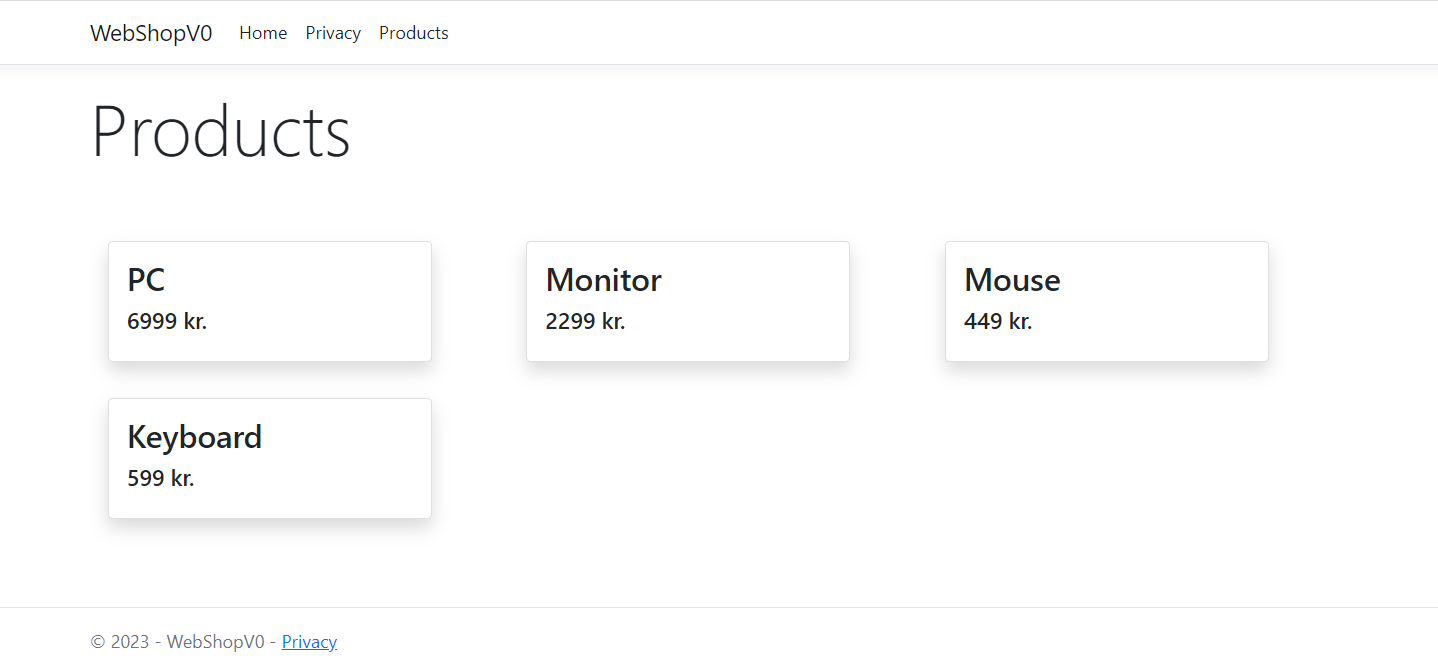
Razor Pages repetitionsopgave

I denne opgave laver vi et ganske minimalt *Razor Pages* projekt fra bunden, og får lavet en enkelt Page der kan vise et antal ”produkter” som *cards*:



## Trin 1 (WebShopRP)

Opret et nyt projekt af typen: **ASP.NET Core Web App** med navnet: **WebShopRP**. Under punktet *Framework* vælges **.NET 9.0**.

## Trin 2 (Afprøv)

Kør appen og verificér at start-siden vises i din browser.

## Trin 3 (Mappen Models)

Tilføj en ny folder med navnet **Models**. Hint: højreklik på projektet, og vælg *Add -> New Folder*.

## Trin 4 (Klassen Product)

Tilføj en ny klasse **Product** til mappen **Models**. Klassen **Product** skal have følgende properties: **Id** (int), **Name** (string) og **Price** (double) samt en passende constructor. Alle tre properties skal blot have en **get**. Efter temperament kan man vælge at fjerne det omsluttende namespace.

