Document: Backend Programming, fall 2016, Mandatory Assignment 2, MbmStore.

Students: Jens Christian Rasch, Brian Munksgaard

Indhold

Introduktion 1

ASP.NET MVC 1

- Arkitektur 1
- Folderstruktur 1
- Routes 2
- Controllers og child actions 3
- Layouts, views og partial views 3
- Views og ViewModels 4
- Helpers 5
  - Inline helpers 5
  - HTML Helpers 5
  - Custom Helpers 6

MBMStore 6

- Model 6
- Domain model 6
- Modellering af controllers og views 7
- Routes 8
- Controllers og child actions 9
- Layouts, views og partial views 10
- Views og ViewModels 12
- Helpers 13
  - Inline Helpers 13
  - HTML helpers 14
  - Custom helper 14

Introduktion

Opgaven i denne anden mandatory assignment var at lave en fungerende webshop til salg af bøger, musik og film ved at benytte ASP.NET MVC frameworket: Velkommen til MBMStore.

Formålet med denne rapport er at lave en gennemgang af de dele og konventioner der udgør ASP.NET MVC website, samt at beskrive hvordan vi har benyttet disse i udviklingen af MBMStore sitet.

Vi vil starte med at gennemgå de konventioner som ASP.NET MVC gør brug af og hvordan disse kan benyttes og rettes til efter behov. Vi vil derefter forsætte med at forklarer hvordan vi har benyttet disse konventioner samt give nogle eksempler på vores brug af disse.

ASP.NET MVC

ASP.NET MVC (herefter kaldet MVC) er MicroSofts implementering af Model-View-Controller frameworket.

Arkitektur

Den arkitektur som MVC bygger på er lavet for at hjælpe med at strukturere den kode der laves samt at understøtte DRY princippet. Den bygger på ”convention over configuraiton” hvilket betyder at den benytter konventioner for folderstruktur og navngivning af de forskellige komponenter.

Et MVC projekt består af 3 grundlæggende komponenter:

- Controllers
- Views
- Models

Models repræsentere de data der skal arbejdes med i projektet samt en evt. forretningslogik der måtte være.

Views er skabeloner til dynamisk at generere HTML’en til visning af sitets UI. De definere hvordan udseendet af sitet skal være og hvordan de forskellige elementer skal placeres. Et view kan enten være løs eller stærkt type baseret. At et view er strækt type baseret vil sige at det har en tæt kobling til en Model, som viewet kan benytte i layout og visning af data.

Controllers er koblingen mellem modellerne og views. Controllers står for at håndtere alle URL requests til sitet og for at hente data fra modellerne og sende relevante data videre til Views. Controllers kan også indeholde logik til at vælge mellem forskellige views der skal vises eller mapning mellem en Model og en ViewModel.

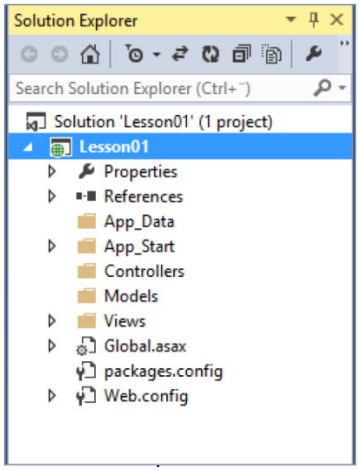
Vi vil komme mere ind på disse 3 komponenter i de følgende afsnit.

Folderstruktur

Et MVC projekt har per konvention en folderstruktur som man kan vælge at følge i sit projekt. Det er ikke en struktur man er tvunget til at benytte. Konventionen for folderstrukturen kan ses nedenfor, sammen med en forklaring på formålet med de enkelte foldere.

Folder	Formål
/Controllers	Her placeres alle controller klasser der håndtere URL requests.
/Models	Her placeres klasserne der håndtere data og forretningslogik.
/Views	Her placeres HTML skabelonerne der håndtere visning af UI for brugerne.
/App_Data	Her kan placeres data filer der skal læses og skrives.
/App_Start	I denne folder placeres konfigurations kode for ting som routing, bundling og Web API.
/Scripts	I denne folder placeres de JavaScript filer og scripts der benyttes i sitet.
/Content	Denne folder indeholder filer som css, billeder og andet indhold som benyttes af sitet.

Når man laver et nyt MVC projekt i Visual Studio (herefter VS) får man automatisk givet en folderstruktur som følge konventionen ovenfor (se figur 1). Som nævnt er det ikke en folderstruktur man er tvunget til at følge, men hvis man benytter denne er der mange ting i et MVC website, som man som udvikler ikke behøver tænke over eller tage stilling til.



Figur 1 - Standard folderstruktur for et MVC projekt

Routes

I folderen "App_Start" findes en fil kaldet "RouteConfig.cs". Denne fil indeholder konfiguration af, hvordan en URL request skal håndteres af MVC sitet. En URL til et MVC site følger som regel følgende opbygning: http://<server>/<Controller>/<Action>/<Id>. Hvordan denne URL håndteres kommer så an på, hvilke routes der er sat op i RouteConfig.cs.

