

Razor Pages opgavesæt

Dette opgavesæt rummer et antal opgaver omhandlende brug af *Razor Pages*.

Opgaverne 1-5 er oprindeligt lavet af **Henrik Høltzer**, og er i forhold til originalerne kun blevet ganske let modificeret her. Al kredit til Henrik Høltzer for at lave disse opgaver!

Opgaverne 6-9 og frem er tilføjet af undertegnede (Per Laursen).

BEMÆRK: Opgave 9 er IKKE færdiggjort endnu.

Bemærk også, at opgaverne generelt fokuserer mere på funktionalitet end på visuel finish. Fokus er på frameworket som sådan, samt hvordan de enkelte dele i arkitekturen spiller sammen. Man kan derfor sagtens bruge løsningerne til opgaverne som afsæt for at arbejde mere målrettet med den visuelle præsentation.

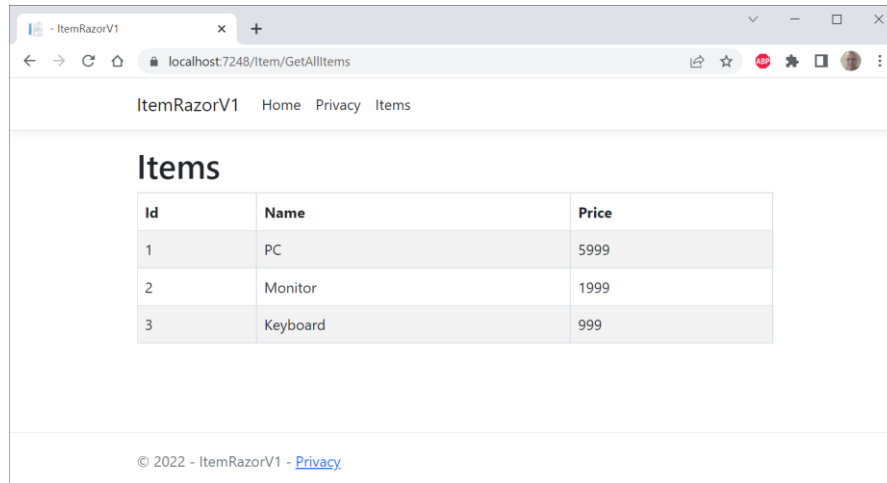
Opgaverne udgør et samlet hele, på den måde at løsningen til en opgave udgør udgangspunktet for den næste opgave. I *Unsolved*-mappen udgør projektet i mappen *ItemRazorVx* således udgangspunktet for opgave x (bemærk at der ikke er noget udgangspunkt for opgave 1, da den starter helt fra bunden), og i *Solved*-mappen udgør projektet i mappen *ItemRazorVx* tilsvarende en løsning for opgave x.

ItemRazor.1 (Start using Razor Pages)	2
ItemRazor.2 (Create new Items).....	5
ItemRazor.3 (Validation, Search and Filter Items).....	10
ItemRazor.4 (Edit and Delete Items)	16
ItemRazor.5 (JSON-file, Background Image, Icons and more Bootstrap features)	24
ItemRazor.6 (Complete implementation for new domain class)	34
ItemRazor.7 (Complete implementation for new complex domain class).....	35
ItemRazor.8 (Refactoring - Repositories)	43
ItemRazor.9 (TBD)	48

ItemRazor.1 (Start using Razor Pages)

Dette er første opgave i opgave-serien **ItemRazor**.

Her skal vi lave det første projekt med *Razor Pages* og få lavet en simpel side der kan vise *Computer Items* i en tabel:



Id	Name	Price
1	PC	5999
2	Monitor	1999
3	Keyboard	999

Trin 1 (ItemRazorV1)

Opret et nyt projek af typen: **ASP.net Core Web App** med navnet: **ItemRazorV1**.

Trin 2 (Afprøv)

Kør projektet og verificer at test siden vises i Browseren.

Trin 3 (Mappen Models)

Tilføj en ny folder (mappe) med navnet: **Models**.

Hint: højreklik på projektet vælg *Add -> New Folder*.

Trin 4 (Klasse Item)

Tilføj en ny klasse **Item** til mappen **Models**. Klassen **Item** skal have følgende properties: **Id** (int), **Name** (string) og **Price** (double) samt diverse constructors (incl. default-konstruktøren).

Trin 5 (Mappen Pages/Item)

Tilføj en ny folder (mappe) med navnet **Item** til mappen **Pages**

Trin 6 (GetAllItems)

Tilføj en ny *Razor Page* (af typen *empty*) til mappen **Item**. Giv siden navnet **GetAllItems**.

Trin 7 (GetAllItems.cshtml.cs)

Tilføj en property **List<Models.Item> Items** til klassen **GetAllItemsModel** i filen *GetAllItems.cshtml.cs*. **Items** skal have en *private set* metode og skal initialiseres med en liste der indeholder et antal **Item**-objekter ala:

```
public List<Models.Item> Items { get; private set; } = new List<Models.Item>
{
    new Models.Item(1, "PC", 5999),
    new Models.Item(2, "Monitor", 1999),
    new Models.Item(3, "Keyboard", 999)
};
```

Trin 8 (GetAllItems.cshtml)

Indsæt kode i *GetAllItems.cshtml* der lister alle **Item**-objekterne fra **Items**-listen i en tabel ala:

```
@page
@model ItemRazorV1.Pages.Item.GetAllItemsModel
@{
}
<h1>Items</h1>

<table class="table table-bordered table-hover table-striped">
    <thead>
        <tr>
            <th>
                Id
            </th>
            <th>
                Name
            </th>
            <th>
                Price
            </th>
        </tr>
    </thead>
    <tbody>
        @if (Model.Items != null)
        {
            foreach (var item in Model.Items)
            {
                <tr>
                    <td>
                        @item.Id
                    </td>
                    <td>
                        @item.Name
                    </td>
                    <td>
```

```

        @item.Price
      </td>
    </tr>
  }
</tbody>
</table>

```

Trin 9 (_Layout.cshtml)

Indsæt et *list-item-tag* med et *anchor-tag* <a> der router til den nye side ala:

```

<li class="nav-item">
  <a class="nav-link text-dark" asp-area="" asp-page="/Item/GetAllItems">Items</a>
</li>

```

Trin 10 (Afprøv)

Afprøv at den nye side virker, og at der kan navigeres til den via linket i menuen.

ItemRazor.2 (Create new Items)

Dette er den anden opgave i opgave-serien **ItemRazor**.

I denne opgave skal vi oprette en "mock-data" klasse, en service-klasse med tilhørende interface, samt en ny *Razor Page* til at kunne oprette nye **Item** objekter.

Udgangspunktet er løsningen fra *ItemRazor.1*

Trin 1 (MockData + MockItems)

Mock data er data beregnet til test. Testdata i opgave 1 blev oprettet direkte i siden **GetAllItems**, og det er bestemt ikke en "pæn" løsning. Testdata skal vi have flyttet ud i en separat mappe, hvor alle testdata kan samles.

Opret en ny mappe **MockData** og tilføj en ny klasse **MockItems** til mappen.

Klassen **MockItems** skal indeholde et *static* classfield **_items**, der refererer til en liste af **Item** objekter:

```
private static List<Item> _items = new List<Item>()
{
    new Item(1, "PC", 5999),
    new Item(2, "Monitor", 1999),
    new Item(3, "Keyboard", 999)
};
```

Klassen **MockItems** skal desuden have en *static* metode **GetMockItems()** der returnerer listen **_items**.

Trin 2 (GetAllItems.cshtml.cs)

Vi skal nu have klassen **GetAllItemsModel** refaktoreret (opdateret/omstruktureret).

Først fjernes initialiseringen af property **Items** og så skal vi have **OnGet()** til at assigne **Items** til vores Mock-data (**Items** skal "pege på/referere til" den statiske liste i **MockItems**)

Hint: benyt **MockItems.GetMockItems()**

Trin 3 (Afprøv)

Afprøv at siden stadig viser tabellen med vores testdata.

Vi er nu klar til endnu en refaktoring, hvor vi introducere begrebet *Service* (via mappe, interface og klasse). En *Service* (funktionalitet) kan være en samling metoder der kan udføre forskellige ting ala. beregninger, filtrering af data, sortering af data, søgning i data, hent og gem af data mm.

Trin 4 (Services, IItemService og ItemService)

Vi skal nu implementere en *Service* der skal kunne tilbyde funktionaliteten: "at hente **Items** objekter fra Mock-data".

Tilføj en ny folder (mappe) med navnet: **Service**

Tilføj et nyt Interface: **IItemService** til mappen. Interfacet skal have metoden: **List<Item> GetItems();**

Tilføj en ny klasse: **ItemService**. Klassen skal implementere interfacet: **IItemService**

Klassen **ItemService** skal have et instancefield **_items**, der kan referere til listen med **Item** objekter. **ItemService** skal også have en constructor der kan initialisere **_items** med Mock-data (Hint: benyt **MockItems.GetMockItems()**)

Trin 5 (Startup.cs)

Vores nye *Service* **ItemService** skal registreres før den kan anvendes af vores *Razor Pages*, det sker i *Program.cs*

Åbn *Program.cs*, og tilføj linjen :

```
builder.Services.AddSingleton<IItemService, ItemService>();
```

lige under linjen:

```
builder.Services.AddRazorPages();
```

Bemærk: **AddSingleton** betyder at der kun skabes en instans af klassen **ItemService** (der kan deles af alle de pages hvor servicen injiceres)

Trin 6 (GetAllItems.cshtml.cs)

Vi skal nu bringe vores nye *Service* i anvendelse, derfor skal der i klassen **GetAllItemsModel** tilføjes et nyt instancefield: **_itemService**, der kan referere til vores nye *Service*. **_itemService** skal initialiseres i konstruktøren, hvor *ItemService* injiceres vha dependency injection (**NB**: Bemærk at vi benytter interface-typen **IItemService** som type for vores *Service*; sammenknytningen mellem **IItemService** og **ItemService** har vi jo netop lavet i *Program.cs*):

```
private IItemService _itemService;

public GetAllItemsModel(IItemService itemService)
{
    _itemService = itemService;
}
```

Ydermere skal vi nu refaktorere **OnGet()**-metoden, så den henter Mock-data via vores *Service*, istedet for direkte via **GetAllItems()** fra **MockItems**:

```
public void OnGet()
{
    Items = _itemService.GetItems();
}
```

Trin 7 (Afprøv)

Afprøv at siden stadig virker efter at vi har refaktoreret koden og introduceret vores *Service*.

Vi har nu refaktoreret vores projekt så det benytter vores *Service*, og er nu klar til at udvide programmet så vi også kan oprette nye *Computer Items*.

Trin 8 (CreateItem)

Først skal der oprettes en ny *Razor Page CreateItem* i mappen **Pages/Item**

Trin 9 (CreateItem.cshtml.cs)

Tilføj følgende instancefield til klassen **CreateItemModel**:

```
private IItemService _itemService;
```

Så kan klassen benytte vores *Service*, og kan dermed anvende listen af **Item** objekter.

Trin 10 (CreateItem.cshtml.cs)

Tilføj en property **Item** til klassen.

```
[BindProperty]
public Models.Item Item { get; set; }
```

Det er denne property der skal bindes til fra UI'en (html-siden), så data kan overføres fra inputfields i UI'en til **Item**-properties i **CreateItemModel**.

Bemærk: **Item** er annoteret med: **[BindProperty]**, der angiver at der kan bindes til denne property fra html-siden.

Trin 11 (CreateItem.cshtml.cs)

Tilføj en constructor **CreateItemModel(IItemService itemService)**, der injicerer vores *Service* og initialiserer **_itemService**:

```
public CreateItemModel(IItemService itemService)
{
    _itemService = itemService;
}
```

Trin 12 (CreateItem.cshtml.cs)

Tilføj metoden **OnPost()**. Metoden skal returnere siden selv (**CreateItem**) hvis **ModelState** ikke er valid, ellers kaldes metoden **AddItem(Item)** (implementeres senere) på **_itemService**. Efter **Item** er added til listen, omdirigeres til siden **GetAllItems**:

```

public IActionResult OnGet()
{
    return Page();
}

public IActionResult OnPost()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _itemService.AddItem(Item);
    return RedirectToPage("GetAllItems");
}

```

Trin 13 (CreateItem.cshtml)

Der skal tilføjes et **<form>**-tag med inputfields så der kan indlæses data til et nyt **Item**:

```

<form method="post">
    <div class="form-group">
        <label asp-for="@Model.Item.Id" class="control-label"></label>
        <input asp-for="@Model.Item.Id" class="form-control"/>
    </div>
    <div class="form-group">
        <label asp-for="@Model.Item.Name" class="control-label"></label>
        <input asp-for="@Model.Item.Name" class="form-control" />
    </div>
    <div class="form-group">
        <label asp-for="@Model.Item.Price" class="control-label"></label>
        <input asp-for="@Model.Item.Price" class="form-control" />
    </div>

    <p>
        <div class="form-group">
            <input type="submit" value="Create" class="btn btn-primary" />
        </div>
    </p>
</form>

```

Dette skulle gerne give en side, som ser nogenlunde således ud:

The screenshot shows a web browser window with the title 'ItemRazorV1'. The address bar displays 'localhost:7248/Item/CreateItem'. The page content includes a navigation bar with 'Home', 'Privacy', and 'Items' links. The main heading is 'Create new Item'. Below this, there are three input fields labeled 'Id', 'Name', and 'Price'. A blue 'Create' button is positioned below the 'Price' field. At the bottom of the page, there is a footer that reads '© 2022 - ItemRazorV1 - Privacy'.

