# Webbutveckling med .NET Blazor och MudBlazor

Emil Dahlberg

Examensarbete för ingenjör (YH)-examen

El- och automationsteknik

Vasa 2024

**EXAMENSARBETE**

Författare: Emil Dahlberg
Utbildning och ort: El- och automationsteknik, Vasa
Inriktning: Informationsteknik
Handledare: Susanne Österholm

Titel: Webbutveckling med .NET Blazor och MudBlazor

---

Datum: 17.3.2024 Sidantal: 39

---

**Abstrakt**

Detta examensarbete gjordes på begäran av Hogia Ferry Systems, som är ett företag som utvecklar mjukvara för hantering av passagerar- och fraktbokningar inom sjötrafik runtom i världen. Arbetet fokuserade på utvecklingen av en webbsida för att söka och visa olika case från en databas som tidigare varit knuten till ett CRM-system. Syftet med projektet var att göra det mer effektivt för att hitta och hantera case samt att minska på resurser, såsom serverkostnader, genom att avveckla det äldre systemet. Genomförandet av projektet involverade användning av Blazor och MudBlazor för att bygga användargränssnittet samt ADO.NET för att hantera databasanslutningen. Ytterligare NuGet-paket användes för att implementera specifika funktioner.

Resultatet av projektet blev en enkel och effektiv webbsida med avancerade filtrerings- och sorteringsmöjligheter för att söka efter case. Den detaljerade sidan för varje case var tydligt utformad och innehöll en översiktlig presentation av caseinformation samt möjligheten att se e-postkedjor och bifogade filer. Sammantaget möjliggjorde den utvecklade webbsidan en smidig och användarvänlig hantering av case, vilket bidrog till ökad effektivitet och minskade kostnader för organisationen.

---

Språk: svenska
Nyckelord: BOOKIT, Blazor, MudBlazor, CRM-system, Hogia

**BACHELOR'S THESIS**

Author: Emil Dahlberg
Degree Programme: Electrical Engineering and Automation
Specialisation: Information Technology
Supervisor(s): Susanne Österholm

Title: Web Development with .NET Blazor and MudBlazor

---

Date: March 17, 2024 Number of pages: 39

---

**Abstract**

This thesis was conducted at the request of Hogia Ferry Systems, a company that develops software for managing passenger and freight bookings in maritime traffic around the world. The work focused on the development of a website for searching and displaying various cases from a database that was previously linked to a CRM system. The purpose of the project was to make it more efficient to find and manage cases and to reduce resources, such as server costs, by shutting down the older system. The project implementation involved the use of Blazor and MudBlazor to build the user interface, as well as ADO.NET to handle the database connection. Additional NuGet packages were used to implement specific functionalities.

The result of the project was a simple and efficient website with advanced filtering and sorting capabilities for searching cases. The detailed page for each case was clearly designed and included an overview presentation of case information as well as the ability to view email threads and attached files. Overall, the developed website enabled smooth and user-friendly case management, contributing to increased efficiency and reduced costs for the organization.

---

Language: Swedish
Keywords: BOOKIT, Blazor, MudBlazor, CRM system, Hogia

# Innehållsförteckning

1 Inledning ... 1
1.1 Bakgrund ... 1
1.2 Mål och krav ... 2
1.3 Uppdragsgivare ... 3
2 Tekniker och metoder ... 4
2.1 Microsoft ASP.NET Core ... 4
2.2 ASP.NET Core MVC ... 4
2.3 ASP.NET Core Razor Pages ... 5
2.4 ASP.NET Core Blazor ... 7
3 Databasinteraktion i .NET ... 8
3.1 ADO.NET ... 8
3.2 Entity Framework ... 10
4 Tekniker att söka fram data ... 12
4.1 Filtrering ... 12
4.2 Sökning ... 13
4.3 Sökhistorik ... 15
4.4 Sortering ... 16
5 Layout ... 20
5.1 HTML och CSS ... 20
5.2 CSS-ramverk ... 22
5.3 UI-komponentbibliotek ... 24
6 Resultat och utförande ... 25
6.1 Databaskoppling och val av ramverk ... 25
6.2 Byggandet av webbsidan ... 26
6.3 Bifogade filer ... 29
6.3.1 Bilder ... 30
6.3.2 Dokumentfiler ... 31
6.4 Optimering av sökfunktionaliteten ... 32
6.5 Finjusteringar och detaljer ... 34
7 Diskussion ... 37
8 Litteraturförteckning ... 38

## Ordförklaringar

| | |
|---|---|
| localStorage | localStorage är en funktion i webbläsaren som låter webbsidor spara små mängder data på användarens enhet. Datan förblir tillgänglig även efter att webbläsaren har stängts av. |
| sessionStorage | sessionStorage är en funktion i webbläsare som låter webbsidor spara små mängder data på användarens enhet. Men till skillnad från localStorage, försvinner datan när webbläsarsessionen avslutas |
| NuGet-paket | NuGet-paket är fördefinierade samlingar av kod, bibliotek och andra resurser som distribueras och används för att underlätta utvecklingen av programvaruprojekt. |
| frontend | Frontend är den del av en webbapplikation som användarna ser och interagerar med i sina webbläsare. |
| foreign keys | Foreign keys är attribut i en databastabell som hänvisar till primärnycklar i en annan tabell. Dessa används för att etablera en relation mellan tabeller i en relationell databas. |
| Azure DevOps | Azure DevOps är en molnbaserad tjänst från Microsoft för att hantera och automatisera mjukvaruutvecklingsprocessen. |
| Guid | "Guid" är en förkortning för "Globally Unique Identifier" och används för att skapa unika identifierare inom dataprogrammering. |
| SQL Parameters | SQL-parameters är en metod för att använda variabler i SQL-frågor för att göra dem säkrare och mer effektiva. |
| SQL-injection | SQL-injection är en säkerhetssårbarhet i databasapplikationer där angriparen injicerar skadlig SQL-kod för att manipulera databasen genom att utnyttja brister i inmatningshanteringen. |
| CRM-system | Ett CRM-system hanterar företagets relationer med kunder genom att spåra information. |
| SASS och LESS | SASS och LESS är CSS-preprocessorer som tillåter användning av variabler, mixins och andra avancerade funktioner för att effektivisera och organisera CSS-kod. |

# 1 Inledning

Det här arbetet genomfördes för Hogia Ferry Systems (HFS), där syftet var att skapa en webbsida som gav en möjlighet att läsa data ur ett CRM-system.

När man idag går in på en webbsida, strävar man efter att allting ska vara snabbt och prestera, utmaningen är dock att man i bakgrunden behöver hantera en väldigt stor mängd data utan att det negativt påverkar prestandan. En annan strävan är också att hitta det man söker när man skriver in något i sökfältet. När du söker något på internet idag finns det tekniker man kan använda för att hitta en relevant webbsida. Dessa tekniker har alla en egen logik som körs för att minimera de resultat som ges. Att implementera en liknande logik kräver noggrann analys och forskning för att hitta den bästa lösningen som både uppfyller användarnas behov och ger önskad prestanda.

I detta arbete tog man ta reda på hur man kunde skapa en webbsida som gav en den önskade prestandan man ville ha. Det innebar att man använde olika strategier för att hantera och presentera data, samt granskade olika söklogiker för att säkerställa att användarna snabbt och enkelt kunde hitta det de sökte. Genom ett noggrant och metodiskt tillvägagångssätt blev webbsidan inte bara funktionell och effektiv, utan gav också en positiv användarupplevelse för HFS-användare.

## 1.1 Bakgrund

Under 2022 övergick Ferry Systems, dotterbolag till Hogia, från sitt tidigare CRM-system, som hade använts under många år, till det gemensamma CRM-systemet för företaget, kallat 'Hogia Dynamics 365'. CRM-systemet bestod av flera case som direkt översatt kunde kallas ärenden. Dessa case innehåller information, som till exempel kundönskemål, buggar med mera.

Endast pågående case, samt case där det skett någon aktivitet under det senaste året migrerades till det nya systemet. Men det fanns fortfarande ett behov av att komma åt den information som fanns sparad i vissa case i det gamla systemet. Till att börja med var tanken endast att behålla databasen och stänga ner webbsidan och endast göra SQL-frågor direkt mot databasen, men i ett senare skede föddes tanken att bygga en egen webbsida för att underlätta för användarna att söka och få fram den information de sökte.

## 1.2 Mål och krav

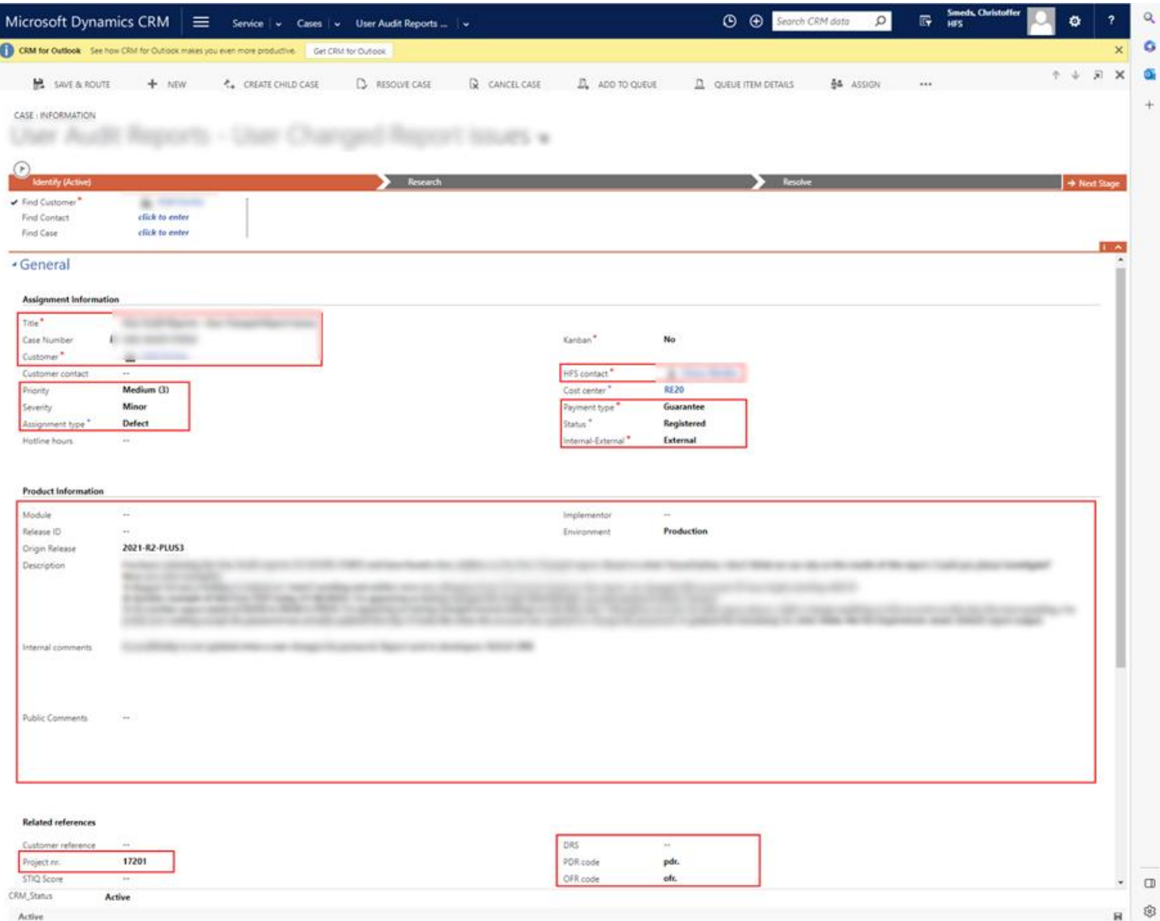
Målet med detta projekt var att utveckla en användarvänlig och effektiv webbsida för sökning ur en databas. Genom att skapa en sökfunktionalitet som är både snabb och smart strävade man efter att erbjuda användarna en smidig och tillfredsställande upplevelse.

För att uppnå detta planerades webbsidans grund enligt följande:

- Webbplatsen skulle ha en effektiv sökfunktionalitet som gjorde det möjligt för användare att söka efter case baserat på en mängd olika kriterier, som nyckelord, datum eller kategorier.

