Projektuppgift

ASP.NET med C#

Projektuppgift
Webbshop

Albin Rönnkvist

Campus Härnösand Universitetsbacken 1, SE-871 88. **Campus Sundsvall** Holmgatan 10, SE-851 70 Sundsvall.
Campus Östersund Kunskapens väg 8, SE-831 25 Östersund.
Phone: +46 (0)771 97 50 00, Fax: +46 (0)771 97 50 01.

MITTUNIVERSITETET
Avdelningen för informationssystem och -teknologi

Författare: Albin Rönnkvist, alrn1700@student.miun.se
Utbildningsprogram: Webbutveckling, 120 hp
Huvudområde: Datateknik
Termin, år: VT, 2019

Sammanfattning

I detta projekt skapade jag en webbshop för ett fiktivt företag som säljer hemelektronik. Webbapplikationen skapades med MVC-ramverket ASP.NET och jobbar mot en SQL Server-databas med hjälp av ett databaslager(DAL).

Innehållsförteckning

Sammanfattning......iii

1 Introduktion......1
1.1 Bakgrund och problemmotivering......1
1.2 Detaljerad problemformulering......1
1.2.1 Databas......1
1.2.2 Databaslager......1
1.2.3 Webbapplikation......2

2 Teori......3
2.1 Databas......3
2.2 SQL......3
2.3 ER-modellen......3
2.4 MVC......4
2.5 ASP.NET......5

3 Metod......6

4 Konstruktion......7
4.1 Databas......7
4.2 Databaslager (DAL)......13
4.3 Webbapplikation......18
4.3.1 Konfigurering......18
4.3.2 MVC......22

5 Resultat......36

6 Slutsatser......37

Källförteckning......38

1 Introduktion

1.1 Bakgrund och problemmotivering

Allt fler väljer att handla på nätet och det är en typ av konsumtion som definitivt kommer fortsätta växa i många år framöver i takt med att den yngre generationen som växt upp med digitalisering blir mer köpstarka. Under det första halvåret 2018 ökade e-handelskonsumtionen med 11 procent och 6 av 10 nordbor, alltså 12 miljoner konsumenter handlar på nätet varje månad. De som handlar mest är konsumenter i åldersspannet 30-49 år och en av de mest populära varukategorierna är hemelektronik. [1] [2]

Eftersom e-handelskonsumtionen ökar så tänker jag att det kan vara nyttigt att lära sig skapa en webbshop. Jag riktar även in mig på hemelektronik eftersom det är en av de mest populära varukategorierna.

Webbapplikationen kommer bestå av två delar:

1. **Kund**: En del där kunder kan navigera genom produkter, logga in och lägga till / ta bort produkter i en varukorg.

2. **Anställd**: En del där anställda kan logga in och skapa, uppdatera och radera produkter / kategorier. Endast användare av användartypen administratör kommer kunna registrera nya konton.

1.2 Detaljerad problemformulering

1.2.1 Databas

Jag kommer börja med att skapa ER-diagram för att beskriva min databas. Efter det kommer jag behöva översätta ER-diagrammen till tabell-diagram för att sedan kunna implementera databasen i Microsoft SQL Server.

1.2.2 Databaslager

Databaslagret konstrueras med modeller där jag skapar detaljer som beskriver databasobjekt och metoder som hanterar objekten och modifierar data i databasen eller hämtar data.

Detaljer

Jag kommer skapa detaljer för produkter, beställningar och användare.
För produkter behöver jag skapa en detalj som beskriver produkterna, en för kategorier samt en för kategorier och produkter sammansatt.
Jag kommer även använda dataannotering och validering.

Metoder

Jag behöver skapa olika metoder för alla olika sätt som jag vill kommunicera med databasen, dvs. alla olika SQL-satser:
Produkter:
- Skapa produkt
- Hämta produkter med tillhörande kategori
- Hämta produkter med tillhörande kategori med filtrering efter kategori.
- Hämta produkter med tillhörande kategori med filtrering efter söksträng.
- Hämta enskild produkt
- Uppdatera produkt
- Radera produkt

Kategorier:
- Skapa kategori
- Hämta kategorier
- Hämta enskild kategori
- Uppdatera kategori
- Radera kategori

1.2.3 Webbapplikation

Jag behöver skapa en webbapplikation i ASP.NET med MVC-struktur. Jag behöver alltså dela upp projektet i modeller, vyer och controllers.

Modeller
Modell-delen kommer bestå av databaslagret som jag gick igenom ovan. Det är så kallade ”detaljer”, dvs. klasser som beskriver databasobjekt med egenskaper. Det är även metoder som använder dessa detaljer tillsammans med SQL-satser för att kommunicera med databasen.

Vyer
Vy-delen kommer bestå av vyer, dvs. det som användaren ser när de besöker. I detta fall kommer det vara olika sidor som besökaren kan navigera sig igenom.

Controllers
I controller-delen slås vyerna ihop med modellerna och agerar som ett mellanting. Data från t.ex. formulär i vyerna kommer skickas till controllern som sedan kan anropa en metod från databaslagret som lagrar den inmatade datan i databasen. Åt andra hållet kommer controllern anropa metoder från databaslagret som läser in data i objekt som skickas vidare till vyn där de kan läsas ut på skärmen.

2 Teori

2.1 Databas

En databas är i grunden en organiserad samling av data(information). Det används för att så effektivt som möjligt spara. bevara och hämta data. Den vanligaste formen av en databas är relationsdatabasen som länge varit i fronten i professionella sammanhang. Denna typ av databas använder sig av relationsmodellen som är en datamodell där data lagras i relationer, även kallat tabeller. Tabellerna ska representera en typ av sak, t.ex. en person eller en bil. I tabellerna finns det kolumner som ska representera sakens attribut eller egenskaper, t.ex. bilfärg eller bilmärke. Den data som matas in i tabellen lagras i rader, även kallat tupler, t.ex. bilfärgen är röd eller bilmärket är Volvo. Sedan kan man koppla ihop dessa tabeller med varandra på olika sätt och hantera data på ett effektivt och strukturerat vis [3]. Ett exempel på en relationsbaserad databashanterare är Microsoft SQL Server[4] som också kommer användas i projektet.

2.2 SQL

För att hantera data i en relationsdatabas så använder man oftast frågespråket SQL. Det är ett standardiserat språk för att lagra, manipulera och hämta data i en databas. Några vanliga SQL-kommandon som ”Select”, ”Insert”, ”Update”, ”Create”, ”Delete” och ”Drop” kan användas för att åstadkomma i princip allt som man behöver göra i en databas [9].

2.3 ER-modellen

Ett *ER-schema* / *ER-diagram* är ett grafiskt gränssnitt som delar upp relationsdatabasen i mindre delar och beskriver den i detalj med tabeller och länkningar.

Namn	Form	Beskrivning
Entitetstyper		Typer av saker, t.ex. personer eller bilar.
Sambandstyper		1:1, N:1, N:M. Beskriver relationen mellan två entitetstyper.
Attribut		Egenskaper, t.ex. kan en person ha hårfärg, namn osv.

Att tänka på när man utformar ett ER-diagram:

- ***Undvik duplicering av information:*** På så sätt blir det effektivare lagring och snabbare sökningar. Dela hellre upp återkommande information i en egen tabell och gör en relation till den.

- ***Grundregler:*** Varje tabell ska beskriva endast en typ av sak. Varje rad i tabellen ska innehålla data om en enda sådan sak. Data vi lagrat för varje sak ska finnas på en enda rad.

- ***PK, FK:*** PK är primärnyckel med ett unikt värde som används för att komma åt en unik rad / tupel. FK är främmande nyckel och används som referens till en primärnyckel i en annan tabell. Med dessa nycklar länkar man ihop tabellerna.

- ***Innehåll:*** Enkla och atomära värden = ett värde i varje kolumn. Bestäm om man kan acceptera Null-värden = avsaknad av värden.

Översättning av ER-diagram:
Ett ER-diagram kan sedan översättas till en relationsdatabas som går att implementera i t.ex. SQL Server.

Översättningen görs i 4 olika steg:

1. Entitetstyper -> Tabell.
   Attribut -> Kolumner.
   Entitetstypens nyckel -> Primärnyckel(helst löpnummer).

2. 1:N sambandstyper blir en främmandenyckel i "många"-tabellen.

3. 1:1 sambandstyper blir en främmandenyckel i en av tabellerna, spelar ingen roll vilken. (ibland kan de slås ihop).

4. N:M sambandstyper blir en egen tabell.

2.4 MVC

MVC är ett design/arkitekturmönster som började bli populärt redan under 1970-talet och används inom all typ av mjukvaru-utveckling. Man kan säga att det är en uppdelning av data, logik och filer för att få en bättre översikt i projekt:

- *Model*: Data / affärsobjekt. Något som innehåller information om något. Ren data och datastruktur utan intelligens.

- *View*: Presentation / UI. Hur saker ska presenteras.

- *Controller*: Logik / beslut. Controllern innefattar all logik, funktionalitet och beslut. Den kan ses som limmet mellan Models och Views.

Exempel: Model med ett databasobjekt, View med HTML-sidor och Controller som bestämmer hur användar-input ska tolkas.

[5]

2.5 ASP.NET

ASP.NET är ett webbapplikations-ramverk som används för att skapa webbapplikationer, webbtjänster och dynamiskt drivna webbplatser. Det skapades av Microsoft som släppte den första versionen i januari 2002. Numera drivs det som ett projekt med öppen källkod av Microsoft och .NET-communityt [6].
ASP.NET Core är en nyare verison av ASP.NET som kan köras på de vanligaste operativsystemen som Windows, Linux och MacOS. Det är likt ASP.NET med samma användningsområden, baserat på öppen källkod och skapat av Microsoft [7].
Dock så finns det en del skillnader mellan de två som man bör ta i beaktning innan man väljer ramverk till ett projekt.
T.ex. om man har olika utvecklingsmiljöer i projektet och behöver ge stöd för cross-plattform så bör man använda .NET Core eftersom det är kompatibelt med de vanligaste operativsystemen.
En nackdel med .NET Core är att all .NET-teknologi inte finns tillgängligt för ramverket som t.ex. Web Forms vilket gör att det i vissa fall är bättre att använda .NET istället [8].

3 Metod

Databas:
Jag börjar med att skapa ER-diagram för att beskriva min relationsdatabas. Risken att skapa en felaktig relationsdatabas är mindre då man gjort ER-diagram eftersom man innan implementation noggrant kan planera dess struktur.
Efter det kommer jag skapa tabell-diagram för att beskriva tabellerna och dess kolumner med datatyper och begränsningar.
Utifrån tabell-diagrammen kan jag sedan simpelt implementera relationsdatabasen i SQL Server.

Databaslager(DAL):
Jag kommer skapa detaljer utifrån tabellerna och databasplaneringen. Detaljerna är klasser som beskriver de olika tabellerna med egenskaper som ska representera kolumnerna och dess datatyper.
Jag kommer skapa metoderna utifrån planeringen. Metoderna kommer ligga i olika klasser och kommer kunna kallas med klassnamn följt av metodnamn.
Databaslagret ligger i Model-delen i MVC-strukturen i webbapplikationen.