I figur 2 nedenfor har vi vist et eksempel på hvordan en route konfiguration kan være sat op. Som angivet i "url" delen, så vil denne route kunne håndtere en URL der følger opbygningen nævnt ovenfor. Ifølge "defaults" delen, så er der angivet standard værdier for 2 dele af URL'en, nemlig controller og action delene, mens id-delen er angivet som Optional hvilket betyder at den kan udelades uden at denne route vil fejle.
Det betyder i praksis at denne route konfiguration både vil kunne håndtere kald som http://minserver/ og http://minserver/Customer/Index/2. Det første eksempel vil svarer til en URL ala http://minserver/Catalogue/Index.

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

Figur 2 - Eksempel på en route konfiguration

Controllers og child actions

Controllers er som tidligere beskrevet bindeleddet mellem Model og View i et MVC site. Det er i controlleren at data hentes og sammensættes inden de sendes videre til det korrekte View. I controlleren kan også forekomme forretningslogik der skal tages højde for inden data sendes til viewet.

Konventionen i ASP.NET MVC angiver at controllers navngives med klassenavnet efterfulgt af Controller, f.eks. CustomerController for en controller der håndtere Customers. Samtidig angiver konventionen at controllerne placeres i en folderen ”Controlleres”, som beskrevet i afsnittet ”Folderstruktur” ovenfor.

Controllers indeholder en række Actions som alle er de metoder der kan kaldes på controlleren. Actions returnere for det meste et ActionResult som i de fleste tilfælde er et View, men i teorien kan en Action returnere alt fra en simpel tekst streng til en kompleks model eller et JSON/XML objekt.

Overholdes konventionerne for Controllers medfører det at en controller vil kunne kaldes med en url ala http://minserver/Customer/Index, hvor Index action på CustomerController controlleren vil blive kaldt. Dette kunne resultere i at det tilhørende View vil blive vist for brugeren. Mere om Views i næste afsnit.

Udover Actions kan en controller også indeholde nogle child actions. Den store forskel mellem Actions og Child Actions er, at en child action er en Action som kan blive kaldt inde fra et specifikt punkt i et view og kun returnere den specifikke afgrænsede del af UI’et. Hvorimod en ”normal” Action på en controller som regel returnere et komplet View.
Child Actions er en god og nem måde at lave små widgets eller dele af et site, som så kan blive genbrugt flere steder og i flere forskellige views. Eksempler på sådan en child action kunne være en indkøbskurv oversigt, eller en breadcrumb sti på siderne eller visning af en menu.

Layouts, views og partial views

Et View er den visuelle del i et MVC site. Det er skabelonen for det HTML der vises for brugerne i browseren og er altså den del der afgør hvordan de data som Controlleren returnere, bliver vist for brugerne.

Alle views tilhører som udgangspunkt én controller og konventionen for Views angiver at hvert view navngives efter navnet på den tilsvarende metode i controlleren. Altså vil det tidligere nævnte view for Index action i Customer controlleren hedde Index.cshtml (.cshtml angiver at programmeringssproget der er benytte er C#). Konventionen angiver også at Views bliver placeret i folderen Views og herunder i en folder navngivet efter den tilhørende controller.

Som sagt er det ikke et krav at følge konventionen, men hvis man følger den kan man fra controller actionen nøjes med at skrive ”return View()” og MVC vil selv kunne finde frem til det korrekt view.

Hvis ikke konventionen benyttes, skal man i stedet sørge for at specificere hvilket view der skal returneres fra en Action i controlleren.

Et view kan også være et Partial View hvilket vil sige at det kun indeholder en del af det komplette UI. Det gør at et Partial View kan genbruges flere steder på et site og som udgangspunkt bliver et partial view returneret af en Child Action. Man kan altså sige at Partial Views er for Child Actions hvad Views er for Actions.

Et View kan enten indeholde alt markup/HTML for den side der skal vises i browseren, eller også kan der refereres til en Layout-fil der indeholder nogle generelle og gennemgående elementer som benyttes af flere Views. Hvorvidt et View benytter en layout fil eller ej kan angives i script-sektion i begyndelsen af filen (se figur XX nedenfor). Her angives enten null for ingen layout eller en relativ sti til den layout-fil der skal benyttes.

```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}
```

Figur 3 - Angivelse af Layout

Man kan også angive dette på et overordnet niveau og få det til at slå igennem på alle Views uden at skulle angive det på hvert enkelt view. Dette gøres ved at angive den samme script sektion som vist i figur 3, men gøre det i en fil kaldet "_ViewStart.cshtml" som placeres i roden af View folderen. Dermed vil dette slå igennem på alle view der laves.