- SQL-frågorna behövde vara formulerade i format som stöds av databashanteraren SQL Server

- Följande ämnen skulle visas för ett case: (syns även i Figur 1)
    - Titel
    - Case-nummer
    - Kundnamn
    - Prioritet
    - Allvarlighetsgrad
    - Uppdragstyp
    - HFS-kontaktperson
    - Betalningsmetod
    - Status
    - Intern eller Extern
    - Modul
    - Release-Id och Origin Release.
    - Beskrivning
    - Interna och externa kommentarer
    - Projektnummer
    - DRS
    - PDR-kod
    - OFR-kod
    - E-postkedjor




**Figur 1: En bild från det gamla CRM-systemet med markering av de ämnen som behövde finnas på nya webbsidan.**

- Utseendemässigt skulle webbsidan ha ett lättförståeligt och användarvänligt gränssnitt som gjorde det enkelt för användare att navigera, söka och hitta information.

## 1.3 Uppdragsgivare

Hogia Ferry Systems är ett helägt dotterbolag till Hogia Ab som är ett programvaruföretag från Sverige med 650 anställda. Det grundades 1980 av Bert-Inge Hogsved. Företaget skapar och säljer administrativ programvara för en mängd olika områden, såsom redovisning, löner, personalresurser och transport. Hogia Ferry Systems huvudprodukt, BOOKIT, är ett bokningssystem som fokuserar på färjeresor och färjefrakt. Hela bokningsprocessen sköts av BOOKIT, från att samla in information om passagerare, bilar och frakt till att göra fakturor och rapporter. Systemet har gjorts för att vara anpassningsbart och flexibelt för alla typer av färjebranscher, och det har funktioner för rutthantering, kapacitetshantering,

prissättning och intäktskontroll. BOOKIT kan också sälja relaterade varor som ombordtjänster, boende och relaterade resebiljetter. (HFS, 2024).

# 2 Tekniker och metoder

I det här avsnittet berättas det om olika teoretiska begrepp som ligger till grund för förståelsen och användningen av de teknologier och metoder som kan användas i detta arbete.

Genom att välja Microsofts egna mallar för att bygga webbsidan skapas inte bara en positiv integration med företagets existerande arkitektur, utan det möjliggör även en bekväm och effektiv arbetsmiljö. Denna strategi ger en dessutom fördelen av att dra nytta av de senaste innovationerna från Microsoft samtidigt som det gynnar den tekniska grunden som företaget använder sig av dagligen.

## 2.1 Microsoft ASP.NET Core

ASP.NET Core är ett kraftfullt och flexibelt ramverk som används för att skapa applikationer som kan köras över olika plattformar, anslutas till molnet och vara uppkopplade till internet. Med ramverket har utvecklare möjlighet att bygga webbapplikationer, tjänster, applikationer för Internet of Things och en mobil backend.

Ramverket är en omarbetad version av den tidigare ASP.NET 4.x-versionen och innefattar strukturella förändringar för att skapa en smidigare och mer modulär ram. ASP.NET Core inkluderar även stöd för moderna front-end-ramverk som Angular, React, Vue och Bootstrap. Ramverket gör det även möjligt för webb-API:er att användas på olika enheter, som webbläsare och mobila enheter. (Microsoft, Overview of ASP.NET Core, 2023).

## 2.2 ASP.NET Core MVC

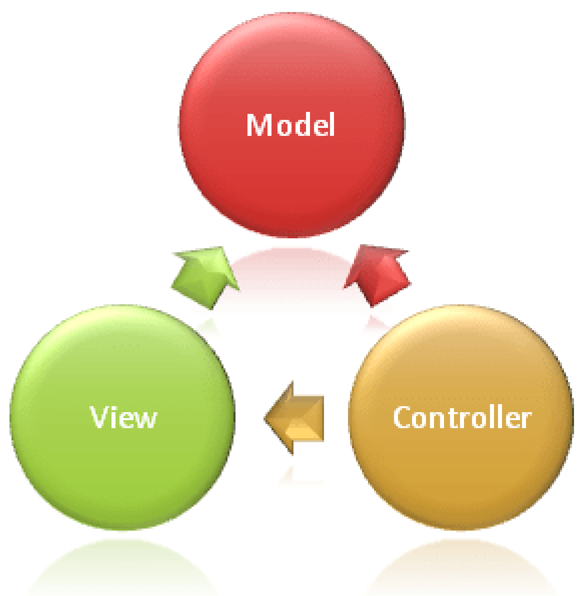
MVC, även känt som Modell-Vy-Kontroll, är en slags plan eller mönster som används av människor när de bygger sina applikationer eller webbsidor. Det hjälper till att organisera koden i tre delar: Modeller, Vyer och Kontrollers.

Modellen är som hjärnan, den har all information och regler om hur saker fungerar. Om något ändras, informerar den de andra delarna.

Vyn är det som användaren ser och integrerar med. Om man klickar på något, berättar vyn för kontrollern att något hände. Den håller också koll på om modellen säger att något har ändrats och uppdaterar sig själv då.

Kontrollern är som dirigenten. Den tar emot signaler från vyn när någon gör något och säger åt modellen vad den ska göra. Den bestämmer också vad användaren ska se, alltså vilken vy som visas.

Det här systemet gör det enklare att jobba med koden. Om man vill ändra hur applikationen ser ut, justerar man bara vyn. Om det handlar om hur saker fungerar, ändrar man bara modellen. Det är också bra när flera personer jobbar på samma applikation, för de kan jobba med olika delar utan att störa varandra.




**Figur 2. De tre komponenterna som definierar MVC. (Microsoft, Overview of ASP.NET Core MVC, 2023).**

## 2.3 ASP.NET Core Razor Pages

Razor Pages är en teknik för att skapa webbsidor. Den använder filer för att bestämma vart man ska navigera. Varje fil för Razor Pages under Pages-mappen betyder en plats man kan navigera till. I Razor Pages, agerar varje C#-komponent som en sidomodell och ansvarar för att definiera beteendet för varje sida. Den fungerar som en hanterare för sidans logik och hanterar hur informationen på sidan ska visas eller bearbetas. Komponenten möjliggör en

ren och välstrukturerad uppdelning av ansvarsområden för att säkerställa att varje sida utför sina specifika uppgifter effektivt. Varje sida jobbar dessutom enskilt med olika saker, till exempel som att visa information eller förmedla data. (Abuhakmeh, 2020).

**Kodexempel 1. En enkel Razor Page.**

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<div class="text-center">
    <h1 class="display-4">Welcome @Model.Name</h1>
</div>

<form method="post" asp-page="Index">
    <div class="form-group">
        <label asp-for="Name"></label>
        <input asp-for="Name" class="form-control" placeholder="Enter Name" auto-
complete="off" />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

Kodexempel 1 beskriver en Razor Pages-sida i en ASP.NET Core-webbapplikation. @page anger sidans sökväg och @model IndexModel specificerar modellen för sidan. ViewData["Title"] sätter titeln för sidan. Resten av koden bygger upp en enkel layout med HTML och tillsammans med Bootstrap ger den en finare layout.

**Kodexempel 2. C#-kod för Razor sidan ovan.**

```
public class IndexModel : PageModel
{
    public string Name => (string)TempData[nameof(Name)];

    public void OnGet()
    {
    }

    public IActionResult OnPost([FromForm] string name)
    {
        TempData["Name"] = name;
        return RedirectToPage("Index");
    }
}
```

Kodexempel 2 är C#-kod i ASP.NET Core för den ovannämnda Razor-sidan. Klassen IndexModel ärver från PageModel.

## 2.4 ASP.NET Core Blazor

Även Blazor är ett frontend webbramverk för .NET, men till skillnad från de tidigare dokumenterade webbramverken kan Blazor köras både server-side och client-side. Det låter dig skapa interaktiva användargränssnitt med C# och dela logik mellan server och klient, allt med en enkel programmeringsmodell. Man kan visa användargränssnittet som HTML och CSS, vilket fungerar bra i olika webbläsare, även på mobila enheter. Blazor gör det även möjligt att bygga applikationer för både skrivbord och mobiler med .NET.

Att använda .NET för klientbaserad webbutveckling har sina fördelar. Koden skrivs i C#, vilket gör det lättare att utveckla och underhålla applikationer. Man kan dra nytta av det befintliga .NET-ekosystemet med .NET-biblioteket och få fördelarna av .NET:s prestanda, pålitlighet och säkerhet. Dessutom kan man vara produktiv oavsett om man använder Windows, Linux eller macOS med utvecklingsmiljöer som Visual Studio eller Visual Studio Code. Blazor bygger på komponenter, som är .NET C#-klasser, och dessa definierar logik för att rendera användargränssnitt, hantera användarinteraktioner och kan återanvändas och distribueras som Razor-klassbibliotek eller NuGet-paket.

Komponenterna skrivs oftast som Razor-markup-pages (.razor-filer), där Razor är en syntax för att kombinera HTML-märkning med C#-kod för utvecklarproduktivitet. Blazor använder naturliga HTML-taggar för att bygga användargränssnittet. Kodexempel 3 illustrerar en komponent som ökar en räknare när användaren klickar på en knapp. Komponenterna renderas till en minnesrepresentation av webbläsarens Document Object Model (DOM) även kallat renderingsträd, som i sin tur används för att uppdatera användargränssnittet på ett flexibelt och effektivt sätt. (Microsoft, ASP.NET Core Blazor, 2023).

**Kodexempel 3. Blazor-komponent.**

```
<PageTitle>Counter</PageTitle>

<h1>Counter</h1>

<p role="status">Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}
```

# 3 Databasinteraktion i .NET

I det här kapitlet kommer det tas upp olika databashanterare som kan tänkas användas i arbetet. Att skapa användbara och effektiva webbsidor kräver att man stannar till och tänker vilken databashanterare som kommer vara bäst för ens applikation. Två alternativ är ADO.NET och Entity Framework som representerar två vanliga sätt att kommunicera med databaser inom .NET-ramverket.

## 3.1 ADO.NET

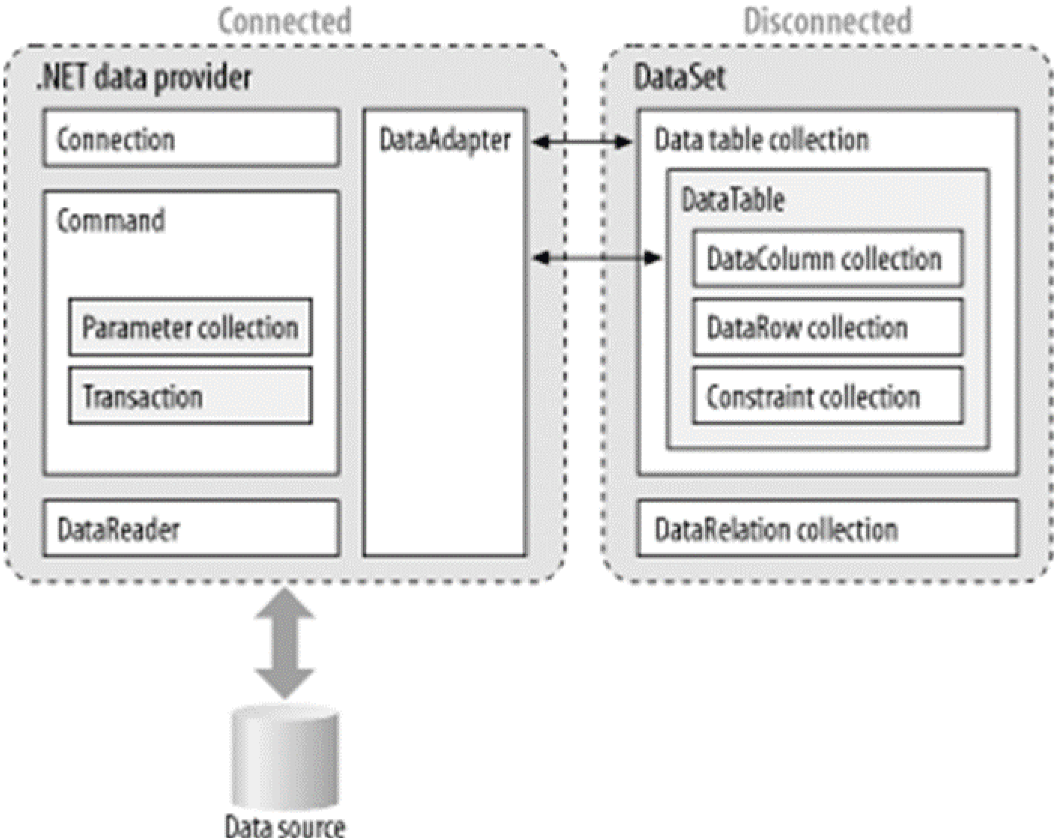
ADO.NET är en del av .NET Framework-komponenten skapad av Microsoft. Med ADO.NET:s samling av klasser och tekniker kan man få åtkomst till datakällor och hämta data till C#- och VB.NET-projekt för att sedan kunna skapa högpresterande samt pålitliga databasbaserade applikationer. (Microsoft, ADO.NET Overview, 2021).