## Trin 5 (Mappen Pages/Products)

Tilføj en ny folder med navnet **Products** til mappen **Pages**

## Trin 6 (All)

Tilføj en ny *Razor Page* (af typen *empty*) til mappen **Products** (højre-klik på mappen). Giv siden navnet **All**.

## Trin 7 (All.cshtml.cs)

Tilføj en property **List<Product> Data** til klassen **AllModel** i filen **All.cshtml.cs**. **Data** skal initialiseres med en liste der indeholder et antal **Product**-objekter:

public List<Product> Data { get; } = new List<Product>

{

new Product(1, "PC", 6999),

new Product(2, "Monitor", 2299),

new Product(3, "Mouse", 449),

new Product(3, "Keyboard", 599)

};

## Trin 8 (All.cshtml)

Indsæt kode i **All.cshtml** der viser alle **Product**-objekterne fra **Data**-listen som et antal Cards:

<h1 class="display-3 mb-5">Products</h1>

<div class="container">

<div class="row">

@foreach (var product in Model.Data)

{

<div class="col-md-4 p-3">

<div class="card shadow" style="width: 18rem;">

<div class="card-body d-flex flex-column">

<h3 class="card-title">@product.Name</h3>

<h5>@(product.Price) kr.</h5>

</div>

</div>

</div>

}

</div>

</div>

## Trin 9 (\_Layout.cshtml, under Pages/Shared)

Find det **<li>**-tag, der rummer et link til **/Privacy**. Lige under dette tag skal du nu indsætte et tilsvarende tag der router til den nye Page:

<li class="nav-item">

<**a** class="nav-link text-dark" **asp-area**="" **asp-page**="/Products/All">Products</**a**>

</li>

## Trin 10 (Afprøv)

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

## Trin 11 (Services)

Tilføj en ny folder til projektet (d.v.s. på samme niveau som **Models**) med navnet **Services**.

## Trin 12 (IProductDataService og ProductDataService)

I folderen **Services**, tilføj et nyt interface **IProductDataService** og en ny klasse **ProductDataService**:

public interface IProductDataService

{

List<Product> GetAll();

}

public class ProductDataService : IProductDataService

{

private List<Product> \_products = new List<Product>

{

new Product(1, "PC", 6999),

new Product(2, "Monitor", 2299),

new Product(3, "Mouse", 449),

new Product(4, "Keyboard", 599)

};

public List<Product> GetAll()

{

return \_products;

}

}

Igen kan man vælge at fjerne det omsluttende namespace.

## Trin 13 (All.cshtml.cs)

Fjern initialiseringen af **Data** – men bibehold **Data** selv, og tilføj en **private set** – i **AllModel**.

Tilføj også et instance field **\_productDataService** af typen **IProductDataService** til **AllModel**.

Tilføj en constructor til **AllModel**, som tager en parameter af typen **IProductDataService**, og bruger den til at initialisere **\_productDataService**.

Endelig skal du i **OnGet**-metoden nu bruge **\_productDataService** til at få initialiseret **Data** med **Product**-objekter (Hint: kald **GetAll**)

**AllModel** bør nu se således ud:

public class AllModel : PageModel

{

private IProductDataService \_productDataService;

public List<Product> Data { get; private set; }

public AllModel(IProductDataService productDataService)

{

\_productDataService = productDataService;

}

public void OnGet()

{

Data = \_productDataService.GetAll();

}

}

## Trin 14 (Program.cs)

Vi mangler at registrere vores produkt-dataservice som en egentlig ”service” i applikationen. Dette gøres ved – i **Program.cs** – at finde linjen **builder.Services.AddRazorPages();**, og under den tilføje:

builder.Services.AddSingleton<IProductDataService, ProductDataService>();

## Trin 15 (Afprøv)

Afprøv at vi stadig får vist vores produkter korrekt.