Bemærk: Når der klikkes på et input feltt af typen **submit** (vist i browseren som en *button* med teksten *Create*) sendes der et *http-request* til serveren med metoden *Post*. Det betyder at metoden **OnPost()** automatisk bliver kaldt på **CreateItemModel** objektet.

Trin 14 (IItemService og ItemService)

Nu mangler vi blot at erklære metoden **AddItem(Item item)** i interfacet **IItemService**, og implementere metoden i klassen **ItemService**:

```
public void AddItem(Item item)
{
    _items.Add(item);
}
```

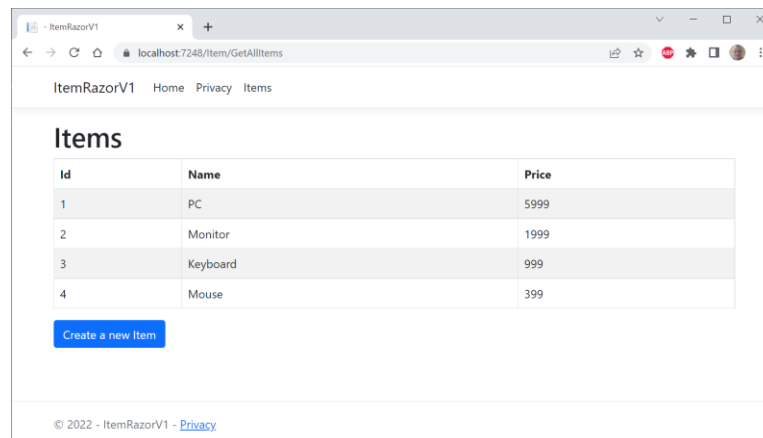
Trin 15 (GetAllItems.cshtml)

For at kunne navigere til vores nye **CreateItem** side indsættes en button, med et *Anchor tag helper* direktiv: **asp-page** til vores **CreateItem** side, under tabellen:

```
<p>
    <a class="btn btn-primary" asp-page="CreateItem">Create a new Item</a>
</p>
```

Trin 16 (Afprøv)

Afprøv at programmet virker, og at der kan tilføjes nye **Item**-objekter via **CreateItem** siden:



ItemRazor.3 (Validation, Search and Filter Items)

Dette er den tredje opgave i opgave-serien **ItemRazor**.

I forrige opgave blev det muligt at oprette nye **Item**-objekter via siden **CreateItem**. I denne opgave vil vi introducere validering af input-data, søgning og filtrering af **Items**.

Udgangspunktet er løsningen fra opgave 2

Trin 1 (Validering - input tag-helper: asp-for - Items.cs)

I sidste opgave benyttede vi *input tag-helper*: **asp-for**. **asp-for** kan hjælpe med at validere, at typen af input-data matcher property'ens type i model-klassen (idet **asp-for** bl.a. sætter HTML "type" attributtet til den tilsvarende .NET type).

Ved at benytte *data annotations* i model-klassen kan vi angive mere specifikke input typer der kan mappes til (f.eks. **[EmailAddress]** vil blive mappet til type="email" og **[DataType(DataType.Date)]** mappes til type="date").

Tilføj følgende *data annotations* i model-klassen **Item**:

```
[Display(Name = "Item ID")]
[Required(ErrorMessage = "Der skal angives et ID til Item")]
[Range(typeof(int), "0", "10000", ErrorMessage = "ID skal være mellem (1) og (2)")]
public int? Id { get; set; }

[Display(Name = "Item Navn")]
[Required(ErrorMessage = "Item skal have et navn")]
public string? Name { get; set; }

[Display(Name = "Pris")]
[Required(ErrorMessage = "Der skal angives en pris")]
public double? Price { get; set; }
```

Bemærk: Typen af **Id** er ændret til **int?** og **Price** til **double?** - det betyder at de også kan være *null* (nullable). Dette er nødvendigt, hvis vi vil udskrive egne **ErrorMessage**, da vi ellers vil få en system errormessage: *"The value is invalid"* (pga **Id** ikke må være *null*).

Overvej:

- Hvad er formålet med **[Display(Name = "...")]** ?
- Hvad gør **[Required(ErrorMessage = "...")]** ?
- Hvordan virker **[Range(typeof(..), ... , ... , ErrorMessage = "...")]** ?

Trin 2 (CreateItem.cshtml)

I siden **CreateItem** tilføjes ****-tags med **asp-validation-for** helper tag samt **class="text-danger"** (Bootstrap - gør teksten rød).

```

<div class="form-group">
  <label asp-for="Item.Id" class="control-label"></label>
  <input asp-for="Item.Id" class="form-control"/>
  <span asp-validation-for="Item.Id" class="text-danger"></span>
</div>
<div class="form-group">
  <label asp-for="Item.Name" class="control-label"></label>
  <input asp-for="Item.Name" class="form-control" />
  <span asp-validation-for="Item.Name" class="text-danger"></span>
</div>
<div class="form-group">
  <label asp-for="Item.Price" class="control-label"></label>
  <input asp-for="Item.Price" class="form-control" />
  <span asp-validation-for="Item.Price" class="text-danger"></span>
</div>

```

Bemærk: **asp-for** behøver ikke at blive foranstillet **@Model** - det er implicit (derfor er de fjernet her - var med i sidste opgave).

Trin 3 (Afprøv)

Afprøv om valideringen virker - du skulle gerne få noget ala dette:

Trin 4 (IItemService, ItemService)

Næste trin er at tilføje en søge-funktion.

Først skal vi tilføje en metode-signatur for funktionen **NameSearch** til **IItemService**. Funktionen skal have en søge-streng som parameter og returtypen skal være en liste af **Items** (vi vælger at benytte det mere generelle Interface **IEnumerable**, i stedet for den specifikke klasse **List** - **List** implementerer **IEnumerable**) :

```

IEnumerable<Item> NameSearch(string str);

```

Dernæst implementeres funktionen **NameSearch** i klassen **ItemService**. Funktionen skal tage en søge-streng som argument og returnere en liste med alle de **Item**-objekter der har et **Name** der indeholder søge-strengen (hvis søge-strengen er tom, returneres alle objekter):

```
public IEnumerable<Item> NameSearch(string str)
{
    List<Item> nameSearch = new List<Item>();
    foreach (Item item in _items)
    {
        if (string.IsNullOrEmpty(str) || item.Name.ToLower().Contains(str.ToLower()))
        {
            nameSearch.Add(item);
        }
    }

    return nameSearch;
}
```

Overvej:

- Hvad er formålet med den lokale liste **nameSearch**?
- Hvorfor benyttes metoden **ToLower()**?
- Hvorfor benyttes Interfacet **IEnumerable** i stedet for klassen **List** som returtype?

Trin 5 (GetAllItems.cshtml.cs)

Ideen er, at det skal være muligt at indtaste en søge-streng på siden **GetAllItems** og få vist alle de **Items** der indeholder denne søge-streng i sidens tabel.

Tilføj en property **SearchString** til klassen **GetAllItemsModel**, og tilføj en annotation **[BindProperty]**, så den kan bindes til et input-field på siden:

```
[BindProperty]
public string SearchString { get; set; }
```

Siden **GetAllItems** kommer til at indeholde en *Search*-button, og når der klikkes på knappen, skal funktionen **NameSearch** kaldes med **SearchString** som argument. Listen **Items** skal assignes til resultatet så det kan blive vist i tabellen. Spørgsmålet er hvordan kobles det sammen?

Vi vælger at benytte en **OnPost**-metode og kalder den **OnPostNameSearch()**. Metoden skal kalde **NameSearch** på servicen og opdatere **Items**:

```
public IActionResult OnPostNameSearch()
{
    Items = _itemService.NameSearch(SearchString).ToList();
    return Page();
}
```

Overvej:

- Hvorfor kaldes metoden **ToList()** på resultatet?
- Hvorfor returneres **Page**?
- Hvordan får **SearchString** tildelt en værdi?

Trin 6 (GetAllItems.cshtml)

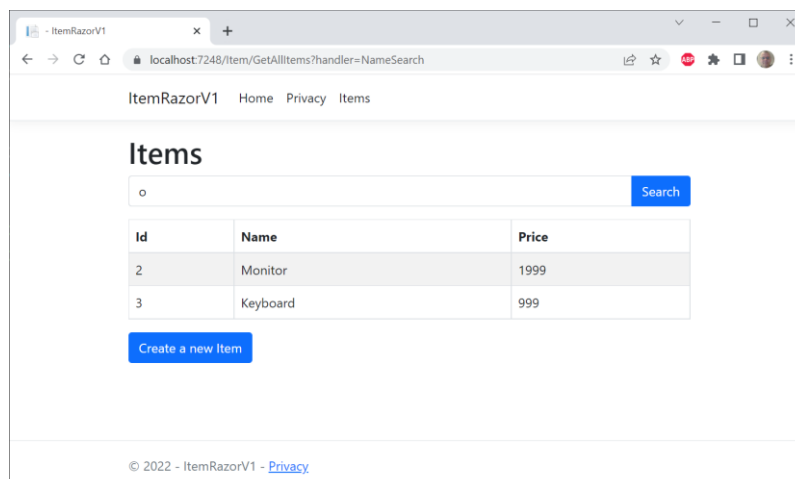
Nu mangler vi blot at tilføje et **<form>**-tag til siden. **<form>**-tagget skal indeholde et **<input>**-tag med et **asp-for** der binder til **SearchString** i model-klassen samt et **<input>**-tag af typen **submit** med et **asp-page-handler** der binder til metoden **NameSearch** i modellen.

```
<form method="post">
  <div class="input-group mb-3">
    <input asp-for="SearchString" class="form-control mr-1"
      placeholder="Enter search term" />
    <input type="submit" asp-page-handler="NameSearch"
      value="Search" class="btn btn-primary" />
  </div>
</form>
```

Bemærk: Da **<form>**-tagget har **method="post"**, skal navnet på handler metoden i model-klassen hedde: **OnPostHandlerMetodeNavn** (her **OnPostNameSearch()** - så kaldes metoden automatisk ved klik på **submit**-knappen (der her har teksten *Search*).

Trin 7 (Afprøv)

Afprøv at programmet virker, og at der kan søges efter **Item**-objekter via søgefeltet ala:



Trin 8 (IItemService, ItemService)

Næste trin er at tilføje en filter-funktion, der kan filtrere **Items** efter en mindste pris og/eller en maks pris. Igen starter vi med **IItemService** og **ItemService**.

Tilføj en metode-signatur for funktionen **PriceFilter** til **IItemService**:

```
IEnumerable<Item> PriceFilter(int maxPrice, int minPrice = 0);
```

Implementér metoden i **ItemService** ala:

```
public IEnumerable<Item> PriceFilter(int maxPrice, int minPrice = 0)
{
    List<Item> filterList = new List<Item>();
    foreach (Item item in _items)
    {
        if ((minPrice == 0 && item.Price <= maxPrice) ||
            (maxPrice == 0 && item.Price >= minPrice) ||
            (item.Price >= minPrice && item.Price <= maxPrice))
        {
            filterList.Add(item);
        }
    }

    return filterList;
}
```

Overvej:

- Hvorfor står der **minPrice=0** i parameter listen?
- Hvornår giver condition: **(minPrice == 0 && item.Price <= maxPrice) || (maxPrice == 0 && item.Price >= minPrice) || (item.Price >= minPrice && item.Price <= maxPrice)** true/false?

Trin 9 (GetAllItems.cshtml.cs)

Ideen er, at det skal være muligt at indtaste en min og en maks pris på siden **GetAllItems** og få vist alle de **Items** der har en pris der ligger inden for grænserne i sidens tabel.