Webbapplikation:
Jag kommer skapa webbapplikationen med webbapplikations-ramverket *ASP.NET* med MVC-struktur.

Webbapplikationen inklusive databaslagret kommer jag utveckla stegvis bit för bit så att jag kan testa alla de olika delarna som hör ihop(MVC). Med detta tillvägagångssätt minskar jag tid nedlagd på felhantering då jag ständigt kan testa applikationen och rätta till fel som annars hade upprepat sig på flera ställen innan jag upptäck det.

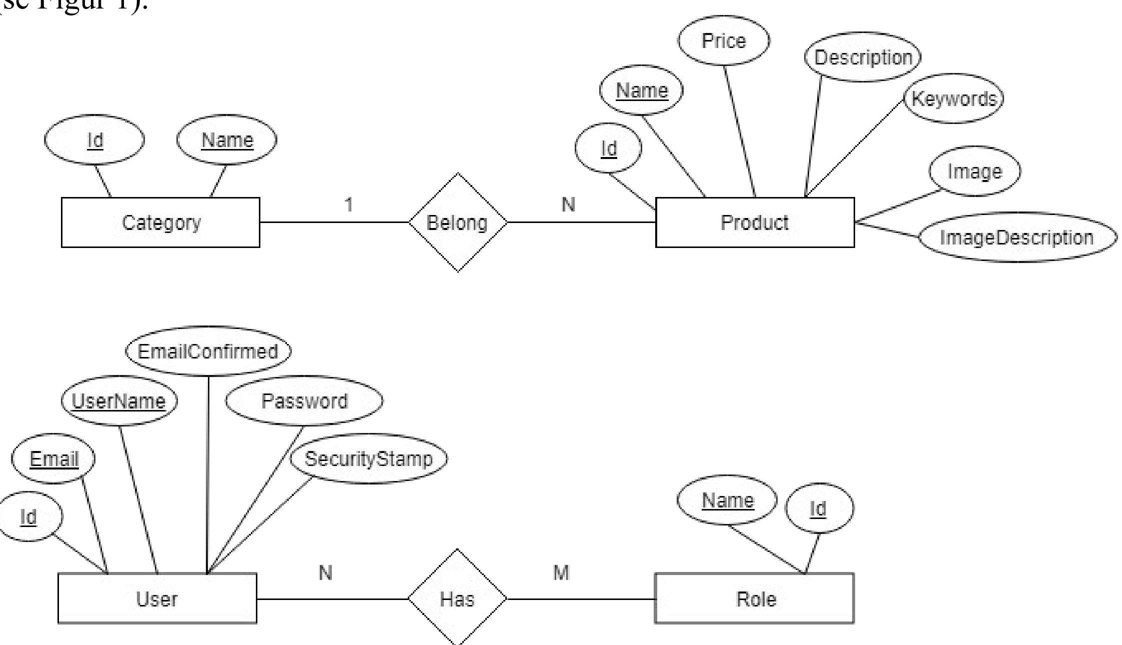
4 Konstruktion

4.1 Databas

Skapa ER-diagram:
Jag börjar med att skapa ER-diagram för att beskriva min databas. Risken att skapa en felaktig databas är mindre då man gjort ER-diagram eftersom man innan implementation noggrant kan planera dess struktur.

Beskrivning:
- En kategori kan innehålla många produkter, en produkt kan endast tillhöra en kategori.
- En användare kan ha flera roller, en roll kan ha flera användare.

(se Figur 1).



Figur 1.

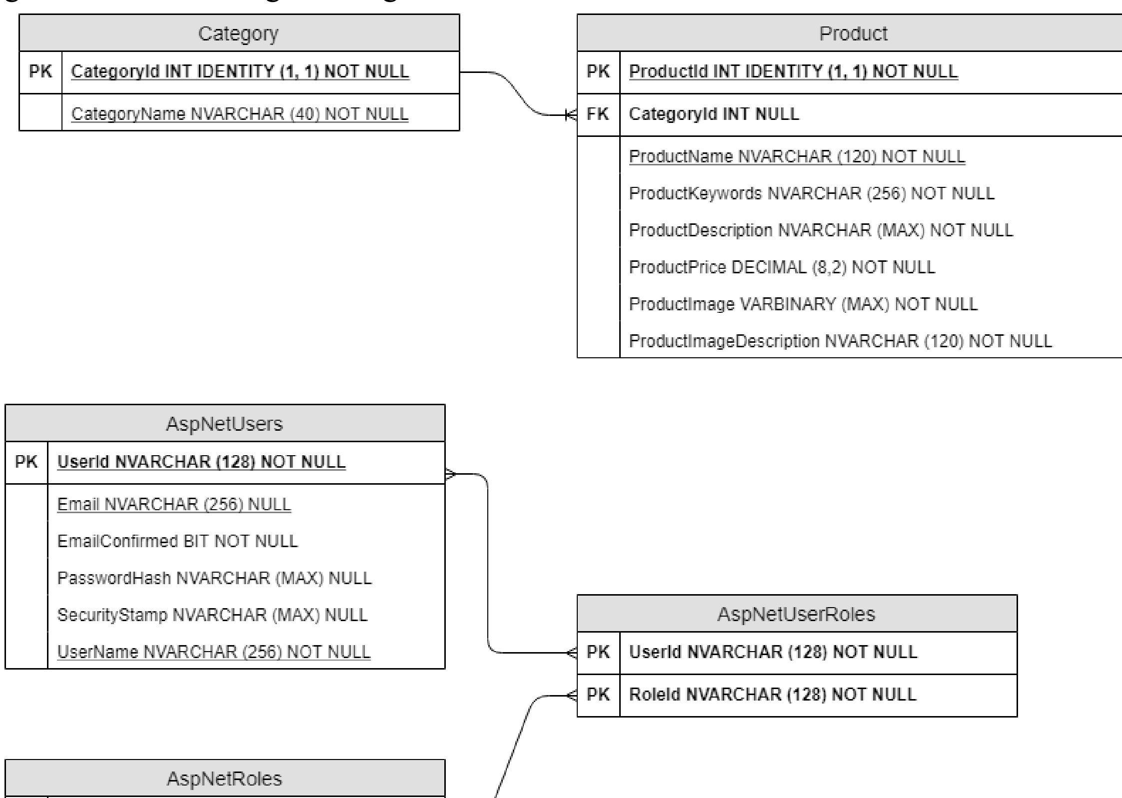
- Kategorierna har ett unikt löpnummer och ett unikt namn.
- Produkterna har ett unikt löpnummer, ett unikt namn, ett pris, en beskrivning, nyckelord, en bild och en bildbeskrivning som används i alt-attributet för bilden senare i applikationen.
- Användare har ett unikt löpnummer, en unik e-postadress, ett unikt användarnamn, en kolumn som ser om e-postadressen är bekräftad eller inte, ett krypterat lösenord och en säkerhetsstämpel eller security stamp som används för att spåra ändringar som görs för användaren, till exempel om lösenord ändras. Denna tabell läggs till automatiskt vid implementering av Identity-systemet där även fler kolumner finns tillgängli-

ga, men ovannämnda kolumner är de jag kommer jobba med i detta projekt.

- Roller har ett unikt löpnummer och ett unikt namn. Även denna tabell läggs till automatiskt av Identity-systemet.

Skapa tabell-diagram

När jag har strukturerat upp min databas i ER-schemat så kan jag översätta det till en relationsdatabas. Se teoridelen ovan för att förstå hur översättningarna går till. Se tabell-diagram i Figur 2 nedan.



Figur 2.

Här anger jag tabellnamn, kolumner och vilken typ av data som ska lagras i varje enskild kolumn, t.ex. en siffra(INTEGER / SMALLINT) eller en sträng(CHAR / VARCHAR), här är VARCHAR ofta att föredra eftersom deklarerade men icke använda tecken inte tar upp lagringsutrymme till skillnad från CHAR som alltid lagrar den deklarerade mängden bytes oavsett antal tecken som används. Om man till exempel lagrar strängen ”Hej” i en kolumn med datatypen VARCHAR(20) så kommer endast 3 bytes för strängen att lagras plus 2 extra bytes för att hålla information om längden. Om kolumnen istället hade haft datatypen CHAR(20) så hade 20 bytes lagrats oavsett längd på strängen. Dock så kan CHAR vara bra om man vet att ett värde alltid kommer ha en spe-

cifik längd men i mitt fall i detta projekt så passar VARCHAR eller NVARCHAR bättre [1]. NVARCHAR är att rekommendera då datatypen även kan lagra Unicode-data [22].
Jag använder datatypen BIT för att lagra boolean-värden. BIT är en datatyp som lagrar integers som kan ha värdet 1, 0 eller NULL. Boolean- / sträng-värdena TRUE och FALSE kan konverteras till BIT-värden där TRUE konverteras till 1 och FALSE till 0 [26].
Jag använder datatypen DECIMAL för att lagra pris där jag sätter begränsningen 8 siffror före decimaltecknet och 2 siffror efter, till exempel: *10000000.00* [27].
Jag använder datatypen VARBINARY med begränsningen MAX för att lagra bilder där kolumndatan kan överskrida 8000 bytes. Denna datatyp lagrar binär data och har ingen fixerad längd precis som VARCHAR [30]. För att kunna lagra en bild på datorn så bryts bilden ner till små element som kallas pixlar. T.ex. så består en bild med upplösningen 1024x798 pixlar av 1024 * 798 pixlar, dvs. 817,152 pixlar. När bilden lagras så presenteras varje pixel av ett binärt värde, en bild lagras alltså som binär data [29].
Jag anger även att vissa kolumner inte får vara tomma med ”NOT NULL”-begränsningen, dvs. att kolumnen inte accepterar NULL-värden.
Jag använder också IDENTITY-egenskapen för att automatiskt ange unika tal till mina egenskapade tabellers löpnummer. Egenskapen tar emot 2 argument, *seed* och *increment*. Seed är värdet som den första raden i tabellen ska ha och increment är det värde som läggs till som värde i nästa rad [23]. Första raden ska alltså ha värdet 1 och därefter ska nya raders värde öka med 1 från föregående rad.
Till sist så anger jag primärnycklar(PK) och främmandenycklar(FK). Primärnycklarna används för att unikt identifiera varje rad i en tabell [24]. Främmandenycklarna används för att länka ihop två eller fler tabeller tillsammans. Det görs genom att man anger en främmandenyckel i en kolumn i en tabell som refererar till en primärnyckel i en annan tabell [25]. Notera att understrukna kolumner ska vara unika.

Implementera databas i Microsoft SQL server
Jag använder mig av två databaser. En för användare och en för produkter. I Web.config skapas en ny databas-anslutning som jag beskriver senare i rapporten vid konfigurering av webbapplikationen. I denna databas kommer jag lägga till mina användar-tabeller. Jag skapar även en egen databasanslutning i Server Explorer där mina övriga tabeller kommer ligga.

Lägg till tabeller till databasen
Som jag nämnde tidigare så läggs användare-, roller-, och användarroller-tabellerna till automatiskt av Identity-systemet. De lägger även till 3 andra tabeller men dessa kommer jag inte använda i detta projekt så jag går endast igenom de tabeller som jag tog med i tabell-diagrammet.

