Joona Mäkinen

JAVA WEB-SOVELLUKSEN PÄIVITYS ASP.NET CORE 9 -VERSIOON

Tekniikka

2025

VAASAN AMMATTIKORKEAKOULU
Tietotekniikka

TIIVISTELMÄ

Tekijä	Joona Mäkinen
Opinnäytetyön nimi	Java Web-sovelluksen päivitys ASP.NET CORE 9 -versioon
Vuosi	2025
Kieli	suomi
Sivumäärä	29
Ohjaaja	Matti Tuomaala

Tämän opinnäytetyön tavoitteena oli kehittää Alajärven puhelinosuuskunnan vanhan Java-pohjaisen asiakashallintajärjestelmän tilalle uusi, moderni web-sovellus. Uusi sovellus toteutettiin hyödyntäen ASP.NET Core 9 -sovelluskehystä.

Projektin keskeisiin tavoitteisiin kuului tutustuminen Microsoftin tarjoamiin teknologioihin, kuten ASP.NET Core, Razor Pages, C#-ohjelmointikieleen sekä tietokannan hallintaan käytettyyn Entity Framework Coreen. Sovelluksen käyttöliittymä toteutettiin Razor Pages -arkkitehtuurilla, joka mahdollistaa yksinkertaisen ja ylläpidettävän rakenteen erityisesti liiketoimintalähtöisiin verkkosovelluksiin.

Opinnäytetyö rajattiin koskemaan asiakastietojen hallintaa, ja muut toiminnot, kuten datan siirto vanhasta järjestelmästä, on jätetty jatkokehitykseen. Projektin aikana hyödynnettiin moderneja ohjelmistokehitystyökaluja, kuten Visual Studio 2022 ja Git-versionhallintaa.

Työ toimi myös oppimisprojektina, jonka kautta kehitettiin osaamista nykyaikaisten web-teknologioiden käytössä ja syvennettiin ymmärrystä sovelluskehityksen periaatteista erityisesti ASP.NET Core -alustalla.

Avainsanat ASP.NET Core, C#, Razor Pages, Entity Framework

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Tietotekniikka

ABSTRACT

Author	Joona Mäkinen
Title	Updating a Java Web Application to ASP.NET Core 9 Version
Year	2025
Language	Finnish
Pages	29
Name of Supervisor	Matti Tuomaala

The aim of this thesis was to develop a new, modern web application to replace the old Java-based customer management system used by Alajärven Puhelinosuuskunta. The new application was built using ASP.NET Core 9 framework.

A key goal of the project was to become familiar with Microsoft technologies such as ASP.NET Core, Razor Pages, the C# programming language, and Entity Framework Core for database management. The user interface was implemented using the Razor Pages architecture, which provides a clean and maintainable structure particularly suited for business-oriented web applications.

The scope of the thesis was limited to customer data management, while other features, such as data migration from the old systems, were left for future development. Modern development tools were used throughout the project, including Visual Studio 2022 and Git for version control.

The project also served as a learning experience to develop skills in modern web technologies and to deepen understanding of application development principles, especially on the ASP.NET Core platform.

Keywords ASP.NET Core, C#, Razor Pages, Entity Framework

SISÄLLYS

KUVAT

1 JOHDANTO

Tämän työn tarkoituksena on tutkia ja kehittää web-pohjainen asiakashallintajärjestelmä, jonka avulla voidaan hallita asiakastietoja sekä lisätä ja muokata asiakkaille liitettyjä palveluita ja tuotteita.

Tarkoituksena on tutustua Microsoftin uuteen ASP.Net Core 9 Frameworkkiin, sen ominaisuuksiin ja siinä käytettävään C#-kieleen. Lisäksi tutustutaan eri malleihin tehdä sovellus, kuten MVC ja Razor Pages, ja Entity Frameworkkiin, jolla tehdään tietokanta. Sovelluksen käyttöliittymä toteutetaan Razor Pagesilla.

Työntilaajana toimii Japo Palvelut Oy on perustettu vuonna 2001 ja se kuuluu Japo-konserniin, johon kuuluu lisäksi vuonna 1925 perustettu Alajärven puhelinosuuskunta. Japo Palveluiden toimiala keskittyy Ohjelmistokehitykseen ja yritysten IT-ratkaisuihin. Emoyhtiön toimintaan kuuluu valokuituverkkojen tarjoaminen.

Asiakashallintajärjestelmä perustuu vanhaan Java-pohjaiseen sovellukseen, joka ei enää vastaa tarpeita ja on hankala jatkokehittää. Järjestelmän uudistaminen ASP.NET 9-teknologiaa hyödyntäen tulisi mahdollistaa parempi ylläpidettävyys ja jatkokehitys.

2 PROJEKTIN TAUSTA

2.1 Projektin tausta ja tarkoitus

Tämän opinnäytetyön taustalla on tarve korvata käytöstä poistuva asiakashallintajärjestelmä. Nykyinen järjestelmä on rakennettu Java-teknologioiden varaan ja sen elinkaari on tullut tiensä päähän. Järjestelmän ylläpito ja jatkokehitys ovat vuosien saatossa vaikeutuneet merkittävästi johtuen koodin ja arkkitehtuurin hankaluudesta.

Vanhan järjestelmä on sisäisessä käytössä oleva Web-sovellus, jonka käyttöliittymä on nykystandardeihin verrattuna vanhanaikainen ja jäykkä. Käyttäjäkokemus on heikko eikä monia toimintoja voida suorittaa suoraan järjestelmässä, vaan ne edellyttävät manuaalisia työvaiheita ja ulkoisten työkalujen käyttöä. Tällainen toimintamalli lisää virheiden riskiä, hidastaa työprosesseja ja aiheuttaa käyttäjätyytymättömyyttä. Lisäksi järjestelmän kankeat tietorakenteen vaikeuttavat uusien toimintojen kehittämistä.

Uuden järjestelmän tavoitteena on korvata kaikki vanhan järjestelmän keskeiset toiminnot ja samalla parantaa käytettävyyttä, tehokkuutta ja luotettavuutta. Tavoitteena on luoda järjestelmä, joka ei ainoastaan palvele nykyisiä tarpeita, vaan toimii myös pohjana tulevalle kehitykselle ja uusien toimintojen lisäämiselle.

Sovellus toteutetaan ASP.NET Core -teknologialla, koska se kuuluu yrityksen käyttämään teknologiapinoon (tech stackiin). Näin varmistetaan yhteensopivuus olemassa olevien järjestelmien kanssa sekä helpotetaan sovelluksen ylläpitoa ja jatkokehitystä yrityksen sisällä.