ADO.NET-arkitekturkoncept är designat så att man kan utföra operationer mot olika typer av datakällor på samma sätt. Det finns två sätt att kommunicera med en databas med ADO.NET.

Det ena sättet är att göra det med Connected-arkitekturen som är en dataåtkomstarkitektur som alltid kräver en öppen anslutning till databasen. DataReader-klassen ger snabbare prestanda för mindre applikationer och datamängder. DataReader-klassen kan läsa databasens data på ett framåt- och endast-läsbart sätt, alltså det går inte att hoppa tillbaka till tidigare data efter att den har passerats. Det används effektivt för att läsa data och kräver en konstant anslutning. (Tuğanlı, 2023).

Det andra sättet man kan använda är Disconnected-arkitekturen som är en dataåtkomstarkitektur som inte kräver en kontinuerlig aktiv och öppen anslutning till databasen. I stället skapas anslutningen endast när information läses eller uppdateras i databasen. I.NET-miljön görs detta med hjälp av DataAdapter, DataSet eller DataTable. Dessa klasser låter dig hämta data från databaser och lagra dem temporärt i minnet för bearbetning. Detta ger bäst prestanda för applikationer som använder fler lager och stora datamängder. (Tuğanlı, 2023).

Figur 3 illustrerar förhållandet mellan en .NET Framework-dataleverantör och ett DataSet.




**Figur 3. ADO.NETs arkitektur.**

Förutsatt att man skapat en anslutningssträng till databasen, som ofta i kodspråk heter connectionString, kan man göra som i Kodexempel 4 för att hämta data som produktnummer, pris och namn av produkter där priset är högre än en specifik prisgräns.

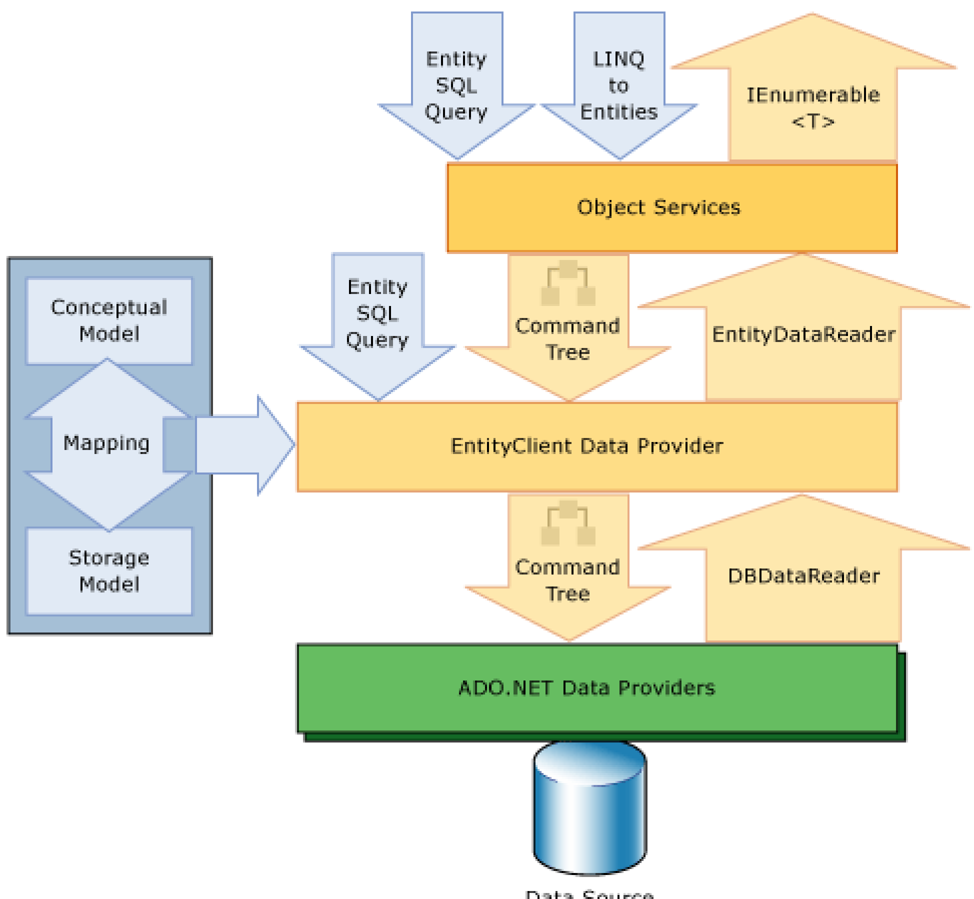
**Kodexempel 4: Ett exempel på hur man kan hämta data med ADO.NET.**

```
const string queryString =
    "SELECT ProduktNr, Pris, ProduktNamn from dbo.produkter"
        + "WHERE Pris > @prisGräns "
        + "ORDER BY Pris DESC;";

const int paramValue = 5;

using (SqlConnection connection =
    new(connectionString))
{
    SqlCommand command = new(queryString, connection);
    command.Parameters.AddWithValue("@prisGräns ", paramValue);

    connection.Open();
    SqlDataReader reader = command.ExecuteReader();
    while (reader.Read())
    {
        Console.WriteLine("\t{0}\t{1}\t{2}",
            reader[0], reader[1], reader[2]);
    }
    reader.Close();
}
```

## 3.2 Entity Framework

Entity Framework som är en samling av ADO.NET-tekniker som hjälper till att skapa dataorienterade mjukvaruapplikationer. Entity Framework gör det möjligt för utvecklare att arbeta med data i form av domänspecifika objekt och egenskaper, såsom kunder och kundadresser, utan att behöva bry sig om de underliggande databastabellerna och kolumnerna där denna data lagras. Med Entity Framework kan utvecklare arbeta på en högre abstraktionsnivå när de hanterar data och kan skapa och underhålla dataorienterade applikationer med mindre kod än i traditionella applikationer. (Microsoft, Entity Framework overview, 2021).

I figur 4 följer ett Entity Framework-arkitekturdiagram som visar Entity Frameworks komponenter och hur de samverkar för att hämta data.




**Figur 4. Entity Frameworks arkitektur. (Microsoft, Entity Framework overview, 2021).**

Entity Framework Core, även kallat EF Core, är en lätt, utbyggbar och plattformsoberoende version av Entity Framework. Ramverket gör det möjligt för .NET-utvecklare att arbeta med en databas med .NET-objekt. Detta eliminerar behovet av det mesta av dataåtkomstkoden

som vanligtvis behöver skrivas. Med EF Core fås dataåtkomst med hjälp av en modell. En modell består av entitetsklasser och ett kontextobjekt som representerar en session med databasen. Kontextobjektet gör det möjligt att fråga och spara data. (Microsoft, Entity Framework Core, 2021).

**Kodexempel 5. En EF Core-modell.**

```
public class ProduktContext : DbContext
{
    public DbSet<Produkt> Produkter { get; set; }
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(
            @"Server=(localdb)\mssqllocaldb;Database=ProductDb;Trusted_Connec-
tion=True");
    }
}
public class Produkt
{
    public int ProduktNr { get; set; }
    public decimal Pris { get; set; }
    public string ProduktNamn { get; set; }
}
```

SQL-frågor till databasen sker med hjälp av LINQ (Language Integrated Query) (se Kodexempel 6).

**Kodexempel 6. SQL-frågor görs enligt denna kod.**

```
var minPris = 3;

using (var db = new ProduktContext())
{
    var products = db.Produkter
                     .Where(b => b.Pris > minPris)
                     .OrderBy(b => b.Pris)
                     .ToList();
}
```

EF Core gör programmering lättare, men kräver kunskap för att förhindra problem i produktionsapplikationer, som till exempel: (Microsoft, Entity Framework Core, 2021).

- För att optimera prestanda måste man kunna förstå databasservern.
- Man behöver kunna genomföra noggranna tester, inklusive prestanda- och versionstest
- Kontroller av säkerhetsanslutningssträngar och hantera känslig information behövs.

- Man behöver även se till att det finns tillräckligt med datainsamling och diagnostik.
- Det krävs även att man förbereder sig för felåterställning och följa noggrant med migrationsprocessen.
- För att undvika problem med produktionsdata behöver man kunna testa och granska noggrant genererade migreringar.

# 4 Tekniker att söka fram data

I detta kapitel berättas det om tekniker man kan använda för att söka, manipulera och hantera data.

När man skapar en applikation för att hämta och visa data från en databas är det viktigt att använda effektiva sökmetoder. En välutvecklad sökfunktion är avgörande för att användarna ska kunna få relevant och snabb tillgång till det de söker. Sökmetoderna i appen är avgörande för att förbättra användarupplevelsen och förbättra funktionaliteten. I denna ständigt föränderliga teknologiska värld är en kraftfull sökfunktion avgörande för att optimera appens användbarhet och framgång.

## 4.1 Filtrering

Filtrering är processen att begränsa resultatuppsättningen till endast de element som uppfyller ett specifikt villkor.

**Kodexempel 7. Filtrering med iterativ metod. (Chidera, 2022).**

```
foreach (var employee in employees)
{
    if (employee.Department == "Software")
    {
        Console.WriteLine(employee.Name);
    }
}
```

I kodexempel 7 används den iterativa metoden. Här loopar man igenom en lista och letar efter medlemmen som uppfyller villkoret i varje iteration. I det här fallet har en lista över anställda filtrerats för att identifiera de som arbetar inom företagets mjukvaruavdelning.

**Kodexempel 8. Filtrering med LINQ-frågesyntax. (Chidera, 2022).**

```
var filteredResults = from employee in employees
                      where employee.Department == "Software"
                      select employee.Name;

foreach (var result in filteredResults)
{
    Console.WriteLine(result);
}
```

Ett annat effektivt sätt att hämta data från C#-källor är genom att använda LINQ-frågor. I kodexempel 8 ser man att det man används en LINQ-frågesyntax med WHERE-delen för att filtrera listan och returnera en ny lista baserat på ett specifikt kriterium.

**Kodexempel 9. Filtrering med LINQ-metodsyntax. (Chidera, 2022).**

```
var filteredResultsTwo = employees.Where(employee => employee.Department ==
"Software");

foreach (var employee in filteredResultsTwo)
{
    Console.WriteLine(employee.Name);
}
```

I kodexempel 9 används LINQ-metodsyntaxen som använder ett lambda-uttryck i stället för en frågesyntax. Lambda-uttrycket är en villkorsfunktion och en genväg till LINQ-frågesyntaxen. (Chidera, 2022).

## 4.2 Sökning

När du skapar SQL-frågor för att söka genom flera tusentals rader i en databastabell är det viktigt att tänka på hur du kan optimera sökningen för att uppnå både snabbhet och träffsäkerhet. När det gäller wildcard-sökningar, där procenttecken placeras före eller efter ett ord för att hitta resultat som innehåller det ordet, kan det ibland resultera i betydande fördröjningar i söktiden. Ett exempel på en ineffektiv SQL-fråga visas i Kodexempel 10.

**Kodexempel 10. Exempel på en resurskrävande SQL-fråga.**

```
SELECT * FROM CustomerTable WHERE CustomerTransport LIKE '%ship%'
```

LIKE-operatorn i SQL kan orsaka oförutsedda prestandaproblem, särskilt när procenttecknet placeras på ett visst sätt. För att optimera, är det bra att undvika procenttecken på fel ställen och överväga att använda fulltextsökning och index. (Winand, 2024).

En LIKE-förfrågan mot en betydande mängd ostrukturerade textdata är mycket långsammare än en motsvarande fulltextförfrågan mot samma data. En fulltextförfrågan mot samma data kan ta några sekunder eller mindre – beroende på antalet rader som returneras – medan en LIKE-förfrågan mot miljontals rader med textdata kan ta minuter. (Microsoft, Full-Text Search, 2023).

För att använda sig av fulltextsökningar kan man ta hjälp av predikaten CONTAINS och FREETEXT. I kodexempel 11 visar man hur man kan hitta alla produkter ur en lista med priset 80.99 som innehåller ordet ”Mountain”.