Et layout indeholder som sagt noget standard HTML layout samt inkludering af fælles stylesheet og scripts. En del af den markup der kan benyttes i en Layout-fil er også kaldene RenderBody og RenderSection. Som udgangspunkt skal et layout have et kald til RenderBody og hvor dette kald er placeret vil indholdet af Viewet automatisk blive indsat.
Det er også muligt at referere til en eller flere sektioner ved at kalder RenderSection. Sektioner er en måde at kunne organisere hvor forskellige elementer placeres på siden. Hver kald til RenderSection kan angive om sektionen er krævet eller ej, altså om det underlæggende View skal sørge for at implementere de samme sektioner.

Som med Views og Controllers så er der også en konvention for placering af Layouts. Alle Layout filer er placeret under folderen Views/Shared og som standard så hedder default Layoutet "_Layout.cshtml". Som med alle andre konventioner i MVC, så er det heller ikke nødvendigt at følge den når det kommer til Layouts. Hvis man ikke benytter standard placeringen af ens layout filer, så skal behøver man blot rette Layout property'en til i ens Views eller i _ViewStart-filen.

Views og ViewModels

Som tidligere beskrevet så benyttes et View til at opbygge en visning af data der hentes fra en Model og sendes videre via en Action i en Controller. Et View kan derfor enten være løst eller stærkt koblet til de data der skal vises. Om et View er løst eller stærkt koblet betyder i bund og grund bare, om viewet har en vide om hvilke data det er der skal vises. Hvis denne viden er til stedet i viewet taler man om at det er et strongly typed view.
At sende data fra en Controller til et View kan gøres på flere måder. Når vi snakker om et løst koblet View, så kan man sende data som propertis på ViewBag eller ViewData objekterne. Snakker vi

derimod om et stærkt koblet View, så kan vi i stedet sende en Model med som parametre når vi kalder View'et til slut i vores Action.

Til tider kan der være tale om, at den model som Controlleren har benyttet til at hente data fra, kan indeholde en del flere informationer end det View'et har behov for, for at kunne vise data for brugeren, eller at de data et View har behov for kommer fra flere forskellige Modeller. Hvis det er tilfældet kan man vælge at oprette en såkaldt ViewModel, som så udelukkende indeholder de data der er relevant for det enkelte View.
Det er Controllerens opgave at sørge for at data der er behov for i en ViewModel bliver hentet fra andre modeller og kopieret ind i ViewModel'en, og efterfølgende bliver sendt til det relevante View, eller Partial View.

At benytte en ViewModel i stedet for en Model når man skal sende data fra Controlleren til View'et kan således være med til at formindske den mængde data der sendes til brugerens browser, samt at forenkle de information der sendes til Viewet. En anden fordel ved at benytte en ViewModel frem for en Model er, at alle relevante data er samlet i en enkelt ViewModel i stedet for at være spredt ud over flere Modeller.

Hvis er View benyttes til at vise simple data, så kan det være en fordel at benytte ViewBag/ViewData til at sende informationer til View'et, men er der tale om at der skal sendes mange informationer, så kan det godt betale sig at benytte sig af et stærkt koblet View.
Det er dog vigtigt at understrege, at det at benytte et stærkt koblet View og sende en Model eller ViewModel til selve Viewet på ingen måde udelukker at man også benytte ViewBag/ViewData. ViewBag/ViewData er f.eks. en god måde at sende paging relevant data fra Controller til View, eller data og informationer som ikke er en del af en Model eller ViewModel.

Helpers

Når man skal lave et View benytter langt de fleste udviklere sig af syntaksen kaldet Razor. Som en del af denne syntaks er, at man kan benytte sig af nogle helpers som gør udviklingen af et View en smule enklere. Disse helpers kommer i forskellige kategorier, men er opdelt i Inline helpers, indbyggede HTML Helpers og Custom Helpers. Vi vil her kort beskrive de 3 forskellige typer og hvad de benyttes til.

Fælles for alle typer af helpers er, at de kan eksistere i både en intern og en ekstern udgave. Det vil sige, at interne helpers alle er angivet og eksistere inde i et enkelt View. En ekstren helper er blevet lavet og eksistere i en eksterne fil i projektet og kan derfor genbruges af mere end ét View.

Vi har eksempler på hvordan man benytter disser helpers i kapitlet om MBMStore senere i denne rapport.

Inline helpers

Inline helpers er en helper man laver internt i et View og som kun kan benytte i det ene View hvor de er eksistere. Disse helpers kan både være løst eller stærkt koblet til de data der vises i Viewet. Angivelse af en Inline helper gøres ved at bruge @helper tag'et.

HTML Helpers

HTML helpers er simple funktioner som kan benyttes til at specificere hvilke HTML elementer der skal placeres på siden. Disse HTML helpers findes i to forskellige udgaver: Standard helpers og Strongly typed helpers.

Forskellen på Standard og Strongly typed helpers er, at med en Strongly typed helper er der en stærk kobling til det felt som helperen referere til. Hvor man vil angive en HTML helper som @Html.TextBox(…), så angiver man en Strongly typed helper ved @Html.TextBoxFor(m => …) hvor m-parameteren er en reference til den model der er givet med til View'et. At bruge Strongly typed helpers giver derfor også mest mening, hvis du har et stærkt koblet view.