Hermed har vi faktisk lavet alle de ”lag”, der er typisk for en *Razor Pages* app:

* **Page**: er defineret i **.cshtml**-filer, kender en **PageModel**-klasse via **@Model**-direktivet, og kan dermed tilgå public properties/metoder i **PageModel** ved notationen **Model.(navn på property)**. Disse properties vil ofte rumme data som skal bruges til at generere en Page.
* **PageModel**: er defineret i **.cshtml.cs**-filer, stiller public properties/metoder til rådighed for Pages. Disse properties vil ofte rumme data som skal bruges til at generere en Page. Disse properties vil ofte blive initialiseret med data – som ofte hentes via en **Service** - i en **OnGet**-metode, **NB**: Hvis en Page har til formål at modtage data fra brugeren, vil dette ofte blive ”indfanget” med properties anno­teret med **[BindProperty]**, og behandlingen af data vil ske i en **OnPost**-metode.
* **Service**: er defineret i almindelige **.cs**-filer, typisk i en **Services**-folder. Rummer funktionalitet som understøtter generering af Pages; meget typisk er data services, som henter data fra en kilde, som f.eks. kan være en database. En service består som regel af et interface og (mindst) en klasse som implementerer interfacet. En service skal registres i **Program.cs**, og f.eks. PageModel-klasser kan derefter få adgang til en service via dependency injection i deres constructor.

## Trin 16 (Create)

Nu vil vi gøre det muligt at oprette nye **Products**. Vi skal derfor lave en **Create**-page. Dette kræver følgende ændringer af koden:

* Tilpasning af **Product**-klassen
* Tilpasning af **IProductDataService** og **ProductDataService**
* Oprettelse af **Create**-page
* Implementation af **CreateModel**
* Implementation af **Create**-page
* Tilpasning af **All**-page

## Trin 17 (Create – tilpasning af **Product**-klassen)

I **Product**-klassen skal du:

* Tilføje en **set** til alle properties.
* Tilpasse din constructor, så en ikke længere tager **id** som parameter. Constructoren skal blot sætte property **Id** til 0.
* Tilføje en default constructor, d.v.s. en constructor der ikke tager nogen parametre.

Herefter kommer **Product** til at se således ud:

public class Product

{

public int Id { get; set; }

public string Name { get; set; }

public double Price { get; set; }

public Product() { }

public Product(string name, double price)

{

Id = 0;

Name = name;

Price = price;

}

}

## Trin 18 (Create – tilpasning af **data-service**)

I **IProductDataService**-interfacet skal du tilføje en metode **Create**:

int Create(Product product);

Opdatér **ProductDataService**, så den ser således ud:

public class ProductDataService : IProductDataService

{

private Dictionary<int, Product> \_products;

public ProductDataService()

{

\_products = new Dictionary<int, Product>();

Create(new Product("PC", 6999));

Create(new Product("Monitor", 2299));

Create(new Product("Mouse", 449));

Create(new Product("Keyboard", 599));

}

public List<Product> GetAll()

{

return \_products.Values.ToList();

}

public int Create(Product product)

{

product.Id = NextId();

\_products[product.Id] = product;

return product.Id;

}

private int NextId() => \_products.Keys.DefaultIfEmpty(0).Max() + 1;

}

Bemærk, at vores data service nu har overtaget ansvaret for at finde et unikt **Id** til vores **Products**. Vi skal egentlig ikke bruge **Id’et** endnu, men vores data-service er nu lidt mere fremtidssikret. Vi har også skiftet til at bruge en **Dictionary**, hvilket gør det meget nemt og effektivt at søge efter et **Product** med et bestemt **Id**.

**NB**: Bemærk, at selv om vi nu kan oprette et nyt **Product**, bliver det ikke gemt permanent på nogen måde, så hvis vi starter appen igen, er vi tilbage til de fire **Products**, vi opretter i constructoren.