**Kodexempel 11. Att använda CONTAINS. (Microsoft, Query with Full-Text Search, 2023).**

```
USE AdventureWorks2022
GO

SELECT Name, ListPrice
FROM Production.Product
WHERE ListPrice = 80.99
   AND CONTAINS(Name, 'Mountain')
GO
```

Strävar man hellre efter att hitta till exempel dokument som innehåller ord relaterade till ett specifikt sökkriterium kan man göra som i kodexempel 12.

**Kodexempel 12. Att använda FREETEXT. (Microsoft, Query with Full-Text Search, 2023).**

```
USE AdventureWorks2022
GO

SELECT Title
FROM Production.Document
WHERE FREETEXT (Document, 'vital safety components')
GO
```

Enskilda ord och fraser kan matchas med CONTAINS-metoden med både exakt och otydlig matchning. Dessutom tillåter den användaren att specificera ords närhet inom ett visst avstånd, returnera träffar med vikt och integrera sökvillkor med logiska operatorer som AND, OR och NOT.

FREETEXT-funktionen följer betydelsen av ord, fraser eller meningar utan att ändra formuleringen. Om en viss term finns i fulltextindexet för en viss kolumn genereras träffar. (Microsoft, Query with Full-Text Search, 2023).

## 4.3 Sökhistorik

Att använda localStorage i webbläsaren för att skapa en sökhistorik är enkelt och användbart för att spara användarens tidigare sökningar. Ett enkelt exempel på hur man kan implementera detta med JavaScript i en Blazor-komponent görs i kodexempel 13.

**Kodexempel 13. Blazor-komponent med localStorage**

```
@page "/"
@inject IJSRuntime JSRuntime


<label for="searchInput">Sök:</label>
<input id="searchInput" @bind="searchTerm" />
<button @onclick="AddToHistory">Sök</button>

<h4>Sökhistorik:</h4>
<ul>
    @foreach (var item in searchHistory)
    {
        <li>@item</li>
    }
</ul>

@code {
    private string searchTerm;
    private List<string> searchHistory = new List<string>();

    protected override void OnInitialized()
    {
        LoadSearchHistory();
    }

    private async void AddToHistory()
    {
        if (!string.IsNullOrWhiteSpace(searchTerm))
        {
            searchHistory.Add(searchTerm);
            await JSRuntime.InvokeVoidAsync("localStorage.setItem",
"searchHistory", JsonSerializer.Serialize(searchHistory));
            searchTerm = string.Empty;
        }
    }

    private async void LoadSearchHistory()
    {
        var storedHistoryJson = await
JSRuntime.InvokeAsync<string>("localStorage.getItem", "searchHistory");
        if (!string.IsNullOrWhiteSpace(storedHistoryJson))
        {
            searchHistory =
JsonSerializer.Deserialize<List<string>>(storedHistoryJson);
        }
```

```
        }
```

I Kodexempel 13 skapar vi med HTML en vanlig söklogik tillsammans med en lista som förtecknar alla sökningar som gjorts. I delen under vår HTML-kod skapas C#-funktioner som bygger logiken. När sidan laddas in, körs funktionen LoadSearchHistory med hjälp av OnInitialized som hämtar all data ur localStorage. För varje gång man trycker på knappen Sök, kommer funktionen AddToHistory köras som gör att sökningen registreras i localStorage.

Men för att denna komponent skall kunna fungera med JavaScript behöver man även inkludera ”@inject IJSRuntime JSRuntime” och skapa en scriptfil med JavaScript funktioner.

**Kodexempel 14. JavaScript fil som lagrar och hämtar saker i localStorage. (Docs, 2024).**

```
window.localStorageFunctions = {
    setItem: function (key, value) {
        localStorage.setItem(key, value);
    },
    getItem: function (key) {
        return localStorage.getItem(key);
    }
};
```

I kodexempel 14 består koden av två funktioner, den första lägger till i localStorage medan den andra hämtar.

Arbetar man med Blazor kan man även implementera localStorage med ett antal NuGet-paket som man kan inkludera i sitt projekt. Vill man hålla sig till metoder  i ASP.NET Core kan man använda sig av Protected Browser Storage som skyddar ens data när man lagrar data i localStorage eller sessionStorage. Skillnaden mellan localStorage och sessionStorage är egentligen bara att datan sparas på olika sätt. Använder man localStorage sparas datan i webbläsarfönstret medan om man använder sessionStorage sparas den endast i den akutella webbläsarfliken. Det vill säga, laddas webbsidan om utan att webbläsaren stängs, kommer datan att vara kvar. Stängs hela webbläsarfönstret så försvinner all data.

## 4.4 Sortering

En viktig och användbar egenskap i en samling är sortering, som gör det möjligt att organisera och presentera data på ett strukturerat sätt. För att sortera data server-side i en databas kan man använda SQL-frågor med ORDER BY-klausulen. Detta gör det möjligt att organisera resultatet av en fråga baserat på värden i en eller flera kolumner i databastabellen.

Till exempel kan man använda ORDER BY för att sortera resultatet av en fråga med användares namn i stigande eller fallande ordning.

Men önskas möjligheten att sortera data client-side och regelbundet kunna sortera till exempel i en tabell så kan det göras på olika sätt beroende på ramverket och teknologin som används.

För att möjliggöra det kan man använda HTML och JavaScript. Man kan låta användare sortera tabellen efter olika kriterier genom att låta dem klicka på tabellrubrikerna. Man börjar med att bygga en HTML-struktur. Man skapar sedan en tabell och inkluderar rubriker för varje kolumn som önskas vara sorterbar. Sedan skapas JavaScript-logik för sorteringsfunktionen. När användaren klickar på en tabellrubrik ska en onClick-händelse fånga klicket och initiera en funktion som sorterar tabellen utifrån den kolumn som användaren har valt.

En annan teknik man kan använda är tabellkomponenten i UI-komponentbiblioteket MudBlazor, som förklaras mer ingående i Kapitel 6.3. Denna komponent har färdigt inbyggd sortering och är väldigt enkel att använda.

**Kodexempel 15. MudBlazor tabell med sortering. (MudBlazor, Table, 2020-2024).**

```
@using System.Net.Http.Json
@using MudBlazor.Examples.Data.Models

@inject HttpClient httpClient

<MudTable Items="@Elements" Hover="true" SortLabel="Sort By">
    <HeaderContent>
        <MudTh><MudTableSortLabel SortBy="new Func<Element,
object>(x=>x.Number)">Nr</MudTableSortLabel></MudTh>
        <MudTh><MudTableSortLabel Enabled="@enabled" SortBy="new
Func<Element, object>(x=>x.Sign)">Sign</MudTableSortLabel></MudTh>
        <MudTh><MudTableSortLabel
InitialDirection="SortDirection.Ascending" SortBy="new Func<Element,
object>(x=>x.Name)">Name</MudTableSortLabel></MudTh>
    </HeaderContent>
    <RowTemplate>
            <MudTd DataLabel="Nr">@context.Number</MudTd>
            <MudTd DataLabel="Sign">@context.Sign</MudTd>
            <MudTd DataLabel="Name">@context.Name</MudTd>
        </RowTemplate>
        <PagerContent>
            <MudTablePager PageSizeOptions="new int[] { 10, 25, 50, 100
}" />
        </PagerContent>
    </MudTable>

    <MudSwitch @bind-Checked="enabled" Color="Color.Info">Enable sorting
on the Sign Column</MudSwitch>

@code {
```

```
    private bool enabled = true;
    private IEnumerable<Element> Elements = new List<Element>();

    protected override async Task OnInitializedAsync()
    {
        Elements = await
httpClient.GetFromJsonAsync<List<Element>>("webapi/periodictable");
    }
}
```

| Nr | Sign | Name ↑ | Position ↑ | Mass |
|---|---|---|---|---|
| 13 | Al | Aluminium | 12 | 26.981539 |
| 51 | Sb | Antimony | 14 | 121.76 |
| 18 | Ar | Argon | 17 | 39.948 |
| 33 | As | Arsenic | 14 | 74.9216 |
| 85 | At | Astatine | 16 | 210 |
| 56 | Ba | Barium | 1 | 137.327 |
| 4 | Be | Beryllium | 1 | 9.012182 |
| 83 | Bi | Bismuth | 14 | 208.9804 |
| 107 | Bh | Bohrium | 6 | 272 |
| 5 | B | Boron | 12 | 10.811 |

Rows per page: 10 ▾ 1-10 of 88 |< < > >|

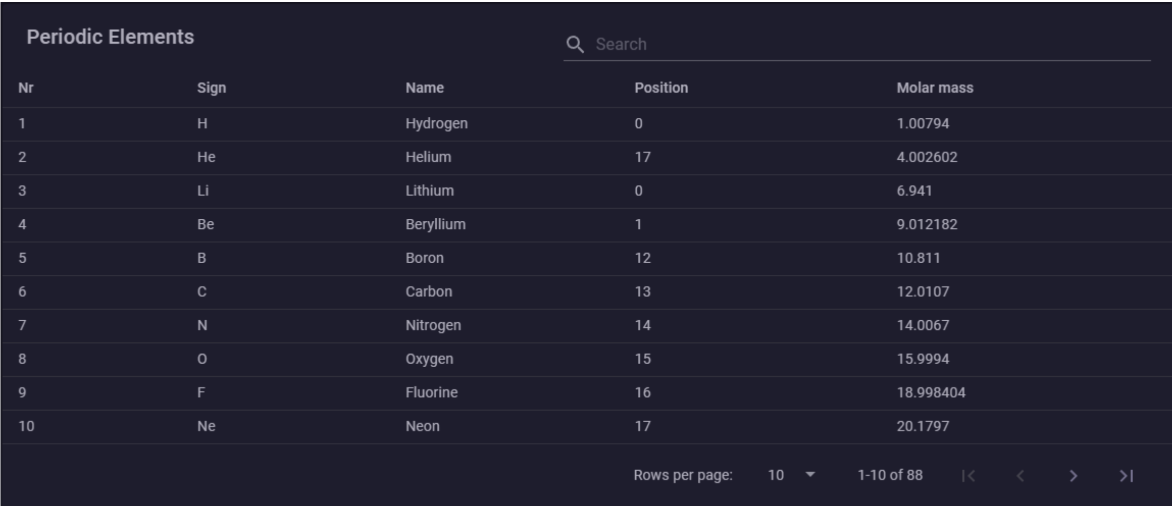
**Figur 5. En MudBlazor-tabell med möjlighet att sortera. (MudBlazor, Table, 2020-2024).**

I denna Blazor-komponent som syns i Figur 5 skapas en tabell med nummer, tecken och namn. Komponenten har inbyggda attribut som ger en möjlighet att välja hur tabellen ska sorteras, hur den ska se ut med mera. Man kan trycka på tabellrubrikerna för att sortera.

MudBlazor erbjuder även en version som använder Server Side filtrering, sortering och sidnumrering.