Tilføj to properties **MinPrice** og **MaxPrice** til klassen **GetAllItemsModel** og tilføj annotation **[BindProperty]**, så de kan bindes til input-fields på siden.

Siden **GetAllItems** kommer til at indeholde en *Filter*-button, og når der klikkes på knappen, skal funktionen **PriceFilter** kaldes med **MinPrice** og **MaxPrice** som argumenter. Listen **Items** skal assignes til resultatet så det kan blive vist i tabellen.

Tilføj metoden **OnPostPriceFilter** til klassen (Hint: minder om **OnPostNameSearch**)

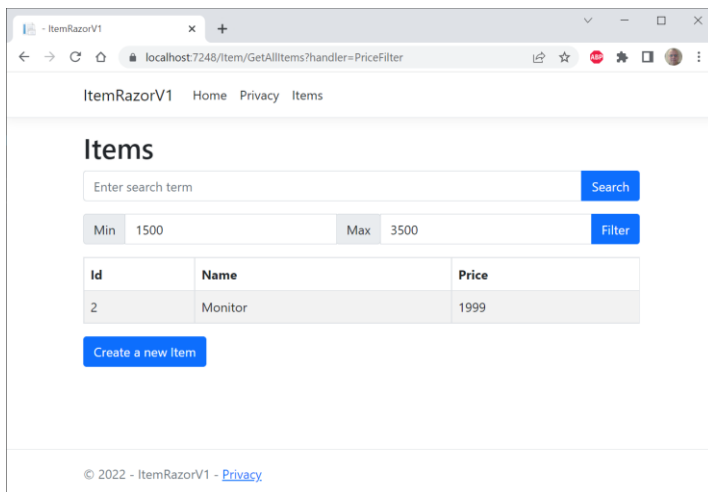
Trin 10 (GetAllItems.cshtml)

Nu mangler vi blot at tilføje et **<form>**-tag til siden. **<form>**-tagget skal indeholde **<input>**-tags med et **asp-for** der binder til **MinPrice** og **MaxPrice** i model-klassen, samt et **<input>**-tag af typen **submit** med et *asp-page-handler* der binder til metoden **PriceFilter** i modellen.

```
<form method="post">
  <div class="input-group mb-3">
    <div class="input-group-prepend">
      <span class="input-group-text">Min</span>
    </div>
    <input asp-for="MinPrice" class="form-control mr-1"/>
    <div class="input-group-prepend">
      <span class="input-group-text">Max</span>
    </div>
    <input asp-for="MaxPrice" class="form-control mr-1"/>
    <input type="submit" asp-page-handler="PriceFilter"
      value="Filter" class="btn btn-primary" />
  </div>
</form>
```

Trin 11 (Afprøv)

Afprøv at programmet virker, og at der kan filtreres efter min og maks priser på **Item**-objekter:



The screenshot shows a web browser window with the URL `localhost:7248/Item/GetAllItems?handler=PriceFilter`. The page title is "ItemRazorV1" and the navigation bar includes "Home", "Privacy", and "Items". The main content area is titled "Items" and contains a search bar with the placeholder "Enter search term" and a "Search" button. Below the search bar is a filter section with two input fields: "Min" with the value "1500" and "Max" with the value "3500", and a "Filter" button. Below the filter section is a table with three columns: "Id", "Name", and "Price". The table contains one row with the values "2", "Monitor", and "1999". Below the table is a "Create a new Item" button. The footer of the page shows the copyright notice "© 2022 - ItemRazorV1 - Privacy".

Id	Name	Price
2	Monitor	1999

ItemRazor.4 (Edit and Delete Items)

Dette er den fjerde opgave i opgave-serien **ItemRazor**.

I forrige opgave blev det muligt at validere data på siden **CreateItem**, samt at søge i **Items** name og filtrere på **Items** price. I denne opgave vil vi gøre det muligt at opdatere (*Edit*) og slette (*Delete*) **Items** fra tabellen.

Udgangspunktet er løsningen fra opgave 3.

Trin 1 (IItemService, ItemService)

Første trin er at tilføje metode-signaturerne **UpdateItem(Item item)** og **GetItem(int id)** til interfacet **IItemService**, og implementere metoderne i klassen **ItemService**.

Metoden **UpdateItem(Item item)** skal tage et **Item** som argument, gennemløbe listen og finde det **Item** der har samme **Id**. **Name** og **Price** skal opdateres for det fundne **Item** (vi tager p.t. ikke højde for om **Item** med givne **Id** findes eller ej, overvej evt hvad man kunne gøre?):

```
public void UpdateItem(Item item)
{
    if (item != null)
    {
        foreach (Item i in _items)
        {
            if (i.Id == item.Id)
            {
                i.Name = item.Name;
                i.Price = item.Price;
            }
        }
    }
}
```

Metoden **GetItem(int id)** skal tage et **Id** og returnere det **Item** der har det givne **Id** (metoden skal returnere *null*, hvis der ikke findes et **Item** med det givne **Id**).

Trin 2 (EditItem)

Næste trin er at oprette en ny *Razor Page* **EditItem** i mappen **Pages/Item**.

Trin 3 (EditItem.cshtml.cs)

Tilføj følgende instancefield til klassen **EditItemModel**:

```
private IItemService _itemService;
```


Så kan klassen benytte vores service, og kan anvende listen af **Item**-objekter.

Trin 4 (EditItem.cshtml.cs)

Tilføj en property **Item** til klassen.

```
[BindProperty]
public Models.Item Item { get; set; }
```

Trin 5 (EditItem.cshtml.cs)

Tilføj en constructor **EditItemModel(ItemService itemService)**, der injicerer **ItemService** og initialisere **_itemService**:

```
public EditItemModel(IItemService itemService)
{
    _itemService = itemService;
}
```

Trin 6 (EditItem.cshtml.cs)

Tilføj metoden **OnGet(int id)**. Metoden skal initialisere property'en **Item** med det **Item**-objekt der skal opdateres. Det sker ved at kalde **GetItem(id)** på servicen, hvor **id** er parameter-overført fra siden (kommer senere). **OnGet(int id)** skal returnere siden selv, nu initialiseret med det **Item** der skal opdateres:

```
public IActionResult OnGet(int id)
{
    Item = _itemService.GetItem(id);
    if (Item == null)
        return RedirectToPage("/NotFound"); //NotFound er ikke defineret endnu

    return Page();
}
```

Trin 7 (EditItem.cshtml.cs)

Tilføj metoden **OnPost()**. Metoden skal kalde **UpdateItem(Item)**, hvor **Item** indeholder de opdaterede properties. Efter opdatering omdirigeres (**RedirectToPage**) til siden **GetAllItems**.

```

public IActionResult OnPost()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _itemService.UpdateItem(Item);
    return RedirectToPage("GetAllItems");
}

```

Trin 8 (EditItem.cshtml)

Tilføj et **<form>**-tag til *EditItem.cshtml*, så de nye værdier for **Name** og **Price** kan indtastes. Der skal være et **<input>**-tag af typen **submit** så der kan sendes et *HTTP-request* (**method="post"**) til serveren og **OnPost()** metoden bliver kaldt:

```

<form method="post">
    <div class="form-group">
        <label asp-for="Item.Id" class="control-label"></label>
        <input asp-for="Item.Id" class="form-control" readonly="@true"/>
    </div>
    <div class="form-group">
        <label asp-for="Item.Name" class="control-label"></label>
        <input asp-for="Item.Name" class="form-control" />
        <span asp-validation-for="Item.Name" class="text-danger"></span>
    </div>
    <div class="form-group">
        <label asp-for="Item.Price" class="control-label"></label>
        <input asp-for="Item.Price" class="form-control" />
        <span asp-validation-for="Item.Price" class="text-danger"></span>
    </div>

    <p>
        <div class="form-group">
            <input type="submit" value="Update" class="btn btn-primary" />
        </div>
    </p>
</form>

```

Overvej:

- Hvad gør `readonly="@true"`?
- Hvorfor vil vi gerne have at `Item.Id` er `readonly`?

Bemærk: når der skal Routes fra siden **GetAllItems** til **EditItem**, skal der sendes en route-parameter med, nemlig **Id** på det **Item** der skal opdateres. Det specificeres øverst på siden i page direktivet **@Page** ved at tilføje **"{id:int}"**. **int** angiver en constraint - at **id** skal være af typen **int**. Havde vi skrevet **"{id:int?}"** var parameteren "optional".

Opdater siden med `@Page "{id:int}"`

Tilføj et `<a>`-tag, med `asp-page="/Item/GetAllItems"` i bunden af siden, så der kan navigeres (routes) tilbage til siden **GetAllItems**:

```
<p>
  <a asp-page="/Item/GetAllItems">Back to List</a>
</p>
```

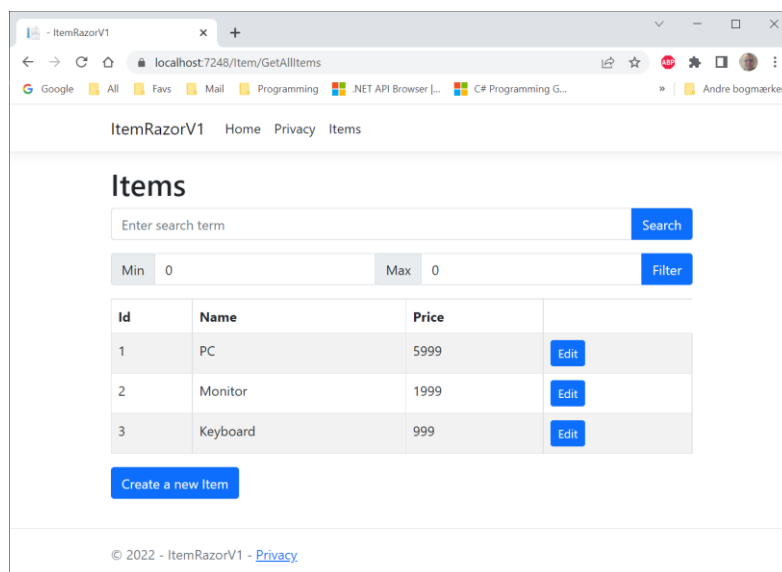
Trin 9 (GetAllItems.cshtml)

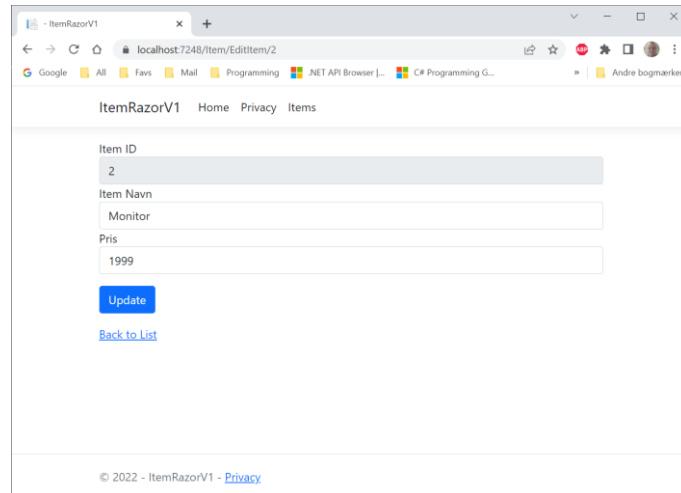
Nu mangler vi blot at tilføje et `<td>`-tag med et `<a>`-tag for hvert **Item** i tabellen (under de andre `<td>`-tag med **Item**-properties). `<a>` tagget skal være af typen **"button"** og skal indeholde et *asp-page helper tag* der router til siden **EditItem**. **Bemærk**: da **id** skal sendes med som router-parameter skal der også være et *asp-route helper tag*: `asp-route-id="@item.Id"` (url'en bliver således: `"/EditItem/id"`)

```
<td>
  <a class="btn btn-primary btn-sm" type="button" title="Edit"
    asp-page="EditItem" asp-route-id="@item.Id">Edit</a>
</td>
```

Trin 10 (Afprøv)

Afprøv at programmet virker, og at det er muligt at opdatere **Item**-objekter ala





Det er nu muligt at lave **CRU** operationerne (**Create, Read og Update**) på **Items**. Nu skal vi have implementeret **Delete**.

Trin 11 (IItemService, ItemService)

Først opdateres **IItemService** og **ItemService** med metoden **DeleteItem(int id)** ala:

```
public Item DeleteItem(int? itemId)
{
    foreach (Item item in _items)
    {
        if (item.Id == itemId)
        {
            _items.Remove(item);
            return item;
        }
    }

    return null;
}
```

Trin 12 (DeleteItem)

Næste trin er at oprette en ny *Razor Page* **DeleteItem** i mappen **Pages/Item**.