Opinnäytetyö toimii samalla oppimisprojektina, jonka kautta perehdytään ASP.NET Core -alustan tarjoamiin työkaluihin ja ominaisuuksiin erityisesti verkkopohjaisen liiketoimintasovelluksen näkökulmasta. Tavoitteena on kehittää osaamista web-teknologioiden käytöstä ja syventää ymmärrystä sovelluskehityksestä.

Tämä opinnäytetyö on rajattu koskemaan ainoastaan asiakkaiden hallintaa. Muut vanhassa järjestelmässä olleet toiminnot ja datan siirto jätetään tämän työn ulkopuolelle ja ne toteutetaan myöhemmässä jatkokehityksessä.

3 TIETOPERUSTA

3.1 ASP.NET Core

ASP.NET Core on Microsoftin kehittämä avoimen lähdekoodin verkkosovelluskehys, joka mahdollistaa modernien ja alustariippumattomien web-sovellusten kehittämisen. Se on seuraaja vanhemmalle ASP.NET-kehykselle, ja sen ensimmäinen versio julkaistiin vuonna 2016 osana .NET Core – alustaa. Kehyksen suunnittelun lähtökohtana oli luoda modulaarinen, kevyt ja helposti ylläpidettävä ratkaisu, joka toimii Windows-, Linux- ja macOS-käyttöjärjestelmissä. (Microsoft, 2024a.)

ASP.NET Core:n yksi etu onkin sen alustariippumattomuus. Sillä kehitetyt sovellukset voivat toimia useissa käyttöjärjestelmissä kuten Windowsissa, macOS:ssä ja Linuxissa. Tämä saavutetaan hyödyntämällä .NET core -alustaa, joka tarjoaa yhtenäisen kehitysympäristön eri käyttöjärjestelmille. (Microsoft, 2024a.)

C# on Microsoftin kehittämä yleiskäyttöinen ja olio-ohjelmointiin perustuva ohjelmointikieli, jota käytetään ensisijaisesti .NET-alustalla. Se on suosituin .NET-kieli. Kieli on suunniteltu tuottavuutta ja suorituskykyä silmällä pitäen, ja sen syntaksi muistuttaa C-, C++-, Java- ja JavaScript-kieliä (Microsoft, 2024b.)

3.2 Entity Framework Core

Entity Framework Core on Microsoftin kehittämä avoimen lähdekoodin Object Relation Mapper (ORM), joka mahdollistaa .NET kehittäjälle tietokantayhteyksien hallinnan käyttäen .NET Objekteja. EF Core on kevyt, laajennettavissa oleva ja alustariippumaton versio suositusta Entity Framewotk teknologiasta. (Microsoft, 2024c.)

Keskeisiin ominaisuuksiin kuuluu mahdollisuus tietokannan käsittelyyn .NET Objektien avulla, mikä vähentää tarvetta kirjoittaa SQL-kyselyitä suoraan. Kehittäjä voi käyttää Language Intregated Query (LINQ) -kyselyitä tietojen hakemiseen ja muokkaamiseen. EF Core tukee useita tietokantoja, muun muassa SQL Server, SQLite, PostgreSQL, MySQL ja Azure Cosmos DB. (Microsoft, 2024c.) Lisäksi EF Core tarjoaa työkalut skeeman hallintaan ja migraatioiden toteuttamiseen, mikä helpottaa tietokannan rakenteen muutosten hallintaa. (Microsoft, 2023a.)

3.3 ASP.NET Core käyttöliittymä

Blazor on Microsoftin SPA ratkaisu. Se mahdollistaa käyttöliittymäkomponenttien rakentamisen C#-kielellä ja tarjoaa vaihtoehtoja palvelinpään ja asiakaspään renderöintiin. Se tukee komponenttipohjaista arkkitehtuuria ja se voidaan myös liittää olemassa oleviin MVC- tai Razor Pages- sovelluksiin. (Microsoft, 2024f.)

ASP.NET Core MVC edustaa perinteistä Model-View-Controller-arkkitehtuuria, jossa erotetaan data, käyttöliittymä, ja ohjauslogiikka selkeästi toisistaan. (Microsoft, 2024f.)

MVC vaihtoehtoon sisältyy myös mahdollisuus yhdistää ASP.NET Core johonkin tunnettuun JavaScript- pohjaiseen käyttöliittymäkehykseen, kuten Reactiin tai Angulariin. Näissä ratkaisuissa asiakaspuolen logiikka toteutetaan JavaScriptillä, ja palvelinpuolen toiminnot hoidetaan REST-apin kautta. Tällainen lähestymistapa voi olla perusteltu silloin, kun tarvitaan hyvin responsiivinen käyttöliittymä ja kehitystiimillä on osaamista JavaScript kehityksestä. (Microsoft, 2024f.)

3.4 Razor Pages

Razor Pages on ASP.NET Coren tarjoama sivupohjainen kehys, joka yhdistää näkymän ja siihen liittyvän logiikan yhteen kokonaisuuteen. Se

perustuu palvelinpuolen renderöintiin, jossa jokainen käyttäjän toiminto johtaa http-pyyntöön ja koko sivun uudelleen lataukseen. Sen etuja ovat nopea käyttöliittymän kehitys ja päivitys, sekä helppo testattavuus ja skaalautuvuus suuriinkin sovelluksiin. (Microsoft, 2024f.)

Razor Pages -sovelluskehityksessä PageModel-luokka muodostaa sivun rakenteen. Vaikka Razor Pages yhdistää näkymän ja toiminnallisuuden, ne on loogisesti eroteltu kahteen eri tiedostoon. Kun näkymä ja toiminnallisuus ovat eri tiedostoissa, näkymästä tulee yksinkertaisempi ja helpompi ylläpitää. Tämä helpottaa monen kehittäjän tiimeissä työskentelyä, sillä näkymää ja toiminnallisuutta voidaan kehittää rinnakkain. (Learn Razor Pages, 2021a.)

Razor Pages -malli hyödyntää Page Controller -arkkitehtuurimallia, jossa jokaisella sivulla on oma kontrolleri. Tämä eroaa perinteisestä MVC:n Controller -mallista, jossa yksi kontrolleri käsittelee useita eri näkymiä. Page Controller -mallissa käsittelylogiikka sidotaan suoraan yksittäiseen sivuun, mikä tekee sovelluksesta yksinkertaisemman ymmärtää. PageModel- luokka toimii myös ViewModel-tyyppisesti. Se tarjoaa näkymälle tarvittavat tiedot ja käsittelylogiikan. (Learn Razor Pages, 2021a.)

