle annetaan oikeudet suorittaa kyselyitä uuden tietokannan sisällä.



Kuva 18. Tietokannan hallinta Datagrip-työkalun avulla

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "Default": "server=localhost;Uid=xptracker;Pwd=test1234;Database=xptracker;"
  }
}
```

Kuva 19. Yhteysmerkkijono tietokantaan.

Viimeiseksi ympäristön pystytykseen kuuluu myös yhteysmerkkijonon lisääminen varsinaisen projektin sisälle, jotta tämä saa yhteyden uuteen testikantaan. Projektissa oli erillinen konfiguraatiotiedosto käytössä, joten rivi lisättiin tänne kuvan 19 osoittamalla tavalla.

4.3 Ohjelmointi

Varsinainen ohjelmointi aloitettiin ohjaimen luomisella projektin sisälle (kuva 20). Koska toteutettava ominaisuus on varsin yksinkertainen luonteeltaan ja projektissa oli nimellisesti käytetty sipulimallia, mutta nämä käytännössä koskivat ainoastaan koneoppimisen toimintoja varten, joten käytin varsinaisena suunnittelumallina normaalia MVC-mallista toteutusta. Tämä oli myös linjassa alkuperäisen käyttöliittymän kanssa.

```
HeroViewModel.cs   HeroFactory.cshtml   HeroController.cs
using System.Diagnostics;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using XpTracker.Infrastructure;
using XpTracker.Models;

namespace XpTracker.Controllers
{
    public class HeroController : Controller
    {

        private readonly ApplicationDbContext _dbContext;

        public HeroController(ApplicationDbContext dbContext)
        {
            _dbContext = dbContext;
        }

        public async Task<IActionResult> HeroFactory()
        {
            var allHeroes = await _dbContext.TeamMembers.ToListAsync();
            var heroViewModel = new HeroViewModel(allHeroes);
            return View(heroViewModel);
        }

        [HttpPost, ActionName("Delete")]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult> DeleteConfirmed(string id)
        {
            var student = await _dbContext.TeamMembers.FindAsync(id);
            if (student == null)
            {
                return NotFound();
            }

            try
```

Kuva 20. Ohjaimen luominen

Seuraavaksi luotiin tietomalli näkymää varten (kuva 21), jolloin kaikki tarvittavat muuttujat olisivat näkymän saatavilla oikeassa muodossaan. Tietoa ei varsinaisesti haluta enää parsia näkymän puolella, joten hyvin mietitty ja yksinkertainen tietomalli takaa tiedon eheyden.

```
using System.Collections.Generic;

namespace XpTracker.Models
{
    public class HeroViewModel
    {
        public HeroViewModel(List<TeamMember> heroes)
        {
            Heroes = heroes;
        }
        public List<TeamMember> Heroes { get; }