En 3. type af HTML herlpers er Templated helpers. Disse benyttes til at lade MVC selv finde frem til hvilke HTML elementer der skal benyttes for at generer en side, alt samme baseret på den Model der er givet til Viewet og de egenskaber der er angivet på denne Model.
Som med HTML helpers, så findes Templated helpers også både i en standard og strongly typed udgave og forskellen mellem disse er nøjagtig magen til den hos HTML helpers.

Custom Helpers

Udover Inline helpers og indbyggede HTML helpers, så er det også muligt at lave sine egen udviklede helpers der kan benyttes i de forskellige Views. Dette gøres ved at lave en extension metode for HtmlHelper klassen eller ved at lave en statisk metode i en utility klasse.
Ved at benytte en utility klasse kan man sikre at ens Custom helper kan genbruges i forskellige Views rundt om i sitet, i stedet for udelukkende at kunne benyttes i ét View.

MBMStore

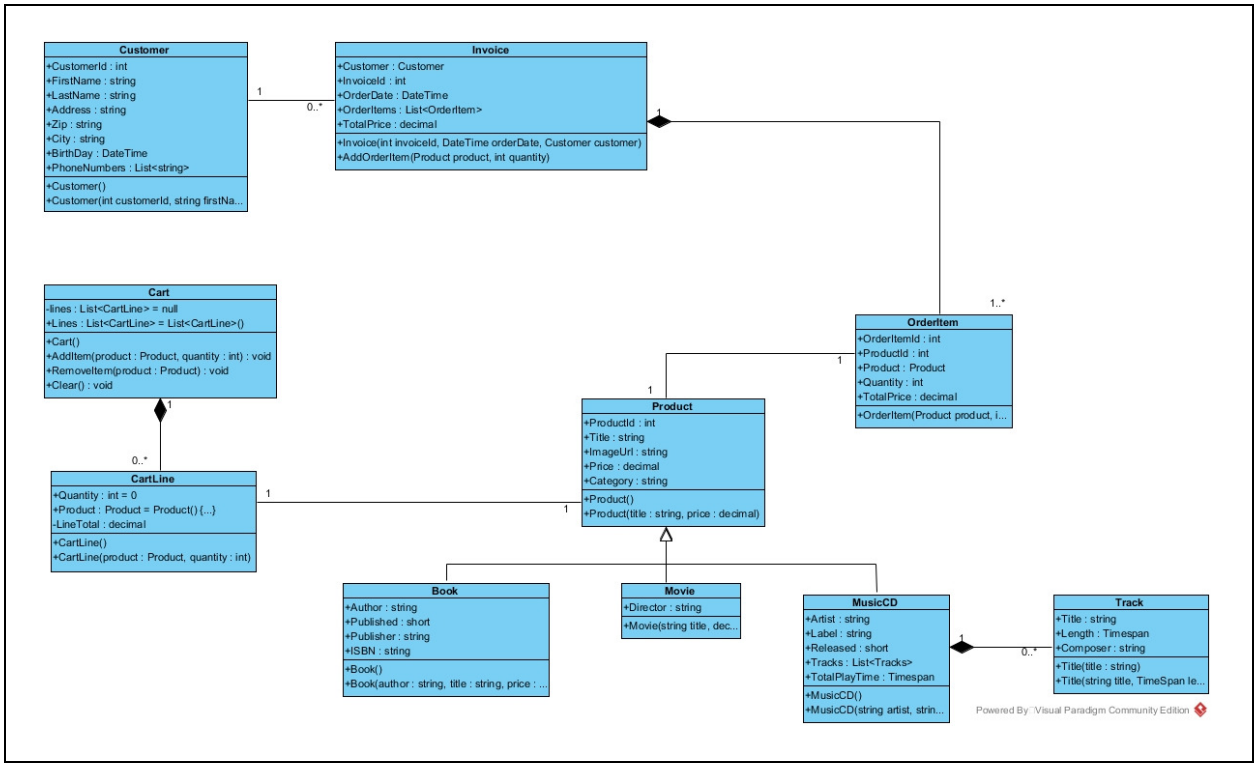
I de følgende afsnit vil vi forsøge at give et indblik i, hvordan vi har brugt de konventioner og elementer som MVC består af, i vores løsningen af opgaven med at lave en webshop til salg af bøger, musik og film: MBMStore.
I vores gennemgang vil vi komme med eksempler på konkrete implementationer vi har lavet i forhold til de elementer der er beskrevet i rapportens forrige kapitel.

For overskuelighedens skyld har vi også vedlagt vores Visual Studio solution, så der kan refereres til denne for at se yderligere implementering af elementerne.