**Kodexempel 16. MudBlazor tabell med server side sortering. (MudBlazor, Server Side Filtering, Sorting and Pagination, 2020-2024).**

```
@using System.Net.Http.Json
@using MudBlazor.Examples.Data.Models
@inject HttpClient httpClient

<MudTable ServerData="@(new Func<TableState, Task<TableData<Element>>>(ServerRe-
load))" @ref="table">
    <ToolBarContent>
        <MudText Typo="Typo.h6">Periodic Elements</MudText>
        <MudSpacer />
        <MudTextField T="string" ValueChanged="@(s=>OnSearch(s))" Place-
holder="Search" Adornment="Adornment.Start"
                      AdornmentIcon="@Icons.Material.Filled.Search"
IconSize="Size.Medium" Class="mt-0"></MudTextField>
    </ToolBarContent>
    <HeaderContent>
        <MudTh><MudTableSortLabel SortLabel="nr_field" T="Element">Nr</MudTa-
bleSortLabel></MudTh>
        <MudTh><MudTableSortLabel SortLabel="sign_field" T="Element">Sign</MudTa-
bleSortLabel></MudTh>
    </HeaderContent>
    <RowTemplate>
        <MudTd DataLabel="Nr">@context.Number</MudTd>
            <MudTd DataLabel="Sign">@context.Sign</MudTd>
        </RowTemplate>
</MudTable>

    @code {
    private IEnumerable<Element> pagedData;
    private MudTable<Element> table;

    private int totalItems;
    private string searchString = null;

    private async Task<TableData<Element>> ServerReload(TableState state)
    {
        IEnumerable<Element> data = await httpClient.GetFromJsonAsync<List<Ele-
ment>>("webapi/periodictable");
        await Task.Delay(300);
        data = data.Where(element =>
        {
            if (element.Sign.Contains(searchString, StringComparison.Ordina-
lIgnoreCase))
                return true;
            if ($"{element.Number} {element.Position} {element.Molar}".Con-
tains(searchString))
                return true;
            return false;
        }).ToArray();
        totalItems = data.Count();
        switch (state.SortLabel)
        {
            case "nr_field":
                data = data.OrderByDirection(state.SortDirection, o => o.Number);
                break;
            case "sign_field":
                data = data.OrderByDirection(state.SortDirection, o => o.Sign);
                break;
        }

        pagedData = data.Skip(state.Page * state.PageSize).Take(state.Pa-
geSize).ToArray();
        return new TableData<Element>() { TotalItems = totalItems, Items = paged-
Data };
```



Periodic Elements

Search

| Nr | Sign | Name | Position | Molar mass |
|---|---|---|---|---|
| 1 | H | Hydrogen | 0 | 1.00794 |
| 2 | He | Helium | 17 | 4.002602 |
| 3 | Li | Lithium | 0 | 6.941 |
| 4 | Be | Beryllium | 1 | 9.012182 |
| 5 | B | Boron | 12 | 10.811 |
| 6 | C | Carbon | 13 | 12.0107 |
| 7 | N | Nitrogen | 14 | 14.0067 |
| 8 | O | Oxygen | 15 | 15.9994 |
| 9 | F | Fluorine | 16 | 18.998404 |
| 10 | Ne | Neon | 17 | 20.1797 |

Rows per page: 10 1-10 of 88

**Figur 6. En MudBlazor-tabell med Server-side filtrering, sortering och sidnumrering. (MudBlazor, Server Side Filtering, Sorting and Pagination, 2020-2024).**

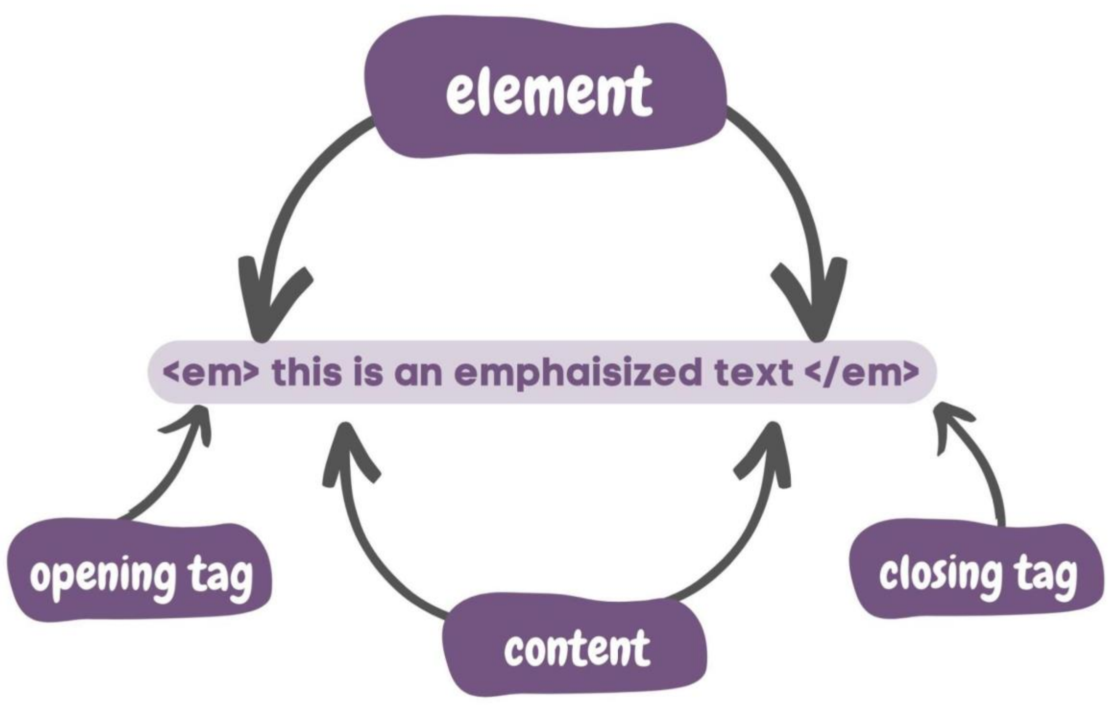
Varje gång en användare byter sida i tabellen (se Figur 6) eller ändrar sorteringen genom att klicka på sorteringssymbolerna i rubriken kommer tabellen att anropa asynkrona funktionen ServerReload (se Kodexempel 16) som i sin tur uppdaterar tabellen. I detta exempel visas också hur man tvingar tabellen att uppdateras när söktextfältet tappar fokus. Detta görs för att tabellen ska ladda om den filtrerade datan på servern. (MudBlazor, Server Side Filtering, Sorting and Pagination, 2020-2024).

# 5 Layout

Layouten är avgörande för användarens upplevelse och interaktion med sidan. Layouter som är lätta att förstå och lätta att navigera kan förbättra användarvänligheten och göra det enklare att hitta det man söker efter.

## 5.1 HTML och CSS

I HTML eller Hyper Text Markup Language, är elementen byggstenarna för ett HTML-användargränssnitt. Den innehåller vanligtvis en öppningstagg, en avslutande tagg och innehållet mellan dem. Det hjälper webbläsare att tolka i klassificeringen av innehållet, såsom rubriker, bilder, stycken och mer.




**Figur 7. HTML-struktur. (Joshi, 2023).**

Figur 7 beskriver hur ett HTML-element är uppbyggt. För att utveckla detta kan man placera elementen i varandra, med andra ord, nästla elementen i varandra (se Kodexempel 17). (Joshi, 2023).

**Kodexempel 17. Nästlade HTML element. (Joshi, 2023).**

```
<html>
    <head>
        <body>
            <div class="parent">
                <div class="child">
                    <p>
                        <span>
                            <b>This is a bold text trying to display nesting</b>
                        </span>
                    </p>
                </div>
            </div>
        </body>
    </head>
</html>
```

Svårare än så är det inte att designa en webbsida med HTML-element. Nästa steg är att ge elementen någons typ av stil. Det får man genom att använda CSS.

Precis som HTML är CSS inte ett programmeringsspråk. Det är inte heller ett märkspråk, utan ett stilmallsspråk. Man använder det för att ge ett element en stil. För att klargöra, vill man till exempel ge paragraf elementet <p> en röd färg kan man göra som i kodexempel 18.

**Kodexempel 18. CSS-exempel.**

```
p {
    color: red;
  }
```

**Figur 8. En webbsida gjord på HTML och CSS. (Patel, 2023).**

Figur 8 är designad endast med HTML och CSS och demonstrerar en funktionell och visuellt tilltalande webbsida utan behov av några andra programmeringsspråk och ramverk.

## 5.2 CSS-ramverk

I grunden består ett CSS-ramverk av flera CSS-stilmallar som är redo att användas av webbutvecklare och designers. Användaren har en färdig CSS-stilmall med ett CSS-ramverk, och för att skapa en webbsida behöver de bara koda HTML med rätt klasser, struktur och ID. Sidfot, bildspel, navigationsfält, hamburgermeny, kolumnbaserade layouter och många andra vanliga webbplatselement finns redan inbyggda i ramverken. Shreya Bose anser att de CSS-ramverk som toppar listan är följande: (Bose, 2023).

- Bootstrap
- Tailwind

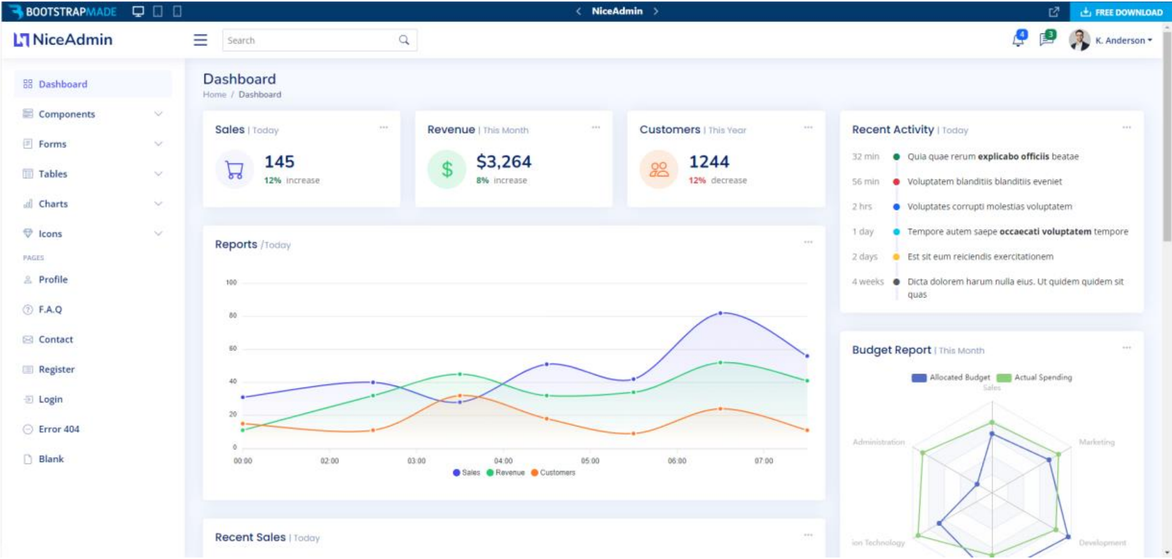
- Foundation

Bootstrap är ett populärt CSS-ramverk som är känt för sin responsiva design och innehåller gränssnittskomponenter med CSS- och JavaScript-baserade mallar. Bootstrap stöder SASS och LESS och erbjuder ett brett utbud av layouts, teman och UI-komponenter.

Tailwind CSS är ett "utility-first" CSS-ramverk som gör det enkelt att ändra teman och styling via konfigurationsfiler. Ramverket erbjuder ett brett utbud av färdiga klasser som gör det möjligt att snabbt skapa anpassade UI-designer utan att behöva skriva mycket CSS.

Foundation, även kallat ”världens mest avancerade responsiva front-end-ramverk” erbjuder en omfattande uppsättning av verktyg för att skapa responsiva webbapplikationer och ger utvecklare mer flexibilitet att anpassa sina användargränssnitt. Det innehåller också ett brett utbud av användargränssnittskomponenter och tillvalsfunktioner, inklusive formvalidering och responsiva bilder. (Bose, 2023).

För att visa ett exempel på ett användargränssnitt designat med ett CSS-ramverk demonstreras i figur 9 en webbsida där Bootstrap används.




**Figur 9. En webbsida grundad HTML och Bootstrap. (BootstrapMade, 2024).**

Från denna källa hittar man även färdiga gjorda templates som är gratis att använda för ens webbsida. (BootstrapMade, 2024).

## 5.3 UI-komponentbibliotek

Ett UI-komponentbibliotek är en samling av förbyggda komponenter som används för att skapa användargränssnitt för webbplatser och applikationer, med en enhetlig look. Dessa bibliotek inkluderar olika färdiga UI-element, såsom knappar, formulär, navigationsmenyer, ikoner och mer, var och en utformad med en modern look och känsla. UI-komponentbibliotek är särskilt användbara i miljöer för design och utveckling, då de hjälper till att säkerställa att den färdiga produkten bibehåller ett professionellt och finslipat utseende. Att använda ett komponentbibliotek har många fler fördelar. Dessa fördelar inkluderar hastighet, kompatibilitet, konsistens och tillgänglighet. (UXPin, 2023).