Trin 13 (DeleteItem.cshtml.cs)

Tilføj instancefield, property og constructor til klassen **DeleteItemModel** (ala **EditItemModel**):

```
private IItemService _itemService;

public DeleteItemModel(IItemService itemService)
{
    _itemService = itemService;
}
```

```
[BindProperty]
public Models.Item Item { get; set; }
```

Trin 14 (DeleteItem.cshtml.cs)

Tilføj metoden **OnGet(int id)**. Metoden skal initialisere property'en **Item** med det **Item** der skal slettes. Det sker ved at kalde **GetItem(id)** på servicen. **OnGet(int id)** skal returnere siden selv, nu initialiseret med det **Item** der skal slettes:

```
public IActionResult OnGet(int id)
{
    Item = _itemService.GetItem(id);
    if (Item == null)
        return RedirectToPage("/NotFound"); //NotFound er ikke defineret endnu

    return Page();
}
```

Trin 15 (DeleteItem.cshtml.cs)

Tilføj metoden **OnPost()**. **OnPost()** skal kalde service-metoden **DeleteItem(Item.Id)**. Efter sletning omdirigeres (**RedirectToPage**) til siden **GetAllItems** ala:

```
public IActionResult OnPost()
{
    Models.Item deletedItem = _itemService.DeleteItem(Item.Id);
    if (deletedItem == null)
        return RedirectToPage("/NotFound"); //NotFound er ikke defineret endnu

    return RedirectToPage("GetAllItems");
}
```

Trin 16 (DeleteItem.cshtml)

Det skal være muligt at fortryde, at man er ved at slette et **Item**. Derfor skal siden indeholde en bekræftelse og en mulighed for at fortryde.

Tilføj et **<button>**-tag af typen **submit** - et klik på denne submit-button vil kalde ovenstående **OnPost()** - metode og **Item** slettes. Tilføj også et **<a>**-tag med et *asp-page helper tag* der router tilbage til **GetAllItems**, hvis der fortrydes.

```
<h1>Delete Confirmation</h1>

<div class="alert alert-danger">
    <h5>Are you sure you want to delete Item - @Model.Item.Name</h5>
    <form method="post">
        <button type="submit" class="btn btn-danger">Yes</button>
        <a class="btn btn-primary" asp-page="/Item/GetAllItems">No</a>
    </form>
</div>
```

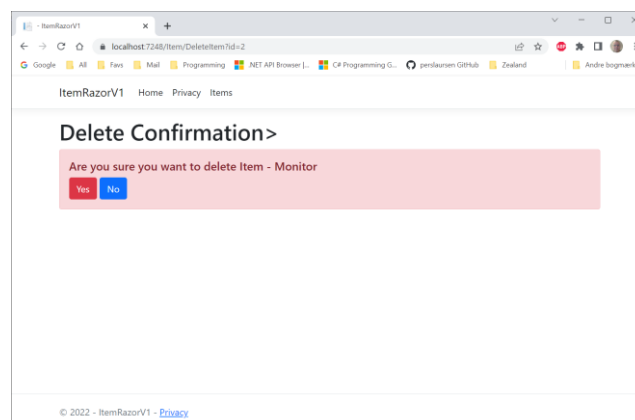
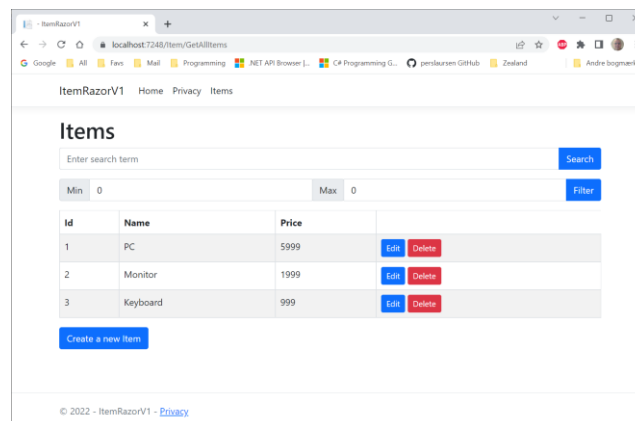
Trin 17 (GetAllItems.cshtml)

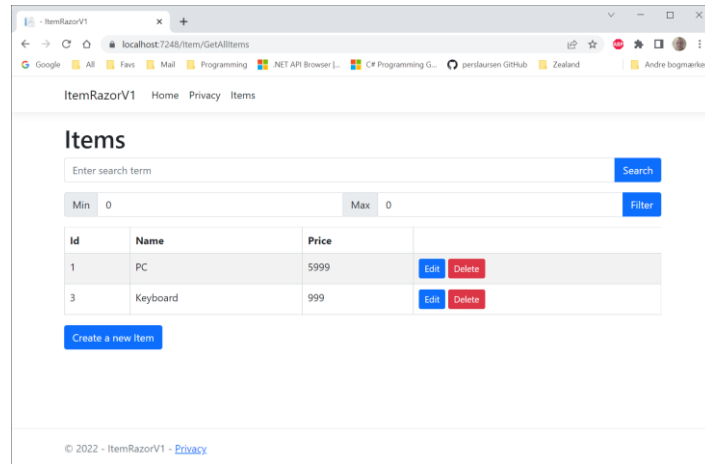
Nu mangler vi blot at tilføje et `<a>`-tag for hvert **Item** i tabellen. `<a>`-tagget skal være af typen **button** og skal indeholde et *asp-page helper tag* der router til siden **DeleteItem**. **Bemærk:** **id** skal sendes med som router-parameter, hvorfor der også her skal være et *asp-route helper tag* **asp-route-id="@item.Id"**. Placer `<a>`-tagget i samme `<td>`-tag som **Edit**-knappen:

```
<td>
  <a class="btn btn-primary btn-sm" type="button" title="Edit"
    asp-page="EditItem" asp-route-id="@item.Id">Edit</a>
  <a class="btn btn-danger btn-sm" type="button" title="Delete"
    asp-page="DeleteItem" asp-route-id="@item.Id">Delete</a>
</td>
```

Trin 18 (Afprøv)

Afprøv at programmet virker og at det er muligt at slette **Item**-objekter ala:





ItemRazor.5 (JSON-file, Background Image, Icons and more Bootstrap features)

Dette er den femte opgave i opgave-serien **ItemRazor**.

I forrige opgave blev det muligt at Editere (update) og Delete (slette) **Item**-objekter via **GetAllItems** siden. I denne opgave vil der være fokus på *persistens*, mere præcist vil der blive introduceret til brugen af JSON-filer, så **Item**-objekter kan hentes når siden indlæses, og gemmes når listen af **Item**-objekter opdateres. Desuden vil der blive introduceret til brugen af Icons, anvendelse af Background Image samt flere andre Bootstrap features inkl. Grid.

Udgangspunktet er løsningen fra opgave 4.

Trin 1 (Data)

Første trin er at oprette en ny mappe **Data**. Mappen skal ligge i **wwwroot**.

Denne mappe kommer senere til at rumme Json-filen *Items.json*, hvor vi vil gemme (persistere) alle vores **Item** objekter.

Trin 2 (JsonFileItemService)

Næste Trin er at oprette en ny klasse **JsonFileItemService** i mappen **Services**. Denne klasse skal benyttes til at hente (**GetJsonItems**) og gemme (**SaveJsonItems**) listen af **Item**-objekter i Json-filen *Items.json*.

Tilføj en property **IWebHostEnvironment WebHostEnvironment**, propertyen skal kun tilbyde **get**:

```
public IWebHostEnvironment WebHostEnvironment { get; }
```

WebHostEnvironment er en service der bl.a. kan benyttes til at få stien til placeringen af vores fil *Items.json*, men først skal den initialiseres. Det gøres ved "Dependency Injection" af servicen i konstruktøren:

```
public JsonFileItemService(IWebHostEnvironment webHostEnvironment)
{
    WebHostEnvironment = webHostEnvironment;
}
```

Nu kan servicen benyttes til at konstruere det fulde filnavn (med path (sti) til placering).

Tilføj en property **string JsonFileName**, propertyen skal kun tilbyde **get**:


```
private string JsonFileName
{
    get { return Path.Combine(WebHostEnvironment.WebRootPath,
                              "Data", "Items.json"); }
}
```

Denne property returnerer det fulde filnavn ved at kombinere stien til **wwwroot** med mappen **Data** og filnavnet *Items.json* (dvs ...**\wwwroot\Data\Items.json**).

Trin 3 (SaveJsonItems)

Når listen af **Item**-objekter skal persisteres, dvs skrives til en fil for at blive gemt, skal objekterne gemmes i et bestemt format. Formatet hedder **JSON** (JavaScript Object Notation) og ser ud som følger:

```
[
  {
    "Id": 1,
    "Name": "PC",
    "Price": 5999
  },
  {
    "Id": 2,
    "Name": "Monitor",
    "Price": 2399
  },
  {
    "Id": 3,
    "Name": "Keyboard",
    "Price": 999
  }
]
```

Et array [...] med objekter {...} bestående af **name:value** - par.

Vores liste af **Item**-objekter skal altså konverteres (serialiseres) til dette format - det kan metoden **JsonSerializer.Serialize<>()** hjælpe med. De serialiserede objekter skal skrives til filen givet ved **JsonFileName**, det gøres ved hjælp af **using(...)** der åbner en **FileStream** til filen oprettet med **File.Create(...)**. Når der skal skrives til filen benyttes en **Utf8JsonWriter** til at "dekore" **Filestreamen** med egenskaben "at kunne skrive i json-format". Det er **Serialize**-metoden der serialisere listen og skriver den til filen:

```
public void SaveJsonItems(List<Item> items)
{
    using (FileStream jsonFileWriter = File.Create(JsonFileName))
    {
        Utf8JsonWriter jsonWriter = new Utf8JsonWriter(
            jsonFileWriter, new JsonSerializerOptions()
            {
                SkipValidation = false,
                Indented = true
            });
        JsonSerializer.Serialize<Item[]>(jsonWriter, items.ToArray());
    }
}
```

Trin 4 (GetJsonItems)

Når den serialiserede liste skal hentes ind igen, er det den modsatte proces. Først benyttes **using(...)** til at åbne en **StreamReader** og **File.OpenText(..)** til at åbne filen. Det er **JsonSerializer.Deserialize<>(...)** metoden der læser fra filen og deserialisere json-objekterne til et array af **Item**-objekter.

Bemærk: Deserializeren benytter default (no arg) constructoren til **Item** til først at oprette et tomt **Item** objekt, Dernæst vil den for hvert **name:value** - par, kalde **set** metoderne med **Name=value** osv. - derfor er det vigtigt at klassen **Item** har en tom constructor **Item(){...}** uden parametre!

Tilføj metoden **GetJsonItems** til **JsonFileItemService**:

```
public IEnumerable<Item> GetJsonItems()
{
    using (StreamReader jsonFileReader = File.OpenText(JsonFileName))
    {
        return JsonSerializer.Deserialize<Item[]>(jsonFileReader.ReadToEnd());
    }
}
```

Trin 5 (Program.cs, ItemService.cs)

Nu er service-klassen, der skal hjælpe med at "persistere" dvs gemme/hente vores **Item** objekter, på plads. Vi skal nu have vores **ItemService** til at benytte den nye service-klasse, i stedet for at anvende MockData (testdata).

Først skal den nye service konfigureres. Tilføj denne linje til *Program.cs*, lige under de andre **Add...** linjer:

```
builder.Services.AddTransient<JsonFileItemService>();
```

Overvej/Undersøg:

- Hvad er forskellen mellem **AddTransient** og **AddSingleton**?
- Hvornår benyttes **AddTransient** hhv **AddSingleton**?

Nu kan den nye service injiceres ind i **ItemService** via konstruktøren. Opdater **ItemService** med:

```
private JsonFileItemService JsonFileItemService { get; set; }

public ItemService(JsonFileItemService jsonFileItemService)
{
    JsonFileItemService = jsonFileItemService;
    // _items = MockItems.GetMockItems();
    _items = JsonFileItemService.GetJsonItems().ToList();
}
```

Bemærk: Første gang vi afprøver programmet findes filen ikke endnu og data ligger kun i **MockItems** - derfor udkommenterer vi initialiseringen via **JsonFileItemService**. Næste gang programmet afvikles er filen dannet og listen gemt (såfremt listen er ændret og **save** kaldt), derfor udkommenteres initialiseringen via **MockItems** og initialiseringen via **JsonFileItemService** indkommenteres igen.