Jag lägger till tabellerna i databasen med hjälp av SQL-satser som utgår efter tabell-diagrammet. För att göra detta så öppnar jag databasanslutningen i *Server Explorer*, högerklickar på *Tables*-mappen och klickar på *New Query*. Då öppnas en ny .sql-fil där jag kan skriva SQL-satser för att skapa tabellerna.
Jag använder SQL-uttrycket ”CREATE TABLE” för att skapa en ny tabell i databasen. Inom paranterserna lägger jag sedan till kolumnerna med tillhörande datatyp, begränsningar och acceptans av NULL-värden.
Jag anger också att Id-kolumnerna(löpnumren) ska vara primärnycklar med ”PRIMARY KEY”-restriktionen. För att döpa denna primärnyckel så anger jag även ”CONSTRAINT [namn_på_nyckel]”.
Jag använder ”[]” runt namnen för att försäkra mig om att jag inte använder några fördefinierade ord.
När jag exekverar detta script så läggs det till i *Tables*-mappen under min databasanslutning. Se tabeller nedan.

- Kategorier:
  Tabellnamn: *Categories*.

```
CREATE TABLE [dbo].[Categories]
(
    [Id] INT IDENTITY (1, 1) NOT NULL,
    [CategoryName] NVARCHAR (60) NOT NULL,
    CONSTRAINT [PK_Categories] PRIMARY KEY ([Id])
)

GO

CREATE UNIQUE INDEX [IX_Categories_CategoryName] ON [dbo].[Categories] ([CategoryName])
```

Figur 3

Om vi kollar i tabelldiagrammet i Figur 2 så ser vi att både Id-kolumnen och namn-kolumnen ska vara unika. Primärnycklar blir per automatik unika i SQL men för vanliga kolumner så behöver jag sätta ”UNIQUE”-begränsningar. Denna begränsning försäkrar att inga duplicerade värden matas in i specifika kolumner som inte är primärnycklar. Det innebär till exempel att det endast kan finnas en kategori med namnet ”TV” och om man försöker lägga till i en ny kategori med samma namn så kommer den inte att lagras [28].

- Produkter:
  Tabellnamn: *Products*.

```
CREATE TABLE [dbo].[Products] (
    [Id]                      INT             IDENTITY (1, 1) NOT NULL,
    [CategoryId]              INT             NULL,
    [ProductName]             NVARCHAR (120)  NOT NULL,
    [ProductKeywords]         NVARCHAR (256)  NOT NULL,
    [ProductDescription]      NVARCHAR (MAX)  NOT NULL,
    [ProductPrice]            DECIMAL (8, 2)  NOT NULL,
    [ProductImage]            VARBINARY (MAX) NOT NULL,
    [ProductImageDescription] NVARCHAR (120)  NOT NULL,
    CONSTRAINT [PK_Products] PRIMARY KEY CLUSTERED ([Id] ASC),
    CONSTRAINT [FK_Products_Categories_CategoryId] FOREIGN KEY ([CategoryId]) REFERENCES [dbo].[Categories] ([Id]) ON DELETE SET NULL
);


GO
CREATE UNIQUE NONCLUSTERED INDEX [IX_Products_ProductName]
    ON [dbo].[Products]([ProductName] ASC);
```

Figur 4

Jag sätter en ”UNIQUE”-begränsning på produktnamn så att två produkter i tabellen inte kan ha samma namn.
Här skapar jag även en främmandenyckel-begränsning för kolumnen *CategoryId* som refererar till *Id*-kolumnen i *Categories*-tabellen. Denna begränsning innebär att jag måste ange ett *CategoryId* som även existerar i tabellen *Categories*. Alltså kan jag inte lägga till en kategori för produkten som inte existerar.
Jag använder också ”DELETE SET NULL” på främmandenyckeln. Det innebär att om jag tar bort en kategori i *Categories*-tabellen och den kategorin refereras till i *Products*-tabellen så kommer alla produkter som tillhör den kategorin att bli av med sin kategori. Då får de produkterna ett NULL-värde i *CategoryId*-kolumnen vilket betyder att de inte tillhör någon kategori.

- Användare:
  Tabellnamn: *AspNetUsers*.

```
CREATE TABLE [dbo].[AspNetUsers] (
    [Id]                   NVARCHAR (128) NOT NULL,
    [Email]                NVARCHAR (256) NULL,
    [EmailConfirmed]       BIT            NOT NULL,
    [PasswordHash]         NVARCHAR (MAX) NULL,
    [SecurityStamp]        NVARCHAR (MAX) NULL,
    [PhoneNumber]          NVARCHAR (MAX) NULL,
    [PhoneNumberConfirmed] BIT            NOT NULL,
    [TwoFactorEnabled]     BIT            NOT NULL,
    [LockoutEndDateUtc]    DATETIME       NULL,
    [LockoutEnabled]       BIT            NOT NULL,
    [AccessFailedCount]    INT            NOT NULL,
    [UserName]             NVARCHAR (256) NOT NULL,
    CONSTRAINT [PK_dbo.AspNetUsers] PRIMARY KEY CLUSTERED ([Id] ASC)
);


GO
CREATE UNIQUE NONCLUSTERED INDEX [UserNameIndex]
    ON [dbo].[AspNetUsers]([UserName] ASC);
```

Figur 6

Detta är alla kolumner som kommer med Identity, se tabelldiagram i Figur 2 för att se vilka jag använder mig av. Jag sparar de resterande kolumnerna utifall jag skulle vilja vidareutveckla min applikation.
Jag sätter en ”UNIQUE”-begränsning på UserName med samma teknik som innan så att två användare inte kan ha samma namn.

- Roller:
  Tabellnamn: *AspNetRoles*.

```
CREATE TABLE [dbo].[AspNetRoles] (
    [Id]   NVARCHAR (128) NOT NULL,
    [Name] NVARCHAR (256) NOT NULL,
    CONSTRAINT [PK_dbo.AspNetRoles] PRIMARY KEY CLUSTERED ([Id] ASC)
);


GO
CREATE UNIQUE NONCLUSTERED INDEX [RoleNameIndex]
    ON [dbo].[AspNetRoles]([Name] ASC);
```

  Figur 7

  Jag sätter en ”UNIQUE”-begränsning på rollnamnen så att två roller inte kan ha samma namn.

- Användarroller:
  Tabellnamn: *AspNetUserRoles*.

```
CREATE TABLE [dbo].[AspNetUserRoles] (
    [UserId] NVARCHAR (128) NOT NULL,
    [RoleId] NVARCHAR (128) NOT NULL,
    CONSTRAINT [PK_dbo.AspNetUserRoles] PRIMARY KEY CLUSTERED ([UserId] ASC, [RoleId] ASC),
    CONSTRAINT [FK_dbo.AspNetUserRoles_dbo.AspNetRoles_RoleId]
    FOREIGN KEY ([RoleId]) REFERENCES [dbo].[AspNetRoles] ([Id]) ON DELETE CASCADE,
    CONSTRAINT [FK_dbo.AspNetUserRoles_dbo.AspNetUsers_UserId]
    FOREIGN KEY ([UserId]) REFERENCES [dbo].[AspNetUsers] ([Id]) ON DELETE CASCADE
);
```

  Figur 8

  Jag använder ”DELETE CASCADE” på främmandenycklarna. Det innebär att om jag tar bort en roll i *AspNetRoles*-tabellen och den rollen existerar i *AspNetUserRoles*-tabellen så kommer alla användarroller med den rollen också att tas bort. Dvs. om jag tar bort rollen ”Anställda” så kommer alla användare som hade den rollen att bli av med sin roll. Detsamma gäller när jag tar bort en användare, då kommer alla rader med den användaren att tas bort i *AspNetUserRoles*-tabellen.

4.2 Databaslager (DAL)

Denna sektion är en beskrivande del där jag förklarar mer ingående vilka metoder jag kommer använda för att kommunicera med databasen från applikationen. Själva databaslagrets implementerade delar kommer beskrivas under MVC-sektionen i modell-delen.
Jag kommer huvudsakligen använda mig av tre metoder för att interagera med databasen: en SelectSingle-metod där jag hämtar ett enstaka objekt från databasen, en SelectAll-metod som hämtar en lista med objekt och en Modify-metod där jag modifierar data i databasen(skapa, uppdatera, radera). Dessa metoder kommer att se lite olika ut beroende på vad som ska hämtas / modifieras men grunden är lik och kommer användas genomgående i hela projektet.

- **SelectSingle-metod:**

  - Skapa metod
  Som ett exempel så hämtar jag en kategori från databasen utifrån ett kategorinamn. Metoden returnerar ett objekt som representerar en kategori utifrån *CategoryDetail*-klassen efter inmatat namn:
  ```
  public CategoryDetail SelectSingleCategory(CategoryDetail cd, out string errormsg).
  ```
  Metoden tar ett *CategoryDetail*-objekt som parameter och skickar med ett felmeddelande till den kallande metoden med *out*-nyckelordet[33].

  - Skapa databasanslutning
  I metoden så börjar jag med att skapa en ny anslutning till min SQL Server-databas med *SqlConnection*-klassen som just representerar en anslutning till en SQL Server-databas [34]. För att det ska fungera så måste jag även inkludera namespacet *System.Data.SqlClient*.
  ```
  SqlConnection dbConnection = new SqlConnection();.
  ```

  Jag använder sedan *SqlConnection.ConnectionString*-egenskapen[35] för att ange en sträng som används för att öppna en SQL Server-databas. Denna sträng hittar man genom att högerklicka på databasanslutningen som man vill använda i Server Explorer och sedan på properties vilket öppnar upp en ny ruta i Solution Explorer där det finns en property med namn *Connection String*.
  ```
  dbConnection.ConnectionString = "[anslutningssträng]";.
  ```

  - Skapa SQL-sats
  När jag är ansluten till databasen så utformar jag en SQL-sats för att hämta alla fält från en rad i tabellen *Categories* som matchar både den inmatade e-postadressen och lösenordet:
  ```
  String sqlstring = "SELECT * FROM [Categories] WHERE [Categories].[CategoryName] = @categoryName";.
  ```

  Denna SQL-sats lagrar jag sedan i ett *SqlCommand*-objekt med hjälp av *SqlCommand*-klassen:
  ```
  SqlCommand dbCommand = new SqlCommand(sqlstring, dbConnection);.
  ```

Där första parametern är min SQL-sats och andra parametern är min databasanslutning. *SqlCommand*-klassen representerar en SQL-sats eller en lagrad procedur som ska exekveras mot en SQL Server-databas [36].

Efter det behöver jag ange vilka parametrar som kan skickas med och vilka värden de får ha. Det är alltså värdena med ett inledande "@" i SQL-satsen som jag vill att användare ska kunna sätta egna värden på. Mer specifikt så behöver jag lägga till parametrar till det tidigare skapade *SqlCommand*-objektet, det gör jag med *SqlCommand.Parameters*-egenskapen som representerar en SQL-sats eller en procedurs parametrar [37]. Med hjälp av *Add()*-metoden så lägger jag till parametrarna till *SqlParameterCollection*-klassen som representerar en samling av parametrar associerade med ett *SqlCommand*-objekt och dess relationer till kolumner i en tabell [38]. Exempel för att lägga till *@categoryName*-parametern:

```
dbCommand.Parameters.Add("categoryName", System.Data.SqlDb-
Type.NVarChar, 60).Value = cd.CategoryName;.
```

Första parametern är namnet på parametern och andra är dess datatyp som jag anger med *SqlDbType*-enum vilket specificerar en SQL Server-specifik datatyp av ett fält eller en egenskap som används i en *SqlParameter*-klass [39]. Jag anger även längd på strängen i tredje parametern. *cd.CategorName* refererar till egenskapen *CategoryName* i *CategoryDetail*-objektet som togs emot som en parameter i själva *GetCategory()*-metoden.