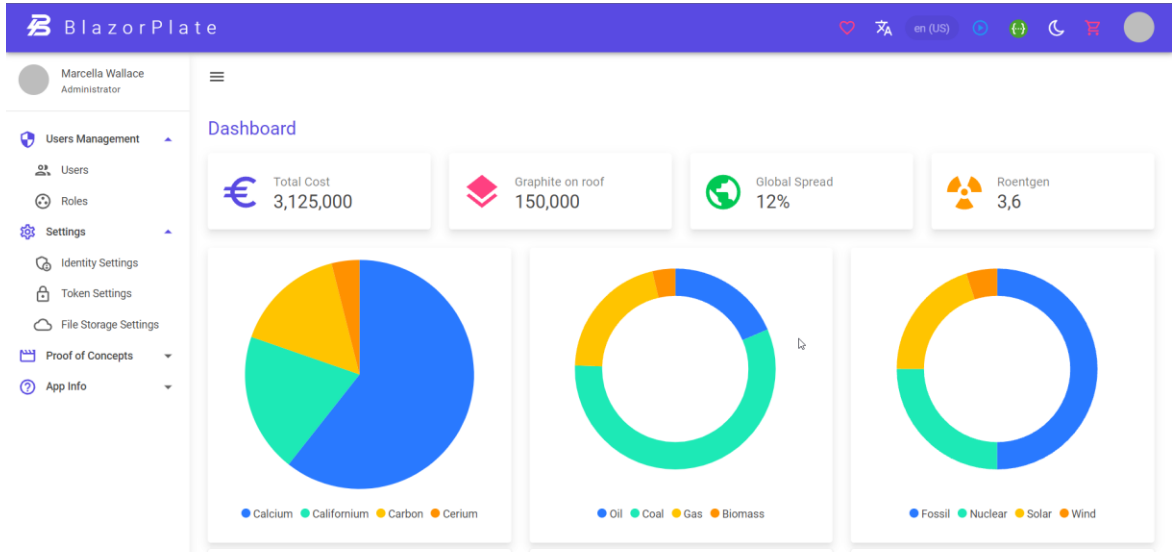
Bland de vanligare .NET UI-komponentbiblioteken hittas bland annat GrapeCitys ComponentOne som erbjuder ett brett utbud av komponenter. Det har många funktioner, som diagram, kalendrar, redigerare, menyer, navigation, rapportvisare och andra datahanteringskontroller. (Dorsey, 2015).

Likväl hittas DevExpress bland de mer populära aktörerna som med sitt paket ger en många alternativ av data- och gridvyer, diagram, scheman, textredigerare, navigationskontroller, rapporter, bilder, dokumentautomation och mycket mer. (Dorsey, 2015).

Även Infragistics är ett eftertraktat alternativ bland .NET utvecklare som med sina lättanvända komponenter tillför grids, hierarkiska grids, pivot grids, data visualizations, redigerare för text, siffror och validering, mobila optimerade kontrollers och mer. (Dorsey, 2015).

Som tidigare nämnt finns även UI-komponentbiblioteket MudBlazor som ett attraktivt alternativ för applikationer som bygger sig på .NET Blazor. Målet med MudBlazor är att göra Blazor-arbete mer njutbart och produktivt för utvecklare. För att uppnå målet har det skapats komponenter med ett modern utseende för väldigt många scenarion. Det inkluderar designelement som färgteman, typografi, layoutkomponenter, navigationsverktyg, knappar, formulärkomponenter, datavisningskomponenter och feedbackmekanismer. (MudBlazor, MudBlazor, 2020-2024).

Figur 10 visar en webbsida som använder UI-komponentbiblioteket MudBlazor och dess element.




**Figur 10. Webbsida designad med UI-komponentbiblioteket MudBlazor. (Garderoben, 2021)**

När man skapar en ny MudBlazor-applikation hittas en navigationspanel på vänster sida, ofta kallad NavBar. Den här panelen gör det enkelt för användaren att navigera mellan olika komponenter på webbsidan. Dessutom finns det en övre panel där användaren har möjlighet att interagera med olika element.

# 6 Utförande och resultat

Projektet genomfördes i flera steg. Det första steget var att välja ett lämpligt ramverk att bygga webbapplikationen med. Följande steg var att skapa en anslutning till databasen som använts i det tidigare CRM-systemet och sedan påbörja arbetet med att gradvis bygga layouten för webbsidan. Projektet integrerades också med Azure DevOps-molnet för kontinuerlig uppladdning och hantering av förändringar för att hålla jämna steg med projektets utveckling.

## 6.1 Databaskoppling och val av ramverk

Eftersom projektet skulle genomföras i en Microsoft-miljö stod valen mellan MVC, Razor Pages och Blazor. Efter att ha funderat vilket ramverk som skulle vara mest lämpligt, valdes ASP.NET Core Blazor. Anledningen till valet var att jag ville ta mig an en utmaning och lära mig något nytt. En ytterligare orsak till att Blazor valdes, var att det fanns möjligheten att skapa en webbapplikation som endast behövde köras server-side, i jämförelse med Razor Pages samt MVC.

Nästa steg var att se till att en anslutning mellan webbapplikationen och databasen från gamla CRM-systemet skapades. För att göra detta behövde man först lägga till en anslutningssträng i en konfigurationsfil för att indikera var databasen finns och hur man ska ansluta till den. Sedan behövde man hantera anslutningssträngen i själva applikationskoden, vilket gjordes genom att skapa en specifik klass för att utföra databasoperationer. Slutligen lades kod till i Program.cs för att konfigurera anslutningen till databasen genom HFSDbContext-klassen. Kodexempel 19 visar hur detta gjordes.

```
    builder.Services.AddDbContext<HFSDbContext>(options => {
options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection
"));});
```

**Kodexempel 19. Denna kod konfigurerar anslutningen till en SQL Server-databas för ApplicationDbContext.**

I detta skede identifierades vilka tabeller som var relaterade till varandra i databasen genom unika foreign keys. Varje CRM-case hade en unik id i form av Guids. Sen skapades dataobjekt med Properties (egenskaper) som motsvarar de fält i databasen som webbapplikationen behövde ha tillgång till.

## 6.2 Byggandet av webbsidan

Efter att relevanta kopplingar hittats började byggandet av webbsidans layout och funktionalitet. Till att börja med planerade man att göra en sökfunktionalitet som listar alla CRM-cases beroende på ett sökkriterium. För att utföra det, skapades med HTML en sökfunktion och en lista för case som uppfyller sökvillkoret (se figur 12).

Case Search

Case ID or Case Title: CAS Search

Search Results:

Case ID: CAS-27063-J5T4S1

Case Title: Test case in new CRM

See more...

Case ID: CAS-27066-F1P5R0

Case Title: Testcase

See more...

**Figur 12. Version 1 av webbsidans sökfunktionalitet.**

Följande sak som planerats var att kunna se mer detaljerad information om ett case. Det genomfördes med att söka fram fler casedetaljer ur databasen och tillsätta Properties till case-dataobjektet. Nu behövdes bara ett sätt att visa case-informationen på webbsidan, så därför skapades en ny Blazor-komponent som skulle sammanfatta och beskriva relevanta delar för ett case.

Case Details

Assignment Information

Case Number:
CAS-22001-R9H8W2

Created on:
01.3.2013 kl. 11:53

HFS Contact:

Title:
Fully paid ticket becomes after embarkation partly paid

Customer:
Wagenborg

Product Information

Case Description:

Internal comments:

Public comments:

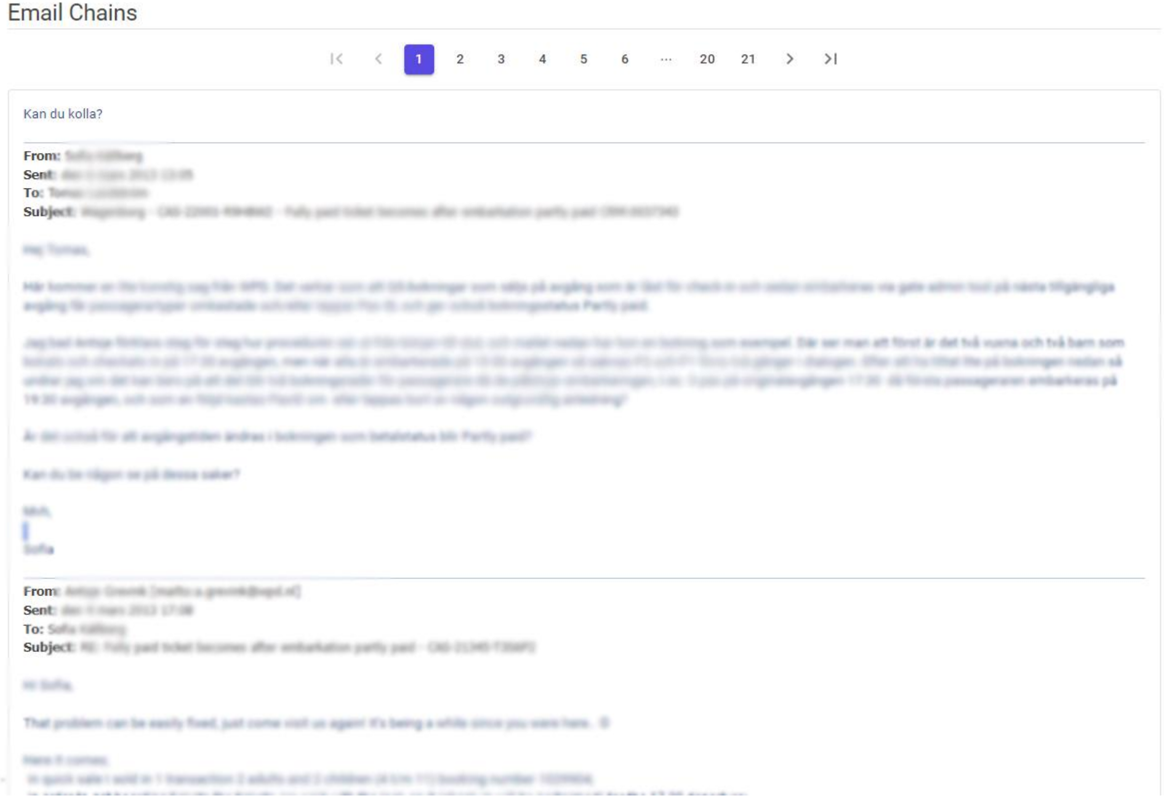
**Figur 13. Version 1 av komponenten som visar detaljerad information av ett case.**

Fält för fält summerades till Blazor-komponenten och man började se den case-information som önskades, men i detta skede noterade jag att webbsidan kändes klumpig och lite vilseledande. Därför började jag forska i metoder som kunde förbättra webbsidans layout och göra den mer användarvänlig. Det var i detta skede min kollega kom in och tipsade mig om att använda MudBlazor när de hört att jag skulle skapa webbsidan med ramverket Blazor.

Så efter en del bekantande och testning av UI-komponentbiblioteket började webbsidan redan se bättre ut. I samband med implementeringen av MudBlazor uppfylldes även kravet att kunna se e-postkedjor länkade till det case man undersöker.

E-postkedjorna lagrades i databasen som HTML i textformat, så jag var tvungen att hitta ett sätt som översatte HTML-texten till HTML-element. Det löstes genom att använda HTMLAgilityPack, ett NuGet-paket som används för att analysera och manipulera HTML-dokument. Deras HTML-Parser-metod laddar HTML-dokumenten på webbsidan från den angivna HTML-texten. (HtmlAgilityPack, 2024).

Detta resulterade i att vi kunde se e-postkedjor i case-detaljkomponenten (se Figur 14).

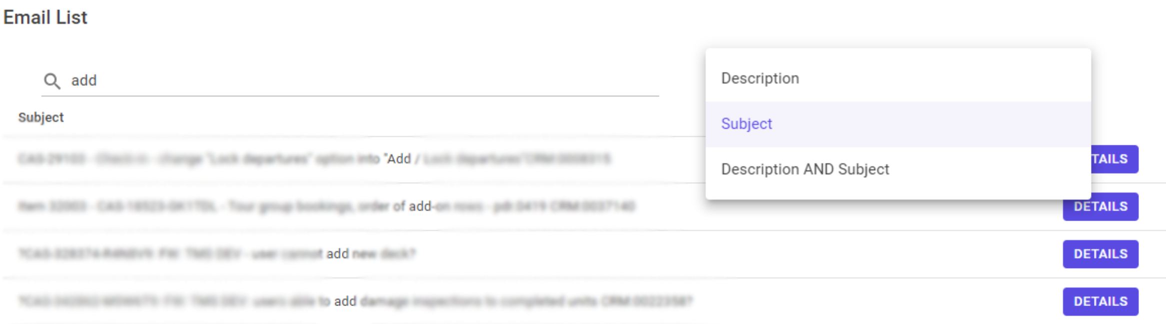



**Figur 14. E-postkedjor länkade till ett case visas under case informationen.**

Jag insåg att det också borde finnas en möjlighet att söka på ord som finns i e-postkedjorna. Därför skapade jag en sökfunktionalitet i samma struktur som sökandet av ett case. Men i detta skede stötte jag på problem. Tiden det tog för att hitta ett case som innehöll ordet jag sökte på blev så lång att det orsakade timeout error. Timeout error uppstår på grund av att applikationen ställer in ett timeout-värde och om timeout-värdet uppnås avbryter applikationen frågan. Även försök vid att ställa SQL-frågan direkt mot databasen orsakade lång väntetid. (Microsoft, Troubleshoot query time-out errors, 2023).