## Trin 19 (Create – Opret Create-page)

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

## Trin 20 (Create – Implementér CreateModel)

I **Create.cshtml.cs** skal vi nu implementere **CreateModel** korrekt. Den får også brug for en reference til vores data-service, men får også brug for en property af typen **Product**, som skal bruges til **data binding**. Samlet set kommer **CreateModel** til at se således ud:

public class CreateModel : PageModel

{

private IProductDataService \_productDataService;

[BindProperty]

public Product Data { get; set; } = new Product();

public CreateModel(IProductDataService productDataService)

{

\_productDataService = productDataService;

}

public IActionResult OnPost()

{

// Tjek om det indtastede data er validt

if (!ModelState.IsValid)

{

return Page();

}

// Send data videre til data-service

\_productDataService.Create(Data);

// Vend tilbage til oversigten.

return RedirectToPage("All");

}

}

Bemærk vores **Data**-property af typen **Product**, som har annoteringen **[BindProperty]**. Denne gør det muligt at ”binde” denne property sammen med vores Page-kode (se senere).

Bemærk også implementationen af **OnPost**. Denne kode vil blive kørt, når vi ”submitter” den form (se senere) hvor vi taster data ind.

## Trin 21 (Create – Implementér Create-page)

I **Create.cshtml.** skal vi nu implementere **Create**-page korrekt. Tilføj denne kode til **Create.cshtml,** under den øverste sektion med @-direktiverne:

<h1 class="display-3 mb-5">Create New Product</h1>

<**form** method="post">

<div class="form-group mb-3">

<**label** **asp-for**="@Model.Data.Name" class="control-label"></**label**>

<**input** **asp-for**="@Model.Data.Name" class="form-control" />

</div>

<div class="form-group mb-3">

<**label** **asp-for**="@Model.Data.Price" class="control-label"></**label**>

<**input** **asp-for**="@Model.Data.Price" class="form-control" />

</div>

<p>

<div class="form-group mb-3">

<input type="submit" value="Create" class="btn btn-outline-primary" />

</div>

</p>

</**form**>

De to første **input**-forms – som blot er tekst-bokse – benytter data binding til at binde til hhv. **Name**- og **Price**-property på det **Product**-objekt som **Data** refererer til.

Den sidste **input**-form er en **submit-button**. Når man klikker på den, vil **OnPost** i **CreateModel** blive kørt. Det indtastede data bliver valideret og sendt videre til vores data-service.

## Trin 22 (Create – Tilpas All-page)

Vores **Create**-page er færdig, men vi kan ikke navigere hen til den… Tilføj derfor følgende kode nederst på **All**-page:

<p>

<**a** class="btn btn-outline-primary mt-3" **asp-page**="Create"> Create a new Product </**a**>

</p>

Nu kan man navigere hen til **Create**-page fra bunden af **All**-page. Afprøv det 😊.

## Trin 23 (Create – Validering)

Lige nu kan man f.eks. angive en negativ pris for et **Product**. Hvis vi gerne vil udføre noget validering af det indtastede data, kræver det to ændringer. Først skal de to properties **Name** og **Price** i **Product** ændres:

[Required(ErrorMessage = "Der skal angives et navn")]

public string Name { get; set; }

[Range(0, 99999.99, ErrorMessage = "Pris skal være mellem 0 og 100.000")]

public double Price { get; set; }

Derefter skal de to første **div**-tags i **Create**-page ændres:

<div class="form-group mb-3">

<**label** **asp-for**="@Model.Data.Name" class="control-label"></**label**>

<**input** **asp-for**="@Model.Data.Name" class="form-control" />

<**span** **asp-validation-for**="@Model.Data.Name" class="text-danger"></**span**>

</div>

<div class="form-group mb-3">

<**label** **asp-for**="@Model.Data.Price" class="control-label"></**label**>

<**input** **asp-for**="@Model.Data.Price" class="form-control" />

<**span** **asp-validation-for**="@Model.Data.Price" class="text-danger"></**span**>

</div>

Nu får vi udført validering! Prøv at eksperimentere lidt med det.