    }
}
```

Kuva 21. Mallin luominen

Varsinaisen käyttöliittymän luomisessa pyrittiin noudattamaan sovelluksen yleistä ulkoasua mahdollisimman jäljittelevästi, jotta lopputulos istuisi samaan tyyliin mitä se on ollutkin. Normaalissa kehitystilanteessa, jossa vaateet tulisivat yrityksen ulkopuoliselta asiakkaalta, laadittaisiin normaalisti jonkinlainen prototyyppi mahdollisesta käyttöliittymästä, jota sitten esiteltäisiin. Varsinaista käyttöliittymäprotoa ei kuitenkaan tässä tapauksessa tarvinnut tehdä projektin sisäisen luonteen takia, vaan lopputulos tehtiin mahdollisimman nopealla kaavalla (kuva 22).

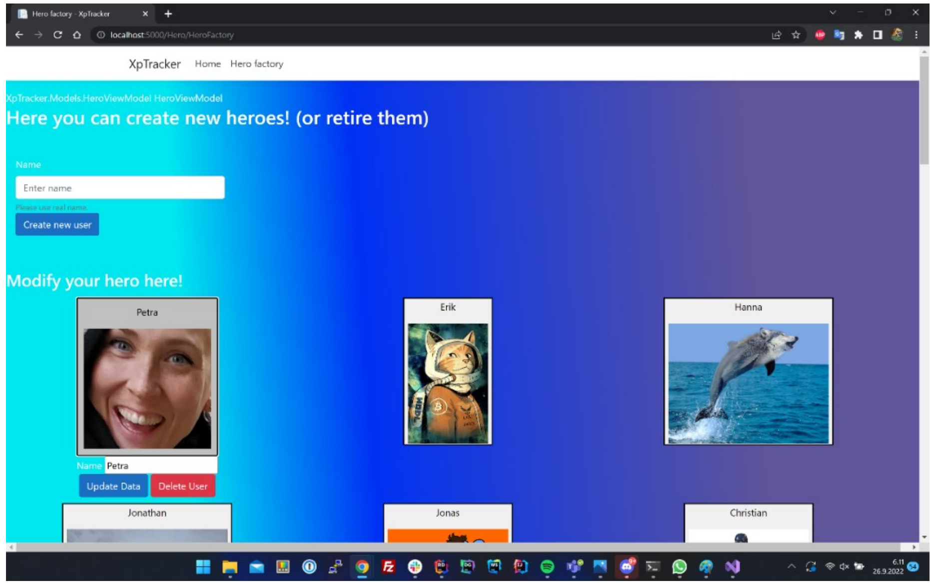
```
HeroViewModel.cs × | HeroFactory.cshtml × | C# HeroController.cs ×
@Model HeroViewModel
@{
    ViewData["Title"] = "Hero factory";
}

<body xmlns="http://www.w3.org/1999/html">
<h2> Here you can create new heroes! (or retire them) </h2>
<br/>
<form>
    <div class="form-group w-25 p-3">
        <label for="exampleInputEmail1">Name</label>
        <input type="text" class="form-control" id="exampleInputEmail1" aria-describedby="emailHelp" placeholde
        <small id="emailHelp" class="form-text text-muted">Please use real name.</small>
        <button type="submit" class="btn btn-primary">Create new user</button>
    </div>
</form>
<br/>
@if (Model.Heroes.Count > 0)
{
    <h3>Modify your hero here!</h3>
    <div id="form-div" class="row">
        @foreach (var hero in Model.Heroes)
        {
            <div class="teamMemberBox">
                <button id="@hero.Id" onclick="JoinTeamMemberToGroup(this.id)">
                    <p>@hero.Name</p>
                    <img src="@hero.PictureUrl" class="profilepicture" alt="@hero.Name">
                </button>
                <br/>
                <div id="form-div-@hero.Id" hidden>
                    Name <input id="fun-@hero.Id" type="text" value=@hero.Name><br>
                <div class="row">
                    <div class="col-lg-1">
                    <button type="button" class="btn btn-primary" asp-action="Edit" asp-route-id="@hero.Id">Edi
                    <button type="button" class="btn btn-danger" asp-action="Delete" asp-route-id="@hero.Id">De
                    </div>
                </div>
                <h3 class="clearfix">@Model.Message</h3>
                </div>
            </div>