- Skapa adapter & ett dataset

Sedan skapar jag en adapter, detta görs med *SqlDataAdapter*-klassen som representerar en samling av data-kommandon och databasanslutningar som används för att fylla på ett *DataSet* och uppdatera en SQL Server-databas [40]. *DataSet*-klassen är en samling av relaterad data [41]. Jag måste även göra en instans av just *DataSet*-klassen och för att kunna göra det så måste jag även använda namespacet *system.Data*.

```
SqlDataAdapter myAdapter = new SqlDataAdapter(dbCommand);
DataSet myDS = new DataSet();.
```

- Exekvera SQL-sats

Jag exekverar SQL-satsen med ett *try...catch...finally*-uttryck.
I *try*-blocket öppnar jag först en databasanslutning:

```
dbConnection.Open();.
```

Sedan fyller jag mitt *DataSet* med data i en tabell som jag döper till ”myUser”:

```
myAdapter.Fill(myDS, "myCategory");.
```

Fill()-metoden lägger till eller uppdaterar rader i ett DataSet för att matcha raderna i Datakällan [42].

Efter det skapar jag en ny räknare som sedan används för att räkna igenom alla rader i mitt *DataSet* med tabellnamnet "myCategory" som jag skapade tidigare:
```
int i = 0;
int count = 0;
count = myDS.Tables["myCategory"].Rows.Count;.
```
Tables-egenskapen hämtar tabellen i mitt DataSet, *Rows*-egenskapen hämtar raderna och *Count*-egenskapen hämtar antal rader.

Jag skapar sedan en if-sats. Om det finns fler än 0 rader i mitt DataSet, alltså om SQL-satsen returnerade någon rad, så skapar jag en ny instans av *CategoryDetails*-klassen och konverterar samt lägger till värdena från raderna i mitt *DataSet* till klassens egenskaper:
```
CategoryDetail cd2 = new CategoryDetail
{
       CategoryName = myDS.Tables["myCategory"].Rows[i]["Category
       Name"].ToString(),
       Id = Convert.ToInt32(myDS.Tables["myCategory"].Rows[i]
       ["Id"])
};.
```

Jag returnerar sedan *CategoryDetail*-objektet:
```
return cd2;.
```

Annars om det inte finns några rader i mitt DataSet, dvs. om SQL-satsen ej returnerade en rad, så returnerar jag null och skickar med ett felmeddelande.

I *catch*-blocket fångar jag eventuella fel med *Exception*-klassen som representerar fel som kan uppstå vid exekvering av applikationen [17]. Om detta sker så returnerar jag även null.

Till sist i *finally*-blocket så stänger jag databasanslutningen:
```
dbConnection.Close();.
```

Jag har nu skapat en metod som kan användas för att hämta kategorier vilket med hjälp av en vy och controller kan användas för att skriva ut en kategori. Detta tillvägagångssätt kommer vara genomgående i alla metoder där jag vill hämta ett enskilt databasobjekt.

- **SelectAll-metod:**
  Denna metod fungerar på ungefär samma sätt som den innan men här returneras en lista med objekt istället för ett enda enskilt objekt. Som ett exempel så tänker jag returnera en lista med alla kategorier.
  ```
  public List<CategoryDetail> SelectAllCategories(out string er-
  rormsg).
  ```

SQL-satsen kan se ut på följande vis:
`"SELECT * FROM [Categories]";`.

Jag skapar sedan en adapter och ett DataSet på samma sätt som tidigare men här skapar jag även en ny lista som jag vid exekvering fyller på med objekt från mitt DataSet:
`List<CategoryDetail> CategoryList = new List<CategoryDetail>();`.

Eftersom det är en lista som ska returneras så måste jag iterera igenom flera rader i mitt DataSet. Det gör jag med en while-loop som körs igenom alla rader i mitt DataSet. För varje iteration så lägger jag till ett objekt i min lista:
`CategoryList.Add(cd2);`.

Till sist returnerar jag listan:
`return CategoryList;`.

Detta tillvägagångssätt kommer vara genomgående i alla metoder där jag vill hämta en lista med databaobjekt.

- **Modify-metod:**
  Denna metod är väldigt lik de två ovanstående. Som ett exempel tänker jag skapa en ny kategori.
  `public int InsertCategory(CategoryDetail cd, out string errormsg)`.

  Jag ansluter till databasen på samma sätt som innan och skapar en SQL-sats som kan se ut på följande vis:
  `INSERT INTO Categories (CategoryName) VALUES (@categoryName);`.
  Sedan lägger jag till parametrar som innan men nu skapar jag ingen adapter eller något DataSet.
  Sedan exekverar jag SQL-satsen mot databasen på ungefär samma sätt som i metoderna innan men här vill jag returnera en integer istället för ett objekt eller en lista. För att göra detta så skapar jag en ny integer med värdet noll som jag sedan sätter till värdet som jag får tillbaka av metoden *SqlCommand.ExecuteNonQuery()*. Denna metod exekverar SQL-satsen mot databasen och returnerar antalet rader som påverkades[43]:
  ```
  int i = 0;
  i = dbCommand.ExecuteNonQuery();
  ```
  Om jag får tillbaka ett värde av noll så innebär det att lagringen misslyckades men om jag får tillbaka ett värde av 1 eller högre så innebär det att en eller flera rader påverkades. Jag kan sedan använda detta värde för att skriva ut felmeddelanden med en if-sats och även returnera värdet för att se hur många rader som påverkades.

  Detta tillvägagångssätt kommer vara genomgående i alla metoder där jag vill skapa, uppdatera eller radera ett databasobjekt.

4.3 Webbapplikation

4.3.1 Konfigurering

Skapa projekt

Jag börjar med att skapa en ny webbapplikation med *ASP.NET Web Application (.NET Framework)*. Jag väljer MVC-mallen och ändrar autentiseringen till *Individual User Accounts*.
När jag klickar på ”ok” så genereras ett projekt som lägger till ett antal NuGet-paket. NuGet är en pakethanterare för .NET som används för att skapa, dela och konsumera användbar kod. Koden läggs till i ett ”paket” som innehåller kompilerad kod och annat innehåll som behövs i projekten för att kunna konsumera paketet [10].
När projektet är genererat så går jag in i pakethanteraren NuGet och uppdaterar alla paket till senaste versionen som sedan sammanfogas i projektet.

Nu har jag skapat en grunden för webbapplikationen med ett fullt fungerande inloggningssystem baserat på *ASP.NET Identity membership system*[11] som erhåller funktionalitet för bland annat registrering, inloggning och ändring av lösenord. Detta system genererar klasser och gränssnitt relaterat till hantering av användare och roller utifrån namnrymden *Microsot.AspNet.Identity* [12] som jag kan konfigurera så att det passar min applikation. Det genererar även en lokal databas med 6 tabeller som beskriver användare och dess egenskaper som användarinformation, roller osv.

Konfigurera inloggningssystem

Validera användare vid registrering

Under mappen *App_Start* finns det en fil med namnet *IdentityConfig.cs* där jag i metoden *Create()* konfigurerar valideringslogiken vid registrering av nya konton.
För att validera användarnamn så använder jag klassen *UserValidator* som erhåller ett antal egenskaper och metoder för att validera användare innan sparning [13].
I klassen sätter jag egenskapen *RequireUniqueEmail* till ”true” vilket talar om för systemet att det inte kan finnas två användare med samma e-postadress, dvs. att e-postadressen måste vara unik [13].
Jag ändrar också egenskapen *AllowOnlyAlphanumericUserNames* och sätter den till boolean-värdet ”false” vilket talar om för systemet att användarnamnet kan innehålla andra tecken än bara de som ingår i en alfanumerisk teckenuppsättning (A-Z, 0-9) [13].

```
manager.UserValidator = new UserValidator<ApplicationUser>(manager)
{
        AllowOnlyAlphanumericUserNames = false,
        RequireUniqueEmail = true
};.
```

Eftersom inloggade användare hanterar det som visas på sidan så vill jag ha extra säkerhet och det kan jag få genom att tvinga användaren att fylla i ett säkert lösenord med många tecken och komplexa kombinationer av bokstäver, siffror och andra typer av tecken. Komplexa lösenord gör det bland annat betydligt svårare för attacker som brute-force och dictionary att gissa rätt lösenord. Dessa attack-tekniker utförs genom att testa sig fram och jämföra ett lösenord med andra vanliga lösenord, oftast med hjälp av ett program som till exempel Cain and Abel [15]. Framförallt dictionary-attacks har en tendens att lyckas när lösenorden är korta, vanliga och simpla [16]. Detta är bara en extra säkerhetsåtgärd och är endast nödvändig om någon obehörig lyckas ta sig in i databasen och få tag på de krypterade lösenorden.
För att validera lösenord så använder jag klassen *PasswordValidator* som erhåller ett antal egenskaper och metoder för just validering av lösenord [14].
I klassen sätter jag egenskapen *RequiredLength* till integer-värdet till ”10” vilket sätter minimum-längden på lösenordet till 10 tecken [14].
Jag sätter egenskapen *RequireNonLetterOrDigit* till ”true” som säger att lösenordet måste bestå av minst ett tecken som inte är en bokstav eller en siffra [14].
Jag sätter egenskapen *RequireDigit* till ”true” som säger att lösenordet måste innehålla minst en siffra [14].
Till sist sätter jag egenskaperna RequireLowercase och RequireUppercase till ”true” som säger att det måste finnas minst en gemen och minst en versal [14].

```
manager.PasswordValidator = new PasswordValidator
{
	RequiredLength = 10,
	RequireNonLetterOrDigit = true,
	RequireDigit = true,
	RequireLowercase = true,
	RequireUppercase = true,
};.
```

Aktivera autentisering

Eftersom endast inloggade användare ska kunna göra vissa saker så måste jag neka åtkomst för icke inloggade användare en del ställen. Detta gör jag med *Authorize*-atributet som specificerar att åtkomsten till action-metoderna begränsad till användare som möter autentiseringskraven [19]. Om icke inloggade besökare försöker gå in på en sådan sida så kommer de direkt bli omdirigerade till inloggningssidan.

```
[Authorize].
```

För att detta ska fungera så måste jag även kommentera / ta bort följande rad i *Web.config* under *<modules>*:

```
<!-- <remove name="FormsAuthentication" /> -→.
```

Konfigurera projekt

Anslut till användardatabas

I root-katalogen finns det en fil som heter *Web.config*. I denna fil ansluter jag till användardatabasen med dess anslutningssträng i elementet *<connectionsStrings>*.

```
<connectionStrings>
	<add name="DefaultConnection"
	connectionString="[anslutningssträng]"
	providerName="System.Data.SqlClient"/>
</connectionStrings>.
```

Anslutningssträngen hämtas i *Server Explorer* under *Data Connections* i anslutningen *DefaultConnection* där jag högerklickar på anslutningen och väljer *Properties* och egenskapen *Connection String* som har värdet av anslutningssträngen.