## Trin 24 (Delete)

Nu vil vi gøre det muligt at slette eksisterende **Products**. Vi skal derfor lave en **Delete**-page. Dette kræver følgende ændringer af koden:

* Tilpasning af **IProductDataService** og **ProductDataService**
* Oprettelse af **Delete**-page
* Implementation af **DeleteModel**
* Implementation af **Delete**-page
* Tilpasning af **All**-page

## Trin 25 (Delete – tilpasning af **data service**)

I **IProductDataService**-interfacet skal du tilføje to metoder **Read** og **Delete**:

Product? Read(int id);

bool Delete(int id)

I **ProductDataService** skal du også tilføje to metoder **Read** og **Delete**:

public Product? Read(int id)

{

return \_products.ContainsKey(id) ? \_products[id] : null;

}

public bool Delete(int id)

{

return \_products.Remove(id);

}

## Trin 26 (Delete – Opret Delete-page)

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

## Trin 27 (Delete – Implementér DeleteModel)

I **Delete.cshtml.cs** skal vi nu implementere **DeleteModel** korrekt. Den får også brug for en reference til vores data-service, og også brug for en property af typen **Product**, som skal bruges til **data binding**. Samlet set kommer **DeleteModel** til at se således ud:

public class DeleteModel : PageModel

{

private IProductDataService \_productDataService;

[BindProperty]

public Product Data { get; set; }

public DeleteModel(IProductDataService productDataService)

{

\_productDataService = productDataService;

}

public virtual IActionResult OnGet(int id)

{

Product data = \_productDataService.Read(id);

if (data == null)

return RedirectToPage("Error");

Data = data;

return Page();

}

public virtual IActionResult OnPost()

{

\_productDataService.Delete(Data.Id);

return RedirectToPage("All");

}

}

Bemærk, at vi i **OnGet** – som tager en parameter, der ser ud til at være et **Id** for et **Product** – søger det **Product** frem, som muligvis skal slettes. Hvis sådan et **Product** findes, initialiseres **Data** til at referere til det. Vi vender tilbage til, hvordan vi så rent faktisk udpeger et **Product** til sletning.

**OnPost** er rimeligt simpel, den sletter blot det udpegede **Product** ved at kalde **Delete** på vores data-service.

## Trin 28 (Delete – Implementér Delete-page)

I **Delete.cshtml.** skal vi nu implementere **Delete**-page korrekt. Tilføj denne kode til **Delete.cshtml,** under den øverste sektion med @-direktiverne:

<h1 class="display-3 mb-5">Vil du slette dette Product?</h1>

<div class="alert alert-danger">

<h5>Er du sikker på, at du vil slette @Model.Data.Name</h5>

<**form** method="post">

<input type="submit" value="Ja" class="btn btn-outline-danger mt-3">

<**a** class="btn btn-outline-primary mt-3" **asp-page**="/Products/All">Nej</**a**>

</**form**>

</div>

Ideen er, at brugeren skal have lov til at fortryde/bekræfte sletning, før den rent faktisk foretages. Derfor er der både en **submit-button** – som vil få **OnPost** til at blive kørt, hvorved sletning foretages – og et **<a>**-tag som navigerer tilbage til oversigten over **Products**. Så det er to forskellige former for kontroller, selv om de tager sig ens ud visuelt. Bemærk, at vi bruger data-binding til at kunne vise **Name** for det valgte **Product**.

## Trin 29 (Delete – Tilpas All-page)

Vi mangler at få implementeret en måde at kunne igangsætte en sletning på. Dette gøres ved at tilføje en *Delete*-knap på det Card, hvorpå det enkelte **Product** vises. I **All.cshtml** skal **card-body** derfor tilrettes således:

<div class="card-body d-flex flex-column">

<h3 class="card-title">@product.Name</h3>

<h5>@product.Price kr.</h5>

<div>

<**a** class="btn btn-outline-danger btn-sm mt-3"

type="button"

title="Delete"

**asp-page**="Delete"

**asp-route-id**="@product.Id">Delete

</**a**>

</div>

</div>

Dette **<a>**-tag navigerer til vores **Delete**-page, og – hvilket er meget vigtigt – parameteren **asp-route-id** søger for, at **OnGet** på **Delete**-page bliver kaldt med netop denne værdi som parameter! På denne måde udpeger vi, hvilket **Product** vi ønsker at slette, hvorefter **Delete**-page kan finde det frem i **OnGet**.