        }
    </div>
}
</body>
```

Kuva 22. Näkymän luominen

Käyttäjän poistaminen ja muokkaaminen vaati käytännössä vain nimen ja uniikin avaimen, joten monimutkaisempi toteutus olisi vain lisännyt turhaa monimutkaisuutta lopputulokseen. Ylläpitosivu tulee myös olemaan todella vähässä käytössä reaalimaailmassa, eikä varsinaista avainkäyttäjää tarvinnut erikseen luoda. Käyttöoikeuksien hallintaa ei siis tarvinnut miettiä tässä tapauksessa.

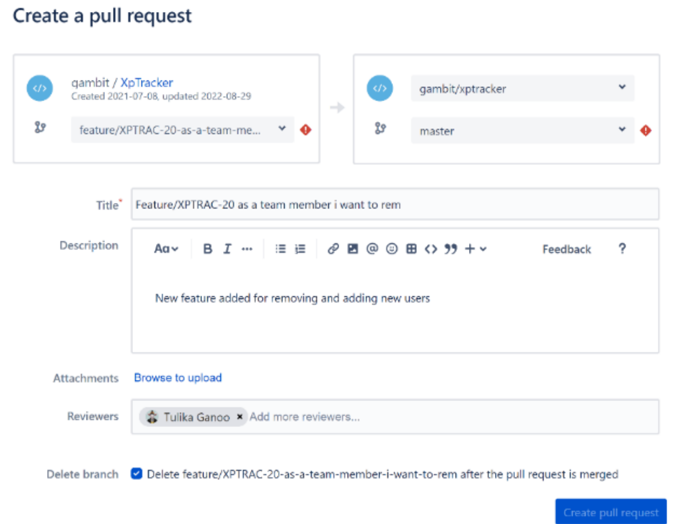
4.4 Toiminnallisuuden testaaminen ja julkaisu

Lopuksi toiminnallisuus testattiin yksinkertaisesti luomalla uusia käyttäjiä tietokantaan uuden sivun kautta (kuva 23), sekä vanhoja käyttäjiä muokkaamalla. Dockerissa pyörivä testikanta mahdollisti tämän monipuolisesti, koska huonossa tapauksessa tiedot voisi aina nopeasti palauttamaan SQL-kyselyllä uudestaan ja uudestaan. Varsinaisten yksikkötestien puuttumisen takia tämä olikin todennäköisesti paras vaihtoehto toiminnallisuuden testaamista varten.



Kuva 23. Kuva käyttöliittymästä

Lopuksi työ viimeisteltiin ajamalla muutokset git-versionhallinnan kautta ja tekemällä vetopyyntö Bitbucket-työkalun kautta (kuva 24). Tällöin koodikatselmointi voidaan suorittaa helposti parin kanssa. Työkalun käyttöliittymä näyttää selkeästi vanhan ja uuden koodiversion erot rivi riviltä.



Kuva 24. Vetopyynnön luominen

Lopputuloksesta suoritettiin koodikatselmointi, jossa käytiin läpi mahdollisia bugeja ja koodin laatua verrattiin olemassa olevaan koodipohjaan. Toiminnallisuus toimi varsin hyvin ja vetopyyntö hyväksyttiin katselmoijan toimesta. Tämän jälkeen käytiin keskusteluja mahdollisesta jatkokehityksestä myöhemmin ja nämä päätettiin toteuttaa uusina työtehtävinä, jolloin yhden oksan koodimuutosten määrä ei kasvaisi liian suureksi.

5 PÄÄTÄNTÖ

Projekti sujui hyvin ja se saatiin toteutettua joustavasti firman sisäisesti. Kehityksessä edettiin mahdollisimman kiinteää kommunikaatiota pitäen ja sisäinen kehitystavoite saavutettiin. Työ kokonaisuuden parissa jatkuu satunnaisesti projektin toimiessa varsinaisesti harjoitusprojektina aina .NET-teknologioihin opetettaessa.

Uusien käyttäjien lisääminen ja vanhojen poistaminen onnistuu nyt kätevästi graafisen käyttöliittymän kautta, jolloin projektia kulloinkin ylläpitävä työntekijä säästyy suoralta tietokannan käsittelyltä SQL-kyselyiden avulla, joka vähentää huomattavasti virheiden mahdollisuutta.

Kehityskokemus .NET-teknologioista ja SQL-tietokannan käsittelystä käytöstä oli opettavainen kokemus. Nämä teknologiat tulevat olemaan hyödyllisiä tulevissa kehitystöissä ja tarjoavat myös hyvän pohjan ammentaa tämän opinnäytetyön toteutuksen aikana opituista asioista käytännön projekteissa tulevaisuudessa. Uutena asiana tuli Docker-kuvan hyödyntäminen tietokannan pystyttämisessä, joka yksinkertaistaa huomattavasti tietokannan pystyttämistä ja paketoimista tulevaisuudessa.

