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: Opgaverne 7-9 er IKKE færdiggjort endnu.

Bemærk også, at opgaverne generelt fokuserer mere på funktionalitet end på visuel finish. Fokus er på frameworket som sådan, samt hvoedan 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 enopgave 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](#_Toc118663751)

[ItemRazor.2 (Create new Items) 5](#_Toc118663752)

[ItemRazor.3 (Validation, Search and Filter Items) 10](#_Toc118663753)

[ItemRazor.4 (Edit and Delete Items) 16](#_Toc118663754)

[ItemRazor.5 (JSON-file, Background Image, Icons and more Bootstrap features) 24](#_Toc118663755)

[ItemRazor.6 (Complete implementation for new domain class) 34](#_Toc118663756)

[ItemRazor.7 (TBD) 35](#_Toc118663757)

[ItemRazor.8 (TBD) 39](#_Toc118663758)

[ItemRazor.9 (TBD) 40](#_Toc118663759)

# 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:

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

## 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 a la:

@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* **<li>** 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 refaktorering, 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 vhj 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:

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Bemærk: Når der klikkes på et input feldt 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:

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

# ItemRazor.3 (Validation, Search and Filter Items)

Dette er den tredie 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 **<span>**-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:

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

## 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:

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

## 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 **IItemsService** 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:

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

# 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(IItemService 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>

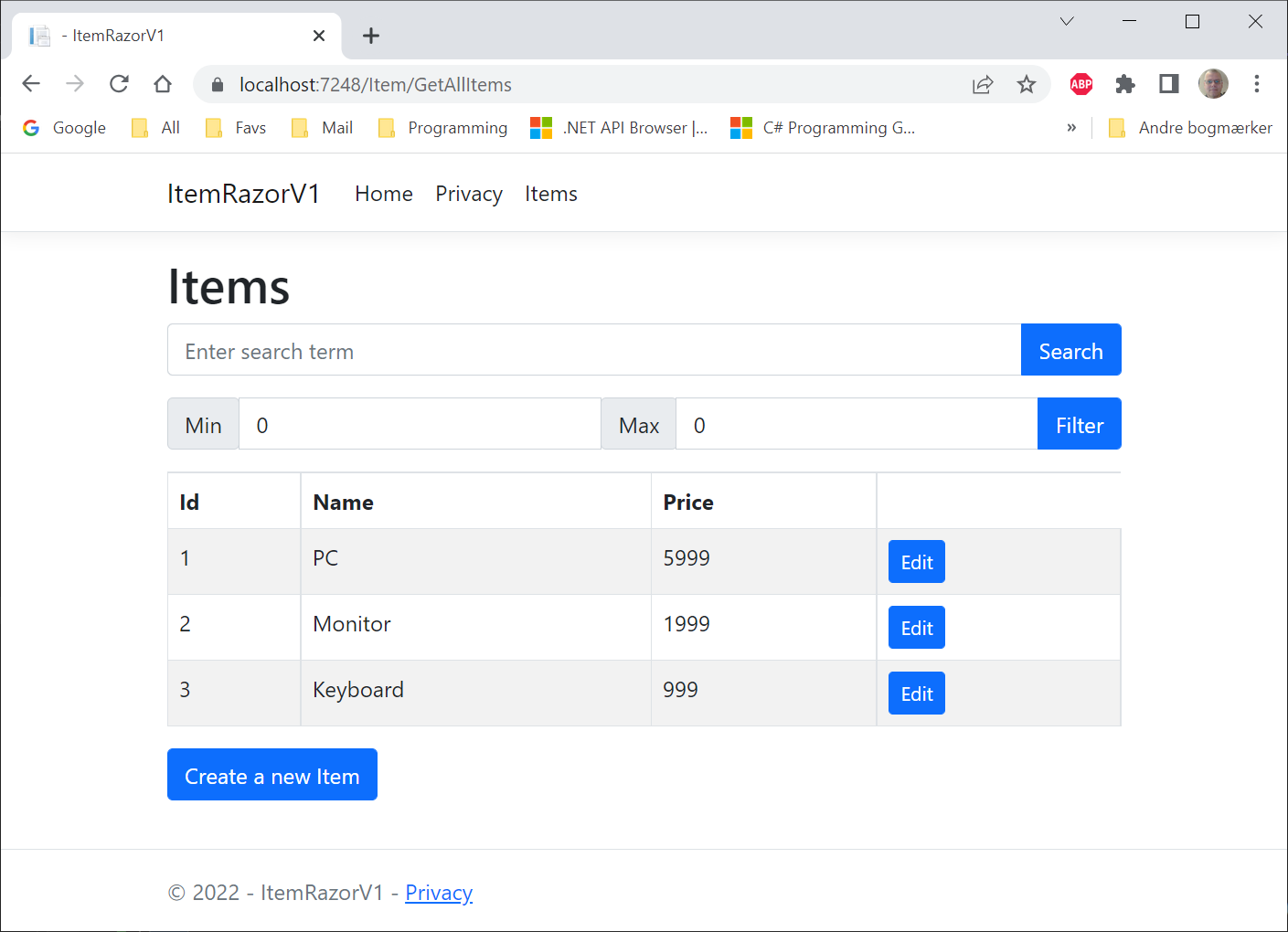
<**a** class="btn btn-primary btn-sm" type="button" title="Edit"