Vaikka Razor Pages -rakennetta on joskus kuvattu MVVM-mallina, kyseessä ei ole puhdas MVVM-toteutus. MVVM perustuu kaksisuuntaiseen tietosidontaan ja näkymän automaattiseen päivittymiseen mallin muuttuessa, mikä ei luonnollisesti tuettuna palvelinpuolen Razor Pages -ympäristössä. (Learn Razor Pages, 2021a.)

3.5 Työkalut

Projektissa käytettiin kehitystyökaluna Microsoftin uusinta Visual Studio 2022 -ohjelmointiympäristöä. Se mahdollistaa koodin kirjoittamisen, muokkaamisen, ajamisen ja julkaiseminen samasta ympäristöstä. Visual studio tukee laajasti erilaisia ohjelmointikieliä, kuten tässä projektissa

tarvittuja C#, HTML/CSS ja JavaScript. Tämä mahdollistaa sovelluksen kehityksen yhdellä työkalulla.

Visual Studio sisältää myös älykkäitä avustimia koodin kirjoittamisen sujuvoittamiseen, mikä samalla myös vähentää virheitä. Se myös integroituu saumattomasti versionhallintaan esimerkiksi GitHubiin, jota myös tässä projektissa käytetään. Lisäksi Visual Studioon saa lisää toiminnallisuutta lataamalla lisäosia Visual Studio Marketplacesta (Microsoft, 2024a).

4 TOTEUTUS

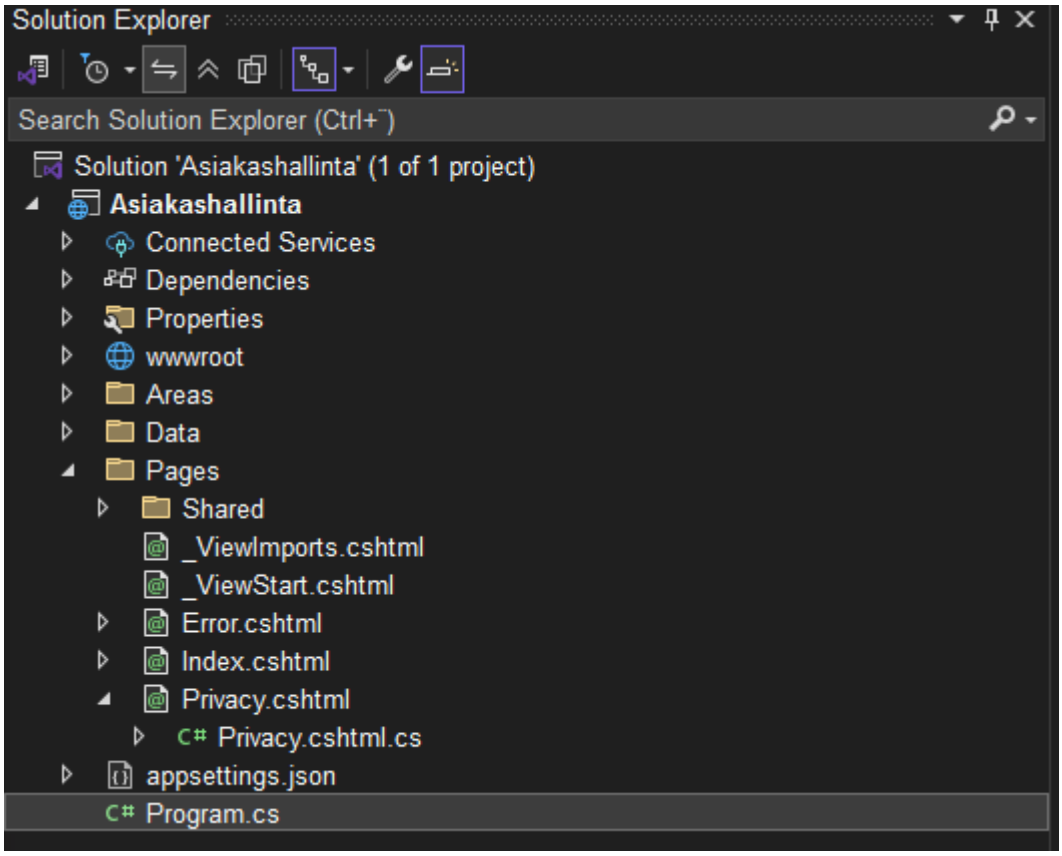
4.1 Versionhallinta

Projektissa käytettiin versionhallintajärjestelmänä Gitiä. Git mahdollisti tehokkaan kehitystyön hallinnan, sillä sen avulla pystyttiin seuraamaan muutoksia koodissa, palaamaan aiempiin versioihin tarvittaessa ja luomaan eri kehityshaaroja. Gitin käyttö loi myös hyvän pohjan myöhemmälle kehitykselle.

Git tarjosi mahdollisuuden luora eri kehityshaaroja, joiden avulla voitiin työskennellä samanaikaisesti useiden ominaisuuksien tai korjausten parissa ilman, että pääkehityshaara häiriintyi.

4.2 Projektin perustaminen

Uutta Razor Pages -projektia luotaessa Visual Studio tarjoaa valmiin projektipohjan, joka sisältää kaikki tarvittavat tiedostot ja kansiorakenteet. Tämä mahdollistaa nopean kehitystyön aloittamisen ilman tarvetta luoda perusrakennetta manuaalisesti.



Kuva 1. Tiedostorakenne

Kuten kuvasta 1 näkyy, sovellus sisältää oletuksena Pages -kansion, jossa sijaitsevat ensimmäiset Razor Pages -sivut, kuten Index.html ja Privacy.html, sekä niihin liittyvät C#-tiedostot eli PageModel-luokat. Razor Pages -sivut tuovat C#- kielen suoraan HTML- sivujen yhteyteen.