Trin 6 (ItemService - AddItem, DeleteItem, UpdateItem)

Hver gang listen ændres skal vi "persistere" ændringen, dvs vi skal kalde **SaveJsonItems**.

Opdater **AddItem**, **DeleteItem** og **UpdateItem**, så **SaveJsonItems()** kaldes når listen opdateres:

```
public void AddItem(Item item)
{
    _items.Add(item);
    JsonFileItemService.SaveJsonItems(_items);
}

public void UpdateItem(Item item)
{
    if (item != null)
    {
        foreach (Item i in _items)
        {
            if (i.Id == item.Id)
            {
                i.Name = item.Name;
                i.Price = item.Price;
            }
        }
        JsonFileItemService.SaveJsonItems(_items);
    }
}

public Item DeleteItem(int? itemId)
{
    Item itemToBeDeleted = null;
    foreach (Item item in _items)
    {
        if (item.Id == itemId)
        {
            itemToBeDeleted = item;
            break;
        }
    }

    if (itemToBeDeleted != null)
    {
        _items.Remove(itemToBeDeleted);
        JsonFileItemService.SaveJsonItems(_items);
    }

    return itemToBeDeleted;
}
```

Trin 7 (Afprøv)

Afprøv at programmet og "persisteringen" virker, dvs. at nyoprettede objekter stadig vil være i listen/tabellen, når applikationen genstartes (samt at andre opdateringer af listen ved brug af **Edit/Delete** bliver gemt).

Husk: Første gang applikationen køres skal data hentes fra **MockItems**! Efterfølgende skal data hentes fra filen (NB: kræver at der er udført en **Add, Delete** eller **Update** inden).

Nu hvor vi har CRUD, Search, Filter og Persistensen på plads, er det tid til at gøre lidt ved UI-designet (Bootstrap m.v.)

Trin 8 (Bootswatch - Themes, _Layout.cshtml)

Det første vi gør er at ændre på "Themes" dvs det *look-and-feel* tema, som vores applikation skal benytte. Det gøres ved i *_Layout.cshtml* filen at tilføje et **<Link>**-tag med en *CDN* reference til det Bootswatch/Bootstrap Theme vi ønsker (her er valgt *slate*). Indsæt følgende link (efter linket til bootstrap):

```
<link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootswatch@5.2.2/dist/slate/bootstrap.min.css">
```

Trin 9 (Afprøv)

Afprøv at linket virker og at applikationen har skiftet Theme til *slate*.

Trin 10 (background-image, ItemRazorStyles.css, _Layout.cshtml)

Den letteste måde, at indsætte et baggrunds billede, er via "styling" af **<body>**-tagget.

Opret en ny *css* fil *ItemRazorStyles.css* i mappen **wwwroot\css** (højreklik på mappen, vælg **Add-> New Item -> Style Sheet**).

Udskift default stylesheet'et *site.css* med det nye:

```
<link rel="stylesheet" href="~/css/ItemRazorStyles.css" asp-append-version="true" />
```

Det billede vi gerne vil benytte er:



Kopier billedet til din egen PC, kald det for *computergear.jpg* , og gem billedet **computergear.jpg** i mappen **wwwroot** (alternativt kan billedet hentes direkte her):

<http://www.heho-zealand.dk/SWC1-2021e/computergear.jpg>

Indsæt følgende *css* i *ItemRazorStyles.css*:

```
body {  
    /* The image used */  
    background-image: url("../computergear.jpg");  
    /* Full height */  
    height: 100%;  
    /* Center and scale the image nicely */  
    background-position: center;  
    background-repeat: no-repeat;  
    background-size: cover;  
}
```

Trin 11 (Afprøv)

Afprøv at baggrunds-billedet bliver vist i applikationen.

Trin 12 (Mere styling, ItemRazorStyles.css, GetAllItems)

Tabellen ser ikke så godt ud med det nye baggrunds-billede, men det kan vi gøre noget ved!

Først ændrer vi på "opacity". Tilføj følgende *css* klasse-definition til *ItemRazorStyles.css*:

```
.table-opacity {  
    background-color: #ffffff;  
    opacity: 0.9;  
}
```

Tilføj den nye *css* klasse til **<table>** i **GetAllItems**:

```
<table class="table table-bordered table-hover table-striped table-opacity">
```

Så er vi klar til lidt mere "styling"! Tilføj et nyt **<th>**-tag under de andre table headers

```
<th>  
    Actions  
</th>
```

Tilføj **class="btn-secondary active"** til **<tr>** under **<thead>**:

```
<thead>  
    <tr class="btn-secondary active">
```

Tilføj `class="btn-secondary"` til `<tr>` i `foreach`-løkken:

```
foreach (var item in Model.Items)
{
    <tr class="btn-secondary">
```

Bemærk: Nu er det filen `_Layout.cshtml`, der skal opdateres. Tilføj `class="d-flex flex-column min-vh-100"` til `<body>`-tagget:

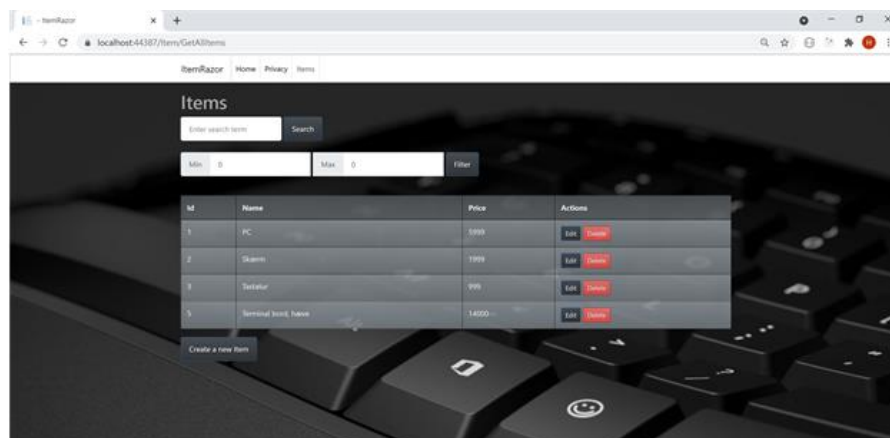
```
<body class="d-flex flex-column min-vh-100">
```

Tilføj klassen `mt-auto` til `<footer>`-tagget:

```
<footer class="border-top footer text-muted mt-auto">
```

Trin 13 (Afprøv)

Du skulle gerne få vist tabellen i et lidt pænere design ala (der kan være små afvigelser):



Bemærk: navigations-baren er lys, for at få den mørk (som er default for *slate*) og teksten hvid:

- Fjern klasserne `navbar-light` og `bg-white` fra `<nav>`-tagget i `_Layout.cshtml`.
- Fjern klassen `text-dark` fra `<a>`-taggene

Trin 14 (Layout - Bootstrap Grid, GetAllItems)

For at kunne styre layoutet på siden benytter vi *Bootstrap Grids*, hvor hver *row* bliver opdelt i 12 *columns*, se evt: <https://getbootstrap.com/docs/5.0/layout/grid/>

Først indsættes i *GetAllItems.cshtml* en **row** med 3 **col** (columns) der indeholder overskrifter til **Search** og **Filter**:

```
<div class="row">
  <div class="col-4"><h5>Search Name</h5></div>
  <div class="col-4"><h5>Filter Price</h5></div>
  <div class="col-4"></div>
</div>
```

Bemærk: **col-4** betyder at **<div>**-tagget spænder over 4 (af 12) kolonner.

Overvej: Hvorfor er der indsat et tomt **<div>**-tag med **col-4** efter de to andre?

Tilføj et **<div class="row">**-tag om de to **<form>**-tags samt **<div class="col-4">** og **<div class="col-5">** om de enkelte forms:

```
<div class="row">
  <div class="col-4">
    <form method="post" class="form-inline">
      ...
    </form>
  </div>

  <div class="col-5">
    <form method="post" class="form-inline">
      ...
    </form>
  </div>

  <div class="col-2"></div>
</div>
```

Bemærk: Der er tilføjet et tomt **<div class="col-2"></div>** efter de to forms - hvorfor?

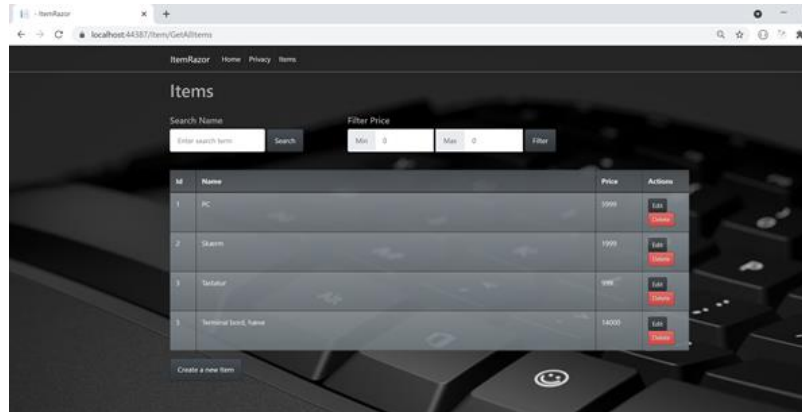
Vi styrer også lige de enkelte kolonnens bredde i tabellen: Tilføj **width="60%"** og de 4 **<col>**-tags:

```
<table class="table table-bordered table-hover
  table-striped table-opacity" width="60%">
  <colgroup>
    <col style="width : 5%">
    <col style="width : 77%">
    <col style="width : 9%">
    <col style="width : 9%">
  </colgroup>
  <thead>
```

Overvej/Undersøg: Hvad er formålet med de enkelte Bootstrap "klasser", hvad er effekten af dem? - måske du kan anvende dem i dit kommende projekt!

Trin 15 (Afprøv)

Du skulle gerne få vist en side ala (der kan være små afvigelser):



Helt korrekt - det ser ikke færdigt ud! Ideen er at knapperne skal udskiftes med *icons*.

Trin 16 (Icons - `_Layout.cshtml`, `GetAllItems`)

Vi vil gerne benytte *Icons*, *font-awesome* har en række gratis icons vi frit kan benytte. Først skal vi tilføje et *CDN-link* til *font-awesome* (i `_Layout.cshtml`):

```
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.7.0/css/font-awesome.min.css">
```

Når der skal indsættes et icon på et link `<a>`-tag indsættes et `<i>`-tag med `class="fa fa-edit"`. Dette er en *css-class* defineret af *font-awesome* der indsætter et *edit*-icon.

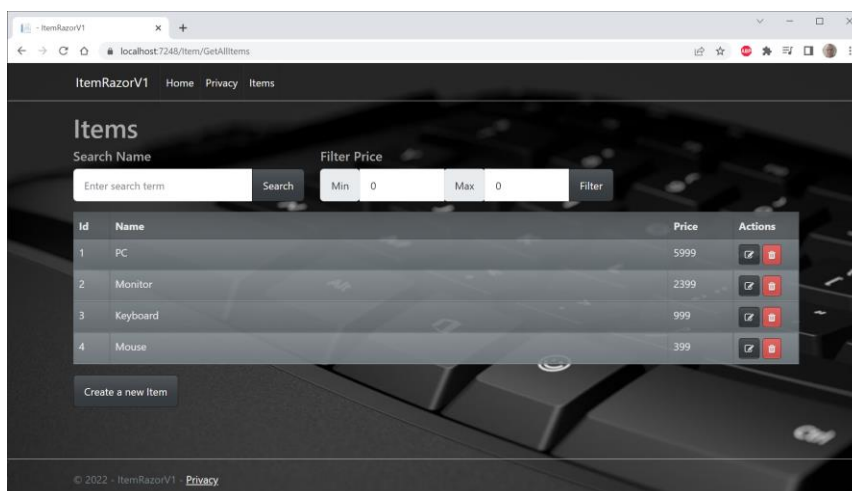
I `GetAllItems.cshtml`, tilføj `<i class="fa fa-edit"></i>` til `<a>`-tag for *edit* og `<i class="fa fa-trash"></i>` til `<a>`-tagget for *delete*:

```
<a class="btn btn-primary btn-sm" type="button" title="Edit" asp-page="EditItem"
asp-route-id="@item.Id"><i class="fa fa-edit"></i></a>
<a class="btn btn-danger btn-sm" type="button" title="Delete" asp-page="DeleteItem"
asp-route-id="@item.Id"><i class="fa fa-trash"></i></a>
```


Overvej/Undersøg: Hvad er formålet med de enkelte Bootstrap "klasser", hvad er effekten af dem? - måske du kan anvende dem i dit kommende projekt!