Ändra grundrouting

I mappen App_Start i filen RouteConfig.cs ändrar jag grundroutingen så att den startar på index-sidan i Home-controllern, dvs. så att webbapplikationen läser in vyn för huvud-startsidan:

```
routes.MapRoute(
	name: "Default",
	url: "{controller}/{action}/{id}",
	defaults: new { controller = "Home", action = "Index", id =
	UrlParameter.Optional }
);.
```

När en MVC-applikation startas så anropas metoden *Application_Start()* som i sin tur anropar metoden *RegisterRoutes()*. Inuti metoden används *MapRoute()*-metoden för att skapa en specifik route med namnet ”default”. Routen matchar matchar en URL-sökväg som ser ut på följande vis: ”*/Home/Index”*. MVC försöker sedan hitta en controller med namnet *Home* och exekvera en action-metod med namnet *Index*.

Route-parametrar:

- {controller=”Home”} definierar *Home* som grund-controller.
- {action=”Index”} definierar *Index* som default-action-metod.
- {id = UrlParameter.Optional} definierar *id* som en alternativ URL-parameter.

[17]

Ändra layout

Under *Views*-mappen i *Shared*-mappen finns det en vy som heter *_Layout.cshtml*. Denna fil definierar en mall som är genomgående i alla vyer. Bland annat titel, meny och footer.

För att läsa ut användarlänkarna i menyn så använder jag en partial som jag inkluderar i vyn med razor-syntax med *Html.Partial()*-metoden:

```
@Html.Partial("_LoginPartial").
```

En partial-vy är en vy som kan återanvändas. På så sätt slipper man skriva om samma kod flera gånger vilket kan bli omständligt i större projekt [18]. Denna _LoginPartial är förinställd där länkarna matchar rätt controller och action-metod så jag behöver endast översätta länk-texten till svenska.
Jag ändrar även vilka länkar som ska visas beroende på om besökaren är inloggad eller inte. Detta görs med en if-sats där jag med hjälp av *IsAuthenticated*-egenskapen hämtar ett värde som bekräftar om anropet är autentiserat eller inte. Egenskapen returnerar ”true” om anropet är autentiserat, dvs. om användaren är inloggad. Annars returnerar den boolean-värdet ”false” [20].

```
@if (Request.IsAuthenticated).
```

Jag kan också ange en roll till if-satsen så att endast användare med en viss roll kan få åtkomst till viss innehåll. Detta gör jag med IsInRole()-metoden som avgör om den nuvarande användaren tillhör en specifik roll eller inte [21]:

```
@if (Request.IsAuthenticated && User.IsInRole("Admin")).
```

4.3.2 MVC

Jag har redan smygstartat med några MVC-filer men nedan följer den huvudsakliga MVC-delen med innehållet i applikationen.

Modeller / databaslager
I modellerna ligger applikationens logik med valideringsregler, databasanslutningar och andra regler. Jag har valt att dela in mina modeller i två delar: detaljer och metoder.
Detaljer:
Detaljer är klasser som används för att skapa objekt i webbapplikationen som jag sedan kan konvertera och använda för att jobba mot databasens tabeller. Jag lägger till automatiserade egenskaper till detaljerna som beskriver kolumnerna i de olika tabellerna. Det är viktigt att använda en datatyp för egenskaperna som går att översätta till datatypen i respektive kolumner i databasen. Jag skapar även en tom konstruktor som initierar alla numeriska egenskaper med värdet 0 och alla sträng- samt objekt-egenskaper med värdet null.
I detaljerna lägger jag till valideringsregler och data-annotering. Detta görs med namespacet *System.ComponentModel.DataAnnotations* som är en del av .NET-ramverket. Först anger jag att mina egenskaper måste ha ett värde med *Require*-attributet. Sedan skriver jag ut felmeddelanden med *ErrorMessage*-attributet [32]. Jag definierar också hur jag vill att egenskaperna ska visas för användarna i vyn i till exempel formulär. Detta gör jag med *DisplayName*-attributet där jag lägger in en parameter med det namn som ska visas i vyn. Se detaljer nedan.

- *AccountViewModels*, *ManageViewModels* och *IdentityModels*
  Dessa är modeller som genereras av Identity-systemet. Jag kommer inte gå igenom de i detalj då de redan är väldokumenterade och ändå kommer att genereras automatiskt. De är kortfattat modeller för att beskriva användare, ändring av kontouppgifter och övrig användarrelaterad data.

- *CategoryDetail*
  Klass som beskriver *Categories*-tabellen (se Figur 9).

```
public class CategoryDetail
{
    // Constructor
    public CategoryDetail() { }

    // Auto-implemented properties
    // Id
    public int Id { get; set; }

    // CategoryName
    [Required(ErrorMessage = "Ange ett kategorinamn")]
    [DisplayName("Kategori")]
    public string CategoryName { get; set; }
}
```
Figur 9.

- *ProductDetail*
  Klass som beskriver *Products*-tabellen (se Figur 10).

```
public class ProductDetail
{
    // Constructor
    public ProductDetail() { }

    // Auto-implemented properties
    // Id
    public int Id { get; set; }

    // CategoryId
    public int CategoryId { get; set; }

    // ProductName
    [Required(ErrorMessage = "Ange ett produktnamn")]
    [DisplayName("Produkt")]
    public string ProductName { get; set; }

    // ProductKeywords
    [Required(ErrorMessage = "Ange nyckelord")]
    [DisplayName("Nyckelord")]
    public string ProductKeywords { get; set; }

    // ProductDescription
    [Required(ErrorMessage = "Ange en beskrivning")]
    [DisplayName("Beskrivning")]
    public string ProductDescription { get; set; }

    // ProductPrice
    [Required(ErrorMessage = "Ange ett pris")]
    [DisplayName("Pris")]
    public decimal ProductPrice { get; set; }

    // ProductImage
    [Required(ErrorMessage = "Ladda upp en bild")]
    [DisplayName("Bild")]
    public byte[] ProductImage { get; set; }

    // ProductImageDescription
    [Required(ErrorMessage = "Ange en bildbeskrivning")]
    [DisplayName("Bildbeskrivning")]
    public string ProductImageDescription { get; set; }
}
```

  Figur 10.

  Som jag nämnde tidigare så lagras en bild som binär data i en databas. Binär data är en sekvens av bytes och kan presenteras som en byte-array(*byte[]*).

- *CreateProductDetail*
  Klass som beskriver *Products*-tabellen och en lista med kategorier från *Categories*-tabellen. Med denna detalj kan jag hämta både en produkt och en lista med kategorier (se Figur 11).

```
public class CreateProductDetail
{
    public ProductDetail Product { get; set; }
    public IEnumerable<CategoryDetail> CategoryList { get; set; }
}
```

  Figur 11.

- *ViewModelPCC*
  En lista med produkter från *Products*-tabellen och en lista med kategorier samt en enskild kategori från *Categories*-tabellen. Ser ut på samma sätt som i Figur 11 fast *ProductDetail*-objektet blir en lista med objekt.

Metoder:
Metoderna är klasser med metoder som använder detalj-objekten för att hämta eller modifiera data i databasen. Jag kommer använda mig av tillvägagångssätten som beskrivs i sektion 4.2 ”Databaslager (DAL)”.
Se metoder nedan.

CategoryMethods
CRUD-funktionalitet för *Categories*-tabellen.

- *SelectAllCategories()*
  SelectAll-metod. Metoden returnerar en lista med alla kategorier.
  `public List<CategoryDetail> SelectAllCategories(out string errormsg)`.
  SQL-sats:
  `SELECT * FROM [Categories]`.

- *SelectSingleCategory()*
  SelectSingle-metod. Metoden returnerar en enskild kategori utifrån det inmatade id:t.
  `public CategoryDetail SelectSingleCategory(int id, out string errormsg)`.
  SQL-sats:
  `SELECT * FROM [Categories] WHERE Id = @id`.

- *InsertCategory()*
  Modify-metod. Metoden tar emot ett *CategoryDetail*-objekt och lägger till den som en ny kategori i databasen.
  `public int InsertCategory(CategoryDetail cd, out string errormsg)`.
  SQL-sats:
  `INSERT INTO [Categories] (CategoryName) VALUES (@categoryName)`.

- *UpdateCategory()*
  Modify-metod. Metoden tar emot ett *CategoryDetail*-objekt och uppdaterar en rad i databasen som matchar kategorins id.
  `public int UpdateCategory(CategoryDetail cd, out string errormsg)`.
  SQL-sats:
  `UPDATE [Categories] SET CategoryName =  @categoryName WHERE Id = @id`.

- *DeleteCategory()*
  Modify-metod. Metoden tar bort en kategori som matchar det inmatade id:t.
  `public int DeleteCategory(int id, out string errormsg)`.
  SQL-sats:
  `DELETE FROM [Categories] WHERE Id = @id`.

ProductMethods
CRUD-funktionalitet för *Products*-tabellen.

- *SelectAllProducts()*
  SelectAll-metod. Metoden returnerar en lista med alla produkter.
  `public List<ProductDetail> SelectAllProducts(out string errormsg)`.
  SQL-sats:
  `SELECT * FROM [Products]`.

- *SelectSingleProduct()*
  SelectSingle-metod. Metoden returnerar en enskild produkt utifrån det inmatade id:t.
  `public ProductDetail SelectSingleProduct(int id, out string errormsg)`.
  SQL-sats:
  `SELECT * FROM [Products] WHERE Id = @id`.

- *InsertProduct()*
  Modify-metod. Metoden tar emot ett *ProductDetail*-objekt och lägger till den som en ny produkt i databasen.
  `public int InsertProduct(ProductDetail pd, out string errormsg)`.
  SQL-sats:
  ```
  INSERT INTO [Products]
  (CategoryId, ProductName, ProductKeywords, ProductDescription,
  ProductPrice, ProductImage, ProductImageDescription)
  VALUES (@categoryId, @productName, @productKeywords,
  @productDescription, @productPrice, @productImage,
  @productImageDescription).
  ```

- *UpdateProduct()*
  Modify-metod. Metoden tar emot ett *ProductDetail*-objekt och uppdaterar en produkt i databasen som matchar id:t.
  `public int UpdateProduct(ProductDetail pd, out string errormsg)`.
  SQL-sats:
  ```
  UPDATE [Products] SET
  CategoryId = @categoryId,
  ProductName =  @productName,
  ProductKeywords = @productKeywords,
  ```

```
ProductDescription = @productDescription,
ProductPrice = @productPrice,
ProductImage = @productImage,
ProductImageDescription = @productImageDescription
WHERE Id = @id.
```

- *DeleteProduct()*
  Modify-metod. Metoden tar bort en produkt som matchar det inmatade id:t.
  ```
  public int DeleteProduct(int id, out string errormsg).
  ```
  SQL-sats:
  ```
  DELETE FROM [Products] WHERE Id = @id.
  ```

- *SelectProductsByCategory()*
  SelectAll-metod. Metoden returnerar en lista med produkter vars kategori-id matchar det inmatade id:t.
  ```
  public List<ProductDetail> SelectProductsByCategory(int
  categoryId, out string errormsg).
  ```
  SQL-sats:
  ```
  SELECT * FROM [Products] WHERE [CategoryId] = @categoryId.
  ```

- *SelectProductsByName()*
  SelectAll-metod. Metoden returnerar en lista med produkter vars namn matchar den inmatade söksträngen.
  ```
  public List<ProductDetail> SelectProductsByName(string
  productName, out string errormsg).
  ```
  SQL-sats:
  ```
  SELECT * FROM [Products] WHERE [ProductName] LIKE
  '%'+@productName+'%'.
  ```

Controllers
AccountController, ManageController
Controllers för att hantera användare. Dessa controllers genereras automatiskt av identity-systemet och de ändringar jag gör är mest för översättningar och routes. För att rapporten inte ska bli för lång så hänvisar jag till Microsofts dokumentation för mer information om dessa controllers.