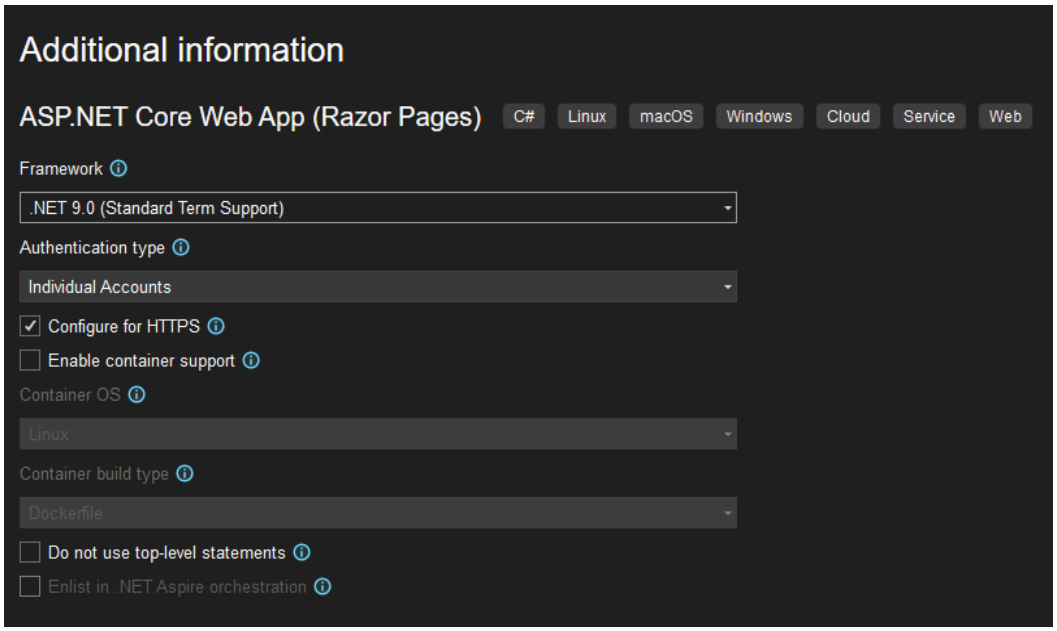
Staattisia tiedostoja kuten tyylitiedostoja, JavaScriptiä ja kuvia varten käytetään wwwroot-kansiota. Tähän kansioon sisältyy valmiiksi Bootstrap- ja jQuery-kirjastot, joilla saa parannettua sovelluksen ulkonäköä ja responsiivisuutta. Sovelluksen ulkoasu pohjautuu Pages/Shared kansiossa sijaitsevaan _Layout.cshtml- tiedostoon, joka toimii pohjana kaikille sivuille ja se sisältää muun muassa navigointipalkin ja viittaa tarvittaviin tyyleihin ja skripteihin.

.NET 6:sta alkaen ASP.NET Core -sovellusten käynnistys ja konfigurointi tapahtuvat Program.cs tiedostossa. Tiedosto yhdistää aiemmin kah-

dessa tiedostossa (Program.cs ja Startup.cs) olleen logiikan yhteen. Samalla on otettu käyttöön top-level statement -ominaisuus, joka mahdollistaa sen, että kooditiedosto ei enää vaadi erillistä Main-metodia. Tämä tekee sovelluksen käynnistyskoodista selkeämmän. Program.cs yhdistää sovelluksen konfiguroinnin, palveluiden määrittelyn ja http-pyyntöjen käsittelylogiikan yhteen tiedostoon. (Learn Razor Pages, 2022.)

4.3 Autentikointi

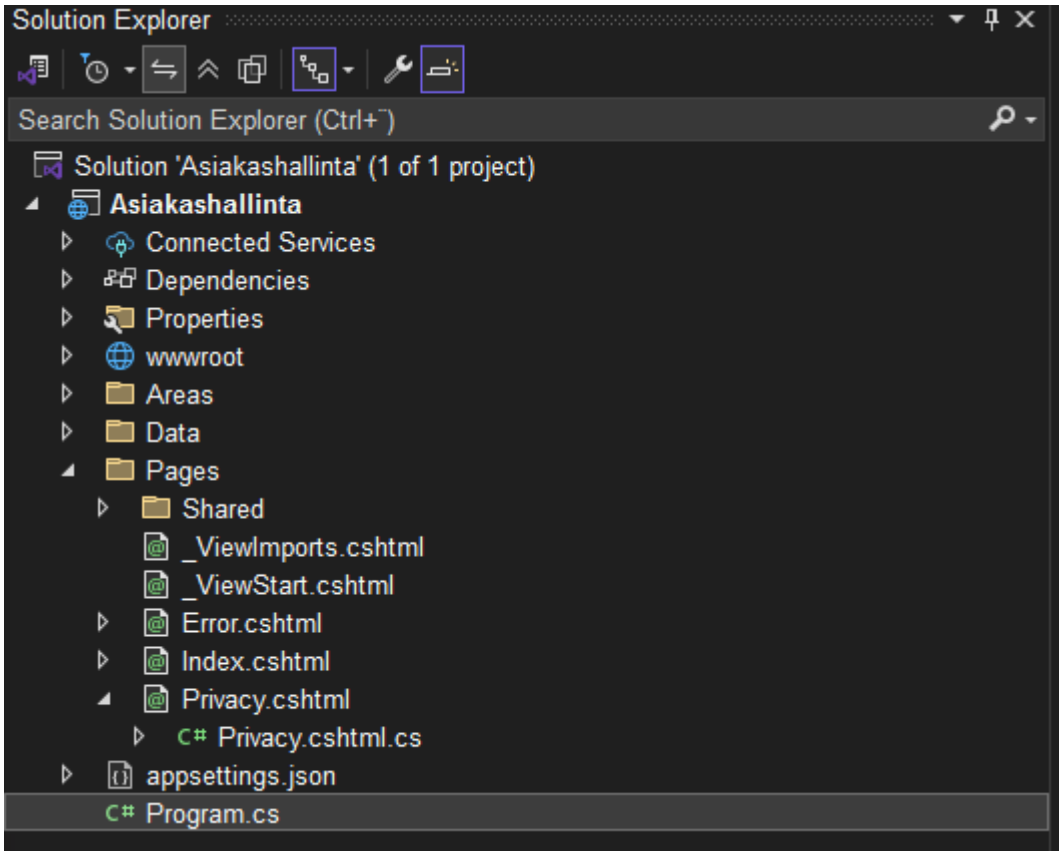
Projektin luonnin yhteydessä valittiin autentikoinnin tyypiksi ”Individual Accounts” kuten kuvassa 2, joka generoi automaattisesti tarvittavat käyttäjänhallintaan liittyvät tiedostot, tietokantamallit ja kirjautumislogiikan. Tämä mahdollisti valmiin käyttäjärekisteröinnin, kirjautumisen ja uloskirjautumisen ilman erillistä toteutusta.



Kuva 2. Individual Accounts.

Autentikointijärjestelmän pohjana toimii ASP.NET Core Identity -kirjasto, joka tarjoaa pohjan käyttäjätunnusten hallintaan. Identity -kirjasto luo tietokannan, johon tallennetaan mm. käyttäjien salasanat, käyttäjäroolit ja tunnukset. Projektissa hyödynnettiin käyttäjärooleja

käyttöoikeuksien hallintaan. Roolit määriteltiin tietokantaan ja niitä käytettiin näkymien ja toimintojen rajaamiseen.