Problemet var att SQL-frågan använde sig av LIKE-operatorn i samband med ett wildcard ”%” som prestandamässigt tvingar databasen att göra en fullständig sökning genom hela kolumnen, i detta fall hamnar den alltså söka genom alla e-postkedjor länkat till ett case vilket kan innehålla extremt mycket text. Så för att lösa problemet forskades metoder att kringgå problemet.

Jag fick överväga att använda fulltextindexering som är optimerat för textbaserad sökning. Så i stället för att använda LIKE-operatorn i SQL-frågorna byttes de ut till CONTAINS and FREETEXT.




**Figur 15. Här kan användare söka efter ett case genom att använda ord från e-postkonversationer.**

I figur 15 visas hur det är möjligt att söka efter case med hjälp av ord från e-postkedjornas ämne, beskrivning eller båda två.

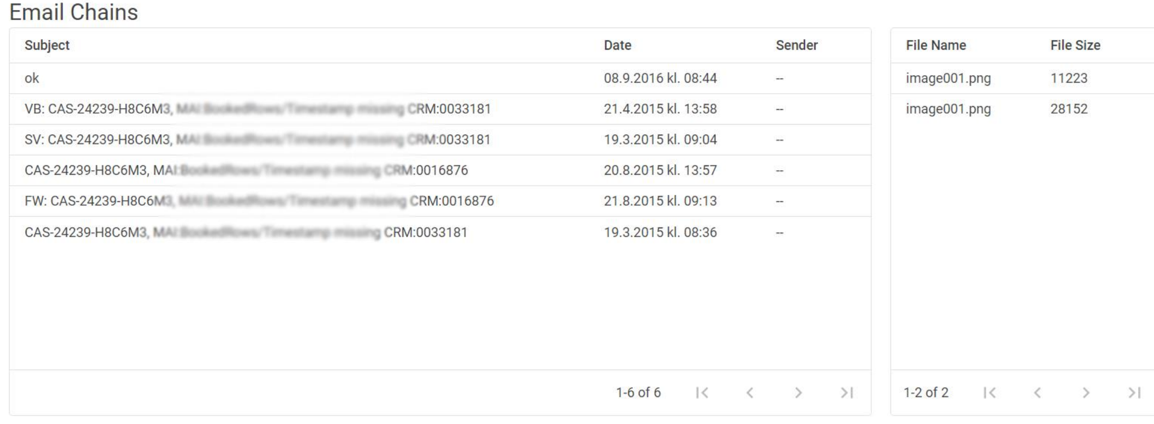
Inledningsvis valdes Entity Framework Core för att hantera datan hämtat från databasen, men på grund av min bättre erfarenhet med ADO.NET och observation att EF Core orsakade dålig prestanda, bestämdes det i stället för att använda ADO.NET. Denna förändring resulterade i betydande förbättrad prestanda och bättre anpassning till webbplatsens krav. I SQL-frågorna användes SQL Parameters för att göra det säkrare emot SQL-injection attacker.

## 6.3 Bifogade filer

Efter första presentationen av webbsidan för användargruppen konstaterades det att man också borde ha möjligheten att se bifogade filer länkade till varje e-postkedja, såsom bilder och PDF-filer. Detta gjordes genom att ta reda på vilka databastabeller som innehöll de bifogade filerna. Det tog en tid för mig att förstå hur jag kan få en bild eller fil att visas med

den information som fanns i databasen. Det visade sig vara att filerna sparades som byte strings, en lång kryptisk text med olika tecken och siffror. Men efter en del forskning och testning hade jag kommit fram till en lösning.

Under case-detaljerna gjordes två tabeller till både e-postkedjor och bifogade filer, från dessa tabeller skulle man ha möjligheten att klicka på den tabellrad för att öppna antingen en e-postkedja eller en fil (se Figur 16).




**Figur 16. Varsin tabell för e-post och bifogade filer.**

### 6.3.1 Bilder

Planen med bilderna var att det skulle finnas en egen Blazor-komponent som visar bilden på hela sidan så man lättare kunde se vad som där stod. Det gjordes på följande sätt (se Kodexempel 20).

**Kodexempel 20. Denna kod resulterar i en bild på webbsidan utifrån en bytestring**

```
<img src="data:image/jpeg;base64,@attachmentItem.Body" alt="@attachmentItem.FileName" class="center-screen" style="width: auto; height: auto;
```

@attachmentItem.Body ersätts med byte-stringen från den bild man valt och på så sätt dyker upp på webbsidan.

I e-postkedjornas HTML-text kunde man även hitta unika ID:n, för de bilder som klistrats in i e-postmeddelandena. Så därför kodades en kontroll-funktion som granskade HTML-texten och jämförde dem med de bifogade filernas ID:n, och ifall det matchade kunde man få bilderna att visas i e-postmeddelandena.

### 6.3.2 Dokumentfiler

Andra filer som kändes relevanta att kunna visas på webbsidan var dokument-filer. För att visa dessa filer på en webbsida kunde man göra exakt som med bilder bara att man ändrade på src-delen och bytte ut img-elementet till embed- eller pre-element beroende på filtypen. (se Kodexempel 21).

**Kodexempel 21. En if-sats som kontrollerar slutet av en fils namn och returnerar rätt element beroende på filtypen.**

```
    if (attachmentItem.FileName.EndsWith(".pdf",
StringComparison.OrdinalIgnoreCase))
    {
                <embed src="data:application/pdf;base64,@attachmentItem.Body"
alt="@attachmentItem.FileName" style="min-width:100%; min-height:1000px;" />
    }
    else if (attachmentItem.FileName.EndsWith(".docx",
StringComparison.OrdinalIgnoreCase))
    {
                <pre style="padding:5px; color:black; font-family:Lato Regular;
max-width:100%; min-height:1000px;">@documentContent</pre>
    }
    else if (attachmentItem.FileName.EndsWith(".xml",
StringComparison.OrdinalIgnoreCase))
    {
                <pre style="padding:5px; color:black; font-family:Lato Regular;
height:auto;">@documentContent</pre>
    }
    else if (attachmentItem.FileName.EndsWith(".txt",
StringComparison.OrdinalIgnoreCase))
    {
                <embed src="data:text/txt;base64,@attachmentItem.Body"
alt="@attachmentItem.FileName" style="min-width:100%; min-height:1000px;" />
    }
```

Andra filer som bifogats i e-postkedjorna behövdes även tas i beaktande. Så därför skapades en ”Download file”-knapp som gav en möjligheten att ladda ner filerna och öppna upp dem på datorn. Detta gjordes med C# och JavaScript på följande vis (se Kodexempel 22).

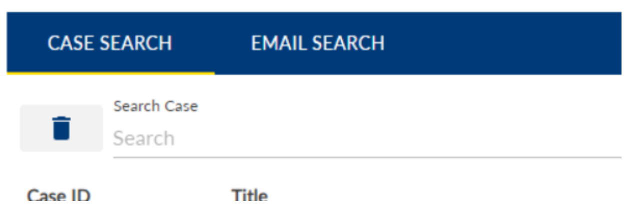
**Kodexempel 22. Koden beskriver hur man kan implementera en ”Ladda ner”-knapp**

```
C#-delen

private async Task DownloadFile()
{
    string base64String =
Convert.ToBase64String(Convert.FromBase64String(attachmentItem.Body));

    await JS.InvokeVoidAsync("downloadFile", base64String,
attachmentItem.FileName);
}



Javascript-delen

window.downloadFile = (base64String, fileName) => {
    const link = document.createElement('a');
    link.href = `data:application/octet-stream;base64,${base64String}`;
    link.download = fileName;
    link.style.display = 'none';
    document.body.appendChild(link);
    link.click();
    document.body.removeChild(link);
}
```

## 6.4 Optimering av sökfunktionaliteten

När jag hade kommit så här långt hade jag ett möte med användargruppen där de fick ge önskemål och feedback om webbsidans nuvarande läge. Här kom det fram att det gärna fick finnas sorteringsmöjligheter i tabellerna som söker case och e-postkedjor, en historik på de ord man sökt med förut och även filtreringsalternativ för att minska på resultaten som sökningarna gav. Jag fick även kritik om att huvudsidan borde vara söktabellen man söker cases i och att jag borde slå ihop case- och e-posttabellerna till en sida i stället för två olika.

Så jag började med att sammanslå ihop Blazor-komponenterna som sökte case och e-postkedjor och skapade två olika flikar som man kan trycka på för att visa den tabell du vill söka i, alltså case eller e-post (se Figur 17).




**Figur 17. Två flikar för Casesöking och E-postsökning.**

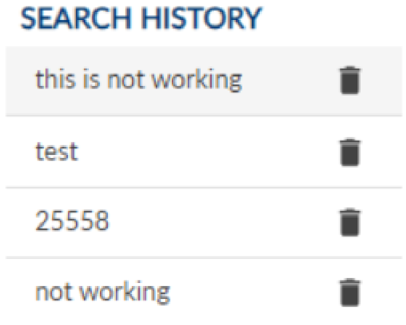
Sorteringsmöjligheterna var lätta att lösa eftersom UI-komponentbiblioteket MudBlazor hade en färdig lösning. Till en början användes den enklare versionen av sortering i en tabell. Men problemet med den var att den inte uppdaterades i realtid, så när man bytte sida i tabellen var de inte sorterade på den andra sidan. Därför fick jag forska i deras mer avancerade version som använder server-side sortering. Denna tabell kallar på en asynkron funktion när användaren navigerar mellan sidorna och klickar på rubrikerna för att sortera. (MudBlazor, Server Side Filtering, Sorting and Pagination, 2020-2024).

I figur 18 kan man trycka på rubriken Created och välja vilken riktning det ska sorteras.

| Created ↑ |
| --- |
| 14.01.2011 kl 10:24 |
| 14.01.2011 kl 10:24 |
| 14.01.2011 kl 10:24 |
| 14.01.2011 kl 10:24 |

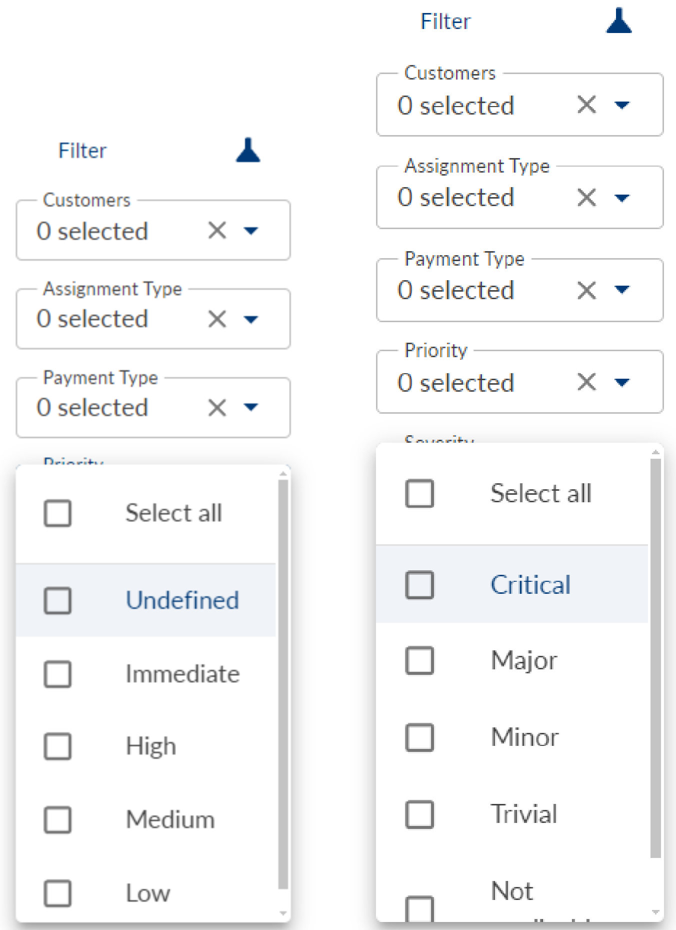
**Figur 18. Tabellsortering.**

Sökhistoriken skapades genom att en lista placerades på höger sida om söktabellen. För varje utförd sökning lades söktermen med i listan och sparades sedan i localStorage. Dessutom lades en ”Ta bort”-knapp vid sidan om varje utförd sökterm som tar bort den ur listan och minnet (se Figur 19).