EmployeeController
Controller för att hantera produkter och kategorier.
Eftersom endast anställda ska kunna hantera produkter och kategorier så begränsar jag åtkomsten till action-metoderna och säger att endast autentiserade användare får besöka den. Detta gör jag med Authorize-attributet:
`[Authorize]`.
Denna roll-begränsning kommer vara genomgående i alla action-metoder för *EmployeeController*.

Hantera kategorier:

- *[HttpGet] Categories()*
  I denna action-metod skickar jag med en lista med alla kategorier till vyn och returnerar den.
  Först anger jag action-metoden är begränsad till endast Http GET-anrop med HttpGet-attributet:
  `[HttpGet]`.
  I action-metoden skapar jag först en instans av klassen CategoryMethods där mina kategori-metoder ligger:
  `CategoryMethods cm = new CategoryMethods();`.
  Sedan skapar jag en ny kategori-lista utifrån CategoryDetail-klassen som beskriver en kategori:
  `List<CategoryDetail> CategoryList = new List<CategoryDetail>();`.
  Efter det anropar jag *SelectAllCategories()*-metoden i klassen *CategoryMethods* som hämtar en lista med alla kategorier som jag sedan lägger till i kategori-listan *CategoryList*.
  `CategoryList = cm.SelectAllCategories(out string error);`.

  Till sist skickar jag med eventuella felmeddelande med en ViewBag och skickar med kategori-listan till vyn.

- *[HttpGet] CreateCategory()*
  Här returnerar jag ett formulär för att skapa en ny kategori. Formuläret skickas till action-metoden *[HttpPost] CreateCategory()*.

- *[HttpPost] CreateCategory()*
  Hit skickas ”skapa kategori”-formuläret som tar emot datan från formuläret som ett *CategoryDetail*-objekt vilket jag sedan kan skicka till metoden *InsertCategory()* i *CategoryMethods*-klassen som lagrar den nya kategorin om inget gick fel.

- *[HttpGet] EditCategory()*
  Här returnerar jag ett formulär för att redigera en kategori.
  För att fylla i formuläret med kategorins nuvarande data så skickar jag med ett kategori-objekt till vyn. Kategorin hämtas med metoden *SelectSingleCategory()* i *CategoryMethods*-klassen utifrån ett id som skickas med som en parameter. Detta id skickas med när vi klickar på redigera-länken:
  `@Url.Action("EditCategory", "Employee", new { id = item.Id })`.

  Vi kan sedan fånga upp det som en parameter i action-metoden:
  `public ActionResult EditCategory(int id)`.
  Och sedan skicka med det som en parameter i *SelectSingleCategory()*:
  `Category = cm.SelectSingleCategory(id, out string error);`.
  Vi skickar sedan med kategori-objektet till vyn:
  `return View(Category)`.

- *[HttpPost] EditCategory()*
  Hit skickas ”redigera kategori”-formuläret som tar emot datan från formuläret som ett *CategoryDetail*-objekt vilket jag sedan kan skicka till metoden *UpdateCategory()* i *CategoryMethods*-klassen som uppdaterar den nya kategorin om inget gick fel.

- *DeleteCategory()*
  När användaren klickar på radera-länken så omdirigeras den till denna action-metod tillsammans med ett id för en kategori:
  `@Url.Action("DeleteCategory", "Employee", new { id = item.Id }`.
  Vi fångar upp detta id som en parameter i action-metoden och skickar sedan vidare det till *DeleteCategory()*-metoden i *CategoryMethods*-klassen som tar bort en kategori utifrån det inmatade id:t.

Hantera produkter:

- *[HttpGet] Products()*
  I denna action-metod skickar jag med en lista med alla produkter till vyn och returnerar den. Detta görs på samma sätt som i action-metoden *[HttpGet] Categories()*.

- *[HttpGet] CreateProduct()*
  Här returnerar jag ett formulär för att skapa en ny produkt. Formuläret skickas till action-metoden *[HttpPost] CreateProduct()*.

- *[HttpPost] CreateProduct()*
  Hit skickas ”skapa produkt”-formuläret som tar emot data från formuläret med 3 input-parametrar.
  Kategorins id som en integer, bild-filen med klassen *HttpPostedFileBase* och resterande egenskaper som ett *CreateProductDetail*-objekt.
  ```
  public ActionResult CreateProduct(CreateProductDetail cpd,
  HttpPostedFileBase file, int categoryId)
  ```
  .

  Jag börjar med att lägga till kategorins id i *ProductDetail*-objektet som finns i *cpd*-objektet:
  ```
  cpd.Product.CategoryId = categoryId;
  ```
  .

  Efter det lägger jag till bilden i samma objekt som ovan. Klassen *HttpPostedFileBase* ger åtkomst till bild-filen som laddades upp av i klienten [46]. Om det laddades upp en bild så skapar jag en ny instans av *MemoryStream*-klassen som representerar en sekvens av bytes som läser och skriver till datorns minne [47]. Som jag nämnde tidigare så är en bildfil en sekvens av bytes.
  Efter det skriver jag över filens bytes till instansen av *MemoryStream*-klassen(*ms*):
  ```
  file.InputStream.CopyTo(ms);
  ```
  .
  Detta görs med *InputStream*-egenskapen som hämtar ett stream-objekt som pekar på en uppladdad fil och förbereder för läsning av innehållet i filen [48]. Ett stream-objekt är ett objekt som består av en sekvens av bytes [49].
  Jag använder sedan *CopyTo()*-metoden som läser av alla bytes från den nuvarande streamen och skriver över de till en annan stream [50].
  Nu återstår det bara att lägga till bilden i *ProductDetail*-objektet:
  ```
  cpd.Product.ProductImage = ms.GetBuffer();
  ```
  .
  GetBuffer()-metoden returnerar en array av bytes från streamen och bilden lagras då som en byte-array.

  När jag nu har lagt till både kategorin och bilden till *ProductDetail*-objektet så skickar jag med det till *InsertProduct()*-metoden i *ProductMethods*-klassen som sparar den nya produkten i databasen:
  ```
  i = pm.InsertProduct(cpd.Product, out string error);
  ```
  .

- *[HttpGet] EditProduct()*
  Här returnerar jag ett formulär för att redigera en produkt. Det görs på samma sätt som i *EditCategory()*-action-metoden. Tyvärr så går det inte att fylla i *<input-type=”file”>*-elementet med den nuvarande bilden p.g.a. säkerhetsrisker, så användaren behöver ladda upp bilden på nytt.

- *[HttpPost] EditProduct()*
  Hit skickas ”Redigera produkt”-formuläret. Det fungerar på samma sätt som *CreateProduct()*-action-metoden fast produkten uppdateras i databasen med metoden *UpdateProduct()*.

- *DeleteProduct()*
  Fungerar på samma sätt som *DeleteCategory()* fast denna action-metod tar bort en produkt istället för en kategori.

HomeController
Controller för att hämta produkter och kategorier. Filtrera efter kategori, Söka eter kategori och sortera efter pris.

- *[HttpGet] Index()*
  Denna action-metod genererar startsidan där jag skickar med ett *ViewModelPCC*-objekt till vyn som består av en lista med produkter och en lista med kategorier samt en enskild kategori.

- *[HttpPost] Index()*
  Hit skickas filtrerings- och sökformulären vars värden jag tar emot som parametrar i action-metoden:
  ```
  public ActionResult Index(string categoryId, string productName).
  ```
  categoryId kommer från listan i filtrerings-formuläret:
  ```
  <select class="form-control" id="categoryId" name="categoryId">.
  ```
  productName kommer från input-fältet i sökformuläret:
  ```
  <input type="text" name="productName" id="productName".
  ```
  categoryId skickar jag till *SelectProductsByCategory()*-metoden och productName skickar jag till *SelectProductsByName()*-metoden i klassen *ProductMethods*.
  För att man ska kunna växla mellan att filtrera och söka utan problem så har jag skapat if-satser som hanterar olika scenarion. Jag skickar sedan med *ViewModelPCC*-objektet till vyn som innehåller en lista med kategorier och en lista med produkter som kan se annorlunda ut beroende på vad användaren har använt för filtreringsmetod. Har användaren filtrerat efter kategori så skickas även ett objekt med en enskild kategori till vyn.

- *[HttpGet] SpecificProduct()*
  Här skriver jag ut information om en specifik produkt. Produkten hämtas med metoden *SelectSingleProduct()* i *ProductMethods*-klassen utifrån ett id som skickas med som en parameter. Detta id skickas med när vi klickar på ”läs mer”-länken i *index.cshtml*:
  ```
  <a href="@Url.Action("SpecificProduct", "Home", new { productId =
  products.Id })">Läs mer</a>.
  ```
  Jag skickar sedan med det returnerade produkt-objektet till vyn.

- *[HttpGet] ShoppingCart()*
  Här returnerar jag bara vyn för kundvagnen(*ShoppingCart.cshtml*).

- *[HttpPost] ShoppingCart()*
  När besökaren trycker på ”köp” i *SpecificProduct.cshtml*-vyn så skickas ett produkt-objekt till denna vy som tas emot som en parameter:
  `public ActionResult ShoppingCart(ProductDetail pd)`.

  Om kundvagnen är tom, dvs. om sessionsvariabeln ”ShoppingList” har värdet null, så skapar jag en ny lista och lägger till det medskickade produkt-objektet till listan. Jag lagrar sedan produkt-objektet i ovannämnda sessionsvariabel (se Figur 12)

```
if(Session["ShoppingList"] == null)
{
    List<ProductDetail> productList = new List<ProductDetail>
    {
        new ProductDetail { Id = pd.Id, ProductName = pd.ProductName, ProductPrice = pd.ProductPrice }
    };

    Session["ShoppingList"] = productList;
}
else
{
    var productList = (List<ProductDetail>)Session["ShoppingList"];
    productList.Add(new ProductDetail { Id = pd.Id, ProductName = pd.ProductName, ProductPrice = pd.ProductPrice });
    Session["ShoppingList"] = productList;
}
```

  Figur 12.

  Om kundvagnen inte är tom så hämtar jag sessionsvariabeln som en lista och lägger till det nya objektet för att till sist lagra den uppdaterade listan i en ny sessionsvariabel med samma namn (se Figur 12 ovan).