Kuva 3. Razor Pages- tiedostorakenne.

4.4 Tietorakenteen määrittely

Tietorakenteen määrittely on tärkeä osa sovelluksen rakennetta, sillä ne määrittävvät, millaista dataa sovelluksessa käsitellään ja miten se tallennetaan tietokantaan.

Projektissa käytössä oleva EF Core toimii siten, että kehittäjä määrittelee tietomallit C#-olioina ja nämä muunnetaan automaattisesti tietokannan tauluiksi. Tämä lähestymistapa tunnetaan nimellä Code First, koska tietokantarakenne muodostetaan suoraan sovelluksen koodista.

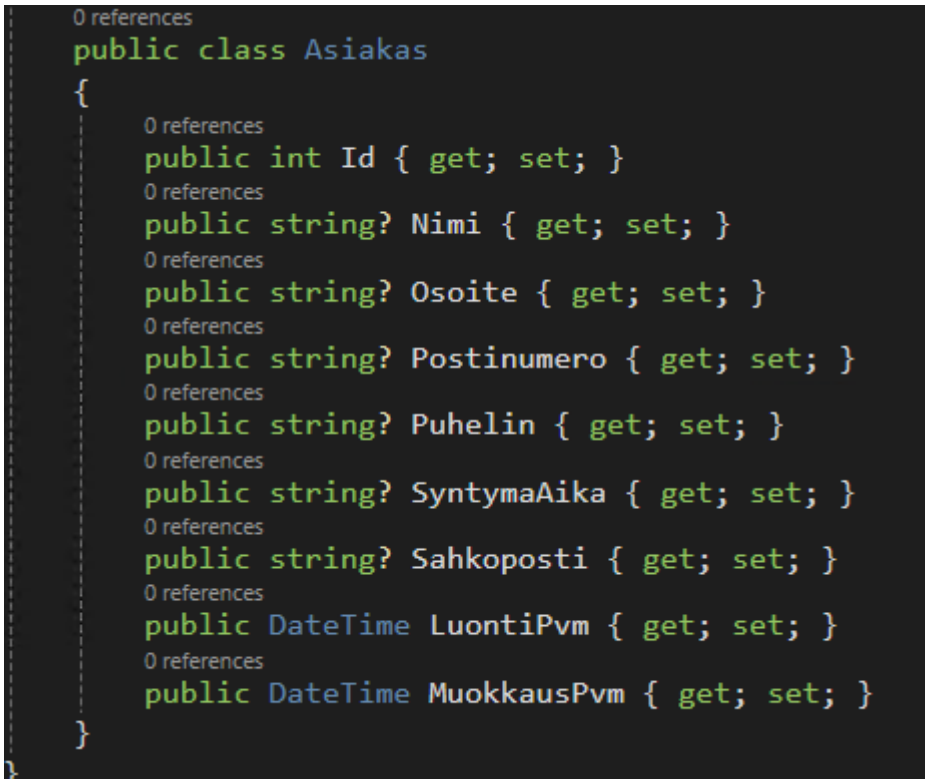
Kehityksessä käytettiin Visual Studion sisäistä paikallista SQL Server -palvelinta (LocalDB) tietokantana. LocalDB on SQL Server -versio, joka

asentuu automaattisesti Visual Studion mukana ja se tarjoaa helpon tavan kehittää sovelluksia ilman erillistä SQL Server -asennusta. EF Core muodostaa yhteyden paikalliseen tietokantaan käyttämällä yhteysmerkkijonoa. Tämä mahdollistaa sen, että tietokanta voidaan luoda ja hallita automaattisesti koodin ja Entity Frameworkin migraatioiden avulla ilman manuaalisia toimenpiteitä. (Microsoft, 2023a.)

Migraatiot ovat osa tietokannan rakenteen hallintaa. Migraatio kuvaa muutoksia, joita on tehty sovelluksen tietomaaliin, ja se mahdollistaa muutosten siirtämisen tietokantaan hallitusti. Uusi migraatio luodaan komennolla *Add-migration (Migraation nimi)*. Kun migraatio on luotu, päivitetään tietokanta vastaamaan sovelluksen tietomallia komennolla *update-database*.

Migraatiot tallentuvat projektin hakemistoon *Migrations*- kansioon ja sisältävät C#-koodia muutosten määrittelyyn, sekä mahdollisuuden palauttaa tietokanta aiempaan tilaan, jos jokin muutos osoittautuu virheelliseksi. (Microsoft, 2023a.)

Tietomallit sijoitettiin erilliseen Models- hakemistoon, joka luotiin projektin juureen selkeyden ja ylläpidettävyyden vuoksi. Jokainen malli vastaa yhtä tietokannan taulua, ja luokan ominaisuudet vastaavat taulun sarakkeita.



Kuva 4. Asiakasluokka.

Kuvassa 4 on Asiakasluokka, jossa Id toimii yksilöllisenä avaimena ja muut ominaisuudet vastaavat asiakkaan perustietoja. EF Core tunnistaa automattisesti Id-kentän ensisijaiseksi avaimen nimeämiskäytännön perusteella. Lisäksi voitaisiin käyttää tietomallien yhteydessä attribuutteja, joiden avulla voidaan asettaa lisärajoitteita ja suhteita tietokannan tasolla.

Sovelluksessa määriteltiin tietokantakonteksti- luokka, joka periytyy DBContext-luokasta. Tämän avulla EF Core tietää, miten mallit yhdistetään tietokannan tauluihin.

4.5 Palveluluokat ja Interfacet

Vaikka Razor Pages -sovellus mahdollistaa liiketoimintalogiikan toteuttamisen suoraan PageModel-luokissa. Sovelluksen ydinlogiikka erotettiin käyttöliittymästä paremman ylläpidettävyyden ja selkeyden saavuttamiseksi. Keskeiset tietokantatoiminnot ja muut sovelluksen toiminnot toteutettiin erillisiin palveluluokkiin, joille määriteltiin rajapinnat.