Model

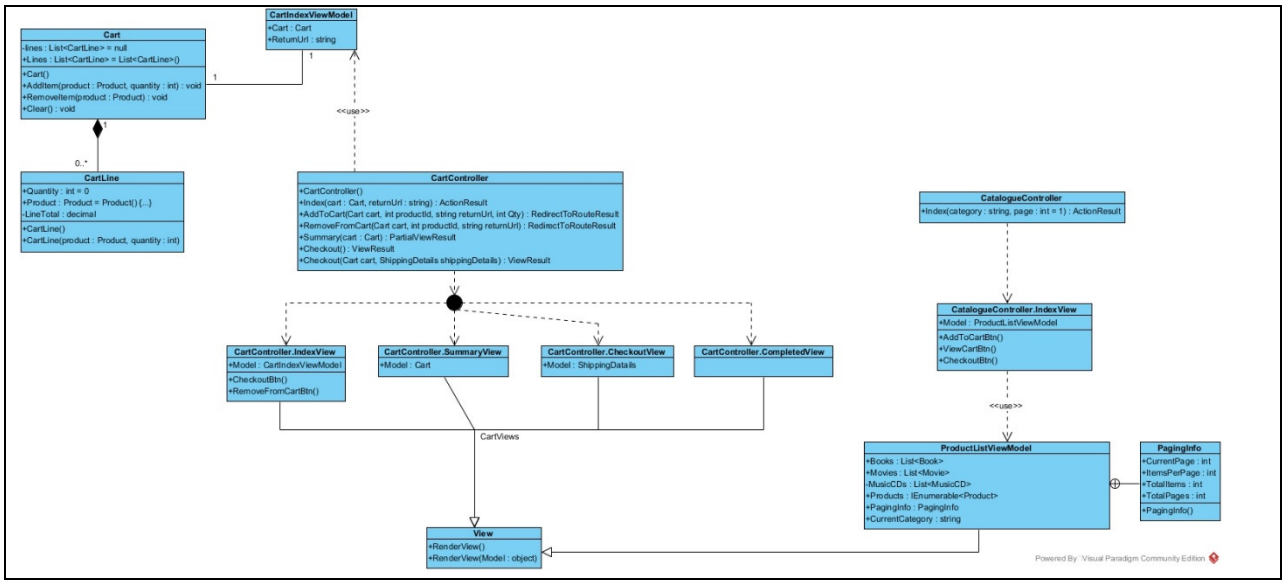
Domain model

Domænemodellen for vores webshop kan ses af nedenstående UML diagram



Modellering af controllers og views

Vi har valgt at modellere controllers og views som vist herunder:

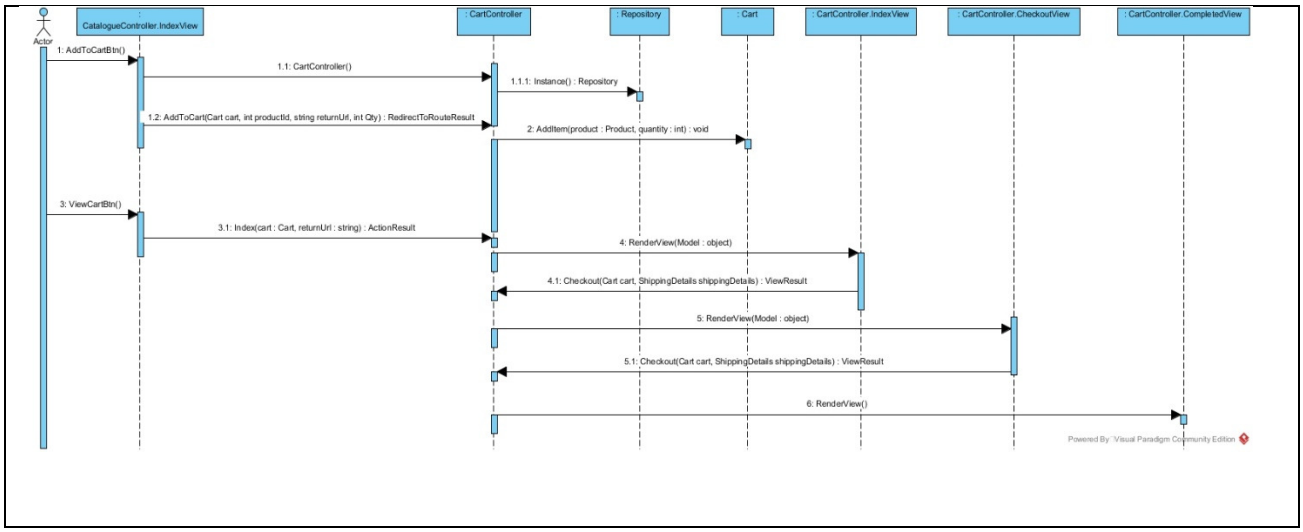


Som det ses, har vi lavet:

- En super klasse View med metoderne RenderView og RenderView(Model: object)
- View klasser, hvor buttons/links etc. er vist som operations/metoder. View data er vist som properties.
- En controller som en helt almindelig klasse.

Vi har så mulighed for at vise hvilke controllers der anvender hvilke views.

Ovenstående giver os mulighed for at lave følgende sekvensdiagram:



Routes

I forbindelse med løsningen af de forskellige opgaver har vi gjort brug af mulighederne for at sætter forskellige Routes op i sitet. Dette er gjort for at gøre navigeringen af shoppen nemmere for brugerne og for at udnytte nogle af de muligheder MVC giver os.