  För att skapa sessioner så använder jag *HttpContect.Session*-egenskapen[54] som gör det möjligt att komma åt egenskaper och metoder i *HttpSessionState*-klassen[55]. Med hjälp av sessioner kan jag lagra data i variabler som inte förstörs om besökaren skickar ett nytt anrop till servern, till exempel om den navigerar till en annan undersida på webbplatsen. På så sätt hålls datan vid liv oavsett vad användaren gör på webbplatsen. Datan lagras även separat för varje användare och är säker eftersom den lagras på servern [56].

- *ClearShoppingCart()*
  När besökaren klickar på ”rensa” i *ShoppingCart.cshtml*-vyn så anropas denna action-metod som tar bort sessionsvariabeln med *HttpSessionState.Remove()*-metoden[57]:
  `Session.Remove("ShoppingList");`.

Vyer

Account, Manage

Vyer för att hantera användare. Dessa controllers genereras automatiskt av identity-systemet och de ändringar jag gör är mest för översättningar och routes.

Employee
Vyer för att hantera produkter och kategorier.

- *Categories.cshtml*
  I denna vy skriver jag ut listan som skickades med från controllern i en tabell. I tabellen Skriver jag ut kategorinamn och två länkar för varje kategori: redigera- och ta bort-länkar. Längst upp börjar jag med att ange vilken modell jag vill jobba med i vyn:
  `@model Ienumerable<Webbshop.Models.CategoryDetail>`.

  När modellen är inkluderad så kan jag komma åt objektets egenskaper med Razor-syntax:
  `@Html.DisplayNameFor(model => model.CategoryName)`.

  Listan som skickades med från controllern kan jag sedan iterera igenom med en foreach-loop och skriva ut varje kategorinamn med Razor-syntax:
  `@Html.DisplayFor(modelItem => item.CategoryName)`.
  För rediga- och ta bort-länkarna så använder jag *Url.Action()*-metoden som tar 3 parametrar: namn på action-metod, namn på controller och ett routevärde som är kategorins löpnummer:
  `@Url.Action("EditCategory", "Employee", new { id = item.Id })`.

- *CreateCategory.cshtml*
  I denna vy skriver jag ut ett formulär utifrån *CategoryDetail*-modellen. Formuläret genereras av *Html.BeginForm()*-metoden som skickar ett POST-anrop till vyns action-metod [44].

- *EditCategory.cshtml*
  I denna vy skriver jag ut ett formulär för att redigera en befintlig kategori. Formuläret fylls i automatiskt vid inladdning med befintlig data som skickades med ett objekt från controllern.
  Formuläret skickas sedan som ett POST-anrop till vyns action-metod på samma sätt som i *CreateCategory*-vyn ovan.

- *Products.cshtml*
  I denna vy skriver jag ut listan med produkter som skickades med från controllern i en tabell.
  Jag skriver ut bilden på följande vis:
  ```
  <img alt="@item.ProductImageDescription"
  src="data:image/jpg;base64,@Convert.ToBase64String(item.ProductImage)"
  />.
  ```
  Här bäddar jag in bild-data direkt i dokumentet med Data URLs som består av fyra delar: ett prefix(data:), en MIME-typ som indikerar vilken typ av data(jpg-filer), Base64-token och själva datan(bilden):
  data:[<mediatyp>][;base64],<data>. [51]
  Jag använder metoden *Convert.ToBase64String()* för att konvertera byte-arrayen(bilden) till en sträng med Base64-kodning. Base64 är en metod för att konvertera binär data till en sträng med 7-bitars ASCII-tecken som kan skickas över olika medier som till exempel webben, utan att korrumpera data.
  Detta är viktigt att göra eftersom vissa former av kommunikationsmetoder(t.ex. HTML och HTTP) har tecken med speciella betydelser vilket kan leda till att kommunikationsmetoden tolkar den binära datan på fel sätt. Därför konverterar man den binära datan till ASCII-teckenkodning som är kompatibel med de flesta former av kommunikationsmetoder [52].

- *CreateProduct.cshtml*
  I denna vy skriver jag ut ett formulär utifrån *CreateProductDetail*-modellen som innehåller ett produkt-objekt en lista med kategori-objekt. Formuläret genereras av *Html.BeginForm()*-metoden som skickar ett POST-anrop till vyns action-metod [44].
  Jag skriver ut kategorierna i en select-meny där den valda kategorins id skickas med från <option>-elementets värde:
  ```
  <option value="@category.Id">@category.CategoryName</option>.
  ```
  För att ladda upp produktbilden så använder jag ett *<input type=”file”>*-element som genereras med Razor-syntax:
  ```
  @Html.TextBoxFor(model => model.Product.ProductImage, new { type =
  "file" }).
  ```
  För att detta ska fungera så måste jag även specificera hur data som formar bodyn i anropet ska kodas. Det görs med enctype-attributet i formuläret:
  ```
  @using (Html.BeginForm(new { enctype = "multipart/form-data" })).
  ```
  Eftersom jag använder ett *<input type=”file”>*-element så kodar jag datan med *multipart/form-data* vilket är värdet som måste användas när man laddar upp filer [45].

- *EditProduct.cshtml*
  Denna vy är likadan som *CreatProduct.cshtml*.

Home
Vyer för kunder med produkter, filtrering, produktdetaljer och kundvagn.

- *Index.cshtml*
  Här skriver jag ut en lista med alla produkter och filtreringsformulär.

- *Specificproduct.cshtml*
  Här skriver jag ut information om en specifik produkt. Jag skickar även med data om produkten som skrivs ut som meta-taggar i *_Layout.cshtml*.
  Jag skriver ut meta-taggarna enligt Open Graph Protocol. Detta protokoll är en uppsättning av meta-taggar som ger detaljerad information om webbplatsen till några av de största sociala nätverksplattformerna som Facebook, Google+, Twitter, Linkedin osv [53].
  Dessa taggar innehåller metadata om: Titel, Typ av objekt(t.ex. Webbsida), Url till specifik sida på webbplatsen, Url till Bild, Nyckelord, Beskrivning.

  Datan hämtas från produkt-objektet som skickades med från controllern och skickas sedan vidare med ViewData till *_Layout.cshtml*, ex på en meta-tagg med keywords:
  ```
  ViewData["Keywords"] = Model.ProductKeywords;
  ```
  .
  I *_Layout.cshtml* skriver jag sedan ut den som en meta-tagg:
  ```
  <meta name="keywords" content="@ViewBag.Keywords" />
  ```
  .

  Exempel på en Open Graph meta-tagg:
  ```
  <meta property="og:title" content="@ViewBag.Title" />
  ```
  .

  Jag skriver även ut ett formulär för att köpa produkten. Detta formulär skickar med ett *ProductDetail*-objekt med den valda produkten till action-metoden *ShoppingCart()*.
  ```
  @using (Html.BeginForm("ShoppingCart", "Home", FormMethod.Post)
  ```
  .
  Objektets värden skickas med från gömda input-fält:
  ```
  @Html.HiddenFor(model => model.ProductName)
  ```
  .

- *ShoppingCart.cshtml*
  Här skriver jag ut innehållet i kundvagnen vilket är en lista med produkter som kommer från sessions-variabeln ”ShoppingList” som genereras i controllern. Jag loopar igenom sessionsvariabeln och skriver ut de produkter som den innehåller med hjälp av en foreach-loop:
  ```
  foreach (Webbshop.Models.ProductDetail p in
  (List<Webbshop.Models.ProductDetail>)Session["ShoppingList"]).
  ```

  Jag skriver även ut två knappar, en för att betala och en för att rensa kundvagnen. Eftersom jag inte har någon betalningsfunktionalitet så rensar båda knapparna kundvagnen med hjälp av action-metoden *ClearShoppingCart()*:
  ```
  <a href="@Url.Action("ClearShoppingCart", "Home")">.
  ```

5 Resultat

Jag har skapat en webbshop som är indelad i två delar. En del där kunder kan navigera genom produkter, logga in och lägga till / ta bort produkter i en varukorg. En del där anställda kan logga in och skapa, uppdatera och radera produkter / kategorier.
Webbshoppen är en databasdriven webbapplikation baserad på ASP.NET MVC. Jag har planerat databasens struktur med ER- och tabelldiagram och implementerat den i Microsoft SQL Server. I Model-delen har jag skapat ett databaslager som kommunicerar med databasen. Jag har även skapat vyer för det visuella i applikationen och controllers som binder ihop modellerna och vyerna.

Jag har använt mig av sessioner och ViewBags för att skicka data mellan vyer och controllers. Jag har även dynamiskt genererat meta-taggar enligt Open Graph Protocol till varje produktsida som presenteras som unika webbsidor.

Den genererade HTML- och CSS-koden validerar korrekt enligt W3C:s rekommendationer. Även inmatningsformulär är skapade i enlighet med WCAG och WAI:s rekommendationer.

Jag har skapat ett inloggningssystem baserat på ASP.NET Identity som med lite konfigurering hanterar användare på ett säkert vis med validering av lösenord, kryptering av lösenord, användarroller osv.
Jag har gjort det möjligt att ladda upp bilder till servern och även att ladda ner de.

6 Slutsatser

Jag har inte hunnit göra allt jag tänkte innan deadline så projektet fick inte all den funktionalitet jag ville att den skulle ha och jag fick inte heller tid över till att finslipa webbplatsen och göra designen bättre. Trots detta så fick jag ändå med de nödvändigaste bitarna och lite till. Applikationen fyller dess funktion och är behaglig att använda men det finns definitivt delar att förbättra och lägga till som skulle göra applikationen mer komplett.

Jag skulle bland annat kunna lägga till sortering efter pris för produkter och även lägga till filtrering i gränssnittet för anställda. Jag skulle också vilja jobba vidare på Identity-systemet med epost-bekräftelse, användarroller vid registrering osv. Nu har jag endast hårdkodat in en administratör-roll i databasen som kan registrera nya användare.

Jag vill också jobba vidare med kundvagnen så att kunder kan ta bort enskilda produkter och ange ett antal av varje produkt. Sedan skulle jag vilja lägga till en funktion för betalning.

Arbetsprocessen har varit lite annorlunda än i tidigare projekt då jag inte riktigt vetat hur jag ska gå tillväga. Jag har testat olika tillvägagångssätt och inte varit helt säker på vad som skulle fungera så det har varit svårt att planera och göra en tydlig struktur men ju längre projektet gick ju mer klarnade allting. Jag har också haft brist på tid p.g.a. diverse anledningar så jag har fått omprioritera delar av projektet för att få med det väsentliga.

Sammanfattningsvis så är jag nöjd med arbetet men det finns saker att jobba vidare på och det finns mer att lära.