Trin 17 (Afprøv)

Du skulle gerne få vist en side ala (der kan være små afvigelser):



ItemRazor.6 (Complete implementation for new domain class)

Dette er den sjette opgave i opgave-serien **ItemRazor**.

I opgaverne 1 til 5 har vi gennemført en fuld implementering af CRUD-funktioner for en enkelt domæne-klasse **Item**. I denne opgave er målet at udføre alle disse trin igen, men nu for en ny domæne-klasse **Customer**, der ligesom **Item** stadig kun har properties af simple typer. Langt de fleste af disse trin kan udføres ved at lægge sig meget tæt op af det tilsvarende trin for **Item**. Derved bliver opgaven måske lidt rutinepræget, men samtidigt begynder man nok at ane, hvor der er potentiale for at lave noget refaktorisering senere...

Udgangspunktet er løsningen fra opgave 5.

Trin 1 (Data og services)

Tilføj en ny **Customer** klasse i **Models** folder. Du kan selv vælge hvilke properties du vil have med, men det anbefales stærkt at du i hvert fald har en **Id** property af typen **int**. Derudover kan du have f.eks. **Navn**, **Email** og **Adresse**, alle af typen **string**.

Tilføj klassen **JsonFileCustomerService** til **Service**-folderen. Den kommer til at blive næsten identisk med **JsonFileItemService**, med alle referencer til **Item**-klassen skiftet ud med **Customer**.

Tilføj interfacet **ICustomerService** og klassen **CustomerService** til **Service**-folderen. Disse bliver også næsten identiske til de tilsvarende for **Item**. Du kan dog undlade **PriceFilter**-metoden fra dette interface, da den ikke giver mening for **Customer**.

Tilføj **MockCustomers** til **MockData**-folderen. Brug **MockItem** som (kraftig) inspiration.

Tilføj **ICustomerService/CustomerService** og **JsonFileCustomerService** som services i **Program.cs**, ved at følge mønsteret for de tilsvarende **Item**-relaterede services (brug **AddSingleton** hhv **AddTransient**).

Trin 2 (Razor Pages)

Tilføj en ny *Razor Page* **GetAllCustomers** i folderen **Pages/Customer**. Den kan med stor sandsynlighed følge mønsteret fra den tilsvarende **GetAllItems** (både for **.cshtml** og **cshtml.cs**). Vær igen opmærksom på, at du ikke får brug for at implementere **OnPostPriceFilter**.

Tilføj tre nye *Razor Pages* **Create/Edit/DeleteCustomer**, i folderen **Pages/Customer**. Surprise: disse vil også stort set være identiske med de tilsvarende **Item**-pages (det bliver forhåbentligt mere og mere tydeligt, at der er basis for senere at indføre nogle base-klasser 😊).

Tilføj **li**-tag for **GetAllCustomers** i **_Layout.cs**.

Trin 3 (Afprøv)

Afprøv om alt virker som forventet (**NB**: Husk det lille "trick" med først at bruge **MockData**, og derefter JSON i **CustomerService**, se evt. Trin 7 i Opgave **RazorPages.5**)

ItemRazor.7 (Complete implementation for new complex domain class)

Dette er den syvende opgave i opgave-serien **ItemRazor**.

I opgaverne 1 til 5 har vi gennemført en fuld implementering af CRUD-funktioner for en enkelt domæne-klasse **Item**, og i opgave 6 alle disse trin igen for klassen **Customer**. For både **Item** og **Customer** gælder det, at alle properties er af simple typer

I denne opgave er målet at udføre alle disse trin endnu engang, men nu for en ny domæne-klasse **Order**, der har nogle mere komplekse properties, der refererer til **Item**- og **Customer**-objekter. Derved bliver flere trin noget mere komplekse.

Flere trin i opgaven handler igen om at sætte noget givent kode ind på det rigtige sted. Noget af dette kode rummer nogle C#-elementer vi ikke har set før; det er ikke så vigtigt i sig selv, det er mere vigtigt at koden (forhåbentligt) gør det den skal 😊.

Udgangspunktet er løsningen fra opgave 6.

Trin 1 (Data)

Tilføj klassen **OrderLine** i **Models**-folderen. Dette er en hjælpeklasse til **Order**-klassen. Bemærk at den refererer til et **Item**.

```
public class OrderLine
{
    private static int _nextId = 0;

    public int Id { get; set; }
    public Item Item { get; set; }
    public int Amount { get; set; }

    [JsonIgnore]
    public double TotalPrice { get { return (Item.Price ?? 0) * Amount; } }

    public OrderLine(Item item, int amount) : this(++_nextId, item, amount) { }

    public OrderLine(int id, Item item, int amount)
    {
        Id = id;
        Item = item;
        Amount = amount;
    }

    public OrderLine() { }

    public override string ToString() { return $"{Item.Name} x {Amount}"; }
}
```

Tilføj klassen **Order** i **Models**-folderen. Bemærk at den refererer til en **Customer**, og rummer en liste af **OrderLine**-objekter.

```
public class Order
{
    public int Id { get; set; }
    public Customer? Customer { get; set; }
    public List<OrderLine> Items { get; set; } = new List<OrderLine>();
    public string? Remark { get; set; }

    [JsonIgnore]
    public double TotalPrice { get { return Items.Select(i => i.TotalPrice).Sum(); } }

    [JsonIgnore]
    public string ItemsSummary { get { return string.Join(" ", Items); } }

    [JsonIgnore]
    public string CustomerInfo { get { return Customer?.Name ?? "(none)"; } }

    public Order() { }

    public Order(int id, Customer? customer) { Id = id; Customer = customer; }

    public OrderLine? GetOrderLine(int itemId)
    { return Items.FirstOrDefault(i => i.Item.Id == itemId); }
}
```

Trin 2 (Services)

Tilføj klassen **JsonFileOrderService** til **Service**-folderen. Den kommer til at blive næsten identisk med de øvrige **JsonFile...** klasser.

Tilføj **IOrderService** og **OrderService** til **Service** folder. I stedet for en **NameSearch** metode skal interfacet have en **CustomerNameSearch** metode. Implementationen i **OrderService** kommer dog til at ligge meget tæt på implementationen af **NameSearch** i f.eks. **CustomerService**. Koden til **UpdateOrder** bliver lidt mere kompleks:

```
public void UpdateOrder(Order order)
{
    if (order != null)
    {
        foreach (Order o in _orders)
        {
            if (o.Id == order.Id)
            {
                o.Customer = order.Customer;
                o.Remark = order.Remark;

                o.Items.Clear();
                foreach (OrderLine orderLine in order.Items)
                    o.Items.Add(orderLine);
            }
        }
        JsonFileOrderService.SaveJsonOrders(_orders);
    }
}
```

Tilføj **MockOrders** til **MockData** (du må naturligvis gerne variere på de konkrete værdier, hvis du har lyst).

```
public class MockOrders
{
    private static List<Order> _orders;

    static MockOrders()
    {
        Order o1 = new Order(1, new Customer(1, "Anders", "and@mail.dk", "Stien 12"));
        Order o2 = new Order(2, new Customer(3, "Carina", "car@mail.dk", "Skolevej 87"));
        Order o3 = new Order(3, new Customer(1, "Anders", "and@mail.dk", "Stien 12"));

        o1.Items.Add(new OrderLine(new Item(1, "PC", 5999), 1));
        o1.Items.Add(new OrderLine(new Item(2, "Monitor", 1999), 2));

        o2.Items.Add(new OrderLine(new Item(3, "Keyboard", 999), 3));

        o3.Items.Add(new OrderLine(new Item(1, "PC", 5999), 1));
        o3.Items.Add(new OrderLine(new Item(3, "Keyboard", 999), 2));

        o1.Remark = "This is the first order";
        o2.Remark = "This is the second order";
        o3.Remark = "This is the third order";

        _orders = new List<Order>() { o1, o2, o3 };
    }

    public static List<Order> GetMockOrders() { return _orders; }
}
```

Tilføj **IOrderService/OrderService** og **JsonFileOrderService** som services i **Program.cs**, ved at følge mønsteret for de tilsvarende **Item/Customer**-relaterede services (brug **AddSingleton** hhv **AddTransient**).

Trin 3 (GetAllOrders Razor Page)

Tilføj *Razor Page* **GetAllOrders** i folderen **Pages/Order**. M.h.t. model-klassen vil den følge mønsteret fra de tilsvarende **GetAll...** model-klasser. M.h.t. page-layoutet vil dette også i store træk følge de tilsvarende, men de to **Order**-properties **ItemsSummary** og **TotalPrice** kan måske gøre gavn...

Trin 4 (Create/Delete/EditOrder Razor Pages - skelet)

Tilføj *Razor Pages* **Create/Delete/EditOrder** i folderen **Pages/Order**, men vent indtil videre med at udfylde dem. I første omgang skal de blot oprettes, så vi kan kompilere vores projekt.

Tilføj **li**-tag for **GetAllOrders** i **_Layout.cs**.

Trin 5 (Afprøv)

Afprøv om du kan få vist **Order**-objekterne fra **MockOrders** i **GetAllOrders**-pagen

Trin 6 (Udfyldning af CreateOrder.cshtml+cshtml.cs)

Vi skal nu have fyldt kode i **CreateOrder**.

Vi antager, at man laver en ny ordre ved at klikke på et ikon i oversigten over **Customers** (d.v.s. i **GetAllCustomers**), ud for den kunde, man vil lave en ny ordre for. Derved kan vi give et customer-id med som parameter til **CreateOrder**, mere specifikt til **OnGet**-metoden i **CreateOrder**.

Det antages ydermere, at det eneste man derudover (eventuelt) tilføjer til den nye ordre en en "remark". Når denne (eventuelt) er tilføjet, klikkes på en "Create"-knap, og man sendes videre til **EditOrder**.

Kode til **CreateOrder.cshtml**:

```
<form method="post">
  <div class="form-group">
    <label asp-for="@Model.CustomerId" class="control-label"></label>
    <input asp-for="@Model.CustomerId" class="form-control" readonly="@true" />
  </div>
  <div class="form-group">
    <label asp-for="@Model.Customer.Name" class="control-label"></label>
    <input asp-for="@Model.Customer.Name" class="form-control" readonly="@true" />
  </div>
  <div class="form-group">
    <label asp-for="@Model.OrderRemark" class="control-label"></label>
    <input asp-for="@Model.OrderRemark" class="form-control" />
  </div>

  <p>
    <div class="form-group">
      <input type="submit" value="Create" class="btn btn-primary" />
    </div>
  </p>
</form>
```

Kode til **CreateOrder.cshtml.cs**:

```
public class CreateOrderModel : PageModel
{
    private IOrderService _orderService;
    private ICustomerService _customerService;

    public CreateOrderModel(IOrderService orderService, ICustomerService customerService)
    {
        _orderService = orderService;
        _customerService = customerService;
    }

    [BindProperty]
    public string OrderRemark { get; set; }

    [BindProperty]
    public int? CustomerId { get; set; }

    public Models.Customer Customer { get; set; }

    public IActionResult OnGet(int custId)
    {
        Customer = _customerService.GetCustomer(custId);
        CustomerId = custId;

        return Page();
    }
}
```

```

public IActionResult OnPost()
{
    if (CustomerId.HasValue)
    {
        Models.Customer customer = _customerService.GetCustomer(CustomerId.Value);

        List<Models.Order> orders = _orderService.GetOrders();
        int orderId = orders.Count == 0 ? 1 : orders.Select(o => o.Id).Max() + 1;

        Models.Order newOrder = new Models.Order(orderId, customer);
        newOrder.Remark = OrderRemark;

        _orderService.AddOrder(newOrder);

        return RedirectToPage("EditOrder", new { id = orderId });
    }

    return RedirectToPage("GetAllOrders");
}
}

```

Vi skal også tilføje den nævnte knap til **GetAllCustomers.cshtml**, på samme sted som de andre knapper:

```

<a class="btn btn-primary btn-sm" type="button" title="CreateOrder" asp-page="/Order/CreateOrder"
asp-route-custId="@customer.Id"> <i class="fa fa-shopping-cart"></i> </a>

```

Bemærk, at vi benytter **asp-route-custId="@customer.Id"**, som således passer sammen med navnet på parameteren til OnGet-metoden i **CreateOrder**.

Trin 7 (Afprøv)

Afprøv om du kan komme fra **GetAllCustomers** til **CreateOrder** ved brug af knappen, og få vist en ordre hvor **Customer** allerede er udfyldt. **NB:** Vi har ikke udfyldt **EditOrder** endnu, så den vil ikke virke endnu!