Afprøv, om du nu kan slette **Products** fra **All**-page, og at du kan fortryde undervejs 😊.

## Trin 30 (Edit)

Som sidste funktionalitet vil vi gøre det muligt at rette eksisterende **Products**. Vi skal derfor lave en **Edit**-page. Dette kræver følgende ændringer af koden:

* Tilpasning af **Product**-klassen
* Tilpasning af **IProductDataService** og **ProductDataService**
* Oprettelse af **Edit**-page
* Implementation af **EditModel**
* Implementation af **Edit**-page
* Tilpasning af **All**-page

## Trin 31 (Edit – tilpasning af **Product**-klassen)

I **Product**-klassen skal du tilføje en metode med navnet **Update**:

public void Update(Product other)

{

Name = other.Name;

Price = other.Price;

}

## Trin 32 (Edit – tilpasning af **data service**)

I **IProductDataService**-interfacet skal du tilføje en metode med navnet **Update**:

bool Update(int id, Product product);

I **ProductDataService** skal du også tilføje en metode med navnet **Update**:

public bool Update(int id, Product product)

{

Product? existingProduct = Read(id);

if (existingProduct == null)

return false;

existingProduct.Update(product);

return true;

}

## Trin 33 (Edit – Opret Edit-page)

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

## Trin 34 (Edit – Implementér EditModel)

I **Edit.cshtml.cs** skal vi nu implementere **EditModel** korrekt. Den får også brug for en reference til vores data-service, og også brug for en property af typen **Product**, som skal bruges til **data binding**. Samlet set kommer **EditModel** til at se således ud:

public class EditModel : PageModel

{

private IProductDataService \_productDataService;

[BindProperty]

public Product Data { get; set; }

public EditModel(IProductDataService productDataService)

{

\_productDataService = productDataService;

}

public virtual IActionResult OnGet(int id)

{

Product data = \_productDataService.Read(id);

if (data == null)

return RedirectToPage("Error");

Data = data;

return Page();

}

public virtual IActionResult OnPost()

{

// Validate data entered by user

if (!ModelState.IsValid)

{

return Page();

}

// Submit data to data service

\_productDataService.Update(Data.Id, Data);

return RedirectToPage("All");

}

}

Hvis man sammenligner denne med de andre PageModel-klasser, vil man se at **OnGet** minder rigtig meget on **OnGet** for **DeleteModel** (de er faktisk helt identiske, hvorfor egentlig det?), mens **OnPost** minder rigtig meget om **OnPost** for **CreateModel** (hvilke forskelle er der?)

## Trin 35 (Edit – Implementér Edit-page)

I **Edit.cshtml.** skal vi nu implementere **Edit**-page korrekt. Tilføj denne kode til **Edit.cshtml,** under den øverste sektion med @-direktiverne:

<h1 class="display-3 mb-5">Edit Product</h1>

<**form** method="post">

<div class="form-group mb-3" hidden>

<**label** **asp-for**="@Model.Data.Id" class="control-label"></**label**>

<**input** **asp-for**="@Model.Data.Id" class="form-control" readonly="@(true)" />

</div>

<div class="form-group mb-3">

<**label** **asp-for**="@Model.Data.Name" class="control-label"></**label**>

<**input** **asp-for**="@Model.Data.Name" class="form-control" />

<**span** **asp-validation-for**="@Model.Data.Name" class="text-danger"></**span**>

</div>

<div class="form-group mb-3">

<**label** **asp-for**="@Model.Data.Price" class="control-label"></**label**>

<**input** **asp-for**="@Model.Data.Price" class="form-control" />

<**span** **asp-validation-for**="@Model.Data.Price" class="text-danger"></**span**>