Källförteckning

[1] Affärsvärlden, ”Nordisk e-handel ökar” https://www.affarsvarlden.se/bors-ekonominyheter/nordisk-e-handel-okar-kraftigt-6932830 Publicerad 2018-09-27. Hämtad 2019-03-05

[2] Postnord, ”E-handeln fortsätter att gasa – och AI kommer att påverka utvecklingen framöver” https://www.postnord.com/sv/media/pressmeddelanden/postnord-sverige/2018/e-handeln-fortsatter-att-gasa-och-ai-kommer-att-paverka-utvecklingen-framover/ Publicerad 2018-09-13. Hämtad 2019-03-05

[3] Databasteknik, ”Relationsmodellen” http://www.databasteknik.se/webbkursen/relationer/index.html Publicerad 2005-07-18. Hämtad 2019-03-05

[4] SearchSQLServer, ”Microsoft SQL Server” https://searchsqlserver.techtarget.com/definition/SQL-Server Publicerad 2017-08. Hämtad 2019-03-05

[5] DotNet mvc Core, ”Teoretisk MVC” http://www2.tfe.umu.se/systemteknik/Webbteknik/DoW/MVCintro/MVC-teori-1/MVC-teori-1.html Publicerad 2018-08. Hämtad 2019-03-05

[6] Wikipedia, ”ASP.NET” https://en.wikipedia.org/wiki/ASP.NET Publicerad 2019-02-24. Hämtad 2019-03-05

[7] Wakefly, ”What is ASP.NET and Why Should I Use It?” https://www.wakefly.com/blog/what-is-asp-net-and-why-should-i-use-it/ Publicerad 2018-11-27. Hämtad 2019-03-05

[8] Microsoft, ”Choosing between .NET Core and .NET Framework for server apps” https://docs.microsoft.com/en-us/dotnet/standard/choosing-core-framework-server Hämtad 2019-03-05

[9] Databasteknik, ”Introduktion till frågespråket SQL” http://www.databasteknik.se/webbkursen/sql/index.html Hämtad 2019-03-05

[10] Microsoft, ”An introduction to NuGet” https://docs.microsoft.com/en-us/nuget/what-is-nuget Hämtad 2019-03-11

[11] Microsoft, ”Introduction to ASP.NET Identity” https://docs.microsoft.com/en-us/aspnet/identity/overview/getting-started/introduction-to-aspnet-identity Publicerad 2019-01-22. Hämtad 2018-03-11

[12] Microsoft, ”Microsoft.AspNet.Identity Namespace” https://docs.microsoft.com/en-us/previous-versions/aspnet/dn253016(v=vs.108) Publicerad 2015-10-27. Hämtad 2018-03-11

[13] Microsoft, ”UserValidator<TUser, Tkey> Class” https://docs.microsoft.com/en-us/previous-versions/aspnet/dn613288(v%3Dvs.108) Publicerad 2015-10-27. Hämtad 2018-03-11

[14] Microsoft, ”PasswordValidator Class” https://docs.microsoft.com/en-us/previous-versions/aspnet/dn613295(v%3Dvs.108) Publicerad 2015-10-27. Hämtad 2018-03-11

[15] Wikipedia, ”Cain and Abel (software)” https://en.wikipedia.org/wiki/Cain_and_Abel_(software) Publicerad 2018-11-10. Hämtad 2018-03-11

[16] Wikipedia, ”Dictionary attack” https://en.wikipedia.org/wiki/Dictionary_attack Publicerad 2019-02-28. Hämtad 2018-03-11

[17] Microsoft, ”ASP.NET MVC Routing Overview (C#)” https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/controllers-and-routing/asp-net-mvc-routing-overview-cs Publicerad 2008-08-19. Hämtad 2018-03-11

[18] TutorialsTeacher, ”Partial View” http://www.tutorialsteacher.com/mvc/partial-view-in-asp.net-mvc Hämtad 2019-03-11

[19] Microsoft, ”AuthorizeAttribute Class” https://docs.microsoft.com/en-us/dotnet/api/system.web.mvc.authorizeattribute?view=aspnet-mvc-5.2 Hämtad 2019-03-11

[20] Microsoft, ”HttpRequest.IsAuthenticated Property” https://docs.microsoft.com/en-us/dotnet/api/system.web.httprequest.isauthenticated?view=netframework-4.7.2 Hämtad 2019-03-11

[21] Microsoft, ”User.IsInRole Method” https://docs.microsoft.com/en-us/dotnet/api/microsoft.visualbasic.applicationservices.user.isinrole?view=netframework-4.7.2 Hämtad 2019-03-12

[22] Microsoft, ”nchar and nvarchar (Transact-SQL)” https://docs.microsoft.com/en-us/sql/t-sql/data-types/nchar-and-nvarchar-transact-sql?view=sql-server-2017 Publicerad 2018-10-22. Hämtad 2019-03-12

[23] Microsoft, ”CREATE TABLE (Transact-SQL) IDENTITY (Property)” https://docs.microsoft.com/en-us/sql/t-sql/statements/create-table-transact-sql-identity-property?view=sql-server-2017 Publicerad 2017-03-14. Hämtad 2019-03-12

[24] W3Schools, ”SQL PRIMARY KEY Constraint” https://www.w3schools.com/sql/sql_primarykey.asp 2019-03-12

[25] W3Schools, ”SQL FOREIGN KEY Constraint” https://www.w3schools.com/sql/sql_foreignkey.asp Hämtad 2019-03-12

[26] Microsoft, ”bit (Transact-SQL)” https://docs.microsoft.com/en-us/sql/t-sql/data-types/bit-transact-sql?view=sql-server-2017 Publicerad 2017-07-23. Hämtad 2019-03-13

[27] Microsoft, ”decimal and numeric (Transact-SQL)” https://docs.microsoft.com/en-us/sql/t-sql/data-types/decimal-and-numeric-transact-sql?view=sql-server-2017 Publicerad 2017-07-23. Hämtad 2019-03-13

[28] Microsoft, ”Unique Constraints and Check Constraints” https://docs.microsoft.com/en-us/sql/relational-databases/tables/unique-constraints-and-check-constraints?view=sql-server-2017 Publicerad 2017-06-27. Hämtad 2019-03-13

[29] TeachWithIct, ”8-bit challenge – Binary Representation if Images” https://teachwithict.weebly.com/binary-representation-of-images.html Hämtad 2019-03-13

[30] Microsoft, ”binary and varbinary (Transact-SQL)” https://docs.microsoft.com/en-us/sql/t-sql/data-types/binary-and-varbinary-transact-sql?view=sql-server-2017 Publicerad 2017-08-16. Hämtad 2019-03-13

[31] Database.guide, ”GETDATE() Examples in SQL Server (T-SQL)” https://database.guide/getdate-examples-in-sql-server-t-sql/ Publicerad 2017-06-15. Hämtad 2019-03-13

[32] Microsoft, ”Adding validation to the Model (C#)” https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-aspnet-mvc3/cs/adding-validation-to-the-model Publicerad 2011-01-12. Hämtad 2019-03-13

[33] Microsoft, ”out parameter modifier (C# Reference)” https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/out-parameter-modifier Hämtad 2019-03-13

[34] Microsoft, ”SqlConnection Class” https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqlconnection?view=netframework-4.7.2 Hämtad 2019-03-13

[35] Microsoft, ”SqlConnection.ConnectionString Property” https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqlconnection.connectionstring?view=netframework-4.7.2 Hämtad 2019-03-13

[36] Microsoft, ”SqlCommand Class” https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqlcommand?view=netframework-4.7.2 Hämtad 2019-03-13

[37] Microsoft, ”SqlCommand.Parameters Property” https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqlcommand.parameters?view=netframework-4.7.2 Hämtad 2019-03-13

[38] Microsoft, ”SqlParameterCollection Class” https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqlparametercollection?view=netframework-4.7.2 Hämtad 2019-03-13

[39] Microsoft, ”SqlDbType Enum” https://docs.microsoft.com/en-us/dotnet/api/system.data.sqldbtype?view=netframework-4.7.2 Hämtad 2019-03-13

[40] Microsoft, ”SqlDataAdapter Class” https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqldataadapter?view=netframework-4.7.2 Hämtad 2019-03-13

[41] Microsoft, ”DataSet Class” https://docs.microsoft.com/en-us/dotnet/api/system.data.dataset?view=netframework-4.7.2 Hämtad 2019-03-13

[42] Microsoft, ”DbDataAdapter.Fill Method” https://docs.microsoft.com/en-us/dotnet/api/system.data.common.dbdataadapter.fill?view=netframework-4.7.2#System_Data_Common_DbDataAdapter_Fill_System_Data_DataSet_System_String_ Hämtad 2019-03-13

[43] Microsoft, ”SqlCommand.ExecuteNonQuery Method” https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqlcommand.executenonquery?view=netframework-4.7.2 Hämtad 2019-03-13

[44] Microsoft, ”FormExtensions.BeginForm Method” https://docs.microsoft.com/en-us/dotnet/api/system.web.mvc.html.formextensions.beginform?view=aspnet-mvc-5.2 Hämtad 2019-03-14

[45] DEV, ”Understanding HTML Form Encoding: URL Encoded and Multipart Forms” https://dev.to/sidthesloth92/understanding-html-form-encoding-url-encoded-and-multipart-forms-3lpa Hämtad 2019-03-14

[46] Microsoft, ”HttpPostedFileBase Class” https://docs.microsoft.com/en-us/dotnet/api/system.web.httppostedfilebase?view=netframework-4.7.2#definition Hämtad 2019-03-18

[47] Microsoft, ”MemoryStream Class” https://docs.microsoft.com/en-us/dotnet/api/system.io.memorystream?view=netframework-4.7.2 Hämtad 2019-03-18

[48] Microsoft, ”HttpPosteeFile.InputStream Property” https://docs.microsoft.com/en-us/dotnet/api/system.web.httppostedfile.inputstream?view=netframework-4.7.2 Hämtad 2019-03-18

[49] Microsoft, ”Stream Class” https://docs.microsoft.com/en-us/dotnet/api/system.io.stream?view=netframework-4.7.2 Hämtad 2019-03-18

[50] Microsoft, ”Stream.CopyTo Method” https://docs.microsoft.com/en-us/dotnet/api/system.io.stream.copyto?view=netframework-4.7.2 Hämtad 2019-03-18

[51] MDN web docs, ”Data URLs” https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Data_URIs Publicerad 2019-01-27. Hämtad 2019-03-18

[52] Quora, ”What is the purpose of base64 encoding, why not just use binary?” https://www.quora.com/What-is-the-purpose-of-base64-encoding-why-not-just-use-binary Publicerad 2017-07-20. Hämtad 2019-03-18

[53] ogp, ”The Open Graph Control” http://ogp.me/ Hämtad 2019-03-23

[54] Microsoft, ”HttpContext.Session Property” https://docs.microsoft.com/en-us/dotnet/api/system.web.httpcontext.session?view=netframework-4.7.2 Hämtad 2019-03-23

[55] Microsoft, ”HttpSessionState Class” https://docs.microsoft.com/en-us/dotnet/api/system.web.sessionstate.httpsessionstate?view=netframework-4.7.2 Hämtad 2019-03-23

[56] C# Corner, ”Introduction To ASP.NET Sessions” https://www.c-sharpcorner.com/UploadFile/225740/introduction-of-session-in-Asp-Net/ Publicerad 2018-11-23. Hämtad 2019-03-23

[57] Microsoft, ”HttpSessionState.Remove(String) Method” https://docs.microsoft.com/en-us/dotnet/api/system.web.sessionstate.httpsessionstate.remove?view=netframework-4.7.2 Hämtad 2019-03-23