**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



Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Det er nu muligt at lave **CRU** operationerne (**C**reate, **R**ead og **U**pdate) på **Items**. Nu skal vi have implementeret **D**elete.

## 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:

Et billede, der indeholder bord

Automatisk genereret beskrivelse

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

# 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** (**J**ava**S**cript **O**bject **N**otation) 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 "dekorere" **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 JsonWriterOptions()

{

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 stadigt 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:

Et billede, der indeholder tastatur, elektronik, indendørs, computer

Automatisk genereret beskrivelse

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:

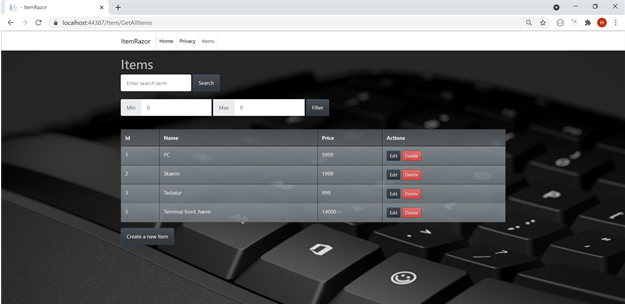
<body class="d-flex flex-column min-vh-100">

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** (colums) 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 kolonners 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):

Et billede, der indeholder tekst, monitor, elektronik, indendørs

Automatisk genereret beskrivelse

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-awsome* 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):

Et billede, der indeholder tekst, sort, elektronik, skærmbillede

Automatisk genereret beskrivelse

# 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 (TBD)

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.

**TBD - Disse trin skal beskrives mere detaljeret**

## 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/DeleteOrder Razor Pages - skelet)

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

**TBD – Disse skal detaljeres**

Tilføj Razor Page ViewOrder i folder Pages/Order (vent med udfyldning)

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

AFPRØV

Udfyld Razor Pages Create/View/DeleteOrder i folder Pages/Order (kode)

Tilføj CreateOrder button til GetAllCustomers (kode)

AFPRØV (Husk swap mellem MockData og JSON i CustomerService)

# ItemRazor.8 (TBD)

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

TBD (skal være med fokus på at refaktorisere hen imod brug af mere generelle Repository-klasser)

Udgangspunktet er løsningen fra opgave 7.

# ItemRazor.9 (TBD)

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

TBD (skal være med fokus på udestående refaktoriseringer)

Udgangspunktet er løsningen fra opgave 8.