Suunnittelumallien tutkiminen työtä varten antoi lisää ymmärrystä niiden soveltamisessa käytännössä ja mahdollistaa monipuolisempien ja paremmin skaalautuvien sovellusten luomisen tulevaisuudessa. Niiden tärkeys myös kristallisoitui varsinkin useamman ihmisen työskennellessä saman koodipohjan kanssa yhtäaikaisesti.

Yhteenvetona tiivistän opinnäytetyön toteuttamisen olleen jollain tasolla haastava varsinkin uutena tulleiden asioiden osalta. Tämä antoi todella paljon uusia työkaluja työkalupakkiin tulevaisuutta ajatellen ja auttaa välttämään työn aikana ilmenneitä sudenkuoppia asiakasprojekteissa, joka lähtökohtaisesti lyhentää niiden kehitysaikaa omalta osaltani.

LÄHTEET

.NET Core Overview. 2022. TutorialsTeacher. WWW-dokumentti. Saatavissa: https://www.tutorialsteacher.com/core/dotnet-core / [viitattu 16.08.2022]

Broadhead, R. 2020 The Copy Paste AntiPattern. WWW-dokumentti. Saatavissa: https://develpreneur.com/the-copy-paste-antipattern-an-easy-trap-to-fall-into/ [viitattu 15.09.2022]

Chaturvedi, V. 2021. What Is Docker & Docker Container ? A Deep Dive Into Docker. WWW-dokumentti. Saatavissa: https://www.edureka.co/blog/what-is-docker-container [viitattu 15.09.2022]

Containers were just the Beginning. 2022 .Docker Inc. WWW-dokumentti. Saatavissa: https://www.docker.com/why-docker/ [viitattu 15.09.2022]

Entity Framework – Overview. 2022. Tutorialspoint. WWW-dokumentti. Saatavissa: https://www.tutorialspoint.com/entity_framework/entity_framework_overview.htm [viitattu 16.08.2022]

Entity Framework Core. 2021. Microsoft. WWW-dokumentti. Saatavissa: https://docs.microsoft.com/en-us/ef/core/ [viitattu 16.08.2022]

Gamma, E. 2005. How to Use Design Patterns A Conversation with Erich Gamma, Part I. WWW-dokumentti. Saatavissa: https://www.artima.com/articles/how-to-use-design-patterns [viitattu 15.09.2022]

GolfPatterns. 2022. Choosing a Design Pattern. WWW-dokumentti. Saatavissa: https://www.gofpatterns.com/design-patterns/module7/how-to-choose-designPattern.php [viitattu 20.09.2022]

Gudabayev, T. 2021. Understanding SOLID Principles: Dependency Inversion. WWW-dokumentti. Saatavissa: https://dev.to/tamerlang/understanding-solid-principles-dependency-inversion-1b0f [viitattu 20.09.2022]

Introduction To ASP.NET Core Razor Pages. 2021. C# Corner. WWW-dokumentti. Saatavissa: https://www.c-sharpcorner.com/article/introduction-to-asp-net-core-razor-pages/ [viitattu 16.08.2022]

Janssen, T. 2018. SOLID Design Principles Explained: The Open/Closed Principle with Code Examples. WWW-dokumentti. Saatavissa: https://www.freecodecamp.org/news/solid-principles-explained-in-plain-english/ [viitattu 20.09.2022]

Jones, M. 2015. What is the Golden Hammer Anti-Pattern?. WWW-dokumentti. Saatavissa: https://exceptionnotfound.net/the-golden-hammer-anti-pattern-primers/ [viitattu 15.09.2022]

Learn Razor Pages. 2021. Why should you use Razor Pages?. WWW-dokumentti. Saatavissa: https://www.learnrazorpages.com/ [viitattu 15.09.2022]

Liang, M. 2021. Understanding Object-Relational Mapping: Pros, Cons, and Types. WWW-dokumentti. Saatavissa: https://www.altexsoft.com/blog/object-relational-mapping/ [viitattu 15.09.2022]