4.5.1 Palvelurakenne

Projektin juureen luotiin Services -kansio, johon sijoitettiin sovelluksen liiketoimintalogiikkaa sisältävät luokat, kuten asiakkaiden käsittelyyn liittyvä kuvassa 4 esiintyvä AsiakasService. Nämä luokat vastaavat kaikista tietokantatoiminnoista, kuten tietojen hakemisesta, lisäämisestä ja päivittämisestä. Razor Pages -sivujen PageModel-luokat keskittyvät ainoastaan käyttöliittymälogiikkaan ja toimivat kontrollerina, jotka kutsuvat tarvittavia palveluja

```
public class AsiakasService
{
    private readonly ApplicationDbContext _context;

    0 references
    public AsiakasService(ApplicationDbContext context)
    {
        _context = context;
    }

    0 references
    public async Task<List<Asiakas>> GetAllCustomersAsync()
    {
        return await _context.Asiakkaat.ToListAsync();
    }

    0 references
    public async Task<Asiakas?> GetCustomerByIdAsync(int id)
    {
        return await _context.Asiakkaat.FindAsync(id);
    }

    0 references
    public async Task AddCustomerAsync(Asiakas asiakas)
    {
        _context.Asiakkaat.Add(asiakas);
        await _context.SaveChangesAsync();
```

Kuva 5. AsiakasService.

4.5.2 Interfacet

Palveluluokkien rinnalle määriteltiin interface-luokat, jotka kuvaavat palvelun tarjoamat metodit ilman toteutusta. Interfacet mahdollistavat irrotetun ja helposti testattavan rakenteen, jossa esimerkiksi palveluiden toteutusta voidaan vaihtaa ilman muutoksia muuhun sovellukseen. Kuvassa 6 on esimerkki AsiakasServiceen toteutetusta rajapinnasta.

```
public interface IAsiakasService
{
    Task<List<Asiakas>> GetAllCustomersAsync();
    Task<Asiakas?> GetCustomerByIdAsync(int id);
    Task AddCustomerAsync(Asiakas asiakas);
}
```

Kuva 6. AsiakasServicen rajapinta.

4.5.3 Riippuvuuksien injektointi

Sovelluksen rakenne hyödyntää riippuvuuksien injektointia (Dependency Injection). Sen avulla ohjelmiston osat, kuten palvelut ja PageControllerit voidaan erottaa toisistaan, jolloin sovellus on helpompi testata ja ylläpitää. Palvelut rekisteröidään Program.cs- tiedostoon kuten kuvassa 7.

```
builder.Services.AddScoped<IAsiakasService, AsiakasService>();
```

Kuva 7. Palvelun rekisteröinti.

4.6 Käyttöliittymä

Kun tietomallit oli määritelty ja tietokantayhteys luotu EF Coren avulla, siirryttiin käyttöliittymään. Razor Pages -sovelluksessa jokainen sivu

koostuu kahdesta tiedostosta näkymätiedostosta (.cshtml) ja sitä vastaavasta taustatiedostosta (.cshtml.cs).

Sivujen taustatiedostot periytyvät PageModel-luokasta ja niiden kautta hoidetaan sivuun liittyvä logiikka, kuten tietojen haut, lomakkeiden käsittelyt ja navigaatiot.

4.7 Asiakkaan lisäyslomake ja tietojen validointi

Asiakkaiden hallintaan liittyvistä toiminnoista yksi tärkeä on uuden asiakkaan lisääminen järjestelmään. Toiminto toteutettiin Razor Pages- sivuna, joka sisälsi lomakkeen asiakkaan perustietojen syöttämistä varten. Lomakkeen kenttien tietojen hakemiseen käytettiin Model Binding-ominaisuutta, jossa selaimessa täytetyt tiedot siirtyvät automaattisesti palvelinpuolen käsittelijälle OnPost metodille ViewModelin avulla

Sivulle luotiin kuvassa 8 esiintyvä Asiakkaan lisäämistä vastaava ViewModel, jossa oli tarvittavat kentät asiakkaan luontia varten. Lisäksi ViewModeliin lisättiin data-annotaatioita kenttien validointia varten

```
public class AsiakasViewModel
{
    public string Nimi { get; set; } = string.Empty;
    public string SyntymaAika { get; set; } = string.Empty;
    public string Osoite { get; set; } = string.Empty;
    public string PuhelinNumero { get; set; } = string.Empty;
    public string Sahkoposti { get; set; } = string.Empty;
}
```

Kuva 8. AsiakasViewModel.

Lomakkeessa käytettiin Tag-helper-elementtejä lomakekenttien määrittelyssä. Tag Helperit *asp-for* ja *asp-validation-for* sidottiin suoraan ViewModelin kenttiin. Tämä helpotti kehitystä ja varmisti, että kentät pysyivät synkronoituna palvelinpuolen mallin kanssa. Validointiviestit tulostettiin automaattisesti niiden kenttien alle, joissa virhe esiintyi.

```
<div class="form-group">
    <label asp-for="asiakasViewModel.Nimi" class="control-label"></label>
    <input asp-for="asiakasViewModel.Nimi" class="form-control" />
    <span asp-validation-for="asiakasViewModel.Nimi" class="text-danger"></span>
</div>
```

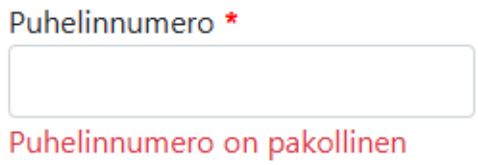
Kuva 9. Tag Helperit.

Kun lomake lähetettiin, palvelinpuolen OnPost- metodi vastaanotti syötetyt tiedot. Ennen tallentamista tiedot validoitiin automaattisesti *ModelState.IsValid* -tarkistuksen avulla

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

Kuva 10. Modelstate.

Jos *ModelState* ei ollut validi, sivu palautettiin takaisin käyttäjälle ja virheilmoitukset näytettiin lomakkeen yhteydessä kuten kuvassa 11.



Kuva 11. Virheilmoitus.

```
public class CreateModel : PageModel
{
    private readonly IAsiakasService _asiakasService;

    public CreateModel(IAsiakasService asiakasService)
    {
        _asiakasService = asiakasService;
    }
```

Kuva 12. Riippuvuuden injektointi.

4.8 Tietoturva

Tietoturva on huomioitu sovelluksen toteutuksessa. Koska käyttäjien hallinta toteutettiin ASP.NET Identityn avulla, voidaan luottaa vahvoihin

oletustoteutuksiin mm. salasanan tallennuksen osalta. Sovelluksen eri osioita voidaan rajata *Authorize* attribuutilla vain tietyille käyttäjärooleille esimerkiksi sovelluksen asetuksiin pääsy vain pääkäyttäjälle.

Kaikki lomakkeet validoidaan sekä asiakas- että palvelinpuolella. Lomakkeiden käsittelyssä hyödynnetään ModelState.IsValid -tarkistusta, jonka avulla estetään vääränmuotoisten ja vaarallisten tietojen tallennus.