**Figur 19. Listan med sökhistorik.**

Filtrering gjordes genom att manipulera SQL-frågorna. WHERE-satser lades till och för varje filter skapades en unik C#-funktion som anropades när användaren sökt efter case med de markerade filteralternativen som användes. Återigen användes UI-komponentbiblioteket MudBlazors egna komponent Multiselect. De valda alternativen returneras som sammanlänkade strängar avgränsade med kommatecken (se Figur 20).




**Figur 20. Filtrering av sökresultat.**

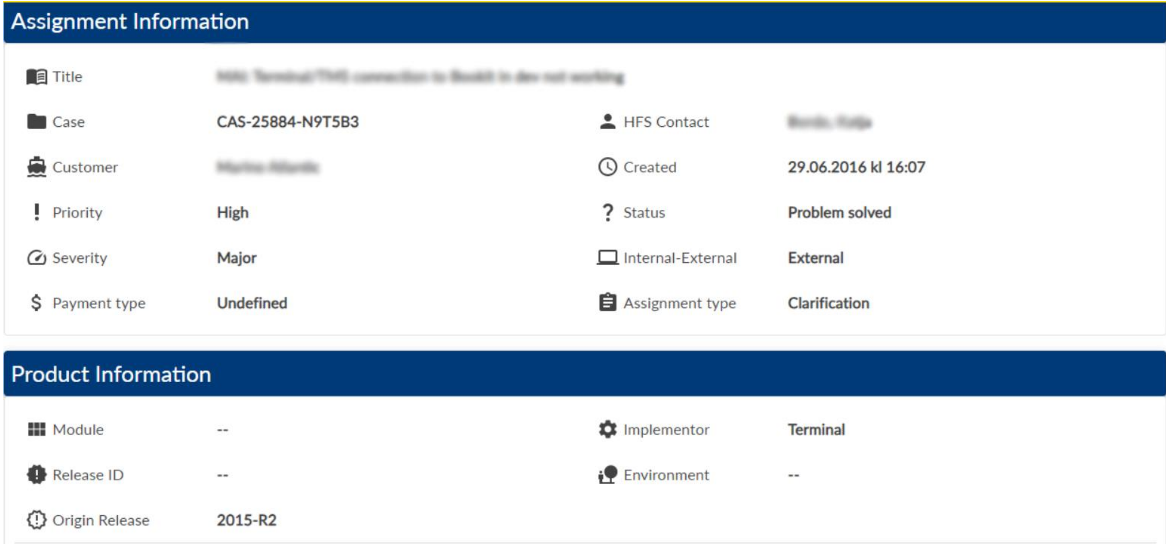
## 6.5 Finjusteringar och detaljer

När webbsidan börjat fungera enligt kraven, fanns det ännu saker som kunde förbättras och optimeras.

Till att börja med gav jag webbsidan färg och kontrast. Färgerna valdes enligt företagets färgkoder och deras logo placerades i vänstra hörnet av sidan som man kunde trycka på för att återvända till startsidan (se Figur 21).

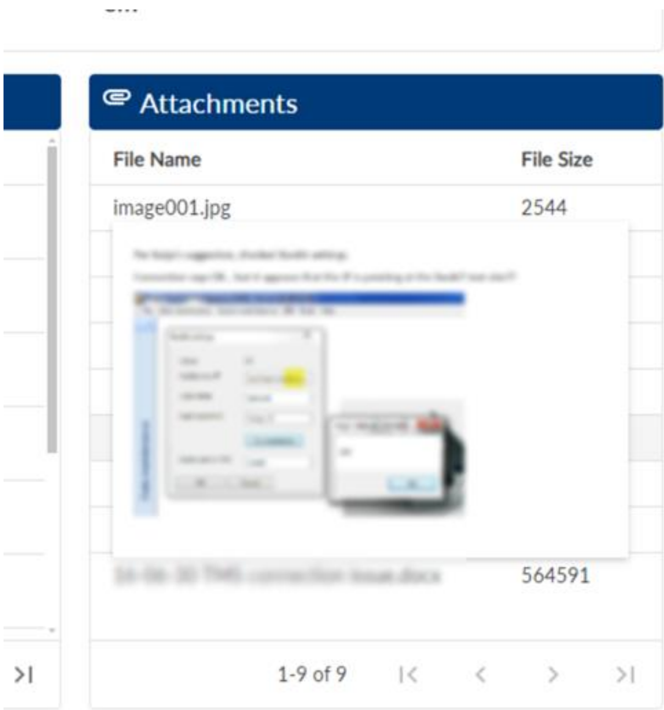
Figur 21: Gult, blått och vitt symboliserar företaget Hogia.

Efter detta placerades även ikoner före varje case-detalj för att ge en mer beskrivande förklarning om vad fälten betyder. Ikonerna kunde hämtas direkt ur MudBlazor UI-komponentbiblioteket (se Figur 22).




**Figur 22. Ikoner för varje case-detalj.**

En annan förbättring som gjordes var att med JavaScript bygga en funktionalitet där man kunde förhandsvisa bilderna ur tabellen med de bifogade filerna. Detta för att inte behöva trycka på varje bild och öppna upp dem i en ny flik när man söker en specifik bild (se Figur 23).




**Figur 23. En förhandsvisning av bilden.**

Efter ytterligare feedback av användargruppen kom det fram att det vore nödvändigt att ha en bildkarusell nere på webbsidan som visade en liten thumbnail av en bild, om man då tryckte på thumbnailen navigerade man till den bilden i stället. Detta implementerades med JavaScript. Ett litet problem jag stötte på då var att vissa case kunde ha väldigt många bilder, så många att karusellen blev för lång och inte rymdes på sidan. Detta löstes med att göra om det till en horisontell rullningslista, alltså en lista med scrollbar. Nästa problem som jag då stötte på var att scrollbaren inte lämnade på det ställe man senast hade den på när man navigerade mellan bilderna. Problemet åtgärdades med att implementera localStorage, då lagrades scrollbarens-position i minnet och återställdes inte längre (se Figur 24).

**Figur 24. En bildkarusell av alla bilder från ett case.**

# 7 Diskussion

När jag började på företaget Hogia Ferry Systems visste jag inte alls vad jag gav mig in på, jag hade fått sommarjobb med uppgiften att skapa en webbsida. Jag hade aldrig heller jobbat på ett It-företag, så jag visste inte vad som väntade. När jag väl fick börja på och fundera på vilka metoder och strategier jag skulle använda för att skapa webbsidan bestämde jag att skapa den med ett ramverk jag inte använt förut. Detta för att ge mig en utmaning och ta lärdom av det. Efter att jag fått bekanta mig med databasen och undersökt tabellerna i den var jag helt chockad. Jag hade aldrig förr jobbat med så mycket data. Dessutom sa chefen att det finns tabeller som inte används eller är irrelevanta för uppgiften. Så att hitta kopplingarna mellan tabellerna och ta reda på vilka kolumner som var de jag behövde var inte alltid det lättaste att finna, men med tiden och hjälp av kollegerna fann jag det jag behövde.

Det blev en intressant utvecklingsperiod med en del hinder och utmaningar på vägen. Men det stoppade mig inte, jag fann alltid en väg till att lösa det jag ville åstadkomma. Det svåraste med utvecklingen tror jag nog var att hitta alla databaskopplingar men även lösningen till hur jag skulle få bifogade filerna att synas och bli nerladdningsbara.

Feedbacken jag fått av användargruppen efter att webbsidan varit klar har varit bra, till och med så bra att de tyckte den var bättre och enklare att använda än Microsofts Dynamics CRM-system som används för nya case.

# 8 Litteraturförteckning

Abuhakmeh, K. 5.6.2020. *Basics of Razor Pages.* Hämtat från Jetbrains: https://www.jetbrains.com/guide/dotnet/tutorials/basics/razor-pages/

BootstrapMade. 29.1.2024. *BootstrapMade*. Hämtat från Nice Admin - Free bootstrap admin HTML template: https://bootstrapmade.com/nice-admin-bootstrap-admin-html-template/

Bose, S. (7.5.2023). *BrowserStack.* Hämtat från Top 5 CSS Frameworks for Developers and Designers: https://www.browserstack.com/guide/top-css-frameworks

Chidera, E. I. (21.12.2022). *Filtering in C# – How to Filter a List with Code Examples.* Hämtat från freeCodeCamp: https://www.freecodecamp.org/news/filtering-in-csharp-how-to-filter-a-list-with-code-examples/

Docs, M. W. (2024). *Window: localStorage property.* Hämtat från MDN Web Docs: https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage

Dorsey, T. (9.8.2015). *Visual Studio Magazine.* Hämtat från 9 Top .NET UI Component Collections: https://visualstudiomagazine.com/articles/2015/09/01/9-top-net-ui-component-collections.aspx

Garderoben. (26.9.2021). *GitHub.* Hämtat från Who's using MudBlazor?: https://github.com/MudBlazor/MudBlazor/issues/2849

HFS. (2024). *Hogia Ferry Systems.* Hämtat från Overview of HFS: https://www.hogia.se/int/ferry-system

HtmlAgilityPack. (2024). *Load Html From String.* Hämtat från Html Agility Pack: https://html-agility-pack.net/from-string

Joshi, M. (10.3.2023). *BrowserStack.* Hämtat från How to build a website using HTML and CSS: https://www.browserstack.com/guide/build-a-website-using-html-css

Microsoft. (15.9.2021a). *ADO.NET Overview.* Hämtat från Microsoft: https://learn.microsoft.com/en-us/dotnet/framework/data/adonet/ado-net-overview

Microsoft. (25.5.2021). *Entity Framework Core.* Hämtat från Microsoft: https://learn.microsoft.com/en-us/ef/core/

Microsoft. (15.9.2021b). *Entity Framework overview*. Hämtat från Microsoft: https://learn.microsoft.com/en-us/dotnet/framework/data/adonet/ef/overview

Microsoft. (14.11.2023). *ASP.NET Core Blazor*. Hämtat från Overview: https://learn.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-8.0&WT.mc_id=dotnet-35129-website

Microsoft. (3.1.2023a). *Full-Text Search.* Hämtat från Microsoft: https://learn.microsoft.com/en-us/sql/relational-databases/search/full-text-search?view=sql-server-ver16

Microsoft. (10.5.2023). *Overview of ASP.NET Core*. Hämtat från Microsoft: https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-8.0

Microsoft. (26.9.2023). *Overview of ASP.NET Core MVC.* Hämtat från Microsoft Documentation: https://learn.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-8.0

Microsoft. (3.1.2023b). *Query with Full-Text Search.* Hämtat från Microsoft: https://learn.microsoft.com/en-us/sql/relational-databases/search/query-with-full-text-search?view=sql-server-ver16

Microsoft. (7.3.2023). *Troubleshoot query time-out errors.* Hämtat från Microsoft: https://learn.microsoft.com/en-us/troubleshoot/sql/database-engine/performance/troubleshoot-query-timeouts

MudBlazor. (2020-2024a). *MudBlazor.* Hämtat från What is MudBlazor?: https://mudblazor.com/mud/introduction

MudBlazor. (2020-2024b). *Server Side Filtering, Sorting and Pagination.* Hämtat från MudBlazor: https://mudblazor.com/components/table#server-side-filtering,-sorting-and-pagination

MudBlazor. (2020-2024c). *Table.* Hämtat från MudBlazor: https://mudblazor.com/components/table#sorting

Patel, R. (25.2.2023). *Medium.* Hämtat från How to create a Website Using HTML and CSS Only: https://rutikkpatel.medium.com/how-to-create-a-website-using-html-and-css-only-18fc6a3f68ab

Tuğanlı, C. (27.4.2023). *Medium.* Hämtat från ADO.NET Architecture: CONNECTED VS DISCONNECTED: https://medium.com/@cemtuganli/ado-net-architecture-connected-vs-disconnected-791f17279fc5

UXPin. (24.11.2023). *UXPin.* Hämtat från What is a Component Library, and Why Should You Use One for UI Development?: https://www.uxpin.com/studio/blog/ui-component-library/

Winand, M. (2024). *Indexing Like Filters.* Hämtat från Use the index, Luke: https://use-the-index-luke.com/sql/where-clause/searching-for-ranges/like-performance-tuning