Trin 8 (Udfyldning af DeleteOrder.cshtml+cshtml.cs)

Vi skal nu have fyldt kode i **DeleteOrder**, som heldigvis stadig er ret simpel, da den vil fungere på samme måde som f.eks. **DeleteCustomer**. Du kan derfor blot copy-paste koden fra **DeleteCustomer.cshtml+cshtml.cs** og tilpasse den til **Order**.

Trin 9 (Udfyldning af EditOrder.cshtml+cshtml.cs)

Vi skal nu have fyldt kode i **EditOrder**, som bliver den mest komplekse page indtil videre. Blot at afklare hvordan editering af en ordre overhovedet skal foregå er ganske komplekst. Her er antaget følgende, kun som et eksempel:

- En ordre kan kun tilrettes m.h.t. hvilke **Items** ordren rummer.
- Hvis en ordre rummer 0 stk. af et givet **Item**, kan man tilføje et nyt **Item** til ordren ved at vælge **Item** fra en *drop-down* liste over alle **Items**. Herved laves en ny "ordre-linje", hvor det er angivet ordren rummer 1 stk. af dette **Item**.
- For en eksisterende ordre-linje kan styk-antallet hæves eller sænkes, ved at klikke på en "+"-knap hhv en "-"-knap. Hvis antallet falder til 0 stk., fjernes ordre-linjen helt fra ordren.

Koden er relativt omfattende. Her er koden til **EditOrder.cshtml**:

```
<form method="get">
  <div class="form-group">
    <label asp-for="Order.Id" class="control-label"></label>
    <input asp-for="Order.Id" class="form-control" readonly="@true" />
  </div>
  <div class="form-group">
    <label asp-for="Order.CustomerInfo" class="control-label"></label>
    <input asp-for="Order.CustomerInfo" class="form-control" readonly="@true" />
  </div>
  <div class="form-group">
    <label asp-for="Order.ItemsSummary" class="control-label"></label>
    <input asp-for="Order.ItemsSummary" class="form-control" readonly="@true" />
  </div>
  <div class="form-group">
    <label asp-for="Order.TotalPrice" class="control-label"></label>
    <input asp-for="Order.TotalPrice" class="form-control" readonly="@true" />
  </div>
  <div class="form-group">
    <label asp-for="Order.Remark" class="control-label"></label>
    <input asp-for="Order.Remark" class="form-control" readonly="@true" />
  </div>

  <div>
    <label class="control-label">Items</label>
    <table class="table table-bordered table-hover table-striped table-opacity" width="60%">
      <colgroup>
        <col style="width : 10%">
        <col style="width : 60%">
        <col style="width : 10%">
        <col style="width : 10%">
        <col style="width : 10%">
      </colgroup>
      <thead>
        <tr class="btn-secondary active">
          <th>
            Id
          </th>
          <th>
            Name
          </th>
          <th>
            Amount
          </th>
          <th>
            Price
          </th>
          <th>
            Actions
          </th>
        </tr>
      </thead>
      <tbody>
        @if (Model.Order.Items != null)
        {
          foreach (var orderLine in Model.Order.Items)
          {
            <tr class="btn-secondary">
              <td>
                @orderLine.Item.Id
              </td>
              <td>
                @orderLine.Item.Name
              </td>
              <td>
                @orderLine.Amount
              </td>
              <td>
                @orderLine.TotalPrice
              </td>
            </tr>
          }
        }
      </tbody>
    </table>
  </div>
</form>
```



```

        <td>
            <form method="post">
                <button asp-page-handler="Decrease" asp-route-itemId="@orderLine.Item.Id"
                    asp-route-orderId="@Model.Order.Id" class="btn btn-primary btn-sm">
                    <i class="fa fa-minus-square"></i>
                </button>

                <button asp-page-handler="Increase" asp-route-itemId="@orderLine.Item.Id"
                    asp-route-orderId="@Model.Order.Id" class="btn btn-primary btn-sm">
                    <i class="fa fa-plus-square"></i>
                </button>
            </form>
        </td>
    </tr>
}
</tbody>
</table>
</div>
</form>

<p>
    <form method="post">
        <label>Add new Item to Order : </label>
        <select asp-for="ChosenItem" asp-items="@Model.ItemList"></select>
        <button asp-page-handler="Add" asp-route-orderId="@Model.Order.Id" class="btn btn-primary btn-sm">
            <i class="fa fa-plus-square"></i>
        </button>
    </form>
</p>

<p>
    <a asp-page="/Order/GetAllOrders">Back to List</a>
</p>

```

Her er koden til **EditOrder.cshtml.cs**:

```

public class EditOrderModel : PageModel
{
    private IOrderService _orderService;
    private IItemService _itemService;

    public EditOrderModel(IOrderService orderService, IItemService itemService)
    {
        _orderService = orderService;
        _itemService = itemService;

        List<Models.Item> items = _itemService.GetItems();
        ItemList = items.Select(i => new SelectListItem
            { Text = i.Name, Value = i.Id.ToString() }).ToList();
    }

    [BindProperty]
    public Models.Order Order { get; set; }

    [BindProperty]
    public string ChosenItem { get; set; }

    public List<SelectListItem> ItemList { get; set; }

    public IActionResult OnGet(int id)
    {
        Order = _orderService.GetOrder(id);
        if (Order == null)
            return RedirectToPage("/NotFound"); //NotFound er ikke defineret endnu

        return Page();
    }

    public IActionResult OnPost()
    {
        return Page();
    }

    public IActionResult OnPostDecrease(int itemId, int orderId)
    {
        Order = _orderService.GetOrder(orderId);
    }
}

```

```

Models.Order updOrder = CloneOrder(Order);
OrderLine? chosenOrderLine = updOrder.GetOrderLine(itemId);

if (chosenOrderLine != null)
{
    chosenOrderLine.Amount--;

    if (chosenOrderLine.Amount == 0)
        updOrder.Items.Remove(chosenOrderLine);
}

_orderService.UpdateOrder(updOrder);

Order = _orderService.GetOrder(orderId);

return Page();
}

public IActionResult OnPostIncrease(int itemId, int orderId)
{
    Order = _orderService.GetOrder(orderId);
    Models.Order updOrder = CloneOrder(Order);
    OrderLine? chosenOrderLine = updOrder.GetOrderLine(itemId);

    if (chosenOrderLine != null)
        chosenOrderLine.Amount++;
    else
    {
        Models.Item chosenItem = _itemService.GetItem(itemId);
        updOrder.Items.Add(new OrderLine(chosenItem, 1));
    }

    _orderService.UpdateOrder(updOrder);

    Order = _orderService.GetOrder(orderId);

    return Page();
}

public IActionResult OnPostAdd(int orderId)
{
    return OnPostIncrease(int.Parse(ChosenItem ?? "0"), orderId);
}

private Models.Order CloneOrder(Models.Order order)
{
    Models.Order clone = new Models.Order(order.Id, order?.Customer);
    clone.Remark = order?.Remark;
    clone.Items = new List<OrderLine>(order?.Items ?? new List<OrderLine>());

    return clone;
}
}

```

Til sidst skal vi have sat knapper ind til at kalde **EditOrder** og **DeleteOrder** fra **GetAllOrders**:

```

<td>
  <a class="btn btn-primary btn-sm" type="button" title="Edit" asp-page="EditOrder"
    asp-route-id="@order.Id"><i class="fa fa-pencil"></i></a>
  <a class="btn btn-danger btn-sm" type="button" title="Delete" asp-page="DeleteOrder"
    asp-route-id="@order.Id"><i class="fa fa-trash"></i></a>
</td>

```

Trin 10 (Afprøv)

Afprøv om du kan oprette og redigere ordre som angivet i trin 9. **NB:** det kan sagtens være, at du vil opdage nogle fejl i koden, da den ikke er blevet testet specielt grundigt ☹️.

ItemRazor.8 (Refactoring - Repositories)

Dette er den ottende opgave i opgave-serien **ItemRazor**.

I opgaverne 1 til 7 har vi gennemført en fuld implementering af CRUD-funktioner for tre domæne-klasser, hvor den sidste var mere kompleks. Vi har ofte fulgt princippet om copy/paste og efterfølgende tilretning, hvilket ikke er så hensigtsmæssigt på den lange bane.

I denne opgave er målet at få udført noget **refaktorisering**, i første omgang udelukkende på repository-delen. Med andre ord skal vi ikke lave ny funktionalitet, men derimod forbedre kode-strukturen. Dette gøres ved at indføre nogle interfaces og base-klasser, der benytter sig af **Generics** (type-parameterisering).

Udgangspunktet er løsningen fra opgave 7.

Trin 1 (Interfaces)

Lav en ny folder med navnet **Repositories** under **Services**-folderen. Lav derefter en ny folder **Interfaces** under **Repositories**-folderen. I denne folder tilføjer vi fire interfaces (koden følger bagefter). Husk at et interface kan tilføjes ved at højreklikke på folderen, og vælge **Add > New Item**, og derefter vælge "Interface" i den dialog der kommer frem.

- **IHasId**: Et meget lille interface, som alle vores domæne-klasser nu skal implementere (det gør de faktisk i forvejen)
- **IUpdateFromOther**: Også et meget lille interface, som alle vores domæne-klasser nu skal implementere. Skal bruge i.f.m. **Update**-funktionen.
- **IRepository**: Et generelt, type-parameteriseret interface for et repository.
- **IJsonFileRepository**: Et generelt, type-parameteriseret interface for et repository baseret på brug af JSON-formatet og tekst-filer.

```
public interface IHasId
{
    int Id { get; set; }
}
```

```
public interface IUpdateFromOther<T>
{
    void Update(T tOther);
}
```

```
public interface IRepository<T> where T : class, IHasId, IUpdateFromOther<T>
{
    int Count { get; }

    IEnumerable<T> GetAll();
    T Create(T t);
    T? Read(int id);
    void Update(int id, T t);
    T? Delete(int id);
    IEnumerable<T> Search(string str);
}
```

```

public interface IJsonFileRepository<T> where T : class
{
    string JsonFileRelative { get; set; }
    string DataRoot { get; set; }

    void SaveToJsonFile(IEnumerable<T> list);
    IEnumerable<T> GetFromJsonFile();
}

```

Trin 2 (Base-klasser)

Lav en ny folder med navnet **Base** under **Repositories**-folderen. Her implementerer vi to klasser:

- **JsonFileRepositoryBase**: Generel, type-parameteriseret implementation af **IJsonFileRepository**-interfacet.
- **RepositoryBase**: En generel, type-parameteriseret implementation af et repository som implementerer **IRepository**, som kan gemme – og efterfølgende hente – data på JSON-format i en tekst-fil.

```

public class JsonFileRepositoryBase<T> : IJsonFileRepository<T> where T : class
{
    public string JsonFileRelative { get; set; }
    public string DataRoot { get; set; } = "Data";
    public IWebHostEnvironment WebHostEnvironment { get; }

    public JsonFileRepositoryBase(IWebHostEnvironment webHostEnvironment, string jsonFileRelative)
    {
        WebHostEnvironment = webHostEnvironment;
        JsonFileRelative = jsonFileRelative;
    }

    private string JsonFileName
    {
        get { return Path.Combine(WebHostEnvironment.WebRootPath, DataRoot, JsonFileRelative); }
    }

    public void SaveToJsonFile(IEnumerable<T> list)
    {
        using FileStream jsonFileWriter = File.Create(JsonFileName);
        Utf8JsonWriter jsonWriter = new Utf8JsonWriter(jsonFileWriter, new JsonWriterOptions()
        {
            SkipValidation = false,
            Indented = true
        });
        JsonSerializer.Serialize(jsonWriter, list.ToArray());
    }

    public IEnumerable<T> GetFromJsonFile()
    {
        using StreamReader jsonFileReader = File.OpenText(JsonFileName);
        return JsonSerializer.Deserialize<T[]>(jsonFileReader.ReadToEnd()) ?? new T[0];
    }
}

```

```

public abstract class RepositoryBase<T> : IRepository<T>
    where T : class, IHasId, IUpdateFromOther<T>
{
    protected Dictionary<int, T> _data;
    protected IJsonFileRepository<T> _jsonRepo;

    public RepositoryBase(IJsonFileRepository<T> jsonRepo)
    {
        _jsonRepo = jsonRepo;
        _data = Load();
    }

    public int Count { get { return _data.Count; } }

    public T Create(T t)
    {
        t.Id = NextId();
        _data[t.Id] = t;

        Save();

        return t;
    }

    public T? Read(int id)
    {
        return _data.ContainsKey(id) ? _data[id] : null;
    }

    public void Update(int id, T t)
    {
        if (t != null && _data.ContainsKey(id))
        {
            _data[id].Update(t);
            Save();
        }
    }

    public T? Delete(int id)
    {
        T? toBeDeleted = Read(id);

        if (toBeDeleted != null)
        {
            _data.Remove(id);
            Save();
        }

        return toBeDeleted;
    }

    public IEnumerable<T> GetAll()
    {
        return _data.Values.OrderBy(t => t.Id);
    }

    public IEnumerable<T> Search(string str)
    {
        return GetAll().Where(t => SearchMatch(t, str));
    }

    protected abstract bool SearchMatch(T t, string str);

    private int NextId()
    {
        return Count > 0 ? GetAll().Select(t => t.Id).Max() + 1 : 1;
    }
}

```

```

private Dictionary<int, T> Load()
{
    return _jsonRepo.GetFromJsonFile().ToDictionary(t => t.Id, t => t);
}

private void Save()
{
    _jsonRepo.SaveToJsonFile(_data.Values);
}
}

```

Bemærk, at koden indeholder flere elementer som vi ikke har set før; det er ikke så vigtigt her, det vigtige er at se hvilke principper denne refaktorisering følger.