I tabellen nedenfor har vi angivet de forskellige Routes som vi har sat op i sitet. Alle disse kan ses i filen RouteConfig.cs som kan findes i folderen App_Start, som jo per konvention er den folder man placere routing og konfigurations filer i.

Route	Beskrivelse

routes.MapRoute(
name: null,
url: "{controller}",
defaults: new
{
controller = "Catalogue",
action = "Index",
category = (string)null, page = 1
}
);
	Denne route vil blive aktiveret hvis sitet kaldes med en url ala http://minserver/Customer eller http://minserver/.
	Hvis ingen controller er angivet, vil sitet automatisk vise side 1 af Index action metoden på Catalogue controlleren og uden at filtrer på hvilken produkt kategori der vises.

routes.MapRoute(
name: null,
url: "{controller}/Page{page}",
defaults: new
{
controller = "Catalogue",
action = "Index",
category = (string)null
},
constraints: new
{
page = @"\d+"
}
);
	Denne route viser på lige fod med den forrige en oversight over produkter på sitet. Forskellen på disse to routes er dog, at denne kan håndtere paging af resultatet fra Controlleren.
	Der er sat en constraint op på denne route som gør, at værdien for page-parameteren udelukkende kan være et tal.

routes.MapRoute(name: null, url: "Catalogue/{category}", defaults: new { controller = "Catalogue", action = "Index", page = 1	Denne route viser udelukkende Index metoden på Catalogue controlleren og med en kategori der er angivet i url'en. F.eks. http://minserver/Catalogue/Books hvor "Books" er produkt kategorien.

});	

routes.MapRoute(
name: null,
url:
"{controller}/{category}/Page{page}",
defaults: new
{
controller = "Catalogue",
action = "Index"
},
constraints: new
{
page = @"\d+"
}
);
	Som med den forrige route, men med den ekstra feature at denne route kan håndtere paging I de resultater som Controlleren leverer.
	På samme måde som nævnt for den anden route i denne liste, så er der sat en constraint op som sikre, at værdien af ”page” parameteren er et tal værdi.

routes.MapRoute(
name: "Default",
url: "{controller}/{action}/{id}",
defaults: new
{
controller = "Catalogue",
action = "Index",
id = UrlParameter.Optional
}
);
	Dette er default routen. Det er denne route der vil blive faldt tilbage på hvis ingen af de andre routes passer med det brugeren har skrevet i adresse feltet i browseren.

Fælles for de Routes vi har beskrevet i ovenstående tabel er, at de alle har defineret nogle default værdier i tilfælde af at disse dele af url’en ikke er blevet udfyldt af brugerne.

Rækkefølgen af vores routes er heller ikke tilfældig, da de forskellige routes der er sat op i RouteConfig.cs alle læses i den rækkefølge de står, og første match til den indkomne url, er den route der bliver benyttet til håndtering af det indkomne kald.

Controllers og child actions

I vores MBMStore har vi 2 steder valgt at benytte et partial view til at vise nogle generelle dele af sitet, som ikke har en direkte tilknytning til én enkelt side. Disse to partial views er vores menu samt vores Indkøbskurv oversigt. Begge disse partial views bliver håndteret af nogle child actions. Child action for menu’en bliver håndteret i Menu child action i NAV controlleren, og for indkøbskurven er det Summary child action i Cart controlleren.

Fælles for begge disse child actions er også, at de bliver kaldt fra vores _Layout.cshtml-fil. Dette gør at disse to elementer vil bliver vist på hver eneste side på vores site, også selvom resten af Viewet henter dens data fra en anden controller og model.

Kaldet til menu’en ser ud som vist i kasserne nedenfor. Den første viser controllerens child action, hvor vi via en Linq query finder frem til de produkt kategorier der findes i systemet og derefter sender vi dem videre som input til et strongly typed partial view.

```
public PartialViewResult Menu(string category = null)
{
  ViewBag.SelectedCategory = category;

  IEnumerable<string> categories = Repository.Instance.Products
    .Select(x => x.Category)
    .Distinct()
    .OrderBy(x => x);
  return PartialView(categories);
```

```
}
```

Nedenfor kan man se hvordan dette strongly typed partial view ser ud. Som det kan ses, så har vi angivet modellen som værende en IEnumerable af typen string.

```
@model IEnumerable<string>

@Html.ActionLink("Home", "Index", "Catalogue", null, new { @class = "btn btn-block btn-default btn-lg" })

@foreach (var link in Model)
{
   @Html.RouteLink(link, new
{
   controller = "Catalogue",
   action = "Index",
   category = link,
   page = 1 }, new
   {
      @class = "btn btn-block btn-default btn-lg" + (link == ViewBag.SelectedCategory ? " btn-primary" : "")
   })
}
```