Kaikki lomakkeet sisältävät automaattisesti Cross-Site Request Forgery -suojauksen Razor Pages sovelluksissa. Lomakkeet generoivat automaattisesti piilotetun kentän nimeltä *_RequestVerificationToken*, ja palvelin tarkistaa tämän tunnisteen jokaisessa POST-pyynnössä. Tämä suojaa sovellusta CSRF-hyökkäyksiltä ilman lisäkonfiguraatiota. (Exception not found, 2021)

Sovellus on oletuksena konfiguroitu käyttämään HTTPS-yhteyttä, mikä varmistaa, että kaikki tiedonsiirto käyttäjän ja palvelimen välillä on salattua. Sovelluksen API-avaimia tai tietokantayhteyden merkkijonoja ei tallenneta lähdekoodiin.

Lisäksi yksi tärkeimmistä tietoturvakäytännöistä on pitää kehitysalusta ja siihen liittyvät kirjastot säännöllisesti ajan tasalla. Päivitykset sisältävät usein korjauksia tunnetuille haavoittuvuuksille. Koska tietoturvauhat kehittyvät jatkuvasti ja uusia haavoittuvuuksia löydetään, frameworkien ja kirjastojen kehittäjät julkaisevat päivityksiä suojatakseen sovelluksia näiltä riskeiltä. Jos päivityksiä ei tehdä, sovellus jää alttiiksi jo tunnetuille haavoittuvuuksille, joita hyökkääjät voivat helposti käyttää hyväkseen. (Escape, 2023)

4.9 Asiakkaiden listaus

Asiakkaiden hallintaan tarvitaan myös selkeä ja toimiva listausnäkymä. Listausnäkymä toimii järjestelmän pääsivuna asiakashallinnan osalta, ja

sen kautta käyttäjä pääsee nopeasti tarkastelemaan ja muokkaamaan asiakkaita. Tämän vuoksi listauksen suunnittelussa korostui käytettävyyden ja suorituskyvyn merkitys, erityisesti silloin kun asiakastietoja on satoja tai tuhansia rivejä.

Listauksessa käytettiin C#-koodia HTML:n sisällä asiakkaiden läpikäymiseen. Tämä toteutettiin kuvan 13 kaltaisella Razor-syntaksilla, jotka mahdollistavat C#-kielen käyttämisen suoraan HTML-rakenteessa.

```
<tbody>
    @foreach (var asiakas in Model.Asiakkaat)
    {
        <tr>
            <td>@asiakas.Nimi</td>
            <td>@asiakas.Sahkoposti</td>
            <td>@asiakas.Puhelin</td>
        </tr>
    }
</tbody>
```

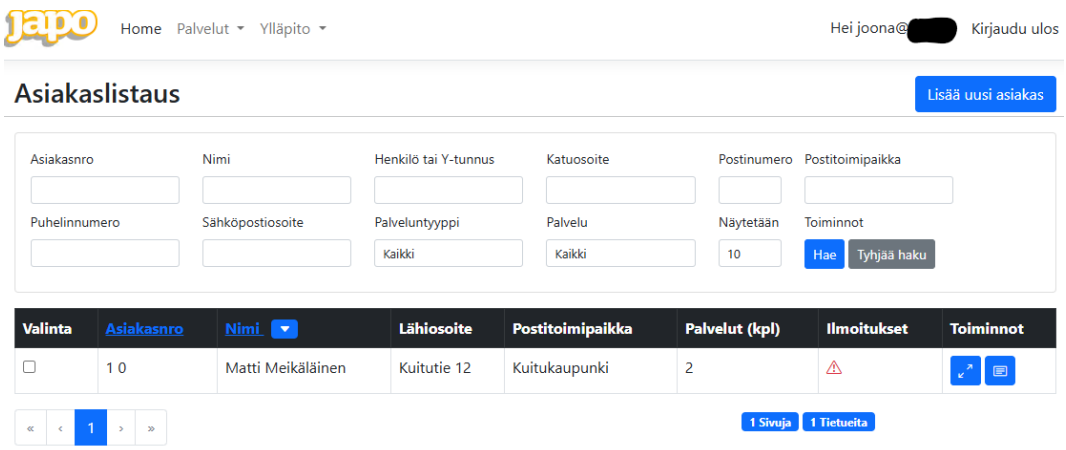
Kuva 13. Razor-syntaksi.

Asiakkaiden listausnäkymän käyttöliittymän toteutuksessa hyödynnettiin Bootstrap-kirjastoa. Bootstrap tarjosi valmiit komponentit esimerkiksi taulukkojen ja painikkeiden toteutukseen. Asiakaslistauksen taulukko toteutettiin käyttämällä table-komponenttia, joka mahdollisti selkeän tietojen esittämisen.

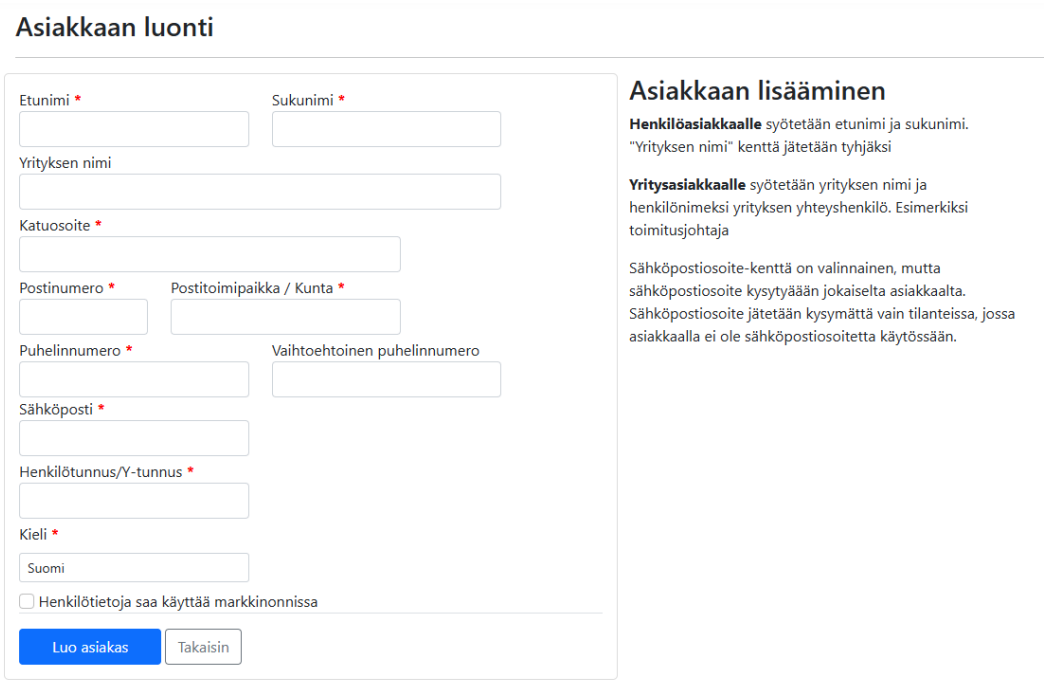
Listauksen toiminnallisuutta parannettiin lisäämällä hakutoiminnot, joiden avulla käyttäjä voi suodattaa asiakkaita eri hakuehdoilla, kuten nimen, paikkakunnan tai asiakasnumeron perusteella. Hakutoiminto toteutettiin lomakkeella, joka lähettää hakuehdot palvelimelle ja päivittää taulukon vastaamaan hakutuloksia. Käyttäjän syöttämät hakuehdot tallennetaan istuntoon, mikä varmistaa sen, että valitut suodattimen säilyvät, vaikka käyttäjä siirtyisi välillä toiseen näkymään ja palaisi takaisin asiakaslistaukseen.