</div>

<p>

<div class="form-group mb-3">

<input type="submit" value="Save" class="btn btn-outline-primary" />

</div>

</p>

</**form**>

Hvis man sammenligner med den tilsvarende kode for **Create**-page, vil man se at de også minder rigtigt meget om hinanden. Dog er der her en ekstra **<div>**, som refererer til **Id**, og denne **<div>** er endda sat som *hidden*… Først og fremmest: hvorfor vil det ikke give mening at tillade at man retter **Id** på et eksisterende **Product**? Hvis man ikke må det, hvorfor så overhovedet have denne **<div>** med? Det er lidt subtilt, men hvis den ikke er med, taber man så at sige forbindelsen mellem det **Id** som **OnGet** bliver kaldt med, og det **Id** man kan få fra **Data**-property i **OnPost**. Når du er færdig med opgaven, kan du prøve at fjerne denne **<div>**, og se hvilken effekt det har.

## Trin 36 (Edit– Tilpas All-page)

Vi mangler at få implementeret en måde at kunne igangsætte en tilretning på. Dette gøres – ligesom for **Delete** – ved at tilføje en *Edit*-knap på det Card, hvorpå det enkelte **Product** vises. I **All.cshtml** skal **card-body** derfor tilrettes således:

<div class="card-body d-flex flex-column">

<h3 class="card-title">@product.Name</h3>

<h5>@product.Price kr.</h5>

<div>

<**a** class="btn btn-outline-danger btn-sm mt-3"

type="button"

title="Delete"

**asp-page**="Delete"

**asp-route-id**="@product.Id">Delete

</**a**>

<**a** class="btn btn-outline-info btn-sm mt-3"

type="button"

title="Edit"

**asp-page**="Edit"

**asp-route-id**="@product.Id">Edit

</**a**>

</div>

</div>

Dette **<a>**-tag navigerer til vores **Edit**-page, og igen sørger parameteren **asp-route-id** for, at **OnGet** på **Edit**-page bliver kaldt med den rigtige værdi. På denne måde udpeger vi, hvilket **Product** vi ønsker at tilrette, hvorefter **Edit**-page kan finde det frem i **OnGet**.

Afprøv, om du nu kan tilrette **Products** fra **All**-page 😊.

## Trin 37 (Reflektion)

Nu er vi ved vejs ende for denne opgave. Vi har implementeret stort set komplet *CRUD*-funktionalitet for en enkelt domæne-klasse (data bliver dog ikke gemt permanent), med en rimelig – men ret elementær – UI. Et godt spørgsmål er nu:

**Hvor meget vil det kræve at gøre noget tilsvarende for en ny domæne-klasse, f.eks. en Customer-klasse?**

Hvis ellers den nye klasse også er rimeligt simpel, vil koden sikkert på rigtig mange måder minde om den kode vi nu har lavet. Måske så meget, at vi burde begynde at tænke på at indføre nogle mere generelle interfaces og klasser. Type-parameterisering og nedarvning vil nok også kunne bruges flere steder. Overvej f.eks. disse spørgsmål:

* Hvor meget forskel vil der være mellem en **Customer**-dataservice og en **Product**-dataservice?
* Hvor meget forskel vil der være mellem f.eks. en **Customer**-CreatePageModel og den tilsvarende for **Product**?
* Hvor meget forskel vil der være mellem f.eks. en **Customer**-AllPage og den tilsvarende for **Product**?
* Hvor meget forskel er der egentlig på de forskellige PageModel-klasser? Mange af dem har brug for en reference til en data-service, og en data-bærende property til data-binding.

Hvis du har mod på det, så fortsæt gerne med f.eks. at lave nogle bud på nogle gode base- og/eller type-parameterisesrede klasser og interfaces, og se hvor langt du kan komme med dette. Alternativt kan du også prøve at implementere CRUD-funktionalitet for en f.eks. en **Customer**-klasse ”direkte”, uden at tænke for meget på base-klasser og lignende. Vælg det du synes der giver mest mening for dig 😊.