Vores kald til visning af totalen på de produkter der er lagt i indkøbskurven er lidt mere enkel. Igen har vi et strongly typed partial view, men her er modellen en instance af en Cart klasse og her kalder summere vi Quantity på alle liner samt viser det totale beløb for varerne i kurven, ved at kalde metoden TotalPrice() på modellen, som her jo er en instans af Cart klassen.

```
@model MbmStore.ViewModels.Cart

<div class="navbar-right">
   @Html.ActionLink("Checkout", "Index", "Cart",
   new { returnUrl = Request.Url.PathAndQuery },
   new { @class = "btn btn-default navbar-btn" })
</div>

<div class="navbar-text navbar-right">
   <b>Your cart:</b>
   @Model.Lines.Sum(x => x.Quantity) item(s),
   @Model.TotalPrice.ToString("n")
</div>
```

Layouts, views og partial views

Alle Views i vores løsning benytter sig af den samme fælles layout fil (_Layout.cshtml). I stedet for at benytte os af konventionerne i MVC og angive layout filen i _ViewStart.cshtml, har vi specificeret layout filen i de enkelte Views.

I _Layout.cshtml (se nedenfor) har vi, som nævnt tidligere, angivet kaldet til de 2 partial views der viser indkøbskurv overblikket og menu'en. Begge disse dele går igen på alle sider på sitet, og derfor giver det mening at have dem med i layout filen i stedet for at skulle angive det på hvert enkelt View. Vi har angivet de enkelte partial views ved at benytte en standard HTML helper @Html.Action() som kalder den controller action der er specificeret ved input parametrene.

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8" />
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@ViewBag.Title</title>
    <link href="~/Content/Site.css" rel="stylesheet" type="text/css" />
    <link href="~/Content/bootstrap.css" rel="stylesheet" type="text/css" />
    <link href="~/Content/bootstrap-theme.css" rel="stylesheet" type="text/css"/>

    <script src="~/Scripts/modernizr-2.6.2.js"></script>
  </head>
  <body class="container">
    <div class="navbar navbar-inverse" role="navigation">
      <a class="navbar-brand" href="#">Music Book Movie Store</a>
      @Html.Action("Summary", "Cart")
    </div>

    <div class="row panel">
      <div id="categories" class="col-xs-3">
        @Html.Action("Menu", "Nav")
      </div>

      <div class="col-xs-9">
        @RenderBody()
      </div>
    </div>

    <footer>
      <p>&copy; @DateTime.Now.Year - The Move Book MusicCD Store</p>
    </footer>

    <script src="~/Scripts/jquery-1.10.2.min.js"></script>
    <script src="~/Scripts/bootstrap.min.js"></script>
  </body>
</html>
```

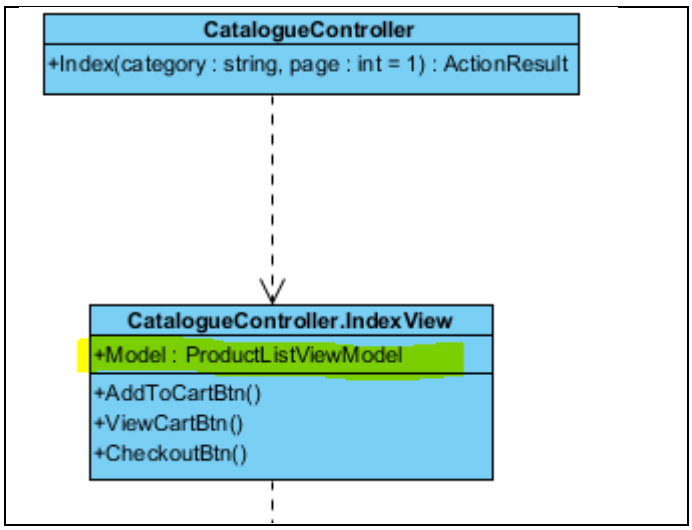
Udover de tidliger nævnte partial views gør vi også brug af nogle partial views for at vise de enkelte produkt kategorier på katalog siden. Vi har lavet et partival view for hver af de tre produkt kategorier og i kassen nedenfor kan man se hvordan vi har implementeret det partial view der viser bøgerne.

```
@model MbmStore.Models.Book

<div class="row">
  <div class="col-md-2">
    <img src="/Content/Images/@Model.ImageUrl" alt="@Model.Title" style="width:100px" />
  </div>
  <div class="col-md-4">
    <strong>Title:</strong> @Model.Title<br />
    <strong>Author:</strong> @Model.Author<br />
    <strong>Price:</strong> @Model.Price<br />
    <strong>Publisher:</strong> @Model.Publisher (@Model.Published)<br />
    <strong>ISBN:</strong> @Model.ISBN<br /> <br />
  </div>
  <div class="col-md-6">
    @using (Html.BeginForm("AddToCart", "Cart"))
    {
      @Html.Hidden("ProductId", Model.ProductId)
      @Html.Hidden("returnUrl", Request.Url.PathAndQuery)
      <div>
        @Html.DropDownList("Qty", new SelectList(Enumerable.Range(1, 20)))
      </div>
      <div>
        <input type="submit" class="btn btn-success" value="Add to cart" />
```

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

Som vi kan er ovenstående View strongly typed (@model …). Dette viser vi også i vores diagrammer ved at have en Model property af den relevante type:



Views og ViewModels

Der er nogle enkelte steder i vores webshop hvor vi har benyttet os af ViewModels. For eksempel har vi ved implementeringen af Index view for CartControlleren lavet en CartIndexViewModel (se nedenfor). Årsagen til dette er, at vores Cart view både skal have information om den indkøbskurv der skal vises, men også om en url for at komme tilbage fra indkøbskurven igen. Det kunne være gjort via brug af ViewBag objektet, men vi valgte i stedet denne løsning