Suodatuslogiikka toteutettiin AsiakasServicessä LINQ-lausekkeilla, jotka rajaavat näkymässä näytettävää asiakasjoukkoa annetun hakukriteerin

perusteella. Näin varmistetaan, että suodatus tapahtuu tehokkaasti jo tietokantakyselyn tasolla eikä vasta käyttöliittymässä, mikä on tärkeää erityisesti suurilla asiakasmäärillä. Listauksen käyttöliittymä suunniteltiin responsiiviseksi, jotta se toimii sujuvasti eri päätelaitteilla, kuten tietokoneilla ja puhelimilla. Kuvassa 14 on kuva asiakaslistausnäkymästä ja kuvassa 15 asiakkaiden lisäysnäkymästä



Kuva 14. Asiakaslistaus.



Kuva 15. Asiakkaan luonti.

5 POHDINTA JA JATKOKEHITYS

Tämän opinnäytetyön tavoitteena oli kehittää nykyaikainen ja ylläpidettävä asiakashallintajärjestelmä Japo Palvelut Oy:lle korvaamaan vanha, Java-pohjainen järjestelmä. Projekti toteutettiin hyödyntäen Microsoftin uusinta ASP.NET Core 9 -kehystä, Razor Pages -käyttöliittymämallia, C#-ohjelmointikieltä ja Entity Framework Corea tietokannan hallintaan. Samalla työ toimi tekijälleen oppimisprojektina, jossa syvennyttiin modernien web-teknologioiden käyttöön liiketoimintasovelluksen kehittämisessä.

Projektin aikana onnistuttiin toteuttamaan toimiva asiakashallintamoduuli, jonka avulla voidaan lisätä, muokata ja hallita asiakastietoja sekä niihin liittyviä palveluita ja tuotteita. Uuden järjestelmän rakenne on selkeämpi ja joustavampi kuin vanhassa sovelluksessa, mikä helpottaa sekä jatkokehitystä että ylläpitoa. Razor Pages -mallin käyttö osoittautui toimivaksi ratkaisuksi sovelluksen sivupohjaiseen rakenteeseen, ja se mahdollisti loogisesti eriytetyn, mutta yhtenäisen kehitysmallin näkymien ja niihin liittyvän logiikan välillä.

Kehitystyökaluna käytetty Visual Studio 2022 tarjosi tehokkaan ympäristön, jossa kaikki sovelluksen osat saatiin rakennettua hallitusti. Versionhallinnan toteuttaminen Gitin avulla loi hyvän pohjan jatkokehitykselle.

Järjestelmä tarjoaa hyvän pohjan jatkokehitykselle. Kehittämistyötä voidaan suunnata tekniseen parantamiseen ja toiminnalliseen laajentamiseen tarpeiden mukaan. Tällaisia tarpeita ovat muun muassa integraatiot muihin järjestelmiin, raportointityökalujen kehittäminen ja mobiilikäytön parantaminen.

Lisäksi ohjelmiston elinkaaren kannalta tärkeää on myös ylläpidettävyys ja tietoturvan huomioiminen. Näihin liittyviä jatkokehitysmahdollisuuksia voivat olla esimerkiksi automattisten testien, versionhallintaputkien ja säännöllisten päivityksien käyttöönotto. Jatkokehityksessä huomioidaan myös käyttäjäpalautetta, jonka avulla järjestelmän käytettävyyttä ja hyödyllisyyttä voidaan parantaa entisestään.

LÄHTEET

Escape. (2023) *ASP.NET security best practices.* Noudettu 28.5.2025 osoitteesta: https://escape.tech/blog/asp-dot-net-security/

Exception not found. (2021) *Using Anti-Forgery Tokens in ASP.NET 5.0 Razor Pages.* Noudettu 28.5.2025 osoitteesta: https://exceptionnotfound.net/using-anti-forgery-tokens-in-asp-net-core-razor-pages/

Learn Razor Pages. (2021a) *The Razor Pages PageModel.* Noudettu 12.5.2025 osoitteesta: https://www.learnrazorpages.com/razor-pages/pagemodel/

Learn Razor Pages. (2022a) *Configuring a Razor Pages application.* Noudettu 12.5.2025 osoitteesta: https://www.learnrazorpages.com/application-configuration/

Microsoft. (2023a). *Migrations Overview - EF Core*. Noudettu 13.4.2025 osoitteesta: https://learn.microsoft.com/en-us/ef/core/managing-schemas/migrations/

Microsoft. (2024a). *Introduction to ASP.NET Core*. Noudettu 10.4.2025 osoitteesta https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core

Microsoft. (2024b). *A tour of the C# language.* Noudettu *10.4.2025 osoitteesta https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/overview*

Microsoft. (2024c). *Overview of Entity Framework Core*. Noudettu 10.4.2025 osoitteesta https://learn.microsoft.com/en-us/ef/core/

Microsoft. (2024d). *Querying Data - EF Core*. Noudettu 12.4.2025 osoitteesta https://learn.microsoft.com/en-us/ef/core/

Microsoft. (2024e). *What is Visual Studio?* Noudettu 15.4.2025 osoitteesta: https://learn.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide?view=vs-2022

Microsoft. (2024f). *Choose an ASP.NET Core web UI.* Noudettu 5.5.2025 osoitteesta https://learn.microsoft.com/en-us/aspnet/core/tutorials/choose-web-ui?view=aspnetcore-9.0