Trin 3 (Nye repository-interfaces)

Vi bibeholder at definere specifikke interfaces svarende til hver domæne-klasse, men disse bliver meget små nu. Faktisk er det et af disse nye interfaces, der rummer egne metoder.

Lav en ny folder med navnet **Model** under **Repositories**-folderen. Her definerer vi tre nye interfaces for de type-specifikke repositories:

```

public interface IItemRepository : IRepository<Item>
{
    IEnumerable<Item> PriceFilter(int maxPrice, int minPrice = 0);
}

public interface ICustomerRepository : IRepository<Customer>
{
}

public interface IOrderRepository : IRepository<Order>
{
}

```

Trin 4 (Nye repository-klasser)

De nye type-specifikke repository-klasser bliver ligeledes ganske små. Tilføj disse tre nye klasser til **Model**-folderen, som blev oprettet i sidste trin:

```

public class ItemRepository : RepositoryBase<Item>, IItemRepository
{
    public ItemRepository(IWebHostEnvironment WebHostEnvironment)
        : base(new JsonFileRepositoryBase<Item>(WebHostEnvironment, "Items.json"))
    {}

    public IEnumerable<Item> PriceFilter(int maxPrice, int minPrice = 0)
    {
        return GetAll().Where(t => (minPrice == 0 && t.Price <= maxPrice) ||
                                   (maxPrice == 0 && t.Price >= minPrice) ||
                                   (t.Price >= minPrice && t.Price <= maxPrice)
        );
    }

    protected override bool SearchMatch(Item t, string str)
    {
        return string.IsNullOrEmpty(str) || t?.Name?.ToLower().Contains(str.ToLower()) == true;
    }
}

```

```

public class CustomerRepository : RepositoryBase<Customer>, ICustomerRepository
{
    public CustomerRepository(IWebHostEnvironment WebHostEnvironment)
        : base(new JsonFileRepositoryBase<Customer>(WebHostEnvironment, "Customers.json"))
    {}

    protected override bool SearchMatch(Customer t, string str)
    {
        return string.IsNullOrEmpty(str) || t?.Name?.ToLower().Contains(str.ToLower()) == true;
    }
}

public class OrderRepository : RepositoryBase<Order>, IOrderRepository
{
    public OrderRepository(IWebHostEnvironment WebHostEnvironment)
        : base(new JsonFileRepositoryBase<Order>(WebHostEnvironment, "Orders.json"))
    {}

    protected override bool SearchMatch(Order t, string str)
    {
        return string.IsNullOrEmpty(str) ||
            t.Customer?.Name?.ToLower().Contains(str.ToLower()) == true;
    }
}

```

Trin 5 (Registrering af nye repositories som services)

Dette gøres som vi efterhånden har set det nogle gange, i **Program.cs**:

```

builder.Services.AddSingleton<IItemRepository, ItemRepository>();
builder.Services.AddSingleton<ICustomerRepository, CustomerRepository>();
builder.Services.AddSingleton<IOrderRepository, OrderRepository>();

```

Husk at sætte denne kode lige efter linjen **builder.Services.AddRazorPages();**

Trin 6 (Skift fra at bruge I...Service til at bruge I...Repository på alle Pages)

Nu er vi klar til at tage disse nye repositories i brug. Det gøres ved at skifte alle referencer til de gamle interface-typer ud med de nye, d.v.s:

- Udskift **IItemService** med **IItemRepository**
- Udskift **ICustomerService** med **ICustomerRepository**
- Udskift **IOrderService** med **IOrderRepository**

Den endelige test er jo så at slette alle de gamle **...Service** interfaces og klasser fra projektet, og se at det hele virker som før 😊.

ItemRazor.9 (Refactoring – Page Models)

Dette er den niende opgave i opgave-serien **ItemRazor**.

I opgaverne 1 til 8 har vi gennemført en fuld implementering af CRUD-funktioner for tre domæne-klasser, og har desuden gennemført en refaktorisering af repository-delen.

I denne opgave er målet at få udført yderligere refaktorisering, nu med fokus på "page-model" klasserne, d.v.s. de C#-klasser, som hver især udgør den ene halvdel af en *Razor Page*. Det er således de klasser, som ligger i **.cshtml.cs**-filerne under **.cshtml**-filerne. For hver domæne-klasse er der således fire af disse klasser (**GetAll...**, **Create...**, **Delete...**, **Update...**). Ser man lidt nærmere på disse klasser, vil man opdage betydelige muligheder for at benytte base-klasser. Vi skal derfor definere og benytte en antal nye base-klasser, der igen benytter sig af **Generics** (type-parameterisering).

Udgangspunktet er løsningen fra opgave 8.

Trin 1 (Definition af PageModelAppBase)

Som det første indfører vi en base-klasse på øverste niveau, som skal være den helt grundlæggende base-klasse for (næsten) alle vores "page-model" klasser.

Lav en ny folder ved navn **Base** i projektet, på niveau med **Models** og **Pages**.

Tilføj en ny klasse **PageModelAppBase** til Base-folderen:

```
public class PageModelAppBase<TRepo> : PageModel
{
    protected TRepo _repository;

    public PageModelAppBase(TRepo repository)
    {
        _repository = repository;
    }
}
```

Intentionen med denne klasse er at modellere det som (næsten) alle "page-model" klasser har tilfælles, nemlig at de har en reference til et repository, og at denne reference bliver initialiseret via **Dependency Injection** i constructoren.

Overvej: Burde denne klasse være **abstract**? Kan vi være sikre på, at **TRepo** altid vil være en reference til et repository? Betyder det egentlig noget, om vi er sikre på det?

Trin 2 (Definition af GetAllPageModelBase)

Nu definerer vi et lidt mere specialiseret klasse **GetAllPageModelBase**, som skal være baseklasse for alle "page model"-klasser hørende til **GetAll**-pages.


```

public class GetAllPageModelBase<TData, TRepo> : PageModelAppBase<TRepo>
    where TData : class, IHasId, IUpdateFromOther<TData>
    where TRepo : IRepository<TData>
{
    public List<TData> Data { get; protected set; }

    public GetAllPageModelBase(TRepo repository) : base(repository) { }

    public virtual void OnGet()
    {
        Data = _repository.GetAll().ToList();
    }
}

```

Bemærk, at både data-typen (f.eks **Customer**) og repository-typen (f.eks. **ICustomerRepository**) er type-parametre til denne klasse. Der stilles også noget større krav til begge disse typer, m.h.t. hvilke interfaces de skal implementere. Bemærk også, at klassen arver fra **PageModelAppBase**.

Overvej: Hvor kommer inspirationen til koden i denne klasse mon fra? Prøv evt at se på koden i klasserne **GetAllItemsModel**, **GetAllCustomersModel** og **GetAllOrdersModel**.

Trin 3 (Definition af CreatePageModelBase)

På tilsvarende vis definerer vi nu klassen **CreatePageModelBase**, som skal være baseklasse for alle ”page model”-klasser hørende til **Create**-pages.

```

public class CreatePageModelBase<TData, TRepo> : PageModelAppBase<TRepo>
    where TData : class, IHasId, IUpdateFromOther<TData>
    where TRepo : IRepository<TData>
{
    private string _onPostRedirectPage;

    [BindProperty]
    public TData Data { get; set; }

    public CreatePageModelBase(TRepo repository, string onPostRedirectPage) : base(repository)
    {
        _onPostRedirectPage = onPostRedirectPage;
    }

    public virtual IActionResult OnGet()
    {
        return Page();
    }

    public virtual IActionResult OnPost()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _repository.Create(Data);

        return RedirectToPage(_onPostRedirectPage);
    }
}

```

Overvej: Hvad formål tjener instance field **_onPostRedirectPage**? Hvorfor er **OnGet** og **OnPost** erklæret som **virtual**?

Trin 4 (Definition af DeletePageModelBase)

Igen på tilsvarende vis definerer vi nu klassen **DeletePageModelBase**, som skal være baseklasse for alle "page model"-klasser hørende til **Delete**-pages.

```
public class DeletePageModelBase<TData, TRepo> : PageModelAppBase<TRepo>
    where TData : class, IHasId, IUpdateFromOther<TData>
    where TRepo : IRepository<TData>
{
    private string _onPostRedirectPage;

    [BindProperty]
    public TData Data { get; set; }

    public DeletePageModelBase(TRepo repository, string onPostRedirectPage)
        : base(repository)
    {
        _onPostRedirectPage = onPostRedirectPage;
    }

    public virtual IActionResult OnGet(int id)
    {
        TData? data = _repository.Read(id);

        if (data == null)
            return RedirectToPage("/NotFound");

        Data = data;
        return Page();
    }

    public IActionResult OnPost()
    {
        TData? deletedData = _repository.Delete(Data.Id);

        if (deletedData == null)
            return RedirectToPage("/NotFound");

        return RedirectToPage(_onPostRedirectPage);
    }
}
```

Overvej: Kan det være et problem, at **"/NotFound"** står direkte i koden?

Trin 5 (Definition af EditPageModelBase)

Som den sidste base-klasse definerer vi nu klassen **EditPageModelBase**, som skal være baseklasse for alle "page model"-klasser hørende til **Edit**-pages.

```
public class EditPageModelBase<TData, TRepo> : PageModelAppBase<TRepo>
    where TData : class, IHasId, IUpdateFromOther<TData>
    where TRepo : IRepository<TData>
{
    private string _onPostRedirectPage;

    [BindProperty]
    public TData Data { get; set; }
```

```

public EditPageModelBase(TRepo repository, string onPostRedirectPage)
    : base(repository)
{
    _onPostRedirectPage = onPostRedirectPage;
}

public IActionResult OnGet(int id)
{
    TData? data = _repository.Read(id);

    if (data == null)
        return RedirectToPage("/NotFound");

    Data = data;
    return Page();
}

public IActionResult OnPost()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _repository.Update(Data.Id, Data);

    return RedirectToPage(_onPostRedirectPage);
}
}

```

Overvej: Hvorfor skal **OnGet** have en parameter **int id**? Hvor kommer denne værdi mon fra?

Trin 6 (Brug af de nye base-klasser)

Nu er vi klar til at gennemgå alle de 12 "page-model" klasser, og se i hvor høj grad vi kan bruge vores nye base-klasser. Dette må vi selv arbejde dig igennem, klasse for klasse... I flere tilfælde kan vi stort set slette alt den oprindelige kode, som det f.eks. er tilfældet for **DeleteItemModel**:

```

public class DeleteItemModel : DeletePageModelBase<Models.Item, IItemRepository>
{
    public DeleteItemModel(IItemRepository repository)
        : base(repository, "GetAllItems")
    {
    }
}

```

I andre tilfælde kan vi godt bruge base-klasserne, men der er også dele af den oprindelige kode, som skal bevares. Dette gælder f.eks. for alle **GetAll**-klasserne (men det åbner måske mulighed at udvide base-klassen...? Det kan du overveje). Endelige er der også et par tilfælde – specifikt for klasserne hørende til **Order** – som er så specialiserede, at vi ikke umiddelbart har gavn af at bruge base-klasserne. Det vil du nok opdage, som du arbejder dig gennem klasserne.

NB: Bemærk, at disse ændringer også vil gøre et nødvendigt at lave nogle mindre ændringer i layout-koden (i **.cshtml**-filerne). Det vil compilatoren nok gøre dig opmærksom på undervejs 😊.

Trin 7 (Afprøv)

Når din kode kan compilere, kan du afprøve om alt virker helt som før. Husk at en refaktorisering ikke skal ændre på funktionaliteten, kun på kodens struktur, forhåbentligt til det bedre.

Tillykke, du er nået igennem hele opgavesættet 😊.