```
public class CartIndexViewModel
{
    /// <summary>
    /// The current cart.
    /// </summary>
    public Cart Cart { get; set; }

    /// <summary>
    /// The URL to return to when continuing shopping.
    /// </summary>
    public string ReturnUrl { get; set; }
}
```

Et andet sted vi har brugt en ViewModel er i forbindelse med Paging ved visning af produkter. Denne ViewModel indeholder information der er relevant for at kunne lave en ordentlig paging. ViewModellen indeholder også en metode der returnere det totale antal sider baseret på oplysningerne i de andre properties.

```
public class PagingInfo
{
    /// <summary>
    /// The current page.
    /// </summary>
```

```
    public int CurrentPage { get; set; }

    /// <summary>
    /// The number of items per page.
    /// </summary>
    public int ItemsPerPage { get; set; }

    /// <summary>
    /// The total number of items.
    /// </summary>
    public int TotalItems { get; set; }

    /// <summary>
    /// The total number of page.
    /// </summary>
    public int TotalPages
    {
      get
      {
        int tp = (TotalItems - 1) / ItemsPerPage + 1;
        return tp;
      }
    }
}
```

Helpers

Inline Helpers

I forbindelse med visning af produkter på katalog siden, har vi gjort brug af nogle inline helpers til at vise de forskellige produkter baseret på deres kategorier. Alle disse kan ses i Index.cshtml for Catalogue controlleren. Nedenfor kan ses hvordan helper metoden ser ud for den del der viser bøgerne.

```
@helper RenderBooks(List<Book> items)
{
  if (Model.Books.Count() > 0)
  {
  <h2>The Books</h2>
    foreach (Book b in items)
    {
    @Html.Partial("ProductSummaryBook", b);
    <br/>
      }
    }
  }
}
```

Som det kan ses i det ovenstående eksempel, så er det denne inline helper metode der benytter kaldet til det partial view vi beskrev tidligere. For at få vist dette partial view benyttes standard HTML helperen Html.Partial() som angiver hvilket partial view der skal vises, og giver en instans af Book med som input parameter.

Selve kaldet til disse inline helpers ser ud som vist nedenfor

```
<div>
  <h1>@ViewBag.Title</h1>
  @RenderBooks(Model.Books)
  @RenderCDs(Model.MusicCDs)
  @RenderMovies(Model.Movies)
</div>
```

HTML helpers

I alle vores Views har vi gjort brug af både standard (for eksempel Html.BeginForm og Html.Hidden) og strongly typed (Html.EditorFor) HTML helpers. Der er så mange eksempler rundt omkring i de enkelte Views at vi har valgt ikke at vise nogle her i rapporten men vil i stedet henvise til den medfølgende kode.

Custom helper

Vi har også benyttet en enkelt customer helper i vores webshop. Denne customer helper har vi lavet som en extension metode på HtmlHelper klassen og benyttes til at håndtere paging links på Index view for Catalogue controlleren.

```
namespace MbmStore.HtmlHelpers
{
    /// <summary>
    /// This class hold html extension method(s) used for
    /// handling pagination on the shop pages.
    /// </summary>
    public static class PagingHelpers
    {
        /// <summary>
        /// Html extension method used to generate previous
        /// and next page links on a page.
        /// </summary>
        /// <param name="html"></param>
        /// <param name="pagingInfo"></param>
        /// <param name="pageUrl"></param>
        /// <returns></returns>
        public static MvcHtmlString PageLinks(this HtmlHelper html, PagingInfo pagingInfo, Func<int, string> pageUrl)
        {
            StringBuilder result = new StringBuilder();
            for (int i = 1; i <= pagingInfo.TotalPages; i++)
            {
                TagBuilder tag = new TagBuilder("a");
                tag.MergeAttribute("href", pageUrl(i));
                tag.InnerHtml = i.ToString();
                if (i == pagingInfo.CurrentPage)
                {
                    tag.AddCssClass("selected");
                    tag.AddCssClass("btn-primary");
                }
                tag.AddCssClass("btn btn-default");
                result.Append(tag.ToString());
            }
            return MvcHtmlString.Create(result.ToString());
        }
    }
}
```

På Index viewet bliver denne customer helper benytte på følgende måde

```
@Html.PageLinks(Model.PagingInfo, x => Url.Action("Index", new { page = x, category = Model.CurrentCategory }))
```