Lucidchart. 2022. What are software anti-patterns?. WWW-dokumentti. Saatavissa: https://www.lucidchart.com/blog/what-are-software-anti-patterns [viitattu 20.09.2022]

Mamani, C. 2020 The God Object Or The God Class Anti-Pattern. WWW-dokumentti. Saatavissa: https://medium.com/@carlos.ariel.mamani/the-god-object-or-god-class-anti-pattern-bfb8c15eb513 [viitattu 15.09.2022]

What is MariaDB. 2022. MariaDB Tutorial. WWW-dokumentti. Saatavissa: https://www.mariadbtutorial.com/getting-started/what-is-mariadb/ [viitattu 15.09.2022]

MVC. 2022. Mozilla. WWW-dokumentti. Saatavissa: https://developer.mozilla.org/en-US/docs/Glossary/MVC [viitattu 15.09.2022]

Parr, K. 2020. Anti-patterns You Should Avoid in Your Code. WWW-dokumentti. Saatavissa: https://www.freecodecamp.org/news/antipatterns-to-avoid-in-code/ [viitattu 15.09.2022]

ReSharper + IntelliJ platform. 2022. JetBrains. WWW-dokumentti. Saatavissa: https://www.jetbrains.com/rider/features/ [viitattu 15.09.2022]

Saxena, M. 2021. What Is Liskov Substitution Principle (LSP)? With Real World Examples. WWW-dokumentti. Saatavissa: https://blog.knoldus.com/what-is-liskov-substitution-principle-lsp-with-real-world-examples/ [viitattu 20.09.2022]

Software Craft. 2020. Interface Segregation Principle: Everything You Need to Know. WWW-dokumentti. Saatavissa: https://reflectoring.io/interface-segregation-principle/ [viitattu 20.09.2022]

SOLID: The First 5 Principles of Object Oriented Design. 2020. Digital ocean. WWW-dokumentti. Saatavissa: https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design [viitattu 16.08.2022]

Source making. 2022. Boat Anchor. WWW-dokumentti. Saatavissa: https://sourcemaking.com/antipatterns/boat-anchor [viitattu 20.09.2022]

Tahaghoghi, S. & Williams, H. 2007. Learning MySQL. O'Reilly. [viitattu 15.09.2022]

The Single Responsibility Principle. 2020. freeCodeCamp. WWW-dokumentti. Saatavissa: https://www.freecodecamp.org/news/solid-principles-explained-in-plain-english/ [viitattu 15.09.2022]

Understanding Onion Architecture. 2018. CodeGuru. WWW-dokumentti. Saatavissa: https://www.codeguru.com/csharp/understanding-onion-architecture/#Principles [viitattu 15.09.2022]

Varma, S. 2022. ORM (Object Relational Mapping) WWW-dokumentti. Saatavissa: https://javabydeveloper.com/orm-object-relational-mapping/ [viitattu 15.09.2022]

Watts, S. 2020. What is Spaghetti Code (And Why You Should Avoid It). WWW-dokumentti. Saatavissa: https://www.bmc.com/blogs/spaghetti-code/ [viitattu 15.09.2022]

What is MySQL?. 2022. Oracle. WWW-dokumentti. Saatavissa: https://dev.mysql.com/doc/refman/8.0/en/what-is-mysql.html [viitattu 15.09.2022]

What is Rider?. 2022. JetBrains. WWW-dokumentti. Saatavissa: https://www.jetbrains.com/rider/ [viitattu 15.09.2022]

What's a design pattern?. 2022. Refactoring guru. WWW-dokumentti. Saatavissa: https://refactoring.guru/design-patterns/what-is-pattern [viitattu 16.08.2022]

Wrapping up. 2022. Codecademy Team. WWW-dokumentti. Saatavissa: https://www.codecademy.com/article/mvc [viitattu 15.09.2022]